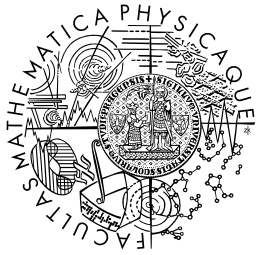

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Vojtěch Čermák

Adversarial examples design by deep generative models

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Roman Neruda, CSc
Study programme:	Master of Computer Science
Study branch:	Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 21.5.2021 signature of the author: Vojtěch Čermák

I would like to thank my supervisor, Roman Neruda, for the valuable advice he has provided throughout my long time writing the thesis. Completing this thesis would have been much more difficult were it not for his valuable insights.

Title: Adversarial examples design by deep generative models

Author: Vojtěch Čermák

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc, Department of Theoretical Computer Science and Mathematical Logic

Abstract: In the thesis, we explore the prospects of creating adversarial examples using various generative models. We design two algorithms to create unrestricted adversarial examples by perturbing the vectors of latent representation and exploiting the target classifier's decision boundary properties. The first algorithm uses linear interpolation combined with bisection to extract candidate samples near the decision boundary of the targeted classifier. The second algorithm applies the idea behind the FGSM algorithm on vectors of latent representation and uses additional information from gradients to obtain better candidate samples. In an empirical study on MNIST, SVHN and CIFAR10 datasets, we show that the candidate samples contain adversarial examples, samples that look like some class to humans but are classified as a different class by machines. Additionally, we show that standard defence techniques are vulnerable to our attacks.

Keywords: adversarial examples, generative models, deep learning, classification

Contents

1 Introduction **3**

2 Preliminaries **5**

2.1 Artificial neural networks 5

2.2 Deep generative models 6

2.2.1 Variational autoencoder (VAE) 7

2.2.2 Generative adversarial network (GAN) 8

2.2.3 Adversarially learned inference (ALI) 9

2.3 Adversarial examples 10

2.3.1 Adversarial attacks 11

3 Related work **13**

3.1 Deep generative models 13

3.1.1 Variational autoencoders 13

3.1.2 Generative adversarial networks 13

3.1.3 Flow based generative models 15

3.2 Adversarial examples 15

3.2.1 Adversarial attacks 15

3.2.2 Adversarial defences 17

4 Latent space perturbations **20**

4.1 Datasets 20

4.2 Models 21

4.2.1 Classifiers 21

4.2.2 Generative models 22

4.3 Interpolation based approach 24

4.3.1 Interpolation with bisection 26

4.3.2 Algorithm for extracting adversarial examples 28

4.4 Gradient based approach 30

4.4.1 Gradient method with bisection 33

4.4.2 Gradient method with mixed labels 34

4.4.3 Algorithm for extracting adversarial examples 36

5 Experiments **38**

5.1 Candidate images 38

5.2 Adversarial examples 42

5.2.1 MNIST 43

5.2.2 SVHN 45
5.2.3 CIFAR10 47
5.2.4 Adversarial examples for robust classifier 48

Conclusion **52**

Bibliography **55**

List of Figures **58**

List of Tables **59**

A Appendix **60**
A.1 Model Architectures 60
A.1.1 MNIST models 60
A.1.2 SVHN models 61
A.1.3 CIFAR models 61

1. Introduction

The undeniable success of deep neural networks in the past decade led to the widespread application of deep learning in practice. Application of those models seems to be limitless, from image recognition to language modelling, malware detection, financial fraud, reinforcement learning, computer vision, speech recognition, language processing and many other areas.

However, the widespread applications of those algorithms to real-life systems raised many security concerns as it creates incentives for adversaries to exploit weaknesses in those algorithms. In nightmare scenarios, we could witness adversaries create adversarial traffic signs designed to mislead autonomous vehicles, adversarial radio songs gaining control over voice-controlled assistants or malware undetectable by filters and many others. Those security concerns rose since Szegedy et al. [2014] found a simple way to create adversarial images for image classification using small perturbations on the input images. In this way, they created two images with differences imperceptible by the human eye, but each is classified differently by computer. Since then, many machine learning researchers have looked into the issue and discovered many new ways to attack machine learning models and defend them from the attacks. For example, Song et al. [2018] proposed a novel approach to the adversarial images, which is not based on small perturbation but creates adversarial examples from scratch. This approach bypassed most of the available defences to this date.

In this thesis, we would like to thoroughly explore how deep generative models can help us understand the nature of adversarial examples. Unveiling the enigma of the topic can help us design universally robust models that could lead to better safety in applying deep learning models. In order to construct reliable defences against adversarial attacks, we need to understand their nature better. From deep generative models, we can learn a lot about sampling and how new samples are created using machines.

To do so, we carry out several experiments that shed light on the topic. First, we explore how linear interpolation combined with bisection can extract images near the decision boundary of an arbitrary classifier. Such images can be subsequently utilized to create a set of potentially adversarial examples. Second, we take the idea behind FGSM of Szegedy et al. [2014], an algorithm used for creating perturbation in images, and apply it on vectors of latent representation. This approach allows us to use additional information from gradients to obtain images near the decision boundary, similarly to linear interpolation, and create another set of potentially adversarial examples. By manually inspecting their

class, we show that such sets contain adversarial images, samples that look like some class to humans but are classified as a different class by machines. We apply our insights to create adversarial examples for MNIST, SVHN and CIFAR10 datasets.

The thesis is structured as follows: The first chapter provides a detailed explanation of the methodology and key concepts used in the rest of the work. In the second chapter, we cover the relevant literature related to adversarial examples and the deep generative model. In the third chapter, we present several experiments to demonstrate how deep generative models can be utilized in adversarial attacks using two distinct methods. We apply our findings to design algorithms for creating adversarial examples. The fourth chapter is designated to discuss the results of introduced algorithms and their properties. In the last chapter, we summarise our findings and suggest topics for further research.

2. Preliminaries

2.1 Artificial neural networks

Artificial neural networks (ANNs) are a family of machine learning models inspired by biological neural networks. The structure of the network resembles how neurons and synapses in the biological brain work. From the computer science point of view, we can view ANNs as a graph with nodes representing neurons. The neurons are connected to each other by edges or synapses with corresponding weights. The synapses enable the transmission of the information signal from one neuron to another.

One of the most basic ANN is the perceptron, which is a model with a single neuron. Perceptron processes the information from incoming connections by summing the weighted signals. Additionally, a constant called bias is also added to the sum. In order to produce its output, the neuron passes the weighted sum of signals through an activation function, which is usually in the form of non-linearity. The most commonly used activation functions are arcus tangent and sigmoid functions. The weighted sum and activation function described the process of potential build-up and firing nerve impulse in biological neurons.

We can extend the idea behind perceptron and stack multiple perceptron neurons into layers. Neurons in multilayered perceptron are usually organized into several layers with different functions. The input layer receives external data, and the output layer produces an output which can be class prediction in case of classification or scalar prediction in case of regression. In between, there can be one or more intermediate layers, also called hidden layers. There are various types of hidden layers, and many have specialized function suitable to specific problems. The most simple layer type are fully connected layers, where each neuron in one layer is connected with every neuron in the next layer. Another common type is convolutional layers which learn weights in convolution kernels. This type of layer can utilize structure in data such as images or sound to efficiently learn patterns.

Most common ANNs are feedforward networks with a directed acyclic graph structure which allows only connections between neurons in adjacent layers. A more general type of ANNs is a recurrent neural network which allows connections between neurons in the same or any previous layers. This ANN type can be represented in the form of a directed graph that allows cycles.

Depending on a learning task, weights in the ANNs can be learned using various training algorithms. For example, in a supervised learning setting, the

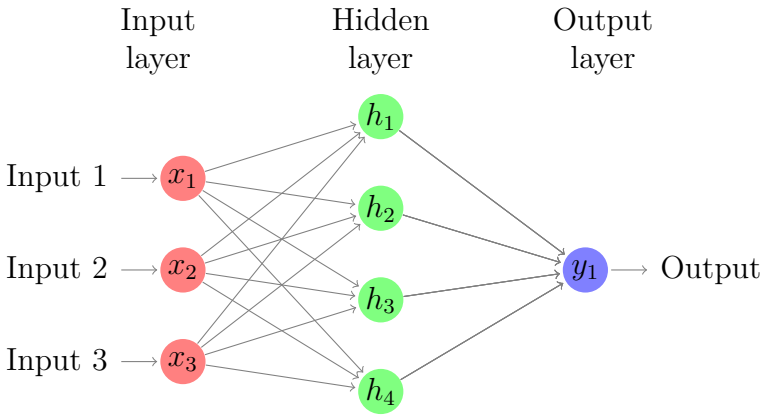

Figure 2.1: Schema of simple feed-forward ANN. It consists of an input layer with three neurons and a hidden layer with four neurons. A single neuron in the output layer can be used for binary classification or regression.

goal is to train the network such that its output is as close as possible to true labels. We can view the training task as an optimization problem where the goal is to minimize some loss function with respect to the network weights.

Most feedforward networks are designed with differentiable layers and activation functions that allow the backpropagation algorithm to efficiently compute the gradient of the loss function with respect to the network weights. Backpropagation uses chain rule for derivatives to iteratively calculate the gradient with respect to each weight, one layer at each iteration, starting from loss function and last layer of the network. Subsequently, the gradient calculated by backpropagation is used for updating the weights using gradient descent or its stochastic variants.

2.2 Deep generative models

Deep generative models are a class of unsupervised deep learning models designed to learn distribution some training and then can generate new data points from this distribution. There are three most commonly used classes of generative models, variational autoencoders (VAE) based on variational Bayesian inference, models utilizing the min-max game between generator and discriminator, such as the generative adversarial networks and flow-based generative models. The last class produces high fidelity samples but is computationally expensive when generating new samples. Given the goal of this thesis, we consider only variational autoencoders and generative adversarial networks as the significant computational expenses are a limiting factor in using the flow-based models.

2.2.1 Variational autoencoder (VAE)

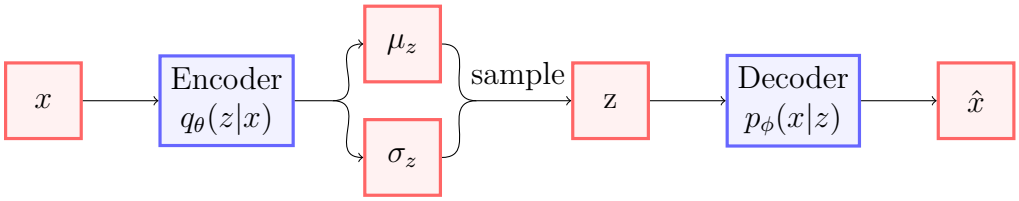

Figure 2.2: Schema of Variational autoencoder. The encoder takes samples from true data distribution x and returns μ_z and σ_z which are used for sampling z via reparametrization trick (Equation 2.3). The decoder takes z as input and returns reconstructed datapoint $\hat{x}$.

From a neural network point of view, a variational autoencoder consists of two neural networks, the encoder and the decoder. The encoder takes datapoints x from true data distribution as inputs and translates them into vectors of latent representation z. From a probabilistic point of view, the encoder network with weights and biases θ parametrizes the approximate posterior distribution $q(z \mid x)$ of the latent vectors z. Based on this, we note encoder as $q_\theta(z \mid x)$.

On the other hand, the decoder network reconstructs datapoints x given vectors of latent representation z. We can view the decoder network with weights and biases ϕ as parametrization for likelihood distribution $p(x \mid z)$ of data x given latent representation z. Hence, we note decoder as $p_\phi(x \mid z)$. Parameters of latent distribution sampled by the encoder are used as input to the decoder network.

The loss function of the VAE, as described in equation 2.1, is composed of expected negative log-likelihood with regularization term.

$$L(\theta, \phi) = -E_{z \sim q_\theta(z|x)} \log p_\phi(x \mid z) + KL(q_\theta(z \mid x) \mid\mid p(z)) \tag{2.1}$$

The first term is expectation of negative log-likelihood $\log p_\phi(x \mid z)$. We also call it reconstruction loss, as it tells us how well the decoder can reconstruct x given z, where the encoder network generates the z. Its exact formula depends on a training dataset. In our case, we train the VAE with the intent to use it for image generation. In this domain, binary cross-entropy and mean squared error are the most often used functions for the reconstruction loss.

The second term, the KL divergence, serves as regularization. It restricts encoder's distribution $q_\theta(z \mid x)$ to some prior distribution $p(z)$. By assuming diagonal unit Gaussian as the prior distribution, we can derive a closed-form solution of the KL divergence. In equation 2.2, J is the number of dimension of the latent space, μ and σ^2 are mean and variance parameters of Gaussian distribution, outputted by the encoder. In an implementation, the logarithm of

variance is usually used instead of variance.

$$-KL(q_\theta(z \mid x) \mid\mid p(z)) = -\frac{1}{2}\sum_{j=1}^{J}(1 + \log \sigma_j^2 - \mu_j^2 - \sigma_j^2) \tag{2.2}$$

Sampling datapoints z from encoder $q_\theta(z \mid x)$ uses random node z, which makes backpropagation impossible. We reparametrize the samples as described by equation 2.3. The samples deterministically depend on the parameters μ, σ and a new source of randomness ϵ. After sampling ϵ, there is no random component, making computation of derivatives needed for backpropagation tractable.

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim N(0, I) \tag{2.3}$$

2.2.2 Generative adversarial network (GAN)

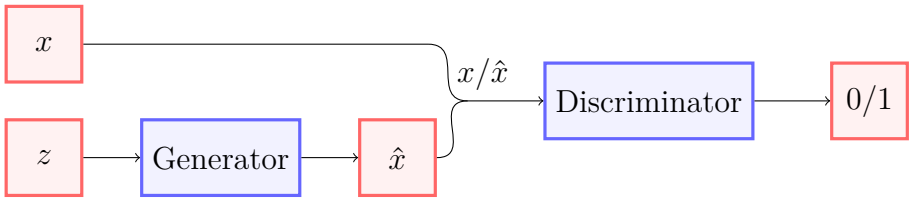

Figure 2.3: Schema of GAN training procedure. Generator creates new sample $\hat{x}$ given latent vector z. Discriminator classifies if input sample is generated or comes from true data distribution.

GAN model (Goodfellow et al. [2014]) converts the unsupervised task of training a generative model to a supervised learning task of two networks, the generator and the discriminator. Generator network $G(z)$ is trained to learn the data distribution and generate new synthetic samples. This is done by mapping vectors from latent space $z \sim p_z(z)$ to the space of training data. As prior for $p_z(z)$, multivariate Gaussian distribution is usually selected. As input, the generator takes points from lower dimensional latent space and outputs points resembling examples from the true data distribution. We can view the training of the generator as a learning process on how to encode high dimensional features from the original dataset to a vector in latent space. The architecture of the generator network usually consists of transposed convolutional layers with strides, batch normalization and ReLU or LeakyReLU non-linearity.

On the other hand, discriminator network $D(x)$ is trained to distinguish if each given sample x is from true data distribution $x \sim p_{data}(x)$ or is generated by the generator. Regarding network architecture, discriminators are standard binary classifier networks.

Both models are trained simultaneously by playing an adversarial game against each other. The discriminator is trained to classify fake and real images correctly, and the generator is trained to produce high fidelity images that fool the discriminator. We can formalize the min-max game by value function described in Equation 2.4.

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)} \log(D(x)) + E_{z\sim p_z(z)} \log(1 - D(G(z))) \quad (2.4)$$

2.2.3 Adversarially learned inference (ALI)

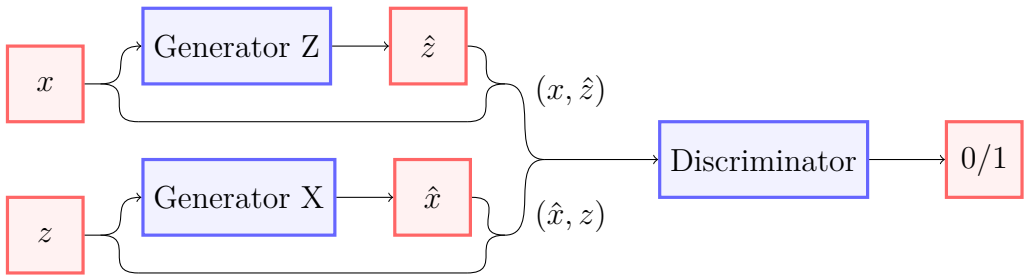

Figure 2.4: Schema of ALI training procedure. Generator X creates new sample $\hat{x}$ given latent vector z. Generator Z creates new sample $\hat{z}$ given true data sample x. Discriminator classifies if input is $(x, \hat{z})$ or $(\hat{x}, z)$ pair.

