Unsupervised Synthetic Data Generation through Rendering Engines

Federico Arenas López

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2021

Abstract

The current pipeline for developing Computer Vision (CV) models suffers from a bottle-neck caused by the often resource expensive task of data collection and data annotation. Synthetic Data Generation (SDG) leverages the recent, rapid development of 3D Rendering Engines, software capable of generating highly realistic views of a 3D model or a 3D scene, to unclog this bottle-neck by automatically generating images and annotations that can be used to train CV models. Recent studies have focused in using this concept to build Unsupervised Synthetic Data Generation architectures, which focuses on using rendering engines to generate synthetic images through unsupervised learning models that jointly minimize the distribution gap between the generated images and a set of real target images, while maximizing performance on a downstream CV task. These models promise to automate the data collection process to any application by only needing a 3D scene in a rendering engine and small set of target images, to generate a large set of training images similar to the target images.

The proposed project systematically investigates the scalability of this type of models to increasingly complex scenarios by focusing on the Meta-Sim architecture, one of the original papers that introduced this specific type of architectures. The study argues that the architecture's ability to scale was briefly studied and the results reported in the original study are insufficient to determine transferability, and therefore focuses on using a new 3D scene and the Unity rendering engine to evaluate this. To use this 3D scene, we develop an efficient architecture design to attach a rendering engine to an Unsupervised SDG architecture. Subsequently, the evaluation is done through an extensive set of experiments that specifically focus on the generated distributions at each complexity increase, and on the gradient flow which determines successful learning.

Finally, by closely examining the results from the generated distributions and gradient flows of the architecture, we show that in order for these models to learn on different 3D scenes, the hyperparameter tuning needs to be focused on controlling their learning instability. Indeed, we find that this is caused by the finite-difference approximation used to estimate the gradients that pass through the renderer. At last, we end the study by pointing out ways of addressing these issues so that future work can continue to improve Unsupervised SDG models.

Acknowledgements

Firstly, I would like to offer my deepest gratitude to my two supervisors, Patric Fulop and Pavlos Andreadis, who since the first meeting were always in the best attitude to help me bring this project to fruition. I would like to extend this gratitude to the Neurolabs company, who funded this project and without whom I would not have been able to run countless GPU-expensive experiments in the cloud. I would like to specially thank Markus Schläfli for being there to advice me on the best way to engineer this architecture.

On a personal level, I'd like to thank my friends and fellow students from the University of Edinburgh for being my family away from home, since being an international student can often be extremely challenging. Moreover, I'd like to thank the COLFUTURO organization and the Colombian Government for funding and allowing me to pursue my MSc in Artificial Intelligence degree at The University of Edinburgh. This Government effort gives us Colombians an opportunity to bring back highly impactful and transformative ideas that will bring our country forward, and improve the challenging conditions we often find ourselves in as fellow Latin Americans. Finally, and perhaps more importantly, I'd like to extend a message of gratitude to my family in Spanish.

Le quiero ofrecer mis más profundos agradecimientos a mi mamá, Myriam López, mi papá, Juan Bernardo Arenas, y a mi hermano, Alejandro Arenas, por brindarme el apoyo necesario para llevar a cabo tan monumental hazaña que es hacer una maestría en el exterior. Sin su acompañamiento nada de esto sería posible. Mis agradecimientos son infinitos. Al resto de mi familia les agradezco por siempre brindarme las mejores energías durante mis años de formación, y por siempre creer en mí. De corazón, gracias.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Federico Arenas López*)

Table of Contents

Chapter 1

Introduction

The field of Computer Vision (CV) has seen rapid advancements in model performance in the past 10 years, due to the introduction of Deep Convolutional Neural Netowork (CNN) architectures, advancements in computing resources, and data availability [71]. The data collection process to train these architectures however has not experienced the same rate of technological advancement. As of now, this step continues to be arguably the most time-consuming and laborious task in the CV model building pipeline, since traditionally humans have been needed to provide the ground truth prediction for each image the model needs to be trained on. Fortunately, advancements seen in the gaming industry have fostered the development of highly realistic 3D Rendering Engines [56], software that allows the user to generate views of a 3D model, from which media can be exracted. This catalyzed the development of architectures that use rendering engines for Synthetic Data Generation (SDG) to train CV models. Since the full information of a 3D model is known in the renderer, the annotation process can be automated. This significantly speeds up data collection, and facilitates the development of the type of architecture this dissertation will focus on: Unsupervised Learning SDG architectures that use rendering engines to generate data which closely resembles a small set of real images [29, 17, 8, 52], and can be used to train CV models deployed in the real world.

Specifically, we focus on the Meta-Sim architecture. Proposed by Kar et al. [29] it generates images by communicating with a 3D rendering engine to generate a synthetic dataset, then compared to a real target dataset through a distribution similarity metric that measures the distance between both distributions. This distance is used as an error by the architecture, and at each learning step, the architecture learns to generate data that resembles the target data the best. Additionally, the architecture can integrate the error from training a TaskNetwork (any CNN-based CV model) with the generated

data, and validated on the target data. This further teaches the architecture to modify the 3D scene in the renderer to generate data that maximizes the performance of an arbitrary CV model. In our study, we seek to extend the Meta-Sim study on the grounds that the original paper only briefly reports performance results when increasing the number of mutable attributes[1] in the scene. We argue that these results are inconclusive in order to determine whether the Meta-Sim architecture is scalable to different applications. We hypothesize that a more complex scene (e.g. having more modifiable attributes, and more objects in it), will mean a decrease in performance. Ultimately, by focusing our study on Meta-Sim, the seminal paper that used distribution matching with a 3D rendering engine, we will shed light on the limitations of such architectures.

To investigate our hypothesis, we develop a fast architecture to generate data from a new 3D scene using the Unity 3D rendering engine. Subsequently, we systematically increase the complexity of this scene. By comparing the generated distributions to the target distributions at each complexity increase, we observe and tune the learning of the network attempting to improve performance. After extensively experimenting with the architecture on the new 3D scene, we find that our hypothesis holds when only increasing the number of mutable attributes in the scene, without increasing the number of objects. The hypothesis does not hold when the inverse is done. From the gradient flow of each experiment, we find that increasing the number of mutable attributes in the scene makes the learning unstable, and therefore no convergence is reached. Interestingly however, adding more objects to the scene regularizes this effect, and helps the network learn more stably. As it is now, we deem the architecture not scalable to different custom cases, since the hyperparameter tuning needed to control the instability in the gradients during learning is too specific in order for the architecture to learn in a new custom case. Nonetheless, the insights from our experiments provide concrete directions to improve the robustness of these models.

Finally, we structure the study as follows. After providing some context into the different solutions to tackle the CV data collection problem, Chapter 3 provides a thorough explanation of the Meta-Sim model, its theoretical limitations, attaching Unity, and how we intend to systematically test Meta-Sim's scalability. Subsequently, Chapter 4 reports and analyses the results obtained from the scalability study. At last, Chapter 5 discusses the results obtained, considers how our study extends the state of the art, poses questions to answer in future work, and presents the conclusions of the study.

[1] *mutable attributes*, or *modifiable attributes*, are the changeable parameters of a given 3D model in a scene. For example, a car in a 3D scene may have mutable attributes like height, rotation, offset, etc.

Chapter 2

Background

Ever since the introduction of Deep Convolutional Neural Networks (CNNs) by Yann LeCun in 1999 [38], and after the introduction of AlexNet in 2012 [35], Neural Network (NN) based models have become increasingly more reliable at performing Computer Vision (CV) tasks, and therefore have been increasingly adopted at industry-level applications where a high level of reliability is needed [45]. Indeed, model accuracy has nearly quadrupled in a matter of only 10 years since 2012 [71]. As of now, NN-based models are widely accepted as the most reliable method for CV applications [45], from retail [55, 15, 47], to medical imaging [60, 54, 32], to other domains [19, 48]. To be more precise, NN-based CV models will be defined as any type of model that receives an image as input, which is composed by an array of pixels, and outputs a prediction of some kind, using a neural architecture. Depending on the task at hand, the kind of prediction will vary. In Object Recognition (OR) tasks it is whether an entity in an image belongs to a given class; in Object Detection (OD) tasks, it is the class to which an entity belongs in the image, and it is location; and in Object Segmentation (OS) tasks it is to which class does each pixel in the image belong to.

These models are inherently data hungry. They require at least 1,000 images to train a model from scratch to recognize one single object [65], and this number increases as the number of different objects to recognize increases, and as the scene in which we wish to detect objects becomes more specific. In addition to taking the photos, each of the images requires a corresponding annotation, which varies according to the type of task we wish our model to perform[1], and which are needed for the model to

[1] In the case of OR tasks, the annotation indicates whether a single object in an image belongs to a given class; in the case of OD tasks, the annotation is the bounding boxes surrounding the objects in the image and their corresponding class; and in the case of OS tasks, the annotation of a single image is a mask of pixels on top of the image that indicates which pixels of the image correspond to which class.

learn to make correct predictions. Manually creating these annotations for thousands of images becomes firstly, a crucial step in the CV model development, but secondly, a cumbersome task that requires going through every image and providing its corresponding annotation. A single annotation of an image with a few classes can easily take a few seconds, which translate into a few weeks of annotation for a dataset of a few thousands of images. This is a very time-consuming task.

In order to address this annotation bottle-neck, multiple possible solutions that specifically speed up the data annotation step have been introduced. Most commonly, these involve crowd-sourcing the annotation process, handing out the task of annotating images to hundreds of people through apps or services such as the Amazon Mechanical Turk [16], online surveys [34], or simply hiring personnel [59]. Perhaps smarter solutions have been proposed, where assisted annotation tools have been developed to speed up the process. These go from simple user interfaces [34], to smarter Semi-Unsupervised models that assist the user during annotation [7, 9], or even annotate them themselves, and only further human verification is needed [10]. Although clever, these methods only employ sophisticated ways of using manpower to annotate the images, so the task is not fully automated in any case.

An alternative, more promising solution is to control the full data collection pipeline by synthetically generating images and annotations. This method seeks to replace the manual real-data collection process using automatic Synthetic Data Generation (SDG): generating images and annotations with a given system, later used to train a NN-based CV model tested and validated on a real life domain. However, for these models to perform well, there needs to be a high degree of similarity between the virtual or *synthetic* and real domains. The *domain gap* is defined as the dissimilarity between domains [46] caused by the difference in layout[2] or *content gap*, or by the difference in appearance between the synthetic domain and the real domain or *appearance gap*. The task of reducing these dissimilarities will be defined as *domain alignment*. For instance, methods such as [69, 41, 28, 27, 39] address the appearance gap by stylizing synthetic images (the *synthetic dataset* or *generated dataset*) to closely resemble a set of images from the real world (the *target data* or *target dataset*). The main trend of methods use variants of Generative Adversarial Networks (GANs) [23]. The domain alignment performed by these models is done at a pixel-level, adapting the source data style to the same as the target data.

In our work, we focus on the solutions that address the content gap using a 3D

[2]This corresponds to the layout of the 3D scene and the type and number of objects present in it.

rendering software, which we will refer to as a *3D rendering engine* or simply *renderer* to automatically generate an arbitrary amount of images, but maybe more importantly, annotations from a 3D scene, since all object coordinates and information are known [18, 67, 33, 56]. Early success from this approach has been shown in areas such as autonomous driving [20, 51], robotics [63, 33, 49], and others [57, 14, 13]. Using a renderer however has significant challenges of its own. For example, Hodan et al. [25] have proven that the design decisions necessary to create a faithful scene result in a significant expense of time that defeats the purpose of SDG. To circumvent this expense, Unsupervised Learning methods such as Domain Randomization [64], and Domain Separation Networks [12] have been developed. These methods, however, do not generate data that respects the real-world constraints, or that follows the same layout in the real data (e.g. a couch shouldn't be in top of a chair in a living room scene).

These challenges are addressed by methods such as [44, 21] directly addressing the content gap by representing the 3D scene's parameters as probabilistic grammars [70]. A Graph Neural Network (GNN) [68] is then used to encode the graph representations of the scenes which are then fed to the simulator to generate synthetic images automatically from these graphs. Specifically, [52] uses the loss from training a downstream task with the generated data and performs domain alignment by minimizing this loss. Due to the non-differentiable nature of current rendering engines [30, 44], the gradients from this loss are estimated using the finite-differences gradient estimator [58], which are the updates to the parameters of the GNN. This trains a model on synthetic images optimized to perform well on a given downstream task. Moreover, Meta-Sim [29], and Meta-Sim2 [17] explicitly represent the content gap by measuring the distance between the generated distribution and the target distribution using a distribution similarity metric. This content gap loss, or distribution loss, is added to the downstream task's loss. This trains a system that can create synthetic data that strictly follows the target data distribution, and that performs well on a given task. In our study, we decide to specifically focus on the Meta-Sim model, since further work needs to be done to shed light into how the content gap is reduced as the number of parameters from the 3D scene is increased, and how the overall learning behaves, given that gradient approximation is being used. We justify this in depth in the following section, starting by fully explaining the architecture.

Chapter 3

Methodology and Implementation

In this section we dive deep into the Meta-Sim architecture to give the reader a comprehensive definition of the architecture as an Unsupervised SDG architecture, followed by a critical discussion of the model pointing out *where* our study wishes to extend the original study. Subsequently, we provide an overview of the implementation of the model, but more importantly, our implementation with a separate rendering engine. We finish the section by explaining *how* our study wishes to extend the original study.

3.1 The Meta-Sim Architecture

The main goal of Unsupervised SDG architectures is, given a small set of real images taken from a real scene (the target dataset) to learn to produce a synthetic dataset of images from a synthetic scene that are similar in content and appearance to the real dataset (the generated dataset) which at the same time can be optimized to perform well on a downstream task tested on the target dataset. To achieve this goal, Kar et al.[29] developed the Meta-Sim architecture, an unsupervised learning method that uses a Probabilistic Scene Grammar (PSG) to define a hierarchical representation of the real scene. Meta-Sim uses the PSG to teach a a Graph Convolutional Network (GCN) autoencoder [31] to generate graph representations of the scene. These graph representations are rendered as images by a 3D rendering engine to generate a dataset D_g that is compared to the target dataset $D_t = X_t$ using a distribution similarity metric. The distance between both distributions, the generated distribution and the target distribution, is used as a loss that gets minimized during the training of the parameters θ of the GCN, the Meta-Sim training. Finally, the architecture can be optimized to

generate data to train a TaskNetwork[1] T_ϕ (or simply TaskNet) validated on the annotated target dataset $D_t = (X_t, Y_t)$. This section will walk the reader through the inner workings of this architecture, first providing an overall graphical explanation, and then a more formal, mathematical definition of the model.

3.1.1 Understanding the architecture graphically

Let us express the model graphically in Figure 3.1, where we can visualize the information flow when passing through its components. As a general rule, the architecture is trained in two main stages, first the *GCN pretraining* depicted by dashed arrows in green, and then the *Meta-Sim training* or *MMD training* depicted by black arrows for its forward pass, and red arrows for its backward pass. The GCN pretraining consists in training the GCN at encoding and decoding batches of graph representations initially sampled from the PSG. The pretraining is done for a given number of *pretraining epochs*, where the architecture learns how to classify each graph node's categorical values, and to reconstruct each graph node's continuous values.

Theoretically, the pretraining initializes the GCN at reproducing sensible graph representations for the following training stage, the Meta-Sim training. For this stage, during the forward pass the GCN generates batches of graphs fed to the renderer with a loaded 3D scene strictly compatible with the PSG. The renderer must have the capability of parsing each node in the graph, for every graph in the batch, which corresponds to an instance of the attribute values of each entity in the loaded 3D scene. Each graph in a batch is rendered as an image, for all batches in the dataset, generating a full dataset D_g, and used to train the TaskNet. After training, the TaskNet is validated against the target dataset D_t from which we obtain a performance *score*.

Crucially, this graphical representation allows us to see how the full backpropagation must be done for all parameters from the GCN (θ), TaskNetwork (ϕ) and Renderer (γ) to be compatible. In red we can see that after the forward pass, the backward pass uses the generated dataset, the target dataset, and the *score* to calculate the *distribution loss* or *MMD loss*, and the *task loss*. The former is calculated by translating both datasets to feature maps using the InceptionV3 network, and then measuring their distribution similarity using the Maximum Mean Discrepancy (MMD) [24] metric. The latter is calculated by using the REINFORCE score function estimator [66] which reformulates the *score* obtained from the TaskNet with parameters ϕ as a task loss

[1] This is usually OR, OD, or OS.

compatible with the GCN parameters θ^2. Both losses are summed up to obtain a single *Loss* before being passed through the renderer using the Finite-differences gradient approximation method. The full loop is trained for a given number of *Meta-Sim training epochs*.

Finally, this representation allows us to see the modular nature of the architecture where, theoretically, we can use different renderers, PSGs, and 3D scenes. This graphic representation also shows that it is possible to omit using the task loss and TaskNet altogether, and only use the distribution loss to train the architecture.

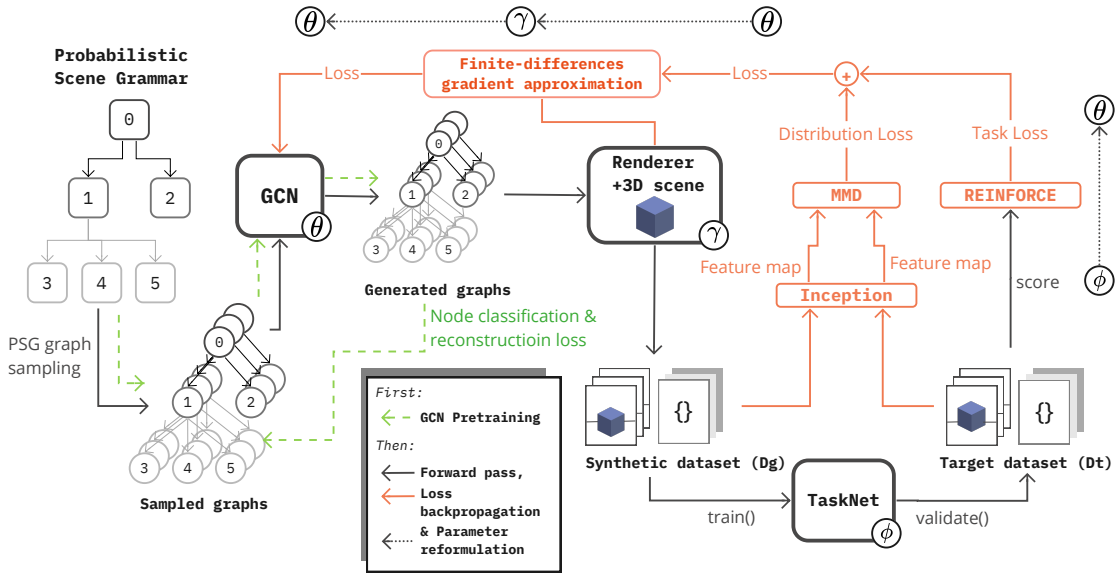

Figure 3.1: **Meta-Sim**: Graphical representation of the Meta-Sim architecture. First the GCN is pretrained for a given amount of *pretraining epochs*, then the Meta-Sim training starts.

