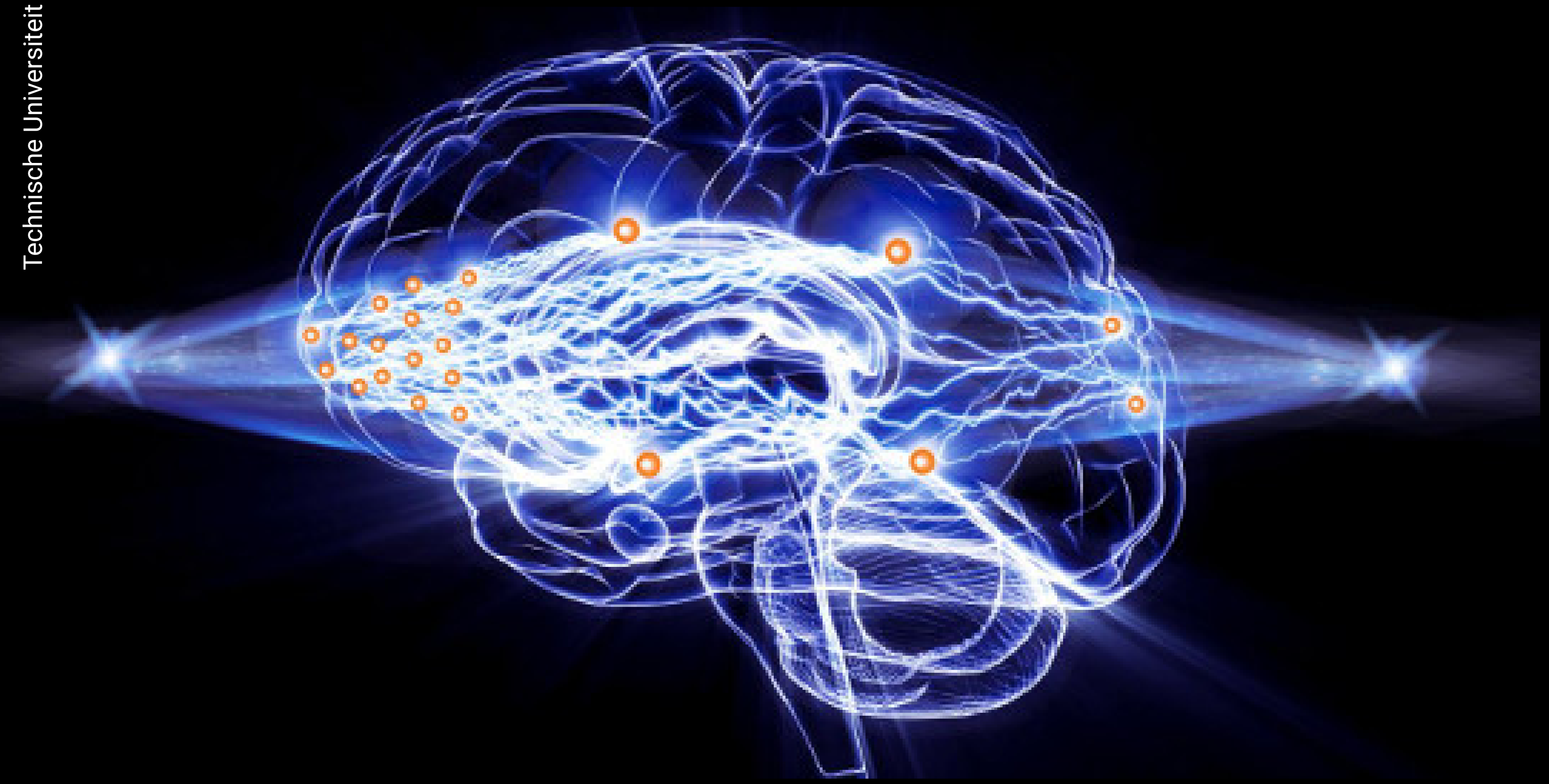
Exploiting neuron activation values for creating adversarial examples

Utilization of intermediate network information in genetic algorithms

Irene van der Blij



Exploiting neuron activation values for creating adversarial examples

Utilization of intermediate network information in genetic algorithms

by

Irene van der Blij

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday March 25, 2021 at 09:00 AM.

Student number:	4385691	
Project duration:	March 2020 – March 2021	
Thesis committee:	Dr. ir. S.E. (Sicco) Verwer,	TU Delft, supervisor
	Prof. dr. R. (Robert) Babuska,	TU Delft
	Prof. dr. ir. RL (Inald) Lagendijk	TU Delft

Abstract

The increasing usage of neural networks forms a threat to the cyber security as attacks with adversarial examples can deceive the networks. Because neural networks can have complex structures with tens of thousands of parameters, they are hard for humans to understand. Hence, existing white-box attacks use very limited network information and most state-of-the-art methods are based on gradient descent. However, intuitively attacks and defenses can be more effective when the user understands the model and uses all information contained by the model. In this work we further investigate the inner workings of a neural network and consider using intermediate network information for the creation of adversarial examples. We show that neuron activation values can be distinguished by the class of the data point and contain meaningful information about the prediction of the network. Based on this information, we propose a new, gradient-free method for creating adversarial examples based on a genetic algorithm. By covering a larger part of the search space and manipulating the neuron activation values, our success rate exceeds most state-of-the-art methods, such as DeepFool and RFGSM. We also find that the trade-off between success rate and distance has a huge impact on the results of a method, wherefore we recommend to carefully balance this trade-off by formulating an optimization formula with a separate loss and distance component.

Preface

When I started my Bachelor's degree, I already dreaded the process of writing a Master's thesis. For the most part of my time as Bachelor's student I sincerely proclaimed that I would stop studying after I had obtained my Bachelor of Science, and never start a Master's degree. Yet here we are! I was not ready to let go of the perks of student life and wanted to enjoy the freedom, parties, and having fun with my friends for a few extra years. I will cherish the good times I had during my studies, but I am also exited to be writing this and start a new chapter in my life.

As I expected, writing a Master's thesis was easier said than done. With a lack of inspiration and passion the process started laboriously. As the people close to me know, I had some personal circumstances and I was going through a rough time. On top of that, after I had had my first few thesis meetings, the Covid-19 crisis hit The Netherlands and the universities had to close. Considering the circumstances, I am very proud to present this work and obtain my Master's degree.

First of all, I would like to thank my supervisor, Sicco Verwer, for his guidance and feedback, and being understanding towards my personal circumstances. Thank you for convincing me to keep going when I wanted to quit and thought it would be better to start all over with a new subject. I also want to thank Robert Babuska and Inald Lagendijk for taking place in my thesis committee and taking the time to evaluate my work. Furthermore, I want to give a shoutout to the other members and students of Sicco's group for the positive vibes and motivational group meetings. Knowing you are not alone is a great consolation.

I also want to thank the people close to me. First of all my parents, who were always there for me through difficult times and financed my studies. Next I want to thank my boyfriend, Bob, for taking care of me when I couldn't and supporting me throughout the whole process. Seeing how proud you are of me makes me feel proud of myself, and I cannot express how thankful I am for having someone like that in my life. Furthermore I want to thank my sisters and friends, who were always interested in my progress and kept me motivated.

Irene van der Blij
Delft, March 2021

Contents

1 Introduction

The use of machine learning techniques and artificial intelligence is growing in many sectors. For example, in the health care sector these techniques can help diagnose breast cancer [1], in the financial sector it can detect credit card fraud [18], and in the car industry the techniques are employed to create self-driving vehicles [9]. The increasing usage of machine learning techniques is explained by its outstanding performance, especially for high-dimensional problems for which the patterns and solutions are not evident to the human mind.

A popular artificial intelligence technique is the neural network. This is a model containing nodes and weights, which are used to classify an input to a certain class. Neural networks can have complex structures with tens of thousands of parameters, and as such they are hard for humans to understand. The user of the model is unaware of the reasoning of the model, making it nearly impossible to detect abnormalities.

This is a threat to the cyber security of the system that uses the model, and unfortunately adversaries use this vulnerability to attack the system. A well-known and successful attack is the adversarial example. This attack aims to make very small changes to the input, such that the data point looks the same, but its class is wrongly predicted by the neural network.

Over the last decade, many methods were proposed to create adversarial examples, and to defend against them [21]. The ongoing developments in this field result in a cat-and-mouse game, where adversaries find new ways to attack the model with unseen methods to create adversarial examples, and the victim responds to this by creating new defense mechanisms against the new attacks. As new, better methods for creating adversarial examples are still found today, it is interesting to look at the inner workings of existing methods and look for possible improvements.

Surprisingly, existing methods use very limited network information. Most state-of-the-art methods are based on gradient descent, using the gradient of the loss function with respect to the input. A neural network, however, likely contains much more valuable information which is not used in existing methods. This is seemingly due to the difficulty of understanding the inner workings of a neural network and the unclear meaning of (intermediate) network information.

An interesting and accessible piece of network information are the intermediate results of the network: the neuron activation values. A neuron activation value is determined by the activation values of the previous layer with their corresponding weights. The result is put through an activation function, producing our desired neuron activation value.

1.1. Motivating example

As an explanatory example, we consider a very simple neural network with two hidden neurons and two output classes. We separate the input data by their actual output class, to be able to see the differences in the neuron activation values. In figure 1.1 the activation values of the first hidden neuron are shown for the separated data. The differences in activation values are clearly distinguishable for the two output classes, and hence we suspect that we are dealing with some valuable information.

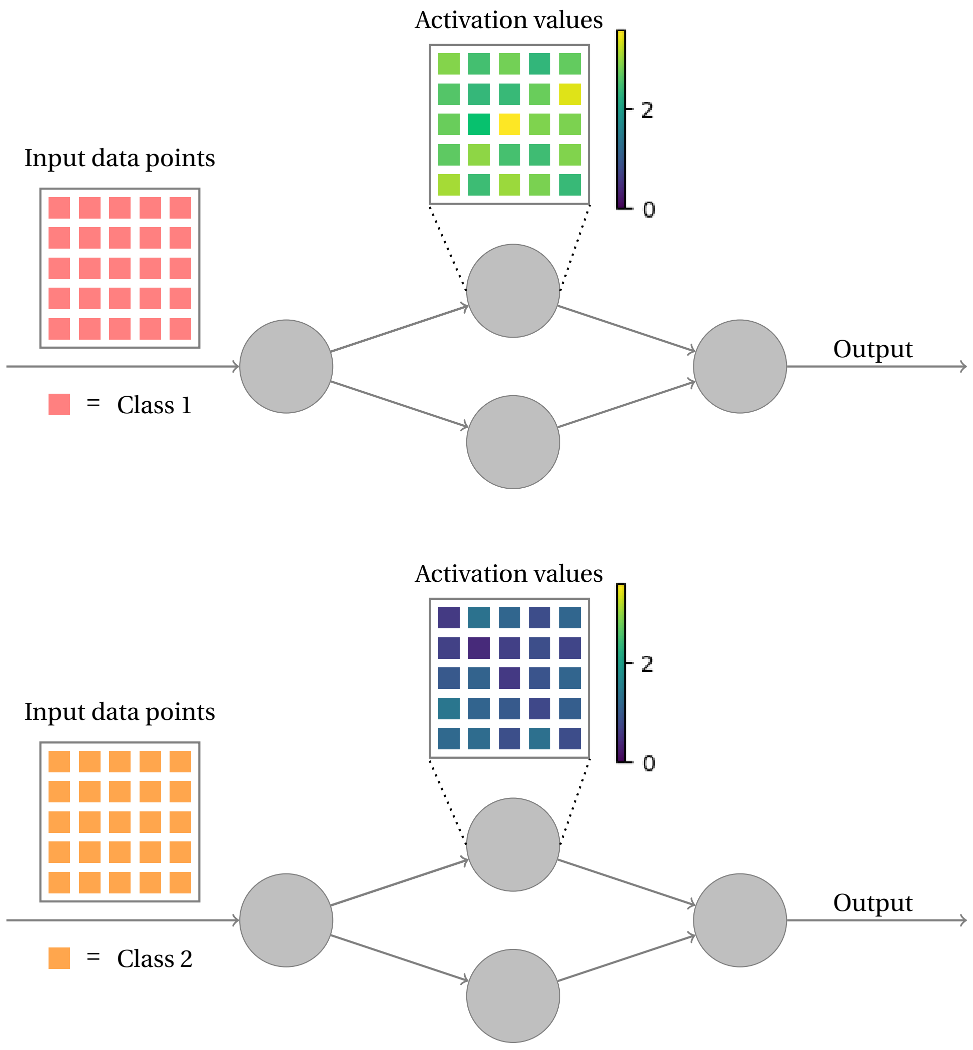


Figure 1.1: Visualization of a simple neural network with two hidden neurons and two output classes. The input data is separated by their class label, showing the evaluation of the network for both classes separately. The top hidden neuron is highlighted by showing the activation values corresponding to the input. The data points belonging to a different class show distinguishable neuron activation values.