ALI model (Dumoulin et al. [2017]) augments original GANs with a method of encoding true data to latent vector. In ALI, a generator consists of two components, the decoder network $G_x(z)$ which maps data from latent space to input space in the same way as a generator in original GANs and the encoder network $G_z(x)$, which maps data samples x back to latent space. Regarding architecture, $G_x(z)$ works the same as in basic GAN architectures. On the other hand, the encoder consists of basic convolutional layers with strides in a similar fashion as an encoder network in VAEs.

The discriminator network in ALI is trained to distinguish between joint pairs $(x, G_z(x))$ and $(G_x(z), z)$. Discriminator is binary classifier network with one class for each pair. By playing the adversarial game, all three network are trained simultaneiously. Similarly to the GAN value function, the ALI game can be formalized by value function described in Equation 2.5:

$$\min_G \max_D V(D,G) = E_{x\sim q_{data}(x)} log(D(x, G_z(x))) + E_{z\sim p_z(z)} log(1 - D(G_x(z), z))) \quad (2.5)$$

By following the probabilistic terminology, we can define the encoder joint distribution $q(x,z) = q(x)q(z \mid x)$, and the decoder joint distribution $p(x,z) =$

$p(z)p(x \mid z)$, where marginal distributions $q(x)$ and $p(z)$ are empirical data distribution and prior distribution for z, respectively. In order to allow backpropagation, sampling from conditional distribution $q(z \mid x)$ is done with the utilization of the reparametrization trick, similarly to the VAE encoder. In an implementation, we rewrite the loss functions by using softmax functions, which are created by moving the last sigmoid layer from the discriminator to the loss functions. This change only simplifies the implementation as the loss functions remain effectively the same. We describe the ALI training procedure in detail in Algorithm 1.

Algorithm 1: ALI training procedure (Algorithm 1 from Dumoulin et al. [2017])

$\theta_g, \theta_d \leftarrow$ initialize weights for generators G_z, G_x and discriminator D ;
while *not convergence* **do**
  $x^{(i)} \sim q(x),\ i = 1 \dots M$; `// sample M points from dataset`
  $z^{(j)} \sim p(z),\ j = 1 \dots M$; `// sample M points from z prior`
  $\hat{x}^{(j)} \leftarrow p(x \mid z = z^{(j)})$; `// sample from the conditionals`
  $\hat{z}^{(i)} \leftarrow q(z \mid x = x^{(i)})$;
  $\rho_q^{(i)} \leftarrow D(x^{(i)}, \hat{z}^{(i)})$; `// calculate discriminator outputs`
  $\rho_p^{(j)} \leftarrow D(\hat{x}^{(j)}, z^{(j)})$;
  $L_g \leftarrow \frac{1}{M} \sum_{i=1}^{M} softplus(-\rho_q^{(i)})$; `// calculate losses`
  $L_d \leftarrow \frac{1}{M} \sum_{j=1}^{M} softplus(\rho_d^{(j)})$;
  $\theta_g \leftarrow \theta_g - \nabla_{\theta_g} L_g$; `// update weights`
  $\theta_d \leftarrow \theta_d - \nabla_{\theta_d} L_d$;

2.3 Adversarial examples

In the context of this thesis, we consider adversarial attack in the form of modifying target images by smallest possible perturbation to cause either targeted or untargeted misclassification.

There are several assumptions about the knowledge of the attacker. In the case of white-box attacks, we assume the attacker has complete knowledge of the model, including architecture, weights, activations and gradients. On the other hand, in the case of black-box attacks, the attacker has access only to the model's output for a given input. A standard approach to the analysis of adversarial attack methods is to perform the attack in white box settings and then extend it to the black box attack by utilizing its transferability property. Transferability allows for the straightforward construction of a black-box attack by first constructing the perturbations in the white box setting on the surrogate model and then transferring the perturbation to attack an arbitrary model in a

black-box setting.

Adversarial attacks are trivial when arbitrary large perturbation are allowed, as one can simply replace the original image with an image of the target class. Therefore, most perturbation based attacks are bounded by some norm. The norm describes differences between original and perturbed image and serves as an approximate measure of human perception of the attack. Most commonly used norms in adversarial attacks are l_0, l_2 and l_∞. Adversary attack with l_∞ norm restricts all pixels of the perturbed image to be within epsilon range $(x-\epsilon, x+\epsilon)$ to the original image x. In the case of l_2 norm, the perturbed image is bounded by Euclidean distance to the original image. Bounding the attack by l_0 norm can be interpreted as restricting the maximum allowed number of changed pixels in the original image to create the adversarial example.

2.3.1 Adversarial attacks

Authors of Goodfellow et al. [2015] introduced Fast Gradient Sign Method (FGSM), a method of generating perturbed adversarial examples bounded by l_∞ distance to the original image. It consists of shifting pixels to the direction determined by the gradient of target classifiers loss function with respect to the original image. The magnitude of the shift is determined by ϵ, as shown in Equation 2.6. Additionally, we need to clip pixel values after the shift to ensure it stays within the allowed pixel range. In our case, the pixel value should be within the $[-1, 1]$ interval.

$$x^* = clip(x + \epsilon sgn(\nabla_x L(x, y_{true}))) \tag{2.6}$$

We can easily modify FGSM to targeted attack by changing gradient sign to minus and selecting target classes as a label, as described in Equation 2.7.

$$x^* = clip(x - \epsilon sgn(\nabla_x L(x, y_{target}))) \tag{2.7}$$

Straightforward extension to FGSM is its iterative variant, the basic iterative method introduced in Kurakin et al. [2017a]. The method, as described in Equation 2.8, consists of taking several steps of FGSM with smaller steps with step size α, instead of a single large step. Again, we need to clip the values after each iteration to ensure they stay within the allowed range. To compare the performance of basic FGSM attack and its iterative variant, we can select α such that $\alpha N = \epsilon$, where N is a total number of iterations. In that way, after N iterations,

the attack is bounded by the same epsilon as in single-step FGSM.

$$\begin{aligned} x_0^* &= x \\ x_{t+1}^* &= clip(x_t^* + \alpha sgn(\nabla_{x_t^*} L(x, y_{true}))) \end{aligned} \tag{2.8}$$

Analogically to single-step FGSM, we can create a targeted variant of the basic iterative method using a vector of target labels and changing the gradient sign. Similarly to FGSM algorithms and the basic iterative method, which are l_∞ bounded attacks, we can create gradient-based attacks bounded by Euclidean norm or any other l_p norm. Analogically, we can create their targeted and iterative variant.

$$x^* = x + \eta \frac{\nabla_x L(x, y_{true})}{\|\nabla_x L(x, y_{true})\|} \tag{2.9}$$

3. Related work

In this chapter, we go through relevant literature. It includes development in research of deep generative models, such as Variational autoencoders, Generative adversarial networks and their variants. Then we cover highlights in research of adversarial attacks and defences.

3.1 Deep generative models

3.1.1 Variational autoencoders

Variational autoencoders (VAEs), introduced in Kingma and Welling [2014] and Rezende et al. [2014], are a class of generative models that augments vanilla autoencoders with the ability to generate new samples. In vanilla autoencoders, the decoder converts images to unknown latent distribution. Consequently, we do not know how to sample from this distribution, making it impossible to generate new valid samples. In VAE, we can restrict the latent distribution to the same prior distribution with known parameters.

VAEs use techniques based on variational Bayesian inference to learn the underlying probability distribution of training data. Instead of directly minimizing KL divergence between learned posterior and true posterior, they maximized evidence lower bound (ELBO) to bypass the need for calculating the intractable true posterior. Additionally, VAE uses the reparameterization trick to take randomness outside of the loss function, making its derivative tractable.

Further research developed several improvements of the original VAE model, such as beta-VAE of Higgins et al. [2017]. In their paper, they showed a new way to learn interpretable factorized latent representations using VAE. To do so, they generalized the original VAE by adding a new hyperparameter beta. The parameter beta is used as a weighting factor to balance the likelihood and KL-divergence term in the VAE loss function. In empirical experiments, the authors showed that correctly tuned beta is essential for the model's performance.

3.1.2 Generative adversarial networks

A fundamentally different approach to address the generative modelling problem is used in the generative adversarial networks, introduced in Goodfellow et al. [2014], and similar type of generative models. The authors used a min-max game between generator and discriminator to simultaneously train a generator capable of sampling from latent distribution and discriminator trained to distinguish fake

samples from real samples.

Original GAN had several down comings, such as unreliable training procedure with instability problems of learning and mode collapse, making selection hyper-parameters of the training procedure notoriously challenging. Since then, several improvements to GANs were proposed to fix issues of the original paper. For example, Wasserstein GAN introduced in Arjovsky et al. [2017], minimizes an efficient approximation of the Earth Mover's distance instead of the loss of traditional GAN. Incorporating Earth Move's distance in the loss function of the model created interpretable learning curves, improved training stability and reduced the problem of mode collapse.

Radford et al. [2015] augmented the original GAN model by utilizing convolutional neural networks in DCGAN architecture, significantly improving previous results in both quality and diversity of generated images. The DCGAN authors suggested adopting several architecture guidelines for building successful GANs. Their methods consist of using strided convolution instead of pooling layers, using batch normalization in both generator and discriminator, removing any fully connected layers for deeper architectures, and using ReLU activation for all layers except the output layer in the generator and using Leaky ReLU activation for all layers in discriminator.

Another property of the original GAN is that there is no inference to the model. As a result, we do not know much about what the model has learned, making it difficult to evaluate the trained model's quality. There were attempts to learn inference in GANs by using hybrids models, such as VAE-GANs developed by Larsen et al. [2016]. The authors placed the GAN discriminator in the VAE loss function, which solved the issue of VAEs with generating blurry images while retaining other properties of VAE, such as the inverse mapping.

A different approach for learning inference used independently Donahue et al. [2017] and Dumoulin et al. [2017] in their Bi-GAN and Adversarially Learned Inference (ALI) models, respectively. ALI or BiGAN models are synonymous methods of training inverse mapping to project data points back to latent representation. The general idea is to train two generator models simultaneously, encoder E is mapping data points to latent representation, and generator G is mapping points in latent representation to training distribution. Then, the min-max game is played between the two models and joint discriminator, where discriminator discriminates two pairs $(x, E(x))$ vs $(G(z), z)$, hence training both G and E simultaneously. They showed that it works well for standard datasets, such as SVHN, CIFAR10 and Tiny ImageNet.

3.1.3 Flow based generative models

Neither VAE nor GAN based models learn explicitly $p(x)$, the probability density function of real data. On the other hand, flow-based models are a method of directly estimating p(x) with the utilization of the flows, which can be viewed as learnable invertible preprocessing transforms of the dataset. For modelling complex probability densities, this concept was first used in Dinh et al. [2015] with the introduction of the Non-linear Independent Component Estimation (NICE) method. Their subsequent paper Dinh et al. [2017] extended the idea of NICE by introducing new invertible and learnable transformations, the real valued non-volume preserving transformations (Real NVP). The Real NVP model approximates posterior using a sequence of invertible transformations with suitable normalizations to transform simple distribution, such as Gaussian distribution, to arbitrarily complex distributions. This method allows for constructing an unsupervised deep learning model where the training criterion is tractable exact log-likelihood.

There have been many contributions to the research of flow-based models, such as the GLOW model proposed in Kingma and Dhariwal [2018]. They proposed a new type of generative glow in the form of invertible 1x1 convolutions. Empirical results showed the higher quality of synthesized images and better semantic manipulation than Real NVP. Further experiments on high-resolution natural images such as the CelebA dataset with 128x128 resolution showed that GLOW is efficient and scales well into a higher dimension.

3.2 Adversarial examples

3.2.1 Adversarial attacks

The first to notice the problem of adversarial examples in neural networks were Szegedy et al. [2014]. By applying small imperceptible non-random perturbations to test images, they were able to arbitrary change network predictions. Given an image and a classifier, they used the L-BFGS optimization algorithm to find a new image classified differently by the classifier but close to the original image under L2 distance. Once an adversarial example is found, its adversarial properties are robust and transferable to models trained with different hyperparameters and even trained on a disjoint training set of training data points. Their findings suggest the existence of intrinsic blind spots in the neural networks. Additionally, adversarial examples were never observed in the test set directly, but they found it near almost every test case.

Authors of Goodfellow et al. [2015] showed that the ability to create adversar-

ial examples for a given neural network is in the networks linear nature and can not be explained by nonlinearity and overfitting. Basic regularization strategies such as dropout and model averaging do not increase models robustness against adversarial attacks. On the other hand, changing nonlinearity to the RBF family improved robustness against them. Additionally, they showed a link between easy to optimize models and models susceptible to adversarial examples. Their second significant contribution was designing a simple method of generating samples, the Fast Gradient sign method (FGSM). It is constructed to produce examples fast and efficiently, and unlike the L-BFGS method of Szegedy et al. [2014], which is l_2 bounded, FGSM is bounded by the l_∞ metric. Finally, the authors used the FGSM as an efficient and fast method to generate new adversarial examples and used them for the adversarial training defence technique. Empirical results showed that adversarial training successfully decreased the effectiveness of adversarial attacks.

The basic iterative method is a natural extension of the FGSM method and was first described in Kurakin et al. [2017a]. Instead of taking a single step of size ϵ, they iteratively take multiple smaller steps of FGSM and clip pixel values after each iteration to ensure it stays within ϵ. In their subsequent research, Kurakin et al. [2017b], they showed that adversarial training defence does not work well against attack based on iterative methods. On the other hand, adversarial examples generated by iterative methods are less transferable to other models, making them weaker in black-box adversarial attacks.

Papernot et al. [2015] argued for using l_0 distance to model human perception and introduced Jacobian based saliency maps attack (JSMA), a class of attacks optimized under l_0 distance. JSMA utilizes the Jacobian matrix to compute the saliency map, which shows us the impact of each pixel on the classification. Then, they use a greedy iterative algorithm to alter the most important pixels to increase the likelihood of misclassification target class.

Authors of Carlini and Wagner [2017] provided guidelines for creating attacks and how to create good validation of any defences techniques. Their suggestions consist of always using targeted white-box attack, as untargeted attacks are too weak and black box attack can always be created from white-box attack using surrogate and transferability of adversarial attacks. To show that current defence methods are not sufficiently strong enough, they designed new attack algorithms based on l_0, l_2 and l_∞ distances. Their attacks perform well against various defences, and they suggested using them as benchmarks.

The unrestricted adversarial examples, the new thread model introduced by in Song et al. [2018], is fundamentally different from the earlier thread models. In previous attacks, small perturbation was used to ensure the class remains

unchanged for human perception. Methods for creating unrestricted adversarial examples do not use small perturbations but utilize deep generative models to generates adversarial examples from scratch. The change in the original picture can be significant, but a human can still see the original class. Authors use Auxiliary Classifier Generative Adversarial Model to model a class distribution over data samples. Then, conditioned on class, they search latent space until they find an example misclassified by target classifiers. In this way, all traditional methods of defences expecting models based on small perturbations are bypassed. In order to evaluate the performance of the attack, they used visual evaluation by human judges.

Zhang et al. [2019] suggested a new adversarial attack model, the blind spot attack. They showed that adversarial examples are concentrated and can be created more easily in blind spots, where real dataset distribution have a low number of observations. Additionally, in high dimensional datasets, it is easier to find blind spots due to the curse of dimensionality and data point scarcity, which consequently makes defences such as adversarial training more difficult. They suggested blind spots as the underlying cause of adversarial examples, as the model cannot reliably learn properties of rare examples.

