TUHH
Hamburg University of Technology

ICS
Institute of Control Systems

Master's thesis

Domain Adaptation and Meta-Learning for End-to-End Autonomous Driving

Author:
Felix Wege

Supervisor: Patrick Göttsch, M.Sc.

Examiner: 1. Prof. Dr. Herbert Werner
2. Prof. Dr. Thomas Bräunl

May 25, 2020

ICS
Institute of Control Systems

Master's thesis
for Mr. Felix Wege

M — 11 — 2019
Student ID: 21379567

Study Course: IMPMEC

Title:
Domain Adaptation and Meta-Learning for End-to-End Autonomous Driving

Project Description:
Deep learning has been used to solve a number of problems ranging from robotic object manipulation to playing video games, [1], [2]. End-to-end learning is particularly interesting because it creates a mapping directly from sensor input to action, encompassing every interaction in between. These techniques and variations thereof have been successfully applied to the field of autonomous driving, [3].

Simulations are a powerful tool to test algorithms before deploying them on a real robot or automobile. However, machine learning algorithms can usually not be transferred from simulation to reality seamlessly, [1]. While training data in simulation is abundant, a recurring problem is the lack of training data in reality. This thesis aims to facilitate the transfer of behavior learning in simulation to reality. A variety of approaches exist to bridge the gap between simulation and reality, ranging from domain classification [4] and randomization [1] to meta-learning [5]. The focus of this thesis is to compare and combine these to enable a small-scale vehicle to follow lanes.

Tasks:

1. Literature survey on domain adaptation, meta-learning and related concepts
2. Learn to follow lanes in simulation
3. Implement approaches to transfer from simulation to reality
4. Compare and evaluate them in experiments

References:

[1] 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", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 23–30.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602*, 2013.

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars", *arXiv preprint arXiv:1604.07316*, 2016.

[4] Y. Ganin and V. Lempitsky, "Unsupervised domain adaptation by backpropagation", *arXiv preprint arXiv:1409.7495*, 2014.

[5] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks", in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1126–1135.

Supervisor: Patrick Göttsch, M.Sc.
Examiner: 1. Prof. Dr. Herbert Werner
2. Prof. Dr. Thomas Bräunl

External: University of Western Australia

Start Date: 16.12.2019
Due Date: 16.05.2020

May 25, 2020, Prof. Dr. H. Werner

Hereby I declare that I produced the present work myself only with the help of the indicated aids and sources.

Hamburg, May 25, 2020 Felix Wege

Abstract

Transferring systems desinged in simulation to reality is a difficult undertaking. The problem has been studied for decades, yet impedes the application and transferability of systems to date. With the advent of deep learning this problem intensifies since such systems are prone to overfitting. This thesis aims to employ deep learning methods to help bridge the gap between simulation and reality, applied to a lane-following task. It compares three different types of approaches to achieve this: Domain randomization, domain adaptation, and meta-learning. Utilizing randomized synthetic data, regularization between domains, and auxiliary datasets, these approaches learn a task from few examples in reality. A new approach using a combination of domain adaptation and randomization is proposed. All algorithms are implemented and evaluated on EyeBots, a mobile robotics platform developed by the University of Western Australia's Robotics and Automation Lab. Experiments show promising results on a held-out test dataset and on a real-world test course.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ADR active domain randomization

AR autonomy rating

cGAN image-conditioned generative adversarial network

CNN convolutional neural network

CyCADA cycle-consistent adversarial domain adaptation

DDC deep domain confusion

FCL fully connected layer

FOMAML first-order model-agnostic meta learning

GAN generative adversarial network

GDR guided domain randomization

KL Kullback-Leibler

L2S learning to simulate

LRUA least recently used access

LSTM long short-term memory

MAML model-agnostic meta learning

MANN memory-augmented neural network

MMD maximum mean discrepancy

MPMSE masked pairwise mean squared error

NTM Neural Turing machine

PixelDA pixel-level domain adaptation

RCAN randomized to canonical adaptation network

ReLU rectified linear unit

RKHS reproducing kernel Hilbert space

RNN recurrent neural network

SDR structured domain randomization

SVPG Stein variational policy gradient

UDR uniform domain randomization

UWA University of Western Australia

Chapter 1

Introduction

1.1 Deep Learning in Autonomous Driving

There are two main approaches to machine learning: Reinforcement learning and imitation learning. In reinforcement learning an agent is trained to take actions that maximize a cumulative reward. Imitation learning or supervised learning is concerned with approximating a function, given input-output pairs. Both can been applied to autonomous driving problems, [1], [2]. This thesis focuses on imitation learning, but all algorithms evaluated here can also be adapted and applied in a reinforcement learning setting.

A number of different problem settings arise in autonomous driving, such as obstacle avoidance, lane-following and lane-changing, parking. The environment can be uncharted or subject to changes and intentions of other road users are unknown. Perception of the environment is done by sensors, most prominently cameras and Lidar sensors. The focus of this thesis is lane-following using a single camera.

Autonomous driving encompasses many different tasks, such as recognition of surroundings and other vehicles, pedestrians, and traffic signs; prediction of future states of the environment, most importantly the movement of other road users; and planning to navigate successfully in the environment, integrating information about the it, [1]. Traditionally, these tasks would be solved independently by different subroutines but recently end-to-end learning has successfully solved a number of challenging tasks, ranging from autonomous driving [2] to playing video games [3]. This thesis focuses on end-to-end deep learning, i. e. mapping directly from sensor input to output action. End-to-end learning has the advantage of optimizing internal representation but comes at the disadvantage of reduced interpretability.

1.2 Motivation

The application of deep learning is hampered by its dependence on large datasets and countless training iterations, [4]. Collecting large dataset can be expensive or even

impossible for some applications. Also, a model trained on a specific dataset does not necessarily transfer well to another set of data. The error on previously unobserved data is known as the generalization error, [5].

Generalization error becomes an issue when training on simulated data. Since simulated data follows a different data-generating process than reality models trained in simulation do not transfer to reality without taking additional measures. For example, Hoffmann et al. report a drop in accuracy from 93 % to 54 % when deploying a semantic segmentation road detection model trained in simulation to reality, [6]. Clearly, there exists a gap between simulation and reality. This thesis aims to help bridge this gap.

Aside from the gap between simulation and reality there are other application for this kind of knowledge transfer. For example, consider the effects day and night or the seasons on the appearance of images. While the image of a road in summer differs from that of a road in winter the two are related and an intelligent system ought to understand this relation. Knowledge transfer is a universal problem in machine learning and not limited to autonomous driving.

1.3 Aim of this Thesis

The aim of this thesis is to implement a lane-following algorithms using end-to-end deep learning using as little labeled training data from reality as possible. Lane-following is to be evaluated on a track in simulation and reality using an EyeBot vehicle. The track is inspired by the Carolo-Cup, an annual competition of universities to advance autonomous driving [1]. To be able to run in real-time the runtime per iteration on the target platform should be no higher than 100 ms. Three different approaches are compared in this thesis: Domain randomization, domain adaptation, and meta-learning.

Domain randomization aims generalize to new domains by training on randomized data. If training data in simulation is sufficiently diverse reality appears to be like just another randomization, [4].

Domain adaptation tries to bridge the gap between reality and simulation by penalizing models that behave differently in those two domains. This is often achieved by adding loss terms to the objective function that penalize a distance metric, [7].

Meta-learning trains a model to be able to adapt to new tasks fast by incorporating learning new tasks over and over again during training, [8]. A model can then be fine-tuned on a few examples with a few gradient steps.

1.4 Chapter Outline

This thesis is structured as follows. Chapter 2 briefly describes fundamentals of deep learning that are required for the following chapters. In chapter 3 a variety of approaches

[1] Visit `carolo-cup.de` for more information.

to transfer deep learning from simulation to reality are explained. The hardware and software used for implementation of few selected algorithms is described in chapter 4. In the following chapter, chapter 5, experimental results are compared and discussed. Finally, chapter 6 concludes this thesis and gives an outlook to future work.

Chapter 2

Fundamentals

This chapter is not an introduction to deep learning. It assumes the reader is already familiar with deep learning. This chapter merely states some important formulae and ideas that serve as reference and are extended upon in the following chapters. For an in-depth treatment of deep learning see the cited sources, first and foremost Goodfellow et al., [5].

First, Section 2.1 introduces nomenclature used throughout this thesis. Section 2.2 defines loss functions and metrics that are commonly used in machine learning. Finally, Section 2.3 briefly describes some components of neural networks and the basic algorithm to train them: gradient descent.

2.1 Nomenclature

The following overview sheds light on nomenclature used in this thesis. It is inspired by Goodfellow et al., [5].

$f(x;\theta)$	A function f of x parameterized by θ
$\frac{\partial y}{\partial x}$	Partial derivative of y w. r. t. x
$\nabla_x y$	Gradient of y w. r. t. x
$P(x \mid y)$	Conditional probability of x given y
$x \sim P$	A random variable x with distribution P
$\mathbb{E}_{x\sim P}[f(x)]$	Expectation of $f(x)$ w. r. t. $P(x)$

2.2 Losses and Metrics

This section describes common machine learning loss functions and metrics that are used throughout this thesis. The difference between a loss function and a metric is that a metric is not used to compute gradients during training but merely serves evaluation purposes.

Softmax

The softmax function produces a probability distribution over n classes. Typically it is used to transform logits or unnormalized log-probabilities z_i of the final layer to a probability distribution. It is a generalization of the sigmoid function. The softmax function is defined as follows.

$$\operatorname{softmax}(z_i) = \frac{\exp z_i}{\sum_{j=1}^{n} \exp z_j} \tag{2.2.1}$$

Each $\operatorname{softmax}(z_i)$ is between 0 and 1. The sum of all $\operatorname{softmax}(z_i)$ is 1. Using a softmax output is convenient for maximum log-likelihood estimation because logarithm and exponentiation cancel out, [5].

Categorical Cross-Entropy

Given two discrete probability distributions p and q, the cross-entropy $H(p, q)$ is defined as

$$H(p, q) = \sum_i p_i \log \frac{1}{q_i} \tag{2.2.2}$$

$$= -\sum_i p_i \log q_i. \tag{2.2.3}$$

The categorical cross-entropy is often used as a loss function in classification settings. Let p be the ground-truth labels and q be the output of a classifier. Further, let the labels be one-hot encoded, i. e. . p_i is 1 for the correct class and 0 otherwise. Thus, all summands of Equation (2.2.2) are 0 except for one.

L1 and L2 Loss

The L_1 loss is given by

$$L_1 = \|x\|_1 \tag{2.2.4}$$

$$= \sum_i |x|, \tag{2.2.5}$$

whereas the L_2 loss is given by

$$L_2 = \|x\|_2 \tag{2.2.6}$$

$$= \sqrt[2]{\sum_i x^2}. \tag{2.2.7}$$

Both are commonly used in machine learning for regularization. The L_2 loss is the distance to the origin. It penalizes large values disproportionally. The L_1 loss is the sum of absolute values. It is useful when small deviations from zero should be penalized, [5].

Accuracy

In binary classification, accuracy is defined as the proportion of correct results, i. e. true positives and true negatives divided by the total number of positives and negatives.

$$\text{binary accuracy} = \frac{\text{TP} + \text{TN}}{\text{P} + \text{N}} \tag{2.2.8}$$

In can be extended to multiclass classification by accumulating true positives for each class and dividing by the total number of data points.

$$\text{categorical accuracy} = \frac{\sum \text{TP}}{\sum \text{P}} \tag{2.2.9}$$

Cosine Similarity

Cosine similarity—also called cosine distance—is a measure of similarity between two vectors. Given two vectors a and b it is defined as the inner product divided by the norm.

$$\begin{aligned} \text{cosine similarity}(a, b) &= \cos(\theta) \\ &= \frac{a \cdot b}{\|a\| \|b\|} \end{aligned} \tag{2.2.10}$$

The cosine similarity is between -1 and 1.

2.3 Neural Networks

This section briefly describes convolution, the building block of convolutional neural networks (CNNs), before introducing two variations of them: depthwise separable convolutions and inverted residuals with linear bottlenecks. It goes on to explain long short-term memorys (LSTMs), a recurrent component of neural networks. Next, the idea behind generative adversarial networks (GANs) is described. Finally, gradient descent is briefly outlined as it is the basic method for training neural networks.

Convolution

Convolutions are a common neural network operation for data arranged in an array, such as an image. They have contributed greatly to the success of neural networks. Three advantages of convolutions over fully-connected layers are sparse interactions, parameter sharing, and equivariant representations, [5].

Assuming a square input and output, let D_F be the input height and width, D_K be the kernel width and height, and M and N be the input and output depth, respectively. A convolution is parameterized by a $D_K^2 \times M \times N$ kernel K. It maps an input F of size $D_F \times D_F \times M$ to a $D_F \times D_F \times N$ output G, given appropriate padding, as follows, [9].

$$G_{k,l,n} = \sum_{i,j,m} K_{i,j,m,n} \cdot F_{k+i-1,l+j-1,m} \tag{2.3.1}$$

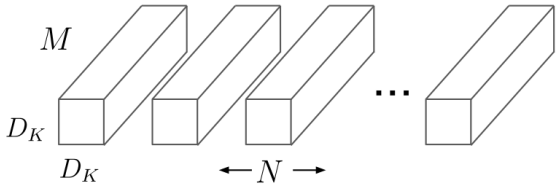
Convolutions have a computational cost of $D_K^2 \cdot M \cdot N \cdot D_F^2$.

Depthwise Separable Convolution

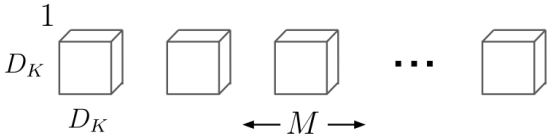
Depthwise separable convolutions are a factorization of standard convolutions. They were contrived by [10] and popularized by [9] through their application in MobileNetV1. A depthwise separable convolution consists of a depthwise convolution and a pointwise convolution. A depthwise convolution applies a single filter to every input channel. A pointwise convolution applies a 1×1 convolution to the outputs of the depthwise convolution, combining them. Given a depthwise convolutional kernel $\hat{K}$ of size $D_K \times D_K \times M$ a depthwise convolution can be written as

$$\hat{G}_{k,l,m} = \sum_{i,j} \hat{K}_{i,j,m} \cdot F_{k+i-1,l+j-1,m}. \tag{2.3.2}$$

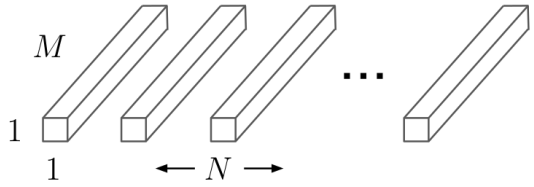
The depthwise convolution is followed by a pointwise convolution to compute a linear combination. The authors recommend following each of the two layers with batch normalization and rectified linear unit (ReLU), [9]. Figure 2.3.1 contrasts depthwise separable convolution filters with standard ones.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Figure 2.3.1: Comparison of standard, depthwise, and pointwise convolution filters. Source: Howard et al. [9]

Depthwise separable convolutions reduce number of parameters and computational cost. The computational cost of a depthwise separable convolution is $D_K^2 \cdot M \cdot D_F^2 + M \cdot N \cdot D_F^2$.

The reduction in computational cost and number of parameters compared to standard convolutions is

$$\frac{J_{standard}}{J_{separable}} = \frac{D_K^2 \cdot M \cdot D_F^2 + M \cdot N \cdot D_F^2}{D_K^2 \cdot M \cdot N \cdot D_F^2} \tag{2.3.3}$$

$$= \frac{1}{N} + \frac{1}{D_K^2}. \tag{2.3.4}$$

A smaller computational cost accelerates training and inference while a smaller number of parameters reduces the model capacity, [5].

Inverted Residual with Linear Bottleneck

Inverted residual with linear bottleneck blocks are used in MobileNetV2, an improvement of MobileNetV1, [11]. They use residual connections between low-dimensional bottleneck layers. The low-dimensional layers are linear while the high-dimensional expansion layers are non-linear.

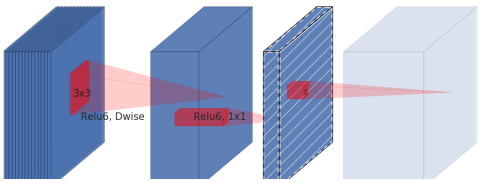
A bottleneck block takes a high-dimensional input and compresses it to a low-dimensional intermediate representation. Then, it applies a filter before mapping it back to many dimensions. The low-dimensional subspace embeds all necessary information. An expansion block does the opposite of a bottleneck. It takes a low-dimensional input, expands it to high dimensions, applies a filter, and maps back to low dimensions. Each layer uses depthwise separable convolutions.

Residual connections were proposed to tackle degrading accuracy of networks that were too deep to train, [12]. A residual connection is a connection between layers that are not adjacent; they are a shortcut between layers. They help with convergence rate and training stability. Residual connections are typicaly established between high-dimensional layers with bottlenecks in between. Inverted residuals on the other hand connect low-dimensional layers with expansion blocks between them. They are motivated by the fact that low-dimensional layers contain all necessary information. Figure 2.3.2 juxtaposes bottleneck, expansion, residual, and inverted residual blocks. Experiments suggest that inverted residuals perform better if some layers are linear, [11].

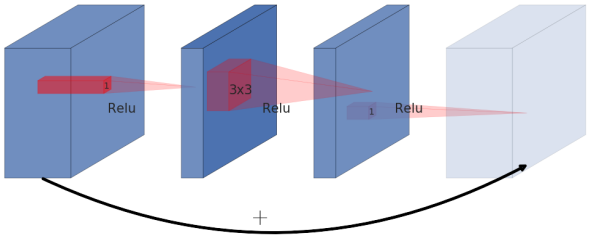
Long Short-Term Memory

LSTMs are a special type of recurrent neural network (RNN) that were first introduced in 1997 by Hochreiter and Schmidhuber, [13]. They are explicitly designed to capture long term dependencies and have been successfully employed in a number of applications, ranging from speech recognition to image captioning, [5]. LSTMs networks consist of a sequence of cells. Each unit has a cell state c and a hidden state h, both of which can be altered by previous states and inputs. Whether and to what degree the states are changed depends on a number of gating mechanisms.

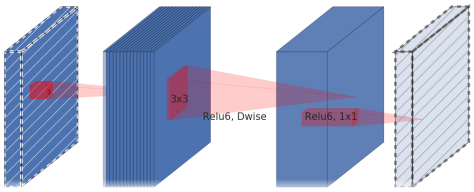
The forget gate controls how much of the previous cell state c_{t-1} is passed to the next cell



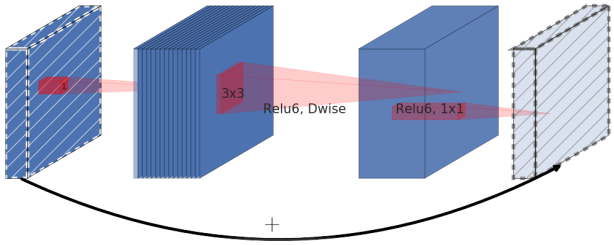
(a) Bottleneck block



(b) Residual block



(c) Expansion block



(d) Inverted Residual block

Figure 2.3.2: Comparison of different bottleneck and residual blocks. A block consists of multiple layers. Layer thickness indicates relative number of channels. Hatched texture layers do not use non-linearities. The last layer in lighter color marks the beginning of the next block. When stacked, bottleneck block and expansion block are the same, save for the beginning and end of the stack. The same is true for residual and inverted residual block, except the layers connected have different dimensions.

Source: Sandler et al. [11]

state c_t. Given the weights U_f, W_f and bias b_f the forget gate equation is given by

$$f_t = \sigma(U_f x_t + W_f h_{h-1} + b_f). \tag{2.3.5}$$

The forget gate output f_t is given by the sigmoid of the sum of bias plus weighted sum of inputs x_t and previous hidden state. It is in the interval $[0, 1]$, where 0 means "forget everything" and 1 means "remember everything".

The second gate is the input gate. It determines how much new information is added to the cell state. The input gate is given by

$$i_t = \sigma(U_g x_t + W_g h_{t-1} + b_g), \tag{2.3.6}$$

where U_g, W_g, and b_g are weights and biases. The input gate yields the intermediate cell state $\tilde{c}_t$.

$$\tilde{c}_t = \sigma(U x_t + W h_{t-1} + b) \tag{2.3.7}$$

The cell state is then updated by the following equation.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{2.3.8}$$

It is the sum of previous cell state c_{t-1} and modified state $\tilde{c}_t$, weighted by forget gate f_t and input gate i_t respectively.

The output gate controls what value to output and pass on to the next cell. It is computed from input x_t and previous hidden state h_{t-1} using output weights U_o and W_o and bias b_o.

$$q_t = \sigma(U_o x_t + W_o h_{t-1} + b_o) \tag{2.3.9}$$

The new hidden state h_t is given by the hyperbolic tangent of the output gate.

$$h_t = \tanh(c_t) q_t \tag{2.3.10}$$

Figure 2.3.3 shows the structure of an LSTM cell.

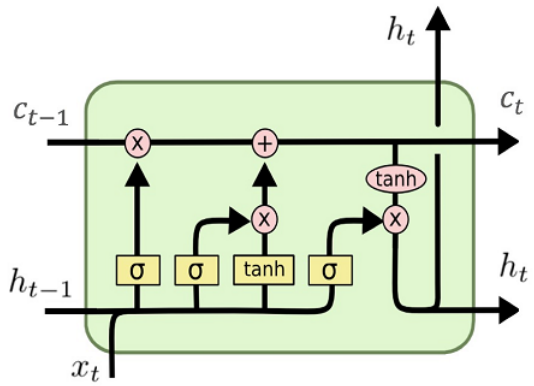


Figure 2.3.3: Long short-term memory memory cell structure. Cell state c and hidden state h are updated using input x and a number of additions, multiplications, sigmoid units, and hyperbolic tangent operations.

Source: Olah [14]

Generative Adversarial Networks

GANs juxtapose two competing networks: Generator G and discriminator D. The generator generates samples while the discriminator tries to distinguish generated sampled from real ones. The objective for training generator G and discriminator D is given by

$$L_{GAN}(G, D) = \mathbb{E}_x[\log D(x)] + \mathbb{E}_x[\log(1 - D(G_x(x)))]. \tag{2.3.11}$$

Both generator and discriminator are parameterized by a CNN. GANs can be used to generate synthetic training data or to map images from one domain to another, [5].

Gradient Descent

Gradient descent and variations thereof are the most prevalent method for training neural networks. Given a loss function $L(\theta)$ and its partial derivate w. r. t. parameters θ given by $\nabla_{\theta_{i-1}} L(\theta)$ the objective can be minimized iteratively by subtracting the gradient from the parameters, [5].

$$\theta_i = \theta_{i-1} - \alpha \nabla_{\theta_{i-1}} L(\theta) \tag{2.3.12}$$

Here, α is a learning rate that determines the step size. Gradient descend moves parameters in the direction that yield the best improvement of the objective function. The magnitude of the learning rate is important for convergence and stability and needs to be set carefully. Numerous improvement of gradient descent have been proposed using higher-order derivatives or momentum, [5].

Chapter 3

Transfer from Simulation to Reality

This chapter describes different approaches to bridge the gap between simulation and reality. The first two sections give a brief introduction to the problem of domain transfer. The remaining sections focus on three different categories to solve said problem: Domain randomization, domain adaptation and meta-learning.

First, Section 3.1 explains transfer learning in general and different aspects of it. Then, Section 3.2 looks at transfer from simulation to reality as a form of multi-task learning. Many algorithm presented in the following three sections can be formulated as multi-task. Section 3.3 describes domain randomization, a technique that aims to achieve generalization by randomization of training data. The next section, Section 3.4 is concerned with domain adaptation. Domain adaptation aims to generalize by minimizing some sort of loss that measures discrepancy between domains. Finally, Section 3.5 presents meta-learning algorithms that can be used to learn fast from few data points, which can be used to adapt to a new domain.

3.1 Transfer Learning

Most machine learning models perform worse on data from a different distribution and need to be retrained. Transfer learning acknowledges that training and test data can be drawn from a different distribution. Collecting new data with labels and retraining can be costly and impractical. Transfer learning aims to transfer knowledge to new tasks and domains. It is motivated by the fact that humans, when confronted with a new environment or new task, often benefit from previously learned tasks that are similar.

Let $D = \{X, P(X)\}$ be a domain, where X is a feature space and $P(X)$ is its marginal probability distribution. Two domains are different if their feature space, its marginal probability distribution or both are different. Given a domain D a task is defined by $T = \{Y, f(Y)\}$, where Y is a label space and f is an unobservable objective function. Two tasks are different if their label space, objective function or both are different.

In general there can be multiple source and target domains as well as multiple tasks. For the remainder of this chapter only one source domain D_S and one target domain D_T

with one task each (T_S and T_T, respectively) are considered. With the above definition of domains and tasks transfer learning is formally defined as follows, [15].

Definition 3.1.1 *Given a source domain D_S and target domain D_T with respective tasks T_S and T_T, the objective of transfer learning is to maximize performance of a predictor f_T on T_T using information from source domain and task, where $D_S \neq D_T$ and/or $T_S \neq T_T$.*

3.1.1 Types of Transfer Learning

There are several different types of transfer learning. When $D_S = D_T$ and $T_S = T_T$ the problem becomes regular machine learning. For all other combinations a different type of transfer learning occurs. Table 3.1 summarizes different types of transfer learning. The different types of transfer learning are called inductive, transductive, and unsupervised transfer learning. They are described in the following.

Table 3.1: Different types of transfer learning for different relationship between source and target domains and tasks. Domains and tasks can be the same (=) or different but related (≠), resulting in either inductive, transductive, unsupervised or no transfer learning. Table inspired by Pan et al., [15].

Learning setting		D_S and D_T	T_S and T_T
No transfer learning		=	=
Type of transfer learning	Inductive	=	≠
	Transductive	≠	=
	Unsupervised	≠	≠

Inductive Transfer Learning

Inductive transfer learning is defined as learning a new task in the same domain, i. e. $T_S \neq T_T$ but $D_S = D_T$. This setting is also referred to as multi-task learning (cf. Section 3.2). The task is source domain is exploited to improve performance on target task in the same domain. Inductive transfer learning requires at least a few labeled data points in target domain to induce the target task. An example for inductive transfer learning in autonomous driving is to predict features of a road in addition to predicting the steering, [16]. Both tasks are learned on data from the same domain. Learning multiple tasks can improve generalization.

