Improving Robustness of Deep Learning Systems with Fast and Customizable Adversarial Data Generation

by

Mehmet Melih Arıcı

B.S., Computer Sceince, Özyeğin University, 2018

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2021

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor Prof. Alper Sen for patiently guiding and mentoring me with his experience and knowledge. I would like to thank fellow jury members Asst. Prof. İnci Meliha Baytaş and Asst. Prof. Reyhan Aydoğan for their invaluable feedback. I am also grateful to my parents Mustafa and Ayşe Melahat Arıcı, my sister Merve Çakıroğlu and my grandmother Nezire Turna for their unlimited support. I would not complete this thesis without them.

ABSTRACT

Improving Robustness of Deep Learning Systems with Fast and Customizable Adversarial Data Generation

Deep Learning (DL) is the force behind the success of solving many complicated tasks in recent years. With the use of DL systems in safety-critical applications, it has become of great importance to make these systems robust against adversarial attacks. Adversarial data generation is an effective tool to make DL systems robust against such attacks, with the help of adversarial training. Recent studies focus gradient-based adversarial attacks. Although they can successfully generate adversarial samples, high computation cost and lack of flexibility over input generation arise the need for an efficient and flexible adversarial attack methodology.

In this thesis, we present a fast and customizable adversarial data generation framework towards bridging this gap. Convolutional autoencoders with custom loss functions, enable user-configurable data generation within a much shorter time compared to the state-of-the-art attack method called PGD. We integrate suspiciousness metric from traditional software engineering and a feature importance metric into our custom loss functions. Experiments show that our technique produces adversarial samples faster than PGD and using these samples in adversarial training, allows comparable robustness against adversarial attacks.

ÖZET

Hızlı ve Özelleştirilebilir Hasım Veri Üretimi ile Derin Öğrenme Sistemlerinin Sağlamlığını İyileştirme

Derin Öğrenme (DÖ), son yıllarda birçok karmaşık problemin çözülebilme başarısının arkasındaki güçtür. DÖ sistemlerinin güvenlik açısından kritik uygulamalarda kullanılması ile, bu sistemleri düşman saldırılarına karşı sağlam kılmak önem kazanmıştır. Hasım veri üretimi, hasım eğitimin yardımıyla DÖ sistemlerini bu tür saldırılara karşı sağlamlaştırmak için etkili bir araçtır. Son çalışmalar, gradyan temelli hasım ataklara odaklanmaktadır. Başarılı bir şekilde hasım örnekleri oluşturabilmelerine rağmen, bu atakların yüksek hesaplama maliyetine sahip olmaları ve girdi oluşturma sürecinde yeterince esnek olamamaları, verimli ve esnek bir hasım atak metodolojisine olan ihtiyacı ortaya koymaktadır.

Bu tezde, bu açığı kapatmaya yönelik hızlı ve özelleştirilebilir bir hasım veri üretme çerçevesi sunuyoruz. Özel kayıp fonksiyonlarına sahip evrişimli otomatik kodlayıcılar, PGD adı verilen son teknoloji saldırı yöntemine kıyasla çok daha kısa sürede, kullanıcı tarafından yapılandırılabilir veri üretimine olanak tanıyor. Geleneksel yazılım mühendisliğinden gelen şüphelilik metriğini ve bir özellik önemi metriğini, özel kayıp fonksiyonlarımıza entegre ediyoruz. Deneylerimiz gösteriyor ki, tekniğimiz hasım örnekleri PGD'den daha hızlı üretebilmektedir. Ayrıca, üretilen hasım örnekleri hasım eğitimde kullanmak, hasım saldırılara karşı kıyaslanabilir bir sağlamlık sağlamaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES ix
LIST OF SYMBOLS x
LIST OF ACRONYMS/ABBREVIATIONS xii
1. INTRODUCTION 1
2. LITERATURE SURVEY 4
3. BACKGROUND 8
3.1. Adversarial Samples, Attacks and Defenses 8
3.1.1. Adversarial Samples 8
3.1.2. Adversarial Attacks 9
3.1.3. Fast Gradient Sign Method (FGSM) 9
3.1.4. Projected Gradient Descent (PGD) 10
3.1.5. Adversarial Defense 10
3.2. Autoencoders 11
3.3. Suspiciousness 13
3.3.1. Software Fault Localization (FL) 13
3.3.2. Adaptation of FL to Artificial Neural Networks 14
3.4. Layer-wise Relevancy Propagation (LRP) 15
4. METHODOLOGY 16
4.1. Data Generation Pipeline 16
4.2. Data Generator Network (DGN) Architectures 19
4.3. Custom Loss Functions 21
4.3.1. LRP Custom Loss 21
4.3.2. NCE Custom Loss 22
4.3.3. Suspiciousness Custom Loss 23

4.4. Adversarial Verification of Generated Samples 25
4.5. Adversarial Defense 26
4.6. Evaluation 27
4.6.1. Evaluation Setup 27
4.6.1.1. Datasets 27
4.6.1.2. Target Models 27
4.6.1.3. Training Configurations 28
4.6.1.4. Experimental Design 28
4.6.1.5. Research Questions 30
4.6.2. Evaluation Results 31
4.6.2.1. RQ1 (Attack Performance) 32
4.6.2.2. RQ2 (Defense Performance) 33
4.6.2.3. RQ3 (Comparison) 38
4.6.2.4. RQ4 (Effectiveness) 38
4.6.2.5. RQ5 (Cost) 38
5. CONCLUSIONS 42
REFERENCES 43

LIST OF FIGURES

Figure 3.1. Adversarial sample generation 8

Figure 3.2. A basic illustration of an autoencoder. 12

Figure 3.3. An illustration of layer-wise relevancy calculation 15

Figure 4.1. Data generation workflow 16

Figure 4.2. Attack procedure of our approach. 17

Figure 4.3. Defense procedure of our approach. 18

Figure 4.4. Evaluation procedure of our approach. 19

Figure 4.5. DGN architecture for MNIST dataset. 20

Figure 4.6. DGN architecture for CIFAR dataset. 20

Figure 4.7. Generated images by DGN networks 32

Figure 4.8. Data generation time cost comparison 39

LIST OF TABLES

Table 3.1. Suspiciousness measures 14

Table 4.1. Training configurations of DGNs 28

Table 4.2. DGN-adversarial training configurations 29

Table 4.3. FGSM and PGD-adversarial training configurations 30

Table 4.4. Adversarial attack perturbation budgets 30

Table 4.5. Attack performances on MNIST dataset. 33

Table 4.6. Attack performances on CIFAR dataset. 34

Table 4.7. Defense performances against attacks on MNIST dataset. 35

Table 4.8. Defense performances against attacks on CIFAR dataset. 36

Table 4.9. Performance of DGN models against each other 37

Table 4.10. Total time costs of DGN models 40

Table 4.11. Total time costs of all attacks 40

LIST OF SYMBOLS

a_i	Activation value of neuron n_i
C	Number of classes
$\mathcal{D}$	Target neural network model for attack
f	Classifier under attack in custom losses
F	Future map
g_{θ_i}	Data generator network for class i
$\mathcal{G}$	A set of data generator networks
h_i	The largest admissible feature value
J	Cost function
l_i	The smallest admissible feature value
m	Training batch size
n	An artificial neuron
N	Neural network (or non-suspicious indices in custom losses)
p	Program element (or a pixel index in custom losses)
P	Input program (or all pixels indices in custom losses)
$\mathcal{P}$	Problem specific properties
P_R	A set of the most relevant pixel indices
P_N	A set of non-relevant pixel indices
R_i	Relevancy score of input at index i
R_j	Relevancy score of neuron n_j
S	A set of suspicious indices
T	Test dataset (or test suite in fault localization)
w_{ij}	Weight between input x_i (or neuron n_i) and neuron n_j
x	Input sample
x'	Generated input sample
x_i	Input image at data index i
$\widetilde{x}_i$	Generated input image at data index i
$x_{i,p}$	p^{th} pixel value of original input image x_i

$\widetilde{x}_{i,p}$	p^{th} pixel value of generated input image $\widetilde{x}_i$
X_i	Training data for class i
X'_i	Generated training data for class i
$\mathcal{X}_T$	Test dataset
$\mathcal{X}_{T_c}$	Test dataset for class c
$\mathcal{X'}_T$	Generated test dataset
$\mathcal{X'}_{T_c}$	Generated test dataset for class c
y	Label of input sample
ϵ	Adversarial perturbation limit
η	Calculated adversarial perturbation
θ	Model weights
σ	Softmax function

LIST OF ACRONYMS/ABBREVIATIONS

AAE	Adversarial Autoencoder
AE	Autoencoder
BIM	Basic Iterative Method
CAE	Convolutional Autoencoder
C&W	Carlini & Wagner
DGN	Data Generator Network
DL	Deep Learning
DNN	Deep Neural Network
FGSM	Fast Gradient Sign Method
FL	Fault Localization
GAN	Generative Adversarial Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
HGD	High-level Representation Guided Denoiser
JSMA	Jacobian Saliency Map Attack
LRP	Layer-wise Relevancy Propagation
LSTM	Long Short Term Memory
MSE	Mean Squared Error
NCE	Negative of Categorical Cross Entropy
PGD	Projected Gradient Descent
ReLU	Rectified Linear Unit
RSE	Random Self-Ensemble
SUSP	Suspiciousness
VAE	Variational Autoencoder

1. INTRODUCTION

The amount of accessible data, which has grown significantly every day, has enabled data-hungry deep learning (DL) systems to be included in many industrial products. Undoubtedly, the superior problem-solving ability of DL systems also plays a big role in this situation. As might be expected, such powerful systems are expected to be used in solving challenging problems such as image classification [1], speech recognition [2] and natural language processing [3]. Since these problems are core parts of many applications in software industry including air traffic control systems [4], autonomous vehicles [5] and medical diagnostics tools [6], it is easy to say that DL systems are indispensable for safety-critical applications.

An error that may occur in a safety-critical system can pose a threat to human life, damage the environment, cause mass deaths or serious financial losses. Considering the widespread use of DL systems in safety-critical applications, it is of great importance to make these systems more secure against potential threats. In addition, the fact that safety-critical systems are processing large amounts of data in real time reveals that a possible solution to these threats should be fast. Recently, there have been examples including autonomous vehicle accidents [7] and financial losses [8] that show how serious this problem can be.

The use of artificial neural networks in safety-critical applications is questioned after the discovery of adversarial examples that mislead neural networks [9–11]. Starting with Fast Gradient Sign Method (FGSM) [12], many adversarial attack techniques have been proposed [13–16] to generate adversarial samples. Projected Gradient Descent (PGD), a multi-step form of FGSM, has emerged as the strongest first-order adversarial attack [17].

Given the brittle nature of neural networks against adversarial samples, defense mechanisms against potential attacks are needed. Many adversarial defense techniques

have been developed [18–25] to this end. The idea of incorporating adversarial samples into training process together with original samples is aroused in [10,12]. This technique is called as adversarial training which is one of the most successful adversarial defense approaches. Time required for adversarial data generation is a significant criterion to make a safety-critical DL system robust against attacks, using adversarial training. Despite the high attack power of state-of-the-art PGD, high data generation cost makes the use of PGD in safety-critical systems impractical.

While the protection of safety-critical systems is a priority, the extensive use of neural networks in various domains, requires the development of problem-specific attack and defense techniques. Hence, a framework which allows customizable robustness against domain specific attacks at a reasonable cost, is needed. To meet this need, we propose a fast and customizable adversarial data generation framework that uses Convolutional Auto Encoders (CAE) with custom loss functions.

Together with CAEs we also employ several loss functions to develop a customizable adversarial data generation framework. One such loss function is based on suspiciousness metric from software fault localization. Suspiciousness allows finding suspicious neurons of a network that are potentially responsible for erroneous network behaviours [26]. We use suspiciousness in a custom loss function to force our system to generate adversarial samples which increase activations of suspicious neurons. Another methodology that we integrated in our custom losses is Layer-wise Relevancy Propagation (LRP), which allows understanding of the most relevant (i.e. important) and non-relevant input features for a given network. LRP is used to force our system to generate adversarial inputs by increasing weights of non-relevant features and decreasing weights of relevant features. Finally, we integrate the Negative of categorical Cross Entropy (NCE), into one of our custom loss functions.

