# UNSUPERVISED ANOMALY DETECTION IN MULTI-CLASS DATASETS USING GENERATIVE ADVERSARIAL NETWORKS

BY

WALKER L. DIMON

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Master's Committee:

Clinical Associate Professor of Practice Michael F. Lembeck
Research Assistant Professor Huy T. Tran

# Abstract

Presented in this thesis is a novel Generative Adversarial Network, or GAN, based method, D-AnoGAN, for detecting anomalies in complex datasets containing disconnected data manifolds. Current state of the art methods treat disconnected data manifolds as a single, continuous one to learn from. The key contribution of D-AnoGAN is specifically accounting for the discontinuity between manifolds within a dataset during training. To achieve this, a multi-generator network is first implemented, where each generator is responsible for learning a unique manifold of data. Second, a machine learning mechanism called a "bandit" is implemented to find the optimal set of generators required to cover all data manifolds through unsupervised prior-learning. Finally, the multi-generator and bandit are used to cluster data from the same manifold together during training, allowing them to be learned in a disconnected fashion. The proposed method's effectiveness is demonstrated on two publicly available datasets, as well as a new experimental dataset developed in-house, where state of the art results are achieved.

# Acknowledgements

Dr. Michael Lembeck, I am deeply grateful for having the opportunity to work with you at The University of Illinois. You have not only helped guide me through my graduate program, but through life in general in the midst of a global pandemic. Your mentorship and friendship have been invaluable to my experience here. Thank you for everything.

Dr. Huy Tran, I am thankful for how graciously you took me under your wing and opened my eyes to the beautiful world of artificial intelligence. Working with you has been a wonderful experience, and I want to thank you for inspiring me to pursue a career in the exciting field of AI research.

Thank you to my colleagues and friends Hongrui Zhao, Marc Akiki, Eric Alpine, Nick Chase, Jacob Heglund, and Neale Van Stralen, as well as everyone else in my research groups for bringing added light to my graduate school experience. I look forward to seeing what your futures have in store.

I would not be here today without the support of my mother, Terri Dimon, my father, William Dimon, and my sister, Franni Dimon. I love and thank you all.

Finally, thank you to Emily Renne, Babe, Rosie, and Suzie for being by my side every day of my graduate program. The four of you have taught me what true happiness in life is all about.

*To my hero and grandfather, Gerald Dimon.*

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In today's technological world, a fascinating area of research focuses on the automation of tasks, traditionally performed by human beings, through the use of machines and artificial intelligence. Machine learning, a branch of artificial intelligence, automates tasks using computer models, which learns a mapping of representative input data to desired output responses, to develop understanding from vast amounts of data [1]. Inspired by the way human beings themselves learn, machine learning models follow an iterative learning process in order to discover meaningful insights about datasets, predicting outcomes and detecting emergent patterns within the data as a result.

For a machine model to learn, a training process is performed to gain an understanding of the data that it is being fed. During training, a special type of algorithm, known as an artificial neural network [2], is often used. Artificial neural networks (neural networks for short) are able to process large amounts of data quickly and effectively, thanks in large part to their unique algorithmic structure [3, 4]. A neural network is composed of a collection of connected nodes called "neurons," modeled after the structure of the human brain. These neurons are used to pass data information from an input to an output, applying a summation of "weighted" multiplication operations to each data point. This weighted summation is then modified by an "activation function" and passed through the neuron as output. Activation functions help a neural network learn complex patterns in the data, allowing important pieces of information to pass through the neuron only. A visualization of this node operation is shown in Figure 1.1.

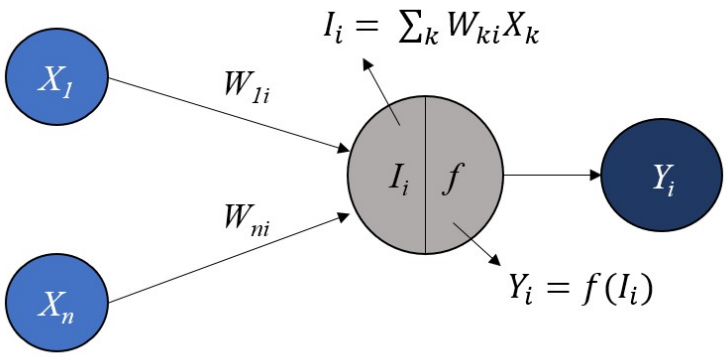


Figure 1.1: Neurons Within a Neural Network. Data information $X$ is adjusted by a set of weights $W$ as input in to the neuron and summed together, resulting in $I_i$. The data summation, $I_i$, is then modified using an activation function $f$, resulting in an output $Y_i$.

Typically, these neurons are assembled into hierarchical " hidden layers" that effectively act as a transfer function, mapping inputs to outputs throughout the network. In a fully connected network, each data input is passed to every neuron within a layer, where every connection to the neuron contains its own weight. As a neural network learns, these weights are updated to produce a desirable output. Each layer of neurons within a neural network uses the same activation function for selective information passing. Activation functions commonly found in practice are the sigmoid function, tanh (or Hyperbolic Tangent) function, and ReLU (or Rectified Linear Unit) function, although there are many others. A visualization of a simple network architecture is provided in Figure 1.2.

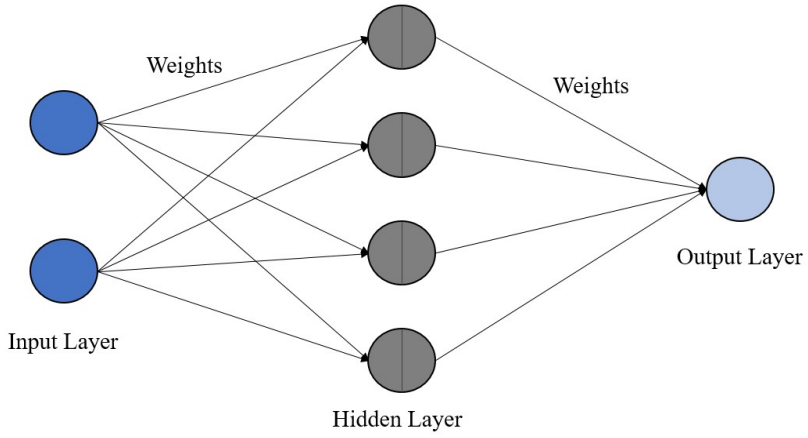


Figure 1.2: Network neurons can be assembled into layers that act as a transfer function, mapping inputs to outputs throughout the neural network. Each neuron within a layer has its own weight, followed by an activation function before its output.

When training neural networks to learn data representations, data information passes from the network's input nodes through a specified set of hidden network layers, finishing with an output layer of information [5]. The job of the hidden layers is to perform nonlinear transformations to the input data. During each iteration of training, commonly referred to "episodes" or "epochs", the set of weights associated with each hidden layer is updated based on a loss metric between the network's output and a targeted output defined by the neural network designer. Updating these weights will influence which node information is passed between the network layers until an optimal set of nodes are discovered.

The objective of a neural network during training is to optimize an algorithm used to fit training data to a desired output [6]. During each training iteration, data samples are input to the network and the difference between the computed output and desired output is calculated. Training a neural network involves adjusting the weights on each input line within the network such that the difference error is iteratively reduced, and the desired output obtained. During the update process, weights are updated through backpropagation, which computes the gradient of the loss with respect to the weights of the network. This process is illustrated in Figure 1.3.

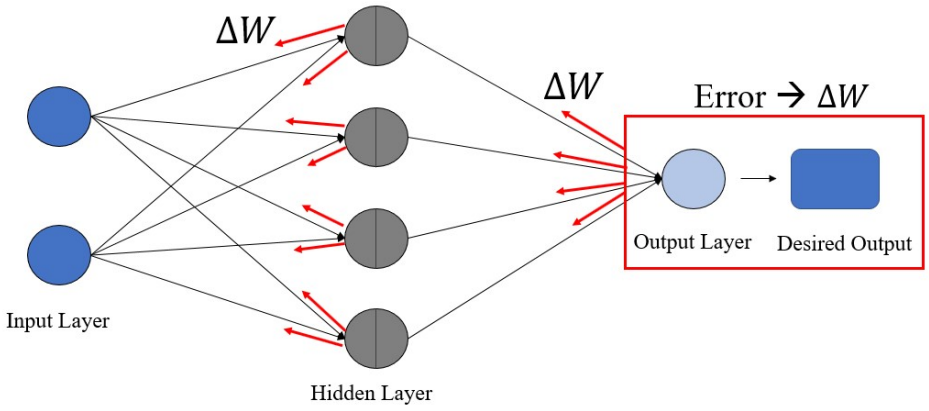


Figure 1.3: Backpropogation in a Neural Network. The output layer of a neural network layer is compared to the desired output for the network, where an error is derived. The gradient of this error, or loss, is used to apply a change to each weight in the network, $\Delta W$.

As training progresses, the learning process of neural networks are controlled by a set of parameters, often referred to as hyperparameters. Several hyperparameters are employed to influence the learning process, including the learning rate, which controls the step size at each iteration of training while moving toward a minimum, the network size (input and output dimensions), the number of hidden layers, and the batch-size of data samples used before weight updates throughout each training epoch. Finding an optimal set of hyperparameters is up to the architectural designer, and is very important. In practice, each neural network architecture requires a unique set of hyperparameters based on its application, and tuning these hyperparameters is critical for training performance.

When training a neural network-based model, one of two learning frameworks, "supervised" or "unsupervised" learning, is followed [7]. In supervised learning, the model is fed data that is pre-labeled (or known) by a human in order to perform training in a directed, supervised fashion. This framework is typically used for classification and regression tasks, where the end objective is to learn a function that best approximates the relationship between input data and a desired output. Alternatively, in unsupervised learning, the goal is to infer the natural structure present within a dataset. Here, the data used to train the model is unlabeled. With this, it is up to the model to discover meaningful insights within the data on its own without any human supervision. Common unsupervised learning tasks involve processes related to clustering (or grouping) of data, representation metrics of data, and density estimation (or probability distribution) of data. Figure 1.4 serves as a visual guide to help distinguish between supervised and unsupervised learning. After training is complete, a model can then be deployed to perform a certain task.

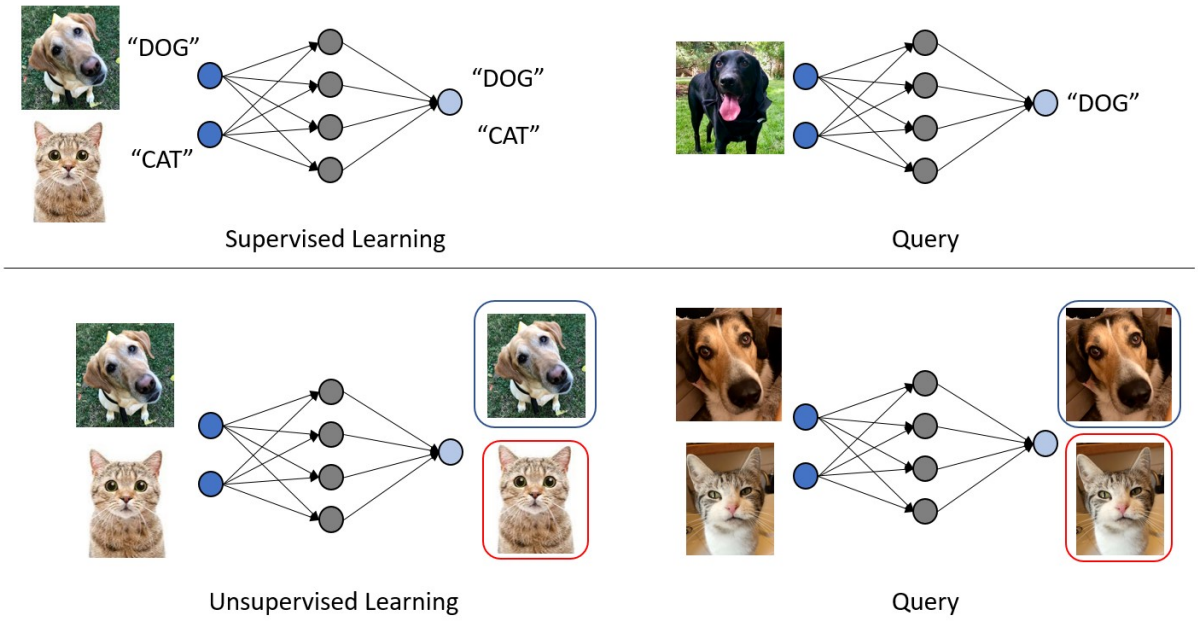


Figure 1.4: Supervised Learning vs Unsupervised Learning. Supervised learning (**top**) is used to perform tasks such as classifying unseen query images of cats as "CAT" and dogs as "DOG". Alternatively, unsupervised learning (**bottom**) is used to perform tasks that do not require training data labels, for instance separating unseen query images of dogs and cats into different clusters.

In recent years, machine learning models have been shown to be very useful in the task of anomaly detection, which is the identification of a data sample as abnormal when compared to other related data samples in a set. For example, Figure 1.5 shows a set of automotive wheels used to train a network on acceptable wheel types. When presented with a wheel containing some feature not contained within the trained dataset, the network responds to the query with an anomalous indication.

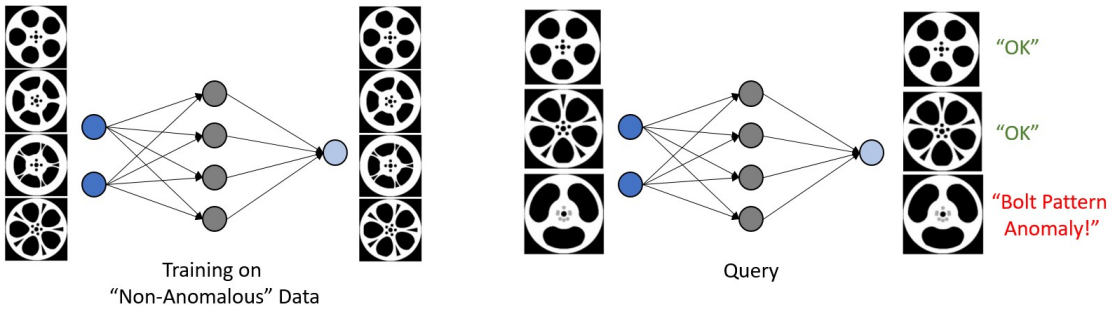


Figure 1.5: A notional example of anomaly detection tasks. Using machine learning, the task of anomaly detection can be automated to detect samples abnormal relative to dataset of "non-anomalous" data used to train a model, such as a set of wheel patterns.

The development of anomaly detection models can have a profound impact on society, benefiting a wide range of fields. In the health care industry, the automation of anomaly detection has led to advances in cancer tissue diagnosis within patients [8–10], enhancing the capability to track both disease progression and treatment monitoring. Detecting anomalous manufacturing processes, such as defects and surface-finish irregularities, have also been aided through automation [11]. Autonomous vehicles have incorporated anomaly detection models into their software to sense anomalous objects in front of them [12]. Additionally, as a matter of national security, anomaly detection models have been used to detect harmful objects within luggage compartments across US airports [13].

To automate the task of anomaly detection, neural networks have been deployed in an variety of ways, training models to detect anomalies in either a supervised or unsupervised manner [14]. For supervised anomaly detection, all data is labeled a priori as either non-anomalous or anomalous, where a neural network is then trained to distinguish between the two. Alternatively, in unsupervised anomaly detection, anomalous samples are detected without any prior knowledge that they exist. For these models, it is only assumed that all data used to train a neural network is considered non-anomalous. This is beneficial in areas where one can not anticipate all potential anomalous data samples that can occur.

With the advancement of computational capability, the size of neural network algorithms used to train machine learning models has substantially increased in recent years. Stemming from this, a neural network framework known as deep learning has become the standard for tasks such as anomaly detection. These algorithms are composed of multiple layers of neurons to extract more information out of the data being fed into it [15]. One common deep learning framework used for unsupervised anomaly detection are "deep generative methods," which learn a representation of non-anomalous data to generate new, similar data points not presented in a training dataset [16].

Deep generative methods provide particular benefit for anomaly detection tasks that utilize computer vision. In computer vision, computers are trained to interpret and understand the visual world through data in the form of digital images or videos, allowing them to "see" similar to the way humans do [17]. Computer vision-based deep generative frameworks learn the representation of an image dataset, where the model is then used to generate a reconstruction of an input query image it has not been presented before. For anomaly detection, this reconstruction is used to determine whether or not a sample is anomalous by comparing it to the input query image, which usually involves taking a pixel-level difference between the two. Reconstructions that vary greatly from a query image infers that the representation of that query was not learned during training, and are labeled as an anomalous sample. This method is visualized in Figure 1.6.

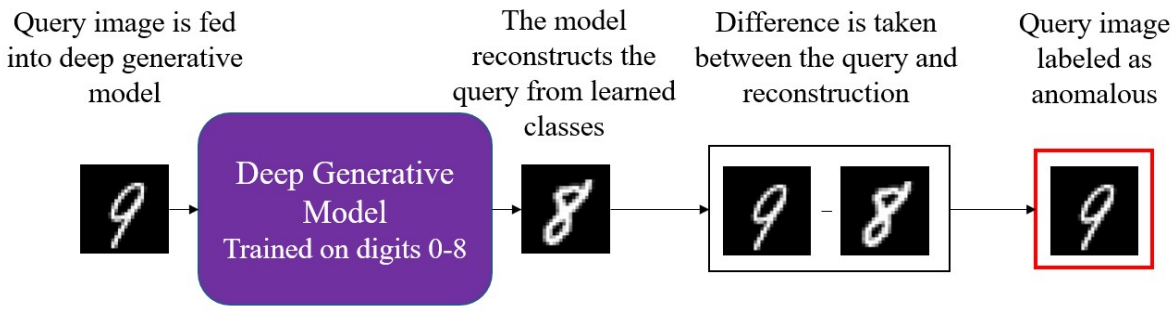


Figure 1.6: An overview of deep generative models. These models identify anomalies by taking the difference between a query image and its reconstruction. Reconstructions that vary greatly from the query are flagged as anomalous.

Two popular deep generative frameworks for learning these data representations are autoencoders (AEs) [18] and generative adversarial networks (GANs) [19]. Although these two methods have shown sufficient performance for several unsupervised anomaly detection tasks [8, 9, 11, 20, 21], the learned representation used to generate new samples takes the form as one single, continuous manifold. A manifold in this context represents a set of closely-related data points. For instance, if a dataset contains red candies, blue candies, and green candies of different sizes, then one manifold in the data is the collection of all green candies. This manifold is said to be "disconnected" from the red and blue manifolds. Treating a learned representation as a single, continuous manifold works well in cases where anomaly detection is performed on data samples that fall outside of a single learned non-anomalous class, but breakdown in the case where multiple classes are considered non-anomalous. In this scenario, the model must combine the learned representation of the separate, disconnected manifolds (representative of each class) into a single, continuous one. If an anomaly shares features across manifolds of non-anomalous data, current generative models are capable of reconstructing those features, potentially resulting in a situation where the anomaly is reconstructed with a small amount of variation. If this happens, those anomalies may go undetected by the model, which is a problem. Figure 1.7 highlights this possibility for both AE and GAN models. Developing a deep generative model that can account for anomalies that share features across non-anomalous training manifolds has the potential for improving the task of unsupervised anomaly detection in multi-class datasets.