We want to exploit this information for creating adversarial examples by combining it with a suitable and powerful algorithm. The algorithm should be conscious of the fact that adversarial examples are more often found when a larger part of the search space is considered, and the algorithm needs to be capable of optimization for high-dimensional problems. Hence, we take a genetic algorithm as these are designed to explore large input spaces and are known for their good performance on high-dimensional data.

Combining the genetic algorithm with the neuron activation values, we aim to perturb the input in such a manner that the activation values of the data point change. The goal is to find a perturbation for which the activation values are similar to a different class, while the input still looks like the actual class. Let us clarify this by going back to the example in figure 1.1. Say we have a data point for which we want to find an adversarial example. The class label of the data point is ’1’, corresponding to a red square in the image.

Looking at the activation values, a data point of class ’1’ usually has activation values between 2 and 4, as opposed to data points of class ’2’ (orange squares) where the activation values lie between 0 and 2. Our goal now is to change the red square in such a way that its activation value will look like an activation value of an orange square, while still looking like a red square. This is shown in figure 1.2

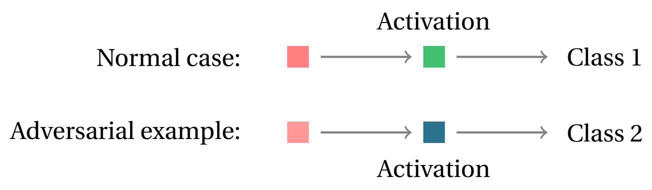


Figure 1.2: Visualization of our goal: perturb a data point to create an adversarial example based on the changed activation values. The input data point still has to look like it belongs to its original class, so the color of the square may slightly change, but not too much as we do not want it to look like another class. The desired perturbation results in activation values similar to a different class, causing the network to misclassify the perturbed data point.

This example furthermore shows the importance of covering the search space. Methods based on gradient descent will search for corner cases where the color of the square is somewhere between red and orange, finding the obvious adversarial examples. However, what would happen if we feed the network a purple square? The network may have never seen a similar input before, causing unwanted or unexpected behaviour and neuron activations, possibly resulting in new adversarial examples.

1.2. Research objectives

In this work we attempt to get a better understanding of the inner workings of a neural network, specifically concerning the neuron activation values. To achieve this, we investigate the behaviour and patterns of neuron activation values and try to identify to which extent changes in neuron activation values can result in finding new adversarial examples. The main research question is therefore formulated as follows:

How do neuron activation values behave and how can we use this information to create new adversarial examples?

The main research question is broken down into three sub-questions. The first step is to analyze neuron activation values in very simple networks, such that potential differences between the values of distinct classes become visible and we can reason about the meaningfulness of neuron activation values. Subsequently we investigate whether this also holds for larger, more complicated networks. The first sub-question therefore is:

1. **What differences are visible in neuron activation values for different classes, and does this hold for different network sizes?**

With the findings of this sub-question we want to investigate whether the neuron activation values are an indicator of the class label that the network classifies the data point to, and how we can translate neuron activation values to activation distributions and class probabilities. The option of using these activation distributions to create adversarial examples will be explored. The second sub-question therefore is:

2. **How can we deduce the class label from neuron activation distributions, and how can we use this to create adversarial examples?**

As we hope to develop a new method for creating adversarial examples, primarily based on neuron activation values, we want to evaluate its performance. The adversarial examples that are produced by the method will be compared to adversarial examples created by state-of-the-art methods, considering amongst others the success rate, distance, and efficiency of the methods. The third sub-question therefore is:

3. **What is the performance of the new method using neuron activation values, and how does this compare to the state-of-the-art?**

1.3. Contributions

In this work, we try to gain a deeper understanding of the information contained by neuron activation values, and use this to propose a gradient-free method for generating adversarial examples. Our main contributions are:

- **Network insight**: This work provides new insights into the inner workings of a neural network and the value of network information for classifications. Most importantly, we find that neuron activation values are very indicative of class labels.
- **GenNeuAct**: We propose a new method for generating adversarial examples. The basis of the method is a genetic algorithm, with a fitness function based on neuron activation values. The fitness of a data point is determined by scoring it against the estimated density function of each neuron and each output class. The success rate of the method is higher than most state-of-the-art methods, such as FGSM, BIM, DeepFool and RFGSM. Only C&W achieved superior success rates. The quality of the adversarial examples created by GenNeuAct is comparable to the state-of-the-art. A big strength of this method is the network coverage, as the method explores a larger part of the search space. Also, to our knowledge, this is the first work to utilize intermediate network information for classification.
- **Identification of ingredients for success**: We compare and analyze GenNeuAct and the state-of-the-art methods for generating adversarial examples, and look for properties causing the success of a method. We find that the trade-off between success rate and distance has a huge impact on the results of a method. Our findings on the C&W attack show that this trade-off can be carefully balanced by formulating an optimization formula with a separate loss and distance component.

1.4. Outline

The rest of this thesis is structured as follows: In chapter 2 we give background information about neural networks and genetic algorithms. In chapter 3 state-of-the-art white-box attacks are discussed and evaluated, and we look at previous works considering neuron activation values. Chapter 4 investigates what information can be extracted from neuron activation values and how this information can be used to make network predictions and adversarial examples. In chapter 5 the adversarial examples created by the method using neuron activation values is compared to the state-of-the-art for four different datasets. Chapter 6 discusses the limitations and recommendations of the research, and the conclusions are presented in chapter 7.

2 Background

This chapter presents and explains the background knowledge that is needed to understand this work. We elucidate the basics of how a neural network is constructed and how they work. Furthermore, the principles of a genetic algorithm are explained.

2.1. Neural networks

An artificial neural network (further referenced as neural network) is a circuit of interconnected neurons, inspired by the biological neural networks in the brain. The neural network takes as input a vector of real numbers, and also outputs a vector of real numbers. Between the input and output there are 'hidden layers' with neurons which can transmit a signal to other neurons. An example of a feed-forward neural network is shown in figure 2.1.

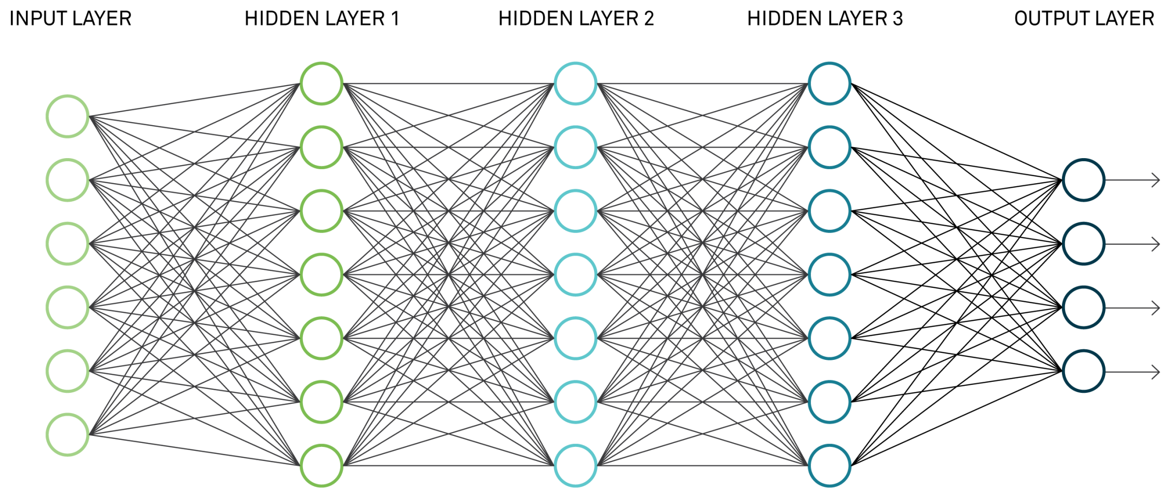


Figure 2.1: Example of a feed-forward neural network with 6 input neurons, 4 output neurons, and three hidden layers with each 7 neurons

2.1.1. Components

To get from the input to the output, the network does a long series of computations. An overview of the computations for a single neuron is visualized in figure 2.2. As visible in figure 2.1 and 2.2, a neural network is build up out of different components, which are repeated throughout the network. The next subsections will explain each component present in this overview.

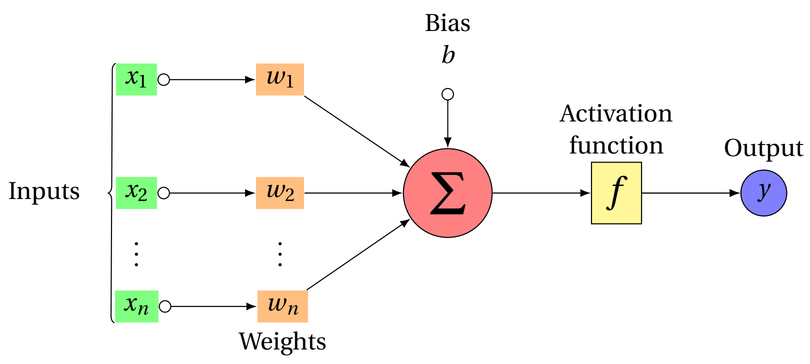
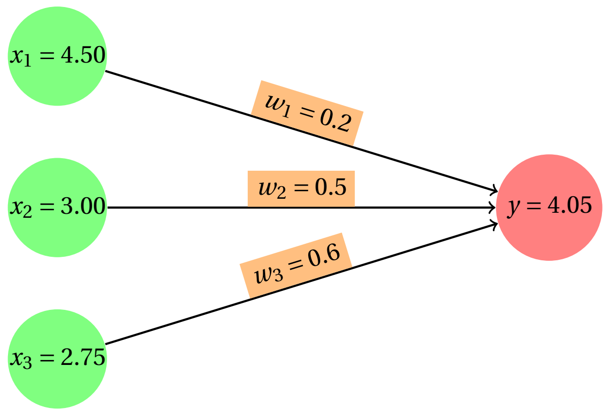


Figure 2.2: Overview of the inner computations for a single neuron in a neural network. All inputs are multiplied by the corresponding weight and summed. Next, the bias is added to this sum. The obtained value is inserted in the activation function, resulting in the final output of the neuron.

Weights

The connections between the neurons are called edges, and each edge has its own weight. For a fully connected network, each input neuron has a connection to each neurons in the first hidden layer, each neuron in the first hidden layer has a connection to each neuron in the second hidden layer, and so on. In the last hidden layer, each neuron has a connection to each of the output neurons. These weights increase or decrease the strength of the signal at a connection. To compute the value of a hidden neuron, the first step is to multiply the weight of each incoming edge with the value of the neuron the edge originated from. These values are summed at the neuron. An example calculation for a single neuron is shown in figure 2.3.



$$\begin{aligned} y &= \textstyle\sum_i x_i * w_i \\ &= x_1 * w_1 + x_2 * w_2 + x_3 * w_3 \\ &= 4.500.2 + 3.00 * 0.5 + 2.75 * 0.6 \\ &= 4.05 \end{aligned}$$

Figure 2.3: First step in calculating a hidden neuron value: sum input times weight

Biases

Except for the neurons in the input layer, each neuron has a value called the bias. This bias is a scalar value and is added to the result of the calculation in the previous section. The bias can be compared to a constant in a linear functions: it does not affect the steepness of the line, but shifts the entire line left or right. When a bias is high, the resulting neuron value will inherently also be higher. This makes it more likely for the neuron to be activated by the activation function.

Activation functions

The final step in calculating the output value of a neuron is applying the activation function. The most popular training techniques use a method called backpropagation (more about this later), which makes use of the gradient of the activation function. Therefore, we need a non-linear activation function. The non-linearity of the activation function also allows us to learn more complex data patterns. Some commonly used non-linear activation functions and their derivative are shown in figure 2.4.

