

The Dimension Effect on Adversarial Learning Phenomena

Amit Leibovitz

A thesis submitted for partial fulfillment for
Master in Computer Science

Supervisor: Dr. Adi Shraibman

School of Computer Science
The Academic College of Tel Aviv-Yaffo
Tel-Aviv, Israel

April, 2020

Copyright 2020
AMIT LEBOVITZ
All Rights Reserved

Abstract

Machine-based systems, especially those based on deep neural networks, have become an integral part of our everyday life. The performance in many fields, including translations, voice recognition, spam detection, and image processing, has surpassed that of humans. Their impact will even increase soon with the release of various autonomous systems, led by the autonomous vehicle.

With the improved capabilities we get by using neural networks come new vulnerabilities as well. An industry of attacks on these networks called *adversarial learning* has been developed. In these attacks, the input data is altered slightly, in a way a human eye or ear cannot detect, causing the networks to change their behavior dramatically. Several methods have been presented over the years to deal with this phenomenon, some diagnoses the sensitivity of the network to minor changes, some add the perturbed examples to the training stage (this is called *adversarial training*). But, as far as we know, no generic defense system has been found for all types of attacks and domains.

In this paper, we examine the vulnerability to adversarial attacks in one of the core parameters - the multiplicity of the input dimension. We explain why the high dimension of the input domain increases the vulnerability to an adversarial attack and we examine the effect of several reduction-based protection methods on diverse datasets, including those with multiple color channels (CIFAR10), and high dimension (INTEL). Our conclusion is that dimension reduction can improve the resilience of a network against adversarial attacks. But, it is difficult to find one method which fits all cases. Therefore our recommendation is an ensemble approach that combines several methods in order to obtain an optimal result.

Content

1.	INTRODUCTION	1
1.1.	CONTRIBUTIONS.....	2
1.2.	PAPER OUTLINE	3
2.	BACKGROUND	4
2.1.	THE CLASSIFICATION PROBLEM	4
2.2.	DEEP NEURAL NETWORK	5
2.3.	CONVOLUTIONAL NEURAL NETWORK.....	6
3.	PROBLEM DESCRIPTION - ADVERSARIAL LEARNING	10
3.1.	DEFINITIONS OF TERMS	11
3.2.	ADVERSARIAL CLASSIFICATION ATTACKS	12
3.2.1.	<i>First Discovery - Box constrained L-BFGS.....</i>	<i>12</i>
3.2.2.	<i>Linear and Gradient-based - Fast Gradient Sign Method (FGSM).....</i>	<i>13</i>
3.2.2.1.	One-step target class method.....	14
3.2.2.2.	Basic Iterative Method (BIM).....	14
3.2.2.3.	Iterative Least-likely Class Method (ILCM).....	14
3.2.3.	ℓ_0 norm based - JSMA and One Pixel Attack.....	14
3.2.4.	Universal based Attack.....	15
3.2.5.	Neural Network-based - Adversarial Transformation Network (ATN).....	15
3.3.	ADVERSARIAL DEFENSE STRATEGIES.....	16
3.3.1.	Training Modification - Adversarial Learning	16
3.3.2.	Data Modification	16
3.3.3.	Network Modification	17
3.3.4.	External Models	18
4.	THE DIMENSION EFFECT	19
4.1.	LINEARITY HYPOTHESIS	19
4.2.	THE CURSE OF DIMENSIONALITY.....	19
4.3.	THE ADVERSARIAL CURSE OF DIMENSIONALITY FOR LINEAR MODELS	19
5.	DIMENSIONALITY REDUCTION DEFENCE.....	21
5.1.	DIMENSIONALITY DEFENCE METHODS.....	21
5.1.1.	<i>Image resize and rescale</i>	<i>21</i>
5.1.2.	<i>K-means color quantization.....</i>	<i>22</i>
5.1.3.	<i>PCA.....</i>	<i>23</i>
5.1.4.	<i>Filtering</i>	<i>25</i>
5.1.4.1.	Low pass filter.....	26
5.1.4.2.	Gaussian filter.....	27
5.1.4.3.	Gradient	28
5.1.4.4.	Bilateral filter.....	31
5.1.5.	<i>Edge detection</i>	<i>31</i>
5.2.	ENSEMBLE DEFENCE METHODS.....	33
5.2.1.	<i>What is ensemble.....</i>	<i>33</i>
5.2.2.	<i>Voting.....</i>	<i>34</i>
5.2.3.	<i>Stacking with meta-learner</i>	<i>34</i>

5.3.	EXPERIMENT SETUP.....	35
5.3.1.	<i>Metrics</i>	35
5.3.2.	<i>Datasets</i>	36
5.3.3.	<i>Network Architecture</i>	37
5.3.4.	<i>Adversarial Examples</i>	38
5.3.5.	<i>Experimental Stages</i>	39
6.	RESULTS.....	47
6.1.	DIMENSIONALITY REDUCTION RESULTS	47
6.1.1.	<i>Image Resize and Rescale</i>	48
6.1.2.	<i>K-means Color Quantization</i>	49
6.1.3.	<i>PCA</i>	50
6.1.4.	<i>Low Pass Filtering</i>	51
6.1.5.	<i>Gaussian Filtering</i>	52
6.1.6.	<i>Median Filtering</i>	53
6.1.7.	<i>Gradient Filtering</i>	54
6.1.8.	<i>Bilateral Filtering</i>	55
6.1.9.	<i>Canny Edge Filtering</i>	56
6.1.10.	<i>Summary</i>	57
6.2.	ENSEMBLE RESULTS.....	59
6.2.1.	<i>Motivation</i>	59
6.2.2.	<i>Voting and Stacking Results</i>	61
7.	CONCLUDING REMARKS AND FUTURE WORK.....	63
8.	REFERENCES	64

1. Introduction

In recent years we witness increased use of products and technologies based on machine learning and artificial intelligence. This phenomenon covers almost every aspect of our daily lives: web search, online translation, phone’s voice personal assistant and fingerprint locking, target advertising and much more. In the coming years, we expect even more advanced technologies that until recently were considered fiction rather than science. This includes autonomous vehicle and IoT products, which will make a significant change in our lives.

Like the revolution of the personal computer, which became common in every home during the 1980s and the 1990s, the endless race to upgrade and progress has sometimes resulted in security neglect. Machine learning algorithms have always been tested for how much the model was right: accuracy, FP/FN, precision/recall and so on. There has never been a criterion for examining the durability of a model against intentional malformed input. Just imagine the impact of a cyber-attack adding minor changes to the sensors of the autonomous vehicle, causing it to be confused between a stopover sign and a highway sign.

In this research, we explore a new field that emerged only in 2014 and focuses on methods of deliberate deception of learning models. This field is named *Adversarial Learning*. In their article, Szegedy et al. [1] were the first to discover a disturbing phenomenon in neural networks, which until then were considered to be incredibly accurate for many tasks. It turned out that small perturbations to an image, such that in many cases may not be distinguished by the human eye, can cause the network’s classification to change dramatically, and with high confidence.

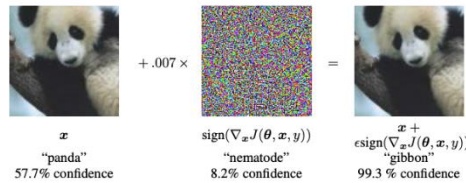


Figure 1: An adversarial image generated by the FGSM algorithm. Left - the original panda image; Middle - small perturbation added to the image; Right - an adversarial image, classified as a gibbon (Source: Explaining and harnessing adversarial examples, Goodfellow et al [16])

Figure 1 illustrates the problematic nature of this phenomenon. In this case, a small perturbation was added to the original panda image. This small noise was calculated using the Fast Gradient Sign Method suggested by [16]. Although the change is negligible and we can clearly see that the output is a panda image, the neural network identified it as gibbon with high confidence.

The FGSM was the first algorithm to propose a scheme to produce adversarial images. It uses the sign elements of the cost function gradient with respect to the input image to calculate a small noise vector to be added. The magnitude of the change can be controlled by multiplying the noise vector with the epsilon parameter. A large value of epsilon increases the likelihood of being mistaken on the network, but, of course, also detract from image quality and make it easier to recognize that manipulation has been performed. Figure 2 shows the effect of different epsilon values on the test accuracy in the case of CIFAR10 and INTEL datasets.

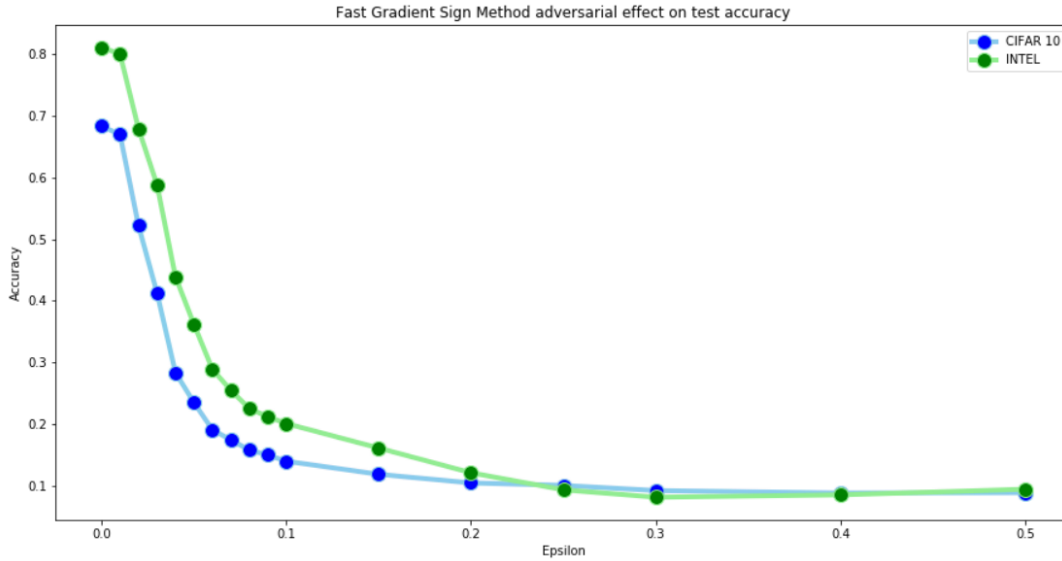


Figure 2: FGSM algorithm for creating adversarial images. This figure shows how the epsilon parameter, controlling the magnitude of the noise vector, affects the accuracy achieved by the neural network model in two different datasets.

We focused on the effect of the input dimension on the attacker’s ability to mislead the network, examining several different dimensionality reduction approaches on the classification accuracy in the case of both standards and manipulated input data.

1.1. Contributions

We studied a large number of dimensionality reduction defense strategies to combat adversarial machine learning attacks: PCA, KMeans, image rescaling, LPF, edge detection and other types of filtering including Gaussian, median, gradient and bilateral. Most of the prior work that took an approach of dimensionality reduction [30,31,32,33,35,36,40] deal with certain methods and hence the difficulty in making a qualitative comparison. Here we tried to perform an overview of as many methods as possible on the same architecture and data set.

For each of these methods, we show a significant accuracy improvement compared to the original model. The models for which we got the best results were KMeans and PCA. KMeans with $k = 20$ for example shows an improvement from 25.4% accuracy to 60.5% when $\epsilon = 0.07$ on the INTEL dataset while PCA with variance percent preserve equals 99.9% got an accuracy of 53.9%. CIFAR-10 dataset showed the same trend where KMeans and PCA presented the best improvement on small epsilon values where KMeans with $k = 20$ improved the accuracy of 28.3% in the original model to 58% when $\epsilon = 0.04$. We witnessed a dramatic improvements in classification accuracy for most of the dimensionality reduction methods, mostly in the low range of epsilon values. Note that small values for epsilon are the more likely scenario for an attack, considering the attacker’s desire not to be detected.

Our work as well as other recent work suggest that there is no magic solution that is suitable for all problems or all network architectures and data types. Therefore we suggest three novel defense ensemble strategies for adversarial learning defense. Given a large number of models that have been trained on data that has been downgraded with different processes of dimensionality reduction, if by a variety of methods or by using a different set of parameters, we would like to train a meta-learner that will take the advantage of an ensemble approach to make a more accurate result based on the original results.

We suggest to use three main methods. The first method is a simple voting among all the results for decision-making. A more sophisticated method is to use a stacking approach in which we give a new

meta-learner the dimension reduction predictions to perform training and testing. One option we tried is to give the model the predicted classes and perform one-hot encoding. Another is to be trained on all the probability vectors of all the models combined. Every model produced a distribution vector on every prediction that can be an indication of the confidence level of the model.

For small epsilon value in the range of $[0,0.07]$ these 3 methods got a higher score than the best dimension reduction method at each point. For example, on the CIFAR-10 dataset and adversarial rate of $\epsilon = 0.03$ the original model got an accuracy of 41.2% while the best dimension reduction method got a score of 58.5%. We were able to achieve an accuracy of 63% on both stacking approaches.

We consider this a general approach for solving the problem of adversarial learning. This approach can be extended at any time by further models of different dimensionality reduction methods. Proper training of the meta-learner can take the best of all models to give the optimum result. To the best of our knowledge, there was no previous attempt of dealing with the adversarial learning problem with a stacking ensemble approach.

1.2. Paper Outline

The outline of this writeup is as follow. We first briefly provide the background required in chapter 2. We describe the problem of classification and then focus on neural networks and different relevant architectures.

In Chapter 3 we elaborate on the problem domain. A full description of the adversarial learning phenomenon is given as well as several common attack methods from recent years. This chapter ends with a presentation of several common approaches for both detection and prevention.

Chapter 4 will focus on the properties of multi-dimensional input. First, we describe a general phenomenon that makes it difficult for machine learning models to solve high-dimensional problems and has received the ominous term "the curse of dimensionality". Then we will see how high dimensionality is much more problematic when considering deliberate attacks on the model.

Chapter 5 describes several dimension reduction methods that have been tested. We describe each of the methods in detail, define the experimental conditions and the measured metrics.

Chapter 6 presents the results concentrate on the conclusions that follow.

Chapter 7 summerises the contribution of this study as well as options for future research.

2. Background

Machine Learning is the science of getting computers to act without being explicitly programmed. In the past decade, machine learning has given us self-driving cars, practical speech and image recognition, effective web search, and a vastly improved understanding of the human genome.

Machine Learning was first mentioned in 1959 by Arthur Samuel [2] while trying to solve the game of checkers. As an interdisciplinary field, ML shares common properties with other fields such as Artificial Intelligence, statistics, information theory, game theory, and optimization. Evolved from data mining and pattern recognition, ML explores the study of algorithms that can learn from examples and make predictions and decisions based on a given data [3].

The ability to 'learn' is defined by progressively improve performance on a specific task. According to Tom M. Mitchell [4] 'A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E'.

Machine learning is divided into three main different classes: supervised learning, unsupervised learning and reinforcement learning. In supervised learning, the learning process is guided. That is, the algorithm trains the model with a labeled set of samples (the data), each of them is labeled by a 'teacher'. Unsupervised learning is a subfield in ML that helps find previously unknown patterns in a dataset without pre-existing labels. This includes tasks such as clustering, dimension reduction, and outlier detection. In the more sophisticated problem of reinforcement learning, the user gives the algorithm feedback on its decisions, in order to improve performance over time.

2.1. The Classification Problem

We focus on supervised learning, and more specifically classification problems. In the basic statistical supervised learning model, the model has a set of N labeled points which will be our *training set*.

Meaning, the training process includes N input vectors $\{x^i\}_{i=1}^N, x^i \in \mathbb{R}^n$ and N class labels

$\{y^i\}_{i=1}^N, y^i \in \{0,1\}$ for the discrete *classification* problem and $\{y^i\}_{i=1}^N, y^i \in \mathbb{R}$ for the continuous *regression* problem. The training model will give a prediction for every new data point. Testing the model with a new labeled dataset, called the *test set*, can give us a measure of how good our model is. If we define our hypothesis output for new data point x as $h(x)$ and the real known output as $f(x)$, then the accuracy of our model can be defined as

$$(1) L(\Theta) = \sum_{i=1}^n l(f(x_i), h(x_i, \Theta))$$

where Θ is the learning parameters of the model and l is a loss function measuring the error of the prediction with respect to the real value. Some examples of such loss functions can be the square loss function and the logistic loss function. The total L function is referred to as the *Training Loss Function*.

In order to avoid overfitting and keeping the model as simple as possible, we also define the regularization function $\Omega(h(\Theta))$ which measures the complexity of the model.

So, the objective function we wish to minimize in this optimization problem should be

$$(2) Obj(\Theta) = L(\Theta) + \Omega(h(\Theta))$$

The machine learning process illustrated in Figure 3 shows the main stages during a supervised learning experiment [5]. The labeled data is divided into *training and test sets*. After *Exploratory Data Analysis*,

which includes cleaning, transforming and visualization of the input data, a feature extraction stage gives us input X. This input will be inserted into a machine learning model resulted in a predicted value. The model includes some learned parameters. The actual value and the metric value can be used to define the accuracy of our model.

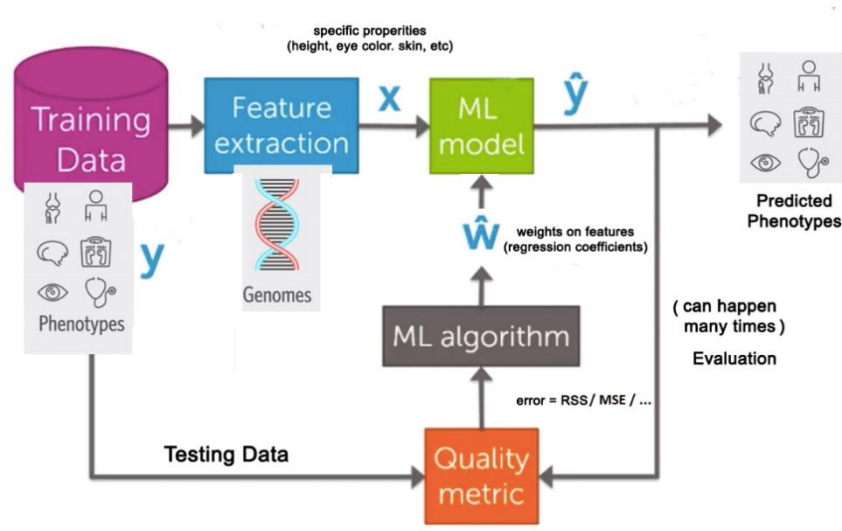


Figure 3: The Supervised Learning Process
(Source: Coursera, Machine Learning Foundations: A Case Study Approach [5])

2.2. Deep Neural Network

In recent years we witness large domains where the ML technologies based on deep learning lead to breakthroughs that was not seen before. These domains include tasks that not so long ago seemed very difficult, such as voice recognition (like Google assistance, Siri and Alexa [6]), natural language tasks (speech recognition[7][8][9] and translation [10][11], auto complete [12], machine summarization [13]) and real-time computer vision classifiers (the autonomous vehicle for example [14]).

F. Rosenblatt [17] introduced the perceptron in 1957 as an algorithm for supervised learning of binary classifiers. As shown in figure 4, all inputs $\{x^i\}_{i=1}^N$ are multiplied with their weights $\{w^i\}_{i=1}^N$ and then summed up.

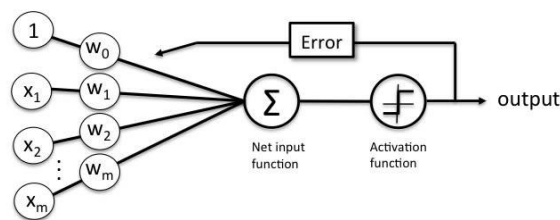


Figure 4: The perceptron (source: mlxtend)

The next stage is to apply a nonlinear activation function $\sigma(x)$. In more general cases, the sigmoid or the RELU function will be used.

$$(3) \sigma_1(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma_2(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

Figure 5: The Sigmoid and RELU Activation Function

The artificial neural network uses the repeated application of the perceptron model called neurons. The network applies layers of neurons. In each layer, every neuron outputs a signal real number, which is passed to every neuron in the next layer. At the next layer, each neuron forms its own weighted combination of these values, adds its own bias, and applies the activation function. The real numbers produced by the neurons on one layer are collected into a vector a , and then the vector of the outputs from the next layer has the form

$$(4) \sigma(Wa + b)$$

where W is the weights matrix and b is the bias vector. The number of columns in W will be the number of neurons at the previous layer that made the vector a . The number of rows in W and the size of vector b will be the number of neurons at the current layer.

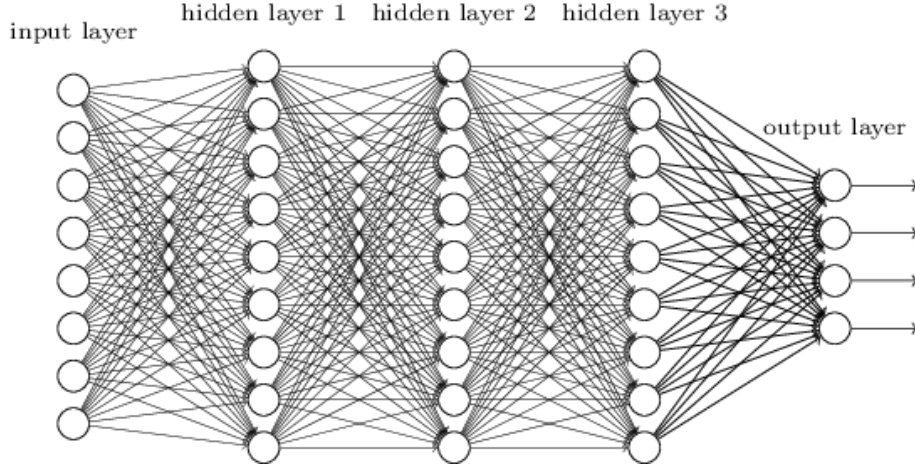


Figure 6: Full-Connected Neural Network

We can now introduce the general form of an artificial neural network called the Fully Connected Neural Network. Assume our network has L layers, where layer 1 is the input layer, L is the output layer, and layer l contains n_l neurons. Notice that our network will map from \mathbb{R}^{n_1} space to \mathbb{R}^{n_L} space. We use $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$ to note the weights at layer l and $b^{[l]} \in \mathbb{R}^{n_l}$ for the vector of biases in layer l .

We will assign an input data point x to the first layer as:

$$(5) a^{[1]} = x \in \mathbb{R}^{n_1}$$

The recursive flow of the network will be as follow

$$(6) a^{[l]} = \sigma(W^{[l]}a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l}, \quad \text{for } l = 2, 3, \dots, L$$

where $y = a^{[L]}$ is the output of the model.

2.3. Convolutional Neural Network

Another form of a common ANN architecture is the Convolutional Neural Network (known as CNN). These networks were used constantly to obtain some remarkable results in object recognition for the ImageNet challenge (see [15] and Figure 7).

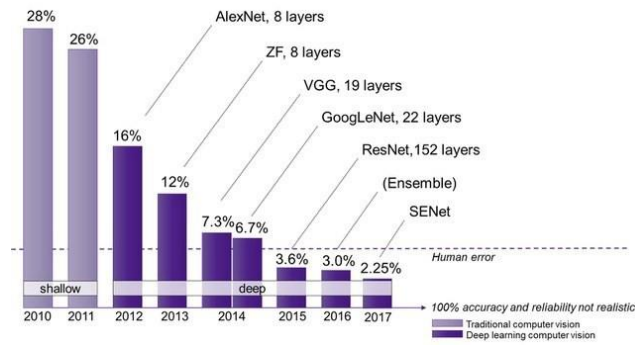


Figure 7: Results of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) over the years

Instead of connecting each neuron to all other neurons as it was defined in the Fully Connected Neural Network architecture, we will try to capture more local phenomena by connecting it to its neighbors only. In the image domain where our input neurons are pixels, we will connect each one of them to pixels in its local area. As a result, CNNs have much fewer connections and learning parameters and so they are easier to train, while their theoretical performance is likely to be only slightly worse.

A convolutional neural network has four main building blocks:

- Convolution layer
- Activation (ReLU)
- Pooling layer
- Fully Connected layer

Convolutional is a linear operation for feature extraction where an element-wise product is applied between a small window of the input and a small array of numbers vector called a kernel. The output is called a feature map. This procedure is repeated multiple times in different locations and neighborhoods (Figure 8).