3.1.2 Understanding the architecture formally

Let us now dive deeper into the formal definition of the model, starting by the definition of the PSG, which is a hierarchical graph representation of the 3D scene from which we will generate images [42]. Each node in the graph is either a parent of one or multiple nodes, or a child of a node. Each node is an entity in the scene, and each entity has *mutable* or *modifiable attributes* —attributes that the architecture can manipulate to minimize its objective function— and immutable attributes. As a general definition, a *scene* or a *3D scene* normally contains multiple entities of different classes such as a cube, the floor, a camera to take pictures of the scene, light to illuminate the scene, and

[2] According to this formulation, the *MMD loss* does not need labels, and the *task loss* does.

the scene itself. Each entity can have mutable attributes such as the cube's rotation, or the light's illumination, and immutable attributes, such as the cube's scale, or the light's pose. The PSG defines the range of values that each mutable attribute from each entity can take. We formally define the PSG structure used across the study in Figure 3.2.

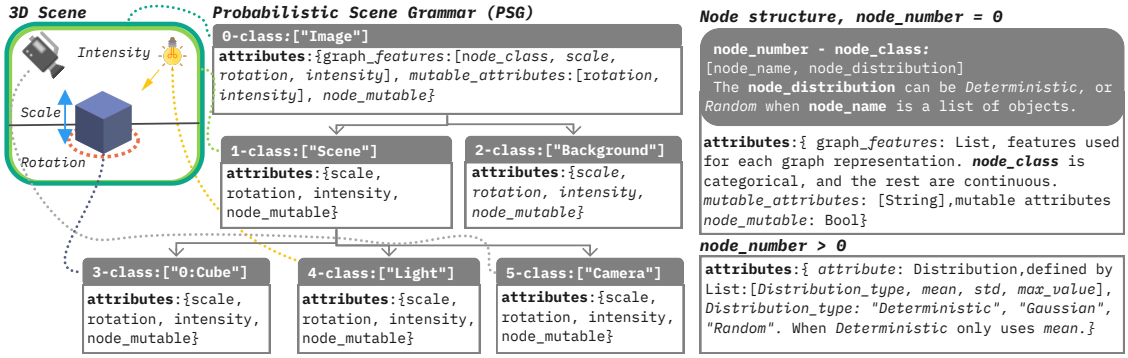

Figure 3.2: **PSG**: Probabilistic Grammar Representation of a 3D scene, with explained node structure. This scene can be real or synthetic.

The first stage of the architecture's training involves pretraining the GCN autoencoder's parameters θ to generate sensible graph structures that represent meaningful variations of the 3D scene. Each graph structure, or simply *graph*, is an instantiation of the PSG with specific attribute values sampled according to the PSG's predefined distribution parameters. During this pretraining stage, we train the GCN for a given amount of *pretraining epochs* to learn to instantiate graphs in accordance to the PSG. This involves initially training the GCN to minimize two losses, the *classification loss*, the error of the network at classifying each node class, and the *content loss*, the error of the network at correctly reconstructing each node's attribute values. Both errors are calculated against the graphs sampled from the PSG.

The second stage of the architecture's training starts by using the pretrained autoencoder to generate batches of graphs $G_g^{(b)}$ that are sent to the renderer R_γ with parameters γ. These parameters will remain out of reach to the Meta-Sim architecture, since the current nature of rendering engines makes them intractable and untraceable [30]. The renderer however will have the task of parsing each graph in the batch $G_g^{(b)}$, modify the loaded 3D scene's attributes according to the values in each graph, and return a batch of images $D_g^{(b)}$ to the architecture.

$$D_g^{(b)} = R_\gamma(G_g^{(b)}) \tag{3.1}$$

The first loss from the architecture comes from the *appearance gap* between the generated batches of images $D_g^{(b)}$, and a batch of target images $D_t^{(b)}$, which are repre-

sented as the last pooling layer from the InceptionV3 Network [61] pretrained on the ImageNet dataset [53]. The difference between both feature representations, the distribution loss, is measured using the Maximum Mean Discrepancy distribution matching metric [24], using a Gaussian kernel. This is best known as the Kernel Inception Distance (KID) [11].

$$\mathcal{L}_{Dist} = MMD_{\theta}(D_g^{(b)}, D_t^{(b)}) \tag{3.2}$$

The second loss in the architecture comes from the performance *score* of the TaskNetwork T_ϕ which can take the form of any Computer Vision Neural Architecture that performs tasks such as object classification, object detection, or object segmentation. According to the task, this score can vary from being a simple accuracy metric, to the precision of the network at a given Mean Intersection over Union (mIoU) threshold. T_ϕ is trained on the full generated dataset D_g and validated on the target dataset D_t.

$$T_\phi = \texttt{train}(T_\phi,\, D_g), \tag{3.3}$$

$$score_\phi = \texttt{validate}(T_\phi,\, D_t), \tag{3.4}$$

This *score* is reformulated as a negative log likelihood loss by using the REINFORCE score function estimator, which subtracts a moving average of the score to reduce variance in gradient calculation, and multiplies this by the log likelihoods of the generated graphs G_g[3], which from Eq.3.6, is a reformulation that translates a score of parameters ϕ to a loss of parameters θ.

$$REINFORCE(score_\phi) = -(score_\phi - \overline{score_\phi}) \cdot logp_{G_g\theta}, \tag{3.5}$$

$$\mathcal{L}_{Task} = REINFORCE(score_\phi) \tag{3.6}$$

The task loss is now compatible to be summed with Eq. 3.2 to obtain a total *Loss*.

$$\mathcal{L} \;=\; \mathcal{L}_{Dist} + \;\mathcal{L}_{Task} \tag{3.7}$$

The objective during the training of Meta-Sim is to minimize this *Loss* during a given amount of *MMD training epochs* by generating data that minimizes the *distribution loss* while maximizing the TaskNetwork *score*. In general terms, this means minimizing the appearance gap between the generated images and the target images, while improving performance of the generated data in a given computer vision task. Specifically, given the task of generating a synthetic dataset D_g of N batches $G_g^{(b)}$ of generated graph samples, using a renderer R_γ and a target dataset D_t from which we randomly sample batches $D_t^{(b)}$, we wish to learn θ to generate a synthetic dataset D_g that satisfies the following optimization problem

[3]We will not go into detail into how this is done since it is a non-essential detail to our study, we refer the keen reader to *provide citation or refer to Appendix*

$$\theta = \arg\min_{\theta} \frac{1}{N} \sum_{b}^{N} \mathcal{L}_{Dist}^{(b)} + \mathcal{L}_{Task}^{(b)} \tag{3.8}$$

$$s.t. \begin{cases} \theta = \arg\min_{\theta} \frac{1}{N} \sum_{b=0}^{N} MMD_{\theta}(R_{\gamma}(G_g^{(b)}), D_t^{(b)}) \\ \phi = \arg\max_{\phi} \frac{1}{N} \sum_{b=0}^{N} score_{\phi}(R_{\gamma}(G_g^{(b)}), D_t^{(b)}) \end{cases} \tag{3.9}$$

This however is not the typical optimization objective, given that the parameters γ from the renderer R_{γ} are non-differentiable due to the intractable nature of current rendering engines [30]. The gradients for this function need to be approximated using a finite-difference gradient approximation [58], where the gradient of the loss from a given batch of generated graphs is approximated by perturbing each generated graph g using a small delta δ, and measuring the finite difference between the perturbations after passing these through the renderer[4].

$$\nabla_{\gamma}^{(i)} \approx \frac{1}{2 \cdot \delta}(R_{\gamma}(g^{(i)} + \delta) - R_{\gamma}(g^{(i)} - 2 \cdot \delta)) \tag{3.10}$$

3.2 Extending Meta-Sim: what we seek to shed light on

From what we just explained, the above architecture is a Synthetic Data Generation architecture that uses Distribution Matching to generate data of a 3D scene that closely resembles data from a similar real scene. This means that the architecture, as defined above and in the original study, should be able to be reproduceable with different 3D scenes, with varying *complexity*— varying number of entities in the scene and mutable attributes. In other words, the architecture should be *scalable*— transferable and adaptable to different scene complexity. The original study however provides inconclusive data for us to confirm that this architecture is indeed *scalable*, since it only implements the above proposed architecture in one 3D scene, with no available implementation to the public. The scalability testing reported in the study reported overall results for the TaskNet performance when adding types of mutable attributes. The results showed improved performance when using Meta-Sim, compared to images from a 3D scene with random entity attribute values. These improved results however can be attributed to the photorealism of the 3D scene, and the pretraining of the GCN, and not to the Meta-Sim training itself. Indeed, for a reader to conclude that the architecture works at Distribution Matching, the study would need to report results such as a comparison

[4]This equation is according to the implementation given by [29], the original finite-differences equation $\nabla_{\gamma}^{(i)} \approx \frac{1}{2 \cdot \varepsilon}(R_{\gamma}(g^{(i)} + \delta \cdot \varepsilon) - R_{\gamma}(g^{(i)} - \delta \cdot \varepsilon))$, does not use a factor of 2, and uses *epsilon* to denote the perturbation magnitude, and d as the direction of the perturbation in optimization space [1].The factor 2 from Equation 3.10 should be excluded in future work.

between the generated distribution and the target distribution, and a plot of the MMD loss converging towards a local minimum.

Furthermore, using finite-differences approximation has been proven to be an unstable method for gradient estimation, and even more unstable when increasing the dimensions of the data [58]. Indeed, as [30] mentions it, and [43] proves it, determining the value of the small delta d for each application is very difficult, and explains that the correct finite-differencing delta on images is pixel-dependant, which may mean that when comparing disturbed images, the delta might be too large for some pixel areas in the images, and too small for others. This might render the learning unstable, and may explain why no results of the MMD loss were originally reported. According to multiple studies [58, 30, 43], it has been proven that finite-difference estimation is a tricky endeavor. The Meta-Sim study only focuses on reporting the TaskNetwork performance, when the crucial metric is the distribution loss, and the comparison of the generated distributions versus the distribution of the target distribution, since at its core the Meta-Sim problem is a Distribution Matching problem. This also undermines the study's decision to use a Gaussian kernel for the MMD loss, which is only seconded by [40], where the MMD loss is used for MNIST digit and face matching on a differentiable rendering function context, but never in the context of 3D scenes using a non-differentiable renderer. This renders (pun intended) the use of MMD with a Gaussian kernel arbitrary and possibly unfit for the theoretical task Meta-Sim is trying to do, which is using this metric to update the GCN autoencoder's parameters to generate graphs. This is something to keep in mind.

All of this to say that this study will focus in completing a rigorous scalability analysis of the Meta-Sim architecture, in the hopes of conducting a thorough distribution matching study on increasingly complex 3D scenes, using a publicly available 3D rendering engine. Originally, we hypothesize that the architecture's distribution matching performance will decrease as the complexity of the architecture increases. We define this performance as the architecture's ability to make the MMD loss converge, and as a qualitative evaluation of the similarity between the generated distribution after Meta-Sim training, and the target distribution. We hope to provide the literature with a systematic study that provides conclusive results to whether this type of architectures have potential to be used as data generation frameworks.

3.3 Implementation: author's baseline on MNIST case

We will depart from the only available implementation which was provided by the authors [5]. This implementation is only implemented to generate MNIST [37] digits, but it is an excellent starting point that contains all the necessary components mentioned previously. Globally, this implementation contains 3 main classes: *MetaSimTrainer*, *Models*, and *Data*. And one configuration file *Configuration*. *MetaSimTrainer* inherits all methods and attributes from the two classes, and all information regarding the configuration of the network from the configuration file. *Data* contains the GraphSampler, which samples graphs from the PSG, the TargetData, and the Renderer, which in this case has type="MNIST", since the original implementation's renderer generates digits. *Models* contains the GCN, the MMDInception calculator, and the TaskNetwork. Figure 3.3 illustrates this structure.

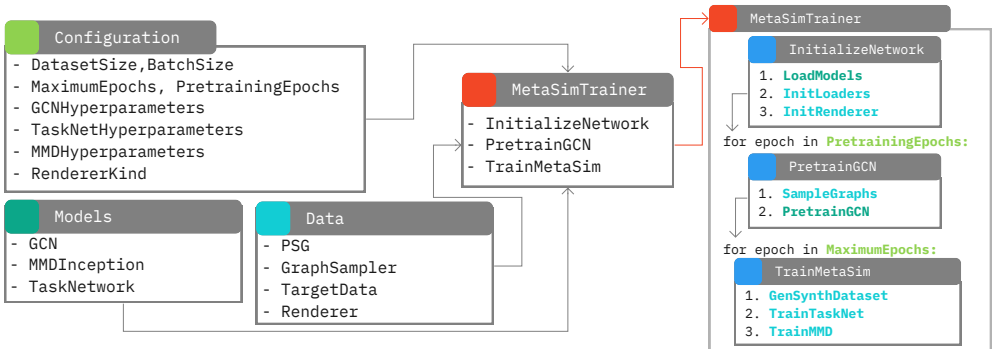

Figure 3.3: Meta-Sim implementation structure.

To train Meta-Sim, *MetaSimTrainer* contains 3 methods that are run consecutively in the main loop, InitializeNetwork initializes all models imported from the *Models* class with the hyperparameters from the *Configuration* file, and all data loaders and renderer from the *Data* class. Following initialization, the GCN is pretrained for PretrainingEpochs epochs, and finally, the Meta-Sim architecture is trained for MaximumEpochs epochs. The latter contains 3 main steps, where the first, GenSynthDataset generates the full synthetic dataset that is used to train the TaskNet at the second step. The third and final step is in charge of the MMD training, and uses the MetaSim model to generate graphs that are then passed to the Renderer which then generates images that are compared to the target dataset during the MMDLoss calculation. This loss is summed up with the TaskLoss, which is a reformulation of the taskAccuracy obtained in step 2, and then backpropagated through the network to update Meta-Sim's and therefore the GCN's parameters. The complete pseudo-code for the 3 steps is shown in Figure 3.4.

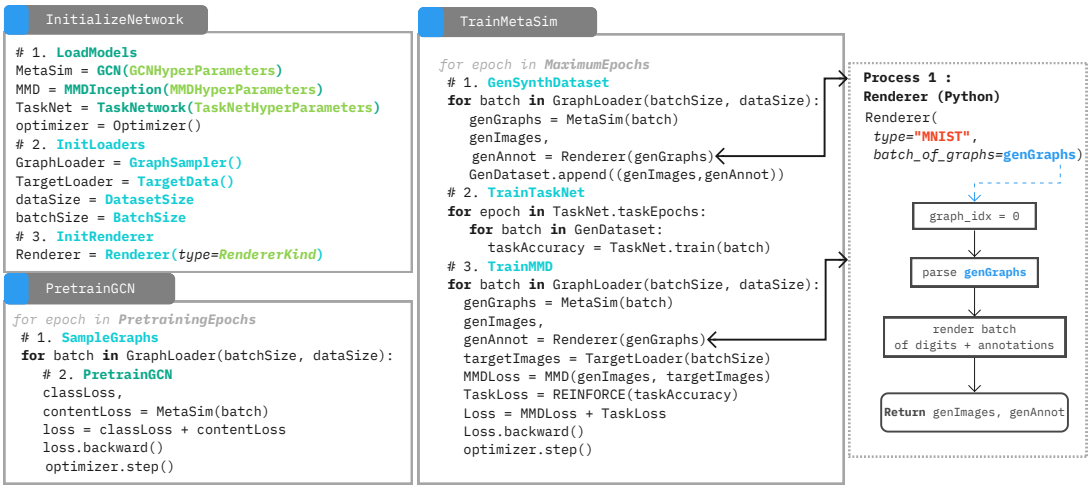

Figure 3.4: Pseudo-code for the 3 necessary blocks to train Meta-Sim. The Renderer process gets called during the synthetic dataset generation and MMD training.

3.4 Implementation: our Custom 3D dataset

From the above implementation, the `Renderer(type="MNIST")` is a method implemented inside the Data class, which itself is implemented in Python, as shown in Process 1 in the right side of Figure 3.4. In this implementation, the authors hardcode a digit renderer in Python, which makes all interactions with the renderer very simple and trivial, since it is a sequential process that gets run inside Python. This of course, is not what we want, since we'd like to use any other commercially available 3D renderer, which is normally a black box outside of Python, and is not evidently accessible.

In our study, one of our main contributions is a general implementation framework for attaching any renderer to the Meta-Sim architecture, but more generally any ML architecture that needs to generate data inside its pipeline. This implementation is not provided by the authors, nor anywhere in the literature [3, 4, 2]. In our case, we decide to attach Unity [6], which is commercially available and optimized, so rendering is very fast. This implementation is explained in the following section.

3.4.1 Renderer: attaching Unity

Following the conventions from the previous section, we want to add a second implementation for a `RendererKind=="Custom"`. In this case `Process 1`, which is the process called when the `Renderer` method is called, is going to be running in parallel to a second process, `Process 2`, which is the rendering engine, and both processes will be publishing files to an `Interface` composed of file directories. Process 1 publishes the generated batch of graphs in a JSON format to *graph_dir*, and Process 2 publishes

the generated batch of images and annotations to *image_dir*. Process 1 checks whether *image_dir* contains a full batch of images, and Process 2 checks whether *graph_dir* contains a full batch of graphs. When true, both continue to execute the following actions. When the `Renderer` method is not called, Process 2 is still running in the background, parsing *graph_dir* which is always empty, unless the `Renderer` is called.

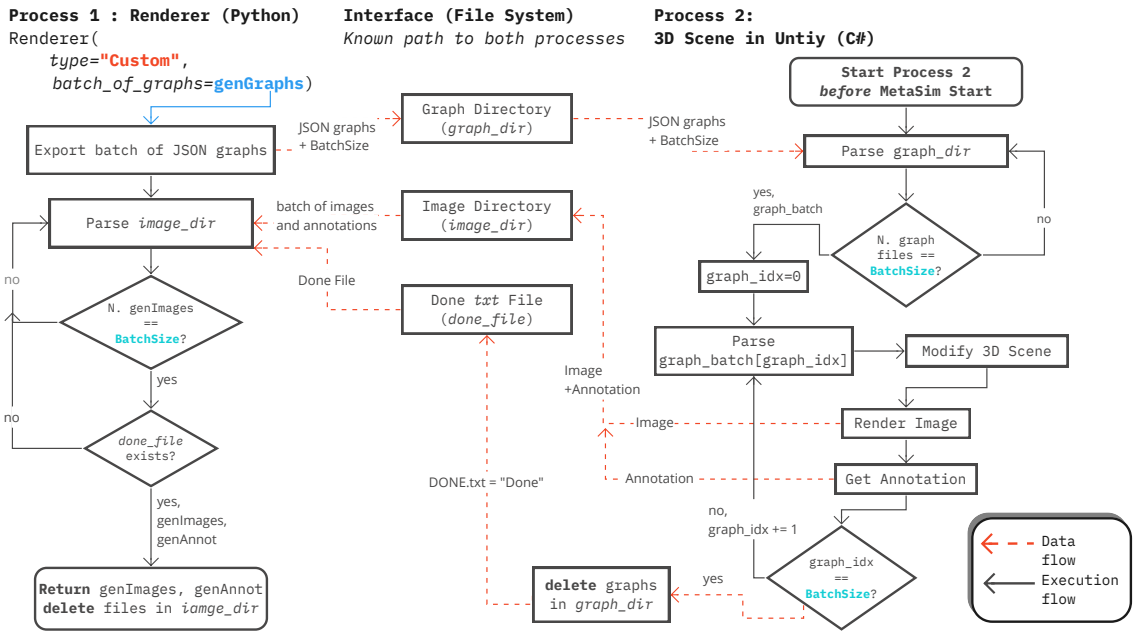

Figure 3.5: Architecture logic for interaction between the Renderer class in Python whithin the Meta-Sim process, and the 3D rendering engine.