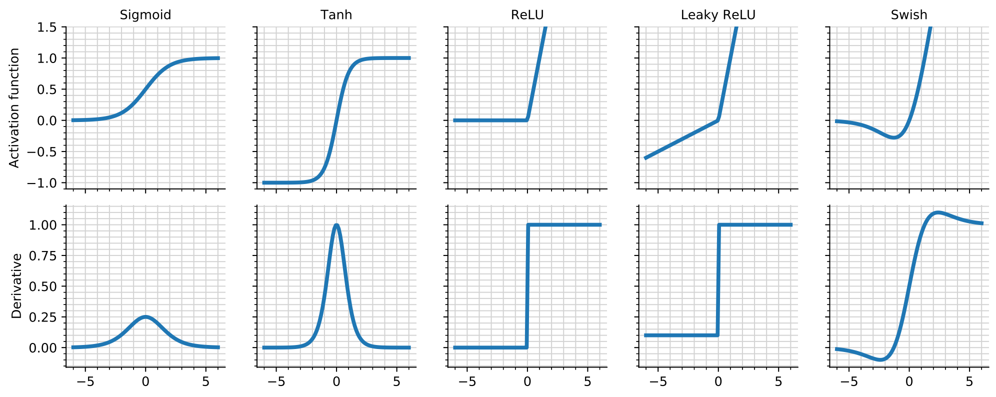


Figure 2.4: Different activation functions and their derivative

Sigmoid The Sigmoid function outputs values between 0 and 1, such that the neuron output is normalized. It gives clear predictions for x-values above 2 or below -2, and the function has a smooth gradient. However, for very small or very large x-values, we encounter the so-called *vanishing gradient problem*. The gradients for such values are so small that the weights do not change (anymore) and the network stops learning. Another problem is that this method is computationally expensive. This is important because the activation function is sometimes calculated for thousands or even millions of neurons for each data point. Another downside of this method is that the outputs are not zero-centered, this makes the optimization of the network harder and/or slower.

TanH The Hyperbolic Tangent (TanH) is very similar to the Sigmoid function. The biggest difference is that TanH is zero-centered, making it easier to optimize the network. Other than that, the advantages and disadvantages are the same as for the Sigmoid function.

ReLU The Rectified Linear Unit looks a lot like a linear function. However, since the x-values below zero are all mapped to 0, the function is non-linear and allows for backpropagation. In contrast to the Sigmoid and TanH function, this function is very computationally efficient, allowing the network to converge more quickly. A downside of this method is the so-called *dying ReLU problem*. This occurs when the input of the activation function approaches zero or is negative. The gradient of the function then becomes zero, also causing the loss function to be zero. From now on, the update step of the weights will remain the same as it is dependent on the value of the loss function. When a neuron gets into this state, we say that the neuron is dead.

Leaky ReLU The Leaky ReLU function was designed to solve the dying ReLU problem. Instead of mapping all x-values below zero to 0, there is a small positive slope. This allows for backpropagation even with negative values, preventing the dying ReLU problem. This method is still computationally efficient. Unfortunately the Leaky ReLU does not solve all problems, as the predictions for negative input values are not consistent.

Swish A much newer method is the Swish activation function, developed by Google. According to the authors, Swish tends to perform better than ReLU [37]. They show this on deeper models across various challenging datasets. The paper does not compare the computational efficiency of the new activation function to ReLU. As the Swish function has more complicated computations, it is inherently more computationally inefficient than ReLU, but this might be worth the gain in accuracy.

Throughout the rest of this work, the ReLU activation function is used, unless stated otherwise. This decision is base on the limited use of the Swish method and the computational efficiency of the ReLU function.

2.1.2. Training a network

When training a neural network, the goal is to learn the values of the weights and biases such that the network makes good predictions. The to-be-learned dataset is split into a training set and a test set. The training set is used to learn the weights and biases, the test set is used to calculate the network's accuracy on unseen data.

Loss function

To train a network, a method is needed to quantify how good the neural network performs with the current parameters. To this end we can use a loss function. The most well-known loss function is the mean squared error (MSE) loss:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2,$$

where n is the number of training samples, y_{true} is the actual class the data point belongs to, and y_{pred} is the class prediction of the network.

For classification problems, cross entropy loss (CEL) is the most commonly used loss function. The mathematical formulation is as follows:

$$CEL = -(y_{true}\log(y_{pred}) + (1 - y_{true})\log(1 - y_{pred})).$$

The cross entropy loss increases as the predictions by the network diverge from the true class. When training a network, we try to minimize this loss.

Updating rules

Based on the result of the loss function, we want to alter the weights and biases to lower the loss. If we compute the gradient of the loss function with respect to the weights for a single data point, we can take a step in the direction of the gradient to lower the loss:

$$w_{new} = w_{old} - lr\left(\frac{\partial Loss}{\partial w_{old}}\right).$$

We can do the same for the biases:

$$b_{new} = b_{old} - lr\left(\frac{\partial Loss}{\partial b_{old}}\right).$$

lr is the learning rate, which is a hyper-parameter determining the size of the steps we take.

Backpropagation

Getting the gradients at each weight and bias in the network is done using a method called backpropagation. This method will be explained based on the notations in figure 2.5, which is a tiny neural network for explanatory purposes. Each neuron has a value a, which denotes its activation value (output of the activation function for that neuron). The superscript of the activation indicates in which layer the neuron is located, and the subscript indicates its position in that layer. The same goes for the biases b, and the weights w.

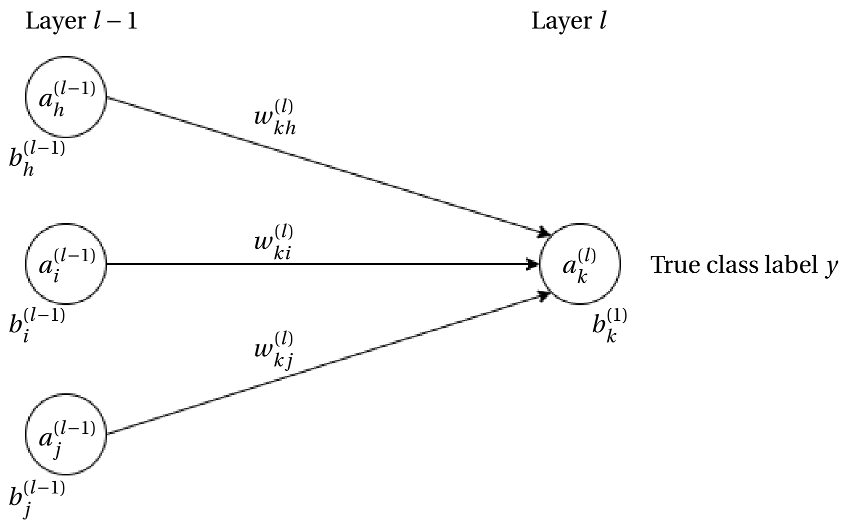


Figure 2.5: Example neural network with the corresponding mathematical notations

As the name backpropagation suggests, we start at the output layer of the network, and work our way to the left. For simplicity we use the mean square error loss function. As was shown before, the activation value of neuron k in layer l can be calculated as follows:

$$a_k^{(l)} = \sigma\left(\sum_i w_{ki}^{(l)} a_i^{(l-1)} + b_k^{(l)}\right),$$

where σ is the activation function of choice, for example ReLU. For convenience we also declare a variable z, which is the value before the activation function is applied:

$$z_k^{(l)} = \sum_i w_{ki}^{(l)} a_i^{(l-1)} + b_k^{(l)}.$$

For a certain data point, we can use the loss function (MSE) to compute the loss of the network:

$$Loss = \sum_{k=0}^{n_L - 1} (a_k^{(l)} - y_k)^2.$$

Now we can take a look at the gradient we actually want to compute. Using the chain-rule (visualized in figure 2.6), the gradient can be split up into three different partial derivatives:

$$\frac{\partial Loss}{\partial w_{ki}^{(l)}} = \frac{\partial z_k^{(l)}}{\partial w_{ki}^{(l)}} \frac{\partial a_k^{(l)}}{\partial z_k^{(l)}} \frac{\partial Loss}{\partial a_k^{(l)}}$$

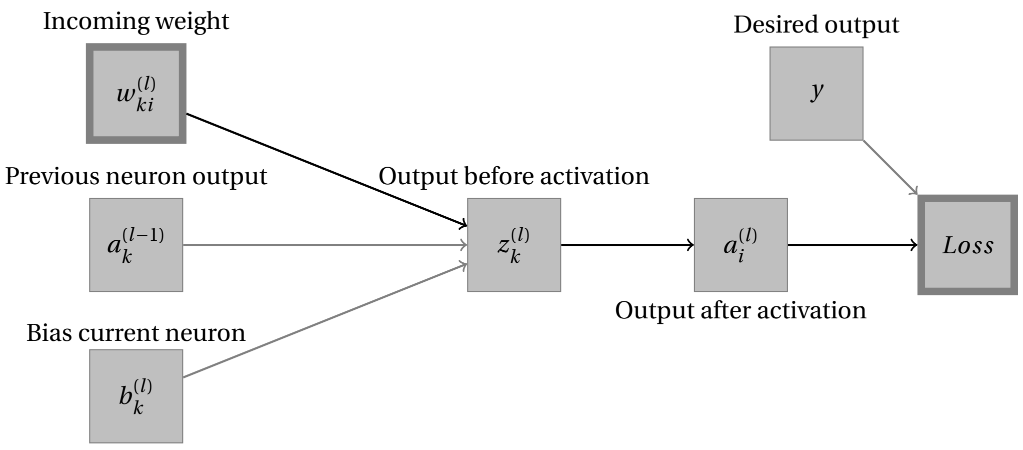


Figure 2.6: Visualization of the different components used in the chain rule.

These partial derivatives are easier to compute:

$$\frac{\partial z_k^{(l)}}{\partial w_{ki}^{(l)}} = a_i^{(l-1)},$$

$$\frac{\partial a_k^{(l)}}{\partial z_k^{(l)}} = \sigma'(z_k^l),$$

$$\frac{\partial Loss}{\partial a_k^{(l)}} = 2(a_k^{(l)} - y_k)$$

such that

$$\frac{\partial Loss}{\partial w_{ki}^{(l)}} = a_i^{(l-1)}\, \sigma'(z_k^{(l)})\, 2(a_k^{(l)} - y_k).$$

Similarly, we compute the gradient for the bias:

$$\frac{\partial Loss}{\partial b_k^{(l)}} = \sigma'(z_k^{(l)})\, 2(a_k^{(l)} - y_k).$$

The gradients are computed for each weight and bias present in the network, resulting in the gradient vector. This gradient vector will then be used to update the weights and biases according to the update formulas discussed before. This process is repeated for all the data points in the training set, until some condition is met. An easy and commonly used stopping criterion is setting a fixed number of epochs; one epoch is putting all training data points through the backpropagation method once. Another stopping condition that is commonly used is to stop training when the loss of the network is below a certain predetermined threshold.

2.2. Genetic algorithms

As mentioned before, there are numerous methods with which you can find adversarial examples. This work uses, amongst other things, a genetic algorithm to find adversarial examples. Genetic algorithms are based on the biological phenomenon of natural selection, which is about the change and adaption in heritable traits of populations of living organisms. Some organisms have a certain phenotype that makes their chances of survival higher. This phenotype will eventually get the upper hand in the reproduction of the organism, also known as *survival of the fittest*. A genetic algorithm uses operators inspired by natural selection, such as selection, crossover and mutation, to search for an optimal solution to the problem at hand.

2.2.1. Terminology

As we want to use the natural selection in a non-biological setting, we need to 'translate' the terminology. The process starts out with an initial population. In biology, this represents the heritable traits of each organism, in our setting it is a set of data points. Each data point in the population is an individual solution to the problem we want to solve. A single data point in the population is also called a chromosome. Each chromosome is build up out of genes, which in our case are the variables of a data point. An example/overview for binary data is shown in figure 2.7.

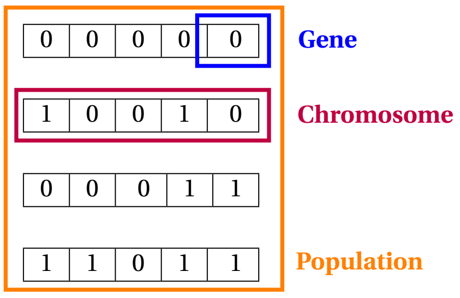


Figure 2.7: Overview of the terminology for a binary dataset with 4 data points with each 5 variables.