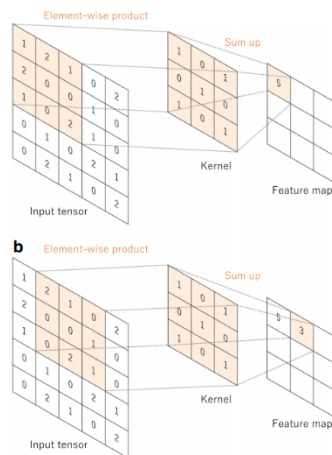


Figure 8: Convolutional Layer (Source: Convolutional neural networks: an overview and application in radiology [18])

Different kernels may be applied to capture different characteristics of the input image. Two main hyperparameters that define the convolution operation are the size of the window and the number of kernels (meaning, the numbers of neurons in the convolutional layer). Typical size for the kernel will be a 3×3 , 5×5 or 7×7 window. Other parameters may be the *Strides* and *Padding*. Strides are the number of pixels shifts of the window when going over the input matrix. A typical value will be 1 or 2. Padding is used when the filter does not fit the input image. We can then decide to drop this part or pad the pictures with zeroes.



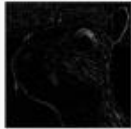




Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Figure 9: Common convolutional kernels and their output

The outputs of the linear operation of convolutional are then passed through a non-linear activation function such as sigmoid, hyperbolic, ReLu or tangent (ReLu and tangent are common choices). We are applying these activation function on the feature maps to increase the non-linearity of our model, as the image themselves are non-linear.

The pooling layer is a subsampling of the previous layer. It reduces the spatial size of the input, and therefore the number of computation and learnable parameters. The pooling layer selects a pooling operation to apply on each feature map separately and creates a new set with the same number of features maps. Typical types of pooling operators are *average pooling* and *max pooling*. There are non-learnable parameters for the pooling layer as filter size, stride and padding are predefined.

The last phase is the classification where the feature learning layers output is usually flattened to a one-dimensional vector and then passed through several fully connected layers, also called dense layers. The activation layer for this phase is usually different from other activation function, where the softmax or sigmoid are the proper choices, and the numbers of output neurons are usually as the number of classes where each output represent the probability for the matching class.

The step where the input data is transformed into output through these layers is called forward propagation. A typical schema for CNN architecture using these building blocks are shown in Figure 10.

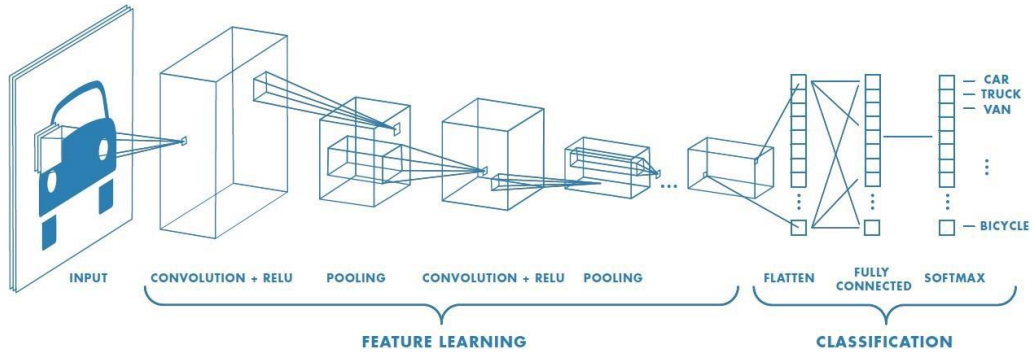


Figure 10: A typical CNN architecture consists of several repetitions of a stack of convolutional layers and a pooling layer followed by one or more fully connected layers.

3. Problem Description - Adversarial Learning

Adversarial examples are deliberately manipulated inputs created with the purpose of confusing a neural network and decreasing its performance. Adversarial examples are similar to regular example, differing only by a small perturbation. Even though these perturbations are usually indistinguishable to the human eye, they cause the network to misclassify with high confidence.

In 2014, a group of researchers from Google and NYU were the first to discover this intriguing weakness of deep neural networks in the image classification domain [1]. In their paper, the researchers found some disturbing properties among neural networks that one of them was that the smoothness assumption that is typically valid for computer vision problems and kernel method does not hold. One would expect that given a small radius $\epsilon > 0$ and an input image x , a $x + r$ satisfying $\|r\| < \epsilon$ will be classified with high probability to the same class as x . But, Szegedy et al. [1] showed that a simple optimization process breaks this assumption. It turns out that deliberate changes to some specific image could cause the network to identify it as another targeted class and with high probability. To make matter even worse it was found that the adversarial examples were relatively robust and shared by different models. Meaning, the same adversarial images were found to be hard for a family of networks that were trained with different hyperparameters or set of examples.

This sparked a new field, the adversarial field, with intensive research. Many and varied methods have been suggested since then to deceive neural networks. In 2017, Moosavi-Dezfooli et al. [19] showed the existence of a universal and very small perturbation vector that causes state-of-the-art deep neural networks to be misclassified with high probability.

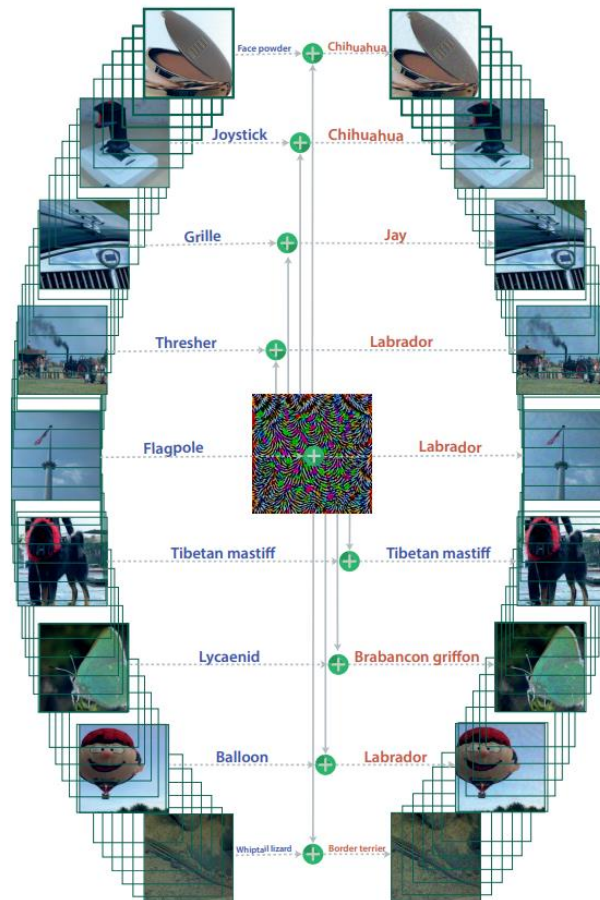


Figure 11: Universal perturbation. Left - original images with the true label. Middle - small universal perturbation to be added. Right - Adversarial examples and their estimated label. (Source: [19])

In the same year, some studies demonstrated the existence of adversarial learning in the real physical world. Kurakin et al. [20] showed that adversarial images that were printed and recaptured using a cell phone camera, were still misclassified by the network (Figure 12). Another research introduced some methods for road sign attacks where stickers on a stop sign were managed to fool the network [21].

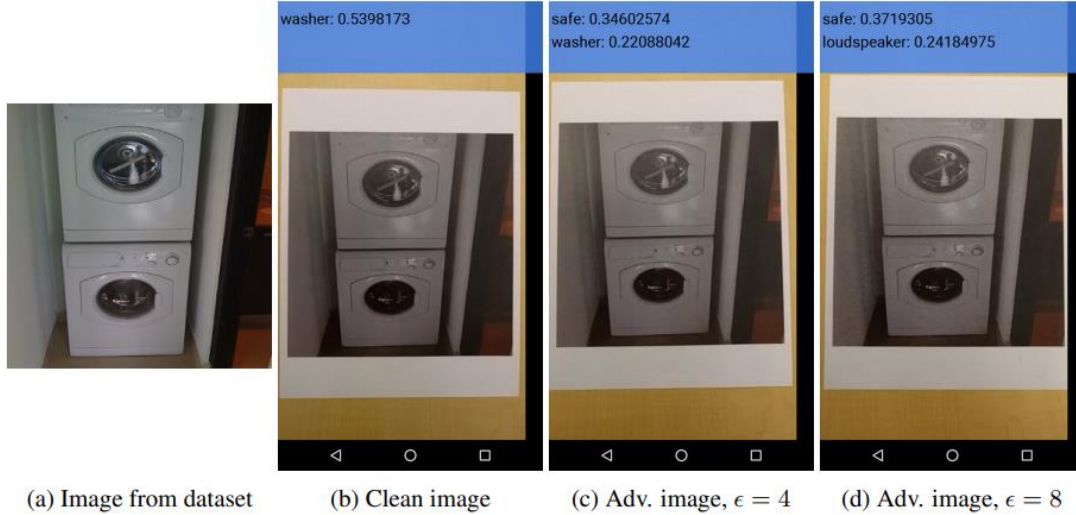


Figure 12: Black box attack on a phone image classification application. On the left is the original image. This image was printed after adding small perturbations defining by the epsilon parameter. The model failed to classify it correctly (Source: [20])

3.1. Definitions of Terms

In this section we describe common technical concepts and terms used in the world of adversarial learning. The focus is on the image processing domain.

- *Adversarial example* x_{adv} - A malformed version of an original input image that was deliberately altered to deceive a learning model.
- *Adversarial training* - A defense strategy against adversarial learning where adversarial examples are intentionally added to the training process of the model.
- *Black-box attack vs White box attack* - In the black-box scenario, the attacker doesn't have any prior knowledge about the model, its parameters, loss function, architecture or algorithm in contrast to the white-box case.
- *Clean image* x_c - The original input image.
- *Fooling ratio* - Refer to the percentage of test images that changed their true label after been perturbed.
- *Image-specific vs universal attack* - Whether the perturbation vector is calculated for each image separately or there is a universal perturbation vector for all test images.
- *One-shot vs iterative method* - One-shot method is when the calculation process is in a single step compared to an iterative process that in most cases will be computationally expensive.
- *Perturbation* - A small noise vector that was added to a clean image making it an adversarial example.
- *Targeted vs non-targeted attack* - In a targeted attack, the model was fool to believe the adversarial example belonged to a specific label wherein non-targeted any label different from the true label is valid.
- *Transferability* - The ability of an adversarial example to mislead other models as well.

3.2. Adversarial Classification Attacks

In this section, we review some popular methods for adversarial classification attacks in the image processing domain. There are many other attack methods on different architectures and domains such as attacks on *Recurrent Neural Networks*, *Autoencoders* and *Reinforcement Learning* tasks, etc', but we focus on the common case of classification problems. Of course, there are many different approaches, where some of them are the evolution of each other, and we cannot mention all of them. Therefore we give a review of only the more common methods or those that have formed a milestone in adversarial history while trying to give a varied list that presents the nature and properties of different attacks. The interested reader can see the surveys [22],[23],[24] for further reading.

3.2.1. First Discovery - Box constrained L-BFGS

Properties: White-box, Non-targeted, Image-specific, One-shot.

Szegedy et al. [1] first introduced the term of *adversarial learning* by finding what they called blind spots in neural networks. They formalize the following minimization problem as the search for adversarial examples:

Let $f : \mathbb{R}^m \rightarrow \{1, \dots, k\}$ be a deep neural network classifier mapping images to a discrete set of classes with a continuous loss function $loss_f : \mathbb{R}^m \times \{1, \dots, k\} \rightarrow \mathbb{R}^+$. Then for an input image $x \in \mathbb{R}^m$ and a target label $l \in \{1, \dots, k\}$ we wish to solve the following optimization problem

$$(7) \arg \min_p \|p\|_2$$

$$s.t. \quad f(x + p) = l, \quad x + p \in [0, 1]^m$$

where p is the perturbation vector. This problem has a nontrivial solution in case of $f(x) \neq l$. The original problem is hard, so the authors suggest approximating it by using a box-constrained L-BFGS. They look for the minimum $c > 0$ for which the minimizer p of the following problem satisfies $f(x + p) = l$:

$$(8) \min_p c \cdot |p| + loss_f(x + p, l)$$

$$s.t. \quad x + p \in [0, 1]^m$$

This will result in the exact solution in the case of a convex loss function, but this is not the general case in neural networks.

As mention, this was the first suggested algorithm for finding adversarial examples, which cause a wide interest of researches in the adversarial attacks field.

3.2.2. Linear and Gradient-based - Fast Gradient Sign Method (FGSM)

Properties: White-box, Non-targeted, Image-specific, One-shot.

Goodfellow et al. [16] tried to explain the adversarial examples transferability across architectures and training sets with the linear nature of deep neural networks. They suggested having a linear process for creating perturbations. The adversarial examples were generated in the following way:

$$(9) x_{adv} = x_c + \varepsilon \cdot \text{sign}(\nabla J(\theta, x_c, y_{true}))$$

where θ are the model parameters, x_c is the input image, y_{true} is the true label of x_c , and ∇J is the gradient of the cost function. ε is a small scalar number controlling the amplitude of the perturbation vector. They name it Fast Gradient Sign Method (FGSM).

Figure 13 illustrates some visual results for the FGSM algorithm on the CIFAR 10 dataset. It can be seen that for small values, it is difficult to discern that the image has been manipulated but still in many cases the network will be surprisingly wrong. For example, in the case of the bird, the deer, the dog, the frog, the ship and the truck even for epsilon of 0.01, the classification was wrong, although it is clearly seen that the nature of the image has not changed. A large value of epsilon increase the likelihood of being mistaken on the network, but, of course, also detract from image quality and make it easier to recognize that the input has been manipulated.

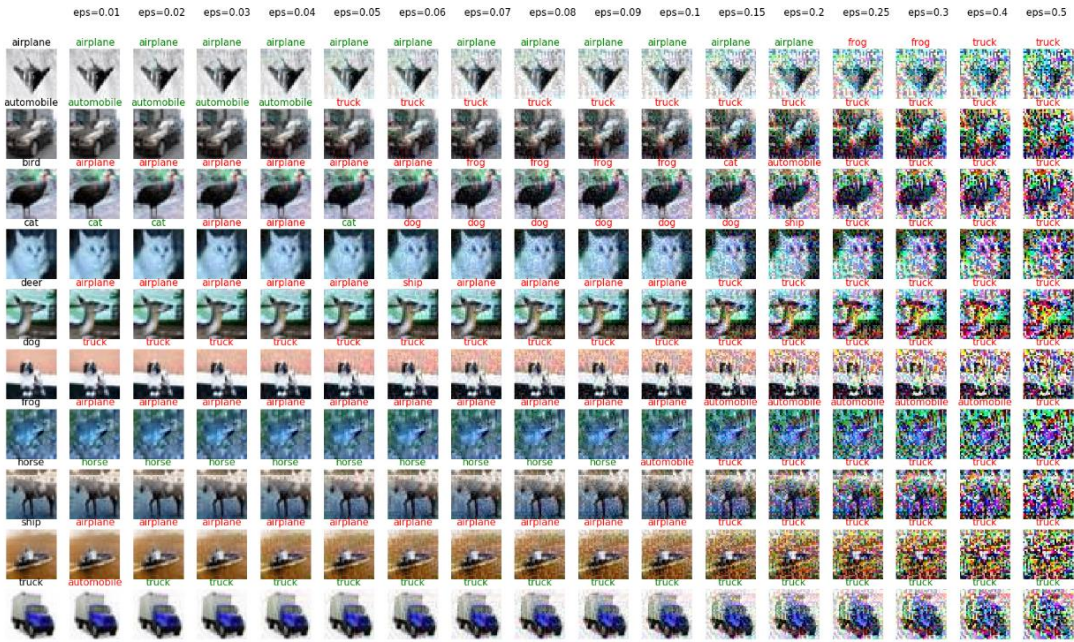


Figure 13: A demonstration of the Fast Gradient Sign Method (FGSM) on the CIFAR10 dataset. The leftmost image is the original file and its category while moving right express different values of epsilon during the adversarial attack and the prediction of the network. Green color means correct classification, while red color indicates an error.

FGSM is one of the most used adversarial techniques and will be our main algorithm in this paper. Other variations of FGSM were published during the years.

3.2.2.1. One-step target class method

Properties: White-box, Targeted, Image-specific, One-shot.

Kurakin et al. [25] propose an alternative approach to maximize the probability of a specific label differs from the true label. The new formula will be as followed:

$$(10) x_{adv} = x_c - \varepsilon \cdot \text{sign}(\nabla J(\theta, x_c, y_{target}))$$

3.2.2.2. Basic Iterative Method (BIM)

Properties: White-box, Non-targeted, Image-specific, Iterative.

Till now we saw only one-shot methods taking a single large step in the direction of increasing the loss function. Another approach is taking some small steps to get a better result. This extension [25] got the name Basic Iterative Method (BIM):

$$(11) x_{adv}^0 = x_c, \quad x_{adv}^{N+1} = \text{Clip}\left\{x_{adv}^N + \alpha \cdot \text{sign}(\nabla J(\theta, x_c, y_{true}))\right\}$$

where α is the step size (the authors chose $\alpha = 1$, meaning they change the pixel by 1 at each step) and $\text{Clip}\{\cdot\}$ clips the values of the values to fit the interval $[x_{c,i,j} - \varepsilon, x_{c,i,j} + \varepsilon]$. The authors determined the number of iterations as $\min(\varepsilon + 4, 1.25 \cdot \varepsilon)$.

3.2.2.3. Iterative Least-likely Class Method (ILCM)

Properties: White-box, Non-targeted, Image-specific, Iterative.

By combining (10) and (11) we can get an iterative method of choosing the least likely class:

$$(12) x_{adv}^0 = x_c, \quad x_{adv}^{N+1} = \text{Clip}\left\{x_{adv}^N - \alpha \cdot \text{sign}(\nabla J(\theta, x_c, y_{least-likely}))\right\}$$

This extension got the name Iterative Least-likely Class Method (ILCM) [25] and produce adversarial examples that shown some serious effect on the classification accuracy.

3.2.3. ℓ_0 norm based - JSMA and One Pixel Attack

Properties: White-box (JSMA) / Black-box (One Pixel), Non-targeted, Image-specific, Iterative.

Most methods are using ℓ_2 and ℓ_∞ norms when minimizing the perturbation vector. Different approaches suggest looking at ℓ_0 norm, meaning the number of pixels that were changed during the adversarial process. Papernot et al. [26] created a saliency map using the gradient of the network layers output to monitor how the change of each pixel affects the probability of misclassification. Once the map is ready it is easy to choose the most effective pixel. The algorithm restricts the number of pixels that can be altered. This method got the name Jacobian-based Saliency Map Attack (JSMA).

Taking it to the extreme, Su et al. [27] demonstrated how a single pixel change can mislead the network.

3.2.4. Universal based Attack

Properties: White-box, Non-targeted, Universal, Iterative.

Moosavi-Dezfooli et al. [19] described the first method defining a general perturbation for a network that is not image-specific. Meaning, by adding the perturbation to any image the network will misclassify on it with high probability. Let μ denote a distribution of clean images in \mathbb{R}^m . Their goal is to find a perturbation p that will be as small as possible and that will cause the network to be wrong in “most” cases. To formalize it the following 2 conditions should be applied:

$$(13) \quad \begin{aligned} \|p\| &\leq \xi \\ P_{x \sim \mu} (f(x+p) \neq f(x)) &\geq 1 - \delta \end{aligned}$$

where δ controls the fooling ratio and the parameter ξ limits the magnitude of the perturbation vector. To solve this the authors used an iterative approach going through each image input adding to the current universal perturbation the smallest possible vector that will make him the network to misclassified the current image.

$$(14) \quad \Delta p^i \leftarrow \arg \min_r \|r\|_2 \quad s.t. \quad f(x_c^i + p + r) \neq f(x_i)$$

3.2.5. Neural Network-based - Adversarial Transformation Network (ATN)

Properties: White-box, Non-targeted, Image-specific, Iterative

Up to this point, most of the methods have focused on the direct calculation of the gradient or solving an optimization problem. Baluja and Fischer [28] idea was to train a feed-forward neural network generating adversarial examples that are misclassified by a target network. They call it the Adversarial Transformation Network (ATN). They did it by minimizing a loss function combining the task of similarity between the original image and the adversarial version while having a misclassification by the target network.

3.3. Adversarial Defense Strategies

In this section, we provide an overview of some of the main paradigms to defend against an adversarial attack. It is important to mention that there is currently no general method that provides a proper solution to all types of adversarial attacks and this is an ongoing research field. The main difficulty is due to the lack of theoretical tools to deal with the inputs that were deliberately crafted while dealing with non-linear and non-convex environment and hard optimization problems to solve. In addition, adding mechanisms for protecting the models may affect the model accuracy on clean images as well as running time (training or testing) and resource consumption.

Current approaches can be divided into the following families:

- Training Modification - like adversarial training
- Data modification - may be on the training phase, test phase or both
- Network modification - make it more robust to small changes (fix the gradient, change the loss function or make the network deeper)
- External models - like detection systems

Here are some of the most popular methods to defend against adversarial attacks.

3.3.1. Training Modification - Adversarial Learning

One of the most popular methods in research today is adversarial learning. In this approach, the objective is to increase the robustness of the model by injecting random adversarial examples into the training phase alongside the original images. This method differs from the classical augmentation which includes image transformation such as rotation and results in a true input image. Adversarial training enriches the data with images that are unlikely to receive as input.

Szegedy et al [1], who were the first to point out the existence of blind spots in neural networks, were also the first to examine the effect of adding these examples to the training set. Many studies suggesting new attack methods have at the same time introduced adversarial learning as the main defensive approach. Goodfellow et al [16] offered an alternative objective function combining both the original and the adversarial example generated by the Fast Gradient Sign Method:

$$(15) \quad \tilde{J}(\theta, x_c, y_{true}) = \alpha J(\theta, x_c, y_{true}) + (1 - \alpha) J(\theta, x_c + \varepsilon \cdot \text{sign}(\nabla J(\theta, x_c, y_{true})), y_{true})$$

where J is the original loss function and $\alpha \in (0, 1)$.

A mentionable drawback of adversarial learning is that it tends to overfit to the specific attack method used while training. Moreover, the defense is not robust to black-box attacks.

3.3.2. Data Modification

Many attack methods create a high-frequency noise that is not distinguished by the human eye. This phenomenon led to several studies focusing on the preprocessing of the image data as a defense strategy. Several main approaches can be identified:

- Image compression
- Image denoising
- Feature squeezing
- Randomization

Image compression is probably the most common method, and especially the lossy compression of JPEG. One of the main features of JPEG compression making it relevant as an adversarial defense approach is the ability to remove high-frequency signals. Dziugaite et al [30] were the first to examine the effect of JPEG compression on adversarial samples calculated by FGSM [16]. Their research showed that JPEG compression is effective for large scale perturbations while small interferences may survive the process. Das et al [31] took a similar approach and extend the experiment to DeepFool method [32] as well. Their main focus was on the trade-off between image quality and accuracy improvement. They suggested taking an ensemble approach running several different compressions with different qualities while having a vote between all of the results. Shin et al [34] showed how to generate an adversarial sample that will survive JPEG compression.

Further research by Cornell and Facebook [33] reached similar conclusions where other methods for feature squeezing and image denoising were tested such as image cropping and rescaling, bit depth-reduction and Variance minimization. An ensemble approach has also been tested while averaging the results. Shaham et al [35] have also experimented the effectiveness of low pass filtering, PCA, JPEG compression, soft thresholding and low-resolution wavelet approximation as defense approaches against adversarial attacks. Sahay et al [40] on the other hand used autoencoder to denoise the input image and then used another autoencoder last hidden layer to apply a dimensionality reduction.

A slightly different approach was taken by Xie et al [36] who offered to combine random elements (see Figure 14) and came second in the NIPS 2017 adversarial defense challenge.

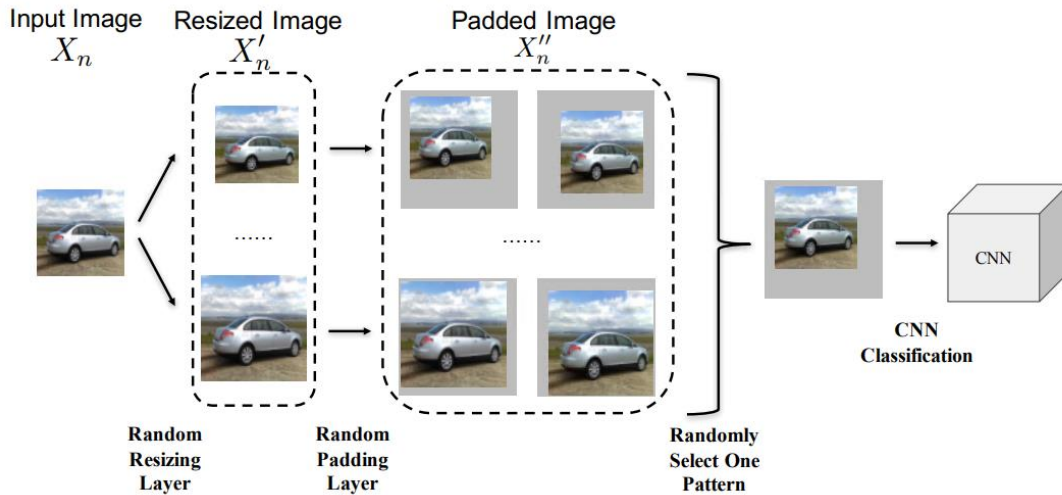


Figure 14: In this research Xie et al [36] used two randomization layers as a defense against adversarial examples: random image resizing and random zero paddings around the image.