Overall, the main contributions of this thesis as follows:

- We propose an adversarial data generation framework which allows domain-specific attacks. Samples generated by our framework are used in an adopted version of adversarial training presented in [12] to obtain robust networks.
- We present three custom loss formulations; LRP, Suspiciousness and NCE to generate adversarial inputs that exploit significant structures of a neural network.
- We evaluate both attack and defense capacity of our approach against the state-of-the-art adversarial attacks on two public datasets (MNIST [27], CIFAR10 [28]). Our technique is faster than other attacks in data generation and shows comparable defense performance against them.
- Source code of our framework and data generation neural networks are freely available at https://github.com/FastCustomizableCAE/DeepCustom.

To the best of our knowledge, our work is the first framework that allows customizable adversarial data generation and it integrates a software engineering metric called suspiciousness into neural network loss functions.

Rest of this thesis is structured as follows. In Chapter 2, we present related studies with our work and in Chapter 3, we provide necessary background information. Chapter 4 covers our methodology and shows experimental results. Finally in Chapter 5, we conclude this thesis and give a discussion about future works.

2. LITERATURE SURVEY

In this chapter, we present our literature survey on related studies. For this purpose, we investigated works related to DNN testing, adversarial attacks, defenses and data generation. We start our discussion with DNN testing and corner-case test generation frameworks [29–31] since the work presented in this thesis offers generation of adversarial samples (i.e. corner-case test inputs) for DNNs.

The black-box nature of DNNs makes it difficult to find neurons, layers that responsible for wrong classification decisions and identify corner-case input that cause DNNs to make false predictions [32]. Integrated from software engineering, white-box testing techniques are adapted to DNN testing to expose faulty parts of DNNs and generate corner-case inputs.

DeepXplore [29] proposed a metric called *neuron coverage* similar to code coverage in traditional software engineering. Generating corner-case behaviors by triggering more neurons in a DNN (i.e. increasing neuron coverage) is formulated as an optimization problem. As a result, thousands of corner-case behaviors in state-of-the-art dataset and models are found using DeepXplore [29].

The positive effect of increasing neuron coverage to generate corner-case behaviors, urged researchers to work on new metrics. DeepGauge [30] presented a set of multi-granularity coverage metrics to test DNNs. Achieving high coverage results on adversarial samples shows the effectiveness of their metrics to test DNNs.

Autonomous car software is a safety-critical application where DNNs are used. To test DNNs in autonomous cars, DeepTest [31] offered a systematic framework to automatically detect erroneous behaviors of DNN-driven autonomous car systems. DeepTest [31] generates corner test cases which contain real-world conditions such as different lighting effects or challenging weather conditions (e.g. rain, fog). The aim

of generating such inputs is to maximize neuron coverage. Similar to DeepXplore [29], DeepTest [31] found thousands of erroneous behaviors on state-of-the-art autonomous driving datasets.

The generation of adversarial samples which cause DNNs to make false predictions, started with [10], then [12] offered Fast Gradient Sign Method (FGSM) to generate adversarial samples. Adversarial samples can also be found in physical world [13]. Subsequently, many adversarial attacks were proposed [13–17].

Basic Iterative Method (BIM) is emerged as an multi-step iterative version of FGSM where the algorithm clip pixel values of intermediate results after each step [13]. Jacobian Saliency Map Attack (JSMA) [14] uses forward derivatives to construct adversarial saliency maps which show the amount of perturbations for input features to include, in order to produce an adversarial sample in the end. DeepFool [15] efficiently calculates minimal perturbations to generate adversarial samples for DNNs. Carlini & Wagner (C&W) attack [16] is presented as an adversarial attack that is effective against defensive distillation [33]. Among other attacks, Projected Gradient Descent (PGD) [17] which is similar to BIM, is appeared to be a universal state-of-the-art attack that uses local first order information of target DNN models. None of these adversarial attacks allow full control of users on data generation. In our evaluation, we compare our technique with FGSM and PGD attacks in different configurations.

In adversarial defense side, [12] offered adversarial training using FGSM attack and [17] advanced this approach using PGD. Besides adversarial training, other defense approaches have been proposed [18–25, 33] to increase robustness of DNNs.

Random Self-Ensemble (RSE) [18] prevents effects of strong gradient-based attacks by adding random noise layers. A set of transformations is applied to image dataset to defend against adversarial attacks in [19]. MagNet [20] uses detector and reformer networks to differentiate adversarial samples from originals. Similar to MagNet, [21] utilizes small detector networks which are capable of distinguishing adversarial

data, to augment DNNs. High-level representation guided denoiser (HGD) [22] as an adversarial defense for image classification, designed a loss function which is the difference between target model's outputs on clean image and denoised image. HGD does not suffer from the error amplification effect which damages success of standard denoisers [22]. Defense-GAN [23] employs GANs [34] to model distribution of unperturbed images and hence it provides adversarial robustness by finding the closest unperturbed image output of a given adversarial image and feeding the output to the classifier during inference. Thermometer Encoding [24] discretises and quantizes DNN inputs to defend against adversarial samples. Similar to Defense-GAN [23], PixelDefend [25] purifies adversarial perturbed image by moving it back towards an unperturbed version which can be observed in training data. Defensive distillation [33] technique trains a second network which has the same architecture with original network, but it is trained with *soft labels* (i.e. more-smooth version of softmax function is used instead of traditional softmax during training) to avoid over-fitting against training data. Reducing over-fitting effect removes *blind spots* [10] where adversarial samples lie and hence provides robustness. Our approach uses adversarial training with FGSM, PGD and DGN (i.e. Data Generator Network presented in Section 4.1) generated adversarial samples.

Robustness improvements are further investigated in elimination of critical nodes or weights of a model [35, 36]. To identify design problems and vulnerabilities of DNN systems, random testing is used in many studies [29, 31, 37, 38]. An efficient technique to verify robustness of DNN models with ReLU activations is proposed in [39]. Beyond feed-forward and convolutional networks, [40] presents an algorithm to quantify robustness of recurrent, LSTM and GRU networks.

Autoencoders (AE) [41] are used for efficiently learning useful features of data distribution. Convolutional autoencoder (CAE) which is successful at capturing useful features in more complicated data such as medical images [42], is a modified version of traditional AE. An improved version of AE, Variational Autoencoder (VAE) which encodes input as a distribution over latent space, is used in data generation [43,44]. [45] used VAE for synthetic data generation for imbalanced data and [46] used VAE for

generating molecular structures using parse trees from a context-free grammar. We use CAE networks for adversarial data generation in our work.

Adversarial autoencoder (AAE), a probabilistic AE that utilizies Generative Adversarial Networks (GAN)s [34], is proposed in [47] and can be used in generative modeling and semi-supervised classification. An adversarially regularized autoencoder is offered in [48], generates textual outputs, allows training of latent representation for textual style transfer. [49] presented an autoencoder based framework which enhances the performance of black-box attacks.

3. BACKGROUND

In this chapter, we discuss previous studies on which this thesis is based upon. Initially, we will discuss adversarial samples, attacks and defenses. Then we study autoencoders which are core architectures that our work is built upon. Then, we finally discuss about suspiocusness from software fault localization and layer-wise relevancy propagation, which are used for shaping data generationg process.

3.1. Adversarial Samples, Attacks and Defenses

3.1.1. Adversarial Samples

Adversarial samples are inputs crafted from original dataset inputs by adding small perturbations to cause deep learning models to make false predictions [12]. The amount of adversarial perturbation ϵ is often controlled by distance metrics and differs from dataset to dataset. On famous MNIST [27] dataset, 0.3 is a widely-used value for ϵ, where pixel values are in between 0 and 1. On CIFAR [28] dataset where pixel values are from 0 to 255, $\epsilon = 8$ is a common assignment.

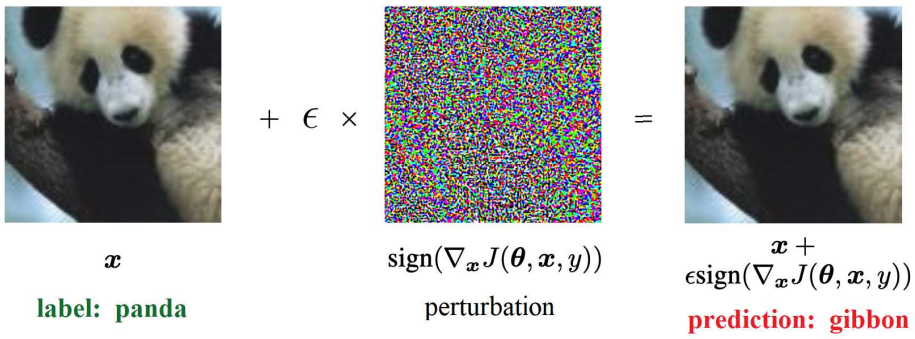

Figure 3.1. An adversarial sample is generated from original sample by adding an adversarial perturbation [12].

Figure 3.1 shows the generation of adversarial samples from its original version. In the left side, an original image is classified as class *panda* by a target deep learning model. A perturbation is calculated using a formula and is multiplied by a constant factor. Finally, the calculated perturbation is added over the original image to obtain an *adversarial sample*. The generated adversarial sample is very similar to the original sample but classified differently by the target model as *gibbon*.

3.1.2. Adversarial Attacks

Adversarial attacks are used to generate adversarial examples as explained in Section 3.1.1. The aim of an adversarial attack is generating samples such that a target model incorrectly classifies the generated samples. Hence, if the accuracy of the target model on generated samples is low, the adversarial attack can be considered as successful. The accuracy of the target model on adversarial samples is called as **robust accuracy**, while accuracy on original samples is called as **natural accuracy**.

3.1.3. Fast Gradient Sign Method (FGSM)

FGSM is a fist order gradient based adversarial attack technique which calculates an adversarial perturbation based on loss function of a target model. FGSM calculates the adversarial perturbation η as sign of gradient of the cost function with respect to input, as shown in Equation 3.1.

$$\eta = \epsilon sgn(\nabla_x J(\theta, x, y))), \tag{3.1}$$

where x is input, θ is parameters of original model, $J(\theta, x, y)$ is a cost function [12]. Adversarial image x' can be obtained by adding the perturbation on top of original image as shown in Equation 3.2.

$$x' = x + \eta. \tag{3.2}$$

The intuition behind this technique is adding scaled noise whose direction is the same as the gradient of the cost function. Adding such noise will cause an increase in the output of the cost function of the target model. Hence, this objective helps to

generate adversarial samples that cause high error on the target model.

3.1.4. Projected Gradient Descent (PGD)

PGD is a state-of-the-art adversarial attack technique, but actually a multi-step variant of FGSM. PGD has the following formulation to calculate adversarial perturbation:

$$x^{t+1} = \Pi_{x+\mathcal{S}}(x^t + \alpha sgn(\nabla_x J(\theta, x, y))), \tag{3.3}$$

where x^t is an input generated at time t and x^{t+1} is an input generated at time $t+1$ by adding a calculated perturbation on top of x^t. $\mathcal{S}$ defines a set of allowed perturbations. For image datasets, perturbations that preserve perceptual similarity between images such as l_∞-ball around x, is a meaningful perturbation set for $\mathcal{S}$. $\Pi_{x+\mathcal{S}}$ is a projection operation to project perturbation back to allowable ϵ-norm ball if it is greater than the allowable perturbation limit ϵ.

PGD is an iterative algorithm. The idea is that in each step, a perturbation is added over the original image in the direction of gradient of loss. The perturbation is projected back into ϵ-norm ball if it is more than allowable perturbation limit. As step size increases, PGD becomes more powerful for generating adversarial samples. However, time required to generate a sample also increases with step size. Although its attack power, the high computation need makes PGD intractable to use in real time systems.