Transductive Transfer Learning

Transductive transfer learning aims to learn the same task in a new domain, i. e. $T_S = T_T$ but $D_S \neq D_T$. This setting is also referred to as domain adaptation (cf. Section 3.4). It requires no labeled data in target domain, but can make use of such data if available. Transductive transfer learning exploits unlabeled target domain data in combination with

labeled source domain data to transfer a task across domains. An example for transductive transfer learning is learning a task in simulation and transferring it to reality. The task is identical but data in source and target domain have different marginal probability distributions. This thesis focuses on transductive transfer learning to transfer deep learning from simulation to reality.

Unsupervised Transfer Learning

In unsupervised transfer learning both domains and tasks are different, i. e. $T_S \neq T_T$ and $D_S \neq D_T$. The objective is to transfer one task to a different task in a different domain. In general, no labeled data is available for the target task in target domain. Unsupervised transfer learning is typically concerned with clustering or dimensionality reduction.

3.1.2 Approaches to Transfer Learning

There are several approaches to transfer learning. This section describes four common approaches: instance, feature representation, parameter, and relational knowledge transfer. Some of them are only applicable to certain types of transfer learning.

Instance Transfer

Instance transfer aims to transfer knowledge by weighting or sampling data from source domain to match the marginal probability distribution in target domain. This is justified if some of the data from source domain is assumed to be useful for the task in target domain. The sampling procedure itself can be regarded as an optimization with the objective to minimize target loss by sampling and weighting source data. Resampling or reweighting is only suitable for domain adaptation if the domains shift is limited to a change in marginal probability distribution, [17]. It can not bridge the gap between different visual domains.

Feature Representation Transfer

The goal of feature representation transfer is to find a feature representation that works well in multiple domains or for multiple tasks. Feature representation transfer can be achieved by additional loss terms that minimize a metric in feature space. Alternatively, when training a model to perform multiple tasks in multiple domains more robust features emerge. Feature representation transfer is applicable if source and target domain are structurally similar but visually different.

Parameter Transfer

Parameter transfer aims to find model parameters that can be shared across domains or tasks. The underlying assumption is that models for related tasks or domains should have similar parameters. Often, parameter transfer is used to initialize model parameters in a

reasonable way before fine-tuning on target domain data. Many deep learning applications use models pre-trained on large datasets such as ImageNet, [5]. While the dataset is unrelated to the specific task it provides a general-purpose feature extraction initialization. Some meta-learning algorithms explained in Section 3.5 can be used for optimal parameter initialization.

Relational Knowledge Transfer

Relational knowledge transfer differs from the other three approaches in that it explicitly tries to establish a relationship between source and target domain. As opposed to the other three approaches relational knowledge transfer relaxes the assumption of independent and identitally distributed source and target domain. Relational knowledge transfer postulates that a mapping between two related domains can be found. GAN-based approaches to domain adaptation explained in Section 3.4.2 belong to this category.

3.2 Multi-Task and Multi-Domain Learning

Multi-task learning refers to learning multiple tasks at the same time. It is related to inductive transfer learning (cf. Section 3.1.1). Multi-domain learning means to learning a task in multiple domains. As such, it is related to transductive transfer learning (cf. Section 3.1.1). In general, whenever more than one loss term is optimized, the problem setting can be formulated as a multi-task problem, [18]. Analogously, whenever data from more than one source is used, multi-domain learning occurs. Both are often employed with the intention to improve generalization. Multi-task and multi-domain learning are closely related. In a way, learning the same task in a new domain can be seen as learning an additional task. Thus, many of the algorithms described in this chapter constitute a form of multi-task learning.

It can be advantageous to learn multiple tasks simultaneously because by doing so a bias is introduces, thereby preferring some hypotheses over others, [18]. Learning multiple tasks results in an implicit data augmentation which reduces the risk of overfitting. In the same way it can be advantageous to learn a task in multiple domains to generalize well. Also, learning multiple tasks enables models to focus on relevant features, ignoring others. Some features might be easier to learn for one task than another. Furthermore, multi-task learning allows *eavesdropping* on other tasks to learn useful features, [16].

Multi-task and multi-domain learning is often implemented by parameter sharing. Parameter sharing is the practice of tying some or all parameters of two or more models together. There are two forms of parameter sharing: hard parameter sharing and soft parameter sharing. In hard parameter sharing weights of shared layers are set to the exact same values. In other words, only one model with output layers for every task or domain is used, separating task-specific layers and shared layers, [16], [19]. It can be shown to reduce the risk of overfitting by N, where N is the number of tasks, [18]. On the other hand, soft parameter sharing introduces a distance metric between parameters, [7]. Each

task is solved by a separate model. The weights are regularized by a distance metric. This enforces shared weights to be similar but not necessarily the same.

An important question in multi-task learning is which tasks to add to improve generalization. The most obvious auxiliary task is a closely related task, [16]. Another possibility is an adversarial task. An adversarial tasks is a task that should not be achieved. By reversing the gradient of the adversarial loss performance on the main objective can be improved, [20]. Also, an attention mechanism can be used to focus on informative features, [21]. A recent addition to the array of auxiliary tasks is reproduction of input from low-dimensional feature space via an auto-encoder, [22]. The auto-encoder enforces a representation that is complete, i. e. the input can be reconstructed from it. In the three remaining sections these ideas are explained in greater detail.

3.3 Domain Randomization

Many problems in machine learning require large amounts of training data. Simulations can be used to lower the cost of collecting data. They are also often used to prototype before working on a real system to prevent damage.

While simulations are invaluable they have limits when it comes to transferability. There are two main differences between simulations and reality hampering successful transfer between them: simulated physics and image rendering, [4]. First, simulated physics and real system differ. If simulation parameters are not tuned the simulated system does not capture reality. System identification can be used to optimize them, which can be time-consuming and error-prone. Furthermore, many simulators have unmodeled effects such as non-rigid bodies, gear backlash, and wear. Second, simulated sensors produce readings that differ from their real-world equivalents. Images rendered in simulation are different in terms of appearance and noise. High-fidelity image renderers come with significant computational costs, offsetting the low-cost data collection in simulation. Low-fidelity image renderers are faster but further increase what is called the *reality gap*. All of these differences result in machine learning models trained in simulation to be unable to transfer to reality.

Domain randomization offers a way to increase generalization to reality from simulation. Instead of trying to match the real world in simulation as closely as possible simulation parameters are randomized to vary physics and appearance. In effect, a model is trained on a range of simulations. If the variability is high, models are able to generalize well, [4]. In general domain randomization is applicable to imitation learning and reinforcement learning.

Domain randomization can be seen as a form of meta-learning. During training the network is presented with new data and forced to adapt. This bears similarity to algorithms described in Section 3.5.3.

Uniform Domain Randomization

The most basic form of domain randomization is uniform domain randomization (UDR). UDR randomizes simulation parameters uniformly in a given range. It can by applied universally but requires expert knowledge to select parameter subject to randomization and to choose their ranges.

Tobin et al. use domain randomization for robotic object localization and grasping, [4]. The objective is to map a camera image to object position in Cartesian coordinates. They use a VGG-16 network trained using reinforcement learning. The following parameters are randomized.

- Number, shapes, positions and textures of objects
- Background textures
- Position, orientation, and field of view of camera
- Lighting
- Noise added to images

For each training episode new parameters are sampled from a closed interval using a uniform distribution.

Andrychowicz et al. extend domain randomizatin to physics parameters in addition to appearance, [23]. The task at hand is manipulation of a cube by a robotic hand. They randomize the following parameters.

- Dimensions and masses of links
- Damping coefficients of joints
- Actuator controller gains
- Surface friction coefficients
- Gravity vector

Also, a model for motor backlash and action delays is employed. However, many effects remain unmodeled in simulation. Output noise is added to take any unmodeled behavior into account.

Structured Domain Randomization

Structured domain randomization (SDR) acknowledges the structure inherent in a scene during randomization using hierarchy. It has three layers: global parameters, context splines, and objects. Prakash et al. use SDR for bounding box detection of cars, but it can be adapted to other use cases , [24].

To generate a new scene in simulation a scenario s is selected at random from a range of predefined scenarios. For bounding box detection of cars scenarios are different road types and settings. Next, global parameters g such number of objects, lighting and weather

conditions are sampled. Global parameters also determine splines c that model the course of the road and surroundings. Splines are given a random color and texture such as grass or asphalt color. Additional randomizations such as potholes or oil spills are superimposed. In the last step objects o are placed on the splines. Figure 3.3.1 shows the hierarchical generation of a scene using SDR.

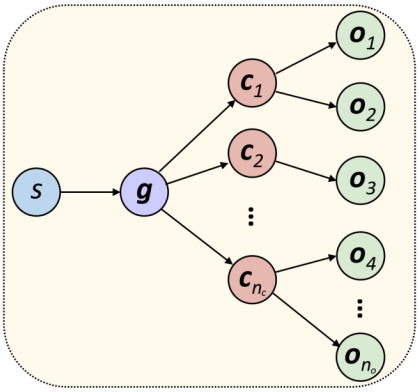


Figure 3.3.1: Structured domain randomization. The structure is implemented using a graph of conditional probabilities.

Source: Prakash et al. [24]

SDR poses constraints on the randomization process. These constrains can be formalized as conditional probabilities. Given a random scene $s \sim p(s)$, global parameters are sampled using $g \sim p(g \mid s)$. Having determined the global parameters g the splines are sampled using $c \sim p(c \mid g)$. Continuing this patterns leads to the following expression.

$$p(I, s, g, c, o) = p(I \mid s, g, c, o) \prod_{j=1}^{n_o} p(o_j \mid c_i) \prod_{i=1}^{n_c} p(c_i \mid g) p(g \mid s) p(s) \tag{3.3.1}$$

The probability of generating an image I depends on scenario s, global parameters g, splines c, and objects o. Each item's probability in the dependency chain depends on that of its predecessors. This encodes the structure of the scene. SDR generates randomized scenes with a plausible structure. In contrast, UDR samples each random variable independently and hence does not have conditional probabilities between simulation parameters. Adding structure to the scene can be more data efficient, [24].

3.3.1 Guided Domain Randomization

Guided domain randomization (GDR) seek to actively find a randomization scheme that yield improved generalization.

SimOpt

The aim of SimOpt is to narrow down the distribution of parameter randomization required for closing the reality gap, [25]. In UDR simulation parameters are sampled from a uniform

distribution in fixed intervals. These intervals need to be tuned by humans, which requires expert knowledge. If the interval is too wide training becomes inefficient since a lot of data contains irrelevant or redundant information. On the other hand, if the interval is too narrow no generalization is achieved. Ideally, the distribution of simulation parameters would be just wide enough to bridge the gap to reality.

The SimOpt framework tries to improve the sampling procedure. It starts with an initial distribution of simulation parameters p_ϕ and improves on them iteratively. Let π_{θ,p_ϕ} be an agent's policy that is trained in a simulation using parameters sampled as $\xi \sim p_\phi$. The agent can be trained using standard deep reinforcement learning algorithms by maximizing the expected cumulative reward as follows.

$$\max_\theta \mathbb{E}_{\xi\sim p_\phi}\left[\mathbb{E}_{\pi_\theta}\left[R(\tau)\right]\right] \tag{3.3.2}$$

Given a discrepancy metric D the objective of SimOpt is to minimize the distance between observed trajectories in simulation τ_ξ^{ob} and reality τ_{real}^{ob}. This can be achieved by minimizing the following objective.

$$\min_\phi \mathbb{E}_{\xi\sim p_\phi}\left[\mathbb{E}_{\pi_{\theta,p_\phi}}\left[D(\tau_\xi^{ob}, \tau_{real}^{ob})\right]\right] \tag{3.3.3}$$

Computing the discrepancy without further adjustments requires a roll-out of the policy in reality. This can be expensive, time-consuming or even infeasible. However, the inputs of the policy and observations to compute $D(\tau_\xi^{ob}, \tau_{real}^{ob})$ do not need to be the same. Thus, Chebotar et al. propose to reuse previous roll-outs as long as the Kullback-Leibler (KL) divergence between the parameter distributions used is sufficiently small. They use relative entropy policy search to minimize Equation (3.3.3), a gradient-free optimization algorithm suitable for working with a non-differentiable simulator. Figure 3.3.2 illustrates the training procedure of SimOpt.

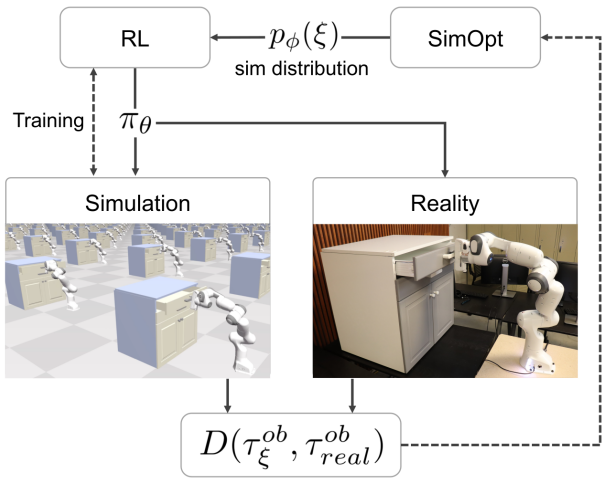


Figure 3.3.2: Overview of SimOpt framework. Agents are trained in simulation. The discrepancy between roll-outs in simulation and reality is used to optimize simulation parameter sampling.

Source: Chebotar et al. [25]

Learning to Simulate

Learning to simulate (L2S) is an algorithm that tries to find domain randomization parameters that generalize well to the target domain, [26]. The simulation configuration that works best is not necessarily the one that resembles reality as closely as possible. L2S has several advantages to UDR. The effect of UDR depends on domain knowledge for defining and tuning intervals to sample from. By optimizing the way simulation parameters are sampled the amount of human labor required is reduced. Furthermore, smaller datasets can be used to achieve similar or better performance compared to UDR. Therefore, training times are reduced.

In supervised learning settings the objective is to find a function f_θ that minimizes the loss, given training data sampled from a distribution $p(x, y)$. The real distribution of data $p(x, y)$ is unknown and only a sample of size N is known. Simulated data is sampled from a distribution $q(x, y \mid \psi)$, where ψ are parameters of the simulation regarding rendering and physics. While data from reality is limited, data in simulation is abundant.

The goal is to find simulation parameters ψ such that the loss L for f_θ given validation data D_{val} from target domain is minimized. L2S formulates this as a bilevel optimization problem, optimizing simulation parameters ψ and model parameters θ jointly.

$$\psi^* = \arg\min_{\psi} \sum_{(x,y)\in D_{val}} L(y, f_\theta(x; \theta^*(\psi))) \tag{3.3.4}$$

$$\theta^* = \arg\min_{\theta} \sum_{(x,y)\in D_{q(x,y|\psi)}} L(y, f_\theta(x, \theta)) \tag{3.3.5}$$

The outer optimization Equation (3.3.4) finds simulations parameters ψ that generate data in an optimal manner. The loss is computed on validation data using optimal model parameters ψ^*. The inner optimization Equation (3.3.5) optimizes models parameters θ to solve the actual task on data sampled from $q(x, y; \psi)$.

Model parameters can be optimized straightforwardly using gradient descent. However, Equation (3.3.5) depends explicitly on simulation parameters. To solve the problem using gradient-based methods the inner optimization problem needs to be smooth, twice differentiable, and have an invertible Hessian, [26]. In general, these constraints are not satisfied. Hence, Ruiz et al. propose to use policy gradients to optimize ψ, [26].

Let π be a policy for generating simulation parameters $\psi \sim \pi$. The simulation acts as a generative model $G(\psi)$ that yields data pairs (x, y) conditioned on ψ. During optimization models are trained of generated datasets $D_q(x, y \mid \psi)$ until convergence or for a fixed number of iterations using gradient descent. This solves the inner bilevel optimization problem. For the outer optimization task the data generation policy π is updated using the negative loss on validation data, given a trained model, as reward. L2S bears similarities with optimization-based meta-learning (cf. Section 3.5.3). The training procedure is visualized in Figure 3.3.3. Note that unlike UDR L2S is not suitable for zero-shot learning because it requires a labeled validation set.

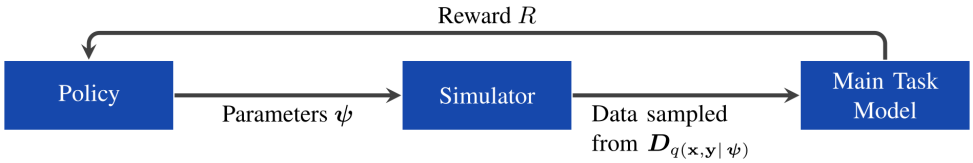


Figure 3.3.3: Schematic of the learning to simulate procedure. Synthetic data $D_{q(x,y|\psi)}$ is generate in simulation parameterized by ϕ. A model is trained to solve the main task on this data. The model performance on a validation set is used to train a policy to find better simulation parameters.

Source: Ruiz et al. [26]

Active Domain Randomization

Active domain randomization (ADR) deems some randomizations more informative than others and focuses on those during training, [27]. Informative, in this context, refers to being more efficient at achieving generalization across domains. Instead of sampling simulation parameters uniformly, ADR introduces a sampling policy to find parameters that generate more simulation instances. The sampling policy is implemented using Stein variational policy gradient (SVPG) to learn an ensemble of policies or particles. The randomization space is regarded as a search space for optimization. Mehta et al. use deep reinforcement learning to solve the main task as well as the search for randomization parameters. Each particle's state is a set of randomization parameters. Since the agent trained on simulation instances changes over time, the optimization of simulation parameters is non-stationary.

The ADR algorithm works as follows. At the beginning agent policy and SVPG particles are initialized randomly. Multiple simulation instances E_i are generated using SVPG particles as simulation parameters. The current agent policy π_θ is rolled out in these environments. The resulting trajectories τ_i are compared to a roll-out of the agent in a reference environment E_{ref} by the discriminator D_ψ. The discriminator is trained for binary classification between randomized environment and reference environment. The reference environment is not used for training the agent directly; the agent is trained on randomized simulations only.

The discriminator reward r_D is given by

$$r_D = \log D_\psi(y \mid \tau_i \sim \pi_\theta(.; E_i)). \tag{3.3.6}$$

The discriminator is rewarded for exploring regions of the search space that evoke different behavior in the agent than the reference environment. Thus, the discriminator iteratively finds harder environments while the agent becomes better at adapting to diverse settings. Training two competing networks that each make each others objective harder is reminiscent of GANs. Figure 3.3.4 illustrates the ADR training loop.

ADR and L2S both optimize simulation parameters to achieve generalization. While L2S is concerned with the performance of a model on a small labeled validation set from target domain ADR compares the roll-out of a trained model between simulation and reality.

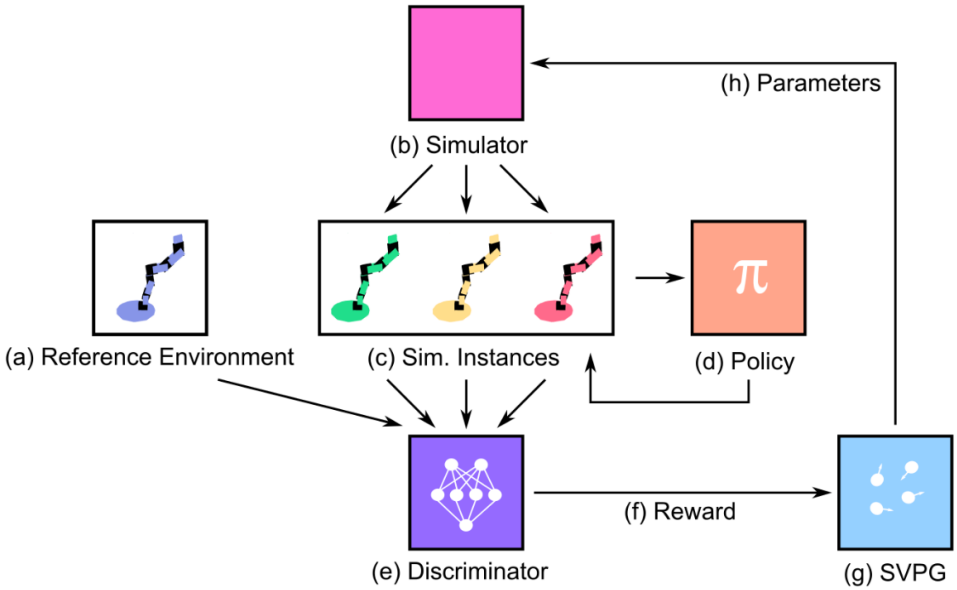


Figure 3.3.4: Active domain randomization. Various simulation instances (c) are generated by a simulator (b) using parameters (h) provided by SVPG particles (g). For each simulation instance the agent policy(d) is rolled out to find those instance that are difficult. This is done by comparing them to a reference environment (a) by the discriminator (e) which learns a reward (f). The reward is used to train SVPG particles to find parameters that are challenging for the current agent, closing the loop.

Source: Mehta et al., [27]

Randomized to Canonical Adaptation Network

Randomized to canonical adaptation networks (RCANs) learn to map simulated and real images to a canonical representation without using real world data, [28]. They use randomized simulations to translate from real images to shared domain. Thus, they are a hybrid of domain adaptation and domain randomization. They fall into the category of input-level domain adaptation as describes in Section 3.4.2, operating on raw input images. The RCAN framework introduces a third domain along with source and target domain: the canonical domain. The canonical domain is an abstract representation of the scene, lacking real world detail and noise. It contains the essence of the scene in uniform colors and constant lighting. No randomizations are present in the canonical simulation. Learning with domain randomization can sometimes slow training down or destabilize it, [28]. By reducing the visual complexity, learning from a simplified canonical representation is easier.

The translation from randomized simulation and reality to canonical representation is done using an image-conditioned generative adversarial network. The generative part G maps an image from any domain to canonical. During training, the generator is presented with tuples (x_s, x_c, m_c, d_c), where x_s is a source image from the randomized domain, x_c is an image from canonical domain matching the source image, m_c is a segmentation mask, and d_c is a depth image. Randomized and canonical image need to be paired, i. e. each

randomized image needs to have a canonical counterpart. This is achieved by collecting data in each simulated scenario twice, once with randomizations and once without. After the GAN is trained, the main task can be solved using reinforcement learning or imitation learning. Figure 3.3.5 outlines how images are mapped to a common domain using the generator before training an agent.

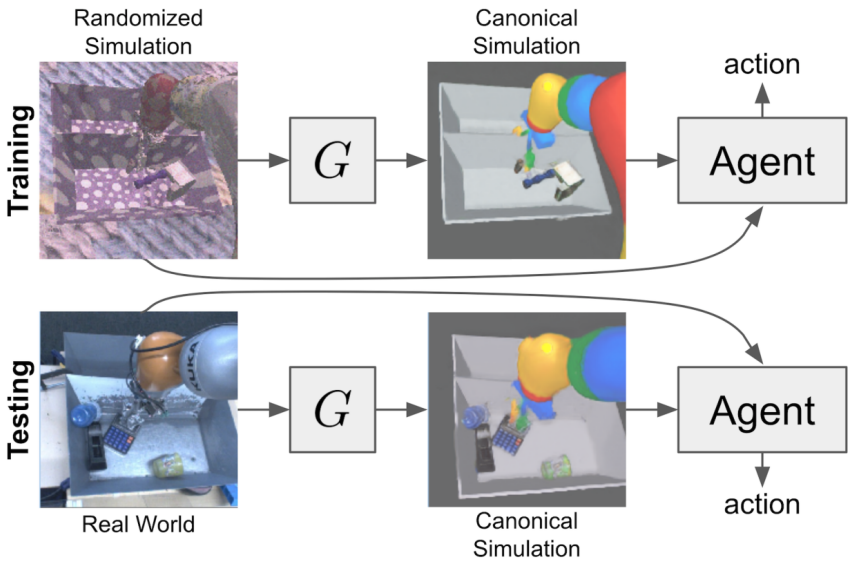


Figure 3.3.5: Randomized to canonical adaptation network. Images from simulation and reality are mapped to a canonical domain in which the agent is trained.
Source: James et al. [28]

The objective of RCAN is to learn a generator $G(x_s) \rightarrow \{x_a, m_a, d_a\}$, where x_a is the generated canonical image, m_a the generated segmentation mask, and d_a the generated depth image. The generator is trained by minimizing the difference between generated and canonical image while also making sure the semantic and depth information is similar. This is achieved using a combination of three loss terms: $L_x(x_c, G_x(x_s))$ for visual equality, $L_m(m_c, G_m(x_s))$, for semantic equality, and $L_d(d_c, G_d(x_s)$ for depth equality. The weighted sum of equality losses is given by

$$L_{eq}(G) = \mathbb{E}_{x_s,x_c,m_c,d_c}[\lambda_x L_x(x_c, G_x(x_s)) + \lambda_m L_m(m_c, G_m(x_s)) + \lambda_d L_d(d_c, G_d(x_s)], \quad (3.3.7)$$

where λ_x, λ_m, and λ_d are weights. Individual loss terms are computed using L1 or L2 loss between pixel values. At the same time, the discriminator is trained to distinguish between generator output and actual canonical image. The generator is trained to minimize the standard GAN loss (cf. Equation (2.3.11)) and equality loss jointly, i. e.

$$\min_G \max_D L_{GAN}(G, D) + L_{eq}(G). \quad (3.3.8)$$

The main task is learned in canonical domain. No real data is used during training the generator. RCAN relies on domain randomization to generalize translating real images to canonical domain. If this fails, any behavior learned from canonical images is unlikely to succeed.