3.3.3. Network Modification

Several methods have tried to address the phenomenon of adversarial learning by modifying the network and making it more robust to small changes.

One of these methods is *gradient masking* or *gradient hiding*, which tries to hide information about the gradient from the adversary. Many white-box attacks use the gradient of the model to calculate the perturbation vector, meaning that turning it to non-differentiable or close to zero makes it totally useless. Ross and Doshi-Velez [42] suggested a defense process that penalizes the degree to which small changes in inputs can alter model predictions. This method may be useful to gradient-based attacks but double the complexity of the training process.

3.3.4. External Models

Another defensive option may include add-on detectors to find out whether or not the image has been poisoned. An example of this approach can be found in the papers of [37] and [38] similar approach was proposed where different methods of feature squeezing have been used and all of the predictions were examined alongside the original image prediction. A large difference will be an indicator of an adversarial image in most cases.

Methods based on external detectors require a large number of adversarial examples and tend to suffer from overfitting on the adversarial algorithm that generated the attack.

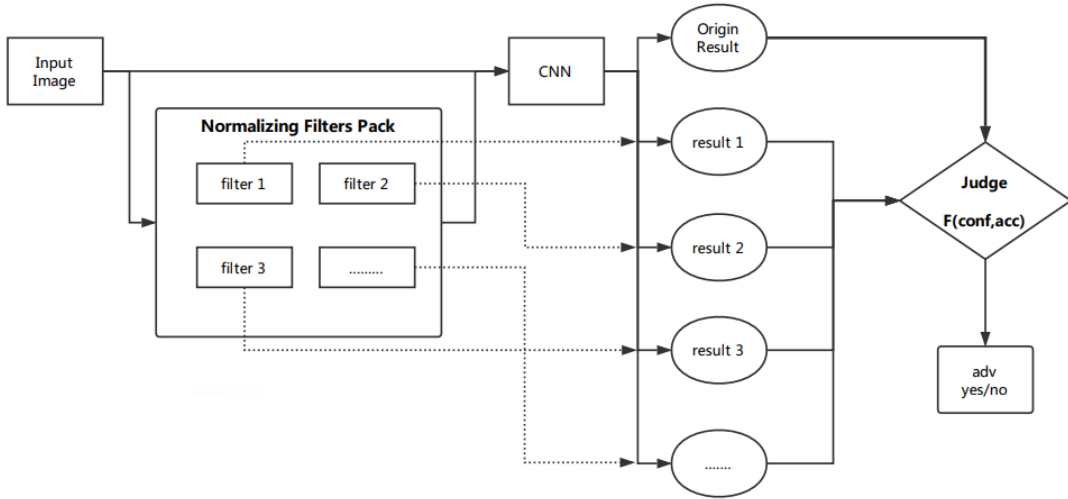


Figure 15: Detector for adversarial examples. In this architecture, the model gets both the original image and the transformed samples. All the predictions are sent to a judge module where large different is an indicator for an adversarial example (source: Gu et al [37])

Another approach that uses Generative Adversarial Network (GAN) was introduced by Lee et al [41]. In their research, they train simultaneously a network whose goal was to correctly classify clean and adversarial images and a network that tries to produce images that would fool the network. As in any GAN based architecture, the two networks compete with each other. This can be seen as a combination of adversarial training and an external model. Using the GAN-based defense method may be effective on gradient-based attacks but requires longer training time and resources.

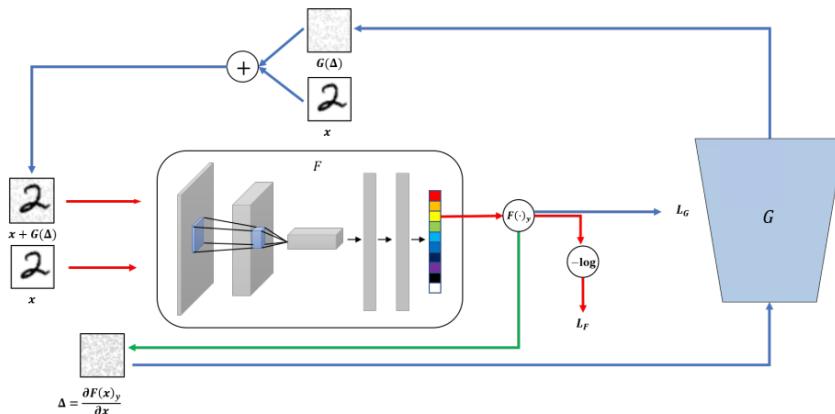


Figure 16: The classifier F is trained on both clean and adversarial images. A generative network G is training to generate adversarial examples using the clean image and its gradient (source Lee et al [41])

4. The Dimension Effect

Goodfellow et al. [16] were the first to point on the high dimension as one of the main factors that increase the probability of adversarial example to deceive the network, because the same amount of noise in each dimension will result in larger Euclidian distance from the original instance. This assumes a linearity behavior of NN which is known as the *linearity hypothesis*.

4.1. Linearity Hypothesis

Szegedy et al. [1] first discovered the existence of adversarial examples, but could not explain the cause. At first, it was widely assumed that this was due to the complexity and the nonlinearity nature of neural networks. It was also thought that it was just an example of overfitting.

The leading hypothesis today, although not a consensus, is the *linearity hypothesis* suggested by Goodfellow et al. [16]. They claimed that the original hypothesis could not explain why simple and shallow models suffer from adversarial examples as much as deep models. Meaning the complexity and overfitting were failed to explain this phenomenon. Adversarial examples are able to fool models different from those they were originally derived from and with the same prediction. If adversarial is a form of overfitting, each network should react differently to these examples.

Goodfellow et al. [16] presented the *linear hypothesis* which argued that the source of adversarial examples is that the model behaves extremely linearly as a function of its inputs. They claim that neural networks are too linear to resist linear adversarial perturbation. For easier optimization process, most of the NN architecture such as LSTM, ReLUs and maxout were intentionally designed to behave in a linear manner. As a result, they suggested a linear method for generating adversarial examples - the FGSM algorithm [16]. Their main hypothesis was that the neural network is too linear to resist linear adversarial.

4.2. The Curse of Dimensionality

Real-life datasets typically come with high dimensions, like the number of pixels in an image or the number of different words in a text document. The true dimension is often much lower. For example, if we are looking at a 28x28 handwritten digits image like in the MNIST dataset, the input dimension is $[0,1]^{28 \times 28}$ which is the total possible for input images. If we choose a random image from this domain, most chances it will not be a handwritten digit, but some random black and white image. In fact, handwritten is only a tiny fraction of events in this large input space. So, why high dimension input space could be a problem?

The Curse of Dimensionality describes a phenomenon in which when the input dimension increases, the volume of the space increases exponentially, making it sparse. If for example, 10 data points seem reasonable for 1-dimensional space, in 2-dimensional space we'll need 100 points for the same density of points and 1000 points for the 3-dimensional case. Most machine learning algorithms are statistical by nature, using counting of observations in various regions of some space and distance measures. Those two fails when the dimension is increasing.

4.3. The Adversarial Curse of Dimensionality for Linear Models

Goodfellow et al. [16] tried to explain how a combination of high dimension along with the linearity hypothesis of neural network models can lead to adversarial examples.

In many cases, each feature has a precision limit. This could be a digital image that is stored as 8-bit so that any color channel in a pixel cannot measure any change lower than $1/255$, or an accuracy limitation

of any of the sensors. For input x , adversarial image $\tilde{x} = x + p$ and an activation function $f(x)$ it is reasonable to think that the model will respond the same for both if every element of the vector p is smaller than the precision of the features. Meaning, we expect $f(x) \approx f(x + p)$ for $\|p\|_\infty < \varepsilon$, where ε is smaller than the sensor precision. Consider a linear model we can describe the output for the adversarial input as follow:

$$(16) \quad f(\tilde{x}) = w^T \tilde{x} = w^T x + w^T p$$

The adversarial perturbation p will cause the activation function $f(x)$ to grow by $w^T p$. To maximize this growth we can assign $p = \text{sign}(w)$. For an n -dimensional vector w with an average magnitude of m , the activation function will grow by εmn .

$$(17) \quad f(\tilde{x}) \approx f(x) + \varepsilon mn$$

Meaning, the activation value will grow linearly with n for high dimensional problems. This can explain why linear models with high dimension inputs may suffer from adversarial examples.

5. Dimensionality Reduction Defence

In this section, we provide a detailed description of all dimension reduction methods that have been tested as a way to protect against adversarial examples. We then describe the unique approach we decided to take - an *ensemble* of several models. We will take a look at what the *ensemble* approach is and describe the *stacking* method we chose to implement.

Then, we present each experiment - what metrics we chose to use, the datasets to work on, the network architecture, the generation of the adversarial examples and the experimental steps.

5.1. Dimensionality Defence Methods

In this section, we will cover the dimensionality reduction methods that have been examined in this study. For each method, we will provide some theoretical background, the original Python source code that was applied and some image examples.

5.1.1. Image Resize and Rescale

We will start with a simple image squeezing of rescaling and resizing. Using this method, we wish to shrink an image by a certain factor and then expand it again to its original size. This is done with the *skimage* package by applying *rescale* and *resize* functions.

```
from skimage.transform import rescale, resize

def resize_dim_reduction(im, factor):
    new_im = resize(rescale(im, 1. / factor, multichannel=True),
                    (im.shape[0], im.shape[1], im.shape[2]))
    return new_im
```

Rescale operation resizes an image by a given scaling factor. Resize serves the same purpose, but allows to specify an output image shape instead of a scaling parameter.

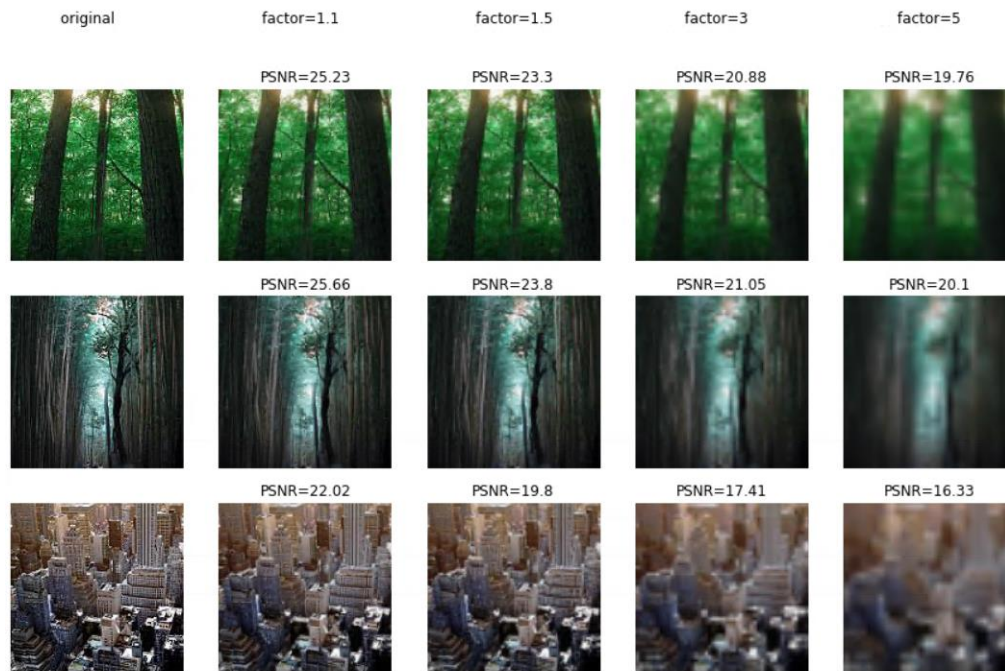


Figure 17: Rescaling an image by a given factor and then resizing it to its original shape. PSNR reflects the damage in image quality. The images are taken from the INTEL dataset.

5.1.2. K-Means Color Auantization

K-means is one of the most popular and widely used clustering algorithms both in literature and industry. Its simplicity and speed give K-means a major advantage over other, more accurate, methods. Given an integer K and a set of n data points $x \in \mathbb{R}^d$, the goal is to find k clusters centers C that minimizes ϕ the total squared distance between them.

$$(18) \phi = \sum_{i=1}^n \min_{c \in C} \|x_i - c\|^2$$

This problem is NP-hard, but has an iterative approximation algorithm suggested by Lloyd [43] in 1982:

1. Choose a random initial k centers $C = \{c_1, c_2, \dots, c_k\}$
2. For each data point $x_i : i \in [1, n]$ assign it to its nearest center
3. For each $i \in [1, k]$ set c_i to be the center of mass of all the points that were assigned to the group C_i , meaning $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.
4. Repeat steps 2 and 3 until no update has been made

Here we perform a pixel-wise vector quantization of the image to reduce the number of colors. A random codebook that is taken from each image is been used to quantized image colors.

```
from sklearn.utils import shuffle
from sklearn.cluster import KMeans

def recreate_image(codebook, labels, w, h):
    d = codebook.shape[1]
    image = np.zeros((w, h, d))
    label_idx = 0
    for i in range(w):
        for j in range(h):
            image[i][j] = codebook[labels[label_idx]]
            label_idx += 1
    return image

def kmeans_dim_reduction(im, k, n_sample_pixles=1000):
    im = im.astype('float64')

    w, h, d = tuple(im.shape)

    image_array = np.reshape(im, (w * h, d))
    image_array_sample = shuffle(image_array, random_state=0)[:n_sample_pixles]

    kmeans = KMeans(k)
    kmeans = kmeans.fit(image_array_sample)

    labels = kmeans.predict(image_array)

    new_image = recreate_image(kmeans.cluster_centers_, labels, w, h)

    return new_image
```

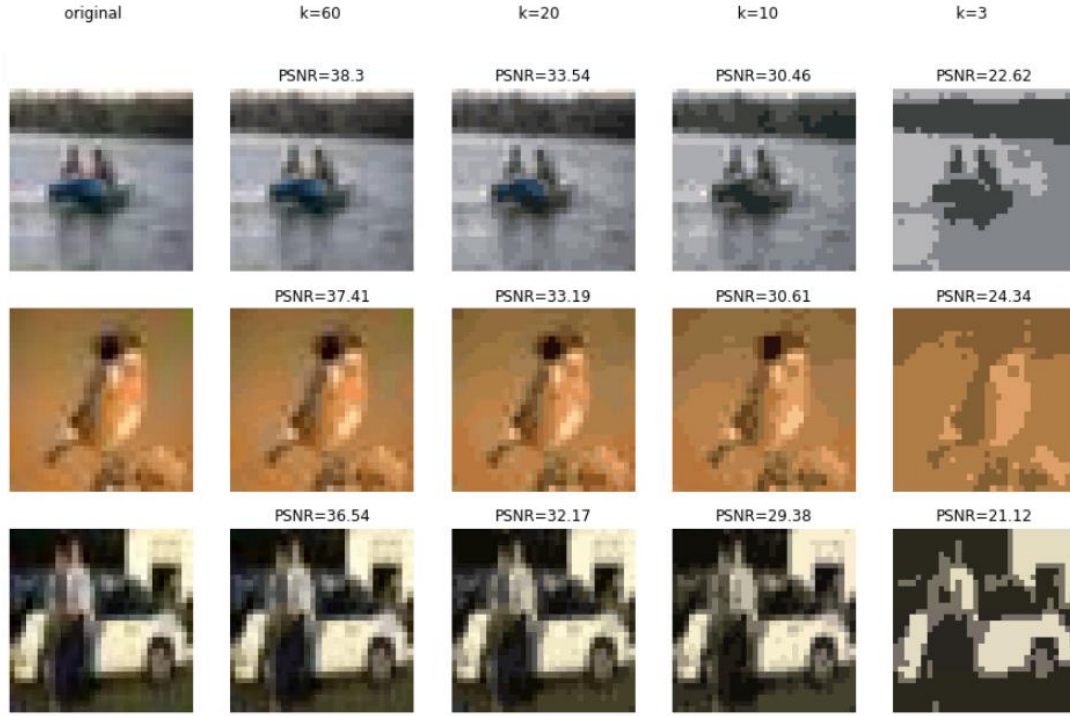


Figure 18: K-means quantization of an image with a given k value. PSNR reflects the damage in image quality. The images are taken from the CIFAR10 dataset.

5.1.3. PCA

PCA is a dimensionality reduction and denoising process that try to keep as much variance as possible in the original input matrix. This is done by finding the k leading principal component of the input, representing it in the PC space, and mapping it again to the input space.

For $\{x_1, x_2, \dots, x_m\} \in \mathbb{R}^d$ we would like to reduce their dimension with a linear transformation $W \in \mathbb{R}^{k,d}$, meaning having a mapping $y = Wx : W \in \mathbb{R}^{k,d}$. Then we wish to recover the original vector using a linear transformation $\tilde{x} = Uy : U \in \mathbb{R}^{d,k}$ such that the compressed vector will be close to its source. To find the compression matrix W and the recovering matrix U that minimize the total squared distance we aim to solve the following optimization problem:

$$(19) \arg \min_{\substack{W \in \mathbb{R}^{k,d} \\ U \in \mathbb{R}^{d,k}}} \sum_{i=1}^m \|x_i - UWx_i\|_2^2$$

In the optimal solution of (19) the columns of U are orthonormal and will be the first largest eigenvectors of the covariance matrix $A = \sum_{i=1}^m x_i x_i^T$ and $W = U^T$. For full solution and proof please refer to [3].

We perform the PCA calculation, meaning finding the eigenvector, for the all training set and not for each image separately. Attached is the python implementation of this linear transformation.

```

from PIL import Image
import glob
import os
import numpy as np
import pickle
from sklearn.preprocessing import normalize
from sklearn.decomposition import PCA

def learn_pca_model(path, percent):
    file_name = path.replace("\\", "_") + '_pca_' + str(percent) + '_model.pkl'

    if not os.path.isfile(file_name):

        imlist = glob.glob(path + "\\**/*.jpg", recursive=True)

        # dimensions
        im = np.array(Image.open(imlist[0]))
        m, n = im.shape[0:2]
        imbr = len(imlist)

        # matrix with flattened images
        print('Found ' + str(imbr) + ' images.')
        print('Read and flatten images.')
        immatrix = np.asarray([(np.array(Image.open(im)) / 255.0).flatten() for im in
imlist], 'f')

        # pca
        print('Learning PCA model.')
        pca = PCA(percent)
        pca.fit(immatrix)

        with open(file_name, 'wb') as output:
            pickle.dump(pca, output, pickle.HIGHEST_PROTOCOL)

    else:

        with open(file_name, 'rb') as inp:
            pca = pickle.load(inp)

    return pca

def pca_dim_reduction(im, pca_model):
    X = [im.flatten()]

    lower_dimensional_data = pca_model.transform(X)

    approximation = pca_model.inverse_transform(lower_dimensional_data)

    approximation = approximation.reshape(1, im.shape[0], im.shape[1], 3)

    return approximation[0]

```



Figure 19: Performing PCA of an image with a given factor value representing the percent of variance to preserve. PSNR reflects the damage in image quality. The images are taken from the INTEL dataset.

5.1.4. Filtering

Image filtering has many applications, including de-noising, smoothing, sharpening, and edge detection. Linear filtering of an image is obtained by performing a convolution operation defined by a kernel. In the general form, convolution is defined by:

$$(20) \quad g(x, y) = \omega * f(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b \omega(s, t) f(x-s, y-t)$$

where $f(x, y)$ is the original image, ω is the kernel and $g(x, y)$ is the filtered image. In our case, it is a weighted sum of neighboring pixels as can be seen in figure 20.

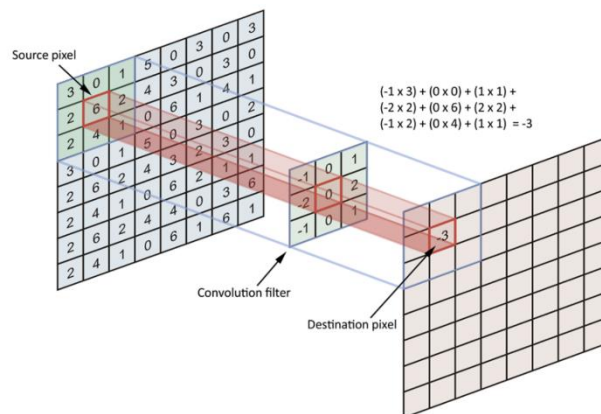


Figure 20: An image filtering applying as a linear convolutional operation on each pixel.

Several kernels were examined.

5.1.4.1. Low pass filter

Images can be filtered with low-pass filters (LPF) or high-pass filters (HPF). LPF is useful in noise removal or blurring the image while HPF helps in edge detection. The smoothed image is achieved by averaging nearby pixels. A simple example is a kernel of all ones divided by the number of elements within the kernel:

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In this case, the kernel size is 3 but can be any odd number. Other filters may include more weighting for the current pixel or different smoothing in the x and y-axis.

Attached is a sample code for low pass filtering using a matrix of ones and a kernel size parameter.

```
import numpy as np
import cv2

def low_pass_filter_dim_reduction(im, factor):
    kernel = np.ones((factor, factor), np.float32) / (factor * factor)
    new_im = cv2.filter2D(im, -1, kernel)

    return new_im
```

Below are some examples of different kernel sizes.

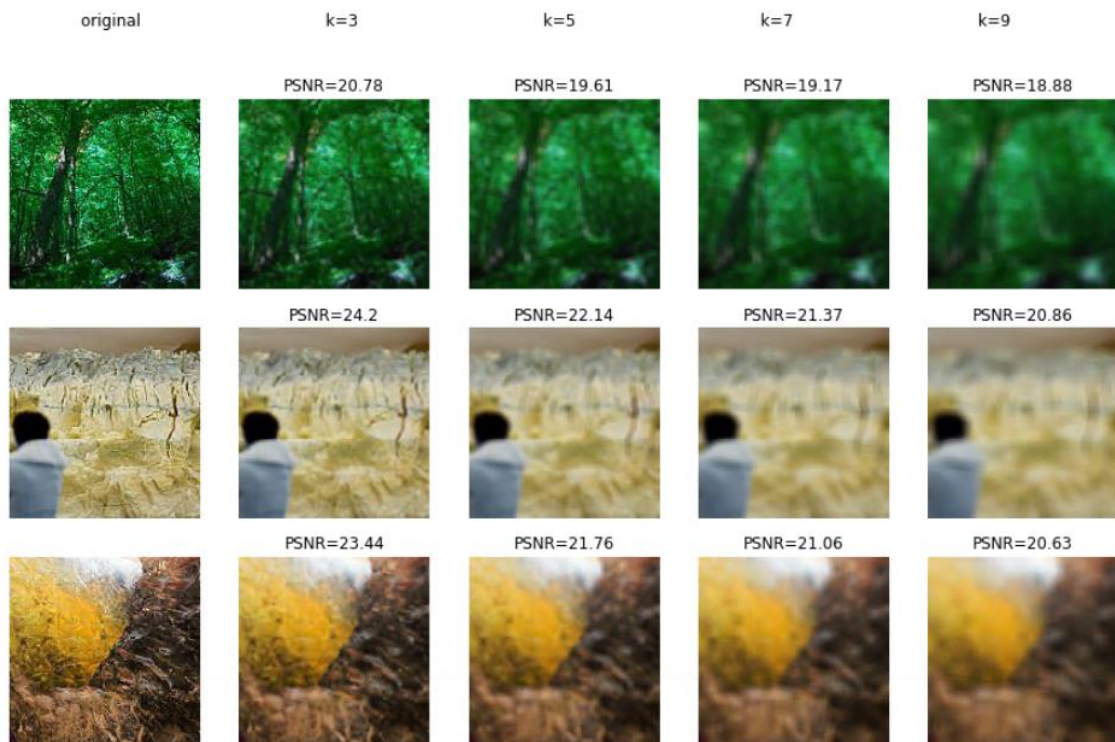


Figure 21: Performing Low Pass Filtering of an image with a given factor value representing kernel size. PSNR reflects the damage in image quality. The images are taken from the INTEL dataset.

5.1.4.2. Gaussian filter

Gaussian filter is widely used to reduce image noise. The gaussian function express the normal distribution for calculating the transformation in each pixel. The gaussian formula for 2 dimensions is

$$(20) G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where x and y are the distance from the axes and σ is the standard deviation. The distribution is shown in figure 22.

