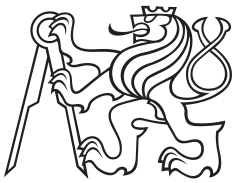
Master thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Simulating depth measuring sensors for autonomous learning and benchmarking

Otakar Jašek

Supervisor: doc. Ing. Karel Zimmermann, PhD.
Field of study: Open Informatics
Subfield: Computer Vision and Digital Image
May 2018

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jašek Otakar** Personal ID number: **420148**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer Vision and Image Processing**

II. Master's thesis details

Master's thesis title in English:

Simulating Depth Measuring Sensors for Autonomous Learning and Benchmarking

Master's thesis title in Czech:

Simulace hloubkových senzorů pro autonomní učení a testování

Guidelines:

Accurate perception is an essential component for many fundamental capabilities such as emergency braking, predictive control for active damping, safe turning on a road intersection or self-localization from offline maps. Consequently, any fully-autonomous vehicle requires an algorithm which process low-level data such as RGBD measurements and provides a high-level interpretation of the actual situation, such as positions of pedestrians and cars in the close neighbourhood of the expected vehicle trajectory. State-of-the-art approaches such as supervised learning of deep convolutional neural networks has started to achieve super-human level, however millions of annotated training data are required for both learning and validation. Collecting and annotating real world data for is extremely demanding. On the other hand, pure physical simulation of RGBD sensors has not yet achieved sufficient level of maturity, despite of increasingly growing game industry. We propose to simulate realistic sensor measurements by introducing not-easy-to-model systematic failures "noise" learned from real captured data.
1. Familiarize yourself with Valeo data-interface and create the calibrated dataset with RGBD images and corresponding annotations.
2. Study state-of-the-art methods for Generative Adversarial Networks such as [1,2,3] and try available implementations [4].
3. Propose a method for data-driven simulation of a depth sensor. Optionally, extend proposed method for an RGBD sensor.
4. Evaluate proposed method and discuss typical failure cases.

Bibliography / sources:

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, Generative Adversarial Nets. Proceedings Neural Information Processing Systems Conference, 2014
[2] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, Image-to-Image Translation with Conditional Adversarial Networks. ArXiv, 2016. https://arxiv.org/pdf/1611.07004v1.pdf
[3] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Josh Susskind, Wenda Wang, Russ Webb, Learning from Simulated and Unsupervised Images through Adversarial Training, CVPR 2017 best paper award.
https://github.com/zhangqianhui/AdversarialNetsPapers
[4] https://github.com/zhangqianhui/AdversarialNetsPapers

Name and workplace of master's thesis supervisor:

doc. Ing. Karel Zimmermann, Ph.D., Vision for Robotics and Autonomous Systems, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **08.01.2018** Deadline for master's thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

doc. Ing. Karel Zimmermann, Ph.D. Supervisor's signature	doc. Ing. Tomáš Svoboda, Ph.D. Head of department's signature	prof. Ing. Pavel Ripka, CSc. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

Declaration

Abstract

Keywords:

Supervisor: doc. Ing. Karel Zimmermann, PhD.

Abstrakt

Klíčová slova:

Překlad názvu: Simulace hloubkových senzorů pro autonomní učení a testování

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

A large number of machine learning applications rely on a vast amount of data from the real world to infer useful relations. However, obtaining these data is not always a viable option. For example, you would like to teach your autonomous car to recognize approaching collision in order to avoid it, but crashing real cars in order to capture the image or depth data is not possible. Luckily, computer graphics started to become more and more realistic in recent years, and it is now possible to capture image and depth data from computer games. These data have one significant advantage – it can simulate almost any scenario such as crashing, unusual environment, etc., as long as it is possible within the game.

Although the data captured from the modern computer games look almost realistic, it suffers from many problems to be readily usable by machine learning applications. The most significant drawback is the fact, that they look *too* perfect – real-world sensors often measure data with noise or fail altogether.

Our goal in this thesis is to find such a relation between the in-game data and the real world data to be able to transform the in-game data to look as realistically as possible. Since we do not have a one-to-one mapping between these data, it is necessary to apply methods of *unsupervised* learning. Recently, a new method suitable for unsupervised generative learning called CycleGAN emerged and it is based on Generative Adversarial Networks (GANs). This method was shown to work on various unpaired image datasets, however, as far as we know this is the first work which is trying to apply it on depth data.

1.2 Thesis structure

In this first chapter, we set up motivation and reasoning for this work and also briefly summarize contribution of this thesis. The next chapter is an overview of related theoretical work. The first section of said chapter briefly summarizes recent work in the field, while the next section explores more deeply neural networks used in this thesis. The last section of this chapter

describes operation of LiDAR which we are trying to simulate in chapter 5.

Chapter 3 is dedicated to the used datasets. In this chapter, we summarize key characteristics of the datasets and how they were obtained.

Chapter 4 describes all the programs written for the purpose of this thesis and shows their functionality. This chapter can also serve as a user guide for the programs.

Chapter 5 is a showcase of performed experiments. We also describe all the drawbacks we encountered during the experiments. The chapter ends with an evaluation of results.

In the last chapter, we analyze all the results and discuss the achieved contributions of this thesis, followed by plans for the future work and conclusion.

1.3 Contribution

Chapter 2

Theory

2.1 Related work

2.2 Used neural networks

The neural networks (sometimes also called artificial neural networks or ANNs) are a powerful tool of today's machine learning. The main component is an *artificial neuron*, a computational unit which takes an input and computes a predefined (usually linear) function with its internal parameters. This output is then optionally fed through (usually nonlinear) *activation function* to introduce nonlinearities in the output. The artificial neurons can be stacked next to each other to form layers, and if we connect multiple layers together, we have a neural network. We can think of the neural network as a nonlinear transformation function with multiple internal parameters.

The process of training the neural network to give the output we desire then consists of feeding the input data into the network and evaluating the performance by the *loss function*, which computes a real-valued number associating the actual output of the network with a notion of a "badness" in comparison with the expected output. This loss function could be for example a norm of a difference between the output of the network and the output given by a human in the case of image labeling or it could be more complex function altogether. The loss function is then minimized with respect to the internal parameters of the neural network by gradient descent algorithm. The most used gradient descent algorithm is a stochastic descent and its variants such as Adam [1] or Adagrad [2].

If the loss function is well defined over the problem set, then the network will give the desired results at the global minimum of the loss function. However, since this function is a function of the *parameters* of the network, it is generally not convex and highly dimensional, and therefore it is hard to reach the global minimum. LeCun [3] gave numerous tricks to improve the likelihood of finding a good enough local minimum.

The neural layers we introduced above are usually called *fully connected* because all the outputs from one layer are connected to all the neurons from the next layer. This was one of the first formulations of artificial neural

networks [4]. However, these fully connected layers are not well suited for computer vision applications, since we would like to have the same response to the particular object in the image regardless of its position or orientation. To overcome this issue, the *convolutional* layers [5], which perform a mathematical operation of convolution over the input, were introduced. Quite often these convolutional layers are followed by the fully connected layers at the end of the network.

2.2.1 Generative adversarial network (GAN)

A generative adversarial network is a concept by Ian Goodfellow [6] aimed at learning to generate a sample from a particular distribution. The main goal is to train a *generator* network to produce samples from the target distribution given a sample from some noise distribution. In order to achieve this, a second network called *discriminator* is introduced, and its main goal is to distinguish between the samples from the real target distribution and the "fake" samples produced by the generator network given a noise sample. If the whole setup is modeled in such a way that the trained discriminator outputs a scalar assigning a probability of the sample coming from the target distribution, the generator is then trying to produce the samples that are convincing enough to the discriminator so that discriminator's output for the generated samples is as close to 1 as possible. This setup was originally formulated within the maximum log-likelihood estimation setup. It could be seen as a two-player minimax game with the value function $V(G, D)$ shown in equation 2.1, where G and D are generator and discriminator functions respectively, p_{target} is a target distribution and p_{noise} is a noise distribution that is usually chosen as a uniform, however it could be any other source of noise data.

$$\min_G \max_D V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{target}}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{noise}}[\log(1 - D(G(\boldsymbol{z})))] \quad (2.1)$$

This formulation immediately yields loss functions for the generator (equation 2.2) and discriminator (equation 2.3) where $\boldsymbol{x}$ are the samples from the target distribution that we present to the networks during the learning and $\boldsymbol{z}$ is a noise sampled at each iteration of the training algorithm.