DeceptionNet

DeceptionNet uses a neural network for pixel-level randomization, [22]. While a task network is trained to solve the task at hand, a deception network is trained hinder the task network by perturbations. The task network $T(x;\theta_T) \to \hat{y}$ is a CNN that maps input image x to prediction $\hat{y}$. The deception network $D(x_s,\theta_D) \to x_d$ is an auto-encoder that takes an input from source domain x_s, encodes it in a low-dimensional latent vector z, and decodes it to a deception image x_d. In contrast to regular auto-encoders, [22] propose multiple decoding modules M_i. The deception image x_d is the weighted sum of decoder module outputs, i. e. $x_d = \sum_i w_i M_i(z)$, where w_i are masks. A decoding module can change some characteristics of the input image, for example background, light, distortion, or noise.

Training is done using a min-max approach similar to GANs in two alternating steps. The task loss is minimized while the deception loss is maximized. Formally, the objective is given by

$$\min_{\theta_T} \max_{\theta_D} L_T(T(D(x;\theta_D),y;\theta_T)). \tag{3.3.9}$$

In the first step weights of the task network θ_T are frozen—i. e. they are not updated—by setting their gradients to zero. An input image is transformed by the deception network and fed through the task network. The task loss is computed and its gradients reversed (cf. Section 3.4.1). Thus, the deception network is trained to disturb the task network. In the second step weights of the deception network θ_D are frozen and the task network is trained using deceptive images. Hence, it becomes robust to domain changes. The training procedure is illustrated in Figure 3.3.6.

No real world images are required for training DeceptionNet. DeceptionNet is similar to ADR and L2S but unlike them DeceptionNet operates on directly pixels instead of optimizing simulation parameters.

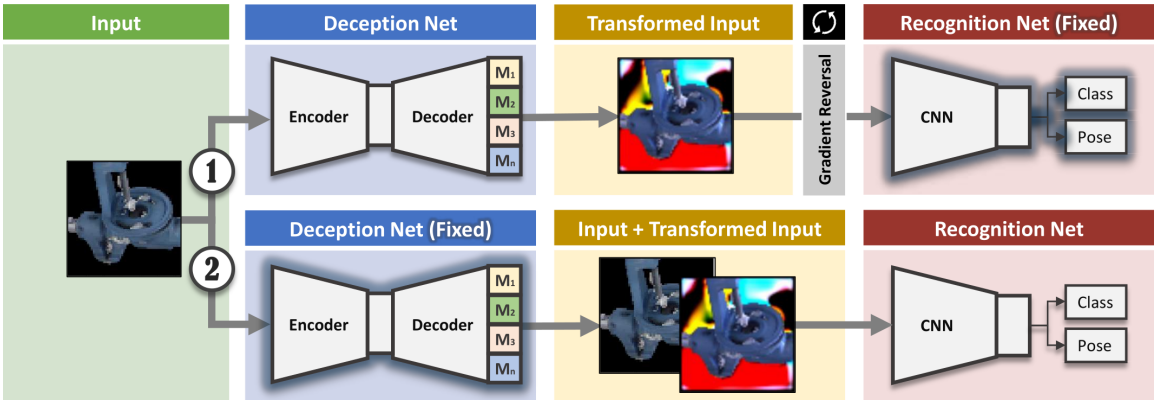


Figure 3.3.6: DeceptionNet architecture and training. Input images are transformed by a deception network. The task network is trained on transformed images. Training alternates between deception and task network.

Source: Zakharov et al. [22]

3.4 Domain Adaptation

Domain adaptation is a field in machine learning that attempts to transfer models form one domain to another. This can be achieved by learning a transformation between them or by finding domain-invariant feature representations. Two different kinds of domain adaptation are presented in the following sections: Feature-level and input-level domain adaptation. The former operates on feature representations in different domains while the latter is concerned with transforming inputs, [29].

3.4.1 Feature-level

Feature-level domain adaptation aims to find feature representation that allow a network to be trained in source domain and deployed in target domain. Thus, feature-level domain adaptation can be considered to be a form of feature representation transfer learning. Three approaches are introduced: Domain classification with gradient reversal, deep domain confusion (DDC), and weight regularization across domains. The latter is a hybrid approach to transfer learning, combining feature representation transfer with parameter transfer. All of them introduce additional loss terms, making them multi-task learning algorithms.

Domain Classification with Gradient Reversal

Domain classification with gradient reversal is a method to produce domain-invariant features, [20]. It uses labeled data from source domain and unlabeled data from target domain.

Domain classification with gradient reversal uses one network with two outputs. The network predicts the output class or variable y for classification and regression, respectively. It consists of three parts: feature extractor $G_f(\theta_f)$, predictor $G_y(\theta_y)$, and eponymous domain classifier $G_d(\theta_d)$, each with their own set of parameters θ. Typically, the feature extractor is a CNN that maps an input x to a flattened feature tensor f. A series of fully connected layers (FCLs) maps features to y, an output class vector for classification or an output variable for regression problems. Parallel to that a second stack of FCLs maps the same feature tensor to a domain label d. The domain classification network architecture is detailed in Figure 3.4.1.

During training, data from source and target domain are mixed and presented to the network. Source domain data is assumed to be labeled completely and the objective loss L_y can readily be computed. On the other hand, only few or even none of the target domain examples have ground truth labels available. For those no objective loss can be computed; it is set to zero. The objective loss, if it is available, can be minimized using standard optimization techniques. In addition to the objective loss a domain classification loss L_d is computed from the domain label. The domain classification loss should be maximized since the network should not be able to tell source and target domain apart. Equivalently, the negative classification loss is to be minimized. This motivates the

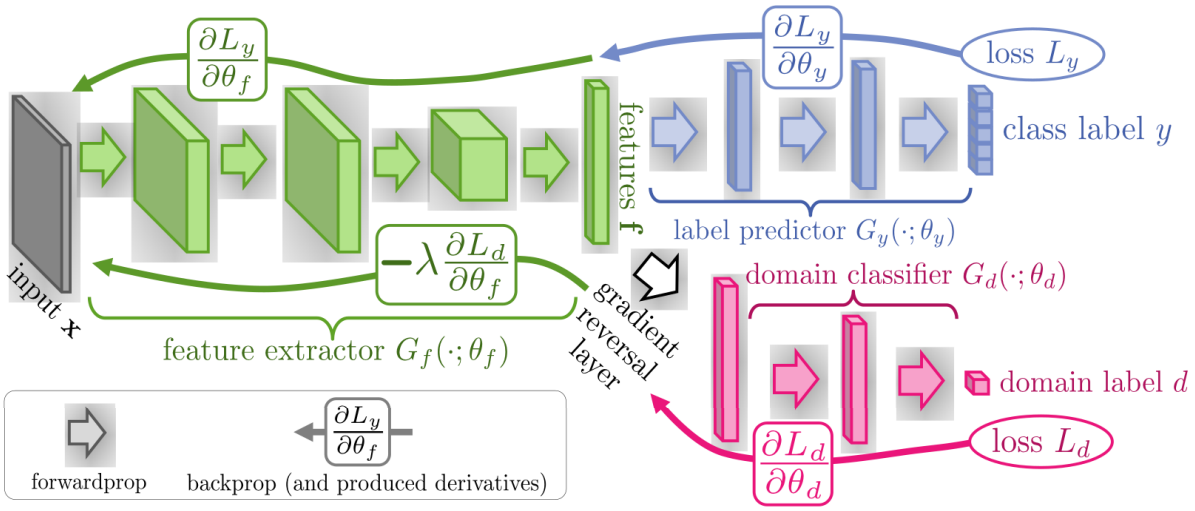


Figure 3.4.1: Domain classification with gradient reversal. Feature extractor network $G_f(\theta_f)$ extracts features f from input image x. Class label y is predicted from features by the predictor network $G_y(\theta_y)$. In parallel, a domain label d is predicted by the domain classifier network $G_d(\theta_d$. The reversed domain classification gradient is used to attain domain-invariant features.

Source: Ganin et al. [20]

introduction of the gradient reversal layer. The gradient reversal layer, located between feature tensor and domain classifier, multiplies the gradient of all subsequent layers by -1 during backpropagation. Formally, the domain classification loss is given by the following equation.

$$L = L_y(y_i, G_y(G_f(x_i;\theta_f);\theta_y)) - \lambda L_d(y_d, G_d(G_f(x_i;\theta_f);\theta_d)) \tag{3.4.1}$$

A weight hyperparameter λ can be used to balance the two terms. Parameter updates using gradient descent with step size μ take the following form.

$$\theta_f \leftarrow \theta_f - \mu\left(\frac{\partial L_y}{\partial \theta_f} - \lambda\frac{\partial L_d}{\partial \theta_f}\right) \tag{3.4.2}$$

$$\theta_y \leftarrow \theta_y - \mu\frac{\partial L_y}{\partial \theta_y} \tag{3.4.3}$$

$$\theta_d \leftarrow \theta_d - \mu\frac{\partial L_d}{\partial \theta_d} \tag{3.4.4}$$

Parameters of predictor and domain classifier are updated using their respective loss. The shared parameters of the feature extractor are updated using both losses. Thus, discriminative, domain-invariant features are produced. Minimizing two losses jointly makes domain classification with gradient reversal a multi-task learning algorithm.

Adding a domain classification loss can be viewed as a form of regularization. This regularization favors networks that can not distinguish between domains.

Deep Domain Confusion

An approach similar to adding a domain classification loss was proposed by Tzeng et al., [30]. Instead classifying the domain, a domain confusion loss is added. The domain confusion loss encourages features to be domain-invariant by minimizing a metric between source and target domain features. DDC can be applied to semi-supervised and unsupervised domain adaptation.

The domain confusion loss makes use of maximum mean discrepancy (MMD). MMD is a metric defined on two distributions. It operates on a representation of distributions given by a mapping $\phi : X \to H$, where H is commonly referred to as reproducing kernel Hilbert space (RKHS). When applying MMD for domain adaptation of neural networks the distributions in question are source and target domain data X_S and X_T, respectively. The representation is given by a forward pass through the network. The MMD is then given by the difference of means of two distributions.

$$\mathrm{MMD}(X_S, X_T) = \left\| \frac{1}{|X_S|} \sum_{x_S \in X_S} \phi(x_S) - \frac{1}{|X_T|} \sum_{x_T \in X_T} \phi(x_T) \right\| \tag{3.4.5}$$

Given a task loss $L_y(y, X_L)$ for labeled data X_L (from source and possibly target domain) the objective of DDC is stated as follows.

$$L = L_y(y, X_L) + \lambda \mathrm{MMD}^2(X_S, X_T) \tag{3.4.6}$$

By minimizing task loss and MMD between features jointly both discriminative and domain-invariant features are found. The domain adaptation layer that computes the domain confusion loss is typically placed after a fully-connected layer close to the output. Figure 3.4.2 shows the network architecture used for DDC.

Weight Regularization Across Domains

Rozantsev et al. combine MMD with weight regularization, [7]. While acknowledging that maximizing feature invariance can be used for domain adaptation, the authors state that forcing weights to be shared across domains can be adverse to generalization. Most approaches for domain adaptation described in this chapter use one set of parameters for all domains and jointly minimize objective loss along with some form of domain invariance loss. In contrast, weight regularization allows model parameters to diverge within bounds for different domains.

Weight regularization uses a two-stream architecture. The source stream is pre-trained on data from source domain only. It serves as a starting point for weights. The second stream is trained on unlabeled target data. While the source stream can be trained using a classification loss, e. g. softmax, in general no such loss is available for the target stream since no labels are provided. The target stream is trained using a domain discrepancy loss such as MMD. Additionally, weights between corresponding layers of source and target stream are regularized. Figure 3.4.3 illustrates the two-stream architecture using various loss terms.

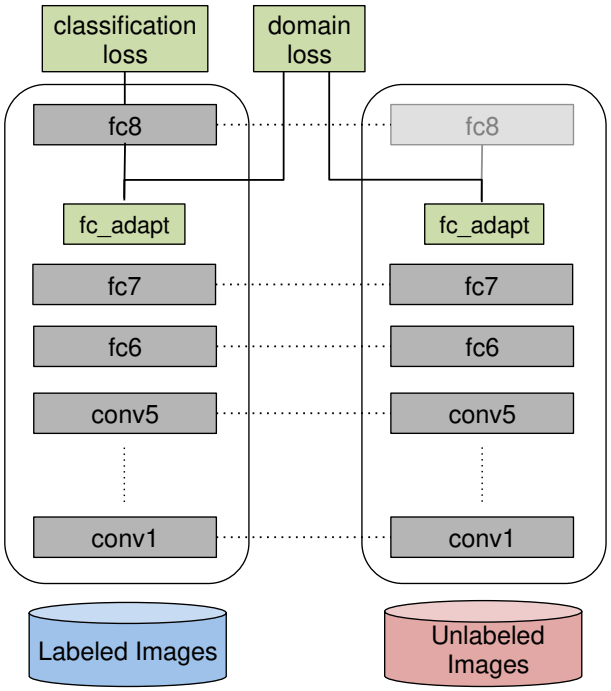


Figure 3.4.2: Deep domain confusion architecture. Two losses are minimized during training: classification loss to solve the main task and domain confusion loss to achieve domain invariance. The domain confusion loss is computed between abstract, high-dimensional feature vectors near the final layer. The weights of all layers are shared between source and target domain as indicated by dotted lines.

Source: Tzeng et al. [30]

The combined loss of both streams is given by

$$L(\theta^s, \theta^t) = L_s(\theta^s) + L_t(\theta^t) + \lambda_w L_w(\theta^s, \theta^t) + \lambda_d L_d(\theta^s, \theta^t) \tag{3.4.7}$$

Source and target classification loss $L_s(\theta^s)$ and $L_t(\theta^t)$ depend only on their respective parameters. If target domain data is fully unlabeled $L_t(\theta^t)$ is zero. The MMD loss encourages feature distributions between source and target domain to be aligned. It is defined in Equation (3.4.5). Weight regularization on the other hand ensures model parameters of source and target stream do not diverge too far. This allows differences in parameters to arise but at the same time ties them together. Individual loss terms are weighted with hyperparameters λ_w and λ_d.

The weight regularization loss is defined as the L2-norm between two sets of parameters.

$$L_w(\theta^s, \theta^t) = \sum_i \left\| \theta_i^s - \theta_i^t \right\| \tag{3.4.8}$$

Weight regularization between source and target network is similar to weight decay described in Section 4.4.3. The difference is that while weight decay draws weights back to the origin, weight decay for domain adaptation keeps weights in a hypersphere around source stream parameters. In fact, setting $\theta_s = 0$ leads to the weight decay Equation (4.4.1).

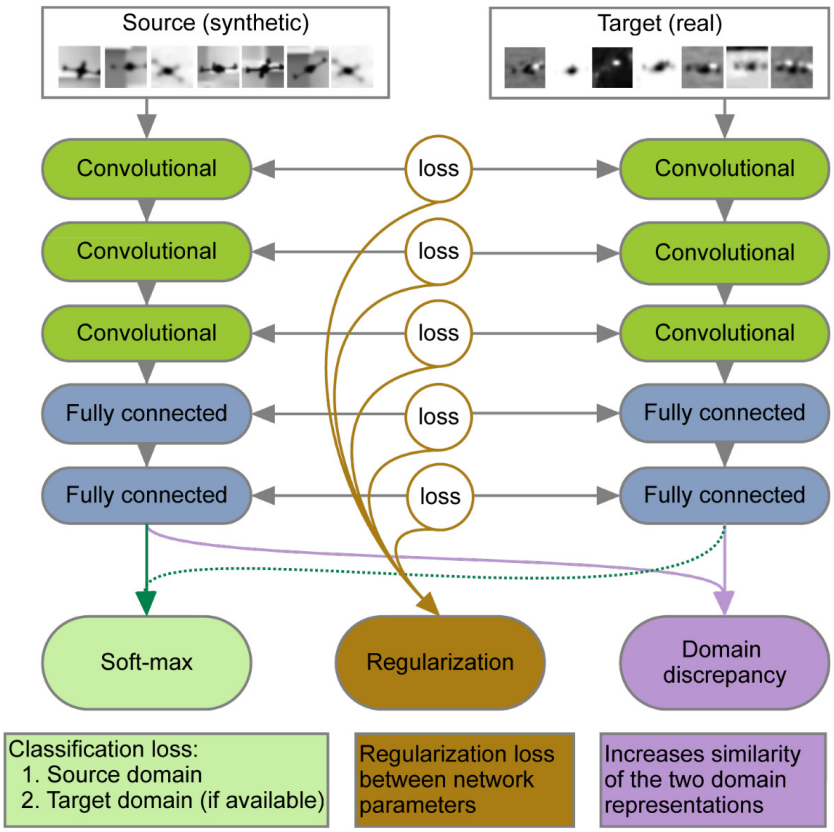


Figure 3.4.3: Weight regularization across domains. Two networks with separate weights are trained on data from their respective domain. A domain discrepancy loss encourages domain-invariant features. At the same time, weights are regularized to prevent overfitting to a particular domain.

Source: [7]

To explicitly model a transformation between source and target stream parameters a linear transformation can be added in the following way,

$$L_w(\theta^s, \theta^t) = \sum_i \left\| a_i \theta_i^s + b_i - \theta_i^t \right\|, \tag{3.4.9}$$

where a_i and b_i are learnable parameters that describe the domain shift. Other transformations are conceivable.

3.4.2 Input-level

Input-level domain adaptation aims to find a mapping from one domain to another. It tries to make data form source domain appear as if it were sampled from target domain. Many recent approaches use GANs to model this mapping. An advantage of input-level domain adaptation is interpretability: Transformed input images are easier to interpret than feature vectors. This comes at the disadvantage of increased computational cost and

potential instability during training. The computation cost poses further constrains on image size.

Pixel-Level Domain Adaptation

Pixel-level domain adaptation (PixelDA) is an input-level approach to domain adaptation, [29]. Given labeled images in source domain and unlabeled images in target domain it seeks to find a mapping from one domain to the other using a GAN. Once this mapping is found learning the main task is straightforward.

PixelDA consists of three networks: A generator $G(x_s, z; \theta_G) \rightarrow x_f$, a discriminator $D(x; \theta_D) \rightarrow \{\text{real}, \text{fake}\}$, and a task network $T(x_f; \theta_T) \rightarrow \hat{y}$. The Generator maps an image from source domain to target domain, given a noise vector z, creating fake images x_f. On the other hand, the discriminator is trained to distinguish between real and fake target domain images x_t and x_f. The task network is trained on source domain and fake target domain images x_s and x_f. No target domain images are used for training the task network. Figure 3.4.4 visualizes the high-level architecture of PixelDA and how the sub-networks are related to each other.

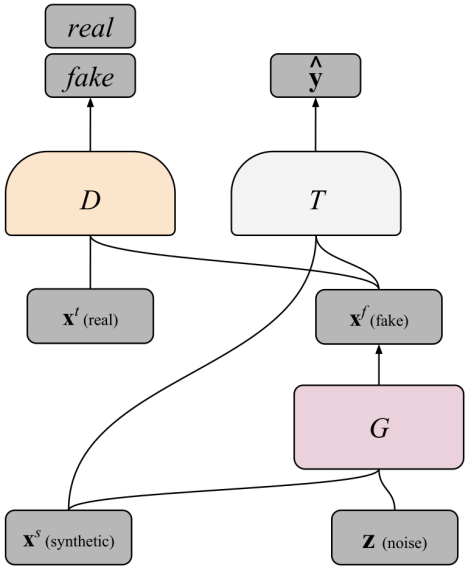


Figure 3.4.4: Relationship between generator G, discriminator D, and task network T for PixelDA. Generator and discriminator are trained so that the generator outputs source domain images in the style of the target domain. The target network is trained on these images in conjunction with source domain images.

Source: Bousmalis et al. [29]

Generator and discriminator follow the standard GAN design of Equation (2.3.11), with the addition of a task loss L_T and content-similarity loss L_C.

$$\min_{\theta_G, \theta_T} \max_{\theta_D} L_{GAN}(G, D) + \lambda_T L_T(G, T) + \lambda_C L_C(G) \tag{3.4.10}$$

The task loss can be any loss that is suitable for the task, for example categorical cross-entropy (cf. Section 2.2). Bousmalis et al. propose to use masked pairwise mean squared error (MPMSE) for content-similarity loss, defined as

$$L_C(G) = \mathbb{E}_{x_s,z}\left[\frac{1}{k}\sum(d \circ m)^2 - \frac{1}{k^2}\left(\sum d \circ m\right)^2\right], \tag{3.4.11}$$

where $d = x_s - G(x_s, z)$ is the pairwise difference between source image x_s and generator output $x_f = G(x_s, z)$, m is a binary mask, and $\circ$ is the Hadamard or element-wise matrix product, [29]. The binary mask m is used to select the foreground. MPMSE penalizes differences between pairs of pixels. It encourages the generator to reproduce the overall shape of objects in the foreground but allows small variations.

Cycle-Consistent Domain Adaptation

Cycle-consistent adversarial domain adaptation (CyCADA) extends PixelDA by introducing cycle-consistency, [6]. The cycle-consistency or reconstruction loss is meant to enforce semantic consistency of generated images. This is supposed to help generate more realistic target domain images and reduce training instability. CyCADA also combines image-level adversarial learning with feature-level adversarial learning inspired by Ganin et al., [20]. Again, the aim is to find a transformation from one domain to another to generate suitable training data to solve the main task.

Like PixelDA, CyCADA consist of three networks: generator G, discriminator D, and task network T. As usual, the generator maps a source domain image to target domain image, i. e. $G_{S\to T}(x_S) = x_T$. In CyCADA, the same generator is also used to map the generated target domain image back to source domain, i. e. $G_{T\to S}(x_T) = G_{T\to S}(G_{S\to T}(x_S)) = x_S$. In other words, the reconstruction loss demands that $G_{T\to S}(G_{S\to T}) = \mathbb{I}$, where $\mathbb{I}$ is identity. Also, the generator should be able to map a target domain image to a source domain and image and back, i. e. $G_{S\to T}(G_{T\to S}(x_T)) = x_T$. The reconstruction loss uses the L1 norm to achieve this.

$$\begin{aligned} L_{cyc} =& \mathbb{E}[\|G_{T\to S}(G_{S\to T}(x_S)) - x_S\|_1] \\ &+ \mathbb{E}[\|G_{S\to T}(G_{T\to S}(x_T)) - x_T\|_1] \end{aligned} \tag{3.4.12}$$

In addition to cycle-consistency of images CyCADA enforces a semantic consistency. Semantic consistency is achieved if source domain images x_s and generated target domain images $G_{S\to T}(x_s)$ are classified in the same way by the source task network f_S. Given an arbitrary task loss L_{task} and source domain labels y_S the semantic consistency loss is given by

$$\begin{aligned} L_{sem} =& L_{task}(y_S, f_S(x_S)) \\ &+ L_{task}(y_S, f_S(G_{S\to T}(x_S))). \end{aligned} \tag{3.4.13}$$

The intention of semantic consistency is that the transformation applied by $G_{S\to T}$ should preserve information about the content of the image.

Semantic consistency can be thought of as task network output consistency. Another form of consistency introduced by CyCADA is feature consistency. Feature consistency is inspired by feature-level domain adaptation techniques. It seeks to maintain similar feature representations for different domains and is implemented by a GAN loss using an additional feature-level classifier D_{feat}.

$$L_{feat} = \log D_{feat}(f_T(G_{S\to T}(x_s))) + \log(1 - D_{feat}(f_T(x_t))) \tag{3.4.14}$$

The feature-level discriminator is trained to tell feature vector of task network f_T applied to real and generated target domain images x_T and $G_{S\to T}(x_s)$, respectively, apart.

The final CyCADA loss is given by the sum of all losses explained above plus the standard GAN loss.

$$L_{CyCADA} = L_{task} + L_{GAN} + L_{cyc} + L_{sem} + L_{feat} \tag{3.4.15}$$

Figure 3.4.5 visualized how the losses are related to individual components of CyCADA.

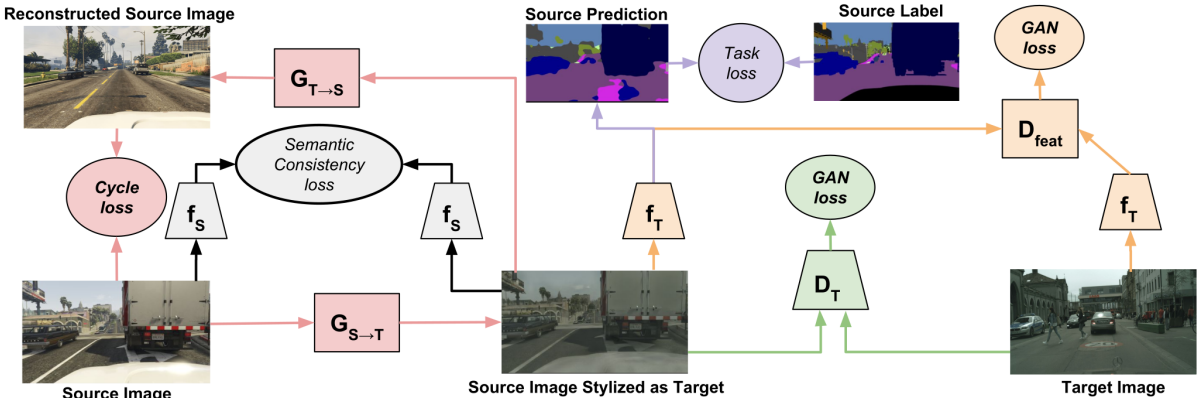


Figure 3.4.5: Overview of cycle-consistent adversarial domain adaptation. The generator G is trained to generate target domain images while the discriminator D is trained to differentiable between generated and real target domain images (shown in green). In addition, the generator is trained to reconstruct source domain images from target domain images generated by itself using a cycle-consistency loss L_{cyc} shown in red. The semantic consistency loss L_{sem} is shown is gray. Shown in yellow is an additional feature-level GAN loss L_{feat}. The task network f_T is trained on generated target domain images to perform the main task, in this case semantic segmentation (shown in purple).

Source: Hoffmann et al. [6]

3.5 Meta-Learning

Many deep learning algorithms require a large amount of data to perform well. In contrast, humans are able to learn new task after a few attempts by leveraging knowledge from previous tasks. Such behavior is desirable in neural networks. A learning setting in which a network is required to learn a new task few examples is called few-shot learning. When no data for the target task is given the setting is called zero-shot learning.

Meta-learning attempts to optimize the learning process itself. It is also called "learning to learn". There are three different types of meta-learning: metric-based, model-based, and optimization-based.

