

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Daniel Prado Mendes de Mello

**REDES GENERATIVAS ANTAGÔNICAS PARA CLUSTERIZAÇÃO
HIERÁRQUICA**

Belo Horizonte
2021

Daniel Prado Mendes de Mello

**REDES GENERATIVAS ANTAGÔNICAS PARA CLUSTERIZAÇÃO
HIERÁRQUICA**

Versão final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Renato Martins Assunção

Coorientador: Fabrício Murai Ferreira

Belo Horizonte
2021

Daniel Prado Mendes de Mello

GENERATIVE ADVERSARIAL NETWORKS FOR HIERARCHICAL CLUSTERING

Final version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Renato Martins Assunção

Co-advisor: Fabrício Murai Ferreira

Belo Horizonte
2021

Mello, Daniel Prado Mendes de.

M527r Redes generativas antagônicas para clusterização hierárquica
[manuscrito] / Daniel Prado Mendes de Mello. – 2021.
69 f. il.

Orientador: Renato Martins Assunção.

Coorientador: Fabrício Murai Ferreira.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f.65-69

1. Computação – Teses. 2. Método de clusterização – Teses. 3. Aprendizado profundo – Teses. 4. Redes generativas – Teses. I. Assunção, Renato Martins. II. Ferreira, Fabrício Murai. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*85(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

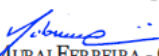
FOLHA DE APROVAÇÃO

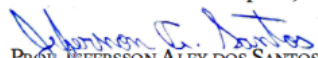
Redes Generativas Antagônicas Para Clusterização Hierárquica

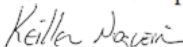
DANIEL PRADO MENDES DE MELLO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. RENATO MARTINS ASSUNÇÃO - Orientador
Departamento de Ciência da Computação - UFMG


PROF. FABRÍCIO MURAI FERREIRA - Coorientador
Departamento de Ciência da Computação - UFMG


PROF. JEFERSSON ALEX DOS SANTOS
Departamento de Ciência da Computação - UFMG


PROF. KEILLER NOGUEIRA
STIR - University of Stirling

Belo Horizonte, 28 de Maio de 2021.

Acknowledgments

My family for their support and my advisors for their time and attention.

Resumo

Métodos de clusterização profunda alcançaram recentemente um progresso substancial, aproveitando o poder de representação de arquiteturas de aprendizagem profunda para aprender representações que são ideais para análise de clusters. No entanto, poucos esforços foram feitos para tentar combinar clusterização hierárquica, uma abordagem clássica muito útil que não assume um número fixo de clusters, com métodos profundos. Neste trabalho, propomos uma nova maneira de realizar clusterização profunda de forma hierárquica *top-down*, onde cada divisão binária em uma árvore de subdivisões é realizada por uma rede generativa antagônica de dois geradores. Mostramos o quão bem nosso método se compara a outras técnicas de clusterização profunda em bases de dados para clusterização, obtendo resultados competitivos, bem como uma exploração da árvore de clusterização hierárquica, verificando como ela organiza com precisão os dados de treinamento em uma hierarquia de características semanticamente coerentes, conforme esperado.

Palavras-chave: Aprendizagem Profunda. Clusterização. Redes Generativas Antagônicas.

Abstract

Deep clustering methods have recently achieved substantial progress by leveraging the representation power of deep architectures to learn embedding sub-spaces that are optimal for cluster analysis. Nonetheless, very few effort has been done in combining hierarchical clustering, a very useful classical approach that does not assume a fixed number of clusters, with deep methods. In this work, we propose a new way of performing deep clustering in a top-down hierarchical manner, such that each binary split in a tree of subdivisions is performed by a two-generator generative adversarial network. We show how well our method compares to other deep clustering techniques on clustering datasets, obtaining competitive results, as well as an exploration of the hierarchical clustering tree, verifying how it accurately organizes the training data in a hierarchy of semantically coherent characteristics, as expected.

Keywords: Deep Learning. Clustering. Generative Adversarial Networks.

List of Figures

2.1	Illustration of mode collapse occurring during the training of a GAN trained with a mixture of 8 Gaussian distributions (red). Generated samples (blue) are incapable of representing all the modes of the real distribution.	19
2.2	Generator used in the DCGAN architecture. Image reproduced from [Radford et al., 2016]	19
2.3	Illustration of the training progress of a MGAN with 8 generators trained with a mixture of 8 Gaussian distributions (red). Generated samples (blue) are capable of representing all the modes of the real distribution, eliminating mode collapse.	20
2.4	Example of an architecture with multiples generators. Image reproduced from [Hoang et al., 2018].	21
2.5	Example of result after training an architecture of 10 generators with the MNIST dataset, with each line corresponding to generated samples of one of the generators.	21
3.1	Simplified view of our method’s hierarchical clustering. Each cluster k is associated with a membership vector \mathbf{s}_k , which contains the probabilities of each i -th training example belonging to k . Each split block receives a \mathbf{s}_k as input and creates two new membership vectors from it, each associated with a subcluster of cluster k	26

3.2	Complete hierarchical tree constructed over the MNIST dataset. Each vector \mathbf{s}_k is placed together with 25 real examples sampled from cluster k 's probability distribution. This distribution is formed by sampling an example i with weights given by the i -th probability in \mathbf{s}_k . To indicate the most prevalent classes in each \mathbf{s}_k , we represent in parenthesis, scaled by font size, the sum of the probability mass of all the examples belonging to the class inside a given \mathbf{s}_k . To be more precise, the probability mass of a class A inside \mathbf{s}_k is given by $\sum_i^N s_{k,i} \cdot \mathbf{1}(c_i = A)$, where $\mathbf{1}$ is an indicator function and c_i is the class of the i -th example. The scale linearly ranges from font size 2 (closer to 0% mass, 0 mass) to font size 7 (closer to 100% of mass, or approximately 7000 of mass for the approximately 7000 examples per class in MNIST).	27
3.3	Raw split and refinement sub-blocks inside a split block. Each $\mathbf{s}_l^{(t)}$ and $\mathbf{s}_m^{(t)}$ constitute membership vectors estimates that are gradually transformed into the final estimates \mathbf{s}_l and \mathbf{s}_m	31
3.4	(Best viewed in color) Visualization with a MNIST example of a \mathbf{s}_k , whose probabilities are mostly associated with 3's and 5's, passing through a raw split sub-block and then 2 refinement sub-blocks. The probabilities of each individual example in the sample is indicated by its intensity in gray scale, with more white indicating close to 100 % probability. Like we did for Figure 3.2, we once again use the font size scale, in parenthesis, alongside each block, to indicate the amount of probability mass per class assigned to it. Note that at each t , the probabilities sum up to \mathbf{s}_k , <i>i.e.</i> , $\mathbf{s}_m^{(t)} + \mathbf{s}_l^{(t)} = \mathbf{s}_k$	33
3.5	(Best viewed in color) Training the Raw Split Components. Two generators $G\alpha_k$, $G\beta_k$, one discriminator D_k and a classifier C_k . $\mathcal{P}_{\mathbf{s}_k}$ draws samples weighted by the amount of probability each example has in \mathbf{s}_k	34
3.6	(Best viewed in color) Clustering the dataset with the raw split classifier.	35
3.7	(Best viewed in color) Training the Refinement Components.	38
3.8	(Best viewed in color) Clustering the dataset with the refinement classifiers	40
4.1	Random samples from the MNIST dataset.	45
4.2	Random samples from the Fashion MNIST dataset.	46
4.3	Random samples from the CIFAR-10.	47
4.4	Top 10 real MNIST images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.	51
4.5	Top 10 real Fashion MNIST images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.	51

4.6	Top 10 real CIFAR-10 images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.	52
4.7	Complete hierarchical tree constructed over the Fashion MNIST dataset. Each vector \mathbf{s}_k is placed together with a square grid containing 25 real examples sampled from cluster k 's probability distribution (a distribution formed by sampling a real example i with weights given by the i -th probability in \mathbf{s}_k). The leaf nodes samples are indicated with a segmented red line for the borders of the square grid. Along with each square grid, in parenthesis, we indicate the name of the classes in varying font sizes, with each font size proportional to the amount of probability mass the class has in \mathbf{s}_k , using 10 font sizes, linearly representing 10 intervals, ranging from closer to 0% of total mass (or 0 mass) to closer to 100% of total mass, (or 7000 of mass).	53
4.8	Complete hierarchical tree constructed over the CIFAR-10 dataset. Each vector \mathbf{s}_k is placed together with a grid containing varying numbers of real examples sampled from cluster k 's probability distribution (a distribution formed by sampling a real example i with weights given by the i -th probability in \mathbf{s}_k). The leaf nodes samples are indicated with a segmented red line for the borders of the grid. Along with each grid, in parenthesis, we indicate the name of the classes in varying font sizes, with each font size proportional to the amount of probability mass the class has in \mathbf{s}_k , using 10 font sizes, linearly representing 10 intervals, ranging from closer to 0% of total mass (or 0 mass) to closer to 100% of total mass, (or 6000 of mass).	55
5.1	Raw split sub-block adapted for data augmentation	61

List of Tables

4.1	Main results: Clustering performance of different algorithms on 3 datasets based on ACC and NMI. The column “Deep?” indicates if the respective algorithm is based on deep learning or not. *: Algorithms and results that made use of data augmentation techniques aimed for clustering.	49
4.2	Architecture settings.	57
4.3	Specific hyperparameter settings used for each dataset.	58

Contents

1	Introduction	14
2	Background and Related Work	17
2.1	GANs	17
2.2	GANs with Multiple Generators	20
2.3	Deep Clustering	22
3	Methodology	25
3.1	Method Overview	25
3.2	Basic Notation for Hierarchical Soft Clustering	26
3.2.1	Notation for soft clustering	28
3.2.2	Expanding the notation for hierarchical soft clustering	28
3.2.3	Hierarchical clustering initial state: before the first split	28
3.2.4	Generalization: after j splits	29
3.2.5	Referring to membership vectors at the nodes of the tree	29
3.2.6	Describing a tree with an example	30
3.3	Performing the Split	30
3.3.1	Raw Split Phase	32
3.3.2	Refinement Phase	38
4	Experiments	45
4.1	Datasets	45
4.1.1	MNIST	45
4.1.2	Fashion MNIST (FMNIST)	46
4.1.3	CIFAR-10	46
4.2	Evaluation Metrics	47
4.3	Main Results	48
4.3.1	Baselines and state-of-the-art methods	48

4.3.2	Method Comparison	48
4.3.3	Qualitative Analysis	50
4.4	Implementation Details	56
4.4.1	Deep learning framework and hardware	56
4.4.2	Architecture	56
4.4.3	Hyperparameters	57
5	Future Works	59
5.1	Training Time	59
5.2	Hyperparameter and Architecture Choices	60
5.3	Data Augmentation	60
6	Conclusion	63
	Bibliography	65

Chapter 1

Introduction

Cluster analysis is a fundamental research field of unsupervised learning, with a wide range of applications, especially for computer vision related tasks [Shi and Malik, 2000, Achanta and Susstrunk, 2017, Joulin et al., 2010, Liu et al., 2018]. Its goal is to assign similar points of the data space to the same cluster, while ensuring that dissimilar points are placed in different clusters. One of the main challenges within this approach is to quantify the similarity between objects. For low-dimensional data spaces, similarity might be straightforwardly defined as minimizing some geometric distance (*e.g.* euclidean distance, squared euclidean distance, Manhattan distance). On the other hand, choosing the distance metric becomes unfeasible for high dimensional data distributions. Images are a clear example of this problem, since any distance metric based on raw pixel spaces is subject to all sorts of low-level noisy disturbances irrelevant for determining the similarity. This problem motivates the need for some dimensionality reduction technique, in which the fundamental relationships between objects projected onto the resultant embedding space become more consistent with geometrical distances.

In recent years, deep clustering techniques have spearheaded the dimensionality reduction approach to clustering by employing the highly non-linear latent representations learned by deep learning models [Krizhevsky et al., 2012]. Considering the unsupervised nature of cluster analysis, the models that naturally arise as candidates for deep clustering are unsupervised deep generative models, since these must learn highly abstract representations of the data as a requirement for realistic and diverse generated samples. One of such models are the Generative Adversarial Networks (GAN) [Goodfellow et al., 2014], whose extremely realistic results in image generation, semantic interpolation and interpretability in the latent space, are evidence of their capacity of learning a powerful latent representation capable of capturing the essential

components of the data distribution. In short, the basic formulation consists of a min-max game between two players: the discriminator and the generator, both modeled by neural networks. The discriminator is trained to distinguish between a real data distribution and a fake distribution, while the generator is trained to generate this fake distribution in a sufficiently convincing manner, such that it fools the discriminator into classifying a fake generated sample as belonging to the real distribution. This scenario tends to result in a generated distribution increasingly closer to the real distribution, which is the main objective of generative learning. Nonetheless, very few works have proposed GAN architectures designed for clustering. Some of the few works exploring this possibility are [Mukherjee et al., 2019, Chen et al., 2016], where the authors showed that by manipulating the Generator’s architecture in a specific way we could control to which class of the training data a generated sample belongs, even if the class labels aren’t available during the training.

Some recent works employed a GAN architecture with multiple generators [Ghosh et al., 2018, Hoang et al., 2018, Zhang et al., 2018] in order to obtain a greater diversity of image generation, as well as an alternative way of stabilizing the training. In these works, a curious side effect phenomenon is observed, in which each generator tends to specialize in generating examples belonging to a specific class of the dataset, strongly suggesting the possibility of clustering. In this case, the clustering would be made possible by employing a classifier in charge of distinguishing between the generators, and this classifier could later be applied to the real dataset in order to classify real examples without the use of labels. However, this alternative was not explored in the cited works. Additionally, the question of how to infer the number of generators to be used to represent the classes of the training set is not clear in these works, which could configure a problem in a real clustering task, where the number of clusters is assumed to be unknown.

In this work, we intend to take advantage of the GAN architecture with multiple generators, having exclusively the clustering task as a goal. The main motivation for employing this model is that we believe that, by employing multiple generators with each generator specializing in representing a particular cluster of the training distribution, we might obtain an even stronger capacity of representation, and thus more meaningful clusters, than it would be possible with a single generator covering all clusters, such as in previous works that explored GANs for clustering. Additionally, we design our method so that it not only performs the clustering of the training data, but also does it in a hierarchical way, creating new generators as divisions of subsequent clusters become necessary, *i.e.*, it permits the user to control different clustering granularity levels according to the task at hand.

The main objective of this work is to demonstrate, through the experimental route, the potential of GANs with multiple generators for the unsupervised clustering task, obtaining, with a new training algorithm, state-of-the-art results of clustering accuracy in image datasets and a first deep learning based clustering alternative performed hierarchically.

Chapter 2

Background and Related Work

2.1 GANs

In a game theory context, a GAN is modeled by a game between two adversarial components: a discriminator D and a generator G . Both usually are parameterized by neural networks, with parameters θ_D e θ_G . The generator performs a mapping from the latent space to the training data space, represented as $\mathcal{G} : \mathcal{Z} \rightarrow \mathcal{X}$. The discriminator, on the other hand, performs a mapping from the training data space to a real number, represented as $\mathcal{D} : \mathcal{X} \rightarrow \mathbb{R}$, where the real number measures the probability of the input data being fake or authentic. The latent space \mathcal{Z} is usually modeled by either a Gaussian or a uniform distribution, with dimensionality much smaller than \mathcal{X} . The GAN minimax game under a loss function $\mathcal{L}_{adv}(G, D)$ is defined by

$$\min_{\theta_G} \max_{\theta_D} \mathcal{L}_{adv}(G, D) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{data}} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z} \sim \mathcal{P}_z} [\log(1 - D(G(\mathbf{z})))] \quad (2.1)$$

where \mathbf{x} is a random variable sampled from the real data distribution, and \mathbf{z} is the random variable sampled from latent distribution, which is transformed into a generated image by the functions modeled by the generator.

Equivalently, we might as well represent Equation (2.1) with a substitution of the transformation $G(\mathbf{z})$ by the variable \mathbf{x} sampled from the distribution \mathcal{P}_{gen} of the generated images:

$$\min_{\theta_G} \max_{\theta_D} \mathcal{L}_{adv}(G, D) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{data}} [\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{gen}} [\log(1 - D(\mathbf{x}))]. \quad (2.2)$$

In Equation (2.2), the GAN simultaneously optimizes both G and D , so that at the ending of the procedure, ideally, the distributions \mathcal{P}_{gen} and \mathcal{P}_{data} become indistinguishable from each other; ending up in a situation where, if we assume a perfect generator, the optimum strategy for the discriminator becomes to assign 50% probability to any given image being real or false. More precisely, in [Goodfellow et al., 2014] it was shown that the minimization of the generator’s loss function, in the presence of a perfect discriminator, would be equivalent to minimizing $D_{JS}(p_{data}(\mathbf{x})||p_{gen}(\mathbf{x}; \boldsymbol{\theta}_G))$, the Jensen-Shannon divergence between the generated distribution and the real distribution. which only reaches its minimum when both the distributions are the same. This minimization is also equivalent to the process of maximizing the likelihood of the generated distribution.

Despite these theoretical guarantees, in the minimax game of Equations 2.1 and 2.2, the generator suffers from the problem of having gradients too close to zero when the discriminator correctly assigns a probability close to zero to the generated images. This situation compromises the generator’s learning, especially at the initial stages of the training, when it is easier to discriminate between real images and fake images. A heuristic alteration was proposed by [Goodfellow, 2016] in order to overcome this problem: instead of *minimizing* the component $\mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{gen}}[\log(1 - D(\mathbf{x}))]$, the only part dependent of $\boldsymbol{\theta}_G$ in the minimax formulation, the generator is now optimized to *maximize* a new loss function $\mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{gen}}[\log(D(\mathbf{x}))]$. This change might be understood as a way of making the generator maximize the probability of the discriminator mistakenly classifying false positives, instead of minimizing the probability of the discriminator correctly classifying true negatives, as in the minimax formulation. This new loss function for the generator became known as non-saturating loss, once it was conceived with an intention of facilitating the gradient flux for the generator. The loss function for the discriminator remains unchanged.

In practice, training a GAN involves a series of difficulties, with the main ones related to convergence problems during the minimax game. A common example is called mode collapse, which is when the generator reaches a local minimum in its optimization function and ends up generating samples with little diversity, whose quality is very little improved during the rest of the training. In Figure 2.1 we can see an example of this phenomenon. As the training progresses, the generated samples (in blue) end up concentrating in only one of the modes of a distribution formed by a mixture of 8 Gaussian components (red), and alternating between modes while doing so, which demonstrates this architecture’s incapacity to represent all the modes of the dataset at the same time.