$$\mathcal{L}_G = \log(1 - D(G(\boldsymbol{z}))) \quad (2.2)$$

$$\mathcal{L}_D = -(\log D(\boldsymbol{x}) + \log(1 - D(G(\boldsymbol{z})))) \quad (2.3)$$

It was theoretically shown [6] that in the case of generator and discriminator having enough capacity this setup allows to train the generator to be able to generate samples indistinguishable from the samples from p_{target}. However this is not easily achievable in practice. One of the main problems is that generator usually does not have an infinite capacity. More problems stem from the fact that the original loss function (equation 2.2) for the generator does not provide strong enough gradients early in the process of training,

therefore a new loss function with the same theoretical properties, but stronger gradients was introduced as shown in the equation 2.4.

$$\mathcal{L}_G = -\log(D(G(\boldsymbol{z}))) \tag{2.4}$$

In practice, we are trying to find the Nash equilibrium [7] of a highly dimensional, non-convex function and while we can obtain gradients for this function using training samples, there is no known algorithm to solve this game exactly [8]. To overcome this obstacle, it is recommended [6] to alternate between training step of the generator and the discriminator on the same data with discriminator being trained first. It is helpful for the training process to train discriminator near its optimum before updating generator and to achieve this, it might be necessary to train discriminator more than once for one batch of samples.

Another problem that could very easily occur is a *mode collapse* of the generator – a point, where generator does not use its full potential and generates a fixed point for multiple inputs keeping discriminator in the dark. Since the generator receives its gradients from the discriminator and discriminator cannot give any useful information anymore, the generator will not be updated in any sensible direction past the point where the mode collapse occurred. After the mode collapse, it does not make any sense to train the affected networks anymore.

2.2.2 GAN variants

Since the inception of GANs, many variants emerged trying to overcome some of the issues outlined in the previous subsection. According to DeepHunt [9], there were 354 papers proposing a variation of GAN as of 10$^{\text{th}}$ May 2018. Most of these improvements revolve around redefining the loss functions and introducing various tricks to achieve better training stability.

In the following subsections will be shortly described some of these variants with their particular improvements and differences from the original GAN.

DCGAN – Deep Convolutional GAN

DCGAN [10] is not a variant of GAN per se, as it mostly involves guidelines for stable training of GAN where discriminator and generator consist of multiple convolutional layers. The said guidelines can be briefly summarized as:

- Use strided convolution and deconvolution instead of pooling layers. The reasoning behind this is to allow the networks to find their own representations of up-sampling and down-sampling operations.
- Use batch normalization [11] everywhere applicable (i.e. in every layer except the last one). This allows to get more stable gradients for back-propagation.

- Do not use fully connected layers that are not the direct output of the discriminator. If there are hidden fully connected layers, then the model stability might improve, however it reduces convergence speed of the training process.
- Use ReLU [12] activation for generator's layers except the last layer using hyperbolic tangent. It was observed, that ReLU helps convergence speed of the training process the most.
- Use LeakyReLU [13] activation for discriminator's layers. This seems to be especially helpful in higher resolution settings.
- Use Adam [1] optimizer with *different* hyperparameters than the usual default. Especially necessary is to lower the learning rate and momentum terms.

LSGAN – Least Squares GAN

The main idea behind LSGAN [14] is not to use maximum log-likelihood framework, but to use least squares instead. The formulation of the generator and discriminator loss functions can be then seen in equations 2.5 and 2.6, where a, b and c are target values of the discriminator that we are aiming for. In most applications, $c = a = 1$ (or 0.9 to introduce label smoothing, as proposed by [8, 15]) and $b = 0$.

$$\mathcal{L}_G = \frac{1}{2}(D(G(\boldsymbol{z})) - c)^2 \tag{2.5}$$

$$\mathcal{L}_D = \frac{1}{2}(D(\boldsymbol{x}) - a)^2 + \frac{1}{2}(D(G(\boldsymbol{z})) - b)^2 \tag{2.6}$$

The reasoning for this reformulation of the loss functions is mostly to provide better gradients and to move the generated samples closer to the decision boundary. In traditional GAN, samples that pass the decision boundary do not provide strong enough gradients and do not contribute to the learning process. However with the LSGAN, there is only one flat point of the loss functions without strong gradients.

WGAN and WGAN-GP – Wasserstein GAN (with gradient penalty)

One of the ideas of the original GANs we have not talked about before is minimizing some metric between the generated and the target distributions. This metric is usually well defined by the respective loss function and for original formulation of GAN it is Kullback-Leibler divergence [16] and for LSGAN it is Pearson χ^2 divergence [17].

WGAN [18] was introduced to minimize Wasserstein-1 distance [19], also known as Earth-Mover. This definition yields following loss functions as seen on equations 2.7 and 2.8.

$$\mathcal{L}_G = -D(G(\boldsymbol{z})) \tag{2.7}$$

$$\mathcal{L}_D = D(G(\boldsymbol{x})) - D(\boldsymbol{z}) \tag{2.8}$$

However, to enforce a Wasserstein-1 distance, it is necessary to satisfy the condition that function D is K-Lipshitz continuous for any given K. This is not easy to achieve and the method used in the original paper was weight clipping to a tight bounding box after each update. It is worth noting the authors admit that this solution is impractical and obviously wrong, however they could not think of a better solution at the time.

The reformulation of WGAN called WGAN-GP [20] emerged soon after and introduced less drastic way to enforce K-Lipshitz continuity. The discriminator's loss function would receive an additional term called gradient penalty forcing the gradients of the function to be "approximately 1 almost everywhere". This gradient penalty removes the need for the weight clipping in the original paper and it is shown in equation 2.9, where $\tilde{\boldsymbol{x}} = \epsilon\boldsymbol{x} + (1-\epsilon)G(\boldsymbol{z})$ and ϵ is a uniform random number from the range [0; 1].

$$\mathcal{L}_{GP} = (\|\nabla_{\tilde{\boldsymbol{x}}} D(\tilde{\boldsymbol{x}})\|_2 - 1)^2 \tag{2.9}$$

It is now even more critical than in the original formulation of GAN to train the discriminator to almost optimum. This stems for the fact that better discriminator yields much stronger gradients than poorly trained one.

Authors of WGAN-GP showed, that the gradient penalty is superior to the weight clipping since the original WGAN exhibited either vanishing or exploding gradients quite often. To demonstrate higher stability, WGAN-GP authors trained many architectures with this criterion on ImageNet [21] dataset and measured the Inception score [8] (score based on ability to produce samples from different ImageNet classes with high classification rate by Inception network [22]) achieved by the network. It was shown [20], that many more architectures were able to obtain high Inception score when trained by WGAN-GP loss functions instead of original GAN loss functions.

Quite an important thing to note is the fact, that WGAN-GP *cannot* use popular Batch normalization [11] layers as it alters the gradients of the layers and makes the gradient penalty useless. The recommendation by the article authors is to use Layer normalization [23] layers instead. It is also beneficial to widen the range of the discriminator function to [-1; 1] (as opposed to [0; 1] in the original GAN) and to achieve this, it suffices to only change the activation function of the last layer of the discriminator to hyperbolic tangent.

2.2.3 SimGAN

SimGAN [24] is GAN variant aiming to learn a *refinement* of simulated data to make them look realistic enough. The paper called the generator network with the name *refiner* to emphasize the fact, that it received simulated data (instead of noise) on the input. The paper introduced three important concepts in the field of GANs. The first one is *local* adversarial loss – this idea means, that the discriminator should not produce only scalar output, but

a response map, where each part of this map corresponds to the particular patch of the evaluated image. The reasoning behind this is to allow patches with not so dominant features to contribute to the loss of the discriminator and by extension of the refiner as well.

Second key idea is adding a so-called *self-regularization* term to the refiner's loss function. This term can be seen in equation 2.10, where $\boldsymbol{x}$ is a sample from simulated data (refiner's input), $\tilde{\boldsymbol{x}}$ is a refined sample (refiner's output), λ_{reg} is a relative weight given to the importance of this loss term and ψ is a feature mapping from image space to the feature space. This feature mapping could be any function with important properties (such as classification of the data), or it could be simple identity and then the $\mathcal{L}_{reg}$ would become pixel-wise distance.

$$\mathcal{L}_{reg} = \lambda_{reg} \left\| \psi(\boldsymbol{x}) - \psi(\tilde{\boldsymbol{x}}) \right\|_1 \tag{2.10}$$

This self-regularization term allows to teach the refiner to keep the most important information of the image during the refinement process.