3.5.1 Metric-Based

Metric-based meta-learning classifies new classes based on a metric to known examples of other classes. It is conceptually related to nearest neighbor methods for classification. Metrics are computed between feature embeddings that are subject to optimization. In the following two metric-based meta-learning algorithms are described: Siamese networks and matching networks.

Siamese Networks

Siamese networks were first proposed in 1994 by Brombley et al. for signature verification, [31]. They rate similarity of inputs by computing a metric between extracted features. Similar input images are expected to be close together in feature space. Thus, they are likely to belong to the same class. This can be conceptualized as an image matching algorithm.

Siamese networks consist of two sub-networks that are tied together by a distance function between their respective high-level features. The weights are shared between both sub-networks. There are no restrictions on the architecture of the sub-networks other than that they must be deterministically compute a feature vector. Figure 3.5.1 shows an exemplary shallow Siamese network with an input layer, a hidden layer, a distance layer, and an output layer. There are two inputs (x_1, x_2), one for each twin. Each twin applies the same hidden layers in the forward pass. In the distance layer a metric is applied to the outputs of the hidden layer. The original paper proposes to use the cosine of the angle between these feature vectors (cf. Equation (2.2.10)), [31]. Koch et al. use the L_1 norm instead, [32]. A fully-connected output layer with sigmoid activation functions yields the normalized similarity score between two inputs.

During training, the network is presented with pairs of images (x_1, x_2) randomly sampled from a dataset. These images either belong to the same class or not. The Siamese network performs binary classification by thresholding the similarity score output. The cross entropy loss function (cf. Equation (2.2.2)) is used to compute the loss value and update the weights, [32].

Networks based on the Siamese architecture can be used for one-shot classification. Take a network trained for verification on an image dataset. Consider a test image x and a set of images $\{x_c\}_{c=1}^{C}$ of classes $c = 1, \ldots, C$, none of which were used during training. For each pair (x, x_c) determine the similarity by feeding it through the Siamese network. The predicted class of x is the one with the greatest similarity, i. e. $\hat{y} = \arg\max_c p_c$. To illustrate, imagine a network trained on a glyphs from a number of alphabets that is confronted with glyphs from new alphabet drawn by different people and having to decide which glyphs

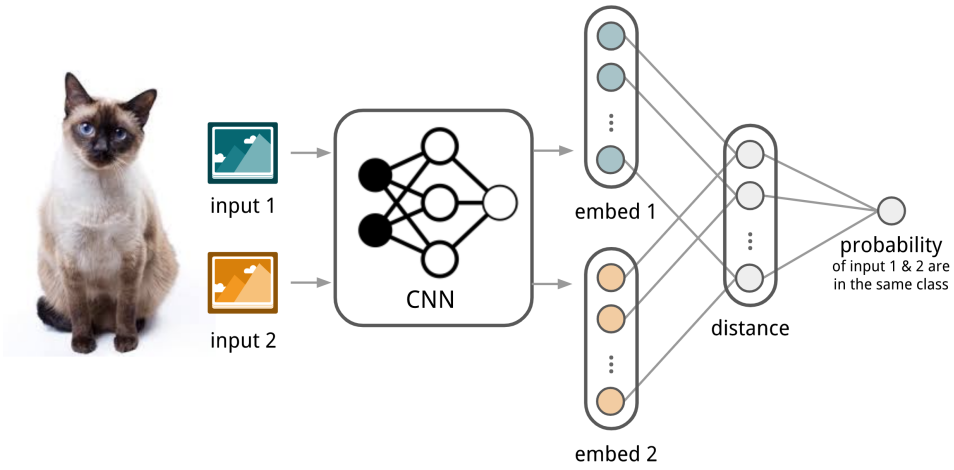


Figure 3.5.1: Siamese network. Two inputs are fed through a neural network. A distance between high-dimensional feature embeddings is computed to determine whether the inputs belong to the same class or not.

Source: Weng [33]

represent the same character. Evidently, the computational cost is proportional to the number of classes. Siamese networks are similar to nearest neighbor classification.

Koch notes two properties of Siamese networks, [34]. First, Siamese networks are *symmetric*, meaning that for a pair of two images it does not matter which image is fed into which input. Second, Siamese network predictions are *consistent*, i. e. similar images yield similar features. This assumption is the basis of why Siamese networks work. In light of recent advancements in adversarial attacks on neural networks, in particular one pixel attacks, this assumption can be doubted, [35]. Two extremely similar images that are indistinguishable by humans can be shown to result misclassification by neural networks, fooling them.

Matching Networks

Matching networks use a memory mechanism to take advantage of a small labeled support set, [36]. They are similar to Siamese networks in that they try to measure similarity between images to classify from a few examples. In contrast to Siamese networks training and testing is done on the same task.

Matching networks make use of a set of labeled images for the task. This support set S contains k examples of labeled images $\{(x_i, y_i)\}_{i=i}^{k}$, making this a k-shot learning algorithm. A Matching network classifies a test image x by defining a probability distribution over outputs $\hat{y}$

$$P(\hat{y} \mid x, S) = \sum_{i=1}^{k} a(x, x_i) y_i, \tag{3.5.1}$$

where $a(x, x_i)$ is an attention mechanism. Equation (3.5.1) expresses output probability as a linear combination of support set labels. It can be understood as a form of associative memory because it associates input with similar examples in the support set, [36].

For the attention mechanism the softmax over cosine distances between embeddings is used. Recall the definitions of softmax and cosine distance form Section 2.2. Inserting Equation (2.2.10) in Equation (2.2.1) yields the following expression for the attention mechanism.

$$\begin{aligned} a(x, x_i) &= \text{softmax}(\text{cosine similarity}(f(x), g(x_i))) \\ &= \frac{\text{cosine similarity}(f(x), g(x_i))}{\sum_{j=1}^{k} \text{cosine similarity}(f(x), g(x_j))} \\ &= \frac{\frac{f(x)g(x_i)}{\|f(x)\|\|g(x_i)\|}}{\sum_{j=1}^{k} \frac{f(x)g(x_i)}{\|f(x)\|\|g(x_j)\|}} \end{aligned} \tag{3.5.2}$$

In Equation (3.5.2) f_θ and g_θ are embedding functions, parameterized by θ. Embedding functions map an input x to a feature vector $z \in \mathbb{R}^{n_z}$. In this context CNNs are used to implement them, mapping an image input to a point in feature space. The embedding function g_θ is used to map examples from S to feature space while f_θ is used to embed the input. Instead of embedding each labeled example in isolation, Vinyals et al. propose a full context embedding, where $g_\theta(x_i, S)$ considers the entire support set when embedding x_i, [36]. This allows the network's embedding to be optimized, e. g. to differentiate between two examples that have similar embeddings but belong to different classes. Embedding functions and linear combination of labels are visualized in Figure 3.5.2.

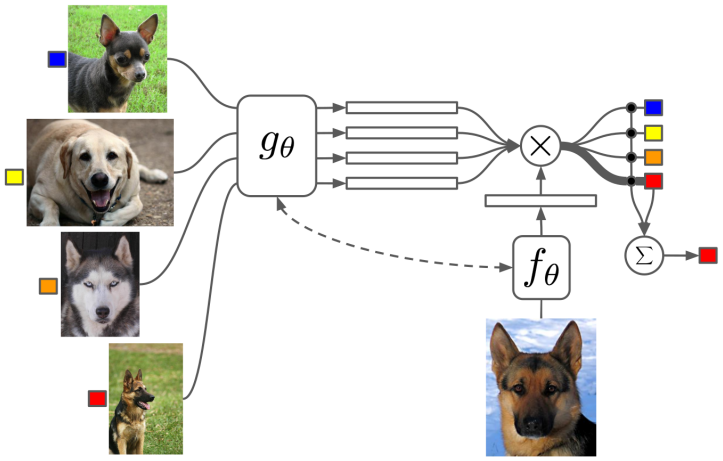


Figure 3.5.2: Matching network. Four support set examples are embedded by g_θ. An input image is embedded by f_θ and the cosine similarity to support set embedding is computed. The output class is a linear combination of support set classes weighted by similarity.
Source: Vinyals et al. [36]

For training matching networks tasks T_i are samples from a distribution of tasks $p(T)$, where each tasks is a collection of input data and labels. The task is split into a labeled

support set S and an unlabeled batch B. Formally, the training objective is given by

$$\theta = \arg\min_{\theta} \sum_{T_i \sim p(T)} \sum_{S,B \sim T_i} \sum_{x,y \in B} L(y, P(\hat{y} \mid x, S, \theta)). \tag{3.5.3}$$

The objective is to minimize the loss of predicting labels $\hat{y}$ for data x in the unlabeled part of T_i, given support set S.

3.5.2 Model-Based

Model-based meta-learning enhance neural networks additional components to learn from few examples. The following section explains memory-augmented neural networks (MANNs), neural network with added memory for fast learning.

Memory-Augmented Neural Networks

MANNs use an external memory to reduce the cost of learning. The external memory can be used to store and load information, [37]. This allows for fast learning from new data.

MANNs are based on Neural Turing machines (NTMs), [38]. NTMs add addressable memory to neural networks. They consist of two parts: a memory bank and a memory controller, embodied by a neural network. A block diagram of a NTM is shown in Figure 3.5.3.

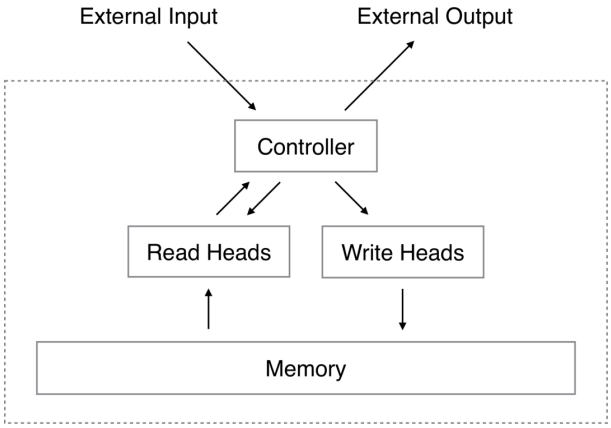


Figure 3.5.3: A Neural Turing machine. The controller recieves an external input and moves the read and write heads accordingly. The heads interact with the memory bank, producing the output.

Source: Graves et al. [38]

A NTM's memory can be thought of as an $N \times M$ matrix, where N is the number and M the size of memory locations. Interaction with memory is done through read and write operations. Given a weight vector w_t of length N at time t a read operation on memory matrix M_t is given as follows.

$$r_t = \sum_{i=0}^{N-1} w_t(i) M_t(i) \tag{3.5.4}$$

Writing is divided into two parts: erasing and adding. To erase memory the following operation is performed, given an erase vector $e_t \in (0, 1)$.

$$\tilde{M}(i) = M(i) + (1 - w_t(i))e_t \tag{3.5.5}$$

The new value can then be written to the modified memory $\tilde{M}_t(i)$ with add vector a_t in the following way.

$$M(i) = \tilde{M}(i) + w_t(i)a_t \tag{3.5.6}$$

Santoro et al. propose a least recently used access (LRUA) module for writing. The new value is written to the memory location that was used least recently. This encourages encoding of relevant information. All operations of a NTM are differentiable, making it trainable by gradient descent methods, [38].

The weight vector in Equations (3.5.4) to (3.5.6) are produced by read and write heads. Reading and writing to memory taking into account a weight vector is a blurry memory addressing mechanism. There are two types of addressing: content-based and location-based. Content-based addressing takes a key vector k of length M and compares it to each row $M_t(i)$ in memory. The key vector k is the output of a neural network. A similarity score is computed for each pair $(k, M_t(i))$. The authors of the original NTM paper propose a softmax cosine similarity metric (cf. the matching network attention mechanism, Equation (3.5.2)), [38]. As a result similar inputs and rows in memory produce higher weight values. For location-based addressing, a shift vector s_t is applied to the weight vector.

$$w_t(i) = \sum_{j=0}^{N-1} w_t(j)s_t(i - j) \tag{3.5.7}$$

Note that Equation (3.5.7) is a convolution in 1D.

During training a MANN stores representation information in memory. This information can be retrieved later to classify new examples. Information retrievel is done by content-similarity. Among other advantages, the use of a memory block makes multiple forward passes as used by many metric-based meta-learning algorithms redundant.

3.5.3 Optimization-Based

LSTM Meta-Learning

LSTM meta-learning aims to find an initialization and update rule for parameters that lead to good performance in few-shot learning settings, [39]. It uses an LSTM cell for storing and updating model parameters.

Ravi and Larochelle propose to use an LSTM cell to learn an update rule, using the cell state to represent learnable model parameter, [39]. Recall the gradient descent Equation (2.3.12).

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} L_t$$

The update step can be formulated as cell state update of an LSTM (cf. Equation (2.3.8)) by setting $f_t = 1$, $c_{t-1} = \theta_{t-1}$, $i_t = \alpha_t$, and $\tilde{c}_t = -\nabla_{\theta_{t-1}} L_t$.

$$c_t = f_t c_{t-1} + i_t \tilde{c}_t \tag{3.5.8}$$

Input and forget gate are optimized during meta-training. In parametric form, the input gate equation for LSTM meta-learning is given by

$$i_t = \sigma(W_i \cdot [\nabla_{\theta_{t-1}} L_t, L_t, \theta_{t-1}, i_{t-1}] + b_i), \tag{3.5.9}$$

where $[\cdot, \cdot]$ denotes a concatenation of vectors. In the context of LSTM meta-learning the input gate can be thought of as the learning rate. The input gate controls to what degree the gradient is applied to the parameters. It is a function of current gradient, current loss, previous parameters and previous learning rate. In a similar manner the forget gate can be parameterized as follows.

$$f_g = \sigma(W_f \cdot [\nabla_{\theta_{t-1}} L_t, L_t, \theta_{t-1}, f_{t-1}] + b_f), \tag{3.5.10}$$

There is no equivalent to a forget gate in gradient descent. However, Ravi and Larochelle note that this mechanism can help escape from local minima, [39]. Gradient descent always retains the full value of previous parameters, relying on the gradient to be non-zero. If the gradient vanishes the learning algorithm can be stuck at a local minima. By having the option to scale parameters at every update step it becomes possible to escape such situations. In addition to the update rule the initial cell state c_0 is learned during training to provide an optimal initialization for model parameters.

During meta-training training and test dataset D_{train} and D_{test} are sampled from a distribution of datasets. A copy of parameters is updated using Equation (3.5.8) by training on batches of D_{train}. After T training steps these parameters are evaluated on D_{test}. Finally, the meta-learner's parameters are updated using the loss on the test set. The training procedure is illustrated in Figure 3.5.4.

The LSTM meta-learning algorithm requires the computation of second-order derivatives which is computationally expensive. Ravi and Larochelle assume that gradients of the learner are independent of gradients of the meta-learner, thus avoiding the computation of higher-order derivatives. They show experimentally that their proposed LSTM meta-learner outperforms existing approaches.

Model-Agnostic Meta-Learning

Model-agnostic meta learning trains a model on a multitude of tasks with only few examples and training steps to produce parameters that generalize well to new tasks. It does not make any assumption on the model other than that it is trained with gradient descent, making it model-agnostic. It can be applied to classification, regression, and reinforcement learning with fully-connected, convolutional or recurrent neural networks.

The objective of meta-learning is to produce a parameterization that is useful across many tasks, thus being adaptable to new ones by fine-tuning. In other words, the loss functions

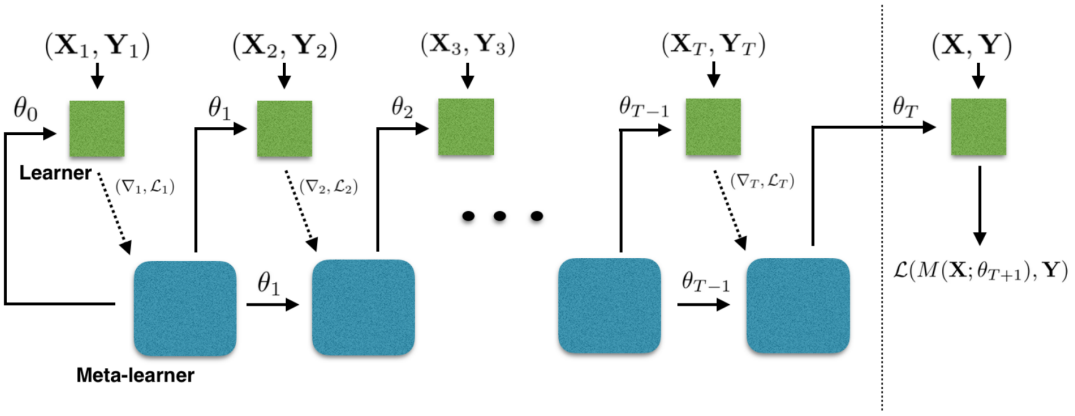


Figure 3.5.4: LSTM meta-learning. The learner updates model parameters on batches of training data (X_i, Y_i). After T steps the meta-learner updates meta-parameters using Equations (3.5.8) to (3.5.10).

Source: Ravi and Larochelle [39].

should be sensitive to new tasks. A high sensitivity to new tasks means small changes to model parameters θ cause large changes in the loss value $L(\theta)$. Figure 3.5.5 visualizes loss sensitivity.

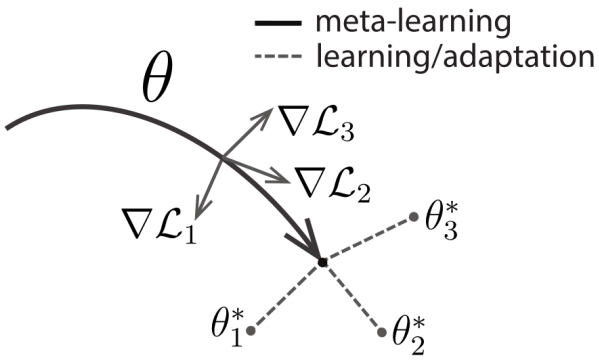


Figure 3.5.5: Model-agnostic meta learning. Model parameters θ are optimized during meta-training such that adapting to new tasks (θ_1, θ_2, and θ_3) requires only a few gradient steps.

Source: Finn et al. [8]

Consider a set of tasks T for supervised learning. Each task T_i consists of input-output tuples (x_i, y_i). During the meta-learning phase the model is trained on samples of n tasks drawn from a distribution $p(T)$. For each task k examples are sampled, loss $L_{T_i}(f_\theta)$ and gradients $\nabla_\theta L_{T_i}(f_\theta)$ are computed and task-specific parameters $\theta_i{}'$ are computed with gradient descent as follows (cf. Algorithm 1, line 6).

$$\theta_i^{'} = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$$

The parameters trained on this task for one or more training step are then tested on held-out samples of the same task. The loss $L_{T_i}(f_{\theta_i'})$ of a model f with parameters $\theta_i^{'}$ on

these test samples is to be minimized.

$$\min_{\theta} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) = \min_{\theta} \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta - \alpha \nabla_\theta L_{T_i}(f_\theta)}) \tag{3.5.11}$$

It measures how well the the model generalizes when trained on a new task. The test error of learning from k samples is used as training error for meta-learning. Applying the gradient descent update rule of Equation (2.3.12) to the objective formulated in Equation (3.5.11) leads to the folowing expression.

$$\begin{aligned} \theta &\leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'}) \\ &= \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta - \alpha \nabla_\theta L_{T_i}(f_\theta)}) \end{aligned} \tag{3.5.12}$$

Equation (3.5.12) computes the gradient of a gradient, in other words a second-order partial derivative. Experiments have shown that first-order model-agnostic meta learning (FOMAML), a first-order approximation of MAML, can be sufficient, [8]. Algorithm 1 outlines the complete MAML algorithm.

Algorithm 1 Model-agnostic meta learning, [8]

Require: $p(T)$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $T_i \sim p(T)$
4: **for all** T_i **do**
5: Evaluate $\nabla_\theta L_{T_i}(f_\theta)$ with respect to k examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta L_{T_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} L_{T_i}(f_{\theta_i'})$
9: **end while**

Two different step size parameters or learning rates are used, α for the task update step and β for the meta update step. After meta-learning the model performance is evaluated on tasks held out from meta-training. In contrast to other initialization methods MAML optimizes fast adaptability to new tasks. A network pre-trained using this framework is faster at adapting to new tasks than random initializations or generic ImageNet initializations. To adapt to a new problem parameters trained with meta-learning need to be fine-tuned on samples of a new task.

Chapter 4

Implementation

This chapter explains the implementation of algorithms to transfer learned behavior from simulation to reality. First, different approaches presented in chapter 3 are compared in section 4.1 and a selection of them is chosen to be implemented. Then, section 4.2 describe the hardware and software used in experiments, namely EyeBot and EyeSim. Section 4.3 sheds light on the process of collecting data in simulation and reality. Finally, section 4.4 describes details regarding training such as network architecture and measures to prevent overfitting.

4.1 Comparison and Selection of Approaches

This section compares approaches to bridge the gap between simulation and reality presented in chapter 3. There are three main categories of such approaches: Domain randomization (section 3.3), domain adaptation (section 3.4), and meta-learning (section 3.5). Each of them has a number of subcategories with advantages and disadvantages for different types of problem settings. For each main category an algorithm is selected to be implemented in this thesis and subsequently evaluated.

Domain Randomization

Among the domain randomization techniques GAN-based GDR have come to dominate others in the past few years. They show great performance on benchmark datasets such as MNIST and work well for real-world applications, [22], [28]. However, they are computationally expensive since they encompass training multiple networks. Furthermore, they are potentially unstable during training. Some of them require occasional real-world roll-outs or a labeled validation dataset.

Unguided domain randomization on the other hand is less computationally expensive when it comes to generating a synthetic dataset. It simply varies selected parameters uniformly in a given range. There is no guarantee for its success since no optimization of parameters

is done. It relies on expert knowledge to select which simulation parameters to randomize. However, it has often been shown to yield good results, [4], [23].

The necessity of real-world roll-outs and labeled validation dataset are in conflict with the goals set out in section 1.3. For reasons of simplicity and to limit the scope of this thesis unguided domain randomization is selected. The task at hand—following a lane—is relatively simple compared to grasping objects of unknown shape as done by Tobin et al., [4]. There are less degrees of freedom in the actuator and the environment is more well-defined.

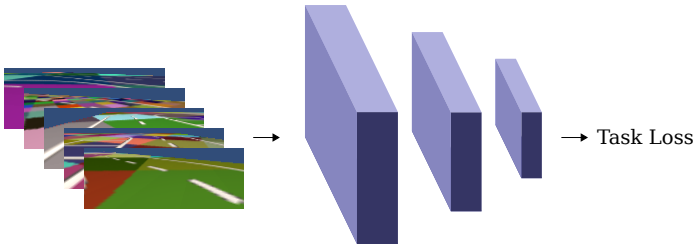


Figure 4.1.1: Schematic domain randomization. A network is trained using large amounts of randomized synthetic data to learn the target task. Generalization to target domain is achieved through diversity of training data.

Domain Adaptation

Input-level domain adaptation is closely related to GDR using GANs. It suffers from the same drawbacks: high computational complexity and training instabilities. Furthermore, there is an additional disadvantage: If one were to use a GAN to map a real-world image to another domain at test time before presenting the task network with its input, said GAN needs to be able to run in real-time. EyeBots use a Raspberry Pi 4 with limited computational power. Thus, input-level domain adaptation with image translation in real-time is likely infeasible.

Among the feature-level domain adaptation algorithms weight regularization combined with domain confusion stands out (cf. section 3.4.1). According to Rozantsev et al. it achieves high accuracy on the Office dataset, a common domain adaptation benchmark dataset, [7]. It is the most complex feature-level domain adaptation approach presented in this thesis with respect to the number of loss terms and hyperparameters is has. However, it is still simple to implement but may require some tuning of loss weights. Thus, weight regularization with added domain confusion is chosen.

Furthermore, a variation of conventional domain adaptation is implemented. The source dataset generated in simulation is replaced with a randomized dataset. The coalescence of domain adaptation and randomization has the potential to yield better generalization results.

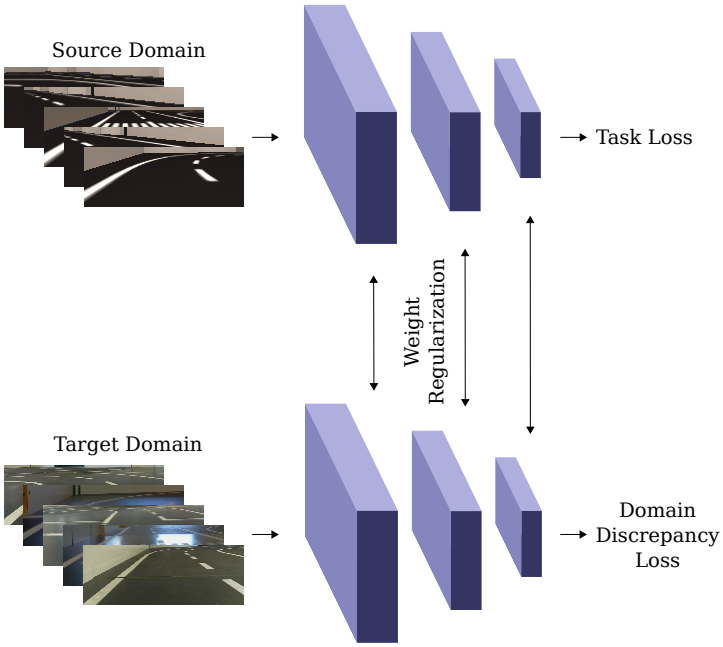


Figure 4.1.2: Schematic domain adaptation. Two networks are trained simultaneously: The source network is trained to perform the main task while the target network is trained to be domain-invariant. Weights of the two networks are regularized to prevent divergence while allowing minor adjustment.

Meta-Learning

Metric-based meta-learning can be shown to be able to correctly classify previously unseen examples. Typically, they use an appropriate metric to compare feature representations. This requires multiple forward passes at test time. For reasons stated above this is infeasible on the hardware used.

Model-based meta-learning does not require multiple forward passes. It uses a memory mechanism to store information and retrieve it as they see fit. This makes them superior for applications with limited computational resources. However, they require modification of the network architecture which is undesirable.