Finally, in order for Meta-Sim to efficiently generate images with the attached 3D Renderer, Process 2 needs to be started before MetaSim is run, and needs to be kept running until `TrainMetaSim` is done training. This is important because Process 2 is a parallel process that when started, builds the rendering scene, and while running surveys whether there are graphs in *graph_dir*. In order for the architecture to run efficiently, the scene building[5] needs to be done once[6]. The complete architecture logic is shown in Figure 3.5. This architecture design allows for fast batch rendering, generating a batch of 50 128x128 images in 125 miliseconds in the Unity rendering engine. The speed of the implementation is crucial, since we will run several experiments with hyperparameter modifications in order to evaluate the performance of the architecture when increasing scene complexity.

[5]Building a scene refers to initializing the renderer with a loaded 3D scene.

[6]A previous version of this implementation called the building of the scene every time the Renderer class would be called, this slowed down the MetaSim training by a factor of 40.

3.4.2 How we intend to test the Meta-Sim architecture under the new Custom 3D scene

Now that the architecture is attached to the Unity 3D rendering engine, we can define a 3D scene that will allow us to test the architecture under increasingly complex settings, this scene is explained in Figure 3.6, and contains all possible Scene Classes in orange, and all possible Scene Attributes in green. Our testing methodology will be guided by starting from the simplest scene, up until the most complex one. This scene is inspired from the Toyota Light dataset (TYO-L or TYOL) where an image contains a single object class [26]. We will refer to this scene as the *3D-TYOL* or *custom* scene.

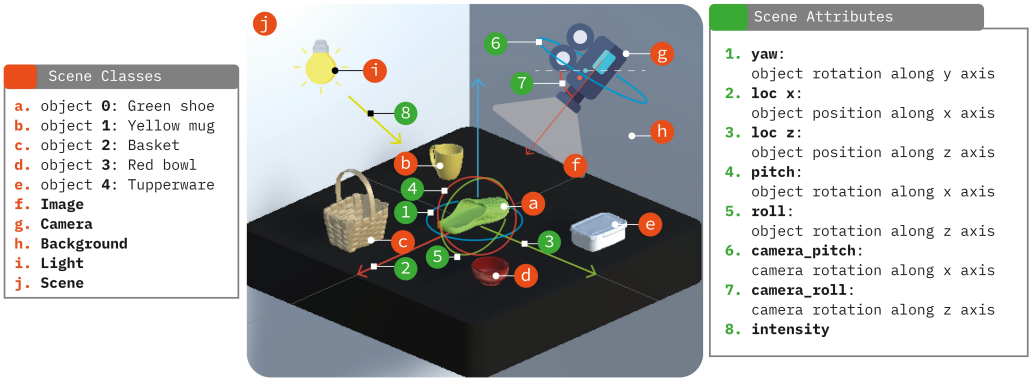

Figure 3.6: Unity 3D scene insipired from the TYO-L dataset [26] with the nomenclature used for all experiments to generate data, and to generate the target dataset.

The starting point will be the *single-class setting*, where we only include one of the 5 object classes, object 0, and the rest of the classes are excluded in the scene. We will run experiments on the *single-class setting* with 1, 3, 5, and 8 Scene Attributes, included in the same order as listed in Figure 3.6. We will then finalize our experiments with the *multi-class setting*, which will include all 5 object classes, and will incrementally add Scene Attributes in the same way as the *single-class setting*.

For all of these configurations we will train the Meta-Sim architecture only using the MMD loss, discarding the `TrainTaskNet` step altogether. We prove correct convergence using only this loss with initial experiments performed on the MNIST dataset, whose results can be found in Appendix **??**. This will reduce the amount of components of the architecture, and therefore failure points, while still theoretically allowing convergence of the network [29]. In the same spirit, we will use synthetically generated target images from the same 3D scene, where we will control the target distribution of attributes. All generated and target images will be 128x128 RGB images. At each epoch, the architecture will generate a dataset of 1,000 images and annotations.

Chapter 4

Analysis and Evaluation

As interesting and as promising the Meta-Sim architecture may seem, during this section we wish to thoroughly explore its scalability under ever more complex settings, in order to find out whether the architecture works on different 3D scenes than the one used in the original study. For this, and as explained in Section 3.4.2, all experiments are conducted using only the MMD loss. We use synthetically generated images as the target distribution, making the domain alignment problem much simpler by remove the synthetic to real domain gap. We start by experimenting with a single class setting, increasing its number of modifiable attributes from 1, all the way up to 8, and we repeat the same study on a 5-class setting.

4.1 Experiments on a single-class setting

4.1.1 1 mutable attribute: learning to rotate an object 90°

We start from the simplest case, where we attempt to generate 1,000 images that fit a target distribution of 1,000 images[1], where the object's *yaw rotation*, or simply *yaw*, distribution is centered around 90°. This means that, after a given amount of MMD training epochs, our model should be able to learn to generate images where the object is rotated approximately 90 degrees along the *y axis*. We will first explore how the architecture fits different target distributions, departing from different initial distributions. This will allow us to understand which is the best type of distribution we should use for the architecture to work properly. We will later on experiment with ways to improve the architecture's learning performance to generate wide distributions.

[1] We will use the same amount of generated and target images for the rest of the study.

4.1.1.1 Fitting a Dirac target distribution with the MNIST configuration

In order to transition smoothly from the MNIST case to the 3D-TYOL case, we start by using the same learning hyperparameters[2] as the ones used in the MNIST experiments. We pretrain the GCN autoencoder for 7 epochs, and we plot its losses during this stage. We plot the pretraining results in Figure 4.1, and given that the loss converges to a low of under 0.05, we assume that the architecture learns to generate graphs that represent our 3D-TYOL scene correctly.

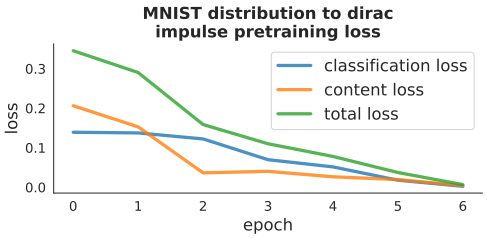

Figure 4.1: GCN autoencoder loss performance after 7 pretraining epochs.

After pretraining, we train the current configuration for 500 MMD training epochs and we record the generated images, and their samples for their *yaw* attribute value, every 125 epochs. In Figure 4.2, we plot all *yaw* values for the generated images at each epoch, we will call this the generated *yaw* distribution. Separately, we also plot the generated *yaw* distribution after the GCN pretraining stage and before MMD training (at epoch 0 of MMD training), and also the target distribution, which will be depicted in bright pink in all distribution plots for the rest of the study. We want to be clear that the generated distribution at epoch 0 of MMD training is the distribution learned from the PSG after GCN pretraining. From epoch 1 onwards, the generated distributions are affected by the MMD loss. This remains the same for the rest of the study.

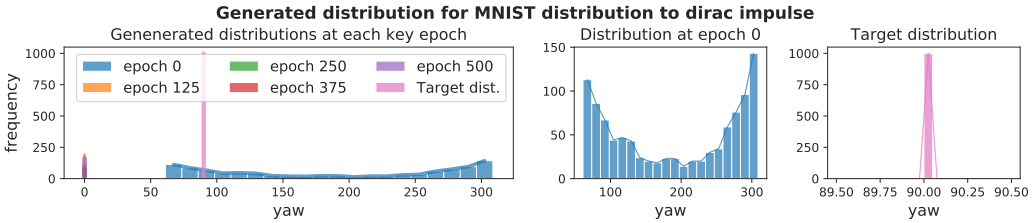

Figure 4.2: Yaw values of generated samples every 125 epochs of MMD training, compared to the target distribution. We observe an inverted distribution at epoch 0, and a Dirac impulse centered around 90° for the target distribution.

From this plot we observe that after epoch 0, all other generated distributions are collapsing to angles very close to 0. This is a clear sign of mode collapse [62], and we will further analyse its cause in the following experiments. We can also observe that epoch 0 distribution is a shifted normal distribution. This is because the MNIST PSG was defined with the *yaw* attribute as a Gaussian of $\mu = 0°$ and $\sigma = 90°$. This distribu-

[2] All hyperparameters available to tune Meta-Sim are explained in Appendix A.

tion will produce negative values. However, after sampling, the architecture takes the modulus of the sampled value and the maximum value. According to this, any negative value will be mapped to its positive counter part. This effectively produces an inverted distribution. In the following experiments we will replace the target distribution to be a Gaussian distribution of pre-defined mean and standard deviation, so the distribution matching can be from Gaussian to Gaussian, and not from Gaussian to Dirac, which makes the problem a lot harder [50].

4.1.1.2 Fitting a Gaussian distribution departing from a Gaussian distribution, with 1 pre-training epoch

We define a new initial distribution with $\mu = 180°$ and $\sigma = 10°$. This will avoid the inversion effect we observed previously. We decide to pretrain the GCN only for 1 epoch to allow the MMD training to attempt to fit the target distribution, with no influence from the pretraining stage. We train this configuration for 100 epochs, and record the generated images and distributions every 25 epochs, which are depicted in Figure 4.4.

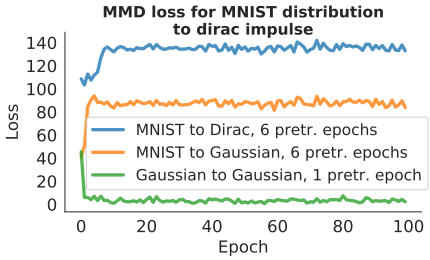

Figure 4.3: MMD loss performance after 100 epochs of MMD training for the 3 distribution matching experiments.

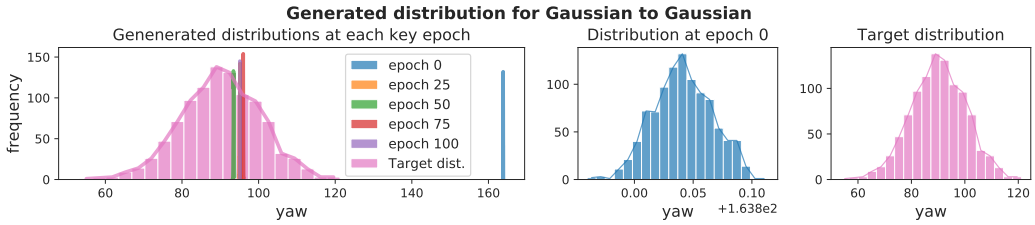

Figure 4.4: Yaw values of generated samples for Gaussian to Gaussian experiments. We observe correct convergence towards the target mean. We also observe that the generated distributions are Gaussians.

As we can see, the architecture finally attempts to fit the target distribution by shifting its mean from $\approx 165°$, to $\approx 95°$ at epoch 50. This is an important improvement with respect to the previous architecture configurations, since it now learns to generate images that resemble the target ones. We can verify this by looking at the generated images in Figure 4.5.

Furthermore, in Figure 4.3 we plot the MMD loss from this experiment, compared to the one from the previous section, and a third one that uses the current Gaussian

Figure 4.5: Randomly sampled images of generated dataset every 30 epochs of MMD training, and randomly sampled image from target dataset, for Gaussian to Gaussian experiment.

distribution, but departs from the MNIST distribution[3]. We confirm that the current configuration allows MMD loss convergence and, by looking at the gap between the blue and orange lineplots, observe that using a Gaussian target distribution significantly lowers the MMD loss. We further investigate this by looking at the gradient flow through the GCN decoder (since the encoder weights remain frozen during MMD training) as shown in Figure 4.6. Epoch 0 experiences a high gradient, and therefore the drop in loss above. The gradients from the previous experiment seem to vanish after epoch 0, which explains the mode collapse observed in Figure 4.2. On the contrary, the gradient flow from the Gaussian to Gaussian experiment seems much healthier, where the gradients don't vanish immediately, and there is a gradual convergence towards the local minima located at an angle of $\approx 95^\circ$.

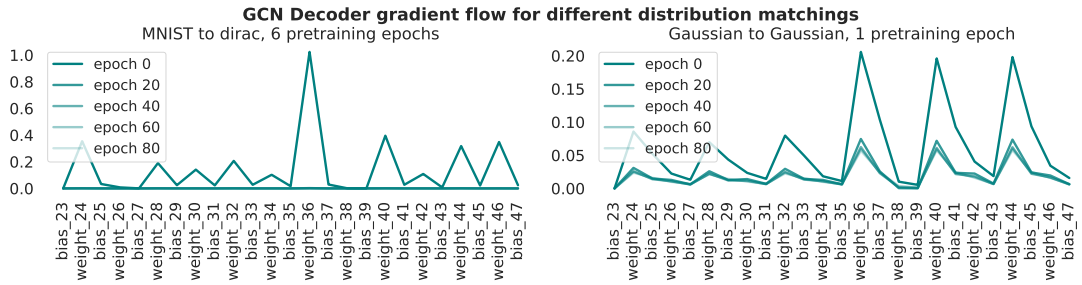

Figure 4.6: Gradient flow through GCN autoencoder decoder for MNIST to dirac, and Gaussian to Gaussian experiments. We observe that the latter has healthier, more stable gradients than the former.

So what is important here is that the architecture is learning the target mean successfully. It is not learning the standard deviation of the target distribution, but this is to be expected, according to the Meta-Sim study the standard deviation of the generated distribution is learned during the pretraining phase, and it is the one specified in the PSG. The generated distribution at epoch 0 *is* a Gaussian, just a very narrow Gaussian. This configuration seems to be the correct one for this experiment to work. However,

[3]The full study for the MNIST to Gaussian experiemnt can be found in Appendix D.

this exploration gives us clear evidence that the Meta-Sim architecture is not a very robust architecture when it comes to generic cases involving 3D assets that are mutable, let's investigate this claim further by attempting to widen the generated distribution.

4.1.1.3 Attempting to widen the generated distribution

Increasing the number of pre-training epochs and using learning rate decay It is now evident that the GCN learns to generate a wider distribution during the pre-training phase, which is the one specified in the probabilistic grammar, and it is the blue distribution generated at epoch 0 of MMD training in all distribution plots in this study. The pretraining however has shown to destabilize learning of the target mean. In this set of experiments we gradually increase the number of reconstruction epochs to widen the generated distribution, while trying to learn the correct mean.

We will start by increasing the number of reconstruction epochs to 5, and 10 epochs, and employing learning rate decay using Step Learning Rate Decay [22], every two MMD training epochs, and we leave all other hyperparameters as per the experiment in the previous section. Based on the evidence from Appendix B and Section 4.1.1.2, the architecture is supposed to converge after 10 MMD training epochs, so all experiments from now on are trained for this long. We hope that, by increasing the number of pre-training epochs, the architecture learns to generate the distribution specified in the probabilistic grammar. We also hope that by using learning rate decay we can control the instability in the gradients. We depict the GCN pretraining loss after 10 pretraining epochs in Figure 4.7.

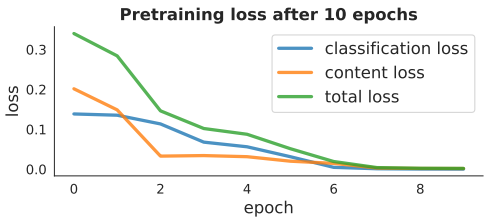

Figure 4.7: GCN autoencoder loss performance after 10 pretraining epochs.

Furthermore, by plotting the generated distributions in Figure 4.8, we can see that the learned distribution is clearly affected by the number of pre-training epochs we use. We can see that the 10 epochs of pre-training help the GCN generate a similar distribution as the one specified in the probabilistic grammar. Furthermore, the use of learning rate decay keeps the distributions from collapsing to $\sigma \approx 0^{\circ}$. This shows that the use of learning rate decay conserves the learned distribution through the MMD training epochs. This can especially be seen when we use 10 pre-training epochs and learning rate decay every 2 epochs.

From the previous plots we can see that the architecture rather learns a closer mean to the target distribution, or learns to generate a distribution as specified in the proba-

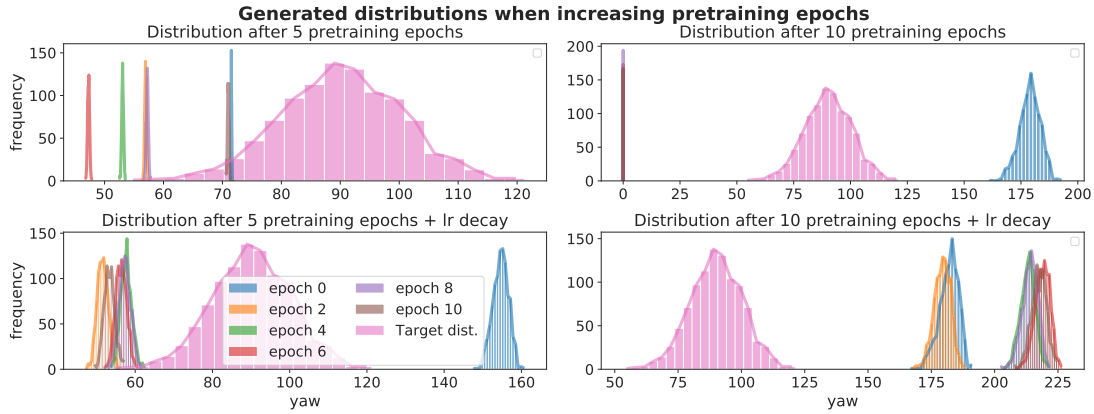

Figure 4.8: Yaw values of generated samples for increasing pretraining epochs experiments and adding learning rate (lr) decay.

bilistic grammar. It however doesn't seem to be converging towards the target mean, and gets stuck in local minima around other angles. We can confirm this from the non-convergence observed in the MMD loss of the 4 cases shown below.

If we look at the gradients, there seems to be no apparent vanishing gradient problem. In fact, the gradients seem to be healthier than the ones in our previous experiment. This may indicate that with this pre-training configuration, the architecture can learn, but the updates are too large, and therefore the learning is unstable. In Appendix E we show that nor reducing the learning rate, or implementing weight decay alleviate this problem. Let us continue our experiments by adding more mutable attributes.