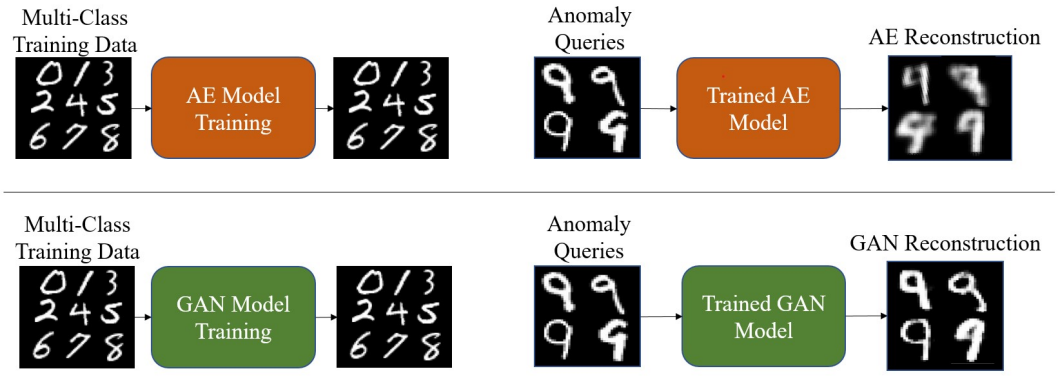


Figure 1.7: Anomalous Reconstructions for Current State of the Art. Here, two deep generative models, an AE (**top**) and a GAN (**bottom**), are trained on multiple classes (or manifolds) of data, digits 0-8. In cases where the anomalous samples share features across learned manifolds, the nine digit for example, both methods have the potential of reconstructing them, even though they were not trained to learn their data representation. This will result in the anomalous samples going undetected by the model.

The proposed approach utilizes a number of individual generators, working cooperatively in a GAN, each trained to look for particular features in the input training data. Through doing this, each generator is responsible for learning a unique manifold of data only. Two publicly-available disconnected manifold datasets are used to demonstrate this approach, where one manifold is removed during training and used as an anomaly during testing. Additionally, a new software tool is introduced for generating simulated automobile wheels that can serve as a fully controlled benchmark for anomaly detection in disconnected data manifolds. This software is used to generate a sample dataset of both non-anomalous and anomalous manifolds to demonstrate its capability on a real-world application. Through these results, it is shown that accounting for the discontinuity between manifolds helps achieve better anomaly detection performance over other state of the art methods.

In summary, the main contributions of this work are as follows:

- Present a novel GAN architecture for unsupervised anomaly detection within disconnected data manifolds.
- Demonstrate that this proposed model achieves better anomaly detection performance on disconnected manifold datasets compared to other state of the art methods.
- Present a new software tool used to produce fully controllable, disconnected manifold datasets for benchmarking multi-class anomaly detection models.

# Chapter 2

# Neural Network Architectures

## 2.1 Convolutional Neural Networks

For machine learning tasks using computer vision, computers are trained to interpret and understand the visual world through data in the form of digital images, allowing them to "see" similar to the way humans do. When training a neural network to perform these tasks using image-based datasets, deep generative methods are often used [22], which learn a representation of non-anomalous data to generate new, similar data points not presented within a training set. A popular neural network framework used for deep generative methods are convolutional neural networks [23]. In convolutional neural networks, the basic neural network architecture is altered to a matrix grid for information passing. This allows matrix-based calculations to be performed on an input image, which is treated as a matrix, where every matrix index is a pixel-value of the image. These matrices can either take a two-dimensional shape for gray-scale images, or three-dimensions for RGB images, where each matrix layer is associated with a respective color plane. When using these networks to train deep generative models, two processes are followed [24]. First, a convolutional network is used to reduce the dimensionality of the data, commonly referred to as down sampling, into a low-dimensional representation of the data, where important features of the data are learned. A second convolutional network then takes this low-dimensional representation and expands its dimensionality, known as up sampling, back to a fully-dimensional reconstruction of the input image. An overview of this network structure is shown in Figure 2.1.

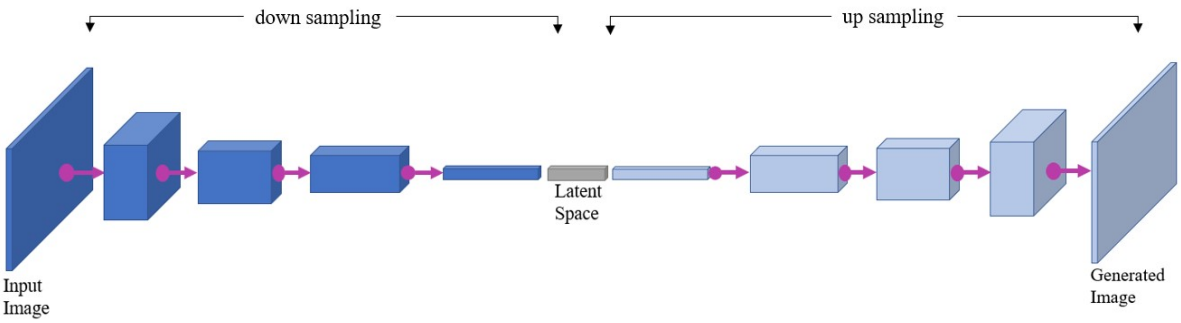


Figure 2.1: Simple Convolutional Neural Network. When training a deep neural network on datasets composed of images using generative methods, convolutional neural networks are often used. These networks first down sample an input image into a low-dimensional representation, referred to as a latent space, then up sample that low-dimensional representation into a generated reconstruction of the input.

When an image is down sampled into its low-dimensional representation, the goal is to reduce the data sample into a learned set of important features, commonly referred to as a "latent space" representation. This operation is achieved by passing the image through a series of convolutional layers. When the latent space is obtained, a full-sized image is simplified into a set of features that define the image. For example, latent spaces in facial recognition algorithms might use the length and angle of connecting lines between facial features to identify a person's face from trained images.

In each convolutional layer, a series of filters moves across the image, which are responsible for extracting different image features. The filters themselves are uniquely-weighted matrices that apply a matrix multiplication operation to the image's pixel values, resulting in a dimensionality reduction of the image between each layer. The weights of these filters are updated during training to optimize down sampling performance. The dimensionality of the image is reduced using a set of hyperparameters associated with the convolutional operation: filter size, stride, and channel size [25]. The filter size dictates the size of the matrix moving along the image, and controls the dimension of the convolutional output. The stride refers to the step size taken by the filter as it moves across the image, therefore also controlling the dimension of the convolutional output. The channel size is the number of filters applied to the image for a given convolutional layer, where each filter is responsible for detecting a unique feature within the image. The initial layers of convolutional neural networks are responsible for extracting high-level features of the image, so fewer filters (and a lower channel size) are needed. As the dimensionality of the data is reduced between layers, more filters are necessary to pass extracted features along to the latent space. Typically, the channel size is increased by a factor of two between each convolutional layer. Figure 2.2 serves as a visualization aide for these convolutional operations. In this example, a 4x4 resolution data sample is fed into a convolutional neural network, which

has a filter size of two, a stride of one, and a channel size of four. The 2x2 filter moves across the image from left-to-right and top-to-bottom, applying a matrix multiplication operation between the pixels and filter values, which are then added together. A 3x3 matrix is outputted at the end of the convolution. This process is performed a total of four times, resulting in a 3x3x4 data matrix as final output. Before the convolutional process is complete, a linear transformation is performed on the final convolutional output, changing the dimensionality of the data from a matrix to a one-dimensional latent vector representation.

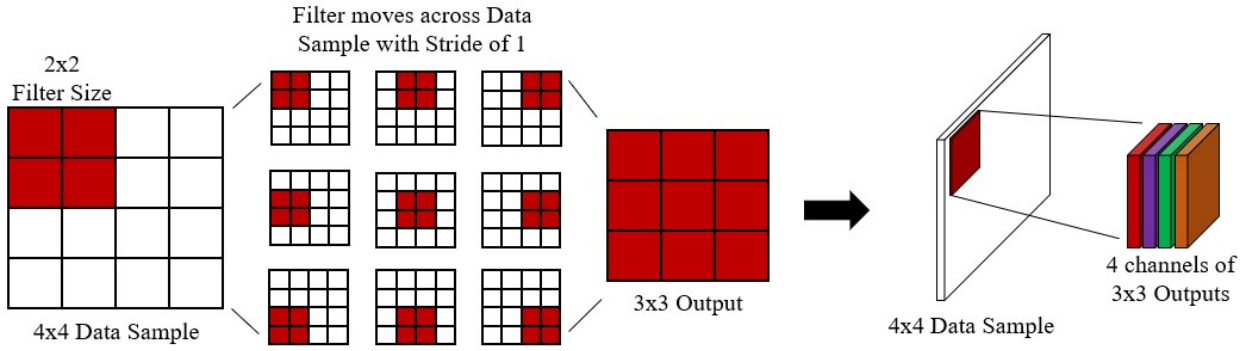


Figure 2.2: Convolutional Operation on Data. A 4x4 resolution image is passed into a convolutional neural network, which has a filter size of 2, a stride of 1, and a channel size of 4. A 3x3 matrix is outputted at the end of the convolution. This process is performed a total of four times, resulting in a 3x3x4 data matrix as final output.

For up sampling an image from a low-dimensional representation to a image reconstruction, the same ideology applies, but instead by performing deconvolutions on the data between each layer. For each deconvolution, the data dimensionality is *increased* between the input and output. Channel size is also typically reduced by a factor of two between each deconvolutional layer. After the deconvolutional process is complete, the image reconstruction has the same dimensionality as the original image.

As a data sample travels through convolutional/deconvolutional network layers, activation functions are applied to each matrix index of the layer's output. These activation functions take each index value and decide whether or not to "fire" the value in an unchanged state to the next layer. Several different activation functions are used within convolutional neural networks today, including but not limited to sigmoid, tanh, and Rectified Linear Unit (ReLU) [26].

## 2.2 Deep Generative Methods

Deep generative methods learn a representation of non-anomalous data to generate new, similar data points not presented in a training dataset. For computer vision tasks, these frameworks learn the representation of an image dataset, where the model is used to reconstruct query images, or images not present within the training set, as output. For anomaly detection, this reconstruction is used to determine whether or not a query sample is anomalous.

Two of the most commonly used deep generative methods are autoencoders (AEs) and generative adversarial networks (GANs). An autoencoder consists of two neural networks, an encoder and decoder, to perform down sampling and up sampling processes, respectively. During training, the encoder network down samples images into a low-dimensional representation, which stores latent information about the input data distribution. The decoder is responsible for taking this low-dimensional representation and up sampling it into a reconstruction of the input sample image into the encoder. This model is illustrated in Figure 2.3.

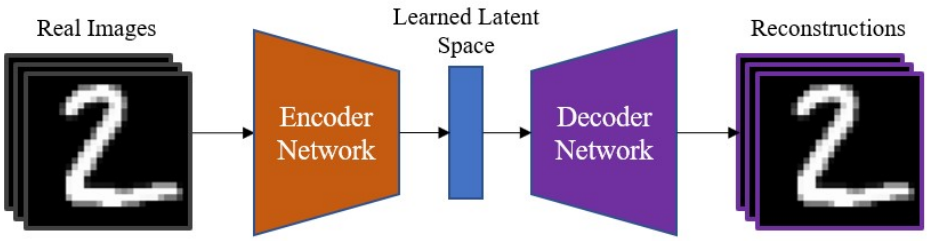


Figure 2.3: Illustration of an autoencoder architecture. These architectures are composed of two networks, an encoder and decoder. The encoder network learns to encode real image samples into low-dimensional representation. The decoder is responsible for taking this latent representation and up sampling it into a reconstruction of the real image.

The training objective for an autoencoder is to minimize the difference between real input images and their reconstructions, or reconstruction error. In order to update the weights of both the encoder and decoder networks, the mean-squared error between the real sample input and its reconstruction is used. Once training is complete, the model should be capable of reconstructing never before seen non-anomalous data samples with a high level of accuracy. Building off of this traditional autoencoder, variational autoencoders (VAEs) [27] are autoencoders whose encoding distribution is regularised during training in order to ensure that its latent space has good properties that allow for continuous generation of new non-anomalous data samples never seen during training.

Similarly, GANs consist of two neural networks, a generator and a discriminator, that play against each other in an adversarial game. During training, the generator's goal is to produce data samples that fool the discriminator into thinking they're real, while the discriminator's goal is to distinguish generated (fake) data from real data. This model is illustrated in Figure 2.4.

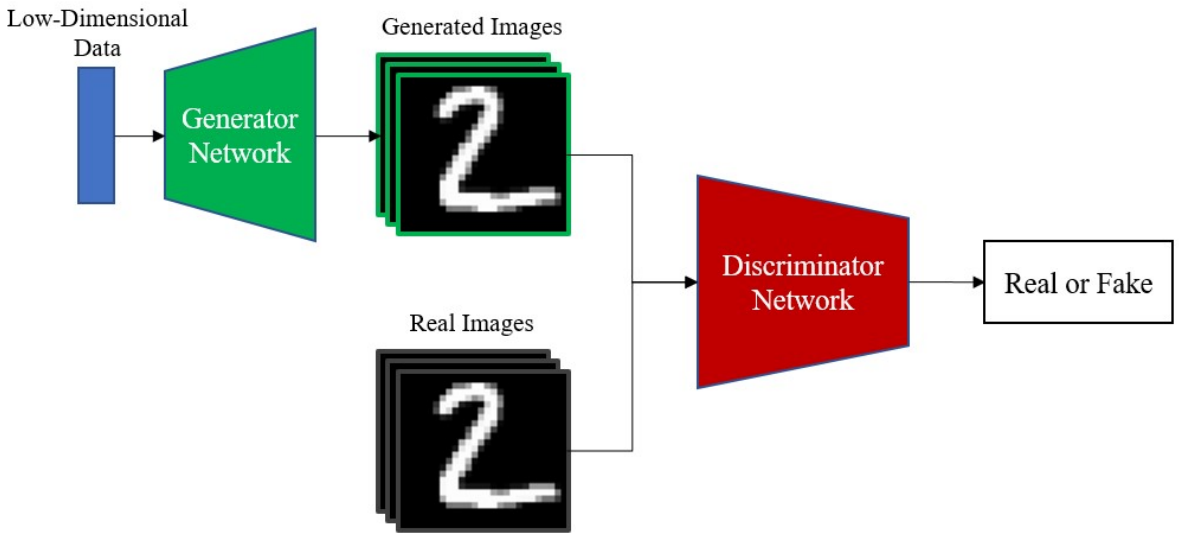


Figure 2.4: Illustration of generative adversarial network architecture. These architectures are composed of two networks, a generator and a discriminator. The generator's goal is to take low-dimensional data and produce images that fool the discriminator into thinking they're real. The discriminator's goal is to distinguish generated (fake) data from real data.

Like the decoder network of an autoencoder, the generator network of a GAN takes lower-dimensional data as its input and outputs a full-sized image reconstruction. The discriminator takes as input both generated image samples from the generator network and real image samples, and outputs a classification of those inputs as either being a generated or real image sample. During training, the generator network is updated by taking the average classification output of generated image samples by the discriminator network. The discriminator is updated by taking the average classification output of real and generated image samples. By the end of training, the goal is for the generator to produce samples that resemble real images well enough that the discriminator is incapable of distinguishing between real and fake data.

In order to apply GAN networks for the task of anomaly detection, input image samples must be down sampled into a low-dimensional latent representation for the generator network to then up sample into a reconstruction of the input [28]. Because of this, anomaly detection-based GANs require an encoder network added the model framework for down sampling of data, similar to an AE. GAN models of this type are illustrated in Figure 2.5. During training, the encoder network is updated based on the reconstruction of real images, using the mean-squared error between real image inputs and their reconstructions from the generator network. With this evolved framework, GAN models can then be trained to generate samples of non-anomalous data. When these trained models are fed both non-anomalous and anomalous data samples during testing, the difference between the input sample query and its reconstruction are used to determine whether or not the sample is anomalous.

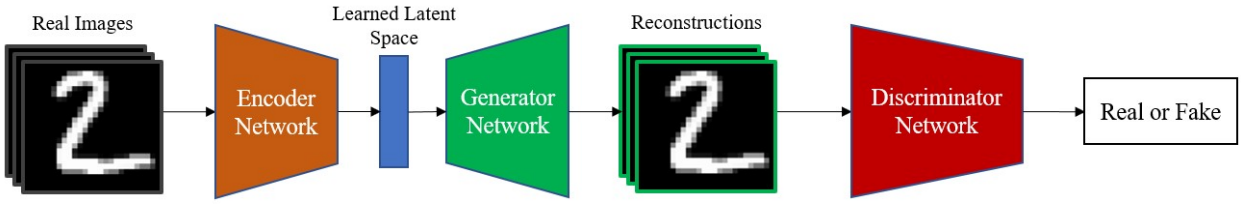


Figure 2.5: Illustration of generative adversarial network architecture used for anomaly detection. These architectures require the addition of an encoder network to down sample real image samples into a learned latent representation for reconstruction from the generator network.

A final important point to address when discussing GAN models is taking into consideration the recent improvement made to the training process by [29], referred to as a Wasserstein GAN. The Wasserstein GAN, or WGAN, is an extension of the traditional GAN framework, and has been shown to improve training and produce better quality images. Instead of using a traditional discriminator to classify the image as real or fake, the WGAN implements a critic network that scores the"realness" or "fakeness" of an image. This change is driven by the argument that training the generator should seek to minimize the distance between the distribution of real data samples in the training dataset and the distribution of generated samples. This work adopts the critic method used in the Wasserstein GAN model. However, with this distinction in mind and for the sake of consistency to other work, the critic network will continue to be referred to as the discrimnator throughout the rest of this thesis.

Now, with an overview of deep generative methods in mind, a discussion on how they have been applied for anomaly detection with current state of the art will now be examined.

# Chapter 3

# Related Work

## 3.1 One-Class Anomaly Detection

Anomaly detection tasks fall into one of two categories: one-class and multi-class. In one-class anomaly detection, models are trained on a single class of data, commonly referred to as a manifold, with all data points outside of that learned manifold considered anomalous. In recent years, several deep-generative methods have shown to be effective at one-class anomaly detection. In [30], the standard GAN encoder and generator networks are combined into a single network, referred to as the refinement network, that learns the single input class. During training, an input noise distribution is added to the data to refine non-anomalous data and distort anomalous data.

In [31], the traditional GAN framework is modified to focus training within the latent space outputted by the encoder network specifically, constraining it to exclusively represent the non-anomalous class. This work also presents a dual-discriminator framework, where both discriminators are adversarially trained using the same latent space. One of these discriminators is used to ensure that encoded representations of in-class examples resemble uniform latent-sized random samples, while the other ensures that randomly drawn latent samples generate real-looking data.

In [32], an AE model is equipped with a parametric density estimator that learns the probability distribution of its latent representations through an autoregressive procedure. ConAD implements a multi-hypothesis AE for one-class anomaly detection, with a discriminator network mimicing a GAN discriminator [33]. Their method splits the final layer of the decoder into different hypothesis clusters, optimizing one cluster for non-anomalous data, with the remaining detecting anomalies.

Although these methods perform well at detecting anomalies outside of a single learned class, many real-world datasets are comprised of multiple manifolds and require frameworks that take this distinction into account.

## 3.2 Multi-Class Anomaly Detection

In multi-class anomaly detection, models are trained on a dataset containing multiple disconnected (or separate) manifolds, with all data points outside those manifolds considered anomalous. In recent years, variations of GANs traditionally used for one-class classification have been reconfigured for use in multi-class anomaly detection. This has been achieved by increasing the convolutional depth of the generative network, which helps the generator learn the combined feature set across all non-anomalous classes.

The first model of this type is AnoGAN [8], which adversarially trains a generator to map a uniformly sampled latent space to the distribution of training data. At test time, back propagation is used to find the generator's closest reconstruction of the query image using the reconstruction loss as the anomaly score. By adding a secondary training process that trains an encoder to be the inverse of the generator, the time consuming back propagation was reduced significantly through development of a fast version of AnoGAN called f-AnoGAN [9]. In this version, the query image is passed through the encoder to generate the latent vector which is then passed through the generator to create a reconstruction of the image. The image reconstruction error along with a mean squared error of an intermediate discriminator layer is used for classifying samples as non-anomalous or anomalous.