We also need a method to determine how likely a chromosome is to survive. This can be done by the use of a fitness function, which takes as input a single data point and outputs its score. The higher the fitness, the more likely the data point is chosen for reproduction. Fitness functions are very similar to loss functions, which were discussed earlier this chapter. Maximizing a fitness function is in essence the same as minimizing a loss function. All different types of loss functions explained before can therefore also be used in a genetic algorithm.

2.2.2. Selection

To find the optimal chromosome composition for an organism, we can simply let the original population evolve and check the genes multiple generations later as we know that the fittest chromosomes are most likely to survive and reproduce. In the selection process of the genetic algorithm, parents are selected for reproduction. We want the offspring to be as fit as possible, thus we choose the data points with the highest results from the fitness function as the parents. The number of parents selected for reproduction is a predetermined number, and thus a hyper-parameter of the algorithm.

2.2.3. Crossover

The next step is for the parents to combine their 'DNA' and create offspring. This process is called crossover. Originally this was often done using a randomly chosen crossover point. After the crossover point is reached, all genes coming thereafter are switched with the genes of the other parent. An example for binary data is shown in figure 2.8. It is also possible to randomly select multiple crossover points, or apply uniform crossover where for each gene the parent is randomly chosen.

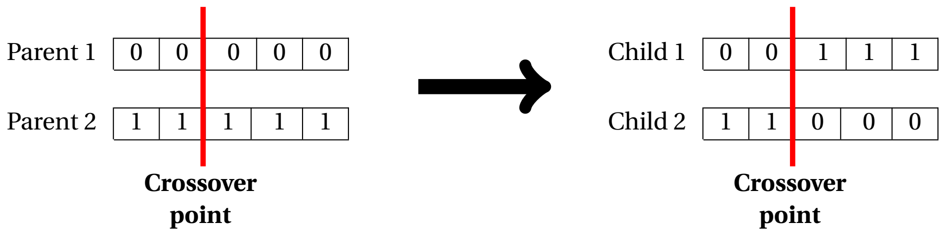


Figure 2.8: Crossover for binary data. The crossover point is after the second gene, causing each gene thereafter to switch with the gene of the other parent.

For non-binary data, there are a lot more ways to do crossover. When dealing with real numbers, one could for example take the average value of both parents for each gene. Another possibility is to give the chromosome or genes of one parent higher weights, causing the child to be more like this parent. BLX-a, or Blend Crossover, is a method for doing crossover for which it has been shown that it has a good search ability [16]. This method uses the following formulas:

$$d_i = |P1_i - P2_i|$$

$$X^{min} = min(P1_i, P2_i) - \alpha d_i$$

$$X^{max} = max(P1_i, P2_i) + \alpha d_i$$

where $P1$ and $P2$ represent the two parents, and i indicates the index of the gene. α is a positive parameter, for which 0.366 is a good value [43]. For each gene, the offspring value is chosen randomly from the interval $[X_i^{min}, X_i^{max}]$ following the uniform distribution.

2.2.4. Mutation

The final step is mutation. The genes of the newly created offspring can be subject to mutations with a low random probability. For the binary data shown before, this would mean that for each gene the bit is flipped with a certain predetermined probability. The process of mutation is important, as it maintains diversity in the population and provides possibilities to escape from local optima. This can also prevent premature convergence of the algorithm.

For real-valued data there are, as can be expected, more options for creating mutations. Some popular methods are:

Uniform A certain predetermined percentage of genes will be mutated, the genes are randomly determined. The mutated genes are replaced with a uniform random value between predetermined bounds for that gene.

Non-Uniform The mutation-rate or mutation-impact is lowered as generations pass. In the early stages of the algorithm it prevents stagnation, and in the later stages this method allows for fine tuning.

Gaussian When using a Gaussian mutation, a randomly chosen value from the Gaussian distribution is added to the chosen gene. If the result exceeds the predetermined bounds, the result is clipped.

After the mutations, the creation of the offspring is finished. Depending on the preferences of the coder, the whole population can be replaced with the new offspring, or the new population is a combination of the offspring and some very fit parents. With the new population, the whole process can be repeated until there are no significant changes anymore in the offspring produced, or when a predetermined number of generations is reached.

2.2.5. Evolution

In a genetic algorithm the previously explained steps are repeated a predetermined number of times, called the number of generations. In each generation the best solutions are combined to create possibly even better solutions, and the mutation phase allows for exploration of unseen solutions. A visualisation of the entire algorithm is shown in figure 2.9. The example shows the evolution of the population, in which the properties of the fittest individuals prevail and are magnified.

2.2.6. Uses

Genetic algorithms are popular due to the understandable concept and its good performance. Genetic algorithms are very suitable for multi-objective optimization and it does not depend on the computation of derivatives. Furthermore, genetic algorithms are relatively robust to local minima and maxima compared to other optimization methods, as the algorithm is stochastic. Genetic algorithms work well in noisy environments and they are suitable for discrete as well as continuous problems.

Genetic algorithms are widely used in different application areas [7]. A well-known application of genetic algorithms is in the field of machine learning [19], but also other sectors have benefited from this method. In economics they are, for example, used to solve the economic dispatch problem [11]. In aeronautics, genetic algorithms even have been used for helicopter conceptual design [12]. In this work we further explore the application of genetic algorithms in neural networks.

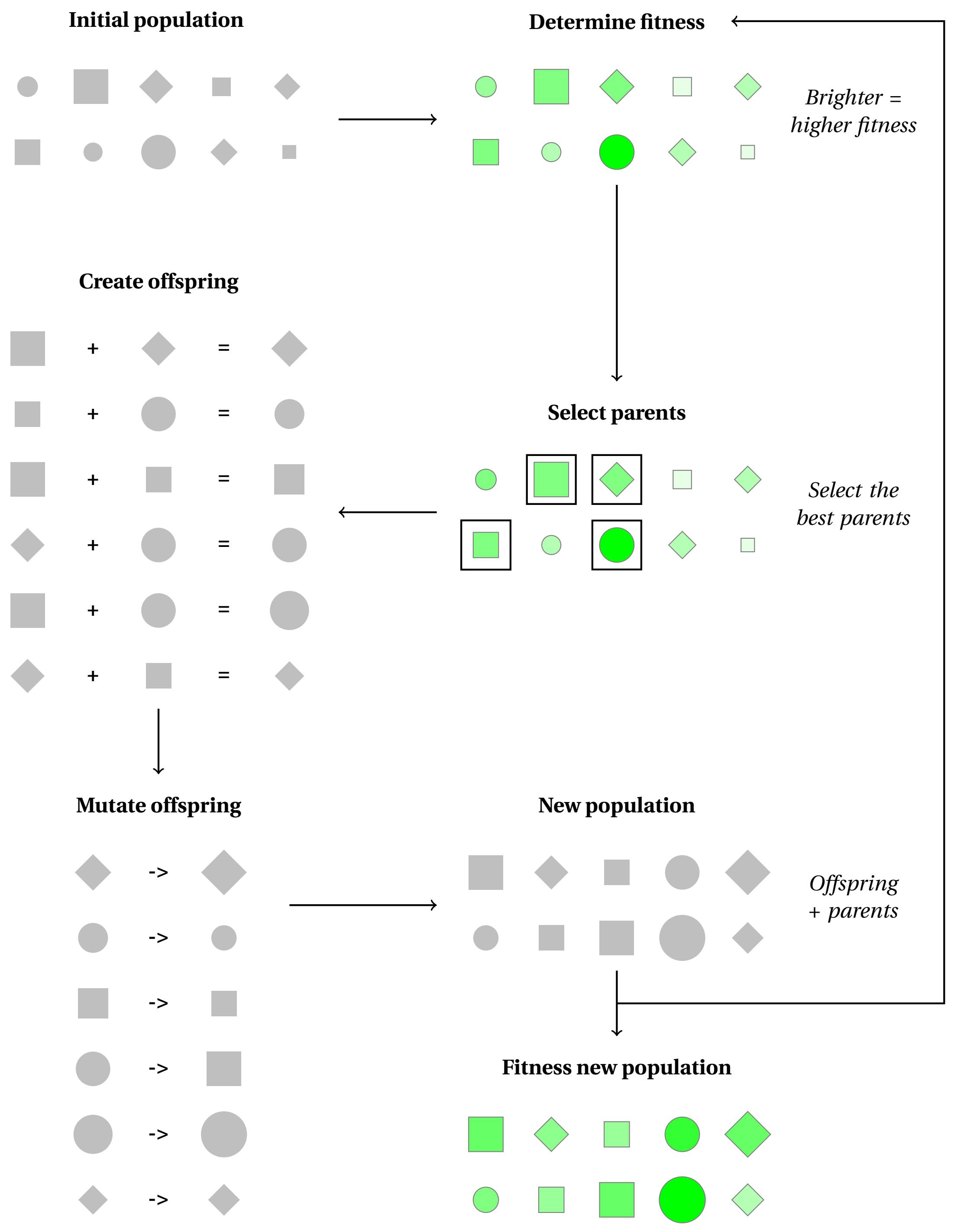


Figure 2.9: Overview of a genetic algorithm. In this specific example the fitness is influenced by shape and size, where being large and/or a circle is best. Throughout the generations of the algorithm, the population evolves and will consist of more large shapes and circles.

3 Related Work

In the previous chapter the basics of neural networks were explained, and we saw that the predictions made by neural networks can be useful in many industries. There are, however, attackers who want to fool neural networks such that they predict the wrong output class. This can be done by adding a small perturbation to a data point, causing misclassification by the network, but no noticeable difference in the data point for the human eye. The resulting data point is called an adversarial example [20]. A classic example of an adversarial example for an image as input data is shown in figure 3.1. Even though images are most often the target of this type of attack, adversarial examples can be created for many different types of data.

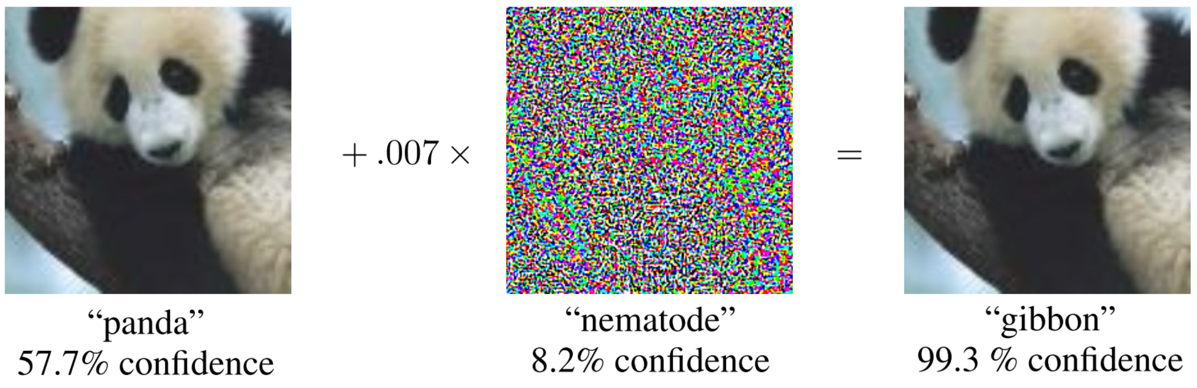


Figure 3.1: A visual representation of an adversarial example from Goodfellow et al. [20]. The picture on the left is (correctly) classified as a panda by GoogLeNet, with 57.7% confidence. After adding a small, carefully designed perturbation, GoogLeNet misclassifies the panda as a gibbon with a very high confidence of 99.3%. The difference between the images is not visible to the human eye.

Neural networks and adversarial examples are a popular research topic. As such, this chapter provides an overview of the relevant related work. Firstly, the current state-of-the-art white-box attacks are explained and evaluated. Thereafter, we take a look at the limited existing research into (the use of) neuron activation values, and consider the network coverage of existing methods. Finally, our findings are concluded by summarizing the research gap.

3.1. White-box attacks

Attacks on neural networks can be distinguished by their type; It is either a white-box method or a black-box method. For a black-box attack no knowledge about the model architecture is needed, the attack is solely based on the inputs and outputs of the model. A white-box method uses the architecture and/or parameters of the network for the attack. Unfortunately, white-box methods cannot always be applied, as the underlying model needs to be accessible and known. White-box methods are, however, very interesting as they allow the attacker to look for vulnerabilities in the model and use these to carry out a more targeted attack. The upcoming sections discuss some of the current state-of-the-art white-box attack methods.