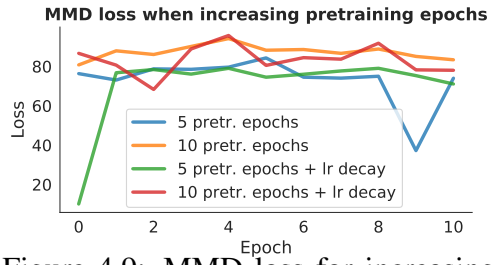

Figure 4.9: MMD loss for increasing pretraining epochs and adding learning rate decay experiments.

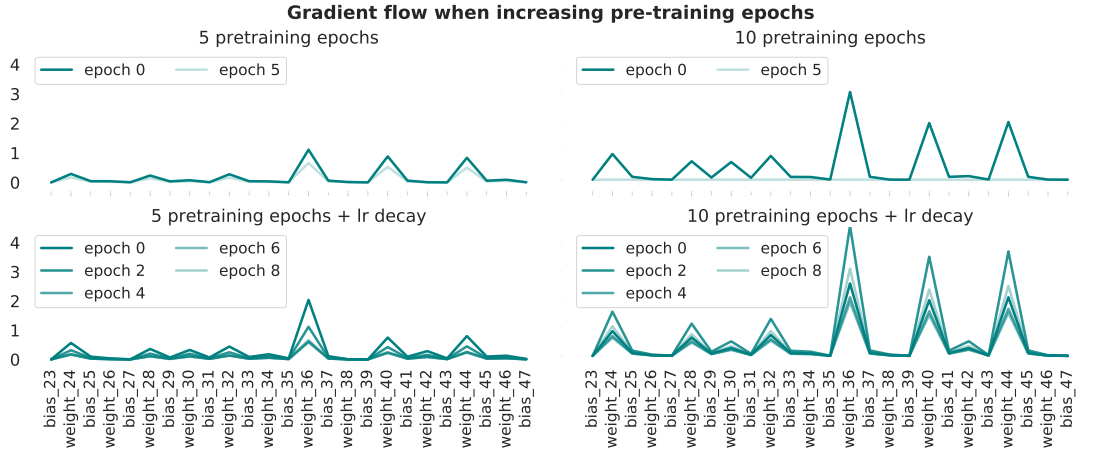

Figure 4.10: Gradient flow through GCN decoder for increasing pretraining epochs and adding learning rate decay experiments.

4.1.2 3 mutable attributes: learning to rotate an object 90° and learn a translation

We thus move to learning a distribution over the *yaw* attribute, but also the *loc_x* and *loc_y* attributes. The last two are initialized with $\mu = 1$ and $\sigma = 0.15$, and the target distribution with $\mu = 1.16$ and $\sigma = 0.025$. The *yaw* distribution is left as before. Initially, we allow the model to fully play with the values of the locations or *translations*, bearing in mind that the model can generate translations that take the object out of frame.

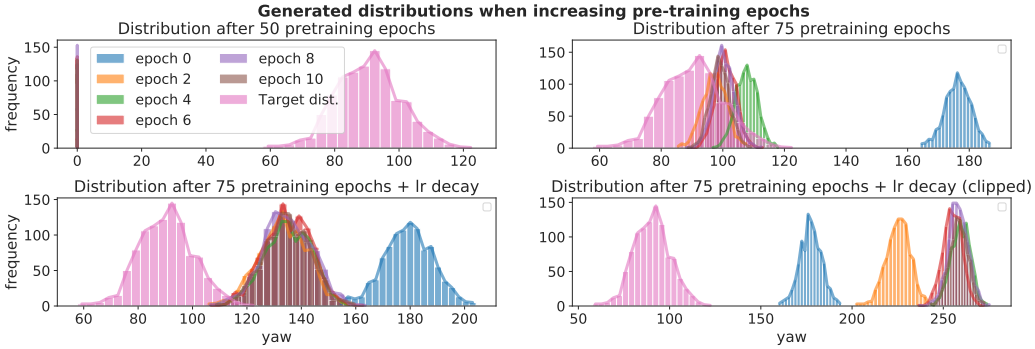

Figure 4.11: Distribution of yaw values for the generated samples when using 3 attributes, compared to the yaw values of the target distribution.

We start by looking at the generated yaw distributions after using 25, 50, 75 reconstruction epochs, and 75 reconstruction epochs + learning rate decay. We use these values since we now want to be more precise in the initialization of the location, because if not, the model will collapse into values that take the object out of frame. From Figure 4.11 we can see that the architecture is not able to learn the *yaw* distribution correctly with 50 pretraining epochs, since the distribution simply collapses to 0. However, after 75 epochs of pretraining, the architecture is able to learn a correct distribution, and get very close to the target distribution's mean.

This however is not enough for us to conclude that the architecture is converging towards the target distribution. Let us now look at the generated distributions for the locations of the object shown in Figure 4.12. We omit showing the distribution for 50 pretraining epochs, since it collapsed with the *yaw* distribution.

From the generated distributions of locations we observe that the network is generating normal distributions stably after 75 reconstruction epochs. However, after epoch 0 the object is completely out of frame. Indeed, after the second epoch of learning, the object is taken out of frame and the learning stagnates in a black screen, without any further updates. To avoid this, in the last experiment we *clip* the locations of the objects

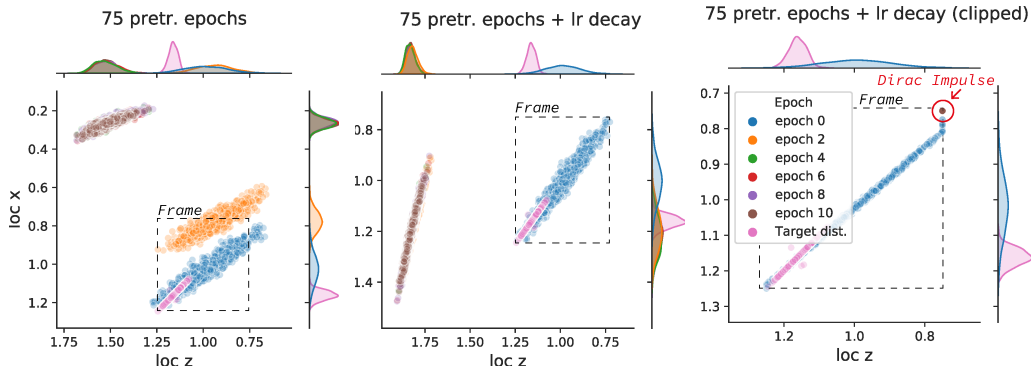

Figure 4.12: Scatter plot showing the distribution of *loc z* and *loc x* values for the generated samples every 2 epochs, compared to the target distribution's values. The frame of the camera is shown as a black dashed line.

to the frame of the camera, so it isn't driven out of frame and the network can still have some distribution input to perform the MMD calculation. This is the red encircled dot we see in the last plot, where all distributions are being mapped to that single spot for the rest of the training epochs.

As we can see from the MMD losses in Figure 4.13, there is still no convergence from the architecture when increasing the number of modifiable attributes to 3. Using 75 pretraining epochs helps the network converge during the first epoch but the next update is still so high that it brings the object out of frame. The architecture continues to have high gradients, as shown in Figure 4.14, which in turn just completely offsets the learning to local optima, which in our case is a the default black background.

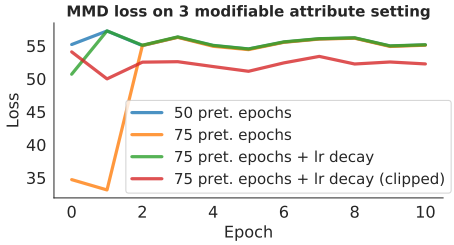

Figure 4.13: MMD loss for the 4 experiments performed using 3 attributes and trained for 10 MMD training epochs.

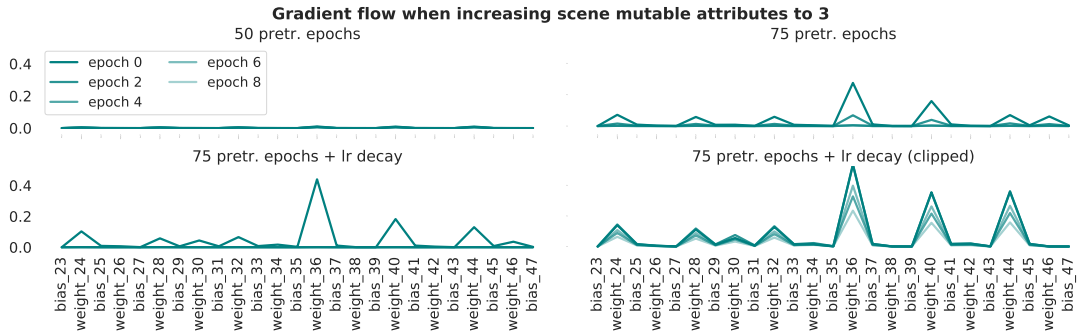

Figure 4.14: Gradient flow through GCN decoder for experiments using 3 attributes. We observe vanishing gradients for all but one of the experiments.

From Figure 4.14 we can see that the gradients for all experiments except the

clipped one are vanishing to 0 after the second epoch. This is to be expected according to the observed distributions, since they are being driven out of frame, except for the last experiment. Indeed, this indicates that the MMD training is providing very large updates, and that the gradients at each epoch need to be kept in between 0.1 and 0.25 to allow healthy learning. It thus becomes an optimization stabilization problem, since the gradients from the finite-differences estimation are so unstable. To illustrate how the images are driven out of frame, we can see the following figure from the 75 epochs of pretraining, *clipped* and *not clipped*.

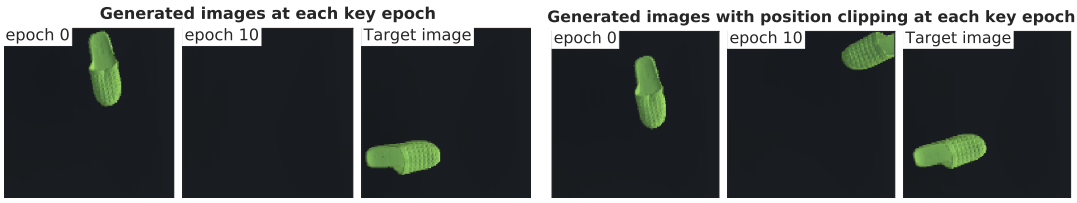

Figure 4.15: **(left)** Randomly sampled images from generated datasets every 10 epochs for unclipped model. **(right)** Randomly sampled images for generated datasets every 10 epochs. Clipping prevents the object from moving out of frame after 10 epochs.

Finally, clipping the locations of the objects lowers the general MMD loss, since the green spot is still in frame. However, this is still harmful to the ability of the network to learn a distribution, since we are mapping all distributions to a single Dirac impulse. From the initial experiments, we have proven that this is not desirable. Therefore, for the rest of the experiments we clip each sample that is driven out of frame by sampling from a Gaussian centered around just below the limit of the frame.

4.1.3 5 mutable attributes: learning to change an object's pose and location

We continue to increase the complexity of the scene by adding the other rotation components of the object, namely the *yaw*, *pitch* and *roll*, plus both location attributes. The *pitch* and *roll* are initialized with $\mu_{pitch} = 360, \mu_{roll} = 360$ and $\sigma_{pitch} = 10, \sigma_{roll} = 10$, and the target distribution with $\mu_{pitch} = 360, \mu_{roll} = 420$ and $\sigma_{pitch} = 5, \sigma_{roll} = 20$. We decide to run 3 experiments where we use 50, 75, and 100 pretraining epochs. We start by looking at the generated *yaw*, *pitch* and *roll* distributions as a 3D plot shown in Figure 4.16. From this plot we can see that there is no distribution matching happening, and we notice a similar pattern when there is no learning, which is the convergence of samples towards 0, in one or two of the angles. This is a sign that the pose of the object

is not well represented in the feature representation of the images, and therefore is not really being learned.

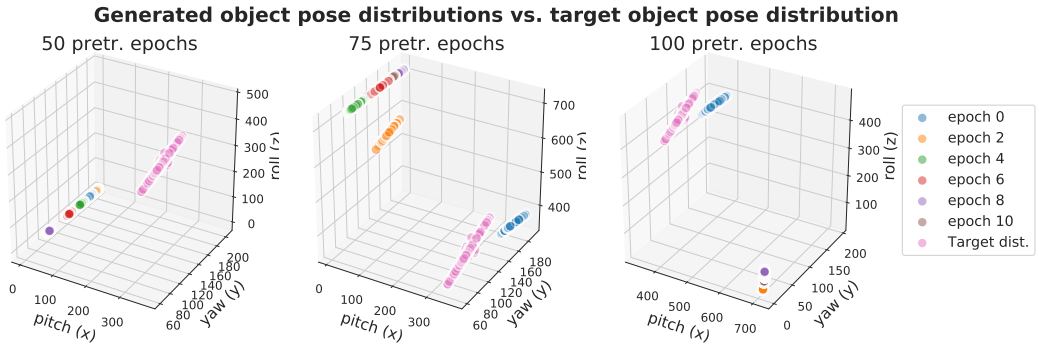

Figure 4.16: 3D scatter plot depicting the yaw, pitch, and roll values of the generated distributions every 2 epochs, compared to the values from the target distribution.

We continue our analysis by looking at what the location distributions of the three experiments in Figure 4.17. From these plots we can notice that the new Gaussian-clipping is working, and now the samples that are out of frame are brought back by sampling from a Gaussian distribution. This however doesn't seem to be enough for the model to learn to fit the target location distributions. We can actually observe that moving from 50 to 75 pretraining epochs shifts the location along z of the generated distributions towards the target distribution, but moves further away the location along x of the generated distribution.

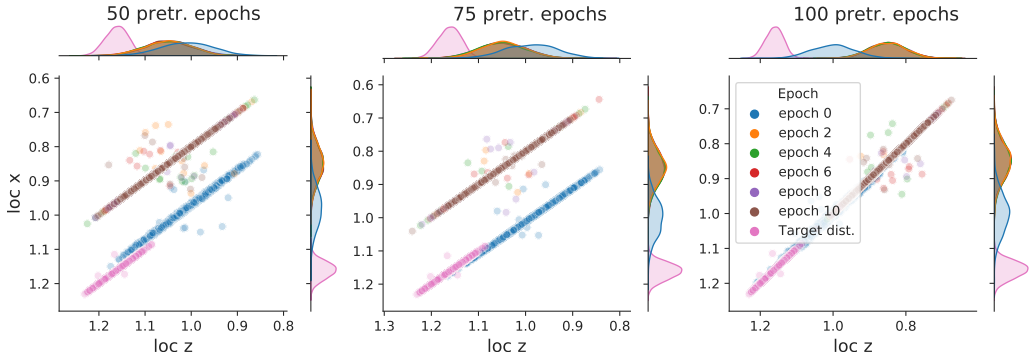

Figure 4.17: Scatter plot showing the distribution of *loc z* and *loc x* values for the generated samples every 2 eppochs, compared to the target distribution's values, during the 5 attribute, single-class setting experiments.

The architecture doesn't seem to be learning at all. This can be confirmed by looking at the MMD loss in Figure 4.19 from the 3 experiments where the experiment with 50 pretraining epochs seems to have the lowest loss, where there is apparent local convergence close to a loss of 30.

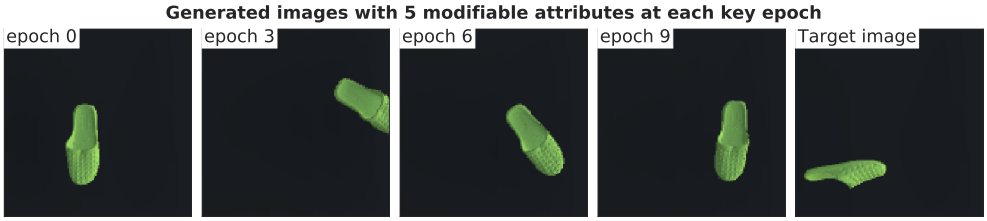

Figure 4.18: Randomly sampled images from generated datasets every 3 epochs of MMD training, compared to the a randomly sampled image from the target dataset.

We can look at the generated images from the experiment with 50 pretraining epochs in Figure 4.18. We can see that from epoch 0 to epoch 9 there seems to be local convergence towards the left bottom corner of the image, but no rotation along the other axis is being learned, rather than some variation along *yaw*. This is in accordance to the distribution shown in Figure 4.16, where the other two components (*pitch* and *roll*) collapse to 0.

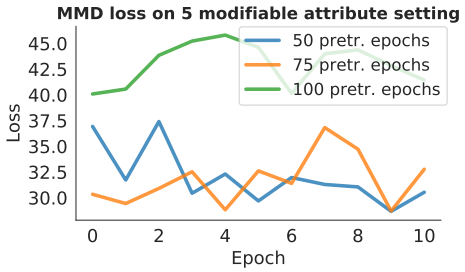

Figure 4.19: MMD loss during the 5 attribute, single-class setting experiments after 10 epochs of MMD training.

4.1.4 8 mutable attributes: learning to rotate the camera and change lighting intensity

Let's increase the number of modifiable attributes to 8 by adding two attributes that modify the rotation of the camera, *camera_roll* and *camera_pitch* (rotation of the camera along z and x axis, respectively), and the *intensity* of the light, which are all initialized with $\mu_{croll} = 360, \mu_{cpitch} = 65, \mu_{int} = 0.5$ and $\sigma_{croll} = 10, \sigma_{croll} = 10, \sigma_{int} = 0.01$, and the target distribution with $\mu_{croll} = 360, \mu_{cpitch} = 60, \mu_{int} = 0.8$ and $\sigma_{croll} = 5, \sigma_{croll} = 5, \sigma_{int} = 0.5$.

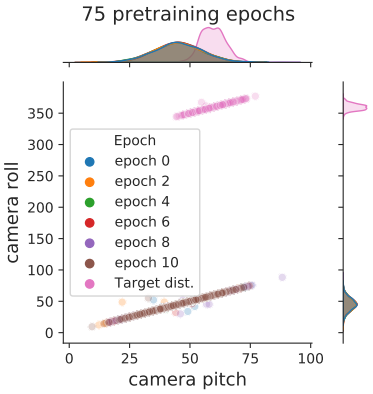

Figure 4.20: Scatter plot showing the *camera pitch*, and *camera roll* values of the generated samples, on the 8 attribute setting.

We can look at the camera attributes to verify whether the architecture is learning to shift the distribution towards the target distribution. Let's look at the camera pitch and roll distributions in Figure 4.20 where we observe that the architecture is stuck at $\approx 50°$ for both attributes. This is a

clear sign that the architecture is simply not learning.

This can be confirmed by plotting the MMD loss in Figure 4.21. From this plot we can clearly see no apparent sign of learning, or even convergence in this amount of epochs. Since the architecture shows random performance, we conclude that the architecture does not scale well to such a complex case in the single-class setting and move forward to experimenting on a multi-class setting.

