Master of Science Thesis

Deep Reinforcement Learning for Goal-directed Visual Navigation

M. Kisantal

February 10, 2018

Faculty of Aerospace Engineering · Delft University of Technology

TU Delft Delft University of Technology

Challenge the future

Deep Reinforcement Learning for Goal-directed Visual Navigation

Master of Science Thesis

For obtaining the degree of Master of Science in Aerospace Engineering at Delft University of Technology

M. Kisantal

February 10, 2018

Faculty of Aerospace Engineering · Delft University of Technology

Copyright © M. Kisantal
All rights reserved.

DELFT UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF
CONTROL & SIMULATION

The undersigned hereby certify that they have read and recommend to the Faculty of Aerospace Engineering for acceptance a thesis entitled **"Deep Reinforcement Learning for Goal-directed Visual Navigation"** by **M. Kisantal** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: February 10, 2018

Supervisor: ______________________
Dr. G.C.H.E. de Croon

Daily Supervisor: ______________________
Ir. K. Hecke, MSc

Reader: ______________________
Dr.ir. E. van Kampen

Reader: ______________________
Dr.ir. J. Kober

Acknowledgements

In December 2016, I had the privilege to attend the Neural Information Processing Systems Conference as part of the AImotive delegation, where I was interning at that time. This conference ended up being a formative experience for me: I learned about the state-of-the-art in Machine Learning, got an impression that I might contribute too to advance the field further, and most importantly I got some inspiration and ideas for this thesis.

First and foremost, I want to thank my supervisors, prof. Guido de Croon and Kevin van Hecke. Guido's advices were vital for this research, and I learned a lot from him during our meetings and discussions, not just about machine learning and artificial intelligence, but also about conducting research in general. Furthermore, I am really grateful for the freedom I was given during this research to pursue topics I was really interested in. Kevin was a great help just as well, especially those days all my efforts seemed to fail. His cool-headed and rational approach to debugging complex ML models was invaluable and got me back on track those times when I started to loose confidence.

This thesis would not have been possible without the support I got from AImotive Inc. I would like to offer my special thanks to Dávid Kiss and Márton Görög for mentoring me, CEO Lászlo Kishonti for generously supporting this research with a high-end computer, and a number of other colleagues for the amazing months I spent working with them.

Finishing my studies is also a great opportunity to reflect back to the years that led here. I got first-rate mentorship during my undergrad years from Bálint Vanek and István Gőzse, who made it possible for me to continue my studies at TU Delft. I also wish to acknowledge the help of Györgyi Varga-Umbrich, Mária Varga and Pál Péter Fekete, who got me started on this path.

Finally, I want to thank my family for their invaluable support through all my studies, and Ági, for always being there for me.

Thank you.

Máté Kisantal

Abstract

Safe navigation in a cluttered environment is a key capability for the autonomous operation of Micro Aerial Vehicles (MAVs). This work explores a (deep) Reinforcement Learning (RL) based approach for monocular vision based obstacle avoidance and goal directed navigation for MAVs in cluttered environments. We investigated this problem in the context of forest flight under the tree canopy.

Our focus was on training an effective and practical neural control module, that is easy to integrate into conventional control hierarchies and can extend the capabilities of existing autopilot software stacks. This module has the potential to greatly improve the autonomous capabilities of MAVs, and their applicability for many interesting real world use-cases. We demonstrated training this module in a visually highly realistic virtual forest environment, created with a state-of-the-art computer game engine.

Acronyms

Asynchronous Advantage Actor-Critic	**A3C**
Backpropagation Through Time	**BPTT**
Convolutional Neural Network	**CNN**
Deep DPG	**DDPG**
Deep Learning	**DL**
Deep Q-Network	**DQN**
Deep Recurrent Q-Network	**DRQN**
Deterministic Policy Gradient	**DPG**
First Person Shooter	**FPS**
Flight Control Unit	**FCU**
Gated Recurrent Unit	**GRU**
Generative Adversarial Network	**GAN**
Global Positioning System	**GPS**
Grand Theft Auto	**GTA**
Hardware-in-the-Loop	**HTIL**
Histogram of Oriented Gradients	**HOG**
Learning from Demonstrations	**LfD**
Light Detection And Ranging	**LIDAR**
Long Short-Term Memory	**LSTM**
Machine Learning	**ML**
Markov Decision Process	**MDP**
Maximum Mean Discrepancy	**MMD**
Micro Aerial Vehicle	**MAV**
Multi-Layer Perceptrons	**MLP**
Multi-Task Learning	**MTL**
Neural Fitted Q-Iteration	**NFQ**
Neural Network	**NN**
Oriented FAST and rotated BRIEF	**ORB**
Rectified Linear Unit	**ReLU**

Recurrent Neural Network	**RNN**
Reinforcement Learning	**RL**
Root Mean Square Propagation	**RMSProp**
Self-Supervised Learning	**SSL**
Software-in-the-Loop	**STIL**
Stochastic Gradient Descent	**SGD**
Temporal Difference	**TD**
Trust Region Policy Optimalization	**TRPO**
Unreal Engine 4	**UE4**
Variational Auto-Encoder	**VAE**
Vision-based Neural Navigation Module	**VNNM**

List of Symbols

Greek Symbols

α	Momentum hyperparameter
γ	Discount factor
ϵ	Learning rate
μ	Deterministic policy
θ	Vector of parameters
π	Stochastic policy

Roman Symbols

$\mathcal{A}$	Action space
c_S, c_T	Source and target classifiers
D	Discriminative model
$d_{\mathcal{H}}(.,.)$	$\mathcal{H}$ divergence
$\mathcal{D}_S, \mathcal{T}_S$	Source and target domain probability distributions
$\mathbb{E}$	Expectation operator
$\mathcal{F}$	Function class
F	Feature extractor
g	Gradient estimate over mini-batch
G	Generative model
G_t	Discounted return from timesetep t
$\mathcal{H}$	Hypothesis class
$I[.]$	Binary indicator function
$J(.)$	Loss function

$L(.)$	Per-example loss
m	Training set size
m'	Mini-batch size
$\mathbb{P}$	Probability
$\mathcal{P}_{ss'}^{a}$	State transition probability function
p, q	Probability distributions
$Q(s, a)$	Action-value function
r	Reward
$\mathcal{R}$	Reward function
$\mathcal{S}$	State space
S, T	Sets of source and target domain samples
v	Velocity for momentum
$V(s)$	State value function
x	Input vector
X, Y	Sets of training samples
y	Output vector

Contents

Acronyms **i**

List of Symbols **i**

I Introduction **3**

1 Background **5**

1.1 Content and structure of this report 7

II Article **9**

III Preliminary Thesis - Literature Review **23**

2 Deep Learning **25**

2.1 Artificial Intelligence and Machine Learning 25

2.2 Artificial Neural Networks 26

2.3 Interpretations of Deep Learning 27

2.3.1 Exponential advantages of deep neural architectures 28

2.3.2 Manifold Learning 28

2.4 Training 29

2.5 Regularization 31

2.6 Neural Network Architectures 33

2.6.1 Convolutional Neural Networks 33
2.6.2 Recurrent Neural Networks 35

3 Domain Adaptation 37
3.1 Supervised Domain Adaptation 38
3.2 Semi-supervised Domain Adaptation 39
3.3 Unsupervised Domain Adaptation 40
3.3.1 Aligning second-order Statistics 41
3.3.2 Maximum Mean Discrepancy based methods 41
3.3.3 Domain adversarial training 44
3.4 Domain adaptation for Reinforcement Learning 48
3.4.1 Changing input distributions 49
3.4.2 Domain adaptation in a general case and transfer learning 50
3.5 Training with synthetic data 51

4 Deep Reinforcement Learning 53
4.1 General Reinforcement Learning 53
4.1.1 Markov Decision Processes 55
4.2 Value and policy based RL algorithms 56
4.2.1 Value based methods 57
4.2.2 Policy Gradient Reinforcement Learning Methods 58
4.3 Deep Reinforcement Learning 59
4.3.1 Difficulties of using Deep Neural Networks in RL 60
4.3.2 Deep RL Algorithms 61

5 Auxiliary Training 65
5.1 The auxiliary training in supervised training 66
5.1.1 Multi-Task Learning 66
5.1.2 Aiding supervised learning with unsupervised learning 66
5.2 Auxiliary training for RL 68
5.2.1 RL agent for FPS playing 69
5.2.2 NAV agent 70
5.2.3 UNREAL agent 72
5.3 Auxiliary training tasks 75

5.3.1 Depth prediction . . . 75
5.3.2 Optical Flow . . . 77
5.3.3 Ego-motion estimation . . . 78
5.3.4 Detection and semantic segmentation . . . 78
5.3.5 Mapping . . . 79
5.3.6 Unsupervised auxiliary tasks . . . 80

6 Literature Review Discussion 83
6.1 Deep Learning . . . 83
6.2 Domain Adaptation . . . 84
6.3 Deep Reinforcement Learning . . . 84
6.4 Auxiliary training . . . 85

IV Preliminary Thesis - Preliminary Analysis 87

7 The learning task 89
7.1 MAV control hierarchy . . . 89
7.2 Obstacle Avoidance and Navigation module . . . 90
7.3 Reward function . . . 91

8 Auxiliary training with real data 93
8.1 Incorporating real images in the training process . . . 93
8.2 Mixing synthetic and real data in the training process . . . 94
8.3 Sources of real training data for auxiliary training . . . 94
8.3.1 Public Datasets . . . 94
8.3.2 Collecting new datasets . . . 95
8.4 Specificity problems of the real datasets . . . 96

9 Network architecture 97
9.1 Base RL network . . . 97
9.2 Auxiliary tasks . . . 98
9.3 Domain Adversarial training . . . 99
9.4 The complete neural architecture . . . 100

10 Remaining Thesis Work **103**

10.1 Simulation Environment Setup 103

10.2 RL agent and Neural Network architecture setup 104

10.3 Auxiliary training experiments in simulation 105

10.4 Setting up MAV experiments 105

10.5 Domain Adaptation Experiments 106

List of Figures

1.1 Examples of cluttered outdoor and indoor environments 5

1.2 Screenshot from our simulator (left), and photograph of a real forest (right) 6

2.1 Illustration of a simple feedforward neural network with two hidden layers 26

2.2 Convolution with a single filter 34

3.1 Visualization of domain distributions with t-SNE 40

3.2 Network architectures for different MMD based domain adaptation methods 42

3.3 Symmetric and asymmetric architectures for Domain Adversarial Training 46

3.4 Synthetic image and semantic segmentation ground truth from GTA V . . 52

4.1 Breakout, a popular Atari computer game 53

4.2 General RL setting 54

5.1 Schematic picture of an autoencoder architecture 67

5.2 A deep neural network augmented with an autoencoder 68

5.3 Architecture with shared and task specific sub-networks 69

5.4 Network architecture of the augmented DRQN 69

5.5 Architecture of the NAV agent 71

5.6 Network architecture of the UNREAL agent 74

5.7 Depth prediction results 76

5.8 FlowNet2.0 optical flow prediction 77

5.9 Single image optical flow prediction results 78

5.10 Semantic segmentation of a traffic scene 79
5.11 Output visualizations for the cognitive mapper 80

7.1 Typical control system for a MAV 89
7.2 Control system augmented with the RL trained Obstacle Avoidance & Navigation module 90

8.1 Example images from the NYU Depth dataset 95
8.2 Sparse optical flow ground truth for a sequence from the KITTI dataset . 95

10.1 Images generated with UnrealCV 104

Part I

Introduction

Chapter 1

Background

Today Micro Aerial Vehicles (MAVs) are getting more and more accessible, as various manufacturers are providing mass-produced MAVs at reasonable prices. Their low size and cost, along with their agility make MAVs an ideal platform for a number of tasks. However, most MAVs are still used for hobby and recreational purposes, while their commercial use is limited. One reason is that for efficient, scalable and safe commercial use more reliable autonomous capabilities are needed.

Obstacle avoidance and goal-directed navigation are key capabilities for autonomous operation of MAVs. For this, MAVs need to perceive their environment, understand the situation they are in, and react appropriately. From this point of view, flying in cluttered outdoor environments (such as forests), or in GPS-denied indoor environments is highly challenging (see some examples of cluttered environments in Fig. 1.1).

Figure 1.1: Examples of cluttered outdoor and indoor environments: a forest scene; busy street with buildings and overhanging wires; cluttered warehouse environment. (*credit: CC BY-SA 3.0 Peter Bond, Sebastian Kasten, Andreas Praefcke*)

Given the size, weight, power and cost limitations of MAVs the choice in sensors is limited. LIDARs for example are suitable for depth sensing, but their price, weight and power consumption make them less suitable for MAV applications. Cameras are cheap and lightweight sensors that can provide images with an abundance of information. Still obstacle avoidance and navigation of MAVs based on a monocular camera image input

is challenging. It requires understanding the 3D structure of the scene, which relies on correct interpretation and integration of a number of visual clues.

Our goal was to create a vision-based navigation module, that can extend the autonomous capabilities of MAVs with the ability to traverse cluttered environments based on monocular vision input. To this end, we defined the module and its interfaces such that it can be easily integrated into existing autopilot software stacks.

Figure 1.2: Screenshot from our simulator (*left*), and photograph of a real forest (*right*)

A common approach to vision-based obstacle avoidance is to decouple the perception and control problems: first a perception algorithm creates a 3D representation of the scene from the input image(s), and then a path planning/control algorithm is responsible for safe traversing according to the reconstructed 3D space. In this work, we are exploring an integrated approach to this problem, where perception and control are strongly coupled and co-adapted. This concept holds the promise of finding more efficient solutions, however, in most cases it relies on learning-based techniques.

We address the training of a module that integrates perception and control with a Reinforcement Learning (LR) based approach. RL is a particularly flexible learning paradigm, that does not require explicit supervision. Instead it lets agents learn by interacting with an environment, while providing evaluative feedback [1]. Deep RL (the combination of RL with deep neural network function approximators) has shown promising results in learning vision-based policies (e.g. mastering a suite of Atari games [2]).

However, the required training time of Deep Reinforcement Learning algorithms is still prohibitively high for training in real environments; furthermore initial exploratory policies might damage the test vehicle itself. These are the main obstacles that prevent learning in the physical domain for MAVs, and generally for robots. These problems can be partially addressed with training in a virtual domain. However, training on virtual data introduces a new problem, as the transferability of the learned policies to the real domain is not trivial. In the literature review we discuss various domain adaptation techniques that can help closing the reality gap between the simulated environment and the real world. Within the scope of this thesis we only address this problem with using a highly realistic simulated environment that is built upon a state-of-the-art computer game engine, which features a high quality rendering pipeline (see the the side-by-side comparison to a real image in Fig. 1.2). Testing the effectiveness of the trained policies in real environment and exploring different domain adaptation alternatives to improve the

performance is left for future work.

1.1 Content and structure of this report

The first part of this thesis is a scientific paper that summarizes the most important results and findings of this research, and briefly describes our methodology.

Part III. and IV. are from the Preliminary Thesis, included for completeness and to provide some additional context and background information for the article.

In particular, Part III. is a literature review, which discusses the theoretical background of this research. First in Chapter 2 Deep Learning is discussed, with the related optimization and regularization issues, as well as some common network architectures. The topic of Chapter 3 is Domain Adaptation. This chapter discusses the performance issues of neural networks when transferred to different domains, and reviews the existing techniques to alleviate these problems. Next, in Chapter 4 we introduce the basics of Reinforcement Learning, as well as some recent Deep RL algorithms. Chapter 5 concludes the the literature review with a discussion of auxiliary training. This chapter also contains a brief overview of a number of learning tasks that are relevant for the main task (obstacle avoidance and goal-directed navigation), and on which successful deep learning was demonstrated. Finally, we discuss the finding of the literature review in Chapter 6.

Part IV. is a preliminary analysis, that dives more into the core problems. First, we specify the reinforcement learning task in Chapter 7. Then in Chapter 8 we discuss some considerations related to the incorporation of real data in the training process. This is followed by the description of the proposed network architecture in Chapter 9. Finally, a brief research planning is given in 10.

Part II

Article

Deep Reinforcement Learning for Goal-directed Visual Navigation

Máté Kisantal[1], Kevin van Hecke[2] and Guido de Croon[3]

Abstract— **Safe navigation in a cluttered environment is a key capability for the autonomous operation of Micro Aerial Vehicles (MAVs). This work explores a (deep) Reinforcement Learning (RL) based approach for monocular vision based obstacle avoidance and goal directed navigation for MAVs in cluttered outdoor environments, such as forests. Our focus was on training an effective and practical neural control module, that is easy to integrate into conventional control hierarchies and can extend the capabilities of existing autopilot software stacks. We demonstrate learning neural policies in a visually highly realistic virtual forest environment.**

I. INTRODUCTION

Today MAVs are getting more and more accessible, as various manufacturers are providing mass-produced MAVs at reasonable prices. Their low size and cost, along with their agility make MAVs an ideal platform for a number of tasks. However, most MAVs are still used for hobby and recreational purposes, while their commercial use is limited (mostly to drone inspection under pilot control). One reason is that for efficient, scalable and safe commercial use more reliable autonomous capabilities are needed.

Safe navigation in cluttered environments is a key capability for autonomous operation of MAVs. Safe navigation is essentially a trade-off between goal-directed motion and obstacle avoidance. In order to avoid obstacles, MAVs need to perceive their environment, understand the situation they are in, and react appropriately.

In this work we address the problem of MAV forest flight under the tree canopy. Providing an MAV with the capability of safely traversing such environments would open the way for novel use-cases, like flying autonomous search and rescue missions or surveying forest wildlife and vegetation. In this environment the MAV has to actively avoid trees, branches and various other obstacles. Since precise mapping of such environments for generating safe pre-planned trajectories is overly expensive for most use-cases, ideally the MAV has to be capable of generating safe trajectories locally based on its own observations.

Obstacle avoidance relies on the correct sensing of the 3D environment around the MAV. Some sensors (e.g. LIDAR) can directly provide this information [1], [2]. However, given the size, weight, power and cost limitations of MAVs the choice in sensors rather is limited. Cameras are cheap, lightweight and low-power sensors that can provide images with an abundance of information. These properties make cameras very suitable for MAVs. Still, vision-based obstacle avoidance is challenging, as it requires understanding the 3D structure of the scene, which relies on correct interpretation and integration of a number of visual clues.

A common approach to vision-based obstacle avoidance is to decouple the perception and control problems: first a perception algorithm creates a 3D representation of the scene from the input image(s), and then a path planning/control algorithm is responsible for the safe traversal according to the reconstructed 3D space (e.g.[3], [4], but also some more recent deep learning based approaches such as [5] and [6]). An advantage of this framework is that it benefits from developments both in 3D scene reconstruction and in motion planning algorithms [7]. Furthermore, as the intermediate 3D representation is easily interpretable, debugging these systems is easier. On the other hand, decoupling the two processes, and constraining an intermediate representation may be less efficient than a solution that integrates these processes.

Integrated approaches sidestep the need for exact state estimation, and map the states to higher level outputs (typically collision probabilities, or directly to actions; e.g. [8], [7], [9], [10]). In practice, obtaining accurate 3D representations of the full 3D scene, or having highly accurate localization might not be essential for successful navigation and obstacle avoidance. In many cases just an approximate relative localization, and knowing obstacle distances in a few directions is enough already. However, determining what features are really needed for a particular task is a difficult problem in itself, which is strongly dependent on both the available observations, and on the properties of the controller that uses the features. Here lies the advantage of integrated approaches: control and perception are strongly coupled and co-adapted, and since internal representations are not explicitly constrained, ideally integrated approaches have the potential to be more effective. Such integrated approaches typically rely on machine learning techniques that can optimize the internal representations.

RL provides a flexible framework to learn policies for a variety of control problems, and has already been explored on a variety of robotics problems[11]. Recent advancements in Deep Reinforcement Learning (DRL) (where deep neural networks are used as function approximators) has shown promising results, training successful agents in highly challenging problems (such as a suite of Atari games [12]) from raw visual inputs, in some cases outperforming even human performance. DRL algorithms are capable of learning perception and control jointly, and the internal representations

[1] Student, Faculty of Aerospace Engineering, Technical University Delft kisantal.mate@gmail.com

[2] Daily Supervisor, Faculty of Aerospace Engineering, Technical University Delft

[2] Supervisor, Faculty of Aerospace Engineering, Technical University Delft

in their neural networks are learned and optimized for the task at all hierarchical levels.

We employ DRL to train a Vision-based Neural Navigation Module (VNNM), that can be used to extend the autonomous capabilities of current autopilot software stacks. To this end, we set the boundaries of this module and define the interfaces such that it can be incorporated in conventional control hierarchies. In order to interface with higher level navigation modules, the trained policy is a function of both the observations and a goal direction input. The output of our policy is a velocity command, that can be easily tracked with conventional feedback control methods.

Our main contribution is the definition, training and evaluation of a vision-based neural controller (the VNNM) that can add vision-based navigation capabilities to existing MAVs to make them able to operate in cluttered environments autonomously. We demonstrate the ability of the system to learn goal directed navigation in a virtual forest environment. Furthermore, we release a photo-realistic simulator of a forest environment with high quality graphics that can be used to train and and evaluate vision-based approaches for obstacle avoidance, path planning and navigation.

II. Related work

Vision based obstacle avoidance and navigation for MAVs has attracted significant interest in the research community over the last few years. Through this section our focus is on learning-based, integrated approaches for these problems.

A. Obstacle avoidance

Most learning-based approaches cast obstacle avoidance as a supervised learning problem. In imitation learning this supervision comes straight from an expert, and a direct mapping from raw inputs to expert actions is learned [13], [14], [15]. However, collecting demonstrations might be a tedious work, and accounting for the changing state distributions makes repeated iterations necessary [16]. Another possible source of expert trajectories can be a *model predictive control* trajectory planner, that can find optimal trajectories given full information of the environment. With these expert trajectories a neural network policy can be trained to follow the same trajectories based on partial observations. Recent work explored this idea, but it was demonstrated only in simplistic simulated environments [17], [18].

Another way to formulate obstacle avoidance as a supervised problem is learning to predict collision probabilities based on the observations. Some approaches check if an action results in a collision immediately [19], or over a certain time-horizon following a policy [20], [8]. Despite being an RL algorithm, we mention the recently introduced CAD^{2}RL algorithm [7] here, as it can also be seen as very similar method. In this work the authors defined the reward to be 0 at collisions and 1 when the agent is further from obstacles than a threshold distance. Due to this particular reward definition, the action value function is closely related to collision probabilities.

Given a learned collision probability function, simple hand-engineered policies are used in the above mentioned examples, that are able to follow safe trajectories by choosing actions according to the collision probabilities. In particular, [7] follows a greedy policy (choosing the safest action) while [8] determines a safe flight direction by evaluating left, right and center crops of the image. In theory collision probability learning approaches can be integrated with higher level navigation objectives (e.g. moving towards a goal location if the required action has low risk, and making detours otherwise). However, this approach raises two problems. First, the collision probabilities are only valid as long as the policy is the same that was used during the training (typically greedy obstacle avoidance policies that use the command with the least collision probability). By following a different policy (e.g. one that takes goal-directed navigation into account) this condition does not hold anymore. Second, in most cases the policies are hand-engineered and not optimized. Typically a certain collision probability threshold is picked, but this does not necessarily represents a good trade-off between obstacle avoidance and goal directed motion. In contrast, our proposed approach learns safe goal directed navigation in an end-to-end manner, and the RL training directly optimizes the action selection.

Besides supervised approaches, training obstacle avoidance with RL was also addressed in previous work [9], [21]. Khan et al. introduced a model-based RL method for obstacle avoidance, Generalized Computation Graphs (GCG) [9]. Their most successful GCG instantiation relies on formulating the learning task as prediction of discrete future events. This formulation works really well in the case of simple obstacle avoidance, where the predicted discrete events can be collisions. However, their method is difficult to extend to goal oriented navigation. As our approach does not rely on a special formulations of the RL problem, it allows greater flexibility in the task and reward function definitions, which lets us train goal-oriented navigation directly. A previous work of Khan et al. proposed an uncertainty-aware model-based RL method for collision avoidance [21], however its effectiveness was demonstrated in a rather limited setting only (avoiding a single obstacle, based on 16 by 16 grayscale images). Model-based RL methods in general are learning a predictive model that gives the expected reward (or in an obstacle avoidance scenario the collision probability) given observations and a sequence of actions. This model is usually used in a model predictive control planner, to find the best sequence of actions within a certain time horizon. The separate planning process may consider the goal location in choosing the best path, however it requires multiple model evaluations in test time for finding acceptable actions. This can be a problem for constrained platforms such as MAVs. Our approach is model free, thus the inference directly outputs an action.

B. Path Detection

Recent approaches managed to achieve safe traversal in natural environments by constraining the motion to man-

made paths. In this case the learning task is the prediction of the path direction, for which supervision can be provided. Forest paths typically indicate traversable 3D corridors, thus following them sidesteps the need for obstacle detection and path planning.

Giusti et al. [10] made recordings with three cameras looking to different directions, and trained a network to recognize the direction of the camera relative to the trail. Another work extended their datasets with videos recorded with different lateral offsets [22]. While constraining the motion to forest trails can be useful in a number of practical scenarios and completely sidesteps the problem of path planning, it does not make use of the fundamental capability of MAVs they can fly freely irrespective of the terrain conditions This significantly limits the accessible areas for the MAV operation.

C. Goal-directed navigation

Several recent works addressed the problem of vision-based navigation using learning based methods [23], [24], [25]. The agent in [23] is trained to locate a certain object in various mazes, but it does not generalize to other goal objects/locations. Zhu et al. trained a policy, that is a function of both a visual input and an image of a target object. Their approach generalizes to various targets, however specific layers of their network need to be finetuned for particular environments to learn room layouts and object arrangements [25]. In general, semantic navigation approaches are rather specialized, which renders them less useful for the kind of general-purpose navigation that we seek to achieve for MAVs. Gupta et al. also investigates vision-based navigation as a geometric task, where the goal location is defined as relative coordinates, however their method assumes the environment to be a grid-world[24]. A common shortcoming of the above-mentioned methods from an MAV implementation point-of-view that they do not consider collision avoidance.

Somewhat similar to our work, Smith at al. focuses on a learning based approach that is easy to integrate to traditional navigation stacks [26]. However, their approach relies on depth sensing, and the primary role of trained neural network in their setting is avoiding exhaustive search by selecting the most probable candidate trajectories for a planner. In the context of indoor ground robots, Pfeiffer et al. proposes a similar solution to goal-directed navigation as ours [27], however they simplify the perception problem by using 2D laser range finder inputs, and the model is trained in a supervised fashion using expert demonstrations from a path planning algorithm.

III. Training an Obstacle Avoidance module for MAVs

In the introduction we argued that a learning-based obstacle avoidance module, which integrates perception and control can be a promising direction towards more autonomous MAVs. In this section we discuss how such a module can augment the usual MAV control hierarchy to provide vision based obstacle avoidance, and how the obstacle avoidance task can be framed as an RL problem. We also describe the simulated environment that was used during training, discuss the RL algorithm and the neural architecture of the agent, and finally report the training details.

A. Obstacle Avoidance module in the MAV control hierarchy

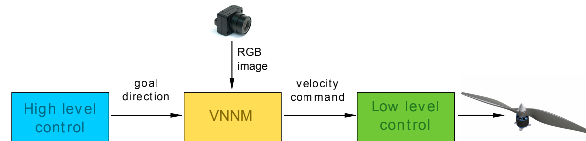

Fig. 1. VNNM module in the MAV control hierarchy.

The control hierarchy of a typical MAV can be divided to two parts: low and high level control. The *low level control* typically consists of cascaded layers of feedback controllers, that stabilize the MAV, provide adequate handling characteristics, and in most cases control the attitude and velocity of the MAV. The *high level control* usually include some sort of a navigation control loop, and additionally further high level mission planner and decision making modules. Alternatively, high level control may also come directly from a human operator.

In theory, a learning based control module can replace any part of the control hierarchy, or the whole hierarchy at once in and end-to-end fashion. However, since our main goal is to provide obstacle avoidance and goal-directed navigation capability, it is advantageous to keep some parts of the original hierarchy. Furthermore, limiting the scope of the learning task makes training more feasible. The reusability and easy integration into existing autopilot software stacks were our main consideration in setting the boundaries and defining the interfaces of the trained module.

On the low-level side, traditional feedback control methods are more reliable and better understood, than learning based methods. Also, keeping low-level feedback control in place may allow reusing the trained policy across different MAVs, as suitable feedback control can mask the differences between platforms at the level of more abstract actions. This abstraction can also help narrowing the reality gap in case of training in a simulator [28], and also allows for using simpler motion models, that can neglect the low-level dynamics of the MAV. Therefore the outputs of the obstacle avoidance module are higher level velocity commands, and a conventional low-level cascaded control system is responsible for tracking these. Since we chose to maintain a constant forward speed and altitude, this velocity command is a heading command in our case.

When extending the module to higher levels, the main consideration is keeping the module flexible so that it can be used in many contexts. As an example, if the module is trained directly for a high-level navigation objective (e.g. searching for a certain object) then its use would be significantly constrained (searching for another object would require the re-training of the module). Based on reusability

considerations the input for the obstacle avoidance module from higher levels was chosen to be the relative direction of the navigational goal. This signal only relies on knowing the current heading of the MAV, which can be provided even in GPS-denied environments by an on-board magnetometer. With this input it is easy to integrate the module with waypoint-based navigation systems. Another use case might be an intelligent steering mode, in which an operator can steer the MAV in a desired direction, while the module prevents it to collide with obstacles.

This formulation essentially leads to a module, that is making a trade-off based on visual information between flying directly towards the goal and making avoidance manoeuvres, thus providing a safe trajectory locally. In a sense the role of this model is similar to the path planning a horse does in an ordinary riding scenario: while the rider can control the general direction of movement, the horse still has some authority to avoid running into obstacles.

B. Training Environment

Teaching goal-based navigation for MAVs using RL on real platforms is problematic, given the trial-and-error nature of RL and the possibly unsafe initial exploratory policies. Safe RL is a direction in RL research that is trying to explicitly minimize the risks of training (e.g [29]). In this work we sidesteps the safety concerns of RL by training the agents in a virtual environment. Besides the safety benefits, virtual environments allow for speeding up the training process by running the simulation faster than real time, and also give an opportunity to parallelize training. Furthermore, given access to the internals of the simulator, perfect information is available about the state and various other aspects of the environment. However, training on a virtual environment introduces a domain discrepancy, and the performance of the agent on the real environment is not guaranteed.

Therefore creating a suitable virtual environment for training an RL agent is a challenging task. On one hand, as our proposed neural control module operates on the level of velocity commands, the physical fidelity is relatively easy to achieve with a simple motion model. On the other hand, the required visual fidelity is significantly more difficult to achieve, especially since we do not make use of any explicit domain adaptation technique.

Fig. 2. Screenshot from our simulator (*left*), and photograph of a real forest (*right*)

The computer game industry, however, has put significant efforts and resources for creating visually realistic gaming experiences [30]. In this work we are making use of the open-source, cross-platform Unreal Engine 4 game engine, which features an exceptionally high quality rendering pipeline [31]. We created a virtual forest environment for RL training using the free Open World Demo Collection of Epic Games. The environment features 229 collidable trees scattered around in a roughly $100\ m \times 100\ m$ area. Latter evaluation plots are displaying a map of the environment with coarse obstacle locations, e.g. in Fig. 7 or 10. The traversable areas are surrounded with a rock wall. A few screenshots of the simulator are show in Fig. 3 that showcase the high quality models and textures, and the various light conditions across the environment. A side-by-side comparison to a real image is given in Fig. 2.