3.1.5. Adversarial Defense

Adversarial defense is a set of techniques to defend against adversarial attacks. The aim of an adversarial attack is to fool a target neural network by generating adversarial samples. The aim of adversarial defense techniques is to make a target neural network more robust against adversarial samples.

Adversarial training is a basic but successful adversarial defense technique that we utilized in our work. In normal training, a target model is trained with original training samples. In adversarial training however, the idea is to incorporate adversarial samples into training process of the target model together with original samples. Based on this explanation, it is clear that we need adversarial samples during adversarial training. Here is a basic formulation of adversarial training:

$$\widetilde{J}(\theta, x, y) = \alpha J(\theta, x, y) + (1 - \alpha) J(\theta, x', y) \tag{3.4}$$

An adversarial sample or a batch of adversarial samples is generated during each training step by a given adversarial attack. The generated samples are then used to calculate adversarial loss. At the end, original and adversarial losses are combined to obtained final loss function of adversarial training. In Equation 3.4, α is a parameter to balance original loss $J(\theta, x, y)$ and adversarial loss $J(\theta, x', y)$. The optimum value for α is suggested as 0.5 in [12]. Note that, target network weights θ in Equation 3.4, change during adversarial training. Hence in each step, training objective adapts itself to changing weights.

Adversarial training definitely increases robust accuracy against adversarial test samples but it is also common to observe a slight accuracy decrease on original test accuracy. This decrease is reasonable since we integrate modified samples into the original dataset which doubles dataset size and increases variety. The slight natural test accuracy decrease is acceptable considering high accuracy gain for robust accuracy.

3.2. Autoencoders

Autoencoder (AE) is an artificial neural network that learns efficient data representations. Autoencoders compress given data into a latent representation and reconstruct it back to a form which is closed to its original version. Figure 3.2 shows a basic structure of an autoencoder. *Encoder* and *Decoder* are two main parts of an autoencoder.

The encoder part $h = f(x)$ takes input and maps it into a latent representation h. Latent representation h is a low dimensional encoding of input data x. The decoder part $r = g(h)$, reconstructs the input back from the latent representation h. Autoencoders learn useful features of input data to construct it back. Note that, this is not copying input to output. Instead, decoder networks learn to generate data from a lower dimensional input.

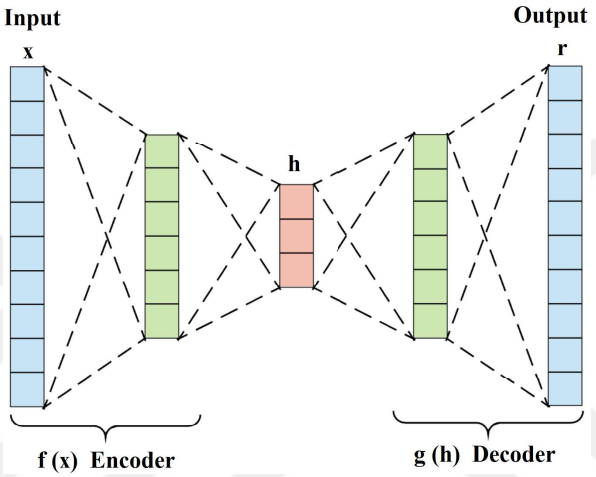

Figure 3.2. A basic illustration of an autoencoder.

An *undercomplete autoencoder* is an autoencoder type where the dimension of latent representation is lower than input dimension. This architecture forces AE to learn the most prominent features of inputs. When the dimension of the latent representation is equal or greater than the input dimension, AE is considered as *overcomplete*. Overcomplete AE networks can learn to copy input to output (identity function) rather than learning useful input properties. To avoid such a case, keeping encoder and decoder networks shallow and to limit latent representation dimension can be a solution but a small AE architrecture may become useless for a complex input distributions. An ideal solution is using *regularized autoencoders* which use a loss function which forces the model to have other properties besides just copying input to output. [41].

Convolutional autoencoder is an advanced version which uses convolutional and deconvolutional layers [50]. Convolutional layers are effective on image inputs where

feed-forward conventional autoencoders are having difficulty to learn complex input distribution. Deconvolutional layers are also known as up-sampling layers which are used to increase given input dimensions. Hence, they are used to generate an image from latent representation through layers of generator network.

3.3. Suspiciousness

Suspiciousness measure from software fault localization (FL) [51] has been integrated to deep neural networks. The aim of FL is to identify suspicious program elements in a software program. In spectrum-based FL, counts of successful and failed test executions are used to calculate a suspiciousness score for each element. Considering that neurons are the main elements of an artificial neural network, suspiciousness allows finding suspicious neurons of a neural network, which are more responsible for erroneous DNN behaviours [26]. We will give a short discussion about software fault localization, where suspiciouss measure comes from.

3.3.1. Software Fault Localization (FL)

Every software has bugs that affect working mechanism of them. Identifying bugs, namely problematic parts of a given software, is crucial to solve underlying problems. FL at this point, is useful to detect problematic parts of a software code that cause buggy behavior. Statements and declarations are some examples of software code.

Notation. P is an input program and T is an input test suite for P. FL process tests P against T. During process, FL extracts passed and failed test cases. In spectrum-based FL, the spectrum tuple $(attr^{as}, attr^{af}, attr^{nf}, attr^{ns})$ is calculated for each program element $p \in P$. $attr^{as}$ and $attr^{af}$ represent cases where p is executed (i.e. $attr^{as}$: active and succeed, $attr^{af}$: active and failed). On the other hand, $attr^{nf}$ and $attr^{ns}$ are the cases where p is not executed (i.e. $attr^{ns}$: not-active and succeed, $attr^{nf}$: not-active and failed). Members $attr^{as}$,$attr^{ns}$ are cases where the test succeeds and $attr^{af}$,$attr^{nf}$ are cases where the test fails. Spectrum of each program element p is calculated in spectrum-based FL, which then will be used suspiciousness calculations of those programs elements. Note that $attr^{as}, attr^{af}, attr^{nf}, attr^{ns}$ are the numbers that

show how many times the condition it represents occur. For instance if $attr^{as}$ of a program element p is 200, then p was active in 200 times when the test case succeeds.

FL technique enables finding spectrum of each program element p. One can ask how to find suspiciosness scores of each p, given their spectrum tuples in a program. At this point, suspiciousness measures will be used where they have been defined in Table 3.1.

Table 3.1. Suspiciousness measures. * is a variable greater than 0.

Measure	**Formula**
Tarantula [52]	$\frac{attr_n^{\mathrm{af}}/(attr_n^{\mathrm{af}} + attr_n^{\mathrm{nf}})}{attr_n^{\mathrm{af}}/(attr_n^{\mathrm{af}} + attr_n^{\mathrm{nf}}) + attr_n^{\mathrm{as}}/(attr_n^{\mathrm{as}} + attr_n^{\mathrm{ns}})}$
Ochiai [53]	$\frac{attr_n^{\mathrm{af}}}{\sqrt{(attr_n^{\mathrm{af}} + attr_n^{\mathrm{nf}}) \cdot (attr_n^{\mathrm{af}} + attr_n^{\mathrm{as}})}}$
D* [54]	$\frac{attr_n^{\mathrm{af}*}}{attr_n^{\mathrm{as}} + attr_n^{\mathrm{nf}}}$

3.3.2. Adaptation of FL to Artificial Neural Networks

Eniser et al. [26] showed that spectrum-based FL can be applied to artificial neural networks. A neural network N can be considered as an input program where spectrum tuple of each neuron is calculated using a test dataset T. Note that in this approach, neurons correspond to program elements of software FL. After calculation of spectrum tuples of each neuron in network, one of suspiciousness measures presented in Table 3.1, are applied to find suspiciousness scores of each neuron. Suspiciousness measures help to identify the most suspicious neurons which are more often activated by test inputs when DNN made an incorrect decision, and less often activated by test inputs when DNN made a correct decision [26].

3.4. Layer-wise Relevancy Propagation (LRP)

Neural networks are known for their black-box nature. Layer-wise relevancy propagation (LRP) is one of the techniques to interpret a neural network's decisions using deep Taylor decomposition [55]. After a forward-pass, LRP propagates output back through the network from output layer to input values and assigns a relevancy score to neurons and input elements. Hence, LRP allows understanding of the most relevant (i.e. important) input features for a given network.

$$R_i = \sum_j \frac{x_i w_{ij} - l_i w_{ij}^+ - h_i w_{ij}^-}{\sum_{i'} x_{i'} w_{i'j} - l_i w_{i'j}^+ - h_i w_{i'j}^-} R_j, \tag{3.5}$$

where x_i is an input pixel that satisfies $l_i \leqslant x_i \leqslant h_i$ such that $l_i \leqslant 0$ and $h_i \geqslant 0$ are the smallest and the largest admissible pixel values for each dimension. w_{ij} is the weight between the input pixel and j^{th} neuron in the next layer. R_j is the relevancy score of jth neuron in the next layer, where it is cumulatively calculated from the output layer towards previous layers. w_{ij}^+ refers to positive and w_{ij}^- refers to negative part of w_{ij} [55]. Equation 3.5 shows the calculation of relevancy of input feature at position i. If our input is an image, then relevancy score of each pixel is calculated using Equation 3.5.

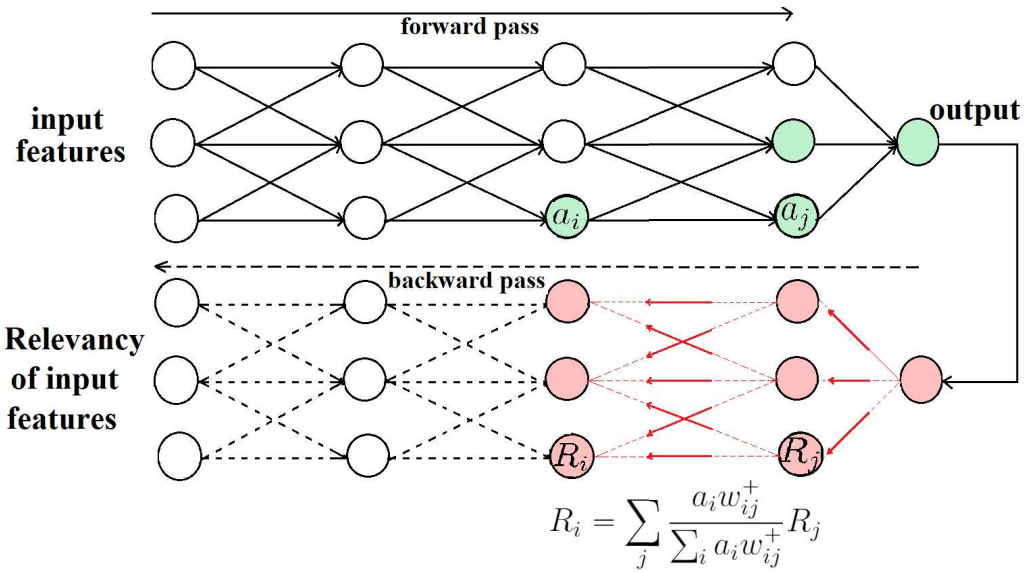

Figure 3.3. An illustration of relevancy calculation for a feed-forward neural network. From output layer to the input layer, relevancy is calculated cumulatively.

4. METHODOLOGY

The core idea behind our technique is using deep CAEs with custom loss functions that are defined based on user requirements. Hence, we offer a flexible framework to users. We studied three different custom loss functions to this end, which are explained in Section 4.3 and compared their abilities to produce adversarial data.

4.1. Data Generation Pipeline

Figure 4.1 shows the data generation pipeline of our framework. Adversarial data is produced through deep CAE networks called Data Generation Networks (DGNs) which are a form of regularized autoencoders [41]. There are k DGNs in the pipeline where k is the number of classes in a given dataset. Each DGN is trained with training instances that belong to its class. DGNs produce a batch of generated samples from a clean batch in each iteration during training. The clean and generated batches can be used in custom loss functions to calculate loss and update weights via back propagation.