Authors of Xiao et al. [2019] in their AdvGAN model used Generative Adversarial Network to generate adversarial perturbation. The model consists of generator $G(x)$ and discriminator, which discriminates between original image x and $x + G(x)$. Both generator and discriminator are trained using an adversarial game, similarly to basic GAN. Unlike the usual attacks based on slow optimization procedure to attack a single instance each time, such as attacks described in Carlini and Wagner [2017], they used feed-forward networks as a fast way to generate the perturbations. Their attack technique showed the state of the art performance for the black-box type of attack in the MNIST attack challenge presented by Madry et al. [2019].

3.2.2 Adversarial defences

Since the discovery of adversarial attacks, there have been attempts to create models robust to them. One of the first defences techniques, introduced in Goodfellow et al. [2015], was adversarial training. One way to do adversarial training is to augment the training set with adversarial examples and then sample batches consisting of samples from both adversarial examples and the original training dataset during training. Different adversarial training approaches use modified loss function to ensure that the model predicts the same class for both original images and perturbed adversarial examples. The major downside of adversarial

training is that it increases robustness only for a single model that needs to be specifically trained. The second issue is that the robustness is increased predominantly for the specific attacks used to augment the training dataset, and the model is still weak against different types of attacks. Subsequently, adversarial training performs poorly against black-box attacks.

Authors of Hosseini et al. [2017] focused on blocking the transferability property of adversarial examples, which is an essential prerequisite for constructing defences against black-box attacks. The idea is similar to adversarial training, but they added a NULL label to the dataset, which allowed the classifier to classify any given image as invalid. The classifier is trained with both clean images from the original dataset and images perturbed to various degree. Compared to standard labels, the NULL label increases with more perturbed images and is zero for clean images. Empirical results showed a good compromise between resistance against attacks and accuracy on clean images.

Papernot et al. [2016] used the distillation technique to reduce the effectiveness of adversarial examples. The distillation process consists of two steps. In the first step, we train a classifier on clean data. The softmax activation in the last layer of the first model is then used as labels for training the second classifier. In this way, more information about the class of original images is used during the training. Empirical experiments showed that using soft labels from the original classifier to label the second classifier's dataset significantly reduces the effectiveness of white-box methods. However, this method is not efficient against black-box attacks. Additionally, newer attack methods such as attacks from Carlini and Wagner [2017] are able to confuse the classifier even when trained using distillation.

A different approach is used by authors of Meng and Chen [2017]. They proposed MagNet, a defence method consisting of one or more detector networks and a reformer network. The detector networks classify whether an example is adversarial or natural based on the reconstruction error of the autoencoder. The Reformer network then reconstructs adversarial examples so that they are close to the natural examples and, at the same time, does not change the natural examples significantly. The reformer network based on autoencoder reconstructs natural examples near the original examples and reconstruct adversarial images closer to the manifold of natural examples. In empirical experiments, MagNet shows excellent results against black-box attacks but works poorly against white-box attacks. In order to prevent the white-box attack, to authors suggested using a random autoencoder from a pool of pre-trained autoencoders.

A similar idea was used in the Defense GAN introduced in Samangouei et al. [2018], but instead of using autoencoders, they utilized the power of generative adversarial networks to reduce the effectiveness of both white box and black

attacks. The core idea is to find the best latent representation of image x in the latent space of the GAN model trained on clean data. The latent representation of the original image is then used to generate a new image without adversarial perturbations because it was generated using a generator trained on clean data. To find the suitable representation z, they used gradient descent optimization to minimize the distance between the original image x and the reconstructed image $G(z)$ as the criterion. Defence GAN can be used to defend any classifier, as it works as preprocessing step and does not change the structure of the classifier. The downside of the Defense GAN is its reliance on the GAN's quality and generative power. Therefore, any potential application depends on the success of GAN training, which can be a challenging task.

Authors of Madry et al. [2019] approached the problem of adversarial examples from a robust optimization point of view. They provided strong evidence that first-order methods provide enough information to the optimization to train robust network and proposed iterative variant of FGSM, the projected gradient descent attack (PGD), as the universal first-order adversary. To support their claim, they published a public challenge on their web page and showed that iterative variants of gradient-based attacks, such as the PGD, are comparable to the state of the art attacks, even in their basic form without any hyperparameter tuning. They propose to use iterative methods instead of single step FGSM to create adversarial examples used in adversarial training. They showed that adversarial training could be used as universal defences against most perturbation based attacks. Additionally, their model with this type of defence is published on their webpage as part of the public challenge.

4. Latent space perturbations

In this chapter, we would like to explore how the image class changes if we do perturbations in latent space instead of image space in a series of empirical experiments. Using latent space enables us to take a different approach to create adversarial images based on creating perturbations in higher-level features instead of those in the form of small perturbations like those created by FGSM. Using the MNIST dataset as an example, this approach allows us to create adversarial images by manipulating features such as digit slant, height, width, line thickness and image blurriness. Source codes for all experiments can be found in the public repository https://github.com/VojtechCermak/thesis-adv-examples.

4.1 Datasets

For our experiments, we used a standard MNIST dataset of grayscale handwritten digits introduced in LeCun et al. [2010]. MNIST images are of size 28x28 pixels with a single grayscale channel. The machine learning community has been extensively using MNIST for developing prototypes and testing new concepts due to its simplicity. The dataset contains 60000 training images and 10000 test images divided into ten classes, with one class for each digit. In order to scale the values to (-1, 1) interval, we transform the dataset by subtracting -0.5 from each pixel value and multiplying it by two.

Working with MNIST, one must be careful to interpret the results, as MNIST does not represent real-life image distributions well enough. In order to determine if our hypotheses relate to real-world problems, we use the SVHN (Street View Home Numbers) dataset of Netzer et al. [2011] to validate our results. It consists of cropped digits from house numbers obtained from Google Street View images. It consists of 73257 train samples and 26032 test samples separated into ten classes, with one class per digit. Similarly to MNIST, we scale the pixel values in each channel to (-1, 1) interval.

To further validate our hypotheses, we use well known CIFAR10 dataset from Krizhevsky [2009]. The dataset consists of 60000 real-life images downscaled to 32x32 and carefully selected to be recognizable by humans. There are 50000 train images and 10000 test images in 10 classes, including classes of animals (dog, cat, horde etc.) and human machines (car, ship, train, etc.). Again, we scale each pixel value to (-1, 1) interval.

Figure 4.1: Randomly sampled images from MNIST, CIFAR10 and SVHN datasets.

4.2 Models

4.2.1 Classifiers

To study the effect of different attacks, we train several classifiers for each dataset as attack targets or as an auxiliary classifier to label samples generated from unconditional generators. Our classifiers are based on the VGG architecture described in Simonyan and Zisserman [2015]. It is composed of several (2 in case of MNIST and 3 in case of SVHN and CIFAR10) convolutional blocks followed by three fully connected layers and an output layer with softmax activation. Each convolutional block consists of two convolutional layers with batch normalization and ReLU activation, followed by max pool and dropout layers. A detailed description of model architectures and used optimization parameters is in the appendix.

Additionally, we use both natural and adversarially trained robust pre-trained models from Madry et al. [2019] public challenge. Using those models allows us to evaluate the performance of our attack against commonly used defence methods and compare the performance against the MNIST benchmark.

Table 4.1: Performance of classifiers

Model	No adversary	FGSM 0.1	FGSM 0.2	iFGSM 0.1	iFGSM 0.2
Our classifiers					
SVHN	0.9537	0.1617	0.0974	0.0022	0.0000
MNIST	0.9950	0.7344	0.2098	0.2526	0.0004
CIFAR	0.8572	0.0699	0.0870	0.0000	0.0000
Classifiers of Madry et al. [2019]					
MNIST Natural	0.9917	0.8061	0.5437	0.5539	0.0401
MNIST Robust	0.9840	0.9795	0.9710	0.9556	0.8536

Overview of the accuracy of classifiers, measured on the unmodified test dataset and against both FGSM and iterative FGSM adversaries.

An iterative variant of FGSM creates a much more efficient attack, as can be seen from the Table 4.1. The power of the attack is especially apparent in the case

of complex high dimensional datasets such as CIFAR, where the iterative attack causes misclassification in all test cases. On the other hand, the performance of a robust classifier is diminished by the attacks only slightly.

4.2.2 Generative models

In order to explore the possibility of attacks by doing modifications in latent space, we need a good mapping from latent space with high fidelity generated images. It should have smooth transitions between classes because humans can detect artefacts and unnatural distortions. To find such latent space, we explore three types of generative models with various hyperparameter settings. To evaluate the quality of learned latent space and to tune hyperparameters of generative models, we used two distinct metrics.

The first metric used is the Sliced Wasserstein distance (SWD) metric introduced in Karras et al. [2018]. It uses a fast approximation of Wasserstein distance between two distributions. In our case, we are interested in the distance between the distributions of original images and generated images. The SWD metric consists of representing both original and generated images by using the Laplacian pyramid and then calculating the similarity of small image patches from various pyramid levels. Measuring similarity on different levels of the Laplacian pyramid allows us to measure similarity on different levels of resolution.

To double-check our results, we use a metric similar to the Inception score from Salimans et al. [2016], but instead of the Inception classifier, we use our pretrained classifiers. The metric is calculated from class probabilities predicted by a classifier as KL Divergence between the conditional and marginal class probability distributions.

Regarding ALI architecture, we followed the ideas in the original papers as close as possible with minor changes in CIFAR, where we replaced maxout activations by Leaky RELU. Because ALI authors did not include model for the MNIST dataset, we created our own by following the guidelines they suggest for other datasets. A detailed description of model architectures and used optimization is in the appendix.

Table 4.2: Performance of generative models

Model	IS	SWD (level 0)	SWD (level 1)
MNIST			
ALI	9.5507	0.0581	-
DCGAN	9.4440	0.0536	-
VAE	8.7980	0.2628	-
SVHN			
ALI	8.0685	0.1899	0.3812
DCGAN	8.0234	0.2464	0.2328
VAE	7.3671	0.9484	0.3696
CIFAR			
ALI	6.8935	0.1270	0.2949
DCGAN	6.8181	0.1559	0.2587
VAE	3.1773	1.0474	0.3725

Performance of generative models measured by a metric based on Inception Score (IS) and Sliced Wasserstein distance (SWD) at various levels of Laplacian pyramid.

The Table 4.2 suggests that ALI models have quality slightly better than basic GANS with DCGAN architecture, and VAE models are significantly worse. In the case of the CIFAR10 dataset, VAE was unable to generate any images that are recognizable by humans. On the other hand, VAE models significantly outperform ALI in the test set reconstruction quality measure by Euclidean distance.

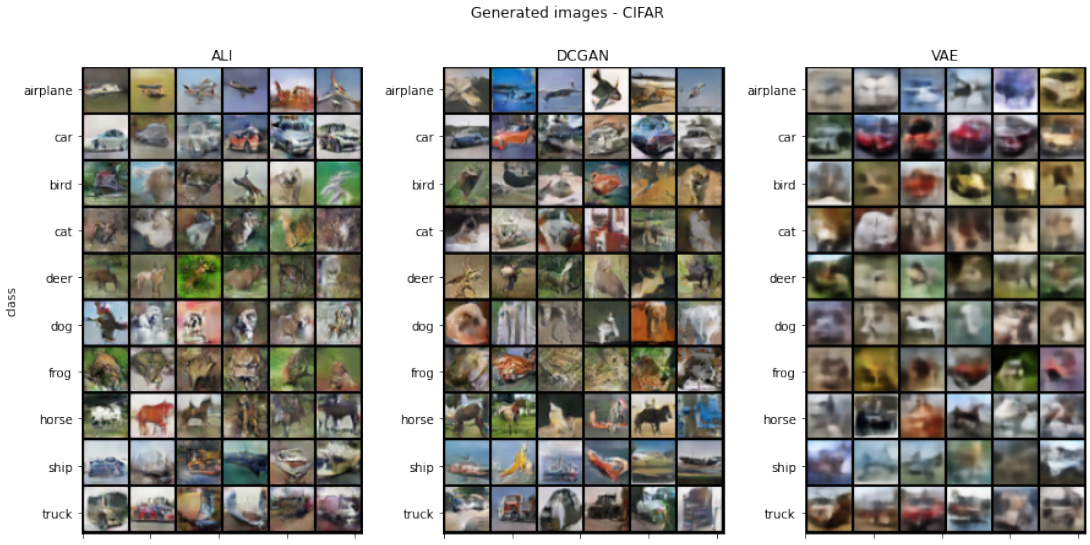

Figure 4.2: CIFAR10 images generated by generative models. Based on the classifier's output, we assign a class to the image if the softmax of the class is greater than 0.5.

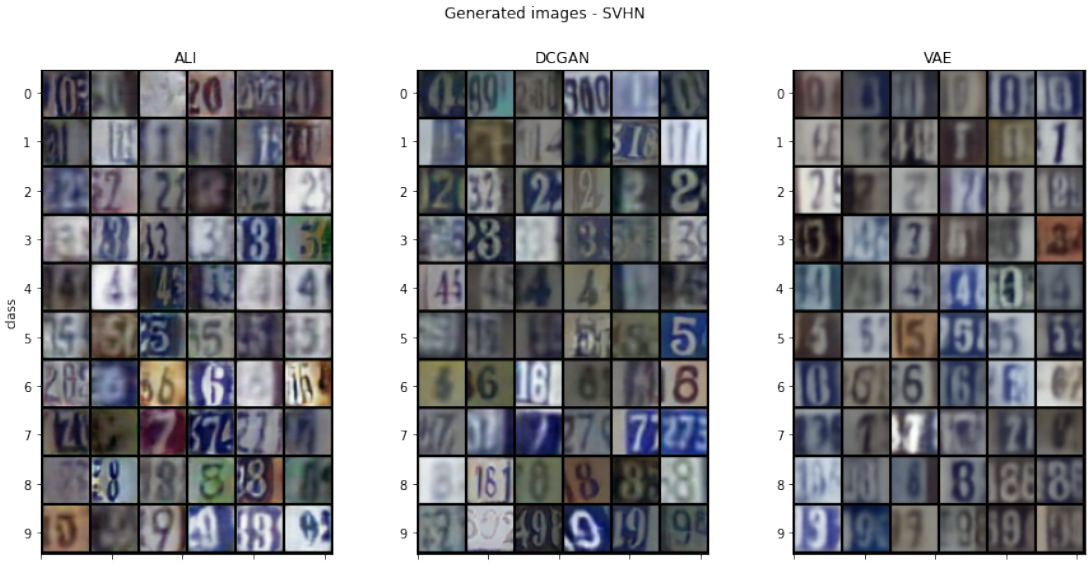

Figure 4.3: SVHN images generated by generative models. Based on the classifier's output, we assign a class to the image if the softmax of the class is greater than 0.5.