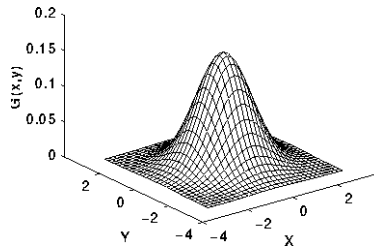


Figure 22: 2-D Gaussian distribution with mean (0,0) and $\sigma = 1$

Here is an example for 5x5 Gaussian kernel:

$$\frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

We used the implementation of the OpenCV python package with $\sigma = 0$ for both axes, x and y.

```
import cv2

def gaussian_filter_dim_reduction(im, factor):
    new_im = cv2.GaussianBlur(im, (factor, factor), 0)

    return new_im
```

Attached are some examples of applying this method.

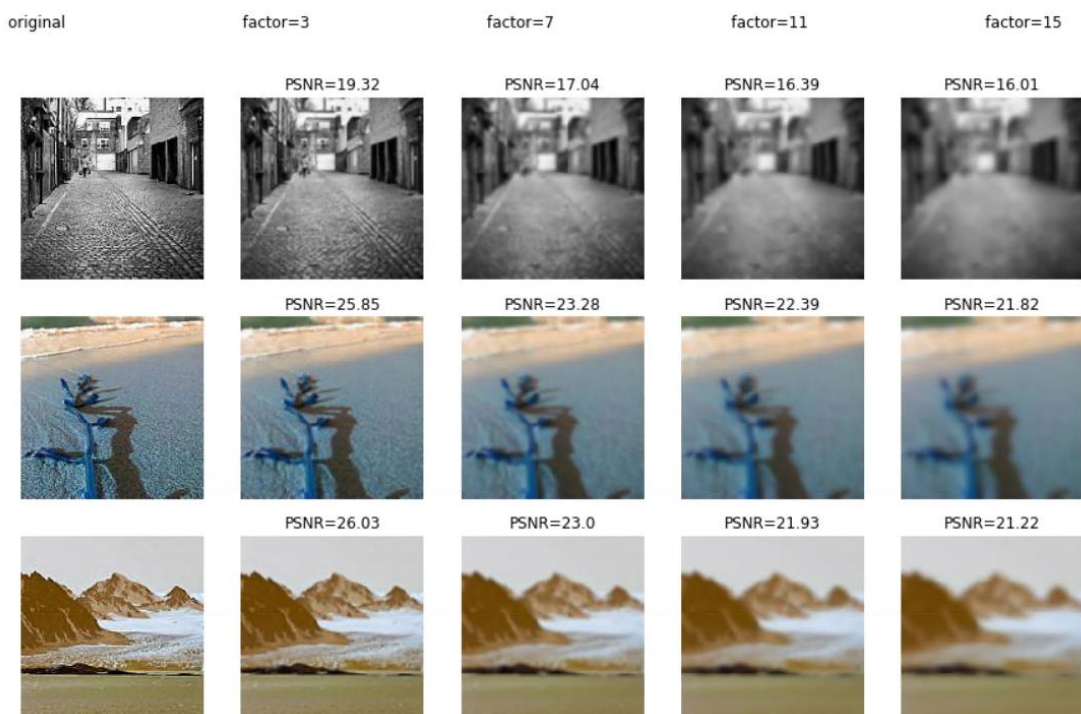


Figure 23: Performing Gaussian Filtering of an image with a given factor value representing kernel size. PSNR reflects the damage in image quality. The images are taken from the INTEL dataset

5.1.4.3 Median filter

A median filter is an effective non-linear technique for reducing random noise while preserving edges. This is done by a sliding window placing the median value across the input window. Unlike averaging or Gaussian filtering which can create new colors, the median filter will place some pixel value from the image.

Bellow is the median filtering implementation using the python OpenCV package.

```
import cv2

def median_filter_dim_reduction(im, factor):
    new_im = cv2.medianBlur(im, factor)

    return new_im
```

Figure 24 illustrates the effect of the median filter on images with random Gaussian noise. The most important impact, as can be seen, is the edge-preserving.

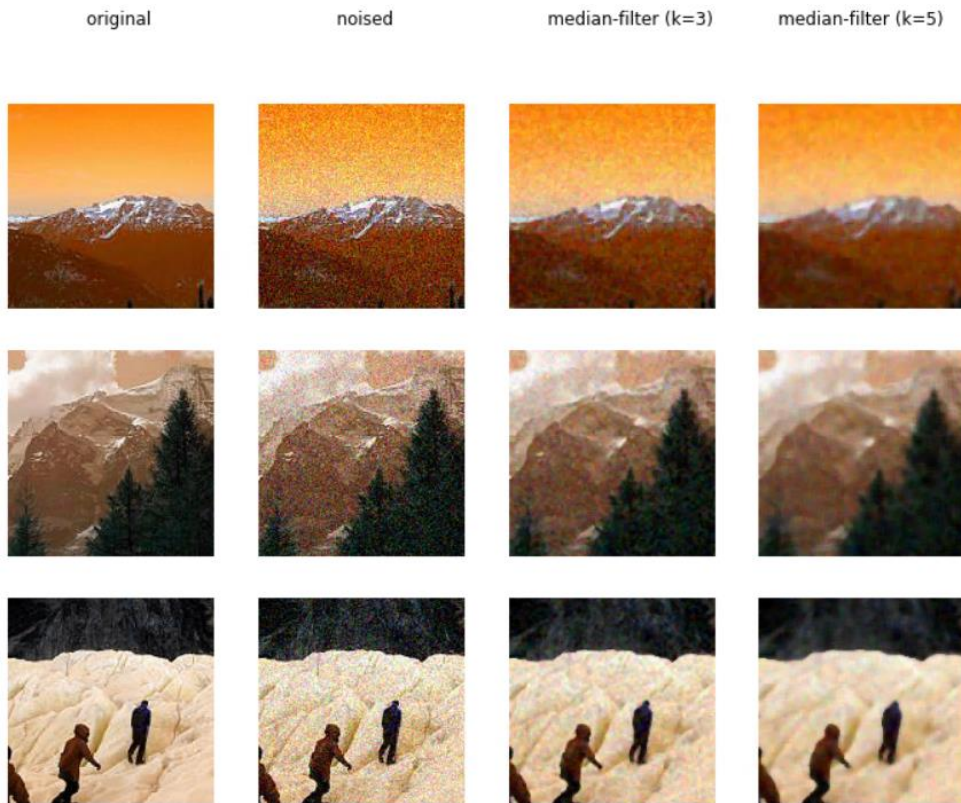


Figure 24: Performing Median Filtering of an image with a given factor value representing kernel size. On the left is the original image and beside it is the image with some random Gaussian noise. The next two images re median filtered images with a window size of 3 and 5. The images are taken from the INTEL dataset

5.1.4.3. Gradient

Gradient filters is a high pass filter (HPF) which is looking for a directional change in color or intensity. In general, the gradient is expressed as

$$(21) \nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

The direction of the gradient may be in the x-axis only, y-axis or any other vector. The derivative in the image case may be approximated by finite differences, for example, the gradient of image A on the y-axis can be written as a 1-D convolution

$$\frac{\partial f}{\partial y} = \begin{bmatrix} -1 \\ +1 \end{bmatrix} * A .$$

We examined 3 types of gradient filters: Laplacian, Sobel, and Scharr. Sobel and Scharr use a 3x3 kernel for finding edges along the x or y-axis. The two discrete filters described above with the x and y-axis version:

$$C_{sobel_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad C_{sobel_y} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$C_{scharr_x} = \begin{bmatrix} -3 & 0 & +3 \\ -10 & 0 & +10 \\ -3 & 0 & +3 \end{bmatrix} \quad C_{scharr_y} = \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Laplacian, on the other hand, is given by the formula:

$$(22) \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Since the image input is represented as discrete pixels, it is common to use an approximation sliding window to the second derivatives in the definition of the Laplacian. Two commonly used kernels are shown above:

$$C_{laplacian_1} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \quad C_{laplacian_2} = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 25 illustrates the difference between all the mentioned gradient methods.

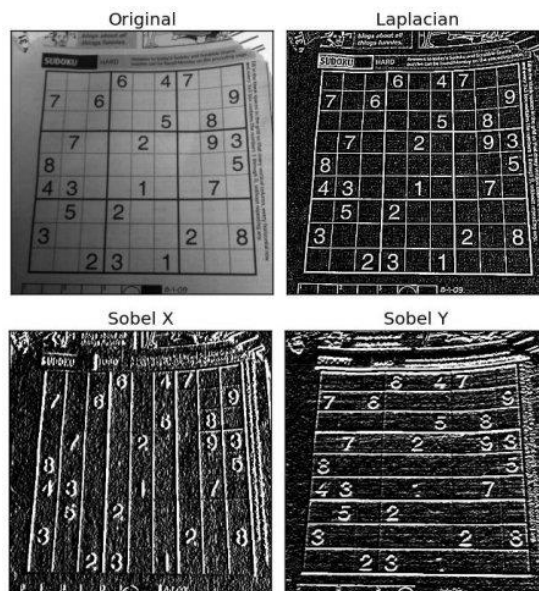


Figure 25: Performing different kinds of Gradient Filtering on an image. On the upper left is the original image. The other images show the output of Laplacian, Sobel x and Sobel y kernels. The images are taken from the [OpenCV documentation](#)

Attached is the python source code for implementing the gradient masking using the open cv package.

```

import cv2
import numpy as np

def laplacian_derivatives_dim_reduction(im):
    # | 0 1 0 |
    # | 1 -4 1 |
    # | 0 1 0 |
    im = (im * 255).astype(np.uint8)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    laplacian = cv2.Laplacian(gray, cv2.CV_64F)

    return laplacian

def sobel_x_derivatives_dim_reduction(im):
    # Sobel operators is a joint Gausssian smoothing plus differentiation operation,
    # so it is more resistant to noise.
    # | -1 0 +1 |
    # | -2 0 +2 |
    # | -1 0 +1 |
    im = (im * 255).astype(np.uint8)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=5)

    return sobelx

def sobel_y_derivatives_dim_reduction(im):
    # Sobel operators is a joint Gausssian smoothing plus differentiation operation,
    # so it is more resistant to noise.
    # | +1 +2 +1 |
    # | 0 0 0 |
    # | -1 -2 -1 |
    im = (im * 255).astype(np.uint8)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=5)
    return sobely

def scharr_x_derivatives_dim_reduction(im):
    # More accurate
    # | -3 0 +3 |
    # | -10 0 +10 |
    # | -3 0 +3 |
    im = (im * 255).astype(np.uint8)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    scharrx = cv2.Scharr(gray, cv2.CV_64F, 1, 0)

    return scharrx

def scharr_y_derivatives_dim_reduction(im):
    # More accurate
    # | +3 +10 +3 |
    # | 0 0 0 |
    # | -3 -10 -3 |
    im = (im * 255).astype(np.uint8)
    gray = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)

    scharry = cv2.Scharr(gray, cv2.CV_64F, 0, 1)

    return scharry

```

5.1.4.4. Bilateral filter

Most of the filters that were presented tend to blur edges as they are some kind of a weighted average of pixel values in the neighborhood of the pixel, such as in the case of LPF and Gaussian. The weights will be proportional to the distance from the pixel in Gaussian filtering. Bilateral filtering, on the other hand, adds to these spatial weights that look at the pixel values. This method gives the weights of the pixels a value according to how much they are close in colors to the center pixel. The two weighting systems allow having a Gaussian blurring while preserving the edges.

We implemented this method by using the OpenCV package with a signal parameter controlling the diameter, the color sigma and the distance sigma (in this order).

```
import cv2

def bilateral_filter_dim_reduction(im, factor):
    new_im = cv2.bilateralFilter(im, factor, factor * 2, factor / 2)

    return new_im
```

Figure 26 illustrates the bilateral filtering with factor equals 5 (meaning kernel size = 5, color sigma = 10 and distance color = 5) on the famous picture of Lena. Here we can see the blurring effect while keeping the edges of the picture.



Figure 26: Performing Bilateral Filtering on an image. On the left is the original image. The right image is the filtered image with factor equals to 5.

5.1.5. Edge Detection

Here we used the popular Canny edge detection algorithm that was developed by John F. Canny [44] in 1986. In this algorithm, we first reduce noise by using a Gaussian filter so it won't affect our result. Then the algorithm finds the gradient of each pixel using the Sobel kernel in both horizontal and vertical directions. From these 2 kernels G_x and G_y we calculate the angle and the magnitude of the gradient:

$$(23) \quad \begin{aligned} G_{\text{magnitude}} &= \sqrt{G_x^2 + G_y^2} \\ G_{\text{Angle}}(x, y) &= \tan^{-1} \left(\frac{G_x}{G_y} \right) \end{aligned}$$

Pixels with gradient magnitude lower than a minimum threshold will be removed and those higher than a maximum threshold will be marked as edges. Pixels that are on a route of 2 edge points and the direction of the gradient will mark as edges as well.

In our implementation, we used the OpenCV python package which takes a parameter lower and upper value. As these values are difficult to know because they are different from image to image and it also becomes a function in two dimensions, we work with the median of the image to determine these values and the external parameter sigma to control its sensitivity.

```
import cv2
import numpy as np

def canny_edge_detection_dim_reduction(im, sigma):
    x = np.uint8((im*255).astype(int))
    v = np.median(x)
    lower = int(max(0, (1.0 - sigma) * v))
    upper = int(min(255, (1.0 + sigma) * v))
    edged = cv2.Canny(x, lower, upper)
    return edged
```

In figure 27 we can see different values of sigma and the output using the Canny edge detection algorithm.

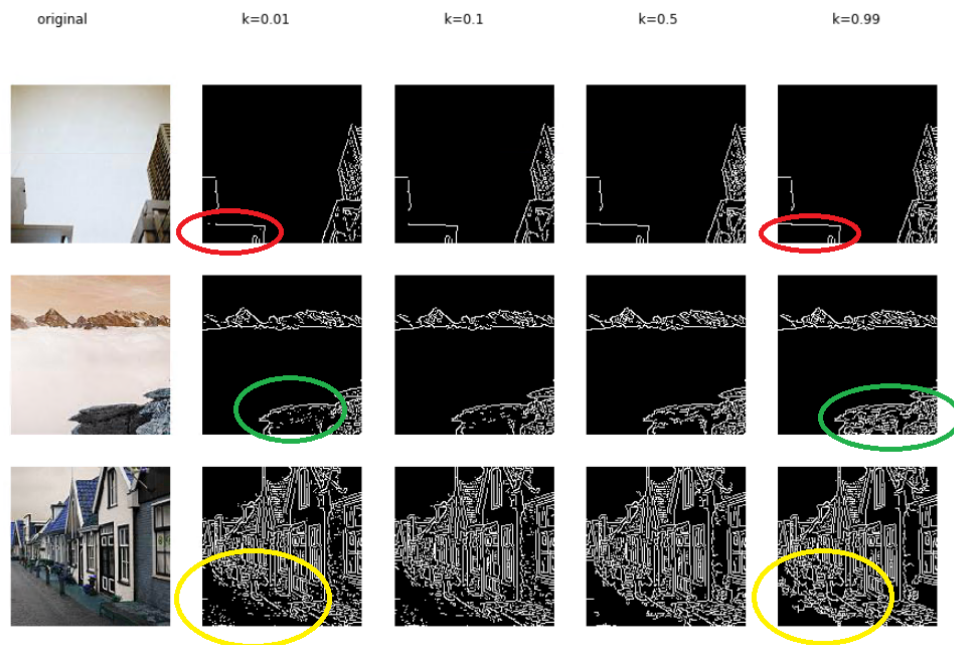


Figure 27: Performing Canny edge detection on an image with a given factor value representing the sensitivity parameter. On the left is the original image. The next images are Canny edge detection filtered images. The images are taken from the INTEL dataset

5.2. Ensemble Defence Methods

So far, we have shown several methods of dimensionality reduction for adversarial defending separately. In this section, we try to combine predictions of several methods to get a better classification, a well-known approach called an *ensemble*. As far we know, no such method has been tested for dimensionality reduction and with stacking approach as we will introduce later.

5.2.1. What is Ensemble

Ensemble learning is a machine learning paradigm where multiple learning algorithms, sometimes called “weak learners”, are trained to solve the same problem and their results are combined to obtain better predictive performance than could be obtained from any of the models alone.

Several major kinds of ensemble meta algorithms aim at combining weak learners are:

- Bagging
- Boosting
- Stacking

Bagging stands for Bootstrap Aggregation. Bootstrapping is a process of resampling the training set to reduce overfitting and decrease the variance. Every model in the Bagging process is trained separately and in parallel while their answers are combined in a deterministic averaging process.

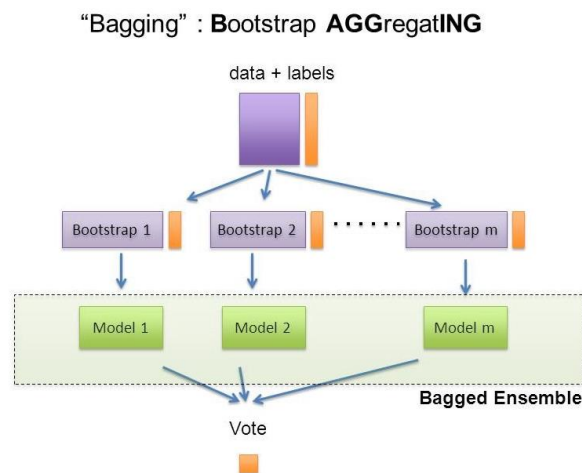


Figure 28: The Bagging process. Given a set of labeled data we are resampling it on each stage to create a new model. All the answers are combined with a weighted averaging.

Boosting learns the model sequentially in an adaptive manner. Starting from a base model, boosting try to improve on each step by creating a new model based on the errors of the previous one.

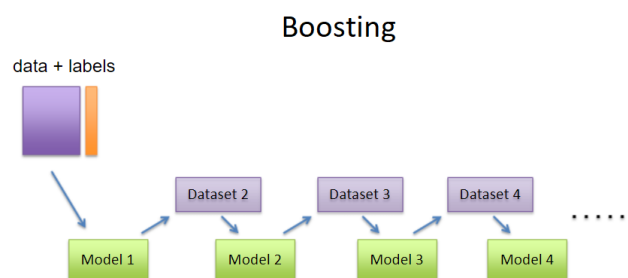


Figure 29: The boosting process. Given a set of labeled data we train a model such that in each step the new model is based on the previous one and its errors.

Stacking learns several different models in parallel and combines them by training a meta-model to output a prediction based on the weak learners' predictions.

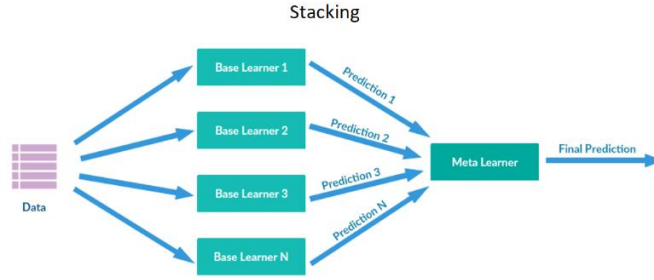


Figure 30: The stacking process. Given a set of labeled data, we train some different models in parallel and their predictions are inputs for a meta learner to predict.

Bagging and boosting don't seem like methods that can be suitable for our use case because of the need for a large amount of image in resampling or because we have only a set of different models predictions. On the other hand, stacking seems like a choice that can give value.

We chose to work with two types of stacking - simple voting among the predictions and give them to a meta-learner.

5.2.2. Voting

The simplest way for working with the results of the models is to have a vote among all predictions. This way, we get as input the predicted class of each model on each image. For each image, we select the class with the most votes.

Suppose we have a set of N models h_1, \dots, h_N and an instance input x_i . Then the classification of the meta-model g will be

$$(24) \quad g(x_i) = \text{mode}\{h_1(x_i), \dots, h_N(x_i)\}$$

A noteworthy disadvantage of this method is that all models have a similar impact so a plurality of weak learners in a given area of the input space will bias the result if they have a majority there (or in other words - the majority is not always right). To overcome this difficulty, we examined the stacking approach.

5.2.3. Stacking with Meta-Learner

Stacking takes the prediction of each model and combined them to compute a more accurate prediction. Let h_1, \dots, h_N be a set of N models and x_i an input point. Then the prediction of the meta learner g is

$$(25) \quad g(x_i) = \sum_{j=1}^N \beta_j h_j(x_i)$$

where the optimum $\{\beta_1, \dots, \beta_N\}$ are the solution for the least-squares optimization problem:

$$(26) \quad \beta^* = \underset{\beta}{\operatorname{argmin}} \sum_{j=1}^N \left(y_i - \sum_{j=1}^N \beta_j h_j(x_i) \right)^2$$

In this version, we gave the meta-model only the prediction. To do it more efficient we can give it the probability of the predicted class or all the prediction vector in a similar way.

5.3. Experiment Setup

We design an experiment with image dimensionality reduction transformations that alter the structure of perturbation and hopefully will raise the ability of the neural network to defend against such an attack without damage the input quality. In this section we will give a full description of neural network architecture and implementation, the datasets and attack methods that we will use and the approach of measuring the success ratio.

5.3.1. Metrics

There are several ways to evaluate the performance of machine learning systems, including accuracy, precision, recall and f1 score. Because in our case there is no preference for FP over FN we chose to work with accuracy:

$$(27) \text{ accuracy} = \frac{\sum_{i=1}^n \mathbb{1}[h(x_i) = y_i]}{N}$$

Accuracy measures among all data points the portion of true classification. This is the most general evaluation metric to look for. A more accurate metric in the case of an adversary may be the misclassification success rate or the fooling ratio which measures among only the correct classified image how many instances the adversary was able to fool.

$$(28) \text{ Fooling_ratio} = \frac{\sum_{i=1}^n \mathbb{1}[h(\tilde{x}_i) \neq y_i \text{ and } h(x_i) = y_i]}{\sum_{i=1}^n \mathbb{1}[h(x_i) = y_i]}$$

where x_1, \dots, x_n are the input images, $\tilde{x}_1, \dots, \tilde{x}_n$ are the adversary images and $h(x)$ is the hypothesis represented by the model.

So far we have only measured the model classification performance but another thing to be concerned about when reducing the dimensionality is the input quality. To do so we will use the PSNR to evaluate how much we damaged the image. PSNR uses the mean square error (MSE) of the low dimensional input from the original input image.

$$(29) \quad \begin{aligned} \text{MSE} &= \frac{1}{N \cdot M} \sum_{i=1}^N \sum_{j=1}^M (x(i, j) - x_{low_dim}(i, j))^2 \\ \text{PSNR} &= 20 \cdot \log_{10} \left(\frac{\text{MAX-PIXEL}}{\sqrt{\text{MSE}}} \right) \end{aligned}$$

5.3.2. Datasets

All the experiments in this paper were done with convolutional neural networks on two image datasets: the CIFAR-10 and the INTEL datasets.

The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 different classes, and with 6000 images per class. The classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. We used 25,000 images for training, 25,000 as validation (and for the ensemble stage) and 10,000 for testing.

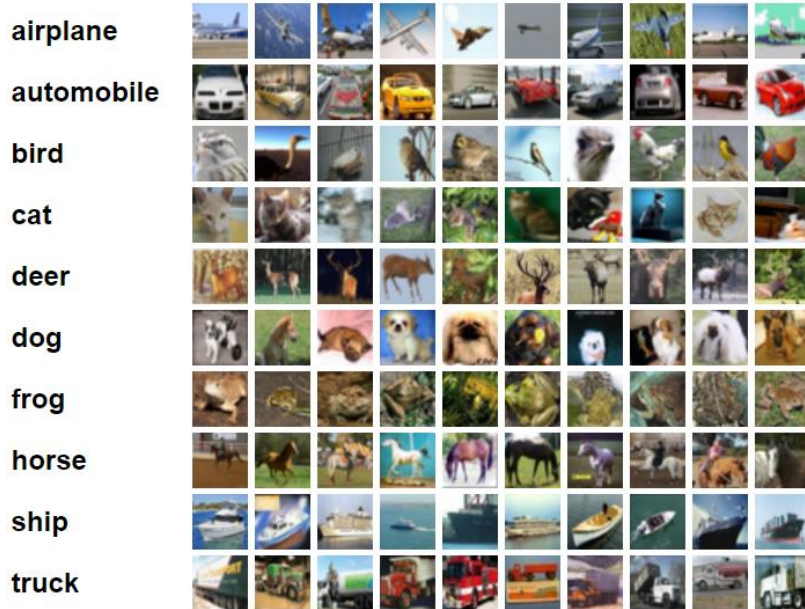


Figure 31: The CIFAR-10 dataset consists of 10 different classes. Here we can see 10 random examples for each of these classes.