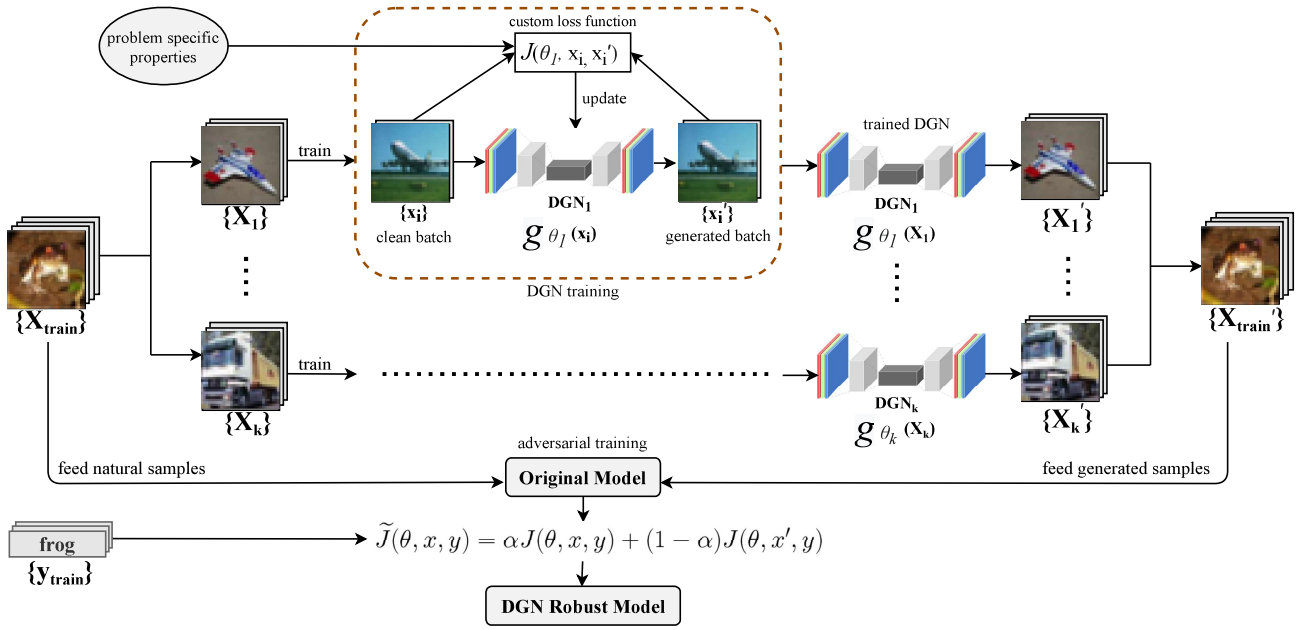

Figure 4.1. Data generation workflow. After generating adversarial training data using DGNs, both original and generated data are used in adversarial training to obtain a DGN robust model.

In addition to clean and generated batches, custom loss functions used in DGNs may require *problem specific properties* (p.s.p). For LRP, problem specific properties are *relevant pixels* which are pixels of images that affect the decision of the original model the most. For suspiciousness, properties are *suspicious neurons* of the original model. The NCE loss function on the other hand, does not need any problem specific properties to calculate the loss. In a nutshell, loss functions of DGNs are customizable and can be shaped according to the desired purpose.

Once DGNs for each class are trained, original training samples of each class are fed to the corresponding DGN to obtain adversarial samples. Generated samples are then collected to build an adversarial training dataset that will be used in adversarial training of the original model together with original training samples.

1: **Require** $\mathcal{D}$: Target Model.
2: $\mathcal{X}$: Training Dataset.
3: $\mathcal{P}$: problem specific properties
4: $\mathcal{X}' \leftarrow \varnothing$ {generated data vector}
5: $\mathcal{G} \leftarrow \varnothing$ {trained DGNs set}
6: **for** all class $i \in \{1, ..., k\}$ **do**
7: $x_c = \textsc{FindClassSamples}(\mathcal{X}, i)$
8: $p_c = \textsc{FindProperties}(\mathcal{P}, i)$ {class specific properties}
9: $g = \textsc{TrainDGN}(x_c, p_c)$
10: $x'_c = g(x_c)$ {generate data using the trained DGN}
11: $\mathcal{X}' = \mathcal{X}' \bigcup x'_c$ {collect generated training data}
12: $\mathcal{G} = \mathcal{G} \bigcup g$ {save the trained DGN}
13: **end for**
14: **Return** $\mathcal{X}'$, $\mathcal{G}$

Figure 4.2. Attack procedure of our approach.

Figure 4.2 shows data generation algorithm in high level. Target model $\mathcal{D}$, training dataset $\mathcal{X}$ and p.s.p $\mathcal{P}$ are inputs. For each class, algorithm finds training samples (line 7) and p.s.p (line 8) for that class. After a DGN is trained (line 9) using these

samples and corresponding p.s.p, trained DGN is used for generating adversarial training data for that class (line 10). Generated data (line 11) and trained DGN (line 12) is collected to be used later in adversarial training and adversarial test data generation, respectively.

Although it is not a must to use a separate DGN for each class, we experimentally noticed that class-specific DGNs (i.e. DGNs with class-specific custom loss properties) are better in generating samples that satisfy problem-specific features. For instance in LRP, working with relevant pixels for each class is more meaningful than using relevant pixels for whole dataset, since relevant pixels vary a lot from class to class. The same situation applies to suspiciousness where suspicious neurons that are found for one class are significantly different from neurons found for other classes. However, depending on custom loss used, it is not necessary to train k different DGNs. We observed that using different DGNs for each class gives better results for custom losses we incorporated. A different custom loss function design may not require this setting.

1: **Require** $\mathcal{D}$: Target Model.
2: $\mathcal{X}$: Original Training Dataset.
3: $\mathcal{X}'$: Adversarial Training Dataset.
4: ϵ: Maximum Allowed Adversarial Perturbation.
5: $\mathcal{X}' = \textsc{EpsilonClipping}(\epsilon, \mathcal{X}')$
6: $\mathcal{D}_R = \textsc{AdversarialTrain}(\mathcal{D}, \mathcal{X}, \mathcal{X}')$ {obtain robust model}
7: **Return** $\mathcal{D}_R$

Figure 4.3. Defense procedure of our approach.

Figure 4.3 shows our defense algorithm. Adversarial training data $\mathcal{X}'$ which is returned in attack algorithm presented in Figure 4.2, is used in our defense algorithm. At first, we apply *epsilon clipping* which is described in Section 4.4 to adversarial training data generated by our DGNs, in line 5. The aim of this step is to verify that adversarial samples satisfy required adversarial perturbation budget. Then, verified adversarial training samples, original training samples and target model are used in *adversarial training* to obtain a robust target model in line 6. Finally, algorithm returns

the robust model in line 7.

1: **Require** $\mathcal{D}$: Target Model.
2: $\mathcal{D}_R$: Robust Target Model
3: $\mathcal{X}_T$: Original Test Dataset.
4: $\mathcal{G}$: Data Generator Networks
5: ${\mathcal{X}'}_T \leftarrow \varnothing$ {adversarial test dataset}
6: **for** all class $i \in \{1, ..., k\}$ **do**
7: $\quad g_c = \mathcal{G}[i]$ {get dgn for class c}
8: $\quad {\mathcal{X}'}_{T_c} = g_c(\mathcal{X}_{T_c})$ {generate adversarial test data for class c}
9: $\quad {\mathcal{X}'}_T = {\mathcal{X}'}_T \bigcup {\mathcal{X}'}_{T_c}$ {collect generated test data}
10: **end for**
11: **Return** $\mathcal{D}(\mathcal{X}_T), \mathcal{D}({\mathcal{X}'}_T), \mathcal{D}_R(\mathcal{X}_T), \mathcal{D}_R({\mathcal{X}'}_T)$ {Natural and robust accuracies}

Figure 4.4. Evaluation procedure of our approach.

The attack algorithm trains DGN models and generates adversarial training data using DGNs. The defense algorithm verifies generated adversarial training data and apply adversarial training to obtain a robust model which is adversarially robust version of original target model. The evaluation algorithm presented in Figure 4.4, used DGNs to generate adversarial test data to evaluate original and robust target models. The evaluation algorithm returns original test accuracy of original model, robust test accuracy of original model, original test accuracy of robust model and robust test accuracy of robust model respectively in line 11.

4.2. Data Generator Network (DGN) Architectures

The DGN models used in data generation are convolutional autoencoders. Due to the difference in difficulty of generating data for MNIST and CIFAR datasets, we designed two different DGN architectures for MNIST and CIFAR. Both DGN models use 3x3 filters in their convolutional layers to encode data. For decoding, transposed convolutional (i.e. deconvolutional) layers are used. Batch normalization is applied after each deconvolutional layer.

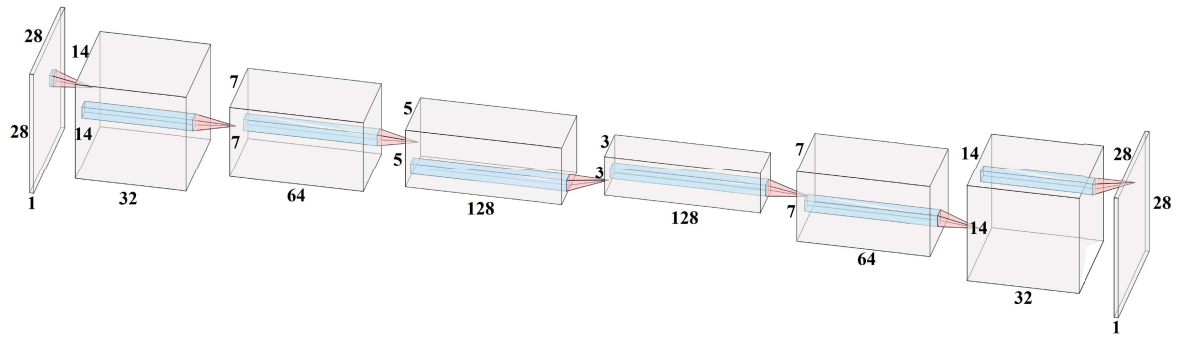

Figure 4.5. DGN architecture for MNIST dataset.

Figure 4.5 shows DGN architecture for MNIST dataset. In encoder part, it consists of three convolutional layers with stride 1 and ReLU activations. First two convolutional layers use the same padding (i.e. padding that causes the output as the same as the input) and the third one uses no padding. After each convolutional layer, there is a MaxPool layer which uses 2x2 pool size with stride 2. The size of latent representation is 3x3x128. In decoder part, three deconvolutional layers with 2x2 filters and no padding are used. The first deconvolutional layer uses stride of 3, while others use stride of 2. Finally, the output of the final deconvolutional layer is passed through sigmoid activation function to keep output values between 0-1 which are valid pixel bounds for MNIST.

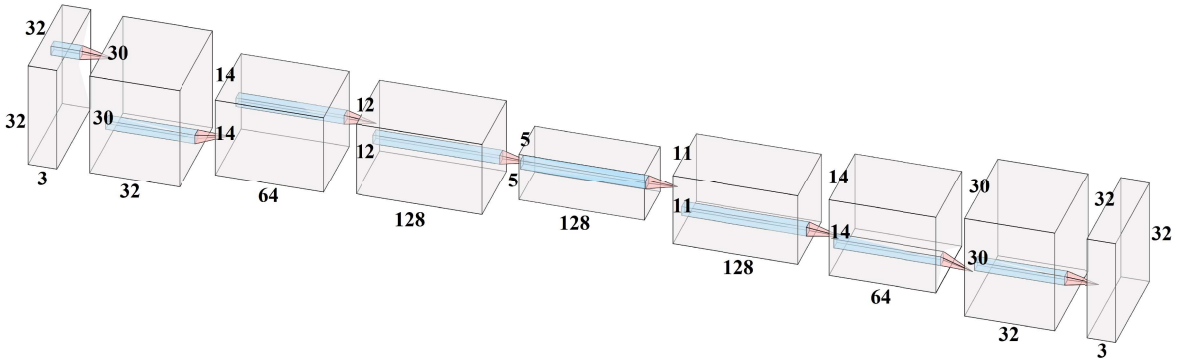

Figure 4.6. DGN architecture for CIFAR dataset.

