

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

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

1 Introduction	1
1.1 Motivation	1
1.2 Thesis structure	1
2 Theory	3
2.1 Related work	3
2.2 Used neural networks	3
2.2.1 Generative adversarial network (GAN)	3
2.2.2 GAN variants	3
2.2.3 CycleGAN	3
2.3 Description of LiDAR	3
3 Datasets	5
3.1 Depth sensors	5
3.1.1 Grand Theft Auto dataset . . .	6
3.1.2 Valeo dataset	7
3.2 RGB sensors	8
3.2.1 Grand Theft Auto dataset . . .	8
3.2.2 KITTI dataset	8
4 Programs	11
4.1 mod-cycle-gan	11
4.1.1 Exported classes	11
4.2 dat-unpacker	14
4.3 data-processing	14
5 Experiments	15
5.1 Depth sensors	15
5.2 RGB sensors	15
5.3 Evaluation	15
6 Conclusion	17
6.1 Discussion	17
6.2 Future work	17
6.3 Conclusion	17
Bibliography	19

Figures

Tables

3.1 Example of images from GTA . . .	6
3.2 GTA point clouds	7
3.3 First channel of LiDAR-like data from GTA dataset	8
3.4 Valeo point clouds	9
3.5 First channel of LiDAR-like data from Valeo dataset	9



Chapter 1

Introduction



1.1 Motivation



1.2 Thesis structure

In this first chapter we set up a motivation and reasoning for this work. The next chapter is an overview of a 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

2.2.1 Generative adversarial network (GAN)

2.2.2 GAN variants

2.2.3 CycleGAN

2.3 Description of LiDAR

One way to overcome this obstacle is to use technique such as spatial pyramid pooling [1], however it is not suitable for a point cloud representation since the dimensions of such point cloud will be $n \times 3$.

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.

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



Figure 3.1: Example of images from GTA

$64 \times 2084 \times 3$, 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

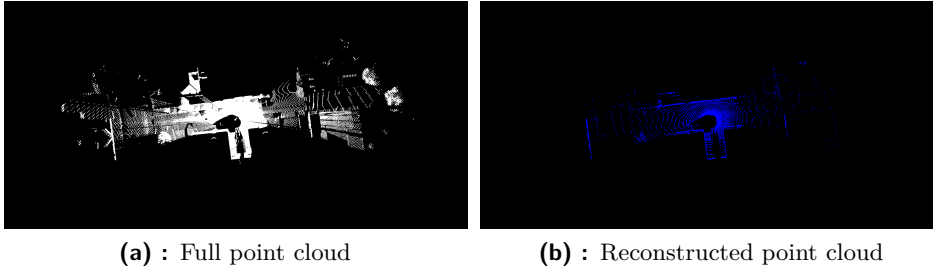


Figure 3.2: GTA point clouds

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 0.75 meters above the camera centers (1.75 meters above the car center).

We don't have intensity information from this LiDAR simulation, so we set intensity to all valid rays as 0.5 (maximal intensity is 1, minimal is 0). 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 pointcloud, 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 the movement of the car. The matrices served for transforming particular



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.

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 22nd February 2017 till 28th 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

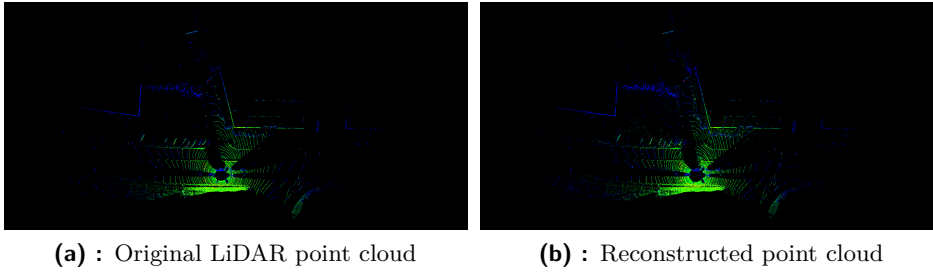


Figure 3.4: Valeo point clouds

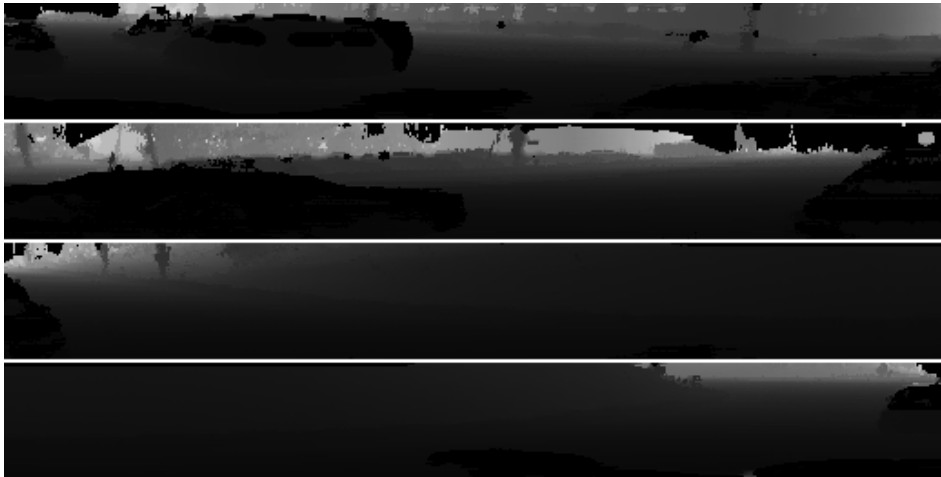


Figure 3.5: First channel of LiDAR-like data from Valeo dataset. For easier viewing, the strip of data is divided into 4 equal stripes stacked on top of each other.

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 visualising 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_normmer`, `X_denormmer`, `Y_normmer`, `Y_denormmer`) 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__()`

`feed()`

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

`__init__()`

`run()`

- `utils.DataBuffer` – Class implementing history pool according to [7].

`__init__()`

`query()`

- `nets.BaseNet` – Base class for mapping networks (Generator and Discriminator).

```
__init__()  
  
transform()  
  
weight_loss()
```

- `nets.GAN` – Implementation of GAN [8]. Uses original loss functions.

```
__init__()  
  
gen_loss()  
  
dis_loss()
```

- `nets.LSGAN` – Implementation of Least Squares GAN [9]. Subclass of `nets.GAN`.


```
__init__()  
  
gen_loss()  
  
dis_loss()
```

- `nets.WGAN` – Implementation of Wasserstein GAN with gradient penalty [10]. Subclass of `nets.GAN`.

```
__init__()  
  
gen_loss()  
  
dis_loss()  
  
grad_loss()  
  
full_dis_loss()
```




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.
- [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 networks. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.
- [4] Velodyne LiDAR Inc. *User’s manual and programming guide*, May 2013.
- [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. 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] 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.

- [9] 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.
- [10] Ishaan Gulrajani, Faruk Ahmed, Martín Arjovsky, Vincent Dumoulin, and Aaron C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.