3.1.1. L-BFGS

One of the first methods for creating targeted adversarial examples was the box-constrained Limited-memory Broyden-Fletcher-Goldfarb-Shanno attack (L-BFGS) [42]. This method tries to find an adversarial example x' for input x with classified label l by minimizing the L_2 distance. This is a hard problem to solve, so the problem is approximated by the following formula:

$$\min[c * ||x - x'||_2^2 + L(x', l')]$$
$$\text{such that } C(x') = l' \text{ and } x' \in [0,1]^n$$

By minimizing $L(x', l')$ (where L is a loss function), we try to obtain an adversarial example x' such that the classifier gives x' the desired label l'. By minimizing $||x - x'||_2^2$ we try to obtain an adversarial example x' that is close to the original input x. Line search is used to find the value of c for which the equation yields the minimal value.

3.1.2. FGSM

Even though the L-BFGS attack can create close adversarial examples, it it not used often since the algorithm is very slow. Hence, the Fast Gradient Sign Method (FGSM) was developed [20]. Instead of finding a very close adversarial example, it was primarily designed to be fast. Another difference with the L-BFGS attack is the distance metric: while L-BFGS is optimized for the L_2 distance, FGSM is optimized for the L_∞ distance. Given input x, FGSM tries to find adversarial examples using the following formula:

$$x' = x - \epsilon * \text{sign}(\Delta L(x, l))$$

$\Delta L(x, l)$ is the gradient of the loss function, and can be computed by using the back-propagation algorithm. By moving the input x in the direction of this gradient, the probability of the adversarial example x' being classified to label l is decreased. The value of epsilon (ϵ) determines the magnitude of the perturbation, hence epsilon is usually a very small value such that the perturbation remains undetectable. For every data point in the input (e.g. for every pixel in an input image) the moving direction is determined using the gradient, after which all data points are moved at the same time. An example of the course of this method is shown in figure 3.2.

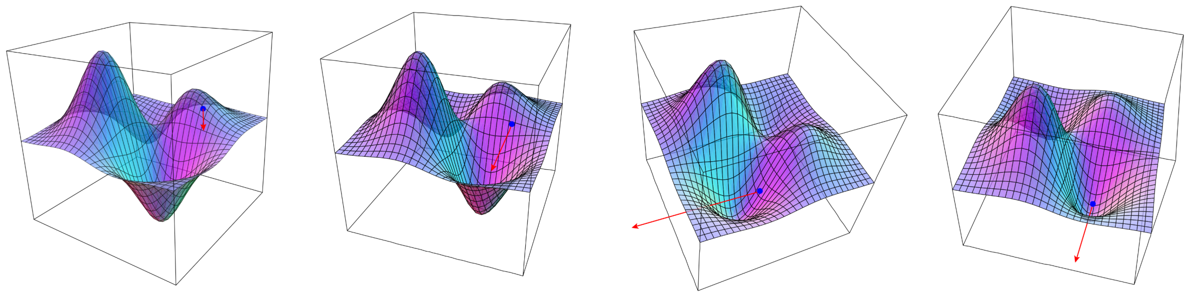


Figure 3.2: A visualization of gradient descent. The surface represents the loss function, which we try to optimize. By taking a step in the direction of the gradient, shown by the red arrows, we hope to get closer to the (local) optimum.

3.1.3. BIM

Later, the FGSM was extended such that perturbations are added in multiple steps. This method is known as the Iterative FGSM or Basic Iterative Method (BIM) [25]. To avoid large changes in a data point, the step of size ϵ in the direction of the sign of the gradient is substituted for multiple smaller steps of size α, which are clipped by the value of epsilon:

$$x'_0 = 0$$
$$x'_i = x'_{i-1} - \text{clip}_\epsilon(\alpha * \text{sign}(\Delta L(x'_{i-1}, l'))$$

By using the clip, in each iteration the change to the adversarial example x'_i is limited, and it is ensured that x'_i stays within the L_∞ ϵ-neighbourhood of input x. According to [25], BIM can produce better results than the regular FGSM. However, they also found that FGSM is more robust to photo-transformation than iterative methods.

3.1.4. RFGSM

Shortly after BIM was published, a new variant on the FGSM was introduced: the random-step FGSM (RFGSM) [44]. This method is a simple alteration to the original FGSM, prepended by a small random perturbation. The method uses the following formula:

$$x' = x + \alpha * \text{sign}(N(0^d, I^d)) + (\epsilon - \alpha) * \text{sign}(\Delta L(x + \alpha * \text{sign}(N(0^d, I^d)), l))$$

N represents the noise, i.e. the small random perturbation. A new parameter, α, is introduced, which needs to be smaller than ϵ. The addition of a small random noise can be helpful as the true direction of the highest gradient is sometimes disguised by sharp curvatures in the function. The authors also find that the random perturbations decrease the transferability of adversarial examples.

The RFGSM is computationally efficient, and the authors claim that the method significantly outperforms FGSM, with and without adversarial training.

3.1.5. JSMA

This method uses Saliency Maps [40], which show the impact of each data point of input x on the classification. An example of a Saliency Map for an imagery input is shown in figure 3.3. The Jacobian-based Saliency

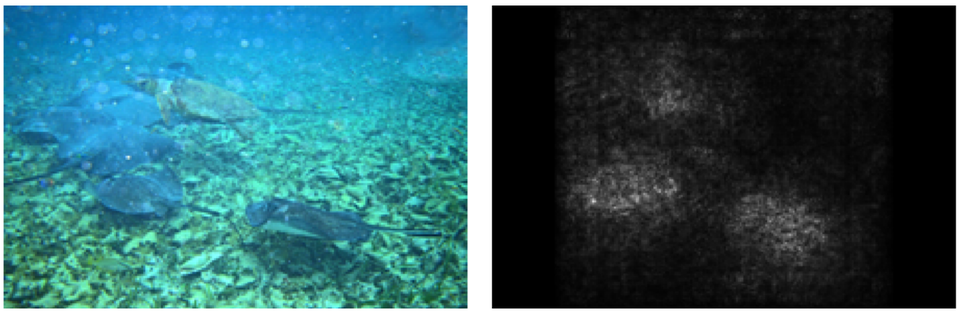


Figure 3.3: Original image (left) and its Saliency Map for the top-1 predicted class. The Saliency Map shows the pixels that were most important for the classification. [40]

Map Attack (JSMA) [33] uses these Saliency Maps in selecting the most important data points for a certain target class, which are then altered to decrease the likelihood of correct classification. This process is repeated until the threshold for the amount of pixels to change is reached or the attack succeeds in changing the classification. The Saliency Map can be represented by the following formula:

$$S^+(x_i, c) = \begin{cases} 0 \text{ if } \frac{\partial F_c(x)}{\partial x_i} < 0, \\ 0 \text{ if } \sum_{c' \neq c} \frac{\partial F_{c'}(x)}{\partial x_i} > 0, \\ -\frac{\partial F_c(x)}{\partial x_i} * \sum_{c' \neq c} \frac{\partial F_{c'}(x)}{\partial x_i} \text{ otherwise} \end{cases}$$

The left part of the equation, $S^+(x_i, c)$, shows how much x_i positively correlates with class c and also negatively correlates with all classes other than c ($c' \neq c$). If the correlation with c is not positive, or if the correlation with $c' \neq c$ is not negative, the saliency is reset to zero. The F in the formula represents the output of the softmax layer. An attacker can exploit this Saliency Map by picking a target class t for input x, not matching the correct class label. The saliency for the target class can be computed, which allows for selecting data points that are important for classifying to class t and not to the correct class label. By increasing the value of these data points, the likelihood of classification to target class t increases, possibly causing misclassification. This method showed good results, but has high computational cost which makes the algorithm slow and less attractive to use.

3.1.6. DeepFool

DeepFool [32] is a method to create non-targeted adversarial examples, optimized for the L_2 distance metric. The method starts by 'assuming' linearity, such that each class is separated by a hyperplane (resulting in a polyhedron). In every iteration the input is perturbed such that the resulting example is moved to the closest boundary of the polyhedron. As the linearity was only an assumption since neural networks do not have linear decision boundaries, DeepFool takes a step in the direction of the simplified solution. This is repeated until it finds a true adversarial example.

DeepFool shows some very promising characteristics, as it beats the previously mentioned white-box attacks on different aspects. Compared to L-BFGS, DeepFool is efficient and the adversarial examples that are produced by DeepFool are closer to the original input. For FGSM and JSMA DeepFool also created adversarial examples with smaller perturbations.

3.1.7. C&W Attack

The Carlini and Wagner (C&W) attack [10] formulates the misclassification of the adversarial example x' in a way that is better for optimization. They define a function $f(x')$ such that $f(x') \geq 0$ if and only if $C(x') = l'$. The idea is that this function $f(x')$ tells us how close we are to x being classified as class label l'. The authors evaluated seven different implementations for this function $f(x')$, with the following implementation yielding the best results:

$$f(x') = \max(\max_{i \neq l'}(Z(x')_i) - Z(x')_{l'}, -\kappa)$$

Here $Z(x')_i$ represents the logit of class i for (adversarial) example x', hence $\max_{i\neq l'}(Z(x')_i)$ is the most probable non-target logit. $Z(x')_{l'}$ then represents the probability of the target class l', i.e. how probable is it that x' is misclassified to target class l'. Say we have an image of a cat that we want our network to classify incorrectly as a dog, as shown in figure 3.4. x' is the perturbed image of the cat we feed to the network, the original class is $class2 : cat$ and the target class is $class3 : dog$. $Z(x')_{l'}$ is the logit of the target class, which is the input of the dog class to the (Log)Softmax function at the final layer of the network. The most probable

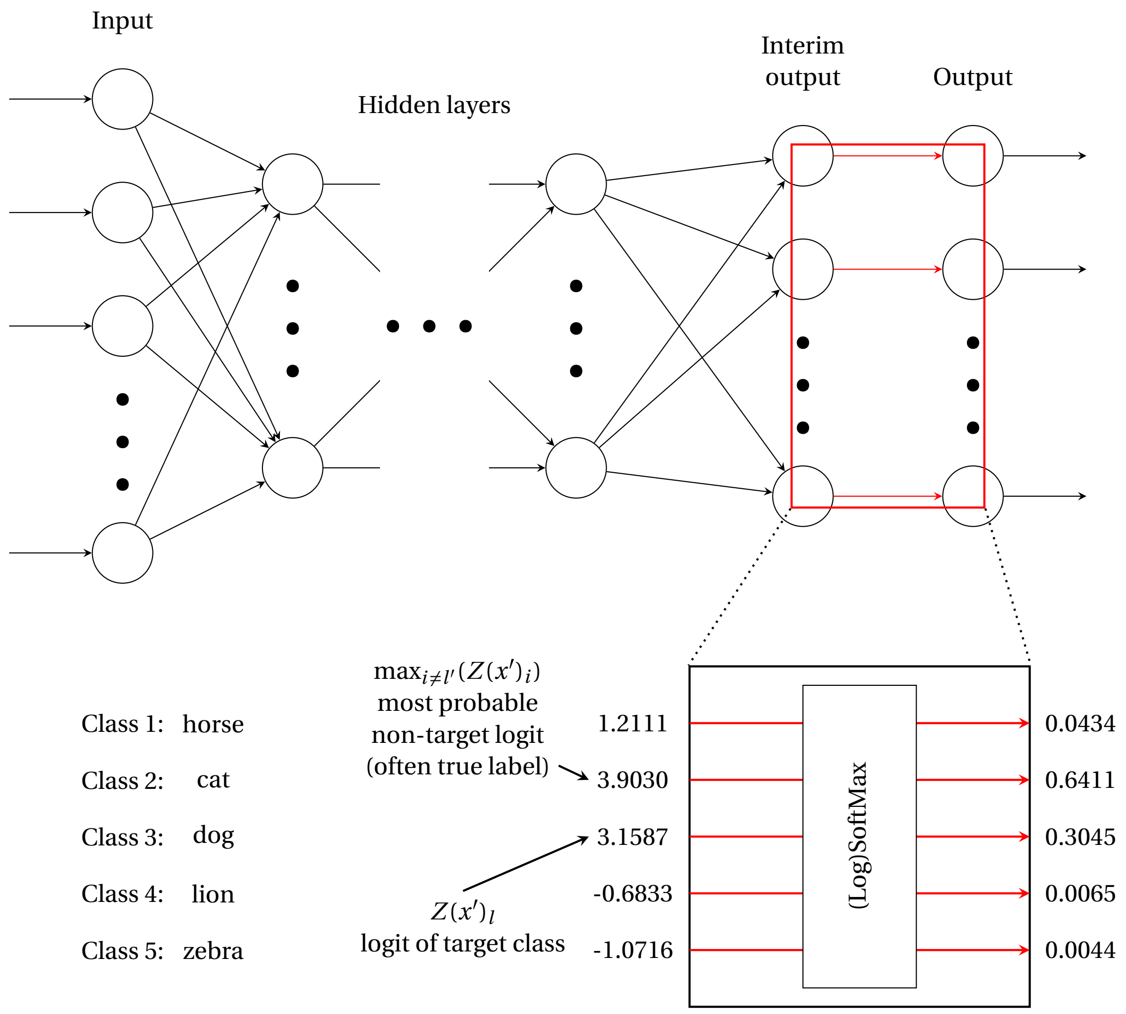


Figure 3.4: Visualization of the neural network values that are used by the C&W attack. Neural networks often use a (Log)Softmax activation function in the output layer; C&W uses the inputs to this activation function. This example considers a toy network with five output classes as shown on the left.

non-target logit, $\max_{i \neq l'}(Z(x')_i)$, is the highest logit excluding the logit of the target class. As in this example, this is often the logit of the true class label (in this example the logit of $class2 : cat$).