The INTEL dataset [45] was taken from the open datasets center of Kaggle. It consists of 17,034 150x150 images in 6 categories: buildings, forest, glacier, mountain, sea, and street. The train includes 6813 images, the validation 6814 images (for the ensemble stage as well) and the test set has 3407 images.

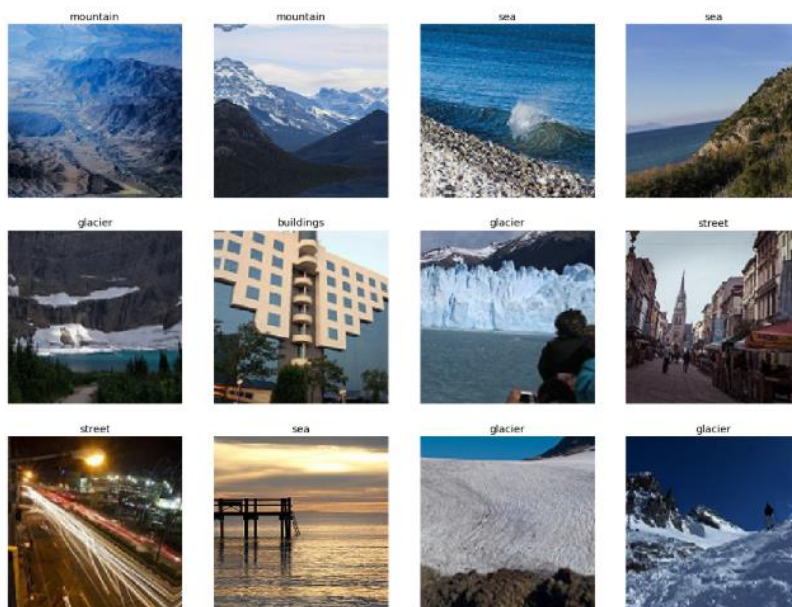


Figure 32: The INTEL dataset consists of 6 different classes. Here we can see some random examples for each of these classes.

5.3.3. Network Architecture

We used the Keras implementation of a convolutional neural network with 4 Conv-Pooling blocks, having Conv filter depth of 32, 64, 128 and 256 respectively. The Conv filter size is 3x3 and the Pooling filter has a size of 2x2 with rectified linear units (ReLUs) as activation functions.

This is followed by a Flatten layer and a Dropout regularization layer. Finally, we have two blocks of fully connected layers with size 512 and 128 respectively, that feeds into a softmax output layer with the number of classes.

```
from keras.models import Sequential
from keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout

def create_model(input_size, num_classes):
    """
    Create a standard classification model with Keras
    """
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1',
                    / input_shape=input_size))
    model.add(MaxPooling2D((2, 2), name='maxpool_1'))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same', name='conv_2'))
    model.add(MaxPooling2D((2, 2), name='maxpool_2'))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_3'))
    model.add(MaxPooling2D((2, 2), name='maxpool_3'))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same', name='conv_4'))
    model.add(MaxPooling2D((2, 2), name='maxpool_4'))
    model.add(Flatten())
    model.add(Dropout(0.2))
    model.add(Dense(512, activation='relu', name='dense_1'))
    model.add(Dense(128, activation='relu', name='dense_2'))
    model.add(Dense(num_classes, activation='softmax', name='output'))

    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    model.summary()

    return model
```

5.3.4. Adversarial Examples

We evaluated the performance of different dimensionality reduction methods using the implementation of FGSM [16] provided by the Cleverhans library [46]. Several values of epsilon in the range of $[0, 0.5]$ were examined.

```
from cleverhans.utils_keras import KerasModelWrapper
from cleverhans.attacks import FastGradientMethod
from keras import backend
from skimage.io import imread_collection
import imageio
import sys, os

epsilons = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,
            0.15, 0.2, 0.25, 0.3, 0.4, 0.5]

def create_adversarial_directory(input_path, output_path, model, eps=0.1):
    # delete if exists
    make_new_dir(output_path)

    total_counter = sum([len(files) for r, d, files in os.walk(input_dir)])
    counter = 0

    for subdir, dirs, files in os.walk(input_path):

        for directory in dirs:
            current_dir = os.path.join(subdir, directory)
            make_new_dir(current_dir.replace(input_path, output_path))
            sys.stdout.write("\r{:%}".format(round(float(counter)/total_counter*100, 2)))
            sys.stdout.flush()

            # read all images
            col = imread_collection(current_dir + r'\*')
            images = [col[i] for i in range(len(col))]
            images = np.asarray(images)

            # adv
            sess = backend.get_session()
            wrap = KerasModelWrapper(original_model)
            fgsm = FastGradientMethod(wrap, sess=sess)

            fgsm_params = {'eps': eps,
                           'clip_min': 0.,
                           'clip_max': 1.}

            adv = fgsm.generate_np(images / 255, **fgsm_params)

            # write to file
            for i, image in enumerate(adv):
                original_file_path = col.files[i]
                new_file_path = original_file_path.replace(input_path, output_path)
                imageio.imwrite(new_file_path, (image * 255).astype(int))
                counter = counter + len(col.files)

# Create adversarial
input_dir = r'original\test'
output_dir = r'original\test_adv'

for epsilon in epsilons:
    current_output_dir = os.path.join(output_dir, str(epsilon))
    make_new_dir(current_output_dir)
    print('Current Epsilon Value = ' + str(epsilon))
    create_adversarial_directory(input_dir, current_output_dir, original_model, epsilon)
```

5.3.5. Experimental Stages

We trained the model for 200 epochs with an early stopping option using categorical cross-entropy loss and used the Adam optimizer.

```
import keras, pickle, os
from keras.callbacks import ModelCheckpoint, EarlyStopping

class AccuracyHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.acc = []
        self.val_acc = []

    def on_epoch_end(self, batch, logs={}):
        self.acc.append(logs.get('acc'))
        self.val_acc.append(logs.get('val_acc'))

    def to_file(self, model_file_name):
        with open(model_file_name, 'wb') as output:
            pickle.dump((self.acc, self.val_acc), output, pickle.HIGHEST_PROTOCOL)

    def from_file(self, model_file_name):
        with open(model_file_name, 'rb') as history_input:
            self.acc, self.val_acc = pickle.load(history_input)

def fit_model(model, train_generator, validation_generator, new_run=True,
              model_file_name='my_model', early_stopping_patience=30, epochs=200,
              verbose=1):
    if (new_run) or not os.path.isfile(model_file_name + '_model.h5'):
        history = AccuracyHistory()
        early_stopping_monitor = EarlyStopping(patience=early_stopping_patience)

        mcp_save = ModelCheckpoint('.mdl_wts.hdf5', save_best_only=True,
                                   monitor='val_loss', mode='min')

        STEP_SIZE_TRAIN = train_generator.n // train_generator.batch_size
        STEP_SIZE_VALID = validation_generator.n // validation_generator.batch_size
        fit_history = model.fit_generator(train_generator,
                                         steps_per_epoch=STEP_SIZE_TRAIN,
                                         epochs=epochs, verbose=verbose,
                                         validation_data=validation_generator,
                                         validation_steps=STEP_SIZE_VALID,
                                         callbacks=[early_stopping_monitor, mcp_save,
                                                  history])

        plot_compare(fit_history)
        model.load_weights(filepath='.mdl_wts.hdf5')
        model.save(model_file_name + '_model.h5')
        history.to_file(model_file_name + '_history.pkl')

    else:
        model = load_model(model_file_name + '_model.h5')
        history = AccuracyHistory()
        history.from_file(model_file_name + '_history.pkl')
        plot_compare(history)

    return model, history

#create and compile model
original_model = create_model(input_size=
                              num_classes=len(original_train_generator.class_indices))

#train the network
original_model, original_history = fit_model(original_model, original_train_generator,
                                             original_validation_generator,
                                             new_run = False,
                                             model_file_name=dataset_name+'_original',
                                             epochs =200 )
```

To test the model we simulate it on the original test set files and the adversarial files for each epsilon value. We repeated this process for each of the dimensionality reduction methods that were mentioned with different parameters.

```

def test_model(model, test_generator, silent=False, file_name=''):
    """
    Get the X_test and y_testb (as categorical) and return predcitions, classes, loss,
    accuracy, fl-score, percision, recall, confusion-matrix and classification report.
    """
    if not os.path.isfile(file_name + '_predict_results.pkl'):
        score = model.evaluate_generator(test_generator, steps=test_generator.n //
                                       test_generator.batch_size, verbose=0)

        if not silent:
            print('Test loss:', score[0])
            print('Test accuracy:', score[1])

        pred = model.predict_generator(test_generator, steps=test_generator.n //
                                      test_generator.batch_size)
        pred_classes = np.argmax(pred, axis=1)
        classes = test_generator.classes

        r = classes == pred_classes
        print('Accuracy:')
        print(sum(r) / len(r))

        cm = confusion_matrix(classes, pred_classes)
        cr_str = classification_report(classes, pred_classes, output_dict=False)
        if not silent:
            print(cm)
            print('-' * 50)
            print(cr_str)

        cr = classification_report(classes, pred_classes, output_dict=True)
        test_result = {'loss': score[0],
                      'accuracy': score[1],
                      'predictions': pred,
                      'f1': cr['weighted avg']['f1-score'],
                      'precision': cr['weighted avg']['precision'],
                      'recall': cr['weighted avg']['recall'],
                      'confusion matrix': cm,
                      'classification report': cr_str}
        with open(file_name + '_predict_results.pkl', 'wb') as output:
            pickle.dump(test_result, output, pickle.HIGHEST_PROTOCOL)
    else:
        with open(file_name + '_predict_results.pkl', 'rb') as input:
            test_result = pickle.load(input)
    return test_result

# test the model
test_result = test_model(original_model, original_test_generator, silent=False,
                        file_name=dataset_name+'_original_test')

```

For the ensemble stage, we created 2 types of dataframes containing all the results of all models on all samples. The first type will be the size of the number of models and the number of samples when each cell [i,j] contains the predicted class of image i with model j. In the second type of matrix the prediction of each model j on the image i will be the distribution vector with the size of the number of classes. That is, each column that previously contained a single number, that is, the class number, will now be split into the number of classes with the probability for each class. These matrices will be later used by the meta-learner as input for the training and testing and all our adversarial results.

```

from keras.datasets import cifar10
import glob, itertools

epsilons = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1,
            0.15, 0.2, 0.25, 0.3, 0.4, 0.5]
dataset_name = cifar10
train_generator, validation_generator, test_generator = load_dataset_generators(
                                                    dataset = 'original')

# 1. get true classes
#-----
validation_classes = validation_generator.classes
test_classes = test_generator.classes

# 2. find all models
#-----
model_files = glob.glob(dataset_name+'*_model.h5')
def get_dir_name_from_model_name(model_name):
    return model_name.replace(dataset_name+'_', '').replace("_model.h5", "")
models = list(map(get_dir_name_from_model_name, model_files))

# 3. define dataframe with all class results for - validation, test and adv
#-----
columns = ['true_value']+models
validation_models_predict_class_df = pd.DataFrame(columns = columns)
# validation
test_models_predict_class_df = pd.DataFrame(columns = columns)
# test
test_adv_models_predict_class_df_dict =
    {eps : pd.DataFrame(columns = columns) for eps in epsilons}      # test adv

# add a true_value column
validation_models_predict_class_df['true_value'] = validation_classes
# validation
test_models_predict_class_df['true_value'] = test_classes
# test
for eps in epsilons:
# test adv
    test_adv_models_predict_class_df_dict[eps]['true_value'] = test_classes

# 4. do the same for the probabilityresult dataframe
#-----
classes = list(train_generator.class_indices.keys())
columns = ['true_value'] + list(
    itertools.chain.from_iterable(
        [[x + '___' + model for x in classes] for model in models]))

validation_models_predict_prob_df = pd.DataFrame(columns=columns) # validation
test_models_predict_prob_df = pd.DataFrame(columns=columns) # test
test_adv_models_predict_prob_df_dict =
    {eps: pd.DataFrame(columns=columns) for eps in epsilons} # test adv

validation_models_predict_prob_df['true_value'] = validation_classes # validation
test_models_predict_prob_df['true_value'] = test_classes # test

```

```

validation_models_predict_class_df['true_value'] = validation_classes # validation
test_models_predict_class_df['true_value'] = test_classes # test
for eps in epsilons: # test adv
    test_adv_models_predict_prob_df_dict[eps]['true_value'] = test_classes

# 5. fill all dataframes
#-----
for i, model_file in enumerate(model_files, start=0):
    folder = models[i]
    print(f'Progress = {str(i + 1)}/{len(model_files)}, folder = {folder}')
    print(model_file)
    # load model
    model = load_model(model_file)

    # read data sets
    train_generator, validation_generator, test_generator =
        load_dataset_generators(dataset=folder)

    # run model on validation and test
    validation_result = test_model(model, validation_generator, silent=True,
                                   file_name=dataset_name + '_validation_ensemble_' +
                                   folder)
    test_result = test_model(model, test_generator, silent=True, file_name=dataset_name
                              + '_test_ensemble_' + folder)

    # get predictions on validation and test
    validation_pred = validation_result['predictions']
    test_pred = test_result['predictions']

    # get pred classes
    validation_pred_classes = np.argmax(validation_pred, axis=1)
    test_pred_classes = np.argmax(test_pred, axis=1)

    # add to classes df
    validation_models_predict_class_df[folder] = validation_pred_classes
    test_models_predict_class_df[folder] = test_pred_classes

    # add to predict prob df
    columns = [x + '___' + folder for x in classes]
    validation_models_predict_prob_df.loc[:, columns] = validation_pred
    test_models_predict_prob_df.loc[:, columns] = test_pred

    for eps in epsilons:
        current_directory = folder + "\\test_adv\\" + str(eps)

        adv_generator = load_dataset_generator(current_directory)

        adv_result = test_model(model, adv_generator, silent=False,
                                file_name=dataset_name + '_test_adv_' + folder +
                                '_eps_' + str(eps) + '_adv')

        test_adv_pred = adv_result['predictions']

        test_adv_pred_classes = np.argmax(test_adv_pred, axis=1)

        test_adv_models_predict_class_df_dict[eps][folder] = test_adv_pred_classes
        test_adv_models_predict_prob_df_dict[eps].loc[:, columns] = test_adv_pred

```

Let's see how our data looks like. Below is the first type of matrix that contains for each image and model the predicted class. For example, here the model `original_bilateral_filter_10` (meaning, the factor value is 10) predicted that image 2 belongs to class 3 where the true value should be 0.

validation_models_predict_class_df							
	true_value	original_bilateral_filter_10	original_bilateral_filter_2	original_bilateral_filter_3	original_bilateral_filter_4	original_bilateral_filter_5	original_bilateral_filter_10
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
2	0	3	7	3	0	3	3
3	0	8	8	8	0	8	8
4	0	0	0	0	0	0	0
...
24995	9	9	9	9	9	9	9
24996	9	9	9	9	9	9	9
24997	9	9	9	9	9	9	9
24998	9	9	9	9	9	9	9
24999	9	9	9	9	9	9	9

25000 rows × 61 columns

The second type of matrix will expand each column for all classes' possibilities. For example, for the `original_bilateral_filter_10` model and 10 possible different classes in the CIFAR-10 case, we will split the column into 10 different columns according to the probability of the class. And so, below we can see that the cell `[2, 0__original_bilateral_filter_10]` tells that the `bilateral_10` model predicted that the likelihood of image 2 to belong to class 0 is 0.060611.

validation_models_predict_prob_df							
	true_value	0__original_bilateral_filter_10	1__original_bilateral_filter_10	2__original_bilateral_filter_10	3__original_bilateral_filter_10	4__original_bilateral_filter_10	...
0	0	0.763911	0.001489	0.145023	0.014805	1.97124e-05	...
1	0	0.987272	0.000043	0.001850	0.000035	6.30311e-06	...
2	0	0.060611	0.002237	0.029430	0.589650	4.73485e-05	...
3	0	0.323592	0.029280	0.000246	0.000017	7.49374e-06	...
4	0	0.663715	0.000689	0.203205	0.012648	9.44434e-05	...
...
24995	9	0.000064	0.006278	0.000001	0.000002	1.08063e-06	...
24996	9	0.000071	0.001531	0.000003	0.000005	1.96792e-06	...
24997	9	0.004519	0.003884	0.001199	0.011742	1.16061e-05	...
24998	9	0.000029	0.203127	0.000002	0.000037	4.48334e-06	...
24999	9	0.000219	0.074475	0.000003	0.000130	5.03429e-06	...

25000 rows × 601 columns

Now to implement ensemble voting, we wish to look at the first type of matrix on each row and select the maximum class and this will be the meta-learner prediction to every image. From this, we can easily find the percentage of accuracy by comparing it to the true value.


```

# Voting
#-----

# validation
voting_validation_predicted = validation_models_predict_class_df.drop(
    ['true_value'], axis=1).mode(axis=1)[0].astype('int64')
validation_score = sum(voting_validation_predicted ==
    validation_models_predict_class_df['true_value'])
    /len(validation_models_predict_class_df)

ensemble_results_table_total.at['validation', 'ensemble_voting'] = validation_score

# test
voting_test_predicted = test_models_predict_class_df.drop(
    ['true_value'], axis=1).mode(axis=1)[0].astype('int64')
test_score = sum(voting_test_predicted ==
    test_models_predict_class_df['true_value'])
    /len(test_models_predict_class_df)

ensemble_results_table_total.at['test', 'ensemble_voting'] = test_score

ensemble_results_table_total

# adv test
for eps in epsilons:
    voting_adv_predicted = test_adv_models_predict_class_df_dict[eps].drop(
        ['true_value'], axis=1).mode(axis=1)[0].astype('int64')
    adv_score = sum(voting_adv_predicted ==
        test_adv_models_predict_class_df_dict[eps]['true_value'])
        /len(test_adv_models_predict_class_df_dict[eps])

    ensemble_results_table_total.at['adv_epsilon_'+str(eps), 'ensemble_voting'] =
        adv_score

```

For stacking implementation we will have the same experiments twice - one for the predicted class matrix and the other for the probabilities. The code will be the same so we will give it only for the first case. We will organize the data for the model – separation of x and y, shuffling all instances and applying one-hot encoding for the case of using the predicated classes.

```

# Stacking
#-----

train = validation_models_predict_class_df.astype(str)
test = test_models_predict_class_df.astype(str)
test_adv_dict = {k:v.astype(str) for k,v in
test_adv_models_predict_class_df_dict.items()}

# shuffle the data

train=train.iloc[np.random.permutation(len(train))]
train=train.reset_index(drop=True)

test=test.iloc[np.random.permutation(len(test))]
test=test.reset_index(drop=True)

for k,v in test_adv_dict.items():
    test_adv_dict[k] =
test_adv_dict[k].iloc[np.random.permutation(len(test_adv_dict[k]))]
    test_adv_dict[k] = test_adv_dict[k].reset_index(drop=True)

```

```

# separate x and y

train_y = train['true_value']
train_x = train.drop(['true_value'], axis=1)

test_y = test['true_value']
test_x = test.drop(['true_value'], axis=1)

test_adv_dict_x = dict()
test_adv_dict_y = dict()
for k,v in test_adv_dict.items():
    test_adv_dict_y[k] = test_adv_dict[k]['true_value']
    test_adv_dict_x[k] = test_adv_dict[k].drop(['true_value'], axis=1)

# apply one hot encoding of categorical features

for col in train_x.dtypes[train_x.dtypes == 'object'].index:
    for_dummy = pd.Categorical(train_x.pop(col), categories=list(set(train_y)))
    train_x = pd.concat([train_x, pd.get_dummies(for_dummy, prefix=col)], axis=1)

for col in test_x.dtypes[test_x.dtypes == 'object'].index:
    for_dummy = pd.Categorical(test_x.pop(col), categories=list(set(train_y)))
    test_x = pd.concat([test_x, pd.get_dummies(for_dummy, prefix=col)], axis=1)

for k, v in test_adv_dict_x.items():
    for col in test_adv_dict_x[k].dtypes[test_adv_dict_x[k].dtypes == 'object'].index:
        for_dummy = pd.Categorical(test_adv_dict_x[k].pop(col),
categories=list(set(train_y)))
        test_adv_dict_x[k] = pd.concat([test_adv_dict_x[k], pd.get_dummies(for_dummy,
prefix=col)], axis=1)

```

For training and evaluating the meta-learner, we used the H2O Auto-ML [47]. H2O automates the process of training a large selection of candidate models. We've tried several models including DRF (Distributed Random Forest), XGBoost, fully-connected neural network and more. We restricted the algorithm to a 12-hour run (after noticing that additional time has little effect).

```

# run h2o auto ml for model selection
import h2o
from h2o.automl import H2OAutoML

h2o.init()

htrain = h2o.H2OFrame(pd.concat([train_y, train_x], axis=1))
htest = h2o.H2OFrame(test_x)
hadv = {}
for k,v in test_adv_dict_x.items():
    hadv[k] = h2o.H2OFrame(v)

x = htrain.columns
y = 'true_value'
x.remove(y)

htrain[y] = htrain[y].asfactor()

aml = H2OAutoML(max_runtime_secs = 60*60*12, stopping_metric='misclassification',
include_algos = ["GLM", "DeepLearning", "DRF", "XGBoost", "GBM",
"DeepLearning", "StackedEnsemble"], stopping_rounds=0)

aml.train(x=x, y=y, training_frame=htrain)

```

After finding the optimal model in the next step we can evaluate it on the training set test set, and all the adversarial examples.

```
# train

train_y_pred = aml.leader.predict(htrain)
train_y_pred = train_y_pred.as_data_frame()['predict'].astype(str)

ensemble_results_table_total.at['validation', 'ensemble_stacking_class(H2O)'] =
    accuracy_score(train_y, train_y_pred)

# test

test_y_pred = aml.leader.predict(htest)
test_y_pred = test_y_pred.as_data_frame()['predict'].astype(str)

ensemble_results_table_total.at['test', 'ensemble_stacking_class(H2O)'] =
    accuracy_score(test_y, test_y_pred)

# adv

for eps in epsilons:
    print('current eps=' + str(eps))
    test_adv_y_pred = aml.leader.predict(htest_adv[eps])
    test_adv_y_pred = test_adv_y_pred.as_data_frame()['predict'].astype(str)

    ensemble_results_table_total.at
        ['adv_epsilon_' + str(eps), 'ensemble_stacking_class(H2O)'] =
            accuracy_score(test_adv_dict_y[eps], test_adv_y_pred)
```

All the results of our experiments are presented in the next section.

6. Results

In this section, we will show the efficiency of each dimensionality reduction defense mechanism that was mentioned in the previous section. We will measure our performance with the metrics defined in the previous section.

6.1. Dimensionality Reduction Results

We will now look at the results obtained for each dimension reduction method. For each method, we trained a network with a training set that was reduced using this method. In all experiments, we obtained a convergence of the validation set after 15-40 epochs as seen in the example in figure 33.

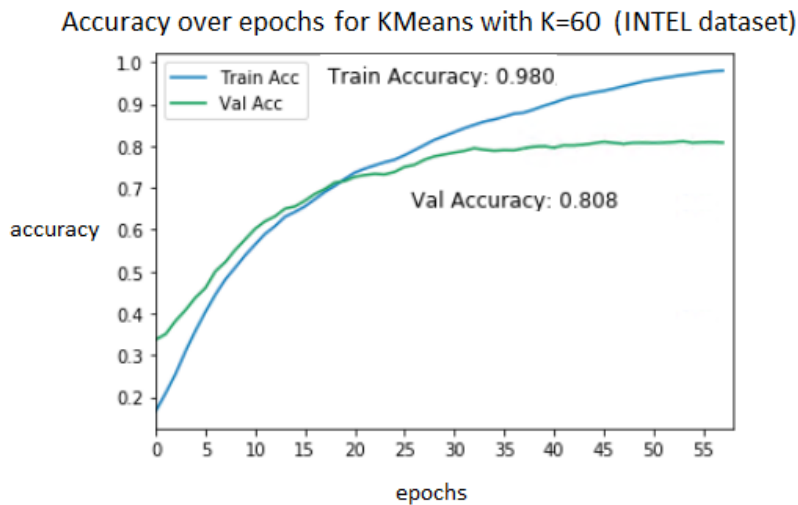


Figure 33: Accuracy results over epochs. Here we see the progress for the accuracy in the case of the INTEL dataset and dimension reduction using the K-means method with parameter $k = 60$. It can be seen that the validation accuracy stabilizes at around 0.8 after the 30th epoch

After training, we ran a prediction for every method and epsilon value on the data, measuring the accuracy, fooling rate and the median PSNR value. The FGSM algorithm for generating adversarial examples used the following epsilon values: [0.01,0.02,0.03,0.04,0.05,0.06,0.06,0.07,0.08,0.09,0.1, 0.15,0.2,0.25,0.3,0.4,0.5]

6.1.1. Image Resize and Rescale

The resizing and rescaling algorithm includes a squeezing parameter. The following values were examined: [1.1,1.2,1.3,1.4,1.5,1.7,2.0,2.5,3.0]. Figure 34 shows the accuracy and the fooling rate for this experiment. We can see that for epsilons values smaller than 0.3-0.4 all the alternative models got better and similar results than the original model on adversarial examples.