Despite these difficulties, many training techniques and architecture variations

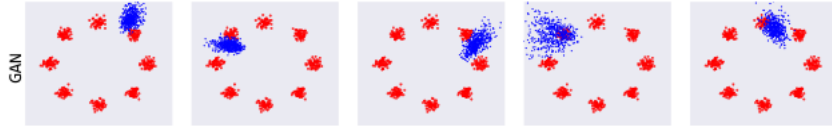


Figure 2.1. Illustration of mode collapse occurring during the training of a GAN trained with a mixture of 8 Gaussian distributions (red). Generated samples (blue) are incapable of representing all the modes of the real distribution.

have been proposed recently aiming to make the training more stable, with one of the first seminal works with this approach being [Radford et al., 2016], where the authors proposed a GAN architecture parameterized by convolutional neural networks (CNN), referred to as deep convolutional generative adversarial network (DCGAN), as well as a series of practical guidelines that helped to stabilize the training. In Figure 2.2 we reproduce the generator architecture used in the DCGAN work. Besides expressive results in image generation that were obtained by the authors, notable properties of interpolation in the generator’s latent space were also demonstrated, where it was possible to perform arithmetic operations with the latent vectors of generated images, whose result would be semantically coherent with such operations. This property demonstrated the representation power of GANs, with the possibility of disentangling high-level semantic characteristics present in the dataset.

The architecture we shall employ in this work is strongly influenced by the DCGAN architecture, the main difference being that it is generalized to multiple generators.

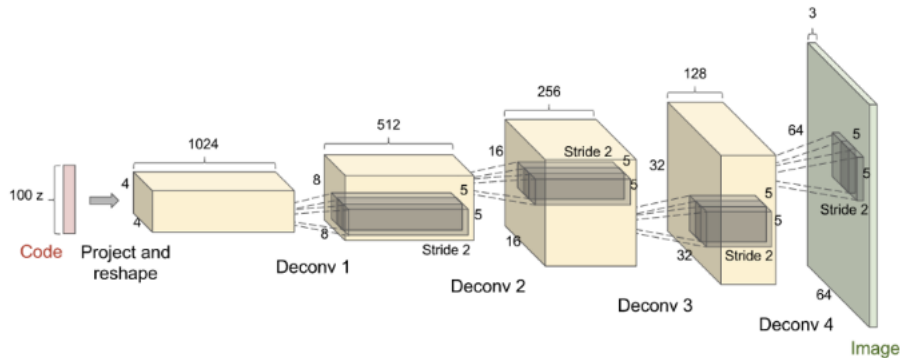


Figure 2.2. Generator used in the DCGAN architecture. Image reproduced from [Radford et al., 2016]

Recently, many works employing GANs have obtained increasingly more expressive results, especially in tasks related with computer vision, such as video and image generation [Radford et al., 2016, Brock et al., 2019, Tulyakov et al., 2018], image inpainting [Yu et al., 2018, Liu et al., 2019], semantic segmentation [Zhang et al., 2018],

image-to-image translation [Isola et al., 2017], style transfer [Karras et al., 2019], image synthesis from text [Reed et al., 2016].

2.2 GANs with Multiple Generators

As a way of solving the problem of modal collapse in GANs and accelerate the training convergence, several works [Ghosh et al., 2018, Zhang et al., 2018, Hoang et al., 2018, Khayatkhoei et al., 2018] proposed architecture variations that employ multiple generators trained simultaneously with a discriminator, which we refer to as MGANs.

The main motivation behind this approach is to facilitate the role of each generator in the adversarial game, since each generator is able to specialize in some specific mode of the distribution and still fool the discriminator, as a result of the presence of other generators specializing in other modes. This phenomenon is depicted in Figure 2.3, where, as in Figure 2.1, the training progression for a mixture of 8 Gaussian distributions is shown, but this time a MGAN with 8 generators is used. We can notice how each of the 8 modes of the distribution is being much better represented, thus eliminating problems related to convergence and modal collapse. This is a result of each generator’s specialization in one of the 8 modes of the real distribution.

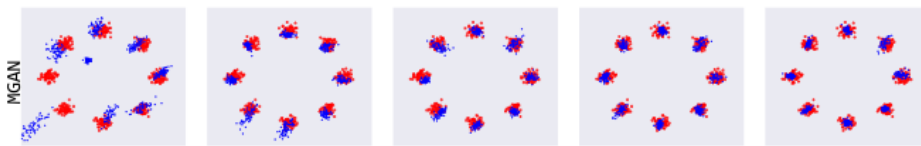


Figure 2.3. Illustration of the training progress of a MGAN with 8 generators trained with a mixture of 8 Gaussian distributions (red). Generated samples (blue) are capable of representing all the modes of the real distribution, eliminating mode collapse.

We can observe in Figure 2.4 an example of this type of architecture. A number of N generators are simultaneously trained, sharing the same latent distribution. The loss function is similar to the loss of a GAN with 1 generator, however we sum the expectations for each generated distribution obtained with each of the N generators employed. Hence, we shall have an optimization within the minimax game performed

with the parameters $\theta_{G_1}, \theta_{G_2} \dots \theta_{G_N}, \theta_D$:

$$\min_{\theta_{G_1, \dots, G_N}} \max_{\theta_D} \mathcal{L}_{adv}(G_{1,2, \dots, N}, D) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{data}} [\log D(\mathbf{x})] + \frac{1}{N} \sum_{k=1}^N \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G_k}} [\log(1 - D(\mathbf{x}))]. \quad (2.3)$$

In Figure 2.5 we show an example of application of this type of architecture trained with the MNIST dataset, where each line corresponds to samples generated by each generator in an architecture with 10 generators after the training is finished. We can see how each of the generators manages at the end to specialize in one of the classes of the dataset, *i.e.*, one of the digits.

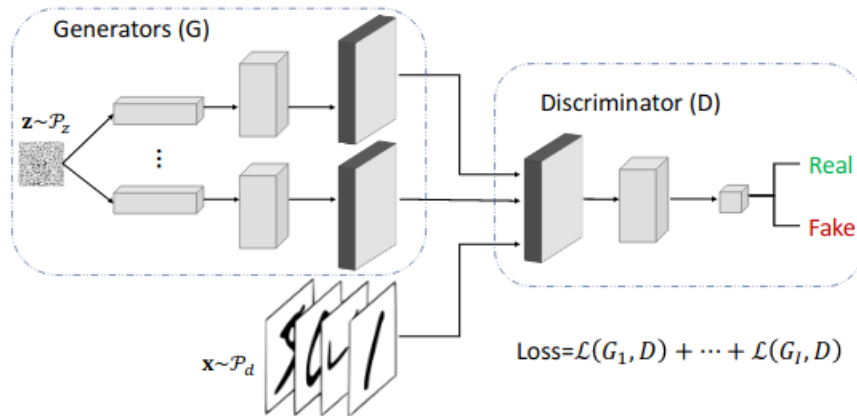


Figure 2.4. Example of an architecture with multiples generators. Image reproduced from [Hoang et al., 2018].



Figure 2.5. Example of result after training an architecture of 10 generators with the MNIST dataset, with each line corresponding to generated samples of one of the generators.

Ghosh et al. [Ghosh et al., 2018] were the first to propose this type of architecture, empirically demonstrating its properties of facilitating both the training convergence and the diversity of generation. Hoang et al. [Hoang et al., 2018] proposed to introduce a classifier of generators that is independent of the discriminator, whose classification loss function would also be minimized by the generators, encouraging the diversity of generated images. Zhang et al. [Zhang et al., 2018] contributed with various theoretical guarantees for a increase in convexity of the loss function as more generators are introduced. It is important to emphasize that in none of these works the possibility of using the architecture with multiple generators for clustering tasks was conceived, which is precisely the focus of the present work.

2.3 Deep Clustering

A very important task in the field of unsupervised learning is cluster analysis or clustering. The goal of cluster analysis is to group examples belonging to some data distribution into clusters, keeping examples similar to each other in the same cluster, and examples different from each other in different clusters.

Among the different classical approaches to cluster analysis, we can cite the k-means algorithm [MacQueen, 1967] as one of the most well-known and widely employed to an enormous range of tasks. The k-means algorithm is specially suited for clustering data samples that are approximately spread around separated cluster centers, called cluster centroids. Each centroid is defined by the mean of the elements assigned to it. The algorithm is iterative, fixing a group of k centroids randomly spread at a first step, while it determines the examples in the dataset that are closer to each of these k centroids; for the next step, it fixes a new clustering, assigning examples to a new cluster defined in regard to which centroid each example was closest to, and then it estimates new means for the centroids defined by these new clusters, thus obtaining a new set of k centroids. The process is then repeated, assigning a new clustering to previously estimated centroids at a step, and re-estimating new centroids for the new clusters at the next step. The algorithm is repeated until some sort of convergence or until a predefined limit of iterations.

Regarding the use of GANs for clustering tasks, we can cite the architecture named ClusterGAN [Mukherjee et al., 2019]. In the ClusterGAN work, the authors were strongly influenced by another architecture, the well-known InfoGAN [Chen et al., 2016], a type of GAN whose latent variable is formed by, besides the usual multidimensional variable z , a set c of one-dimensional variables c_1, c_2, \dots, c_N that

are expected to unsupervisedly capture semantic information in a disentangled manner (*i.e.*, with each variable codifying isolated interpretable features of the real data). For obtaining this, the authors of InfoGAN proposed an additional term in the generator’s loss function that maximized $I(c; G(z, c))$, the mutual information between a generated image and the latent variable that originated it. The variables c could be chosen to represent both categorical and continuous features. The ClusterGAN work adapted this architecture for the clustering task. For this architecture, the generator learns to generate a certain class of the real distribution in correlation with a given one-hot format for the latent variables c . To obtain this, they proposed to use an inference encoder network capable of performing the mapping $\mathcal{E} : \mathcal{X} \rightarrow \mathcal{Z}$, which is the inverse of the Generator’s mapping and similar to an encoder’s mapping for an autoencoder architecture. This encoder network is trained jointly with the generator for classifying a latent variable associated with each generated sample, such that after the training it can be employed to classify real data samples accordingly to which latent variable is mostly correlated with it, thus providing the clustering. One main difference in regard to our work is our use of multiple generators instead of multiple discrete latent variables, which enables us to discover clusters with much more representation capacity. Even more importantly, all the mentioned methods in this section assume a fixed and known optimum number of clusters that describe the real distribution (horizontal clustering), while for our approach we propose a hierarchical way of deep clustering with GANs, which can be performed recursively, without assuming any optimum number of clusters.

In [Kundu et al., 2019] the authors propose the GAN-Tree framework, which resembles our approach superficially, since it also involves a hierarchical structure of independent GAN nodes capable of generating samples related to different levels of a similarity hierarchy. There are multiple relevant differences, however. The most important one being that the main motivation for GAN-Tree was a framework capable of addressing the trade-off between quality and diversity when generating samples from multi-modal data. The authors claimed that their approach could be readily adapted for clustering tasks, but no definitive experiments with clustering benchmarks were provided. Other important difference lies in their splitting procedure, which was performed with a latent $\hat{\mathbf{z}}$ inferred by an encoder E for a sample image \mathbf{x} , that is, $\hat{\mathbf{z}} = E(\mathbf{x})$. For a node of the tree, they decompose their prior for \mathcal{P}_z into a mixture of two Gaussians with shifted means. They determine the prior component to which $\hat{\mathbf{z}}$ is more likely related, and then train the encoder to maximize the likelihood to this prior. For a clustering task, this approach would heavily rely on the assumption that the inference made by E is of sufficient semantic relevance, as well as that the decomposed

Gaussians in \mathcal{Z} will also be sufficient to capture clustering patterns with sufficient semantic relevance. The split in our approach, on the other hand, is directly embedded into the GAN training, with each generator automatically learning to represent each cluster. Therefore, in our work the clustering semantic quality is directly tied to the GAN’s well known representation learning capacity, and, in particular, to the tendency of different generators in MGANs to cover different areas of the training distribution with high semantic discrepancy.

Chapter 3

Methodology

3.1 Method Overview

In this section, we present a high-level overview of how our method operates. All the components and important characteristics will be defined more precisely in the next sections. Our method constructs a hierarchical clustering binary tree like the one depicted in Figure 3.1. Adopting a soft clustering approach, each \mathbf{s}_k at the leaf nodes refers to a vector of probabilities measuring how likely each example is to belong to a certain cluster k , and we refer to \mathbf{s}_k as the membership vector of cluster k . Hence, the number of clusters for a given tree construction is always equal to the number of leaf nodes. The tree is iteratively grown, starting from top to bottom. The mechanism by which the tree grows is encapsulated in each block referred to as split: a soft clustering operation that takes as input a node \mathbf{s}_k , previously a leaf node at that iteration, and divides its probabilities masses into two new leaf node probability vectors. Each of these probability vectors is related to subspaces of cluster k and whose sum equals \mathbf{s}_k . The term “split” does not imply some sort of hard assignment clustering, in which individual examples are thoroughly separated. Rather, we refer to a “soft split” done over each example’s probability mass given in \mathbf{s}_k . Since a split might be performed at any leaf node at a given iteration, each level of the binary tree is not necessarily even (*i.e.*, complete). We decide which \mathbf{s}_k to split next based on the total probability mass allocated to it. More precisely, we take the \mathbf{s}_k with the largest total mass, which roughly measures how big each clustering is, since more mass means more examples associated with it. Figure 3.2 presents a practical example with the result of our method’s complete hierarchical clustering tree constructed with the MNIST dataset, where the amount of probability mass per class in each \mathbf{s}_k is represented by the font size of each class in parenthesis. Notice that at each split of the tree shown in Figure

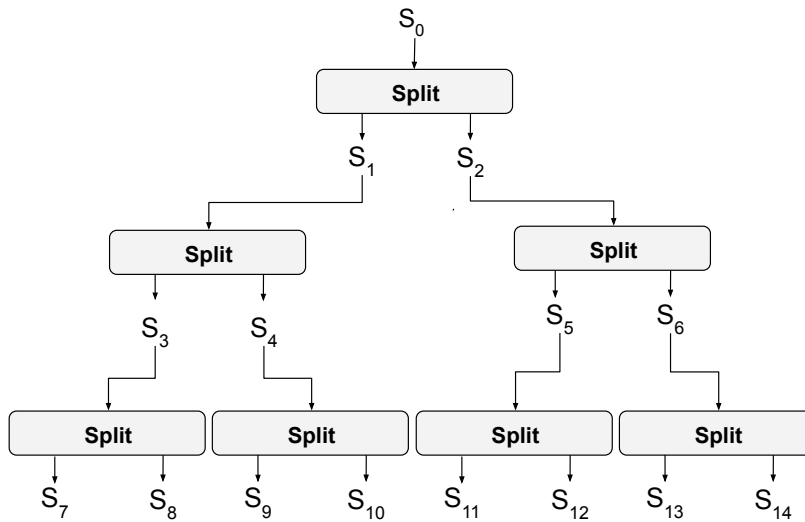


Figure 3.1. Simplified view of our method’s hierarchical clustering. Each cluster k is associated with a membership vector \mathbf{s}_k , which contains the probabilities of each i -th training example belonging to k . Each split block receives a \mathbf{s}_k as input and creates two new membership vectors from it, each associated with a subcluster of cluster k .

3.2 the classes mostly present at the parent \mathbf{s}_k are divided into two subdivisions. At the leaf nodes ($\mathbf{s}_6, \mathbf{s}_8, \mathbf{s}_9, \mathbf{s}_{12}, \mathbf{s}_{13}, \mathbf{s}_{14}, \mathbf{s}_{15}, \mathbf{s}_{16}, \mathbf{s}_{17}, \mathbf{s}_{18}$), we have the final clusterization. But some clusters might not be perfectly separated, like \mathbf{s}_{18} , for instance. Notice how \mathbf{s}_{18} is mostly associated with 9, but a big portion of the probability mass assigned to 9’s was lost during the 5-th split, when it ended up in \mathbf{s}_9 . We still arrive, however, at a clearly visible 1-to-1 association of each class with almost every other \mathbf{s}_k at a leaf node.