BiGAN [20] has also been used for multi-class anomaly detection. In this model, the generator is trained similarly to a standard GAN architecture with an encoder added that learns the inverse mapping of the generator. This is done by passing the discriminator both the image and the latent vector for both generated images and encoded training images. At test time the query image is passed through the encoder to produce the latent-sized vector, which is then passed through the generator to create a reconstruction.

GANomaly [13] takes the traditional GAN framework and adds an additional encoder into the pipeline, resulting in a encoder-generator-encoder architecture. The discriminator is then trained using output data from both the generator and the second encoder. Recently, a "skip connection" version of the model [34] was developed, which uses the output of an individual convolutional layer in the encoder network as an additional input into the mirrored deconvolutional layer in the generator network [35]. This configuration resulted in improved model performance in some applications of anomaly detection.

When trained on disconnected manifold datasets, these methods do not explicitly learn the unique features of individual manifolds separately, instead collapsing the representation into a single, continuous manifold. Figure 3.1 helps illustrate this point in the case where a model is tasked with learning nine classes of hand-written digits 0-8. Here, each colored point within the latent space is associated with a unique data manifold. Data points from different manifolds share boundaries with each other, and some data points from a particular manifold even share the same "area" of latent space with other manifolds. This learning framework can be problematic when anomalies share features found across unique manifolds that represent individual learned classes. In this case, anomalous classes can be falsely considered as learned and labeled as non-anomalous.

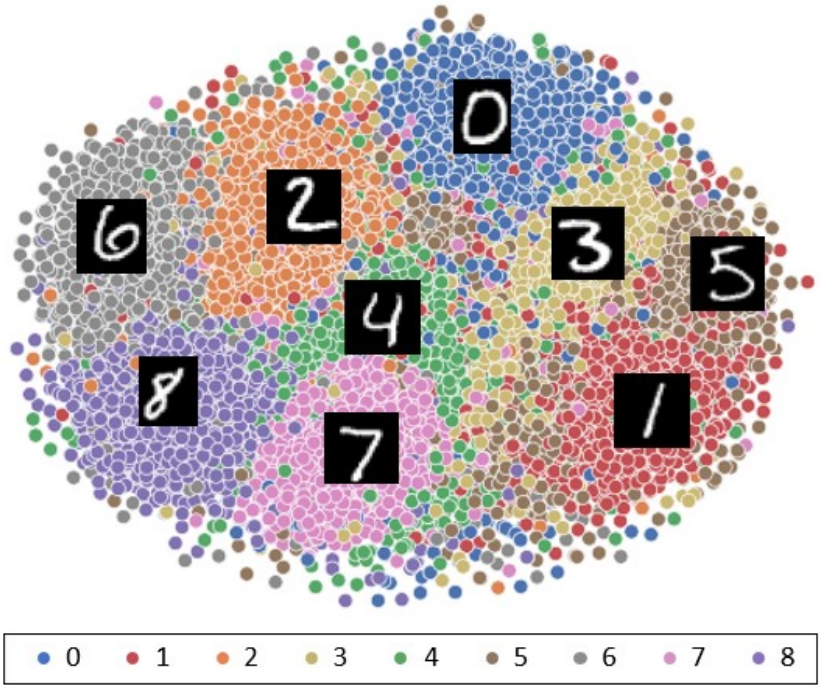


Figure 3.1: Example Latent Space for Current State of the Art. During training, current state of the art deep generative methods treat the learned representation (or latent space) of all disconnected manifolds as a single, continuous one. This is highlighted in the example where a model is trained on a dataset of digits 0-8. Anomalous samples that share features across this latent space may then go undetected during testing.

## 3.3 Accounting for Disconnected Manifolds During Training

Two recent advances in GAN models attempt to address the discontinuity in manifolds during training through clustering (or grouping) data points of the same manifold together. The first model is ClusterGAN [36], which clusters latent-space representations of disconnected manifolds (i.e. each class) during training. This architecture also adds an encoder network to the GAN framework, which works alongside the generator in adversarial training by aiding in the latent-space clustering task.

The second model is Disconnected Manifold GAN with Prior Learning (DM-GAN-PL) [37], which uses a multi-generator architecture to learn each unique manifold representation independently. This model also learns a cluster prior in an unsupervised fashion. Similar to ClusterGAN, a classifier network is used to learn the cluster association to a given manifold sample and is trained through penalizing the generator for mapping multiple generated manifolds to one single cluster. Additionally, a set of parameters are added to learn the clustering distribution, called priors, by training off the classifier's estimate of the real data cluster distributions. By learning the prior, the model is unsupervised, as no cluster prior is required for training.

Although these models *do* account for potential discontinuity between training manifolds, neither are capable of performing anomaly detection. Building on these methods, the goal of this work, illustrated in Figure 3.2, is to develop a novel GAN-based anomaly detection model through clustering of disconnected manifolds using multiple generators.

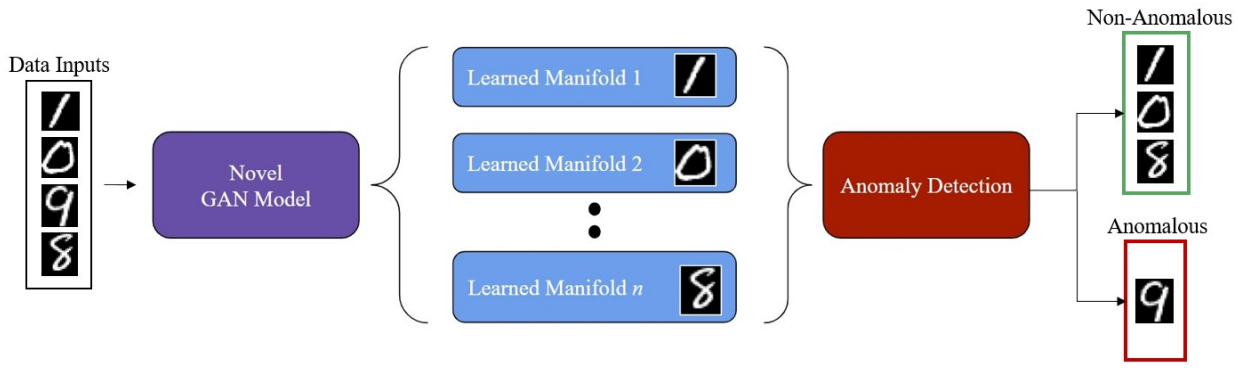


Figure 3.2: Notional Overview of Research Goals. This work aims to create a novel GAN model that learns each manifold representation independently from each other through clustering in order to increase anomaly detection performance. The goal is for the novel model to detect anomalous samples (the nine digit in this example) within a batch of non-anomalous samples (digits zero, one, and eight in this example) better than current state of the art by learning the representation of non-anomalous samples separately.

# Chapter 4

# Proposed Model: D-AnoGAN

## 4.1 Model Architecture

The main contribution of this thesis is the novel anomaly detection deep learning network architecture, Disconnected Anomaly GAN. Disconnected Anomaly GAN, or D-AnoGAN, consists of four convolutional neural networks: a multi-generator, a multi-encoder, a discriminator, and a classifier. It also includes a mechanism, referred to as a bandit, to discover the optimal set of generators (and thus encoders) to cover all manifolds within a given dataset. An overview of the proposed architecture is shown in Figure 4.1. An in-depth explanation of each model component is provided in the following pages.

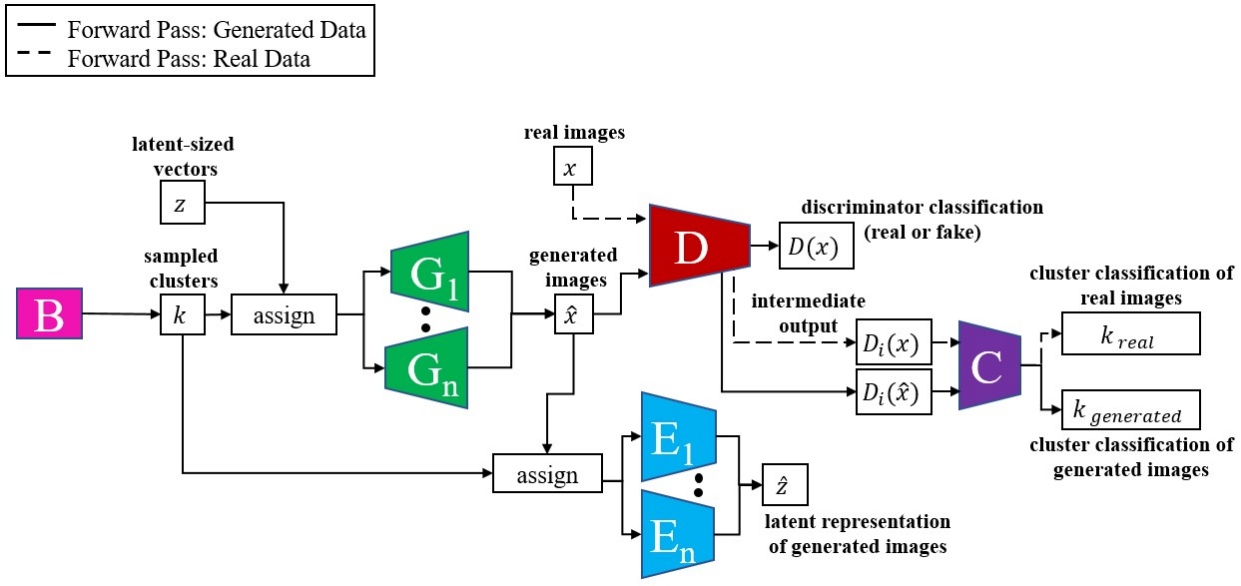


Figure 4.1: Model Architecture for D-AnoGAN. The model consists of four convolutional neural networks, a multi-generator $G$, a multi-encoder $E$, a discriminator $D$, and a classifier $C$, as well as a bandit $B$.

### 4.1.1 Multi-Generator

As previously mentioned, current state of the art GAN models learn multi-class datasets with a single generator network. To account for discontinuity between classes (or manifolds), a multi-generator network, $G$, is implemented. This network contains $n$ generators, where $n$ is a hyperparameter assumed to be larger than the number of manifolds within the dataset. Each generator is responsible for learning one unique data manifold, which is performed through clustering latent representations of data manifolds together and training a generator off of a single cluster. To train the multi-generator, two main steps are followed. First, the network is fed a latent vector $z$ and a cluster number $k$, where $z$ is sampled from the uniform distribution $U(-1, 1)$, and $k$ is sampled from the bandit. A cluster number $k$ is associated with a single latent representation of a manifold. The multi-generator then uses $k$ to select a generator to map $z$ into a generated sample, $\hat{x}$, that is representative of a real manifold in the non-anomalous data space $x$. These two processes are illustrated in Figure 4.2

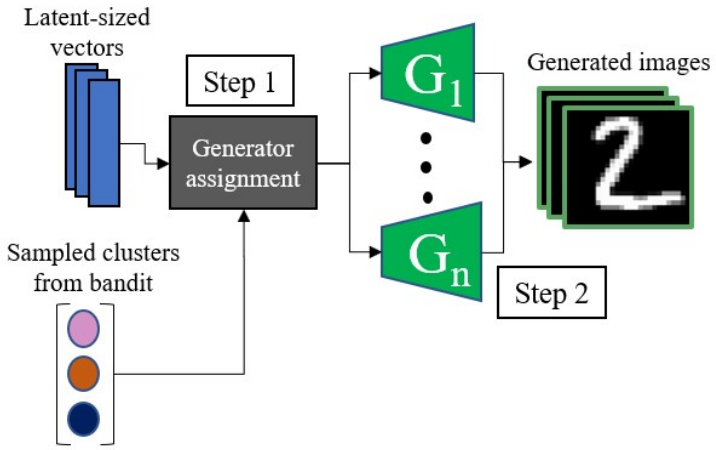


Figure 4.2: The multi-generator network. This network consists of $n$ generators, each responsible for learning a unique manifold of data. To train this network, latent-sized vectors are fed into an individual network assigned by sampled clusters from the bandit. The network outputs generated images.

At the beginning of training, there is a uniform probability of each cluster being associated with a manifold in the dataset. As training progresses and generators begin learning manifold representations, the probability of clusters associated with a unique manifold increase increase, while the probability of clusters with no manifold association are reduced to zero. By the end of training, the goal is for each manifold of data to be associated with a unique cluster number and a single generator, and for extra generators unassociated with a unique manifold to be shutdown, or removed from the model, by the bandit. An example of this process is shown in Figure 4.3.

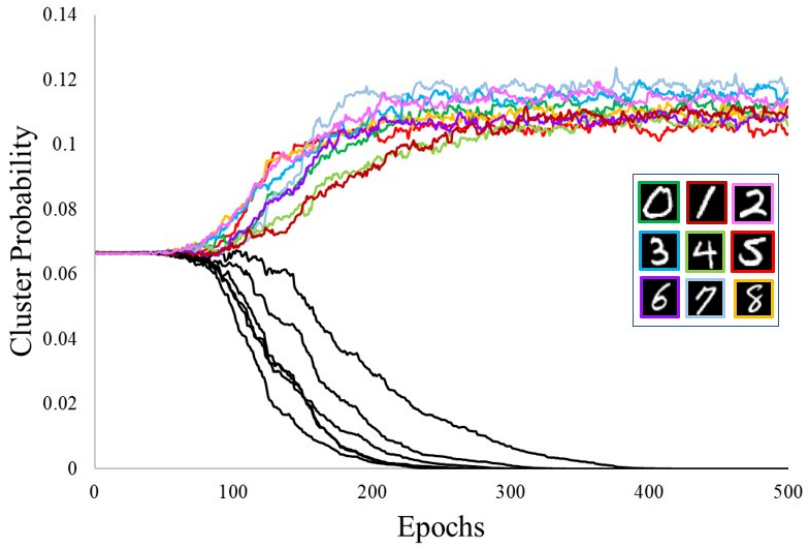


Figure 4.3: Example model clustering performance. As generators learn manifold representations during training, the probability of clusters associated with a unique manifold increase increase, highlighted in color. The probability of clusters with no manifold association are reduced to zero (highlighted in black), with the respective generators for those clusters shut down from training. By the end of training, all data manifolds (digits 0-8 in this example) are successfully clustered and separated, with excess generators shutdown by the bandit.

The output from the multi-generator, $\hat{x}$, is fed to the discriminator, $D$, where an intermediate discriminator layer output, $D_i(\hat{x})$, is fed to the classifier, $C$, for cluster-classification of $\hat{x}$. Using this output from the discriminator and classifier, the generator loss, $l_G$, is calculated as:

$$l_G = -\mathbb{E}[D(\hat{x})] - \lambda l_C, \tag{4.1}$$

where the first term of the generator loss is the Wasserstein GAN loss [29], and the second term is the classifier loss defined in Eq. (4.5) with a penalty coefficient $\lambda$. The Wasserstein GAN loss uses the discriminator output to penalize generators for outputs that do not belong to the real data manifolds. It also penalizes the multi-generator for not collectively mapping the full set of real data manifolds in $x$. The classifier loss, explained in the classifier section, is shared with the generator to prevent multiple generators from learning the same manifold of non-anomalous data in $x$.

### 4.1.2 The Bandit

The bandit, B, inspired from [37], is a set of parameters equal in size to $n$ that is used to discover the optimal set of generators required to cover all data manifolds within a set. The bandit does this by first sampling clusters based on their probability of manifold-association, which are then used to assign latent-sized vectors into the multi-generator network. By the end of training, the goal is for a one-to-one mapping betwen clusters and data manifolds. An overview of the bandit is illustrated in Figure 4.4.

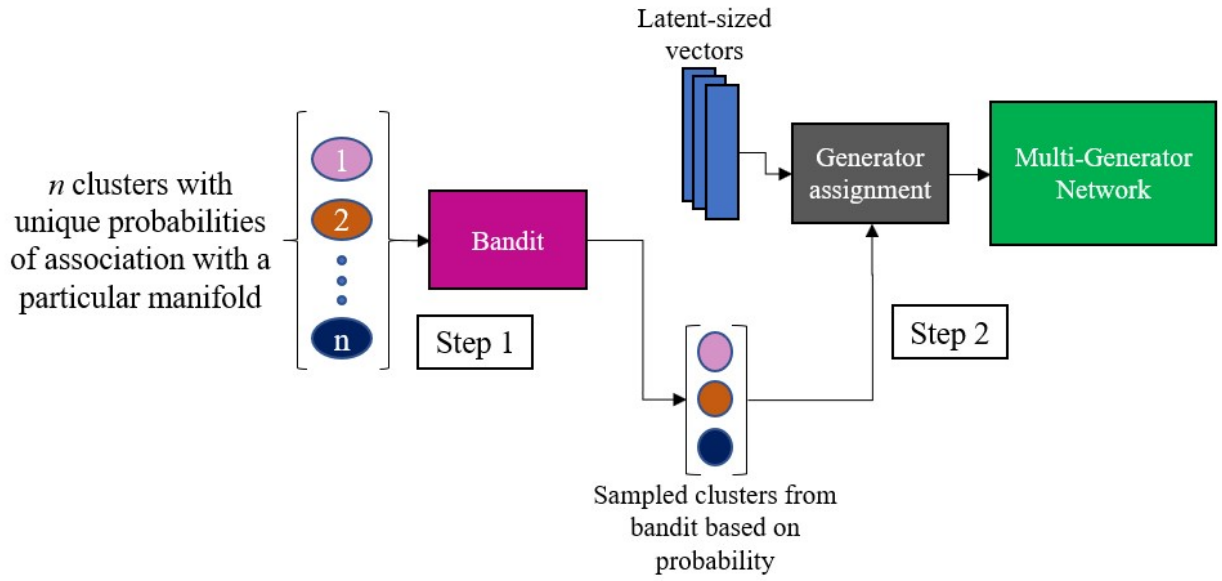


Figure 4.4: Overview of the bandit. The bandit samples from $n$ clusters used by the multi-generator based on their probability of being associated with a manifold of data. This probability is updated throughout training, as an optimal subset of clusters corresponding to each manifold of data is discovered by the bandit. By the end of training, the goal is for a one-to-one mapping between clusters and data manifolds.

In order to shutdown extra generators in the multi-generator, the bandit network must learn the correct distribution between training data manifolds and optimal generators, known as the "prior". This is achieved using an expectation maximization (EM) method where the approximation of the prior is the expected value outputted by the classifier network, which will be discussed in Section 4.1.5. Given sufficiently large batches of real data $x$, the expected value of the classifier's output, $C(x)$, is a good approximation of the prior.

For the EM method, the multi-generator must already have learned a mapping from the latent space $Z$ to all data manifolds in the input space $X$ and the classifier must already be trained to predict cluster numbers. The classifier's output can not be used to estimate the prior at the start of training. Instead, the bandit starts with a uniform prior and an entropy regularization loss term that maximizes the entropy of the estimated prior. This entropy regularization term keeps the estimated prior close to uniform at the beginning of training to allow the generators to learn to generate data on the real data manifolds and the classifier to predict cluster numbers before the bandit starts to turn generators off. The bandit is updated using the following loss, $l_B$:

$$l_B = \mathbb{E}[H(B(k), C(D_i(x))] - \beta^t \nu H(C(D_i(x)), \tag{4.2}$$

where the first term is the cross entropy between the current approximation of the prior, $B(k)$, and expected value of the classifier's output given the training data. The second term is the entropy regularizer term using the bandit's estimate of the prior. For the regularizer, $\beta$ is a decaying constant for the entropy regularization, $t$ is the training step, and $\nu$ is a regularizer loss coefficient.