Optimization-based meta-learning appears to solve issues that metric-based and model-based meta-learning entail. It has no impact on the runtime at test time and does not dictate a particular network architecture. MAML is chosen since it outperforms LSTM meta-learning, [8].

4.2 Hardware and Simulation Environment

This section described hardware and simulation environment used in the experimental part of this thesis. Experiments are carried out on EyeBots and in EyeSim.

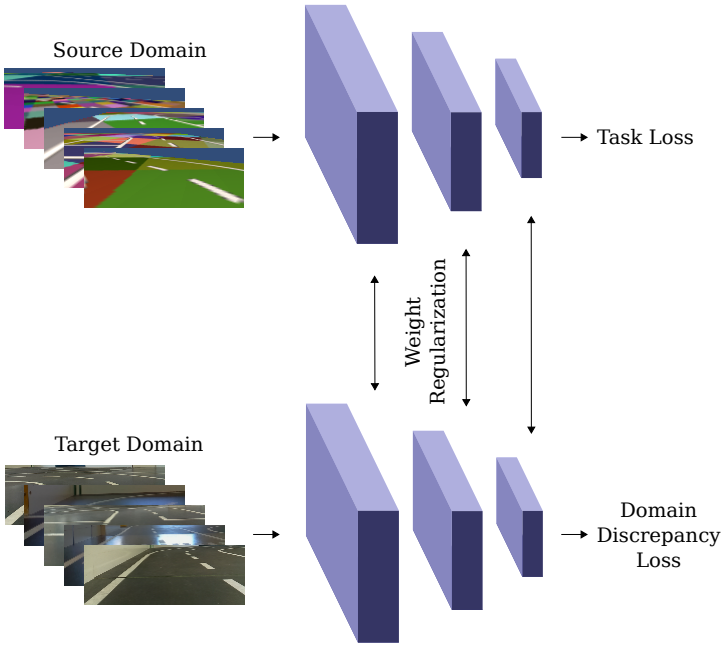


Figure 4.1.3: Schematic domain adaptation and randomization. This setting is similar to domain adaptation, except source domain data is replaced by randomized data. Thus, the target network is adapted to a plethora of source domains instead of merely a single manifestation.

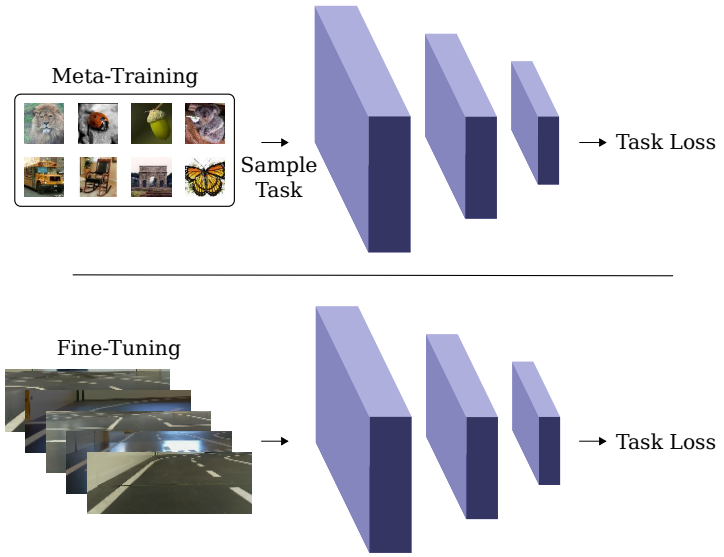


Figure 4.1.4: Schematic meta-learning. During meta-training the network is trained on tasks sampled from a collection of tasks. It is trained explicitly to learn new tasks fast. Then, it can be fine-tuned on the target domain task using few data and iterations.

4.2.1 EyeBot

EyeBot is an embedded controller for robotics applications developed by the University of Western Australia (UWA) Robotics and Automation Lab, [40]. Its most recent version is based on a Raspberry Pi 4, using Raspbian 10 (Buster) as its operating system. The controller is connected to an IO-board via USB. The IO-board includes hardware and software drivers for various sensors and motors. Among the sensors are three infrared distance sensors, a Raspberry Pi camera module, and encoders to read motor positions. Two servo motors can be controlled independently. Optionally, an LCD screen can be connected to the Raspberry Pi. An Ethernet port and Wi-Fi connection are available. Figure 4.2.1 depicts an Eyebot following a lane.

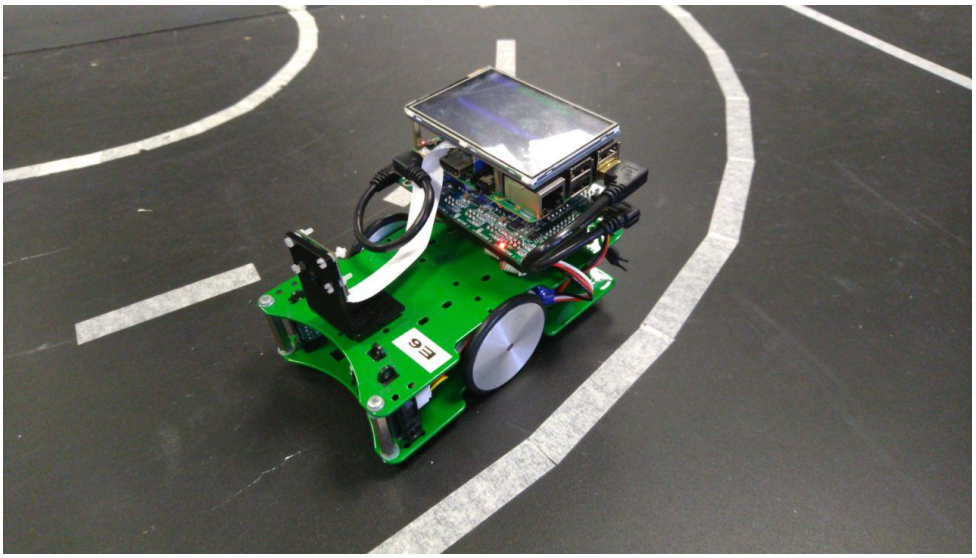


Figure 4.2.1: Photo of an EyeBot following a lane.

A software library called RoBIOS provides an abstraction layer for sensors and motors. It comes with C, C++ and Python bindings. To use it, the user needs to include an `eyebot.h` file if using C or C++. For Python an `eye.py` module can be imported. The API provides access to sensor readings, camera image, basic image processing, LCD display, and a differential driving interface. A full list of available functions can be found in [40].

4.2.2 EyeSim

EyeSim VR is a simulation environment that simulates EyeBot functionality, [41]. It is based on the Unity game engine. Sensor readings and robot movement are simulated in real-time. Robots in EyeSim respond to the RoBIOS API. It is possible to create and load custom robot models, objects, and environments. EyeSim can be used to prototype and test algorithms in a safe environment. It is not meant to be photo-realistic. Figure 4.2.2 shows a screenshot of EyeSim.

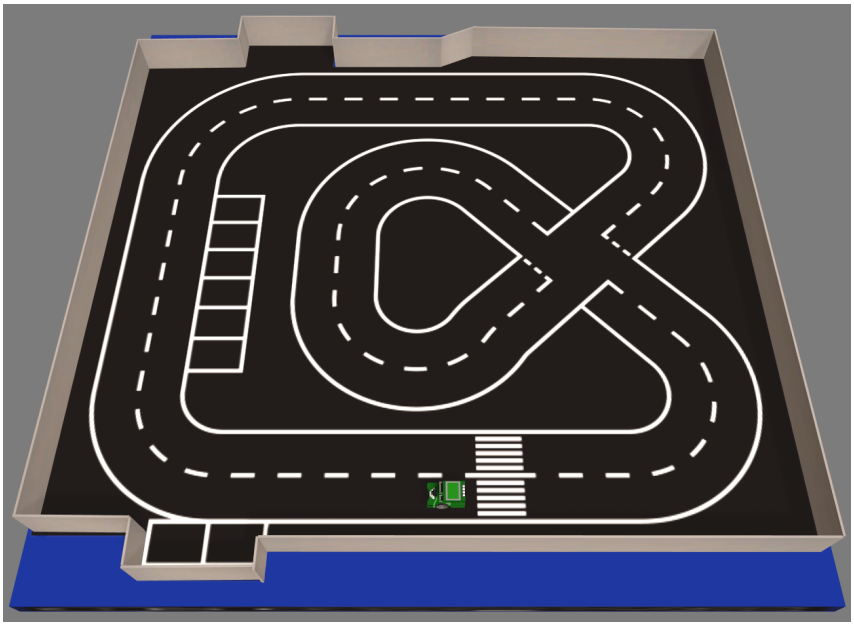


Figure 4.2.2: Screenshot of EyeSim VR.

4.3 Data Collection

This section describes the process of collecting data for experiments with EyeBots. In the context of this thesis data refers to image-action-pairs. Images can be taken from a real or simulated robot. Data collection differs between the two domains because additional information about robot position is available in simulation. For some of the real-world images no ground-truth action is available. All datasets used in experiments are summarized in table 4.1[1].

Table 4.1: Overview of datasets used in experiments.

Domain	# D_{train}	# D_{val}	# D_{test}
Reality (labeled)	2000	500	500
Reality (unlabeled)	8000		
Simulation	10000	2000	
Domain Randomization	10000	2000	

During data collection the robot is controlled using differential driving with a constant linear velocity. The angular speed controls the steering direction. Angular speed is normalized to $[-1, 1]$ for training, where positive values steer left, negative values steer right and zero means go straight. To simplify the problem the continuous angular speed is discretized to five classes with values $[-0.7 - 0.25, 0.0, 0.25, 0.7]$.. The classes correspond to "hard right", "slight right", "straight", "slight left", and "hard right", respectively. While

[1] All datasets are publicly available at `https://www.kaggle.com/felixwege/eyebot-autonomous-driving-dataset`.

discretization of steering may cause loss of precise steering for difficult maneuvers it is sufficient for lane-following as demonstrated by Xu et al., [42].

Simulation

Data collection in simulation is automated to gather large amounts of data effortlessly. An EyeSim world file resembling the simplified Carolo-Cup course is used as shown in figure 4.2.2. EyeSim provides some additional functions that give information about position and velocity of the vehicle that is unavailable in reality. This information is used to implement a PID controller to follow the road. The controller has two input variables: distance between vehicle and lane center as well as difference between vehicle heading and desired heading to follow the road. Controller gains are tuned by hand.

Images retrieved by the RoBIOS API are available in various resolutions with a 4:3 aspect ratio. To reduce file size and computational cost images are downsampled to 160×120 pixels. Note that all information about the course is in the bottom half of the image. Hence, images are cropped to the bottom 160×60 pixels, further reducing size and cost. Images are saved with three color channels using the Portable Network Graphics format. Actions are saved in a comma-separated values file. Data collection is done at 10 Hz. An example image from simulation is shown in figure 4.3.1b.

Gathering data in simulation in an automated fashion enables collection of almost arbitrary amounts of data. For experiments conducted in this work 10000 simulated training images and corresponding steering angles are recorded. To supplement training data a validation set of 2000 labeled images is collected. Additionally, 8000 unlabeled images are collected to be used for domain adaptation.

(a) Reality (b) Simulation (c) Domain Randomization

Figure 4.3.1: Example images from the perspective of an EyeBot in reality, simulation, and domain randomization as they are used for training. The three images depict a similar scenario but differ in appearance.

Domain Randomization

Data collection for domain randomization follows the procedure outlined in the previous section, except that the floor texture is randomized. The floor texture is randomized by pasting colored squares on a canvas and superimposing the road markings on top. Domain randomization data collection is discussed in detail in section 3.3. An example image collected using domain randomization is shown in figure 4.3.1c. In total 10000 training images and 2000 validation images are gathered using domain randomization.

Since domain randomization and data collection in simulation can be automated, large amounts of data can be collected with little effort.

Reality

Data collection in reality is less straightforward since no perfect knowledge about vehicle position and velocity is available. Different solutions are conceivable. For example, one could implement a lane-following using classical image processing and control theory. Alternatively, EyeBots can be controlled remotely while recording data. Since only a small amount of labeled data from reality is required this approach is chosen. The vehicle is driven around the course manually to collect 2000 training and 500 validation data. An example image collected in reality is shown in figure 4.3.1a. In addition, 500 test images are gathered. There are used to evaluate and compare performance of algorithms in the following sections. They are never used for training or validating models or hyperparameters. The simplified Carolo-Cup course as it is set up in the UWA's Robotics and Augmentation Lab is shown in figure 4.3.2.

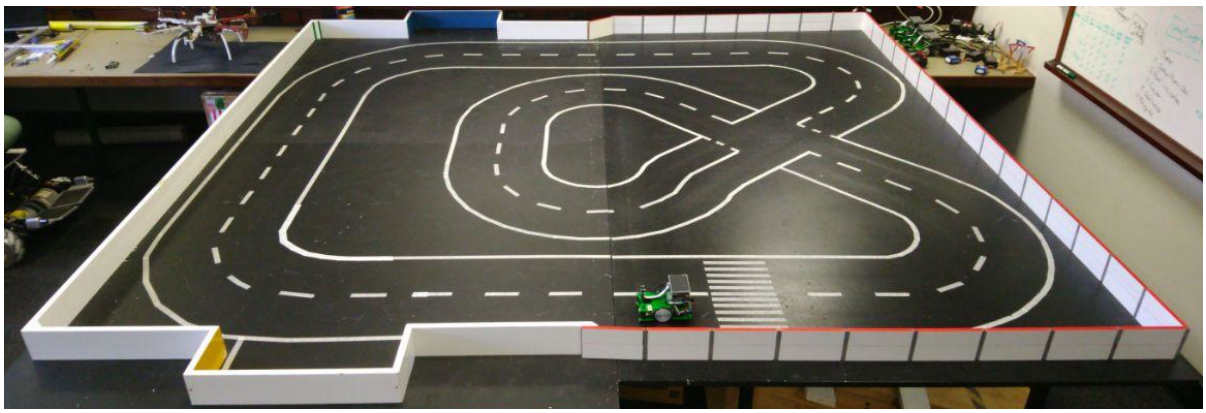


Figure 4.3.2: Photo of an EyeBot on the Carolo-Cup track.

4.4 Training

4.4.1 Software Libraries

All software in this thesis are written in the Python programming language, using common packages such as NumPy for array operations and OpenCV for image processing. TensorFlow 2 is used to implement deep learning algorithms. It is chosen over other frameworks such as PyTorch because is offers more support and optimization for deployment on mobile and embedded devices. Version 2 of TensorFlow offers some advantages over version 1, for example eager execution, cleaner API and a more "pythonic" style.

TensorFlow has integrated Keras, a high-level interface to constructing and training deep learning models. Keras comes with numerous models, layers, loss functions, optimizers,

metrics, and utilities. It also provides base classes from which to inherit, making it easy to implement custom objects.

While Keras provides useful tools to train simple models, advanced deep learning algorithms often require the implementation of a custom training loop. To train a model using gradient descent, the following steps are necessary: Forward pass of input tensors, computation of loss, computation of gradients w. r. t. trainable parameters, and update of said parameters. Implementing a custom training loop gives full control over implementation of extravagant architectures, loss functions and gradient updates described in chapter 3. TensorFlow's `GradientTape` is used for automatic gradient computation. It can be used as a Python context manager. Within the context all trainable variables and operations using them are recorded.

TensorFlow 2 offers eager execution which simplifies debugging. Ordinarily, TensorFlow constructs a computational graph from operations. When using eager execution all operations are evaluated immediately. However, this has a negative effect on execution time. TensorFlow 2 introduces a Python function decorator `tf.function`. Python functions that are annotated with this decorator are converted to tensorflow graph operations to speed up computations.

The input pipeline utilizes TensorFlow's `Dataset` API. With this API large dataset can be handled. It can be used to apply transformations and data augmentation to data. Dataset objects can consume Python generator expressions. When iterating over a dataset only the current batch is loaded into memory. Each input tensor is of size $n_{batches} \times h \times w \times n_{channels}$, where h and w are image height and width, respectively. Actions are one-hot encoded after discretization, resulting in a $n_{batches} \times n_{classes}$ tensor.

To monitor performance during training TensorBoard is used. TensorBoard can visualize losses, metrics, models, images, and other data. During training, losses and metrics are evaluated on the training set during each step. Periodically, the model is evaluated on the validation set and images with predicted and ground-truth are saved. Evaluation and writing to TensorBoard is done using custom callbacks that are called at the beginning and end of every step and epoch.

Models and their parameters are saved using TensorFlow's `SavedModel` format. A saved model contains all operation and weights associated with a model. It can be loaded to continue training or used for deployment. At regular intervals the current model is saved. In addition to that, n models with the best performance w. r. t. to a selected metric are kept. This constitutes a form of early stopping. Details on early stopping and other regularization techniques are described in section 4.4.3.

TensorFlow Lite offers tools to optimize performance on mobile and embedded devices. Mainly, it consist of two components: a converter and an interpreter. The converter can transform a saved model into a serialization that can be process by the interpreter, optimizing size and performance by quantization of weights. Instead of using the default of 32 bit floating point numbers for weights, 8 bit integers can be used to reduce file size by up to 4 times and performance by a similar factor. This is explored in section 4.4.2. The interpreter can load an optimized model and perform inference.

Ultimately, models are supposed to run on an Eyebot with Raspberry Pi. At the time of writing this thesis Raspbian 10 does not support TensorFlow 2. For inference on EyeBots all saved models are converted to a TensorFlow 1 compatible format before deployment.

4.4.2 Network Architecture and Hyperparameters

Training neural networks often involves several hyperparameters, such as learning rate, weight decay, and the network architecture itself. Hyperparameter are not optimized during training, but have an influence on it. Hyperparameter optimization is a field of interest of its own, with approaches ranging from grid or random search to evolutionary algorithms, [5]. In this thesis most hyperparameters are set to sane default values. Only learning rate and domain adaptation weights are optimized. Table 4.2 lists hyperparameters used in experiments. Some of the parameters, such as learning rate, dropout rate, and weight decay, are further explained in the following sections. This thesis uses the Adam optimizer in all experiments, [43].

Table 4.2: Hyperparameters

Hyperparameter	Value
Input shape	$160 \times 60 \times 3$
Batch size	128
Epochs	1000
Learning rate	10^{-4}
Dropout rate	0.5
Label smoothing	0.1
Weight decay	0.01

MobileNetV2

Before conducting any other experiments the network architecture is determined. Two architectures are compared: MobileNetV2 and a CNN that has the same number of layers as MobileNetV2 with the same input and output dimensions. Figure 4.4.1 shows the basic network architecture[2]. MobileNetV2 has 19 inverted residual with linear bottleneck layers (cf. section 2.3), [11] It has $2,290,437$ trainable parameters. In contrast, a CNN with the same layer structure has $5,208,357$ trainable parameters. This has an impact on model capacity, accuracy, inference speed, and file size. The two different networks are compared in the following experiment.

Experiments Networks are trained on 2000 labeled images from reality to compare the architectures. To increase inference speed they are converted to TF Lite. Four different

[2]Network architecture figure was generated using `https://github.com/HarisIqbal88/PlotNeuralNet`.

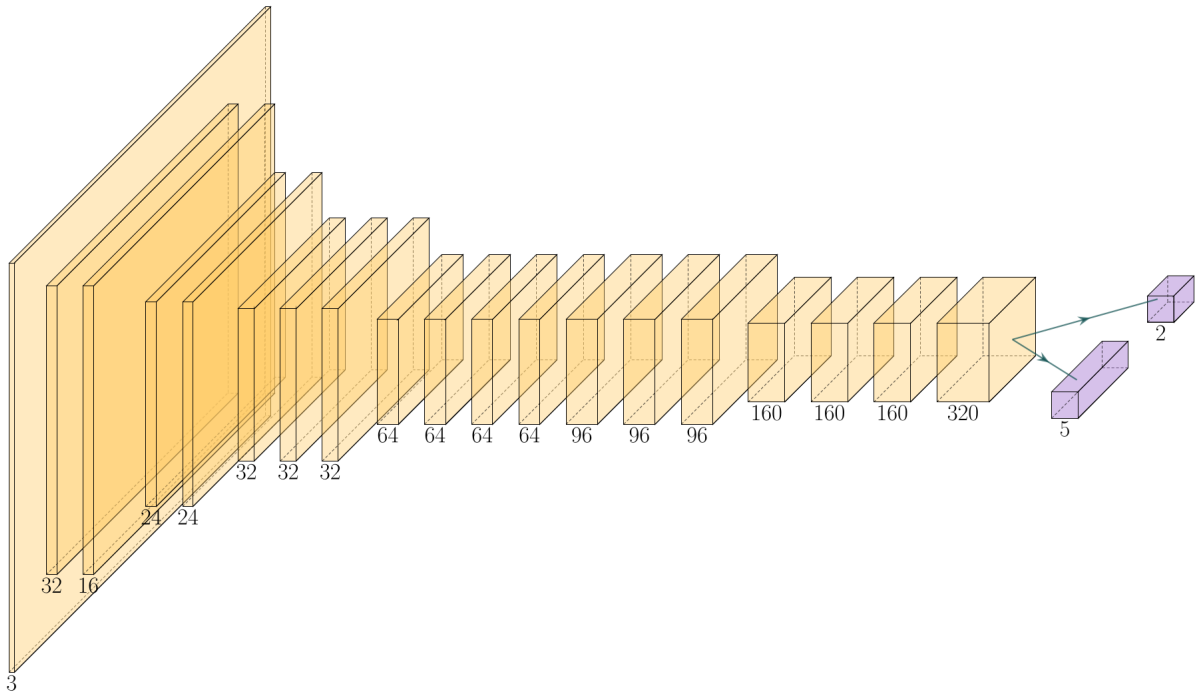


Figure 4.4.1: The Network architecture is based on MobileNetV2. It uses a custom head with two outputs: A 5-way classification for steering and a binary domain classification (both shown in purple). Feature maps of convolutions are shows in yellow. The number of filters is written underneath each feature map. The input dimension is $160 \times 60 \times 3$. As the size is reduced by pooling operations or strided convolutions the number of filters is increased. Except for the output layer the network is fully convolutional.

networks are compared: CNN, CNN (TF Lite), MobileNetV2, and MobileNetV2 (TF Lite). Their accuracy on the test set and inference time on an EyeBot are compared in table 4.3. Figure 4.4.2 shows box plots of inference times.

Table 4.3: Comparison of MobileNetV2 and CNN, with and without TF Lite. The accuracy is computed on the test set of 500 images. Time per iteration is the median of 1000 iterations on an EyeBot.

Network	# Params	Accuracy	Time/it
CNN	5.2 M	90.20 %	142 ms
CNN (TF Lite)	-	90.00 %	77 ms
MobileNetV2	**2.3 M**	**91.00 %**	140 ms
MobileNetV2 (TF Lite)	-	90.40 %	**66 ms**

Results All networks achieve upwards of 90 % accuracy on the test set. There is no significant difference in accuracy between MobileNetV2 and equivalent CNN for this task. TF Lite conversions have a slightly lower accuracy than their unconverted counterparts.

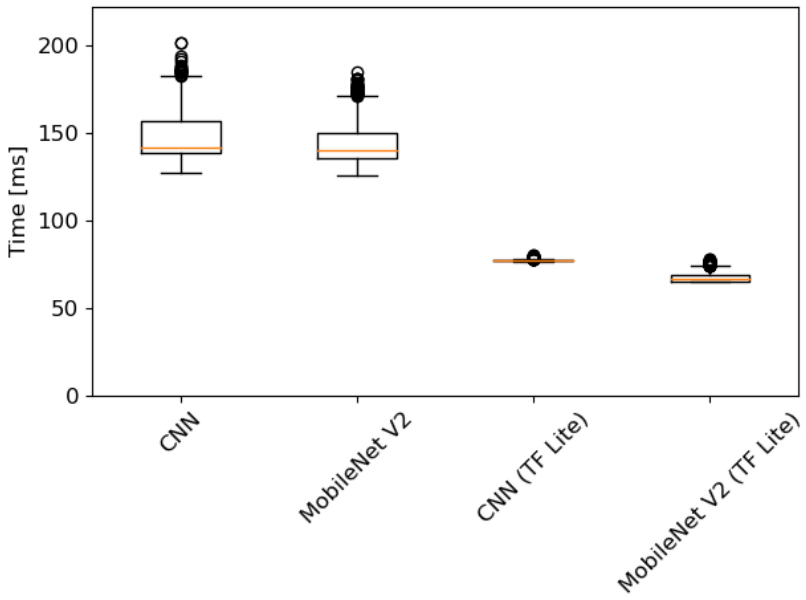


Figure 4.4.2: Box plots of inference time of different models on an EyeBot. Models converted to TF Lite are twice as fast as their counterparts. MobileNetV2 is slightly faster than its CNN equivalent.

However, the time per iteration of unconverted networks is twice as long. The median iteration time of MobileNetV2 on an EyeBot is 140 ms while the network converted to TF Lite only takes 66 ms. TF Lite is vital to meeting the runtime requirement set out in section 1.3. Given that MobileNetV2 converted to TF Lite is the fastest at almost no loss in accuracy, this architecture is chosen for all remaining experiments.

Learning Rate

Learning rates are crucial for training deep learning models, [44]. In the gradient descent update equation (2.3.12) the learning rate α determines to what extend the gradient is applied to trainable parameters. If the step size is too small training converges slowly or becomes stuck in a local minimum, [5]. Higher learning rates can yield faster convergence and escape local minima. However, they can cause overshooting and divergence. Often the learning rate is decayed during training to find a balance between exploration and exploitation.

To find a sensible initial learning rate the following algorithm is implemented. Starting from a learning rate that is far too small the learning rate is increased exponentially after every step. Loss and gradient are computed and network parameters are updated. The loss is monitored during this procedure. Typically, the loss decreases slowly at first but decreases increasingly faster. At a certain point the loss reaches a minimum and increases again. This is the point where a large learning rate causes divergence. At this point training is aborted. The point where the loss decreased the fastest is a suitable learning

rate to start training with. Algorithm 2 exemplifies this algorithm. This learning rate algorithm is based on [44]. An example of learning rate finding is shown in figure 4.4.3.