3.2 Basic Notation for Hierarchical Soft Clustering

In this section, we define the notation used in our work. Matrices or vector random variables are denoted by either Greek or non-italicized Roman letters, in uppercase, printed in a boldface font (*e.g.*, $\mathbf{\Delta}, \mathbf{D}$); vectors are denoted by either Greek or non-italicized Roman letters, in lowercase, printed in a boldface font (*e.g.*, $\mathbf{\delta}, \mathbf{d}$); sets are denoted by non-italicized Roman letters, in uppercase, printed in non-boldface font (*e.g.*, D); scalars and functions are denoted by either Greek or italicized Roman letters, either in uppercase or lowercase, printed in non-boldface font (*e.g.*, Δ, δ, D, d); scalar random variables are denoted by italicized Roman letters, in uppercase, printed in non-boldface font (*e.g.*, D); because of conventions widely employed in similar works,

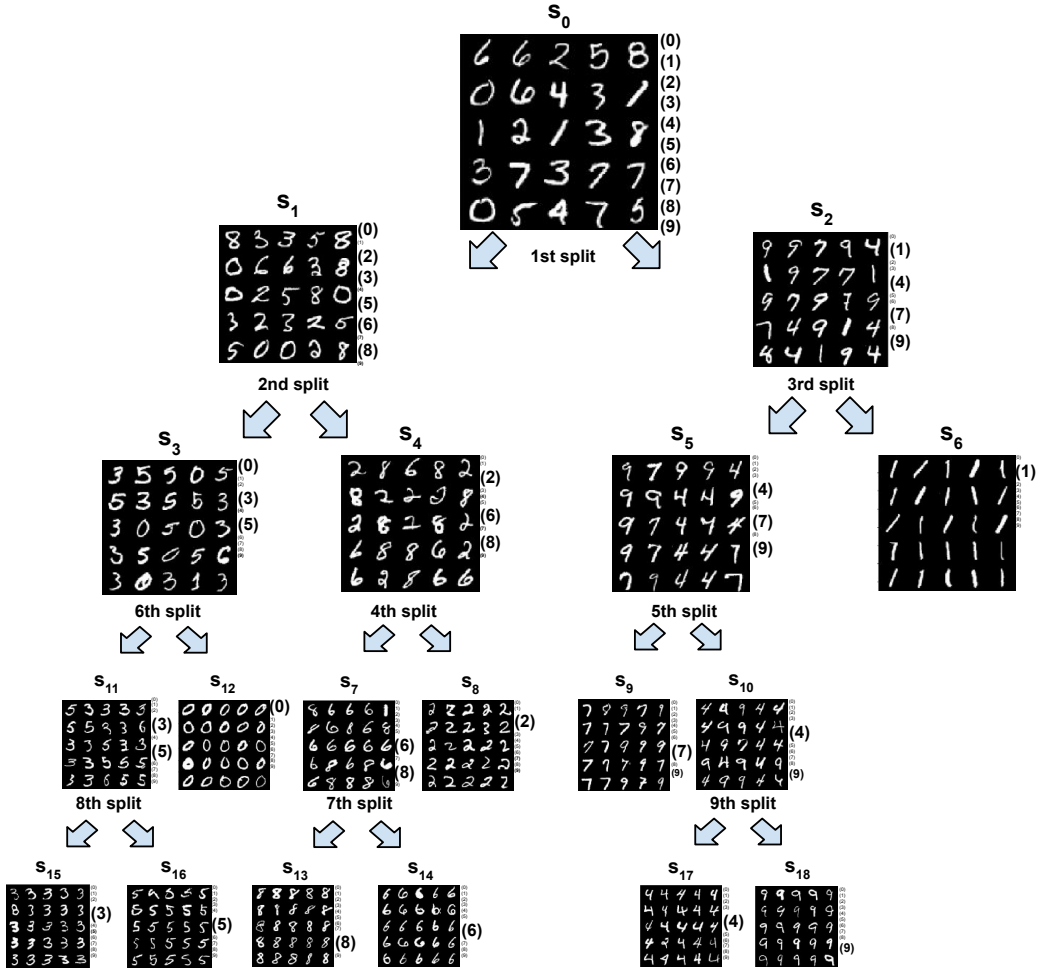


Figure 3.2. Complete hierarchical tree constructed over the MNIST dataset. Each vector \mathbf{s}_k is placed together with 25 real examples sampled from cluster k 's probability distribution. This distribution is formed by sampling an example i with weights given by the i -th probability in \mathbf{s}_k . To indicate the most prevalent classes in each \mathbf{s}_k , we represent in parenthesis, scaled by font size, the sum of the probability mass of all the examples belonging to the class inside a given \mathbf{s}_k . To be more precise, the probability mass of a class A inside \mathbf{s}_k is given by $\sum_i s_{k,i} \cdot \mathbf{1}(c_i = A)$, where $\mathbf{1}$ is an indicator function and c_i is the class of the i -th example. The scale linearly ranges from font size 2 (closer to 0% mass, 0 mass) to font size 7 (closer to 100% of mass, or approximately 7000 of mass for the approximately 7000 examples per class in MNIST).

we adopt some exceptions to the aforementioned rules, which are \mathcal{L} for loss functions, \mathbb{E} for expectation over a random variable, ∇ for gradients, \mathbb{R} for the set of all real numbers, \mathcal{P} for probability distributions, $\mathbf{1}$ for a vector of ones.

3.2.1 Notation for soft clustering

Given a collection of N training examples $X_{\text{Data}} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$, and a total of C clusters, a soft clustering model parameterized by θ defines a membership matrix: a matrix $\mathbf{M} \in \mathbb{R}^{N \times C}$, where each $m_{i,k} = p(Z_i = k \mid \mathbf{x}_i, \theta)$ measures the probability of example i belonging to cluster k given \mathbf{x}_i and our model's parameters θ . The hidden random variable Z_i is used to represent the association of the i -th example to each cluster k . Consequently, we also have that $\sum_{k=1}^C m_{i,k} = 1$, *i.e.*, the probabilities in each row sum up to 1.

3.2.2 Expanding the notation for hierarchical soft clustering

Our method iteratively grows a hierarchical tree of clusters, from top to bottom, **with each leaf node representing a cluster**, and each split operation over a leaf node creating two new nodes, equivalently splitting a previous cluster into two subdivisions. This means that the matrix \mathbf{M} must change, both in its dimensions and in its values, at each new split performed on a leaf node of the associated tree. It must remain a membership matrix, so the values, even though modified, must still be probabilities that sum up to 1 along each row. We use $\mathbf{M}^{(j)}$ to indicate the membership matrix after j split operations were performed. We use $\mathbf{m}_{:,k}^{(j)}$ to refer to the column vector formed by the k -th column of the matrix $\mathbf{M}^{(j)}$, and also $m_{i,k}^{(j)}$ to refer to its scalar probability at row i and column k . Additionally, we will employ $\mathbf{1}_N \in \mathbb{R}^{N \times 1}$ as an all-ones column vector with N rows.

3.2.3 Hierarchical clustering initial state: before the first split

We start the tree with $\mathbf{M}^{(0)} \in \mathbb{R}^{N \times 1}$, the initial 1-column membership matrix describing a single trivial cluster to which each example in the dataset has probability 1 of belonging, *i.e.*, $\mathbf{m}_{:,1}^{(0)} = \mathbf{1}_N$, while the corresponding tree has a single node. Performing the first split transforms the 1-column matrix $\mathbf{M}^{(0)}$ into the 2-column matrix $\mathbf{M}^{(1)} \in \mathbb{R}^{N \times 2}$ and connects 2 new leaf nodes to the previous 1-node tree, making the number of leaf nodes equal to 2. The two new columns $\mathbf{m}_{:,1}^{(1)}$ and $\mathbf{m}_{:,2}^{(1)}$ are such that $\mathbf{m}_{:,1}^{(0)} = \mathbf{m}_{:,1}^{(1)} + \mathbf{m}_{:,2}^{(1)}$. By following this condition, it is clear that $\mathbf{m}_{:,1}^{(1)} + \mathbf{m}_{:,2}^{(1)} = \mathbf{m}_{:,1}^{(0)} = \mathbf{1}_N$.

3.2.4 Generalization: after j splits

For the j -th split, we have $\mathbf{M}^{(j)} \in \mathbb{R}^{N \times (j+1)}$, and a tree consisting of $2j + 1$ nodes (each split generates 2 new nodes) and $j + 1$ leaf nodes (each split replaces the previous leaf node by 2 new leaf nodes, adding 1 to the current count of leaf nodes). In order to perform the $(j + 1)$ -th split and obtain $\mathbf{M}^{(j+1)} \in \mathbb{R}^{N \times (j+2)}$, we must choose a leaf node of the tree associated with $\mathbf{M}^{(j)}$, and hence, its corresponding column $\mathbf{m}_{:,c}^{(j)}$. After the choice is made, the $(j + 1)$ -th split takes $\mathbf{m}_{:,c}^{(j)}$ as input and substitutes it by two new columns $\mathbf{m}_{:,c}^{(j+1)}$ and $\mathbf{m}_{:,c+1}^{(j+1)}$ in $\mathbf{M}^{(j+1)}$, such that $\mathbf{m}_{:,c}^{(j)} = \mathbf{m}_{:,c}^{(j+1)} + \mathbf{m}_{:,c+1}^{(j+1)}$. Let us formulate an induction hypothesis and assume that the sum along $\mathbf{M}^{(j)}$'s rows equals 1, *i.e.*, $\sum_{k=1}^{j+1} \mathbf{m}_{:,k}^{(j)} = \mathbf{1}_N$. We can verify that the same will hold for $\mathbf{M}^{(j+1)}$, as is shown by the following Equation 3.1:

$$\begin{aligned}
 \sum_{k=1}^{j+2} \mathbf{m}_{:,k}^{(j+1)} &= \sum_{k=1}^{j+1} \mathbf{m}_{:,k}^{(j)} - \mathbf{m}_{:,c}^{(j)} + \mathbf{m}_{:,c}^{(j+1)} + \mathbf{m}_{:,c+1}^{(j+1)} \\
 &= \sum_{k=1}^{j+1} \mathbf{m}_{:,k}^{(j)} - \mathbf{m}_{:,c}^{(j)} + \mathbf{m}_{:,c}^{(j)} \\
 &= \sum_{k=1}^{j+1} \mathbf{m}_{:,k}^{(j)} \\
 &= \mathbf{1}_{1,N}.
 \end{aligned} \tag{3.1}$$

Because the condition $\mathbf{m}_{:,1}^{(1)} + \mathbf{m}_{:,2}^{(1)} = \mathbf{m}_{:,1}^{(0)} = \mathbf{1}_N$ holds at the first split, we have thus demonstrated with the induction hypothesis and Equation 3.1 that $\sum_{k=1}^{j+1} \mathbf{m}_{:,k}^{(j)} = \mathbf{1}_{1,N}$ will hold for any j .

3.2.5 Referring to membership vectors at the nodes of the tree

The membership matrix $\mathbf{M}^{(j)}$ notation presented thus far was useful for precisely describing the transitions between soft clustering states that occur for our hierarchical tree model when a split is performed and new leaf nodes are added. However, it has a shortcoming: a given $\mathbf{M}^{(j)}$ is not enough for describing the entire tree, but only its leaf nodes after j splits have been performed. In order to visualize the entire hierarchical structure, we need a notation for referring to the membership vectors created at each node of the tree, not only the membership vectors at the leaf nodes that are defining the current soft clustering state. So we employ \mathbf{s}_k for referring to the membership vector created at the k -th created node of the tree. For referring to the i -th example probability in \mathbf{s}_k , we use $s_{k,i}$. It must be clear that both $\mathbf{m}_{:,k}^{(j)}$ and \mathbf{s}_k refer to membership

vectors, even though not necessarily to the same specific membership vector. While $\mathbf{m}_{:,k}^{(j)}$ is used in reference to the columns of $\mathbf{M}^{(j)}$, \mathbf{s}_k is used in reference to the nodes of the trees. Since $\mathbf{M}^{(j)}$ describes the membership vectors associated only with the leaf nodes of the tree, for each column vector in $\mathbf{M}^{(j)}$ we have an equivalent \mathbf{s}_k at the tree, but there is at least one \mathbf{s}_k at a non-leaf node of the tree for which there is no counterpart $\mathbf{M}^{(j)}$. To be more precise: for each k column of $\mathbf{M}^{(j)}$ there is a node k' at the tree such that $\mathbf{s}_{k'} = \mathbf{m}_{:,k}^{(j)}$; but, for $j > 0$, there is at least 1 node k' at the tree for which there is no column k in $\mathbf{M}^{(j)}$ such that $\mathbf{s}_{k'} = \mathbf{m}_{:,k}^{(j)}$.

3.2.6 Describing a tree with an example

In order to clarify the use of the notation even further, we refer back to Figure 3.2 depicting a real example of the tree constructed for with the MNIST dataset, and describe the sequence of splits occurred during its growth:

- Before the first split, at $j = 0$, we have a single node and $\mathbf{m}_{:,1}^{(0)} = \mathbf{s}_0$ or, equivalently, $\mathbf{M}^{(0)} = \mathbf{s}_0 = \mathbf{1}_{1,N}$.
- After the 1st split, $j = 1$, we have $\mathbf{m}_{:,1}^{(1)} = \mathbf{s}_1$, $\mathbf{m}_{:,2}^{(1)} = \mathbf{s}_2$, or, equivalently, $\mathbf{M}^{(1)} = [\mathbf{s}_1 \ \mathbf{s}_2]$, with $\mathbf{s}_1 + \mathbf{s}_2 = \mathbf{s}_0 = \mathbf{1}_{1,N}$, where $[\]$ denotes a concatenation of column vectors.
- After the 2nd split, $j = 2$, we have $\mathbf{m}_{:,1}^{(2)} = \mathbf{s}_3$, $\mathbf{m}_{:,2}^{(2)} = \mathbf{s}_4$, $\mathbf{m}_{:,3}^{(2)} = \mathbf{s}_2$, or, equivalently, $\mathbf{M}^{(2)} = [\mathbf{s}_3 \ \mathbf{s}_4 \ \mathbf{s}_2]$, with $\mathbf{s}_3 + \mathbf{s}_4 + \mathbf{s}_2 = \mathbf{s}_1 + \mathbf{s}_2 = \mathbf{1}_{1,N}$.
- After the 3rd split, $j = 3$, we have $\mathbf{m}_{:,1}^{(3)} = \mathbf{s}_3$, $\mathbf{m}_{:,2}^{(3)} = \mathbf{s}_4$, $\mathbf{m}_{:,4}^{(3)} = \mathbf{s}_6$, or, equivalently, $\mathbf{M}^{(3)} = [\mathbf{s}_3 \ \mathbf{s}_4 \ \mathbf{s}_5 \ \mathbf{s}_6]$, with $\mathbf{s}_3 + \mathbf{s}_4 + \mathbf{s}_5 + \mathbf{s}_6 = \mathbf{s}_3 + \mathbf{s}_4 + \mathbf{s}_2 = \mathbf{1}_{1,N}$.
- And so on.

3.3 Performing the Split

In this section we detail the deep clustering components and procedures occurring inside the split blocks, which were shown as nodes of the hierarchical clustering tree in Figure 3.1 and were treated as black boxes in Sections 3.1 and 3.2. But firstly, we describe a split block in terms of sub-blocks and what transformations they perform on a \mathbf{s}_k membership vector in order to output the two subsequent membership vectors that are soft subdivisions of the probabilities in \mathbf{s}_k . These sub-blocks inside a split block are of two types: 1) **raw split sub-block** and 2) **refinement sub-block**. Figure 3.3

shows how these sub-blocks are connected inside a split block. We might also refer to the procedures occurring inside the raw split sub-block as the *raw split phase* and to the chain of procedures encompassing the refinements sub-blocks as the *refinement phase*. The following Subsections 3.3.1 and 3.3.2 will discuss in detail, respectively, the raw split and the refinement phases, presenting *how* the GAN/MGAN components inside each sub-block manage to perform the binary clustering operations.

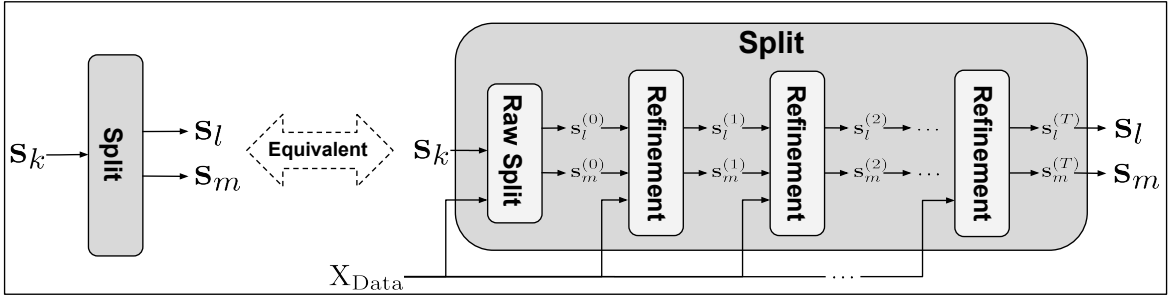


Figure 3.3. Raw split and refinement sub-blocks inside a split block. Each $\mathbf{s}_l^{(t)}$ and $\mathbf{s}_m^{(t)}$ constitute membership vectors estimates that are gradually transformed into the final estimates \mathbf{s}_l and \mathbf{s}_m .

We have already established in Section 3.2 that the split operation over cluster k takes as input the membership vector \mathbf{s}_k and returns two new membership vectors whose sum equals \mathbf{s}_k . Let us refer to these two vectors as \mathbf{s}_l and \mathbf{s}_m , with $\mathbf{s}_l + \mathbf{s}_m = \mathbf{s}_k$. Referring to Figure 3.3, we can see how, inside a split block, before being transformed into the final \mathbf{s}_l and \mathbf{s}_m , the vector \mathbf{s}_k firstly passes through a raw split sub-block, which outputs two initial membership vectors $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, that are subdivisions of \mathbf{s}_k (do not confuse this superscript notation, which refers to sequences of refinements performed on a membership vector inside a split block, with the superscript notation previously used for the membership matrix in $\mathbf{M}^{(j)}$, which referred to sequences of splits performed on the hierarchical tree). Sometimes the true classes with most probability mass in \mathbf{s}_k become well separated into soft clusters described by $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, and we could even use them as the final split block outputs \mathbf{s}_l and \mathbf{s}_m . But since $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$ are usually just rough estimates, we use the refinement sub-blocks to get them progressively closer to what we expect the ideal soft clustering assignment to be. Referring back to Figure 3.3, we can see how $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$ pass through a refinement sub-block, yielding two new membership vectors $\mathbf{s}_l^{(1)}$ and $\mathbf{s}_m^{(1)}$, which are also passed through a next refinement sub-block. The process goes on for T refinement sub-blocks, until we achieve our final result $\mathbf{s}_l^{(T)}$ and $\mathbf{s}_m^{(T)}$ that will be used for \mathbf{s}_l and \mathbf{s}_m . It must be noted that $\mathbf{s}_m^{(t)} + \mathbf{s}_l^{(t)} = \mathbf{s}_k$ for every t . This whole split procedure is also defined in Algorithm 3.1.

Algorithm 3.1 Split

```

input:  $X_{\text{Data}}, \mathbf{s}_k$ 
1 # RAW SPLIT PHASE
2  $\mathbf{s}_l^{(0)}, \mathbf{s}_m^{(0)} \leftarrow \text{raw\_split}(X_{\text{Data}}, \mathbf{s}_k)$ 
3 # REFINEMENT PHASE
4  $\mathbf{s}_l^{(t)}, \mathbf{s}_m^{(t)} \leftarrow \mathbf{s}_l^{(0)}, \mathbf{s}_m^{(0)}$ 
5 for  $T$  iterations do
6   |  $\mathbf{s}_l^{(t)}, \mathbf{s}_m^{(t)} \leftarrow \text{refinement}(X_{\text{Data}}, \mathbf{s}_l^{(t)}, \mathbf{s}_m^{(t)})$ 
7 end
8  $\mathbf{s}_l, \mathbf{s}_m \leftarrow \mathbf{s}_l^{(t)}, \mathbf{s}_m^{(t)}$ 
9 Return  $\mathbf{s}_l, \mathbf{s}_m$ 

```

In order to visualize how the refinement sub-blocks progressively improve the raw split results, we present an illustrative example with 25 MNIST samples in Figure 3.4. In the next subsections we will reuse this same example to explain how the raw split and refinement sub-blocks perform their respective operations with it. The samples are depicted inside the square grid under X_{Data} , and each of the probabilities of each example of the sample for a given \mathbf{s}_k are depicted below X_{Data} . For this \mathbf{s}_k in particular, 3's and 5's are the classes with most probability mass assigned to it. We would expect the final split result to be two membership vectors with one receiving all the 3's probability mass while the other vector receives all the 5's probability mass. Firstly, we pass \mathbf{s}_k through the raw split block, and we can observe that the probability mass of 3's and 5's are roughly divided between $\mathbf{s}_l^{(0)}$ (with more 3's mass than 5's) and $\mathbf{s}_m^{(0)}$ (with more 5's mass than 3's), with some examples that are harder to identify not receiving the expected probability in the membership vector mostly associated with other examples of its class. After we pass $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$ through the first refinement and obtain $\mathbf{s}_l^{(1)}$ and $\mathbf{s}_m^{(1)}$, we can observe more probability mass being assigned for 3's in $\mathbf{s}_l^{(1)}$ and more probability mass assigned for 5's in $\mathbf{s}_m^{(1)}$. After the second refinement, the probability mass of 3's and 5's are almost completely separated in $\mathbf{s}_l^{(2)}$ and $\mathbf{s}_m^{(2)}$.