### 4.1.3 Multi-Encoder

For anomaly detection, a multi-encoder network is used to encode query images $x_{query}$ to the multi-generator's latent space $Z$. This network is illustrated in Figure 4.5. Similar to the multi-generator, the multi-encoder, $E$, is initialized with $n$ encoders. During training, two steps are followed. First, the network takes as input generated data $\hat{x}$ from the multi-generator, and the sampled cluster $k$ from the bandit. The cluster number is then used by the multi-encoder to select which encoder to use to encode $\hat{x}$ as $\hat{z}$. As training progresses, encoders paired with shutdown generators are also shutdown. The multi-encoder is trained using a mean squared error loss, $l_E$, that compares the original latent vector to the encoded latent vector, calculated as:

$$l_E = \lambda \mathbb{E}[(z - \hat{z})^2]. \tag{4.3}$$

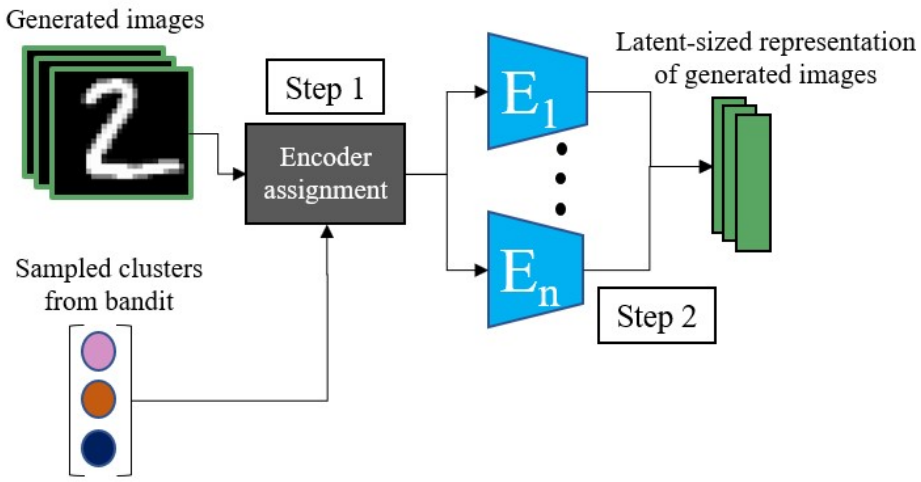


Figure 4.5: The multi-encoder network. This network consists of $n$ encoders, each responsible for learning a unique latent-representation associated with a single manifold of data. To train this network, two steps are followed. First, generated image samples outputted by the multi-generator are fed into an individual network assigned by sampled clusters from the bandit. The network then down samples the generated images and outputs latent-sized representations of them.

### 4.1.4 Discriminator

The discriminator, $D$, is tasked with distinguishing between real data $x$ and generated data $\hat{x}$. To train the discriminator, two steps are followed. First, the discriminator is fed both real data, $x$, and generated data from the multi-generator, $\hat{x}$. The discriminator then down samples this data into a classification of being either "real" or "fake". This network is visualized in Figure 4.6.

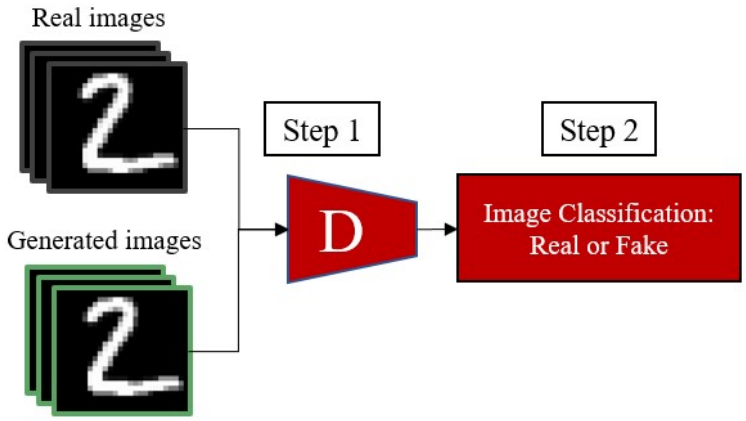


Figure 4.6: The discriminator network. When a generated image is classified is fake, the generator network is penalized. By doing this, the discriminator trains the generator network to produce image samples that closely represent real data. Once this is achieved, the discriminator network will not be able to distinguish between real data and fake data.

The discriminator's objective is to maximize the Wasserstein Distance [38], a measure of the distance between two probability distributions, between its output for real and generated data. Additionally, the standard gradient penalty is added to enforce a 1-Lipschitz constraint with a penalty coefficient $\lambda$ [29, 39]. This constraint ensures that the gradient of discriminator output on generated samples is close to one. A gradient of one implies that normal and generated samples closely resemble each other. The loss function, $l_D$, is summarized below:

$$l_D = \lambda \mathbb{E}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2] - \mathbb{E}[D(\hat{x})] - \mathbb{E}[D(x)], \tag{4.4}$$

The discriminator network is the most important component of D-AnoGAN. If this network is not able to classify poorly generated samples from the multi-generator as fake, the quality of reconstructions can be greatly diminished. If this were to happen, the likelihood of anomalous samples being detected during testing can drop, due to the fact that the difference between non-anomalous queries and their reconstructions could also be large.

### 4.1.5 Classifier

The classifier, $C$, is a single-layer network, responsible for learning the cluster association for a given manifold of generated images $\hat{x}$, similar to methods used in [37]. The classifier trained using two steps. First, the network is fed an intermediate output from the discriminator, $D_i(\hat{x})$, associated with $\hat{x}$. The intermediate discriminator output is then used by the classifier to predict the cluster number $k$. This network is illustrated in Figure 4.7.

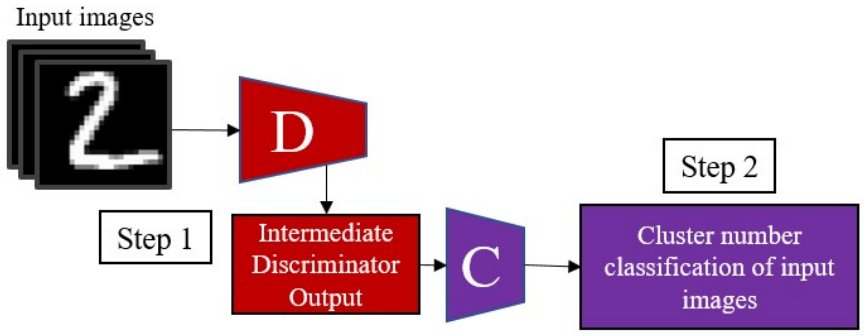


Figure 4.7: The classifier network. responsible for learning the cluster association for a given manifold of generated images. In order to do this, it takes an intermediate output from the discriminator network as input, and outputs a distribution of cluster association corresponding to the image fed into the discriminator.

By using this input, instead of $\hat{x}$ directly, over fitting a unique manifold to multiple clusters from the classifier is prevented as this requires it to use latent features learned by the discriminator. This is important as the classifier loss is used during training to shut down generators mapping to the same data manifold; this process would be hindered by an overfitting classifier. The classifier is also required for anomaly detection as the multi-encoder and multi-generator need a predicted cluster number $k$ associated with $x_{query}$. The classifier is trained using the cross-entropy loss between the predicted cluster number and the ground truth cluster number $k$, calculated as:

$$l_C = -\mathbb{E}[H(k, C(D_i(\hat{x}))], \tag{4.5}$$

## 4.2 Model Training

D-AnoGAN combines these model components to generate disconnected data manifolds, while also learning the inverse mapping of the generator's input and the cluster prior distribution. During every training epoch, each mini-batch of data is used to train the discriminator. The multi-generator, multi-encoder, classifier, and bandit are trained every five iterations of discriminator training. Once initial training is complete, a second training step is performed, referred to as $xzx$-training. This training step, inspired by [9], locks the network parameters of the multi-generator, discriminator, and classifier networks, where no additional weighting updates take place. Additional training is then performed on the multi-encoder using real data $x$. The overall training algorithm for D-AnoGAN can be found in Appendix A.

During $xzx$-training, the multi-encoder is updated by comparing real data, $x$, to its reconstruction, $\hat{x}$, outputted by the multi-generator. The pipeline for xzx-training follows seven steps. First, a query image is fed through the trained discriminator network, where the intermediate output layer from it is passed through the trained classifier network. The model then moves onto the second phase, where the classifier network will output a cluster classification for the query image. In the third phase, the query cluster label outputted from the classifier is used to assign the query image into its corresponding encoder within the trained multi-encoder network. Next is the fourth phase, where the multi-encoder down samples the query into a latent representation. During the fifth phase, the latent representation vector outputted from the multi-encoder is then passed through the trained multi-generator network, where the classifier label is used again to assign the latent vector to a corresponding generator. The multi-generator network up samples the latent vector into a reconstruction of the query image for the sixth phase. Finally, the difference between the query image and reconstruction image is then used to update the multi-encoder network. The xzx-training pipeline is visualized in Figure 4.8.

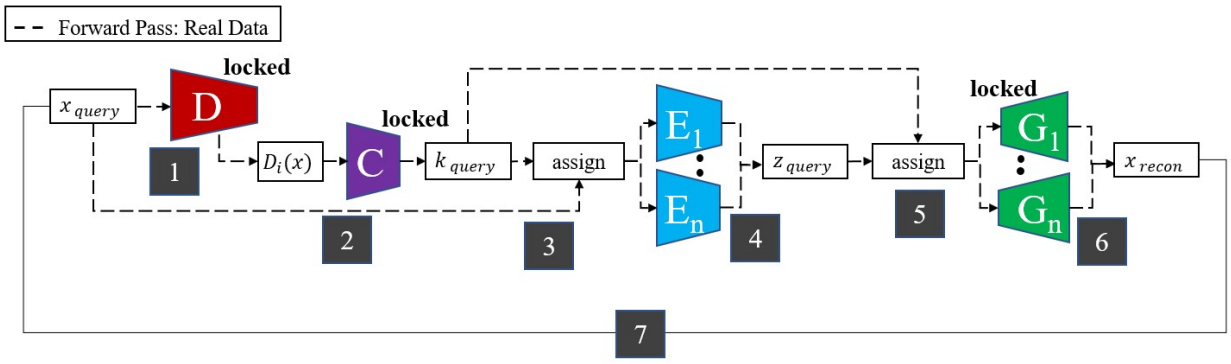


Figure 4.8: Xzx-Training Pipeline. During xzx-training, the model parameters for the multi-generator, discriminator, and classifier networks are locked, where no additional weighting updates take place. Further training updates are then performed to the multi-encoder network by comparing the real data, $x_{query}$, to its reconstruction, $x_{recon}$.

It was found that this additional training reduces reconstruction error of real samples, increasing anomaly detection performance in all tested datasets. An example of this increased performance is shown in Figure 4.9. Here, a non-anomalous query sample is fed into D-AnoGAN, where it is reconstructed as output. The pixel-level difference between the input and output (visualized here as a heat map) is used to show the accuracy of the non-anomalous reconstructions, which is important during anomaly detection testing. For this heat map, large pixel differences are highlighted in red, and low pixel differences are highlighted in blue. It can be seen in this figure that reconstructions of a query image have a lower pixel-level difference with additional xzx-training.

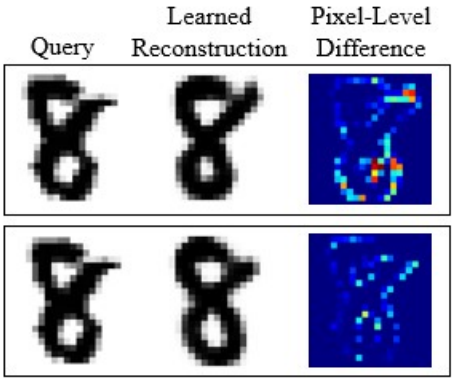


Figure 4.9: Impact of Xzx-Training on Sample Reconstruction. From this figure, it is clear that reconstructions from D-AnoGAN without additional xzx-training (**top**) have a higher pixel-level difference than those with xzx-training (**bottom**).

## 4.3 Model Testing

Once the initial training and xzx-training phases are complete, anomaly detection testing can be performed. This process is broken up into seven main phases. In the first phase, a query image is fed through the trained discriminator network, where the intermediate output layer from it is passed through the trained classifier network. Next, the classifier network will output a cluster classification for the query image. In the third phase, the query cluster label outputted from the classifier is used to assign the query image into its corresponding encoder within the trained multi-encoder network. The multi-encoder then down samples the query into a latent representation. This latent representation vector then outputted from the multi-encoder is then passed through the trained multi-generator network, where the classifier label is used again to assign the latent vector to a corresponding generator. The multi-generator network then up samples the latent vector into a reconstruction of the query image during the sixth phase. Finally, the difference between the query image and reconstruction image is then used to assess whether or not a sample is anomalous. A visualization of the anomaly detection testing pipeline is shown in Figure 4.10.

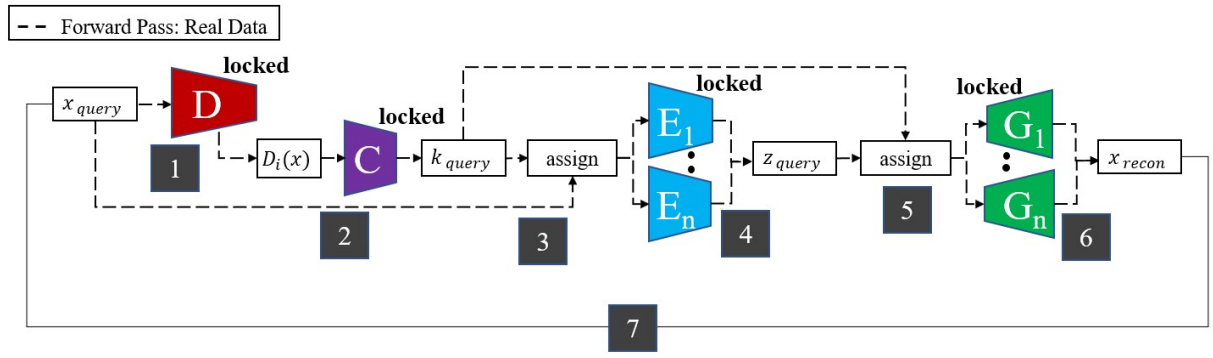


Figure 4.10: Testing pipeline of D-AnoGAN for Anomaly Detection. Here, a query image, $x_{query}$, is fed through the model, where a reconstruction of the query, $x_{recon}$, is outputted. The difference between the query image and the reconstruction is used for classifying the query as either non-anomalous or anomalous.

# Chapter 5

# Experimental Results

## 5.1 Datasets

D-AnoGAN's anomaly detection performance is tested against other state of the art anomaly detection methods using two publicly available multi-class datasets, MNIST [40] and Fashion-MNIST [41], as well as a third, novel, custom-generated disconnected-manifold dataset, Wheel. A brief overview of all three datasets is provided below.

### 5.1.1 MNIST

The MNIST dataset is composed of ten classes of 32x32 resolution, gray-scale images of handwritten digits 0-9. For testing on MNIST, a total of ten tests were performed, where for each test, one of the ten digit classes was removed from the training set and used as an anomalous class during testing. 80% of the nine classes treated as non-anomalous were randomly selected for training, with the remaining 20% used for testing. This dataset is visualized in Figure 5.1

Figure 5.1: MNIST dataset: 10 classes of handwritten digits 0-9. During training, one of these ten classes are removed and considered anomalous during testing.

### 5.1.2 Fashion-MNIST

The Fashion-MNIST dataset consists of 32x32 resolution, gray-scale images of fashion apparel and accessories, split into ten unique classes. A total of ten tests was also performed with this dataset, where each test involved removing a single class (apparel/accessory) from the training set, and considering that class as anomalous during testing. 80% of the nine classes treated as non-anomalous were randomly selected for training, with the remaining 20% used for testing. This dataset is visualized in Figure 5.2

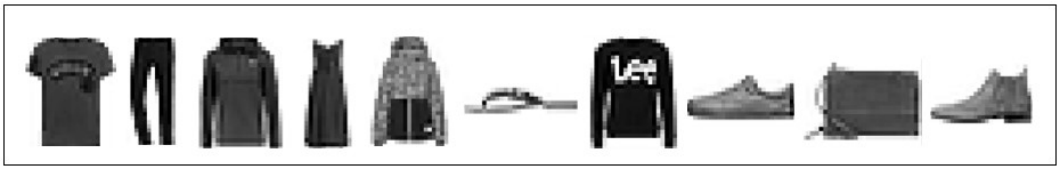

Figure 5.2: Fashion-MNIST dataset: 10 classes of fashion apparel items. During training, one of these ten classes are removed and considered anomalous during testing.

### 5.1.3 Wheel

The purpose of developing the Wheel dataset was to implement D-AnoGAN within a real-world setting of anomaly detection on images representative of wheel patterns found in the automotive manufacturing industry, with different characteristics such as number and types of spokes and bolts. The key contribution with this dataset is that the user has full control over the manifold designs being trained and tested. For example, each wheel pattern can vary the number of bolts and spokes, with those spokes being either closed or forked.

This dataset consists of eight non-anomalous manifolds and two anomalous manifolds of 256x256 resolution, gray-scale wheel images. Examples of these ten manifolds can be found in Figure 5.3. The non-anomalous manifolds contain one of four spoke patterns (three-closed, three-forked, five-closed, and five-forked) and either four or five bolt holes arrayed around a center axle. The rim and hub diameter are continuously varied within each manifold. 80% of these eight manifolds were randomly selected for training, with the remaining 20% used for testing. The two anomalous manifolds were designed to represent potential irregularities during the manufacturing process of the eight non-anomalous wheel patterns. The first consists of non-black shading along the forked-region of the spoke, representing potential surface finish irregularities of the wheel, since the spokes appear to be neither closed or forked. The second anomalous manifold contains wheels with both the four and five bolt hole patterns on the hub, which could occur in the event of a hub casting irregularity.

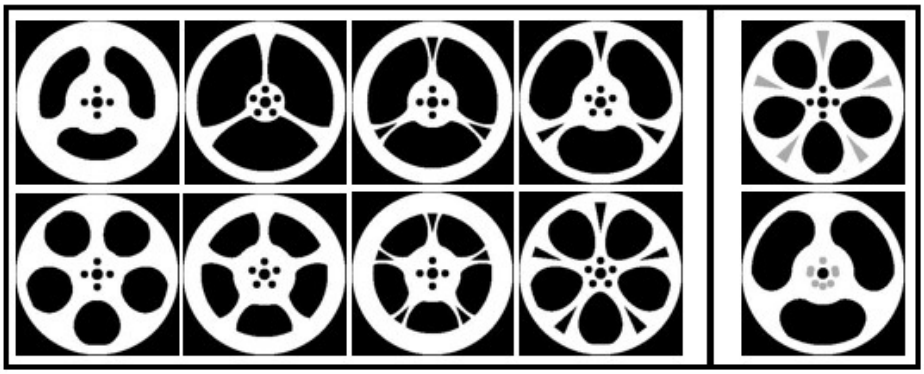

Figure 5.3: **Left:** Sample images of the eight non-anomalous wheel manifolds. **Right:** Sample images of the two anomalous wheel manifolds. The anomalies consist of low-contrast forks and a combination of the two acceptable bolt patterns.

## 5.2 Experimental Setup

The code-base used to train and test D-AnoGAN was developed using PyTorch, a python-based machine learning library developed by Facebook [42]. PyTorch provides the tools necessary to design all the neural networks used within D-AnoGAN, as well as to execute all processes throughout the entire training pipeline, including feeding image data into the model and the back-propagation steps required to update network weights. The MNIST and Fashion-MNIST datasets were obtained using the PyTorch dataset library. All model training procedures were executed using on a GPU for efficient image processing.