Figure 4.6 shows DGN architecture for CIFAR dataset. The architecture is deeper and has much more capacity in comparison to MNIST version. The encoding part has four convolutional layers with no padding where the first and the third have stride size of 2, and others have stride size of 1. The size of latent representation is 5x5x128. Four deconvolutional layers with different stride and kernel sizes (i.e. filters) are used

for decoding. Kernel sizes of 2,1,2,1 and strides of 3,4,4,3 are used, but no padding is used.

4.3. Custom Loss Functions

The loss functions of DGNs are customizable which allows generating data according to user requirements. We have developed three different *custom loss functions*, namely LRP, NCE and Suspiciousness custom losses, in order to produce adversarial data. All three custom losses have different objectives that addresses adversarial data generation from different perspectives. However, any other custom loss function can be incorporated as well.

Notation. In all custom losses, f represents an original model which is the classifier under attack and g refers to a DGN network. x is an original input and $\widetilde{x}$ is the input generated by DGN network using x. Hence, $g_\theta(x) = \widetilde{x}$, where θ stands for the weights of the DGN. The mini-batch inputs $x_1, ..., x_m$ are used in each DGN training iteration where the batch size is m. Finally, P represents all pixel indices of an image. Since each MNIST image has 784 pixels, for an MNIST image, P is a set of numbers from 1 to 784. $x_{i,p}$ and $\widetilde{x}_{i,p}$ refer to the pth pixel values of original input x_i and the generated input $\widetilde{x}_i$, respectively.

4.3.1. LRP Custom Loss

LRP allows us to identify *relevant pixels* that contribute most to the decision taken by a neural network. Taking advantage of this technique, we split the data by class and find most relevant k pixels for each class. At the end of this problem-specific process, we have the indices of k most relevant pixels for each class, which are then used in the following LRP custom loss:

$$\mathcal{L}_{LRP} = \frac{1}{m}\sum_{i=1}^{m}\left(\alpha\frac{\sum_{p\in P_R}((x_{i,p}-\epsilon)-\widetilde{x}_{i,p})^2}{\|P_R\|} + (1-\alpha)\frac{\sum_{p\in P_N}((x_{i,p}+\epsilon)-\widetilde{x}_{i,p})^2}{\|P_N\|}\right) \quad (4.1)$$

where P_R is the set of most k relevant pixel indices where the relevancy score of each pixel is calculated using the following LRP-z rule [55] which uses deep Taylor decomposition presented in Equation 3.5.

P_N in $\mathcal{L}_{LRP}$, is a set of *non-relevant* pixel indices such that $P_N = P - P_R$. Note that, relevant and non-relevant pixel indices are the same for all images belonging to the same class. The distance value ϵ is used to keep adversarial perturbation in L_∞ norm-ball. α is a parameter to balance relevant and non-relevant losses.

The aim of $\mathcal{L}_{LRP}$ loss is to force the DGN network to generate inputs such that the values of relevant pixels decrease while the values of non-relevant pixels increase by ϵ, with the help of Mean Squared Error (MSE) loss. By using such a loss function, the impact of non-relavant pixels (i.e. relavant pixels of other classes) will be increased and the impact of relevant pixels, which are responsible for the original network's correct decision, will be decreased. Hence, the generated images will be incorrectly classified by the original model, in accordance with the purpose of an adversarial attack.

4.3.2. NCE Custom Loss

The second custom loss we use is NCE custom loss which penalizes DGN network with the negative of categorical cross entropy loss used in the original model. Hence, if DGN generates an input that causes a high error on the original model, DGN's loss will be lowered, or vice versa. More formally, NCE custom loss is defined as follows:

$$\mathcal{L}_{NCE} = \frac{1}{m}\sum_{i=1}^{m}\left(\alpha\frac{\sum_{p\in P}(x_{i,p}-\widetilde{x}_{i,p})^2}{\|P\|} - \beta\Big(-\sum_{j}^{C} y_j log(\sigma(\widetilde{y}_i)_j)\Big)\right) \tag{4.2}$$

where $\sigma(\widetilde{y}_i) = e^{\widetilde{y}_i}/\sum_k^C e^{\widetilde{y}_k}$ is the softmax function to convert logits into class probabilities. C represents number of classes, y_j is the ground truth one-hot vector and $\widetilde{y}_i$ is the output of the original model on the generated input $\widetilde{x}_i$. The output of function $\sigma(\widetilde{y}_i)_j$ in cross-entropy loss, gives the output probability of the original model for class j on generated input $\widetilde{x}_i$. The parameters α and β regularize MSE and negative cross entropy losses.

Similar to LRP loss, we use MSE in the first part of equation but this time the aim is to make generated input pixels close to the original input pixels. At the same time, we also add the negative of original cross entropy loss in the second part to force DGN to generate inputs that are incorrectly classified by the original model.

4.3.3. Suspiciousness Custom Loss

The purpose of suspiciousness custom loss is to enforce DGN to produce inputs that increase the activation values of *suspicious neurons* in the original model while keeping the activation values of non-suspicious neurons the same. Identification of suspicious neurons is the problem-specific part of this approach. Although increasing the activations of x number of suspicious neurons is effective for dense networks, we observed that the same effect is very limited for convolutional networks since they have much more capacity (i.e. contain many neurons) in comparison to dense networks. Therefore, we define the following suspiciousness custom loss function differently for dense and convolutional layers.

$$\mathcal{L}_{Susp} = \frac{1}{m}\sum_{i=1}^{m}\left(\lambda\frac{\sum_{s\in S}(f(x_i)_s - f(\widetilde{x}_i)_s)}{\|S\|} + \gamma\frac{\sum_{n\in N}|f(x_i)_n - f(\widetilde{x}_i)_n|}{\|N\|}\right) \tag{4.3}$$

For dense layers of the original model, S is a set of suspicious neuron indices, $f(x_i)_s$ and $f(\widetilde{x}_i)_s$ are activation values of the suspicious neuron at index s for input x_i and $\widetilde{x}_i$, respectively. N is a set of non-suspicious neuron indices, $f(x_i)_n$ and $f(\widetilde{x}_i)_n$ are activation values of the non-suspicious neuron at index n on input x_i and $\widetilde{x}_i$, respectively. For convolutional networks, we consider looking at the most suspicious feature maps. For a feature map F which contains X number of neurons in it, Tarantula score of F can be found as stated in Equation 4.4. Hence, Tarantula score of a feature map is the average of Tarantula scores of its neurons.

$$Tarantula_F = 1/X\sum_{n\in F} Tarantula_n \tag{4.4}$$

Similar to finding the most suspicious neurons, top k suspicious feature maps can be found after sorting feature maps based on their Tarantula scores in descending order and selecting first k of them.

While it is possible to continue with just neurons for convolutional layers, choosing a suitable number for k (i.e. number of suspicious neurons to consider) is hard among huge number of neurons. Moreover, feature maps learn different parts of an image. Hence, each feature map represents certain type of features in an image. It would be possible that all suspicious neurons are came from same feature map, which would cause generation of inputs that focus only a certain type of feature. Therefore, suspiciousness based on feature maps, is more meaningful than neurons, for convolutional networks.

The definition of Equation 4.3 changes for convolutional layers as follows. For convolutional layers, S is a set of suspicious feature map indices, $f(x_i)_s$ and $f(\widetilde{x}_i)_s$ are the average activation values of neurons in the suspicious feature map at index s for input x_i and $\widetilde{x}_i$ respectively. N is a set of non-suspicious feature map indices, $f(x_i)_n$ and $f(\widetilde{x}_i)_n$ are the average activation values of neurons in the non-suspicious feature map at index n for input x_i and $\widetilde{x}_i$ respectively. The parameters λ and γ balance suspicious and non-suspicious losses, which can take different values based on dense or convolutional layers.

$\mathcal{L}_{Susp}$ subtracts suspicious activations for the generated input $\widetilde{x}_i$ from suspicious activations for the original input x_i, which allows generation of inputs such that their suspicious activations are higher than suspicious activations of their original versions. Moreover, the right part of the equation uses absolute value to keep non-suspicious activations the same. Finally, $\mathcal{L}_{MSE}$ is used to keep generated pixels similar to original ones:

$$\mathcal{L}_{MSE} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\sum_{p\in P}(x_{i,p}-\widetilde{x}_{i,p})^2}{\|P\|}\right) \tag{4.5}$$

Note that, convolutional networks with both convolutional and dense layers use both versions of $\mathcal{L}_{Susp}$ described above as well as $\mathcal{L}_{MSE}$ for loss calculation. Whereas, dense networks only use the version of $\mathcal{L}_{Susp}$ for dense layers only together with $\mathcal{L}_{MSE}$ for loss calculation. The final formulation for $\mathcal{L}_{Susp}$ is shown in Equation 4.6.

$$\mathcal{L}_{Susp} = \alpha\mathcal{L}_{Susp} + (1-\alpha)\mathcal{L}_{MSE}, \tag{4.6}$$

where α is used to regularize $\mathcal{L}_{Susp}$ and $\mathcal{L}_{MSE}$.

4.4. Adversarial Verification of Generated Samples

DGN models together with their custom loss formulations, are capable of generating adversarial samples. Although custom loss formulations force DGNs to generate adversarial samples, generated samples are not guaranteed to comply with the allowable adversarial perturbation limit. Hence, we add an extra step called *epsilon clipping* after data generation process to ensure that generated samples are adversarial. Each generated sample should satisfy the condition which is presented in Equation 4.7.

$$\|\widetilde{x} - x\| \leqslant \epsilon, \tag{4.7}$$

where $\widetilde{x}$ is generated and x is original data sample, $\|\widetilde{x} - x\|$ is L_∞ distance between original and generated samples, ϵ is allowable perturbation limit. If the condition presented in Equation 4.7 does not hold, meaning that the calculated L_∞ distance is greater than perturbation budget ϵ, the conditional formula presented in Equation 4.8 is used to adjust adversarial perturbation to a maximum valid perturbation value.

$$f(x_i, \widetilde{x}_i) = \begin{cases} -\epsilon, & \text{if } x_i - \widetilde{x}_i > 0 \\ \epsilon, & \text{if } x_i - \widetilde{x}_i < 0 \\ x_i - \widetilde{x}_i, & \text{otherwise} \end{cases} \tag{4.8}$$

We do not want to change the sign of perturbations. Hence, Equation 4.8 sets perturbation amounts to valid values without changing sign of original perturbations. For instance, if an original feature x_i is greater than a generated feature $\widetilde{x}_i$ in DGN output, the adversarial perturbation is set to -ϵ since $x_i + (-\epsilon) = x_i - \epsilon = \widetilde{x}_i$. In this

setting, $x_i = \widetilde{x}_i + \epsilon$. Hence the original feature is still greater than the generated feature and the sign of perturbation is preserved.

The validated perturbation ϵ is then added over original sample feature x_i to obtain generated sample feature $\widetilde{x}_i$.

4.5. Adversarial Defense

In addition to using DGNs for attack, DGN generated samples can be used to make original networks robust against adversarial attacks using one of the most successful defense techniques, called adversarial training. We adopt an adversarial training technique used in [12] into our work, in which adversarial samples are generated during the training phase to calculate adversarial loss which is combined with the original loss to obtain the final loss value:

$$\widetilde{J}(\theta, x, y) = \alpha J(\theta, x, y) + (1 - \alpha) J(\theta, x', y), \tag{4.9}$$

where x is an original input and x' is a generated input, y is a training label and α is the parameter that balances original and adversarial losses. The parameter α is set to 0.5 as suggested in [12]. The difference of our approach from [12] in adversarial training is that we generate adversarial samples *before adversarial training* and then use them in adversarial training together with original samples. The reason why we use this approach is that DGN models are trained once in the beginning based on the original model parameters and using the altered model weights during adversarial training would require training DGN models with changed problem-specific properties in each batch update which will be costly. Moreover, it would be redundant to use such approach for DGN (LRP) where the custom loss does not need original network parameters.