Putting this together, the left argument of the outer max shows the difference between what class label the classifier most probably predicts for input x', and what we want the classifier to predict. The parameter κ is a tunable constant that controls the confidence in the adversarial examples. In the original work the authors set the value of κ to 0. Now the original problem of finding a good adversarial example x' can be reformulated as follows:

$$\min_{x'} ||x - x'||_2^2 + c * f(x')$$

$$\text{such that } x' \in [0,1]^n$$

The value of the constant c is determined by binary search. To get rid of the box-constrain, the method *change of variables* is used, where instead of optimizing over the perturbation we optimize over w: $x' = \frac{1}{2} * (\tanh(w) + 1)$. This formulation allows for the use of general optimizers in deep learning. The final optimization problem is as follows:

$$\min_{w} ||\frac{1}{2} * (\tanh(w) + 1)||_2 \; + \; c * f(\frac{1}{2} * (\tanh(w) + 1))$$

The results show that C&W attack creates better adversarial examples than FGSM and JSMA. The C&W attack is considered to be a strong attack, but a big downside is that it is computationally expensive.

3.1.8. Overview & discussion

The presented white-box attack methods are summarized in table 3.1, where for each attack it is shown what model information it uses, whether it is a computationally efficient method, and how its performance compares to the other presented methods. As can be expected, the table shows a trade-off between performance

	Network information used	**Computational efficiency**	**Performance**
L-BFGS	Gradient of the loss function with respect to the input	Expensive linear search	One of the first methods for creating adversarial examples
FGSM	Sign of the gradient of the loss function with respect to the input	One step gradient update, more efficient than L-BFGS	Designed to be fast, rather than optimal. Therefore does not produce the minimal adversarial perturbations
BIM	Sign of the gradient of the loss function with respect to the input	Similar to FGSM	Produces superior results to FGSM
RFGSM	Sign of the gradient of the loss function with respect to the input	Similar to FGSM	Produces superior results to FGSM
JSMA	Gradient of the output of the softmax layer with respect to the input	High computational costs	Very good results with minimal modifications
DeepFool	Differences in the gradient of the model output of the actual class with respect to the input and the gradient of the output for other classes with respect to the input	More computationally efficient than JSMA	Smaller perturbations than FGSM and JSMA
C&W	Output of the network before the softmax layer	Computationally expensive due to its extensive search for the most effective perturbation	Can compromise defensive distilled models on which L-BFGS and DeepFool fail to find adversarial examples

Table 3.1: Overview of state-of-the-art white-box methods to create adversarial examples, including the network information the method uses, the computational efficiency and the performance compared to other methods.

and efficiency, as better performance often comes at a greater computational cost. It is therefore not possible to proclaim which method is the best in all cases.

Interestingly enough, the state-of-the-art white-box methods seem to use a limited amount of network information. The first three methods, L-BFGS, FGSM, and BIM, only use the gradient of the loss function with respect to the input. The loss function is calculated with the results from the output layer, and thus no other model information about previous layers and neurons is used. For JSMA, DeepFool and C&W this is very similar: these methods only use the output of the softmax layer to calculate its gradient with respect to the input, the network output to compute gradients with respect to the input, and the output of the network before the softmax layer respectively. No other parameters or network properties are taken into account.

Neural networks are known for their difficulty of interpretation. It is a complex task to extract what a network has exactly learned and why. In this area of research there are a lot more steps to be taken, and it is likely that we can extract more meaningful information from trained models than the current state-of-the-art methods do.

3.2. Neuron activation values

An example of network information considering the whole network, instead of just one (output) layer, are neuron activation values. Each neuron in the network has a neuron activation value for each possible input. The neuron activation value is the final output of a neuron, which is the result of the activation function. An example of different neuron activation patterns is shown in figure 3.5, which visualizes the intuition that neuron activation values contain meaningful information.

To our knowledge, only limited research has been done on the use of neuron activation values, and there are few works exploiting this model information. The known applications for neuron activation values will be discussed next.

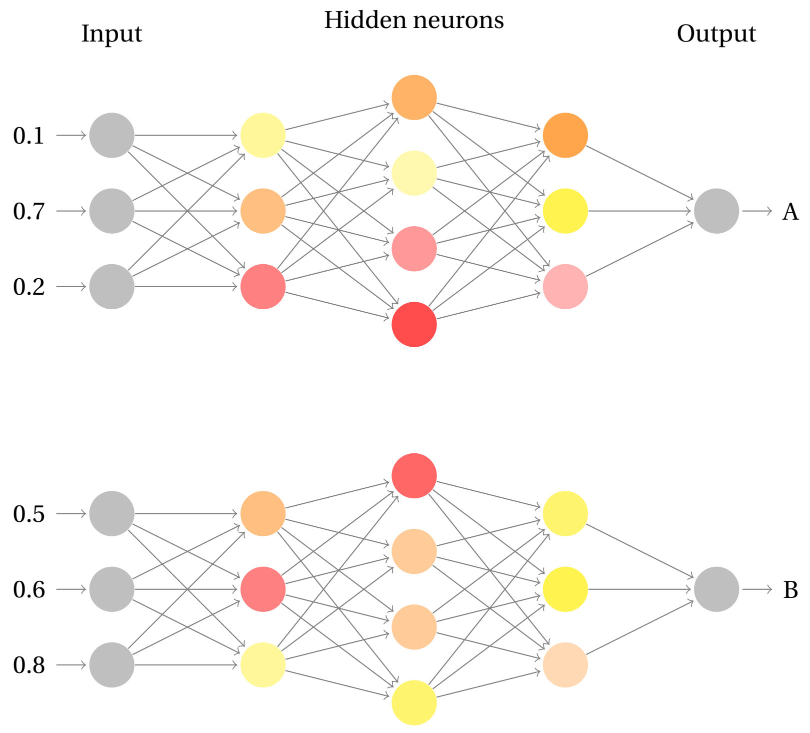


Figure 3.5: Example neural network for two different inputs. The example network takes as input 3 scalar values between 0 and 1, and outputs class A, B or C. The color of the hidden neurons show the height of their activation value. We see that the two different network inputs lead to very different neuron activation. These neuron activation patterns contain valuable information about the class label, but this information is not used in methods that only use the output of the network.

3.2.1. Priority neuron network

In 2018, researchers proposed a new resource-aware neural network, called the priority neuron network (PNN) [3]. In the paper a training algorithm is proposed which uses regularization techniques. With these techniques the neuron activation values are constrained and a priority is assigned to each neuron. For each neuron, its ordinal number is used as the priority criteria, as the neuron priority is inversely proportional to its ordinal number in the layer. This idea was inspired by the multirate prediction idea in [8]. The priority criteria imposes a relatively sorted order on the activation values.

Even though the neuron activation values are not used directly, but only indirectly influenced, this was one of the first works we could find which specifically uses the neuron activation values. Since the method was mainly developed to decrease training time and memory usage by creating a reconfigurable network at runtime, the neuron activation values are not used to get a deeper understanding of the inner workings of the neural network and increase its performance or performance of adversarial examples.

3.2.2. Large margins

In machine learning, one way to determine the loss of a classifier is by looking at the minimal distance of the prediction to the decision boundary, called the margin. For a linear classifier, for example, the euclidean distance to the separating hyperplane can be calculated and used as the margin. A two-dimensional example is shown in figure 3.6.

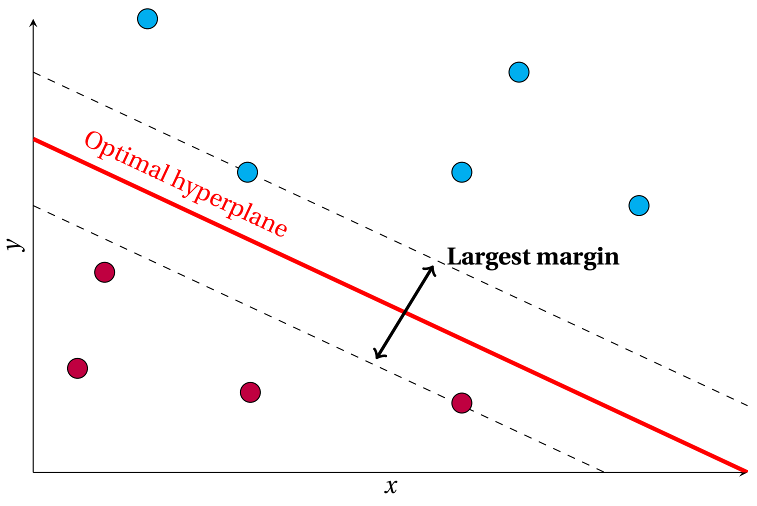


Figure 3.6: Two-dimensional example of a large margin classifier. The red and blue data points represent different output classes, which are separated by the optimal hyperplane which has the largest possible margin between the two closest points of each class.

In more complex classifiers, such as a neural network, it is a lot harder to define a decision boundary. Widely used loss functions only use a decision boundary and margin at the final output layer. In other words, the margin is based on the distance of the input data to the decision boundary of the output. In figure 3.6, x and y combined give the input data, and the output class is represented by the color of the data point. Hence, the largest margin is an input-output association.

Elsayed et al. [15] introduced a new loss function, called large margin. In their work they consider the neuron activation values as some intermediate representation of the data. Hence, they can now calculate the margin or loss at any neuron by replacing the input data with the intermediate representation. The intermediate margins are combined to formulate a margin loss at each layer. The authors claim that neural networks trained with their new loss function perform well in a number of practical scenarios compared to baselines on standard datasets.

3.2.3. Generalization gap

Jiang et al. expanded on the work of Elsayed et al. by using the (approximated) intermediate decision boundaries to predict the generalization gap [22], which is the difference between the training accuracy and the test accuracy at the end of training. They define the *margin distribution* at each layer l, which is the distribution of the distances to the decision boundary at each layer for all the training data. The distribution is normalized to make it invariant to scaling, after which a compact signature of the distribution is created based on the quartiles, since a compact signature is easier to analyse. Using this signature, a method to measure the generalization gap is formulated. The results show that the method has high predictive power of the generalization gap and is thus successful. In the paper the researchers also state that the usage of hidden layers (i.e. the neuron activation values) is crucial for the predictive power.

3.2.4. Overview & discussion

As was mentioned before, the research into the significance of neuron activation values remains limited. The works discussed above show that intermediate network information can be valuable in classification tasks for neural networks.

In the priority neuron network, neurons are prioritized based on their ordinal number. The reason for this specific prioritization remains unexplained, but it does inspire us to further investigate the difference in importance of neurons. More specifically, it would be interested to find out whether certain neurons are more important and more often activated for certain output classes. This could lead to distinguishable neuron activation patterns, as was visualized in figure 3.5.

In the work about large margins and the generalization gap we also see a promising future for the use of intermediate network information. They create a loss function which takes into account the loss margins at each hidden layer, which seems to perform well. Hence, this method encourages the use of intermediate information in neural networks.

Even though the works discussed above use the neuron activation values in some way to, for example, create a loss function or predict the generalization gap, the works do not elaborate much on the information contained in the neuron activation values and why these values are important for the classification of the neural network. With this knowledge, however, one may be able to exploit it and significantly improve the network's performance or the performance of adversarial examples.