With each dataset, 80% of the total number of non-anomalous data images were used to train the model, with the other 20% used during testing alongside of anomalous data images. This 80%-20% split of non-anomalous data is a standard practice for training and testing machine learning models. D-AnoGAN was used to train a total of ten models for both the MNIST and Fashion-MNIST datasets, with one model associated with each removed-class. A single model was trained on the Wheel dataset, which consisted of all eight non-anomalous manifolds. The same Wheel model was tested on both anomalous Wheel manifolds.

For all three datasets, D-AnoGAN was trained to a point where all excess generators within the multi-generator network were shutdown and for the others to converge to a final state, leaving a one-to-one relationship between remaining generators and data manifolds present within the dataset. An example of this convergence metric is shown in Figure 5.4, which was monitored in real time using Tensorboard software. This training time, as in the amount of epochs cycled through, was explored during the initial phases of this research, and varied between each dataset. Training converged by 500 epochs for MNIST, 600 epochs for Fashion-MNIST, and 1,000 epochs for Wheel.

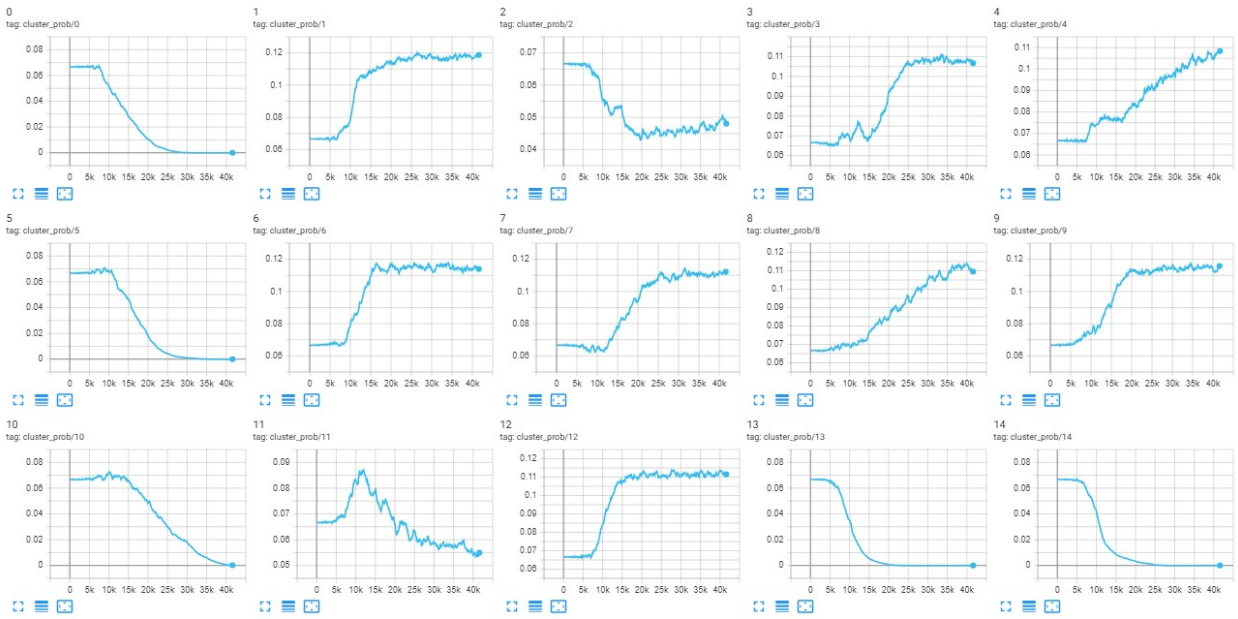


Figure 5.4: Clustering Convergence During Training. Each plot represents the cluster probability of association to a unique manifold within a given dataset. These plots are used to monitor the convergence of training a particular D-AnoGAN model. Training is complete once every plot reaches a steady-state final probability. These plots were monitored in real-time through Tensorboard software.

Once training was complete, the non-anomalous testing set was combined with anomalous samples (one of the ten classes for MNIST and Fashion-MNIST, custom anomalous samples for Wheel), and fed through the trained model. D-AnoGAN's anomaly detection performance was compared against other state of the art methods. For comparison tests using MNIST and Fashion-MNIST, the same model architecture published by all other methods was used for both datasets, as was the case for D-AnoGAN as well. This was done to ensure an equal playing field for all models during testing on Fashion-MNIST. For testing on Wheel, additional convolutional network layers were added to all comparison models to account for the higher resolution images (256x256), as was done with D-AnoGAN. These additional layers were required in order to follow the same down sampling/ up sampling procedure used for the 32x32 image resolution size of the MNIST and Fashion-MNIST datasets.

## 5.3 Network Hyper-Parameters

During model development, D-AnoGAN was tuned to an optimal set of hyperparameters for each individual network for the best anomaly detection performance possible. These hyperparameters consisted of a filter size used during convolutions, a stride (used by the filter as it moves along the image), and a padding size, which are pixels added to all four sides of an image when it is being processed by the filter. The generator network uses a filter size of five, a stride of one, a padding of two, and a scale-factor of two for up sampling between layers. All convolutions are followed by a *SELU* (or Scaled Exponential Linear Unit) activation function, with a final *tanh* (or Hyperbolic Tangent) activation placed before the output. A base filter size of 32 was used for this network, with each layer increasing the number of filters by a factor of two. The base form of the generator network is composed of three layers, although six layers were required for learning on the Wheel dataset to accommodate for the larger image resolution (256x256).

The encoder network uses a filter size of four for all layers except the final layer, which uses a filter size of three. Each layer uses a stride of two, a padding of one, and a down sampling scale-factor of two. Like the generator, *SELU* and *tanh* activation functions are implemented in this network as well. The encoder uses a base filter size of 32, each layer increasing the number of filters by a factor of two. The base form of this model is composed of three layers, but also requires six layers for the Wheel dataset to accommodate the larger image resolution (256x256).

The discriminator consists of three layers using a filter size of five, a stride of two, and a padding of two. A *Leaky-ReLU* (or Leaky-Rectified Linear Unit) activation function is used after each convolution. This network uses a base filter size of 32, each layer increasing the number of filters by a factor of two.

The classifier is a single-layer network which uses a filter size of five, a stride of two, a padding of two, and a down sampling scale-factor of two. The single-layer uses an input filter size 64 and output filter size of 128. Batch normalization and a *Leaky-ReLU* activation are performed after the single convolutional operation.

## 5.4 Evaluation

Anomaly detection performance is assessed by using the Area Under the Curve (AUC) of the Receiver Operating Characteristics (ROC) curve [43]. An ROC curve measures the performance of a classification model, where two classification parameters are plotted against each other: the true positive rate vs the false positive rate. The Area Under the ROC curve measures the entire two-dimensional area underneath the entire ROC curve. A common way to interpret AUC is by thinking of it as the probability that a model ranks a random anomalous sample *higher* than a random non-anomalous sample. AUC ranges in value from 0 to 1, where a model whose anomaly predictions are 100% wrong has an AUC of 0.0, and a model whose anomaly predictions are 100% correct has an AUC of 1.0. Not only that, but a model with an AUC of less than 0.5, where a score of 0.5 means there is a 50% chance of a correct classification, mislabels non-anomalous samples as anomalous more often than not.

For each anomaly detection test, D-AnoGAN's AUC is found by calculating an anomaly score, $\mathcal{A}$, which is a weighted sum of the Kullback–Leibler divergence between the estimated distributions of $x$ and $\hat{x}$ (from the classifier output) and the pixel-level difference between $x$ and $\hat{x}$, as follows:

$$\mathcal{A} = (1-\alpha)\left\|x-\hat{x}\right\|^2 + \alpha \mathrm{KL}\left[C(x) \parallel C(\hat{x})\right], \tag{5.1}$$

where $\alpha \in [0,1]$ is an anomaly scoring weight hyperparameter. During testing, it was found that tuning $\alpha$ for different datasets influenced the AUC score. The results for MNIST and Fashion-MNIST are reported with $\alpha = 0.9$; Wheel results are reported with $\alpha = 1.0$. These $\alpha$ values resulted in the highest AUC value for each respective dataset across all tests.

## 5.5 Results

D-AnoGAN was tested against all comparison methods for each of the three datasets. For MNIST (Table 5.1), the top row is associated with the class, or digit, removed during model training. Individual AUC scores for each test are reported in an individual column, as well as the mean AUC across all tests. An AUC of 1.0 indicates that a model correctly classified all of both the anomalous and non-anomalous samples correctly. An AUC of 0.0 indicates incorrect classification for all anomalous and non-anomalous samples. An AUC of 0.5 indicates that there is a 50% of a model classifying an anomalous sample correctly. For each anomaly detection test, the best-performing model's AUC score is boldfaced in black. The highest mean-AUC score across all tests for a given dataset is also boldfaced. For tests where the zero, two, and five digits were treated as an anomaly, [18] achieved the highest anomaly detection performance. D-AnoGAN achieved the best anomaly detection performance for tests that treated the one, three, four, six, seven, eight, and nine digits as anomalous. D-AnoGAN also achieved the highest mean-anomaly detection performance across all models tested with an AUC score of 0.91.

| **MNIST** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | MEAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| VAE [18] | **0.96** | 0.73 | **0.98** | 0.91 | 0.79 | **0.92** | 0.92 | 0.76 | 0.91 | 0.62 | 0.85 |
| f-AnoGAN [9] | 0.80 | 0.10 | 0.79 | 0.64 | 0.58 | 0.72 | 0.71 | 0.47 | 0.71 | 0.45 | 0.60 |
| OCGAN [31] | 0.86 | 0.46 | 0.93 | 0.83 | 0.83 | 0.78 | 0.83 | 0.73 | 0.59 | 0.67 | 0.75 |
| BiGAN [20] | 0.82 | 0.43 | 0.83 | 0.71 | 0.72 | 0.74 | 0.74 | 0.48 | 0.72 | 0.47 | 0.67 |
| GANomaly [13] | 0.78 | 0.16 | 0.83 | 0.68 | 0.69 | 0.73 | 0.82 | 0.53 | 0.84 | 0.56 | 0.66 |
| D-AnoGAN | 0.91 | **0.91** | 0.92 | **0.94** | **0.91** | 0.88 | **0.93** | **0.91** | **0.94** | **0.89** | **0.91** |

Table 5.1: Disconnected data manifold anomaly detection results for MNIST. AUC scores for each test associated with a removed-class, as well as the mean AUC across all tests, is reported. For each test, the AUC score of the best-performing model is boldfaced in black. The model with the highest average AUC score is also boldfaced in black. For both MNIST, D-AnoGAN achieves the best anomaly detection performance with the highest mean AUC.

For Fashion-MNIST (Table 5.2), the top row is associated with the class, or apparel item, removed during model training. Individual AUC scores for each test are reported, as well as the mean AUC across all tests. For each anomaly detection test, the best-performing model's AUC score is boldfaced in black. The highest mean-AUC score across all tests for a given dataset is also boldfaced. Several different model achieved the best anomaly detection performance for different tests. Overall anomaly detection performance is also lower in this dataset than in MNIST. This could be due to the fact that Fashion-MNIST has been documented as a more challenging dataset than MNIST for anomaly detection bench marking, due low image resolution and similar nature between manifolds. D-AnoGAN performed the best for tests where the "dress" class (column 3) and "handbag" class (column 8) were treated as anomalies during testing. D-AnoGAN also achieved the highest mean-anomaly detection performance across all models tested with an AUC score of 0.71.

| **Fashion-MNIST** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | MEAN |
|---|---|---|---|---|---|---|---|---|---|---|---|
| VAE [18] | 0.57 | 0.84 | **0.61** | 0.63 | **0.58** | 0.83 | 0.56 | 0.59 | 0.94 | 0.79 | 0.69 |
| f-AnoGAN [9] | 0.51 | 0.61 | 0.53 | 0.56 | 0.53 | 0.70 | 0.51 | 0.68 | 0.77 | **0.82** | 0.62 |
| OCGAN [31] | 0.54 | 0.84 | 0.52 | 0.62 | **0.59** | 0.84 | 0.48 | 0.60 | 0.48 | 0.70 | 0.62 |
| BiGAN [20] | 0.67 | 0.50 | 0.59 | 0.53 | 0.56 | 0.65 | **0.65** | 0.58 | 0.83 | 0.69 | 0.62 |
| GANomaly [13] | **0.68** | **0.85** | 0.52 | 0.71 | 0.49 | **0.95** | 0.53 | **0.67** | 0.97 | 0.81 | 0.70 |
| D-AnoGAN | 0.62 | 0.82 | 0.57 | **0.73** | 0.53 | 0.87 | 0.57 | 0.64 | **0.97** | 0.81 | **0.71** |

Table 5.2: Anomaly detection results for Fashion-MNIST. AUC scores for each test associated with a removed-class, as well as the mean AUC across all tests, is reported. For each test, the AUC score of the best-performing model is boldfaced in black. The model with the highest average AUC score is also boldfaced in black. D-AnoGAN achieves the best anomaly detection performance with the highest mean AUC.

For Wheel (Table 5.3), the top row relates to the anomalous manifold tested on with the trained Wheel model. D-AnoGAN achieved the best anomaly detection performance for the anomalous spoke test, with an AUC of 0.88. [31] achieved the best performance for the anomalous hub test, with an AUC of 0.98. D-AnoGAN also achieved the highest mean anomaly detection performance with a mean AUC of 0.93.

| **Wheel** | Spoke | Hub | MEAN |
|---|---|---|---|
| VAE [18] | 0.60 | 0.52 | 0.56 |
| f-AnoGAN [9] | 0.74 | 0.71 | 0.72 |
| OCGAN [31] | 0.84 | **0.98** | 0.91 |
| BiGAN [20] | 0.82 | 0.66 | 0.74 |
| GANomaly [13] | 0.62 | 0.54 | 0.58 |
| D-AnoGAN | **0.88** | 0.97 | **0.93** |

Table 5.3: Disconnected data manifold anomaly detection results for Wheel. AUC scores for both anomalous manifolds, as well as the mean AUC across both tests, is reported. For each test, the AUC score of the best-performing model is boldfaced in black. The model with the highest average AUC score is also boldfaced in black. D-AnoGAN achieves the best anomaly detection performance with the highest mean AUC.

Figure 5.5 provides further insight into how D-AnoGAN achieves strong anomaly detection performance in multi-class datasets over other state of the art. In this Figure, the latent spaces of D-AnoGAN and [18] are visualized, composed of non-anomalous digits 0-8 and the anomalous digit 9 from the MNIST dataset. Here, D-AnoGAN successfully clusters manifolds of non-anomalous data into individual learned representations. As a result, anomalous queries are encoded into a latent space associated with a single learned manifold representation only, and are therefore not reconstructed accurately, as is the case with an example 9-digit being reconstructed into a 4-digit. In contrast, [18] must treat the latent space of all learned manifold representations as a single, continuous one. Due to this, anomalous queries that share features with learned non-anomalous manifolds, such as the example 9-digit, can be reconstructed with a high level of accuracy, even though its representation was never learned by the model.

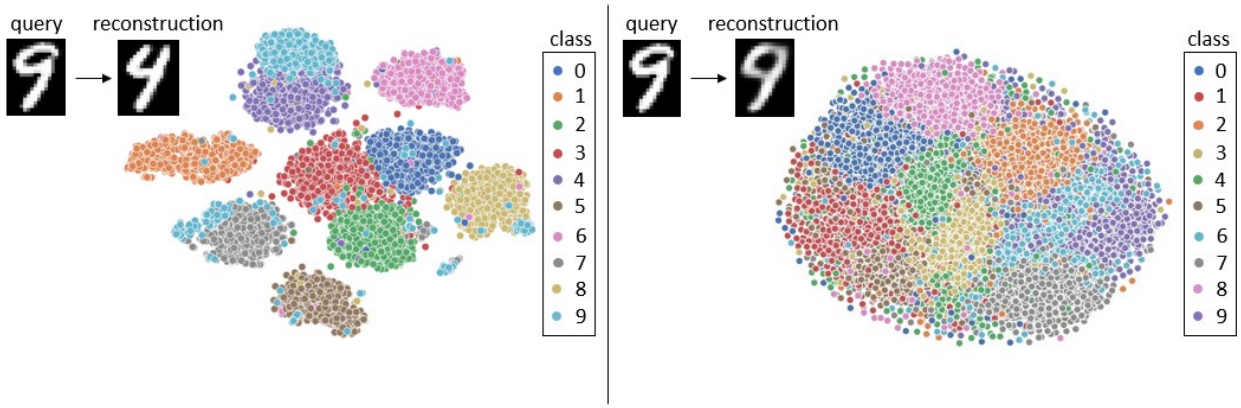


Figure 5.5: Latent space visualizations of D-AnoGAN (**left**) vs VAE [18] (**right**), composed of non-anomalous digits 0-8 and the anomalous digit 9 from MNIST. D-AnoGAN successfully clusters manifolds of non-anomalous data into individual learned representations. In contrast, the VAE must treat the latent space of all learned manifold representations as a single, continuous one. Becuase of this, the VAE is able to reconstruct a nine digit that was not present during training with a high level of accuracy. D-AnoGAN is not able to do so.

A study on how D-AnoGAN's model size influences anomaly detection is summarized in Table 5.4. It was found that achieving state of the art performance using the MNIST dataset required a smaller model than Fashion-MNIST, although models of the same size are reported in Tables 5.1-5.3. A potential explanation for this could be due to the feature complexity of the Fashion-MNIST dataset. For training on Wheel, it was found that the standard practice of doubling the convolutional filters for each network layer, resulting in over 342 million trainable parameters, was not required to achieve state of the art performance over other state of the art. This could potentially be due to the fact that unique features of each Wheel manifold only existed along the spoke and hub locations, with the rest of each manifold being quite simlar to each other. In order to produce a model of 57 million parameters, filters were only doubled every two layers instead of every single layer. Model sizes resulting in underachieving performance are also reported in Table 5.4 to showcase benchmarks required for achieving state of the art anomaly detection for each dataset using D-AnoGAN.

| MNIST | Fashion-MNIST | Wheel |
|---|---|---|
| 0.860 (852,280) | 0.700 (852,280) | 0.840 (25,756,045) |
| 0.914 (2,441,344) | 0.700 (2,441,344) | **0.930** (57,434,413) |
| **0.910** (7,831,312) | **0.710** (7,831,312) | 0.932 (342,387,469) |

Table 5.4: Mean AUC corresponding to different model sizes for D-AnoGAN trained on MNIST, Fashion-MNIST and Wheel. Number of trainable parameters are in parentheses. Boldfaced values are associated with results reported in Tables 5.1-5.3.

In addition, a comparison of model sizes used for training D-AnoGAN and all other methods on both MNIST and Wheel is summarized in Table 5.5. This comparison highlights the trade off for using a method like D-AnoGAN, which requires an increased model size over most other methods in order to account for the discontinuity of data manifolds.

| **Model Size** | MNIST | Wheel |
|---|---|---|
| VAE [18] | 237,656 | 16,865,340 |
| f-AnoGAN [9] | 149,505 | 2,233,153 |
| OCGAN [31] | 3,353,842 | 6,688,645 |
| BiGAN [20] | 12,036,490 | 20,384,738 |
| GANomaly [13] | 6,400,768 | 188,681,344 |
| D-AnoGAN | 7,831,312 | 57,434,4139 |

Table 5.5: Comparison of model sizes for experimentation on MNIST and Wheel.