Third key idea is to introduce memory for the discriminator's learning process. The discriminator should be unable to forget about the images it has seen an epoch ago. In order to facilitate this memory, a history buffer which keeps the refined samples is introduced. When performing single optimization step on discriminator weights, approximately half of the used batch is replaced with the samples from this history buffer. To be able to limit the buffer size, after performing the training step, half of the samples from the batch is randomly selected and random samples in the history buffer are replaced by this selection.

2.2.4 CycleGAN

CycleGAN [25] is a concept aiming to match two different distributions by the means of GANs. The core idea is to have *two* GANs trained simultaneously with one generator learning the mapping from the first distribution to the second and the other generator learning the reverse mapping. To enforce this reverse mapping, new term called *cycle consistency loss* is added to the loss function of the generators. This loss term can be seen in the equation 2.11, where p_X, p_Y are the distributions between which we try to find a mapping $\boldsymbol{x} \sim p_X$, $\boldsymbol{y} \sim p_Y$, $G_{X \to Y}$ is a generator mapping from p_X to p_Y, $G_{X \to Y}$ is a generator mapping from p_Y to p_X and λ_{cyc} is a relative weight given to the importance of this loss term.

$$\mathcal{L}_{cyc} = \lambda_{cyc}(\left\| G_{Y \to X}(G_{X \to Y}(\boldsymbol{x})) - \boldsymbol{x} \right\|_1 + \left\| G_{X \to Y}(G_{Y \to X}(\boldsymbol{y})) - \boldsymbol{y} \right\|_1) \tag{2.11}$$

This approach of training two GANs simultaneously can give us mapping between these two distributions *without* having a pair-to-pair correspondences. It is important to note that even though this closely relates to the style transfer problem, the resulting mappings should work in both directions which is usually not the case with more common approaches [26] to the style

transfer. Since this seems like an approach that could help us model the mapping between real-life and in-game data of the cars' sensors, we decided to use CycleGAN as a basis for our experiments using various underlying architectures of GANs.

The CycleGAN does not concern itself with the particular type of GAN used as a generator mapping, however original results were published using LSGAN with instance normalization [27], $\lambda_{cyc} = 10$ and local adversarial loss used for discriminator.

2.3 Description of LiDAR

Chapter 3

Datasets

3.1 Depth sensors

Using outputs from depth sensors in neural network can be quite challenging. The main difficulty stems from the fact that even though most of the depth sensors (including LiDAR) capture data on a regular grid, there is usually needed some post-processing of the data which discards some invalid points (for various reasons, i.e. the point is too far to be considered reliable or the laser did not return any response). This post-processing usually results in a point cloud (with additional information such as intensity) of an *irregular* shape – meaning there is not the same amount of points in one measurement. This is a problem that is not easily solved by neural networks with convolutional and fully-connected layers. The reasoning for why this is an obstacle is provided in section 2.2.

Since we are aiming at *generating* data using CycleGAN [25], we ideally want measurement from both datasets to have equal shape. If that would not be possible for some reason, the least constraining requirement is that there is a mapping representable by a neural network which transforms a measurement from one dataset to a measurement from other dataset with matching shapes and vice versa.

To ease the work of neural networks, we decided to use representation as close to LiDAR as possible for both datasets. Velodyne HDL-64E (the LiDAR used by Valeo company) has 64 lasers (each with different vertical angle) and by default spins at 600 RPM, which according to the LiDAR manual [28] means that horizontal resolution is 0.1728°. We can then create a grid of 64×2084 virtual lasers, where this grid corresponds to all data points collected during one full rotation of LiDAR. The process of creating such grid consists of casting a ray from the camera center corresponding to the particular horizontal and vertical angle to the point cloud and finding the closest point to this ray. Then, threshold of the distance of the point from the ray is necessary to make sure our closest point is not too far away. We set up this threshold as 0.5 % of the distance from the camera to simulate conic nature of the laser. This reasoning immediately shows that a multi-channel grid is necessary where at least one of the channels encodes validity of the corresponding ray. One measurement therefore consists of an "image" of size

(a) : RGB image **(b) :** Depth image

Figure 3.1: Example of images from GTA

64×2084×3, where first channel corresponds to the distance of the ray from the camera center, second channel corresponds to intensity of the response (information that real LiDAR outputs as well) and third channel corresponds to validity of a particular ray.

Another way a particular laser in this "image" can be invalid is if the corresponding point found in point cloud is either too far or too close from the camera center. These limits come again from the LiDAR manual, minimal distance is 0.9 meters, maximal distance is 131 meters.

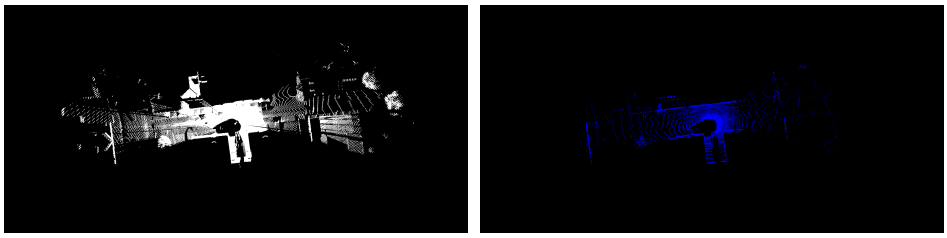
It could happen, that substantial amount of information would be missing from one measurement – especially if the point cloud was rather sparse (as it was in Valeo dataset case). Even worse, the missing information could look entirely random. To remedy this, we employed linear interpolation of the rays, that are marked as invalid and have at least half of their neighbors in a neighborhood of size 3 valid. The said interpolation involved distance and intensity as well.

There are numerous advantages of this representation. One is that such representation could be easily treated as an image by neural networks and therefore convolution is applicable. Also, for neural networks, fixed-size input is often desired. Another advantage is that this representation is easily transformable to the point cloud representation. And if we take first channel separately and mask it with validity channel, then it can be easily displayed as a depth image of size 64×2048.

The only thing considering depth dataset creation we have not talked about yet is the method of obtaining the corresponding point clouds, camera center and starting rotation. Those aspects vary depending on the dataset and therefore we will talk about them in the next two subsections.

3.1.1 Grand Theft Auto dataset

Thanks to Matěj Račinský, who did tremendous work on exploiting Grand Theft Auto and extracting information from it automatically (such as depth, stencil buffer, etc.), we only had to use the data provided by his scripts. The data came in the form of the depth image such as 3.1b from in-game camera and corresponding camera matrix transforming the image to the world coordinates. However, due to the game limitations, it was always possible



(a) : Full point cloud **(b) :** Reconstructed point cloud

Figure 3.2: GTA point clouds

to capture only one camera at a time and it took non-zero time to switch the cameras to capture another image. Because of these limitations it took about one second of in-game time to capture the full 360° scene around the car. Data in Valeo dataset produce a full scan at a rate of 10 Hz and since we wanted to match the Valeo data as closely as possible, we simply interpolated positions of the car with 100 ms intervals. This actually created more measurements than depth images, however they are all taken from a different position in the in-game world.

All four virtual cameras sit at the height of one meter from the car center, which later proved to be too low and therefore quite a large portion of the car was reflected in LiDAR-like image. To correct for that, the center of the virtual LiDAR was shifted by 1.5 meters above the camera centers (2.5 meters above the car center).

Since the intensity of real-life LiDAR largely depends on the color of the surface, we decided that the intensity component of each ray would be determined by gray-scale value of the corresponding pixel of the in-game camera. The dataset has 14046 LiDAR-like measurements, and it was split into two parts – training and testing. Training portion of the dataset consists of 8427 measurements, testing contains 5619 measurements. Data from testing portion were not seen by the network during the training phase. The size of the dataset translates into 1404.6 seconds of in-game time that was recorded continuously.

Figure 3.2a shows an example of the original point cloud, figure 3.2b shows the recreated point cloud from the data from GTA dataset, figure 3.3 shows an example of a first channel of the data. To ease the viewing, the horizontal stripe of 64×2084 is cut into 4 pieces stacked on top of each other, creating the new size of 256×521.