Algorithm 2 Learning rate finder

Require: $\gamma > 1$
1: Initialize α
2: **while** $\alpha <$ threshold or loss not divergent **do**
3: Compute loss, update weights
4: $\alpha \leftarrow \gamma \cdot \alpha$
5: **end while**
6: **Return** α where loss decreased most

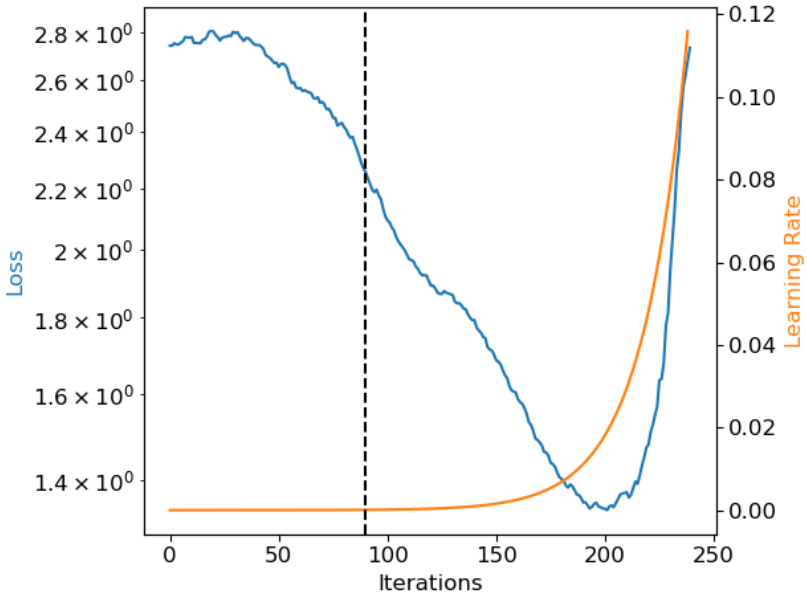


Figure 4.4.3: The learning rate (shown in orange) is increased exponentially until the training loss explodes. Training loss is plotted against a semi-logarithmic scale. It decreases slowly at first, decreases increasingly faster as the learning rate grows until the loss explodes. The optimal learning rate occurs at the dashed line.

4.4.3 Measures to Prevent Overfitting

This section briefly describes measures to prevent overfitting that are common practice in deep learning.

Data Augmentation

Data augmentation is a technique to generate new data for training applying transformations to existing training data. It is especially effective for computer vision applications.

Images can be translated, rotated, flipped, and changes to brightness, contrast, and hue can be made to multiply the amount of data, [5]. Also noise can be added. Table 4.4 list randomized translations that are applied.

Table 4.4: Data augmentation parameters

Randomization	Value
Brightness	$[-20\,\%, 20\,\%]$
Contrast	$[50\,\%, 100\,\%]$
Salt and Pepper Noise	$1\,\%$
Flip vertically	$50\,\%$

Weight Decay

Weight decay, also known as squared L2 weight regularization, penalizes model parameters θ for growing too large. To perform weight decay the L2 norm, i. e. the squared sum of all weights, is added the loss $L(\theta)$, resulting in a new loss $\tilde{L}(\theta)$.

$$\tilde{L}(\theta) = \frac{\alpha}{2}\theta^T\theta + L(\theta) \tag{4.4.1}$$

Here, α is the weight decay rate. It is divided by two to make the gradient more convenient. In this thesis $\alpha = 0.01$ for all experiments. The gradient is given by

$$\nabla_\theta \tilde{L}(\theta) = \alpha\theta + \nabla_\theta L(\theta). \tag{4.4.2}$$

The gradient descent update step with weight decay can be expressed as follows.

$$\begin{aligned} \theta_i &= \theta_{i-1} - \epsilon\nabla_\theta\tilde{L}(\theta) \\ &= \theta_{i-1} - \epsilon(\alpha\theta + \nabla_\theta L(\theta)) \\ &= (1 - \epsilon\alpha)\theta_{i-1} - \epsilon\nabla_\theta L(\theta) \end{aligned} \tag{4.4.3}$$

Weight decay shrinks the weights in every update. Thus, weights are encourages to remain reasonably small, [5].

Early Stopping

Another measure to prevent overfitting is early stopping. Early stopping exploits a common phenomenon when training models with sufficient capacity to overfit. Often, the training loss decreases steadily. However, the validation loss, after initially decreasing, starts to increase again. After this point the model starts to overfit, [5]. Figure 4.4.4 shows a typical learning curve. The model parameters that lead to the lowest validation loss are likely to perform well on the test set. Early stopping proposes to prefer these parameters and stop training when the validation loss has not decreased for some time. In this thesis every model is trained for 1000 epochs and the best-performing one on the validation set is used for testing.

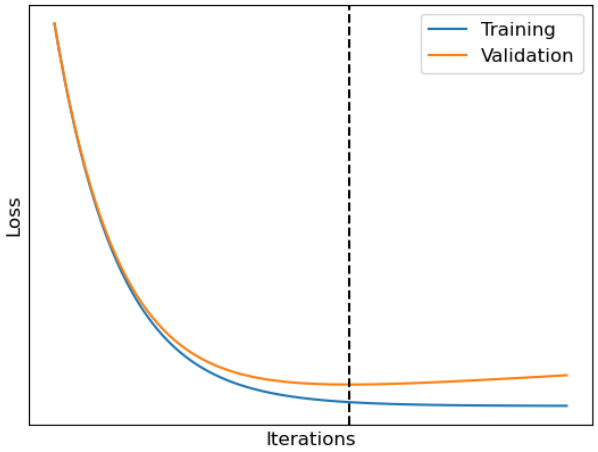


Figure 4.4.4: Typical U-shaped learning curve of a model overfitting. The training loss, shown in blue, decreases steadily. The validation loss stops decreasing and increases again after a certain number of iterations.

Dropout

Dropout is related to ensemble learning, i. e. training multiple models to perform the same task and averaging their output, [5]. It is computationally inexpensive. Dropout is implemented by removing a random subset of units from a network during training (by multiplying them with zero). The probability of an individual unit being "dropped" is given by the dropout rate (cf. table 4.2). During inference all units are used. Dropout reduces the risk of overfitting by encouraging robust features and redundant paths.

Batch Normalization

Batch normalization is an adaptive reparametrization that facilitates training of deep neural networks. In practice neural networks are trained in batches, i. e. multiple input images are processed in parallel. Gradients are averaged across the batch and applied to the weights. Batch normalization proposes to replace the batch of activations H of each layer with the following expression at training time.

$$\tilde{H} = \frac{H - \mu}{\sigma} \tag{4.4.4}$$

The activations H are normalized by subtracting the mean μ and dividing by the standard deviation σ of the batch of activations. This operation is included in back-propagation. Thus, the gradient can not increase mean or standard deviation of activations. At test time a moving average of activations can be used. While batch normalization was initially proposed to improve optimization it also has a regularizing effect, [5].

Chapter 5

Evaluation

This chapter describes experiments conducted with algorithms selected in section 4.1. All algorithms are evaluated on a test dataset of 500 real images as explained in section 4.3 w. r. t. categorical accuracy. Real world lane-following capabilities are evaluated using an autonomy rating.

At first, naive transfer from simulation to reality is explored in section 5.1. Domain randomization results are discussed in section 5.2. Next, section 5.3 explores the application of unsupervised and semi-supervised domain adaptation to lane-following. In section 5.4 MAML and its first-order approximation are compared. Trained networks are demonstrated to learn the detection of reasonable road features in section 5.5. Finally, all of the above approaches are deployed on an EyeBot and their performance in reality is evaluated in section 5.6 Two different settings are contrasted: 20-shot learning and 0-shot learning.

5.1 Transfer Learning

This section describes transfer learning using a pre-trained MobileNetV2 for lane-following. The network is pre-trained on ImageNet. It is fine-tuned on data from simulation and reality to explore the effect different amounts of data from those domains has.

5.1.1 Transfer Learning to Simulation

Experiments Before moving on to real data, this sections shows that the chosen architecture and hyperparameters are capable of solving the problem in simulation. A number of models are trained using increasing numbers of data to verify the amount of data is sufficient. Each model is trained for 1000 epochs. Measures against overfitting as described in section 4.4.3 are used. The model with the highest validation accuracy is used for comparison.

Results Figure 5.1.1 shows classification accuracy for models trained with amounts of data ranging from 100 to 10000 examples. Training and validation dataset consist

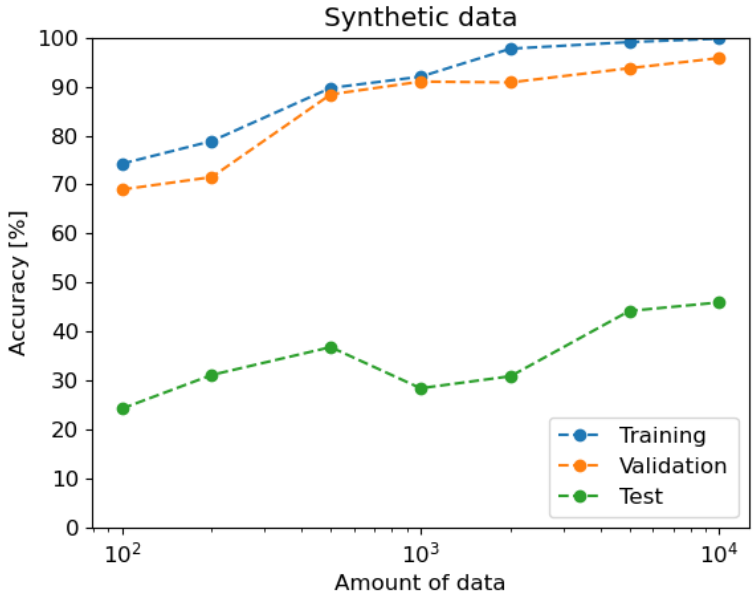


Figure 5.1.1: Classification accuracy for models trained with 100, 200, 500, 1000, 2000, 5000, and 10000 simulated images on a logarithmic x-axis. Training and validation accuracy are high while there is a large gap in performance when testing real data.

of simulated images while the test set consists of real images. Accuracy on training and validation dataset are generally high, starting at 74.25 % training accuracy and 68.99 % validation accuracy for 100 examples and increasing steadily as the amount of data becomes larger. With 10000 training images training accuracy rises to 99.99 % and validation accuracy reaches 95.80 %. The test accuracy on the other hand is no higher than 45.80 %. This experiment shows there is a gap between reality and simulation. However, training with larger amounts of synthetic images does increase performance on test dataset. In 5-way classification the chance of guessing the correct class is 20 % and 45.80 % is significantly above that. This shows that while the two domains are certainly different there is some information shared between them.

5.1.2 Transfer Learning to Reality

Experiments In the next experiment models are trained on real data. As outlined in section 4.3 only 2000 labeled training images from reality are available.

Results The results are shown in figure 5.1.2. For 100 data points training accuracy is 81.94 %, validation accuracy is 59.26 % and test accuracy is 47.20 %. They increase continuously as the amount of training data is increased. The highest test accuracy is 90.93 % when using all 2000 labeled images. For small amounts of data the gap between training, validation, and test accuracy is large. This likely happens due to overfitting to small amounts of data. As the amount of data is increased the gap shrinks while all

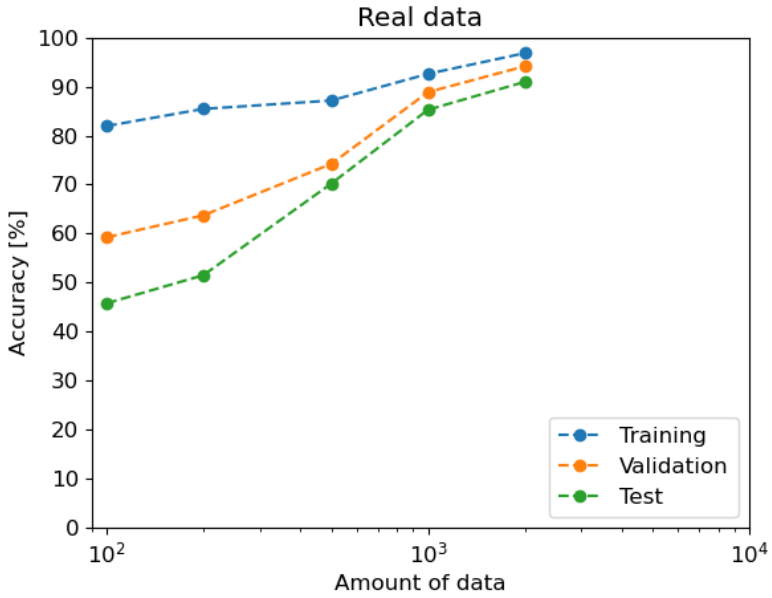


Figure 5.1.2: Classification accuracy for models trained with 100, 200, 500, 1000, and 2000 real images on a logarithmic x-axis. The gap between training, validation, and test accuracy shrinks when using larger amounts of data.

accuracies increase. This reveals that 100 labeled images are insufficient for the problem at hand. Collecting more data is likely to improve performance but time-consuming. The following chapters evaluate ways to improve performance without collecting more data.

5.2 Domain Randomization

This sections describes and evaluates the application of domain randomization (cf. section 3.3) to lane-following.

Experiments Domain randomization is implemented in simulation to generate randomized synthetic images. The EyeSim world file specifies a 550×550 pixel image that is used as floor texture. The Carolo-Cup scene uses a dark gray background with road markings (see figure 4.2.2). To apply domain randomization the background image is randomized during data collection. The gray background is replaced with colored squared resembling a patchwork rug. Figure 5.2.1 shows a screenshot of EyeSim with randomized floor texture.

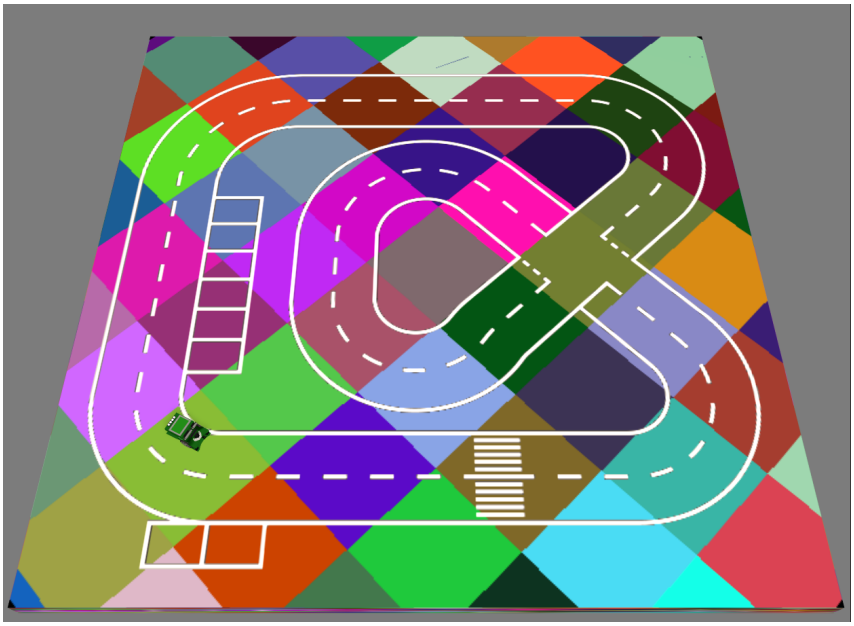


Figure 5.2.1: Bird's-eye view of randomized floor texture in EyeSim.

The following parameters are randomized using a uniform distribution.

- Color of squares
- Size of squares
- Orientation of squares
- Road marking brightness

The color of squares is randomized by choosing a random RGB value. Their size is a random value between 10 and 100 pixels and the random orientation is between 0° and 89°. The brightness of road markings is randomized to 50 % to 100 % of its original value. Figure 5.2.2 shows nine examples of images as they are collected using domain randomization.

Results Figure 5.2.3 shows classification accuracy for models trained with amounts of randomized data ranging from 100 to 10000 examples. Training and validation dataset are collected using domain randomization while the test set consists of real images. For 100 images there is a large gap between training and validation and test accuracy. Training accuracy is at 77.14 % while validation and test accuracy are at 40.00 % and 31.53 %, respectively. This shows that learning a task using randomization is more difficult. Compared to using real data this is low, but this already is an improvement over using simulated data. The improvement becomes larger as more domain randomization data are used. When using 10000 images the test set accuracy rises to 60.80 %, which is an increase by 15 percentage points. This demonstrates that domain randomization is suitable to bridge the reality gap at least partially.

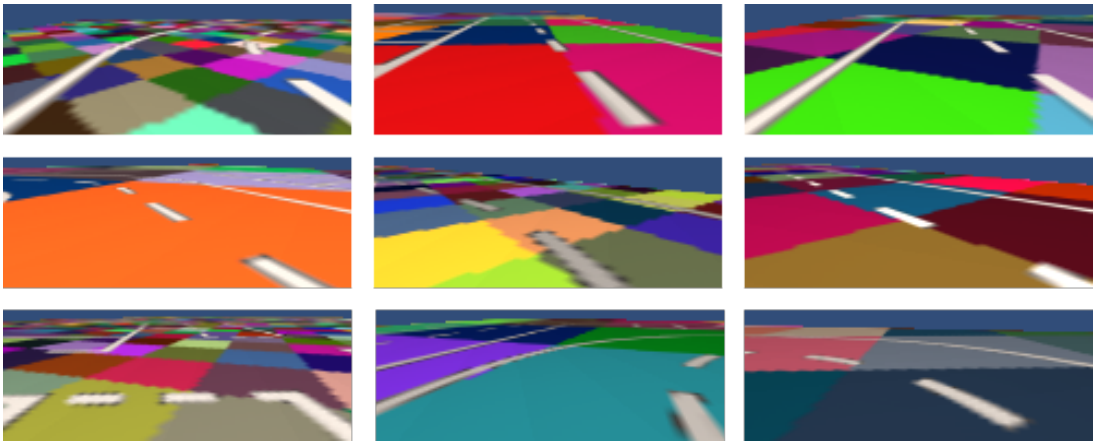


Figure 5.2.2: Nine examples of domain randomization images from the perspective of the vehicle. The floor texture consists of squares of different colors, sizes and orientations.

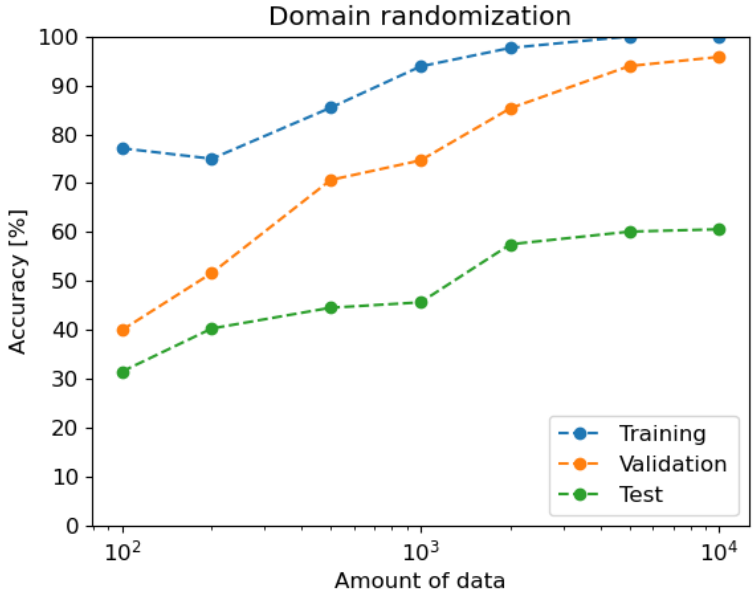


Figure 5.2.3: Classification accuracy for models traine with 100, 200, 500, 1000, 2000, 5000, and 10000 real images on a logarithmic x-axis. Test set accuracy increases as larger amounts of domain randomization data are used, outperforming models trained on regular data from simulation.

5.2.1 Ablation Study

Experiments To evaluate the influence of different randomizations an ablation study is conducted. For each type of randomization a new dataset of 10000 images is collected where only this parameter is varied while others are fixed. One of them varies the size, another one the orientation, and the last one randomizes the brightness of road markings. All datasets except for one use squares of random color. These experiments are compared with using all randomizations combined and with using no randomization at all, i. e. using

synthetic images as described in section 5.1.1. Table 5.1 summarizes the results of the ablation study.

Table 5.1: Results of the domain randomization ablation study. "None" means that no randomization is employed, i. e. data from simulation as discussed in section 5.1.1. Combined means that all randomizations are applied as discussed in the previous experiment.

Randomization	Accuracy
None	45.80 %
Color of squares	53.20 %
Color & size of squares	54.40 %
Color & orientation of squares	**59.60 %**
Road marking brightness	48.40 %
Color & road marking brightness	55.80 %
Combined	**60.80 %**

Results Replacing the gray background texture with colored squares achieves 53.20 % accuracy, a 7.4 percentage point gain over un-randomized synthetic images. Adding squares of random size and color yields an additional improvement by 1.2 percentage points over only using random colors. Using a random orientation for colored squares has the largest positive impact on performance. With an accuracy of 59.60 % this is almost as good as using all randomizations. Using a gray background with randomized brightness of road markings results in a slight increase by 2.6 percentage points to 48.40 %. Finally, using colored squares in conjunction with random road marking brightness results in 55.80 % accuracy.

The experiments show that using domain randomization with colored squares of random orientation offers the largest benefit. Even using colored squares of fixed orientation results in a large improvement. Randomizing the size of squares only yields a small improvement compared to squares of fixed size. The same is true for varying the road marking brightness. In conclusion, all of the randomization parameters have an impact on performance, with colored squares with random orientation having the largest.

5.3 Domain Adaptation

This section describes experiments conducted using domain adaptation and discusses results. Domain adaptation is evaluated in two different settings: Unsupervised and semi-supervised. In unsupervised domain adaptation no labeled target domain data is provided. Semi-supervised domain adaptation makes use of a small amount of labeled target domain data. Before any experiments are conducted loss weights λ_w and λ_d of equation (3.4.7) are optimized.

5.3.1 Optimization of Loss Weights

Domain adaptation using weight regularization and domain confusion has two major hyperparameters: weight regularization weight λ_w and domain confusion weight λ_d. The former controls to what extend the weight regularization loss is reflected in the total loss. The latter determines how much to penalize the difference between source and target domain feature representation. Note that the weight regularization loss is different from weight decay described in section 4.4.3. It is applied to the difference of weights between models trained on source and target data.

Hyperparameter optimization is a vast field of research and this work cannot possibly cover more than a fraction of it. The two simplest algorithms for hyperparameter optimization are grid search and random search. Grid search evaluates a finite set of hyperparameter values spread uniformly or logarithmic in a given interval. The best-performing combination of hyperparameters on a validation set is chosen for further experiments. The cost of grid search grows exponentially with the number of hyperparameter in consideration. Random search selects random hyperparameter values in a given range and evaluates them on a validation set. In many applications random search is more efficient than grid search, [5]. Thus, domain adaptation weights are optimized using random search.

Experiments Random search using log-uniform sampling is implemented to find good values for domain adaptation weights. Both λ_w and λ_d are sampled in $[10^{-3}, 10^3]$. To perform log-uniform sapmling uniform sampling is exponentiated. In total 28 trials are done, each with a new sample of weights. In the interest of short training times networks are trained on a subset of training data; 2000 images are used, 1000 labeled images from simulation and 1000 unlabeled images from reality.

Results Figure 5.3.1 shows results of random search experiments. The accuracy is computed on the validation set consisting of real images to quantify the capability to generalize. It varies greatly throughout the experiments, ranging from 23.40 % to 48.20 %. The highest accuracy is achieved for $\lambda_w = 2.434$ and $\lambda_d = 0.9987$. In general, larger values for λ_w perform better than small values, although there is a high variation. On the other hand, large values λ_d are detrimental to validation accuracy. Values account 1 perform well, with an additional cluster of values around $3 \cdot 10^{-3}$. Random search for optimal domain adaptation weights serves as an ablation study. The fact that optimal weights are far from the lower bound demonstrates that both of them have a contribution.

The highest accuracy at 48.20 % is only slightly higher than the accuracy achieved when training on synthetic data only, but domain adaptation is expected to perform better when trained on the full dataset. More on that in the following sections.

5.3.2 Unsupervised Domain Adaptation

Experiments Unsupervised domain adaptation forbids the usage of any labeled target domain data. Thus, 10000 labeled images from simulation and 10000 unlabeled images

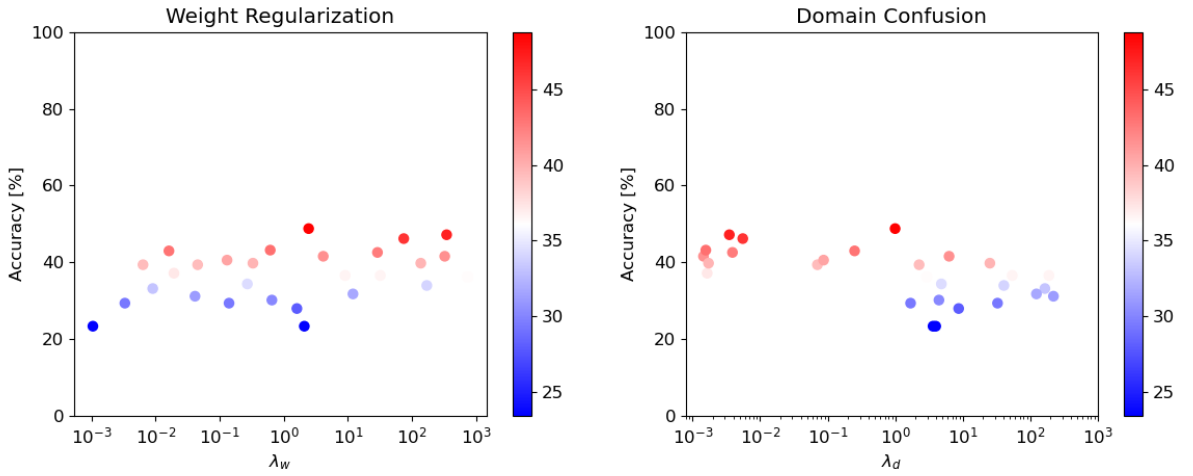


Figure 5.3.1: Random search for domain adaptation loss weights. Each figure depicts a side-view on the axis of interest of a three-dimensional scatter plot. The color of dots indicates levels of accuracy. A blue dot represent low accuracy while red is used for large values.

from reality are used. Generalization to target domain is achieved through encouraging domain-invariant features and parameter-tying. All experiments are conducted using the hyperparameters discussed in section 5.3.1. Domain adaptation with weight regularization requires two networks at training time: One network for source and one network for target domain. The source domain network is pre-trained on synthetic data. Target domain network weights are tied—but not shared—with source domain network weights as described in section 3.4.1.