We access the internals of the simulator using the UnrealCV plugin [30], that allow us moving the agent in the environment, detecting collisions and getting RGB and depth images.

Fig. 3. Scenes from our realistic forest simulation environment built with UE4. The top three images show different parts of the environment from the perspective of the agent, while the bottom two show further details of the environment.

In order to evaluate the difficulty of avoiding obstacles in this environment, we measured the non-dimensional *traversability* of the environment [32], which is closely related to the obstacle density. Traversability is calculated by spawning the agent to random locations with random headings, and averaging the collision-free straight path lengths s over n runs. Non-dimensional traversability is normalized

with the radius of the MAV:

$$Traversability = \frac{1}{n \cdot r} \sum_{0}^{n} s. \tag{1}$$

The average collision-free trajectory length in our experiments was $23.4\ m$, calculated over 600 trials. Since the collision radius of the MAV in our experiments is $0.5\ m$, the traversability of the forest environment is 48.6.

C. Defining obstacle avoidance as an RL problem

RL is a machine learning paradigm in which an agent is interacting with its environment, sequentially takes actions and receives scalar rewards, along with observations of the state of the environment [33]. The goal of the agent is to maximize the cumulative reward it receives. By defining the reward function ("shaping the reward") we can encourage the agent learn a particular task. The objective in our case is learning a policy that navigates the drone towards a goal location in a safe manner avoiding obstacles. This section casts this task as an RL problem: we define the reward function, the action space and the observations the agent receives from the environment.

The task has two, sometimes competing goals: flying towards the goal location, and avoiding collisions. These goals are directly reflected in the reward function:

$$R_t = r_{direction} + r_{collision} \tag{2}$$

The direction reward is calculated as the dot product of the (planar) normalized goal and displacement vectors at each timestep. This provides a dense reward signal with values in the interval $[-1, 1]$ that encourages flying directly to the goal. The collision reward is -10 if the command results in getting closer to an obstacle than one meter, and 0 otherwise. Upon collision the episode is terminated.

Apart from the reward, the agent receives observations (s) from its environment. The primary observation is a 84×84 RGB image, where pixel values are normalized between ± 1. The agent gets information about the goal direction as the sine of the heading error ($sin(\psi_{goal} - \psi_{MAV})$), this signal is again conveniently between ± 1.

The action space in our experiments is discrete. It consists of three angular acceleration actions (a) for steering the drone: changing the heading rate of the drone ($\pm 5°/step^2$), or keeping a constant heading rate. Since the maximum heading rate is limited at $\pm 15°/step$, the heading rate can take the following discrete values: $\dot{\psi} \in [-15,\ -10,\ -5,\ 0,\ 5,\ 10,\ 15]$. Our motivations for angular acceleration commands were the smooth resulting paths, and that acceleration commands are easily tracked by the baseline feedback controllers. In order to keep the observability of the problem, the inputs of the agent are augmented with the heading rate state.

D. RL Algorithm

A fundamental problem in Deep RL is that on-line observation sequences from an agent are highly correlated [12]. The first successful deep RL agents were using off-policy algorithms to overcome this problem, as it allowed learning from randomly sampled minibatches from an experience replay memory [12]. As Mnih et al. proposed, uncorrelated experiences may also come from multiple agents acting in separate instances of the environment simultaneously. Our algorithm is largely based on their Asynchronous Advantage Actor-Critic (A3C) algorithm [34]. The critic estimates the state-value function ($V(s_t; \theta_v)$), and the actor uses its predictions to improve the policy ($\pi(a_t|s_t, \theta)$). The critic is trained to minimize the *temporal difference error*, while the actor updates are calculated with the *policy gradient theorem*, using the *advantage* as a baseline [33]. To lower the variance of the advantage estimate, we use Generalized Advantage Estimation (GAE) [35]. Following [34] the entropy of the policy is added to the objective function to encourage exploration and prevent early convergence to sub-optimal policies. We are using eight separate agents (workers), each collects experiences in their separate instances of the simulator, and send gradient updates to a master network. The network parameters of the workers are updated after each episode.

Another difficulty in deep RL that these algorithms are learning from a single, scalar reward signal. This leads to high sample complexity and excessive training times. A possible solution to alleviate this problem can be training the network with multiple loss functions simultaneously. This idea is closely related to Multi-Task Learning in the context of supervised learning [36]. An early example of this approach in RL context was that deep actor-critic implementations usually did not use separate actor and critic networks, instead most weights were shared between the two (e.g. in [34] only the last layer is separate). This decision can be justified by arguing that the features needed for predicting the value of a state can be just as useful for the policy, and vice versa.

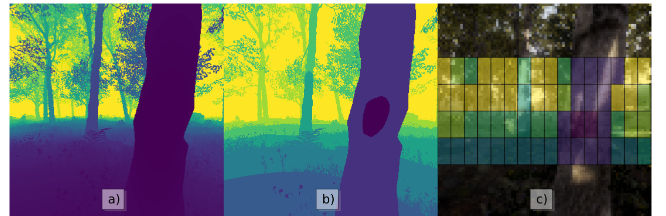

Fig. 4. High resolution ground truth depth image (*a*); quantized depth image (*b*); cropped and down-scaled depth image overlaid on the corresponding input image (*c*)

Recent works introduced auxiliary tasks for RL, demonstrating that both supervised [34], [37], unsupervised and RL tasks [38] can be introduced to improve the RL training process, both in terms of sample complexity and the performance of the learned policies. In particular we build upon the experiences of Mirowski et al., who showed that an RL agent that learns to navigate in mazes can significantly benefit from learning depth and loop closure prediction as auxiliary tasks [23].

Vision-based navigation and obstacle avoidance relies

heavily on correctly interpreting the 3D scene from single images. Because of the relatedness of the two task, the RL training can benefit the auxiliary depth prediction task. An important consideration is that the auxiliary task has to be easier to solve, to allow the main task to make use of the learned features. Therefore (following [23]) we define the depth prediction task as classification of the pixels of a low resolution depth image into eight depth bins. The original depth image from the simulator is shown in Fig. 4 *a)*, while *b)* shows the depth image quantized into exponentially spaced bins. To significantly reduce the number of depth pixels (thus making the auxiliary task simpler), only the most important central part of the image is covered in the depth prediction task, with lower resolution along the less significant vertical axis. The down-scaled depth image, that consists of 64 pixels is shown on Fig. 4 *c)*, overlaid on the visual input the agent receives.

For a pseudo code of the complete RL training algorithm please refer to Appendix I.

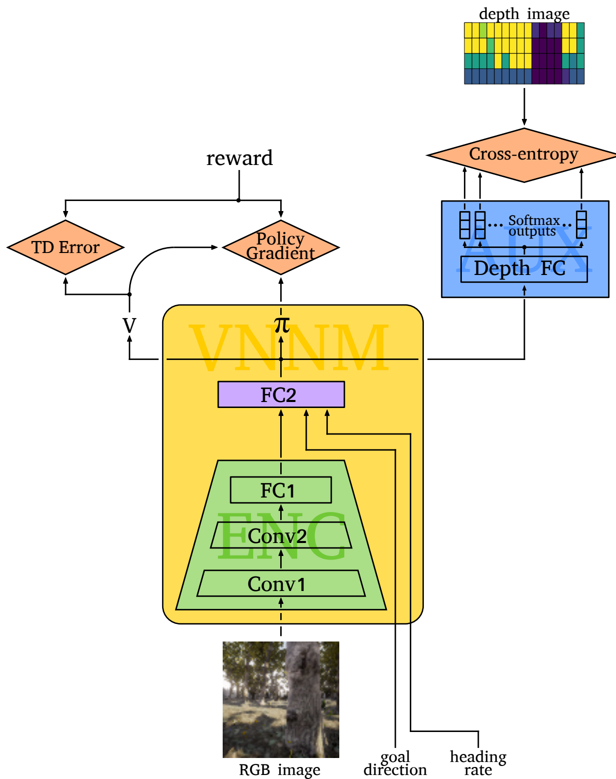

Fig. 5. Neural architecture of the RL agent.

E. Agent architecture

The neural network architecture of the agent is shown in Fig.5. The setup closely follows the DQN [12] and NAV [23] architectures.

The RGB input image ($84 \times 84 \times 3$) is fed to an encoder, that consists of two convolutional layers and a fully connected layer. The receptive fields of the convolutional layers are 8×8 and 4×4 respectively (stride 4 and 2), with 16 Exponential Linear Units (ELU) [39] in the first layer and 32 ELUs in the second. The encoder ends with a fully connected layer (FC1) that has 256 ELUs.

The next fully connected layer (FC2) consists of 256 ELUs, that are fed with both the encoded image, the goal direction input, and the heading rate state.

As a typical actor-critic architecture, the neural network outputs a value prediction, and the action-distribution of the stochastic policy. The policy (π) is a softmax output from FC2, while the value prediction (V) is a single linear output. The value prediction is trained to minimize the temporal difference (TD) error [33]. The policy is trained by taking steps in the policy gradient direction [33].

The for the auxiliary depth prediction task, the network has a dedicated layer (Depth FC) with 128 ELUs, which outputs independent softmax predictions, across 8 bins for each of the 64 depth pixels (following the architecture described in [23]). The cross-entropy (CE) error of the depth predictions is minimized, using ground truth data from the simulator. Note that this part of the network is only used during training time.

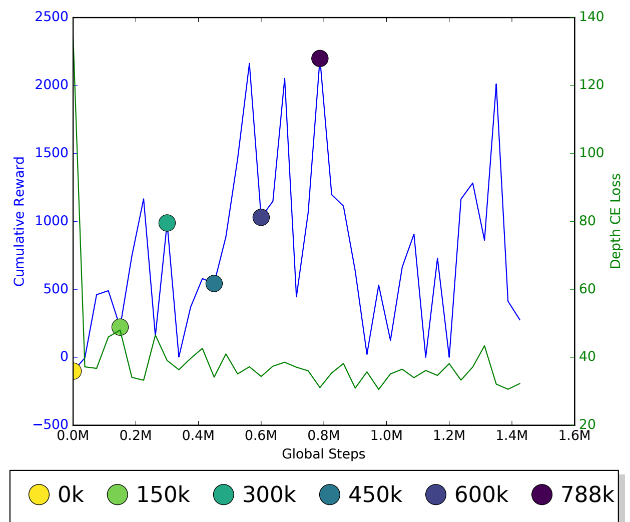

Fig. 6. Learning curves of the main RL task, and the auxiliary depth prediction task.

F. Training Details

We trained the network over 1.4 million agent perceived steps, with 8 agents, using the Adam optimizer[40], with a constant learning rate of 10^{-4}. During the training, the updates were calculated over 10 step long episodes.

The progress of the RL agent was tested with evaluation runs, where we logged the cumulative reward on a fixed setting (same goal and start locations, random initial heading, max 500 steps). For each evaluated model, we averaged the cumulative reward over 10 runs. During these evaluations we also logged the average cross-entropy error on the auxiliary depth prediction task. The learning curves are plotted in Fig. 6, and for a subset of the evaluated models (marked with

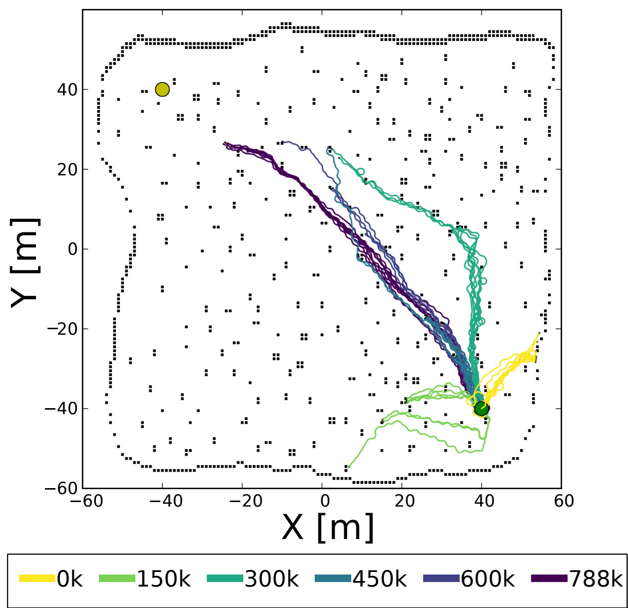

Fig. 7. Trajectories produced by models from different stages of the training.

coloured dots) we also plotted the evaluation trajectories on Fig. 7. The plotted trajectories show that those models, which achieve higher cumulative reward are producing qualitatively better trajectories (more goal directed motion, longer flight paths). This suggests that the reward definition captures the most important aspects of the learning task well.

Fig. 6 clearly shows, that the RL training process is not stable enough. There are significant performance drops, where the agent seems to forget the skills it has acquired already. In the following sections we analyze the best performing model, that was saved at $788k$ global steps.

IV. Experimental Results

We evaluated the performance of the trained VNNM module in the simulated forest environment.

A. Quantitative Evaluation

For the quantitative evaluation, goal and start locations with a distance of $60\ m$ were randomly sampled in the central $80\ m\ \times 80\ m$ area of the environment. We choose the goal distance such that it is significantly longer than the dimensional traversability of the environment ($23.4\ m$), yet it is short enough to capture different parts of the forest (e.g. longer goal distances fit only diagonally to the environment making the agent fly similar paths). Runs were terminated upon crashing, or after taking 1000 steps (which is equivalent to $200\ m$ in the environment), or when the agent arrived within a $2\ m$ radius of the goal location .

For the quantitative evaluation we run 400 trials in the environment. In 43.0% (172) of the trials the module was able to navigate the MAV to the goal location, the average path length to the goal was $67.50\ m$ $(\sigma = 4.77\ m)$.

All unsuccessful runs were due to crashes, and none of the agents reached the 1000 steps episode limit. On average, the unsuccessful runs ended $29.30\ m$ $(\sigma = 17.85\ m)$ from the goal location.

To put these results in context, we compare the performance of the VNNM to a simple straight flying policy. Straight flight in identical conditions on average crashed $38.68\ m$ $(\sigma = 15.59\ m)$ from the goal. In 16.0% (64) of the trials the straight policy was able to reach the goal, therefore in terms of successful runs the VNNM module performed almost three times better. On average, the VNNM got $15.8\ m$ closer to the goal ($\sigma = 2.82\ m$).

In Fig. 8 we plotted the histograms of the distances from the goal location at the end of the path, both for the VNNM module and a baseline straight flying policy (400 trials each). To show that the difference between the two distribution is statistically significant, we conducted a randomized bootstrap test, and concluded that the probability that the two set of samples are coming from the same distribution is negligible ($p \ll 10^{-5}$).

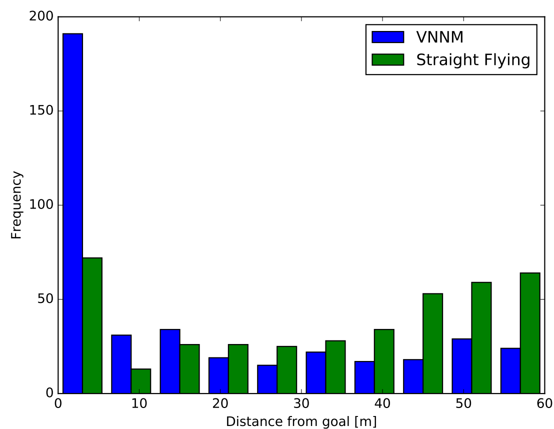

Fig. 8. Histogram of distances from the goal location at episode termination.

B. Qualitative Evaluation

In order to analyze the behavior of the navigation performance, we plotted some selected representative trajectories in Fig. 9. The two upper sub-plots (*a)* and *b)*) are showing trajectories in which the VNNM module was able to navigate the agent to the goal successfully, avoiding multiple obstacles along the way. Two unsuccessful trials are plotted on subplot *c)* and *d)*.

The motion of the agent is mostly rather goal-directed, but in some cases limited detours are made (e.g. *b*). Another common characteristics of the control policy that the VNNM learned that it is commanding right or left angular acceleration actions at almost all steps. This behavior results in sinuous ("wiggling") trajectories, as seen on all subplots of Fig. 9.

In Fig. 10 we demonstrate the ability of the agent to track changing goal direction commands. We changed the goal

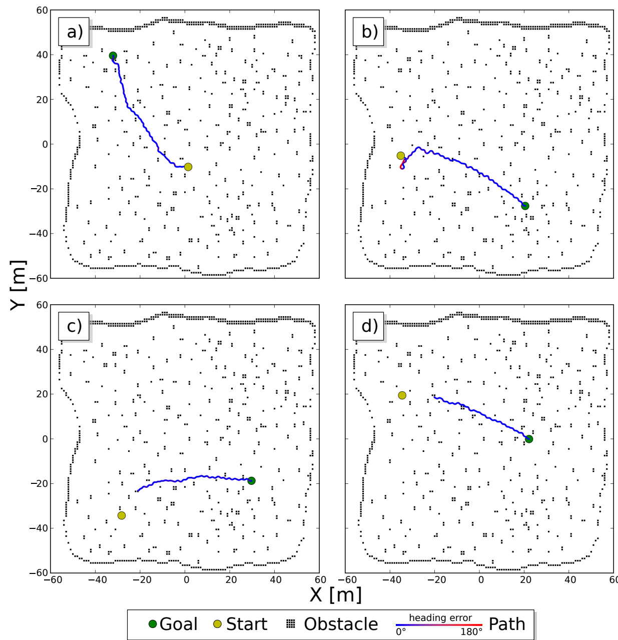

Fig. 9. Selected trajectories from the evaluation.

location every 200 steps, the steps of the trajectory taken under a certain goal setting were colours as the corresponding goal location. The agent clearly follows the actual heading direction input, but makes smaller detours if needed and actively avoids obstacles, and flew almost 200 m.

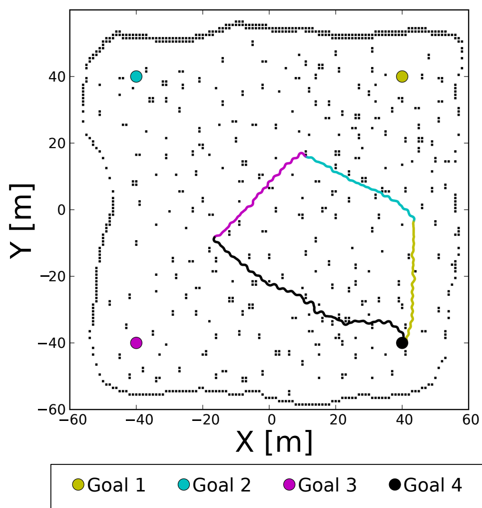

Fig. 10. Trajectory with changing goal locations.

C. Depth prediction

We plotted depth prediction results in Fig. 11 (bottom row), along with the ground truth depth classes (top row), overlaid on the input images.

Both prediction is doing really well on detecting the horizon line: the bottom row of pixels are consistently from a closer depth class, while the rest of the image is mainly detected as the furthest depth class. Since both the attitude (except for heading) and the height over the ground is fixed during the training, the horizon line is in the same position in all observations. This property is therefore easily learned early on the training (see the significant initial drop of the cross-entropy loss in Fig. 6).

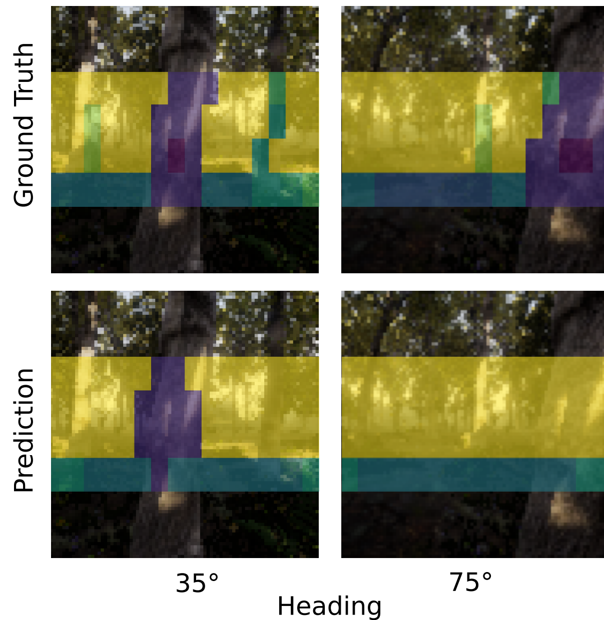

Fig. 11. Depth predictions and ground truths overlaid on the corresponding input images, for two conditions.

We found that the model was able to detect close trees in the middle of the image (a representative example is the 35° heading condition). However, on the side of the image it was not detecting the tree (see 75° heading condition, which image was taken at the same position under a different heading). The network was also unable to detect farther trees.

These performance characteristics may be related to the main RL task. For the goal-directed obstacle avoidance and navigation, the most relevant information is if there is any tree close in front of the agent. More distant trees, or those that are not in front of it are less relevant. Therefore the shared encoder might be trained for detecting close frontal trees with all loss functions, while tree detection on other distances and headings are only trained with the depth loss.

V. Discussion and Conclusions

In this paper we introduced a neural control module, the VNNM, that can extend the capabilities of existing autopilot software stacks with vision-based navigation capabilities. Its interfaces (goal direction input towards higher level modules, and a velocity command to lower level control loops) allow easy integration to conventional MAV control hierarchies.

We demonstrated training this module for autonomous flight in a cluttered natural (forest flight under the tree canopy), using a simulator with high visual fidelity. The

trained VNNM outperformed the baseline straight flying controller, and demonstrated tracking changing goal direction inputs while avoiding obstacles. However, the obstacle avoidance performance did not meet our initial expectations and the requirements for reasonably safe obstacle avoidance.

On the one hand, the asynchronous RL training was not proven to be stable enough. Asynchronous RL depends on having multiple agents acting in parallel. Running multiple instances of our high visual fidelity simulators represents a heavy load, and we were not able to have more than 8 agents acting at the same time given our computational resources. This might well be the reason for the unstability of the training process.

On the other hand, we left most of the hyperparameter-space unexplored. In most cases we relied on good hyperparameters reported in [34] and [23]. However, our training task was different, which might already necessitate different hyperparameters. Furthermore, the reward function and the neural architecture added several other design decisions, which should be further tested in order to achieve better training results.

We demonstrated that a goal directed obstacle avoidance can be learned using RL in a highly realistic virtual environment. However, it is still yet to be explored how well these trained policies generalize to real environments. Incorporating domain adaptation techniques to this training process to close the domain gap between the simulation and the domain of intended use is an interesting direction for future research.

VI. Acknowledgement

The authors would like to thank AImotive Inc. for supporting this research.

References

[1] Abraham Bachrach, Ruijie He, and Nicholas Roy. Autonomous flight in unstructured and unknown indoor environments. 2009.

[2] Slawomir Grzonka, Giorgio Grisetti, and Wolfram Burgard. Towards a navigation system for autonomous indoor flying. *2009 IEEE International Conference on Robotics and Automation*, pages 2878–2883, 2009.

[3] Lionel Heng, Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. Autonomous obstacle avoidance and maneuvering on a vision-guided mav using on-board processing. *2011 IEEE International Conference on Robotics and Automation*, pages 2472–2477, 2011.

[4] Friedrich Fraundorfer, Lionel Heng, Dominik Honegger, Gim Hee Lee, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. Vision-based autonomous mapping and exploration using a quadrotor mav. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4557–4564, 2012.

[5] Linhai Xie, Sen Wang, Andrew Markham, and Agathoniki Trigoni. Towards monocular vision based obstacle avoidance through deep reinforcement learning. *CoRR*, abs/1706.09829, 2017.

[6] Punarjay Chakravarty, Klaas Kelchtermans, Tom Roussel, Stijn Wellens, Tinne Tuytelaars, and Luc Van Eycken. Cnn-based single image obstacle avoidance on a quadrotor. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6369–6374, 2017.

[7] Fereshteh Sadeghi and Sergey Levine. $(CAD)^2$RL: Real singel-image flight without a singel real image. *arXiv preprint arXiv:1611.04201*, 2016.

[8] Dhiraj Gandhi, Lerrel Pinto, and Abhinav Gupta. Learning to fly by crashing. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3948–3955, 2017.

[9] Gregory Kahn, Adam Villaflor, Bosen Ding, Pieter Abbeel, and Sergey Levine. Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation. *CoRR*, abs/1709.10489, 2017.

[10] A. Giusti, J. Guzzi, D. C. Cirean, F. L. He, J. P. Rodrguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. D. Caro, D. Scaramuzza, and L. M. Gambardella. A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667, July 2016.

[11] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *I. J. Robotics Res.*, 32:1238–1274, 2013.

[12] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 1(518):529533, 2 2015.

[13] S. Ross, N. Melik-Barkhudarov, K. S. Shankar, A. Wendel, D. Dey, J. A. Bagnell, and M. Hebert. Learning monocular reactive uav control in cluttered natural environments. In *2013 IEEE International Conference on Robotics and Automation*, pages 1765–1772, May 2013.

[14] Klaas Kelchtermans and Tinne Tuytelaars. How hard is it to cross the room? - training (recurrent) neural networks to steer a uav. *CoRR*, abs/1702.07600, 2017.

[15] Dong Ki Kim and Tsuhan Chen. Deep neural network for real-time autonomous indoor navigation. *CoRR*, abs/1511.04668, 2015.

[16] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011.

[17] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 528–535, 2016.

[18] Gregory Kahn, Tianhao Zhang, Sergey Levine, and Pieter Abbeel. Plato: Policy learning using adaptive trajectory optimization. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3342–3349, 2017.

[19] Gregory J. Stein and Nicholas Roy. Genesis-rt: Generating synthetic images for training secondary real-world tasks. *CoRR*, abs/1710.04280, 2017.

[20] Kevin Lamers, Sjoerd Tijmons, C. De Wagter, and Guido C. H. E. de Croon. Self-supervised monocular distance learning on a lightweight micro air vehicle. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1779–1784, 2016.

[21] Gregory Kahn, Adam Villaflor, Vitchyr Pong, Pieter Abbeel, and Sergey Levine. Uncertainty-aware reinforcement learning for collision avoidance. *CoRR*, abs/1702.01182, 2010.

[22] Nikolai Smolyanskiy, Alexey Kamenev, Jeffrey Smith, and Stanley T. Birchfield. Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4241–4247, 2017.

[23] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *CoRR*, abs/1611.03673, 2016.

[24] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. *CoRR*, abs/1702.03920, 2017.

[25] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J. Lim, Abhinav Gupta, Fei fei Li, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3357–3364, 2017.

[26] J. S. Smith, J.-H. Hwang, F.-J. Chu, and P. A. Vela. Learning to Navigate: Exploiting Deep Networks to Inform Sample-Based Planning During Vision-Based Navigation. *ArXiv e-prints*, January 2018.

[27] Mark Pfeiffer, Michael Schaeuble, Juan I. Nieto, Roland Siegwart, and Cesar Cadena. From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1527–1533, 2017.

[28] Kirk Y. W. Scheper and Guido C. H. E. de Croon. Abstraction as a mechanism to cross the reality gap in evolutionary robotics. In *Simulation of Adaptive Behavior*, 2016.

[29] Tommaso Mannucci, Erik-Jan van Kampen, Cornelis C de Visser, and Qiping Chu. Safe exploration algorithms for reinforcement learning controllers. *IEEE transactions on neural networks and learning systems*, 2017.

[30] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. *arXiv preprint arXiv:1609.01326*, 2016.

[31] EPIC Games Inc. *https://www.unrealengine.com/*, 2017. Accessed: 2017-06-25.

[32] Clint Nous, Roland Meertens, C. De Wagter, and Guido C. H. E. de Croon. Performance evaluation in obstacle avoidance. *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3614–3619, 2016.

[33] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 2 edition, 2017.

[34] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[35] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

[36] Rich Caruana. Multitask learning: A knowledge-based source of inductive bias. In *ICML*, 1993.

[37] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, 2017.

[38] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *CoRR*, abs/1611.05397, 2016.

[39] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289, 2015.

[40] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

APPENDIX I
RL ALGORITHM

Algorithm 1 A3C with auxiliary training, pseudo-code for each worker thread

Require: Master network parameter vector θ_{master}, global step counter T

repeat ▷ Training Loop
 $d\theta \leftarrow 0$ ▷ reset accumulated gradients
 $\theta_w = \theta_{master}$ ▷ synchronize worker parameters from master
 $t = 0$
 Start new episode ▷ reset environment, spawn agent
 Get s_t and o_t ▷ Initial state and auxiliary observations
 repeat ▷ Episode Loop
  Perform a_t according to $\pi(a_t|s_t;\theta_w)$ ▷ stochastic action selection
  Receive r_t, s_{t+1}, o_{t+1} ▷ collecting experiences
  $t \leftarrow t+1, \quad T \leftarrow T+1$
 until Terminal s_t or $t > t_{max}$
 $R = \begin{cases} 0, & \text{for terminal } s_t \\ V(s_t,\theta_w), & \text{otherwise.} \end{cases}$ ▷ bootstrapping value for non-terminal states
 $A_{GAE} = 0$
 for $i \in \{t-1,\dots,0\}$ **do** ▷ gradient calculations
  $R \leftarrow r_i + \gamma R$ ▷ discounted n-step return
  $\delta_i = R - V(s_i;\theta_w)$ ▷ TD error
  $A_{GAE} \leftarrow \delta_i + \lambda A_{GAE}$ ▷ Generalized Advantage Estimation
  $d\theta_w \leftarrow d\theta_w + \nabla_{\theta_w}\left[\log\pi\left(a_i|s_i;\theta_w\right)A_{GAE}\right]$ ▷ policy gradient update
  $d\theta_w \leftarrow d\theta_w + \nabla_{\theta_w}\left[\delta_i^2\right]$ ▷ value function update
  $d\theta_w \leftarrow d\theta_w + \nabla_{\theta_w}\left[\pi\left(a_i|s_i;\theta_w\right)\log\pi\left(a_i|s_i;\theta_w\right)\right]$ ▷ policy entropy regularization
  $d\theta_w \leftarrow d\theta_w + \nabla_{\theta_w}\left[\mathrm{CE}\left(p\left(d|s_t;\theta_w\right), o_t\right)\right]$ ▷ auxiliary classification update
 end for
 Perform asynchronous gradient training step on θ_{master} with $d\theta_w$ using shared ADAM optimizer
until $T > T_{max}$

Part III

Preliminary Thesis - Literature Review

Chapter 2

Deep Learning

Deep Learning (DL) is a very promising approach to Machine Learning (ML), popularized recently after its success at the ImageNet large-scale visual recognition competition [3]. Since then deep learning produced state-of-the-art results in image classification [4], depth prediction [5], and semantic segmentation [6], just to name a few problems with high dimensionality that are relevant for robotics. DL algorithms generally require large training sets, and the training process is computationally intensive. On the other hand, it requires less hand-engineering, and the algorithms are applicable for many different problems. The available computational resources are expected to continue growing, and even specialized hardware for DL is being developed. At the same time more datasets are becoming publicly available. Since DL directly benefits from the increasing computation resources and data, these trends are making it a promising approach for future applications.

This chapter provides an overview of the topic of DL, positioning it within the broader field of Artificial Intelligence research and among other Machine Learning approaches. We also cover the basics of Artificial Neural Networks with related training and regularization issues. Finally the most important practical neural network architectures are briefly described.

2.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) is a field of Computer Science. It is concerned with the study and creation of intelligent agents, with the aim of creating machines that exhibit intelligent behaviour. Kurzweil defined AI as "*the art of creating machines that perform functions that require intelligence when performed by people*" [7].