3.1.2 Valeo dataset

Valeo company provided us with two types of data – raw and converted. Raw data contained UDP packets from various sensors before any processing with most prominent being Velodyne HDL-64E LiDAR and OXTS xNAV 550 which is a GNSS-aided inertial measurement system. Converted data consisted of point clouds and transformation matrices. Each point cloud corresponded to one full rotation of LiDAR and was already compensated for

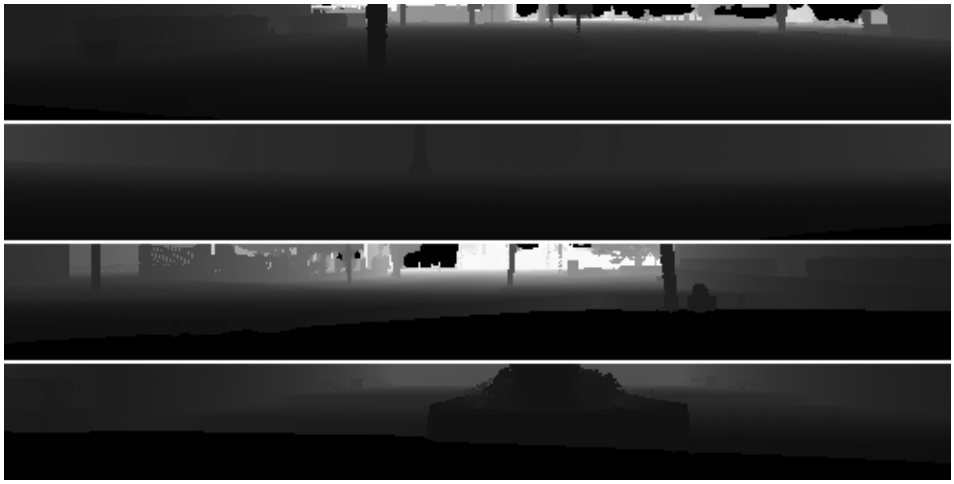


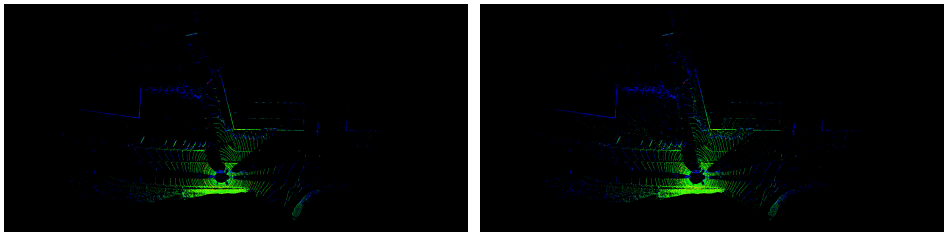
Figure 3.3: First channel of LiDAR-like data from GTA dataset. For easier viewing, the strip of data is divided into 4 equal stripes stacked on top of each other.

the movement of the car. The matrices served for transforming particular point clouds into common reference frame. This reference frame was usually the same as the coordinate frame of the first point cloud – therefore the first point cloud had *identity* as this transformation matrix. The origin of the coordinate frame seemed to be in the car center – we moved the virtual LiDAR center by two meters up to simulate it being on top of the roof of the car.

We decided it would be easier to use *converted* data – mostly because it seemed that it contained precisely the same data as the raw, but without the hassle of parsing and processing Velodyne and OXTS UDP packets. Converted data also contained intensity measurements.

The dataset consists of 22 runs of lengths from 60 to 80 seconds in a cityscape only, resulting in total of 17393 full scans. Training portion of the dataset contains 10435 samples, testing part has 6958 measurements. The data were recorded from 22$^{\text{nd}}$ February 2017 till 28$^{\text{th}}$ March 2017 with two different cars.

Figure 3.4a shows an example of the original LiDAR full scan, figure 3.4b shows the recreated point cloud from LiDAR-like data and figure 3.5 shows an example of a first channel of the data. It is partitioned similarly as in figure 3.3.



(a) : Original LiDAR point cloud **(b) :** Reconstructed point cloud

Figure 3.4: Valeo point clouds

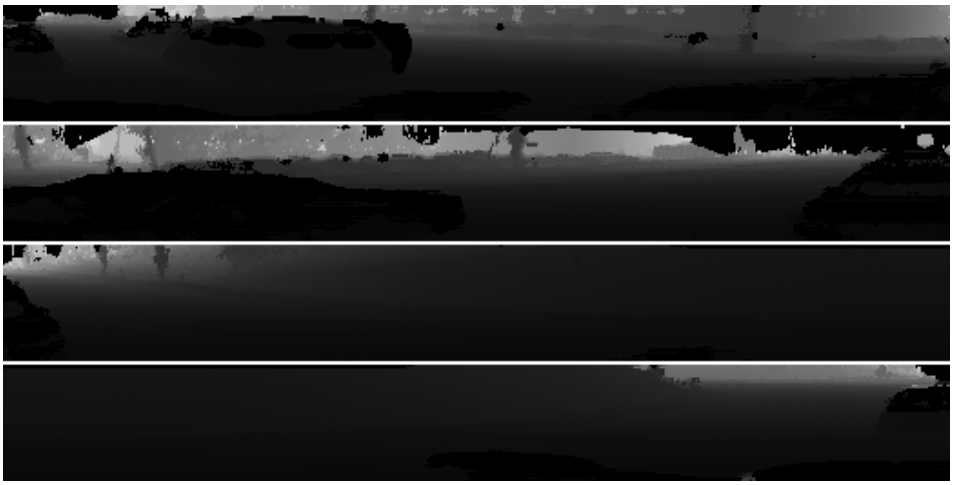


Figure 3.5: First channel of LiDAR-like data from Valeo dataset. For easier viewing, the strip of data is divided into 4 equal stripes stacked on top of each other.

Chapter 4

Programs

The main program developed for the purpose of this thesis was a Python package `cycle` implementing modular CycleGAN [25] in TensorFlow [29] and two programs built on top of this package. This package and associated programs reside in a directory `mod-cycle-gan` at `https://gitlab.fel.cvut.cz/jasekota/master-thesis/tree/master/code/mod-cycle-gan` and will be therefore together referenced as `mod-cycle-gan`. There is also an utility program written in C++ called `dat-unpacker` which reads ADTF DAT files used by Valeo company and extracts data from them into an intermediate format similar to the one gathered from GTA. Last portion of the code developed for this thesis is a folder with various Python modules (with critical part of the code written in Cython) and scripts with simple name `data-processing`. These three programs/packages will be described more in depth in the following sections.

All of the developed code is in the directory `code` on attached CD and also available at `https://gitlab.fel.cvut.cz/jasekota/master-thesis/tree/master/code`.

4.1 `mod-cycle-gan` – Python package `cycle` and programs `train.py` and `test.py`

Python package `cycle` is the implementation of CycleGAN [25] with large inspiration from GitHub repository of Van Huy at `https://github.com/vanhuyz/CycleGAN-TensorFlow`.

4.1.1 Exported classes

- `CycleGAN` – Main class implementing CycleGAN.

  `__init__()`
  : Constructor of this class takes numerous arguments. First two arguments (`XtoY, YtoX`) correspond to GANs to be set in cycle fashion (instances of `nets.GAN` or its subclasses), another two (`X_feed, Y_feed`) are for TFRecords file readers (`utils.TFReader`) and another two (`X_name, Y_name`) correspond to names of the dataset

for pretty printing of logs and Tensorboard messages. Following argument (`cycle_lambda`) is a λ for cyclic loss function (for more detail see section 2.2.4). Next argument (`tb_verbose`) is a boolean for deciding whether to create summaries for Tensorboard and following argument (`visualizer`) is a function to use for visualizing the data in Tensorboard – if this argument is set to `False` or `None` then no function will be used for visualization.

Next four arguments (`learning_rate, beta1, steps, decay_from`) control optimization process – namely initial learning rate for Adam optimizer [1], parameter beta1 of said optimizer, number of steps (where one step corresponds to one batch) and number of steps after which the learning rate starts to decay to eventually stop at zero. Following argument (`history`) indicates, whether to use history pool according to [24] and finally, last argument (`graph`) specifies the computational TensorFlow graph in which the model should be created. If it is left as `None`, then new graph will be created.

`get_model()`

This method actually creates the full model in TensorFlow graph. As such, it should be only called once. It sets up all the losses and their respective optimizers. This method has no arguments.