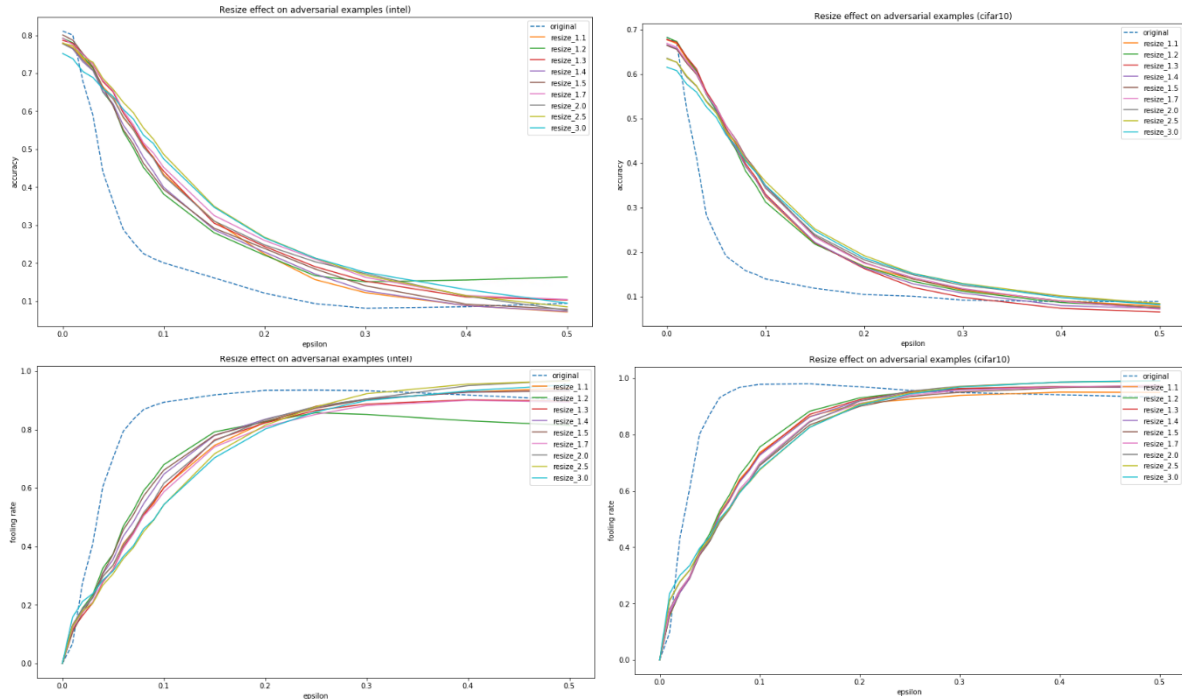


Figure 34: The effect of dimension reduction using the resizing and rescaling method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Figure 35 shows the effect on image quality as a function of the squeezing parameter in PSNR units. Indeed, consistent and slow decay can be observed as the compression factor increases.

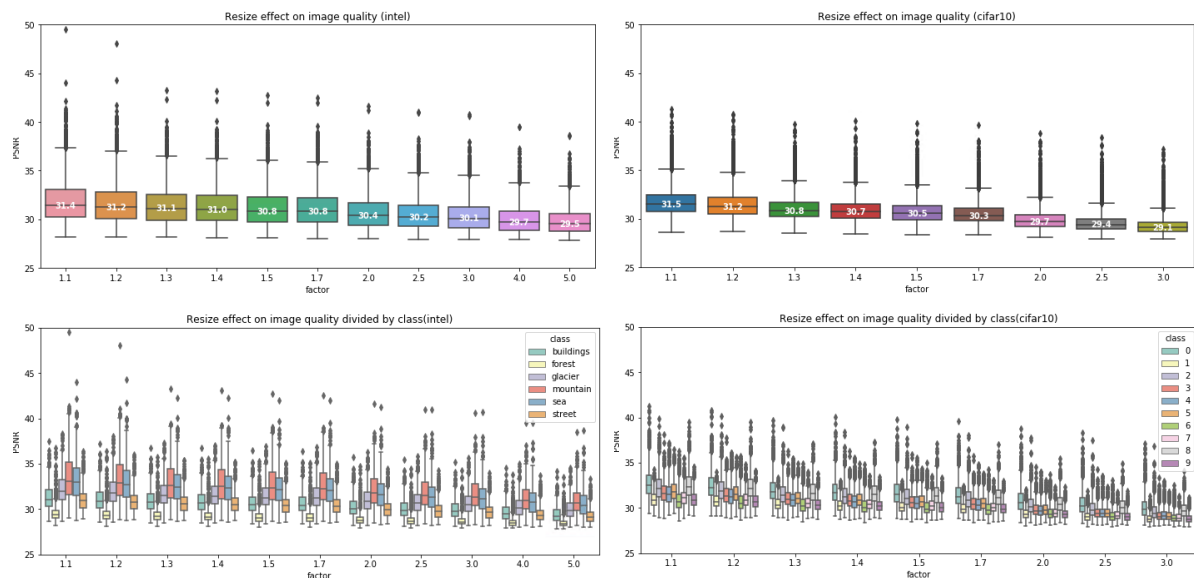


Figure 35: The effect of dimension reduction using the resizing and rescaling method on the quality of the image. In the x axis we can see different sizing factor values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different factor values. The graphs at the bottom are separate for each class.

6.1.2. K-means Color Quantization

For the k-means algorithm, the following k values were examined: [3,5,8,10,15,20,25,30,40,50,60]. As can be seen in figure 36, that shows the accuracy and fooling rate, all the examined k values gave better results than the original model. In many cases, the accuracy improvement was more than 2 or 3 times higher than the original model. As for epsilon equals 0, meaning there is no adversarial attack, all the model acted similar to the original model as they got an accuracy of 78-80% vs 80% in the original model for INTEL dataset and similar in the case of CIFAR-10 except for lower k values (3 and 5).

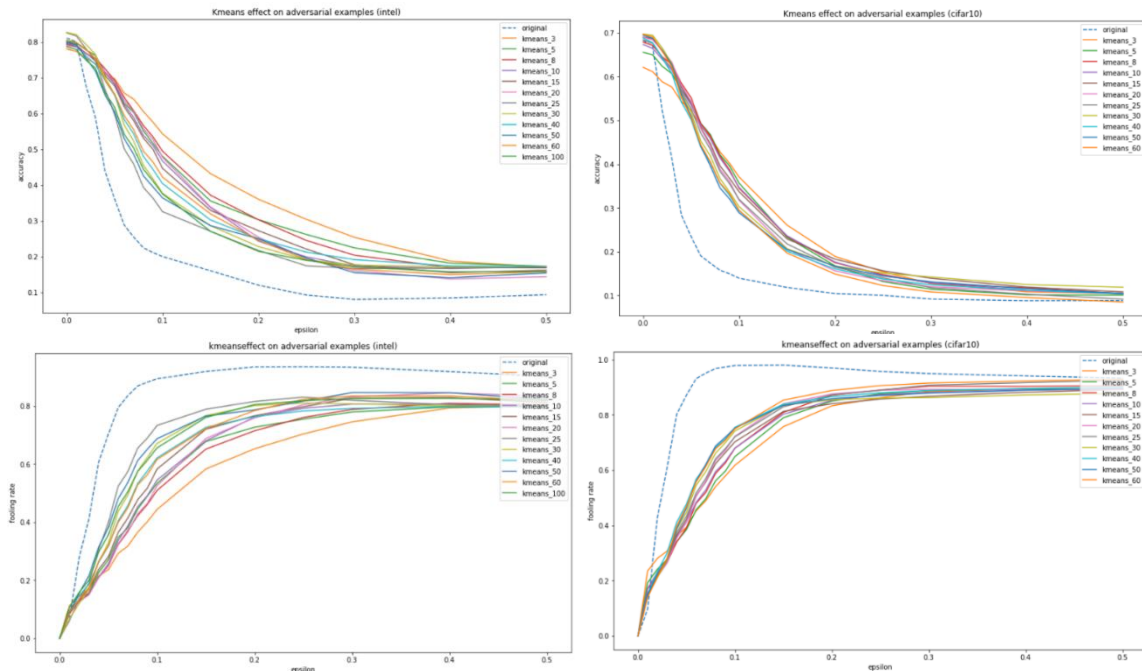


Figure 36: The effect of dimension reduction using the kmeans method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Figure 37 shows the effect of k-means dimensionality reduction on image quality using the PSNR value. As you might assume, the smaller the number of colors (k) in the image, the higher the quality (and the variance of the results).

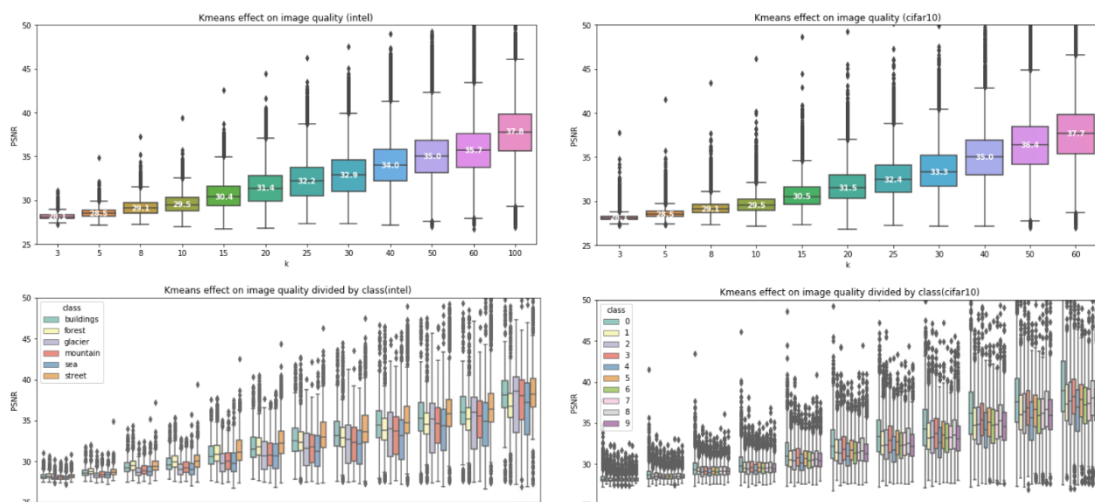


Figure 37: The effect of dimension reduction using the kmeans method on the quality of the image. In the x axis we can see different k values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different k values. The graphs at the bottom are separate for each class.

6.1.3. PCA

Running the PCA algorithm shows an improvement in classification, whereas for the INTEL dataset it is clear that lowering the variance percentage parameter improved the results while for a CIFAR-10 a mixed trend was observed - for the lower values of variance percentage (0.85 and 0.8) it was observed that for small epsilon values the results were lower than other models and got the highest scores for when epsilon passed 0.1 value.

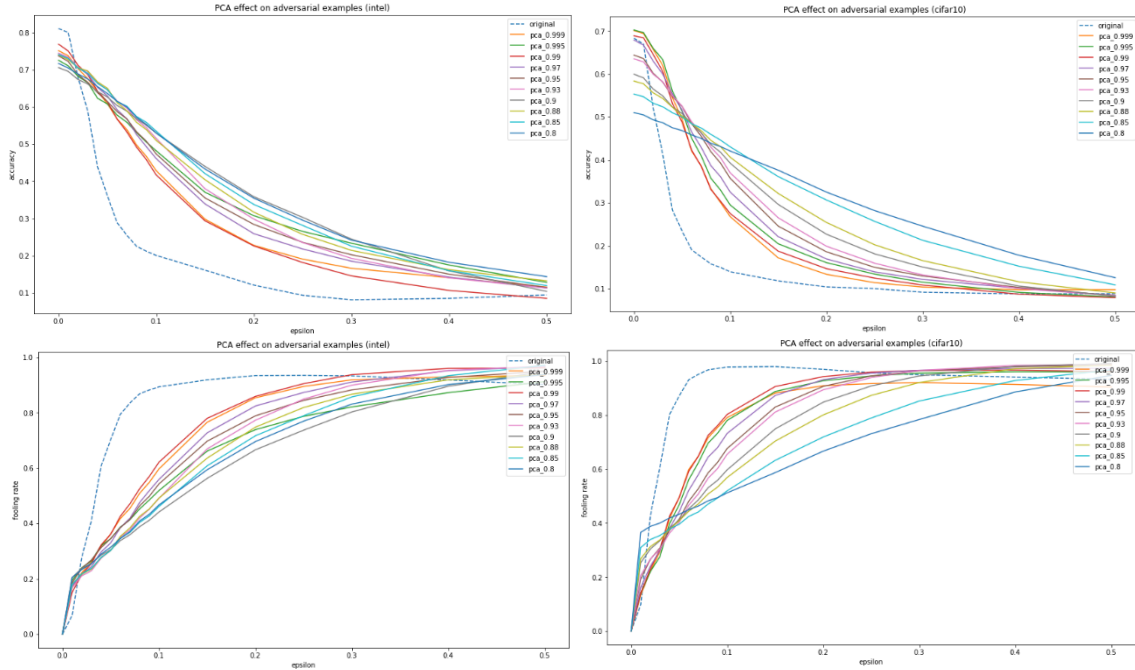


Figure 38: The effect of dimension reduction using the *pca* method on correctly identifying adversarial examples. The *x* axis is the *epsilon* value in the FGSM adversarial method, while the *y* axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Selecting PCA severely damage image quality on small resolution images (such as CIFAR-10) even when taking a high percentage of variance (like 0.999). In contrast, a higher quality image (INTEL) shows that selecting high variance percentage values (>0.99) yields good results but very quickly results in serious damage to image quality when going below this threshold.

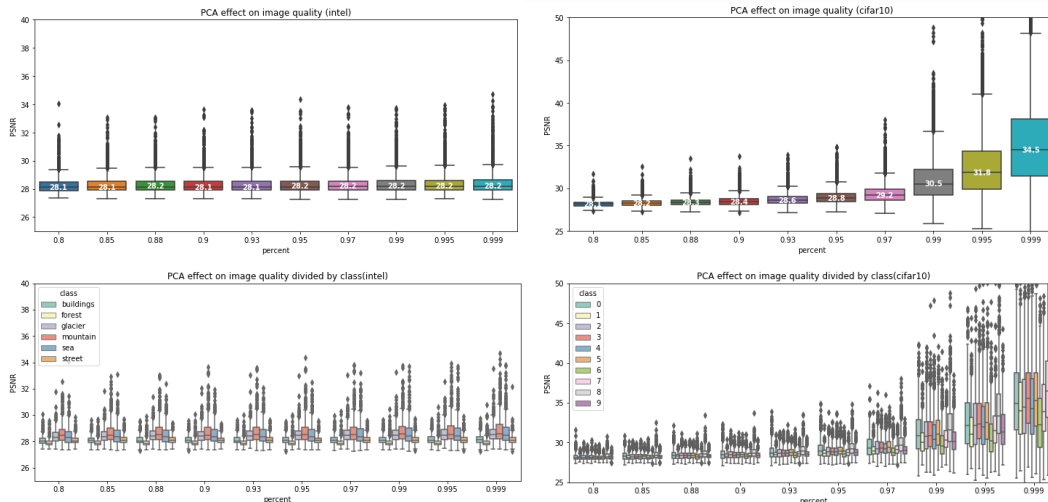


Figure 39: The effect of dimension reduction using the PCA method on the quality of the image. In the *x* axis we can see different *k* values, while the *y* axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different variance percentage values. The graphs at the bottom are separate for each class.

6.1.4. Low Pass Filtering

In the low pass filtering case a similar improvement can be observed for different window size values while epsilon value is below 0.3-0.4. The only exception is a small size of 3 for the convolutional window in the INTEL dataset that performs slightly less, but still better from the original model.

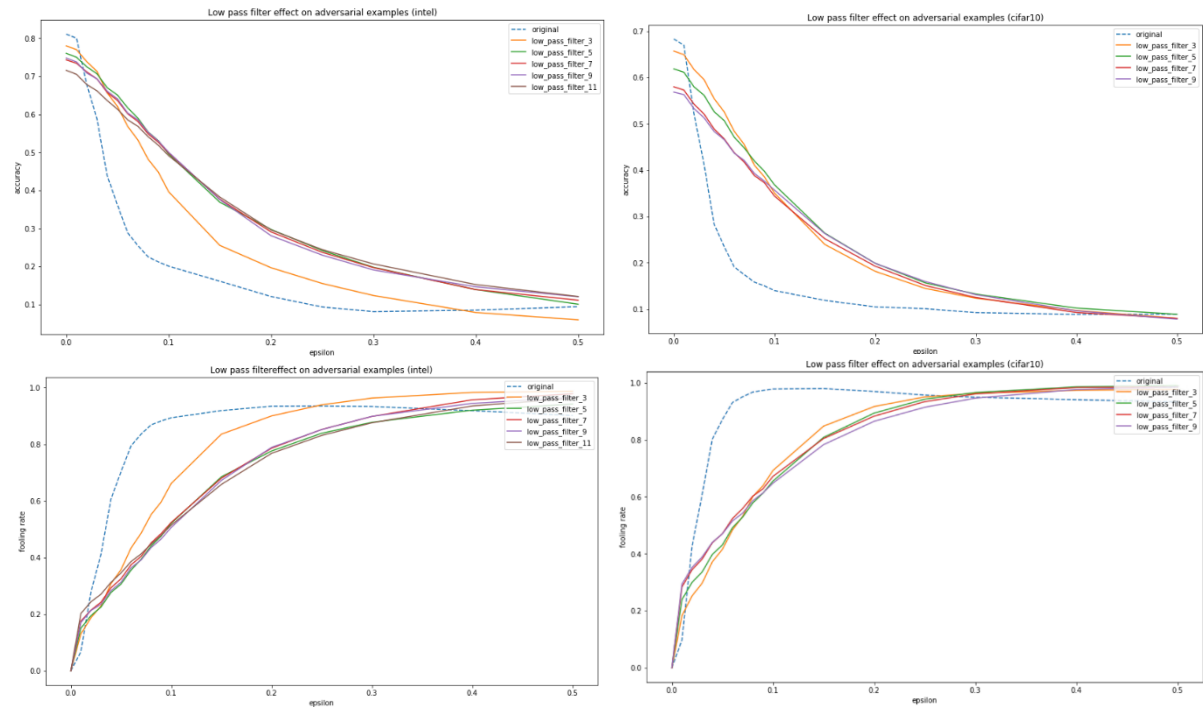


Figure 40: The effect of dimension reduction using the low pass filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Indeed, the image quality in this case stands out significantly over the other cases, as can be seen in figure 41.

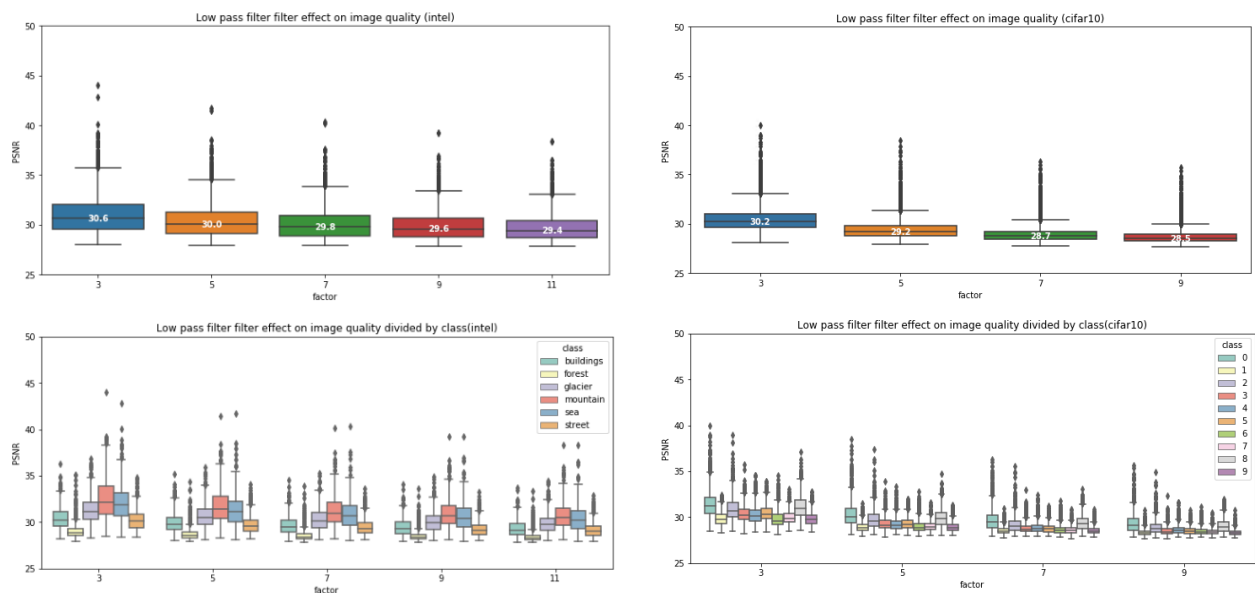


Figure 41: The effect of dimension reduction using the low pass filter method on the quality of the image. In the x axis we can see different factor values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different factor values. The graphs at the bottom are separate for each class.

6.1.5. Gaussian Filtering

By running gaussian filtering we will get better results than the original model as long as epsilon value is less than 0.3-0.4. It is notable that enlarging the window size, meaning more blurring result, we get better classification on adversarial examples.

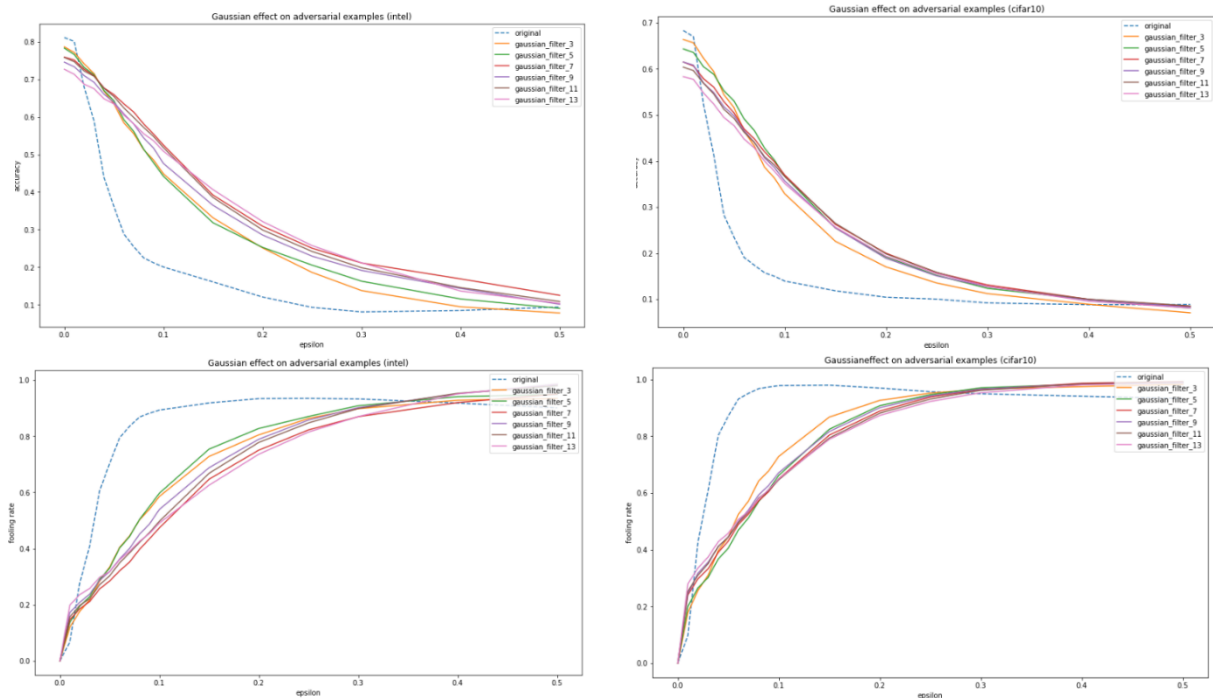


Figure 42: The effect of dimension reduction using the gaussian filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

On the other hand, increasing the blurring will result in obvious damage to image quality.