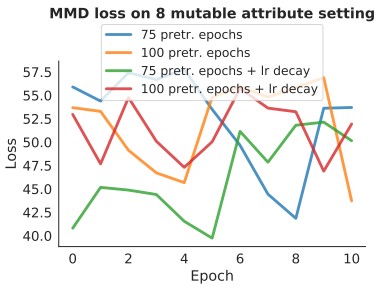

Figure 4.21: MMD loss after 10 epochs of MMD training for the 4 experiments performend on the 8 attribute, single-class setting.

By completing this part of the study, we seek to understand whether using a multi-class setting alleviates the problems encountered in the single-class setting, due to its simplicity and therefore tendency to have vanishing gradients. We will conduct this study as the previous one, but simplifying the hyperparameter search, since the previous case proved that learning rate decay and the number of pretraining epochs are the two hyperparameter settings that most affect the performance of the network.

4.2 Experiments on a 5-class setting

4.2.1 1 mutable attribute: learning to rotate 5 objects 90°

We will initialize all generated distributions and define all target distributions as per the previous experiments. Also, as per the previous experiments, we wish our model to converge towards the target mean while keeping the standard deviation $\sigma = 10^\circ$. This was not achievable under the explored settings in the previous sections.

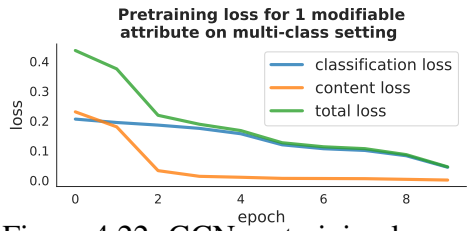

Figure 4.22: GCN pretraining loss after 10 pretraining epochs on the multi-class setting.

Let us start by analyzing the pretraining performance of the GCN autoencoder in Figure 4.22. We can clearly see that, in contrast to the single class setting, in this case the classification loss is significantly higher than the content loss. This is because in this case we have a random sampler that assigns different random classes to the object node in the graph. The GCN here needs to learn how to sample according to the randomness initiated by the probablistic grammar sampling.

According to this, we decide to train our model for 10 MMD epochs after 1, 5, and 10 pretraining epochs, and similarly with learning rate decay every 2 epochs. In

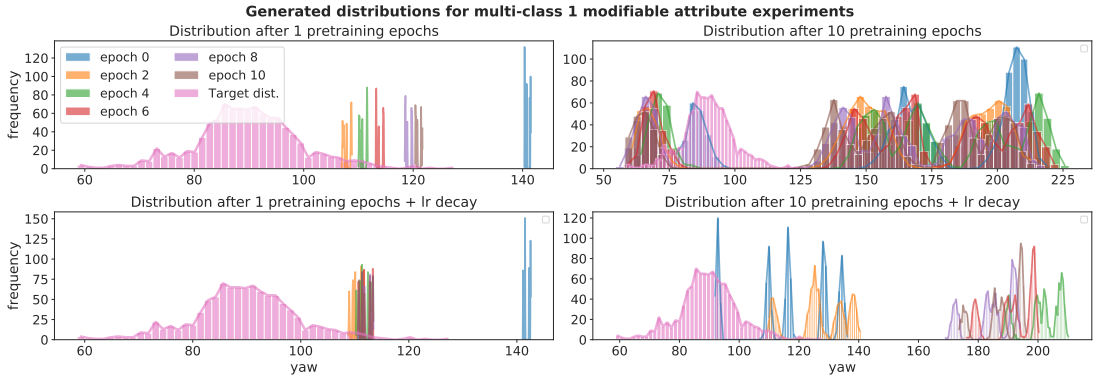

Figure 4.23: Distribution of *yaw* values from generated datasets every 2 epochs, compared to the target attribute value distribution, on multi-class 1 attribute experiments.

the distributions plotted in Figure 4.23 we only plot results for 1 and 10 pretraining epochs, since these expressed more information about the impact of pretraining and learning rate decay in a Meta-Sim multi-class setting. From these distribution plots there is interesting phenomena going on. Firstly, let's look at the experiments with 1 pretraining epoch. As in the homologous single-class setting, the mean seems to be converging towards the target distribution's mean, but there is no standard deviation due to the limited pretraining.

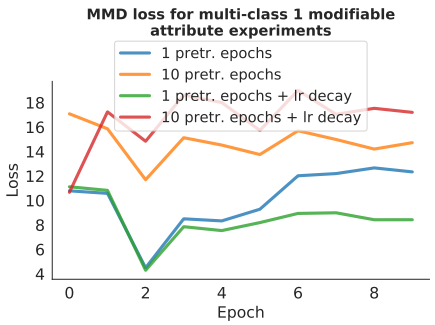

Figure 4.24: MMD loss after 10 MMD training epochs for the multi-class 1 attribute experiments.

We can also see that, unlike the single-class setting, we have multiple peaks per epoch. This seems to be another of the many nuances that the Meta-Sim model has, this time in the distribution initialization. For the target distribution, we define one Gaussian distribution per attribute, and then assign a sampled value from this Gaussian to a randomly chosen class, obtaining a single distribution for all classes. On the contrary, in the generated distribution case, the PSG initial sampling smaples a random class first, and then assigns a value to each attribute, sampled from a Gaussian distribution. This is learned by the GCN during pretraining, and then passed on during MMD training[4]. Moreover, the learning rate decay keeps the distributions from moving further away from the target mean. This is because the updates become smaller, the model has less chances to jump out of the local optima at $\approx 106°$.

[4]Further investigating the impact of this initialization remains out of the scope of this study, and we don't rule out that this might slightly improve performance in the multi-class setting, since we would like to compare distributions of the same nature for a more compatible MMD calculation [36].

Secondly, if we now shift our attention to the right column of Figure 4.23, we can see that actually pretraining the GCN for longer will generate more robust Gaussians around different means. This actually hints to a problem of using multi-modal Gaussians to initialize the GCN. If we observe the distributions generated with 10 pretraining epochs, we can see that one of the Gaussians corresponding to one of the classes is breaking away and almost matching the target distribution. This is harmful because we want all distributions to shift equally towards the target distribution. Additionally, and as in the 1 pretraining epoch case, the learning rate decay seems to narrow down the distributions and get them closer together, but the target mean is not learned at all. The effect of this phenomena impacts the MMD loss plotted in Figure 4.24. From this plot we observe that using 10 pretraining epochs is harmful to the convergence of the network. On the contrary, we find that there is convergence with 1 pretraining epoch, and the learning rate decay helps the network become stable around local optima. Finally, let's look at the generated images in Figure 4.29.

Figure 4.25: Randomly sampled images from generated datasets every 3 epochs, compared to a randomly sampled images from the target dataset, after 1 pretraining epoch and using learning rate decay

4.2.2 3 mutable attributes: learning to rotate 5 objects $90°$ and change their location

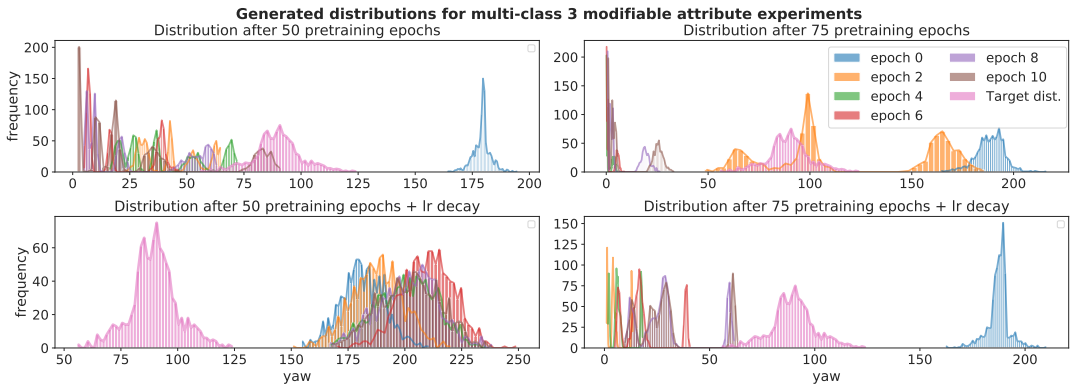

Figure 4.26: Distributions of the *yaw* value from generated datasets every 2 epochs, compared to the target distribution for the 3 attribute setting.

We continue to increase the complexity of the scene by adding the location along x and along z in the 3D scene as mutable attributes. We choose to significantly increase the number of pretraining epochs to 50 and 75, since the number of mutable attributes and classes has increased. We therefore perform 4 experiments with 50 and 75 pretraining epochs, and then add learning rate decay to each of these cases. We plot the generated *yaw* distributions in Figure 4.26.

From these distributions we can see that a similar phenomenon to the previous experiment occurs. In this case though the epoch 0 distribution remains as one single distribution and not multiple distributions. It is when the MMD loss starts updating the network (after the 1st epoch), that we observe multiple distributions being formed at each epoch. Similar to the previous set of experiments, it is when we add learning rate that the distributions for each epoch get closer together, as it is the case for the lower row of the plot.

Moreover, we plot the distribution of generated locations in Figure 4.27, where we can see that there is no clear learning for the experiments without learning rate decay. Indeed, the distributions shift in the opposite direction to the target distribution, and stagnate at $loc\ z \approx 0.85$, and at $loc\ x \approx 1.05$. This is because the gradients of the network are stagnating at a local optima that puts the object out of frame. Thus, our model is forced to bring it into frame by sampling from a Gaussian distribution that is centered around $loc\ z \approx 0.85$, and at $loc\ x \approx 1.05$.

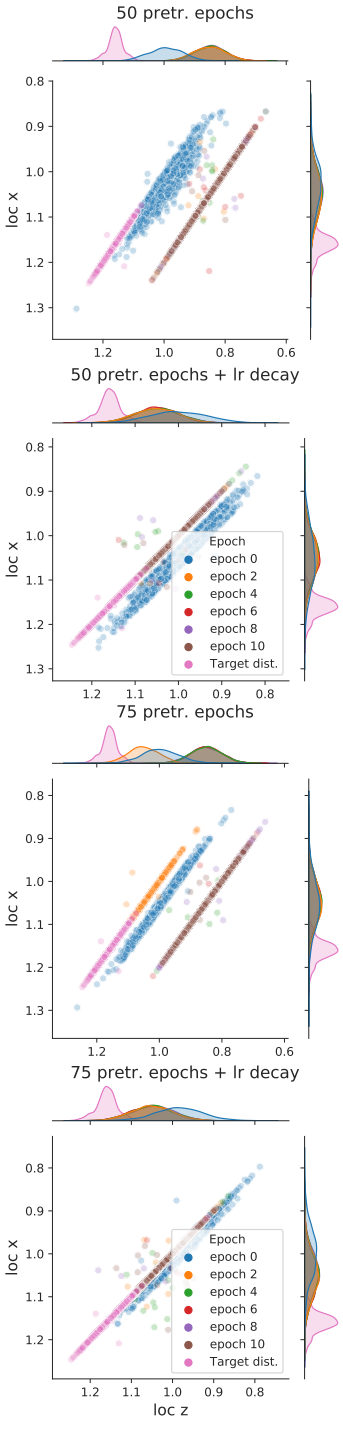

Figure 4.27: Distribution of the $loc\ z$ and $loc\ x$ values for the generated datasets every 2 epochs of MMD training.

On the contrary, when we add learning rate decay, the distributions seem to shift in the right direction, and more as the number of pretraining epochs increases. Indeed, the experiment with 75 pretraining epochs and learning rate decay seems to be the one that performs the best. It doesn't fully converge towards the target distributions, but it

at least shifts in its direction.

This can be confirmed by looking at Figure 4.28, where it is evident that the learning rate decay is keeping the loss at a much lower level than the experiments without learning rate decay. There is still no definitive convergence but the loss stabilizes at a fairly low level for the cases with learning rate decay. direction.

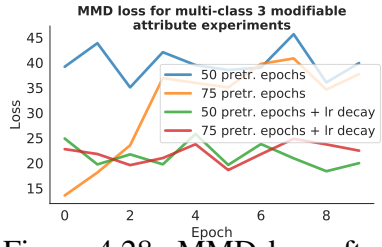

Figure 4.28: MMD loss after 10 MMD training epochs, for the 4 experiments on the 3 attribute setting.

Finally, we can visualize below some of the generated images every 3 epochs of MMD training. From this we can see that by epoch 9 the network learns to generate samples that are located around the bottom left corner. The network however doesn't learn to rotate correctly the green shoe. Juding from the image at epoch 9 the shoe should be horizontal, and in this case it is oriented vertically.

Figure 4.29: Randomly sampled images from generated images every 3 epochs, compared to a randomly sampled image from the target dataset.

4.2.3 5 mutable attributes: learning to change 5 object's pose and location

Let's increase the complexity of the scene by adding the other two components for the rotation of the object, *pitch* and *roll*, on the multi-class setting. We continue to analyze the behavior of the network and its components when more attributes are added. We specifically want to observe how the full

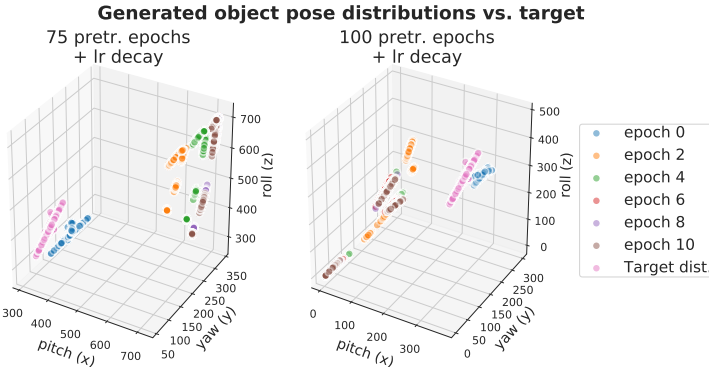

Figure 4.30: 3D scatter plot of *pitch*, *yaw*, and *roll* values for generated dataset every 2 epohs of MMD training, compared to the target distribution attribute values.

distributions are generated for each attribute, and how the fact that more attributes result in more model flexibility.

Indeed, and we will observe this clearly in the next section, the fact that more attributes are added gives the model more chances to fail and get stuck in local optima, because more attributes need to be tuned to a specific distributions to generate images that allow correct convergence of the model.

We train 4 different models, pretrained during 75 and 100 epochs, and we add learning rate decay in the later two experiments for which we visualize all *pitch*, *yaw*, and *roll* sample values in Figure 4.30. In this case, the same phenomenon we saw of multi-modal distribution splitting is happening with the rotations. We can see that only at epoch 0 the distributions seem to be grouped together in a similar shape as the target distribution. It is just after that epoch, at epoch 2, that we can notice 5 separate blobs forming, we assume this corresponds to the distributions of the 5 classes.

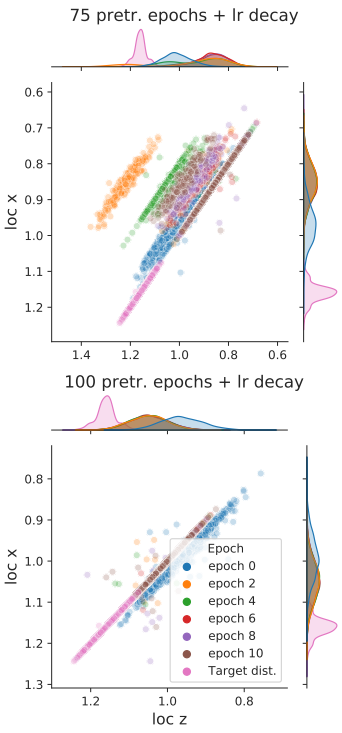

Figure 4.31: 2D distribution of *loc z* and *loc x* values from generated dataset every 2 epochs of MMD training, compared to the target dataset attribute values.

However, even though these two experiments can still reproduce distributions after 4 epochs of training, they still don't learn the target distribution for the full rotation attributes. If we focus on the 100 pretraining epochs + learning rate decay experiment, we can see that the generated samples are close to the target distribution in the *yaw* (y) and *roll* (z) dimensions, but not on the *pitch* (x) dimensions. If we now focus our attention to the generated location distributions plotted in Figure 4.31, we can see that it is only using 100 pretraining epoch and learning rate decay that allows the network to generate distributions of locations that are closer to the target distributions. From Figure 4.32, the epoch 10 image contains an image of a Tupperware. We can see that the location is closer to the bottom left corner, and pose of the object is similar to the target Tupperware, except for the yaw, where the generated is oriented vertically ($\approx 180°$), and the target is oriented almost horizontally ($\approx 90°$).

Finally, this set of experiments sheds more light into the difficulty of adding more variation in the scene for the Meta-Sim model to manipulate. Indeed, the more vari-

Figure 4.32: Randomly sampled images from the generated datasets every 2 epochs of MMD training, compared to a randomly sampled image from the target dataset.

ation we add, the more unstable the learning becomes. The generated samples have too many degrees of freedom and they can easily become too different to the target samples and let the learning stagnate at some local optima. This is exacerbated in the following (and final) set of experiments, where we add the camera rotation and light intensity attributes.

4.2.4 8 mutable attributes: learning to rotate the camera and change light intensity

This is the final test that the Meta-Sim architecture will undergo, and we wish to understand how giving the model freedom over the rotation of the camera along two axis (*pitch* and *roll*), and the *intensity* of the light, will affect both the learning and the generated samples. To test this, we use the same learning parameters as the previous experiment and we investigate the generated distributions. In this case, we will limit ourselves at looking at the generated distributions for the 3 new modifiable attributes, in order to understand how the architecture behaves when allowing it to control two more nodes in the 3D scene.

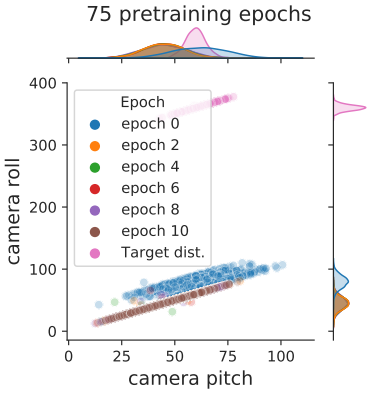

Figure 4.33: *camera pitch* and *camera roll* attribute values from generated dataset every 2 epochs of MMD training, compared to the target dataset attribute values.

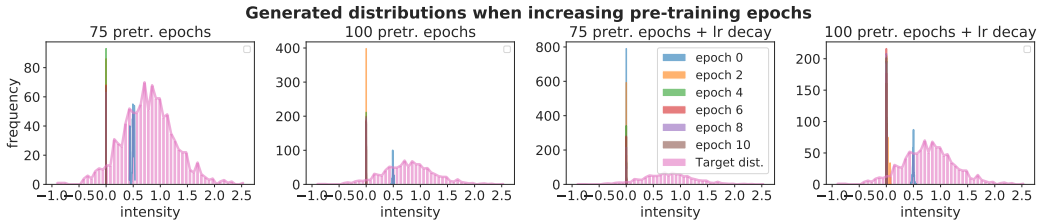

Figure 4.34: Distribution of light *intensity* values from the generated dataset every 2 epochs of MMD training, compared to the target dataset attribute values.