`train()`

This method is the main training loop. The only required argument (`checkpoints_dir`) is the top-level checkpoints directory in which a new directory for this session is either created if needed or selected as a loading point in case of retrying training. Next two arguments (`gen_train, dis_train`) specify how often should be generator and discriminator trained within one training step. Next argument (`pool_size`) specifies the size of the history pool. Following argument (`load_model`) specifies a directory from which to load a saved model if retrying. Note that it is a path relative to top-level checkpoints directory. If this argument is `None`, new directory with current timestamp is created and new training starts. Next argument (`log_verbose`) is a boolean specifying whether to log current loss periodically or not. Next argument (`param_string`) specifies string which is a serialized version of arguments with which `train.py` script was executed. This string will be saved to checkpoint directory with name `params.flagfile`. Last argument (`export_final`) specifies whether to export final model after training as a binary protobuf used for testing.

`export()`

This method requires two arguments – first argument (`sess`) is a session in which a model was run and the second (`export_dir`) is a directory in which to save the model. There will be two saved models of names `{Xname}2{Yname}.pb` and `{Yname}2{Xname}.pb` which can then be used for testing. This method is automatically at the end of the `train()` method if the last argument (`export_final`)

was set to `True`.

`export_from_checkpoint()` – **static method**

This method is a static counterpart of the `export()` method. It requires more arguments than method `export()` because it does not have all the book-keeping information the instance method has. First two arguments (`XtoY, YtoX`) are instances of GANs with the same model as used for training, another four arguments (`X_normer, X_denormer, Y_normer, Y_denormer`) are normalization and denormalization functions to be used for both datasets prior feeding the examples to network and converting them back to useful values. Next argument (`checkpoint_dir`) corresponds to checkpoint directory where the model is stored and following argument (`export_dir`) specifies directory in which the output models will be saved. Last two arguments (`X_name, Y_name`) correspond to the names of the datasets for easier identification of created models.

`test_one_part()` – **static method**

This method tests the stored exported model with a NumPy file and saves the important outputs of the network to a new NumPy file. First argument (`pb_model`) is a path to an exported binary protobuf model of the network to test. Another argument (`infile`) specifies the path to the input file to test and next argument (`outfile`) corresponds to a path of output file. This output file will be a Npz NumPy file with three fields – `output` (output generated by corresponding generator), `d_input` (value of corresponding discriminator evaluated on input data) and `d_output` (value of corresponding discriminator evaluated on output data).

Last argument (`include_input`) is a boolean specifying whether to include input data in the output file or not. If set to `True`, `outfile` will become larger, however it will be more self-contained.

- `utils.TFReader` – Class for reading TFRecords file, which is a TensorFlow binary format for efficient storage of data and features based on Protobuf.

  `__init__()`

  First argument (`tfrecords_file`) specifies the path to the TFRecords file to read. Another argument (`name`) specifies the name of the dataset. This name is rather important, because it needs to be the same as set in TFRecords file `utils.TFWriter` for parsing of the TFRecords file to be successful. Next argument (`shape`) is a shape of one example stored in TFRecords file. Since all the data in TFRecords file are stored as flattened arrays, this needs to be set to correct size in order to reshape it to desired size. Next argument (`shuffle_buffer_size`) is passed to the method `shuffle()` of `tf.data.Dataset` and as such represents the number of samples used for shuffling. Following two arguments (`normer, denormer`)

are functions operating on tensors for normalizing and denormalizing elements of the dataset (i.e. casting to correct type, squeezing or expanding range, etc.) By default these are identity functions. These functions should be able to accept a keyword `name` (they are used for exporting). Next argument (`batch_size`) is a size of one batch produced by this reader and the last argument (`num_threads`) specifies how many threads should be used in operations concerning creating the dataset where applicable.

`feed()`
: This method returns new batch of elements from the dataset when run in TensorFlow session. It is used by `CycleGAN` methods `get_model()` and `train()`. It does not accept any parameters.

- `utils.TFWriter` – Class for creating TFRecords file from NumPy files.

  `__init__()`
  : The constructor accepts three arguments – first argument (`name`) is a name of a dataset. Next argument (`infiles`) is either a single NumPy file or a list of such files comprising the full dataset. It is expected, that all dimensions except the first one will match within the files. The first dimension represents number of *single* elements (with possibly more complex shape, such as $64 \times 2084 \times 3$ as is the case in LiDAR-like data used in our experiments) of datasets. Last argument (`process_data`) is a function operating on these single elements and tweaking them somehow if needed before storing the data to TFRecords file. The reason, why this argument might be useful (instead of using argument `normer` of class `utils.TFReader`) is that this function does not operate on tensors which makes most functions more limiting.

  `run()`
  : This method takes one argument (`outfile`) specifying path to the output TFRecords file. It will report progress to the default logger (with level `INFO`) every 10 examples processed.

- `utils.DataBuffer` – Class implementing history pool according to [24]. This class is used by method `train()` of class `CycleGAN` and should not be instantiated on its own.

  `__init__()`
  : Constructor of this class takes three arguments. First argument (`pool_size`) manages the size of the pool to be used. It has to be either -1 (where essentially there is no pool and this particular instance of the class has no effect) or at least as large as the second argument, `batch_size`. Third argument (`old_prob`) controls the probability with which the older image is returned by method `query()` instead of the one that was given. This probability comes to play only when the buffer is filled to its maximum (`pool_size`).

`query()`
: This method will return the data of the same shape it was fed (by first argument – `data`). If the internal buffer was already filled, then it will replace randomly elements of the data forming a batch with a probability given in the constructor by argument `old_prob` and swaps the elements it replaced into its buffer in the places of the elements that are used in replacement. If the buffer is not filled yet, it will only store the elements from `data` argument and not replace them.

  Second argument (`cur_step`) indicates the global step of the training process in order to be able to return the same data for the same step (for example, if training uses multiple training step for discriminator or generator).

- `nets.BaseNet` – Base class for mapping networks (Generator and Discriminator). This class encapsulates mapping network and if you want to create your own mapping network and feel limited by the capabilities of this, you should subclass it and re-implement method `transform()`.

  `__init__()`
  : Constructor of this class takes five arguments. First argument (`name`) is a name of the network as it will appear in TensorFlow computational graph and this name will encapsulate all of the operations of the network. Second argument (`network_desc`) is a string describing layers of the network and will be dissected in more detail at subsection 4.1.2. Third argument (`is_training`) is a boolean indicating whether the network is in a training or testing mode. This is mostly important for norms.

    Next argument (`weight_lambda`) is a λ of weight term of the loss function for this particular network. This was motivated by having lot of networks with skip connection where we wanted to make only small changes to the image and thus minimizing the weights of generators and therefore generating only small perturbations.

    Last argument (`norm`) is a type of normalization used in network layers. Can be either `'instance'` [27], `'batch'` [11], `'layer'` [23] or anything else for no normalization between layers. It does not make any sense to use different normalizations within one network so this setting can be made global for the whole network.

  `__call__()`
  : This is the way how the mapping induced by this object will be called. It essentially just wraps the call to `transform()` method inside a variable scope of the name specified by the first argument of the constructor (`name`) and collects all the trainable variables into the list called `variables`. This book-keeping is done to ease the subclassing since now the only method to be replaced is `transform()` without the need to worry about collecting all the trainable variables and placing it under same particular variable scope.

`transform()`

This method accepts one argument (`data`) – tensor that will be transformed by the network. It is the core of the class `nets.BaseNet` and if you decide to write your own mapping network by subclassing, this method *must* be implemented. By default, it creates the mapping network according to the argument `network_desc` supplied to the constructor. The syntax of this simple string will be more in depth explained at subsection 4.1.2.

`weight_loss()`

This method return weight loss of the mapping network defined by equation 4.1, where θ_F is a set of trainable variables of a mapping function F represented by this object, $\lambda_{\mathrm{w}F}$ is a multiplier denoting the contribution of this loss term to the overall loss function and $\mathcal{L}_{\mathrm{w}F}$ is a loss term returned by this method.

$$\mathcal{L}_{\mathrm{w}F} = \lambda_{\mathrm{w}F} \frac{1}{|\theta_F|} \sum_{\boldsymbol{w} \in \theta_F} \|\boldsymbol{w}\|_2^2 \tag{4.1}$$

- `nets.GAN` – Implementation of Generative Adversarial Network (GAN) [6]. Uses original loss functions.