3.3. Network coverage

In the process of creating adversarial examples it is not only important to utilize the available network information, but it can also be valuable to consider the network coverage of the algorithm. Logically, with a wider search of possible inputs one has a higher chance of finding adversarial examples. Since most well-known and used methods to find adversarial examples use gradient descent, it is interesting to look at some downsides of gradient descent (and see if we can overcome these). As gradient descent always takes a step into the direction of the negative of the gradient, it is possible that the algorithm converges to a local minimum instead of a global minimum [28]. Even though adversarial examples may already be found in a local minimum of the loss function, it can still be valuable to be able to search for adversarial examples in other parts of the input space. Perhaps a better adversarial example can be found, or for an adversarially trained neural network where the local minimum does not provide a successful adversarial example, the wider search may succeed in finding an adversarial example. A wider search can also find inputs for which the network is clueless, resulting in unpredictable behaviour.

3.3.1. Neuron coverage

Most published methods to find adversarial examples do not consider or elaborate on the covered search space. Recently, as neural networks are increasingly deployed in critical domains, the need for testing neural networks rose and some methods that consider coverage of the input space were developed. Pei et al. [35] created the first white-box testing framework for systematically testing deep learning systems. The main testing metric of this system is neuron coverage, where a neuron is covered when its activation value is above a predetermined threshold. The method also wants to maximize the coverage of behavioral differences between similar deep learning systems. These metrics are combined into a joint optimization problem, which is solved by an efficient gradient-based algorithm. This method is a tool for evaluating a trained network with respect to its neuron coverage, finding incorrect corner case behaviors. The authors suggest to use these adversarial examples for retraining to improve the accuracy of the model.

3.3.2. Neuron selection

Since then, several coverage metrics for neural networks have been introduced [24, 29, 41]. A recent work [27] claims to be more effective than the previously published white-box and grey-box testing techniques, for coverage as well as finding adversarial inputs. Their technique is based on a neuron selection strategy, where the top neurons are selected by scoring their feature vector. This feature vector can contain constant features and variable features, both boolean. A constant feature cannot change during testing; An example of a constant feature is whether the neuron is located in the front 25% of the layers. A variable feature may change during testing, for example whether the neuron has been activated when an adversarial input was found. The paper considers 29 different features, which are scored in a way that the top features indicate properties that selected neurons should have to increase the coverage (and bottom features indicate properties that selected neurons should not have). In this manner, the neuron selection strategy adapts to the neural network at hand. With this technique they were able to reach a remarkably higher coverage than existing methods across all metrics and models.

3.3.3. Overview & discussion

Testing a neural network is not yet a standard, and it is often forgotten about. The wide usage of neural network testing is also hindered by the difficulty to produce meaningful coverage criteria. This is because the control flow of the network does not represent all the information that is learned by the network during training, making it unclear how structural coverage criteria should be defined [5]. Untested or poorly tested neural networks are likely to produce unwanted results to unseen input data. This can be exploited by an attacker, by covering a larger part of the input space and finding adversarial examples.

Current research into network coverage primarily considers methods to test the coverage of neural networks, based on the idea that neural networks with poor coverage at training time are more vulnerable to adversarial examples. Attackers can, of course, also exploit such vulnerabilities by considering a wider search space when looking for adversarial examples. Most well-known and state-of-the-art methods, however, us gradient descent, for which the network coverage is limited. Hence, these findings inspire us to look at methods for creating adversarial examples beyond gradient descent, with higher network coverage.

3.4. Research gap

Having evaluated the previous works in the fields of white-box methods, neuron activation values, and neuron coverage, we identified three research gaps:

- The state-of-the-art white-box methods seem to use a limited amount of network information. Specifically, most methods only do some calculations with the (gradient of the) output layer, neglecting other model information about previous layers and neurons.
- The research into the significance of neuron activation values remains limited. It remains uncertain what information is exactly contained in the neuron activation values and why these values are important for the classification of the neural network.
- Most training methods as well as methods for finding adversarial examples do not consider or elaborate on the covered search space, possibly resulting in vulnerabilities or missed opportunities for finding adversarial examples.

4 Hidden Neuron Activations

In this chapter we dive into the neuron activation values of the hidden neurons. After two datasets are introduced, we visualize the neuron activation values to obtain a better understanding of their meaningfulness. In section 3 neuron activation values of different class labels are compared to find out whether a predictive measure follows from it. Finally in section 4 we present a method for finding adversarial examples based on difference in neuron activation values.

4.1. Datasets

We would like to find out and show what information is contained in neuron activation values and how we can use this information to our advantage. To do this, we make use of two well-known datasets, which are small enough to create some meaningful visualizations.

Iris flower dataset

The Iris dataset [4, 17] contains measurements of three different Iris species: *Iris Setosa*, *Iris Versicolour* and *Iris Virginica*. For each specie, 50 data samples are presented in the dataset. Besides the class label (specie), there are four features: sepal length, sepal width, petal length, and petal width. Using these features, different models have shown to perfectly predict the class label of test data [31]. A scatter matrix of the Iris dataset is shown in figure 4.1.

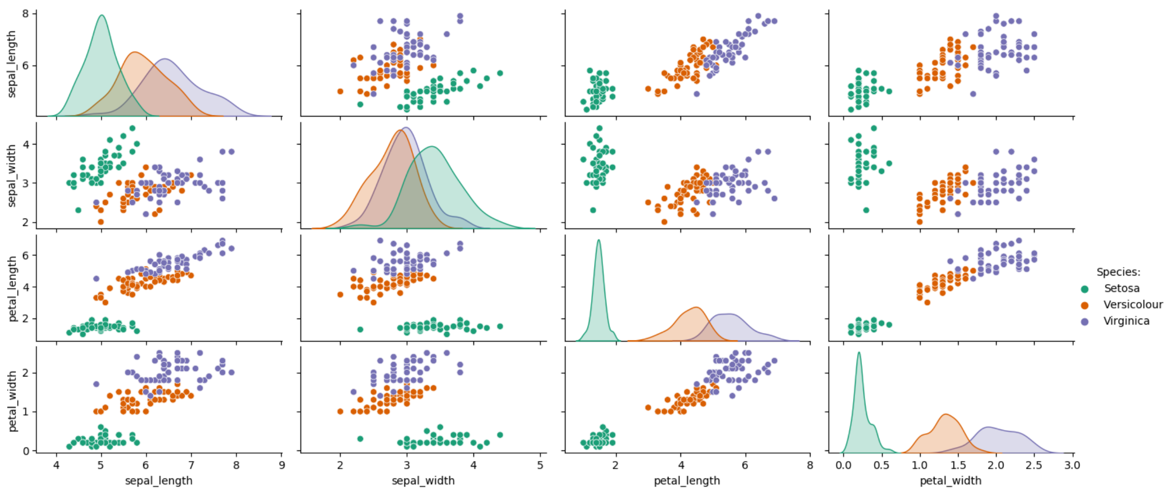


Figure 4.1: Overview of the Iris flower dataset

Pima Indian diabetes dataset

The second dataset that is used in this chapter is the Pima Indian diabetes dataset [38]. This is a dataset from the National Institute of Diabetes and Digestive and Kidney Diseases, in which human features are measured to find correlations between (the combination of) these features and having diabetes. The features are: number of pregnancies, glucose tolerance, diastolic blood pressure, triceps skin fold thickness, insulin, Body Mass Index (BMI), Diabetes Pedigree Function (DPF), and age. The dataset contains features of 768 people, of which 268 have a positive diagnosis of diabetes. The others (500) have tested negative for diabetes. A scatter matrix of the Pima Indian diabetes dataset is shown in figure 4.2.

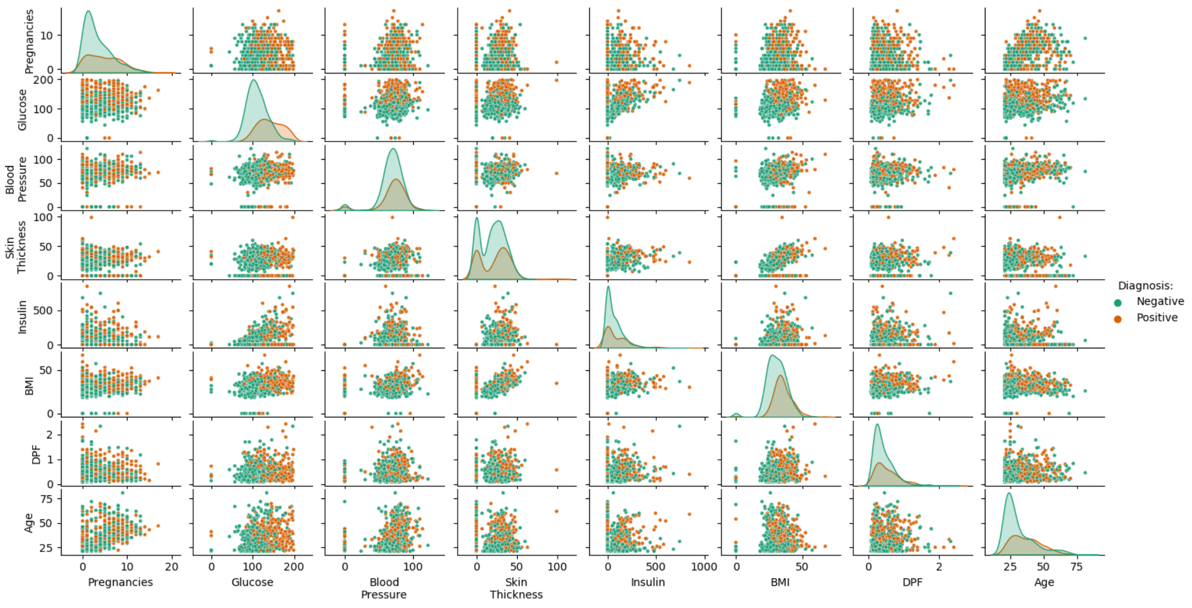


Figure 4.2: Overview of the Pima Indian diabetes dataset

Whereas the Iris dataset is simple enough such that some models can achieve perfect accuracy scores, the Pima Indian diabetes dataset is more complicated. To our knowledge, the highest accuracy score of 98.35% was obtained by deep learning using five-fold cross-validation [6].

4.2. Visualizing neuron activations

As we want to get a clearer picture of neuron activation values, this section visualizes the neuron activation values. We look into the difference in activation values for different class labels in the same neuron, to find out whether they are distinguishable. This is done for different network sizes, after which a conclusion and intuition are formulated.

4.2.1. Single hidden neuron

To investigate the meaningfulness of neuron activation values, we start by training a very small neural network: a network with only one hidden neuron. We train the network on the Iris dataset with a test split of 20%. The prediction accuracy on training data as well as test data is 97%. In figure 4.3 the activation values in the single hidden neuron is shown for all data points. The corresponding density distributions are shown in figure 4.4.

As figures 4.3 and 4.4 sort the neuron activation values by the true class label, we can see the difference in activations for each class. For data points with true class 0, all activations are 0 (each data point is colored purple, which corresponds to 0 according to the color-legend). For data points with the true class 1, we see that the average activation value is about 3. Every data point, however, generates a different activation value, creating outliers with activation values below 1 and above 5. For data points with true class 2, we see an average activation value of just above 6. Here we can also see some outliers below 5, such that the two density distributions overlap and correctly predicting the class label becomes harder.