Let us visualize the generated distributions for the light *intensity* parameter in Figure 4.34. From the figure we can quickly conclude that there is mode collapse happen-

ing for this attribute. Indeed, the architecture is not even learning the initial distribution of the sampled attributes, which results in all values eventually collapsing to 0.

This is a sign that the architecture was not pretrained for long enough, and further exploration into this issue is left as future work, since there seems to be a combination of issues that cause mode collapse, and this is present from the simplest case, up until the most complicated case. The same phenomenon happens with the camera attributes in Figure 4.33, which happens with the camera attributes for the other 3 experiments.

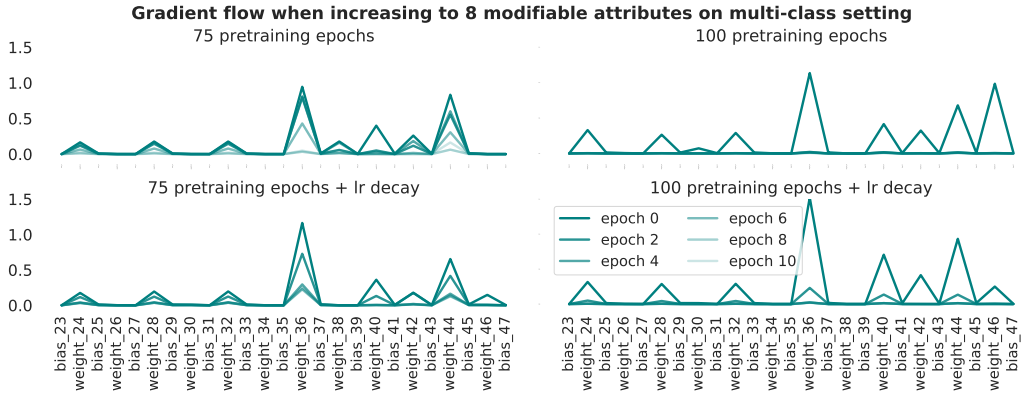

Figure 4.35: Gradient flow through the GCN decoder layers for the experiments performed on the 8 attribute multi-class setting.

We can prove mode collapse by looking at the gradients for each experiment passing through the GCN decoder in Figure 4.35. All gradients except for the 75 pretraining epochs with learning rate decay experiment are eventually collapsing to 0. This is reflected in the generated images shown in Figure 4.36, where the generated images stagnate with no illumination.

Figure 4.36: Randomly sampled images from the generated dataset every 2 epochs of training, compared to a randomly sampled image from the target dataset.

Chapter 5

Discussion and Conclusions

This study focused on testing the feasibility of Unsupervised Synthetic Data Generation pipelines by thorougly and extensively experimenting with the Meta-Sim architecture while connected to a different 3D scene than the one originally used. This was motivated by the fact that the pipeline is only briefly studied on a single 3D scene, with no implementation available to the public, and therefore many open questions with respect to its scalability to different rendering engines, different scenes, and different quantities and types of modifiable attributes. Consequently, our study focused on first developing a way to attach 3D renderers to the architecture, or any architecture that needs to generate data as it trains, second on defining a new 3D scene, and third on systematically testing whether the architecture learns to generate data that resembles a synthetically generated dataset from this 3D scene, under increasingly complex settings. Let us discuss the results obtained from developing these 3 aspects of our study.

The attachment of the 3D rendering engine was the main engineering hurdle that needed to be achieved to carry on with the study. This implementation had strict efficiency requirements to allow us to experiment extensively on the Meta-Sim architecture, generating thousands of images in seconds, so the architecture could be trained for 100 epochs in 10 hours on a Tesla K80 GPU. Unsuccessfully, we attempted this with the Blender rendering engine proving to be too slow for the fast experimenting framework we needed. This engine generates 100 128x128 RGB images in $\approx$1.5s in the GPU. To successfully generate a synthetic dataset of 1,000 images, the Meta-Sim architecture needs to render $\approx$3,000 images at every epoch. For a 100 epoch experiment using Blender, this would amount to $\approx$5.2 days of training which is simply too time-expensive. We therefore moved to the Unity rendering engine [6], which due to

its game-development optimized nature takes ≈ 0.12s to generate 100 images of the same size which theoretically reduces training of the same experiment to 10 hours. It is important to mention however that fast rendering time is not everything in determining the efficiency of the Unsupervised Data Generation architecture. The attachment also needs to be efficient. For instance, our first implementation of Meta-Sim with Unity needed to build the Unity 3D scene every time a batch of graphs needed to be rendered. Due to CPU limitations, a Unity build can take up to 6s, even if, after the build, the renderer takes only 0.12s to generate a batch of images. This meant a complete architectural change that is formalized in Section 3.4.1, which builds the Unity 3D scene once and then runs in parallel to Meta-Sim. All of this to say that this architectural design is one of the main contributions of the study, since the Meta-Sim authors provided only a renderer limited to digits and provided little hints into how to attach a 3D renderer, which is evidently not a trivial task in the way to using Meta-Sim as an Unsupervised Synthetic Data Generation architecture. On the contrary, it is the most difficult implementation task of the architecture.

Moving away from the implementation and focusing on the experiments, we found the Meta-Sim architecture to have particularly fragile gradients, that made the learning volatile, and convergence of the network very difficult. This was experienced both in the *single-class setting* as in the *multi-class setting*. The *single-class setting* proved to be a very difficult configuration for Meta-Sim to learn to generate samples that had a wide standard deviation and converged towards the mean of the target distribution. Indeed, only the simplest 1 attribute case converged towards the target mean, with further unsuccessful attempts to learn the correct target mean with a wide enough distribution. Only on this setting, we learned that within the Meta-Sim pipeline, more pretraining epochs mean a wider generated distribution, but higher gradients, which mean more unstable learning, and therefore no convergence towards the mean. On the contrary, less pretraining gradients meant smaller gradients, and therefore convergence towards the target mean. Two configurations that contradict each other in network performance. Contradictions that were only exacerbated by adding more attributes on the *single-class setting*. Since more attributes mean more distributions to generate correctly during pretraining, that made it essential to use more pretraining epochs, which further destabilized convergence towards a target mean.

Furthermore, the *multi-class setting* exhibited better performance, and the fact that multiple distributions were being generated, each corresponding to each class, served as a regularizing effect to the high gradient problem. Indeed, the 1 attribute experiment

converged towards a $\approx 106^{\circ}$ mean *yaw* angle, with an MMD loss oscillating around ≈ 7.6. These results suggest correct convergence, but the generated distribution is still not reproducing the specified standard deviation, and the network still only converges when pretrained during 1 epoch. The following 3 attribute and 5 attribute experiments exhibited no real convergence in the MMD loss, but their generated samples did exhibit correct behavior, generating images that were generally similar to the target images. This hints that a multi-class setting on a multi-attribute setting regularizes the overall gradients of the network and generates samples that are close to the target samples.

Nevertheless, when increasing scene complexity to 8 attributes, the network was not able to generate samples with the correct attribute values and exhibited mode collapse and vanishing gradients. Indeed, by increasing the amount of mutable attributes, the network's freedom to manipulate the 3D scene also increases, this is counterproductive since the network can simply move the camera or the objects of the scene towards a point that puts the objects of interest out of frame in the generated image. This point can mean a local optima in optimization, since the generated samples might be the same with no content, meaning that all samples would be concentrated around one point in feature space, converting the distribution into a Dirac distribution and stagnating all learning. This is something initially addressed in Section 4.1.2 by clipping all samples that get out of frame with a Gaussian distribution. This however can also allow the network to stagnate at this Gaussian distribution, something observed in the experiments on 8 mutable attributes in Sections 4.1.4 and 4.2.4.

Interestingly, these results both support and refute the original hypothesis of more complexity hurting the network's performance, and we must redefine what complexity means in this case. Indeed, our results suggest that adding more mutable attributes to a given 3D scene demand more pretraining epochs and therefore more unstable gradients, and therefore no convergence at all. However, our results also suggest that adding more classes to the scene act as a regularizer to these gradients and therefore improve performance. We therefore advice adding more classes as more attributes are added, in order to improve network performance. This phenomena of learning instability is most likely sourced in the finite-differences step used to approximate backpropagation through the renderer, which can be seen as a non-differentiable black box function. The approximation must therefore be providing unstable gradients that need to be more robustly controlled. Indeed, this seems to be the main issue [43], where our study reveals the importance of stabilizing learning for correct convergence of functions that use approximate methods. This is specially important when using

non-differentiable rendering engines.

5.1 Open questions

These results give rise to multiple questions that should be answered in future work, in order to improve Meta-Sim's ability to converge under new settings, given that it doesn't robustly work under different configurations than the originally used by [29]. The questions are listed in order of relevance at solving the architecture's current problems.

How does the finite-differences approximation affect gradient stability and how can it be directly regularized? As mentioned before, the fact that non-differentiable renderers are used in Unsupervised SDG architectures raises the need of using approximative methods to estimate the gradient. According to [43], tuning the δ value during the approximation is crucial for the estimation to be correct. Indeed, we suspect that the δ value of 0.03 used during this study is perhaps too high, probably explaining the high gradients. Further work investigating the impact of different δ magnitudes is crucial to understand whether the learning can be stabilised and MMD convergence can be reached more generally.

What is the impact of using different feature representations on the network's ability to calculate the distribution loss? Even though we used multiple types of generated and target distributions to be matched, the same study needs to be performed for the multi-class setting, where the same types of multi-modla, or uni-modal distributions are compared. Not only that, but further exploration can be done in the way the images are represented. Indeed, in this case we were only using the final pooling layer of the InceptionV3 network, but it is also advised using the last fully connected layer from InceptionV3's the auxiliary classifier [61] as a feature representation. Additionally, fine-tuning these models during MMD training can also be an alternative to obtain better feature representations and better distribution matching performance.

Are architectures that approximate gradients to backpropagate through non-differentiable renderers even necessary given the recent proposal of differentiable renderers? We must however act the question if further pursuing using non-differentiable rendering is worth the effort. On one side, given that most renderers in the market are non-differentiable, being able to succesfully incorporate them into an Unsupervised SDG architecture is very tempting, since these are currently the most developed renderers in the market, in terms of photorealism and speed. On the other side, differentiable

renderers are on the technological horizon [30], and shifting towards developping Unsupervised SDG architectures that use these technologies may mean exploiting and realising their potential from an early technological stage. Both approaches are worth pursuing, but this needs to be done strategically. The full potential of Unsupervised SDG frameworks must be realised one way or another. These frameworks promise unlocking the data collection problem for any CV application.

5.2 Conclusions

Overall, we developed an architecture to use 3D rendering engines as data generators in an Unsupervised SDG pipeline, and used it to systematically study the Meta-Sim architecture under multiple, increasingly complex settings. Initially, we show that attaching a rendering engine to a Machine Learning pipeline is not an evident task, and that rendering speed and parallel processing are both necessary to train architectures that generate thousands of images in one single training epoch, in a reasonable amount of time, with a reasonable amount of resources.

Moreover, we define a new 3D scene and use this implementation to define an experimentation framework to test Meta-Sim's performance under increasingly complex settings that vary in number of classes in the scene, and mutable attributes, modifiable parameters that the architecture can use to change the conditions of the classes in the scene. Through this experimentation, we show that the architecture has an important amount of nuances that make the learning highly fragile and difficult with a new 3D scene, different to the ones used in the original study. This is supported by an extensive set of experiments and results that demonstrate the architecture's inability to robustly generate a target dataset whose attributes resemble the ones from a target dataset.

Therefore, this study leads us to conclude that the Meta-Sim architecture, as it is, is not scalable to different 3D scenes, with different 3D rendering engines, and that the learning hyperparameters necessary to perform well in a new setting are too specific, and difficult to tune for it to be deemed reproducible to different settings. Furthermore, we don't rule out necessary modifications performed in future work that may address the learning stability problem under complex settings, which may lead to increased robustness and therefore performance.

Bibliography

[1] How to test gradient implementations — Graduate Descent.

[2] Learning To Simulate — Papers With Code.

[3] Meta-Sim: Learning to Generate Synthetic Datasets — Papers With Code.

[4] Meta-Sim2: Unsupervised Learning of Scene Structure for Synthetic Data Generation — Papers With Code.

[5] nv-tlabs/meta-sim: Meta-Sim: Learning to Generate Synthetic Datasets (ICCV 2019).

[6] Unity Real-Time Development Platform — 3D, 2D VR & AR Engine.

[7] Bishwo Adhikari and Heikki Huttunen. Iterative bounding box annotation for object detection. *Proceedings - International Conference on Pattern Recognition*, pages 4040–4046, 2020.

[8] Harkirat Singh Behl, Atilim Güneş Baydin, Ran Gal, Philip H.S. Torr, and Vibhav Vineet. AutoSimulate: (Quickly) Learning Synthetic Data Generation. Technical report, 2020.

[9] Amanda Berg, Joakim Johnander, Flavie Durand de Gevigney, Jorgen Ahlberg, and Michael Felsberg. Semi-Automatic Annotation of Objects in Visual-Thermal Video, 2019.

[10] Simone Bianco, Gianluigi Ciocca, Paolo Napoletano, and Raimondo Schettini. An interactive tool for manual, semi-automatic and automatic video annotation. *Computer Vision and Image Understanding*, 131:88–99, 2 2015.

[11] Mikołaj Binkowski, Dougal J. Sutherland, Michael Arbel, and Arthur Gretton. Demystifying MMD gans. Technical report, 2018.

[12] Konstantinos Bousmalis, George Trigeorgis, Nathan Silberman, Dilip Krishnan, and Dumitru Erhan. Domain separation networks. *Advances in Neural Information Processing Systems*, pages 343–351, 8 2016.

[13] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. 6 2016.

[14] Daniel J. Butler, Jonas Wulff, Garrett B. Stanley, and Michael J. Black. A naturalistic open source movie for optical flow evaluation. Technical Report PART 6, 2012.

[15] Fabio De Sousa Ribeiro, Francesco Caliva, Mark Swainson, Kjartan Gudmundsson, Georgios Leontidis, and Stefanos Kollias. An adaptable deep learning system for optical character verification in retail food packaging. *2018 IEEE International Conference on Evolving and Adaptive Intelligent Systems, EAIS 2018*, pages 1–8, 6 2018.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. pages 248–255, 3 2010.

[17] Jeevan Devaranjan, Amlan Kar, and Sanja Fidler. Meta-Sim2: Unsupervised Learning of Scene Structure for Synthetic Data Generation. Technical report, 2020.

[18] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio López, and Vladlen Koltun. CARLA: An open urban driving simulator. *arXiv*, 11 2017.

[19] Annalisa Franco, Davide Maltoni, and Serena Papi. Grocery product detection and recognition. *Expert Systems with Applications*, 81:163–176, 9 2017.

[20] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. VirtualWorlds as Proxy for Multi-object Tracking Analysis. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-Decem:4340–4349, 5 2016.

[21] Yaroslav Ganin, Tejas Kulkarni, Igor Babuschkin, S. M.Ali Eslami, and Oriol Vinyals. Synthesizing Programs for Images using Reinforced Adversarial Learning. Technical report, 2018.

[22] Rong Ge, Sham M. Kakade, Rahul Kidambi, and Praneeth Netrapalli. The Step Decay Schedule: A Near Optimal, Geometrically Decaying Learning Rate Procedure For Least Squares. *Advances in Neural Information Processing Systems*, 32, 4 2019.

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 10 2020.

[24] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Alexander Smola, Bernhard Schölkopf, and Alexander Smola GRETTON. A Kernel Two-Sample Test Bernhard Schölkopf. Technical report, 2012.

[25] Tomas Hodan, Vibhav Vineet, Ran Gal, Emanuel Shalev, Jon Hanzelka, Treb Connell, Pedro Urbina, Sudipta N. Sinha, and Brian Guenter. Photorealistic Image Synthesis for Object Instance Detection. *Proceedings - International Conference on Image Processing, ICIP*, 2019-Septe:66–70, 2 2019.

[26] Tomáš Hodaň, Frank Michel, Eric Brachmann, Wadim Kehl, Anders Glent Buch, Dirk Kraft, Bertram Drost, Joel Vidal, Stephan Ihrke, Xenophon Zabulis, Caner Sahin, Fabian Manhardt, Federico Tombari, Tae Kyun Kim, Jiří Matas, and Carsten Rother. BOP: Benchmark for 6D object pose estimation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11214 LNCS:19–35, 8 2018.

[27] Judy Hoffman, Eric Tzeng, Taesung Park, Jun Yan Zhu, Phillip Isola, Kate Saenko, Alexei A. Efros, and Trevor Darrell. CyCADA: Cycle-Consistent Adversarial Domain adaptation. Technical report, 2018.

[28] Xun Huang, Ming Yu Liu, Serge Belongie, and Jan Kautz. Multimodal Unsupervised Image-to-Image Translation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11207 LNCS:179–196, 4 2018.

[29] Amlan Kar, Aayush Prakash, Ming Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate synthetic datasets. *Proceedings of the IEEE International Conference on Computer Vision*, 2019-Octob:4550–4559, 4 2019.

[30] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable Rendering: A Survey.

[31] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 9 2016.

[32] Dimitrios Kollias, Athanasios Tagaris, Andreas Stafylopatis, Stefanos Kollias, and Georgios Tagaris. Deep neural architectures for prediction in healthcare. *Complex & Intelligent Systems*, 4(2):119–131, 6 2018.

[33] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Daniel Gordon, Yuke Zhu, Abhinav Gupta, and Ali Farhadi. AI2-THOR: An interactive 3D environment for visual AI. *arXiv*, 12 2017.

[34] Adriana Kovashka, Olga Russakovsky, Li Fei-Fei, and Kristen Grauman. Crowdsourcing in computer vision. *Foundations and Trends in Computer Graphics and Vision*, 10(3):177–243, 2016.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[36] Atsutoshi Kumagai and Tomoharu Iwata. Unsupervised Domain Adaptation by Matching Distributions Based on the Maximum Mean Discrepancy via Unilateral Transformations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):4106–4113, 7 2019.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998.

[38] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1681:319–345, 1999.

[39] Peilun Li, Xiaodan Liang, Daoyuan Jia, and Eric P. Xing. Semantic-aware grad-GAN for virtual-to-real urban scene adaption. *arXiv*, 1 2018.

[40] Yujia Li, Kevin Swersky, and Richard Zemel. Generative Moment Matching Networks. Technical report.

[41] Ming Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 701–709. Neural information processing systems foundation, 3 2017.

[42] Tianqiang Liu, Siddhartha Chaudhuri, Vladimir G. Kim, Qixing Huang, Niloy J. Mitra, and Thomas Funkhouser. Creating consistent scene graphs using a probabilistic grammar. *ACM Transactions on Graphics*, 33(6), 11 2014.

[43] Matthew M Loper and Michael J Black. OpenDR: An Approximate Differentiable Renderer.

[44] Gilles Louppe, Joeri Hermans, and Kyle Cranmer. Adversarial variational optimization of non-differentiable simulators. Technical report, 2017.