In the next sections, we describe the raw split and refinement phases in detail.

3.3.1 Raw Split Phase

For the raw split phase, we use a MGAN architecture with two generators, which we adapt for binary clustering by taking advantage of the fact that each of its two generators learns to specialize in generating samples from one sub-region of the real data distribution, with this sub-region tending to correlate with a specific set of classes of the dataset.

Figure 3.5 depicts the training of this MGAN, where the MGAN components are the 2 generators $G\alpha_k, G\beta_k$, the discriminator D_k and the classifier C_k . We again

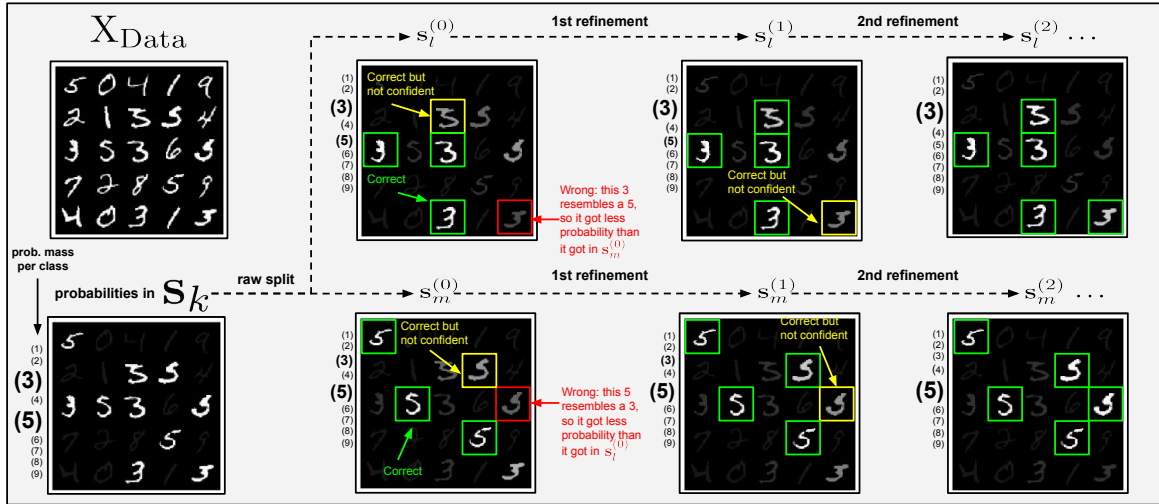


Figure 3.4. (Best viewed in color) Visualization with a MNIST example of a s_k , whose probabilities are mostly associated with 3's and 5's, passing through a raw split sub-block and then 2 refinement sub-blocks. The probabilities of each individual example in the sample is indicated by its intensity in gray scale, with more white indicating close to 100 % probability. Like we did for Figure 3.2, we once again use the font size scale, in parenthesis, alongside each block, to indicate the amount of probability mass per class assigned to it. Note that at each t , the probabilities sum up to s_k , *i.e.*, $s_m^{(t)} + s_l^{(t)} = s_k$.

reuse the same MNIST sample and s_k that was presented in Figure 3.4. We need each generator to specialize in sub-regions of s_k , so real data samples must reflect the proportions of probability mass allocated inside s_k . For the example shown in the figure, we would need samples that are mostly 3's and 5's. We perform this by sampling from a \mathcal{P}_{s_k} distribution defined by drawing weighted random samples from real examples in X_{Data} with probability proportional to their masses in s_k . More precisely, a \mathbf{x}_i in X_{Data} will be sampled with probability $\frac{s_{k,i}}{s_{k,1} + s_{k,2} + \dots + s_{k,N}}$. We then proceed to train the MGAN with these real samples, with the usual adversarial game between generators $G\alpha_k$, $G\beta_k$ and the discriminator D_k , while C_k is trained to distinguish between the generated samples. After some epochs of training, we observe that one generator is generating mostly 3's while the other is generating mostly 5's. A non-essential but helpful implementation detail is that we also train the generators to help in C_k 's classification loss, which increases the incentive for $G\alpha_k$ and $G\beta_k$ to generate samples from different sub-regions. This is non-essential because most of the times each generator naturally specializes in one sub-region. The reason for this specialization is that both generators tend to reach a local minimum in their adversarial game with the discriminator, in which it is simply not necessary for a generator to learn the

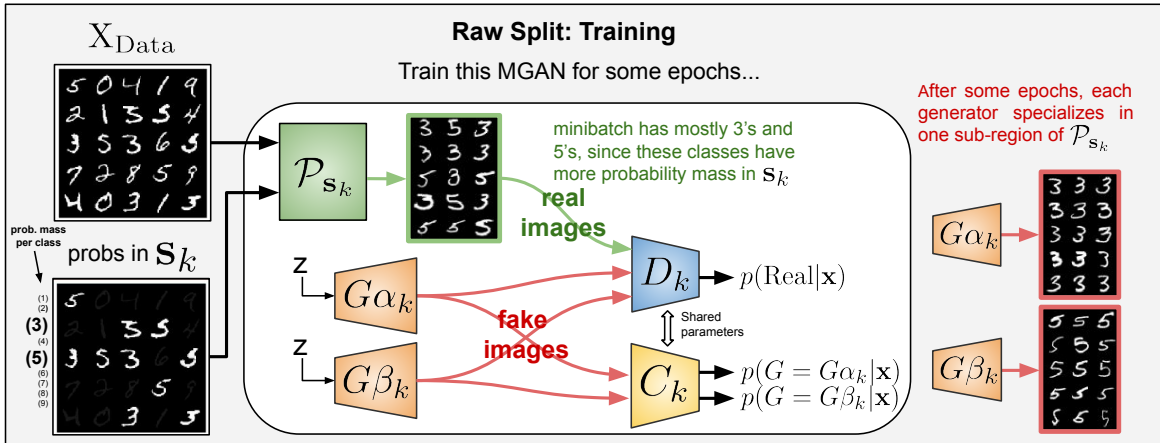


Figure 3.5. (Best viewed in color) Training the Raw Split Components. Two generators G_{α_k} , G_{β_k} , one discriminator D_k and a classifier C_k . \mathcal{P}_{S_k} draws samples weighted by the amount of probability each example has in S_k .

other generator’s sub-region, as long as the union of both generated sub-regions is enough to cover the entire data distribution, thus creating a sufficiently challenging situation for the discriminator (more precisely, in this situation the generators do not receive strong gradients encouraging them to expand their generation into other sub-regions). However, forcing the generators to also minimize the classifier’s loss increases the incentive for both generated distributions staying distinct, which is helpful for more complex data distributions, where the sub-regions aren’t so well defined, and it is harder to separate them with the adversarial game alone. Another more specific implementation detail is that we share some parameters between D_k and C_k , since it is more desirable for C_k to perform its classification in a higher-level feature space which is learned by D_k . This is a common practice in works involving MGANs, and we provide further explanation in the implementation details in Section 4.4.

After training the MGAN for enough epochs, we are able to perform the clustering, as depicted by Figure 3.6. We use C_k to perform the clustering on the real data according to the two generated distributions it learned to identify, and which we expect to correlate with different classes of the dataset. In this example, the first generated distribution resembled 3’s, while the other resembled 5’s. So we expect C_k to mostly assign real 3’s probability mass to the first soft cluster and real 5’s probability class to the second. As we can see in Figure 3.6, C_k manages, as expected, to roughly assign the 3’s probability mass to the cluster related to G_{α_k} ’s distribution (which generated mostly 3’s), and it also roughly assigns the 5’s probability mass to the cluster related with G_{β_k} ’s samples (which generated mostly 5’s). Note that C_k has to perform its in-

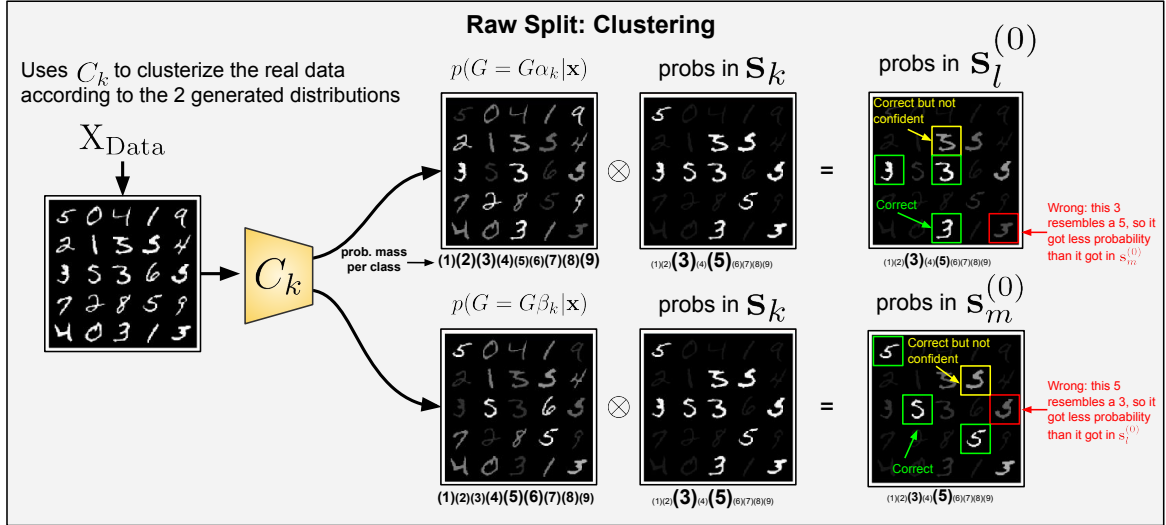


Figure 3.6. (Best viewed in color) Clustering the dataset with the raw split classifier.

ference for all examples of every class in the dataset, regardless if they were present in the generated distributions, with $p(G = G\alpha_k | \mathbf{x}) + p(G = G\beta_k | \mathbf{x}) = 1$. We do not care for the classifications of the examples with low probability in \mathbf{s}_k , since most of their probability mass has already been assigned to other clusters. To enforce the already established condition that the result of each example’s probabilities in the subsequent membership vectors parting from \mathbf{s}_k ’s node in the tree must sum up to their probability in \mathbf{s}_k , we multiply the probabilities of each example predicted by C_k by its probability in \mathbf{s}_k . The result is $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, which had already been shown in Figure 3.4, we’ve enforced that $\mathbf{s}_l^{(0)} + \mathbf{s}_m^{(0)} = \mathbf{s}_k$. Finally, by noting again that certain examples weren’t well separated in $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, we are able to conclude that the two generated distributions weren’t sufficiently diverse and/or high quality to account for the entire set of 3’s and 5’s the MGAN had access to, and this reflected in the classifier not learning a classification that was suited for identifying all 3’s and all 5’s with sufficient accuracy.

We now provide a more formal definition for the MGAN game occurring for the raw split phase. We can define the objective function of the two-generator MGAN game happening at a raw split of membership vector \mathbf{s}_k as an optimization of a sum of two cost functions \mathcal{L}_{adv} and \mathcal{L}_{cls} , described by Equation 3.2. \mathcal{L}_{adv} describes the cost function for the adversarial minimax game between generators and discriminator, and is given by Equation 3.3. \mathcal{L}_{cls} describes the classification cost that is minimized by both generators and classifier, and is described by Equation 3.4. Note that we multiply \mathcal{L}_{cls}

by a regularization parameter λ to weight its impact on the total cost.

$$\min_{\boldsymbol{\theta}_{G\alpha_k}, \boldsymbol{\theta}_{G\beta_k}, \boldsymbol{\theta}_{C_k}} \max_{\boldsymbol{\theta}_{D_k}} \mathcal{L}(G\alpha_k, G\beta_k, D_k, C_k) = \mathcal{L}_{adv}(G\alpha_k, G\beta_k, D_k) + \lambda \mathcal{L}_{cls}(G\alpha_k, G\beta_k, C_k) \quad (3.2)$$

$$\mathcal{L}_{adv}(G\alpha_k, G\beta_k, D_k) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{s_k}} [\log D_k(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G\alpha_k}} [\log(1 - D_k(\mathbf{x}))] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G\beta_k}} [\log(1 - D_k(\mathbf{x}))] \quad (3.3)$$

$$\mathcal{L}_{cls}(G\alpha_k, G\beta_k, C_k) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G\alpha_k}} [\log(C_k(\mathbf{x}))] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G\beta_k}} [\log(C_k(\mathbf{x}))] \quad (3.4)$$

The remainder of this subsection provides a detailed line-by-line description of Algorithm 3.2 to train the MGAN components in the raw split phase. This part is very specific, and the reader might skip it without hindering the overall comprehension of the rest of the method.

Initialization: For a raw split over cluster k , a mini-batch with n_S real examples is sampled from a probability distribution \mathcal{P}_{s_k} . Then, mini-batches with n_G generated samples are drawn from each of $G\alpha_k$ and $G\beta_k$ (lines 5-6). **Discriminator training:** The generated and real samples are used to compute the real and fake losses for D_k , both computed with binary cross-entropy using labels 1 for real and labels 0 for fake, having the gradients of their sum back-propagated to update D_k 's parameters $\boldsymbol{\theta}_{D_k}$ (lines 7-9). **Classifier training:** C_k 's loss is computed with fake samples from $G\alpha_k$ and $G\beta_k$, using cross-entropy with two classes and whatever labels we choose for differentiating between $G\alpha_k$ samples and $G\beta_k$ samples (we use the superscripts on $C_k^{(\alpha-out)}$ and $C_k^{(\beta-out)}$ as a notation for indexing the probability outputs for each class), having its gradients back-propagated to update C_k 's parameters $\boldsymbol{\theta}_{C_k}$ (lines 10-11). **Generators training:** $G\alpha_k$ and $G\beta_k$ are trained together to fool D_k with the same loss used for D_k 's training, but with the labels flipped (line 12); $G\alpha_k$ and $G\beta_k$ are also trained with a regularization classification loss computed with the same loss and labels used for C_k 's loss (line 13) to encourage the generation of samples that minimize it, which can only be achieved if $G\alpha_k$'s samples are distinguishable from $G\beta_k$'s and vice-versa (this classification regularization is important to help each generator specialize in different regions of the training set while trying to fool the discriminator, which is necessary for a well separated binary clustering at the split phase); and finally, both losses are summed, with a regularization λ multiplied to the classification loss to weight its impact on the gradients, which are then computed and back-propagated to update the

parameters of both generators, given by θ_{G_k} (line 14). **Split phase:** After training the MGAN components for sufficient iterations, we create two new membership vectors \mathbf{s}_l and \mathbf{s}_m initialized with zeros (line 18), each to be associated with subdivisions l and m of the cluster k with which \mathbf{s}_k was associated; we then iterate through each training example of the entire training set X_{Data} and use C_k to perform inference on it, *i.e.*, outputting two probabilities $C_k^{(\alpha-out)} = p(\alpha | \mathbf{x}_i)$ and $C_k^{(\beta-out)} = p(\beta | \mathbf{x}_i)$, where α and β symbolize the event that a given example came from $G\alpha_k$ (closer to one sub-region of cluster k) and $G\beta_k$ (closer to another sub-region of cluster k), respectively (lines 22-23); since C_k outputs probabilities ranging from 0 to 1 for each example, *i.e.*, the sum of its outputs $p(\alpha | \mathbf{x}_i)$ and $p(\beta | \mathbf{x}_i)$ is 1, so we must multiply them by i -th example probability in \mathbf{s}_k , so that the condition for $\mathbf{s}_l + \mathbf{s}_m = \mathbf{s}_k$ is satisfied (also at lines 22-23); and the algorithm finally returns the newly created vectors \mathbf{s}_l and \mathbf{s}_m (line 25).

Algorithm 3.2 Raw Split

input: $X_{\text{Data}}, \mathbf{s}_k$

- 1 Creates Components $G\alpha_k, G\beta_k, C_k, D_k$
- 2 **#TRAINING PHASE**
- 3 #trains components for a given number of iterations
- 4 **for** *training_ iterations* **do**
- 5 $\mathbf{x}_s \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{\mathbf{s}_k}\}_{i=1}^{n_s}$
- 6 $\mathbf{x}_{G\alpha} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G\alpha_k}\}_{i=1}^{n_G}$
- 7 $\mathbf{x}_{G\beta} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G\beta_k}\}_{i=1}^{n_G}$
- 8 $\mathcal{L}_{D_k}^{(real)} \leftarrow -\frac{1}{n_s} \sum_{i=1}^{n_s} \log(D_k(\mathbf{x}_s^{(i)}))$
- 9 $\mathcal{L}_{D_k}^{(fake)} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log(1 - D_k(\mathbf{x}_{G\alpha}^{(i)})) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log(1 - D_k(\mathbf{x}_{G\beta}^{(i)}))$
- 10 Updates θ_{D_k} with Adam and $\nabla_{\theta_{D_k}}(\mathcal{L}_{D_k}^{(real)} + \mathcal{L}_{D_k}^{(fake)})$
- 11 $\mathcal{L}_{C_k} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log(C_k^{(\alpha-out)}(\mathbf{x}_{G\alpha}^{(i)})) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log(C_k^{(\beta-out)}(\mathbf{x}_{G\beta}^{(i)}))$
- 12 Updates θ_{C_k} with Adam and $\nabla_{\theta_{C_k}}(\mathcal{L}_{C_k})$
- 13 $\mathcal{L}_{G_k}^{(disc)} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log(D_k(\mathbf{x}_{G\alpha}^{(i)})) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log(D_k(\mathbf{x}_{G\beta}^{(i)}))$
- 14 $\mathcal{L}_{G_k}^{(clasf)} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log(C_k^{(\alpha-out)}(\mathbf{x}_{G\alpha}^{(i)})) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log(C_k^{(\beta-out)}(\mathbf{x}_{G\beta}^{(i)}))$
- 15 Updates θ_{G_k} with Adam and $\nabla_{\theta_{G_k}}(\mathcal{L}_{G_k}^{(disc)} + \lambda \mathcal{L}_{G_k}^{(clasf)})$
- 16 **end**
- 17 **#CLUSTERING PHASE**
- 18 #creates new vectors \mathbf{s}_l and \mathbf{s}_m initialized with zeros
- 19 $\mathbf{s}_l = \mathbf{s}_m = [0, 0, \dots, 0]$
- 20 #uses C_k to estimate the i -th example membership probability and update its value in \mathbf{s}_l and \mathbf{s}_m
- 21 #also multiplies the i -th example's prob by its prob in \mathbf{s}_k to enforce that $\mathbf{s}_l + \mathbf{s}_m = \mathbf{s}_k$
- 22 **for** \mathbf{x}_i in X_{Data} **do**
- 23 $s_{l,i} = C_k^{(\alpha-out)}(\mathbf{x}_i) \cdot s_{k,i}$
- 24 $s_{m,i} = C_k^{(\beta-out)}(\mathbf{x}_i) \cdot s_{k,i}$
- 25 **end**
- 26 Return $\mathbf{s}_l, \mathbf{s}_m$

3.3.2 Refinement Phase

After we have performed the raw split over membership vector \mathbf{s}_k , its probability mass was redistributed into two new membership vectors $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, which shall now pass through the first refinement sub-block and be transformed into new membership vectors $\mathbf{s}_l^{(1)}$ and $\mathbf{s}_m^{(1)}$ with improvements in clustering quality, such as was described by Figure 3.4. We proceed now to show how the refinement sub-block performs this first transformation, whose results are easy to generalize for the subsequent refinement sub-blocks.