## 5.6 Ablation Study

To study how different components of D-AnoGAN's network architecture impact overall anomaly detection performance, an ablation study, which studies the performance of an AI system by removing certain components, was performed to understand the contribution of the component to the overall system [44]. Three scenarios were studied, where models with the removed components were then tested to detect the same set of anomalies on the MNIST and Wheel datasets. Mean AUC scores for each study are shown in Table 5.6.

| | MNIST | Wheel |
|---|---|---|
| Training w/o bandit | 0.92 | 0.89 |
| Training w/o xzx | 0.90 | 0.93 |
| Normal Training | 0.91 | 0.93 |

Table 5.6: Ablation study results for D-AnoGAN. The ablation study was conducted by removing pieces of the D-AnoGAN model and running anomaly detection testing with those models for the MNIST and Wheel datasets.

For the first ablation study, the bandit was disabled during training, and the number of clusters was set equal to the set of manifolds within the dataset. This forced data samples to cluster in a supervised fashion, since no discovery of optimal generators was taking place by the bandit. Anomaly detection performance was still high, which was expected, for both datasets. It was predicted that disabling the bandit would *improve* anomaly detection performance, since no discovery was taking place, which was the case with MNIST. An interesting observation was that disabling the bandit *decreased* performance slightly in the Wheel dataset. A potential explanation for this could be that similar non-anomalous Wheel manifolds ended up sharing generators during training.

For the second ablation study, additional xzx-training of the multi-encoder was omitted during training. Doing so would result in a lower reconstruction accuracy of non-anomalous samples during training, potentially decreasing anomaly detection performance due to our pixel-level difference metric. It was found that omitting additional xzx-training has little to no effect on anomaly detection performance, which was an interesting find. This indicates that the additional training step may not be necessary for a model like D-AnoGAN that accounts for discontinuity between manifolds.

Finally, for a baseline comparison to the other two studies, the third ablation study used the full standard training pipeline. Through this study, it is shown that the full D-AnoGAN pipeline is useful with unsupervised anomaly detection tasks.

# Chapter 6

# Future Work

The research put into developing D-AnoGAN provides a great foundation that can be built upon in future work. First off, all of the datasets used for anomaly detection bench-marking consisted of gray-scale images, although many anomaly detection datasets related to real-world applications consist of RGB images. Implementing D-AnoGAN for RGB datasets was beyond the scope of this work, but would be a suggested expansion to the work presented here.

Designing even more complex datasets for the training and anomaly detection testing process is also an opportunity for future projects. The anomalies D-AnoGAN was tasked with detecting when using the canonical MNIST and Fashion-MNIST datasets were fairly straight-forward. During the development of the Wheel dataset, the anomalous manifolds that were designed are a good starting point for the types of anomalies D-AnoGAN excels with detecting (containing combinations of non-anomalous features), but more exploration into subtle anomalies that can exist in real-world applications would be an interesting research direction.

Another method for detecting anomalies of this type could be through handling anomalies within another domain of data, such as graph structures. This type of data representation has shown promise in other machine learning domains, and could serve as an additional direction for future work.

Finally, an additional research direction that would be of interest as computational hardware advances is to utilize D-AnoGAN in a real-time anomaly detection environment, where the model is tasked with testing on a live-video feed. Because of the network complexity of a model like D-AnoGAN, the reconstruction of a query image necessary to perform anomaly detection is computationally expensive, and a bit inefficient at this time. This may not be the case within the next handful of years, allowing anomaly assessment to be performed to video feed with many frames every second. This could be applied to tasks such as detecting anomalous objects within a scene, where real-time detection is critical for a given scenario.

# Chapter 7

# Conclusions

For the task of anomaly detection, current state of the art deep generative methods learn the representation of training data in the form of a single, continuous manifold. This approach works well in cases where anomaly detection is performed on data samples that fall outside of a single learned non-anomalous class, but breakdown in the case where multiple classes are considered non-anomalous within the training data. This work set out to explore how accounting for potential discontinuities between data manifolds could improve anomaly detection performance within multi-class datasets. To do this, a novel GAN framework was developed consisting of a multi-generator network, where each generator was tasked with learning a single unique manifold within a given dataset. A bandit was also implemented to discover an optimal set of generators responsible for covering all manifolds within a dataset, and to guide the clustering of each manifold into a separate generator. The proposed model, D-AnoGAN, was developed using open-source computer software, and tested against other state of the art methods at anomaly detection performance.

It has been shown that accounting for discontinuity between manifolds results in improved anomaly detection performance by as much as 5% over a variety of multi-class datasets, outperforming several other recent state of the art GAN models as well as alternative deep generative methods. This work also aimed to gain a sense of how GAN-based methods could be used in real-world anomaly detection settings, developing a novel Wheel dataset representative of manifolds found in the automotive industry. Through experimentation on this dataset, it is clear that GAN-based models can be successfully implemented in real-world datasets for anomaly detection.

Potential avenues for future work involving D-AnoGAN have also been considered, for example training the model on more real-world datasets using live video feed. It would also be interesting exploring the model size constraints of D-AnoGAN, studying how large a size of unique manifolds within a dataset the model can learn with maintaining acceptable performance.

# Appendix A

# D-AnoGAN Training Algorithm

*Input:* Training dataset $X$, training epochs $N$

**for** *training epochs 1 to N* **do**
  **for** *each mini-batch x* **do**
    Discriminator Update:
    $\hat{x} \leftarrow G(z, k)$
    $l_D \leftarrow D_{GP} - D(\hat{x}) - D(x)$
    *Back-propagate* $l_D$ *to change* $D$
    **for** *every five discriminator updates* **do**
      Multi-Generator Update:
      $l_C \leftarrow H(k, C_{\hat{x}})$
      $l_G \leftarrow -D(\hat{x}) - l_C$
      *Back-propagate* $l_G$ *to change* $G$
      Multi-Encoder Update:
      $\hat{z} \leftarrow E(\hat{x}, k)$
      $l_E \leftarrow (z - \hat{z})^2$
      *Back-propagate* $l_E$ *to change* $E$
      Classifier Update:
      $C_{\hat{x}} \leftarrow C(D_i(\hat{x}))$
      $l_C \leftarrow H(k, C_{\hat{x}})$
      *Back-propagate* $l_C$ *to change* $C$
      Bandit Update:
      $B(k) \approx p(k)$
      $C_x \leftarrow C(D_i(x))$
      $l_B \leftarrow H(B(k), C_x) - H(C_x)$
      *Back-propagate* $l_B$ *to change* $B$
    **end**
  **end**
**end**

*Output:* Network Models: Multi-Generator, Multi-Encoder, Discriminator, Classifier, Bandit

**$xzx$-training**

*Input:* Trained network models, training dataset $X$, training epochs $M$

**for** *training epochs 1 to M* **do**
  **for** *each mini-batch x* **do**
    Multi-Encoder Update:
    $C_x \leftarrow C(D_i(x))$
    $z \leftarrow E(x, C_x)$
    $\hat{x} \leftarrow G(z, C_x)$
    $l_E \leftarrow (x - \hat{x})^2$
    *Back-propagate* $l_E$ *to change* $E$
  **end**
**end**

*Output:* Network Model: Multi-Encoder

# Appendix B

# Python Code for D-AnoGAN

## B.1 run.py

```
import numpy as np
from numpy import savetxt
import torch
from DM_AnoGAN_PL import DM_AnoGAN_PL
import argparse
import torch.cuda
from torch.utils.data import DataLoader
from utils import process_anomaly_scores
import random
import pdb
import os
import datetime
from datetime import date

parser = argparse.ArgumentParser()
parser.add_argument("--G_lr", type=float, default=2e-4)
parser.add_argument("--D_lr", type=float, default=2e-4)
parser.add_argument("--E_1_lr", type=float, default=1e-3)
parser.add_argument("--E_2_lr", type=float, default=2e-4)
parser.add_argument("--B_lr", type=float, default=1e-3)
parser.add_argument("--E_loss_weight", type=float, default=1.0)
parser.add_argument("--model_name", type=str, default="DM_AnoGAN_PL")
parser.add_argument("--batch_size", type=int, default=128)
parser.add_argument("--c_dim", type=int, default=15)
parser.add_argument("--z_dim", type=int, default=30)
parser.add_argument("--num_eval_out", type=int, default=5)
parser.add_argument("--eval_int", type=int, default=100)
parser.add_argument("--epochs", type=int, default=7500)
parser.add_argument("--D_steps", type=int, default=5)
parser.add_argument("--load_epoch", type=int, default=0)
parser.add_argument("--save_int", type=int, default=250)
parser.add_argument("--decay_rate", type=float, default=0.999)
parser.add_argument("--bandit", action='store_true')
parser.add_argument("--train", action='store_true')
parser.add_argument("--anomaly_test", action='store_true')
parser.add_argument("--train_xzx", action='store_true')
parser.add_argument("--seed", type=int, default=0)
parser.add_argument("--gpu", type=int, default=0)
args = parser.parse_args()

#create logging file for setting information
if not os.path.isdir('./Settings_README/'):
    os.makedirs('./Settings_README/')
filename = str(args.model_name) + '_settings' + '.txt'
with open('Settings_README/' + filename, 'a') as log:
    log.write('Model Name: ' + str(args.model_name) + '\n')
    log.write('Job Execution Date: ' + str(date.today()) + '\n')
    log.write('Job Ran by Walker Dimon (wdimon2@illinois.edu) \n')
    log.write('Job Ran on Campus DL Machine \n')
    log.write('Batch Size: ' + str(args.batch_size) + '\n')
```

```
    log.write('Encoder (E2) Momentum Settings found in DM_AnoGAN_PL.py \n')
    log.write('Epochs: ' + str(args.epochs) + '\n')
    log.write('Seed: ' + str(args.seed) + '\n')
    log.write('Conda Environment: Torch.yml (on GitHub)')
    log.write('\n')

def main():

    #set cuda GPU
    if torch.cuda.is_available():
        torch.cuda.set_device(args.gpu)
        device = torch.device("cuda")
    else:
        device = torch.device("cpu")

    # Pytorch recommended settings for reproducibility
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # Setting manual seed for reproducibility
    random.seed(args.seed)
    torch.manual_seed(args.seed)

    # Initialize DM_AnoGAN_PL model
    AnoGAN = DM_AnoGAN_PL(args)

    if args.train:

        # Load data and create train and test dataloaders
        training_data = torch.load('./data/MNIST/processed/training_processed.pt')
        train_data_loader = DataLoader(training_data, batch_size=args.batch_size,
shuffle=True)

        # Initialize iterations counter
        D_iter = 0

        for epoch in range(args.epochs):
            print('Training Epoch {}/{}'.format((epoch+1), args.epochs))
            for data, labels in train_data_loader:

                # Only train with full batches
                if len(data) == args.batch_size:

                    # Train discriminator
                    AnoGAN.train_D(data.cuda())

                    # Train generator, encoder, and bandit
                    if D_iter % args.D_steps == 0:

                        AnoGAN.train_G()
```

```
                        AnoGAN.train_E_1()
                        AnoGAN.train_E_2()
                        AnoGAN.estimate_real(data.cuda())

                        if args.bandit:
                            AnoGAN.train_B()

                    # Log metrics to tensorboard output every epoch
                    if D_iter % args.D_steps == 0:
                        AnoGAN.log_results(D_iter/args.D_steps, args.bandit)

                    D_iter += 1

            # Output batch of generated images
            if (epoch+1) % args.eval_int == 0:
                AnoGAN.output_generated_images(epoch)

            # Save model at save interval
            if epoch > 0 and (epoch+1) % args.save_int == 0:
                AnoGAN.save_model(epoch, args.train_xzx)

    if args.train_xzx:

        # Load data and create train dataloader
        training_data = torch.load('./data/MNIST/processed/training_processed.pt')
        train_data_loader = DataLoader(training_data, batch_size=args.batch_size,
shuffle=True)

        # Train E_1 using xzx training
        for epoch in range(args.epochs):
            for i, (data, labels) in enumerate(train_data_loader):
                AnoGAN.train_xzx(data.cuda(), i + epoch*len(train_data_loader))

            # Save model at save interval
            if epoch > 0 and (epoch+1) % args.save_int == 0:
                AnoGAN.save_model(epoch, args.train_xzx)

    if args.anomaly_test:

        # Load data and create anomaly and normal dataloaders
        anomaly_data =
torch.load('./data/MNIST/processed/test_HOLDOUT_processed.pt')
        normal_data = torch.load('./data/MNIST/processed/test_processed.pt')
        anomaly_data_loader = DataLoader(anomaly_data, batch_size=args.batch_size,
shuffle=True)
        normal_data_loader = DataLoader(normal_data, batch_size=args.batch_size,
shuffle=True)

        MSE_anomaly = []
        MSE_D_i_anomaly = []
```

```
        KL_z_c_anomaly = []

        for i, data in enumerate(anomaly_data_loader):

            # Forward pass test anomaly batch through DM-AnoGAN-PL and calculate
anomaly scores
            MSE, D_i_MSE, z_c_MSE = AnoGAN.anomaly_test(data, 'anomaly', i)

            # Append batch anomaly scores
            MSE_anomaly = np.concatenate((MSE_anomaly, MSE.cpu().numpy()), axis=0)
            MSE_D_i_anomaly = np.concatenate((MSE_D_i_anomaly,
D_i_MSE.cpu().numpy()), axis=0)
            KL_z_c_anomaly = np.concatenate((KL_z_c_anomaly, z_c_MSE.cpu().numpy()),
axis=0)

        # Stack anomaly scores for anomalous test samples
        output_anomaly = np.stack((MSE_anomaly, MSE_D_i_anomaly, KL_z_c_anomaly),
axis=-1)

        MSE_normal = []
        MSE_D_i_normal = []
        KL_z_c_normal = []
        i = 0

        for data, labels in normal_data_loader:

            # Forward pass test non-anomalous batch through DM-AnoGAN-PL and
calculate anomaly scores
            MSE, D_i_MSE, z_c_KL = AnoGAN.anomaly_test(data, 'real', i)

            # Append batch anomaly scores
            MSE_normal = np.concatenate((MSE_normal, MSE.cpu().numpy()), axis=0)
            MSE_D_i_normal = np.concatenate((MSE_D_i_normal, D_i_MSE.cpu().numpy()),
axis=0)
            KL_z_c_normal = np.concatenate((KL_z_c_normal, z_c_KL.cpu().numpy()),
axis=0)

            i += 1

        # Stack anomaly scores for non-anomalous test samples
        output_normal = np.stack((MSE_normal, MSE_D_i_normal, KL_z_c_normal),
axis=-1)

        # Calculate AUC values and output ROC curve for best alpha
        process_anomaly_scores(output_anomaly, output_normal, args.model_name)


if __name__ == '__main__':
    main()
```

## B.2 D-AnoGAN.py

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import os
from utils import z_sampler, weights_init, Logger, calc_gradient_penalty,
save_images, entropy
from networks import Discriminator, Encoder_zc, Bandit, MultiGenerator, MultiEncoder
import pdb
from torchsummary import summary

class DM_AnoGAN_PL(object):
    def __init__(self, G_lr, D_lr, B_lr, E_1_lr, E_2_lr, z_dim, b_size, c_dim,
E_loss_weight, decay_rate,
                 model_name, save_int, load_epoch, num_eval_out, train_xzx,
anomaly_test, summary):

        self.z_dim = z_dim
        self.c_dim = c_dim
        self.b_size = b_size
        self.E_loss_weight = E_loss_weight
        self.decay_rate = decay_rate
        self.save_int = save_int
        self.model_name = model_name
        self.load_epoch = load_epoch

        # Create networks that make up model
        self.Gen = MultiGenerator(z_dim, c_dim).cuda()
        self.Disc = Discriminator().cuda()
        self.Enc_1 = MultiEncoder(z_dim, c_dim).cuda()
        self.Enc_2 = Encoder_zc(c_dim, z_dim).cuda()
        self.Ban = Bandit(c_dim).cuda()

        #output network sizes
        if summary:
            summary(self.Gen)
            summary(self.Disc)
            summary(self.Enc_1)
            summary(self.Enc_2)
            summary(self.Ban)

        # Initialize bandit update counter
        self.count = 0

        if train_xzx or anomaly_test:
            self.optimizer_E_1 = torch.optim.Adam(self.Enc_1.parameters(), lr=1e-3,
betas=(0.5, 0.5))


            if load_epoch == 0:
```

```
                print('xzx training, or anomaly testing, can only be done after
normal training')
                raise SystemExit
            else:
                # Load model parameters and optimizer state variables
                load_dir = './model_save/' + model_name + '/' + str(load_epoch) +
'.pt'
                checkpoint = torch.load(load_dir)
                self.Gen.load_state_dict(checkpoint['gen_state_dict'])
                self.Disc.load_state_dict(checkpoint['disc_state_dict'])
                self.Enc_1.load_state_dict(checkpoint['enc_1_state_dict'])
                self.Enc_2.load_state_dict(checkpoint['enc_2_state_dict'])
                self.Ban.load_state_dict(checkpoint['ban_state_dict'])

        else:
            # Set optimizers for all networks
            self.optimizer_G = torch.optim.Adam(self.Gen.parameters(), lr=G_lr,
betas=(0.9, 0.99), weight_decay=5e-5)
            self.optimizer_E_1 = torch.optim.Adam(self.Enc_1.parameters() ,
lr=E_1_lr, betas=(0.5, 0.5))
            self.optimizer_E_2 = torch.optim.Adam(self.Enc_2.parameters(),
lr=E_2_lr, betas=(0.9, 0.99))
            self.optimizer_D = torch.optim.Adam(self.Disc.parameters(), lr=D_lr,
betas=(0.5, 0.5))
            self.optimizer_B = torch.optim.Adam(self.Ban.parameters(), lr=B_lr,
betas=(0.5, 0.5))

            if load_epoch > 0:
                # Load model parameters and optimizer state variables
                load_dir = './model_save/' + model_name + '/' + str(load_epoch) +
'.pt'
                checkpoint = torch.load(load_dir)
                self.Gen.load_state_dict(checkpoint['gen_state_dict'])
                self.Disc.load_state_dict(checkpoint['disc_state_dict'])
                self.Enc_1.load_state_dict(checkpoint['enc_1_state_dict'])
                self.Enc_2.load_state_dict(checkpoint['enc_2_state_dict'])
                self.Ban.load_state_dict(checkpoint['ban_state_dict'])

self.optimizer_G.load_state_dict(checkpoint['gen_optimizer_state_dict'])

self.optimizer_D.load_state_dict(checkpoint['disc_optimizer_state_dict'])

self.optimizer_B.load_state_dict(checkpoint['ban_optimizer_state_dict'])
                self.Enc_1.load_state_dict(checkpoint['enc_1_state_dict'])
                self.Enc_2.load_state_dict(checkpoint['enc_2_state_dict'])

            else:
                # Initialize the weights
                self.Gen.apply(weights_init)
                self.Disc.apply(weights_init)
```

```
                self.Enc_1.apply(weights_init)
                self.Enc_2.apply(weights_init)

        # Create Tensorboard logger
        self.logger = Logger('./logs/' + model_name + '_')

        # Generate fixed noise vectors and fixed clusters for testing generator
        self.fixed_z_u = 2 * torch.rand(num_eval_out, z_dim).cuda() - 1
        self.fixed_z_c = torch.LongTensor(range(0, c_dim)).unsqueeze(1).cuda()



    # train discriminator network with real images and generated images
    def train_D(self, real):

        self.Gen.train()
        self.Disc.train()
        self.Enc_1.train()
        self.Enc_2.train()
        self.Ban.train()
        self.Disc.zero_grad()

        # Generate z uniform noise batch and bach of clusters sampled from learned
prior
        self.z_u, self.z_c_oh, self.z_c = z_sampler(self.Ban, self.b_size,
self.z_dim, self.c_dim)

