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.

1.2 Thesis structure

In this first chapter, we set up motivation and reasoning for this work. 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 used datasets and is divided into two parts corresponding to depth and RGB 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 contribution of this thesis, followed by plans for the future work and conclusion.

Chapter 2

Theory

2.1 Related work

2.2 Used neural networks

One way to overcome this obstacle is to use technique such as spatial pyramid pooling [1], however it is not siutable for a point cloud representation since the dimensions of such point cloud will be n×3.
Another possible approach to use could be PointNet [2], however since we want to be able to have point cloud as an output of the network as well, this approach seem to be unsuitable. This is the reasoning why point cloud representation is not suitable at all.

2.2.1 Generative adversarial network (GAN)

2.2.2 GAN variants

2.2.3 CycleGAN

2.3 Description of LiDAR

Chapter 3

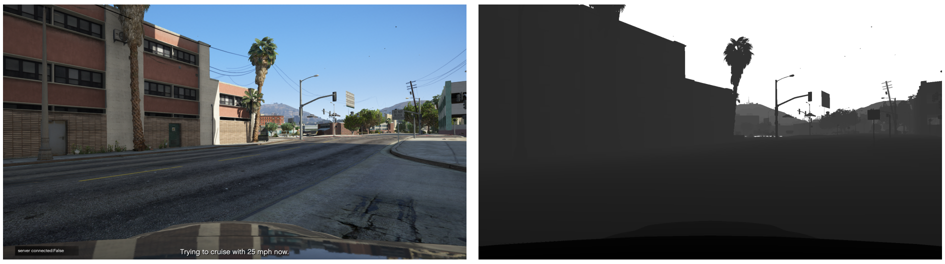
Datasets

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 [3], 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 [4] 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.

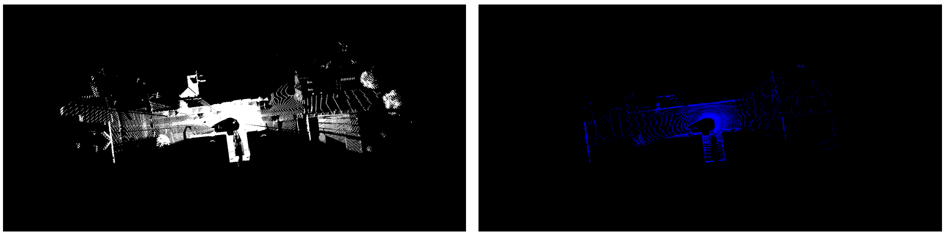
It could happen, that substantial amount of information would be missing from one measurement – especially if the point cloud was rather sparse (as it was in Valeo dataset case). Even worse, the missing information could look entirely random. To remedy this, we employed linear interpolation of the rays, 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

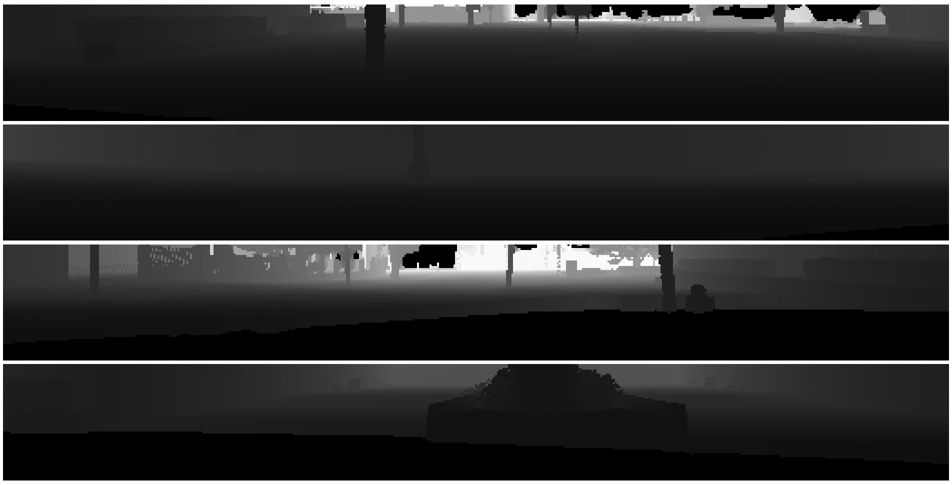


Figure 3.3: First channel of LiDAR-like data from GTA dataset. For easier viewing, the strip of data is divided into 4 equal stripes stacked on top of each other.

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.

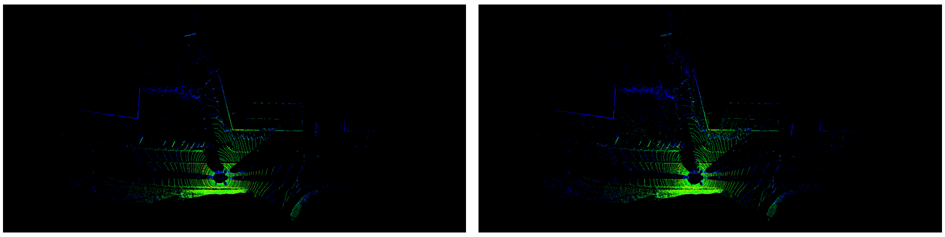
The dataset consists of 22 runs of lengths from 60 to 80 seconds in a cityscape only, resulting in total of 17393 full scans. Training portion of the dataset contains 10435 samples, testing part has 6958 measurements. The data were recorded from 22$^{\text{nd}}$ February 2017 till 28$^{\text{th}}$ March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