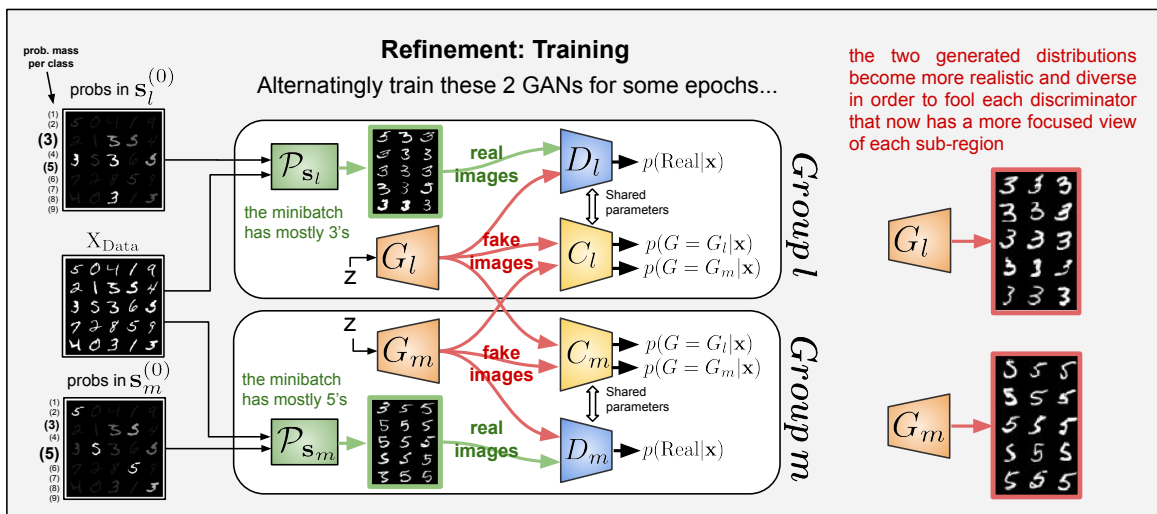


Figure 3.7. (Best viewed in color) Training the Refinement Components.

The components of the first refinement sub-block are depicted by Figure 3.7, with the same $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$ used as example in Figures 3.4 and 3.6. The components are divided in two groups l (formed by a generator G_l , a discriminator D_l and a classifier C_l) and m (formed by a generator G_m , a discriminator D_m and a classifier C_m), which we might also refer to as *refinement groups* l and m . Group l takes $\mathbf{s}_l^{(0)}$ as input, and group m takes $\mathbf{s}_m^{(0)}$. Group l has its own independent GAN game occurring between G_l and D_l , and group m has another separate game occurring between G_m and D_m (the role of classifiers C_l and C_m will become clearer shortly). This scheme with two separated GANs is designed to obtain a more focused generative representation of each sub-region of \mathbf{s}_k , with each sub-region described by $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$, than we were able to obtain at the raw split phase, with a single MGAN's discriminator having to learn to discriminate the entire region described \mathbf{s}_k . *By providing a more focused view of one sub-region to one discriminator, it encounters less variance among the real examples it receives,*

and thus its discriminative task becomes easier. We expect its adversarial generator’s response to be a more diverse and convincing generation of examples associated with that particular sub-region. Referring back to Figure 3.7, each GAN in groups l and m has its real samples drawn from, respectively, distributions $\mathcal{P}_{\mathbf{s}_l}$ and $\mathcal{P}_{\mathbf{s}_m}$, which are formed like $\mathcal{P}_{\mathbf{s}_k}$ was formed during the raw split, *i.e.*, by weighted random sampling of real examples from X_{Data} with weights given by each example’s probability mass in $\mathbf{s}_l^{(0)}$ and $\mathbf{s}_m^{(0)}$. As a result we can see how the minibatches that the GANs in each group receive reflect the probability mass allocated to its group’s respective membership vector, *e.g.*, $\mathcal{P}_{\mathbf{s}_l}$ draws mostly 3’s, since 3’s have more probability mass in $\mathbf{s}_l^{(0)}$, but it might eventually draw some 5’s as well, since there’s still some mass for 5’s in $\mathbf{s}_l^{(0)}$.

The role of classifiers C_l and C_m in these two separated GAN games is similar to the single classifier C_k used during the raw split phase: learn to differentiate between generated samples from G_l and G_m , thus providing a way to clusterize the real data afterwards. The reason for using two separated classifiers trained for the same task (instead of a simpler formulation with a single classifier like during the raw split), is that this permits each classifier to share its parameters with each discriminator, and consequently, as mentioned during the raw split explanation, it forces the classification to occur in a higher-level feature space, which is more desirable (once again, we provide further explanations in Section 4.4). Another practical detail for using the classifiers here, and slightly harder to visualize, is that we can also train G_l and G_m to minimize the classifiers’ losses, which in turn increases the incentive for each generator generating samples more characteristically associated with each sub-region (we did something similar for the 2 generators in the MGAN of the raw split).

After alternatingly training the 2 separate refinement groups for enough epochs, we are able to perform a clustering re-estimation, as depicted by Figure 3.8. This is very similar to how C_k was used to clusterize the real data in the raw split procedure, as previously described by Figure 3.6. Since we trained two classifiers, we take the average between the probabilities that C_l and C_m estimate for the same example in the dataset. Once again, each classifier was trained to distinguish between two generated distributions, which we expect to correlate well with certain classes. Only this time we expect an increase in the quality of the clustering, since these two generated distributions are now assumed to be more informative of each sub-region of the dataset than the two generated distributions obtained during the raw split. In this example, G_l ’s distributions resembled 3’s, while G_m ’s resembled 5’s, so we expect the classifiers to assign more of the 3’s probability mass to the cluster inferred to G_l and more of the 5’s to the cluster inferred to be G_m . The inferred probability results sum up to 1, *i.e.*, $p(G = G_l|\mathbf{x}) + p(G = G_m|\mathbf{x}) = 1$. Thus, the same way we did during the raw

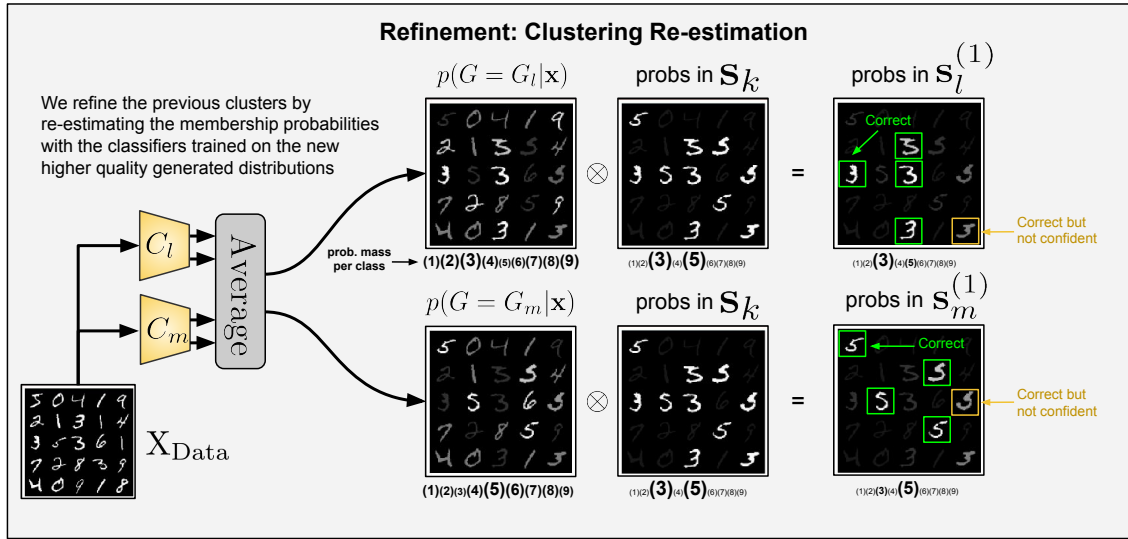


Figure 3.8. (Best viewed in color) Clustering the dataset with the refinement classifiers

split clusterization, we multiply the classification probabilities by the probabilities in s_k , enforcing the two new membership vectors $s_l^{(1)}$ and $s_m^{(1)}$ sum up to s_k . s_k can be obtained by simply summing the membership vector that a refinement block receives as input, *i.e.*, $s_k = s_l^{(0)} + s_m^{(0)} = s_l^{(t)} + s_m^{(t)}$ for every t -th refinement. The new clustering result yielding new membership vectors $s_l^{(1)}$ and $s_m^{(1)}$ is merely a reproduction of what was previously shown in Figure 3.4, which was already verified to be an improvement over $s_l^{(0)}$ and $s_m^{(0)}$ in regard to the reference classes.

For the consecutive refinement, we might expect that providing $s_l^{(1)}$ and $s_m^{(1)}$ to train newly created refinement groups l and m might yield generated distributions that constitute an even more defined representation of the sub-regions represented by $s_l^{(1)}$ and $s_m^{(1)}$, thus providing even more information for the classifiers to perform their clustering and obtain improved membership vectors $s_l^{(2)}$ and $s_m^{(2)}$, such as was previously shown in Figure 3.4. Hence, by repeating the process for T refinements, we hope that the generated distributions and subsequent clustering re-estimation tend to become increasingly more characteristically associated with the initially obtained sub-regions.

We now provide a more formal definition for the two simultaneous GAN games occurring for the training of the components of the refinement phase. From the perspective of refinement group l , the training can be defined as an optimization of a sum of two cost functions \mathcal{L}_{adv} and \mathcal{L}_{cls} , described by Equation 3.5. \mathcal{L}_{adv} describes the cost function for the adversarial minimax game between generator G_l and discriminator D_l , that only involves group l components, and is given by Equation 3.6. \mathcal{L}_{cls} describes the

classification cost that is minimized in respect to G_l 's parameters and C_l 's parameters, but also involves G_m and C_m for computing the cost, and is described by Equation 3.7. Note that we multiply \mathcal{L}_{cls} by a regularization parameter λ to weight its impact on the total cost. Equations 3.6 and 3.7 can also be analogously formulated for the training from the point of view of refinement group m , which is simultaneously optimized, by simply inverting m and l components.

$$\min_{\theta_{G_l}, \theta_{C_l}} \max_{\theta_{D_l}} \mathcal{L}(G_l, D_l, C_l) = \mathcal{L}_{adv}(G_l, D_l) + \lambda \mathcal{L}_{cls}(G_l, C_l) \quad (3.5)$$

$$\mathcal{L}_{adv}(G_l, D_l) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{data}}[\log D_l(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G_l}}[\log(1 - D_l(\mathbf{x}))] \quad (3.6)$$

$$\mathcal{L}_{cls}(G_l, C_l) = \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G_l}}[\log C_l(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G_m}}[\log C_m(\mathbf{x})] + \mathbb{E}_{\mathbf{x} \sim \mathcal{P}_{G_m}}[\log C_l(\mathbf{x})] \quad (3.7)$$

The remainder of this subsection provides a detailed line-by-line description of Algorithm 3.3, which implements an external training loop responsible for coordinating the alternating training of refinement groups l and m , and Algorithm 3.4, which is called as function by Algorithm 3.3 and trains the components of a given refinement group isolatedly with a single update iteration. This part is very specific and relatively dense, and the reader might skip it without hindering the overall comprehension of the rest of the work.

Initialization on Algorithm 3.3: For the refinement of groups l and m at iteration t of the refinement procedure, each group receives membership vectors $\mathbf{s}_l^{(t)}$ and $\mathbf{s}_m^{(t)}$, as well as newly created generators, discriminators and classifiers for each group. Firstly, we draw mini-batches with n_S real examples \mathbf{x}_{s_l} and \mathbf{x}_{s_m} , each respectively sampled from probability distributions \mathcal{P}_{s_l} and \mathcal{P}_{s_m} (lines 3-4 of Algorithm 3.3), with \mathcal{P}_{s_l} and \mathcal{P}_{s_m} being created exactly like \mathcal{P}_{s_k} was created for the raw split procedure: sampling from the training set X_{Data} with weights given by the probabilities in \mathbf{s}_l and \mathbf{s}_m . Then, mini-batches \mathbf{x}_{G_l} and \mathbf{x}_{G_m} with n_G generated samples are drawn from G_m and G_l (lines 5-6 of Algorithm 3.3). **First refinement group training call on Algorithm 3.3:** To train the components of a refinement group, we call Algorithm 3.4, which takes as arguments a set RG_{int} for the internal components/data of the current refinement group being trained at that given call, and a set RG_{ext} with some of the external components/data of its neighbor's refinement group that will also be needed for this training; for the first group training call, we train group l 's components with C_m and \mathbf{x}_{G_m} as additional necessary components/data from refinement group m (line

8 of Algorithm 3.3). **Discriminator training on Algorithm 3.4:** The generated samples $\mathbf{x}_{G_{int}} = \mathbf{x}_{G_l}$ and real samples $\mathbf{x}_{s_{int}} = \mathbf{x}_{s_l}$ are used to compute the fake and real losses for $D_{int} = D_l$, both computed with binary cross-entropy using labels 1 for real and labels 0 for fake, having the gradients of their sum back-propagated to update the parameters of $D_{int} = D_l$ (lines 2 to 4 of Algorithm 3.4). **Classifier training on Algorithm 3.4:** the loss of $C_{int} = C_l$ is computed with fake samples $\mathbf{x}_{G_{int}} = \mathbf{x}_{G_l}$ and $\mathbf{x}_{G_{int}} = \mathbf{x}_{G_m}$ (first use of external data), using cross-entropy with two classes and whatever labels we choose for differentiating between internal and external samples (we use the superscripts on $C_{int}^{(int-out)}$ and $C_{int}^{(ext-out)}$ as a notation for indexing the probability outputs for each class), having its gradients back-propagated to update the parameters of $C_{int} = C_l$ (lines 5-6 of Algorithm 3.4). **Generator training on Algorithm 3.4:** $G_{int} = G_l$ is trained to fool $D_{int} = D_l$ with the same loss used for the training of $D_{int} = D_l$, but with the labels flipped (line 7 of Algorithm 3.4); like during the raw split training, $G_{int} = G_l$ is also trained with a regularization classification loss (line 8 of Algorithm 3.4) computed with the same loss and labels used for the loss of $C_{int} = C_l$, but now it is trained not only to aid in the classification of $C_{int} = C_l$, but also in the classification of $C_{ext} = C_m$ (we do this because both classifiers are trained with the generated samples from both groups, *i.e.*, C_m will be trained with the fake samples from G_l when we are training refinement group m); and finally, both losses are summed, with a regularization λ multiplied to the classification loss to weight its impact on the gradients, which are then computed and back-propagated to update the parameters of $C_{int} = C_l$, given by $\theta_{G_{int}}$ (line 9 of Algorithm 3.4). **Second refinement group training call on Algorithm 3.3:** now that the components of l were updated in the last training call, we draw new mini-batches with generated samples (lines 9 and 10 of Algorithm 3.3) \mathbf{x}_{G_l} (since G_l has just been updated) and \mathbf{x}_{G_m} (because C_l has just been trained with the previous sample of \mathbf{x}_{G_m} , and a new \mathbf{x}_{G_m} sample will help with better gradients for G_m minimizing C_l 's loss), and then proceed to train the components of refinement group m by providing the data/components of m inside the RG_{int} set of arguments of Algorithm 3.4 and the needed data/components of l inside the RG_{ext} set of arguments (line 12 of Algorithm 3.3); the entire refinement training occurs for m just like it was described for refinement group l . **Refinement phase on Algorithm 3.3:** After training refinement groups l and m components for sufficient iterations, we create two new membership vectors $\mathbf{s}_l^{(t+1)}$ and $\mathbf{s}_m^{(t+1)}$ (line 16), each to receive newly estimated and hopefully refined membership probabilities; similarly to the raw split procedure, we iterate through each training example of entire training set X_{Data} , but now we use the two classifiers C_l and C_m to perform inference on the real data by taking the average of the estimated probabilities that both output for each

training example. More precisely, the new membership probability $s_{i,l}^{(t+1)}$ for the i -th example on vector $\mathbf{s}_l^{(t+1)}$ is the average between the probability that C_l predicts for it being closer to G_l 's samples (indexed by $C_l^{(l-out)}$) and the correspondent prediction by C_m (indexed by $C_m^{(l-out)}$), and we also do the analogous for $s_{i,m}^{(t+1)}$ (lines 19-20); because this averaged result of two probabilities ranges between 0 and 1, if we multiply it by $s_{k,i} = s_{l,i}^{(t)} + s_{m,i}^{(t)}$, like we did for vectors \mathbf{s}_l and \mathbf{s}_m at the end of the raw split, the condition for $s_{l,i}^{(t+1)} + s_{m,i}^{(t+1)} = s_{k,i} \mathbf{s}_l^{(t+1)} + \mathbf{s}_m^{(t+1)} = \mathbf{s}_k$ is automatically satisfied (the total probability mass after raw split on cluster k is preserved); and we finally update each i -the example membership probabilities (lines 22-23).