A major direction in AI research is the so-called **knowledge base approach**: its main focus is to provide machines with a reasoning ability, using logical inference and a database of hard-coded statements in a formal language. One advantage of this approach over machine learning can be its data-efficiency: it does not need multiple samples for learning,

in fact it might learn new concepts by pure logical inference. However, this paradigm has proven to be difficult to scale up [8].

A different approach to tackle challenging AI problems is **Machine Learning**. Its main aim is to enable systems to acquire knowledge by extracting patterns from raw data, in order to provide them with a capability to tackle problems involving knowledge of the real world [8]. ML approaches are easier to scale-up, and some algorithms can be reused in several different domain.

2.2 Artificial Neural Networks

Deep learning is a ML technique which is based on (artificial) Neural Networks (NN). NNs are a family of general parametric function approximators, consisting of multiple layers of artificial neurons interconnected with weights (see Fig. 2.1). Neurons in a layer typically have simple but non-linear activation functions. Every layer transforms the preceding representation to a more abstract level [9]. Deep Neural Networks with multiple layers can learn a *hierarchy of representations*, while the parallel neurons provide *distributed representations* within each layer.

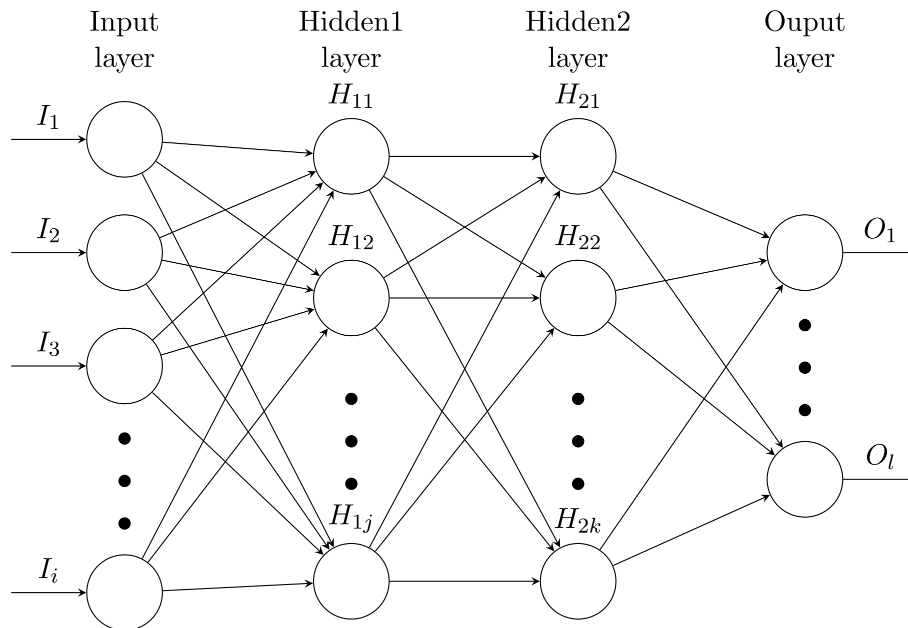

Figure 2.1: Illustration of a simple feedforward neural network with two hidden layers

Neural networks reflect the connectionist approach to AI and cognitive science that draws its inspiration from biological brains and the nervous system. In general, connectionist systems rely on the parallel processing of information with many simple units. This processing is based on statistical properties, instead of using explicit logical rules and serial processing, as in classical systems [10].

The non-linear activations of neurons are a crucial property of neural networks. With linear hidden units, an arbitrarily deep network would be equivalent to a network without

any hidden units. In this case the operation done by the network is repeated matrix multiplication, which can be simplified to a single matrix multiplication. Therefore increasing depth without non-linear activations does not contribute to increasing expressiveness of the network. On the other hand using non-linear activations prevents this simplification, and new layers do increase the expressiveness. However, the difficulty of learning the connections of non-linear hidden units prevented the widespread use of deep networks for decades, as their optimization is a non-convex problem.

Traditionally logistic sigmoid and *tanh* activation functions were popular because of their smooth derivatives. These days most modern deep network use Rectifier Linear Units (ReLU) [11] as hidden units, which has the activation function $g(z) = \max\{0, z\}$. ReLUs are popular because they are well-behaved from an optimization point-of-view [8].

In general, linear models can only learn to represent linear functions. If we consider a two-way classification task, a linear model can only define a hyperplane in the input space for separation. Therefore only linearly separable sets can be discriminated by linear models, which is a serious limitation. On the other hand, linear models are a convenient choice from an optimization point-of-view: the training is a convex optimization, and in some cases even closed form solutions can be found.

A solution to learn non-linear functions while keeping the benefits of convex optimization was to use a non-linear transformation on the inputs. Then a linear model was trained on this new space. For instance in the classification example this had the potential to make the sets of training examples linearly separable. The input space transformation can be a generic nonlinear function, such as Support Vector Machines using the kernel trick [12]. Instead of using a generic nonlinear function, another option is to hand-engineer suitable non-linear functions for specific applications. This manual feature engineering approach led to significant results in many domains (e.g. HOG [13] and ORB [14] features for visual recognition problems), however it required significant domain expertise, and the developed features are usually applicable for only one specific domain [8].

Alternatively, the input transformation can be learned: this approach is called **representation learning**. Representation learning approaches provide the models with the capability to learn from raw, unprocessed data by automatically discovering appropriate representations of the inputs needed to solve the learning task [9]. Although this is a very generic approach the drawback is that the convexity of the optimization is lost. Deep Learning is taking this idea further, essentially doing representation learning at multiple layers, thus learning a whole hierarchy of representations. This hierarchical representation is more powerful, but its optimization is even more difficult.

2.3 Interpretations of Deep Learning

Deep Learning has demonstrated its learning abilities in highly challenging domains with high dimensional inputs. As the number of input dimensions is increasing, the possible different input combinations are growing exponentially. This phenomenon (commonly referred to as the *curse of dimensionality*), is making machine learning on such input spaces difficult, since the available training samples usually cover only a small subset of the possible input combinations. There are two interpretations on why deep neural

networks are suitable for learning even from high-dimensional inputs. The first considers the exponential advantages brought by both the hierarchical structure with several layers, and by the distributed representations within each the layer. The second view is explaining the effectiveness of the learning by the capability of exploring low-dimensional manifolds in the input space, thus reducing the dimensionality of the problem.

2.3.1 Exponential advantages of deep neural architectures

Deep Learning can be interpreted as learning *features at different hierarchical levels*, thus creating representations that are expressed in terms of other simpler representations [8]. In this sense, using a deep neural model implicitly encodes a strong prior belief on the network: the assumption that the goal of the learning can be solved by discovering underlying factors of variation. These factors in turn can also be broken down into even smaller underlying factors of variation recursively, down to the inputs [8]. This composition of several layers of representation (corresponding to different levels of abstraction) is one key to the representational power of deep neural models [9], and their capability to learn complicated concepts.

Another crucial component is the *distributed representation* of information within layers. By using parallel units in layers and creating distributed representations, intermediate representations are shared and reused between many sub-tasks [15]. This can enable deep neural networks to generalize to new combinations of the learned features not seen in the training examples (as an example, having n binary features enables 2^n combinations [9]).

2.3.2 Manifold Learning

Another view explaining the effectiveness of neural networks learning from extremely high dimensional inputs (such as images) is given by the *manifold hypothesis* and manifold learning. A manifold in mathematics is a connected region, which locally resembles an Euclidean space. If the probability density of a high dimensional probability distribution is concentrated around a lower dimensional manifold, then by discovering this manifold the dimensionality of the distribution can be reduced.

From a mathematical point-of-view, images can be considered vectors from a high-dimensional vector space: the pixel coordinate space. Here the number of the dimensions equals to the number of pixels, multiplied by the number of image channels [16]. This would lead to extremely high dimensional problems, if all variations along all dimensions were independent. However real samples are located in an extremely small volume of this high dimensional space. An intuitive example proving this is random generation of images by independently sampling pixels: it is highly unlikely to generate anything close to a real image.

The manifold hypothesis states that real data samples are only from a low dimensional manifold which is embedded in a higher dimension space. Therefore a learning algorithm can efficiently reduce the dimensionality of the problem by discovering manifolds. Deep generative models such as Variational Auto-Encoders (VAE) and Generative Adversarial Networks (GAN) show some indirect evidence that deep learning models are capable of discovering manifolds in the input space [8].

2.4 Training

Training parametric models is basically an iterative process of finding parameters θ for a parametric model $f(x;\theta)$ in order to reduce an appropriate cost function $J(\theta)$. The dynamics of neural network optimization in general is not well understood yet, and still an area of active research (e.g. see [17] [18]). Many of the popular methods (e.g. the momentum method later in this section) are conceptually simple, based on heuristics and intuition. Some gradient-free optimization method, such as genetic algorithms [19] or cross-entropy optimization [20] are a viable alternative for training networks with relatively few parameters. However modern neural networks usually contain parameters in the order of millions, and evolutionary algorithms with inexact gradient calculation are generally considered to be less efficient in this regime [21]. Therefore in this section the focus is on gradient-based optimization, which is the common approach to optimize deep neural networks.

For any optimization problem it is important to have some metrics of the performance. Therefore to measure performance a **cost function** is needed, which is a mapping from the outputs (and from training targets in supervised settings) to a scalar cost. In most cases cost function for machine learning applications can be decomposed as a sum of per-example losses over the training set. Consider a supervised example, where the set of training inputs $x^{(i)}$ and targets $y^{(i)}$ are drawn from the true data generating distribution p_{data}. This training set defines an empirical distribution $\hat{p}_{data}$. The model that is being optimized is a neural network, with θ being the vector of its parameters. Let $L(x, y; \theta)$ denote the per-example loss; then the loss function $J(\theta)$ is given by: [8].

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data}} L(x, y; \theta) = \frac{1}{m}\sum_{i=1}^{m} L(x^{(i)}, y^{(i)}; \theta) \tag{2.1}$$

Note that since the network architecture is kept fixed during the training, the loss is only dependent on the network parameters.

The *backpropagation algorithm* [22] is an important part of all practical learning methods for training large neural networks, which calculates $\nabla_\theta J(\theta)$, the gradient of the cost function with respect to the parameters of the network (θ). It essentially allows information from the cost function to flow backwards to the weights. Backpropagation is simply the recursive application of the chain rule for derivatives: the gradient is calculated iteratively for each weight starting from the output layer, moving backwards in the neural network.

Once the derivatives are calculated, a gradient descent method can take a step in the direction of the derivative, to iteratively decrease the cost and improve the performance. However, calculating the exact gradient over the whole training set before taking a single training step is prohibitively expensive for practical problems where training sets are large. The commonly used *Stochastic Gradient Descent* (SGD) algorithm takes an estimate of the gradient instead, by sampling minibatches randomly from the training set, and taking gradient steps on each minibatch [23]. The estimated value g of the gradient over a mini-

batch is given by:

$$g = \mathbb{E}[\nabla_\theta J(\theta)] = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(x^{(i)}, y^{(i)}; \theta) \tag{2.2}$$

The minibatch size m' controls the trade-off between the variance of the gradient and the required calculations. SGD can significantly speed up the convergence of gradient descent algorithms, as approximate gradients are sufficient for improving the model.

The network weights are then updated by taking a gradient descent step in the (approximate) negative gradient direction:

$$\theta \leftarrow \theta - \epsilon g \tag{2.3}$$

The learning rate ϵ in the above equation is a crucial hyperparameter of the learning process with a significant impact on the convergence of the algorithm. Typically some form of gradually decreasing scheduling of the learning rate is important for efficient training. Instead of predetermined scheduling, a number of algorithms have been proposed to *adjust the learning rate* adaptively during the training process, separately for each parameter. For example RMSProp [24] scales the gradients by an exponentially weighed moving average of past gradients, while Adam [25] calculates an unbiased estimate for both the first (gradients) and second order (velocity) moments.

A commonly used intuitive method to speed up learning is incorporating a *momentum* term in the network update [26]. It can accelerate learning by amplifying small but persistent components of the gradient, which is especially beneficial when the gradient is noisy, or in the presence of high curvature in some directions [27]. The update incorporates a velocity term v that incorporates an exponential moving average of the previous gradients, weighed by the hyperparameter α [8]:

$$v \leftarrow \alpha v - \epsilon g \tag{2.4}$$

$$\theta \leftarrow \theta + v \tag{2.5}$$

This update rule can be seen as taking two steps: one in the gradient direction, with the gradient of the starting point, and another with the momentum accumulated from previous steps. However if the gradient direction is significantly different at the place after taking a momentum step, then moving further in the previous gradient direction will lead to less optimal updates. This issue is addressed by a variation of the momentum method, the *Nesterov momentum* [27]. This method first takes a momentum induced step, then evaluates the gradient at this new location, before taking a gradient descent step with the new gradient.

$$v \leftarrow \alpha v - \epsilon \mathbb{E}[\nabla_\theta J(\theta + \alpha v)] \tag{2.6}$$

$$\theta \leftarrow \theta + v \tag{2.7}$$

The advantage of this method that the calculated gradient can be more relevant. Note that the computational requirements are the same for the two algorithms, since the gradient is only evaluated at one location for both cases.

To conclude our brief overview of neural network training, a further important design consideration that strongly affects the convergence properties is *parameter initialization*. An essential requirement for parameter initialization is 'symmetry breaking' between hidden units. Units receiving the same input have to be initialized with different weights [8]. Otherwise these units would be updated in a same way, wasting computations and expressivity, and preventing developing distributed representations. Therefore usually initializations for weights are chosen from random distributions (typically either uniform or Gaussian) with zero mean, and the variance is set to provide appropriate activation and/or gradient variance in the following layer [28]. Biases and other extra parameters are typically initialized heuristically [8].

Another choice for finding good initial parameters is using pre-trained weights and biases. The first successful attempts for neural network training relied on greedy layer-wise unsupervised pre-training [29]. In this approach the network was first trained on the training set with an unsupervised algorithm, then the learned parameters were used as an initialization for the main supervised training on the same training set. Since then it was demonstrated that given enough training data and sufficient computation resources, random initialization is sufficient. However, initializing with network parameters from supervised training on similar training tasks is still common (see transfer learning in Section 3.1).

2.5 Regularization

Deep models are learned on a training set; however the ultimate goal of learning is to provide good performance on the test set, and on other unseen examples. This capability is called *generalization*. Deep neural networks typically have high model capacity, which makes them susceptible to overfitting on the training data.

Regularization is a family of methods in machine learning aimed at improving the generalization and preventing overfitting in the learned models. It basically trades off increased bias for reduced variance [8]. In practice, the best results are achieved when large neural networks, with huge model capacity are chosen together with the heavy use of regularization methods.

A way of regularizing parametric models is to impose penalties on some norms of the network parameters. The most common form is *weight decay*, which penalizes the size of the parameters by an extra term in the cost function, usually measured by the L^1 or L^2 norm. Weight decay is trying to drive the weights towards zero. Weights which contribute significantly to the reduction of cost function are less affected, while less important weights are driven towards zero more aggressively. The most important difference between using the two norms is that for the L^2 norm, the gradient of the weight decay term is dependent on the weight itself; for L^1 it is constant. Therefore the latter has more significant contribution on weights close to the origin, thus it can result in more sparse solutions [8]. In general, "nets with large weights have more representational power" [30], causing them

to be more prone to overfitting. Therefore restricting the weights to regions closer to the origin helps generalization.

Another widespread regularization method is *early stopping.* It requires a separate validation set of training examples, on which the model is evaluated periodically. Validation error typically has a U-shaped curve when plotted for training steps. First, both the training and validation error decreases, however at some point the model starts to overfit the training data, and performance on the validation set deteriorates [8]. If the training is stopped as soon as the error on the validation set starts to increase, overfitting on training examples can be limited. As stated above, neural networks have greater representational power with large weights. Since networks are initialized with small weights, during the training first simpler hypotheses are explored before more complex hypotheses, which are prone to overfitting [30]. Therefore stopping before the network gets far from the origin and fits a more complex hypothesis can help generalization. From this point of view, L^2 weight decay and early stopping are acting in a similar way. However, the latter has the advantage that is less sensitive to hyperparameters: the optimal amount of regularization is automatically discovered by monitoring the validation performance [8]. On the other hand, early stopping requires setting aside some training data for the validation set.

Learning algorithms can generally benefit from increased number of training examples. *Dataset augmentation* artificially generates more examples from existing ones. This can help the model to be invariant for certain transformations and other factors of variation that are indecisive for the task to be learned [31]. These methods work particularly well for image inputs [3], but the amount of transformation applied has to be tuned carefully (for supervised methods the transformation has to preserve the original labels, i.e. the transformation should not push examples to other categories).

Injecting noise can also be seen as dataset augmentation. Noise can be injected either at the input, at the hidden unit activations, or even to output targets. Using noisy training targets, referred to as label smoothing can account for mislabelled training examples [8]. Noise applied to hidden units can be seen as augmenting the dataset at different hierarchical levels [32]. Finally noisy inputs can be used to increase the noise robustness of the network. In the presence of noisy inputs good performance can only be achieved if the function is relatively insensitive to variations. This encourages finding local minima during the optimization, which is surrounded by flat regions [33].

Ensemble methods are based on the practical finding that different models tend to make different errors; thus by training separated models, the average results can reduce generalization error. Therefore training an ensemble is a simple way to regularize models, on the other hand it is computationally rather expensive both at training an test time. *Dropout* [34] provides a less expensive way to approximate an exponentially large ensemble of neural networks [8]. At training time, input and hidden units are removed from the model with a certain probability, thus essentially training all possible sub-networks jointly. This forces the neurons to learn useful functions in many different contexts, and restricts the co-adaptation of neurons [34].

2.6 Neural Network Architectures

The most simple neural network architectures are fully connected, feedforward networks, also called Multi-Layer Perceptrons (MLP). These are directed acyclic computational graphs, without any feedback from outputs to neurons of preceding layers. Because of the dense connections, the number of trainable parameters grow exponentially with the size of the network. Thus it is difficult to scale them up to large network sizes, or to accommodate high dimensional inputs (such as images).

Parameter sharing is a common way to efficiently scale up for large network sizes: it can greatly simplify the learning problem by forcing sets of parameters to be equal [8]. Parameter sharing is typically used when the input exhibit a certain structure or invariance. In this section we describe two popular neural network architectures: Convolutional Neural Networks, which work on grid-like inputs and share parameters through different positions; an Recurrent Neural Networks that share the same parameters through time.

2.6.1 Convolutional Neural Networks

Convolutional Neural Networks [35] are a family of neural networks architectures that are particularly efficient in processing data with a known grid-like structure [8], such as images or time series. Instead of general matrix multiplication and dense connections CNNs feature a convolution operation with a kernel. CNNs are directly inspired by neuroscience, and exhibit a very similar structure and working principles to the V1 visual cortex in mammal brains.

Convolution in general is a fundamental operation on two real valued functions in signal processing and analysis; usually the first argument of this operation is referred to as the input and the second as kernel. In this context however convolution is discrete: the input is a tensor of data (inputs or hidden activations) and the kernel is a tensor of parameters adapted by the learning algorithm [8]. Also the kernel flipping is usually omitted from implementations, and in most cases convolution is done in a parallel manner on multiple input channels and kernels.

Figure 2.2 illustrates convolution with a single filter. The input image is 7×7 pixel, and extended to 9×9 with zero padding. Zero padding gives control over the output size, sometimes it is used to preserve the input dimensions. The *receptive field* in this case is 3×3 patch of the input image. The distance between subsequent input patches is defined by the *stride*, which is 2 in our example. A *convolutional filter* contains weights for every pixel in the receptive field, thus it is also 3×3. By multiplying every pixel in the receptive field with its respective weight in the filter and summing up the products, we get one element of the output map. The outputs are usually fed through an activation function (e.g. ReLU, not illustrated in the figure). As we see in this example, further differences between general convolution and the convolution operation used in neural networks is that it might involve downsampling by using a stride other than 1, and it can preserve the input dimensions using zero padding.

Two properties make CNNs very efficient for such inputs: local connectivity and shared weights. *Local connectivity* is sufficient for understanding an input patch, since in grid-like inputs local variables are often highly correlated. These correlated regions are “forming

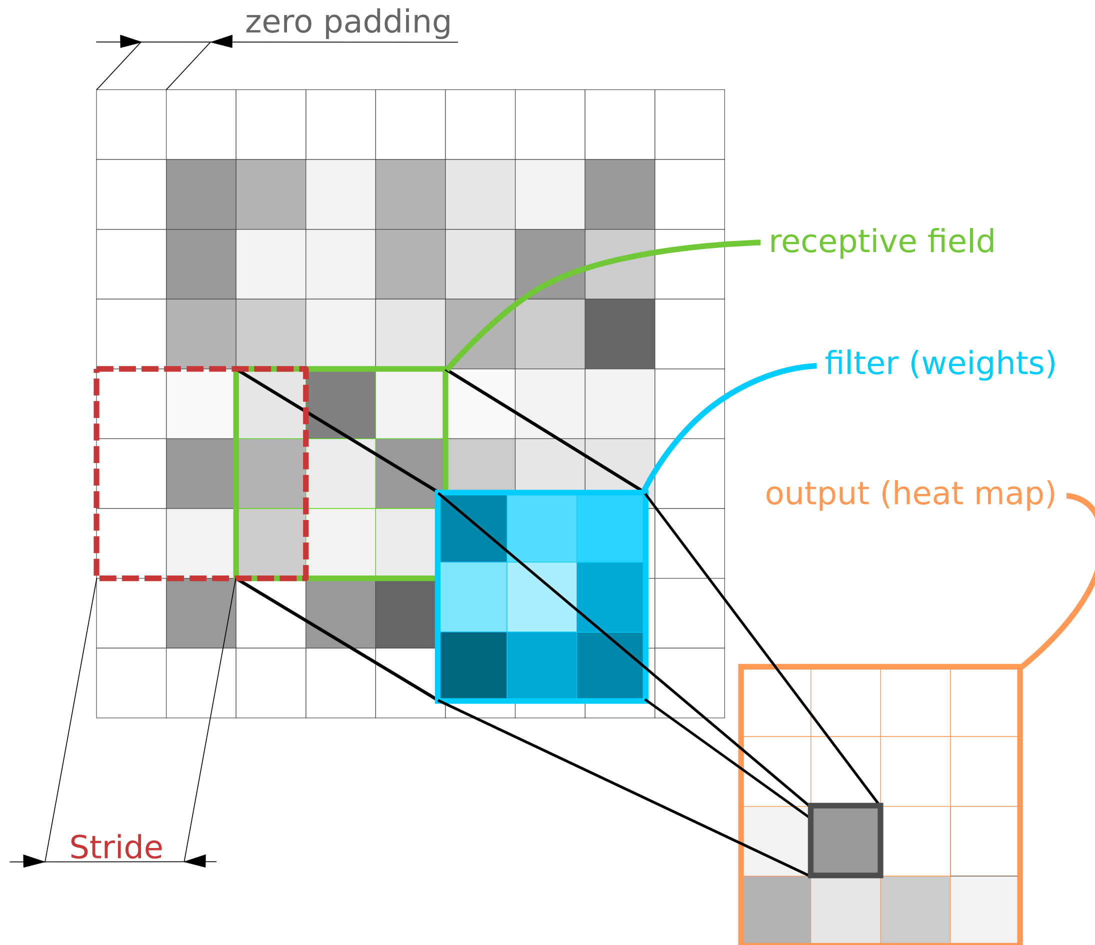

Figure 2.2: Convolution with a single filter

distinctive local motifs that are easily detected" [9] (such as oriented edges, end-points or corners in images [36]). Therefore dense connectivity (connecting an output neuron to *all* input neurons) is not required in lower layers, and local connectivity greatly reduces the number of connections and parameters. Furthermore, *weights can be shared* between different input patches. This is because the same patterns can appear in various parts of the input, in other words the local input statistics are invariant to translation [9]. An object or motif should be detected the same way irrespective its location. Therefore a detector learned on one part of the input might be useful in other regions as well, thus justifying weight sharing.

Besides the convolutional layers another common component of CNNs are pooling layers. Pooling reduces the size of the output layer by summarizing the statistics of nearby outputs of the preceding layer. This is most commonly done by calculating the average output (average pooling), or by choosing the maximum output (max-pooling) from a small neighbourhood of neurons. Besides efficiently reducing the output dimensions, pooling also provides small-scale translation invariance. [8].

Even though successful applications of CNNs are dating back to the 90's [36], the wide-

spread use of CNNs started in 2012, when AlexNet [3] won the ILSVRC challenge by a huge margin. Since then numerous extensions were developed, and the state-of-the-art has been pushed further with new improvements being published in a fast pace. The general trend is that networks are becoming deeper and larger. Just to name a few examples from recent developments, VGG Net [37] for example increased the number of layers (up to 19) while keeping the number of parameters relatively low, by using only 3x3 convolutions. The GoogLeNet [38] went over 100 layers, and introduced the inception module, where the network can do pooling and convolution with different kernel sizes in parallel. ResNet [4] proposed using skip connections, to help the backpropagation of gradients in very deep networks, performing deep residual learning with 152 layers.

2.6.2 Recurrent Neural Networks

Recurrent Neaural Networks (RNN) are computational graphs, which (as opposed to feedforward networks) contain cycles. The recurrent computations can be unrolled to a repetitive graph, thus RNNs can be interpreted as models sharing parameters across time. Their main use is in sequential data processing [8].

The recurrent connection can connect hidden units to other hidden units, or it can connect previous output to the hidden units. This latter case, output recurrence is significantly easier to train. While normally the recurrent connections are feeding the network with its own previous predictions, during training the known correct outputs can be fed back instead. This method (known as *teacher forcing* [39]) makes training easier, and and allows for decoupling training steps, which makes the training more efficient. Teacher forcing is not applicable in case of hidden unit recurrence, as the correct outputs of hidden units are normally unknown.

The conventional method of learning in networks with recurrence between hidden units is Backpropagation Through Time (BPTT), which is simply the application of the general backpropagation algorithm [22] to the unrolled computational graph. It is however still very difficult to learn long-term dependencies with BPTT: as the same weight occurs many times, its effect is exponential. Therefore gradients propagated through many steps may either vanish or explode [40].

One solution used by Echo State Networks [41] to overcome the exploding/vanishing gradient problem is setting the hidden-to-hidden weights upfront to values capturing past inputs sufficiently, and fixing these weights. Thus in these models only the output weights are learned, which imposes serious limitations on the their expressivity. Another way of addressing the problems of the gradient backpropagation is adding 'skip-connections' (connections with unit weight over multiple time-steps, also referred to as 'shortcuts'), or completely removing length-one connections. These solutions effectively decrease the number of steps for backpropagation, thus mitigating the exploding/vanishing gradient problem [8].

The most effective RNN models used in practical applications are *gated RNNs*. Gated RNNs are using Gated Recurrent Units (GRU). The most popular GRU is Long Short-Term Memory (LSTM) [42] units. LSTM units have internal recurrence, a self-connection with weight 1. This self-connection can enforce constant error flow for long durations.

Gate neurons are used to access the state of the LSTM unit, or to erase or update its content. The weights governing the use of these gate neurons are learned.

Chapter 3

Domain Adaptation

Ideally machine learning algorithms are trained and tested on the same domain, and the samples used are coming from a fixed distribution. However in many cases we might wish a model, which was trained on a *source domain* to be used in a significantly different *target domain*. The shift between training and test distributions is referred to as *domain discrepancy*, and this problem is addressed by *domain adaptation* techniques [43].

The need for domain adaptation usually arises in the case when in some domain the available training examples are abundant or easy to acquire, while the domain of intended use has less or no labeled examples available. Using synthetic images to train an agent for operating in the real environment is a typical case of domain adaptation. The domain discrepancy between virtual and real domains is often called the *reality gap*.

Within supervised learning, domain adaptation settings might differ in the availability of labelled examples from the target domain. In the *supervised domain adaptation* setting labels are available for all target domain samples. For the *semi-supervised domain adaptation* only a subset of target samples are labelled (possibly only for a subset of categories). Finally, *unsupervised domain adaptation* is using only unlabelled samples from the target domain [44]. For an unsupervised learning setting, only unsupervised domain adaptation methods can be used as both the source and target domain samples are unlabeled (applicability of domain adaptation methods for Reinforcement Learning is discussed in Section 3.4).

The domain adaptation problem can be framed as a regularization problem: the aim is to prevent overfitting on the source domain, and ensure generalization to the target domain. A natural question to ask: under what conditions can we expect a model to perform well on the target domain [45]?

First, if no ideal model is capable of performing well on both the source and target domains, we cannot expect good performance regardless the domain adaptation. If the underlying function that we are trying to fit with our model (for example a labeling function in a supervised case) is fundamentally different in the two domains, then knowledge learned in the source domain is simply not applicable on the target domain. Since domain

adaptation is only used when this contribution to the generalization error is expected to be small (i.e. the task is similar in the two domains), this problem is not addressed by domain adaptation methods.

Therefore the main reason why test error on the target domain is normally higher is the discrepancy between the target and the source domain distributions. Evaluation of the differences in distributions is therefore a key part of domain adaptation. Determining whether two sets of observations are coming from the same underlying distribution is referred to as the two-sample problem in statistics.

In this chapter the main focus is on unsupervised domain adaptation, as one goal of this research is to enable the transfer of knowledge learned in a virtual domain to the real world robots. The two other settings are only addressed briefly. Also, some prior work was only applicable to shallow networks; here we focus on solutions for deep neural networks instead. Furthermore, we shortly discuss a different research direction, which sidesteps the domain adaptation problem: domain randomization. Finally, we discuss some domain adaptation results in the field of Reinforcement Learning, where not only the input distributions, but the underlying dynamics and Markov processes might differ (i.e. a domain might not only appear different, but may also behave differently).

3.1 Supervised Domain Adaptation

Supervised domain adaptation is a rather common practice in deep learning. In fact most deep convolutional networks are trained first on large image datasets (such as ImageNet), and then finetuned for the specific task. This procedure is called *transfer learning*, and normally it does not involve any explicit domain adaptation. In this setting, the first phase is essentially just providing a very good initialization for training on the target domain.

The reason why transfer learning is so useful in many cases lies in the fact that features, which networks are learning can be rather general and applicable for a variety of tasks. This is especially true in the first few layers in a deep network. A usual phenomenon in deep convolutional networks trained on real images is that the features learned in the first layer resemble Gabor-filters and color blobs, regardless of the datasets and training objectives [46]. This already suggests that transfer learning can be a valid strategy for domains where available training data is not sufficient for training deep models.

Bengio et al. further investigated the generality vs. specificity of layers in deep networks, by studying the effectiveness of transferring (and finetuning) one or more layers in a convolutional network [46]. In their experiments the authors pre-trained a network on ImageNet examples from a subset of categories, and reused some layers in another network, which was trained on a different subset of ImageNet categories. The main finding was that the first layers are rather general, and pre-training those layers on a similar domain improves the performance of the network on the target domain. This was due to the fact that these general layers were able to benefit from the increased amount of training data. On the other hand, task specificity increases in the final layers, which deteriorates transferability of those layers. These results justify the common strategy of replacing only the last layers with new random initialized layers when transferring a network to the target domain or target task.

Sometimes parameters of the transferred layers are frozen, and just the new layers are involved in learning on the target domain. However, another important finding in [46] was that allowing the transferred layers to learn, can improve performance. This is partly because this way the transferred layers can still improve on the target domain. Also when neighbouring new and transferred layers can learn together, they can co-adapt better than with one layer being frozen [46].

On the other hand, during the initial phase of the training with random top layers, the network outputs are random and therefore the backpropagated errors are high. This might lead to unnecessary big changes in the transferred layers, while with converged top layers only slight changes for co-adaptation would be required. Therefore it is also a common practice to set different learning rates for the transferred layers and for the new, random initialized layers. The new layers are therefore more flexible to learn, while changes in the transferred layers are less significant. Still, transferred layers are able to adapt to the new layers and to the new task.