3.2 RGB sensors

3.2.1 Grand Theft Auto dataset

3.2.2 KITTI dataset



(a) : Original LiDAR point cloud

(b) : Reconstructed point cloud

Figure 3.4: Valeo point clouds

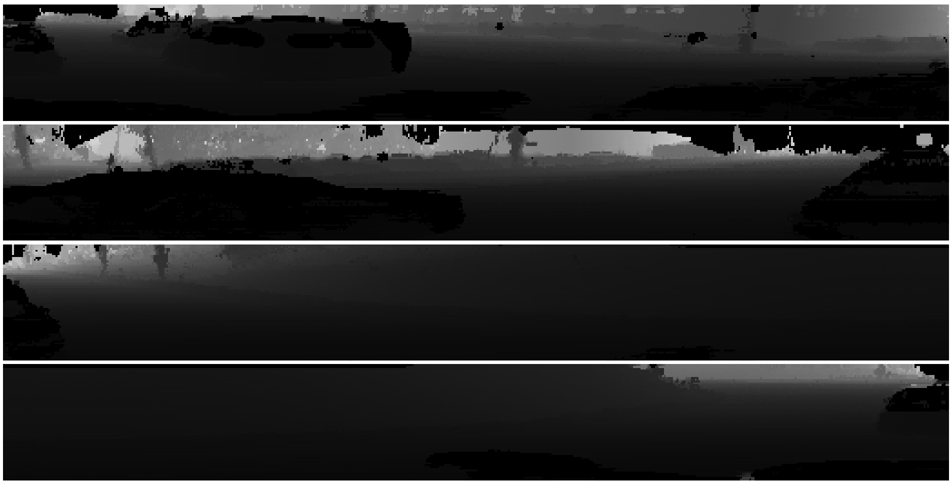


Figure 3.5: First channel of LiDAR-like data from Valeo dataset. For easier viewing, the strip of data is divided into 4 equal stripes stacked on top of each other.

Chapter 4

Programs

The main program developed for the purpose of this thesis was a Python package `cycle` implementing modular CycleGAN [3] in TensorFlow [5] 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 code developed for this thesis is a simple Python module (with critical part of the code written in Cython) with simple name `data-processing`. These three programs/packages will be described more in-depth in following sections.

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

Python package `cycle` is the implementation of CycleGAN [3] 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 2.2.3). 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 visualisation.

Next four arguments (learning_rate, beta1, steps, decay_from) control optimization process – namely initial learning rate for Adam optimizer [6], 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 [7] 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 [7]. 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 4.1.1. 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'` [8], `'batch'` [9] 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 4.1.1.

`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 $L_{\mathrm{w}F}$ is a loss term returned by this method.

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

- `nets.GAN` – Implementation of Generative Adversarial Network (GAN) [10]. 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 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.

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.

`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 L_G is a loss term returned by this method.

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

Note, that the original formulation of the generator loss function by [10] 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.

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

`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 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.

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

- `nets.LSGAN` – Implementation of Least Squares GAN [11]. 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 4.5. 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 [12, 13].

$$L_G = \lambda_G \left\| D(G(\boldsymbol{x})) - 0.9 \right\|_2^2 \tag{4.5}$$

  `dis_loss()`

  This method implements the discriminator loss function according to the equation 4.6. The meaning of the used symbols is the same as in equation 4.4.

$$L_D = \frac{\lambda_D}{2}(\left\| D(\hat{\boldsymbol{x}}) - 0.9 \right\|_2^2 + \left\| D(\boldsymbol{y}) \right\|^2) \tag{4.6}$$

- `nets.WGAN` – Implementation of Wasserstein GAN with gradient penalty [14]. 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 4.7. The meaning of the symbols is the same as in equation 4.2.

  $$L_G = -\lambda_G D(G(\boldsymbol{x})) \tag{4.7}$$

  `dis_loss()`
  : This method implements the discriminator loss function according to the equation 4.8. 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()`

  $$L_D = \frac{\lambda_D}{2}(D(\boldsymbol{y}) - D(\hat{\boldsymbol{x}})) \tag{4.8}$$

  `grad_loss()`
  : This method computes the gradient penalty term according to the equation 4.9, where λ_{DGP} 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 random number from [0,1] and L_{DGP} is a loss term returned by this method.

  $$L_{DGP} = \lambda_{DGP}(\|\nabla_{\tilde{\boldsymbol{x}}} D(\tilde{\boldsymbol{x}})\|_2 - 1)^2 \tag{4.9}$$

  `full_dis_loss()`
  : This method computes the full loss of the discriminator, $L_{DGP} + L_D$.

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'`} 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 [15] in order to keep the scale of the gradients approximately the same across all layers. Normalization (either instance or batch) 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 [16]`

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 [17] used as activation function.

`'b' - 2D Convolutional ResNet [18] 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 [19]`

This layer was originally deconvolutional [20] layer, however to mitigate checkerboard artifacts stemming from using deconvolutional layer [19], 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.

Available activation functions.

- `'r'` – Rectified Linear Unit (ReLU) [17]

- `'l'` – Leaky Rectified Linear Unit (Leaky ReLU) [21]
- `'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].

Models

4.2 `dat-unpacker` – C++ utility

4.3 `data-processing` – Python package

Chapter 5
Experiments

5.1 Depth sensors

5.2 RGB sensors

5.3 Evaluation

Chapter 6

Conclusion

6.1 Discussion

6.2 Future work

6.3 Conclusion

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *CoRR*, abs/1406.4729, 2014. URL `http://arxiv.org/abs/1406.4729`.

[2] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *arXiv preprint arXiv:1612.00593*, 2016.

[3] 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.

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

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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`.

[20] 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.

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