4.6. Evaluation

4.6.1. Evaluation Setup

4.6.1.1. Datasets. We conduct experiments on popular MNIST [27] and CIFAR-10 [28] datasets. The details of each datasets are as follows:

- MNIST dataset consists of 28x28 images of handwritten digits from 0-9. The dataset has 60000 training and 10000 test samples.
- CIFAR dataset contains 32x32 RGB images of 10 different classes. The dataset has 60000 training samples where each class has 6000 images. The test set size of the dataset is 10000. The classes of CIFAR is as follows; airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck.

4.6.1.2. Target Models. We used different target model architectures for MNIST and CIFAR datasets since each dataset has a different difficulty level. The details of each target model are as follows:

- MNIST baseline model is a fully connected neural network which has five fully connected layers. Each layer contains 30 neurons and final output layer consists 10 neurons since MNIST has 10 classes. In total, the model has 160 neurons and 27580 trainable parameters. Rectified Linear Unit (ReLU) is used as an activation function of neurons in the model. The test accuracy of the model is 96.56%.
- CIFAR target model is a convolutional neural network, which contains three convolutional and two dense layers. Convolutional layers use 3x3 filters with stride 1 and no padding. There are MaxPool layers with 2x2 pool size after first two convolutional layers. After convolutional layers, there is a dense layer with 64 neurons and a final output layer with 10 neurons. ReLU activations are used in convolutional and dense layers. The model has 122570 trainable parameters in total and achieves 70.75% test set accuracy.

4.6.1.3. Training Configurations. Table 4.1 shows training configurations of DGN models. All training parameters are determined experimentally to avoid training problems (e.g. underfitting, overfitting). All models use Adam optimizer and learning rate decay.

Table 4.1. Training configurations of DGNs

Dataset	**Custom Loss**	**Training Parameters**		
		Learning Rate	***Epoch Size***	***Batch Size***
MNIST	***LRP***	$1e-4$	150	100
	NCE	$1e-4$	100	100
	Suspiciousness	$1e-4$	600	100
CIFAR	***LRP***	$1e-4$	80	100
	NCE	$1e-4$	100	100
	Suspiciousness	$1e-3$	60	100

Table 4.2 shows adversarial training configurations where adversarial samples used are generated using DGN models with different custom losses. For instance, the first row of Table 4.2 shows adversarial training configurations of original model when the attack is DGN-LRP on MNIST dataset. Similar to DGN training configurations, training parameters are determined experimentally to overcome training challenges.

Table 4.3 shows training parameters of FGSM and PGD adversarial training. The first row of Table 4.3 indicates adversarial training configurations of original model when the attack is FGSM on MNIST dataset for instance. The same training parameters are applied to all PGD attacks with different step sizes and random initializations.

4.6.1.4. Experimental Design. **Problem specific parameters**. In LRP, the number of relevant pixels k=20 and k=80 is used for MNIST and CIFAR, respectively. For suspiciousness, the number of suspicious neurons is 20 for MNIST and the number of suspicious feature maps is 20 for CIFAR.

Table 4.2. DGN-adversarial training configurations

Dataset	**Custom Loss**	**Training Parameters**		
		Learning Rate	***Epoch Size***	***Batch Size***
MNIST	***LRP***	$1e-3$	10	100
	NCE	$1e-3$	10	100
	Suspiciousness	$1e-4$	50	100
CIFAR	***LRP***	$1e-4$	100	100
	NCE	$1e-4$	50	100
	Suspiciousness	$1e-4$	50	100

Custom loss parameters. For LRP, $\alpha = 0.5$ is used. For Suspiciousness loss in MNIST, where we only need suspiciousness loss for dense layers, $\alpha = 0.8$, $\lambda = 0.5$, $\gamma = 0.5$. For Suspiciousness loss in CIFAR, where both dense and convolutional suspiciousness losses are needed; $\alpha = 0.9$, $\lambda = 0.5$ and $\gamma = 0.01$ for both dense and convolutional losses. For NCE, both α and β are set to 1.

Adversarial Attack Configurations. For all attacks (e.g. FGSM, PGD, DGN models) in our experiements, we used perturbation budgets presented in Table 4.4 in L_∞ distance. While setting the perturbation budgets, we integrated the budgets used by the recent studies in this domain [17].

We further applied *epsilon clipping* to DGN generated samples to ensure that generated samples are in allowable perturbation bounds. FGSM and PGD algorithms are also capable of not exceeding the specified perturbation limit.

Experiments were run on a server with Intel i7 8700K Processor, 16 GB RAM and NVidia Geforce GTX 1080 GPU.

Table 4.3. FGSM and PGD-adversarial training configurations

Dataset	**Custom Loss**	**Training Parameters**		
		Learning Rate	***Epoch Size***	***Batch Size***
	FGSM	$1e-3$	20	100
MNIST	***PGD***	$1e-3$	20	100
	FGSM	$1e-4$	80	100
CIFAR	***PGD***	$1e-4$	50	100

Table 4.4. Adversarial attack perturbation budgets

Dataset	**Distance Metric**	**Budget**
MNIST	L_∞	0.3
CIFAR	L_∞	8/255

4.6.1.5. Research Questions. We proposed research questions which will be answered with our experiments. Here are the questions we will look answers for:

- *RQ1* (Attack Performance): How effective are DGN networks in attacking PGD and FGSM robust defense networks?
- *RQ2* (Defense Performance): How effective is adversarial training which uses DGN generated adversarial samples in the defense against FGSM and PGD attacks?
- *RQ3* (Comparison): How do DGN networks with different custom losses perform against each other?
- *RQ4* (Effectiveness): What are the most effective custom losses for attack and defense?
- *RQ5* (Cost): How do data generation times differ between DGN and other attacks?

4.6.2. Evaluation Results

Our framework uses regularized convolutional autoencoders (CAE) which we called as DGNs. Together with CAE, variational autoencoders (VAE)s and generative adversarial networks (GAN)s are generative architectures that are widely used in the literature. VAE learns undelying probability distribution of training data and generates new data based on learned distribution. We did not use VAE since our aim is to give adversarial data generation ability to autoencoders through our custom loss functions and see the effects of our custom losses for adversarial data generation. Moreover, VAEs due to their imperfect reconstruction, often generate blurry images which do not comply with allowable adversarial perturbation limits [56]. GANs are quite adept at producing adversarial data. However, they are unstable and difficult to train, which makes their usage unfavorable on complex goals such as increasing suspicious neuron activations of a neural network. Moreover, GANs generate all image pixels in one shot and hence objectives similar to LRP custom loss, is hard for GANs since they are impractical to guess value of one pixel given another pixel [57].

We investigated our approach from two perspectives, namely, using DGNs as an adversarial attack technique and using them as an adversarial defense technique. Figure 4.7 shows generated adversarial test samples by DGN networks in which they are used in attack side.

We include FGSM and PGD attacks in our experiments to generate adversarial test data. Moreover, FGSM and PGD generated adversarial training data is used in adversarial training of the original model to obtain FGSM and PGD robust defense networks. In all tables, PGD(X) refers to PGD attack with X steps and one random initialization. For example, 10PGD(20) means, 20 step PGD with 10 random restarts. The name of defense networks, comes from attacks that produce adversarial data used (together with original samples) in their adversarial training. For instance, PGD(20) robust defense network is trained using PGD(20) generated adversarial training samples and the original training data. The same style applies to DGN models. For example,

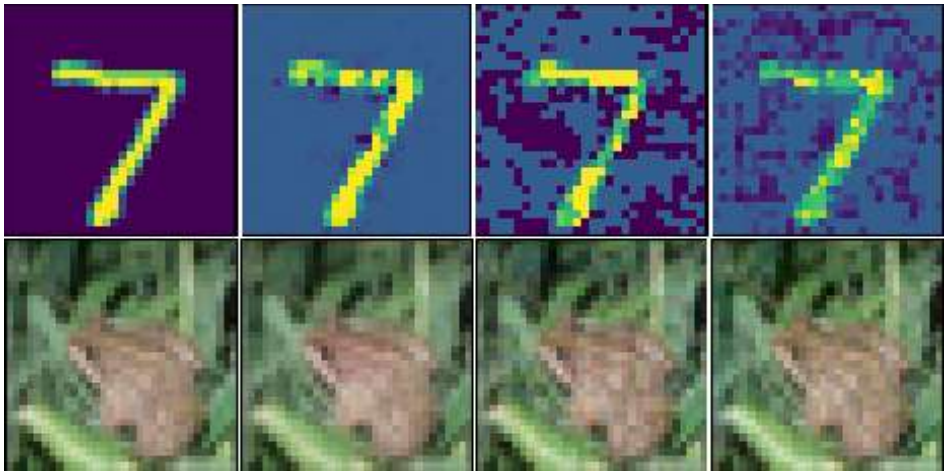

Figure 4.7. Generated images by DGN networks on MNIST (top) and CIFAR (bottom) test data. From left to right we show the original sample, DGN (LRP), DGN (NCE) and DGN (Suspiciousness) generated samples. The label of the original image is 7 in MNIST and 6 (i.e. frog) in CIFAR. In MNIST, DGN (LRP) and DGN (Suspiciousness) generated samples are predicted as 2, DGN (NCE) generated sample is predicted as 3. In CIFAR, DGN generated samples are predicted as 4 (i.e. deer).

DGN(LRP) robust network is trained with DGN(LRP) generated adversarial training samples and the original training data.

4.6.2.1. RQ1 (Attack Performance). On the attack side, we evaluated attack success of DGN generated adversarial test data, together with FGSM and PGD attacks, against FGSM and PGD robust networks. Since the contributions of PGD attacks with step sizes higher than 50 for MNIST and higher than 20 for CIFAR, are very limited to best PGD attacks we used, we did not use them in the evaluation. Tables 4.5 and 4.6 show robust test accuracies of attacks on defense networks. Hence, attacks used in these tables, generate adversarial test samples from original test dataset. On the other hand, defense robust networks are trained with adversarial training dataset which is generated by adversarial attacks using original training dataset.

Table 4.5 shows performances of various DGN, FGSM, and PGD attacks on the original, FGSM-robust and various PGD-robust networks on MNIST dataset. For the original model, DGN(NCE) outperformed the other attacks with 0.0% accuracy

Table 4.5. Attack performances on MNIST dataset.

Attack	**Defense Network Type**					
	Original	***FGSM Robust***	***PGD(10) Robust***	***PGD(20) Robust***	***10PGD(20) Robust***	***PGD(50) Robust***
DGN (NCE)	**0.0**%	93.51%	78.75%	92.10%	91.67%	95.85%
DGN (LRP)	16.12%	**73.00**%	**60.76**%	**64.56**%	**48.73**%	**76.33**%
DGN (Susp.)	51.23%	**70.19**%	**47.87**%	**59.96**%	**41.15**%	**79.25**%
FGSM	2.73%	95.70%	86.15%	89.38%	73.55%	89.24%
PGD (20)	2.31%	95.99%	95.95%	96.49%	94.77%	96.35%
10PGD (20)	**1.13**%	94.41%	91.43%	95.81%	95.34%	96.07%
PGD (50)	2.16%	91.42%	89.92%	95.27%	90.36%	96.39%

which means 100% success rate. Under all settings of FGSM and PGD robust defense networks, DGN(LRP) and DGN(Susp.) are the best two attack techniques.

Table 4.6 shows attack performances on CIFAR dataset. For the original CIFAR network, FGSM and PGD(20) attacks are better than DGNs. However, performances of FGSM and PGD(20) attacks are only slightly better than DGNs on all robust networks, where the accuracy difference is at most 3.62% (on FGSM robust network between DGN(Susp.) and PGD(20) attacks).

4.6.2.2. RQ2 (Defense Performance). We used an adapted version of adversarial training technique (described in previous section) in [12] for our experiments and compare

Table 4.6. Attack performances on CIFAR dataset.