Algorithm 3.3 Refinement

input: $X_{\text{Data}}, \mathbf{s}_l^{(t)}, \mathbf{s}_m^{(t)}$

- 1 Creates Components G_l, D_l, C_l for group l and G_m, D_m, C_m for group m
- 2 **#TRAINING PHASE**
- 3 **for iterations do**
- 4 $\mathbf{x}_{\mathbf{s}_l} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{\mathbf{s}_l^{(t)}}\}_{i=1}^{n_S}$
- 5 $\mathbf{x}_{\mathbf{s}_m} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{\mathbf{s}_m^{(t)}}\}_{i=1}^{n_S}$
- 6 $\mathbf{x}_{G_l} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G_l}\}_{i=1}^{n_G}$
- 7 $\mathbf{x}_{G_m} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G_m}\}_{i=1}^{n_G}$
- 8 #trains group l with some necessary external data/components from group m
- 9 TrainRefinementGroup($\text{RG}_{int} = \{G_l, D_l, C_l, \mathbf{x}_{\mathbf{s}_l}, \mathbf{x}_{G_l}\}, \text{RG}_{ext} = \{C_m, \mathbf{x}_{G_m}\}$)
- 10 $\mathbf{x}_{G_l} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G_l}\}_{i=1}^{n_G}$
- 11 $\mathbf{x}_{G_m} \leftarrow \{\mathbf{x}_i \sim \mathcal{P}_{G_m}\}_{i=1}^{n_G}$
- 12 # trains group m with some necessary external data/components from group l
- 13 TrainRefinementGroup($\text{RG}_{int} = \{G_m, D_m, C_m, \mathbf{x}_{\mathbf{s}_m}, \mathbf{x}_{G_m}\}, \text{RG}_{ext} = \{C_l, \mathbf{x}_{G_l}\}$)
- 14 **end**
- 15 **#CLUSTERING RE-ESTIMATION PHASE**
- 16 #creates new vectors $\mathbf{s}_l^{(t+1)}$ and $\mathbf{s}_m^{(t+1)}$ initialized with zeros
- 17 $\mathbf{s}_l^{(t+1)} = \mathbf{s}_m^{(t+1)} = [0, 0, \dots, 0]$
- 18 #averages C_l and C_m predictions to estimate the i -th example prob for $\mathbf{s}_l^{(t+1)}$ and $\mathbf{s}_m^{(t+1)}$
- 19 #also multiplies this prob by the i -th prob in \mathbf{s}_k to enforce that $\mathbf{s}_l^{(t+1)} + \mathbf{s}_m^{(t+1)} = \mathbf{s}_k$
- 20 #then, updates i -th prob in $\mathbf{s}_l^{(t+1)}$ and $\mathbf{s}_m^{(t+1)}$
- 21 **for \mathbf{x}_i in X_{Data} do**
- 22 $s_{i,k} = s_{i,l}^{(t)} + s_{i,m}^{(t)}$
- 23 $s_{i,l}^{(t+1)} = \frac{1}{2} \cdot \left(C_l^{(l-out)}(\mathbf{x}_i) + C_m^{(l-out)}(\mathbf{x}_i) \right) \cdot s_{i,k}$
- 24 $s_{i,m}^{(t+1)} = \frac{1}{2} \cdot \left(C_l^{(m-out)}(\mathbf{x}_i) + C_m^{(m-out)}(\mathbf{x}_i) \right) \cdot s_{i,k}$
- 25 **end**
- 26 Return $\mathbf{s}_l^{(t+1)}, \mathbf{s}_m^{(t+1)}$

Algorithm 3.4 TrainRefinementGroup

input: $\text{RG}_{int} = \{G_{int}, D_{int}, C_{int}, \mathbf{x}_{s_{int}}, \mathbf{x}_{G_{int}}\}, \text{RG}_{ext} = \{C_{ext}, \mathbf{x}_{G_{ext}}\}$

- 1 $\# \text{RG}_{int}$ receives internal data/components from the current refinement group being trained, RG_{ext} receives external data/components from the neighbor's refinement group that are needed to train the current group
 - 2 $\mathcal{L}_{D_{int}}^{(real)} \leftarrow -\frac{1}{n_S} \sum_{i=1}^{n_S} \log \left(D_{int}(\mathbf{x}_{s_{int}}^{(i)}) \right)$
 - 3 $\mathcal{L}_{D_{int}}^{(fake)} \leftarrow -\frac{1}{n_G} \sum_{i=1}^{n_G} \log \left(1 - D_{int}(\mathbf{x}_{G_{int}}^{(i)}) \right)$
 - 4 Updates $\theta_{D_{int}}$ with Adam and $\nabla_{\theta_{D_{int}}} (\mathcal{L}_{D_{int}}^{(real)} + \mathcal{L}_{D_{int}}^{(fake)})$
 - 5 $\mathcal{L}_{C_{int}} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log \left(C_{int}^{(int-out)}(\mathbf{x}_{G_{int}}^{(i)}) \right) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log \left(C_{int}^{(ext-out)}(\mathbf{x}_{G_{ext}}^{(i)}) \right)$
 - 6 Updates $\theta_{C_{int}}$ with Adam and $\nabla_{\theta_{C_{int}}} (\mathcal{L}_{C_{int}})$
 - 7 $\mathcal{L}_{G_{int}}^{(disc)} \leftarrow -\frac{1}{n_G} \sum_{i=1}^{n_G} \log \left(D_{int}(\mathbf{x}_{G_{int}}^{(i)}) \right)$
 - 8 $\mathcal{L}_{G_{int}}^{(clasf)} \leftarrow -\frac{1}{2n_G} \sum_{i=1}^{n_G} \log \left(C_{int}^{(int-out)}(\mathbf{x}_{G_{int}}^{(i)}) \right) - \frac{1}{2n_G} \sum_{i=1}^{n_G} \log \left(C_{ext}^{(ext-out)}(\mathbf{x}_{G_{int}}^{(i)}) \right)$
 - 9 Updates $\theta_{G_{int}}$ with Adam and $\nabla_{\theta_{G_{int}}} (\mathcal{L}_{G_{int}}^{(disc)} + \lambda \mathcal{L}_{G_{int}}^{(clasf)})$
-

Chapter 4

Experiments

4.1 Datasets

We performed the experiments with 3 datasets: MNIST, Fashion MNIST and CIFAR-10. For all of these datasets, we use images from both the training set and the test set to perform our clustering method (this is a common practice in other deep clustering works).

4.1.1 MNIST

This dataset consists of black-and-white images of hand-written digits with 28x28 resolution. The classes are equivalent to the digit written in each image, ranging from 0 to 9. There are approximately 6000 images for each class in the training set and approximately 1000 images for each class in the test set. In Figure 4.1 we show some samples from this dataset.



Figure 4.1. Random samples from the MNIST dataset.

This is a basic dataset in deep learning tasks related with images. Because of

its simplicity and clearly distinct classes of images, modern image recognition methods usually obtain accuracy close to 99% in this dataset. It is also not a very challenging dataset for unsupervised clustering tasks, with modern deep clustering methods usually scoring above 90% in clustering accuracy.

4.1.2 Fashion MNIST (FMNIST)

This dataset consists of black-and-white pictures of clothing-related objects with 28x28 resolution. The classes of this dataset are: t-shirt, pants, coat, pullover, sandals, sneakers, ankle boot, bag, shirt, dress. There are 6000 images for each class in the training set and 1000 images for each class in the test set. In Figure 4.2 we show some samples from this dataset.

In this dataset, one of the relevant characteristics for unsupervised clustering task is the difficulty of differentiating, even for a human, between certain examples of certain classes, like coats and pullovers, for instance. This becomes even more challenging without the information provided by the labels.



Figure 4.2. Random samples from the Fashion MNIST dataset.

4.1.3 CIFAR-10

This dataset consists of colored images of varied objects and animals with 32x32 resolution. The classes of this dataset are: plane, car, bird, cat, deer, dog, frog, horse, ship and truck. There are 5000 images of each class for the training set, and 1000 images for each class in the test set. In Figure 4.3 we show some samples from this dataset.

This dataset is even more challenging than the Fashion MNIST for deep learning tasks, especially for unsupervised image generation and clusterization. This difficulty is caused by the excessive variance occurring among examples from the same class. This

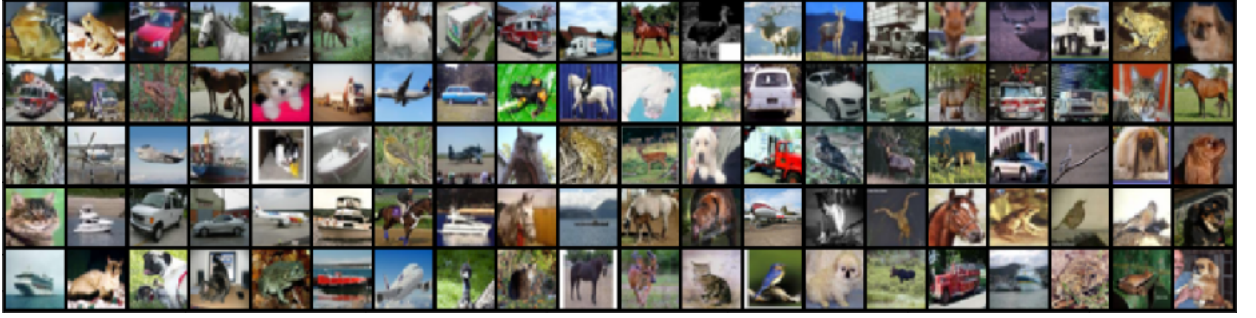


Figure 4.3. Random samples from the CIFAR-10.

dataset is very interesting for clustering because of the huge room for performance improvement that it provides, considering the latest clustering benchmarks that perform experiments with it.

4.2 Evaluation Metrics

We have used two of the most common clustering metrics for evaluating our method’s clustering performance on each dataset: *clustering accuracy* and *normalized mutual information*.

Clustering accuracy (ACC) is described as

$$\text{ACC}\% = \max_M \frac{\sum_{i=1}^N \mathbf{1}(y_i = M(c_i))}{N} \times 100\%, \quad (4.1)$$

where $\mathbf{1}$ represents an indicator function, y_i refers to the *cluster* assigned to the i -th element, c_i refers to the i -th element reference class, and M represents any possible 1-to-1 mapping between the set of reference classes and the set of clusters. The optimum mapping M can be computed by the well known Hungarian Algorithm [Kuhn, 1955]. Since we employ a soft clusterization method, we choose the cluster y_i based on which cluster the i -th example has the highest probability of belonging to, *i.e.*, for the final membership matrix $\mathbf{M}^{(j)}$ (this notation was presented in Section 3.2), obtained after j splits in the hierarchical tree, and hence with $j + 1$ columns (clusters), we select a column k such that $m_{i,k} \geq m_{i,l}$ for every l in $1 \dots j + 1$.

Normalized Mutual Information (NMI) is described as

$$\text{NMI} = \frac{I(C; Y)}{\max\{H(Y), H(C)\}}, \quad (4.2)$$

where $I(C; Y)$ corresponds to the mutual information between the set of reference

classes C and the obtained set of clusters Y , and $H(\cdot)$ refers to the entropy of each set (the entropy for both sets can be straightforwardly computed with the probabilities of a randomly selected example in the training set belonging to one of the 10 reference classes or one of the 10 obtained clusters). The normalization is limited to the interval 0 and 1, with 0 meaning no correlation between C and Y , and 1 meaning perfect correspondence in a 1-to-1 mapping. For defining the set of clusters Y , we choose y_i the same way it was described for computing the ACC.

4.3 Main Results

4.3.1 Baselines and state-of-the-art methods

We now present the baselines and state-of-the-art methods to which we shall compare our results. Results are reproduced from [Zhao et al., 2020]. We use some baselines not based on deep learning: K-means [MacQueen, 1967], SC [Zelnik-Manor and Perona, 2004], AC [Gowda and Krishna, 1978], NMF [Cai et al., 2009]. Deep learning based methods are DEC [Xie et al., 2016], JULE [Yang et al., 2016], VaDE [Jiang et al., 2017], DEPICT [Ghasedi Dizaji et al., 2017], SpectralNET [Shaham et al., 2018], ClusterGAN [Mukherjee et al., 2019], DLS-Clustering [Ding and Luo, 2019], DualAE [Yang et al., 2019], RTM [Nina et al., 2019], NCSC [Zhang et al., 2019], IIC [Ji et al., 2019], DCCM [Wu et al., 2019] and DCCS [Zhao et al., 2020].

4.3.2 Method Comparison

The main results are present in Table 4.1, where we compare the performance of our method with state-of-the-art deep clustering methods and some baseline classical non-deep learning based methods. All of the deep clustering methods we were able to find used horizontal clustering, *i.e.*, assuming a known number of clusters. The clusters are assumed to be equivalent to the classes in each dataset. There are 10 classes for each dataset, and thus 10 clusters are used for each. In order to compare our method’s ACC and NMI results with the other benchmarks, we need to obtain a clustering tree with 10 clusters (leaf nodes), and we do that by simply stopping the tree growth when it reaches the 10-th cluster.

MNIST. Our method performances on MNIST stays behind most other deep methods in terms of ACC. The results are better for NMI, since it surpasses some deep learning based methods like NCSC, ClusterGAN and VaDE. It is worth mentioning

Method	Deep?	MNIST		Fashion MNIST		CIFAR-10	
		ACC	NMI	ACC	NMI	ACC	NMI
K-means	✗	0.572	0.500	0.474	0.512	0.229	0.087
SC	✗	0.696	0.663	0.508	0.575	0.247	0.103
AC	✗	0.695	0.609	0.500	0.564	0.228	0.105
NMF	✗	0.545	0.608	0.434	0.425	0.190	0.081
DEC	✓	0.843	0.772	0.590	0.601	0.301	0.257
JULE	✓	0.964	0.913	0.563	0.608	0.272	0.192
VaDE	✓	0.945	0.876	0.578	0.630	-	-
DEPICT	✓	0.965	0.917	0.392	0.392	-	-
SpectralNet	✓	0.971	0.924	0.533	0.552	-	-
ClusterGAN	✓	0.950	0.890	0.630	0.640	-	-
DLS-Clustering	✓	0.975	0.936	0.693	0.669	-	-
DualAE	✓	0.978	0.941	0.662	0.645	-	-
RTM	✓	0.968	0.933	0.710	0.685	-	-
NCSC	✓	0.941	0.861	0.721	0.686	-	-
IIC*	✓	0.992*	0.978*	0.657*	0.637*	0.617*	0.513*
DCCM*	✓	-	-	0.657*	0.637*	0.623*	0.496*
DCCS*	✓	0.989*	0.970*	0.756*	0.704*	0.656*	0.569*
Our Method	✓	0.930	0.892	0.721	0.691	0.387	0.289

Table 4.1. Main results: Clustering performance of different algorithms on 3 datasets based on ACC and NMI. The column “Deep?” indicates if the respective algorithm is based on deep learning or not. *: Algorithms and results that made use of data augmentation techniques aimed for clustering.

that we did not explore many architecture and hyperparameter tuning possibilities for obtaining a better result for MNIST, with this result being among the first attempts obtained by simply employing the same architecture and very similar set of hyperparameters we used for obtaining a good result with Fashion MNIST. The model employed for MNIST might be operating with excessive capacity, which might undermine the MGAN generators ability to divide the generated dataset into two regions (a crucial step for our clustering to work) since it becomes trivial for each generator to represent the entire training that the MGAN has access to.

Fashion MNIST. Only 2 methods (DCCS, RTM) are able to surpass our method’s ACC performance. Only DCCS surpasses our method in NMI, and by a very thin margin.

CIFAR-10. Our method is able to outperform the non-deep baselines, as well as the deep learning methods DEC and JULE, in the NMI and ACC metrics. However, it performs poorly in comparison with other 3 recent deep learning state-of-the-art methods: IIC, DCCM, DCCS.

It is important to mention that IIC, DCCM, DCCS made use of data augmentation techniques, which for these datasets, especially CIFAR-10, were shown to be extremely effective in boosting clustering accuracy performance. The authors of DCCS reported that their method without the aid of data augmentation achieved 0.692 accuracy on Fashion MNIST and 0.225 accuracy on CIFAR-10, as compared, respectively, to 0.756 and 0.656 with augmentation. Their approach, however, employed an architecture similar to an Autoencoder, while ours is based on GANs, and it is not clear how to configure the training of a GAN with data augmentation, especially a data augmentation specifically adapted for clustering purposes, such as the one the authors of DCCS used. We made some attempts to adapt DCCS’s approach to data augmentation to our method and, even though we did obtain some improvements in isolated parts of the clustering tree, we failed to obtain a sustained global improvement, since these attempts increased the GANs’ instability during the refinements of certain nodes. We will explain these attempts in further detail in Chapter 5. It should be noted that certain types of data augmentation, even though not counting as a type of supervision, might eliminate such an amount of information that is useless for clustering, by eliminating intra-class variation and thus forcing the model to concentrate on inter-class features for performing the cluster, that they are almost as helpful to the model as if it was being supervised by labels. And, in the same way that the class labels might not be known in practice, the specific types of data augmentation which would yield a better clustering result might also not be known in practice. Because of that, we argue that our method might present a more advantageous solution for some practical scenarios, since we are able to achieve a very close performance by making no assumptions about the suitability of some type of data augmentation.

4.3.3 Qualitative Analysis

Figure 4.4 shows the top 10 real MNIST samples most associated with each obtained cluster k at the leaf nodes, *i.e.*, the top 10 images with most probability in each \mathbf{s}_k at a leaf node. Each of the 10 rows is relative to one of the 10 \mathbf{s}_k at the leaf nodes. This image presents a near perfect 1-to-1 association of each class with each cluster.

Figure 4.5 does the same for the Fashion MNIST dataset. We can clearly identify class patterns along each row, with the 1st cluster associated with boots, the 2nd with bags, the 3rd with sneakers, the 4th with sandals, the 5th with pants, the 6th with dresses. For the clusters related to rows 7, 8, 9 and 10, the classes are not so well defined, with both rows 7 and 8 containing t-shirts and shirts, while both rows 9 and

10 contain pullovers and coats.

Figure 4.6 does the same for the CIFAR-10 dataset. We can identify some classes mostly associated with each row, like cats with row 1, dogs with row 2, frogs with row 3, horses with row 4. Some rows have a clear association with two classes, like row 6 with deer and birds, and row 7 with airplanes and also birds. Row 10 does not present a clear association with a specific set of classes of the dataset, instead, it is associated with images with a white background, which is undesirable for a comparison with the reference classes.



Figure 4.4. Top 10 real MNIST images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.