  `__init__()`

  First two arguments (`gen, dis`) of the constructor are the mapping networks – generator and discriminator (in the original paper $G(\cdot)$ and $D(\cdot)$), where discriminator should produce real number in a range [0; 1] due to the way how is loss function defined. If the discriminator function's output is of higher dimension than one, then the appropriate loss function is computed in each dimension independently and the final loss is the arithmetical mean of these loss functions. If discriminator network is a convolutional network, one can think of this as the loss computed at different patches extracted by the network [24].

  Next two arguments (`in_shape, out_shape`) specify input and output shapes of generator *without* the batch size. Though this information could be easily obtained from the generator function, it is there mostly to check that the shapes are correct and indeed what you intended them to be. Next two arguments (`gen_lambda, dis_lambda`) are λs that correspond to the weight of the respective terms in the final loss function. Next argument (`selfreg_lambda`) corresponds to the weight of the self-regularization term [24] of the generator's loss. Last argument (`selfreg_transform`) is a mapping used in the self-regularization term. If set to `None`, then identity mapping will be used.

  `gen_loss()`

  This method expects one argument – data of `in_shape` that will be fed to generator to produce the loss term of the generator

according to the equation 4.2, where λ_G is a multiplier denoting the contribution of this loss term, $G(\cdot)$ is a generator mapping, $D(\cdot)$ is a discriminator mapping, $\boldsymbol{x}$ is input data for generator mapping and $\mathcal{L}_G$ is a loss term returned by this method.

$$\mathcal{L}_G = -\lambda_G \log(D(G(\boldsymbol{x}))) \tag{4.2}$$

Note, that the original formulation of the generator loss function by [6] is slightly different (as seen in the equation 4.3 where all the symbols have the same meaning as in the equation 4.2), however it was suggested in the same paper, that the formulation 4.2 is equivalent with an important advantage of stronger gradients early in the training process.

$$\mathcal{L}_G = \lambda_G \log(1 - D(G(\boldsymbol{x}))) \tag{4.3}$$

`dis_loss()`

This method takes two arguments (`real, fake`) which are both of the `out_shape` – `real` is a real sample from the target distribution and `fake` is a result of applying $G(\cdot)$ to a sample from the input distribution. This method then computes the discriminator term of the loss function specified by the equation 4.4, where λ_D is a multiplier denoting the contribution of this loss term, $D(\cdot)$ is a discriminator mapping, $\hat{\boldsymbol{x}}$ corresponds to `real` argument, $\boldsymbol{y}$ corresponds to `fake` argument and $\mathcal{L}_D$ is a loss term returned by this method. Original paper was maximizing the same function without minus sign, and since we are minimizing all terms of the loss functions, we added minus in front of the loss function. The division by 2 is there only to scale both terms equally with respect to the generator loss.

$$\mathcal{L}_D = -\frac{\lambda_D}{2}(\log(D(\hat{\boldsymbol{x}})) + \log(1 - D(\boldsymbol{y}))) \tag{4.4}$$

`selfreg_loss()`

This method takes two arguments (`orig, conv`) and computes self-regularization [24] term of the generator's loss function according to the equation 4.5, where $\boldsymbol{x}$ corresponds to the `orig` argument, $\tilde{\boldsymbol{x}}$ corresponds to the `conv` argument, $\psi(\cdot)$ is a feature mapping (specified at the constructor), λ_{reg} is a multiplier denoting the contribution of this loss term to the overall loss and $\mathcal{L}_{reg}$ is a loss term returned by this method.

$$\mathcal{L}_{reg} = \lambda_{reg} \left\| \psi(\boldsymbol{x}) - \psi(\tilde{\boldsymbol{x}}) \right\|_1 \tag{4.5}$$

- `nets.LSGAN` – Implementation of Least Squares GAN [14]. This is a subclass of `nets.GAN` and as such all of the methods accept the same arguments. The only difference is in the equations governing the computation of respective terms of the loss function in methods `gen_loss()` and `dis_loss()`.

`gen_loss()`
This method implements the generator loss function according to the equation 4.6. The meaning of the used symbols is the same as in equation 4.2. The reason for number 0.9 comes from the label smoothing proposed by [8, 15].

$$\mathcal{L}_G = \lambda_G \left\| D(G(\boldsymbol{x})) - 0.9 \right\|_2^2 \tag{4.6}$$

`dis_loss()`
This method implements the discriminator loss function according to the equation 4.7. The meaning of the used symbols is the same as in equation 4.4.

$$\mathcal{L}_D = \frac{\lambda_D}{2}(\left\| D(\hat{\boldsymbol{x}}) - 0.9 \right\|_2^2 + \left\| D(\boldsymbol{y}) \right\|_2^2) \tag{4.7}$$

- `nets.WGAN` – Implementation of Wasserstein GAN with gradient penalty [20]. This is a subclass of `nets.GAN` as well, however it introduces one more term to the loss function, so called gradient penalty. This is implemented in method `grad_loss()`.

  `__init__()`
  The constructor takes the same arguments as the constructor of `nets.GAN` with one more argument (`grad_lambda`) specifying the weight of the gradient penalty.

  `gen_loss()`
  This method implements the generator loss function according to the equation 4.8. The meaning of the symbols is the same as in equation 4.2.

$$\mathcal{L}_G = -\lambda_G D(G(\boldsymbol{x})) \tag{4.8}$$

  `dis_loss()`
  This method implements the discriminator loss function according to the equation 4.9. The meaning of the used symbols is the same as in equation 4.4. Note that the gradient penalty term of the loss function is computed in the method `grad_loss()`

$$\mathcal{L}_D = \frac{\lambda_D}{2}(D(\boldsymbol{y}) - D(\hat{\boldsymbol{x}})) \tag{4.9}$$

  `grad_loss()`
  This method computes the gradient penalty term according to the equation 4.10, where λ_{GP} is a multiplier denoting the contribution of the gradient penalty term to the overall loss, $\tilde{\boldsymbol{x}} = \epsilon\hat{\boldsymbol{x}} + (1-\epsilon)\boldsymbol{y}$, ϵ is a uniform random number from [0,1] and $\mathcal{L}_{GP}$ is a loss term returned by this method.

$$\mathcal{L}_{GP} = \lambda_{GP}(\left\| \nabla_{\tilde{\boldsymbol{x}}} D(\tilde{\boldsymbol{x}}) \right\|_2 - 1)^2 \tag{4.10}$$

  `full_dis_loss()`
  This method computes the full loss of the discriminator, $\mathcal{L}_{DGP} + \mathcal{L}_D$.

4.1.2 Parameterization of the mapping networks by `network_desc`

The mapping network could be easily parameterized by a special `network_desc` string. This string comprises of layers separated by semicolon (`';'`). All layers comprise of letter from {`'c'`, `'b'`, `'r'`, `'f'`} specifying the layer type, followed by hyphen (`'-'`) and a list of numerical parameters specific to particular layer each of them separated by hyphen as well. The last part of a layer is again a letter from {`'r'`, `'t'`, `'l'`, `'s'`, `'i'`} specifying the used activation function for this particular layer. All of the trainable variables are initialized using Xavier initializer [30] in order to keep the scale of the gradients approximately the same across all layers. Normalization (either instance, batch, layer or none) is always used before applying activation function.

Note that the parameterization using this notation is rather simple and does not allow all possible configurations of specified layers. If you desire finer control over created layers, then using this `network_desc` string is not ideal for you.

The tail of the `network_desc` string after last semicolon specifies output operation and could be any number of the letters from the set {`'s'`, `'c'`, `'a'`}. These output operations are chained in the order in which they appear in the tail of the `network_desc` string.

Layers description

`'c'` – 2D Convolutional layer [5]
This layer accepts *three* integer arguments – first argument is a kernel size (the kernel will have square shape), second argument is a stride (same in all directions) used in the convolution and third argument specifies the number of filters used (number of channels in the output). For example the string `'c-7-1-64-r'` specifies the convolutional layer with kernel of size 7, stride 1, 64 output channels and ReLU [12] used as activation function.

`'b'` – 2D Convolutional ResNet [31] block
This layer creates a ResNet block and accepts *two* integer arguments. First argument is a kernel size of each layer comprising this block and second argument specifies the number of repetitions of this convolutional layer. Since ResNet block requires the input and output simension to match, stride is implicitly set to 1 and number of the output channels is the same as the number of the input channels. For example, the string `'b-3-2-r'` specifies the most classical ResNet block comprising of two convolutional layers with kernel size 3, stride 1 and ReLU activation in between of those two layers.