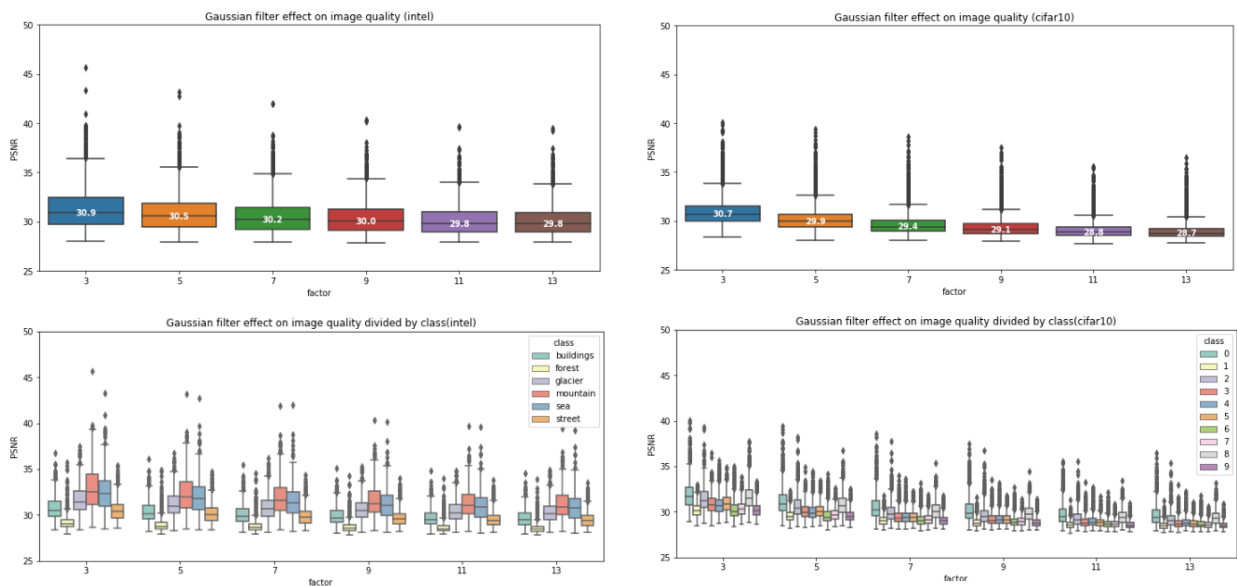


Figure 43: The effect of dimension reduction using the gaussian filter method on the quality of the image. In the x axis we can see different factor values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different factor values. The graphs at the bottom are separate for each class.

6.1.6. Median Filtering

Median filters, which selects existing colors from the image itself, shows an improvement over the original model for epsilon values smaller than 0.3-0.4. The two window sizes that were examined gave similar results.

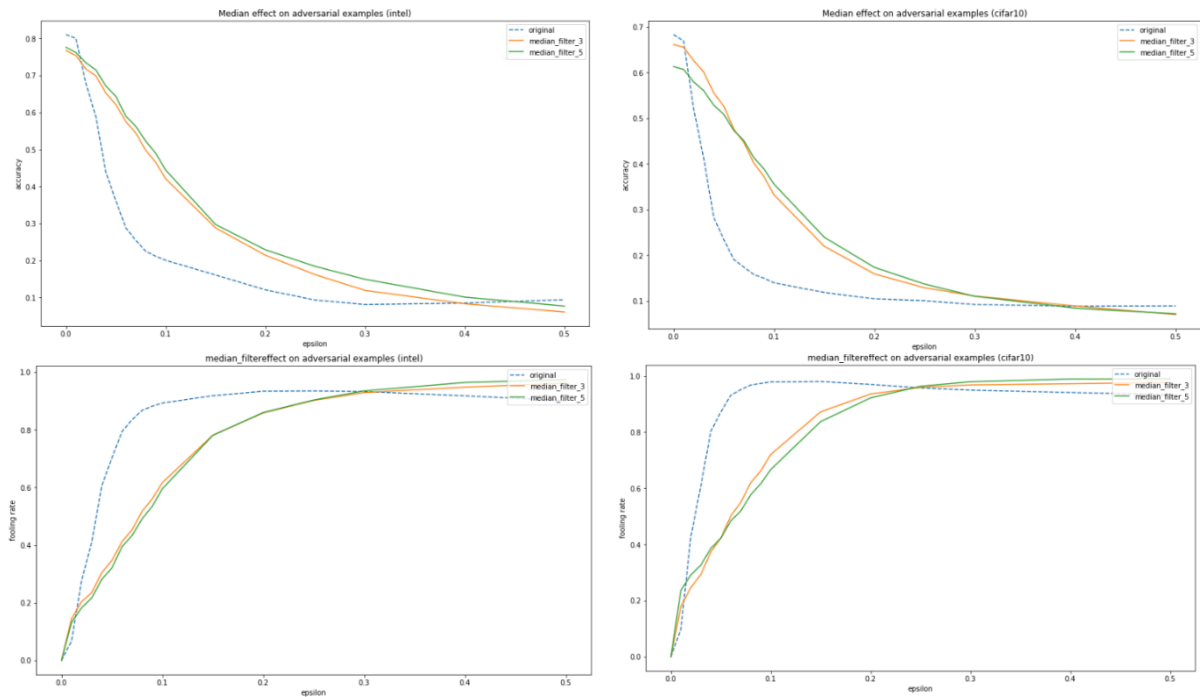


Figure 44: The effect of dimension reduction using the median filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Increasing the window size as expected damaged image quality.

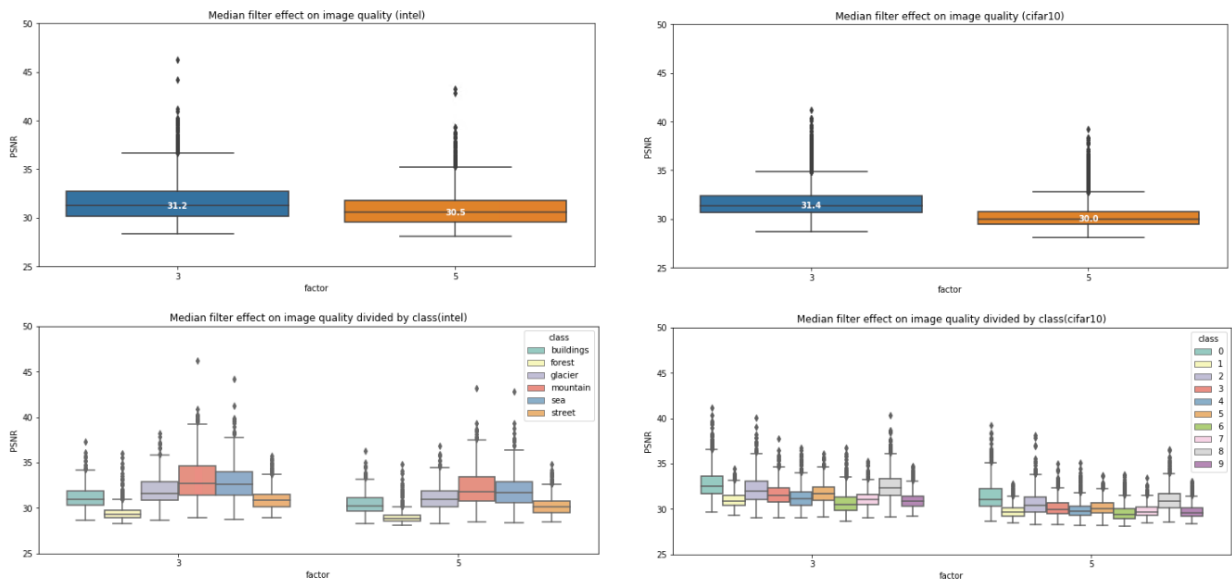


Figure 45: The effect of dimension reduction using the median filter method on the quality of the image. In the x axis we can see different factor values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different factor values. The graphs at the bottom are separate for each class.

6.1.7. Gradient Filtering

Gradient filtering appears to be one of the weakest methods among those that were tested. There seems to be some small improvement in classification accuracy and fooling rate but it only applies for small values of epsilon (lower then $\sim 0.1-0.2$). For very small values the gradient models respond even worse than the original model. This certainly makes sense as the impact of the gradient is to find the map of changes in a certain direction, so the result is quite different from the original image.

In all the experiments Sobel y gave the best results while the laplacian gave the worst results.

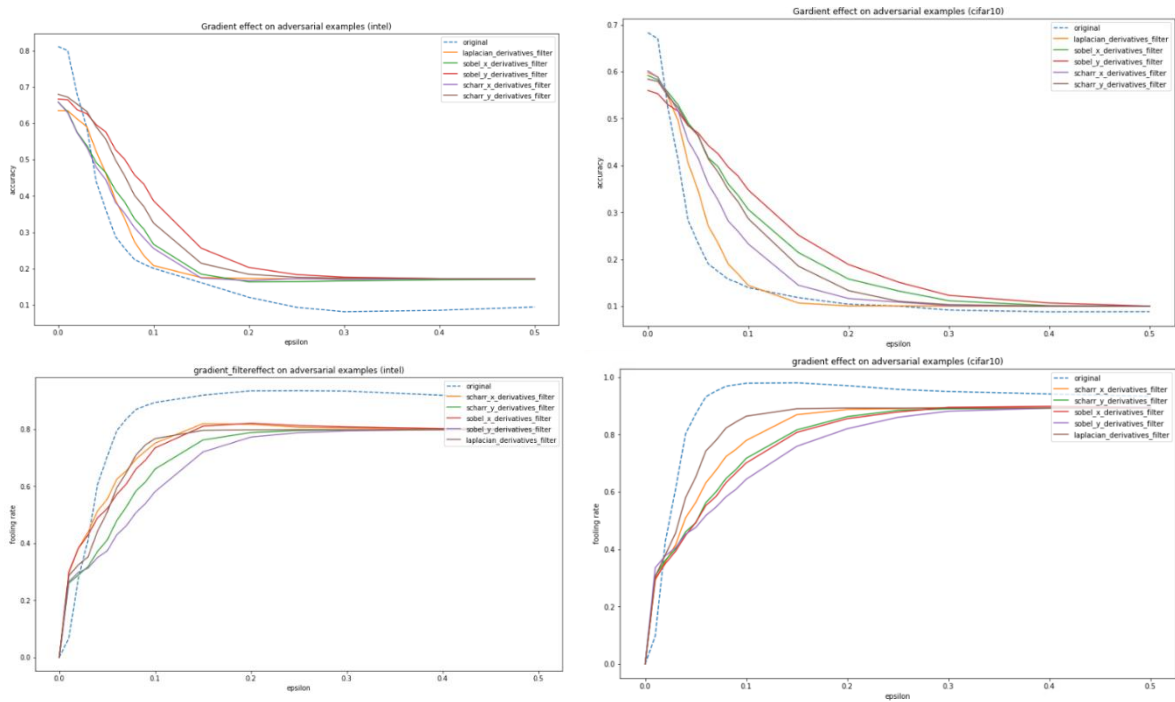


Figure 46: The effect of dimension reduction using the gradient filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

6.1.8. Bilateral Filtering

This filter enjoys both worlds - it performs a weighted average that depends on both the physical distance from the pixel and the color distance. Let us remember that bilateral filtering knows how to blur an image while preserving the edges. Therefore, it produces excellent results for a wide range of epsilon values. We can see that the bilateral filter gives similar results for different factor values, especially in the case of the low-resolution CIFAR-10 dataset.

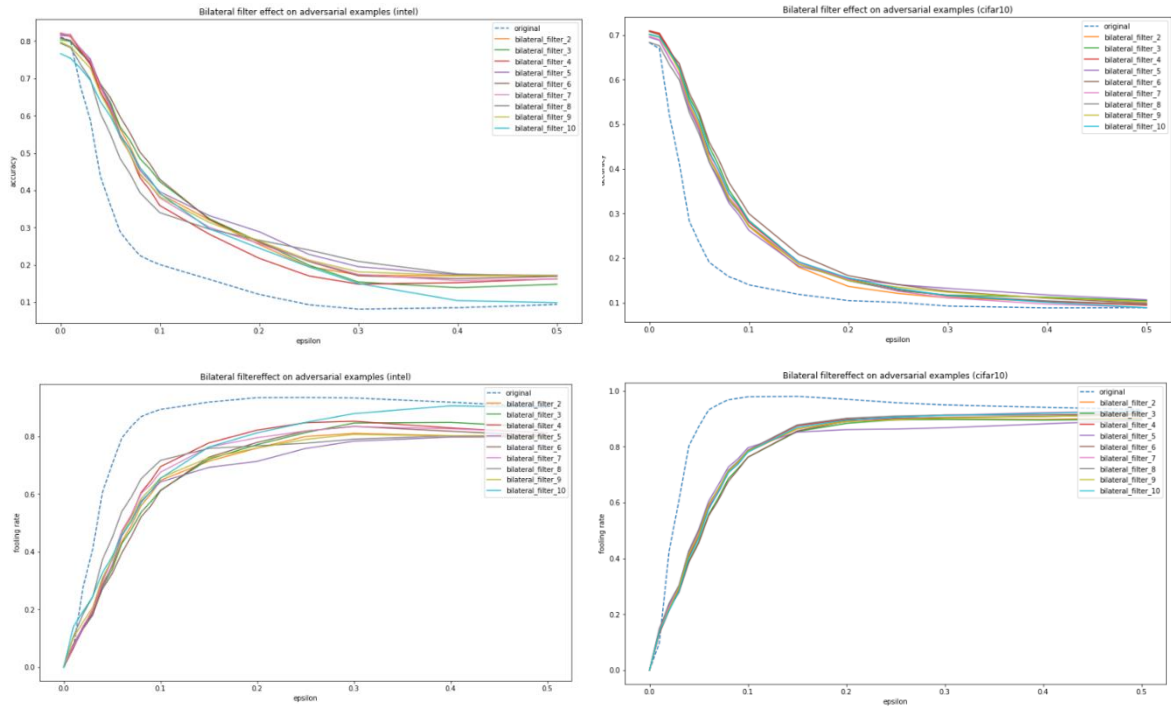


Figure 47: The effect of dimension reduction using the bilateral filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

Notably, the filter retains the quality of the original image best than we have seen so far, based on the high PSNR values. In the case of CIFAR-10 this is even more obvious when for small values of the factor, a significant portion of the images received a maximum value of 100, which indicates that the MSE was equal to zero (meaning, identical images).

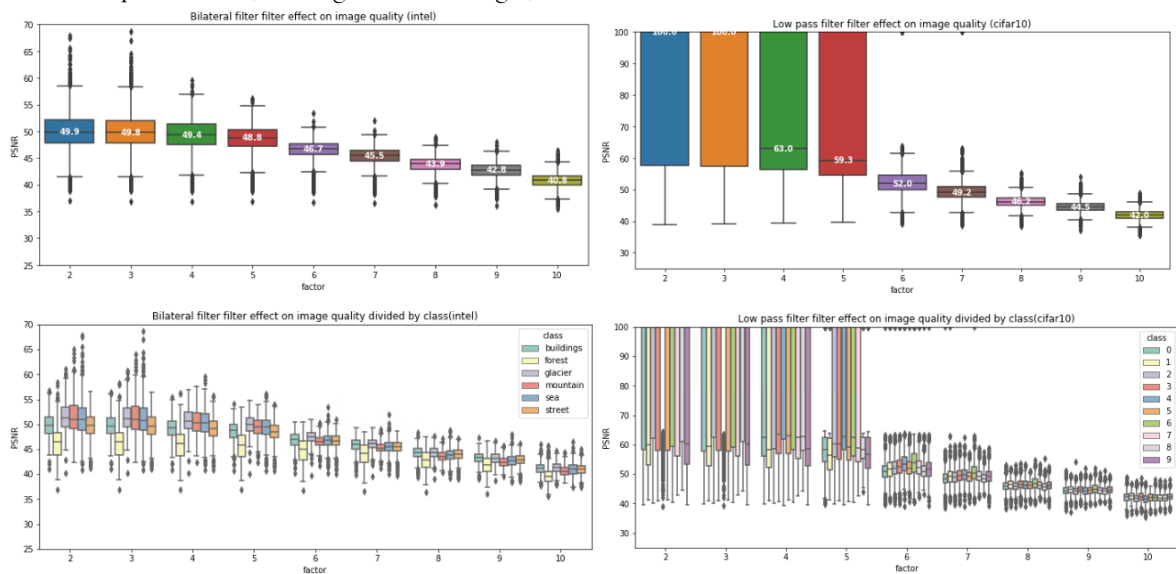


Figure 48: The effect of dimension reduction using the bilateral filter method on the quality of the image. In the x axis we can see different factor values, while the y axis represent PSNR values. The upper graphs are box plot where we can see the 25 & 75 percentile and the median value of the PSNR for different factor values. The graphs at the bottom are separate for each class.

6.1.9. Canny Edge Filtering

As for gradient filtering, canny edge detection filtering is useful for only a certain range of epsilon values. For values too small the model gives a weaker identification than the original model. Again, this was easy to predict because by finding the edges we lost most of the image information, so for small epsilon, we will have less accurate results.

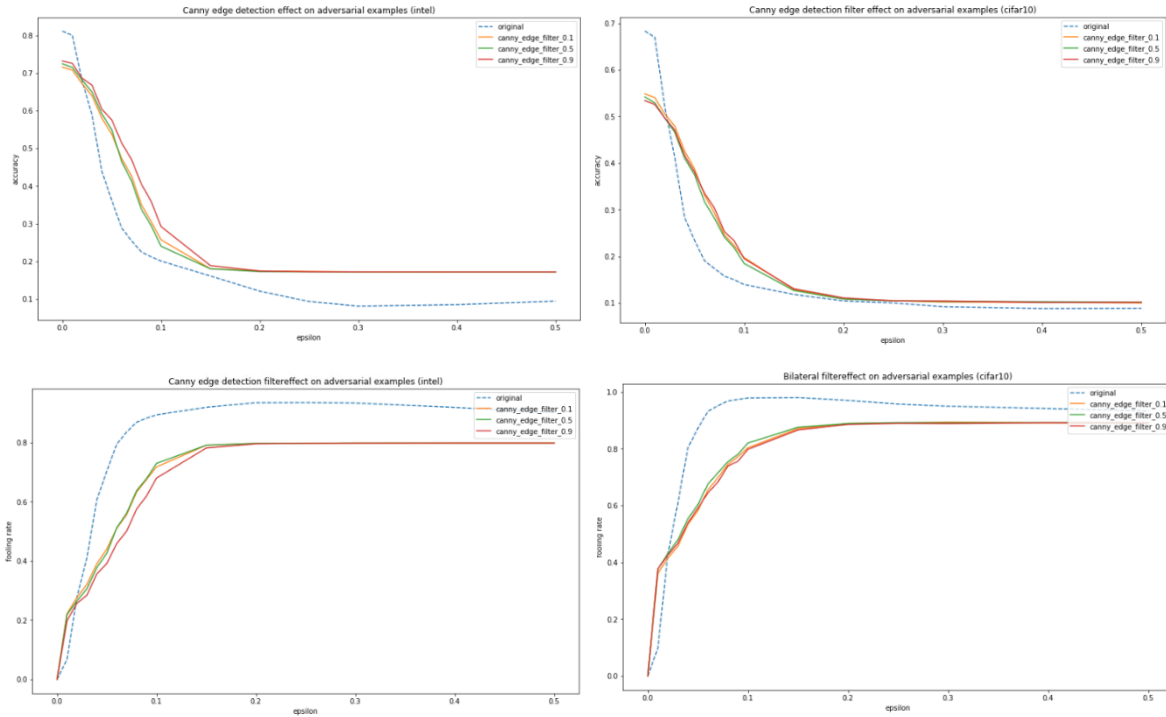


Figure 49: The effect of dimension reduction using the canny edge detection filter method on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate.

6.1.10. Summary

To summarize the nine experiments we performed we wish to examine which dimension reduction methods were more effective in dealing with an adversarial attack using the FGSM algorithm. We also compare all methods to the original model. When comparing effectiveness, we consider the predefined accuracy metrics as well as the damage in image quality.

Figure 50 shows all the results on a single graph when each color represents a particular reduction method and the plurality of graphs from the same color is due to the number of parameters set. The continuous black line expresses the performance of the original model on the test set. It is noticeable that K-means and PCA were able to achieve the best accuracy and fooling rate results. On the other hand, the least successful results were unsurprisingly due to the loss of a lot of information, those that relied on finding maps from the original image - gradient filtering and edge detection.

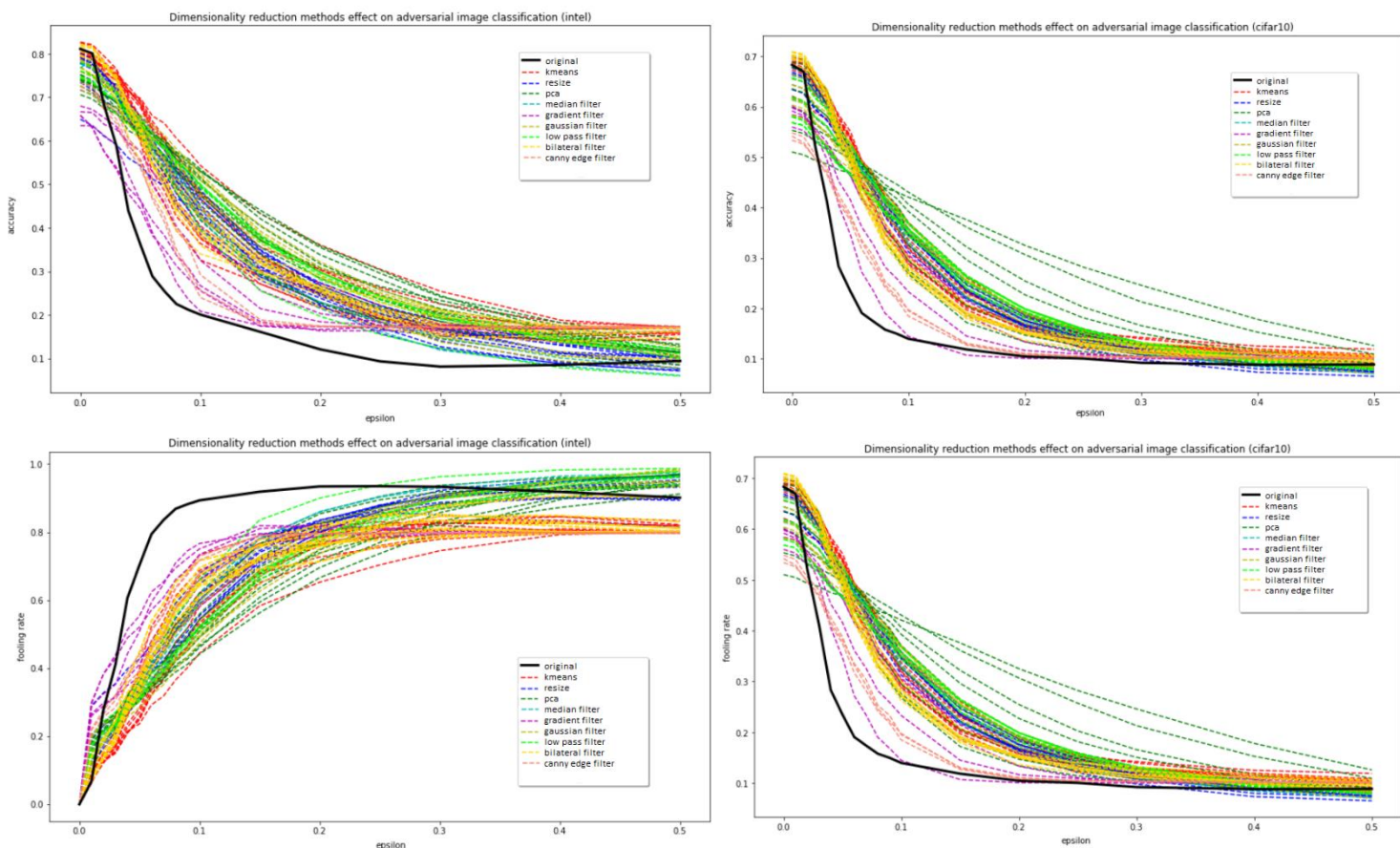


Figure 50: The effect of all dimension reduction methods on correctly identifying adversarial examples. The x axis is the epsilon value in the FGSM adversarial method, while the y axis represent percentage. The two upper graphs shows the accuracy in INTEL and CIFAR-10 datasets while at the bottom we can see the fooling rate. The black line represent the performance of the original model while each color stand for different dimension redction method. Same color lines are due to different parameter values.

Table 1 lists the impact of each method on image quality by the PSNR value. To summarize the box plots presented earlier, we can see the best median obtained for each method, the worst median and the average median for different parameters.