3.2 Semi-supervised Domain Adaptation

In semi-supervised domain adaptation labels are available only for a subset of target samples. Furthermore, target domain labels are not necessarily available for all the categories, possibly only a subset of categories have labeled samples.

A trivial case of self-supervised adaptation is just using the available labelled target domain examples together with the source domain examples in the same training procedure. The remaining unlabelled target samples are then used with any unsupervised domain adaptation method.

Tzeng et al. [47] proposed a method for semi-supervised domain adaptation inspired by network distillation [48], which is a model compression method. Normally supervised neural network training is using hard, one-hot label vectors (i.e. a training sample is either 100% ‘dog’ or 100% ‘car’). However the network output is typically a softmax probability distribution over all categories, and normally the probability is not concentrated on a single class (e.g. the prediction for an image of a dog might be just 80% ‘dog’, and also some probability is given for other furry, four-legged animals, such as ‘cat’ or ‘bear’). Thus softmax output distributions of a trained classifier network contain important semantic information (‘dog’ is closer to ‘cat’ than to ‘car’). Network distillation is making use of this semantic information. Instead of training on the original hard, one-hot label vectors of the training set, soft labels are used. These soft labels are softmax output vectors of a trained larger network (or ensemble of networks). While training on the soft labels the network is learning about multiple categories simultaneously, utilizing the semantic relations discovered by a larger network. In an ideal case, this way the performance of the large network can be achieved with a smaller network and less computations.

Sparsely labeled target examples can be used in a similar way to align semantic structure across domains. First, the network is trained on the source domain. Then the one-hot labels of target domain examples are changed to corresponding softmax network outputs, and these examples are used for training the network further [47]. Compared to simple transfer learning (i.e. using the available examples with the original hard labels), now

the network can also learn about categories for which no labelled target examples are available. This is possible because the semantic relations are assumed to be the same between categories across domains.

3.3 Unsupervised Domain Adaptation

Unsupervised domain adaptation is concerned with the case when no labels are available for samples from the target domain. Differences in input distributions lead to differences in the activation distributions (also called feature distributions) in the hidden layers of the network. Therefore a common strategy for unsupervised domain adaptation for deep neural models is matching the source and target feature distributions at some layers of the network. An example is given in Fig. 3.1, where hidden layer distributions are visualized for the source and target domain samples, with and without domain adaptation. Here the source domain is a training set of synthetic numbers, while the target domain is the Street View House Numbers dataset [49]. Ten distinct clusters can be recognized in the visualizations, corresponding to the then digits of the decimal numeral system. The separate feature distributions without domain adaptation are apparent in the figure, while the adapted neural network produces similar feature distributions on both domains.

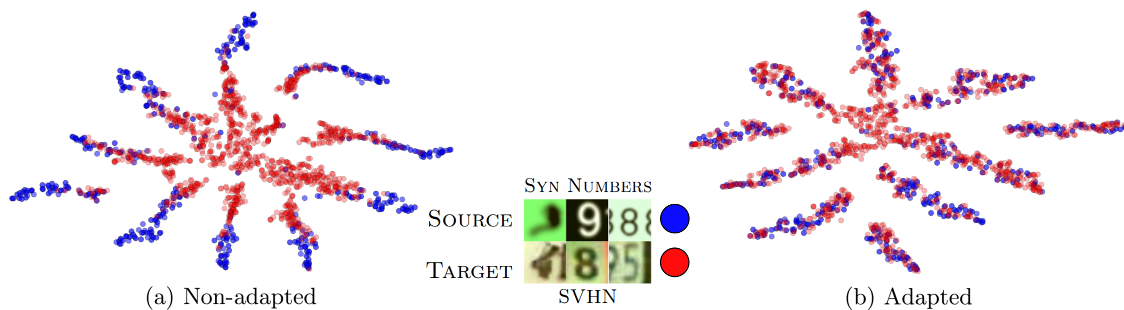

Figure 3.1: t-SNE visualization [50] of a CNN hidden layer feature distributions with (b) and without (a) domain adaptation (source: [44])

As it was stated earlier, matching feature distributions is a central concept of unsupervised domain adaptation. The common hypothesis of most domain adaptation methods is that if feature distributions are matched in the network, then good performance can be achieved on both domains. Feature matching can be done either with explicit transformations of the distribution statistics, or by including an additional loss term during training. This loss is usually based on some distribution distance measure, which penalizes differences in feature distributions. Essentially this loss encourages the network to learn a transformation to an invariant feature space, where samples from both domains result in similar distributions.

Measuring the distance between the domain distribution is key element of unsupervised domain adaptation algorithms. In the following section we review some common approaches: matching correlation distances, methods minimizing the Maximum Mean Discrepancy between feature distributions, and other methods that are using a domain classifier to measure domain discrepancy.

3.3.1 Aligning second-order Statistics

A simple and intuitive way of domain adaptation is simply matching feature distributions by aligning second-order statistics in an unsupervised manner. This can be done with rather straightforward procedures, therefore these methods are sometimes referred to as 'frustratingly easy' domain adaptation methods. One algorithm based on this principle, *Correlation Alignment* [51], starts with normalizing the feature distributions for zero mean and unit standard deviation. Then covariances are matched by first whitening (uncorrelating) the source feature distribution, and then recoloring with the covariance of the target distribution statistics. Opposite transformation is also possible, matching the target feature distributions to the source distribution, however the authors found it to be less effective in practice [51].

3.3.2 Maximum Mean Discrepancy based methods

A commonly used distribution distance metric in machine learning is Maximum Mean Discrepancy (MMD) [55]. MMD measures the domain discrepancy with a function, which has different expectations when evaluated on different domains. MMD is the distance between these mean embeddings. With p and q denoting Borel probability distributions and for a function class $\mathcal{F}$, which consist of functions mapping sets to real numbers ($f : \mathcal{X} \to \mathbb{R}$), MMD is formalized as [55]:

$$\mathrm{MMD}[\mathcal{F}, p, q] = \sup_{f \in \mathcal{F}} \Big(\mathbb{E}_{x \sim p}[f(x)] - \mathbb{E}_{x \sim q}[f(x)] \Big) \tag{3.1}$$

For two set of samples X and Y, consisting of m and n samples drawn from distributions p and q respectively, a biased empirical estimate of the MMD can be given by:

$$\mathrm{MMD}[\mathcal{F}, X, Y] \approx \sup_{f \in \mathcal{F}} \left(\frac{1}{m} \sum_{i=1}^{m} f(x_i) - \frac{1}{n} \sum_{i=1}^{n} f(x_i) \right) \tag{3.2}$$

In practice $f(x_i)$ can be simply the activations in a network at a given layer. In other cases the function f is chosen to be a (positive definite) kernel function, or an optimized mixture of kernels that maximizes the MMD.

The common neural network training loss can be augmented with an MMD dependent term. Backpropagating this error can directly force the network to match feature distributions for target and source domain samples. This matching is improving feature transferability and thus the performance of the network on the new domain. At the same time, the other original terms of the loss function are making the representation discriminative (thus useful) for the original learning task. The effect of the MMD dependent loss term can be controlled by a hyperparameter in practice. Too low MMD loss would not affect the learned representations. The other extreme, a too strong term might lead to degenerate representations, forcing representations of different categories too close together. This may significantly degrade performance on the original learning task [52].

Some approaches directly include an optimization step in the training process, which aims to maximize the measured MMD by the choice of the function f [53], while other domain

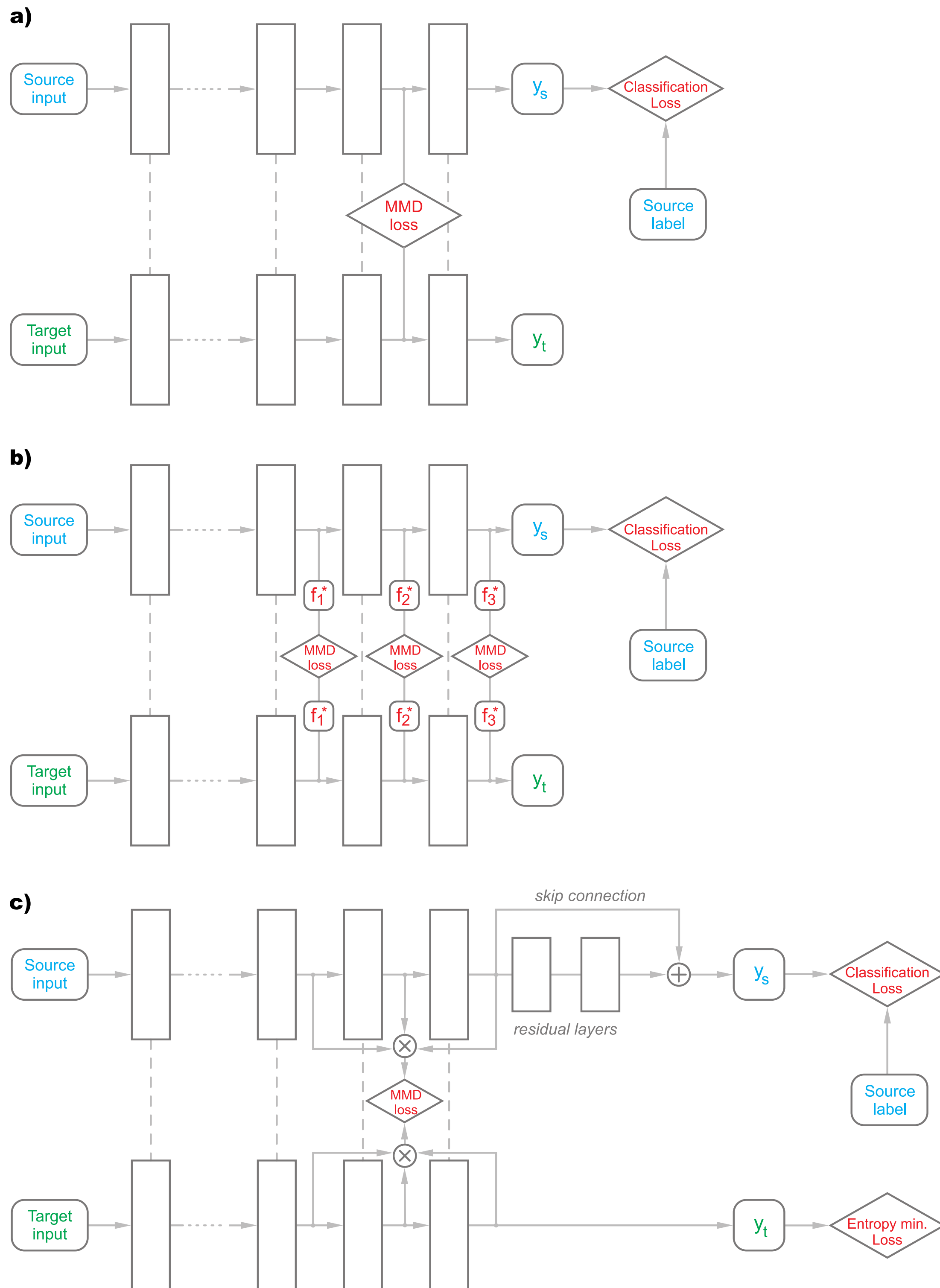

Figure 3.2: Network architectures for different MMD based domain adaptation methods. a) Deep Domain Confusion architecture [52], b) Deep Adaptation Network [53], c) Residual Transfer Network [54]

adaptation methods simply approximate the MMD directly on hidden representations of a layer in a neural network. **Deep Domain Confusion** [52] is an example of the latter. This method introduced an additional, fully connected layer to the original AlexNet convolutional architecture (referred to as the 'adaptation layer', a normal fully connected layer where feature distributions are matched); and an associated domain confusion loss. This loss is based on the MMD calculated on the activations of the adaptation layer. The network architecture is depicted in Fig. 3.2 a). Note that the same layers are used on both domains (dashed lines symbolize weight tying). By trying to explicitly reduce the domain discrepancy in the adaptation layer, the domain confusion loss encourages learning representations in the adaptation layer that are invariant to the input distribution change between the domains. Using an adaptation layer leads to two design choices: choosing the position (depth) and the size (width) of the adaptation layer. In [52] both are optimized using MMD as metric: the first is chosen by evaluating MMD with different adaptation layer placements. For determining the optimal size of the adaptation layer, the authors proposed a grid-search where networks are finetuned with different adaptation layer dimensions and evaluated based on MMD.

Opposed to Deep Domain Confusion, the **Deep Adaptation Network** architecture [53] (in Fig. 3.2 b) is trying to match the representations not just in one, but in multiple layers. Since in deep neural networks lower layers tend to learn more general features, while higher layers are more domain and task specific [46], in Deep Adaptation Networks the domain discrepancy is reduced in these task specific layers independently (on the figure, MMD loss is calculated at the three final layers). Instead of calculating MMD directly on activations in a given layer, Deep Adaptation Networks use a multi-kernel approach for measuring the domain discrepancy, feature distributions are evaluated with a mixture of kernels. Since the mixture of kernels is influencing the testing power of MMD, the kernel for evaluation is optimized (in the figure, f_i^* stands for the optimal kernel that maximizes MMD at layer i). Therefore training of the full architecture is an alternating optimization, with iterated optimization of both the evaluation kernels (for maximizing MMD) and the network (minimizing MMD).

Inspired by the recent success of residual learning[1] [4], **Residual Transfer Networks** (Fig. 3.2 c) [54] relax the usual assumption that the classifier function has to be the same on the two domains. Instead it assumes that the two classifiers (c_S and c_T) only differ in a small perturbation term (Δ_c). Therefore the source classifier activations are given by the element-wise sum of the target classifier activations plus a residual function.

$$c_S = c_T + \Delta_c \tag{3.3}$$

In our example architecture there are two extra layers for the source domain classifier to learn the residualΔ_c. This residual function is learned using the supervised examples, and the target classifier is refined using entropy minimization of the class-conditional distribution on target domain samples.

[1] Residual learning makes use of a deep neural networks containing shortcut (skip) connections. Just like in RNNs, these skip connections are making error backpropagation simpler, thus allowing training deeper networks than before.

With a block of ordinary neural layers ($F(.)$) the goal is to learn a desired mapping ($H(x)$, $H(x) \approx F(x)$). The basic building block of a residual network contains one or more layer, and the inputs (x) are directly 'feedforwarded' to the outputs. Thus the mapping learned by the layers is a residual function ($H(x) \approx x + F(x)$)

In Residual Transfer Networks, simultaneous adaptation of multiple layers is done by fusing the features of multiple layers using tensor products (which is a generalization of the Kronecker/outer product), and then performing feature adaptation on these fused features with a single MMD loss (as seen in Fig. 3.2 c) [54]. Again, the evaluation kernel f can be optimized for MMD.

3.3.3 Domain adversarial training

Another way to assess domain distribution discrepancies is training a domain classifier, and measuring its performance. If the classification task is easy, then the distributions are significantly different. On the other hand, for similar distributions the classifier has a more difficult job, resulting in higher error rates. Therefore the error of a domain classifier can be used to assess distances between distributions.

Using a classifier in this manner to assess domain discrepancy can outperform methods based on some form of a metric loss (e.g. MMD). In some cases there might be subtle, but consistent differences that does not correspond to large metric distances. As an example, consider images with watermarks: the watermark only changes pixel intensities in a few pixels. These differences might not correspond to a significant change in a metric space, still a classifier might be able to spot the pattern and distinguish between images with or without watermarks.

Domain adversarial training [44] is a domain adaptation methods that jointly trains a domain classifier together with a base network, in an adversarial manner. The domain classifier is trying to get better at recognizing the input domains from the activations of a hidden layer. The base networks is trying to build a representation which is indiscriminative for the domains, to fool the domain classifier. At the same time the base network also keeps learning on the main training task. The hypothesis is that if the hidden features for inputs from the source and target domains are matched, then based on those features domain discrimination is not possible. Therefore the rest of the network is not affected by the changes in the input distribution, and the same network can be used on the source and target domains as well [44]. Thus the ultimate goal of the training is *learning a representation that is discriminative with respect to the original task, but indiscriminative for the domains.*

In this section first we introduce the theory of assessing distribution distances with a classifier. The idea of domain adversarial training is directly related to Generative Adversarial Networks [56] (GANs), therefore we review the basic working principles of GANs briefly. Then we compare the common architectures and training objectives used for Domain Adversarial training.

H-divergence

The concept of measuring domain discrepancy with a classifier is formalized in [45] by the notion of $\mathcal{H}$-divergence, which formally assesses domain discrepancy by the separability of the samples. Let $\mathcal{H}$ be a hypothesis class of binary classifiers, then $\mathcal{H}$-divergence is

defined as:

$$d_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) = 2 \sup_{\eta \in \mathcal{H}} \left| \mathbb{P}_{x \sim \mathcal{D}_S}[\eta(x) = 1] - \mathbb{P}_{x \sim \mathcal{D}_T}[\eta(x) = 1] \right| \tag{3.4}$$

$\mathcal{H}$-divergence is high, if there exist a binary classifier, which can distinguish between samples drawn from the source and target distributions, assigning high values either to the target or source domain samples. Empirical $\mathcal{H}$-divergence can be computed from finite samples by finding a classifier which attempts to separate samples by sources. For a symmetric hypothesis class $\mathcal{H}$ it is calculated as [45]:

$$\hat{d}_{\mathcal{H}}(S, T) = 2 \left(1 - \min_{\eta \in \mathcal{H}} \left[\frac{1}{m} \sum_{\boldsymbol{x}:\eta(x)=0} I[\boldsymbol{x} \in S] + \frac{1}{n} \sum_{\boldsymbol{x}\eta(x)=1} I[\boldsymbol{x} \in T] \right] \right) \tag{3.5}$$

where $I[a]$ is a binary indicator function which is one if a is true, and 0 otherwise. A way to find η which minimizes the expression in the square bracket is to assign all source examples with 1 and all target examples with 0 labels. Then a classifier can be trained on this supervised problem to discriminate between source and target distances. Note that despite having domain labels, this setting is still unsupervised domain adaptation (see definitions at the introduction of this chapter).

Generative Adversarial Networks

Generative Adversarial Networks [56] (GANs) are a family of generative models. Generative models are aiming to learn a data generating probability density function p_{data}, in order to generate samples that are similar to training examples coming from p_{data}. Adversarial training jointly optimizes two networks: a discriminative model D and a generative model G. D is trained to estimate the probability of an input sample coming from the training data. At the same time, G is trained to maximize the probability of D making a mistake by falsely identifying generated samples as samples coming from the training set. Therefore the two networks are working against each other, and both are getting better as a result of this joint training.

The joint optimization of the two network is a minimax game with the optimal solution being at a saddle-point, where the discriminator has a maximum and the generator has a minimum. The value function of the minimax game is:

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{data}(\boldsymbol{x})} [\log(D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_z(\boldsymbol{z})} [\log(1 - D(G(\boldsymbol{z})))] \tag{3.6}$$

where $p_z(\boldsymbol{z})$ is a distribution that provides random noise input for the generator.

Rather than matching a density function, this concept is used in the domain adaptation setting to match the hidden representations of a network over the different training domains. Therefore instead of a generator network, a deep representation learning model (a feature extractor) is trained. It is learning to remove domain dependent variation from the hidden representation, while keeping other important factors of variation that are important for the original learning task. This way the network can learn to perform the same task on both domains ideally at the same level.

Network Architecture

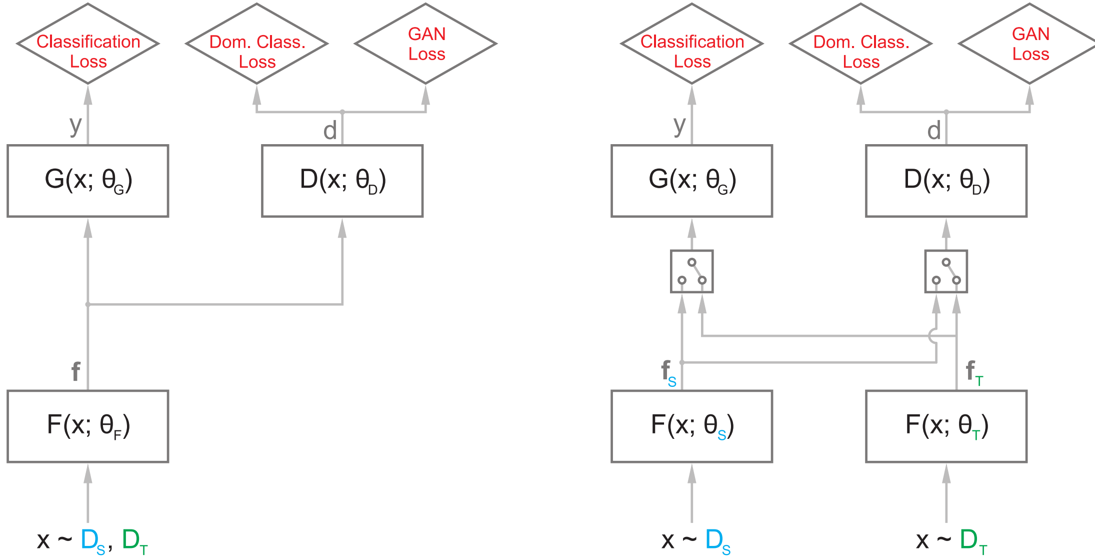

Figure 3.3: Symmetric (*left*) and asymmetric (*right*) network architectures for Domain Adversarial training, with the associated loss functions (classification, domain confusion and GAN losses)

The basic *symmetric architecture* of domain adversarial training (as seen on the left side of Fig. 3.3) can be described as a base network and a domain classifier sub-network that is branched out from an intermediate layer. A domain classifier is trained on this intermediate representation of the base network. We can divide this architecture into three sub-networks and corresponding parameter sets:

- a shared feature extractor $F(\boldsymbol{x}; \theta_F) = f$ that outputs the feature vector f
- the domain discriminator $D(F(\boldsymbol{x}; \theta_F); \theta_D) = d$ that predicts a domain label d based on the feature vector f
- and a task-specific network, $G(F(\boldsymbol{x}; \theta_F); \theta_G) = y$, which can be either a classifier or any other regressive model.

Another architecture for domain adversarial training is maintaining separate feature extractors for both domains [57], $F_s(\boldsymbol{x}; \theta_F^s)$ for the source and $F_t(\boldsymbol{x}; \theta_F^t)$ for the target domain respectively (as seen on the right side of Fig. 3.3). In this *asymmetric* domain adaptation case, the target feature extractor is initialized with the trained source feature extractor. Then, only F_T is trained with domain adversarial loss from the domain classifier; F_S is fixed. The domain classifier D receives either source domain inputs through F_S or target domain inputs through F_T. To confuse the domain classifier, F_T is trained to match the feature distribution of F_S. Once the domain adversarial training has converged, G can be used with features from F_T on the target domain.

Instead of learning a shared feature extractor (which maps both domains to an identical feature space), the asymmetrical architecture adapts only the target feature extractor to

match the source feature distribution. This is generally a more flexible learning paradigm, as it allows both feature extractors to remain domain specific [57], while the symmetric architecture trains a universal feature extractor. The asymmetric adaption is also considered to be an easier task than finding a common feature space, and therefore may yield better results than using a common feature space that ignores the differences [51]. However, this architecture is less suitable for joint training, as it requires a fully trained source feature extractor for initialization.

Adversarial Training

In this section we review different formulations of GAN loss functions, which can be used to train the above introduced network architectures.

The general GAN minimax value function (Eq. 3.6) can be reformulated for the domain adaptation case:

$$\min_F \max_D V(F, D) = \mathbb{E}_{x \sim \mathcal{D}_S} [\log(D(F(x))] + \mathbb{E}_{x \sim \mathcal{D}_T} [\log(1 - D(F(x)))] \tag{3.7}$$

Based on this, the **minimax loss** functions for the domain classifier and for the feature extractor can be thus given as:

$$J^{(D)} = -\mathbb{E}_{x \sim \mathcal{D}_S} [\log(D(F(x))] - \mathbb{E}_{x \sim \mathcal{D}_T} [\log(1 - D(F(x)))] \tag{3.8}$$

$$J^{(F)}_{minimax} = -J^{(D)} \tag{3.9}$$

Here the loss function of the feature extractor is exactly the opposite of the loss function of the domain discriminator.

These equations are used in the **reverse gradient** domain adaptation method [44]. This method backpropagates the classifier error, and flips the sign of the gradient at a *gradient reversal layer* between the domain classifier and the feature extractor. This way their objectives are opposite: while the domain discriminator does gradient descent on the domain classification error, the feature extractor does gradient ascent.

The problem of this approach is that if the domain classifier is able to predict the labels correctly, then the backpropagated gradient vanishes. This case is typical early on the training, as at the beginning the domain distances are usually high, which makes classification easy. This situation makes it even more difficult for the feature extractor to learn: on top of the difficult distribution matching task it does not even get proper gradients.

Therefore Eq.3.9, which is explicitly trying to minimize the probability of a good answer is often replaced with another cost, the **GAN loss**. This new loss is trying to force the feature extractor to maximize the probability of a bad answer instead:

$$J^{(F)}_{GAN} = -\mathbb{E}_{x \sim \mathcal{D}_T} [\log(D(F(x))] \tag{3.10}$$

The advantage of this objective function over the minimax loss formulation is that it has a high gradient even when the domain discriminator is making perfect predictions. These equations are suitable objective functions only for a general GAN, or for an asymmetric domain adaptation architecture, where the feature extractor is not shared.

However, if the feature extractor F is shared for both domains, then both the source and target feature distributions are changing as the training progresses. This is a significant difference between symmetric domain adaptation an the general GAN case. For GANs the p_{data} distribution that generated the training samples is stationary. As noted by [57], for symmetric domain adaptation the GAN objective functions might lead to oscillations, as both feature distributions are changing. Consider the case when J^F objective is satisfied, i.e. the domain classifier is consistently fooled. Then the representations have practically flipped, an this makes them separable again: the domain classifier only needs to flip the sign of its prediction for correct classification.

The **domain confusion loss** [47] is addressing the above mentioned incompatibility of the GAN loss with the symmetric domain adaptation architecture. Instead of trying to 'fool' the classifier, its aim is to 'confuse' it. This means that F is now trying to match the output distribution of the domain classifier with a uniform distribution.

$$J_{conf}^{(F)} = -\frac{1}{2}\mathbb{E}_{x\sim\mathcal{D}_S,\mathcal{D}_T}\left[\log(D(F(x))\right] - \frac{1}{2}\mathbb{E}_{x\sim\mathcal{D}_S,\mathcal{D}_T}\left[\log(1-D(F(x)))\right] \tag{3.11}$$

Besides a version of the above described different adversarial losses, the feature extractor F is also trained on the initial task with gradients backpropagated through G. This ensures that the features are still discriminative for the main learning task, while being indiscriminative for source domains.

3.4 Domain adaptation for Reinforcement Learning

This section focuses on domain adaptation in reinforcement learning settings. The increased difficulty of domain adaptation in a reinforcement learning setting comes from the interactive, sequential nature of these problems. Here not only the inputs are changing, but the underlying physics of the domains might differ as well. For example, the behaviour of a physical system in the real world and in a simulation might significantly differ. Unmodeled dynamics and other physical effects might cause different responses, especially on longer time horizons where even small changes might accumulate.

However, RL would benefit greatly from reliable domain adaptation methods. Recent successes in deep RL showed the potential of learning highly complex control policies in virtual environments. Still, given the data intensive nature of these algorithms the applicability of state-of-the-art deep RL algorithms to real world robotics problems is limited. With suitable simulators and domain adaptation techniques, RL control policies for real robots could be trained safely and efficiently in a virtual environment.

Sequential decision making problems can be described by Markov Decision Processes (MDP). A detailed introduction of MDPs is a topic for the following chapter. Here we only review how it is different from a supervised learning problem, and what implications it has from a domain adaptation point-of-view. An MDP can be described as the tuple

$\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are the state and action spaces, $\mathcal{P}$ is the state transition probability function, and $\mathcal{R}$ is the scalar reward function.

The state space $\mathcal{S}$ of the MDP might change with changing domains. If high-dimensional state spaces (such as pixels of an image) are used, even just changing input distributions might lead to transfer problems. In case only the distributions are shifting over the same state space $\mathcal{S}$ with changing domains, then the problem is similar to domain adaptation in Supervised Learning. For this situation most unsupervised domain adaptation methods described in previous sections are directly applicable to improve performance on the target domain. In a reinforcement learning setting other components of the underlying MDP might differ too when transferring the trained model to a new domain. In domain adaptation usually the task is assumed to be the same in the different domains, therefore the reward function is usually identical. Also most methods are assuming direct correspondence between RL agents, thus action spaces can also be considered alike. However, the state transition probability function and the underlying physics of the domains might significantly differ in domain adaptation.

An example where differences in the state transition probability function $\mathcal{P}$ are particularly critical is the simulation to real world transfer scenario. Even though parameters and properties of the real domain can be measured using system identification methods, some physical factors are usually not captured even in modern physics simulators, like non-rigidity, gear backlash, wear-and-tear and fluid dynamics [58]. Small errors due to under-modeling accumulate, leading to large differences on longer time-frames. Furthermore RL agents can exploit such inaccuracies if those are making the learning task easier. This may easily lead to policies that cannot perform well on the real physical system [59].

In this section we first review domain adaptation examples where only the input distributions are changing. Then we discuss some results for more complex domain adaptation problems.

3.4.1 Changing input distributions

If the underlying MDPs are similar across domains and the changes are only in the input distributions over the state space, then many of the previously described unsupervised domain adaptation methods (ie. domain adversarial training [44], Deep Adaptation Networks [53], etc.) are suitable for domain transfer without any further modification.

In this section some methods are described, which are particularly applicable for domain adaptation in robotics.

Domain randomization [58] addresses the problem of transferring learned models from a simulated, synthetic environment to the real world. In other words, its aim is to close the reality gap between the simulator and the real environment. The main concept behind domain randomization is rather straightforward: instead of trying to accurately model the real environment or using domain adaptation techniques, it simply randomizes many components and parameters of the simulation. The hypothesis is that if the variance of the simulator is high enough, then the real environment may appear just as another variation of the simulator. Therefore if the model has good performance on a highly variable simulation, then the acquired skills can be transferred without any domain adaptation.

While in principle domain randomization can be used for any component of the reality gap, recent experiments were focusing only on differences in input image distributions. Tobin et al. [58] investigated domain randomization for precise object localization, while Sadeghi and Levine [60] were using this technique to learn indoor obstacle avoidance for an MAV.

In some sense domain randomization is a way to make the model invariant for certain factors of variation, like color and textures of objects, or lightning conditions. However this can be seen as hand-engineering transferability, as all the critical factors of variation have to be explicitly randomized by the designer of the simulator. This is a serious disadvantage of this approach. Still, domain randomization can be effective in scenarios when number of factors to be randomized are limited.

Another approach enhances usual unsupervised distribution alignment with **weak pairwise constraints** [61]. Since generating many synthetic examples with large enough variation is relatively easy, for some applications it is reasonable to assume that for each unlabeled examples there exist a matching synthetic image. By finding corresponding pairs and minimizing the Euclidean distance between corresponding hidden unit activations in some layers, the network can be forced to treat images with matching labels in an identical way. Still, this method is mainly applicable for tasks where the variation of the environment is limited (stationary robotic manipulators for example), to make the assumption of the existence of corresponding images viable.

The common advantage of the above described methods or any other unsupervised domain adaptation techniques is that direct transfer of the learned policies is possible, without the need for further training on the real domain.