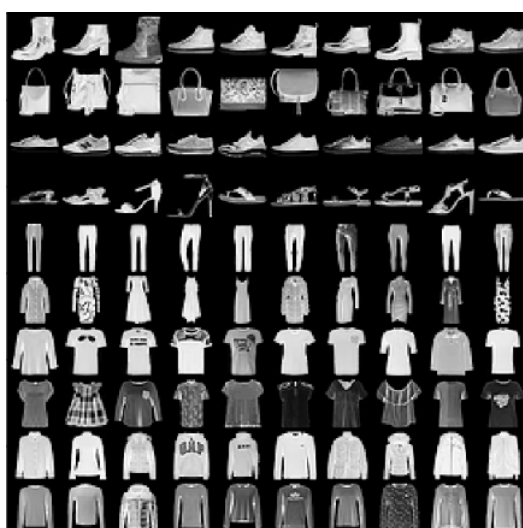


Figure 4.5. Top 10 real Fashion MNIST images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.

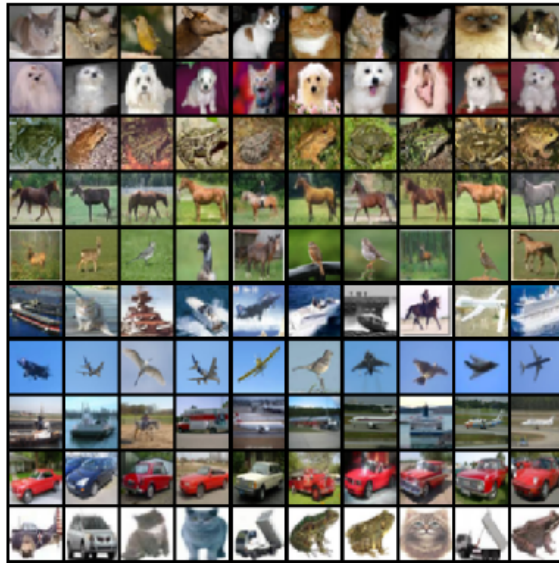


Figure 4.6. Top 10 real CIFAR-10 images most associated with each leaf (cluster) of the clustering tree. Each row refers to a cluster.

We have already presented a visualization for the entire hierarchical clustering tree with the MNIST in Figure 3.2. We provide the same type of visualization with for the clustering tree of the Fashion MNIST dataset in Figure 4.7. We again plotted, for each k -th node of the tree, 25 real examples sampled with weights given by the probabilities in each membership vector \mathbf{s}_k . The most prevalent classes in each \mathbf{s}_k are described in parenthesis and scaled by font size according to how much prevalent they are. By prevalence we mean the sum of the probability mass of all the examples belonging to the class inside a given \mathbf{s}_k . To be more precise, the probability mass of a class A inside \mathbf{s}_k is given by $\sum_i^N s_{k,i} \cdot \mathbf{1}(c_i = A)$, where $\mathbf{1}$ is an indicator function and c_i is the class of the i -th example. The scale of font sizes for each class label in Figure 4.7 varies through 10 font size, with the sizes linearly representing a range from the minimum possible mass a class can have inside \mathbf{s}_k , which is 0, to the maximum possible mass, which for Fashion MNIST is 7000, since there are 7000 examples of each class. Looking at the tree of Figure 4.7, we can notice how the 1st split occurred with almost perfect precision, with examples of the same class having their probability mass nearly entirely allocated to either \mathbf{s}_1 or \mathbf{s}_2 . We begin to notice some imprecision occurring at the 2nd split (with a small portion of probability mass of the coat and t-shirt classes being sent to \mathbf{s}_3 while the largest portion went to \mathbf{s}_4) and 3rd split (with a small portion of sneakers mass being sent to \mathbf{s}_5 and the largest portion going for \mathbf{s}_6). The most imprecise splits occurred during the 4th, 7th and 9th splits, and we can note that the classes involved in these 3 splits (t-shirt, pullover, coat and shirt) are the most visually similar present in the dataset, thus being the hardest to accurately separate

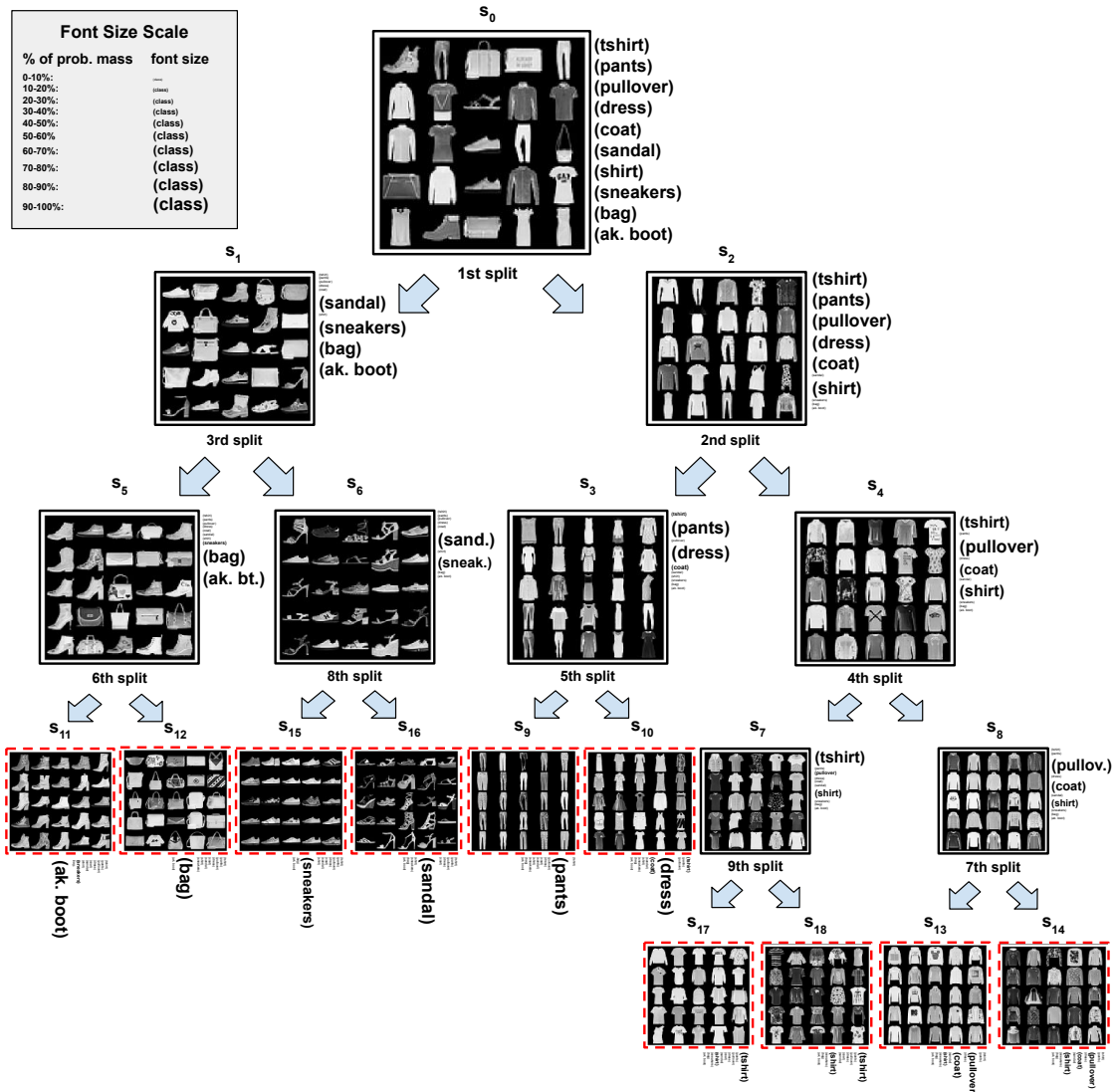


Figure 4.7. Complete hierarchical tree constructed over the Fashion MNIST dataset. Each vector \mathbf{s}_k is placed together with a square grid containing 25 real examples sampled from cluster k 's probability distribution (a distribution formed by sampling a real example i with weights given by the i -th probability in \mathbf{s}_k). The leaf nodes samples are indicated with a segmented red line for the borders of the square grid. Along with each square grid, in parenthesis, we indicate the name of the classes in varying font sizes, with each font size proportional to the amount of probability mass the class has in \mathbf{s}_k , using 10 font sizes, linearly representing 10 intervals, ranging from closer to 0% of total mass (or 0 mass) to closer to 100% of total mass, (or 7000 of mass).

into clusters. The other classes were relatively well separated.

There is a property of this clustering worth noticing that might explain why it had a result more competitive in the benchmarks with NMI than with ACC: notice how even the classes that had a bad accuracy result tended to have its probability mass

sent to no more than two different clusters. For instance, by observing the leaf nodes \mathbf{s}_{13} and \mathbf{s}_{14} we can see that the class coat had one of the worst separation accuracy results, but its probability mass was mostly spread between \mathbf{s}_{13} and \mathbf{s}_{14} instead of being spread among many other different leaf nodes. Something similar occurs for shirt (mostly spread between \mathbf{s}_{18} and \mathbf{s}_{14}), t-shirt (mostly spread between \mathbf{s}_{18} and \mathbf{s}_{17}) and pullover (mostly spread between \mathbf{s}_{13} and \mathbf{s}_{14}). For the ACC metric, there would be no difference if a class A had a given amount of its probability mass wrongfully sent to many different clusters or to only one cluster other than the expected cluster where most of A 's probability mass is located. For the NMI metric, on the other hand, the situation where this same amount of A 's probability mass gets wrongfully sent to only one other cluster provides a better mutual information between clusters and classes than the situation where A 's probability mass gets wrongfully sent to many different clusters.

We now present the same kind of visualization for the hierarchical clustering tree with the CIFAR-10 dataset in Figure 4.8. The number of samples per grid, for each k -th node of the tree, are varied now, aiming to optimize the number of samples as well as the size of images in order to better visualize the clustering of CIFAR-10, whose images are considerably more complex and diverse than the images of MNIST and Fashion MNIST. We can observe that the first split was able to separate the classes according to a cluster related with animals and another cluster related with means of transportation, with considerable accuracy. With the 3rd split, the means of transportation were further divided with good accuracy into clusters of similar objects, with one cluster formed with plane/ship and another with car/truck. The animal cluster, on the other hand, was not as accurately divided after the 2nd split, but roughly it became a cluster with classes dog/frog/cat and another cluster with the classes bird/deer/horse. The following splits were conducted with considerably less accuracy in regard to the reference classes, which might be explained by the intra-class visual differences becoming more dominant than the inter-class differences. An example of this situation is in the separation performed by the 8th split, creating nodes \mathbf{s}_{15} and \mathbf{s}_{16} . By visual inspection, we can conclude that \mathbf{s}_{16} ended up with breeds (mostly dogs) with white fur or feathers, while \mathbf{s}_{15} ended up with general animal breeds with more varied colors. This problem is very hard to overcome without the labels or some data augmentation choice informing the model that the color information should not be adequate in this case for separating the examples. The most well defined associations with classes in the leaf nodes were \mathbf{s}_9 with frogs and \mathbf{s}_{16} with dogs. The other clusters ended up associated with more than 1 class, and \mathbf{s}_{17} , in particular, ended up with almost no probability mass, which greatly hinders the final accuracy result, since we must choose one class to be represented by

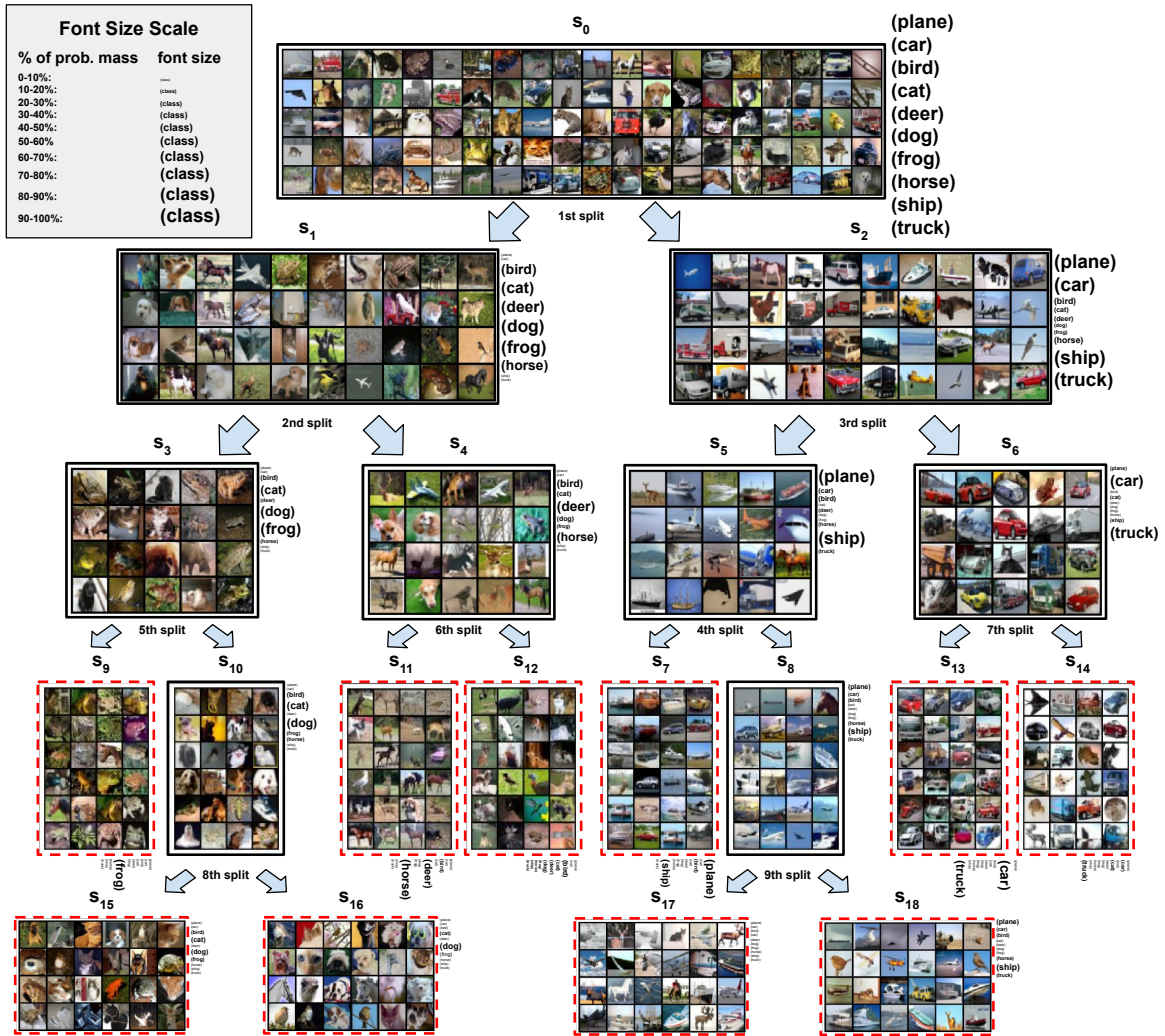


Figure 4.8. Complete hierarchical tree constructed over the CIFAR-10 dataset. Each vector s_k is placed together with a grid containing varying numbers of real examples sampled from cluster k 's probability distribution (a distribution formed by sampling a real example i with weights given by the i -th probability in s_k). The leaf nodes samples are indicated with a segmented red line for the borders of the grid. Along with each grid, in parenthesis, we indicate the name of the classes in varying font sizes, with each font size proportional to the amount of probability mass the class has in s_k , using 10 font sizes, linearly representing 10 intervals, ranging from closer to 0% of total mass (or 0 mass) to closer to 100% of total mass, (or 6000 of mass).

it because of the 1-to-1 mapping from clusters to reference classes.

4.4 Implementation Details

This section describes our experiment settings as well as configurations necessary to reproduce our results.

4.4.1 Deep learning framework and hardware

We have used PyTorch, a well known framework for developing deep learning models. PyTorch provides a high-level interface in Python programming language for the efficient training of neural networks on graphical processing units (GPUs), whose strong capacity for parallel processing accelerates the training speed considerably.

The computer used for this work is owned by the main author. Its hardware settings of relevance for this work are the following: Nvidia RTX 2070 GPU, Intel Core i5-7600K CPU.

4.4.2 Architecture

Table 4.2 describes the architecture settings used by each component. G , D and C respectively refer to each generator, discriminator and classifier created, either for a raw split sub-block or for a refinement sub-block. LN is short for Layer Normalization [Ba et al., 2016], a well known deep learning normalization technique that is also used for stabilizing GAN training [Kurach et al., 2019]. Another important implementation detail to notice is that the convolution layers for D and C share the same weights, *i.e.*, the same $\mathbf{x}_{feature} = \text{conv3}(\text{conv2}(\text{conv1}(\mathbf{x})))$ will be received as input by the non-shared Fully Connected layers of D and C . These convolution weights are shared in order to force the classifier to differentiate between examples in a higher-level feature space, which will be learned by D in its adversarial game with G . Learning to perform the classification in a higher-level feature space is more desirable than doing it in the plain data space (for images, the pixel space), since it is less likely to lead to a overfitted classification that uses low-level information to distinguish between the generated samples, which in turn could impair the quality of the clustering created by the classifiers inference on the training data afterwards. A more specific detail regarding these shared convolutions is that only the gradients of the discriminator’s loss are used to update the weights for these layers, with the classifier’s gradients not being necessary for this update. This is because we empirically verified that the discriminator’s learning was enough to obtain a sufficiently separable feature space for C ’s fully connected layer to perform its classification with high accuracy.

Operation	Kernel	Strides	Feature Maps	LN?	Activation
Generator: $G(\mathbf{z}) : \mathbf{z} \sim \text{Uniform}[0, 1]$					
Fully Connected			100		
Transposed Convolution	4×4	2×2	$\frac{\text{img_dim}^2}{4^2} \times 128$	No	ReLU
Transposed Convolution	4×4	2×2	64	No	ReLU
Transposed Convolution	4×4	2×2	no. of color channels	No	Tanh
Discriminator: $D(\mathbf{x})$					
Convolution (Shared with C)	5×5	2×2	128	Yes	Leaky ReLU
Convolution (Shared with C)	5×5	2×2	256	Yes	Leaky ReLU
Convolution (Shared with C)	5×5	2×2	512	Yes	Leaky ReLU
Fully Connected			1	No	Sigmoid
Classifier: $C(\mathbf{x})$					
Convolution (Shared with D)	5×5	2×2	128	Yes	Leaky ReLU
Convolution (Shared with D)	5×5	2×2	256	Yes	Leaky ReLU
Convolution (Shared with D)	5×5	2×2	512	Yes	Leaky ReLU
Fully Connected			2	No	Softmax

Table 4.2. Architecture settings.