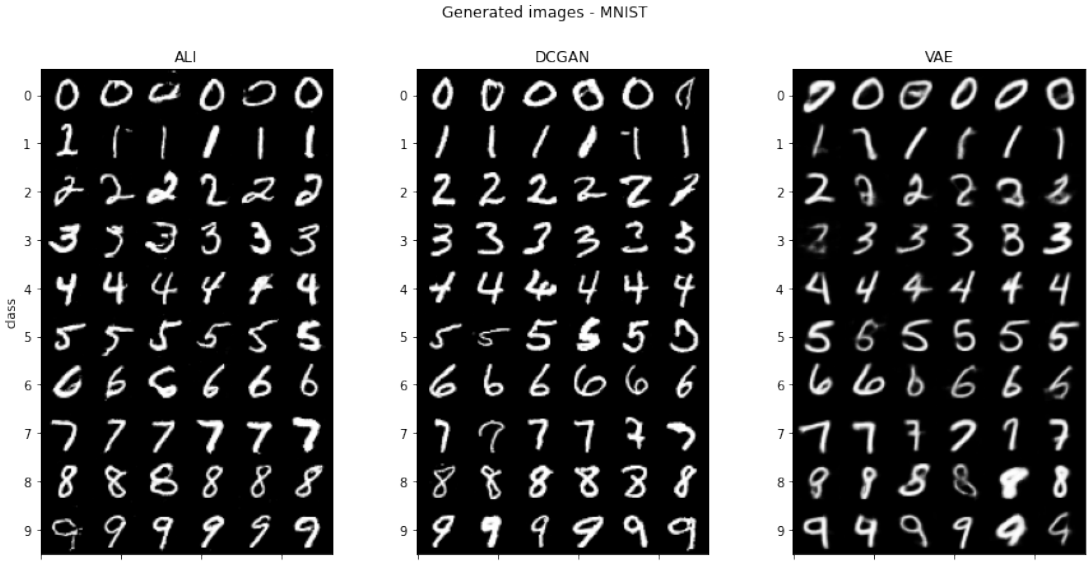

Figure 4.4: MNIST images generated by generative models. Based on the classifier's output, we assign a class to the image if the softmax of the class is greater than 0.5.

4.3 Interpolation based approach

A simple case of manipulations in latent space to change image class is a linear interpolation of latent vectors. Given vectors of latent representation of the original image and target image with a different class, we construct new datapoints as points on the lines created by linear interpolation of every element of those

vectors. We can view the new samples as steps between the original image and the target image with the interpolation step parameter from interval $[0, 1]$, where the original image is 0, and the target image is 1.

In the first few steps, samples are likely to be classified as the original class because the images are close to the original image. After enough steps, the samples are classified as the target image class as we go through a classifier's decision boundary between the two classes. In that way, we can create a sample that lies at the classifier's decision boundary. We are interested in finding such a sample because we expect the classifier to be sensitive to small changes in the sample when it is near the decision boundary. This sensitivity can potentially cause the classifier to misclassify samples that a human evaluator would classify correctly.

To illustrate this process, we design the following example. We generate images using one of the pre-trained generators. Since those images are generated unconditionally without labels, we need to label them using an auxiliary classifier. In order to prevent sampling already ambiguous or distorted samples, we consider an image to be a member of some class if the auxiliary classifier's softmax output for this class is greater than 0.99. Then, we use linear interpolation to create 20 samples as evenly spaces steps between the original image and target image, and we show how the softmax output of the auxiliary classifier changes.

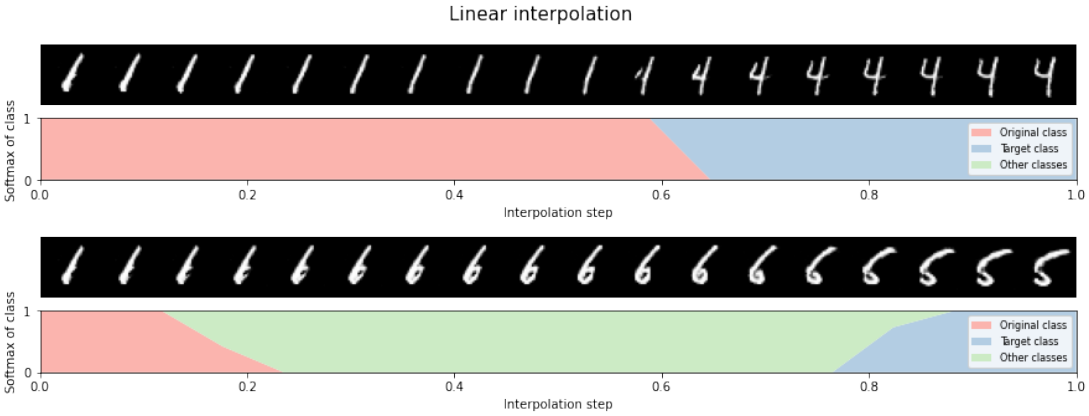

Figure 4.5: The effect of gradually increasing steps of linear interpolation on of both image and classifier's softmax output for the image. The bottom half shows how softmax changes for the original image class (in red) and target image class (in blue). The Sum of softmax activations for all other classes is shown as green. We used latent representations from the ALI model.

As is shown in Figure 4.5, at the beginning of the interpolation, the images closely resemble the original ones and are classified as the same class. Eventually, with increasing interpolation steps, the images transform from original images to target images, and the classification also changes.

Figure 4.5 also shows two potential problems with a linear interpolation that can prevent us from sampling images lying at the decision boundary between the two classes. The first issue can be observed in the first example of Figure 4.5, where the transition between the two images is not a gradual and significant part of the change is done in a few steps. We can see the digit 1 gains features of digit 4 in only two steps. The second issue with linear interpolation is that the images obtained via interpolation can go through different classes than the original and the target classes. We can observe this behaviour in the second example of Figure 4.5, where the original sample with digit 1 first transforms to samples classified as digit 6, although we target digit 5. After few more steps, the image finally transforms to digit 5 as expected.

4.3.1 Interpolation with bisection

In the previous example, we selected interpolation steps such that they are spaced evenly between 0 and 1. The resulting softmax activations are not gradual and steeply decrease as we get near the decision boundary. In order to view samples that correspond to any given original class softmax, we would like to select the interpolation steps such that the softmax values are spaced evenly.

Given the classifier and latent representations of original sample and target sample, we can design function $f(step)$ that takes the interpolation step as input and outputs a single scalar, the softmax of the original sample class. When the interpolation step is zero, the function has value $f(0) = 1$ as the interpolation corresponds to the original sample. By increasing the interpolation step up to one, the function values go to 0 as we interpolate to values closer to the target sample. Although this function is very steep for values around the decision boundary, it is continuous. Therefore, we can use the bisection (as described in Algorithm 2) on this function to obtain interpolation steps that correspond to arbitrary softmax of the original class, even for those in the steep regions.

Algorithm 2: Bisection algorithm

Input : a, b - initial endpoint values, f - continuous function defined on interval $[a, b]$, *target* - target value in domain of f, *threshold*, *maxiter*.

Output: c, such that f(c) differs from *target* by less than *threshold*

```
while i < maxiter do
    c = (a + b) / 2;
    if abs(f(c) − target) < threshold then
        return c ;
    if f(c) < target then
        a = c ;
    else
        b = c ;
```

We recreate the previous example (Figure 4.5), but now we use bisection to find interpolation steps that evenly space original class softmax activations. We keep the same original and target images with the same latent representations as in Figure 4.5. For endpoint values, we select $a = 0$ and $b = 1$ as minimal and maximal allowed values of the interpolation step. As *target*, we have the target softmax of original class. Additionally, we select $threshold = 0.001$ and $maxiter = 100$.

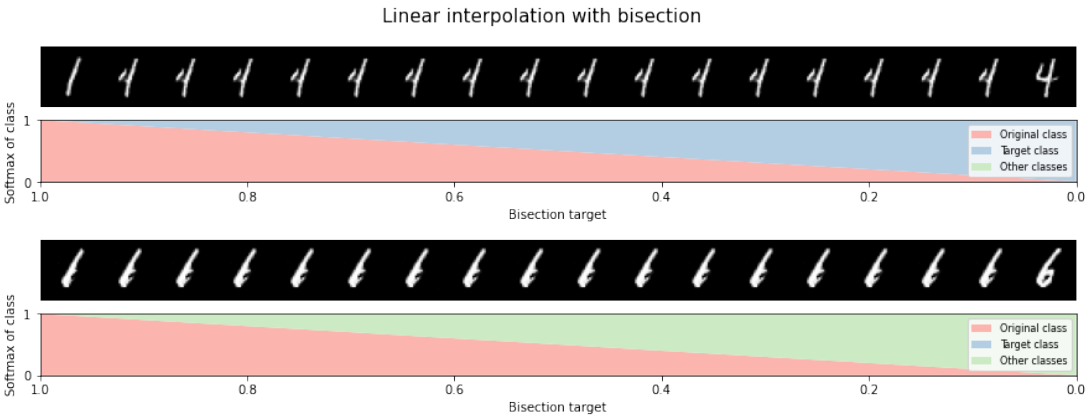

Figure 4.6: The effect of linear interpolation with bisection on both image and classifier's softmax output for the image. The steps of linear interpolation are selected using bisection such that they match the target, which is the softmax of the original class. We initially target high values of softmax of the original class and then gradually decrease the target. The bottom half shows how softmax changes for the original image class (red) and the target image class (blue). The sum of softmax activations for all other classes is shown as green. We used latent representations from the ALI model.

As shown in Figure 4.6, samples obtained using the bisection algorithm have the softmax activations spaced evenly between $[0, 1]$ interval. Bisection allows us to inspect samples located in the steep regions that correspond to the classifier's decision boundary. Such samples would be otherwise difficult to identify and visualize, as we showed in the previous example.

From both examples in Figure 4.6, we can see that the most noticeable changes happen between the first two images and the last two images. In the first example, digit 1 gains some of the attributes of digit 4 between the first and second image. By further decreasing the bisection target, we can observe only slight changes as we navigate directly through the decision boundary. In this region, the classifier is sensitive to small changes in the inputs. Such small changes result in significant changes in softmax activations, while the samples are nearly indistinguishable from each other by human evaluation. Finally, the digits on the image gain rest of the digit 4 features between the last two images, as we leave the sensitive decision boundary.

In the second example of Figure 4.6, we show that we cannot go through samples with different classes than the target classes. This is because the bisection stops when the softmax of the original class is near the value we targeted and does not take into account the softmax value of the intended target sample class. In the second example, we want to transform digit 1 to digit 5, but the algorithm stops as the sample changes to digit 6. The bisection targets are satisfied, although we originally intended to go all the way to digit 5, as in Figure 4.5.

4.3.2 Algorithm for extracting adversarial examples

To generate adversarial examples, we can utilize observation that all samples with original class softmax in the interval $[0.95, 0.05]$ looks visually similar to a human classifier. For example, a human classifier would have difficulty distinguishing between samples with the original class softmax of 0.3 (and 0.7 of some other class) and 0.7 (and 0.3 of some other class). However, the machine classifier would take an index of maximal softmax activation and classify both of those samples differently. Hence, the main idea is to use linear interpolation with bisection to sample various images with softmax output for the original class smaller than 0.5 and then search in those images for images that would humans classify as the original class.

We denote $S_i(x)$ as softmax activation for class C_i of a classifier $f(x)$ on which we target the adversarial examples. The procedure can be described in Algorithm 3:

Algorithm 3: Linear interpolation with bisection attack

Input : $G(z)$ - generator, $f(x)$ classifier under attack, C_1 - original class, C_2 - target class, m - softmax target for C_1

Output: x^* - candidate for adversarial example

Steps :

1. Using $G(z)$, generate samples x_1, x_2 with latent representations z_1, z_2 and classes C_1, C_2, such that $C_1 \neq C_2$.

2. Use bisection with linear interpolation between latent vectors z_1, z_2 to find optimal step such that $S_1(x^*) = m$, where the optimal step corresponds to latent vector z^* which represents sample x^*.

3. Attack is successful if for x^* holds $|S_2(x^*) - (1 - m)| < \epsilon$, where tolerance parameter ϵ is small value near zero.

In Step 1 of Algorithm 3, we use an unconditional generator to generate the samples. Although we can use an arbitrary auxiliary classifier or human judgement to label the unconditionally generated samples, we suggest using the target classifier for the labelling to reduce additional overhead costs associated with training another classifier.

In Step 2, we want to select bisection target $m < 0.5$ so the x^* is classified as C_2 by the classifier. For example, we can use the bisection to target $m = 0.3$. In this way, we create examples with $S_1(x^*) = 0.3$ that are not classified as the original class but are still in the vicinity of the decision boundary.

Step 3 filters out cases such as the second example in Figure 4.6, and it ensures that the transformation does not result in anything other than the target class. In case of unsuccessful transformations, the attack needs to be repeated if we want samples for given input parameters.

Algorithm 3 allows us to identify adversarial images that are not restricted by any norm. We use human evaluation to manually select images that humans would classify as images of the original class from the successful samples. Given an adequately trained generator, we can assume both vectors of z_1 and z_2 represents valid images. Because the prior distribution of z is multivariate standard normal distribution, we can expect all images represented by latent vectors z^* from interpolation between z_1 and z_2 to be valid images as well. Therefore, we expect our procedure to produce valid images.

4.4 Gradient based approach

The linear interpolation does not tell us the optimal way how to traverse from one class to another in the latent space. We can utilize additional information to calculate better steps of transformation.

The goal of training of standard feed-forward neural network is to minimize the network's loss. To do so, we compute the gradient of the loss function with respect to the network weight and use the gradient to update the weights to minimize the loss. On the other hand, in the FGSM attack, we compute the gradient with respect to the input data and use this gradient to adjust input data to maximize the loss.

In other words, the FGSM attack utilizes the gradients to transform the original image into some other image of a different class. We can take this idea and extend it to the latent space and use gradients to select both the best direction and magnitude to create the step in latent space, instead of using the steps from linear interpolation as in the previous section.

In a standard adversarial attack such as FGSM, the target of the attack is a classifier in the form of function $f : X \to \mathbb{R}^k$, that takes a batch of images x and outputs class. In order to use gradients to modify latent space, we need to use a classifier in the form of function $f_z(z)$ that takes a batch of latent representations z and outputs class. The natural way to obtain such a classifier is to train a separate network that uses latent representation vectors as a train set instead of images as a train set with labels obtained from an auxiliary classifier. However, this approach is limiting because the classifier neural network needs to be trained in conjunction with some generator, and as a result, we cannot attack arbitrary classifiers.

We propose universal method of creating function $f_z(z)$ from arbitrary classifier $f(x)$. It consists of concatenating the last layer of a decoder and the first layer of target classifier $f(x)$. The resulting function, the chained classifier, takes latent representation z as input, passes it to decoder D to obtain image x, passes them to classifier $f(x)$ which outputs class probability. The chained classifier is differentiable if both decoder and classifier are differentiable, allowing us to calculate gradients with respect to latent input representation z.

Let D be generator or decoder used for mapping vectors of z in latent space to corresponding image x in image space. Let $f_{D,f}(z)$ denote chained classifier created by concatenating last layer of decoder D and the first layer of classifier $f(x)$. Classifier $f_{D,f}(z)$ takes z as input and outputs softmax for each class. We denote I as the crossentropy loss function of classifier $f_{D,f}(z)$. Finally, α is step size of the shift in z. Single step of the gradient traversal method can be decribed

by following equation:

$$x^* = Decode(z + \alpha \frac{\nabla_z I(z, y_{true})}{\|\nabla_z I(z, y_{true})\|} \tag{4.1}$$

The procedure is similar to the standard untargeted FGSM, but we use $f_{D,f}(z)$ as a classifier under the attack. Additionally, we normalize the gradient using l_2 norm instead of signum. We calculate the gradient of the loss function with respect to z and then use it to select the direction and magnitude of change in z such that it maximizes the loss of the classifier. In the end, the perturbed z is translated from latent space back to images using decoder $.D$

Compared to FGSM, where the step size ϵ has direct interpretation in terms of l_∞ distance, transforming the z in latent space by step size α does not tell us how much the actual image x will be changed in terms of any l_p distance after decoding the z back to image space. We do not know in advance what step size α will be sufficient to change the class or if the result are valid images.

We can ensure the validity of the resulting images by selecting α sufficiently small compared to the distribution of z. The gradient is normalized using l_2 norm, so the sum of its elements is one. Therefore, any element of a latent vector created by the gradient step will differ at most by α from the corresponding element of the original vector. It is reasonable to select α such that variation created by the step matches variation of the z distribution measured its standard deviation. The prior distribution of z is a multivariate standard normal distribution with a standard deviation of 1 in each dimension. Therefore, the maximum allowed value of α should be small multiples of the standard deviation (such as 3) to prevent the worst-case scenario, where only one element of the latent vector is changed by α. However, this scenario is improbable as the normalized gradient values are distributed over dimensions of the latent space. Therefore, we suggest selecting maximal α to be proportional to the dimensionality of latent space.