It seems from the results that bilateral filtering achieved the minimum image quality impairment. Kmeans also achieved significantly high performance for large k values. PCA, on the other hand, showed a mixed trend when CIFAR-10 showed good results but got worst on the INTEL dataset. Other models achieved similar median PSNR results.

Method	INTEL			CIFAR-10		
	Best median	AVG median	Worst median	Best median	AVG median	Worst median
	PSNR	PSNR	PSNR	PSNR	PSNR	PSNR
Resize & Rescale	31.4	30.56	29.5	31.5	30.24	29.4
Kmeans	37.8	29.59	20.1	37.7	32	28.1
PCA	28.2	28.16	28.1	37.5	29.64	28.1
LPF	30.6	29.88	29.11	30.2	29.15	28.5
Gaussian	30.5	30.06	29.8	30.7	29.43	28.7
Median	31.2	30.85	30.5	31.4	30.7	30
Gradient	-	-	-	-	-	-
Bilateral	49.9	46.4	40.8	100	56.8	42
Canny	-	-	-	-	-	-

Table 1: The effect of different dimension reduction methods on image quality using the PSNR measure. For each method and dataset we show the highest, lowest and average of the median PSNR value for each experiment of certain factor.

6.2. Ensemble Results

As mentioned, we gave the results of all the models to a meta-learn for training and evaluation. We have taken several ensembles approaches: voting, stacking of predicated classes and stacking of prediction distribution. Before describing the results we first give the motivation for studying ensemble methods.

6.2.1. Motivation

We know that the multiplicity of dimensions is one of the most influential factors on the likelihood of an adversarial attack. With adversarial training, dimension reduction is one of the most-studied approaches in literature (see chapter 3.3.2) for dealing with this phenomena, and was intensively examined in this research as well. Many methods have given good results for certain data sets and certain attack methods. But, each method also failed in some of the conditions. Hence, there seems to be no magic solution that is suitable for all cases.

We have therefore considered the distribution of model answers. Figure 51 shows for selected epsilon values the right prediction distribution of each model on all instances. In the rows, you can see each of the test models (a dimension reduction method with a parameter value) and all the test set images in the columns. Black cube in a cell $[i, j]$ indicates that model i had the right prediction on the image j . The percentage of black cells per row is the accuracy of the model, and the percentage in the column is the percentage of the models that were right to predict the specific image. Note that the first line expressing the true value is black all the way.

Some conclusions arise from this graph. First of all, of course, you can visually see the accuracy decline with the rise in the value of epsilon. Secondly, it is possible to notice that there is no uniformity in the answers of the models, meaning that each model has the input areas where it is better in prediction than other models and there is no perfect overlap between these areas. Therefore, several models can be harnessed to try to train a meta-learner in the cases in which it tends to be more accurate. So, for example, the meta-learner may be taught that a particular model is better in predicting some kind of class and therefore will prioritize its answer over other models.

True predicted value distribution for each model on every image

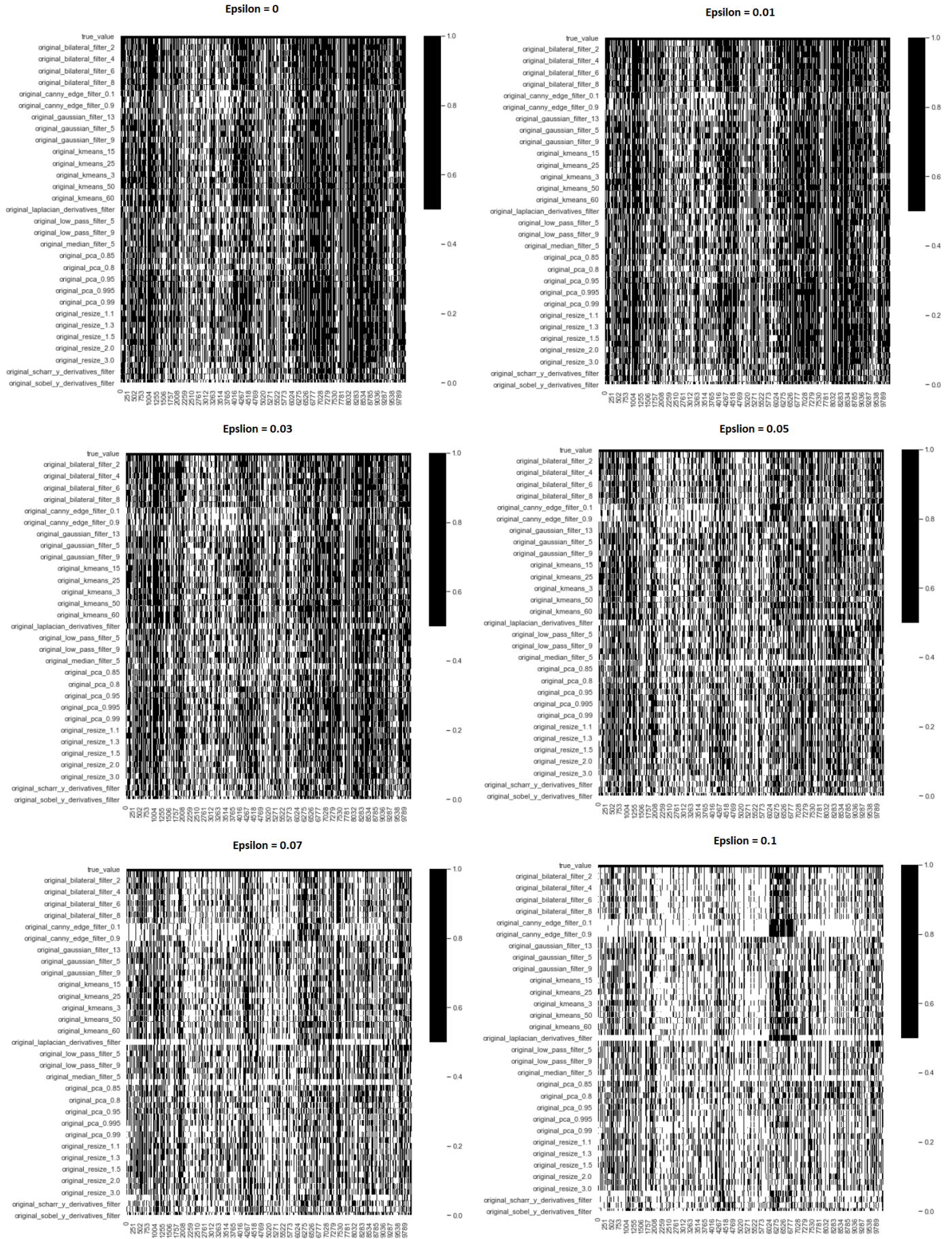


Figure 51: Each graph shows for a particular epsilon value a matrix of results for all models (rows) on all images (columns), with a black value indicating a correct prediction and a white value indicating an error. The first line expresses the true value and is therefore black in length.

6.2.2. Voting and Stacking Results

Figure 52 shows the results for the CIFAR-10 dataset, when the graphs of voting, class stacking and probability stacking are painted in green, red and blue respectively. The continuous black line describes the results of the original model and the broken line is the best dimension reduction model at that point.

The results are notable for the fact that the three methods of ensembling win each model for epsilon values equals or smaller than 0.07. This is, of course, the greatest chance of an attack, since we have seen that it has a very little effect on the impact of the image but critically harms the identification performance. So, for example, the percentage of the original model is dropping up to 0.174 for epsilon equals 0.07, a performance that begins to approach the trivial model that gives a random class uniformly. For larger epsilon values, the ensemble models give better results than most of the models except PCA. Also, the trend between the methods itself is turned off at a certain point, i.e. for low values of epsilon stacking-based methods gave better results and starting from a certain point voting model performs better.

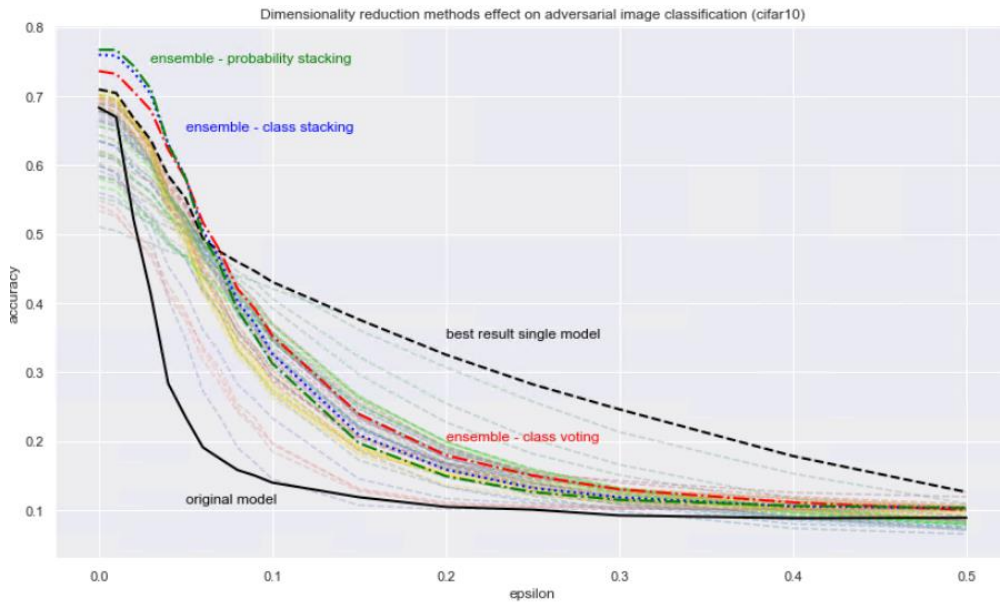


Figure 52: The effect of using ensemble approaches on dimension reduction methods results for correctly identifying adversarial examples. The x-axis is the epsilon value in the FGSM adversarial method, while the y-axis represents the accuracy percentage.

Diving into the numbers themselves in table 2 shows that probability stacking gives the best result in small epsilon values immediately thereafter class stacking and at some point voting.

	original	best_result_single_model	ensemble_voting	ensemble_stacking_class(H2O)	ensemble_stacking_prob(H2O)
train	0.915	0.923			
validation	0.701	0.723		0.779	1.000
test	0.683	0.710	0.736	0.760	0.767
adv_epsilon_0.01	0.670	0.705	0.732	0.759	0.767
adv_epsilon_0.02	0.522	0.669	0.707	0.735	0.744
adv_epsilon_0.03	0.413	0.636	0.680	0.703	0.710
adv_epsilon_0.04	0.284	0.585	0.623	0.630	0.630
adv_epsilon_0.05	0.235	0.552	0.580	0.583	0.582
adv_epsilon_0.06	0.191	0.494	0.517	0.505	0.498
adv_epsilon_0.07	0.174	0.474	0.475	0.463	0.453
adv_epsilon_0.08	0.158	0.459	0.421	0.402	0.389
adv_epsilon_0.09	0.150	0.446	0.391	0.370	0.353
adv_epsilon_0.1	0.140	0.431	0.353	0.326	0.313
adv_epsilon_0.15	0.118	0.376	0.239	0.209	0.197
adv_epsilon_0.2	0.105	0.325	0.179	0.159	0.149
adv_epsilon_0.25	0.100	0.282	0.150	0.132	0.126
adv_epsilon_0.3	0.092	0.246	0.130	0.118	0.115
adv_epsilon_0.4	0.088	0.178	0.111	0.105	0.106
adv_epsilon_0.5	0.089	0.126	0.100	0.103	0.103

Table 2: Accuracy results in the CIFAR-10 dataset for different epsilon values. The table displays the results for the original model, the best result achieved for a dimension reduction model at this point and the ensemble methods of voting, class stacking and probability stacking.

For the INTEL dataset, we see a similar trend when up to epsilon that is less than or equal to 0.05 the ensemble methods gave better results than the best dimension reduction model at this point.

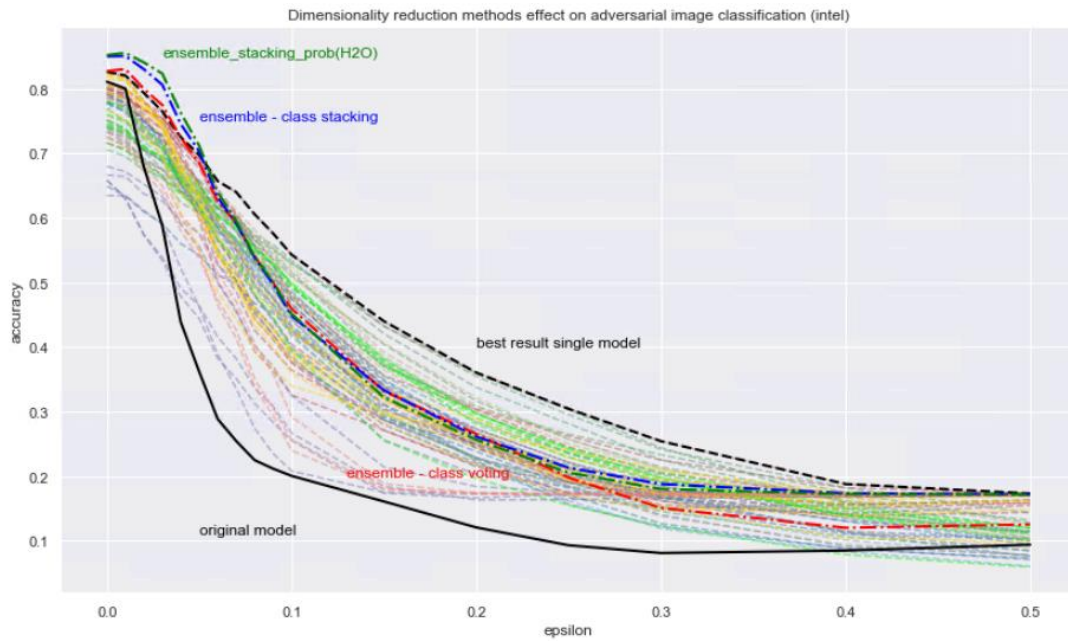


Figure 53: The effect of using ensemble approaches on dimension reduction methods results for correctly identifying adversarial examples. The x-axis is the epsilon value in the FGSM adversarial method, while the y-axis represents the accuracy percentage.

The following table shows the results summary.

	original	best_result_single_model	ensemble_voting	ensemble_stacking_class(H2O)	ensemble_stacking_prob(H2O)
train	0.975	0.997			
validation	0.826	0.842		0.870	1.000
test	0.811	0.827	0.827	0.850	0.852
adv_epsilon_0.01	0.800	0.821	0.831	0.851	0.856
adv_epsilon_0.02	0.679	0.794	0.800	0.830	0.841
adv_epsilon_0.03	0.588	0.765	0.775	0.806	0.824
adv_epsilon_0.04	0.439	0.725	0.724	0.746	0.763
adv_epsilon_0.05	0.362	0.698	0.685	0.703	0.712
adv_epsilon_0.06	0.289	0.657	0.626	0.632	0.640
adv_epsilon_0.07	0.255	0.640	0.593	0.592	0.596
adv_epsilon_0.08	0.225	0.606	0.541	0.539	0.540
adv_epsilon_0.09	0.212	0.576	0.500	0.497	0.501
adv_epsilon_0.1	0.201	0.543	0.460	0.448	0.452
adv_epsilon_0.15	0.161	0.440	0.332	0.333	0.322
adv_epsilon_0.2	0.121	0.361	0.266	0.262	0.257
adv_epsilon_0.25	0.093	0.305	0.198	0.213	0.206
adv_epsilon_0.3	0.081	0.254	0.151	0.188	0.181
adv_epsilon_0.4	0.085	0.188	0.120	0.174	0.172
adv_epsilon_0.5	0.094	0.173	0.126	0.172	0.172

Table 3: Accuracy results in the INTEL dataset for different epsilon values. The table displays the results for the original model, the best result achieved for a dimension reduction model at this point and the ensemble methods of voting, class stacking and probability stacking.

7. Concluding Remarks and Future Work

The use of ensemble approaches, both stacking and voting, on dimensionality reduction methods has proven itself to be a very effective way with dramatic classification performance improvement. The method aims to cover different areas of the input space in which certain models may be better than others and to learn them in order to give the best possible answer. Thus, by implementing a bucket containing a large number of dimension reduction models, it can be enriched at any given moment with thousands of other models.

The ensemble approach and the results that we got using it suggest several directions of future research. First, it is interesting to verify the results that were received on additional attacks and deeper networks. We can also examine the addition of extra methods and parameters on a large scale that we tested here.

Secondly, we put all the models inside the bucket for the meta-learner. A more intelligent approach might examine the models and choose how to filter them or how to better select the parameters for the dimension reduction algorithm. For example, here we examined several k values in the KMeans algorithm. It is interesting to see if one can make a smart choice of the k parameter.

We have presented 3 types of ensemble methods here and noticed, again, that there are different conditions in which each meta-learner is better than others. Therefore, why not build a new meta learner that accepts all existing meta-learners' answers to maximize their answer as well?

In this research we focused only on the classification performance, using the accuracy and fooling rate, and the image quality damage using the PSNR. Since the dimension reduction algorithm is used each time the network classifies a new example and not only in the training phase, the time complexity of each algorithm might also be important for some applications. The different dimensionality reduction methods also differ in running time. This may be critical when talking about systems that require a rapid diagnosis in a short time like for example in the case of the autonomous vehicle.

8. References

- [1] Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., & Fergus, R. (2013). Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199.
- [2] Samuel, Arthur L. (1959). "Some studies in machine learning using the game of checkers". IBM Journal of research and development.
- [3] Shalev-Shwartz, S. & Ben-David, S. (2014). Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press, New York, USA.
- [4] Mitchell, T. (1997). Machine Learning. McGraw-Hill, Portland, Oregon, USA.
- [5] Guestrin, C. and Fox, E. Coursera: Machine Learning Foundations: A Case Study Approach. Available: <https://www.coursera.org/learn/ml-foundations>
- [6] Yoffie, D. B., Wu, L., Sweitzer, J., Eden, D., & Ahuja, K. (2018). "Voice War: Hey Google vs. Alexa vs. Siri". Harvard Business School. Available: https://files.transtutors.com/cdn/uploadassignments/2835385_1_voice-war.pdf
- [7] Graves, A., Mohamed, A. R., & Hinton, G. (2013, May). Speech recognition with deep recurrent neural networks. In 2013 IEEE international conference on acoustics, speech and signal processing (pp. 6645-6649). IEEE.
- [8] Graves, A., & Jaitly, N. (2014, January). Towards end-to-end speech recognition with recurrent neural networks. In International conference on machine learning (pp. 1764-1772).
- [9] Sak, H., Senior, A., & Beaufays, F. (2014). Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. arXiv preprint arXiv:1402.1128.
- [10] Tu, Z., Hu, B., Lu, Z., & Li, H. (2015). Context-dependent translation selection using convolutional neural network. arXiv preprint arXiv:1503.02357.
- [11] Liu, S., Yang, N., Li, M., & Zhou, M. (2014, June). A recursive recurrent neural network for statistical machine translation. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 1491-1500).
- [12] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).
- [13] Denil, M., Demiraj, A., Kalchbrenner, N., Blunsom, P., & de Freitas, N. (2014). Modelling, visualising and summarising documents with a single convolutional neural network. arXiv preprint arXiv:1406.3830.
- [14] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... & Zhang, X. (2016). End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316.
- [15] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).
- [16] Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and harnessing adversarial examples. arXiv preprint arXiv:1412.6572.
- [17] Rosenblatt, Frank (1957). "The Perceptron - a perceiving and recognizing automaton". Report 85-460-1. Cornell Aeronautical Laboratory.
- [18] Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. Insights into imaging, 9(4), 611-629.

- [19] Moosavi-Dezfooli, S. M., Fawzi, A., Fawzi, O., & Frossard, P. (2017). Universal adversarial perturbations. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1765-1773).
- [20] Kurakin, A., Goodfellow, I., & Bengio, S. (2016). Adversarial examples in the physical world. arXiv preprint arXiv:1607.02533.
- [21] Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., ... & Song, D. (2018). Robust physical-world attacks on deep learning visual classification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1625-1634).
- [22] Akhtar, N., & Mian, A. (2018). Threat of adversarial attacks on deep learning in computer vision: A survey. IEEE Access, 6, 14410-14430.
- [23] Chakraborty, A., Alam, M., Dey, V., Chattopadhyay, A., & Mukhopadhyay, D. (2018). Adversarial attacks and defences: A survey. arXiv preprint arXiv:1810.00069.
- [24] Liu, Q., Li, P., Zhao, W., Cai, W., Yu, S., & Leung, V. C. (2018). A survey on security threats and defensive techniques of machine learning: A data driven view. IEEE access, 6, 12103-12117.
- [25] Kurakin, A., Goodfellow, I., & Bengio, S. (2016). Adversarial machine learning at scale. arXiv preprint arXiv:1611.01236.
- [26] Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z. B., & Swami, A. (2016, March). The limitations of deep learning in adversarial settings. In 2016 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 372-387). IEEE.
- [27] Su, J., Vargas, D. V., & Sakurai, K. (2019). One pixel attack for fooling deep neural networks. IEEE Transactions on Evolutionary Computation.
- [28] Baluja, S., & Fischer, I. (2017). Adversarial transformation networks: Learning to generate adversarial examples. arXiv preprint arXiv:1703.09387.
- [29] Kurakin, A., Goodfellow, I., Bengio, S., Dong, Y., Liao, F., Liang, M., ... & Wang, J. (2018). Adversarial attacks and defences competition. In The NIPS'17 Competition: Building Intelligent Systems (pp. 195-231). Springer, Cham.
- [30] Dziugaite, G. K., Ghahramani, Z., & Roy, D. M. (2016). A study of the effect of jpeg compression on adversarial images. arXiv preprint arXiv:1608.00853.
- [31] Das, N., Shanbhogue, M., Chen, S. T., Hohman, F., Chen, L., Kounavis, M. E., & Chau, D. H. (2017). Keeping the bad guys out: Protecting and vaccinating deep learning with jpeg compression. arXiv preprint arXiv:1705.02900.
- [32] Moosavi-Dezfooli, S. M., Fawzi, A., & Frossard, P. (2016). Deepfool: a simple and accurate method to fool deep neural networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 2574-2582).
- [33] Guo, C., Rana, M., Cisse, M., & Van Der Maaten, L. (2017). Countering adversarial images using input transformations. arXiv preprint arXiv:1711.00117.
- [34] Shin, R., & Song, D. (2017, December). Jpeg-resistant adversarial images. In NIPS 2017 Workshop on Machine Learning and Computer Security (Vol. 1).
- [35] Shaham, U., Garritano, J., Yamada, Y., Weinberger, E., Cloninger, A., Cheng, X., ... & Kluger, Y. (2018). Defending against adversarial images using basis functions transformations. arXiv preprint arXiv:1803.10840.

- [36] Xie, C., Wang, J., Zhang, Z., Ren, Z., & Yuille, A. (2017). Mitigating adversarial effects through randomization. arXiv preprint arXiv:1711.01991.
- [37] Gu, S., Yi, P., Zhu, T., Yao, Y., & Wang, W. (2019). Detecting Adversarial Examples in Deep Neural Networks using Normalizing Filters. UMBC Student Collection.
- [38] Xu, W., Evans, D., & Qi, Y. (2017). Feature squeezing: Detecting adversarial examples in deep neural networks. arXiv preprint arXiv:1704.01155.
- [40] Sahay, R., Mahfuz, R., & El Gamal, A. (2019, March). Combatting adversarial attacks through denoising and dimensionality reduction: A cascaded autoencoder approach. In 2019 53rd Annual Conference on Information Sciences and Systems (CISS) (pp. 1-6). IEEE.
- [41] Lee, H., Han, S., & Lee, J. (2017). Generative adversarial trainer: Defense to adversarial perturbations with gan. arXiv preprint arXiv:1705.03387.
- [42] Ross, A. S., & Doshi-Velez, F. (2018, April). Improving the adversarial robustness and interpretability of deep neural networks by regularizing their input gradients. In Thirty-second AAAI conference on artificial intelligence.
- [43] Lloyd, S. (1982). Least squares quantization in PCM. IEEE transactions on information theory, 28(2), 129-137.
- [44] Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence, (6), 679-698.
- [45] Kaggle, Datasets: Intel Image Classification - Image Scene Classification of Multiclass. Available: <https://www.kaggle.com/puneet6060/intel-image-classification>
- [46] Papernot, N., Carlini, N., Goodfellow, I., Feinman, R., Faghri, F., Matyasko, A., ... & Garg, A. (2016). cleverhans v2. 0.0: an adversarial machine learning library. arXiv preprint arXiv:1610.00768.
- [47] H2O.ai. H2O AutoML, August 2017. URL <http://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html>.