[45] Paweł Michalski, Bogdan Ruszczak, and Michał Tomaszewski. Convolutional Neural Networks Implementations for Computer Vision. *Advances in Intelligent Systems and Computing*, 720:98–110, 2018.

[46] Yair Movshovitz-Attias, Takeo Kanade, and Yaser Sheikh. How useful is photo-realistic rendering for visual learning?

[47] Marina Paolanti, Luca Romeo, Massimo Martini, Adriano Mancini, Emanuele Frontoni, and Primo Zingaretti. Robotic retail surveying by deep learning visual and textual data. *Robotics and Autonomous Systems*, 118:179–188, 8 2019.

[48] R. Pierdicca, E. S. Malinverni, F. Piccinini, M. Paolanti, A. Felicetti, and P. Zingaretti. Deep convolutional neural network for automatic detection of damaged photovoltaic cells. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives*, 42(2):893–900, 5 2018.

[49] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. VirtualHome: Simulating Household Activities Via Programs. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 8494–8502, 6 2018.

[50] Wei Qu, Tao Qian, and Guan-Tie Deng. Sparse Approximation to the Dirac-{\delta} Distribution. 8 2020.

[51] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M. Lopez. The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. Technical report, 2016.

[52] Nataniel Ruiz, Samuel Schulter, and Manmohan Chandraker. Learning To Simulate. *arXiv*, 10 2018.

[53] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 9 2014.

[54] Berkman Sahiner, Aria Pezeshk, Lubomir M. Hadjiiski, Xiaosong Wang, Karen Drukker, Kenny H. Cha, Ronald M. Summers, and Maryellen L. Giger. Deep learning in medical imaging and radiation therapy. *Medical Physics*, 46(1):e1–e36, 1 2019.

[55] Bikash Santra and Dipti Prasad Mukherjee. A comprehensive survey on computer vision based approaches for automatic identification of products in retail store. *Image and Vision Computing*, 86:45–63, 6 2019.

[56] Alireza Shafaei, James J. Little, and Mark Schmidt. Play and Learn: Using Video Games to Train Computer Vision Models. *British Machine Vision Conference 2016, BMVC 2016*, 2016-September:1–26, 8 2016.

[57] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Aerial Informatics and Robotics Platform - Microsoft Research. Technical report, 2017.

[58] Hao-Jun Michael Shi, Melody Qiming Xuan, Figen Oztoprak, and Jorge Nocedal. On the Numerical Performance of Derivative-Free Optimization Methods Based on Finite-Difference Approximations. 2021.

[59] Hao Su, Jia Deng, and Li Fei-Fei. Crowdsourcing annotations for visual object detection. *AAAI Workshop - Technical Report*, WS-12-08:40–46, 2012.

[60] Kenji Suzuki. Overview of deep learning in medical imaging. *Radiological Physics and Technology 2017 10:3*, 10(3):257–273, 7 2017.

[61] Christian Szegedy, Wei Liu, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions.

[62] Hoang Thanh-Tung and Truyen Tran. Catastrophic forgetting and mode collapse in GANs. *Proceedings of the International Joint Conference on Neural Networks*, 7 2020.

[63] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *IEEE International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[64] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2018-June:1082–1090, 4 2018.

[65] Pete Warden. How many images do you need to train a neural network?, 2017.

[66] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 1992 8:3*, 8(3):229–256, 5 1992.

[67] Yi Wu, Yuxin Wu, Georgia Gkioxari, and Yuandong Tian. Building generalizable agents with a realistic and rich 3D environment. *arXiv*, 1 2018.

[68] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph Neural Networks: A Review of Methods and Applications. *arXiv*, 2018.

[69] Jun Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. *Proceedings of the IEEE International Conference on Computer Vision*, 2017-Octob:2242–2251, 3 2017.

[70] Song Chun Zhu and David Mumford. A stochastic grammar of images. *Foundations and Trends in Computer Graphics and Vision*, 2(4):259–262, 2006.

[71] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object Detection in 20 Years: A Survey. Technical report, 2019.

Appendix A

Meta-Sim hyperparameters

Hyperparameter	**Description**	**Type**	**Values attempted**
GCN pretraining			
train_reconstruction	Whether to pretrain GCN	Boolean	True
freeze_encoder	Whether to update GCN encoder weights during MMD training	Boolean	True, False
reconstruction_epochs	Number of epochs to pretrain GCN	Integer	1,5,**7**,10,15,25,50,75,100
dropout	Number of epochs to pretrain GCN	None or float	None
MMD training			
batch_size	Batch size of generated data	Integer	16,20,**50**,100,200
num_real_images	Bacth size of target data	Integer	50,100,**200**
max_epochs	Number of epochs for MMD training	Integer	10,25,50,100,**500**
epoch_length	Number of generated samples during one epoch of MMD training	Integer	1,000
mmd_dims	Indexes for pooling layers from Inception used as feature maps	List of values in set: {64,192,768,2048}	[2048]
mmd_resize_input	Whether to resize images to $299x299$	Boolean	False
Adam Optimizer			
lr	Learning rate	Integer	**0.001**, 0.0001, 0.0005
lr_decay	Every how many epochs to apply learning rate decay	Integer	1,2,**None**
lr_decay_gamma	Decaying factor	Floating point	0.5
weight_decay	L2 weight penalty	Floating point	**1.0e-3**, 5.0e-4, 1.0e-5
Finite-differences			
delta	Magnitude of the approximated gradient update	Floating point	*Future work,* $deltas < 0.03$ update

Table A.1: Hyperparameters attempted during the Meta-Sim experiments. In **bold**, initial hyperparameters used during experiments from Section 4.1.1.1

.

Appendix B

Experiments on the MNIST dataset

B.1 Learning to rotate digits

We decide to initally run the baseline implementation provided by the author to prove correct convergence of its components under the baseline experiments. We confirm convergence of the GCN pretraining for the 10 digit, 1 mutable attribute case. We decide to perform two experiments, one including the TaskNet loss, *taskloss*, and the other without it. We focus on evaluating the generated distributions, since this will be the same type of experiments performed during the study. We plot the generated distributions in the following plot.

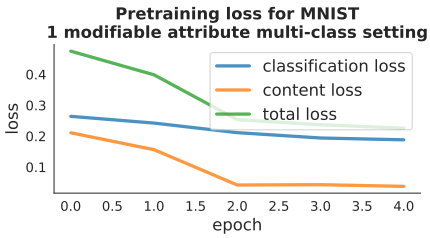

Figure B.1: MNIST pretraining GCN performance after 5 epochs on single attribute setting

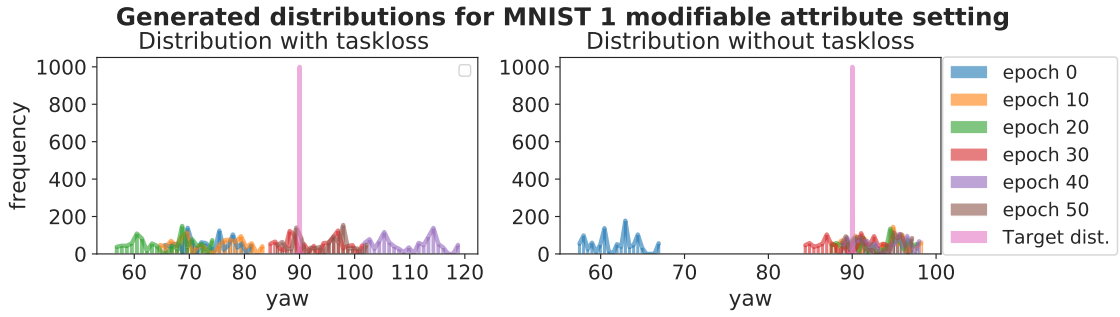

Figure B.2: Distributions of the *yaw* value from generated datasets every 10 epochs, compared to the target distribution for the MNIST 1 attribute.

From this plot we can see that the initial baseline experiments use a Dirac as a target distribution, which may complicate the learning. We can also see that the multi-class nature of the initial distribution defined in the PSG, lets the network generate

multi-modal distributions. We can finally see that the generated distribution correctly converges to the desired mean, with much more variance for the taskloss. This may be due to the first updates from the TaskNet, while it itself hasn't converged.

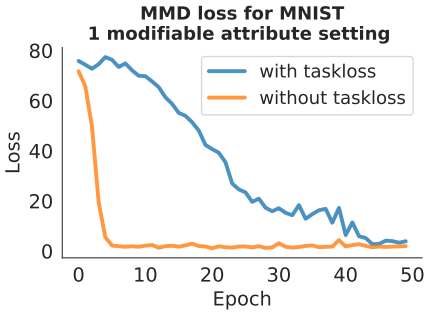

Figure B.3: MMD loss after 50 MMD training epochs for the MNIST 1 attribute setting.

This slowness in convergence can be confirmed by plotting the MMD loss obtained during the training of the network for 50 MMD training epochs. We can confirm that both networks eventually minimize the distribution loss between the target, and generated distribution. This learning can be further seen in the generated digits below.

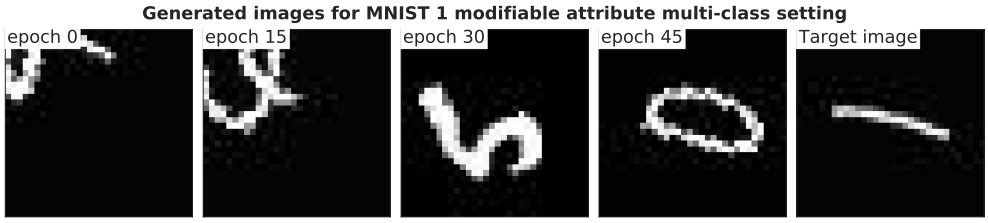

Figure B.4: Randomly sampled images from generated datasets every 15 epochs, compared to a randomly sampled images from the target dataset.

B.2 Learning to rotate and translate digits

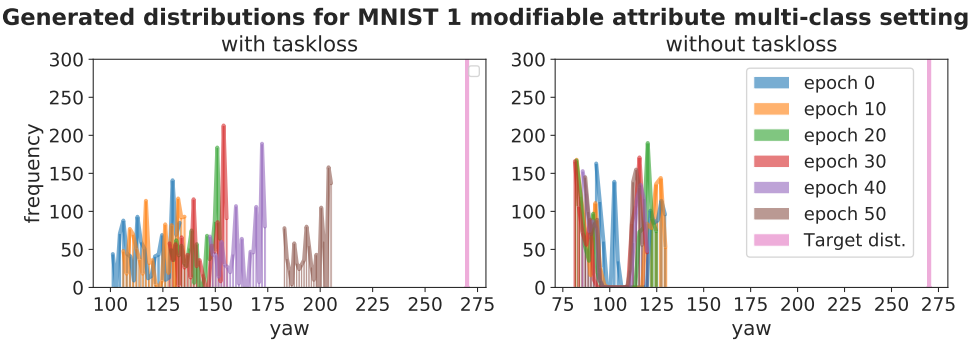

Figure B.5: Distributions of the *yaw* value from generated datasets every 10 epochs, compared to the target distribution for the MNIST 3 attribute setting.

We now experiment with the 3 attribute implementation provided by the authors, where we add the *loc x* and *loc y* attributes and train an architecture with taskloss, and another one without taskloss for 50 MMD training epochs. We observe the generated distributions both for the *yaw* attribute and the *loc x* and *loc z* attribute by plotting the histograms of the former in Figure X, and the scatter plot of the latter in Figure Y.

From these distributions we can see that there is actually no convergence towards the target mean *yaw*. On the contrary, the learned distributions stagnate around $\approx 200°$ and $\approx 100°$ for both experiments. This is interesting, since the architecture should be learning the target mean, according the original study, we will investigate this in depth in the original study. Furthermore, we can see that the architecture is correctly learning the target *location* distribution, much better without the taskloss than with the taskloss.

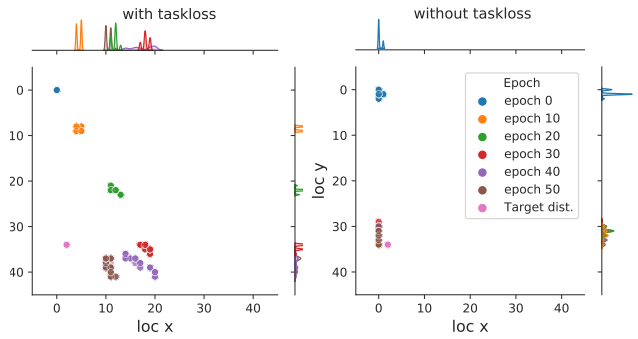

Figure B.6: Distribution of the *loc x* and *loc y* values for the generated datasets every 10 epochs of MMD training.

We can confirm this phenomenon by looking at the convergence of the MMD loss for both experiments. Indeed, the convergence is much faster when we don't use the taskloss. Let us look at the generated MNIST digits in the following figure. From these digits we can see that the architecture *is* learning to rotate the digits, just not to the correct target angle. If we are rigorous, we can say that the architecture is *not* learning the target yaw value.

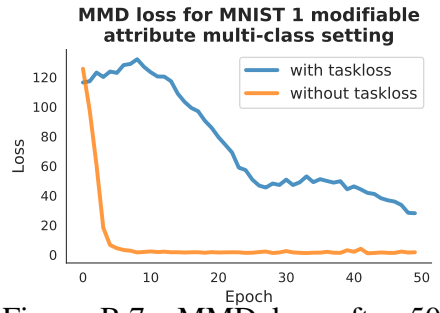

Figure B.7: MMD loss after 50 MMD training epochs for the single-class MNIST experiments.

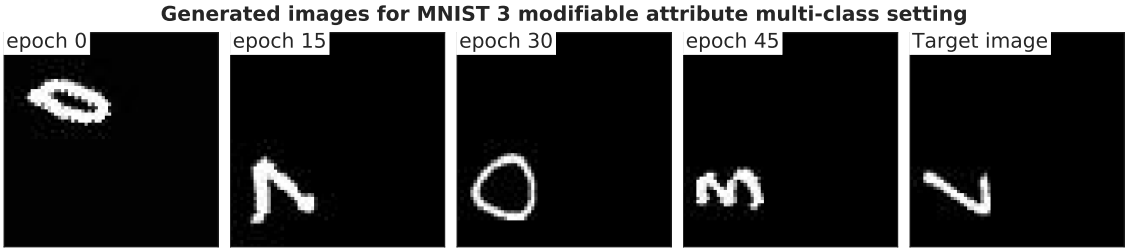

Figure B.8: Randomly sampled images from generated datasets every 15 epochs, compared to a randomly sampled images from the target dataset.

Appendix C

Feature representation of graphs in a 3D low-dimensional space

As per the probabilistic grammar shown above, we observe that there are 4 definite nodes, out of which all attributes are mutable, except for the *yaw* attribute which is mutable on the 3rd node belonging to the object node. As explained before, the GCN learns to reproduce a featurized representation of the graphs sampled from the probabilistic grammar. This featurized representation is reproduced by batches of size (batch_sizexnodes_numberxfeatures_number). We can therefore grab those matrices and map them to a lower dimensional space using Principal Component Analysis , and observe how each node is represented. We can further plot how this representation evolves every 100 epochs in the training.

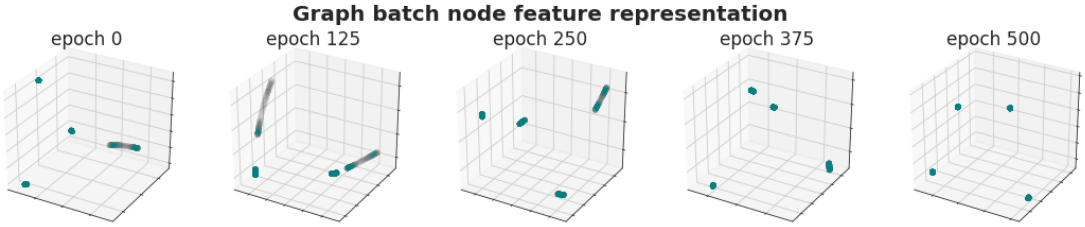

Figure C.1: 3D feature representations from generated graphs, every 200 epochs of MMD training.

From this plot we can observe an initial desired behavior in epoch 0, where 3 nodes are static in space, and the 4th one varies. We assume this node is the one belonging to the object with the modifiable attribute yaw. As the architecture evolves and learns, we would hope to see the 4th node vary, and the rest remain static, until convergence is reached and the target distribution is learned. This however is not the case, from

epoch 200 to epoch 400, the 4 nodes move along the space with no aparent pattern, to finally remain static at epoch 500. We further investigate the evolution of the generated distribution in the following study.

Appendix D

Fitting a Gaussian target distribution with the MNIST configuration

Since the dirac target distribution forces mode collapse in the network's training, we will address this by having a normal distribution of mean 90 degrees, and standard deviation 10 degrees, as our target distribution. We will run this experiment for 100 epochs. We observe the same total loss behavir than before, since we didn't change the number of reconstruction epochs. Let's look at the distribution of the generated samples to see how the generated distributions behaved after 100 epochs of training, every 25 epochs.

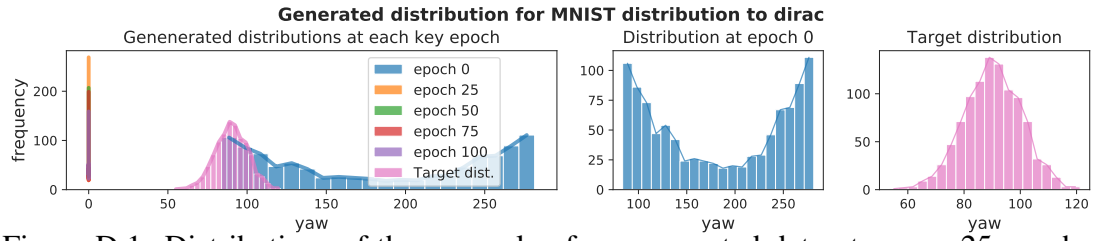

Figure D.1: Distributions of the *yaw* value from generated datasets every 25 epochs, compared to the target distribution for the 1 attribute single-class setting.

As we can see, the same type of mode collapse is happening, and, since we didn't change the probabilistic grammar, the same type of "inverted Gussian" effect is observed on each distribution. We can further confirm no learning from the generated images.