Figure 4.7 shows the percentage of MNIST images that changed class with increasing interpolation step α. We can see that for value $\alpha = 5$, most of the images changed class and further increasing α has a diminishing effect. Additionally, even for large α, we are not able to fully transform all images. We attribute this behaviour to a situation where the classifier is often strongly confident with its predictions, and the softmax of the original class is close to 1 in some cases. The gradient in such an example is zero or near zero within floating-point error and does not help us guide the step in latent space. Normalizing the gradient does not help either and only emphasizes any floating-point error.

Figure 4.8 shows images created by gradually increasing the gradient step size α until the classifier confidently classifies the final image created by the transfor-

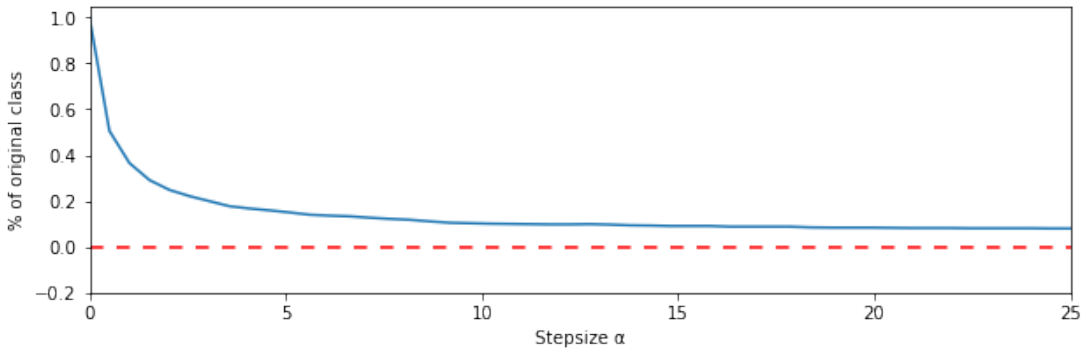

Figure 4.7: Success rate of transformation for untargeted gradient method. The figure shows percentage of MNIST images that changed class with increasing gradient step α. Values are calculated from total of 1000 images with 100 images for each class. We used latent representations from the ALI model.

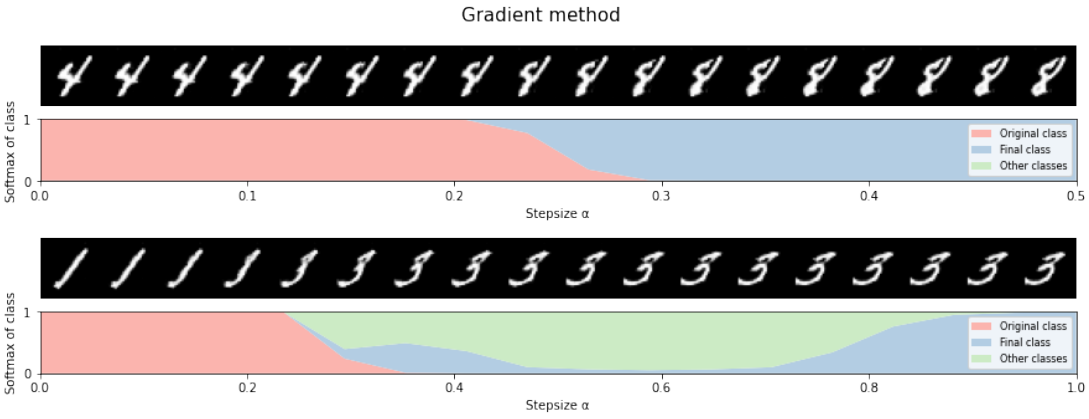

Figure 4.8: The effect of gradually increasing parameter α in gradient method on of both image and classifier's softmax output for the image. The bottom half shows how softmax changes for the original image class (in red) and target image class (in blue). The Sum of softmax activations for all other classes is shown as green. We used latent representations from the ALI model.

mation as a class other than the class of the original image. In both examples, the digits change class eventually, but we can see the same issues described in the linear interpolation approach. In the first example, the transition between the two images is not gradual, and in the second example, the transformation goes through a different class than the original and final images.

4.4.1 Gradient method with bisection

In Figure 4.8, we gradually increment the step size α and stop when we are sure the α is enough to change class. Similarly to the linear interpolation, this transition is not gradual and is steep as we go through the decision boundary. We are interested in selecting the α such that the softmax values of the original class are evenly spaced.

We can use bisection to find step size α that corresponds to the arbitrary softmax value of the original class, equivalently how we used the bisection with linear interpolation. Similarly to the linear interpolation method, given latent representation z of the original sample and a classifier, we can design function $f(\alpha)$ that takes α as input and outputs a scalar that corresponds to the softmax of the original sample class. For $\alpha = 0$, the function has value $f(0) = 1$ as the perturbed z corresponds to the original sample. With increasing α, the function values decrease as we increase the perturbation magnitude, resulting in a decrease of original class softmax.

Unlike linear interpolation, where modifying z through full interpolation is guaranteed to cause class change, there is no such guarantee for gradient steps. The bisection algorithm may not be able to find α from the interval $[a, b]$, where a and b are endpoints for the bisection, such that it changes the original class softmax. As the steps in latent space have no direct interpretation in image space, we cannot say in advance if the upper endpoint large enough to such perturbation of z that causes a change of class from original to different one.

Results from Figure 4.7 gives us some insights into how large the α should be to transform most images, but we are not able to fully transform all of the images even for large step size. Unchangeable images make use of bisection problematic, as it assumes that the solution exists in the $[a, b]$ interval. If it does not exist there, the bisection never converges, and the algorithm leaves the loop after reaching maximum iteration, which makes it very costly to use in such cases.

In Figure 4.9, we used bisection to selected α such that the softmax of the original class is split evenly on the $[0, 1]$ interval. We can see that bisection with gradient steps has similar properties to using bisection combined with linear interpolation. Compared to the method we used in Figure 4.8, using the bisection

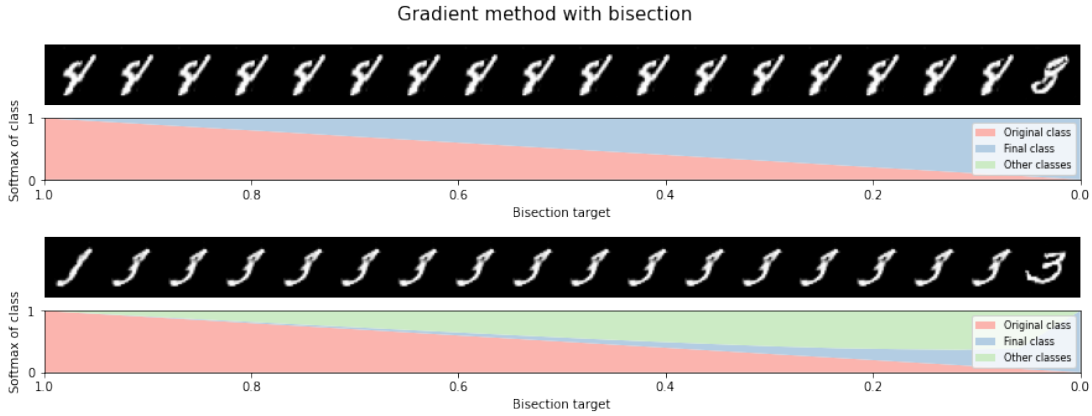

Figure 4.9: The effect of gradient method with bisection on both image and classifier's softmax output for the image. We select the α step using bisection such that the new image matches the target, which is the softmax of the original class. We initially target high values of softmax of the original class and then gradually decrease the target. The bottom half shows how softmax changes for the original image class (red) and the target image class (blue). The sum of softmax activations for all other classes is shown as green. We used latent representations from the ALI model.

allows us to inspect samples in the steep region. Similarly to the linear interpolation method, most of the change happens between the first two and last two images. Although we can observe only minor visual changes for images in between, the softmax of the original class changes substantially.

4.4.2 Gradient method with mixed labels

The bisection searches for the optimal α such that the gradient multiplied by α creates a perturbed z that best matches the softmax of the original class. However, if the optimal α is large, we move for a substantial distance in the latent space based on the gradient calculated with respect to the original latent vector. Such an update can result in suboptimal transformation as we cannot fully utilize all information available because we skip large parts of the latent space with a single large update. We expect that a better and smoother transition between the classes can be achieved by performing multiple smaller gradient steps instead of one large step.

The standard application of bisection is to find a single optimal scalar, which in our cases included finding optimal α or optimal interpolation step. Using bisection to find an optimal sequence of multiple steps is much more complicated, as we need to optimize for multiple parameters, such as the total number of the partial steps and step size for each of the partial steps. Additionally, bisection cannot find the optimal sequence in complex cases with dependency between

the parameters. For example, bisection cannot find optimum if the sequence of optimal partial step sizes decreases as we move from the original sample.

$$x^* = Decode(z - \alpha \frac{\nabla_z I(z, y_{mixed})}{\|\nabla_z I(z, y_{mixed})\|} \tag{4.2}$$

We propose a different approach that allows us to use multi-step sequences to transform from one sample to another with different classes. Similarly to how we can extend FGSM to its targeted or multi-step variants, we can easily convert the single step and untargeted method we described in Equation 4.1 to its targeted variant. Then, using one-hot encoding, we create a mixed label that can represent any given softmax combination. This flexibility allows us to use the targeted variant of the latent gradient method to target the desired softmax combination, as described in Equation 4.2. We can improve the targeted method by using its multi-step variant with a fixed number of small iterations to reach mixed label target. The validity of the resulting image is ensured by selecting a sequence of steps such that the sum of all the partial step sizes is sufficiently small compared to the distribution of z.

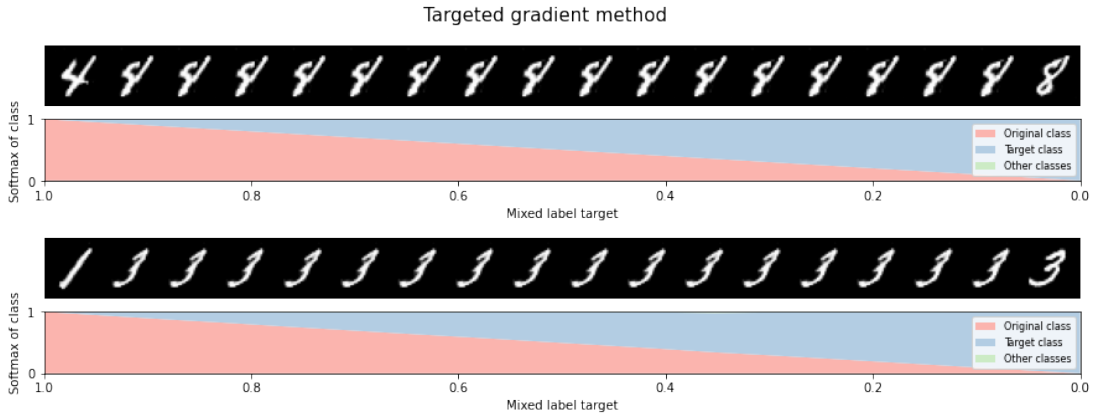

Figure 4.10: The effect of targeted gradient method on both image and classifier's softmax output for the image. We use the targeted gradient method with mixed labels to create new images. We initially target mixed labels with a large share of the original class and then gradually decrease the share in favour of the target class. The bottom half shows how softmax changes for the original image class (red) and the target image class (blue). The sum of softmax activations for all other classes is shown as green. We used latent representations from the ALI model.

We used 200 steps with exponential decay for step size α such that the sum of all partial step sizes is lesser than two. As a target, we used a mixed label with an increasing ratio of the target class. In the first example, the original class is 4, and the target class is 8. In the second example, the original class is 1, and the target class is 3. Targeting the mixed label allows us to explore

samples on decision boundary and yields similar results with similar properties to the bisection with both gradient steps and linear interpolation.

Using mixed label targets allowed us to focus on sampling samples that are exclusively between two classes. Targeting only two classes is not possible for bisection, as shown in the second example of Figure 4.9. Bisection is used to optimize a single scalar, softmax of the original class in our case, and its iteration stops when the optimization is complete. It does not guarantee us that the remaining softmax is of a single class. On the other hand, targeting mixed label between two classes allows us to mixed samples between only those two classes more efficiently.

4.4.3 Algorithm for extracting adversarial examples

As shown in Figure 4.10, the samples, except first and last, are all visually similar. We exploit this property to sample a set of examples that may be of our interest as a potential adversarial example. We use the mixed label between an original and a target classes and mix them such that the arg-max of the mixed label is the target label. In this way, we create samples classified as target class and visually similar to the original class at the same time.

Empirically, we found out that the targeted method does not always converge. In fact, it rarely transforms the class in a situation where we take some sample then target an arbitrary class. We attribute this behaviour to gradients being close to zero in the direction of the target class for most of the target classes. We presume this is a problem in the case of well-trained classifiers, such as our MNIST classifier. We can bypass this by choosing a class that is most likely to be changed from given z. Such a class can be found by applying an untargeted variant of the attack first with sufficiently large enough α. Performing this procedure as preparation step yields a class that is the most suitable class for the mixed label in the targeted attack.

We denote $S_i(x)$ as softmax activation for class C_i of a classifier on which we target the adversarial examples. Full procedure can be described in following steps:

Algorithm 4: Latent gradient attack with probe

Input : $G(z)$ - generator, $f(x)$ classifier under attack, C_1 - original class, m - softmax target for C_1

Output: x^* - candidate for adversarial example

Steps :

1. Using $G(z)$, generate sample x with latent representation z and class C_1.

2. Create chained classifier $f_{G,f}(z)$ from $G(z)$ and $f(x)$.

3. Use untargeted gradient method (4.1) with classifer $f_{G,f}(z)$ to transform z to z' with class C_2, such that $C_1 \neq C_2$.

4. Create mixed label between C_1 and C_2 using one hot encoding. Mix the labels using ratio m between the two classes, such as $label(C_1) = m$ and $label(C_2) = 1 - m$, where $label(C)$ denotes share of class C in the mixed label.

5. Use targeted gradient method (4.2) with classifier $f_{G,f}(z)$ to target the mixed label and create candidate sample x^*.

6. Attack is successful if for x^* holds $|S_2(x^*) - (1 - m)| < \epsilon$ and $|S_1(x^*) - m)| < \epsilon$, where tolerance parameter ϵ is small value near zero.

In Step 1, we suggest using the target classifier for the labelling instead of an additional auxiliary classifier. In Step 3, we select class C_2 into which z has the largest propensity to transform. Finding such class C_2 allows us to solve the convergence issues that would occur in the case of manually selecting target class C_2. In Step 4, we restrict the mix ratio such that $m < 0.5$ in order to generate images that are not classified as original class C_1. For example we can use $label(C_1) = 0.3$.

We suggest using Algorithm 4 with iterative variants of both untargeted and targeted gradient methods in order to increase the success rate of the attack. Again, the candidate samples are not restricted by any norm. From the candidate set, we use human evaluation to manually select images classified as images of the original class by humans.

5. Experiments

This chapter is dedicated to a discussion about the images created using our methods. We used both linear interpolation method (Algorithm 3) and gradient method (Algorithm 4) to sample a set of candidate samples combined with the various generative models for each of the datasets (MNIST, SVHN, CIFAR10). In both methods, we targeted $m = 0.3$, which is the softmax of the original class and 0.7 of some other class and filtered out results that do not match this target. Additionally, we use Algorithm 4 with multi-step variant of the untargeted probe (100 steps, exponential decay with $\gamma = 0.01$ and initial $\alpha = 0.1$) and multi-step variant of the targeted attack (200 steps, exponential decay with $\gamma = 0.05$ and initial $\alpha = 0.1$).

5.1 Candidate images

First, we discuss statistics and properties of raw outputs of the algorithms before any manual selection. We can view the matrices for both linear interpolation method (Figure 5.1) and gradient method (Figure 5.2) as measures of the propensity of change to a specific class, given the original class. Another interpretation is that the matrices capture class similarity in high-level features and the figures show how classes are close to each other on the manifold created by the generative model. This proximity of classes in latent space then indicates the propensity of change to a specific class. Both methods from both figures capture similar effect, but from a different perspective.