`'r'` – Resize and 2D Convolutional layer [32]
This layer was originally deconvolutional [33] layer, however to mitigate checkerboard artifacts stemming from using deconvolutional layer [32],

we decided to use instead resize and convolutional layer. It accepts *three* integer arguments and one float argument. First three arguments correspond to the same arguments as regular convolutional layer and last argument corresponds to coefficient of resizing. The resizing is done using method `tf.image.resize_images` with resize method `tf.image.ResizeMethod.NEAREST_NEIGHBOR`. The number of channels after resizing stays the same as number of channels in the input image. For example, the string `'r-3-1-32-2-r'` will first resize the input image making it twice as large as input and then perform convolutional operation with kernel of size 3, stride 1, 32 output channels and ReLU as activation function.

`'f'` – Fully connected layer
This is the simple fully connected layer. It accepts one integer argument specifying the number of output neurons. The input gets flattened before performing it is fed into the neurons.

Available activation functions

- `'r'` – Rectified Linear Unit (ReLU) [12]
- `'l'` – Leaky Rectified Linear Unit (Leaky ReLU) [13]
- `'t'` – Hyperbolic tangent function defined by $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- `'s'` – Sigmoid function defined by $S(x) = \frac{e^x}{e^x + 1}$
- `'i'` – Identity, no nonlinear activation function is used.

Output operation

These operations are chained in order in which they appear in the tail of the `network_desc`. Note, that this part of the `network_desc` can be empty if you don't want to use any special output operation.

- `'s'` – Sums the current output with input to the whole mapping network.
- `'c'` – Clip the output of the whole network to the range [-1; 1].
- `'a'` – Use activation function on the output. Since the operating range for most of the networks is [-1; 1], the only activation function that *makes sense* to use is tanh, so this function will be used.

Example of the full network

The full network could be for example parameterized by string `'c-7-1-64-r;c-5-2-128-r;b-3-3-r;r-5-1-64-2-r;c-7-1-3-t;sc'`. This network consists of two regular convolutional layers with kernel sizes 7 and 5, strides 1 and 2 and having 64 and 128 output channels. Both of these layers use ReLU as activation function. Convolutional layers are then followed by

one ResNet block with three convolutional layers and kernel size 3 and again, ReLU is used. Next resize operation follows which creates the image double the size and is followed by another two convolutional layers with kernel sizes 5 and 7, both strides equal to 1 and having 64 and 3 output channels. First of these convolutional layers is followed by ReLU activation function, the second one uses tanh as activation. Output of the last convolutional layer is then added to the original input and this sum is then clipped to the range [-1; 1].

4.1.3 Models

The module `cycle` contains a folder called `models` in which the settings for different experiments and mapping networks reside. Each submodule in this folder should correspond to one different parameterization specific to different dataset. Each submodule should export these variables and classes in order to be able to use it as a valid model for CycleGAN:

`X_name, Y_name`
: Names of the X and Y dataset.

`X_DATA_SHAPE, Y_DATA_SHAPE`
: Tuples specifying shape of one element of X and Y dataset.

`XY_Generator, YX_Generator`
: Classes specifying generators from X to Y and from Y to X dataset respectively. Note that these classes should be either `nets.BaseNet` or its subclasses accepting the same arguments in its constructor.

`X_Discriminator, Y_Discriminator`
: Classes specifying discriminators of X and Y datasets. Note that these classes should be either `nets.BaseNet` or its subclasses accepting the same arguments in its constructor.

`X_normer, Y_normer`
: Functions operating on tensors and accepting keyword argument `name`. These functions will be used to normalize data from X and Y datasets before feeding them into the network.

`X_denormer, Y_denormer`
: Functions operating on tensors and accepting keyword argument `name`. These functions will be used to denormalize outputs from the network. They should be inverse functions to functions `X_normer` and `Y_normer`.

`visualize` – optional
: Function to visualize data during training. This function should accept three batches of images, original data, output of first generator and output of second generator applied to the output of the first generator. The function should return *one* batch of images (with same batch size as it received) to show at appropriate place in TensorBoard, therefore

using concatenate operation is preferred. This function is necessary to implement if you specify visualizing in `train.py` script.

For reference implementation of a particular model for LiDAR-like datasets, see folder `mod-cycle-gan/cycle/models/lidar`.

4.1.4 Scripts `train.py` and `test.py`

The folder `mod-cycle-gan` also contains two executable scripts – `train.py` and `test.py`. The script `train.py` is used for training a CycleGAN and has 36 different flags. However, the script is mostly only a wrapper around instantiating various classes and running the method `train` of the class `CycleGAN`. Running the script with the help flag `--helpfull` lists all the available flags and their short description. You can set up almost all of the parameters mentioned in the description of the public methods of exported classes.

The script `test.py` is used for testing the resulting network and has 9 flags. It is a wrapper around the static method `test_one_part` of the class `CycleGAN`. Running the script with the help flag `--helpfull` lists all the available flags and their short description.

4.2 `dat-unpacker` – C++ utility

`dat-unpacker` is a small utility program based on ADTF Streaming library by Audi Electronics Venture GmbH. The reasoning for writing this utility program is to be able to unpack data obtained in ADTF DAT format and further process them. In order to build `dat-unpacker`, C++ compiler, CMake[1] in version at least 3.5, Boost[2] and ADTF Streaming library are needed. Binary copy of ADTF Streaming library is provided in the directory `code/external/adtf-streaminglib`.

The utility was written with Valeo dataset in mind, therefore a lot of features are hardcoded and specific to the data from Valeo dataset only. The utility takes a DAT file as an input, extracts information such as point clouds, timestamps, etc. from it and dumps the data on disk in a very crude format for further processing by Python.

This utility accepts four arguments:

`-i/-input-file [FILE_NAME]` – required

Input DAT file to read. This file needs to contain at least streams with names containing strings `"matrix"`, `"scan"` and `"3dod"`. The number of datablocks within the streams with the names `"matrix"` and `"scan"` needs to be the same. Furthermore, it is expected that each block of the stream `"scan"` will contain $4 \times 4 \times n$ bytes, where every 16 bytes correspond to 4 float numbers indicating one point scanned by Velodyne

[1] `https://cmake.org/`
[2] `https://www.boost.org/`

LiDAR in XYZI format. All points within one data-block will be in the same coordinate frame, where position (0, 0, 0) corresponds to the center of the car, and will correspond to one revolution of LiDAR. Each block of the stream `"matrix"` will contain a string `"<matrix>[`x`]"` where x is a string representation of 16 float numbers encoding a 4×4 matrix used to convert each corresponding block of stream `"scan"` into a *reference* coordinate frame. Stream `"3dod"` contains *one* datablock consisting of $4 \times 4 \times m$ bytes with the same format as blocks from the stream `"scan"` with all the points transformed into one common reference frame and optionally filtered out according to some rule.

The input DAT file may contain additional streams which do not concern the `dat-unpacker` utility. The utility probably will not accept DAT files produced by other companies, unless the adhere to the requirements specified above.

`-o/-output-dir [DIR_PATH]` – required

The directory in which to dump all the useful data. The directory will be created if it does not exist yet. There will be data numbered from 0 with three different extensions – `.ts`, `.matrix` and `.pts`. All of them are only binary representations of the numeric data contained in the input DAT file. Files with the extension `.ts` have size of 8 bytes and it is one signed long integer corresponding to the timestamp of the block with microsecond resolution. Files with the extension `.matrix` have size of 128 bytes and it is 16 doubles corresponding to the transformation matrix. Files with the extension `.pts` have size of $4 \times 4 \times n$ bytes and they correspond to one unmodified block from the stream `"scan"` of the input file.

`-a/-all` – optional

If this flag is specified, there will be one additional file in the output directory called `0000.dod` which contains all the points recorded within the datablock from the stream `"3dod"`.

`-h`

Prints the simple help and exits.

4.3 `data-processing` – various Python modules and scripts

Folder `data-processing` contains two Python modules and one Cython module as well as 4 scripts in a folder `scripts`. The creation of LiDAR-like dataset is split into two parts – first we create unified ZIP files containing all the info we have (metadata such as camera position, rotation, timestamp, etc., point clouds and transformation matrix to one common coordinate frame) and then we create NumPy LiDAR-like data from these ZIP files by casting virtual rays into the point clouds.