        # Generate fake image batch with G
        self.fake = self.Gen(self.z_u, self.z_c)

        # Forward pass real batch through D
        D_x, D_i_x = self.Disc(real)

        # Forward pass fake batch through D
        D_G_z1, D_i_z1 = self.Disc(self.fake.detach())

        # Calculate mean output for D for real samples
        self.D_x_avg = D_x.view(-1).mean()

        # Calculate the mean output for D for fake generated samples
        self.D_G_z1_avg = D_G_z1.view(-1).mean()

        # Calculate gradient penalty
        self.GP = calc_gradient_penalty(self.Disc, real, self.fake.detach(),
self.b_size)

        # Calculate disc loss
        self.errD = self.D_G_z1_avg - self.D_x_avg + self.GP

        # Calculate D gradients
```

```
        self.errD.backward()

        # Update D
        self.optimizer_D.step()

        # Calculate W estimate for logging
        self.w_est = self.D_x_avg - self.D_G_z1_avg



    # train generator network by generating batch of fake images
    def train_G(self, ):

        self.Gen.zero_grad()

        # Forward pass of all-fake batch through D
        D_G_z2, D_i_z2 = self.Disc(self.fake)

        # Calculate average D output for generated samples
        self.D_G_z2_avg = D_G_z2.view(-1).mean()

        # Calculate E_2 logits for generated sampled
        self.E_logits_fake = self.Enc_2(D_i_z2)

        # E1 output for batch of generated samples
        self.E_z_fake = self.Enc_1(self.fake, self.z_c)

        # Calculate E_2 cross entropy loss
        self.E_z_c_loss = self.E_loss_weight *
nn.CrossEntropyLoss()(self.E_logits_fake, self.z_c.squeeze())

        # Calculate E_1 MSE for noise vector
        self.E_z_u_loss = self.E_loss_weight * ((self.z_u - self.E_z_fake) **
2).mean()

        # Calculate generator loss (WGAN loss + E_2 loss)
        self.G_loss = -self.D_G_z2_avg + self.E_z_c_loss

        # Calculate gradients for G and retain graph for E training
        self.G_loss.backward(retain_graph=True)

        # Update G
        self.optimizer_G.step()

    def train_E_1(self, ):

        self.Enc_1.zero_grad()

        # Calculate gradients for E_1 and retain graph for E_2 training
        self.E_z_u_loss.backward(retain_graph=True)
```

```
        # Update E_1
        self.optimizer_E_1.step()

    def train_E_2(self, ):

        self.Enc_2.zero_grad()

        # Calculate gradients for E_2
        self.E_z_c_loss.backward()

        # Update E_2
        self.optimizer_E_2.step()

    # Update bandit using encoder's estimate of real distribution
    def train_B(self):

        self.Ban.zero_grad()

        # Calculate prior entropy for bandit loss
        g_uniform_loss = -(F.softmax(self.Ban(), dim=-1) * F.log_softmax(self.Ban(),
dim=-1)).mean()

        # Calculate bandit loss using encoder's estimate of real and entropy loss
        self.Ban_loss = -(self.E_est_real_avg * F.log_softmax(self.Ban(),
dim=-1)).mean() - 1000*(self.decay_rate**self.count)*g_uniform_loss
        self.count += 1

        # Calculate gradients for B
        self.Ban_loss.backward()

        # Update bandit
        self.optimizer_B.step()

    # Log all metrics in tensorboard log file
    def log_results(self, G_iter, bandit):

        if bandit:
            self.logger.scalar_summary('bandit_loss', self.Ban_loss, G_iter)

            # Calculate prob of each cluster
            c_softmax = F.softmax(self.Ban(), dim=-1)

        # Log metrics for discriminator performance
        self.logger.scalar_summary('w_distance', self.w_est, G_iter)
        self.logger.scalar_summary('gp', self.GP, G_iter)
        self.logger.scalar_summary('disc_loss', self.errD, G_iter)

        # Log metrics for encoder/generator performance
        self.logger.scalar_summary('encoder_z_c_loss', self.E_z_c_loss, G_iter)
```

```
        self.logger.scalar_summary('encoder_z_u_loss', self.E_z_u_loss, G_iter)
        self.logger.scalar_summary('gen_loss', self.G_loss, G_iter)
        self.logger.scalar_summary('recon_real_loss', self.real_mse, G_iter)
        self.logger.scalar_summary('real_est_entropy', self.real_est_entropy,
G_iter)

        # Log per cluster metrics
        for cluster in range(self.c_dim):
            self.logger.scalar_summary('E_real_cluster_prob/' + str(cluster),
self.E_est_real_avg[cluster], G_iter)

            if bandit:
                self.logger.scalar_summary('cluster_prob/' + str(cluster),
c_softmax[cluster], G_iter)


    # Output generated images for qualitative comparison
    def output_generated_images(self, epoch):

        # Set G and E_2 to eval mode for BN layers
        self.Gen.eval()
        self.Enc_2.eval()

        save_dir = './output_' + self.model_name + '/' + str(epoch +
self.load_epoch)

        for c, cluster in enumerate(self.fixed_z_c):

            if not os.path.exists(save_dir + '/' + str(c)):
                os.makedirs(save_dir + '/' + str(c))

            # Create batch of same one hot cluster vectors

            fixed_cluster_z_c = cluster.expand(self.fixed_z_u.size()[0],
self.fixed_z_c.size()[1]).cuda()

            with torch.no_grad():

                # Forward pass of fixed random noise for constant cluster number
                output = self.Gen(self.fixed_z_u, fixed_cluster_z_c)

            save_images(output, save_dir, c)

    # Estimate real distribution, measure reconstruction error, and entropy
    def estimate_real(self, real):

        self.Gen.eval()

        with torch.no_grad():
```

```
            # Forward pass real batch through D
            D_x, D_i_x = self.Disc(real)

            # Calculate E_2 cluster output for generated samples
            z_c_logits_real = self.Enc_2(D_i_x)

            # Calculate softmax cluster probabilities
            z_c_sm = F.softmax(z_c_logits_real, dim=-1)

            # Convert ouput vector to cluster number
            value, z_c = torch.max(z_c_sm, -1)

            # Pass real samples through encoder to get latent representation
            E_z_u_real = self.Enc_1(real, z_c)

            # using average softmax output to estimate prior
            self.E_est_real_avg = torch.mean(z_c_sm, dim=0)

            # generate images with encoder latent representation of real images
            real_recon = self.Gen(E_z_u_real, z_c.unsqueeze(dim=-1))

            # Calculate MSE for reconstruction performance
            self.real_mse = ((real - real_recon) ** 2).mean()

            # Calculate entropy of E_real_estimates
            self.real_est_entropy = entropy(z_c_logits_real)

    def save_model(self, epoch, train_xzx):

        if train_xzx:
            if not os.path.exists('./model_save/' + self.model_name + '_xzx'):
                os.makedirs('./model_save/' + self.model_name + '_xzx')

            save_dir = './model_save/' + self.model_name + '_xzx/' + str(epoch) +
'.pt'

            torch.save({'epoch': epoch,
                        'gen_state_dict': self.Gen.state_dict(),
                        'disc_state_dict': self.Disc.state_dict(),
                        'enc_1_state_dict': self.Enc_1.state_dict(),
                        'enc_2_state_dict': self.Enc_2.state_dict(),
                        'ban_state_dict': self.Ban.state_dict(),
                        }, save_dir)
        else:
            if not os.path.exists('./model_save/' + self.model_name):
                os.makedirs('./model_save/' + self.model_name)

            save_dir = './model_save/' + self.model_name + '/' + str(epoch +
self.load_epoch) + '.pt'
```

```
            torch.save({'epoch': epoch,
                'gen_state_dict': self.Gen.state_dict(),
                'gen_optimizer_state_dict': self.optimizer_G.state_dict(),
                'disc_state_dict': self.Disc.state_dict(),
                'disc_optimizer_state_dict': self.optimizer_D.state_dict(),
                'enc_1_state_dict': self.Enc_1.state_dict(),
                'enc_1_optimizer_state_dict': self.optimizer_E_1.state_dict(),
                'enc_2_state_dict': self.Enc_2.state_dict(),
                'enc_2_optimizer_state_dict': self.optimizer_E_2.state_dict(),
                'ban_state_dict': self.Ban.state_dict(),
                'ban_optimizer_state_dict': self.optimizer_B.state_dict(),
                }, save_dir)

    def anomaly_test(self, query, name, batch_num):
        self.Gen.eval()
        self.Enc_2.eval()

        # Transfer query images to GPU
        query = query.cuda()

        with torch.no_grad():

            # Forward pass query through discriminator
            D_query, D_i_query = self.Disc(query)

            # Forward pass discriminator's intermediate output through E_2
            z_c_query_logits = self.Enc_2(D_i_query)

            # Calculate cluster with max score
            value, z_c = torch.max(F.softmax(z_c_query_logits, dim=-1), dim=-1)

            # Forward pass query and cluster number through E_2 to get z vector
            z_u = self.Enc_1(query, z_c)

            # Forward pass encoded z_u and z_c to generate reconstruction
            recon = self.Gen(z_u, z_c)

            # Forward pass reconstruction through discriminator
            D_recon, D_i_recon = self.Disc(recon)

            # Forward pass discriminator's intermediate output through E_2
            z_c_recon_logits = self.Enc_2(D_i_recon)

        D_i_MSE = ((D_i_query - D_i_recon)**2).view(D_i_query.size()[0], -1).mean(1)
        MSE_image = ((query - recon)**2).view(query.size()[0], -1).mean(1)

        # Calculate KL divergence between query E2 output and reconstruction E2
output
        z_c_KL = nn.KLDivLoss(size_average=False,
reduce=False)(F.log_softmax(z_c_recon_logits, dim=-1), F.softmax(z_c_query_logits,
```

```
dim=-1)).sum(1)

        # Save query image, reconstruction and pixel MSE for first batch for viewing
        if batch_num == 0:
            MSE_pixel = ((query - recon)**2)

            if not os.path.exists('./anomaly_images/' + self.model_name + '/' + name
+ '_MSE/0'):
                os.makedirs('./anomaly_images/' + self.model_name + '/' + name +
'_MSE/0')

            if not os.path.exists('./anomaly_images/' + self.model_name + '/' + name
+ '_query/0'):
                os.makedirs('./anomaly_images/' + self.model_name + '/' + name +
'_query/0')

            if not os.path.exists('./anomaly_images/' + self.model_name + '/' + name
+ '_recon/0'):
                os.makedirs('./anomaly_images/' + self.model_name + '/' + name +
'_recon/0')

            save_images(MSE_pixel, './anomaly_images/' + self.model_name + '/' +
name + '_MSE', 0, True)
            save_images(query, './anomaly_images/' + self.model_name + '/' + name +
'_query', 0)
            save_images(recon, './anomaly_images/' + self.model_name + '/' + name +
'_recon', 0)

        return MSE_image, D_i_MSE, z_c_KL

    # Train encoder using real data only after normal training is complete
    def train_xzx(self, real, iter):
        self.Gen.zero_grad()
        self.Enc_1.zero_grad()
        self.Enc_2.zero_grad()
        self.Gen.eval()
        self.Enc_2.eval()

        # Forward pass real samples through discriminator
        D_x, D_i_x = self.Disc(real)

        # Forward pass discriminator's intermediate output through E_2
        z_c_logits = self.Enc_2(D_i_x)

        # Calculate cluster with max score
        value, z_c = torch.max(F.softmax(z_c_logits, dim=-1), dim=-1)

        # Forward pass real image and cluster number through E_2 to get z vector
        z_u = self.Enc_1(real, z_c)
```

```
# Forward pass encoded z_u and z_c to generate reconstruction
real_recon = self.Gen(z_u, z_c)

# Calculate E_1 MSE between input z noise and encoded z noise
self.E_1_loss = ((real - real_recon) ** 2).mean()

# Calculate gradients E_1
self.E_1_loss.backward()

# Update E_1 only
self.optimizer_E_1.step()

# Log xzx reconstruction error
self.logger.scalar_summary('recon_mse', self.E_1_loss, iter)
```

## B.3 networks.py (MNIST and Fashion-MNIST)

```
import numpy as np
import torch.nn as nn
import torch
import torch.nn.functional as F
from torch.nn import Parameter as P

class MultiGenerator(nn.Module):
    def __init__(self, z_dim, c_dim):
        super(MultiGenerator, self).__init__()

        self.c_dim = c_dim

        self.Generators = nn.ModuleList([Generator(z_dim) for i in range(c_dim)])

    def forward(self, z, zc):

        # create empty output tensor to populate
        out = torch.empty(z.size()[0],1,28,28).cuda()

        # create list of unique clusters in batch
        zc = zc.cpu().squeeze().numpy()
        clusters = np.unique(zc)

        # split each z noise input and feed through corresponding cluster's
generator
        # is there a way to do this in parallel?
        for i, c in enumerate(clusters):
            idx = torch.tensor(np.argwhere(zc == c)).cuda().squeeze()
            g_out = self.Generators[c](z.index_select(0, idx))
            out[idx] = g_out
        return out


class Generator(nn.Module):
    def __init__(self, z_dim):
        super(Generator, self).__init__()

        self.Activation = nn.SELU(inplace=True)

        self.Dense_net_1 = nn.Linear(z_dim, 128*4*4)

        self.cn_1 = nn.Conv2d(128, 64, 5, stride=1, padding=2, bias=True)

        self.cn_2 = nn.Conv2d(64, 32, 5, stride=1, padding=2, bias=True)

        self.zp = nn.ZeroPad2d((-2,-2,-2,-2))
        self.cn_3 = nn.Conv2d(32, 1, 5, stride=1, padding=2, bias=True)


    def forward(self, z):
```

```
        x = self.Dense_net_1(z)
        x = self.Activation(x)
        x = x.view(x.size(0), 128, 4, 4)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_1(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_2(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.zp(x)
        x = self.cn_3(x)
        x = torch.tanh(x)

        return x


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.Activation = nn.LeakyReLU(inplace=True, negative_slope=0.2)

        self.cn_1 = nn.Conv2d(1, 32, 5, stride=2, padding=2, bias=True)

        self.cn_2 = nn.Conv2d(32, 64, 5, stride=2, padding=2, bias=True)

        self.cn_3 = nn.Conv2d(64, 128, 5, stride=2, padding=2, bias=True)

        self.Dense_net_1 = nn.Linear(128*4*4, 1)

    def forward(self, x):
        d = self.cn_1(x)
        d = self.Activation(d)
        d = self.cn_2(d)
        d_i = self.Activation(d)
        d = self.cn_3(d_i)
        d = self.Activation(d)
        d = d.view(d.size(0), 128*4*4)
        d = self.Dense_net_1(d)
        return d, d_i


class MultiEncoder(nn.Module):
    def __init__(self, z_dim, c_dim):
        super(MultiEncoder, self).__init__()

        self.c_dim = c_dim
```

```
        self.z_dim = z_dim

        self.Encoders = nn.ModuleList([Encoder_z_u(z_dim) for i in range(c_dim)])

    def forward(self, x, zc):

        # create empty output tensor to populate
        out = torch.empty(x.size()[0],self.z_dim).cuda()

        # create list of unique clusters in batch
        zc = zc.cpu().squeeze().numpy()
        clusters = np.unique(zc)

        # split each z noise input and feed through corresponding cluster's
generator
        # is there a way to do this in parallel?
        for i, c in enumerate(clusters):
            idx = torch.tensor(np.argwhere(zc == c)).cuda().squeeze()
            e_out = self.Encoders[c](x.index_select(0, idx))
            out[idx] = e_out
        return out


class Encoder_z_u(nn.Module):
    def __init__(self, z_dim):
        super(Encoder_z_u, self).__init__()

        self.Activation = nn.SELU(inplace=True)

        self.cn_1 = nn.Conv2d(1, 32, 4, stride=2, padding=1, bias=True)

        self.cn_2 = nn.Conv2d(32, 64, 4, stride=2, padding=1, bias=True)

        self.cn_3 = nn.Conv2d(64, 128, 3, stride=2, padding=1, bias=True)

        self.Dense_net_1 = nn.Linear(128*4*4, z_dim)


    def forward(self, x):
        z = self.cn_1(x)
        z = self.Activation(z)
        z = self.cn_2(z)
        z = self.Activation(z)
        z = self.cn_3(z)
        z = self.Activation(z)
        z = z.view(z.size(0), 128*4*4)
        z = self.Dense_net_1(z)
        z = torch.tanh(z)

        return z
```

```
class Encoder_zc(nn.Module):
    def __init__(self, c_dim, z_dim):
        super(Encoder_zc, self).__init__()

        self.c_dim = c_dim

        self.Activation = nn.LeakyReLU(inplace=True, negative_slope=0.2)

        self.cn_1 = nn.Conv2d(64, 128, 5, stride=2, padding=2, bias=True)

        self.Dense_net_1 = nn.Linear(128*4*4, c_dim)

        self.bn = nn.BatchNorm2d(128, eps=1e-5, momentum=0.1, affine=True)

    def forward(self, d_i):
        z = self.cn_1(d_i)
        z = self.bn(z)
        z = self.Activation(z)
        z = z.view(z.size(0), 128*4*4)
        z_c_logits = self.Dense_net_1(z)

        return z_c_logits


class Bandit(nn.Module):
    def __init__(self, c_dim):
        super(Bandit, self).__init__()