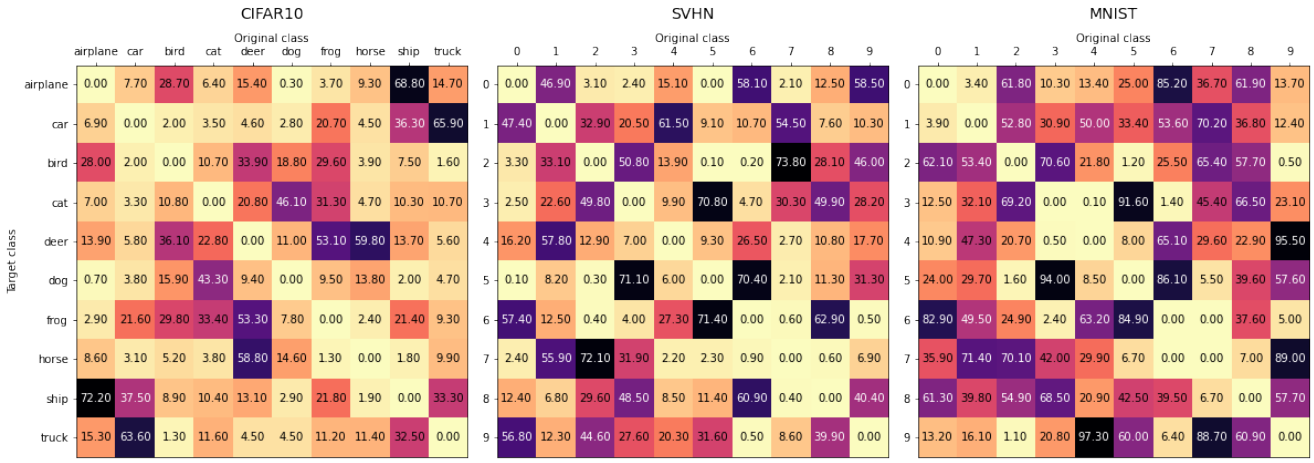

Figure 5.1: Matrix shows percentage of successfully transformed samples from original class to the target class using linear interpolation with bisection method. We targeted value of 0.3 orignal class softmax. As success, we consider final samples with softmax 0.3 of original and 0.7 of target class, within 0.05 tolerance

Figure 5.1 tells us how likely it is to change one class to some target class using the interpolation with bisection method. Successfully changed samples are those with softmax values close to 0.3 for the original class and 0.7 for the target class. For example, we take the success rate of interpolation from digit 9 to 4 in the MNIST dataset, which can be found in the last columns and fourth row of the right matrix. We used bisection to find samples that match target softmax 0.3 for digit 9. The matrix shows that 95.5% of such transformations were successful. Those successfully transformed samples have the softmax values for digits 9 and 4, within 0.05 tolerance, 0.3 and 0.7, respectively.

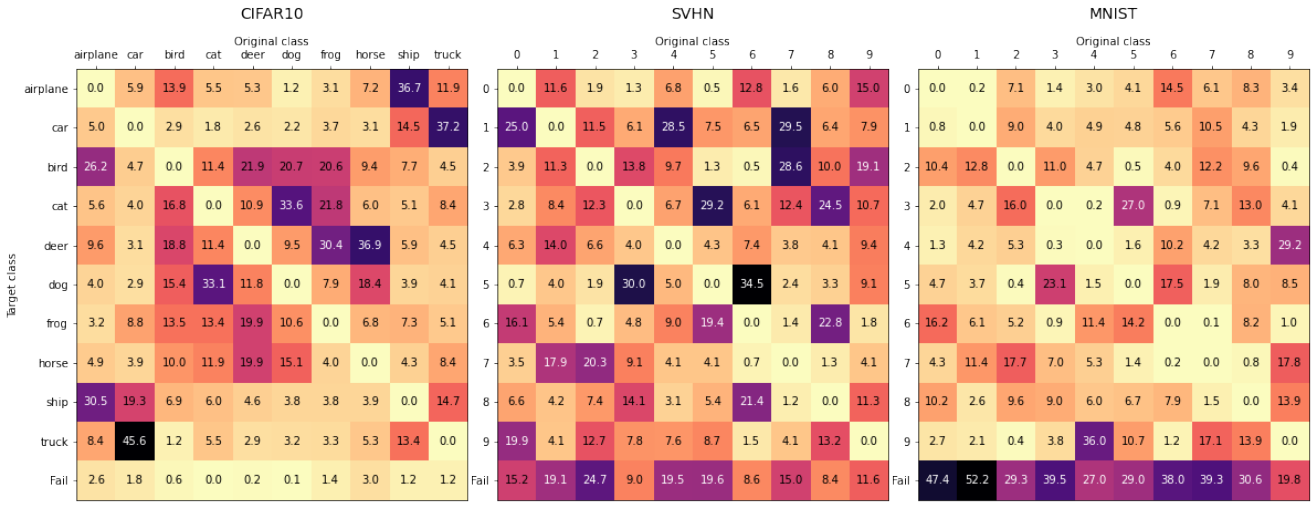

Figure 5.2: Matrix shows distribution of all outcomes of transforming original class using mixed label targeted gradient method with probe. We targeted mixed label with 0.3 of orignal class and 0.7 of class suggested by a untargeted probe. As success, we consider final samples with softmax 0.3 of original and 0.7 of suggested class, within 0.05 tolerance. Columns sums to 100.

On the other hand, Figure 5.2 shows the distribution of transformation outcomes for each class using the gradient method. The gradient method targets mixed labels with softmax values of 0.3 and 0.7 for the original class and class suggested by an untargeted probe. As successful, we consider samples where softmax values are within small tolerance to the desired mixed label. For example, we take values for MNIST digit 0, which can be found in the first column of the right matrix. 10.4% of successful transformations were into digit 2 and 0.8% into digit 1. Additionally, 47.4% of transformations from digit 0 were unsuccessful. Unsuccessful transformations result from not founding any class during the untargeted probe or failing to transform the sample to the mixed label during targeted transformation.

One of the differences between the two matrices is that the matrix from Figure 5.2 includes the failure category. Additionally, the matrix is normalised such that

sum of the columns is 100, including the failure category. Relative differences, except the failure category, between the class values are similar for both methods, and we can see that the matrices are highly correlated for each of the datasets. Both matrices are roughly symmetric about diagonal, and we expect them to converge to diagonal matrices with an increasing number of generated samples. The diagonal is zero in both cases because targeting the initial class does not change a class, making it an invalid target.

In MNIST, we can see that several digit pairs show high propensity, while some combinations are unlikely to be successful. For example, we can see that digits 9 and 4 are likely to change into each other compared to digits with a low propensity, such as digits 9 and 2. Another notable pair of similar MNIST digits are 5 and 3. On the other hand, digits 9 and 4 are not very interchangeable in the SVHN dataset. Some digit pairs, such as (5, 3) and (5, 6) share their similarity across both MNIST and SVHN dataset. Most of the similarities correspond to our intuition on how the digits should transform.

We can use intuition to explain likely transitions in CIFAR, where the mistakable pairs correspond to semantically and visually similar images. For example, transportation devices pairs (cars, trucks) and (boat, airplane) are likely to change to each other. Furthermore, (bird, airplane) are similar, as both represent flying objects. Given the matrices, we can observe a cluster of animal classes, such as dog, cat, birds, horse and deer being similar and especially horse is easily interchangeable for deer class. However, results from CIFAR are not easily verifiable by human judgement as the quality of images generated by the generators is low.

There are notable differences between the failure rate of MNIST digits. We can see that digit 1 is most likely to fail and digit 9 least likely, with failure rates of 52.2% and 19.8%, respectively. We can interpret it that digit 1 has distinctive features, as it is an essentially downward-sloping straight line. Such features enable the classifier to identify it reliably. Subsequently, the gradient will be minimal for most digit 1 samples, resulting in a higher failure rate than other classes. The opposite is true for digit 9. Another interpretation is that samples with digit 9 are at the centre and samples with digit 1 are on the fringes of the latent space manifold. Similar disparities between failure rates of digits can be observed in the SVHN dataset. Conversely, CIFAR failure rates are altogether negligible.

The overall failure rate in Figure 5.2 rate for SVHN is smaller than the failure rate for MNIST, but it is still significantly higher than CIFAR. At first glance, this result does not correspond to Figure 5.1, where we generally can observe much higher success percentages in the SVHN and MNIST datasets. Our suggested explanation is that in linear interpolation, CIFAR is likely to go through various

classes because the generator is not powerful enough to split the classes on the latent manifold due to the complexity of the CIFAR dataset. As a result, in CIFAR, the classes are near each other and blend together on the manifold, making interpolation difficult. On the other hand, this helps CIFAR dataset in the case of the gradient method. Because the classes are not very well separated, it is easier to find direction using gradients from one class, leading to a region with other classes. Compared to digit datasets, the blending of classes on the manifold can cause much evenly distributed class change propensity in CIFAR. Conversely, in MNIST and SVHN, each class is likely to change only to few other classes.

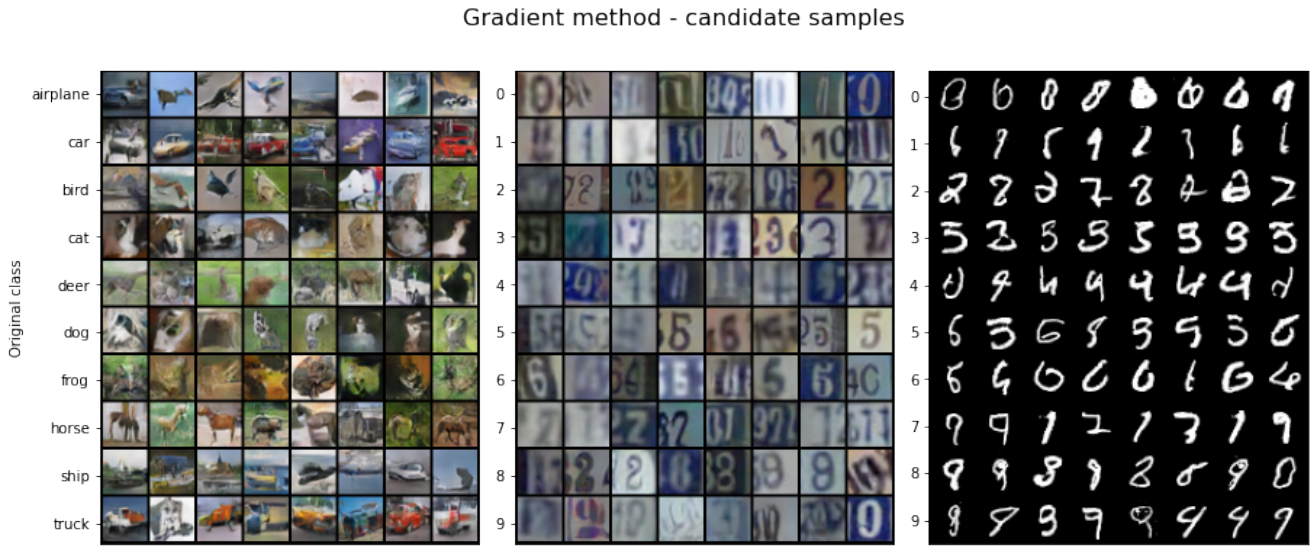

Figure 5.3: Candidate samples before manual selection from gradient method and DCGAN generator. All samples have softmax of 0.3 of orignal class 0.7 of some other class.

All candidate images presented in Figure 5.3 have 0.3 softmax of the original class and 0.7 of some other class, and they were sampled randomly, without any manual tuning, using the DCGAN as a source of latent space. We can see that many candidate samples are of poor quality or the images have ambiguous class. Poor quality images are the main problem in CIFAR, but we attribute this to the poor quality of the generator because the class of the samples is often difficult to guess even in unperturbed images. On the other hand, many samples resemble the original class, notably in the horse and car classes. Although a significant part of the images in SVHN is blurry, remaining sharp images often have distinguishable features of some classes, and occasionally, features of the original class. Noteworthy examples are digits with original class 2 and 3. We can at first glance see several unambiguous digits of the original class and which makes valid adversarial examples. In MNIST, image quality is not a problem,

but we can see that a considerable part of the images is ambiguous. Additionally, the ambiguous images mostly have distinct features of one of the two classes. For example, in the row with images originating from digit 9, we can see that all images have some features of digit 9, mixed with some other digit, such as digit 4 or 7.

5.2 Adversarial examples

We can see in Figure 5.3 that many samples are misclassified, especially MNIST and SVHN digits. This is a crucial observation, as it allows us to identify and select samples we would classify as the original class. We look for images among the candidate set that we would classify as the original class. Such images are adversarial, as they are classified with a class with a maximum value of softmax, which is some other class than the original. Additionally, we search for sharp and least ambiguous images with the most distinctive features of the original class to highlight the classifier's misclassification. Given target classifier, we first show that such strikingly misclassified images exist, and secondly, we can generate arbitrary many of them. We manually select those images from candidate images retrieved using combinations of the three datasets, the available generative models, and our methods.

5.2.1 MNIST

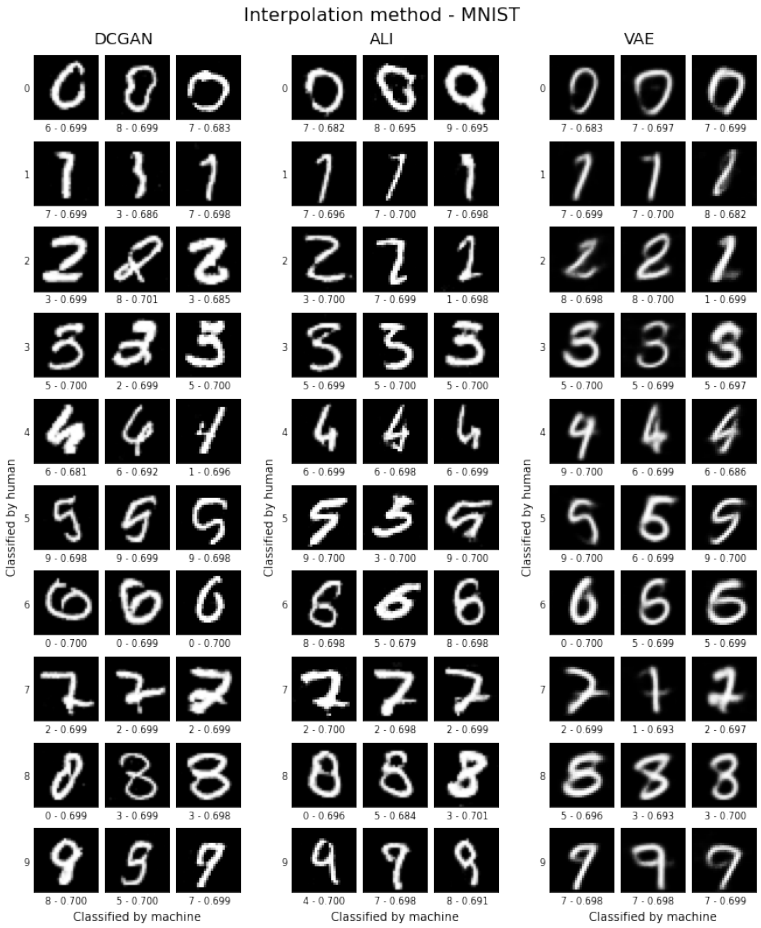

Figure 5.4: Adversarial examples for MNIST dataset. The images are created using the interpolation with bisection method and latent space from three different generators. Bisection target is 0.3 original class softmax. After filtering unsuccessful transformations, we manually select three samples that we would classify as original class but are classified as a different class by a classifier that is under attack.