3.4.2 Domain adaptation in a general case and transfer learning

Next we consider domain adaptation methods, which can handle more fundamental changes between the domain MDPs. In this context *transfer learning* refers to the domain adaptation strategy, when a model first trained on a source domain, then transferred and finetuned on the target domain. In general, transfer learning also refers to training strategies when a learning agent is exposed to increasingly difficult tasks. The general hypothesis behind transfer learning is that the source domain knowledge can serve as a very good prior for further learning. This can greatly help the training, especially when rewards are so sparse in the new domain, that with random exploration the learning would probably never take off [62].

Progressive nets [62] are a general framework that enables easy reusing of previously learned network components for new tasks, by utilizing lateral connections to these previous models. Training of a progressive network starts with training a deep neural network on a task. Upon converging on the first task/domain, the parameters of the first network are frozen, and a new network is initialized. In this new network each layer is receiving inputs from the preceding layer of both the new network, and from the frozen network through lateral connections. Progressive networks have numerous advantages. Since previous network parameters are frozen, already acquired skills are not destroyed by finetuning on a new task. The new network can decide to reuse or ignore previously learned features/skills. Also there are no limitations on the new networks. The models

can be heterogeneous, and even different input modalities can be added. Finally, progressive nets are adding extra capacity to the network, in order to accommodate for the increasingly difficult of the new tasks. To conclude, progressive networks can handle a wide variety of transfer learning tasks, and in a domain adaptation settings they are capable of handling the change in the underlying physics of the domains. On the other hand with its growing network architecture it might not be the most efficient method for domain transfer. Domain transfer might render features and skills preserved in the frozen networks useless on the new domain.

Another interesting transfer learning approach, **invariant feature space** learning [63] is considering multi-agent transfer learning, but the method is also applicable for general domain adaptation. The effectiveness of this method is illustrated by transferring policies between robotic arms with different morphologies. The only assumption is that the reward functions share some structural similarity, other than that the components of the MDPs might be vastly different. The correspondence between the hidden states of the two agent is found by using proxy skills: easy tasks that both robots are capable of doing. Using these proxy skills mappings are found from the hidden states of the two agent to a common feature space. In the common feature space knowledge transfer is possible with matching optimal policy state trajectories.

3.5 Training with synthetic data

In this section we review some results on how synthetic images are used for neural network training and the effect they have on the final performance, without any explicit domain adaptation. Also we compare synthetic samples to real data in terms of learning value, ease of acquisition and ground truth precision.

For some visual problems, obtaining training sets that are sufficiently large for training is difficult and expensive. A good example is semantic segmentation, which requires pixel-level labels. This particular problem became very popular recently, largely due the reliance of autonomous driving on detailed scene understanding (which semantic segmentation can provide). Unfortunately collecting datasets for this problem is difficult: labeling a single image from the CityScapes dataset took 90 minutes on average for a human annotator [64]. Another example is optical flow estimation. Obtaining dense ground truth optical flow data is difficult, as true per-pixel correspondences are hard to determine [65].

Synthetic data generation provides a viable alternative for obtaining large datasets for such problems. However, the applicability of synthetic samples are largely dependent on the fidelity of the data generation process. Modern open-world games often feature extensive and highly realistic virtual environments [66]. One particular game, Grand Theft Auto (GTA) V was already used to generate large, synthetic semantic segmentation datasets by multiple research groups. Richter et al. introduced an efficient semi-automatic semantic segmentation tool, and investigated the effects of augmenting traditional hand-labeled datasets with synthetic data (see Fig. 3.4) [66]. They found that networks trained with synthetic images and just one third of the real CamVid dataset, outperformed a network trained entirely with real data. Furthermore Johnson-Robertson et al. demonstrated that networks trained on synthetic data only, without any real images were able to outperform networks trained entirely on a real dataset [67].

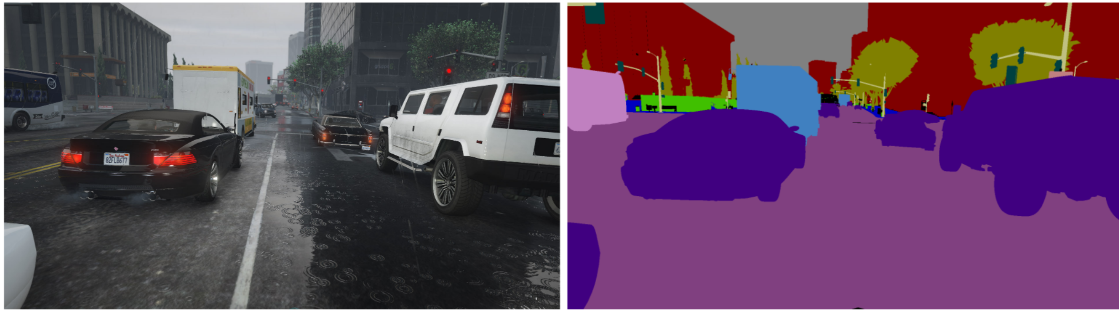

Figure 3.4: Synthetic image and corresponding semantic segmentation ground truth from GTA V [66]

It is worth noting, that the number of samples required was roughly two magnitudes higher. This suggests that the "variation and training value of a single simulation image is lower than that of a single real image" [67]. The diversity of the real environment is a clear advantage of the real images. On the other hand generating significantly more synthetic data is still easier than obtaining real annotated images, as it only requires computational resources but no human labeling effort.

In problems such as optical flow or depth prediction, providing the ground truth for real training data either requires extra sensors for data acquisition, or relies on traditional methods for calculating the ground truth. Even then measurements may contain noise, and traditional methods are imperfect and may fail in some cases (e.g. image correspondence based methods may break down in featureless regions). When this data used for training, neural networks are prone to learn the imperfections of the ground truth prediction, especially if failure in some situation is consistent. Furthermore, this ground truth is sparse in most cases. Therefore a significant advantage of synthetic training data generation is that given full access to the simulator, perfect dense ground truth can be provided for the learning system. As an example state-of-the-art learning based approaches for optical flow prediction can be mentioned, which are relying on synthetic datasets for training [68].

To conclude, adding extra synthetic images to a training set can improve performance, furthermore image generation is a viable alternative data sources in some cases. Even though their applicability largely depends on the fidelity of the virtual environment, recent result suggest that modern game graphics is already at a level where generated images are useful. The inferior training value of synthetic images can be compensated by the ease of data acquisition.

Chapter 4

Deep Reinforcement Learning

Reinforcement Learning (RL) is a family of machine learning algorithms that are particularly suitable for learning control policies. Deep Reinforcement Learning is the combination of RL approaches with Deep Neural Network function approximators. Deep RL is a very active research field nowadays, and it has already brought major breakthroughs in AI research. Examples range from mastering the game of Go [69] (which was a long standing challenge for AI), to learning a wide variety of Atari games [2], and algorithms that decreased energy consumption in Google Data Servers [70].

This chapter first introduces the general Reinforcement Learning setup, and some conventional RL algorithms that are relevant from a Deep RL point-of-view. Then the problems of using deep NN function approximators in an RL framework are discussed. Finally, some recent approaches and algorithms to deep RL are reviewed.

4.1 General Reinforcement Learning

Figure 4.1: Breakout, a popular Atari computer game (*source: McLoaf, Wikipedia*)

For many interesting problems direct supervision is not available for the learning system, simply because the correct answer or the optimal action in a particular situation is un-

known. Nevertheless, assessing the performance might still be possible, with a reward received from the environment. This information from the environment is an 'evaluative' feedback [1], which does not provides information about the alternative decisions, it only assesses the action taken by the learner. The reward is based on the (sometimes delayed) outcomes of the actions. Such feedback is typical in problems where the goal is to learn an effective control policy, or interaction with an environment. A simple illustrative example for interaction problems is Breakout (see Fig. 4.1), an Atari computer game. Without prior experiences, the best action for a given situation is unknown. However, the agent is given feedback from the environment: it is rewarded if the ball hits a brick, and punished when the ball touches the bottom of the screen. Thus the agent can learn in an iterative manner to use the paddle to hit the ball and keep it from falling, based on prior experiences and rewards. Furthermore, with enough experience effective strategies can be discovered (like eliminating the bricks on one side first and guiding the ball to go behind the remaining bricks, which is a highly effective strategy in Breakout).

A biologically inspired family of algorithms for such problems is Reinforcement Learning [71] (RL). The general RL setting (in which an agent is interacting with its environment) is illustrated in Fig. 4.2. The agent receives rewards r_t from the environment and observes its state s_t, and tries to optimize its policy to choose actions a_t in order to maximize the reward received.

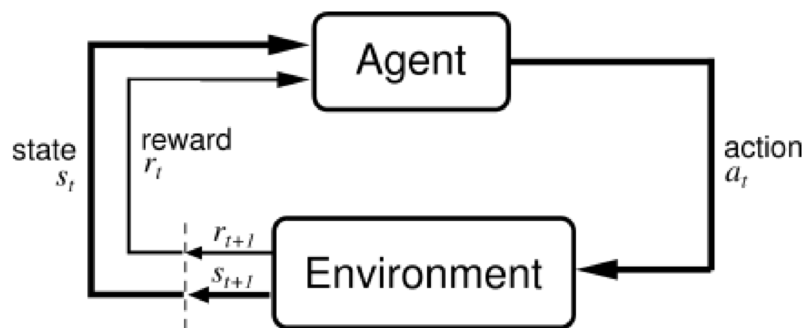

Figure 4.2: General RL setting: the agent-environment interaction in Reinforcement Learning (source: [71])

Sutton and Barto list three distinctive characteristics of RL settings, which set it apart from other learning paradigms such as supervised or unsupervised learning [71]:

1. RL is a closed loop problem. The decisions of the agent may change the state of the environment, and this can influence later inputs the agent receives.

2. No direct instruction or direct supervision is given for the learner. It only receives a scalar, evaluative reward. The best action/correct answer is unknown, furthermore no information on the appropriateness of alternatives is given [1]. Therefore the agent needs to discover the most rewarding actions by repeatedly interacting with its environment.

3. RL agents receive a sequential feedback [1]. Consequences of actions might not be immediate; the effects of actions and the resulting rewards are often delayed. This property brings about the problem of credit assignment.

The second property places RL somewhere between Supervised and Unsupervised learning on the spectrum of machine learning paradigms, based on the amount of the available feedback for the learner. However the other two properties make this setting fundamentally different.

The family of RL methods contain various algorithms. The most significant division is between *policy based algorithms*, which directly learn a policy; and *value based algorithms*, which learn an optimal value function. Another distinction can be made between on- and off-policy algorithms. *On-policy algorithms* are acting in the environment according to the policy that they are improving. In contrast, *off-policy algorithms* can learn a policy while they are acting according to a different, so-called behaviour policy in the environment. Further explanation of these RL algorithms is given in Section 4.2, but before that Markov Decision Processes are introduced in the following section.

4.1.1 Markov Decision Processes

A Markov Decision Process (MDP) is a suitable mathematical framework for modelling the stochastic sequential decision making processes, in which reinforcement learning takes place. Based on the notation in [72], an MDP can be defined with the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. The components of an MDP in a simple case with discrete, finite state and action spaces are the following:

- $\mathcal{S}$ is a finite set of states ($S \in \mathcal{S}$) of the environment, with the Markov property, which means that the actual state of the system captures all relevant information from the history. This property can be formulated as follows:

$$\mathbb{P}\left[S_{t+1}|S_t\right] = \mathbb{P}\left[S_{t+1}|S_1, \ldots, S_t\right] \tag{4.1}$$

- $\mathcal{A}$ is a finite set of actions ($a \in \mathcal{A}$), from which the agent can choose.

- $\mathcal{P}$ is the state transition probability function, giving the probability of arriving to a particular state s', after choosing action a in state s:

$$\mathcal{P}_{ss'}^{a} = \mathbb{P}\left[S_{t+1} = s'|S_t = s, A_t = a\right] \tag{4.2}$$

- $\mathcal{R}$ is the scalar reward function, it gives the expected reward received by the agent after performing action a in state s

$$\mathcal{R}_{s}^{a} = \mathbb{E}\left[R_t|S_t = s, A_t = a\right] \tag{4.3}$$

- γ is a discount factor ($\gamma \in [0, 1]$), used to differentiate between the importance of immediate and future rewards, setting the effective horizon of the agent. It is also useful to ensure the numerical stability of algorithms, preventing infinite sum of rewards. In a different interpretation, the discount factor reduces the variance of the value estimates on the cost of introducing a bias to the estimate [73].

The ultimate goal of an RL agent is to maximize the expected *return* G_t, which is the sum of rewards from a certain time-step t onwards. Future rewards are discounted, using the multiplicative discount factor γ:

$$G_t = R_t + \gamma R_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{4.4}$$

Based on the definition of return, two value functions can be defined. The *state value function* $V(s)$ is the expected return starting from state s:

$$V(s) = \mathbb{E}\left[G_t | S_t = s\right] \tag{4.5}$$

The so-called *Q-function*, or action-value function in turn is the expected return when at state s action a is taken:

$$Q(s,a) = \mathbb{E}\left[G_t | S_t = s, A_t = a\right] \tag{4.6}$$

Finally, a *policy* π, which defines what action is taken in a certain state, can be characterized as a distribution over actions given states:

$$\pi(a|s) = \mathbb{P}\left[A_t = a | S_t = s\right] \tag{4.7}$$

The expected return following a policy π from a given state or from a given state-action pair is given by $V^{\pi}(s)$ and $Q^{\pi}(s,a)$, respectively.

4.2 Value and policy based RL algorithms

Two distinct approaches to RL are policy and value based methods. The first is trying to learn an optimal behaviour by directly optimizing its policy. Value based methods, on the other hand are solving the dual problem instead, which the finding of an optimal value or action-value function $Q^*(a,s)$. The optimal value function in every state determines what the optimal action is. Therefore if $Q^*(a,s)$ is found, then the optimal policy is given by acting greedily according to it (always choosing the action that leads to the highest expected return).

In practice the effectiveness of these algorithms are largely affected by the nature of the problem at hand. For some problems, parametrizing and learning an optimal policy directly is easier; while for other problems the value function might be easier to represent [71]. Also, value-based methods might experience problems when the states are aliased and the Markov property does not hold [74].

This section first introduces the Bellman equation and Value based methods, then policy gradient methods are addressed.

4.2.1 Value based methods

An important decomposition of the value functions is the *Bellman equation*, which provides an intuitive way for learning the value and Q functions. The Bellman equation is based on the fact that the sum of the return can be decomposed into an immediate reward, plus the discounted return from the next step:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k} = R_t + \sum_{k=1}^{\infty} \gamma^k R_{t+k} = R_t + \gamma G_{t+1} \tag{4.8}$$

Similarly, the value and action-value functions can be decomposed as well: to an immediate reward, plus the value of the next state. In the following the equations are given for the action-value function, but the formalization is similar for the state value function as well.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right] \tag{4.9}$$

This equation allows for an intuitive iterative update for learning value functions, the *Value Iteration* algorithm, which (in a tabular setting) is proven to converge to a global optimum [75]. Given an approximation of the action-value function Q_i, once a new reward is observed, a slightly better approximation can be given.

$$y^Q = R_{t+1} + \gamma \max_{a'} Q_i(s', a') \tag{4.10}$$

This new approximation can be set as a target for updating the Q function, yielding the update rule of *Q-learning*.

$$Q_{i+1}(s,a) = Q_i(s,a) + \alpha \underbrace{\left[\overbrace{R_{t+1} + \gamma \max_{a'} Q_i(s', a')}^{y} - Q_i(s,a)\right]}_{TD\ error} \tag{4.11}$$

α in the equation above is the learning rate. The term in the square brackets (the difference between the previous approximation of the Q value and the approximation that encorporates the observed immediate reward) is called the *temporal difference* (TD) error.

SARSA, an on-policy temporal difference method is not using the max operator for selecting the action, but evaluates the Q value following the actual policy [76]. The name refers to the observations that are required for calculating updates: starting state, action, observed reward (s, a, r); and also the new state s' and the action a', taken in that state.

$$y^{SARSA} = R_t + \gamma Q_i^{\pi}(s', a') \tag{4.12}$$

This method provides more accurate value predictions during the exploratory exploratory period of the training, while it converges to the standard Q-learning updates as exploration ceases.

4.2.2 Policy Gradient Reinforcement Learning Methods

Policy Gradient methods solve RL problems by directly improving the policy, calculating the gradient of an objective function with respect to the policy parameters. Compared to value-based algorithms, these methods enjoy better theoretical convergence properties, and are able to learn stochastic policies as well (which is the only way to act optimally in some tasks, think of the game 'Rock-Paper-Scissors' as an example) [72]. Another important advantage of policy based methods is that they handle continuous action spaces better than value based methods. Value based action selection includes a max operation on the possible actions. This calculation might get intractable in case of many actions, or on continuous action spaces.

Even thought policies can be learned using gradient-free optimization methods (such as Evolutionary Algorithms), those methods are less suitable for learning policies with a high number of parameters. This makes them impractical for training policies represented by deep neural networks, therefore the focus of this sub-section is on gradient-based methods.

Numerical policy gradient calculation

A conceptually simple method of calculating the policy gradient is simply calculating the gradient numerically with finite differences, by perturbing every parameter of the policy. This method is applicable for a wide variety of policies (even to non-differentiable functions, or to arbitrary policy representations). However, separate evaluations are required for every parameter of the policy. As usually there are costs associated with evaluations in the environment, this method is impractical for learning models with a high number of parameters, such as deep neural networks.

Analytical policy gradient calculation

An alternative is calculating the policy gradient analytically. The *policy gradient theorem* gives the gradient of the objective function $J(\theta)$ with respect to the policy parameters [77]. Here, the policy is represented with a parametric probability distribution $\pi_\theta(a|s) = \mathbb{P}[a|s;\theta]$, which select actions stochastically based on both the state and the parameter vector θ [78]. The gradient is calculated as the expected gradient of the log of the policy, multiplied by the action-value of the action selected by the policy:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \ln \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right] \tag{4.13}$$

In the above equation, $\log \pi_\theta(s,a)$ is the score function of the policy. Its gradient is showing the direction of parameter change that can increase the log-likelihood of choosing a certain action. Intuitively by multiplying this term with the Q value, the calculated policy gradient will be bigger in the direction of actions with higher Q values. Therefore this update makes choosing more rewarding actions more probable.

However, the availability of the Q values is not trivial. The REINFORCE algorithm is simply using unbiased Monte-Carlo estimates of Q, essentially by evaluating a full episode and summing the experienced rewards [79]. This gives a high variance estimate, while

also being rather sample-inefficient. Actor-Critic algorithms [71] lower the variance and increase sample-efficiency by employing two different function approximators. The actor is parametrizing the policy, while the critic is parametrizing the Q function. The critic can be trained with temporal difference methods, just like algorithms in the previous section. The Q estimate of the critic is then used in Eq. 4.13 to scale the gradient updates of the actor.

A common way to further reduce variance of the policy gradient updates is by subtracting a state-dependent baseline value from the Q function [71]. The most appropriate choice of this baseline is the value function $V(s)$. As discussed earlier, $V(s)$ is the expected reward from a particular state, while $Q(s,a)$ is the expected future reward from a state choosing a particular action. By subtracting $V(s)$ from $Q(s,a)$ we get a measure of how action a compares to other possible actions at that state. The resulting function $A(s,a)$ is called the *Advantage Function*. Intuitively, by using A we are trying to increase the probability of actions that are better than the action chosen by the actual policy, and decrease those that are worse [73]. One option for calculating A is separately learning both Q and V. However, it can be shown that an unbiased estimate of A can be given by the TD error, which can be used directly to learn A [72].

Deterministic Policy Gradient

Stochastic policies were long favoured, because of their smoothness and differentiability, which made gradient calculation with the policy gradient theorem straightforward. At the same time gradient for deterministic policies were considered non-existent, or computable only using a model of the environment [78]. However, Silver et al. showed that gradient calculation of a deterministic policy ($a = \mu_\theta(s)$) is indeed possible [78]. With a deterministic policy, the policy gradient can be decomposed into two terms using the chain rule. The first term is the gradient of the critic with respect to the action, and the second part is the gradient of the actor, with respect to its parameters. This gives the Deterministic Policy Gradient (DPG) update rule:

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}\left[\nabla_\theta Q^\mu(s,a)\right] = \mathbb{E}\left[\nabla_\theta \mu_\theta \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)}\right] \tag{4.14}$$

A disadvantage of a deterministic policy is that it does not provide sufficient exploration of the state-space in itself. Therefore it is required to learn off-policy, using a stochastic behaviour policy to collect the experiences that ensures sufficient exploration.

4.3 Deep Reinforcement Learning

Reinforcement Learning methods traditionally relied on tabular representations of the value functions or policies. However scaling up tabular approaches is difficult, as these representations are susceptible for the curse of dimensionality. Also, every state in this look-up table has to be visited by the agent, since no generalization is made between neighbouring states. A way to overcome these difficulties is to use function approximation. Function approximation not only provides an efficient way to keep the information otherwise stored in large look-up tables; it also allows for generalization to neighbouring

states. Under function approximation, the value functions are parametrized with a set of weights θ:

$$\hat{Q}(s, a; \theta) \approx Q(s, a) \tag{4.15}$$

Linear function approximators were proven to be useful in some cases, either working directly on the inputs, or on some features of the inputs. However in general, linear functions of the inputs are less expressive than non-linear models. Even though this can be compensated using a linear model on top of non-linear features of the raw observations, finding and hardcoding these features still required expertise and significant knowledge of the domain. On the other hand, for a long time RL algorithms were considered unstable when used with non-linear function approximation (see the next section).

As described in the previous chapters, deep neural networks are exceptionally good function approximators, and recent advances in deep learning made it possible to train these networks efficiently. Deep reinforcement learning is utilizing deep neural networks as function approximators; either for parametrizing the policy, or in case of value based methods, the state value or Q functions. This holds the key to enable learning directly from high dimensional sensory input, such as images.

4.3.1 Difficulties of using Deep Neural Networks in RL

Some properties of RL prevented successful training of neural networks, and more generally the use of non-linear function approximators in an RL framework:

- Given the global function approximator nature of neural networks (as opposed to local function approximators, such as Radial Basis Function Networks), a weight change that can improve the network's performance for a particular state-action pair, might deteriorate the performance in other regions of the state-action space, possibly destroying knowledge and skills learned already [80]. This makes Deep RL training more difficult than traditional RL algorithms.

- Usually general supervised and unsupervised deep learning algorithms need 'identical and independently distributed' training data for learning. This criteria is not met for experiences an agent receives during RL training. Usually subsequent states are similar: there are both temporal and spatial correlations between states. This can destabilize the training, or even make it diverge [2].

- TD methods face another destabilization factor: the function to be approximated is also used to set the targets for its own updates (see Eq. 4.10), i.e. the target is also dependent on the network weights. This is a major difference compared to supervised deep learning, where targets are fixed before the training [2].

- Furthermore for value based methods, even small changes in the Q function can lead to significant changes in the policy, and in the resulting data distribution [2]. This property also sets Deep RL apart from other deep learning problems, where the training data distribution is independent from the predictions of the learner.

4.3.2 Deep RL Algorithms

In the following we introduce some recent deep RL algorithms and see how they address the above mentioned challenges.

Neural Fitted Q-iteration

The *Neural Fitted Q-Iteration* (NFQ) algorithm addressed the first two problems based on the principle of storing and reusing experiences [80], with a common RL technique called *Experience Replay.* The main idea is minimizing the negative influence of a new update by offering previous knowledge explicitly. Therefore the gradients for an update are calculated based on *all* the previous experiences, retrieved from an experience replay memory. These batch updates can mitigate the negative global effects of the updates, and also solve the problem of the correlated observations, associated with online updates.

The loss function of the network at iteration i is the squared TD error. The expectation is calculated on the whole replay memory D:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim D}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i) - Q(s,a;\theta_i)\right)^2\right] \tag{4.16}$$

The shortcoming of NFQ is that the batch updates it uses have per-iteration computational costs, proportional to the size of the replay memory. This makes it inefficient for training large networks.

Deep-Q Networks

Deep Q-Networks (DQN) address the batch update problem of NFQ by using SGD, with mini-batches of experiences sampled from the replay memory [2]. The experience memory is a 'First-In-First-Out' memory typically storing the last $10^5 - 10^6$ experiences. Just like SGD made training deep neural networks more feasible in general, in this particular case SGD also manages to keep the benefits of the full-batch updates of NFQ while preventing the linear increase of computational costs with the increased number of stored experiences.

Another key contribution of DQN is the use of a 'target network', which provides fixed targets for the Q-learning updates. This helps with the problem of the correlated targets of the network updates (i.e. the network is used to set its own targets for updates, see Section 4.3.1). The target network is a copy of the Q-network, which is updated periodically, but kept fixed within updates. By fixing the targets the optimization process becomes more similar to supervised learning.

These modifications in practice are usually sufficient for training deep-RL agents, even from high dimensional image-stream inputs. However, it is worth noting that RL training under non-linear function approximation is still not guaranteed to converge, unlike the tabular methods for which theoretical guarantees exist. In practice, DQN demonstrated learning a wide variety of Atari games with fixed hyperparameters, reaching human performance on some games [2].

The DQN loss function is calculated on a mini-batch of experiences sampled from the experience memory D, using the target network weights θ^- as follows:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i) \right)^2 \right] \tag{4.17}$$

To calculate updates for the network, the squared error loss has to be differentiated with respect to the network parameters θ_i.

$$\nabla L_i(\theta_i) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \tag{4.18}$$

Note that with fixed target network weights, the targets are not dependent on the actual network weights, which turns the training to a well-defined optimization problem [2].

In Atari games, which are often used as benchmarks for Deep RL algorithms, not all states of the environment is observable in single images (e.g. velocity of obstacles, or flickering objects). The original DQN architecture made these game observable by stacking four frames together [2]. An alternative architecture, Deep Recurrent Q-Networks (DRQN) is solving partial observability by using a recurrent LSTM layer as the first post-convolutional fully-connected layer with [81]. The recurrent layer allows integration of information over multiple timesteps, thus making the games observable.

Deep Deterministic Policy Gradient

The ideas that made DQN training stable (experience replay + target network) were used in the Deep Deterministic Policy Gradient (DDPG) algorithm, to train Deep RL policies with continuous action spaces effectively [82]. In this approach both the actor and the critic are deep neural networks, and target networks were utilized for both. Since DDPG (and the original DPG) is an off-policy method, experience replay can be used to provide uncorrelated mini-batches of samples for the network updates. Capabilities of the algorithm were demonstrated on simulated robotic grasping tasks amongst others, and the algorithm was able to learn effective policies using image inputs [82].

Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) algorithm [83] is a policy gradient method, in which the size of the gradient step is carefully calculated. As noted by the authors, step sizes are of crucial importance for the success of training RL agents. In a supervised setting it is less of a problem: if a too large step is taken, subsequent gradient steps can make corrections and guide the optimization back. In contrast, changing the policy of an RL agent influences the visited state distribution, and subsequent experiences collected under the deteriorated policy might not be able able to correct the erroneous step. That is why it is difficult in Deep RL to obtain stable and steady improvements. Therefore the TRPO algorithm constrains the step size based on the KL divergence of the old and new policies (measured on the policy parameters), thus providing updates that result in monotonic improvements of the policy.

Asynchronous Deep RL

All the previously introduced methods used experience replay (either replaying the whole memory or a mini-batch of experiences) to get uncorrelated samples. An alternative source of uncorrelated updates can be the experience collected from multiple agents online; this is the basic idea behind *Asynchronous Methods* for Deep RL [84]. In this case multiple agents are acting simultaneously, each in its own instance of the environment. They are experiencing a variety of different states simultaneously, likely exploring different parts of the state-space. The agents are providing these uncorrelated experiences, and the gradient updates calculated on these, to a shared neural network. Asynchronous methods can train deep neural networks with a wide range of existing RL algorithms. Previous methods relied on experience replay for uncorrelating the samples; this made using on-policy methods problematic, since experiences stored in the memory were sampled under previous policies. By sidestepping the need for experience replay, asynchronous methods allow training with on-policy algorithms as well. Also the multi-agent setting allows for larger flexibility (e.g. different agents may utilize different exploration strategies as an example).

Mnih et al. experimented with a number of different RL algorithms in the asynchronous Deep RL setting, namely the off-policy 1-step and n-step Q-learning, and the off-policy Sarsa and actor-critic methods [84]. Their most successful algorithm was the *Asynchronous Advantage Actor-Critic* (A3C) algorithm, which is a regular actor-critic algorithm, trained asynchronously. It maintains an actor and a critic network, and uses a a baseline to reduce the variance of the updates. Compared to DQN, A3C was able to surpass the previous state-of-the-art in half the training time [84].

Chapter 5

Auxiliary Training

As we discussed in 2, one reason for the effectiveness of Deep Learning models lies in the fact that they are trained in an end-to-end fashion: these models are able to learn feature extractors on different hierarchical levels, starting from raw inputs. This way the network training is aiming to optimize the hidden representation for the given task at each layers. However, learning all the parameters for each layer makes the training of these models a rather data-intensive process, so the performance of Deep Learning models in general relies heavily on the amount of available training data.

This problem is maybe even more pronounced in a Deep Reinforcement Learning setting. Here, the only training signal is a scalar reward, which is sometimes even sparse and delayed. Consider a DQN agent learning an Atari game from visual inputs: the agent literally needs to 'learn to see' (i.e. optimize parameters for a convolutional encoder network) using only the scalar reward signal from the environment. This is an important cause behind the high sample complexity and excessive training time of RL agents.

Auxiliary training is addressing the above mentioned problems by augmenting the learning with auxiliary tasks. This method incorporates additional supervised, unsupervised or RL losses in the training process, and also allows for leveraging multimodal sensory inputs [85] [86]. Besides drastically reducing training times, auxiliary training has also demonstrated finding superior final policies.

This chapter first covers some relevant concepts in Section 5.1 from the supervised deep learning literature. Then some recent research are reviewed, where auxiliary training was used to aid RL training (e.g. the NAV agent in Section 5.2.2 which uses supervised auxiliary tasks, or the UNREAL agent in Section 5.2.3 that makes use of unsupervised training). Finally in Section 5.3 we investigate different possible auxiliary tasks, for which successful deep learning has been demonstrated and which are relevant for learning obstacle avoidance and navigation capabilities for MAVs.

5.1 The auxiliary training in supervised training

In this section the focus is on how supervised training methods were demonstrated to benefit either from extra supervised or unsupervised training.

5.1.1 Multi-Task Learning

The idea of introducing new tasks to improve the training process of a neural network dates back to the early '90s. Suddarth and Kergosien referred to these additional tasks as 'hints' [87]. A hint can generally help the training when the capacity is sufficient to learn both functions simultaneously, and both the hint and the original function share some underlying rule. The authors demonstrated how a hint can both reduce training time and eliminate pathological local minima in the example of learning XOR function with an OR function hint. Using hints was compared to chemical catalysis in a sense that both can deform the potentials temporarily, in order to arrive to a better minimum with lower potential. Also, using hints is an effective way to provide prior knowledge about the task or the domain for the learning system.

Caruna extended hints and introduced the term of *Multi-Task Learning* (MTL) for supervised settings [88]. MTL is improving training by making use of the domain information contained in the training signals of other related tasks. The information is shared between tasks by using a shared representation. The underlying prior belief in MTL is that "among the factors that explain variations observed in the data associated with different tasks, some are shared across more tasks" [8]. If this prior is correct, the networks can benefit from the increased amount of training data and extra supervision, which leads to increased performance on the individual tasks and better generalization.