4.3.1 Module `zip_processing`

The module `zip_processing` has 4 functions:

`rgb2gs`
: Simple function converting RGB image (`rgb`) to grayscale *without* gamma correction.

`depth_to_pcl`
: Conversion of GTA depth images into point clouds. Takes depth image (`depth`), RGB image (`rgb`, for using grayscale value as intensity of the point in resulting point cloud), inversion of projection (`proji`) and view matrices (`viewi`) and bounding box (`bbox`) in which to convert the points. The points that are further in distance from the camera center than the limits specified by the bounding box will be thrown away.

`make_gta_zip`
: Creates one ZIP file from GTA data. It expects ID (`i`) of the GTA file, paths to json (`json_file`) with metadata, depth image (`depth_file`) and RGB image (`rgb_file`), bounding box (`bbox`, with same semantics as `depth_to_pcl`) and two optional arguments `outputdir` (if `None`, no data will be saved) and `return_data`.

`make_valeo_zips`
: Creates ZIP files from Valeo DAT files. Expects a DAT file (`datfile`), path to the binary of `dat-unpacker` (`binary`) and a directory in which to store the results (`outputdir`). It also has one optional argument (`remove`) specifying whether to remove temporary files created by `dat-unpacker` after processing them into one ZIP file. It creates one ZIP file for each revolution of LiDAR in the `outputdir`.

4.3.2 Module `datapool`

The module `datapool` contains two classes for manipulating ZIP files created by module `zip_processing` and three functions for reading data from the ZIP file. The class `DataPool` keeps ZIP files in a cache and is primarily intended for Valeo dataset, since point cloud from one ZIP file corresponds to the full rotation of LiDAR. The constructor expects a list of ZIP files in a format produced by functions in module `zip_processing`. The constructor will read only metadata from these ZIP files and fetches the data only when requested by method `load_data`. This method returns a point cloud and car center at a particular timestamp. There is also a method `load_rotmat` which returns a rotational matrix specifying the the angle between a ray in the direction (1, 0, 0) and heading of the car.

The class `GTADataPool` extends the class `DataPool` and serves the same purpose, only for GTA data. The reasoning for this is the fact that it is necessary to concatenate multiple point clouds to simulate a full rotation of LiDAR. The method `load_full_rot` fetches the point clouds at nearest

timestamps, concatenates them together and interpolates the position of the car at particular timestamp.

The module also has functions `read_pcl`, `read_metadata` and `read_both` for reading data from ZIP files in format produced by module `zip_processing`. All of them expect a path to the ZIP file.

4.3.3 Module `rays`

Module rays is a Cython module, so before using it, it is necessary to compile the module. For a convenience, there is a simple Makefile provided which does the compilation. After that you can use it as any you would any other Python module. It has 4 public functions.

`gta_cam_rot`
: Creates rotational matrix from two NumPy vectors of Euler angles. The first vector (`world_rot`) corresponds to world rotation and second vector (`relative_rot`) corresponds to relative rotation, since the camera can be rotated differently than the heading of the car. Third optional argument (`rads`) specifies, whether the angles are already in radians or in degrees.

`interpolate_lidar`
: This function fills in missing values of the LiDAR-like data. This procedure is described more in detail in the section 3. First argument (`lidar_data`) is a NumPy array of LiDAR-like data, second argument (`to_fill`) specifies the minimal percentage of neighboring rays that has to be valid in order to fill in the missing value. Next argument (`mask_size`) specifies the size of the neighborhood across which to interpolate missing values and last argument (`iters`) specifies how many times should the procedure be done. If there is no more values to fill (either all values are filled or no missing value has enough filled neighbors), then this method will do less iterations than prescribed by argument `iters`.

`get_lidar_data`
: This method computes the LiDAR-like data from an instance of either `DataPool` or `GTADataPool` classes at a given timestamp. First argument (`pool`) corresponds to the instance of the `DataPool` class, second argument (`timestamp`) corresponds to the timestamp at which to compute the data (double, in seconds, however with double precision), third argument (`lidar_correction`) corresponds to the shift of the virtual center of the LiDAR if it is not located at the origin of the coordinate frame of the point cloud and could be set to `None`, if no correction is necessary. The last argument (`allowance`) corresponds to the maximal ratio of the distance of the point to the ray and the length of the ray itself in order to be considered valid. Note, that even though the function is written in quite an optimized fashion, it still takes large amount of time on a dense point clouds because it needs to cast 64×2084 virtual rays and find the nearest point in the point cloud from them. It takes about three to four

minutes on a GTA dataset for *one* element of LiDAR like data. It takes significantly less time on Valeo dataset since it is much sparser.

`reconstruct_pcl`
: This method creates point cloud corresponding to the instance of LiDAR-like data. It takes one argument (`lidar_data`) and returns point cloud in the form of NumPy array in the shape of $4 \times n$ where n is a number of valid rays. The returned point cloud is in XYZI format.

4.3.4 `data-processing` scripts

There are three scripts for processing data to create a dataset and one script to get the viewable data from finished experiments. The processing scripts are `gta-lidar.py`, `gta-zip.py` and `valeo-lidar.py` and all of these take three arguments – input folder, output folder and optionally third argument specifying the number of threads to use. If the third argument is not set, the script will use half of the available CPUs. The scripts are mostly wrappers around the functions in the modules in `data-processing folder`.

The last script called `process-output.py` takes two arguments – first argument corresponds to the folder where two files with the names `gta2valeo.npz` and `valeo2gta.npz` are located and second argument optionally specifies the number of samples to process. If it is not set, then the number of samples is assumed to be 100. It creates two folders within the input folder called `gta2valeo` and `valeo2gta` each of them containing 6 files for each sample processed. The files are depth and intensity images (saved as 64×2084 grayscale PNG images) and point cloud stored as a text file (for viewing by for example CloudCompare software[3]) with each row corresponding to one point in the point cloud. The files are stored as original input data to the CycleGAN network and their transformed counterparts. The resulting filenames are therefore `%03d.{depth,inten}.{orig,conv}.png` and `%03d.pcl.{orig,conv}.txt`, where `%03d` is ID number of the sample within the processed files.

[3] `http://www.danielgm.net/cc`

Chapter 5

Experiments

5.1 Depth sensors

5.2 Evaluation

Chapter 6
Conclusion

6.1 Discussion

6.2 Future work

6.3 Conclusion

Bibliography

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

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[3] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[4] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5 (4):115–133, Dec 1943. ISSN 1522-9602. doi: 10.1007/BF02478259. URL `https://doi.org/10.1007/BF02478259`.

[5] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time-series. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

[6] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[7] John F Nash et al. Equilibrium points in n-person games. 1950.

[8] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *CoRR*, abs/1606.03498, 2016.

[9] Avinash Hindupur. The gan zoo, April 2017. URL `https://deephunt.in/the-gan-zoo-79597dc8c347`. [Online; posted Apr-19-2017, updated May-10-2018].

[10] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015.

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[12] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models.

[14] Xudong Mao, Qing Li, Haoran Xie, Raymond Y. K. Lau, and Zhen Wang. Multi-class generative adversarial networks with the L2 loss function. *CoRR*, abs/1611.04076, 2016.

[15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[16] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951. doi: 10.1214/aoms/1177729694. URL `https://doi.org/10.1214/aoms/1177729694`.

[17] Karl Pearson F.R.S. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900. doi: 10.1080/14786440009463897. URL `https://doi.org/10.1080/14786440009463897`.

[18] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *ArXiv e-prints*, January 2017.

[19] Leonid Nisonovich Vaserstein. Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, 5(3):64–72, 1969.

[20] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[22] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[23] J. Lei Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *ArXiv e-prints*, July 2016.

[24] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Josh Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. *CoRR*, abs/1612.07828, 2016.

[25] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networkss. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

[26] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[27] Dmitry Ulyanov, Andrea Vedaldi, and Victor S. Lempitsky. Instance normalization: The missing ingredient for fast stylization. *CoRR*, abs/1607.08022, 2016.

[28] Velodyne LiDAR Inc. *User's manual and programming guide*, May 2013. URL `http://velodynelidar.com/lidar/products/manual/HDL-64E%20S3%20manual.pdf`.

[29] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[30] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

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

[32] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016. doi: 10.23915/distill.00003. URL `http://distill.pub/2016/deconv-checkerboard`.

[33] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *CoRR*, abs/1609.05158, 2016.