Two different experiments are conducted. In the first experiment synthetic data from simulation is used as is. The second experiment uses domain randomization data for the source domain. This setting is called domain adaptation and randomization.

Results Table 5.2 summarizes unsupervised domain adaptation results. Unsupervised domain adaptation achieves 63.80 % accuracy on the real-world test set. This is a significant improvement over using data from simulation only. Performance can be improved further by domain adaptation and randomization. This yields an accuracy of 64.20 %.

5.3.3 Semi-Supervised Domain Adaptation

Experiments This section discusses semi-supervised domain adaptation applied to the lane-following problem. Recall that semi-supervised means that only part of the images are labeled. Two semi-supervised experiments are conducted: one with 100 labeled real world images and one with 2000 labeled real world images. The rest of the real world images for a total of 10000 are unlabeled. Thus, no classification loss can be computed for them. However, they contribute to weight regularization and domain confusion loss. In

addition to 10000 (partially labeled) real world images 10000 labeled synthetic images are used. Domain adaptation is compared to training on few real images only, to training on real images in combination with simulation images without domain adaptation, and to training on domain randomization plus a few labeled real images.

Results Table 5.2 lists semi-supervised domain adaptation results. Semi-supervised domain adaptation using 100 labeled images from reality achieves a test set accuracy of 65.80 %. This is a 15.60 percentage point improvement compared to training on simulated and real images without adding domain adaptation loss terms. Notably, training on synthetic images in combination with real images improves performance compared to training on few real images only—even without domain adaptation. Adding synthetic images acts as a regularization and helps prevent overfitting to a small dataset. Semi-supervised domain adaptation with randomization further improves test set accuracy to 67.20 %. The higher diversity of a randomized dataset appears to be favorable for the emergence of domain invariant features through domain adaptation.

When increasing the number of labeled real world images to 2000 all approaches show significant performance improvement. All of them achieve upwards of 90 %, with domain randomization and real world images yielding the best result. At this point domain adaptation does not offer that much of a benefit since 2000 images are sufficient to learn the task regardless.

Table 5.2: Comparison of semi-supervised and unsupervised domain adaptation. Five different approaches are compared: training on labeled real data only, training on synthetic data and labeled real data, training on domain randomization (DR) and labeled real data, domain adaptation (DA), and domain adaptation with domain randomization (DAR). Note that "Sim + real" and "DR + real" do not actually use real data in the unsupervised setting. The best results are marked bold.

	Unsupervised	Semi-supervised	
# real labels		100	2000
Real only	-	47.20 %	90.20 %
Sim + real	45.80 %	50.20 %	90.40 %
DR + real	60.80 %	62.00 %	**90.80 %**
DA	63.80 %	65.80 %	90.20 %
DAR	**64.20 %**	**67.20 %**	90.60 %

5.4 Meta-Learning

This section describes and discusses results of meta-learning applied to lane-following. MAML and its first order approximation FOMAML are implemented as described in section 3.5.3.

Experiments Since MAML is model-agnostic the same network architecture as in the other experiments can be used. Networks are first meta-trained on a subset of the ImageNet dataset [1]. The subset contains 200 classes with 500 training images and 50 validation images each for a total of 10000 images. The classes range from animals and plants to humans in various contexts to inanimate objects and scenes. Networks are meta-trained for 60000 iterations as done by Finn et al, [8].

Meta-learning is evaluated in a 1-shot, 5-shot, and 20-shot setting, where k-shot learning means that for each class k images are present. Section 4.3 established that there a 5 different classes. Thus, in 20-shot learning a total of 100 images are used in each iteration. During meta-training 5 classes are sampled from all available tiny ImageNet classes. For each classes k images are sampled. The network is trained on those as described in section 3.5.3. After a number of epochs the network is fine-tuned on the lane-following task using k real world images per class.

Results Table 5.3 lists results for meta-learning applied to lane-following. Three different settings are compared: 1-shot, 5-shot, and 20-shot learning. Accuracy is computed on the real world test set (cf. table 4.1). 1-shot learning achieves an accuracy of 43.40 %, 5-shot learning achieved 59.20 %, and 20-shot learning reuslts in 75.20 % accuracy. The more training images are used, the higher the accuracy, as one would expect. Notably, the improvement is large even in 1-shot learning. In this setting, meta-learning greatly outperforms training on data without meta-learning.

5.4.1 First-Order Approximation

Experiments Experiments conducted with FOMAML are the same as the previous ones, except that the first-order approximation as described in section 3.5.3 is used. Again, 1-shot, 5-shot, and 20-shot learning are compared to evaluate the influence of the amount of data. Finn et al. observe that an approximation shows almost no degradation in accuracy but at the same time speed up training significantly. This experiment seeks to verify that.

Results Results of first-order meta-learning are summarized in table 5.3 along meta-learning without approximation. For each of the three settings FOMAML exhibits about 2 to 4 percentage point loss in accuracy. In 20-shot learning 73.60 % accuracy is achieved. FOMAML achieves similar results to MAML but training times are reduced since no second-order derivatives are required.

5.5 Visualization of Road Feature Detection

To demonstrate that networks are actually learning a meaningful representation to predict steering angle the network activation are visualized. Starting at the last layer, network

[1] Tiny ImageNet available under `https://tiny-imagenet.herokuapp.com/`.

Table 5.3: Comparison of MAML and its first-order approximation. Meta-learning is evaluated in three different settings: 1-shot, 5-shot, and 20-shot learning.

	1-shot	5-shot	20-shot
Real only	22.80 %	30.40 %	47.20 %
MAML	43.40 %	59.20 %	**75.20 %**
FOMAML	40.20 %	55.80 %	73.60 %

activations are visualized by averaging the activation of the feature maps, scaling them up to the previous layer's activations, multiplying them together, and repeating this procedure until the input layer is reached, [45]. A deconvolution is used to scale feature maps up. The accumulated activations are thresholded and overlain onto the input image in bright green. The procedure is summarized in algorithm 3.

Algorithm 3 Visualize network activations

Require: Trained network $f(\theta^*)$, input image x
activations $\leftarrow f(x; \theta^*)$
accumulated $\leftarrow 1$
for activation **in** activations **do**
    activation $\leftarrow$ AVERAGE(activation)
    activation $\leftarrow$ DECONVOLUTION(activation) ▷ Scale up to next layer
    accumulated $\leftarrow$ accumulated $\cdot$ activation
end for
accumulated $\leftarrow$ THRESHOLD(accumulated)
Overlay accumulated activation on x

Figure 5.5.1 shows an four example images of visualized network activations. The network in question is trained with the MAML approach is evaluated. Evidently, the network learns to detect road markings in order to predict steering, especially those that are in the foreground. Road markings in the foreground are more informative about the course of the road than those in the background. The networks visibly struggles to make sense of the crosswalk. A detailed visualization of different networks can be found in appendix A.

5.6 Comparison

This section presents results of domain randomization, domain adaptation, and meta-learning applied to lane-following evaluated on the real world Carolo-Cup track. The best-performing ones of each category are compared in a 20-shot and 0-shot setting. A photo of the track is shown in figure 4.3.2.

Inspired by Bojarski et al. an autonomy rating (AR) is used to measure the ability to follow a lane by counting the number of interventions required, [2]. An intervention is required when the vehicle would depart from its course and leave the road otherwise. They

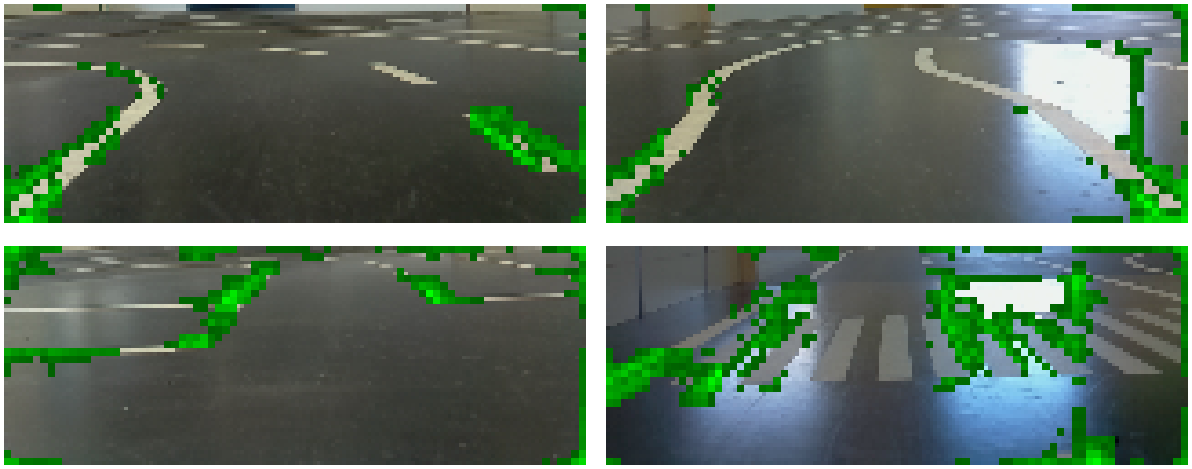


Figure 5.5.1: Visualization of Network Activations. Bright green pixels indicate high activations which in turn are responsible for the network output.

define AR as follows.

$$\mathrm{AR} = \left(1 - \frac{N_{interventions} \cdot 6\,\mathrm{s}}{t[\mathrm{s}]}\right) \cdot 100\,\% \tag{5.6.1}$$

The AR is given by multiplying the number of interventions by 6 s, diving by the elapsed time, and subtracting the result from 1. In other words, one intervention every minute equals a 10 percentage point loss of autonomy. At 10 or more interventions every minute the AR is clipped to zero. Here, a discrete-time autonomy rating is proposed.

$$\mathrm{AR} = \left(1 - \frac{N_{interventions} \cdot 100}{N_{steps}}\right) \cdot 100\,\% \tag{5.6.2}$$

Instead of dividing the number of human interventions by time it is divided by the number of total time steps and multiplied by 100. Thus, one intervention every 1000 steps causes a 10 percentage point loss of autonomy. For reference, the completion of one lap of the Carolo-Cup track takes about 1000 steps. A human intervention is required if the EyeBot leaves its lane.

5.6.1 20-shot Learning

Experiments Five different networks trained on 100 labeled real world images compared, i. e. in a 20-shot setting. A baseline is given by training using a network pre-trained on ImageNet and fine-tune it on the 100 images. It is compared to semi-supervised domain adaptation with and without domain randomization. These two are contrasted with meta-learning and its first-order approximation. Autonomy rating is computed over one completion of the course in clockwise and one completion in counterclockwise direction for each approach.

Results Table 5.4 reports accuracy on the test set and autonomy rating on the test course for 20-shot learning. Meta-learning achieves the highest accuracy at 75.20 %. All domain adaptation, randomization, and meta-learning exhibit significantly better performance than the baseline. The autonomy rating of all approaches is well above above 70 %. Semi-supervised domain adaptation with randomization achieves the highest rating at 78.11 %. It slightly outperforms meta-learning.

The baseline achieves an AR of 58.42 % which is surprisingly high given that its accuracy is far lower than that of all other approaches. This can be explained by the fact that to it is not necessary to always predict the correct outcome to complete the course. However, this model exhibits some perplexity at the crosswalk (see figure 4.3.2). Thus could be caused by the fact that the crosswalk is barely represented in the training data and might be missing or underrepresented in the 100 labeled real world images. Domain adaptation is affected less by the crosswalk since it was trained on 10000 synthetic images from simulation, among them many with crosswalks.

Table 5.4: Comparison of domain adaptation and meta-learning in a 20-shot setting. Accuracy on the test set and autonomy rating on real world Carolo-Cup track are compared. The best results are marked bold.

	Accuracy	Autonomy Rating
Real only	47.20 %	58.42 %
Semi-supervised DA	65.80 %	73.49 %
Semi-supervised DAR	67.20 %	**78.11 %**
MAML	**75.20 %**	77.01 %
FOMAML	73.60 %	75.39 %

Network activation for networks used here are visualized in appendix A.1 and appendices A.4 to A.7. It can be seen that better-performing networks generally cause fewer spurious activations, i. e. activations that ostensibly do not represent meaningful road features. Such activations sometimes appear to occur at the border of the image or in reflections. They are symptoms of overfitting. The fact that MAML and semi-supervised domain adaptation with randomization have less spurious activations goes on to show that the help bridge the gap between simulation and reality.

5.6.2 0-shot Learning

Experiments Four approaches are evaluated in a 0-shot setting. Domain randomization and unsupervised domain adaptation with and without randomization are compared against a baseline that is trained on synthetic data only. Meta-learning is not application to 0-shot learning since it requires at least a small amount of labeled target domain data. Likewise, fine-tuning on real data is not possible.

Table 5.5: Comparison of domain adaptation and meta-learning in a 0-shot setting. Accuracy on the test set and autonomy rating on real world Carolo-Cup track are compared. The best results are marked bold.

	Accuracy	Autonomy Rating
Sim only	45.80 %	54.95 %
DR	60.60 %	68.53 %
Unsupervised DA	63.80 %	67.77 %
Unsupervised DAR	**64.20 %**	**71.61 %**

Results Table 5.5 reports accuracy and autonomy rating for 20-shot learning. Relying only on synthetic data with no means of adapting achieves surprisingly good results. While the accuracy is only 45.80 % the AR is 54.95 %. This means that about 5 interventions are required to complete the track. In general, most interventions are required at the intersection (see figure 4.3.2). Unsupervised domain adaptation with randomization exhibits the best performance in 0-shot learning at 71.61 % autonomy rating. Domain randomization stands out since through its simplicity and good relative results. Without any additional loss terms or other measures it is able to generalize from randomized synthetic images to reality. However, it often goes zig-zag like in straight segments of the course. Note that it is possible to follow the lane with 5-way classification even though the curve radii vary between 25 cm to 50 cm.

Network activation for networks used here are visualized in appendices A.2 to A.5 Again, some networks exhibit more spurious activations than others. Notably, unsupervised domain adaptation is missing out on many road markings. Adding domain randomization alleviates this circumstance.

Chapter 6

Conclusion and Outlook

This chapter summarizes results, draws a conclusion and gives an outlook to future work.

6.1 Conclusion

This thesis set out to bridge the gap between simulation and reality. In Chapter 3 an extensive survey of existing approaches is given. Three different types of approaches are compared: Domain randomization, domain adaptation, and meta-learning. For each type an approach is selected and is considered for implementation and evaluation in the proceeding Chapters 4 to 5. All implemented algorithms are evaluated on a held-out test dataset and on a real-world test course using a model car.

The results of the previous chapter have demonstrated that it is possible to develop a deep learning system for lane-following in the absence of labeled training data in target domain. When such data is available results can be improved further. Among all approaches compared in this thesis model-agnostic meta learning achieved the highest accuracy on the test set at 75.20 %. In lane-following experiments on a test track semi-supervised domain adaptation and randomization achieved the highest autonomy rating at 78.11 %. Visualizations of network activations have shown that networks learn to detect road markings in order to predict steering. Models have an average runtime of 66 ms which is well below the target of 100 ms.

Domain randomization achieved good results with relatively little effort. It significantly outperformed synthetic data without randomization and works in a 0-shot setting. Also, it is agnostic of the target domain. Domain adaptation achieved slightly better results. However, domain adaptation is tailored to a specific target domain. Using domain randomized data for domain adaptation has been shown to yield even better results. Finally, meta-learning performed best on the test set at the cost of extensive meta-training. Its first-order approximation reduced meta-training times while achieving similar results.

While few-shot and zero-shot learning for lane-following is possible none of the approaches presented here can surpass training on a large labeled dataset of real world data. For

applications where system failures incur large costs collecting such datasets is mandatory. Still, domain randomization, adaptation, and meta-learning can be beneficial.

6.2 Outlook

Among domain randomization and adaptation approaches those based on GANs are most promising. They have been discarded in this thesis on the basis of computational costs and to limit the scope, but they show a lot of potential and should be considered for future research. Using GANs, more realistic and diverse synthetic training data can be generated, improving accuracy on the test set.

A recently proposed extension of MAML combines meta-learning with online learning, [46]. Online learning trains on sequences of data, updating the model after each step. Experience from previous tasks can be utilized to update model parameters in the current step. Online meta-learning achieves better results on a number of benchmark tasks than MAML.

All experiments in this thesis were based on the same model architecture: MobileNetV2. While the runtime requirements have been met using inverted residual blocks with linear bottlenecks and TF Lite conversion the model might be larger than necessary. The conversion with TF Lite happens after training. It is conceivable to optimize the model architecture at training time for reduced size, memory usage, and inference time. A number of approaches to achieve this have been proposed, [47]. For example, model pruning seeks to remove redundant parameters, thereby reducing model size and number of operations. Knowledge distillation aims to train a compact model to reproduce the output of a larger model. Furthermore, the number of layers and feature map sizes can be optimized using a genetic algorithm.

MobileNetV2 is a feedforward network without recurrence. Thus, it fails to capture temporal relationships between consecutive data. However, consecutive input-output pairs are related to each other. Introducing recurrent units that are able to model temporal relations could lead to better results. For example, one could add LSTM cells between the final feature vector and the output vector, [48]. This has almost no impact in inference time since LSTMs require few operations compared to the existing architecture. However, some adjustments in the data and training pipeline are necessary.

Accuracy and autonomy rating of the best-performing models presented here are about 75 %. While this is sufficient to follow the lane most of the time there is room for improvement. One possibility to achieve better results is to take a trained model, deploy in on the real world track, and fine-tune it using reinforcement learning. This is a common approach in robotics, where reinforcement learning takes countless iterations and is often destructive. The trained model exhibits the expected behavior except for a few cases. In terms of parameter space, the model parameters are close to optimal and only a few adjustment are necessary.

A different approach to improve results is using semi-supervised learning to automatically label data. Semi-supervised learning can be used to make predictions on unlabeled data and then use said data and the predicted labels to train a better model. This is called

pseudo-labeling or self-training. Recently, a new semi-supervised learning algorithm called FixMatch has been proposed that outperforms other approaches and requires minimal amounts of data, [49]. FixMatch uses consistency regularization and strongly augmented input data to enforce that the model makes similar predictions for strongly perturbed images. It can be used to label large amounts of unlabeled data. Even if some of the labels are erroneous, training a new predictor on these artificial labels can improve performance on the target domain task.

This thesis presented three approaches to solve a simple lane-following task on a small test course, which is more of a toy problem. The next step would be to add tasks such as object detection for other vehicles and traffic signs as well as obstacle avoidance. Other tasks such as lane-switching, overtaking, and behavior at intersections and crossroads are topics of interest. Furthermore, the extension to new and more diverse environments is a necessary next step.

References

[1] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving", *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[2] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars", *arXiv preprint arXiv:1604.07316*, 2016.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning", *arXiv preprint arXiv:1312.5602*, 2013.

[4] 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", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 23–30.

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[6] J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. A. Efros, and T. Darrell, "Cycada: Cycle-consistent adversarial domain adaptation", *arXiv preprint arXiv:1711.03213*, 2017.

[7] A. Rozantsev, M. Salzmann, and P. Fua, "Beyond sharing weights for deep domain adaptation", *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 4, pp. 801–814, 2018.

[8] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks", in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1126–1135.

[9] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications", *arXiv preprint arXiv:1704.04861*, 2017.

[10] L. Sifre and S. Mallat, "Rigid-motion scattering for image classification", *Ph. D. thesis*, 2014.

[11] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] C. Olah, *Understanding lstm networks*, 2015. [Online]. Available: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[15] S. J. Pan and Q. Yang, "A survey on transfer learning", *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, pp. 1345–1359, 2009.

[16] R. Caruana, "Multitask learning", *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[17] B. Sun and K. Saenko, "Deep coral: Correlation alignment for deep domain adaptation", in *European Conference on Computer Vision*, Springer, 2016, pp. 443–450.

[18] S. Ruder, "An overview of multi-task learning in deep neural networks", *arXiv preprint arXiv:1706.05098*, 2017.

[19] H. Nam and B. Han, "Learning multi-domain convolutional neural networks for visual tracking", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4293–4302.

[20] Y. Ganin and V. Lempitsky, "Unsupervised domain adaptation by backpropagation", *arXiv preprint arXiv:1409.7495*, 2014.

[21] G. Kang, L. Zheng, Y. Yan, and Y. Yang, "Deep adversarial attention alignment for unsupervised domain adaptation: The benefit of target expectation maximization", in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 401–416.

[22] S. Zakharov, W. Kehl, and S. Ilic, "Deceptionnet: Network-driven domain randomization", *arXiv preprint arXiv:1904.02750*, 2019.

[23] M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, *et al.*, "Learning dexterous in-hand manipulation", *arXiv preprint arXiv:1808.00177*, 2018.

[24] A. Prakash, S. Boochoon, M. Brophy, D. Acuna, E. Cameracci, G. State, O. Shapira, and S. Birchfield, "Structured domain randomization: Bridging the reality gap by context-aware synthetic data", in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 7249–7255.

[25] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox, "Closing the sim-to-real loop: Adapting simulation randomization with real world experience", in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, pp. 8973–8979.

[26] N. Ruiz, S. Schulter, and M. Chandraker, "Learning to simulate", *arXiv preprint arXiv:1810.02513*, 2018.

[27] B. Mehta, M. Diaz, F. Golemo, C. J. Pal, and L. Paull, "Active domain randomization", *arXiv preprint arXiv:1904.04762*, 2019.

[28] S. James, P. Wohlhart, M. Kalakrishnan, D. Kalashnikov, A. Irpan, J. Ibarz, S. Levine, R. Hadsell, and K. Bousmalis, "Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 12 627–12 637.

[29] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan, "Unsupervised pixel-level domain adaptation with generative adversarial networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3722–3731.

[30] E. Tzeng, J. Hoffman, N. Zhang, K. Saenko, and T. Darrell, "Deep domain confusion: Maximizing for domain invariance", *arXiv preprint arXiv:1412.3474*, 2014.

[31] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a" siamese" time delay neural network", in *Advances in neural information processing systems*, 1994, pp. 737–744.

[32] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition", in *ICML deep learning workshop*, vol. 2, 2015.

[33] L. Weng, *Meta-learning: Learning to learn fast*, 2018. [Online]. Available: `https://lilianweng.github.io/lil-log/2018/11/30/meta-learning.html`.

[34] G. Koch, "Siamese neural networks for one-shot image recognition", Master's Thesis, University of Toronto, 2015.

[35] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks", *IEEE Transactions on Evolutionary Computation*, 2019.

[36] O. Vinyals, C. Blundell, T. Lillicrap, D. Wierstra, *et al.*, "Matching networks for one shot learning", in *Advances in neural information processing systems*, 2016, pp. 3630–3638.

[37] A. Santoro, S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap, "Meta-learning with memory-augmented neural networks", in *International conference on machine learning*, 2016, pp. 1842–1850.

[38] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines", *arXiv preprint arXiv:1410.5401*, 2014.

[39] S. Ravi and H. Larochelle, "Optimization as a model for few-shot learning", 2016.

[40] T. Bräunl, M. Pham, F. Hidalgo, R. Keat, and H. Wahyu, *Eyebot 7 user guide*, 2017. [Online]. Available: `https://robotics.ee.uwa.edu.au/eyebot/EyeBot7-UserGuide.pdf`.

[41] E. V. Team, *Eyesim vr user's manual*, 2017. [Online]. Available: `https://robotics.ee.uwa.edu.au/eyesim/ftp/EyeSim-UserManual.pdf`.

[42] H. Xu, Y. Gao, F. Yu, and T. Darrell, "End-to-end learning of driving models from large-scale video datasets", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2174–2182.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", *arXiv preprint arXiv:1412.6980*, 2014.

[44] L. N. Smith, "Cyclical learning rates for training neural networks", in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, IEEE, 2017, pp. 464–472.

[45] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car", *arXiv preprint arXiv:1704.07911*, 2017.

[46] C. Finn, A. Rajeswaran, S. Kakade, and S. Levine, "Online meta-learning", *arXiv preprint arXiv:1902.08438*, 2019.

[47] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks", *arXiv preprint arXiv:1710.09282*, 2017.

[48] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark", *arXiv preprint arXiv:1609.08675*, 2016.

[49] K. Sohn, D. Berthelot, C.-L. Li, Z. Zhang, N. Carlini, E. D. Cubuk, A. Kurakin, H. Zhang, and C. Raffel, "Fixmatch: Simplifying semi-supervised learning with consistency and confidence", *arXiv preprint arXiv:2001.07685*, 2020.

Unless stated otherwise all URLs were last retrieved on 22.05.2020.

Appendix A

Detailed Visualization of Network Activations

This chapter contains detailed visualizations of network activations overlain on 10 example images in bright green. The following networks are considered: A network trained on 100 real images (appendix A.1), a network trained on 10000 simulated images (appendix A.2), a network trained on 10000 randomized images (appendix A.3), a network trained using unsupervised domain adaptation (appendix A.4), a network trained using unsupervised domain adaptation and randomization (appendix A.5), a network fine-tuned on 100 images using MAML (appendix A.6), and a network fine-tuned on 100 images using FOMAML (appendix A.7). All visualizations are created with algorithm 3.