5.1.2 Aiding supervised learning with unsupervised learning

Before the success of AlexNet [3] on the ILSVRC comptetition in 2012, the common approach for training deep neural networks was combining supervised learning on the main classification task with unsupervised learning from the input images. Unsupervised learning was used for pre-training the networks, to help learning useful representations in the deeper layers. AlexNet demonstrated that large capacity networks with sufficient training data can be trained by using supervised learning exclusively, without any pre-training. These results showed that pre-training is not essential for training deep neural architectures. Alleviating the requirement of compatibility with unsupervised pre-training allowed for a various novel architectural choices [89]. This focused further research more towards novel network architectures in recent years, such as GoogLeNet [38], VGGnet [37] and more recently ResNet [4].

However the idea of using unsupervised learning to improve supervised learning (even in novel architectures) came up again recently. An approach for this is augmenting the supervised neural network with "decoding pathways for reconstruction" [89]. A decoding pathway in practice is an auto-encoder [8]. The general architecture of an auto-encoder (see Fig. 5.1) consists of an encoder and a decoder sub-network. The encoder network transforms the input to a hidden code, and the decoder is trying to reconstruct the

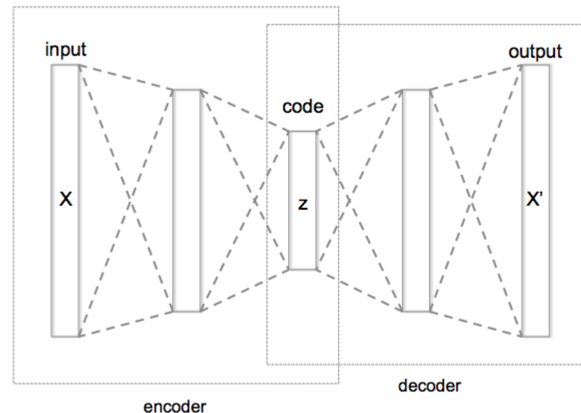

Figure 5.1: Schematic picture of an autoencoder architecture (source: Wikipedia, Chervinskii - CC BY-SA 4.0)

input from this hidden code. Usually the decoder architecture is just a mirrored copy of the encoder (convolutional layers are paired with deconvolution for example). In an auto-encoder the hidden code has low capacity, or some other form of restriction is present. This forces the auto-encoder to learn efficient representations and to prevent trivial copying of the input.

In the context of augmenting a supervised network, the auto-encoder is attached to an intermediate representation of the original supervised architecture, and trained to reconstruct the inputs based on the hidden activations. One possible architecture for this is depicted in Fig. 5.2. Here a deep neural network is augmented with an auto-encoder. The first three convolutional layers are shared: at one hand they are part of the original supervised network, on the other hand they form the encoder layers of the auto-encoder. The hidden representation after the third convolution is connected to the decoder layers of the auto-encoder, which is trained to reconstruct the original image using repeated deconvolution operations. These layers are trained with reconstruction loss(es) in an unsupervised manner. The reconstruction loss can be calculated between the original and reconstructed image (Reconstruction Loss 1), and optionally also between hidden representations (Reconstruction Loss 2 and 3). The fully-connected upper layers are trained with some form of a supervised loss (classification loss in this example).

The shared part of the network is trained with backpropagated errors from both pathways. This way these layers can benefit from both the supervised and the unsupervised training. In practice, the auto-encoder training is trying to maximize the information contained in an intermediate representation of the original supervised network (which is the bottleneck for the autoencoder). This optimized representation can help the network learning on the main supervised task as well. Zhang et al. reported noticeable improvement on the classification test performance of a VGGNet, trained on ImageNet dataset by augmenting the supervised training process with unsupervised objectives using auto-encoders [89].

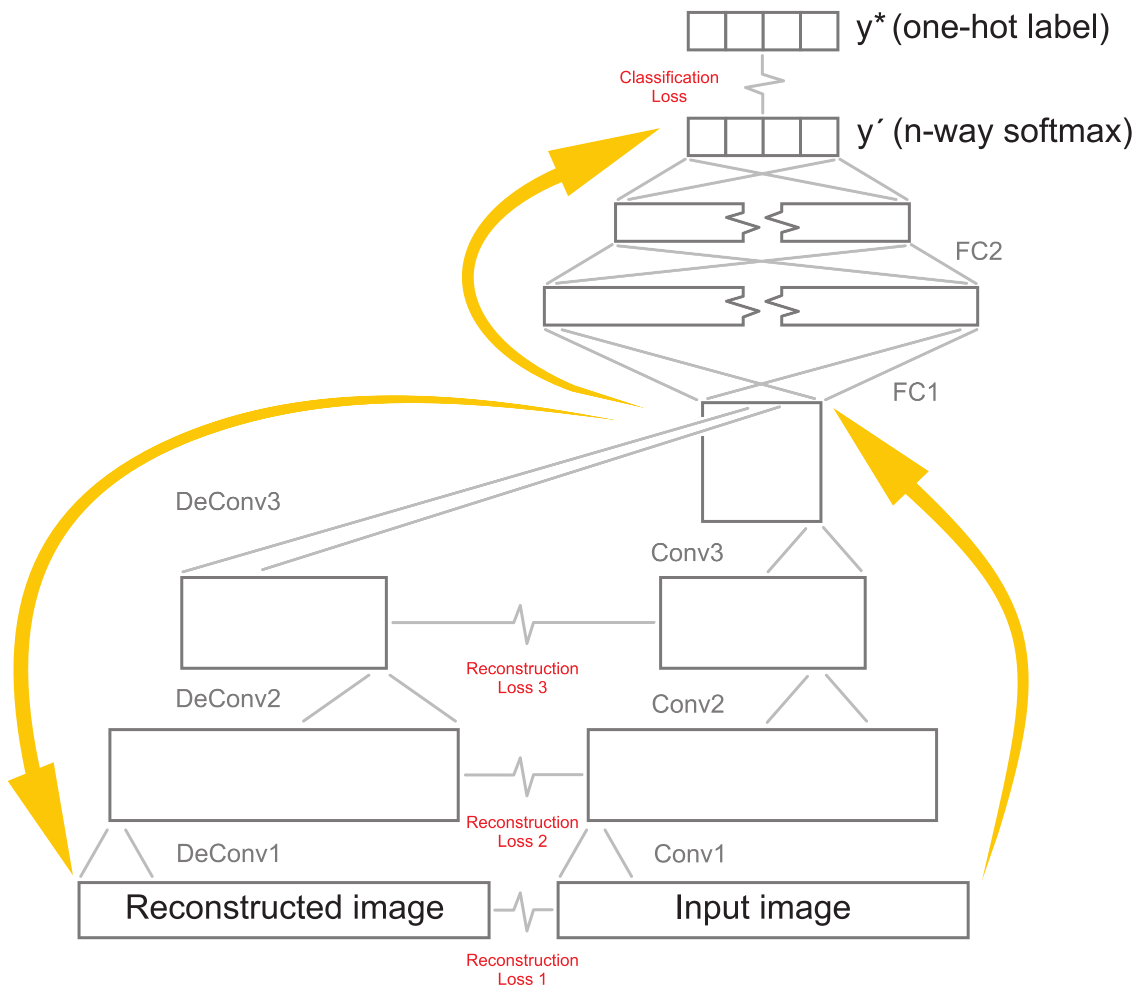

Figure 5.2: A deep neural network augmented with an autoencoder. Yellow arrows indicate the general direction of forward propagation.

5.2 Auxiliary training for RL

Auxiliary training in Reinforcement Learning is rather similar to the above described concepts of additional training in a supervised setting. Here again an intermediate-level representation and inputs are shared between different learning tasks, while some task specific layers remain independent. The main goal is to improve the training progress and the final performance of an RL agent. There are no restrictions on the nature of the auxiliary tasks: they can be supervised, unsupervised and RL tasks as well. The main RL training can benefit from the shared representation, which is trained lot faster and may learn more useful features using all training signals simultaneously. A general network architecture for auxiliary training is illustrated in Fig. 5.3.

A common, but sometimes overlooked example of shared representation in RL is the usual practical implementation of actor-critic architectures with function approximation. While parameters of the actor and the critic are normally shown being independent for

generality, in practice usually some of the parameters are shared. As an example, in an A3C agent all non-output layers are shared [84]. On top of the shared convolutional feature extractor there is a linear output for the value function, and a softmax output for the policy (for discrete action spaces).

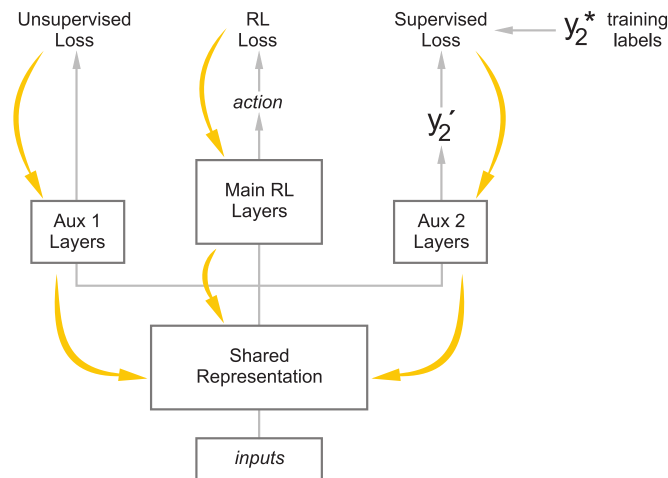

Figure 5.3: Architecture with shared and task specific sub-networks. Black arrows indicate the flow of the information during forward propagation, yellow arrows during backpropagation.

5.2.1 RL agent for FPS playing

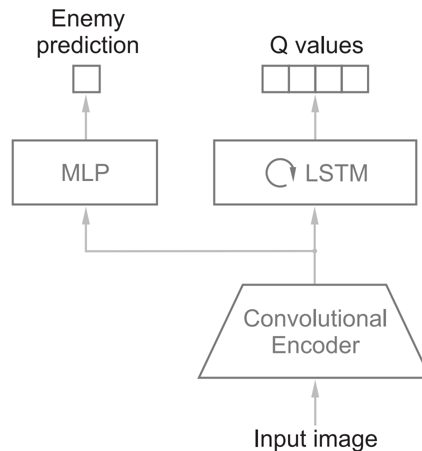

Figure 5.4: Network architecture of the augmented DRQN agent used in [90]

An early attempt to use auxiliary training in RL was made in the context of learning to play in a death-match scenario in an FPS game environment [90]. In this work a Deep Recurrent Q-Network (DRQN) agent [81] is trained on the main task, with the auxiliary task being enemy detection on the input frames. Fig. 5.4 illustrates the network architecture. The convolutional feature extractor of the DRQN agent is shared between

the two tasks, and the detector has a separate fully connected hidden layer. Compared to the DRQN baseline, the agent is able to learn more effective policies in less time.

The authors also compared the performance to another architecture, which featured a completely separated detector network for enemy prediction. In this setting, the output of this separate network is used as an additional input for the DRQN network. If the sole use of the auxiliary training were to provide the agent with information whether an enemy is present, then the performance would not be significantly affected. However, the performance of this architecture dropped down roughly to the level of the base DRQN agent without any enemy information. This experiment showed that learning a shared representation has additional benefits, compared to simply providing information about the presence of enemies [90]. This result suggests that auxiliary training can be used to provide extra domain information and supervision for the agent, focusing the learning efforts on the important aspects of the learning task.

5.2.2 NAV agent

Mirowski et al. investigated the problem of learning navigation abilities in 3D maze environments using RL [85]. Learning this problem from visual inputs is highly challenging: rewards are often distributed sparsely in the environment, and effective navigation in unknown environments requires exploration and the use of a rapid, one-shot memory. For dealing with these complexities the authors introduced the NAV agent.

Network Architecture

The NAV agent (see Fig. 5.5) is an extension of the A3C LSTM agent [84] . Here, on top of a convolutional encoder two LSTM layers are stacked, from which a linear output gives Value function prediction, and an n-way softmax layer outputs the action distribution of the policy. On top of the input image, three extra inputs are used: the most recent reward r_{t-1}, the agent-relative velocity vector $\mathbf{v}_t$ and the last selected action a_{t-1}. With the first LSTM layer being fed with the observed reward and encoded image the authors hypothesize this layer might be able to make associations between rewards and visual inputs and provide this as a context for the second LSTM layer. That layer also receives the encoded image, as well as the other two extra observations in order to build an information rich representation for the value and policy predictions. In training time, this architecture is extended with additional sub-networks, in order to accommodate for the auxiliary training signals.

Auxiliary tasks

Two auxiliary tasks were proposed for the NAV agent: depth prediction and loop closure prediction.

Learning to predict depth from image inputs might help the agent, to learn about the underlying 3D structure of the environment. This knowledge can improve obstacle avoidance and short term trajectory planning capabilities [85]. Ground truth depth images are provided by the simulator. The authors experimented with different formulations of

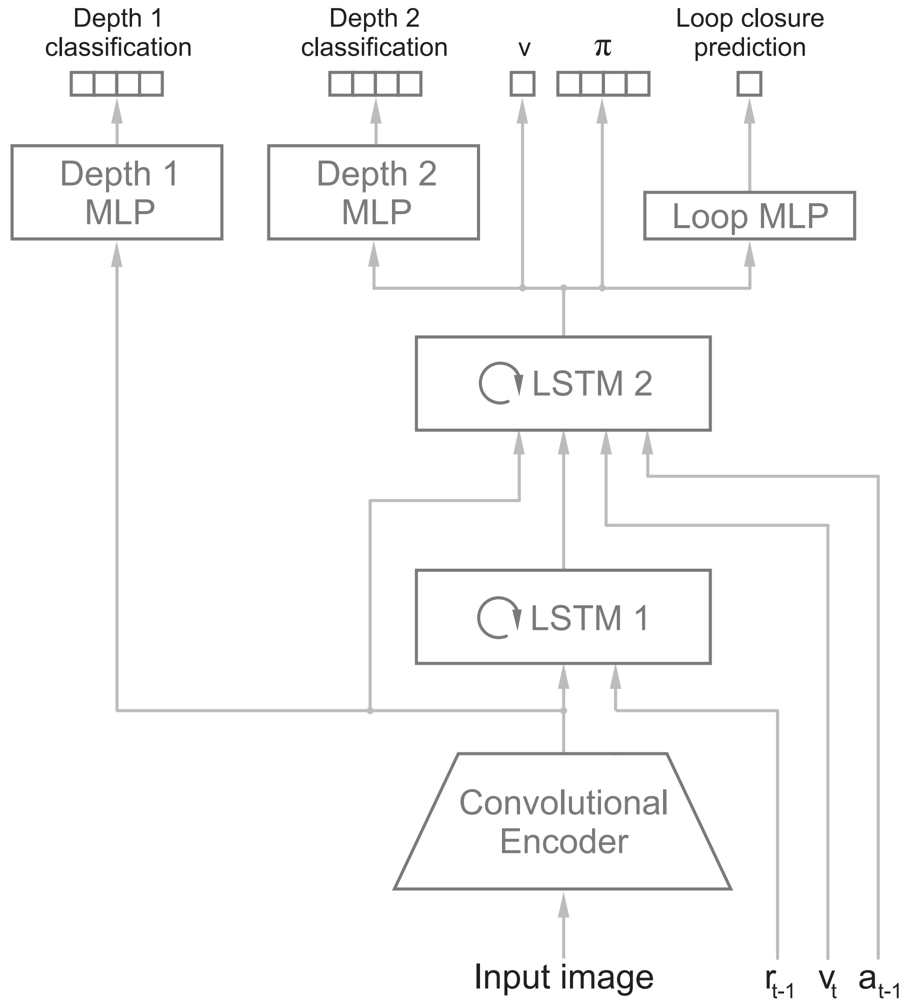

Figure 5.5: Architecture of the NAV agent

the depth prediction task (casting it either as a regression or as a classification problem), and investigated two different architectures for integration (predicting depth after the convolutional layers or from the second LSTM layer output). In order to keep the computational costs of this auxiliary task minimal, the predicted depth image has low resolution (4x16 px), and covers only the most relevant central section of the input image.

Loop closure prediction is trying to encourage the agent to integrate its displacements, thus keeping track of its position. This capability is useful both for efficient exploration and spatial reasoning [85]. Loop closure prediction is a binary classification task (the actual position has either been visited previously, or not), with the ground truth labels labels available from the simulator.

Experimental results

The main findings from the different auxiliary loss combinations were the following:

- Agents with auxiliary depth training outperformed A3C agents, even when A3C agents were given perfect depth information. Therefore in this context depth information is more useful for auxiliary training than as an extra input. This result shows the benefits of the better representation built using auxiliary training over extra information (in line with the results from [90] in Section 5.2.1).
- Auxiliary depth classification gave better results than depth regression, even though a regression loss contains more precise information. A reason for this might be that for the main task less precision is sufficient. However, another important consideration is that auxiliary tasks have to converge significantly faster than the main RL task, to allow the main task to benefit from the representations built by the auxiliary task. Therefore it is beneficial to keep the auxiliary tasks relatively simple.
- Depth2 auxiliary training (prediction from the second LSTM layer) performs better than Depth1. The reason for this can be that Depth2 provides auxiliary training for the LSTM layers too, while Depth1 influences the convolutional layers only.

5.2.3 UNREAL agent

The previously introduced auxiliary training tasks relied on extra supervision from the environment, and also the the auxiliary losses were task-specific. A recent work by Jaderberg et al. focuses on more general auxiliary tasks, which are applicable for a wide variety of RL problems and do not rely on extra supervision from the environment [86]. They refer to these as *unsupervised auxiliary tasks*, and introduce the UNsupervised REinfiorcement and Auxiliary Learning (UNREAL) agent.

Unsupervised learning is traditionally used for learning useful visual representations, typically by reconstructing the input image (e.g. with an auto-encoder), or by predicting the next frame. However in this work the authors focused on learning objectives that are trying to 'predict and control features of the sensorimotor stream' instead, as these objectives might lead to more useful representations on the main task [86]. Two different types of unsupervised auxiliary rewards were proposed: auxiliary control tasks and auxiliary reward tasks.

Auxiliary control tasks

Auxiliary control tasks make use of different pseudo-rewards, by learning optimal policies and value functions to maximize these pseudo-rewards.

The authors introduced *pixel control* as an auxiliary control task. In pixel control, the agent is rewarded for changing the pixel intensities in a region of the input image. This reward is available for the agent without any extra information from the environment. This control task encourages the agent to learn how it can influence the perceptual stream it receives, which can help the agent learning how to control its environment. As an example example, consider a situated agent in an 3D environment with egocentric visual inputs: if the pixel intensity difference is high between the controlled part of the input and the segment right to it, then the agent can maximize intensity change by turning

right. In general, knowing how to control the environment can be a useful capability for a wide variety of tasks. Therefore this auxiliary task is generally applicable for RL agents using visual inputs.

The authors also proposed *feature control* as an auxiliary control task, where the pseudo-rewards are the hidden activations of a higher layer. This auxiliary task is justified based on the observation that the value and policy networks can learn to extract task-relevant high level features. Therefore learning 'how to get more' of a certain feature might be a suitable auxiliary task. However, the effects of this auxiliary task were not investigated in [86].

An important property of off-policy auxiliary training tasks, that for off-policy learning it is sufficient to simply collect experiences using the policy of the main RL task. Off-policy learning enables learning optimal policies for auxiliary control tasks without ever using those policies for acting in the environment.

Auxiliary reward tasks

A key for learning a successful RL policy is that the agent needs to develop features, which can recognize rewarding states. However in case of sparsely distributed rewards it takes a long time to observe enough rewarding experiences for learning these features. Therefore the aim of auxiliary reward tasks it to focus the observations of the agent on these sparse extrinsic rewards.

The well-known experience replay technique (which was used in DQN [2] to stabilize training and to increase data efficiency) can be used in this context as an auxiliary reward task: *value function replay* can provide further off-policy training for the critic of the on-policy A3C agent.

Another auxiliary reward task is *reward prediction*: predicting the reward in the next time-step given some prior inputs. Reward prediction is practically a simpler version of value prediction from a temporal aspect. In the future direction it only considers the immediate reward in the next time step, not the sum of future rewards as value functions do. In the past direction only the three most recent predecessor states are considered rather than the complete history. The learning task is formulated as classification, as only the sign of the next reward is predicted (positive, negative or zero), which makes this training task even simpler.

Since reward prediction is only used for helping representation learning, it is not important to keep the data distribution unbiased for this auxiliary learning task. This way, rewarding experiences can be replayed more often, and this results in significantly faster representation learning. In contrast, value predictions are used as a baseline for the training of the policy network, therefore biasing the value estimate is not acceptable.

Neither of the above mentioned task require any extra information for training, which makes these techniques universally applicable for RL training.

Network architecture

The network architecture of the UNREAL agent is visualized in Fig. 5.6. This architecture is based on an A3C LSTM agent, which contains a convolutional encoder, an LSTM layer,

a linear output for the value prediction and n-way softmax output layer for the policy. Since value function replay does not require any architectural changes, it is not depicted on the figure.

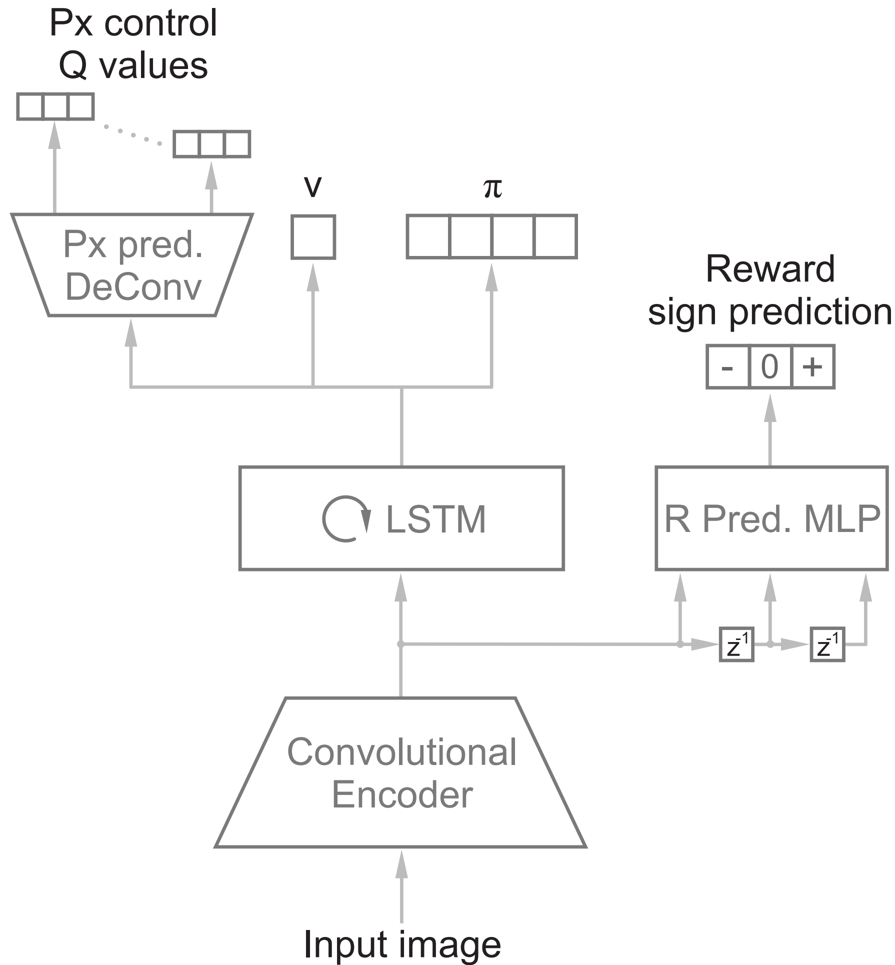

Figure 5.6: Network architecture of the UNREAL agent

Reward prediction has three soft-max outputs for predicting the sign of the next reward. Instead of doing reward prediction from LSTM activations, it is calculated based on the outputs of the convolutional encoder for the preceding 3 time steps. These observations are concatenated with a simpler feedforward network. This way the prediction is only based on the three previous step, not on a longer history captured in the LSTM.

Q values of the pixel control auxiliary task are calculated separately for a $n \times n$ non-overlapping grid placed over the input image. The spatial nature of this control task is exploited by using deconvolutional layers. The deconvolutional architecture allows learning separate Q functions jointly: the weights are shared between Q functions in different positions.

5.3 Auxiliary training tasks

In this section a number of different tasks are reviewed, which might be used as auxiliary tasks for training an RL agent for obstacle avoidance an navigation in a cluttered environment. The main focus is on tasks, where successful NN training was demonstrated, and the information learned is of interest from an obstacle avoidance/navigation point of view.

5.3.1 Depth prediction

Depth prediction is a key component to scene understanding, as it concerns directly the 3D geometry of a scene. However, predicting depth from a single monocular camera image is a challenging problem. One major difficulty is that "the global scale is a fundamental ambiguity in depth prediction" [91]. Even with all inter-image spatial relations correctly predicted, the global scale (or a corresponding mean depth value) is still difficult to obtain, and may lead to erroneous results. However, humans are capable of estimating depth based on monocular cues such as perspective, occlusion, and scaling relative to the known size of a familiar object [92]. This section first gives a brief overview of some current deep learning approaches for single image depth prediction, then discusses how it can be used as an auxiliary training for Deep RL.

Deep learning for depth prediction

For effective single camera depth prediction, multiple local depth cues (such as object sizes, perspective, atmospheric effects) have to be considered and related on a global image level. Eigen et al. addressed this problem with the 'Multi-Scale deep network', which uses both a global 'coarse-scale' network to integrate global understanding of the full screen, and a local 'fine-scale' network, which can make local refinements on the coarse depth map [91]. This network learns direct depth regression using ground truth depth images, and produces dense depth maps. As it was mentioned above, scale ambiguity may lead to large errors using conventional regression error metrics, which makes it difficult to learn the relative positions of objects. The authors therefore proposed a 'scale-invariant error' metric, which focuses the learning efforts on predicting local spatial relations correctly. This error metric uses a log-scale, which is a more natural choice for measuring depth errors than a linear scale (e.g. making 20 cm error at 1 meter is a bigger mistake than 20 cm on 15 meters).

Another approach by Godard et al. is capable of learning depth prediction without depth images, using only a training set of stereo image pairs [92]. This innovation makes collecting datasets easier. The authors are posing depth estimation as an image reconstruction problem, by learning to predict the disparity map from an input image. Given the baseline distance of the training image pairs, the depth can be calculated trivially from the disparity map. The network is trained to predict disparity maps for both left and right images, given only the left image. This added 'Left-Right Disparity Consistency' requirement has proven to increase prediction accuracy.

Even though regression might seem as a natural choice for learning depth estimation given the continuous nature of the depth, casting this problem as classification has its

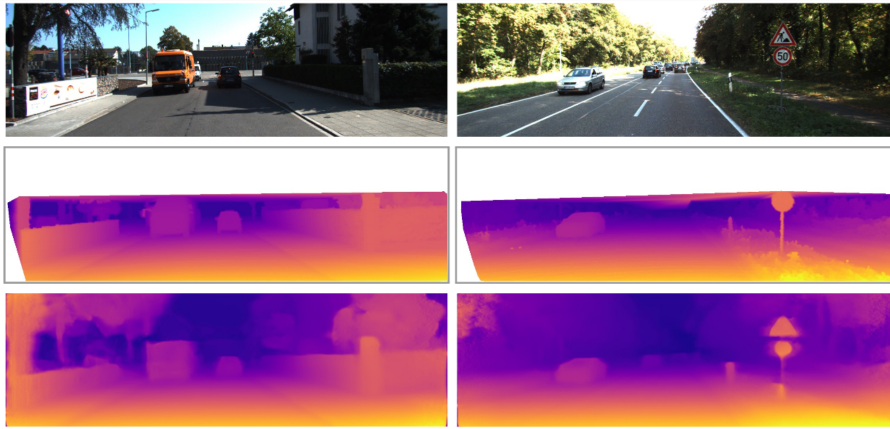

Figure 5.7: Depth prediction results [92] (top: input image, middle: ground truth, bottom: prediction)

advantages as well [93]. With depth quantization, the precision of the predictions can be different for different depth values (e.g. using fine-grained quantization for low distances and a more coarse quantization at higher distances). Also, this formulation gives the confidence of the predictions as well, since the classification outputs are providing a probability distribution. Cao et al. demonstrated that depth classification can outperform depth regression when local smoothness constraints are enforced on the outputs [93].

A closely related problem is *surface normal prediction*. The surface orientation is an important information about the 3D structure of the environment, and carries information about the derivative of depth [94]. Chen et al. explores using surface normal annotated images for improving depth prediction, by including a surface normal related terms in the loss function [94].

Auxiliary depth training

Depth prediction is one of the auxiliary tasks that Mirowski et al. used for the training of the NAV agent [85]. From their results it is apparent that using it as an auxiliary training can help the agent learn about the fundamental 3D nature in which it is embodied.

In their experiments simpler formulation of the depth prediction problem were proven to benefit the agent more: the best results were obtained with a low-resolution depth classification auxiliary training.

There is no prior experience on incorporating surface normal prediction as auxiliary task in RL training. In a sense depth and surface normal information might be redundant, however combining two different sources of information about the 3D structure of the environment may provide additional benefits over depth auxiliary training in isolation. As an example, learning about planar regions might be easier with surface normals, as it is constant over a planar region while the depth is gradually changing over the same region.

5.3.2 Optical Flow

Optical flow (the pattern of apparent motion of a visual scene) contains useful information about the motion of the observer and the 3D structure of the scene [95]. Therefore using this information to provide auxiliary training can be beneficial, especially for learning obstacle avoidance.

The first successful deep neural network that demonstrated end-to-end optical flow learning from pairs of input images was FlowNet [65]. The authors experimented with two architectures. FlowNetS works with stacked input images, and has a simple convolutional architecture, where contractive convolutional layers are followed by deconvolutional layers, to provide per-pixel optical flow estimates. FlowNetC on the other hand features two separate, yet identical streams for the first processing steps, then combines these preprocessed images in a later stage. The authors found that both architecture performed roughly on the same level. Since the available optical flow datasets were not sufficient for training deep models at that time, the authors generated a synthetic 'Flying Chairs' optical flow training set, which contained rendered chairs superimposed on various background images collected from Flickr. Still, the performance of FlowNet was behind traditional state-of-the-art methods. Ilg et al. improved FlowNet in various ways [68]. They found that by using *curriculum learning* (i.e. using more difficult training examples gradually) can significantly improve performance, especially for the FlowNetC architecture. The authors also experimented with iterative refinement, by stacking multiple FlowNets together.

Figure 5.8: FlowNet2.0 optical flow prediction [68]

Even though FlowNet demonstrates the capability of neural networks to learn optical flow prediction, the fact that it processes a pair of input images with a special parallel network architecture makes it less suitable for auxiliary training. Walker et al. on the other hand investigate dense optical flow prediction from single, static images [96]. This problem goes beyond traditional optical flow estimation, where the task can be solved by merely matching corresponding image parts. Predicting optical flow from a static image requires complex understanding of the context and some fundamental knowledge about the objects in the scene. The authors proposed a convolutional architecture, with pixel-wise classification to quantized optical flow directions.

Single image optical flow prediction can be considered as a candidate auxiliary task, especially if the most recently taken action is provided as an extra input. This modified problem (i.e. optical flow prediction based on action) is easier, as it is more deterministic. Furthermore it can teach the agent the connections between its actions and the changes in its perceptual stream, besides providing information about the ego-motion and 3D structure of the environment.