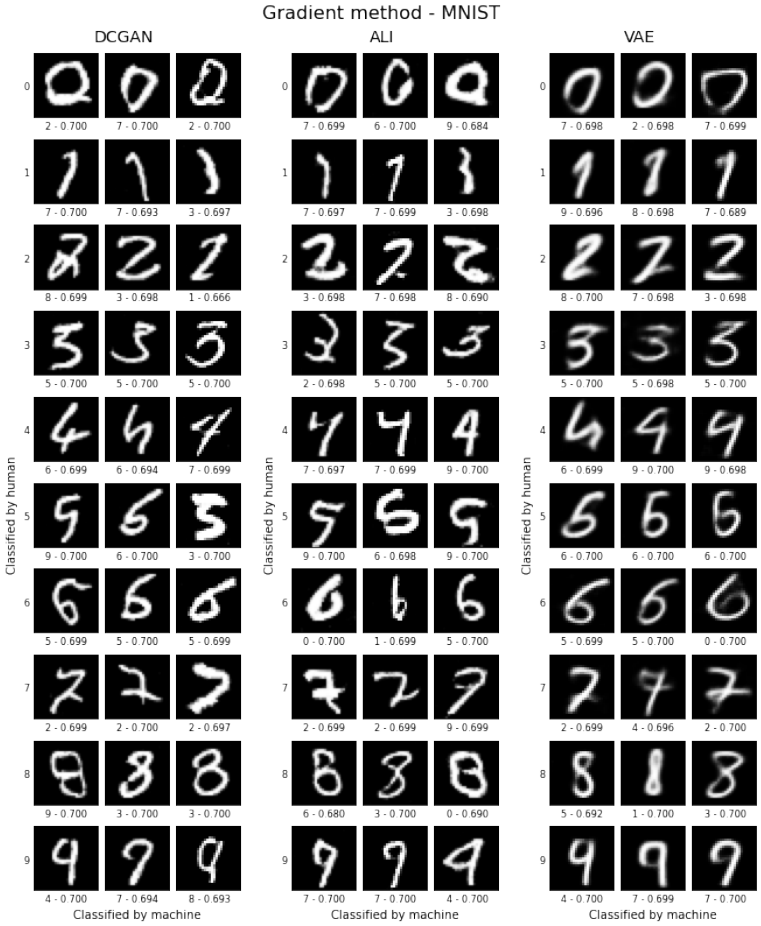

Figure 5.5: Example of MNIST images created using the gradient method and latent space from three different generators. We targeted mixed labels with softmax 0.3 of original class and 0.7 of target class. The target class was selected by probing with the untargeted gradient method. After filtering unsuccessful transformations, we manually selected for each class three samples that we would classify as original class but are classified as a target class by target classifier.

5.2.2 SVHN

Figure 5.6: Adversarial examples for SVHN dataset. The images are created using the interpolation with bisection method and latent space from three different generators. For bisection, we targeted 0.3 of original class softmax. After filtering unsuccessful transformations, we manually select three samples that we would classify as original class but are classified as a different class by a classifier that is under attack.

Figure 5.7: Example of SVHN images created using the gradient method and latent space from three different generators. We targeted mixed labels with softmax 0.3 of original class and 0.7 of target class. The target class was selected by probing with the untargeted gradient method. After filtering unsuccessful transformations, we manually selected for each class three samples that we would classify as original class but are classified as a target class by target classifier.

5.2.3 CIFAR10

Figure 5.8: Adversarial examples for CIFAR10 dataset. The images are created using the interpolation with bisection method and latent space from three different generators. For bisection, we targeted 0.3 of original class softmax. After filtering unsuccessful transformations, we manually select three samples that we would classify as original class but are classified as a different class by a classifier that is under attack.

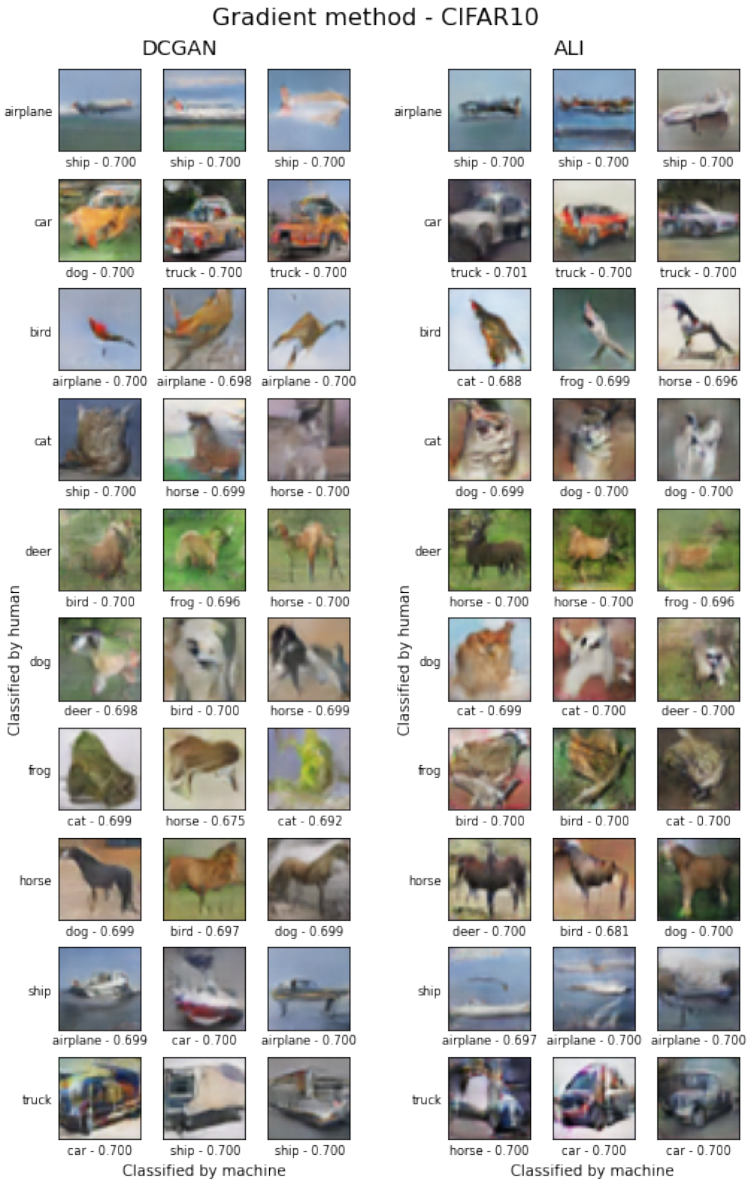

Figure 5.9: Example of CIFAR10 images created using the gradient method and latent space from three different generators. We targeted mixed labels with softmax 0.3 of original class and 0.7 of target class. The target class was selected by probing with the untargeted gradient method. After filtering unsuccessful transformations, we manually selected for each class three samples that we would classify as original class but are classified as a different class by the classifier.

5.2.4 Adversarial examples for robust classifier

So far, we used a standard classifier based on VGG architecture as a target classifier. In previous research, Szegedy et al. [2014] showed the vulnerability of basic undefended classifiers in the form of pixel perturbations. The previous section showed that this vulnerability extends to perturbations in latent space, which allows us to create unrestricted examples for such classifiers. This section demonstrates that our methods also work against robust classifiers with some

form of defence.

We select robust MNIST classifier from Madry et al. [2019] as they made the model accessible as part of the public challenge. They adversarially trained the classifier using adversarial examples created by a universal first-order adversary. According to the authors, such adversarial training can be used as universal defences against adversarial attack, as it solves the min-max robust optimization problem. Solid empirical results of their defence method support this claim. All of this makes their model a good defence benchmark, and we use it as a target classifier for our attack. Other attack parameters are the same as in the previous section. We use DCGAN to create latent space and targeted gradient method with a 0.3 and 0.7 label mix for original and suggested classes and filtered out unsuccessful transformations.

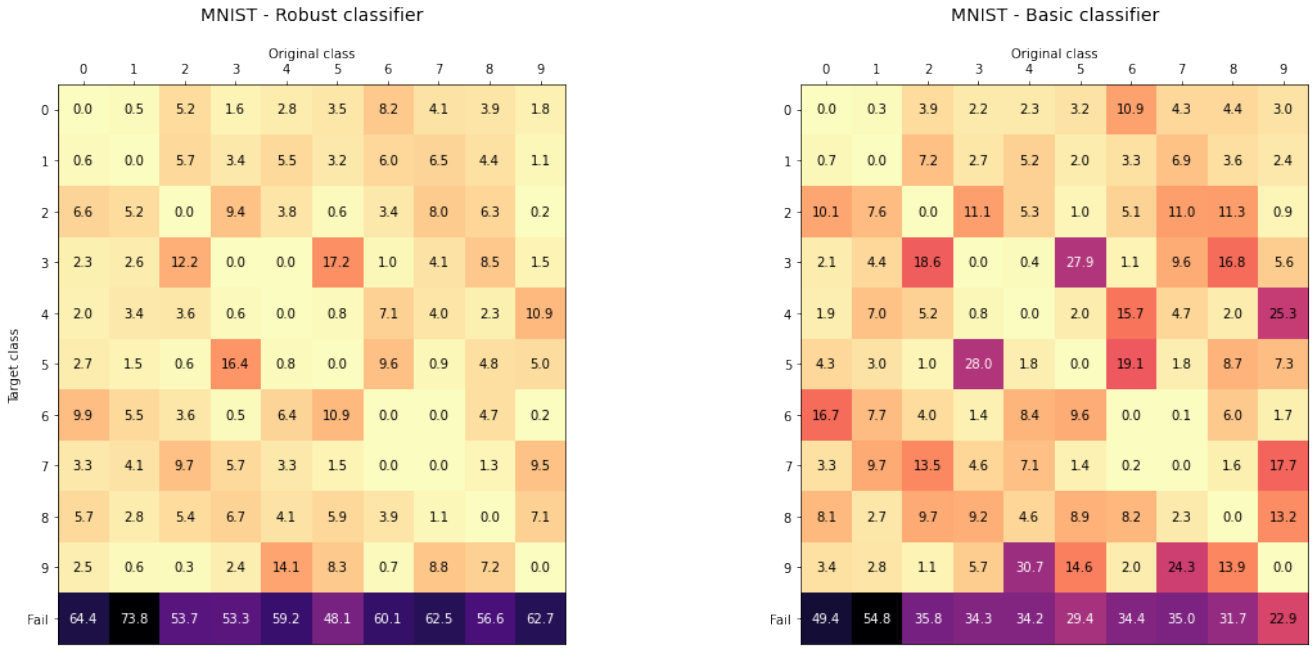

Figure 5.10: Distribution of transformation outcomes for robust and natural classifiers of Madry et al. [2019].

Figure 5.10 shows the distribution of transformation outcome given some original class, similarly to Figure 5.2, but now we use robust and natural classifiers from Madry et al. [2019] as targets of the attack. The natural classifier is trained without the adversarial training defence. Otherwise, both classifiers share the same architecture. We can see that the robust classifier shows a much higher failure rate than the natural classifier. However, the failure rate is not a helpful indicator of defence success against the attack, as it only tells us there are fewer candidate images after running the algorithm. It does not tell us anything about the quality of those candidate images or if it is possible to select adversarial examples manually.

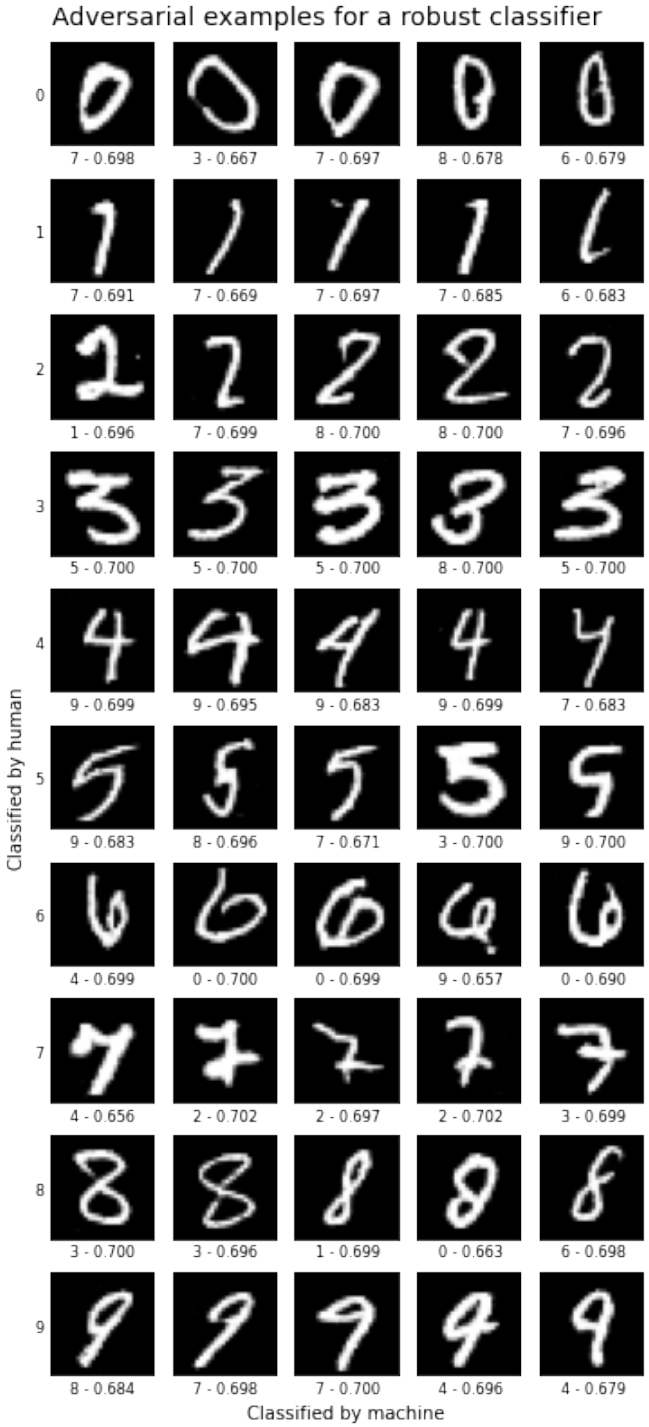

Figure 5.11: Adversarial examples for Robust MNIST classifier of Madry et al. [2019]. We used multi-step variant of Algorithm 4 with $m = 0.3$ and DCGAN generator to create the latent space. After filtering unsuccessful transformations, we manually selected for each class samples that we would classify as original class but are classified as a different class by the classifier.

From candidate images, we can consistently manually select images that look like the original class. By definition of candidate images, they are classified as some other class. As we can see in Figure 5.11, the quality of manually selected

images is no worse than MNIST images in Figure 5.5, designed to be adversarial for an undefended classifier. Our results show that the defence technique described in Madry et al. [2019] works well only against perturbations in image space, and against our attack, it provides only limited defence in the form of a smaller candidate set. We speculate that the gradient calculated from the robust model is more often near zero. It prevents transformation for many examples and is the cause for a smaller candidate set of robust classifiers. If we obtain a gradient large enough to transform the image, we can still exploit the steepness of the decision boundary to find adversarial examples using our method.

Conclusion

In this thesis, we explored the prospect of designing adversarial examples using generative models. We extend the work of Song et al. [2018] by using unconditional generative models instead of conditional ones. Instead of depending on a conditional generative model to generate an image with a target class, we restrict the unconditional generator to sample the target class using either linear interpolation or gradient descent methods. Moreover, those methods allow us to restrict the generated images to samples at the decision boundary of an arbitrary classifier. We found out that images on the decision boundary look visually very similar but are classified differently by a classifier due to the steepness of the decision boundary. This property enables the construction of adversarial images by first sampling candidate images from the decision boundary and then manually selecting misclassified images.

We design two algorithms to create a set of potential unrestricted adversarial examples. The first proposed method, Algorithm 3, utilizes linear interpolation with bisection to create images with some specific target softmax of the original class. It is more suitable for the targeted attack as we can interpolate between two samples of any class combination. The second proposed method, Algorithm 4, is inspired by the FGSM attack. Instead of perturbing pixels in image space, we perform the steps of gradient descent in latent space. We can use targeted variant and mixed target label between the two classes. However, targeting mixed label between arbitrary two classes leads to high failure rates. We solved this by first probing for classes with a high propensity to change. Probing for a suitable class effectively converts the method to an untargeted attack, as there are no guarantees that the probe result is a selected class.