Attack	**Defense Network Type**				
	Original	***FGSM Robust***	***PGD(5) Robust***	***PGD(10) Robust***	***PGD(20) Robust***
DGN(NCE)	64.24%	51.38%	48.03%	47.04%	50.56%
DGN(LRP)	67.69%	52.92%	49.41%	47.09%	50.02%
DGN(Susp.)	68.30%	53.85%	49.54%	48.73%	49.45%
FGSM	**17.92%**	**50.95%**	**47.08%**	**45.51%**	**48.39%**
PGD(20)	**12.99%**	**50.23%**	**46.97%**	**46.07%**	**48.41%**

FGSM and PGD robust models with DGN robust models. For all attacks, we use this adapted adversarial training technique which includes both original and adversarial samples in training process. However, it is also possible to use only adversarial samples in adversarial training. We also evaluated the approach that only trains with adversarial samples to choose the adversarial training technique that suits us best. In DGN-adversarial training, we observed that natural test accuracies of robust models are low when training with only adversarial samples. This was not the case for FGSM and PGD adversarial training. This result showed that the distribution of the samples generated by DGNs is different from the original samples, while the distribution of the samples generated by PGD and FGSM is similar to the originals. In DGN custom losses, since we look at the adversarial input generation objective from different perspectives, it is expected that the samples produced have different distributions. However, this is not an obstacle for DGNs to successfully generate adversarial data. As a result, we chose the adversarial training approach with both original and adversarial samples, where we get better results.

Table 4.7. Defense performances against attacks on MNIST dataset.

Defense Network Type	**Attacks**					
	Original	***FGSM***	***PGD (10)***	***PGD (20)***	***10PGD (20)***	***PGD (50)***
Original	**96.56**%	2.73%	4.89%	2.31%	1.13%	2.16%
FGSM Robust	95.95%	**95.70**%	96.39%	95.99%	94.41%	91.42%
PGD(10) Robust	95.74%	86.15%	**96.45**%	95.95%	91.43%	89.92%
PGD(20) Robust	**96.28**%	**89.38**%	**96.64**%	**96.49**%	**95.81**%	**95.27**%
10PGD(20) Robust	95.63%	73.55%	95.67%	94.77%	95.34%	90.36%
PGD(50) Robust	**96.28**%	89.24%	96.14%	**96.35**%	**96.07**%	**96.39**%
DGN(NCE) Robust	96.21%	15.91%	73.32%	47.69%	46.02%	28.38%
DGN(LRP) Robust	95.03%	15.67%	73.04%	41.92%	32.54%	16.02%
DGN(Susp.) Robust	95.99%	3.90%	39.10%	9.79%	6.26%	4.89%

Table 4.8. Defense performances against attacks on CIFAR dataset.

Defense Network Type	**Attacks**				
	Original	***FGSM***	***PGD (5)***	***PGD (10)***	***PGD (20)***
Original	**70.75**%	17.92%	13.87%	13.22%	12.99%
FGSM Robust	55.19%	**50.95**%	**51.01**%	**50.16**%	**50.23**%
PGD (5) Robust	51.94%	47.08%	47.99%	47.49%	46.97%
PGD (10) Robust	49.50%	45.51%	46.45%	46.25%	46.07%
PGD (20) Robust	53.14%	**48.39**%	49.39%	**48.85**%	**48.41**%
DGN(NCE) Robust	48.80%	42.42%	43.43%	42.75%	42.78%
DGN(LRP) Robust	55.26%	46.32%	47.97%	46.99%	46.48%
DGN(Susp.) Robust	**57.73**%	47.65%	**49.63**%	48.39%	48.03%

Depending on the difficulty of the dataset and the problem, different training parameters are used which are determined experimentally. Unlike the evaluation made in *RQ1*, higher robust test accuracy means a better defense performance of a defense network against a given attack. In Tables 4.7 and 4.8, robust test accuracies of defense networks are shown against adversarial attacks.

Table 4.7 shows defense performances against each attack on MNIST dataset. PGD(20) and PGD(50) robust networks showed better robustness against most of the attacks. DGN(NCE) and DGN(LRP) robust networks showed acceptable robustness

against PGD attacks that have step sizes lower than 20, while DGN(Susp.) robust network's defense performance is worse than others. As expected, defense performance of DGN-robust networks are worse than other robust networks which are adversarially trained with samples generated from attacks of their own type.

On CIFAR, DGN robust networks show comparable defense performance with FGSM and PGD robust networks against FGSM and PGD attacks, as shown in Table 4.8. While FGSM robust network outperformed other defense networks, PGD(20) and DGN (Susp.) robust networks showed a good level of robustness. What draws attention from these results is that, robust accuracies are close to each other, compared to MNIST. This is inherently caused by the high capacity that the original CIFAR network has, which reduces the adversarial effect and transferability of adversarial attacks [17].

Table 4.9. Performance of DGN models against each other

Dataset	**Defense Network Type**	**Attacks**		
		DGN (LRP)	***DGN (NCE)***	***DGN (Susp.)***
MNIST	***DGN(LRP) Robust***	98.91%	30.77%	67.33%
	DGN(NCE) Robust	17.88%	99.94%	15.80%
	DGN(Susp.) Robust	24.41%	9.13%	99.95%
CIFAR	***DGN(LRP) Robust***	64.27%	45.57%	46.86%
	DGN(NCE) Robust	45.03%	76.92%	46.74%
	DGN(Susp.) Robust	54.68%	49.64%	87.65%

4.6.2.3. RQ3 (Comparison). Table 4.9 shows the performances of DGN models with different custom losses against each other. All DGN models showed better performances against attacks of their own type (i.e. DGN attack with same custom loss) as expected. On MNIST, DGN (LRP) robust network showed better robustness against all DGN attacks. DGN (LRP) attack is successful against other DGN robust networks. The best attack result belongs to DGN (NCE) on DGN (Suspiciousness) robust network with **9.13**% robust test accuracy. On CIFAR, DGN (Suspiciousness) robust network showed better robustness against other DGN attacks. The attack performances of all DGN models against each other are very similar. The results observed in *RQ1* and *RQ2* on CIFAR (i.e. similar robust accuracies), continue in this experiment as well.

4.6.2.4. RQ4 (Effectiveness). Based on MNIST attack results in Table 4.5, *LRP* and *Suspiciousness* custom losses are better than *NCE*. However, robust accuracies are lower (i.e. better) when the custom loss is *Suspiciousness*. In Table 4.6, which shows attack results on CIFAR, it is difficult to say that one custom loss is better than another, as robust accuracies are close to each other. We can conclude that *Suspiciousness* custom loss is more preferable for software engineers to generate adversarial data. On defense side, custom loss performances are close to each other. NCE on MNIST and Suspiciousness on CIFAR, are good choices for adversarial defense.

4.6.2.5. RQ5 (Cost). We further analyzed data generation times of adversarial attacks, which is crucial for the cost of adversarial attacks and defenses. Strong PGD attacks are able to generate powerful adversarial samples that enhance attack success. Furthermore, using a stronger adversary in the mini-max formulation of robust optimization [58], decreases the success rate of transferred adversarial examples, hence provides more robust networks [17]. Although it is possible to use even stronger attacks to get better performance, the cost of adversarial sample generation increases significantly, which negatively affects the time required to provide robustness to a network. This makes the use of strong adversaries for real-time streaming data, impractical. On the other hand, after an initial training, which is not required in data generation phase,

our technique can generate samples faster than PGD attacks, with a comparable performance as explained in *RQ1* and *RQ2*. Figure 4.8 shows the time cost of adversarial data generation on MNIST and CIFAR datasets. The cost of DGN networks is lower than all attacks on each dataset. In the case of PGD, as the step size increases, we observed that the cost of data generation increases dramatically. In CIFAR, where the attack and defense performances are close, our technique is **7 - 8.9** times faster. In MNIST against PGD(20), DGN(LRP) showed better attack performance and worse defense performance, but it is **11.3** times faster. Since PGD(50) and 10PGD(20) require 422.4 and 1721.8 seconds respectively for the same task on MNIST, we did not include them in the chart. These results indicate that, in the attack success rate - time cost trade-off, our technique is better than PGD.

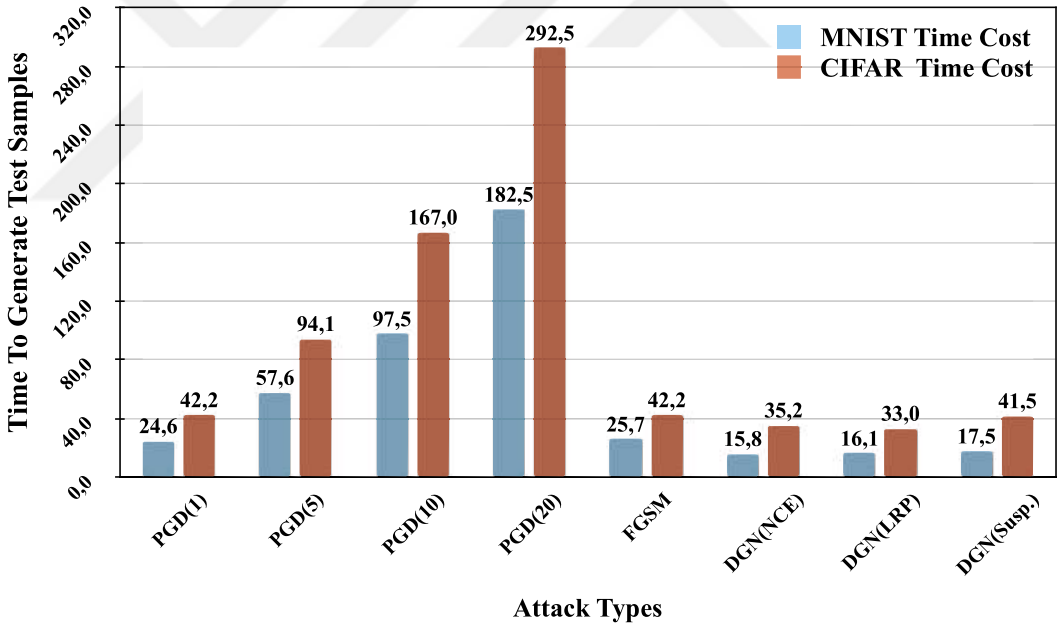

Figure 4.8. Time in seconds for attacks to generate 10K test samples on MNIST and CIFAR datasets (averaged over 10 independent runs). Besides showing comparable attack performance, DGNs are faster than FGSM and PGD attacks in data generation.

We can use our trained DGN models permanently without any additional training while gradient-based techniques have to repeat their process for each new coming data. Since there is no extra training necessity for DGN to generate data, we did not include initial training times in time-cost evaluation of Figure 4.8. Although initial training

Table 4.10. Total time costs of DGN models

Dataset	**DGN Model**	**Time Cost (in seconds)**		
		Training Time	***Data Generation Time***	***Total Time***
MNIST	***DGN (LRP)***	90.30 secs	16.1 secs	106.4 secs
	DGN (NCE)	64.08 secs	15.8 secs	79.88 secs
	DGN (Susp.)	382.44 secs	17.5 secs	399.94 secs
CIFAR	***DGN (LRP)***	110.70 secs	33.0 secs	143.7 secs
	DGN (NCE)	155.86 secs	35.2 secs	191.06 secs
	DGN (Susp.)	125.88 secs	41.5 secs	163.38 secs

Table 4.11. Total time costs of all attacks (in seconds)

Dataset	**Attacks**							
	PGD (1)	***PGD (5)***	***PGD (10)***	***PGD (20)***	***FGSM***	***DGN (NCE)***	***DGN (LRP)***	***DGN (Susp)***
MNIST	24.6 secs	57.6 secs	97.5 secs	182.5 secs	25.7 secs	79.88 secs	106.4 secs	399.94 secs
CIFAR	42.2 secs	94.1 secs	167.0 secs	292.5 secs	42.2 secs	191.06 secs	143.7 secs	163.38 secs

time of DGN models is not an important fact for the cost data generation process, we present initial training times and hence total time cost of DGN models in Table 4.10. Training times of DGN models presented in Table 4.10 is the average training time of

k DGN models. Hence, DGN models are trained simultaneously for this experiment. *DGN(Susp.)* has the longest training time since we trained it for 600 epochs.