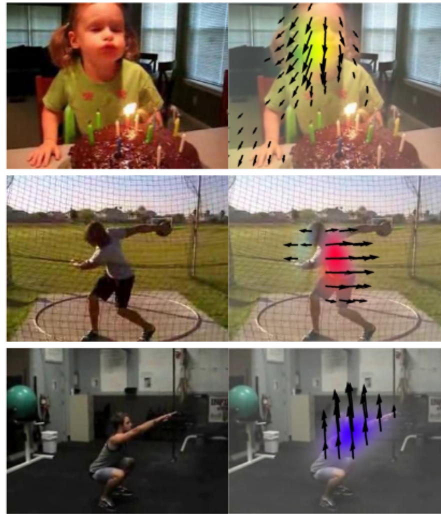

Figure 5.9: Single image optical flow prediction results [96]

5.3.3 Ego-motion estimation

Estimating the ego-motion is a rather useful capability for an autonomous agent that is operating in a 3D environment. The loop closure prediction auxiliary task of the NAV agent encouraged building this capability. The NAV agent receives (egocentric) velocity information, and loop closure prediction requires the agent to integrate these measurements over time.

In the absence of velocity measurements, it is still possible to estimate ego-motion from camera images. Visual odometry is the incremental calculation of the camera ego-motion between consecutive images. Just like depth estimation, monocular visual odometry has the problem of scale ambiguity. Visual odometry is a challenging problem, and algorithms addressing this usually rely on optical flow calculations. A very recent deep learning approach for visual odometry, VINet [97] is also relies on optical flow in a sense that its convolutional layers are initialized with a pre-trained FlowNet [65]. VINet is based on a sequence-to-sequence learning approach, using LSTM cells. Scale ambiguity is handled by integrating visual and inertial measurements. Since inertial measurements typically arrive at an order of magnitude higher rate, inertial input pre-processing has its dedicated LSTMs operating on a corresponding rate. VINet demonstrated comparable performance to traditional visual odometry methods, with its main advantage being robustness to calibration errors [97]. Still, this formulation of ego-motion calculation might be overly challenging for being used as an auxiliary task.

More simple formulation of the ego-motion estimation problem are also possible. As an example, one step displacement or velocity estimates can be calculated, for which the supervision may come from IMU or other measurements.

5.3.4 Detection and semantic segmentation

Detecting or classifying features of the environment from input images is a relatively straightforward choice for an auxiliary task. It provides a very simple way to focus

the representation learning on important aspects of the environment, thus incorporating expert domain knowledge in the training. An example for this is the enemy detector of the FPS playing agent of Lample and Chaplot [90], which turned out to be a decisive factor for learning high performance policies in their experiments.

A shared representation can be made more sensitive to certain environmental features by using detection as an auxiliary task. The opposite effect, representation invariance can also be achieved using auxiliary detection tasks. The key is using an adversarial loss instead of simply backpropagating the detection loss, thus training the shared representation to degrade detection/classification performance. An example for this is domain adversarial training, which aims to train representations that do not differentiate between input domains [44].

Figure 5.10: Semantic segmentation of a traffic scene (source: AImotive.com)

Instead of doing classification on a whole-image level, convolutional neural networks are able to predict class labels on pixel level [6]. This spatially dense, per-pixel prediction task is referred to as *semantic segmentation*. The challenge of semantic segmentation is that global information (i.e. what the object is) needs to be fused with fine-grained local information (which pixels belong to it). Long et al. introduced two architectural changes to adapt typical recognition networks for semantic segmentation. First, they replaced the fully connected layers (which throw away spatial coordinates) with convolutional layers, thus making fully convolutional networks. The output of a fully convolutional network is a 'heatmap' instead of a single probability distribution. Second, predictions are combined from lower layers (with smaller receptive fields but finer details) and from the final output (which is more coarse, but contains global information). Predictions are then up-sampled to the same resolution and summed, to get a final output that combines local predictions that respect the global structure.

Semantic segmentation as an auxiliary task can provide a highly informative training signal to a shared representation, since it requires not only the prediction whether objects or features appear in the input image but also directly pinpoints which input pixels represented the objects on the input image.

5.3.5 Mapping

Humans and animals alike make use of 'cognitive maps' that helps them to reason about obstacles and free space [98]. Gupta et al. demonstrated jointly learning both mapping

a visual input to a 2D metric cognitive map, and route planning using this cognitive map [98]. The 'Cognitive Mapping and Planning' architecture they proposed is fully differentiable and learned end-to-end, using supervised (imitation) learning and the DAGGER algorithm [99]. The mapper part of the architecture is creating a latent 2D metric representation of the environment from first-person camera images, while the planner is making use of these latent maps for path planning based on a modified, hierarchical version of Value Iteration Networks [100].

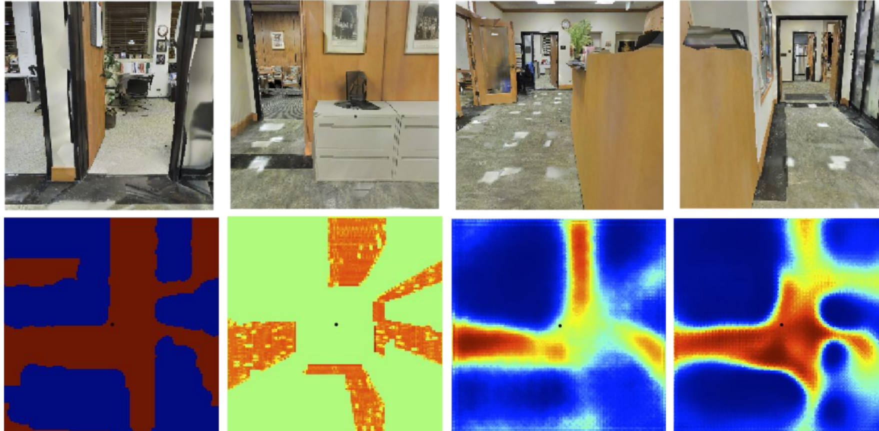

Figure 5.11: Output visualizations for the cognitive mapper. The top row shows the consecutive input images in 4 directions. Bottom row (left to right) shows ground truth free space, analytical projection of depth images, learned predictions using RGB and depth images. (source: [98])

From an auxiliary training point-of-view, mainly the experiments with the mapper in isolation are of interest. In this setting, the agent was rotating around (90° at a time) and made a free-space prediction of its surroundings. Supervision (ground truth free space map) was not provided at every time-step, only at the end of the rotation. Interestingly, the agent was able to give good predictions even for occluded regions from where no observations were available.

Free space mapping as an auxiliary task can teach the agent both about the 3D structure of the actual scene and how to interpret its visual inputs, but it can also provide agents with knowledge about regularities in their environments (like corners usually have a 90° angle, doors are leading to rooms, etc.). For navigation capabilities these auxiliary information might be rather helpful.

5.3.6 Unsupervised auxiliary tasks

Jaderberg et al. identified various useful auxiliary tasks that are useful for reinforcement learning. These tasks do not require any extra supervision from the environment, using only information that is already available for RL agents [86] (see Section 5.2.3).

On the other hand using more conventional unsupervised visual learning tasks for auxiliary training is also an option. One possibility for this is using training auto-ecoders on middle-level representation for reconstructing the input images, in a similar way it was

demonstrated for supervised networks (see Section 5.1.2 and [89]). However, such unsupervised objectives are are trying to reconstruct the whole input images. Even though they might yield efficient low-dimensional representations, still they do not focus the learning efforts on important aspects of the input. Furthermore reconstruction losses are usually based on some metric distance, like mean squared error between input and reconstructed images. Such a loss penalizes missing large features more, while missing small details does not lead to large reconstruction error.

These effects are even more pronounced if the unsupervised feature learning is used in isolation, to yield a low dimensional representation of the input images. Theoretically learning RL tasks would be easier using an informative low-dimensional representation. Still some result suggest that due to missing details the performance is actually worse in this case [101].

Chapter 6

Literature Review Discussion

In the preceding chapters an extensive literature study was given in the topics of Deep Learning, Domain Adaptation, Reinforcement Learning and Auxiliary Training, which together form the theoretical basis of this research. In this chapter the main findings and conclusions of this literature study are summarized.

6.1 Deep Learning

Deep learning is probably the most popular approach to statistical machine learning these days. In the recent years deep learning approaches started redefining the state-of-the-art in speech recognition, visual object recognition, natural language translation and in a number of other domains [9].

One particular network architecture, convolutional neural networks, are very efficient in processing images an other grid like data [8]. Therefore in most cases when working with image inputs the first few layers are convolutional layers, which form a convolutional feature extractor together. This feature extractor can be followed by fully connected or recurrent layers, or alternatively in some cases with further convolutional (or deconvolutional layers). In this project the aim is to demonstrate a vision-based policy for MAVs, therefore starting the network with a convolutional feature extractor is a natural choice in our case as well.

Since the field-of-view of the cameras used on MAVs is limited, with using only single images the state of the environment is not fully observable. Also, determining velocity states from single images is difficult. Despite this, simple reactive policies based on single images might be able to provide some level of obstacle avoidance. However, by integrating information from multiple images a better understanding of the situation can be achieved. For this, recurrent layer(s) can be added to the network, such as layers with LSTM units [42]. Training networks with internal recurrency is making the training process more complicated. An alternative choice is to feed subsequent images to the network at any given time. This is a simpler task to learn but less efficient since full images are

processed repeatedly, instead of carrying over just the relevant processed information in the state of recurrent units. Recurrent units also allow the network to learn retaining information as long as needed, while with feeding preceding frames this time horizon is limited.

For optimizing parameters in neural networks, deep learning typically relies on the backpropagation algorithm [22]. Other methods (such as genetic algorithms) can also be used for optimization, but these methods are less efficient for the high number of parameters common in large neural networks [21]. In our case, using image inputs already pushes the network out of the regime where genetic algorithms are considered to be a competitive alternative to backpropagation.

6.2 Domain Adaptation

A central problem of this research is the transfer of the simulation trained policies to real drones. Due to the systematic differences between the two domains, the performance on real environments is not guaranteed. Since the main learning task is a reinforcement learning problem, mainly unsupervised domain adaptation methods are applicable. The two main approaches are MMD distance based (e.g. [52], [53] and [54]) and domain adversarial training based methods (e.g. [44], [57]). Performance-wise it is not clear yet whether either approach is superior, as the state-of-the-art is being redefined every few months. However, domain adversarial training is easier to integrate to the multiple-objective training framework, due to its similarity to auxiliary tasks.

An interesting approach is domain randomization [58], which sidesteps the need for domain adaptation by randomizing many component of the simulation used. However, it requires hand-engineering variation to the simulator, and some capacity of the network is possibly wasted on adapting to any possible combination of the simulator (while only performance on the real domain is of interest).

When domain adaptation is used for RL tasks, it is important to consider that changes between domains are not necessarily restricted to changes in the input distribution. The behaviour and physics of the two domains might differ as well, which may render trained policies useless on the new domain. This is a significant difficulty for learning low level motor commands. On the other hand, in our case the learning task is higher level control. Higher level physical behaviour is expected to be easier to model, while the low level dynamics of the system can be handled by traditional feedback control.

6.3 Deep Reinforcement Learning

Deep reinforcement learning is a very promising approach for a wide variety of problems, and particularly suitable for learning intelligent control policies from raw image inputs. However, training a neural network with RL is challenging compared to standard supervised learning (which is already difficult in itself). After a short period when value-based methods (DQN [2] and its extensions) dominated deep RL, recently policy-based actor-critic methods are the most promising. At the moment the A3C algorithm seems to be an

optimal choice for our research, since it is a relatively established algorithm with multiple implementations available. On the other hand, an off-policy learning algorithms as DQN might be more flexible, as it allows learning from experiences collected under different policies. This makes off-board learning from real experiences easier. However, the significantly improved efficiency of on-policy algorithms might outweigh the advantages brought by off-policy learning. A3C improved the state-of-the-art in most Atari 2600 Games "in half the training time of other methods" [84].

6.4 Auxiliary training

Auxiliary training is a promising approach to improve the capabilities of neural networks. Auxiliary training allows the incorporation of additional training data in the learning process. On top of that, it also provides extra guidance for the training, as it can help to focus the learning efforts on the important aspects of the environment. In our case, understanding the 3D structure of the scene from monocular camera images is essential for learning obstacle avoidance and navigation capabilities. By giving related auxiliary tasks for the network, the shared representation can be trained to discover important depth clues.

For RL, the lack of information is an inherent problem, as the only feedback given to the learner is a possibly sparse scalar reward signal. Therefore auxiliary training is particularly beneficial for RL, as it provides more information and can significantly boost the learning of useful features. Interestingly, in some cases extra information was more useful when used for auxiliary training, than simply providing that information to the network as an extra input [85], [90].

As auxiliary tasks, both supervised, unsupervised or RL tasks can be used. In the context of learning obstacle avoidance and navigation policies for MAVs, we considered a number of different auxiliary tasks. The selection criteria was the usefulness of the predicted information for the main learning task, and also whether successful deep learning was demonstrated on the task. Another consideration is the suitability of the task for continual Self-Supervised Learning on-board the MAV. The most straightforward task is probably depth prediction, especially since its effectiveness was already demonstrated for related navigational tasks [85]. Detecting other properties related to the 3D structure of the environment (such as surface normals) might also help acquiring a useful feature extractor. Optical flow prediction can teach valuable information about both the 3D structure and the ego-motion of the agent. Detection and segmentation tasks can be used to guide the learning towards acquiring semantic information from the input images. Finally, a mapping auxiliary task can be used to teach regularities of the environment (such as typical rooms having 90° corners).

Part IV

Preliminary Thesis - Preliminary Analysis

Chapter 7

The learning task

In this chapter we define the main learning task, describe its role in a MAV autonomous software pipeline, and specify its inputs and outputs to/from other components of the control system. For this, we first discuss the typical control system hierarchy used in autonomous MAV flight, and then describe how our neural obstacle avoidance and navigation module fits into this scheme.

7.1 MAV control hierarchy

In a typical MAV a cascaded control system is responsible for flying the drone in a desired way. An example block diagram can be seen in Fig. 7.1. This is a high level example, actual control system architectures and implementations may vary.

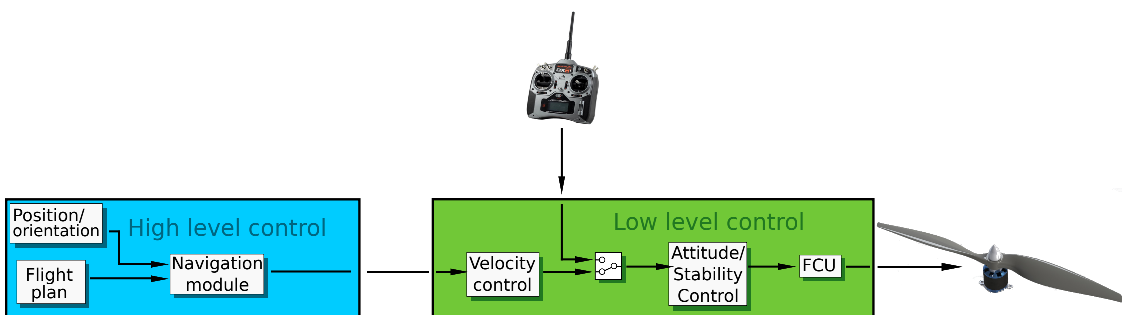

Figure 7.1: Typical control system for a MAV

Starting from the end of the hierarchy, the *actuators* are most often electric motors (aerial control surfaces with servos are used too for fixed/flapping wing MAVs, however the main focus here is on quadrotor MAVs). At the lowest control level a Flight Control Unit (FCU) is used to control the RPM of motors, and thus the force generated by the propellers.

The next control level is usually a *stability augmentation or attitude control loop*, which stabilizes the dynamics of the drone, and gives desirable handling characteristics. If the MAV is manually controlled, then normally it is the outermost control loop in the system.

Autopilots implement further control loops, such as velocity and position control. Usually the highest control loop is some sort of navigation loop. This can guide the MAV to a desired position according to a pre-defined flight plan or goal location, using some positioning system (e.g. GPS).

7.2 Obstacle Avoidance and Navigation module

In this research we are planning to augment the control system described in the previous chapter with a new middle-level module, as shown in Fig. 7.2.

A critical shortcoming of the control system outlined in the previous section is that it lacks a local trajectory planning step. Without this capability, we need to make sure beforehand that 'blindly' executing the flight plan is safe. This can either be done by flying in a safe airspace (high above all obstacles, free of other aerial vehicles), or by having perfect information of the environment and providing meticulous flight plans. However, these stringent requirements are significant constrains on the autonomous capabilities. The *Obstacle Avoidance & Navigation* module is giving the MAV the capability to actively react to its environment based on sensory input from a camera, and deviate from the flight plan locally if it is required for safe flight.

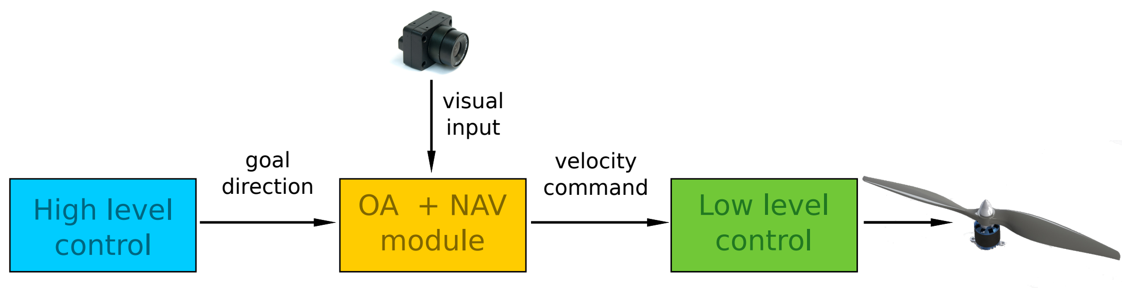

Figure 7.2: Control system augmented with the RL trained Obstacle Avoidance & Navigation module

In order to provide easy interfacing with the higher level navigation module, the input of the obstacle avoidance and navigation module is a relative goal direction. Given a positioning system (such as GPS or OptiTrack [102]), providing a goal direction is trivial. Most drones are also equipped with a magnetometer, which allows for measuring the heading even in GPS-denied environments. The best format for providing angular input for a neural networks is using the sine and cosine of the angle (in this case the relative heading angle of the goal direction), as this representations avoids periodicity issues and both function are restricted between ± 1.

In cluttered environments taking the shortest route and flying directly towards the goal location is usually not possible. The obstacle avoidance and navigation module is responsible to guide the drone towards its goal location in a safe manner. For this, the sometimes

competing goals (flying directly towards the goal vs. flying safely) have to be harmonized. The first priority of this module is obstacle avoidance: to avoid collisions by modifying the trajectory. It is also possible that taking the shortest route would get the MAV stuck in a dead end. In this case, the module should also be able to overcome this myopic behaviour and to help finding alternative directions, escaping these 'local minima'. In a sense this module functions as a local trajectory planner, and overrides the direction given by the navigation module (however there is no explicit trajectory planning, the trajectory is a result of repeated decisions).

The output of the obstacle avoidance module is a reference velocity vector, which is fed as a reference to the low-level controls. To make the RL task easier, the control outputs can be discretized. In the simplest case choosing from a few different horizontal directions is enough for obstacle avoidance, reducing the action space to 1D. Later on vertical directions can be added (2D) as well as speed commands (3D). Furthermore, continuous valued outputs can also used.

Theoretically this neural control module can be extended to both direction in the hierarchy. On one hand, higher hierarchical levels can be included. This would allow for taking higher level navigational goals (such as searching for an object, or exploring efficiently a building). This, however, would make the problem significantly more complex, furthermore it would make the system very specific. In contrast, when only a goal direction is taken as an input the module can be incorporated in various control systems, and used in more contexts.

On the other hand, the module can be extended towards lower hierarchical levels (e.g. directly outputting motor commands). Even though in theory it might lead to more efficient policies, again the learning task difficulty increases. Apart from this, traditional feedback control methods can guarantee stability, but for a neural controller these guarantees can not be provided.

7.3 Reward function

Reinforcement Learning relies on a reward signal, which penalizes the outcome of wrong decisions and rewards solving the RL task. The reward signal can be used to describe how an agent should behave in the environment, and communicate the intended learning goals.

In our case, there are two, sometimes competing goals:

- flying towards the goal location (r_{goal})
- avoiding collisions (r_{crash})

Therefore two corresponding terms can be included in the reward signal:

$$R_t = r_{goal} + r_{crash} \tag{7.1}$$

The first term, r_{goal} in the simplest case is a positive number once the goal is reached, and zero otherwise. However, this might lead to rather sparse rewards, thus slow learning.

A better formulation can be given using a goal distance dependent term, such that the reward increases as the agent is getting closer to the goal location. This acts as a potential field attracting the agent. An alternative reward can be the dot product of the velocity vector and the goal direction. This reward term does not require knowing the exact goal location.

In order to avoid crashes, r_{crash} need to be set to a sufficiently high negative value to penalize collisions. It is also possible to penalize not only crashes but dangerous near-collision situations as well, in order to encourage the agent keeping a safe distance from obstacles.

It is important to set the relative values of these reward terms and the discount factor properly. If the r_{goal} term is too high and the discount factor is low, then the agent might easily get stuck in dead-ends, as flying against the goal location requires trading of some immediate rewards for higher long-term returns. If r_{goal} is high compared to r_{crash} term, then it might encourage the agent to collide into objects just to get closer to the goal location. On the other hand r_{goal} needs to be strong enough to keep the agent moving towards the goal, even when it needs to fly close to obstacles in order to get there.

If needed, other terms can be added to the reward signal, to correct for unintended outcomes of the learning process.

Chapter 8

Auxiliary training with real data

In this project, we are setting up a multi-objective training with auxiliary training tasks. Most of these tasks are supervised learning problems. Supervised learning does not involve interactions with the environment, and this setting allows more flexibility in the training of these auxiliary tasks.

In this chapter we discuss how can we incorporate real images in the virtual training process of an RL agent when auxiliary tasks are used, what benefits we expect, and some related considerations.

8.1 Incorporating real images in the training process

Even though recent algorithms keep improving the sample efficiency of deep RL, training directly on physical robots is still impractical, as it requires either excessive training times or parallelization over multiple machines [103]. Therefore training the RL task in a simulated environment is still easier, furthermore cheaper and safer.

However, as auxiliary tasks do not require interaction with a system, a simulator is not essential for learning these tasks. In principle, any available dataset can be incorporated in auxiliary training processes. This flexibility allows for using training images for these tasks from our target domain, the real environment. Layers trained with added real images can learn features that are useful in both domains, efficiently bridging the domain discrepancy in the feature extractor with supervised domain adaptation. As the feature extractor is shared across all tasks, this way the RL task can also benefit from real images, despite being trained on the simulator only. Compared to unsupervised domain adaptation, supervised domain adaptation is more efficient and generally an easier task. Its main advantage is that it can directly learn features that only exist in the target domain, whereas in unsupervised domain transfer only the feature distributions can be matched.

In an ideal case the feature extractor is learning features that are useful on both domains, therefore the RL agent can work effectively in both domains using the same set of features.

In the worst case however, theoretically it is also possible that the feature extractor learns two separate set of features for the two domains, with the RL agent using only synthetic features. In this extreme scenario the performance in the real domain would be seriously deteriorated. Constraining the capacity of the shared feature representation, or regularization might help preventing this failure. Alternatively, unsupervised domain adaptation might also eliminate this problem. However, most probably the real performance of the system is somewhere between the two extremes. Therefore practical domain adaptation tests are required to assess whether this issue present when real data is used for bridging the domain discrepancy.

8.2 Mixing synthetic and real data in the training process

As we discussed in the previous section, auxiliary tasks can be trained with real images instead of synthetic ones. It is also possible to entirely exclude synthetic images from the auxiliary training process, and use only real images.

This, on the other hand can be a sub-optimal way to train the network, if we already have a simulator with sufficient fidelity. As we discussed in Section 3.5, adding extra synthetic data can help the training process, especially when the amount of available real data is limited. Another advantage is that given perfect knowledge of the synthetic data generation process (if we have access to the simulator), then perfect and dense ground truth can be provided for the synthetic examples. Therefore in some cases it is advisable to keep using synthetic data for these auxiliary tasks. However, if there is a large amount of available real training images with high quality ground truth information (e.g, in the case of SSL), and/or the fidelity of the simulator is not sufficient, then it is better to rely on real data only.

Finding the optimal ratios for real and synthetic images requires experimentation with the training of the agent. Besides this, the ratio does not have to be constant during the training: it can be a scheduled variable. For example we can start with more synthetic data, but gradually shift more towards real data at the end of the training, thus essentially finetuning the auxiliary tasks to the target domain.

8.3 Sources of real training data for auxiliary training

The training of the auxiliary tasks is not constrained to the simulator, as we discussed it earlier this chapter. In this section we discuss two sources for training data: public datasets, or collecting our own datasets.

8.3.1 Public Datasets

Possible sources for training data are the various publicly available datasets. A number of datasets are published for various learning tasks ranging from image classification and various regression problems to sentiment analysis. Multiple depth prediction datasets are available, such as the NYU V2 dataset (see Fig. 8.1) [104]. Also driving-related datasets

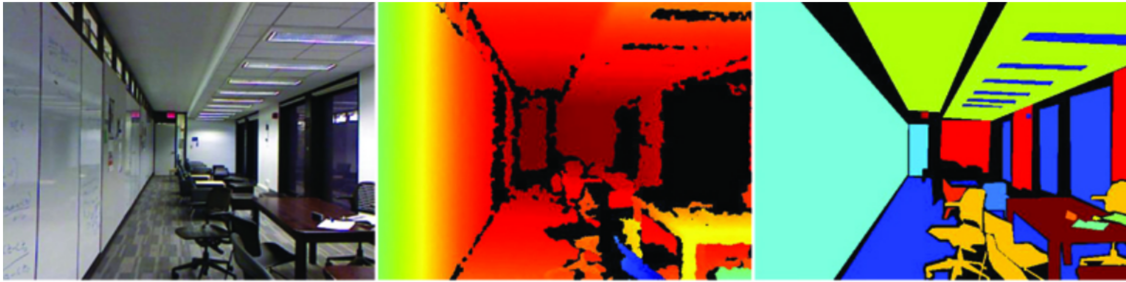

Figure 8.1: An example image (left) with depth (center) and semantic segmentation annotation (right) from the NYU Depth V2 dataset [104]

were made available such as the CityScape [64] datasets with semantic segmentation annotation; or the KITTI dataset [105], where stereo driving sequences are supplemented with ground truth from LIDAR and GPS measurements. The KITTI dataset also contains ground truth for optical flow prediction, as shown in Fig. 8.2.

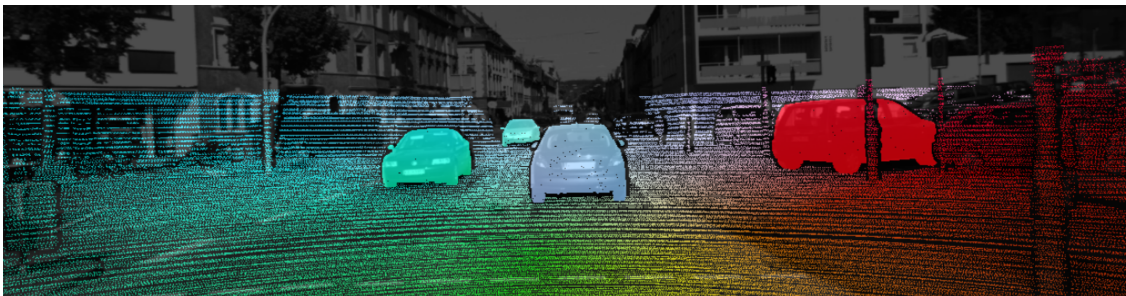

Figure 8.2: Sparse optical flow ground truth for a sequence from the KITTI dataset [105]

It is important to note that using various datasets for training an auxiliary task might require the harmonization of the category labels for classification tasks, or the harmonization of scales for regression tasks.

8.3.2 Collecting new datasets

Another option is collecting our own datasets. These datasets can be collected directly with the MAV, or alternatively with different sensors. The main advantage of collecting our own dataset is that it allows for training on images directly from the environment on which we intend to use the MAV afterwards.

For our multi-objective learning set up, depth ground truth can be provided from stereo images, or with an RGBD camera. For other prediction tasks, such optical flow prediction, ground truth data can be calculated from sequences of images using traditional algorithms. For other tasks related to ego-motion prediction, data from on-board sensors (gyros and accelerometers) or off-board sensors (such as the OptiTrack system [102]) can be used.

8.4 Specificity problems of the real datasets

Using public datasets or collecting our own, however, raises two interesting questions about the specificity of these data sets. First, is it important to collect the training images from the exact same environment, where we intend to fly the drone later? Second, is it important to collect the data under the same policy which governs the MAV later, thus ensuring matching visited state distributions?

So far we have been referring to our target domain as the 'real domain'. However, there is a great variety between real visual scenes, therefore it is unclear whether we can treat real images as if they were coming from the same distribution. In an ideal case, the feature extractor learns to detect general 3D clues, which are applicable to any real environments. The other extreme is learning very scene specific features, for example relying on the color of the wall or other specific properties of an environment. In this case flying to another room, or just the presence of new objects might seriously deteriorate the performance. Also it would practically prevent using public datasets. Probably the network can learn both specific and general features, therefore adding images from the testing environment will likely improve the performance. However the extent of the dependence on such training data needs to be investigated.[1]

The second question concerns the policy under which the training set is collected. The policy is directly related to the visited state distribution. The MAV can enter states which were not encountered during the collection of the training data. This problem is well-known in the context of 'Learning from Demonstrations' (LfD). There a good behaviour is learned from expert demonstrations, however the learner's decisions may cause departure from the expert trajectories [99]. Therefore the learner experiences a different visited state distribution, and may encounter states which were not included in its training, violating the usual i.i.d. assumption. However, while in the LfD setting the policy is being learned directly, with auxiliary training only a shared representation is trained. The different visited state distribution may influence the learned policy, but only indirectly. Therefore experiments are required to find out whether these effects significantly influence the learned policy, or not.

In case either the visited state distribution, or the specific source location turns out to be an important factor during auxiliary training with real images, then persistent Self-Supervised Learning [106] (SSL) can be a suitable learning method. In persistent SSL the MAV can fly under the control of an RL policy, while it collects data and ground truth with its own sensors. In theory, with this information the agent can keep training on the auxiliary tasks on-line, while flying. However, an off-line training with iterated periods of data collection and training (similar to the DAgger algorithm [99] for imitation learning) might be easier to implement in practice. This is partly due to the limited on-board resources. Another important difficulty is that auxiliary tasks need to be trained jointly with the RL task, otherwise the RL task performance would deteriorate. In an offline setting the RL training can still be trained in a simulator.

[1] This first question is also relevant to synthetic datasets to some extent, i.e. it is also a question whether for training we need an exact 3D reconstruction of a specific environment. Training in a virtual forest for example might be less useful for indoor flying, but is it important to model the exact rooms where the MAV is going to operate, or is it sufficient to train in similar virtual office environments?

Chapter 9

Network architecture

Constructing a suitable neural network architecture is a crucial part of this project. In this chapter the design choices are discussed.

In the first section, we introduce the base network which is trained for the main Reinforcement Learning task. Then in the following two sections the architectural considerations brought by auxiliary training and domain adversarial training are discussed.

9.1 Base RL network

Constructing a neural network from scratch involves a number of architectural decisions. Amongst other choices, the number of layers (depth), the number of hidden units in each layer (width), and connectivity patterns between layers have to be decided. These decisions can all be considered hyperparameters of the optimization process.