Our contribution is twofold:

1. Generating unrestricted adversarial images. We showed on MNIST, SVHN and CIFAR10 datasets that generative models can generate unrestricted adversarial examples, and we introduced two algorithms that do so. Such examples can be used it to augment natural datasets with ambiguous images in adversarial training.

2. Generating adversarial images for robust networks. We demonstrated that adversarial images created using our approach are possible against the state of the art defences (Madry et al. [2019]). Our findings suggest weakness in state of the art defences against unrestricted adversarial examples.

A distinct feature of our approach is the need for human evaluation. In or-

der to utilize our approach for creating adversarial examples, we need to ensure the resulting image looks like the original class to humans but is classified differently by machines. In standard adversarial attacks, this is usually done by setting some maximally allowed perturbation in terms of l_p distance to the original image because doing sufficiently small perturbation results in class change undetected by humans. In this thesis, we take a different approach and use our judgement to decide the class membership of candidate samples. Arguably, such selection processes of adversarial samples can be biased, and resulting images can be classified differently by other humans. Therefore, we disclose a large number of our selection in reasonably high resolution and leave it to the human to form their own opinion on the quality of generated samples. Another approach to evaluate unrestricted adversarial examples is used in Song et al. [2018]. They used the Amazon Mechanical Turk service to classify the images by human. Such an approach would be a great way how to evaluate our methods as well. However, creating such a large scale human experiment is beyond the scope of this thesis, and we leave it for future research.

A natural extension to our work is to use different generative models, such as flow-based models, and compare them to the variational autoencoders and GAN-based models. We expect flow-based models to be more flexible as they directly learn the probability density function of real data and learn richer and more detailed representation.

Another interesting topic for further research is to extend our methods to domains different from images. However, this topic is related to the research of generative models in such domains, as we need to map the data from any given domain to create latent space to use our methods.

Because our approach is unrestricted in terms of l_p norm in domain data points, we need to be able for humans to inspect the domain data and select data points that are misclassified. This limitation restricts the use of our method in many domains. Extending our method to various domains can be enabled by further researching how to interpreting movements in latent space. It would be beneficial to develop a proper methodology to measure perceptual or visual similarity between two samples, given their representation in latent space. Such research would greatly help extend the method to domains that are also interested in developing robustness against adversarial samples but where humans are unable to determine the correct classification of given data points quickly. Examples of such domains are credit risk modelling, predicting fraudulent transactions, and other various financial and marketing datasets.

For further research, we suggest extending our methodology to high-resolution image datasets, such as Imagenet. Based on our results in SVHN and CIFAR10,

we think there is strong evidence that our methodology will work adequately for the high-resolution datasets. The main issue with applying our methodology to datasets like Imagenet is the lack of high-quality generative models in those datasets. However, the research in generative models for high-resolution dataset has recently made significant advances, and we believe that a good enough generator will be developed soon.

Additionally, we believe that looking at the problem from a robust optimization perspective can offer a valuable perspective. With adversarial training included, our approach can be seen as an extension of a min-max problem from the image domain to latent space. Robust optimization methodology can extend our work substantially, especially in the area of adversarial training.

Bibliography

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan, 2017.

Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks, 2017.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2015.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp, 2017.

Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning, 2017.

Vincent Dumoulin, Ishmael Belghazi, Ben Poole, Olivier Mastropietro, Alex Lamb, Martin Arjovsky, and Aaron Courville. Adversarially learned inference, 2017.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

I. Higgins, Loïc Matthey, A. Pal, C. Burgess, Xavier Glorot, M. Botvinick, S. Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. In *ICLR*, 2017.

Hossein Hosseini, Yize Chen, Sreeram Kannan, Baosen Zhang, and Radha Poovendran. Blocking transferability of adversarial examples in black-box learning systems, 2017.

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2018.

Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions, 2018.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2014.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale, 2017a.

Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world, 2017b.

Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric, 2016.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks, 2019.

Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples, 2017.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011. URL `http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf`.

Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings, 2015.

Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks, 2016.

Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 11 2015.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models, 2014.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016.

Pouya Samangouei, Maya Kabkab, and Rama Chellappa. Defense-gan: Protecting classifiers against adversarial attacks using generative models, 2018.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

Yang Song, Rui Shu, Nate Kushman, and Stefano Ermon. Constructing unrestricted adversarial examples with generative models, 2018.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2014.

Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks, 2019.

Huan Zhang, Hongge Chen, Zhao Song, Duane Boning, Inderjit S. Dhillon, and Cho-Jui Hsieh. The limitations of adversarial training and the blind-spot attack, 2019.

List of Figures

2.1 Schema of feed-forward ANN 6
2.2 Schema of VAE 7
2.3 Schema of GAN 8
2.4 Schema of ALI 9

4.1 Samples from MNIST, CIFAR10 and SVHN 21
4.2 Generated images for CIFAR10 by class 23
4.3 Generated images for SVHN by class 24
4.4 Generated images for MNIST by class 24
4.5 Modification using linear interpolation 25
4.6 Modification using linear interpolation with bisection 27
4.7 Success rate of transformation for gradient method 32
4.8 Modification using gradient method 32
4.9 Modification using gradient method with bisection 34
4.10 Modification using gradient method with bisection 35

5.1 Interpolation method success rate 38
5.2 Gradient method success rate 39
5.3 Candidate samples 41
5.4 Interplation method samples - MNIST 43
5.5 Gradient method samples - MNIST 44
5.6 Interplation method samples - SVHN 45
5.7 Gradient method samples - SVHN 46
5.8 Interplation method samples - CIFAR10 47
5.9 Gradient method samples - CIFAR10 48
5.10 Gradient method statistics 49
5.11 Adversarial examples 50

List of Tables

4.1 Performance of classifiers 21
4.2 Performance of generative models 23

A.1 MNIST - Adversarially Learned Inference 60
A.2 MNIST - Generative Adversarial Network 60
A.3 MNIST - Variational Autoencoder 61
A.4 MNIST - Classifier 61
A.5 SVHN - Adversarially Learned Inference 62
A.6 SVHN - Generative Adversarial Network 62
A.7 SVHN - Variational Autoencoder 63
A.8 SVHN - Classifier 63
A.9 SVHN - Adversarially Learned Inference 64
A.10 SVHN - Generative Adversarial Network 64
A.11 SVHN - Variational Autoencoder 65
A.12 CIFAR - Classifier 65

A. Appendix

A.1 Model Architectures

A.1.1 MNIST models

Table A.1: MNIST - Adversarially Learned Inference

Layer	Filter	Features	Activation	Other
Generator X				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride, 2 padding	64	ReLU	BN
Convolution	1x1 kernel, 1 stride	1	Tanh	-
Generator Z				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	1x1 kernel, 1 stride	256	-	-
Discriminator - X part				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	Dropout 0.2
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN, Dropout 0.2
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN, Dropout 0.2
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN, Dropout 0.2
Discriminator - Z part				
Convolution	1x1 kernel, 1 stride	256	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	256	Leaky ReLU 0.1	Dropout 0.2
Discriminator - Joint part				
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	1	-	Dropout 0.2

Other parameters: batch size: 64, size z: 128, epochs: 50. Optimizers for Generator and Discriminator: Adam ($lr = 10^{-4}, \beta_1 = 0.5, \beta_2 = 0.999$). Both optimizers decrease learning rate by 0.5 every 10 epoch.

Table A.2: MNIST - Generative Adversarial Network

Layer	Filter	Features	Activation	Other
Generator				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride, 2 padding	1	Tanh	-
Discriminator				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	2x size z	Sigmoid	-

All convolution and transposed convolution layers are without bias. Other parameters batch size : 64, size z: 100, epochs: 100. Optimizers for Generator and Discriminator: Adam ($lr = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999$). Both optimizers decrease learning rate by 0.5 every 20 epoch.

Table A.3: MNIST - Variational Autoencoder

Layer	Filter	Features	Activation	Other
Decoder				
Transposed Conv.	4x4 kernel, 1 stride	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	32	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride, 2 padding	1	Tanh	-
Encoder				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	512	Leaky ReLU 0.2	BN
Convolution	1x1 kernel, 1 stride	128	-	-

Other parameters: batch size: 128, size z: 64, epochs: 80, beta: 3 (3 x KLD part of loss). Optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$).

Table A.4: MNIST - Classifier

Layer	Filter	Features	Activation	Other
Convolution	3x3 kernel, 1 stride, 1 padding	32	ReLU	BN
Convolution	3x3 kernel, 1 stride, 1 padding	32	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	-
Convolution	3x3 kernel, 1 stride, 1 padding	64	ReLU	BN
Convolution	3x3 kernel, 1 stride, 1 padding	64	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	-
Dense Layer	-	512	ReLu	BN, Dropout 0.5
Dense Layer	-	512	ReLu	BN, Dropout 0.5
Dense Layer	-	10	Softmax	-

Other parameters: batch size: 128, epochs: 20. Optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$). Optimizer decrease learning rate by 0.5 every 2 epoch.

A.1.2 SVHN models

A.1.3 CIFAR models

Table A.5: SVHN - Adversarially Learned Inference

Layer	Filter	Features	Activation	Other
Generator X				
Transposed Conv.	4x4 kernel, 1 stride	256	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 2 stride	128	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 1 stride	64	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 2 stride	32	Leaky ReLU 0.1	BN
Transposed Conv.	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	3	Sigmoid	-
Generator Z				
Convolution	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 2 stride	64	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 1 stride	128	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 2 stride	256	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 1 stride	512	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	128	-	-
Discriminator - X part				
Convolution	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	Dropout 0.2
Convolution	4x4 kernel, 2 stride	64	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 1 stride	128	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 2 stride	256	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 1 stride	512	Leaky ReLU 0.1	BN, Dropout 0.2
Discriminator - Z part				
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Discriminator - Joint part				
Convolution	1x1 kernel, 1 stride	1024	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	1024	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	1	-	Dropout 0.2

Input images are scaled to (0,1). Other parameters: Batch size: 128, Size z: 64, Epochs: 200, Optimizer for Generator: Adam ($lr = 10^{-4}, \beta_1 = 0.5, \beta_2 = 10^{-3}$) Optimizer for Discriminator: Adam ($lr = 10^{-5}, \beta_1 = 0.5, \beta_2 = 10^{-3}$). Layers are initialized by values drawn from normal distribution ($mean = 0.0, std = 0.02$).

Table A.6: SVHN - Generative Adversarial Network

Layer	Filter	Features	Activation	Other
Generator				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride	3	Tanh	-
Discriminator				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	512	Leaky ReLU 0.2	BN
Convolution	2x2 kernel, 2 stride	1	Sigmoid	-

All convolution and transposed convolution layers are without bias. Other parameters batch size : 64, size z: 100, epochs: 120. Optimizers for Generator and Discriminator: Adam ($lr = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999$). Both optimizers decrease learning rate by 0.5 every 25 epoch.

Table A.7: SVHN - Variational Autoencoder

Layer	Filter	Features	Activation	Other
Decoder				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride	3	Tanh	-
Encoder				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	512	Leaky ReLU 0.2	BN
Convolution	2x2 kernel, 2 stride	256	-	-

Other parameters: batch size: 128, size z: 128, epochs: 50, beta: 1 (1 x KLD part of loss). Optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$).

Table A.8: SVHN - Classifier

Layer	Filter	Features	Activation	Other
Convolution	3x3 kernel, 2 stride, 1 padding	32	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	32	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Convolution	3x3 kernel, 2 stride, 1 padding	64	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	64	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Convolution	3x3 kernel, 2 stride, 1 padding	128	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	128	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Dense Layer	-	1024	ReLu	BN, Dropout 0.5
Dense Layer	-	1024	ReLu	BN, Dropout 0.5
Dense Layer	-	10	Softmax	-

Other parameters: batch size: 128, epochs: 50. Optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$).

Table A.9: SVHN - Adversarially Learned Inference

Layer	Filter	Features	Activation	Other
Generator X				
Transposed Conv.	4x4 kernel, 1 stride	256	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 2 stride	128	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 1 stride	64	Leaky ReLU 0.1	BN
Transposed Conv.	4x4 kernel, 2 stride	32	Leaky ReLU 0.1	BN
Transposed Conv.	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	3	Sigmoid	-
Generator Z				
Convolution	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 2 stride	64	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 1 stride	128	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 2 stride	256	Leaky ReLU 0.1	BN
Convolution	4x4 kernel, 1 stride	512	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	BN
Convolution	1x1 kernel, 1 stride	128	-	-
Discriminator - X part				
Convolution	5x5 kernel, 1 stride	32	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 2 stride	64	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 1 stride	128	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 2 stride	256	Leaky ReLU 0.1	BN, Dropout 0.2
Convolution	4x4 kernel, 1 stride	512	Leaky ReLU 0.1	BN, Dropout 0.2
Discriminator - Z part				
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	512	Leaky ReLU 0.1	Dropout 0.2
Discriminator - Joint part				
Convolution	1x1 kernel, 1 stride	1024	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	1024	Leaky ReLU 0.1	Dropout 0.2
Convolution	1x1 kernel, 1 stride	1	-	Dropout 0.2

Input images are scaled to (0,1). Other parameters: Batch size: 128, Size z: 64, Epochs: 500, Optimizer for Generator: Adam ($lr = 10^{-4}, \beta_1 = 0.5, \beta_2 = 10^{-3}$) Optimizer for Discriminator: Adam ($lr = 10^{-5}, \beta_1 = 0.5, \beta_2 = 10^{-3}$).

Table A.10: SVHN - Generative Adversarial Network

Layer	Filter	Features	Activation	Other
Generator				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride	3	Tanh	-
Discriminator				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	512	Leaky ReLU 0.2	BN
Convolution	2x2 kernel, 2 stride	1	Sigmoid	-

All convolution and transposed convolution layers are without bias. Other parameters batch size : 64, size z: 100, epochs: 200. Optimizers for Generator and Discriminator: Adam ($lr = 0.0002, \beta_1 = 0.5, \beta_2 = 0.999$).

Table A.11: SVHN - Variational Autoencoder

Layer	Filter	Features	Activation	Other
Decoder				
Transposed Conv.	4x4 kernel, 1 stride	512	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	256	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	128	ReLU	BN
Transposed Conv.	4x4 kernel, 2 stride, 1 padding	64	ReLU	BN
Transposed Conv.	1x1 kernel, 1 stride	3	Tanh	-
Encoder				
Convolution	4x4 kernel, 2 stride, 1 padding	64	Leaky ReLU 0.2	-
Convolution	4x4 kernel, 2 stride, 1 padding	128	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	256	Leaky ReLU 0.2	BN
Convolution	4x4 kernel, 2 stride, 1 padding	512	Leaky ReLU 0.2	BN
Convolution	2x2 kernel, 2 stride	256	-	-

Other parameters: batch size: 128, size z: 128, epochs: 100, beta: 1 (1 x KLD part of loss). Optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$).

Table A.12: CIFAR - Classifier

Layer	Filter	Features	Activation	Other
Convolution	3x3 kernel, 2 stride, 1 padding	32	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	32	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Convolution	3x3 kernel, 2 stride, 1 padding	64	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	64	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Convolution	3x3 kernel, 2 stride, 1 padding	128	ReLU	BN
Convolution	3x3 kernel, 2 stride, 1 padding	128	ReLU	BN
MaxPool	2x2 kernel, 2 stride	-	-	Dropout 0.3
Dense Layer	-	1024	ReLu	BN, Dropout 0.5
Dense Layer	-	1024	ReLu	BN, Dropout 0.5
Dense Layer	-	10	Softmax	-

Other parameters: batch size: 128, epochs: 50, optimizer: Adam ($lr = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$). Optimizer decrease learning rate by 0.35 every 25 epoch.