We further compare total time costs of DGN, FGSM and PGD attacks in Table 4.11. On MNIST dataset, where *DGN(LRP)* outperformed FGSM and all PGD attacks as presented in Table 4.5, total time cost of *DGN(LRP)* is still lower than *PGD(20)* and very close to *PGD(10)*. On CIFAR dataset, where attack success levels of all attacks (i.e. FGSM, PGD, DGN) are very close to eachother, total time cost of *DGN(LRP)* and *DGN(Susp.)* are still better than *PGD(10)* and *PGD(20)* attacks. Also, *DGN(NCE)* has better total time cost than *PGD(20)*.

5. CONCLUSIONS

The existence of adversarial samples is a huge threat for safety and robustness of deep neural networks. It is essential that deep learning systems used in safety-critical applications are safe and robust against potential faults. Some unfortunate scenarios we have seen recently related to autonomous vehicle accidents [7] and financial losses [8] show the importance of robustness improvement methods for DNNs.

Adversarial training [12] is an effective way of providing robustness to DNNs using adversarial samples generated by various attacks. Current work [17] provides state-of-the-art attack performance but suffer from flexibility in data generation and high generation costs. In this work, we present an adversarial data generation framework that consists of convolutional autoencoders with custom loss functions. The proposed approach, enables flexibility on shaping data generation thanks to custom losses. Our technique is general enough to incorporate various custom losses. Specifically, we studied three custom losses LRP, Suspiciousness, and NCE. Besides its customizable nature, our approach differs from others by exploiting the explainability of neural networks to generate adversarial data using LRP and Suspiciousness techniques.

Our evaluations demonstrate that the transferable DGN attacks showed comparable performances on PGD robust networks, while the defense performances of DGN robust networks are acceptable against PGD attacks. Moreover, we showed that DGN attacks are faster than PGD in adversarial data generation.

In future, we will apply our technique on different target model architectures and datasets with different custom loss functions. Moreover, we will enhance our custom losses by using them together as a single loss function. Furthermore, we plan to use DGN models in a reverse order such that input of DGN will be an adversarial and output of DGN will be an unperturbed sample. This approach can be an effective defense mechanism as well.

REFERENCES

1. Wang, F., M. Jiang, C. Qian, S. Yang, C. Li, H. Zhang, X. Wang and X. Tang, "Residual Attention Network for Image Classification", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

2. Hinton, G., L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *IEEE Signal Processing Magazine*, Vol. 29, No. 6, pp. 82–97, 2012.

3. Sutskever, I., O. Vinyals and Q. V. Le, "Sequence to sequence learning with neural networks", *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

4. Julian, K. D., J. Lopez, J. S. Brush, M. P. Owen and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems", *IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pp. 1–10, 2016.

5. Al-Qizwini, M., I. Barjasteh, H. Al-Qassab and H. Radha, "Deep learning algorithm for autonomous driving using GoogLeNet", *IEEE Intelligent Vehicles Symposium (IV)*, 2017.

6. Hu, Z., J. Tang, Z. Wang, K. Zhang, L. Zhang and Q. Sun, "Deep learning for image-based cancer detection and diagnosis- a survey", *Pattern Recognition*, Vol. 83, pp. 134–149, 2018.

7. Lebeau, P., *Google's self-driving car caused an accident, so what now?*, 2016 (accessed Dec 3, 2020), `https://www.wired.com/2016/02/googles-self-driving-car-may- caused-first- crash`.

8. Varshney, K. R., "Engineering safety in machine learning", *Information Theory and Applications Workshop (ITA)*, pp. 1–5, 2016.

9. Biggio, B., I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto and F. Roli, "Evasion attacks against machine learning at test time", *Joint European conference on machine learning and knowledge discovery in databases*, pp. 387–402, Springer, 2013.

10. Szegedy, C., W. Zaremba, I. Sutskever, J. B. Estrach, D. Erhan, I. Goodfellow and R. Fergus, "Intriguing properties of neural networks", *International Conference on Learning Representations*, 2014.

11. Nguyen, A., J. Yosinski and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

12. Goodfellow, I., J. Shlens and C. Szegedy, "Explaining and Harnessing Adversarial Examples", *International Conference on Learning Representations*, 2015.

13. Kurakin, A., I. Goodfellow and S. Bengio, "Adversarial examples in the physical world", *CoRR, abs/1607.02533*, 2016.

14. Papernot, N., P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik and A. Swami, "The limitations of deep learning in adversarial settings", *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 372–387, 2016.

15. Moosavi-Dezfooli, S.-M., A. Fawzi and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2574–2582, 2016.

16. Carlini, N. and D. Wagner, "Towards evaluating the robustness of neural networks", *IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 39–57, 2017.

17. Madry, A., A. Makelov, L. Schmidt, D. Tsipras and A. Vladu, "Towards Deep Learning Models Resistant to Adversarial Attacks", *International Conference on Learning Representations*, 2018.

18. Liu, X., M. Cheng, H. Zhang and C.-J. Hsieh, "Towards robust neural networks via random self-ensemble", *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 369–385, 2018.

19. Guo, C., M. Rana, M. Cisse and L. Van Der Maaten, "Countering adversarial images using input transformations", *International Conference on Learning Representations*, 2018.

20. Meng, D. and H. Chen, "Magnet: a two-pronged defense against adversarial examples", *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 135–147, 2017.

21. Metzen, J. H., T. Genewein, V. Fischer and B. Bischoff, "On detecting adversarial perturbations", *International Conference on Learning Representations*, 2017.

22. Liao, F., M. Liang, Y. Dong, T. Pang, X. Hu and J. Zhu, "Defense against adversarial attacks using high-level representation guided denoiser", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1778–1787, 2018.

23. Samangouei, P., M. Kabkab and R. Chellappa, "Defense-gan: Protecting classifiers against adversarial attacks using generative models", *International Conference on Learning Representations*, 2018.

24. Buckman, J., A. Roy, C. Raffel and I. Goodfellow, "Thermometer encoding: One hot way to resist adversarial examples", *International Conference on Learning Representations*, 2018.

25. Song, Y., T. Kim, S. Nowozin, S. Ermon and N. Kushman, "Pixeldefend: Lever-

aging generative models to understand and defend against adversarial examples", *International Conference on Learning Representations*, 2018.

26. Eniser, H. F., S. Gerasimou and A. Sen, "Deepfault: Fault localization for deep neural networks", *International Conference on Fundamental Approaches to Software Engineering*, pp. 171–191, Springer, 2019.

27. LeCun, Y., *The mnist database of handwritten digits*, 1998 (accessed Dec 3, 2020), `http://yann.lecun.com/exdb/mnist/`.

28. Krizhevsky, A. and G. Hinton, *Learning multiple layers of features from tiny images*, 2009 (accessed Dec 3, 2020), `https://www.cs.toronto.edu/kriz/cifar.html`.

29. Pei, K., Y. Cao, J. Yang and S. Jana, "DeepXplore: Automated Whitebox Testing of Deep Learning Systems", *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18, 2017.

30. Ma, L., F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems", *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 120–131, 2018.

31. Tian, Y., K. Pei, S. Jana and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars", *Proceedings of the 40th International Conference on Software Engineering*, pp. 303–314, 2018.

32. Goodfellow, I. and N. Papernot, *The challenge of verification and testing of machine learning*, 2017 (accessed Dec 3, 2020), `http://www.cleverhans.io/security/privacy/ml/2017/06/14/verification.html`.

33. Papernot, N., P. McDaniel, X. Wu, S. Jha and A. Swami, "Distillation as a defense

to adversarial perturbations against deep neural networks", *IEEE Symposium on Security and Privacy (SP)*, 2016.

34. Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio, "Generative adversarial nets", *Advances in Neural Information Processing Systems*, 2014.

35. Kerlirzin, P. and F. Vallet, "Robustness in multilayer perceptrons", *Neural computation*, Vol. 5, No. 3, pp. 473–482, 1993.

36. LeCun, Y., J. Denker and S. Solla, "Optimal brain damage", *Advances in Neural Information Processing Systems*, Vol. 2, pp. 598–605, 1989.

37. Guo, J., Y. Jiang, Y. Zhao, Q. Chen and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems", *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 739–743, 2018.

38. Ma, L., F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems", *IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 100–111, 2018.

39. Katz, G., C. Barrett, D. L. Dill, K. Julian and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks", *International Conference on Computer Aided Verification*, pp. 97–117, Springer, 2017.

40. Ko, C.-Y., Z. Lyu, T.-W. Weng, L. Daniel, N. Wong and D. Lin, "POPQORN: Quantifying robustness of recurrent neural networks", *International Conference on Machine Learning*, 2019.

41. Goodfellow, I., Y. Bengio and A. Courville, *Deep Learning*, MIT press, 2016.

42. Chen, M., X. Shi, Y. Zhang, D. Wu and M. Guizani, "Deep features learning for medical image analysis with convolutional autoencoder neural network", *IEEE Transactions on Big Data*, 2017.

43. Kingma, D. P. and M. Welling, "Auto-encoding variational bayes", *International Conference on Learning Representations*, 2014.

44. Rezende, D. J., S. Mohamed and D. Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models", *International Conference on Machine Learning*, 2014.

45. Wan, Z., Y. Zhang and H. He, "Variational autoencoder based synthetic data generation for imbalanced learning", *IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–7, 2017.

46. Kusner, M. J., B. Paige and J. M. Hernández-Lobato, "Grammar Variational Autoencoder", *International Conference on Machine Learning*, 2017.

47. Makhzani, A., J. Shlens, N. Jaitly, I. Goodfellow and B. Frey, "Adversarial autoencoders", *International Conference on Learning Representations*, 2016.

48. Zhao, J., Y. Kim, K. Zhang, A. Rush and Y. LeCun, "Adversarially Regularized Autoencoders", *International Conference on Machine Learning*, 2018.

49. Tu, C.-C., P. Ting, P.-Y. Chen, S. Liu, H. Zhang, J. Yi, C.-J. Hsieh and S.-M. Cheng, "Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks", *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33, pp. 742–749, 2019.

50. Dong, G., G. Liao, H. Liu and G. Kuang, "A review of the autoencoder and its variants: A comparative perspective from target recognition in synthetic-aperture radar images", *IEEE Geoscience and Remote Sensing Magazine*, Vol. 6, No. 3, pp. 44–68, 2018.

51. Wong, W. E., R. Gao, Y. Li, R. Abreu and F. Wotawa, "A survey on software fault localization", *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

52. Jones, J. A. and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique", *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273–282, 2005.

53. Ochiai, A., "Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions", *Bulletin of Japanese Society of Scientific Fisheries*, Vol. 22, pp. 526–530, 1957.

54. Wong, W. E., V. Debroy, R. Gao and Y. Li, "The DStar method for effective software fault localization", *IEEE Transactions on Reliability*, Vol. 63, No. 1, pp. 290–308, 2013.

55. Montavon, G., S. Lapuschkin, A. Binder, W. Samek and K.-R. Müller, "Explaining nonlinear classification decisions with deep taylor decomposition", *Pattern Recognition*, Vol. 65, pp. 211–222, 2017.

56. Bengio, Y., *What are the pros and cons of Generative Adversarial Networks vs Variational Autoencoders?*, 2017 (accessed Jan 16, 2021), `https://www.quora.com/What-are-the-pros-and-cons-of-Generative-Adversarial-Networks-vs-Variational-Autoencoders`.

57. Goodfellow, I., *What are the pros and cons of using generative adversarial networks (a type of neural network)? Could they be applied to things like audio waveform via RNN? Why or why not?*, 2016 (accessed Jan 16, 2021), `https://www.quora.com/What-are-the-pros-and-cons-of-using-generative-adversarial-networks-a-type-of-neural-network-Could-they-be-applied-to-things-like-audio-waveform-via-RNN-Why-or-why-not`.

58. Lyu, C., K. Huang and H.-N. Liang, "A unified gradient regularization family for adversarial examples", *IEEE International Conference on Data Mining*, pp. 301–309, 2015.