We can further conclude mode collapse by looking at the MMD loss, and the gradient behavior through the Graph Convolutional Network. Which can be seen in Figure X. Along with the gradients from the other two experiments. In this case, the MMD loss seems to have reached a local optimum in a lower loss than the previous exper-

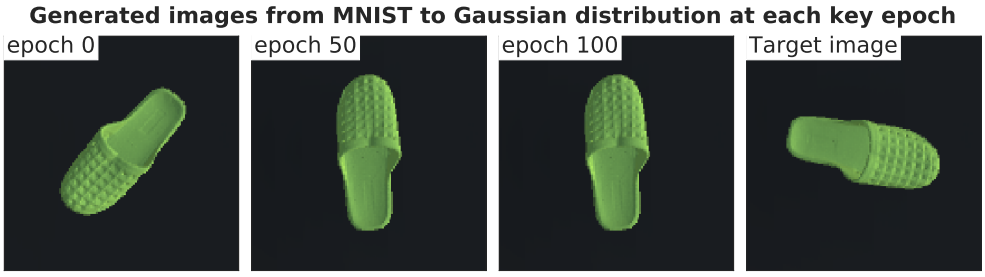

Figure D.2: Randomly sampled images from generated datasets every 50 epochs, compared to a randomly sampled images from the target dataset.

iment, but still a much higher point than desired, and still at angle of 0 degrees. As we can see, the gradients are collapsing much quicklier this time, compared to the previous experiment. The gradients however, are almost half smaller than the previous experiment. This seems to be due to the target distribution being much more spread out than the dirac distribution used in the previous experiment. Let us confirm this by looking at the distribution of generated distribution maps.

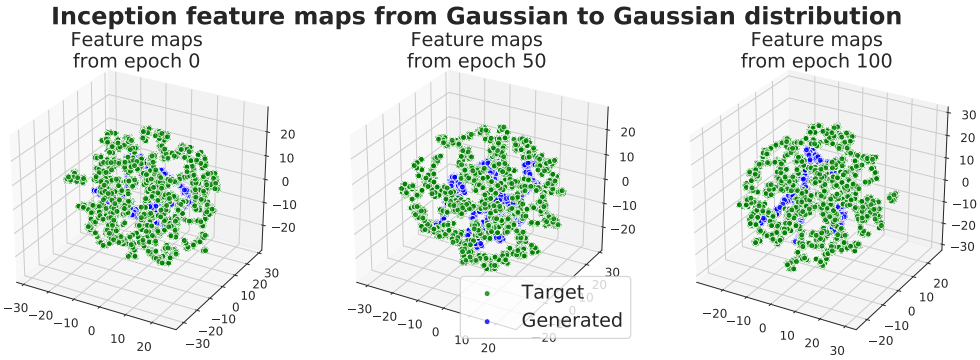

Figure D.3: 3D feature representations from the feature maps generated by the InceptionV3 network, every 50 epochs of MMD training.

We can quickly notice that the target distribution feature representation has significantly changed, thanks to the shift from a dirac impulse to a Gaussian distribution. Also, both distributions seem to be in the same general space, compared to the previous with the dirac impulse, where both 3D distributions where completely far from each other. Also, in future experiments we will keep an eye on the number of samples we use for comparison, because we can evidently see that, thanks to the generated batch size of 50, versus the target batch size of 200, there are much more green samples, than blue ones. This must affect the distribution fitting.

Appendix E

Widening the generated distribution while learning the target mean

E.1 Lowering the learning rate

It seems clear now that in order to recreate the pre-defined distribution we need to use a significant amount of pre-training epochs, this amount increases as the graphs get more complicated (e.g. more nodes in the graphs, and more features). However, it is also evident now, from the results we have seen, that as we increase the number of pre-training epochs, the model is more susceptible to mode collapse. This is why we have to control the gradients in some way, we previously did this using learning rate decay, but in this set of experiments we will simply reduce the learning rate.

Nevertheless, we are aware that the learning rate also affects the pre-training stage, so we will try a learning rate of 0.0005 and 0.0001 (so twice and 10 times smaller than the previous experiments), and then 0.0001 but pre-trained for 100 epochs, since in reality we are also slowing down pre-training progress when reducing the learning rate. This can clearly be seen in the following plots, which show the GCN pre-training performance.

From these plots we can see that the pre-training of the network is being slowed down by the low learning rate. This can be in reality harmful to our objective of learning a wider distribution predefined by the probabilistic grammar. This is why for such a low learning rate as 0.0001, we further increase the pretraining epochs to 100. From the third figure from left to right above we can see that the loss eventually converges to 0. This behavior is reflected in the produced distributions after 10 epochs of MMD training. We can see them below.

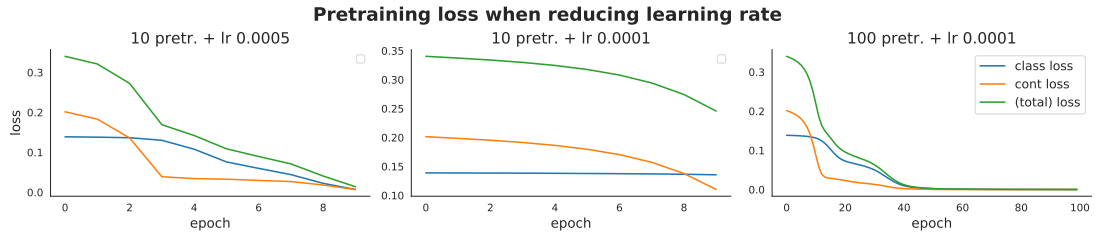

Figure E.1: GCN pretraining loss after 10 pretraining epochs when reducing learning rate, on the single-class 1 attribute experiments.

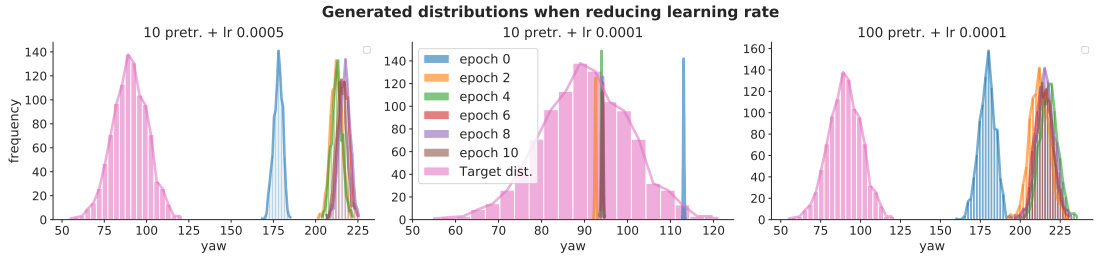

Figure E.2: Distributions of the *yaw* value from generated datasets every 2 epochs, compared to the target distribution for the single-class 1 attribute setting.

From the generated distributions we observe the same paradigm we have been trying to avoid. If we allow the pre-training of the network to minimize the loss up to 0, like in the case of a learning rate of 0.0005, the generated distributions will have a large standard deviation, that resembles the one specified in the probabilistic grammar. This can be seen for both the first and third plot fromn left to right. However, if we don't let it minimize the loss up to 0, this allows the MMD training to learn a mean close to the target mean, but without a large standard deviation, much less the one specified in the probabilistic grammar. This is further observed in the MMD loss, where there is convergence for the learning rate of 0.0001 and 10 epochs, and not for the others.

From these results we can see that using a leearning rate of 0.0001 and 10 epochs allows the network to converge towards a mean around 90 degrees, and therefore an MMD loss of around 1. On the contrary, the other learning rates which were more agressive and allowed a "better" pretraining of the network don't allow it to converge. Once again this is reflected in the gradients of the network, which are signficantly higher for the experiments that did not converge.

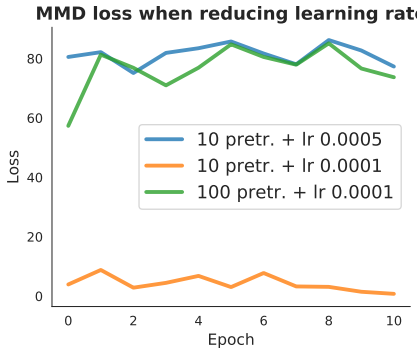

Figure E.3: MMD loss after 10 MMD training epochs for the single-class 1 attribute experiments when reducing learning rate.

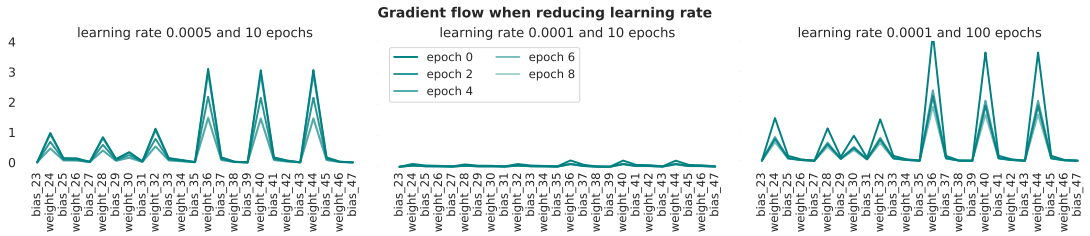

Figure E.4: Gradient flow through the GCN decoder layers for the experiments performed on the 1 attribute single-class setting when reducing learning rate.

From this set of experiments, we can visualize the generated images from training with learning rate 0.0001 with 10 epochs of pretraining. This experiment does provide correct convergence, and the generated distribution is indeed slightly wider than the initial experiment that only used 1 pretraining epochs. In the following figure we can see the output of the renderer.

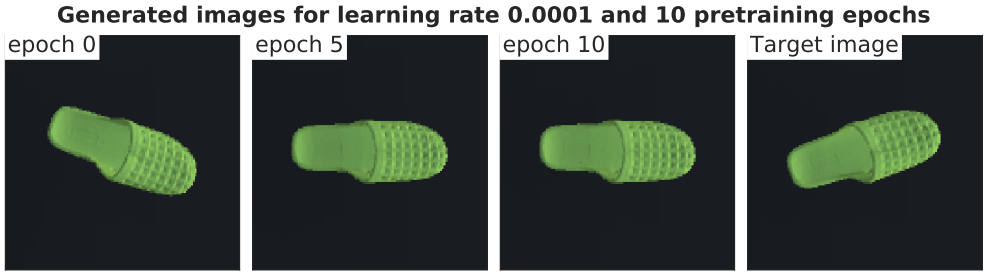

Figure E.5: Randomly sampled images from generated datasets every 5 epochs, compared to a randomly sampled images from the target dataset.

E.2 Increasing weight decay

Additionally, we try increasing the L2 weight penalty value with the hopes of controlling the high gradients resulting from pretraining the GCN for more than 1 epoch. We plot the GCN performance for the three attempted values and observe that evidently this also penalizes the learning during the GCN pretraining.

This penalization is reflected in the width of the generated distributions where, for larger weight decays, the distributions' width is narrower. Since the

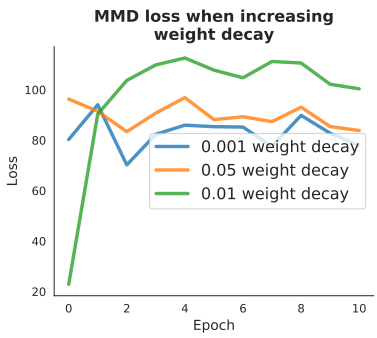

Figure E.6: MMD loss after 10 MMD training epochs for the single-class 1 attribute experiments when increasing weight decay.

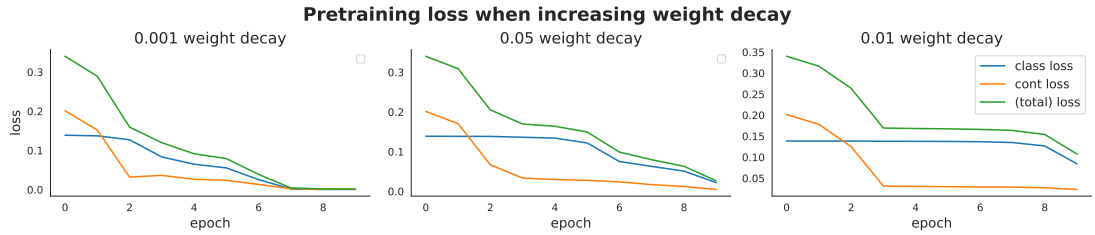

Figure E.7: GCN pretraining loss after 10 pretraining epochs on the single-class 1 attribute setting when increasing weight decay.

architectures with larger weight decay don't fully converge, they don't learn to correctly reproduce the distributions specified in the PSG.

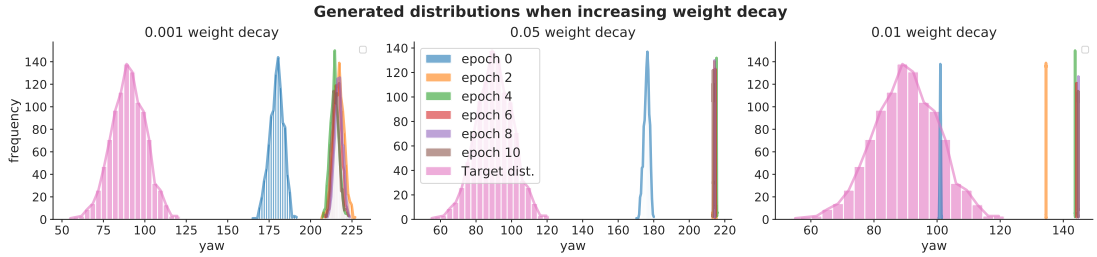

Figure E.8: Distributions of the *yaw* value from generated datasets every 2 epochs, compared to the target distribution for the 3 attribute setting.

We confirm that the weight decay does not help the learning by plotting the MMD loss obtained when increasing the weight penalty. This is further reflected in the gradients seen in Figure X, where we still have large gradients for weight decay of 0.01, and 0.05, and 0.01 simply penalizes the learning too much.

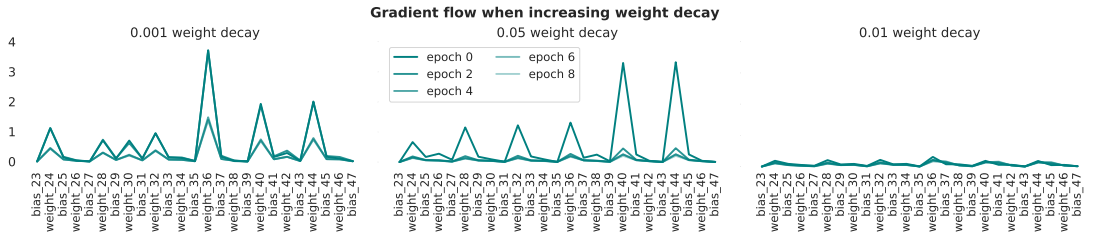

Figure E.9: Gradient flow through the GCN decoder layers for the experiments performed on the 1 attribute single-class setting when increasing weight decay.

E.3 Lowering the MMD loss multiplier

The MMD loss multiplier is a factor used in the implementation to increase the magnitude of the initially calculated loss. We interpret this value as having a direct impact

on the magnitude of the gradients and therefore experiment with 50, and 20, where the original value was 100. We plot the pretraining loss as per usual, and observe that this indeed doesn't affect the pretraining as the previous two hyperparameters, since the multiplication is only done during the MMD training.

We plot the generated distributions, but observe that the values are simply collapsing to 0. This confirms that varying this value does not help the learning of the network, since even though the magnitude of the loss, and therefore the updates is being changed, the initial update from the pretraining may be too high, and therefore vanish the gradients from the first epoch. This may be confirmed by plotting the gradients in Figure Y.

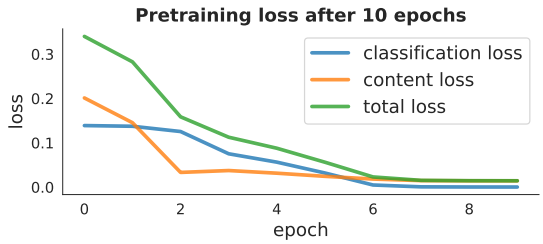

Figure E.10: GCN pretraining loss after 10 pretraining epochs on the single-class 1 attribute setting when reducing the MMD multiplier.

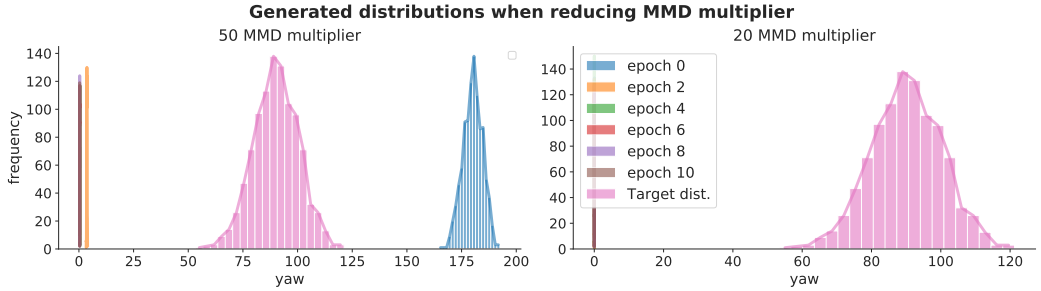

Figure E.11: Distributions of the *yaw* value from generated datasets every 2 epochs, compared to the target distribution for the 3 attribute setting.

We can see that the object is not being rotated by plotting randomly sampled generated images at each epochs, compared to a randomly sampled target image.

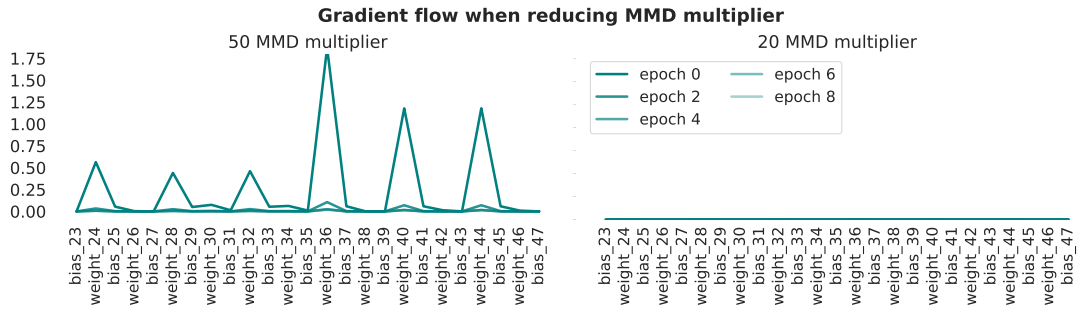

Figure E.12: Gradient flow through the GCN decoder layers for the experiments performed on the 1 attribute single-class setting when reducing the MMD multiplier.

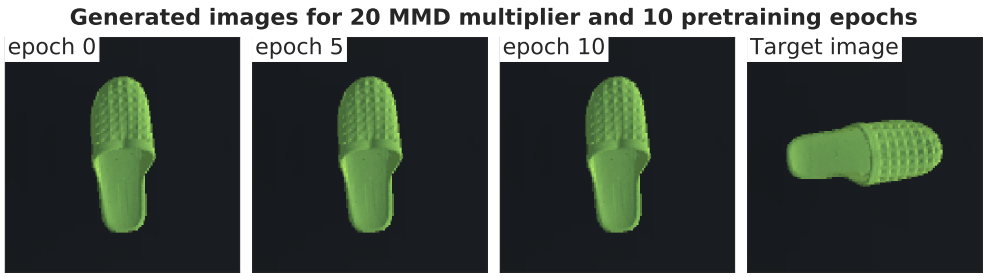

Figure E.13: Randomly sampled images from generated datasets every 5 epochs, compared to a randomly sampled images from the target dataset.