        # initialize c_dim clusters with logits all set to 1.0
        self.c_logits = P(torch.ones((c_dim)), requires_grad=True)

    def forward(self):
        return self.c_logits
```

## B.4 networks.py (Wheel)

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import Parameter as P
import pdb

class MultiGenerator(nn.Module):
    def __init__(self, z_dim, c_dim):
        super(MultiGenerator, self).__init__()

        self.c_dim = c_dim

        self.Generators = nn.ModuleList([Generator(z_dim) for i in range(c_dim)])

    def forward(self, z, zc):

        # create empty output tensor to populate
        out = torch.empty(z.size()[0],1,256,256).cuda()

        # create list of unique clusters in batch
        zc = zc.cpu().squeeze().numpy()
        clusters = np.unique(zc)

        # split each z noise input and feed through corresponding cluster's
generator
        # is there a way to do this in parallel?
        for i, c in enumerate(clusters):
            idx = torch.tensor(np.argwhere(zc == c)).cuda().squeeze()
            g_out = self.Generators[c](z.index_select(0, idx))
            out[idx] = g_out
        return out


class Generator(nn.Module):
    def __init__(self, z_dim):
        super(Generator, self).__init__()

        self.Activation = nn.SELU(inplace=True)

        self.Dense_net_1 = nn.Linear(z_dim, 256*4*4)

        self.cn_1 = nn.Conv2d(256, 256, 5, stride=1, padding=2, bias=True)

        self.cn_2 = nn.Conv2d(256, 128, 5, stride=1, padding=2, bias=True)

        self.cn_3 = nn.Conv2d(128, 64, 5, stride=1, padding=2, bias=True)

        self.cn_4 = nn.Conv2d(64, 64, 5, stride=1, padding=2, bias=True)
```

```
        self.cn_5 = nn.Conv2d(64, 32, 5, stride=1, padding=2, bias=True)

        self.cn_6 = nn.Conv2d(32, 1, 5, stride=1, padding=2, bias=True)


    def forward(self, z):

        x = self.Dense_net_1(z)
        x = self.Activation(x)
        x = x.view(x.size(0), 256, 4, 4)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_1(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_2(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_3(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_4(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_5(x)
        x = self.Activation(x)
        x = F.interpolate(x, scale_factor=2, mode='nearest')
        x = self.cn_6(x)
        x = torch.tanh(x)

        return x


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.Activation = nn.LeakyReLU(inplace=True, negative_slope=0.2)

        self.cn_1 = nn.Conv2d(1, 32, 5, stride=2, padding=2, bias=True)

        self.cn_2 = nn.Conv2d(32, 64, 5, stride=2, padding=2, bias=True)

        self.cn_3 = nn.Conv2d(64, 64, 5, stride=2, padding=2, bias=True)

        self.cn_4 = nn.Conv2d(64, 128, 5, stride=2, padding=2, bias=True)

        self.cn_5 = nn.Conv2d(128, 256, 5, stride=2, padding=2, bias=True)

        self.cn_6 = nn.Conv2d(256, 256, 5, stride=2, padding=2, bias=True)
```

```
        self.Dense_net_1 = nn.Linear(256*4*4, 1)

    def forward(self, x):
        d = self.cn_1(x)
        d = self.Activation(d)
        d = self.cn_2(d)
        d = self.Activation(d)
        d = self.cn_3(d)
        d = self.Activation(d)
        d = self.cn_4(d)
        d = self.Activation(d)
        d = self.cn_5(d)
        d_i = self.Activation(d)
        d = self.cn_6(d_i)
        d = self.Activation(d)
        d = d.view(d.size(0), 256*4*4)
        d = self.Dense_net_1(d)
        return d, d_i


class MultiEncoder(nn.Module):
    def __init__(self, z_dim, c_dim):
        super(MultiEncoder, self).__init__()

        self.c_dim = c_dim

        self.z_dim = z_dim

        self.Encoders = nn.ModuleList([Encoder_z_u(z_dim) for i in range(c_dim)])

    def forward(self, x, zc):

        # create empty output tensor to populate
        out = torch.empty(x.size()[0],self.z_dim).cuda()

        # create list of unique clusters in batch
        zc = zc.cpu().squeeze().numpy()
        clusters = np.unique(zc)

        # split each z noise input and feed through corresponding cluster's
generator
        # is there a way to do this in parallel?
        for i, c in enumerate(clusters):
            idx = torch.tensor(np.argwhere(zc == c)).cuda().squeeze()
            e_out = self.Encoders[c](x.index_select(0, idx))
            out[idx] = e_out
        return out


class Encoder_z_u(nn.Module):
```

```
    def __init__(self, z_dim):
        super(Encoder_z_u, self).__init__()

        self.Activation = nn.SELU(inplace=True)

        self.cn_1 = nn.Conv2d(1, 32, 4, stride=2, padding=1, bias=True)

        self.cn_2 = nn.Conv2d(32, 64, 4, stride=2, padding=1, bias=True)

        self.cn_3 = nn.Conv2d(64, 64, 4, stride=2, padding=1, bias=True)

        self.cn_4 = nn.Conv2d(64, 128, 4, stride=2, padding=1, bias=True)

        self.cn_5 = nn.Conv2d(128, 256, 4, stride=2, padding=1, bias=True)

        self.cn_6 = nn.Conv2d(256, 256, 3, stride=2, padding=1, bias=True)

        self.Dense_net_1 = nn.Linear(256*4*4, z_dim)


    def forward(self, x):
        z = self.cn_1(x)
        z = self.Activation(z)
        z = self.cn_2(z)
        z = self.Activation(z)
        z = self.cn_3(z)
        z = self.Activation(z)
        z = self.cn_4(z)
        z = self.Activation(z)
        z = self.cn_5(z)
        z = self.Activation(z)
        z = self.cn_6(z)
        z = self.Activation(z)
        z = z.view(z.size(0), 256*4*4)
        z = self.Dense_net_1(z)
        z = torch.tanh(z)

        return z

class Encoder_zc(nn.Module):
    def __init__(self, c_dim, z_dim):
        super(Encoder_zc, self).__init__()

        self.c_dim = c_dim

        self.Activation = nn.LeakyReLU(inplace=True, negative_slope=0.2)

        self.cn_1 = nn.Conv2d(256, 256, 5, stride=2, padding=2, bias=True)

        self.Dense_net_1 = nn.Linear(256*4*4, c_dim)
```

```
        self.bn = nn.BatchNorm2d(256, eps=1e-5, momentum=0.1, affine=True)

    def forward(self, d_i):
        z = self.cn_1(d_i)
        z = self.bn(z)
        z = self.Activation(z)
        z = z.view(z.size(0), 256*4*4)
        z_c_logits = self.Dense_net_1(z)

        return z_c_logits


class Bandit(nn.Module):
    def __init__(self, c_dim):
        super(Bandit, self).__init__()

        # initialize c_dim clusters with logits all set to 1.0
        self.c_logits = P(torch.ones((c_dim)), requires_grad=True)

    def forward(self):
        return self.c_logits
```

## B.5 utils.py

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('Agg')
from torch import autograd
import torch
import torch.nn as nn
import os
import tensorflow as tf
import datetime
from torch.distributions import Categorical
import torch.nn.functional as F
from sklearn.metrics import roc_curve, auc, precision_recall_curve
import torchvision
import pandas as pd
import torchvision.transforms.functional as TF
from torchvision import datasets
from torchvision import transforms
import skimage
from skimage import io
from torch.utils.data import Dataset, DataLoader, TensorDataset, random_split
import argparse
import pdb
import PIL

# Logger class used to create a Tensorboard logger with Pytorch
class Logger(object):
    def __init__(self, log_dir):

        log_dir += datetime.datetime.now().strftime("%Y%m%d-%H%M%S")

        self.writer = tf.summary.FileWriter(log_dir)

    def scalar_summary(self, tag, value, step):
        summary = tf.Summary(
            value=[tf.Summary.Value(tag=tag, simple_value=value)])
        self.writer.add_summary(summary, step)


# Xavier weight initialization
def weights_init(m):
    if isinstance(m, nn.ConvTranspose2d):
        if m.weight is not None:
            nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
    if isinstance(m, nn.Conv2d):
        if m.weight is not None:
            nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
```

```
            nn.init.zeros_(m.bias)
    if isinstance(m, nn.Linear):
        if m.weight is not None:
            nn.init.xavier_uniform_(m.weight)
        if m.bias is not None:
            nn.init.zeros_(m.bias)


# sampler function that generates batches of Z noise vectors and Z one hot cluster
vectors using Bandit
def z_sampler(Bandit, b_size, z_dim, c_dim):

    # Generate batch of z-vectors uniformly sampled from -1 to 1
    z_u = 2 * torch.rand(b_size, z_dim).cuda() - 1

    # Generate one hot encoded cluster vectors
    z_c_logits = Bandit()
    z_c_dist = Categorical(logits=z_c_logits).expand(batch_shape=(b_size, 1))
    z_c_index = z_c_dist.sample().cuda()
    z_c_oh = torch.FloatTensor(b_size, c_dim).zero_().cuda()
    z_c_oh.scatter_(1, z_c_index, 1)

    return z_u, z_c_oh, z_c_index

# Caculate gradient penalty for WGAN-GP discriminator loss
def calc_gradient_penalty(netD, real_data, fake_data, b_size):

    # Sample alpha for random interpolation
    alpha = torch.rand(b_size, 1, 1, 1).cuda()
    alpha = alpha.expand_as(real_data)
    alpha = alpha.view(b_size, 1, 256, 256)

    # Create interpolated images between real and generated images
    interpolates = alpha * real_data + (1 - alpha) * fake_data
    interpolates.requires_grad_(True)

    # Forward pass interpolates through discriminator
    disc_interpolates, d_i = netD(interpolates)

    # Calculate gradients of discriminator's output for interpolates
    gradients = autograd.grad(outputs=disc_interpolates, inputs=interpolates,

grad_outputs=torch.ones(disc_interpolates.size()).cuda(),
                              create_graph=True, retain_graph=True,
only_inputs=True)[0]

    gradients = gradients.view(b_size, -1)

    # Single sided gradient penalty that penalizes gradient norm > 1 with
coefficient of 10
```

```
    gradient_penalty = (torch.clamp((gradients.norm(2, dim=1) - 1), min=0) **
2).mean() * 10

    #pdb.set_trace()

    return gradient_penalty


# Cross entropy function for one hot z vectors compared to index labels
def cross_entropy_one_hot(input, target):
    _, labels = target.max(dim=1)
    return nn.CrossEntropyLoss()(input, labels)


def entropy(x):
    b = F.softmax(x, dim=1) * F.log_softmax(x, dim=1)
    b = -1.0 * b.sum()
    return b

# Save images for viewing
def save_images(images, save_dir, c, heatmap=False):

            for i, image in enumerate(images):

                # Convert tensor to numpy for output
                image = torch.squeeze(image).data.cpu().numpy()

                # Move channel axis to match matplotlib standard of channels last
                image = np.moveaxis(image, 0, -1)

                #rotate data 90 degrees for visualization
                image = np.rot90(image, k=1, axes=(0,1))
                image = np.flipud(image)

                if heatmap:
                    # Plot and save MSE heatmap for inspection
                    plt.imshow(image, cmap='jet', vmin=0, vmax=4)
                    plt.axis('off')
                    plt.show()
                    plt.savefig(save_dir + '/' + str(c) + '/' + str(i) +'.png')
                    plt.clf()
                else:
                    # Plot and save image for inspection in grayscale
                    plt.imshow(image, cmap='gray')
                    plt.axis('off')
                    plt.show()
                    plt.savefig(save_dir + '/' + str(c) + '/' + str(i) +'.png')
                    plt.clf()

# Process raw anomaly scores
```

```
def process_anomaly_scores(anomaly_error, normal_error, model_name):

    dir = './anomaly_plots/' + model_name + '/'

    if not os.path.exists(dir):
        os.makedirs(dir)

    a_values = np.zeros((101,))
    AUC_values = np.zeros((101,))
    PRC_AUC_values = np.zeros((101,))
    for j, alpha in enumerate(range(0, 101, 1)):
        alpha = alpha/100

        # Calculate scores for current alpha
        anomaly_scores = (1-alpha)*anomaly_error[:, 0] + alpha*anomaly_error[:, 2]
        normal_scores = (1-alpha)*normal_error[:, 0] + alpha*normal_error[:, 2]
        scores = np.concatenate((anomaly_scores, normal_scores), axis=0)

        # Create labels to go with anomaly scores
        labels = np.ones(np.size(anomaly_scores) + np.size(normal_scores))
        labels[np.size(anomaly_scores):] = 2

        #calculate precisoin and recall for current PR curve
        precision, recall, thresholds = precision_recall_curve(labels, scores,
pos_label=1)

        # Calculate FPR and TPR for current ROC curve
        fpr, tpr, thresholds = roc_curve(labels, scores, pos_label=1)

        # Calculate AUC for current ROC curve
        AUC = auc(fpr, tpr)

        #Calculate AUC for current PR curve
        prc_auc = auc(recall, precision)

        # Save results for current Alpha
        AUC_values[j] = AUC
        PRC_AUC_values[j] = prc_auc
        a_values[j] = alpha

    # Print max and min AUC values
    #print(np.max(AUC_values))
    #print(np.min(AUC_values))

    # print PRC AUC value at alpha=1.0
    print('At alpha=1.0, AUC = ' + str(AUC_values[100]))

    # Calculate scores for best Alpha
    max_idx = np.where(AUC_values == np.max(AUC_values))
    alpha = a_values[max_idx[0]]
```

```
    anomaly_scores = (1-alpha)*anomaly_error[:, 0] + alpha*anomaly_error[:, 2]
    normal_scores = (1-alpha)*normal_error[:, 0] + alpha*normal_error[:, 2]
    scores = np.concatenate((anomaly_scores, normal_scores), axis=0)

    # Calculate FPR and TPR for best ROC curve
    fpr, tpr, thresholds = roc_curve(labels, scores, pos_label=1)

    # Plot and save best AUC ROC curve
    fig, ax = plt.subplots(1, 1)
    ax.plot(fpr, tpr)
    ax.set_title('Best AUC ROC Curve (Alpha = ' + str(a_values[max_idx[0]][0]) +
')', fontsize=18)
    ax.set_xlabel('FPR', fontsize=14)
    ax.set_ylabel('TPR', fontsize=14)
    plt.show()
    plt.savefig(dir + 'Best_ROC.png')
    plt.clf()

    # Plot and save AUC v Alpha
    fig, ax = plt.subplots(1, 1)
    ax.plot(a_values, AUC_values)
    ax.set_title('AUC v Alpha', fontsize=18)
    ax.set_xlabel('Alpha', fontsize=14)
    ax.set_ylabel('AUC', fontsize=14)
    plt.show()
    plt.savefig(dir + 'AUC_v_Alpha.png')
    plt.clf()


# Filter that can be used to filter out model parameters based on layer name
def filter_params(model, filter):
    remaining = []
    filtered = []
    for name, param in model.named_parameters():
        if not param.requires_grad:
            continue
        if filter in name:
            filtered.append(param)
        else:
            remaining.append(param)
    return remaining, filtered

class WheelDataset(Dataset):
    def __init__(self, csv_file, root_dir, transform=None):
        self.annotations = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.annotations)
```

```
def __getitem__(self, index):
    img_path = os.path.join(self.root_dir, self.annotations.iloc[index, 0])
    image = io.imread(img_path)
    y_label = torch.tensor(int(self.annotations.iloc[index,1]))

    if self.transform:
        image = self.transform(image)

    return(image, y_label)
```

# References


[1] L. Moseley, "Introduction to machine learning," *Engineering Applications of Artificial Intelligence*, vol. 1, no. 4, p. 334, 1988. 1

[2] "Neural networks and other information processing approaches," *Introduction to Neural Networks*, p. 7–14, 1991. 1

[3] M. A. Nielsen, *Neural networks and deep learning*. Determination Press, 2015. 1

[4] C. C. Aggarwal, *Neural networks and deep learning: a textbook*. Springer, 2019. 1

[5] K.-L. Du and M. N. S. Swamy, *Neural networks and statistical learning*. Springer, 2019. 3

[6] Y. Shang and B. Wah, "Global optimization for neural network training," *Computer*, vol. 29, no. 3, p. 45–54, 1996. 3

[7] R. Sathya and A. Abraham, "Comparison of supervised and unsupervised learning algorithms for pattern classification," *International Journal of Advanced Research in Artificial Intelligence*, vol. 2, no. 2, 2013. 4

[8] T. Schlegl, P. Seeböck, S. M. Waldstein, U. Schmidt-Erfurth, and G. Langs, "Unsupervised anomaly detection with generative adversarial networks to guide marker discovery," in *International Conference on Information Processing in Medical Imaging*, pp. 146–157, Springer, 2017. 6, 7, 16

[9] T. Schlegl, P. Seeböck, S. M. Waldstein, G. Langs, and U. Schmidt-Erfurth, "f-anogan: Fast unsupervised anomaly detection with generative adversarial networks," *Medical image analysis*, vol. 54, pp. 30–44, 2019. 6, 7, 16, 27, 36, 37, 38, 39

[10] P. Seeböck, S. M. Waldstein, S. Klimscha, H. Bogunovic, T. Schlegl, B. S. Gerendas, R. Donner, U. Schmidt-Erfurth, and G. Langs, "Unsupervised identification of disease marker candidates in retinal oct imaging data," *IEEE transactions on medical imaging*, vol. 38, no. 4, pp. 1037–1047, 2018. 6

[11] P. Bergmann, M. Fauser, D. Sattlegger, and C. Steger, "Mvtec ad–a comprehensive real-world dataset for unsupervised anomaly detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9592–9600, 2019. 6, 7

[12] H. Blum, P.-E. Sarlin, J. Nieto, R. Siegwart, and C. Cadena, "Fishyscapes: A benchmark for safe semantic segmentation in autonomous driving," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019. 6

[13] S. Akcay, A. Atapour-Abarghouei, and T. P. Breckon, "Ganomaly: Semi-supervised anomaly detection via adversarial training," in *Asian Conference on Computer Vision*, pp. 622–637, Springer, 2018. 6, 16, 36, 37, 38, 39

[14] P. Gogoi, B. Borah, and D. K. Bhattacharyya, "Anomaly detection analysis of intrusion data using supervised unsupervised approach," *Journal of Convergence Information Technology*, vol. 5, no. 1, p. 95–110, 2010. 6

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2017. 6

[16] O. Calin, "Generative models," *Deep Learning Architectures Springer Series in the Data Sciences*, p. 591–609, 2020. 6

[17] S. J. D. Prince, *Computer vision: models, learning, and inference*. Cambridge University Press, 2012. 6

[18] Y. Bengio, L. Yao, G. Alain, and P. Vincent, "Generalized denoising auto-encoders as generative models," pp. 899–907, 2013. 7, 36, 37, 38, 39

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, pp. 2672–2680, 2014. 7

[20] H. Zenati, C. S. Foo, B. Lecouat, G. Manek, and V. R. Chandrasekhar, "Efficient gan-based anomaly detection," *arXiv preprint arXiv:1802.06222*, 2018. 7, 16, 36, 37, 38, 39

[21] A. Berg, J. Ahlberg, and M. Felsberg, "Unsupervised learning of anomaly detection from contaminated image data using simultaneous encoder training," *arXiv preprint arXiv:1905.11034*, 2019. 7

[22] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006. 9

[23] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017. 9

[24] J. Dai, Y. Lu, and Y. N. Wu, "Generative modeling of convolutional neural networks," *Statistics and Its Interface*, vol. 9, no. 4, p. 485–496, 2016. 9

[25] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, *A guide to convolutional neural networks for computer vision*. Morgan Claypool Publishers, 2018. 10

[26] "Activation functions used in neural networks," *Wiley Series in Adaptive and Learning Systems for Signal Processing, Communications, and Control Recurrent Neural Networks for Prediction*, p. 47–68. 11

[27] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," 2019. 12

[28] F. Di Mattia, P. Galeone, M. De Simoni, and E. Ghelfi, "A survey on gans for anomaly detection," *arXiv preprint arXiv:1906.11632*, 2019. 14

[29] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," *arXiv preprint arXiv:1701.07875*, 2017. 14, 21, 25

[30] M. Sabokrou, M. Khalooei, M. Fathy, and E. Adeli, "Adversarially learned one-class classifier for novelty detection," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 3379–3388, 2018. 15

[31] P. Perera, R. Nallapati, A. Ai, and B. Xiang, "Ocgan: One-class novelty detection using gans with constrained latent representations," *CVPR 2019*. 15, 36, 37, 38, 39

[32] D. Abati, A. Porrello, S. Calderara, and R. Cucchiara, "Latent space autoregression for novelty detection," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 15

[33] D. T. Nguyen, Z. Lou, M. Klar, and T. Brox, "Anomaly detection with multiple-hypotheses predictions," *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June, pp. 8418–8432, 2019. 15

[34] S. Akçay, A. Atapour-Abarghouei, and T. P. Breckon, "Skip-ganomaly: Skip connected and adversarially trained encoder-decoder anomaly detection," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019. 16

[35] F. Liu, X. Ren, Z. Zhang, X. Sun, and Y. Zou, "Rethinking skip connection with layer normalization," *Proceedings of the 28th International Conference on Computational Linguistics*, 2020. 16

[36] S. Mukherjee, H. Asnani, E. Lin, and S. Kannan, "Clustergan: Latent space clustering in generative adversarial networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4610–4617, 2019. 18

[37] M. Khayatkhoei, M. Singh, and A. Elgammal, "Disconnected manifold learning for generative adversarial networks," *CoRR*, vol. abs/1806.00880, 2018. 18, 22, 26

[38] C. Frogner, C. Zhang, H. Mobahi, M. Araya-Polo, and T. A. Poggio, "Learning with a wasserstein loss," *CoRR*, vol. abs/1506.05439, 2015. 25

[39] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in neural information processing systems*, pp. 5767–5777, 2017. 25

[40] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012. 30

[41] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017. 30

[42] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019. 32

[43] J. Huang and C. Ling, "Using auc and accuracy in evaluating learning algorithms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 3, p. 299–310, 2005. 35

[44] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen, "Ablation studies in artificial neural networks," *CoRR*, vol. abs/1901.08644, 2019. 40