A.1 Real Only

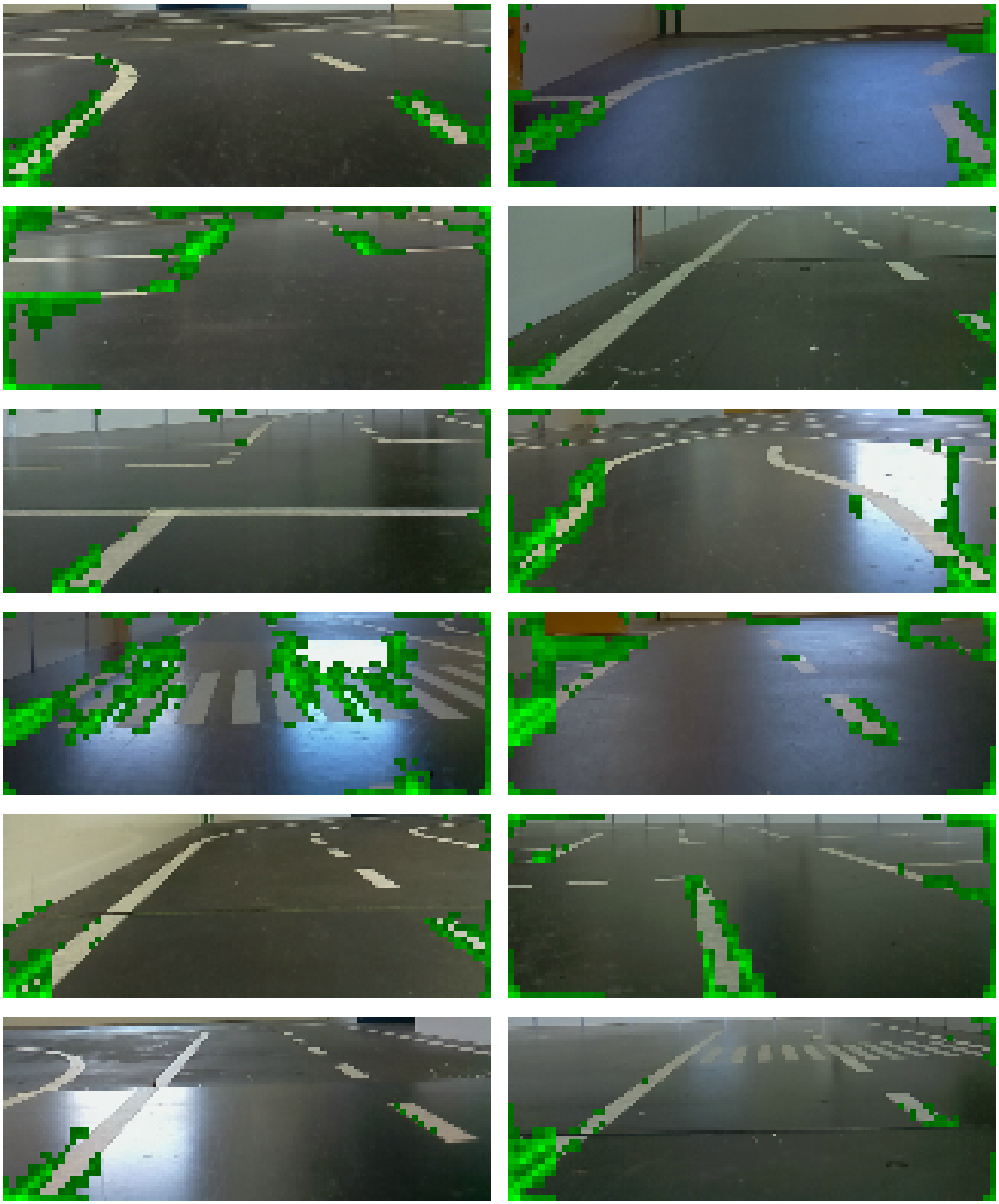


Figure A.1.1: Real Only

A.2 Sim Only

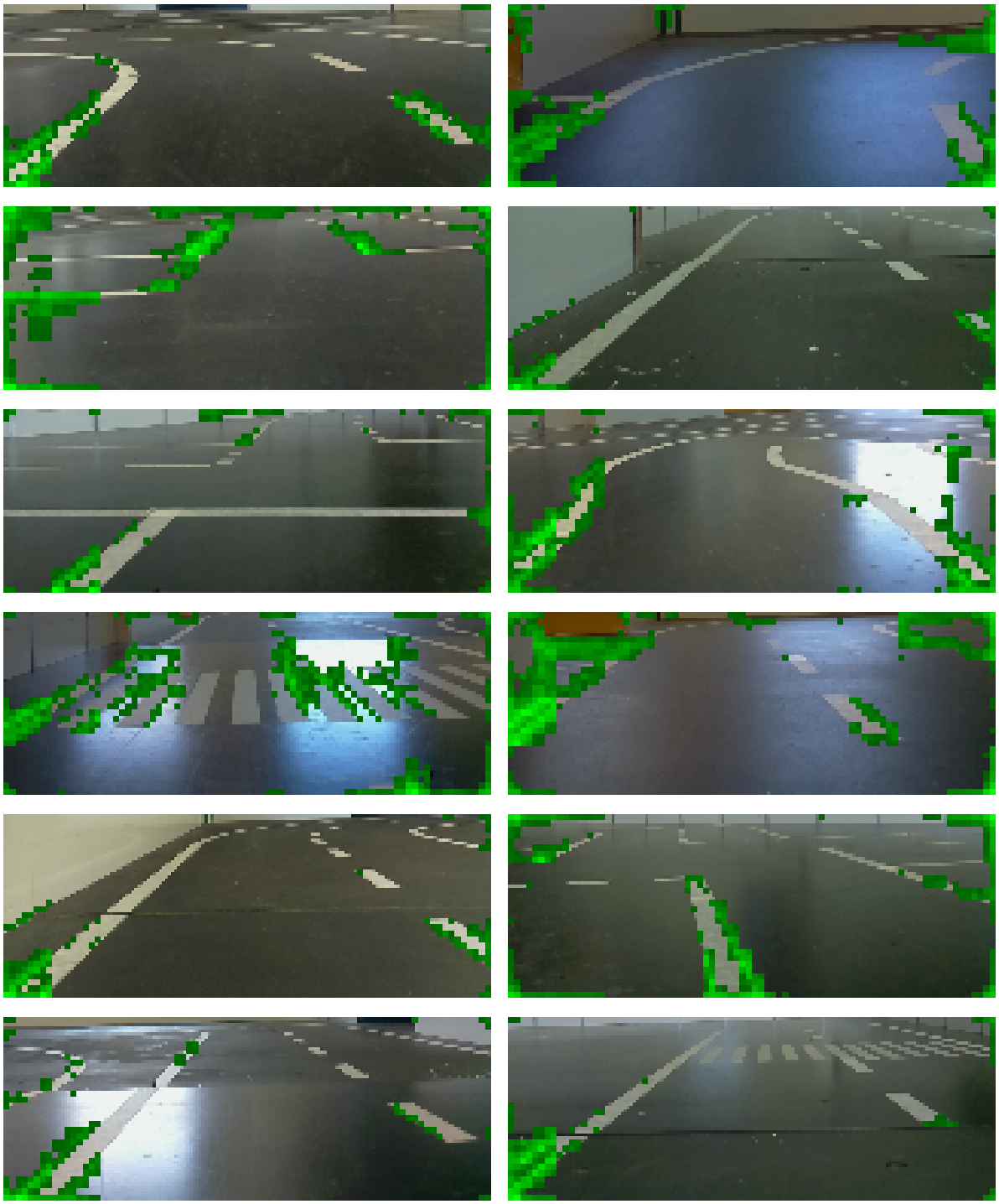


Figure A.2.1: Sim Only

A.3 Domain Randomization

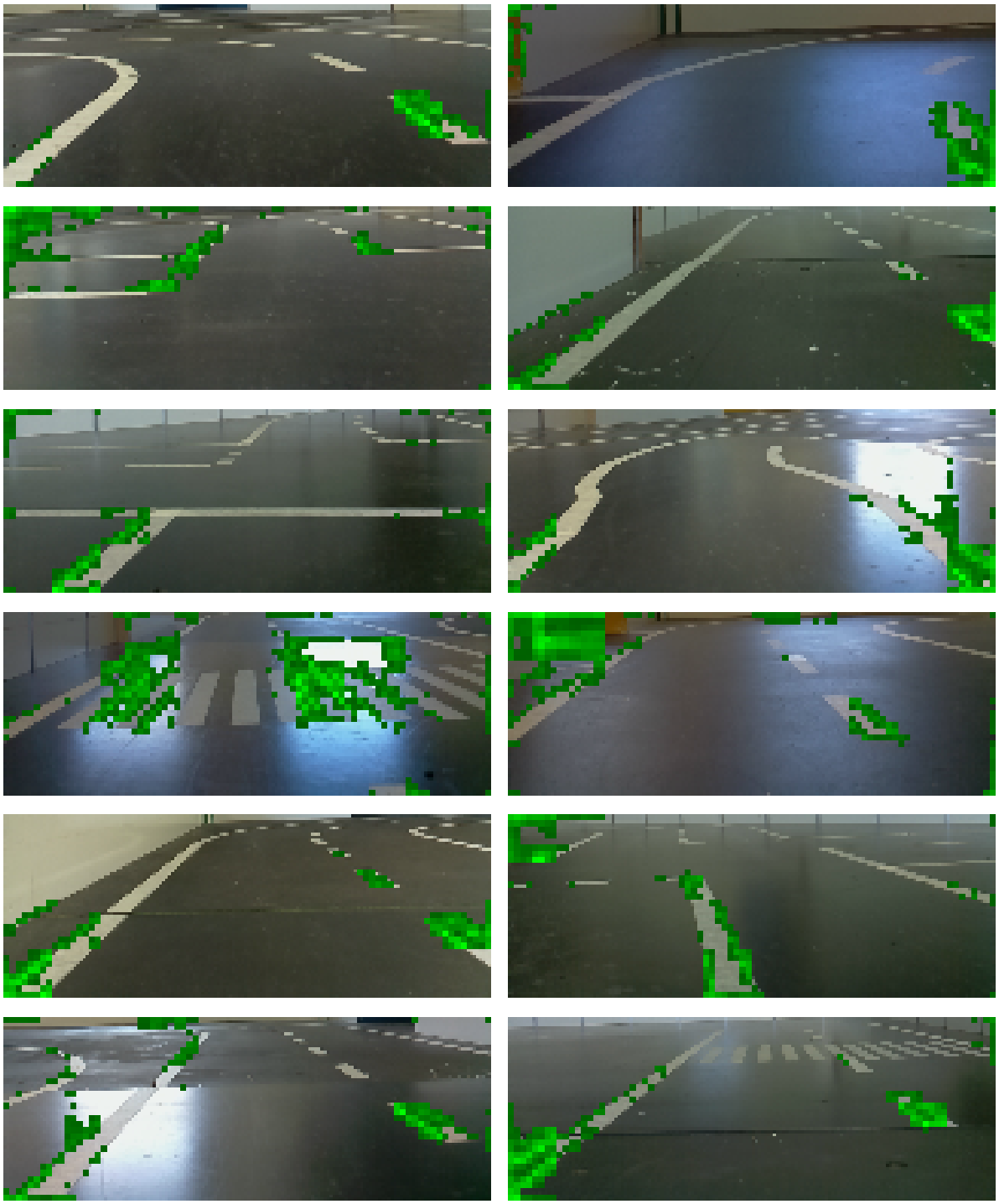


Figure A.3.1: Domain Randomization

A.4 Domain Adaptation

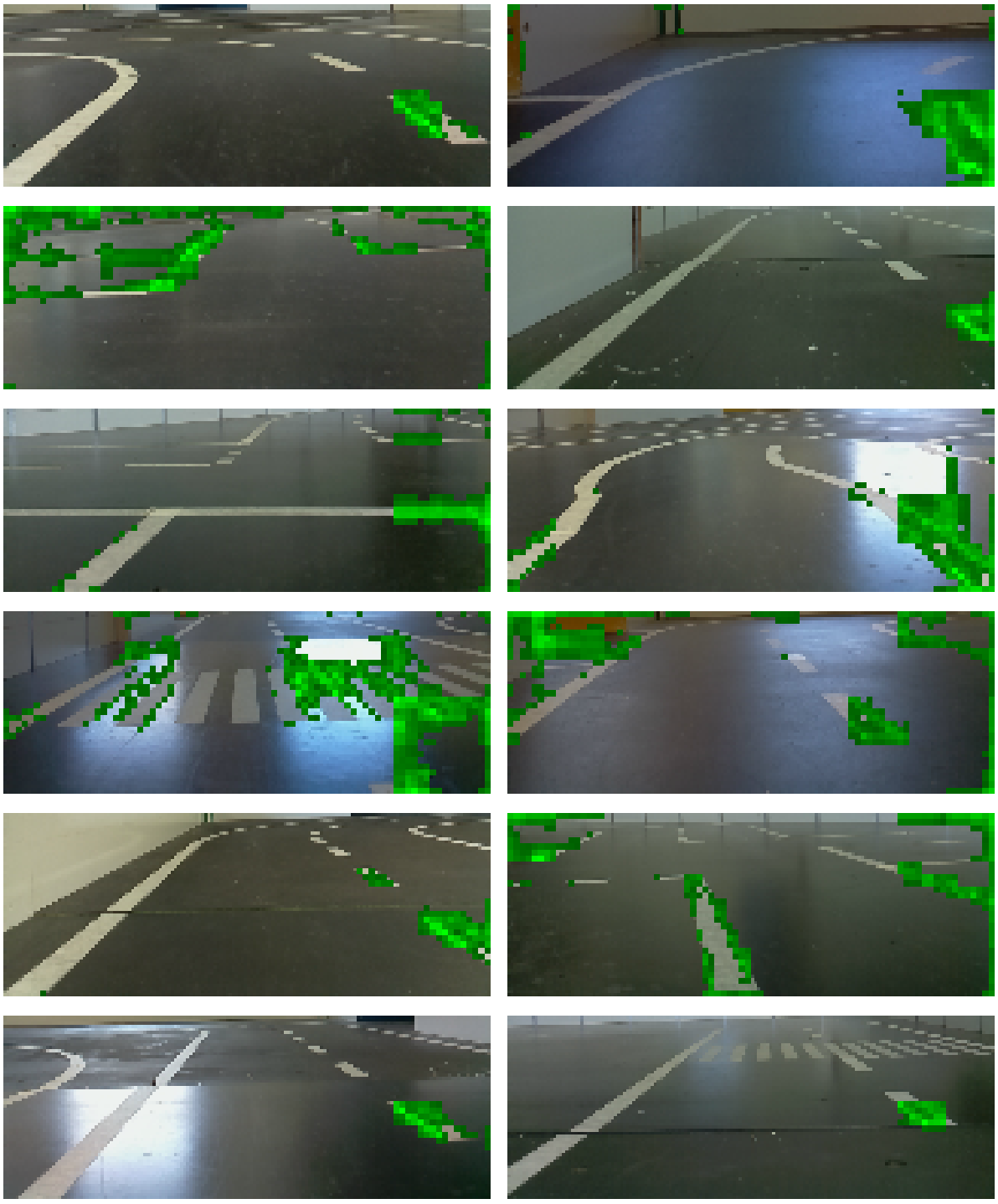


Figure A.4.1: Domain Adaptation

A.5 Domain Adaptation and Randomization

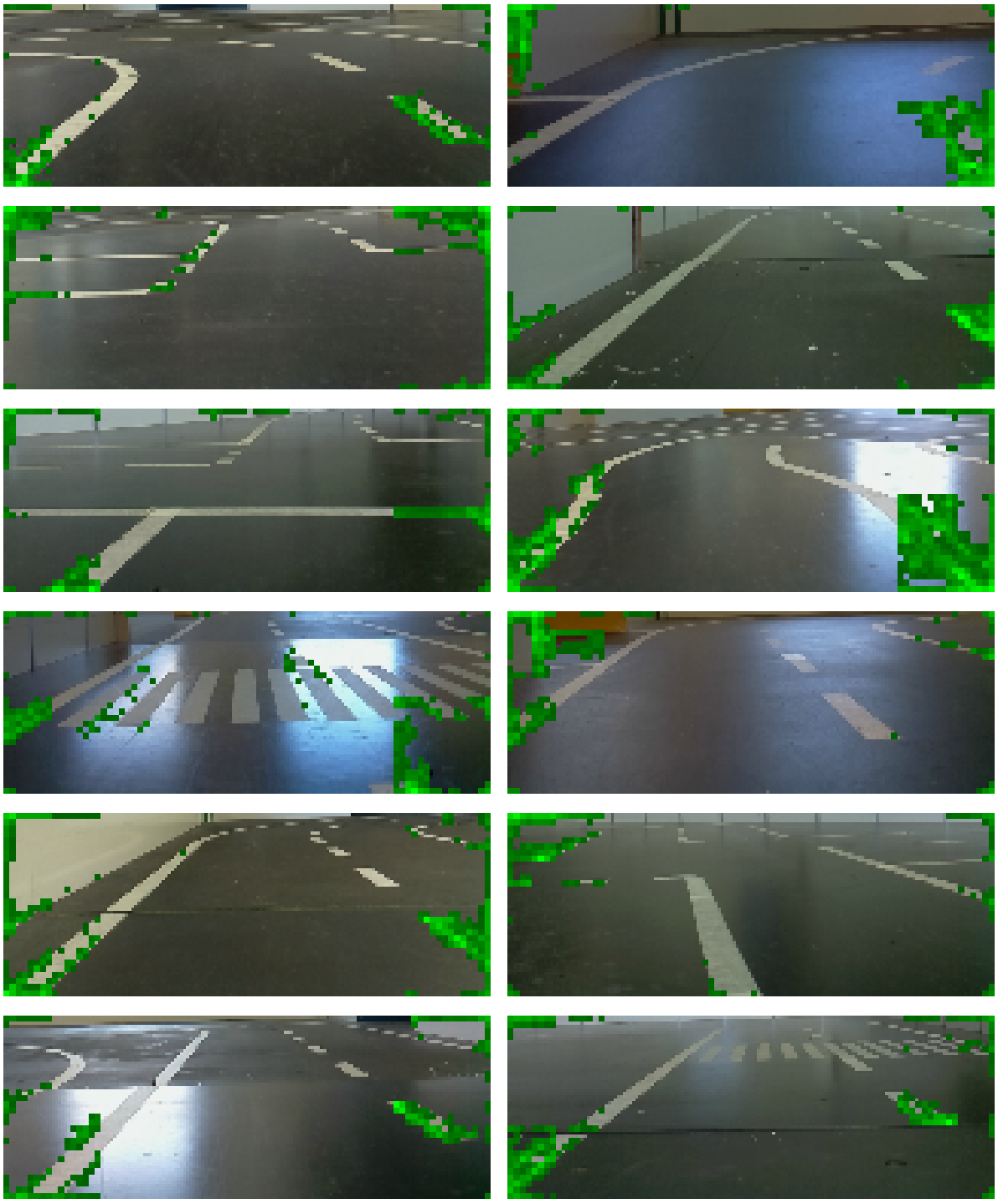


Figure A.5.1: Domain Adaptation and Randomization

A.6 MAML

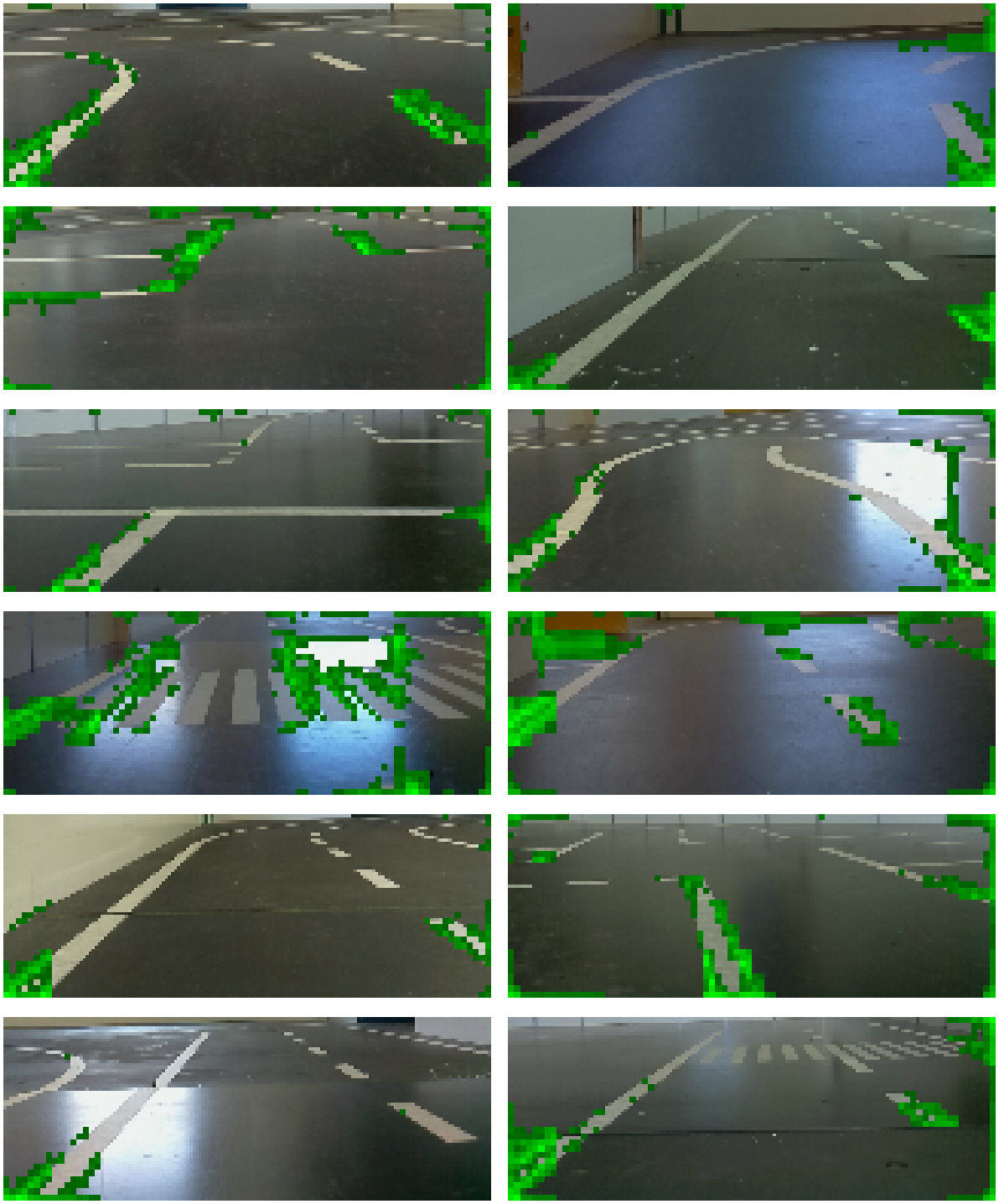


Figure A.6.1: MAML

A.7 FOMAML

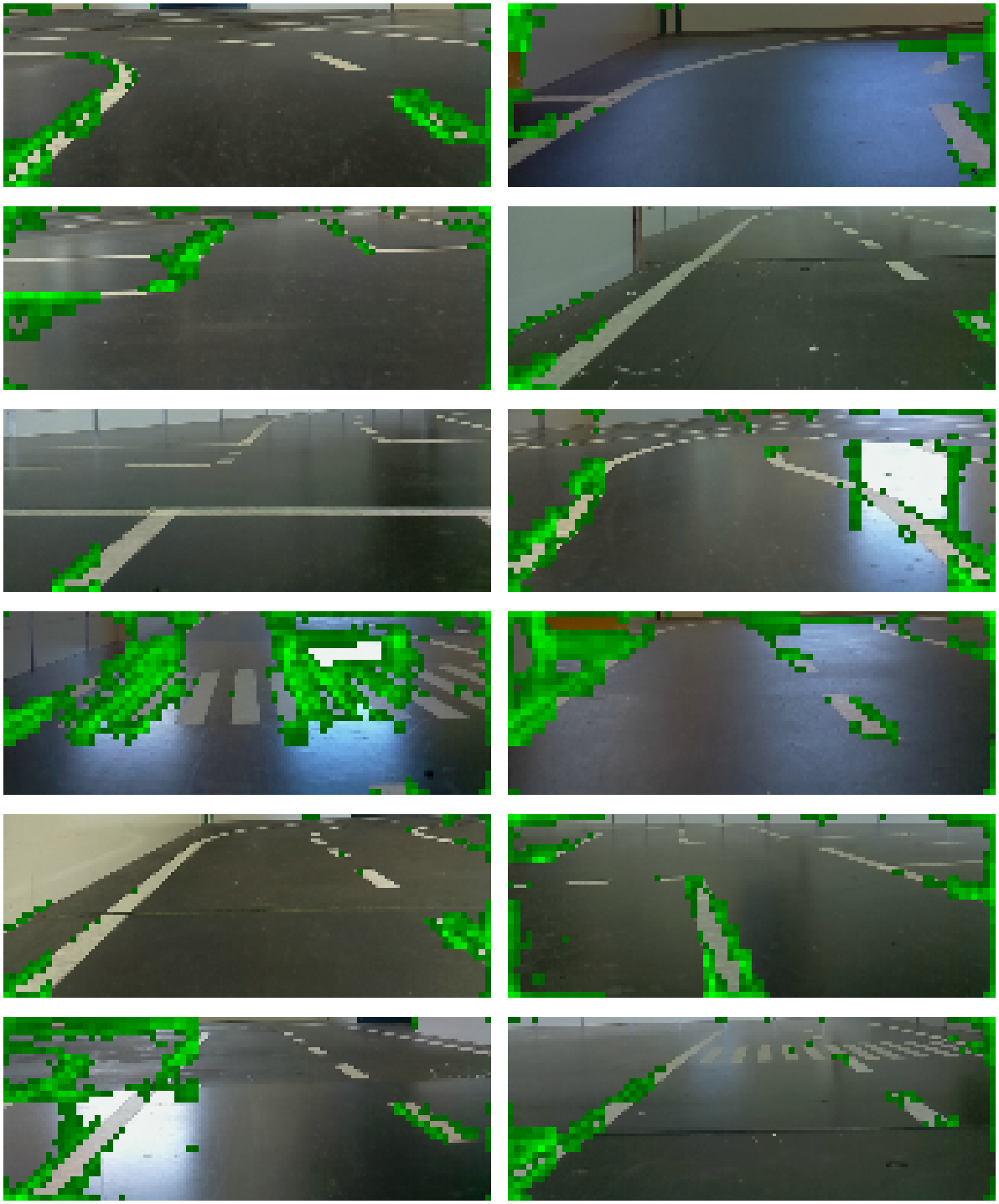


Figure A.7.1: FOMAML