4.4.3 Hyperparameters

Table 4.3 describes the main hyperparameters used for the clustering of each dataset. Most of these settings are well known configurations in deep learning tasks, and they were all chosen based on similar GAN architectures employed for other tasks on these datasets, with some slight fine-tuning modifications that provided a better stabilization for each GAN training along with each classifier. Unusual configurations worth explaining are: the increase in epochs for each refinement, the diversity parameter γ and the initial noise variance. **Increasing training epochs at each iteration of a refinement sub-block:** The reason for this is our empirical verification that in order to obtain improvements over the last refinement result, a GAN created for a refinement t needed to learn a better representation of its observed real data distribution than the GAN created at refinement $t - 1$, and this better representation could be more easily achieved with more training time. **Diversity parameter γ :** regularization parameter used on the generator for weighting the contribution of the classification loss vs the contribution of the adversarial loss, which is explained in detail by the descriptions of Algorithms 3.2 and 3.4. **Initial noise variance:** Adding Gaussian noise (with variance that linearly decays during the training epochs) to both generated and real images is a known stabilization practice for GAN training, as explained in [Jenni and Favaro, 2019].

Parameters Settings	MNIST	Fashion MNIST	CIFAR10
Batch size for real data	100	100	100
Batch size for each generator	100	100	100
Number of epochs (raw split)	80	80	150
Number of epochs (t-th refinement)	$80 + (10 * t)$	$80 + (10 * t)$	$150 + (10 * t)$
Slope of Leaky ReLU	0.2	0.2	0.2
Learning rate for generator	0.0002	0.0002	0.0002
Learning rate for discriminator	0.0001	0.0001	0.0002
Learning rate for classifier	0.00002	0.00002	0.00002
Adam Optimizer	$\beta_1 = 0.5, \beta_2 = 0.999$	$\beta_1 = 0.5, \beta_2 = 0.999$	$\beta_1 = 0.5, \beta_2 = 0.999$
Diversity parameter γ	1.0	1.0	1.0
Initial noise variance	1.5	1.5	1.5

Table 4.3. Specific hyperparameter settings used for each dataset.

Chapter 5

Future Works

In this section, we discuss the results in more detail pointing to aspects that may lead to further improvements.

5.1 Training Time

The main drawback of our model consists in the complexity of sequentially training multiple different GAN/MGAN modules, especially in regard to running time. Each split in the tree creates one MGAN module for the raw split, and two new GAN modules for each refinement iteration. We performed 12 refinement iterations for our experiments, meaning that 24 GANs are trained for the refinements happening in each split. For creating a tree that reaches 10 leaf nodes, which was the case for our experiments, we need 9 splits. So, in total, 9×1 MGAN modules and 24×9 GAN modules were created for each experiment, each one being trained to learn to represent from scratch its input distribution given by \mathbf{s}_k . It should be noted nonetheless that we set the number of minibatch samples for each epoch to $\sum_i^N s_{k,i}$, for a GAN or MGAN receiving its real image samples according to \mathbf{s}_k , and because the probability mass in \mathbf{s}_k decreases at each newly created node, the number of minibatches will also decrease and hence the training time per epoch will also decrease for a GAN/MGAN at each new node. Even then, training so many GANs/MGANs still takes an unreasonable amount of time, taking multiple days to complete an entire tree with a single GPU.

We have tried to overcome the running time issue with the following method: for a refinement iteration of a certain \mathbf{s}_k , instead of creating new GANs at each iteration t and training the components from scratch, we preserve the weights learned by the GAN at the last iteration t . The reasoning behind this is that the GAN at refinement t can be trained for less epochs, since it already starts its training with a previously learned

representation of the data, having only to improve on it based on the newly estimated $\mathbf{s}_k^{(t)}$. This worked well for some nodes of the tree, yielding the same clustering result we achieved before, but with considerably less training epochs per refinement. But for some other nodes, after a certain number of refinements, the Generators’ loss started to increase excessively, creating instability that compromised the image generation and thus worsened the clustering result. We still do not fully understand why this happens, but we believe that this method might work better with some other type of architecture design, especially one that employs a loss function with better robustness and convergence guarantees, such as the Wasserstein loss, instead of the common non-saturating loss used in our work.

5.2 Hyperparameter and Architecture Choices

The excessive time required to train so many GAN/MGAN modules discussed in the last section constitutes in itself a limitation for the search of a set of unique hyperparameters and architectures that work well for each GAN/MGAN. But other than that, each GAN/MGAN is trained with different data distributions for each node of the tree, with each distribution becoming increasingly more homogeneous and thus simpler as nodes are created further away from the root. This difference in distribution complexity between different nodes means that the same set of hyperparameters might not be ideal to train all GANs/MGANs. It would be desirable that the hyperparameter tuning for each GAN/MGAN reflected the complexity of its respective distribution, *e.g.*, for a simpler and more uniform data distribution, we could decrease the capacity of the GAN/MGAN architecture receiving it, or perhaps increase the diversity parameter γ to encourage the 2 generated distributions at each split to become more distinct from each other.

5.3 Data Augmentation

As already mentioned in Subsection 4.3.2, we intended to improve our model’s performance by adapting it to benefit from data augmentation in a similar fashion as the works of [Ji et al., 2019], [Wu et al., 2019] and [Zhao et al., 2020]. More specifically, we tried to adapt the data augmentation approach employed in [Zhao et al., 2020], named DCCS, where the authors developed an Autoencoder architecture for clustering, capable of decomposing its learned representation into a stylistic component \mathbf{z}_s (encoding features common to every class) and a categorical component \mathbf{z}_c (encoding the class

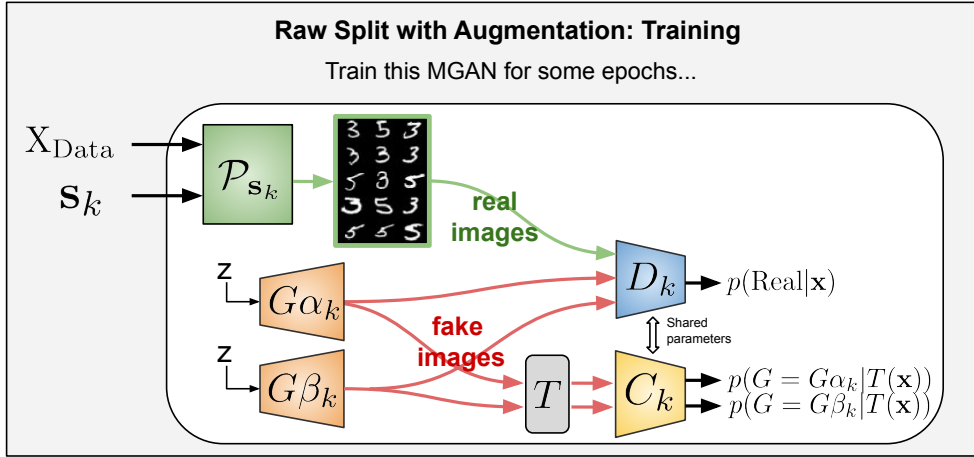


Figure 5.1. Raw split sub-block adapted for data augmentation

information, and being used to form the clusters afterwards). The data augmentation was used in conjunction with a loss enforcing that an example and its augmented version had the same representation in \mathbf{z}_c . To be more precise, for an augmentation function T , \mathbf{z}_c and \mathbf{z}'_c encoded from \mathbf{x}_i and $T(\mathbf{x}_i)$ should be as similar as possible, while the stylistic changes due to the augmentation should be encoded by \mathbf{z}_s and \mathbf{z}'_s . T performs stylistic transformations that do not alter the category of an example (e.g., for images it would be cropping, resizing, variations in brightness and contrast).

At first glance, it is not obvious how our model could be adapted to do something similar to DCCS, since it is not based on Autoencoders and thus lacks an explicit encoder map from \mathbf{x}_i to \mathbf{z}_c . Because \mathbf{z}_c is a categorical vector that forms the clusterization for DCCS, we suspect that the key to attaining an equivalent result lies in enforcing that our component responsible for the clustering becomes insensitive to the style variations caused by the augmentation. In our method, this component is the classifier C (both in the raw split and refinement phases), since it ends up implicitly learning a clusterization (*i.e.*, a categorical encoding) for the real data distribution based on the 2 generated distributions.

Having this in mind, we propose the following: for a generated image \mathbf{x}_G and an augmentation function T , C must still perform its learning as usual, trying to guess from which generator \mathbf{x}_G came from, but receiving as input $T(\mathbf{x}_G)$ instead of \mathbf{x}_G . If C is able to guess correctly, then this means that the component responsible for the clustering has become insensitive to the style variation caused by the augmentation function T . In Figure 5.1, we depict how this alteration would occur for a raw split sub-block on the k -th node, reproducing the raw split scheme previously shown in Figure 3.5, but with a T block before the input to the classifier, representing the augmentation function T (it would be basically the same alteration for a refinement

sub-block, with a T block before the input of each classifier). Note that we should apply the augmentation with T only for the input of C , while D still performs its learning receiving as input \mathbf{x}_G without augmentation. This is because we do not need to have the generators learning the augmented version of the real dataset. This is in accordance with the work of DCCS, since the augmented images were used solely to control the categorical component \mathbf{z}_c , while their style component \mathbf{z}_s didn't make any difference for their algorithm and was not used for any kind of learning.

The first experiments we performed with this alteration yielded improvements for some nodes of the tree constructed with the Fashion MNIST dataset relative to the same experiment but without using augmentation. But for other nodes, performing the augmentation made it too hard for the classifier to distinguish between the 2 generated distributions with good accuracy, which is necessary for a good clustering result afterwards on the real dataset. We believe that adding more capacity to the classifier network might fix this problem.

Chapter 6

Conclusion

In this work, we proposed a method for hierarchical clustering that takes advantage of the deep representation capacity obtained by GANs and MGANs. It constructs a tree of clusters from top to bottom, with each leaf node associated with a cluster. Each cluster is divided in binary splits, with each split occurring in two phases: a raw split and a refinement phase.

The raw split in itself constitutes a new clustering method employing MGANs, taking advantage of the fact that each generator naturally specializes in sub-regions of the dataset to train a classifier to learn clusters for the real dataset based on the generated data. Since we use two generators for the MGAN, each cluster correlates more strongly with one of the two generated distributions. The refinement phase creates two separated regular GANs, each one trained to specialize in one of the previous clusters even further. Similarly to the raw split, new classifiers are also trained to differentiate between the generated data from each GAN, and then are used to re-estimate the clustering results with the real dataset, tending to obtain a new cluster that correlates more with the classes of the dataset than the previous cluster. This refinement procedure is repeated in a loop for some iterations so that it can slightly enhance the previous result.

We have shown how well our method compares to other deep clustering techniques on clustering datasets, obtaining competitive results, especially with respect to the NMI metric. Moreover, our method is, to the best of our knowledge, the only deep clustering method able to construct a hierarchy of clusters. We have explored in detail the clustering tree, verifying that it accurately organizes the examples in a hierarchy of semantically coherent characteristics as expected. We have addressed some shortcomings of the method and proposed some ways to overcome them, especially in regard to the fact that it does not employ data augmentation like other state-of-the-art deep

clustering methods.

Bibliography

- [Achanta and Susstrunk, 2017] Achanta, R. and Susstrunk, S. (2017). Superpixels and polygons using simple non-iterative clustering. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Ba et al., 2016] Ba, L. J., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *CoRR*, abs/1607.06450.
- [Brock et al., 2019] Brock, A., Donahue, J., and Simonyan, K. (2019). Large scale GAN training for high fidelity natural image synthesis. In *International Conference on Learning Representations (ICLR)*.
- [Cai et al., 2009] Cai, D., He, X., Wang, X., Bao, H., and Han, J. (2009). Locality preserving nonnegative matrix factorization. In *21st International Joint Conference on Artificial Intelligence (IJCAI)*.
- [Chen et al., 2016] Chen, X., Duan, Y., Houthoofd, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In *Advances in Neural Information Processing Systems (NIPS)*.
- [Ding and Luo, 2019] Ding, F. and Luo, F. (2019). Clustering by directly disentangling latent space. *CoRR*, abs/1911.05210.
- [Ghasedi Dizaji et al., 2017] Ghasedi Dizaji, K., Herandi, A., Deng, C., Cai, W., and Huang, H. (2017). Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In *IEEE International Conference on Computer Vision (ICCV)*.
- [Ghosh et al., 2018] Ghosh, A., Kulharia, V., Namboodiri, V., Torr, P. H. S., and Dokania, P. K. (2018). Multi-agent diverse generative adversarial networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

- [Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial net. In *Advances in Neural Information Processing Systems (NIPS)*.
- [Goodfellow, 2016] Goodfellow, I. J. (2016). NIPS 2016 tutorial: Generative adversarial networks. *CoRR*, abs/1701.00160.
- [Gowda and Krishna, 1978] Gowda, K. C. and Krishna, G. (1978). Agglomerative clustering using the concept of mutual nearest neighbourhood. *Pattern Recognition*.
- [Hoang et al., 2018] Hoang, Q., Nguyen, T., Le, T., and Phung, D. (2018). Mgan: training generative adversarial nets with multiple generators. In *International Conference on Learning Representations (ICLR)*.
- [Isola et al., 2017] Isola, P., Zhu, J., Zhou, T., and Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Jenni and Favaro, 2019] Jenni, S. and Favaro, P. (2019). On stabilizing generative adversarial training with noise. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Ji et al., 2019] Ji, X., Henriques, J. F., and Vedaldi, A. (2019). Invariant information clustering for unsupervised image classification and segmentation. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.
- [Jiang et al., 2017] Jiang, Z., Zheng, Y., Tan, H., Tang, B., and Zhou, H. (2017). Variational deep embedding: An unsupervised and generative approach to clustering. In *26th International Joint Conference on Artificial Intelligence (IJCAI)*.
- [Joulin et al., 2010] Joulin, A., Bach, F., and Ponce, J. (2010). Discriminative clustering for image co-segmentation. In *Conference on Computer Vision and Pattern Recognition*.
- [Karras et al., 2019] Karras, T., Laine, S., and Aila, T. (2019). A style-based generator architecture for generative adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Khayatkhoei et al., 2018] Khayatkhoei, M., Singh, M. K., and Elgammal, A. (2018). Disconnected manifold learning for generative adversarial networks. In *Advances in Neural Information Processing Systems (NIPS)*.

- [Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Neural Information Processing Systems (NIPS)*.
- [Kuhn, 1955] Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*.
- [Kundu et al., 2019] Kundu, J. N., Gor, M., Agrawal, D., and Babu, R. V. (2019). Gan-tree: An incrementally learned hierarchical generative framework for multi-modal data distributions. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.
- [Kurach et al., 2019] Kurach, K., Lučić, M., Zhai, X., Michalski, M., and Gelly, S. (2019). A large-scale study on regularization and normalization in GANs. In *36th International Conference on Machine Learning (ICML)*.
- [Liu et al., 2019] Liu, H., Jiang, B., Xiao, Y., and Yang, C. (2019). Coherent semantic attention for image inpainting. In *IEEE International Conference on Computer Vision (ICCV)*.
- [Liu et al., 2018] Liu, Z., Lin, G., Yang, S., Feng, J., Lin, W., and Goh, W. L. (2018). Learning markov clustering networks for scene text detection. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [MacQueen, 1967] MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*.
- [Mukherjee et al., 2019] Mukherjee, S., Asnani, H., Lin, E., and Kannan, S. (2019). Clustergan: Latent space clustering in generative adversarial networks. *Conference on Artificial Intelligence (AAAI)*.
- [Nina et al., 2019] Nina, O., Moody, J., and Milligan, C. (2019). A decoder-free approach for unsupervised clustering and manifold learning with random triplet mining. In *IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*.
- [Radford et al., 2016] Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In *4th International Conference on Learning Representations, (ICLR)*.

- [Reed et al., 2016] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H. (2016). Generative adversarial text to image synthesis. In *International Conference on Machine Learning (ICML)*.
- [Shaham et al., 2018] Shaham, U., Stanton, K. P., Li, H., Basri, R., Nadler, B., and Kluger, Y. (2018). Spectralnet: Spectral clustering using deep neural networks. In *6th International Conference on Learning Representations, (ICLR)*.
- [Shi and Malik, 2000] Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence*.
- [Tulyakov et al., 2018] Tulyakov, S., Liu, M.-Y., Yang, X., and Kautz, J. (2018). Moco-gan: Decomposing motion and content for video generation. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Wu et al., 2019] Wu, J., Long, K., Wang, F., Qian, C., Li, C., Lin, Z., and Zha, H. (2019). Deep comprehensive correlation mining for image clustering. In *IEEE/CVF International Conference on Computer Vision (ICCV)*.
- [Xie et al., 2016] Xie, J., Girshick, R., and Farhadi, A. (2016). Unsupervised deep embedding for clustering analysis. In *International Conference on International Conference on Machine Learning (ICML)*.
- [Yang et al., 2016] Yang, J., Parikh, D., and Batra, D. (2016). Joint unsupervised learning of deep representations and image clusters. *CoRR*, abs/1604.03628.
- [Yang et al., 2019] Yang, X., Deng, C., Zheng, F., Yan, J., and Liu, W. (2019). Deep spectral clustering using dual autoencoder network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Yu et al., 2018] Yu, J., Lin, Z., Yang, J., Shen, X., Lu, X., and Huang, T. S. (2018). Generative image inpainting with contextual attention. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
- [Zelnik-Manor and Perona, 2004] Zelnik-Manor, L. and Perona, P. (2004). Self-tuning spectral clustering. In *17th International Conference on Neural Information Processing Systems (NIPS)*.
- [Zhang et al., 2018] Zhang, H., Xu, S., Jiao, J., Xie, P., Salakhutdinov, R., and Xing, E. P. (2018). Stackelberg GAN: towards provable minimax equilibrium via multi-generator architectures. *CoRR*, abs/1811.08010.

- [Zhang et al., 2019] Zhang, T., Ji, P., Harandi, M., Huang, W., and Li, H. (2019). Neural collaborative subspace clustering. In *36th International Conference on Machine Learning (ICML)*.
- [Zhang et al., 2018] Zhang, X., Zhu, X., Zhang, . X., Zhang, N., Li, P., and Wang, L. (2018). Seggan: Semantic segmentation with generative adversarial network. In *IEEE Fourth International Conference on Multimedia Big Data (BigMM)*.
- [Zhao et al., 2020] Zhao, J., Lu, D., Ma, K., Zhang, Y., and Zheng, Y. (2020). Deep image clustering with category-style representation. In *European Conference on Computer Vision (ECCV)*.