Unfortunately finding the optimal values is a non-convex optimization problem in itself, in which evaluation of a hyperparameter set requires complete training of the network. This optimization is computationally extremely expensive. Apart from some recent meta-learning approaches (where a network is trained with RL to design other networks [107], or in other cases networks are evolved [108]), the usual approach to hyperparameter optimization is grid- or random search. These optimization procedures are still expensive as training a single network may take days to finish. Therefore it is common to use a proven network architecture such as AlexNet [3], VGGNet [37] or ResNet [4]. In this section we review some neural architectures for deep RL (see the comparison of the discussed network architectures at Table 9.1).

In deep RL, the first successful network architecture was the Deep-Q Network [2], which has an architecture that was inspired by AlexNet. The input images (four consecutive stacked grayscale images) are fed to a convolutional encoder that has three convolutional layers, and ReLU activations. This is followed by a fully connected layer with 512 neurons, and finally a linear layer connecting to the 4 - 18 outputs (depending on the number of available actions). This network contains over 1.6 million parameters.

	Agent				
Layer	**DQN** [2]	**A-DQN** [84]	**A3C FF** [84]	**A3C LSTM** [84]	**NAV** [85]
Input	84 x 84 x 4			84 x 84 x 3	
Conv1	8 x 8 x 32, stride 4	8 x 8 x 16, stride 4			
Conv2	4 x 4 x 64, stride 2	4 x 4 x 32, stride 2			
Conv3	3 x 3 x 64, stride 1	-			
FC4	512 ReLU	256 ReLU			
FC5				256 LSTM	64/128 LSTM
FC6					256 LSTM
Output	Q-values		V + softmax action probabilities		

Table 9.1: Neural network architectures used for Deep Reinforcement Learning

The asynchronous version of DQN halved the number of convolutional filters and hidden units, thus reducing the number of parameters below 700000 [84]. The A3C algorithm is using the same convolutional encoder, but in this case the fully connected layer is followed by a separate linear output for value prediction, and an n-way softmax for action-selection. The LSTM A3C variant contains an additional layer of 256 LSTM cells before the outputs. Mirowski et al. used the LSTM A3C agent as a starting point and baseline for their NAV agent [85]. If the auxiliary training and extra inputs are excluded, then their base agent is an LSTM A3C agent, with two LSTM layers.

As a starting point for this research project, we can use the an A3C LSTM or NAV base network. This initial network is then augmented with auxiliary tasks and domain adversarial training, as discussed in the following sections.

9.2 Auxiliary tasks

Auxiliary tasks are incorporated in the training process in order to make use of more information in building of a shared representation. The main learning task (the RL task in our case) can therefore benefit from the features of this shared representation, learning faster and possibly converging to better solutions.

There are three new design choices introduced by an auxiliary task:

- The number of separate layers dedicated to the auxiliary task, and the number of neurons in those layers.
- The branching location where this task-specific network is connected to the base network.
- The usage of any additional inputs for the auxiliary task.

As discussed in the previous section, the number of layers (depth) and number of neurons (width) in each layer are usually treated as a hyperparameter in neural network training. These parameters can either be determined in a hyperparameter search, or just set based on prior experiences or heuristics. It is worth noting that auxiliary outputs can

be attached without any extra hidden layers, simply as a linear or softmax output layer. However, in this case the backpropagated auxiliary training losses might act too 'directly' on the shared representation. This is because task specificity increases towards the last layers of a network [46]. The shared representation might be forced to be overly specific for one task, if it is the last hidden layer before an auxiliary output. Therefore it is better to provide more flexibility for learning the auxiliary task by dedicating at least one hidden layer to it. In practice these task-specific network parts are kept relatively simple, with no more than two hidden layers (e.g. [85] [86]).

In order to choose the branching location, again the task specificity of layers has to be considered. On the one hand, the network has to be able to specialize some layers on its main learning task, which favours an early branching. On the other hand, those layers which are not shared cannot benefit from the auxiliary training signals, so from this aspect a late branching is better. When choosing the branching location, also the relevance of the auxiliary task has to be considered. If it is closely related to the main learning task, then more layers can be shared, while for loosely related tasks an earlier branching is more beneficial. Mirowski et al. compared navigation training with auxiliary depth prediction branched out either after the convolutional layers, or from the last hidden (LSTM) layers [85]. In their experiments the latter architecture gave better results, since in this case the auxiliary training is "providing structure to the recurrent layer, not just the convolutional ones" [85].

Finally, additional inputs can be provided for the auxiliary tasks, if needed. As an example consider a task when the network has to predict changes in the visual input: optical flow prediction. Given only the current image, this is a rather challenging task, as the optical flow is dependent on both the 3D structure of the scene, but also on the action that the agent took. Predicting its own actions is less useful for the agent. Instead by providing this information the auxiliary task can be focused on learning about the 3D structure behind the input images and making connections between its actions and changes in the perceptual stream, which capability is more useful for the main learning task. It is important to note that the auxiliary tasks are not performed at test time (except for finetuning with SSL on a real robot). Therefore these auxiliary inputs are only needed during training, it is not necessary to provide them during normal use of the network on the main task.

9.3 Domain Adversarial training

The architectural design choices of domain adversarial training are closely related to those introduced by auxiliary training. In this case again we connect some task specific layers to the original network at a certain location. These layers form the domain discriminator network, which tries to predict the source domain of the input image based on hidden layer activations in the original network. The depth and width of the domain discriminator are hyperparameters.

The effect of the domain adversarial training is the opposite of an auxiliary detection task. While the latter tries to build a representation that is discriminative for the detection task, domain adversarial training strives to make the representation indiscriminative for the source domains. Therefore the branching location in this case determines the hierarchical

level on witch feature distributions are matched. Ideally the goal of domain adversarial training is avoiding to rely on some property of one domain that is not present in the other domain.

To illustrate this, consider a network trained on a car racing game for road scene segmentation. The network may learn to detect the exact textures used to visualize grass in the game. This gives a very high accuracy when evaluated on the game, but less for real input images. If there is a neuron that consistently activates in the presence of the virtual grass texture, then this activation pattern can easily be spotted by the domain discriminator network. Another neuron, which activates for a more general features instead (such as green regions), would be activated on both domains. This neuron cannot be used by the domain discriminator to recognize the source domain, and may be a useful feature on both domains.

Ideally domain adversarial training has to work on such mid-level representations, eliminating domain specific activations and encouraging to detect features present on both domains. If the domain discriminator is attached to one of the first few layers instead, then the feature extractor is given a difficult task, as it has to provide domain independent features with less layers (and thus less flexibility). In this case low level features are matched (such as corner detectors and color blobs), and it might easily lead to degenerate solutions where most of the information is discarded. On the other hand, if the matching is done at the highest layer, then basically the action selection statistics are matched. Even though these statistics might be influenced with changing input distributions, this level might be too far from the inputs for effectively rejecting learning domain-specific features.

9.4 The complete neural architecture

In Fig. 9.1 Agent one possible layout for the complete network is given, which summarizes our discussion about the possible agent architectures.

We illustrated the base network in the central part of the figure. It starts with a convolutional encoder, consisting of 2-3 convolutional layers that are shared across all tasks. Following the A3C LSTM architecture, a recurrent layer with LSTM cells follows. This recurrent layer may or may not be shared with other tasks (even though on the figure we illustrated the branching location for auxiliary tasks after this layer). Finally, the base network may contain a sub-network dedicated to the RL task, which is not shared. It may consist of one or more additional layers (e.g. one more LSTM layer as in the NAV agent). The base network is trained with an RL loss, calculated based on the reward received from the environment. In the figure we illustrated an actor-critic loss, but in principle it can be any other suitable RL loss.

In the figure we also included some possible auxiliary tasks. For simplicity we branched out all auxiliary tasks from the same location, past the LSTM layer. But in practice the branching location has to be considered for each task individually, and in an ideal case it should be optimized.

We can group the auxiliary tasks into two categories: those for which ground truth labels can be provided by self-supervision, and those that rely on some extra information

only available from simulator or from another source. We illustrated the first group of auxiliary tasks at the right side of the base network. For these tasks the training data may either come from the simulator, from public datasets, or alternatively from the MAVs own observations with ground truth from its sensors. Other auxiliary tasks, such as the mapping task may rely on some special information, which can not be provided on-board.

We also included the various data sources on the figure. As we discussed in Chapter 8, various parts of the network can be trained with different datasets. The learning on the RL task is largely relies on the simulator. Ideally given access to all internal states of the simulator, it can be used for any auxiliary task, but in practice some information might be difficult to get (such as optical flow ground truth which is sometimes obtained by modifying the motion-blur pipeline [109]). For some problems, such as depth prediction public datasets are available, which can be incorporated in the training process. Furthermore, given suitable sensors on-board the MAV, training can be done on-board with self-supervised learning. Finally, when using an unsupervised domain adaptation method (such as Domain Adversarial training [44]), unsupervised images from the target domain can be used as well.

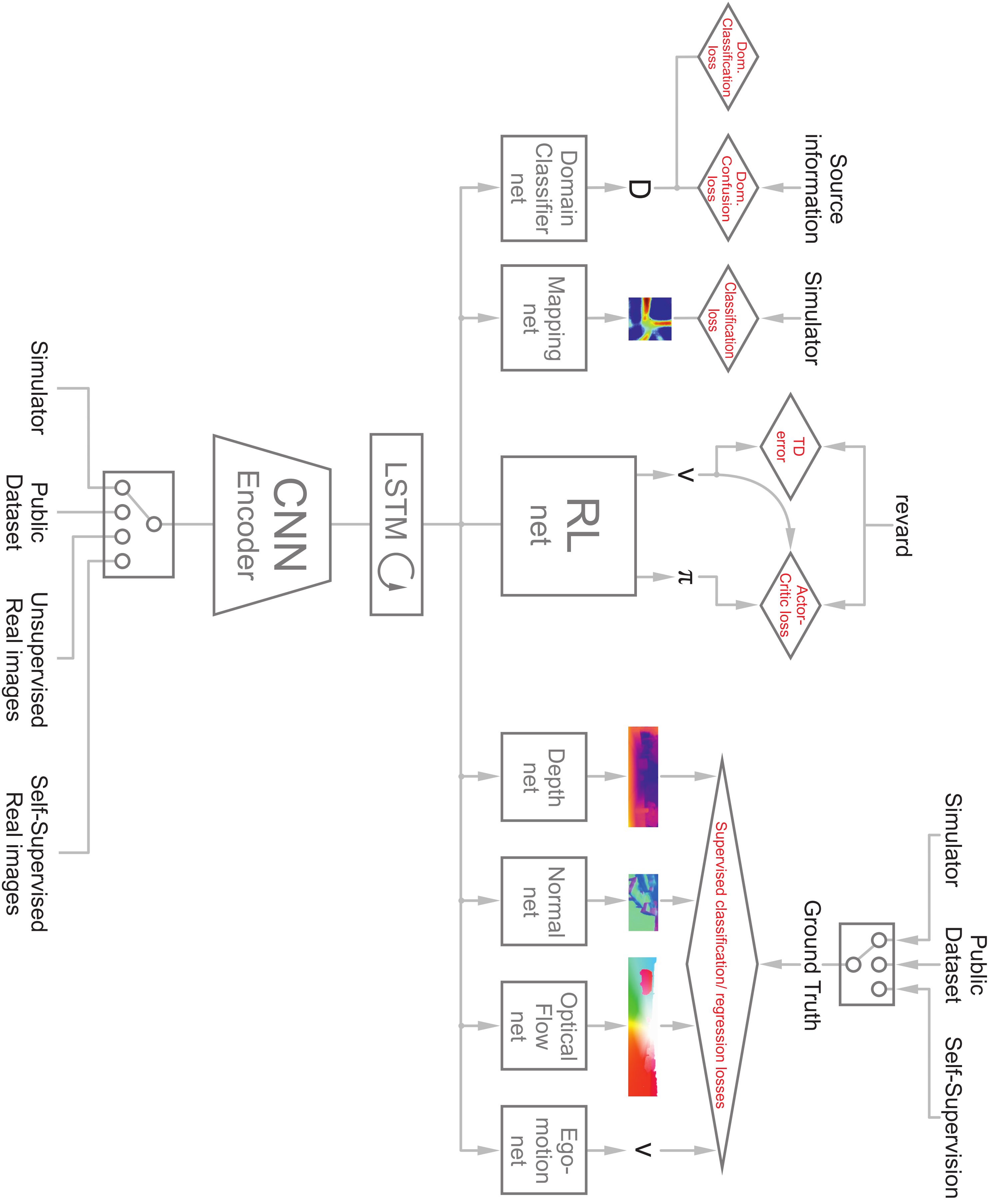

Figure 9.1: Network architecture combining Actor-Critic RL, multiple supervised- and self-supervised auxiliary tasks, and domain adversarial training.

Chapter 10

Remaining Thesis Work

In this chapter we discuss the remaining tasks of the research project. This chapter is accompanied by a project plan GANTT chart in Appendix A.

10.1 Simulation Environment Setup

Creating a suitable virtual environment for training an RL agent is a challenging task. However, the computer game industry spent a lot of effort on creating realistic games [110], and we can make use of this work by creating a simulated environment using a game engine. In this project, for the virtual RL training we chose to use the Unreal Engine 4 (UE4) simulation engine [111]. UE4 is an open-source cross-platform game engine, which features an exceptionally high quality rendering pipeline. UE4 also comes with a marketplace where a number of pre-made environments can be acquired, which allows for training in various environments.

The 3D environments for training might be some generic environments from the UE4 marketplace. However, if we wish to train the agent in an environment that is similar to the environment where we intend to use it, we may also need to create our own 3D environments. This is relatively easy using the built-in Editor of UE4, but modelling still require some work, and the quality of the scene might not reach the professionally designed environments.

Various plugins are available to make UE4 more useful for training RL agents. Microsoft Research has recently open-sourced a complex simulator, Airsim [112], which was built upon UE4. AirSim strives for realism both in terms of perception and physics. Currently it contains a high fidelity quadrotor model, and it supports both software-in-the-loop (SITL)and hardware-in-the-loop (HITL) simulations. AirSim can provide realistic renderings, depth views and object instance masks for training an agent with multimodal data.

An alternative UE4 plugin for agent training is UnrealCV. This plugin is targeted at generating virtual training datasets with multimodal ground truth for computer vision

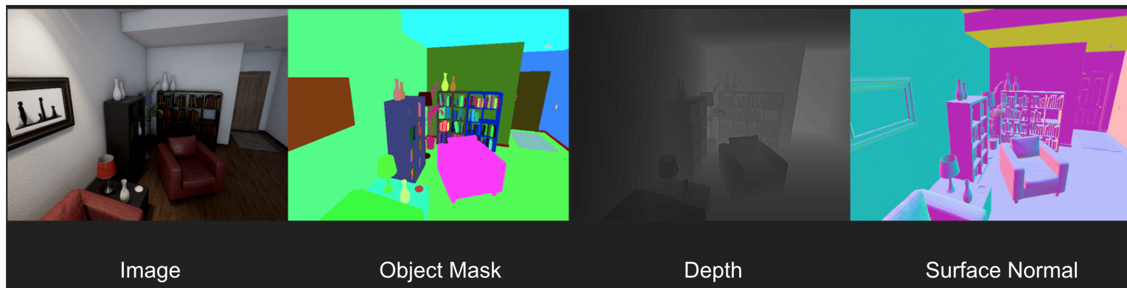

Figure 10.1: Images generated with UnrealCV [110]

algorithms. At this point depth view, object masks and normal maps can be generated as ground truth for learning tasks (see Fig. 10.1).

Even though AirSim and UnrealCV can both be suitable for training an RL agent, the focus of these plugins are different. The main advantage of AirSim is its physical model and HITL/SITL capabilities. However these capabilities are less crucial, as in this project we are training an agent to give higher level velocity commands. The dynamics of the MAV are going to be handled by a low-level control system. Speeding up the simulation is problematic in AirSim due to the real time components, furthermore the synchronization of camera images and ground truth images (depth, object instance masks, etc.) is not solved at the time of writing this report. These properties make it less suitable for training computer vision algorithms. UnrealCV completely lacks the vehicle simulation: the camera just 'teleports', and then remains stationary while the images and corresponding ground truth is rendered. This solution sidesteps the synchronization problem, but also has its disadvantages, most notably the lack of motion blur on the images. Still, at this point UnrealCV is more suitable for training an RL agent in the context of this project.

10.2 RL agent and Neural Network architecture setup

In order to start training an RL agent, first experience handling has to be solved. The experiences the agent collects during its learning process consist of the input image of the start state, ground truth for the auxiliary training tasks, the immediate reward received, and the image of the end state. These experiences are coming from the simulator, which runs an UnrealCV server. The agent can communicate to this server through a Python client, in order to access simulation data and give commands.

Then, the experiences are used in an RL training process, to train a neural network both for the main obstacle avoidance and navigation task, and for the auxiliary training tasks. For constructing the neural network architecture, various libraries can be used (e.g. TensorFlow [113], Theano [114], Caffe [115], etc.). As TensorFlow has currently the largest user community and the most available resources (e.g. multiple open-source A3C implementations are available), it was selected as the preferred library for this project.

10.3 Auxiliary training experiments in simulation

Once both the simulation environment and the RL architecture are finished, training in the simulation environment can be started. As a baseline experiment we will train the agent without any auxiliary task. Then experimentation with auxiliary training can be started. The main goal of this stage is to determine which auxiliary tasks are useful for the main learning task, and what is the best way to incorporate them.

The first auxiliary task can be depth prediction. Depth is relatively easy to access from UE4 thanks to the UnrealCV plugin. Also, the effect of depth prediction has already been studied for similar tasks [85]. Surface normals are easily accessible from UnrealCV as well, and because of its strong connection to depth prediction it might be a natural choice for the next auxiliary task.

Ego-motion prediction can be a relatively easy auxiliary task to implement, given the low dimensionality of this information compared to images.

Another promising auxiliary task is optical flow prediction. However, currently UnrealCV can not provide optical flow ground truth. Therefore implementing some extensions might be required in order to access this information from the simulator. One possible solution is to modify the motion blur pipeline for dense optical flow field generation [109].

To investigate the effects of free-space mapping as an auxiliary task, we need to extract a schematic top-view of the environment from the simulator. Alternatively, it is also possible to generate or create 'maps' of the environment separately, and feed the training process with the corresponding segments of these maps.

The number of experiments we can do with different auxiliary task combinations and architectural choices is limited by the available computational power and the time-frame of this project. Even though exhaustive investigation is out of reach, we still expect to find good auxiliary training combinations that are suitable for the subsequent domain adaptation experiments.

In general, we expect that RL combined with auxiliary training converges faster, and possibly also reaches higher cumulative rewards by the end of the training. If confirmed, then auxiliary training can be considered an effective method to integrate prior domain knowledge for Reinforcement Learning in real robotics tasks.

10.4 Setting up MAV experiments

Experiments on the real drones are a vital part of this project, as our goal is to demonstrate the effectiveness of simulation-trained policies on real MAVs. In order to do neural inference on-board, the sufficient computational resources have to be provided. We are planning to use the S.L.A.M.dunk by Parrot [116], which is an integrated extension kit for the Bebop2 MAV. It comes with an NVIDIA Tegra K1 mobile processor, an IMU, ultrasound sensors and a stereo camera.

From the software side, there are three main implementation tasks. First, we need to run inference on the neural networks on-board. Second, a low level control system is needed

that can execute the velocity commands that the neural network gives. These processes has to be integrated with the autopilot software running on the MAV.

In normal tests only inference of the main (RL related) part of the network is required for action selection. However for Self-Supervised Learning experiments not only inference is required for auxiliary tasks, but also error backpropagation to keep training the network. The required amount of computation might easily exceed the on-board capabilities of the MAV. Therefore either off-board but real time network training is needed, or alternatively a training process with iterated observation collecting and off-board training phases.

10.5 Domain Adaptation Experiments

We are investigating the effects of three different domain adaptation method: domain adversarial training, auxiliary training with real examples, and SSL. Both of these domain adaptation methods require a joint training in the virtual environment, with added real images for domain adaptation.

To establish a baseline, we are going to use an agent trained with auxiliary tasks, but without any domain adaptation technique. Then we see how transferable the original policy is, and will test our domain adaptation methods against this baseline.

To make more accurate comparisons between agents trained with different domain adaptation methods, we can make use of the OptiTrack indoor positioning system [102]. With OptiTrack, we have better control over the circumstances of experiments, for example we can start experiments from the same initial state. Furthermore, the OptiTrack system can accurately track the trajectories of the different agents, and based on these trajectories we can make quantitative comparisons.

With these experiments our expectation is that using a domain adaptation technique will increase the performance of the trained policy in the real domain. This can confirm our hypothesis that Reinforcement Learning methods can benefit from domain adaptation techniques, and that simulated training with domain adaptation might be sufficient for learning high level, intelligent control policies for robots. If policies are still not reliable enough even after domain adaptation, then possibly in order to achieve a required performance for safe operation SSL and training in the environment of intended use is required.

Bibliography

[1] Michael L. Littman. Reinforcement learning improves behaviour from evaluative feedback. *Nature*, 521 7553:445–51, 2015.

[2] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 1(518):529–533, 2 2015.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[5] Yuanzhouhan Cao, Zifeng Wu, and Chunhua Shen. Estimating depth from monocular images as classification using deep fully convolutional residual networks. *CoRR*, abs/1605.02305, 2016.

[6] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1605.06211, 2016.

[7] S Russell and P Norvig. *Artificial Intelligence: Modern Approach*. Pearson Education Limited, 1995.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[10] David A. Medler. A brief history of connectionism. In *Neural Computing Surveys*, chapter 1, pages 61–101. 1998.

[11] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML*, 2010.

[12] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[13] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, 1:886–893 vol. 1, 2005.

[14] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski. Orb: An efficient alternative to sift or surf. *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[15] Yoshua Bengio. Learning deep architectures for ai. *Foundations and Trends in Machine Learning*, 2:1–127, 2009.

[16] Haw minn Lu, Yeshaiahu Fainman, and Robert Hecht-Nielsen. Image manifolds. In *Proceedings of the SIPE Symposium on Electric Imaging: Science and Technology; Conference on Artificial Neural Networks in Image Processing III*, 1998.

[17] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *ICLR*, abs/1611.03530, 2016.

[18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.

[19] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[20] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals OR*, 134:19–67, 2005.

[21] Gregory Morse and Kenneth O. Stanley. Simple evolutionary optimization can rival stochastic gradient descent in neural networks. In *GECCO*, 2016.

[22] David E Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by backpropagating errors. *Nature*, 323:533–536, 9 1986.

[23] Leeon Bottou. Stochastic gradientdescent tricks. In Grégoire Montavon, Geneviève Orr, and Klaus-Robert Muller, editors, *Neural Networks: Tricks of the Trade*, chapter 18, pages 421–436. Springer, 2012.

[24] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 4 2012.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[26] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4:1–17, 1964.

[27] Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

[28] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[29] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *NIPS*, 2006.

[30] Rich Caruana, Steve Lawrence, and C. Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *NIPS*, 2000.

[31] Yoshua Bengio, Frédéric Bastien, Arnaud Bergeron, Nicolas Boulanger-Lewandowski, Thomas M. Breuel, Youssouf Chherawala, Moustapha Cissé, Myriam Côté, Dumitru Erhan, Jeremy Eustache, Xavier Glorot, Xavier Muller, Sylvain Pannetier Lebeuf, Razvan Pascanu, Salah Rifai, François Savard, and Guillaume Sicard. Deep learners benefit more from out-of-distribution examples. In *AISTATS*, 2011.

[32] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli. Analyzing noise in autoencoders and deep networks. *CoRR*, abs/1406.1831, 2014.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Simplifying neural nets by discovering flat minima. In *NIPS*, 1994.

[34] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[35] Yann LeCun. Generalization and Network Design Strategies. Technical Report CRG-TR-89-4, Department of Computer Science, University of Toronto, 06 1989.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[37] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[38] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, June 2015.

[39] Kenji Doya. Bifurcations in the learning of recurrent neural networks. In *IEEE International Symposium on Circuits and Systems*, pages 2777–2780, 1992.

[40] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML*, 2013.

[41] Herbert Jaeger and Harald Haas. Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication. *Science*, 304 5667:78–80, 2004.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[43] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.*, 22:1345–1359, 2010.

[44] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky. Domain-adversarial training of neural networks. *Journal of Machine Learning Research*, 17, 2016.

[45] S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, 2010.

[46] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *CoRR*, abs/1411.1792, 2014.

[47] Eric Tzeng, Judy Hoffman, Trevor Darrell, and Kate Saenko. Simultaneous deep transfer across domains and tasks. In *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[48] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015.

[49] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[50] Laurens van der Maaten, Geoffrey Hinton, and Yoshua Bengio. Visualizing data using t-sne. *Journal of Machine Learning Research*, 1:1–48.

[51] Baochen Sun, Jiashi Feng, and Kate Saenko. Return of frustratingly easy domain adaptation. *CoRR*, abs/1511.05547, 2016.

[52] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. Deep domain confusion: Maximizing for domain invariance. *CoRR*, abs/1412.3474, 2014.

[53] Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I. Jordan. Learning transferable features with deep adaptation networks. In *ICML*, 2015.

[54] Mingsheng Long, Jianmin Wang, and Michael I. Jordan. Unsupervised domain adaptation with residual transfer networks. *CoRR*, abs/1602.04433, 2016.

[55] Karsten M. Borgwardt, Arthur Gretton, Malte J. Rasch, Hans-Peter Kriegel, Bernhard Schölkopf, and Alexander J. Smola. Integrating structured biological data by kernel maximum mean discrepancy. In *ISMB*, 2006.

[56] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *NIPS*, 2014.

[57] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. Adversarial discriminative domain adaptation. *CoRR*, abs/1702.05464, 2017.

[58] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *ArXiv e-prints*, March 2017.

[59] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *I. J. Robotics Res.*, 32:1238–1274, 2013.

[60] Fereshteh Sadeghi and Sergey Levine. $(\text{CAD})^2$RL: Real singel-image flight without a singel real image. *arXiv preprint arXiv:1611.04201*, 2016.

[61] Erik Tzeng, Coline Devin, Judy Hoffman, Chelsea Finn, Pieter Abbeel, Sergey Levine, Kate Saenko, and Trevor Darell. Adapting deep visuomotor representations with pairwise weak constraints. In *Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2016.

[62] Andrei A. Rusu, Matej Vecerik, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets. *CoRR*, abs/1610.04286, 2016.

[63] Abhishek Gupta, Coline Devin, YuXuan Liu, Pieter Abbeel, and Sergey Levine. Learning invariant feature spaces to transfer skills with reinforcement learning. In *International Conference of Learning Representations (ICLR)*, 2017.

[64] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, 2016.

[65] Alexey Dosovitskiy, Philipp Fischer, Eddy Ilg, Philip Häusser, Caner Hazirbas, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, and Thomas Brox. Flownet: Learning optical flow with convolutional networks. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2758–2766, 2015.

[66] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *ECCV*, 2016.

[67] Matthew Johnson-Roberson, Charles Barto, Rounak Mehta, Sharath Nittur Sridhar, and Ram Vasudevan. Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks? *CoRR*, abs/1610.01983, 2016.

[68] Eddy Ilg, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. Flownet 2.0: Evolution of optical flow estimation with deep networks. *CoRR*, abs/1612.01925, 2016.

[69] D. Silver et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 1(529):484–489, 1 2016.

[70] R. Evans. Deepmind ai reduces google data centre cooling bill by 40%. `https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/`, 7 2016. [Online; accessed 13-March-2017].

[71] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 2 edition, 2017.

[72] David Silver. Advanced topics: Reinforcement learning – lecture slides. `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`, 7 2015. [Online; accessed 13-January-2016].

[73] John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438, 2015.

[74] G. de Croon and M. F. van Dartel. Evolutionary learning outperforms reinforcement learning on non-markovian tasks. 2005.

[75] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[76] G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report, Cambridge University Engineering Department, 1994.

[77] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999.

[78] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin A. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[79] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[80] Martin A. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In *ECML*, 2005.

[81] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[82] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[83] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[84] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[85] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *CoRR*, abs/1611.03673, 2016.

[86] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *CoRR*, abs/1611.05397, 2016.

[87] S. C. Suddarth and Y. L. Kergosien. Rule-injection hints as a means of improving network performance and learning time. In *EURASIP Workshop*, 1990.

[88] Rich Caruana. Multitask learning: A knowledge-based source of inductive bias. In *ICML*, 1993.

[89] Yuting Zhang, Kibok Lee, and Honglak Lee. Augmenting supervised neural networks with unsupervised objectives for large-scale image classification. In *ICML*, 2016.

[90] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, 2017.

[91] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *NIPS*, 2014.

[92] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Unsupervised monocular depth estimation with left-right consistency. *CoRR*, abs/1609.03677, 2016.

[93] Yuanzhouhan Cao, Zifeng Wu, and Chunhua Shen. Estimating depth from monocular images as classification using deep fully convolutional residual networks. *CoRR*, abs/1605.02305, 2016.

[94] Weifeng Chen, Donglai Xiang, and Jia Deng. Surface normals in the wild. 2017.

[95] H. C. Lounget-Higgins and K. Prazdyn. The interpretation of a moving retinal image. In *Proceedings of the Royal Society of London. Series B. Biological Sciences*, 1980.

[96] Jacob Walker, Abhinav Gupta, and Martial Hebert. Dense optical flow prediction from a static image. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2443–2451, 2015.

[97] Ronald Clark, Sen Wang, Hongkai Wen, Andrew Markham, and Agathoniki Trigoni. Vinet: Visual-inertial odometry as a sequence-to-sequence learning problem. In *AAAI*, 2017.

[98] Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. Cognitive mapping and planning for visual navigation. *CoRR*, abs/1702.03920, 2017.

[99] Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, 2011.

[100] Aviv Tamar, Sergey Levine, Pieter Abbeel, Yi Wu, and Garrett Thomas. Value iteration networks. In *NIPS*, 2016.

[101] Torstein Sandven. Visual pretraining for deep q-learning. Master's thesis, Norwegian University of Science and Technology, 2016.

[102] NaturalPoint Inc. OptiTrack. `https://optitrack.com/`, 2017. Accessed: 2017-06-18.

[103] Shixiang Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation. *CoRR*, abs/1610.00633, 2016.

[104] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012.

[105] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[106] Kevin van Hecke, Guido C. H. E. de Croon, Laurens van der Maaten, Daniel Hennes, and Dario Izzo. Persistent self-supervised learning principle: from stereo to monocular vision for obstacle avoidance. *CoRR*, abs/1603.08047, 2016.

[107] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[108] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[109] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black. A naturalistic open source movie for optical flow evaluation. In A. Fitzgibbon et al. (Eds.), editor, *European Conf. on Computer Vision (ECCV)*, Part IV, LNCS 7577, pages 611–625. Springer-Verlag, October 2012.

[110] Weichao Qiu and Alan Yuille. Unrealcv: Connecting computer vision to unreal engine. *arXiv preprint arXiv:1609.01326*, 2016.

[111] EPIC Games Inc. OptiTrack. `https://www.unrealengine.com/`, 2017. Accessed: 2017-06-25.

[112] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Aerial informatics and robotics platform. Technical report, February 2017.

[113] Abadi et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[114] Al-Rfou et al. Theano: A python framework for fast computation of mathematical expressions. *CoRR*, abs/1605.02688, 2016.

[115] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[116] Parrot Inc. SLAMdunk - All-In One Integrated Kit for Advanced Navigation Applications. `https://www.parrot.com/us/business-solutions/parrot-slamdunk`, 2017. Accessed: 2017-06-25.