The overlap in the neuron activation value distributions is nicely visualized figure 4.3. If you look at the

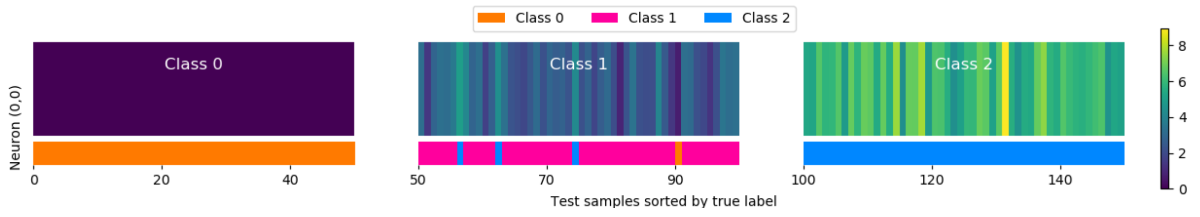


Figure 4.3: Activation values of the single hidden neuron, sorted by the true class label. There are the three colored blocks with *Class 0*, *Class 1*, and *Class 2*, in which the neuron activation values for each data point are shown with the color corresponding to the activation value in the colormap (far right of the figure). The data points are numbered on the y-axis, so each small colorbar within a class block shows the activation value of one data point. Below each class block there is another plot with colors which shows the predictions of the network. The legend on top of the figure is used for this. Since all data points in the lower plot of *Class 0* are labeled orange, all data points are predicted to belong to Class 0. All data points were, therefore, classified correctly by the network. The same goes for the data points belonging to *Class 2*: All data points are predicted as Class 2 (blue), and thus correctly classified. In the lower plot of *Class 1* we see that not all data points were classified correctly, as not every data point has the pink label corresponding to Class 1. For the data points labeled in blue, it means that the data point was incorrectly classified as Class 2, for the orange labeled data points it means that the data point was incorrectly classified as Class 0.

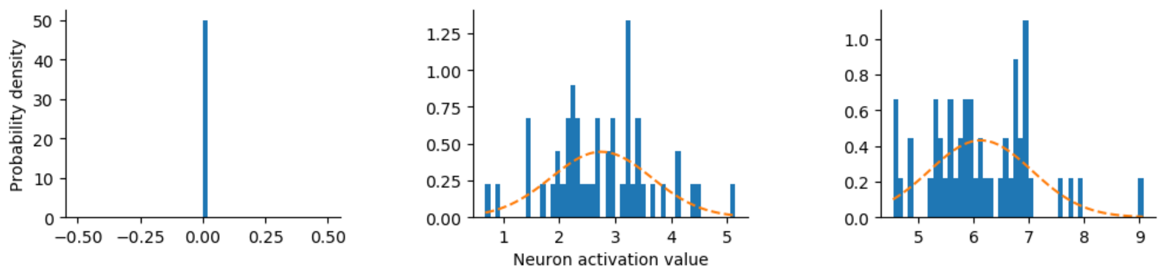


Figure 4.4: Probability density of the neuron activation values for each true class

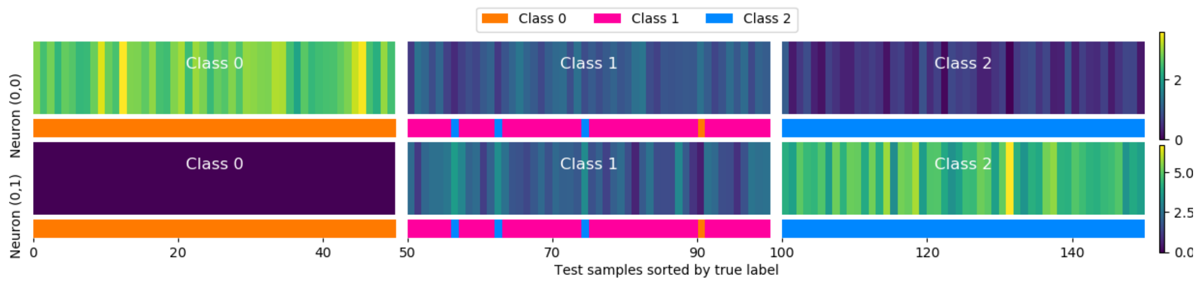
data points that are misclassified, i.e. the data points where the color of the lower plot does not correspond to the correct label, it is evident why these data points were predicted to belong to another class. Let us look at the first three data points that are misclassified. These data points belong to class 1, but they are predicted to belong to class 2. When looking at the corresponding activation values, we see that these values are relatively high for this class label, and thus have a light blue-greenish color. As most of the activation values for class 2 also have this color, the network mistakes these data points for class 2. The fourth point that is misclassified also belongs to class 1, but is predicted to belong to class 0. The plot shows that the corresponding activation value is relatively low for class 1, giving it a dark purple color. This makes the data point look a lot like class 0 data points, leading to misclassification as the network cannot distinguish these data points.

These findings show that neuron activation values contain a lot of information about the neural network's predictions – at least for a single hidden neuron. Next, we will investigate whether these findings hold with more hidden neurons.

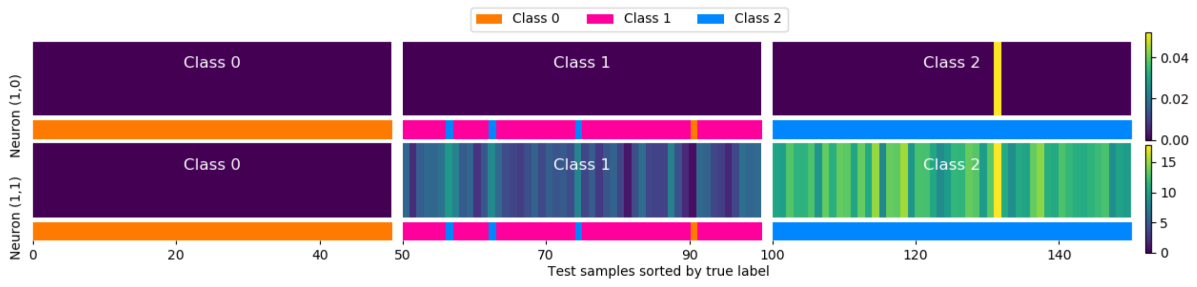
4.2.2. Hidden neuron size of [2,2]

The next network that will be considered is a network with 2 hidden layers, with each two neurons, i.e. with a hidden neuron size of [2,2]. This network is trained on the Iris dataset, with again a test split of 20%. The prediction accuracy on training data as well as test data is 97%. The neuron activation values for each neuron in each layer is shown in figure 4.5.

The figure shows similar findings to the network with a single hidden neuron: the activation values for a data point belonging to a certain class are distinguishable from the activation values of the other classes, and for the data points that are misclassified the activation values are similar to the activation values of the class to which the point was wrongly classified. One thing that strikes the eye is that the neuron at location (1,0) (the top row of figure 4.5b) has activation values of zero for almost all data points, independent of their class label. An explanation for this could be that the neuron is dead, or the current data set is very simple and the network simply does not need more neurons to improve the performance of the network. Therefore, more complex datasets will also be considered.



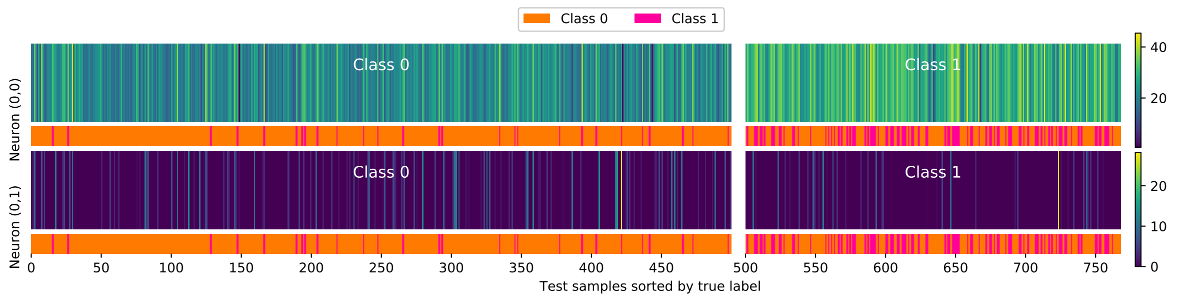
(a) Activation values of the first hidden layer, sorted by the true class label



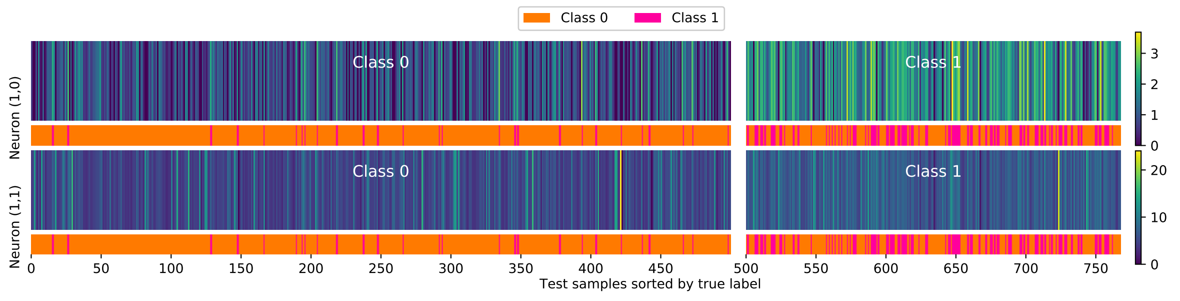
(b) Activation values of the second hidden layer, sorted by the true class label

Figure 4.5: Activation values for trained network on Iris data with hidden neuron size [2,2]

The Pima Indian diabetes dataset, which was discussed earlier this chapter, is a more complex dataset as it is harder to find meaningful correlations in the data. For this dataset a network with with a hidden neuron size of [2,2] was trained, again with a test split of 20%. The prediction accuracy on training data as well as test data is 75%. The neuron activation values for each neuron in each hidden layer are shown in figure 4.6.



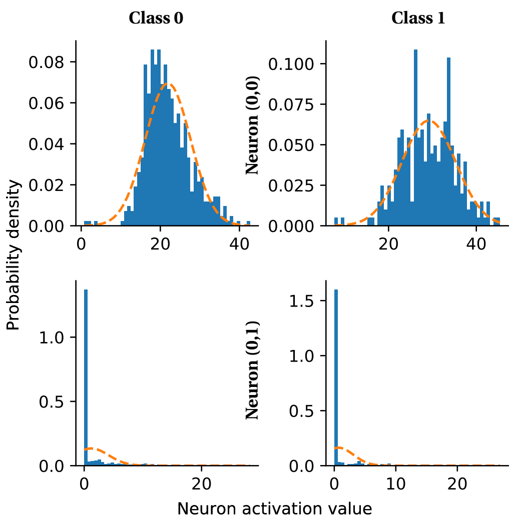
(a) Activation values of the second hidden layer, sorted by the true class label



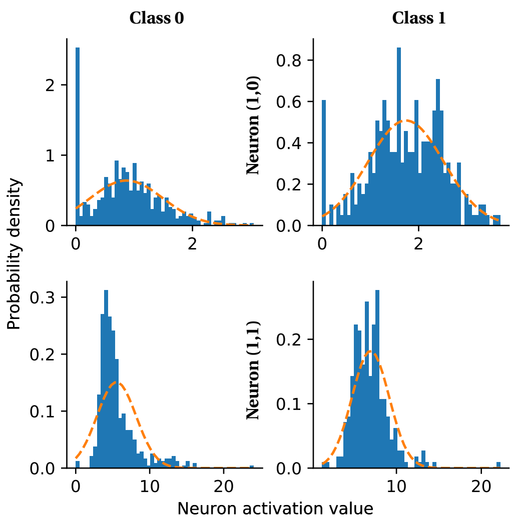
(b) Activation values of the first hidden layer, sorted by the true class label

Figure 4.6: Activation values for trained network on the Diabetes dataset with hidden neuron size [2,2]

As can be expected by more complex data, the neuron activation values for the different class labels are more alike. Nevertheless, meaningful differences are still present. This is more clearly shown in figure 4.7. For the hidden neuron at position (0,0) (the first neuron in the first hidden layer), figure 4.6a shows a noticeable difference in activation values as the samples for class 0 mostly have a blueish color, whereas the samples



(a) First hidden layer



(b) Second hidden layer

Figure 4.7: Probability density of the neuron activation values

from class 1 more often have a greenish-yellow color. This is also shown in figure 4.7a, where we can see that for neuron (0,0) the class 1 histogram lies more to the right than the histogram of class 0, and thus the activations of class 1 are higher. For the second neuron in the first hidden layer, neuron (0,1), it is harder to distinguish between the different classes. In figure 4.6a this is shown by the similar colors for the different classes for this neuron, and figure 4.7a shows the same thing as the histograms for this neuron look very alike. The neurons in the second hidden layer give similar findings, as shown in figures 4.6b and 4.7b. The neuron at position (1,0) shows a clear difference in activation values for the two different classes. For the neuron at position (1,1) the activation values seem less significant for the prediction of the output class.

When looking at the data points that were classified by the network to the wrong output class, it is still visible that these data points create activation values which deviate from the correctly classified data points. This is not visible for every data point in every neuron, but for each misclassified data point it is at least visible in some neurons.

4.2.3. Hidden neuron size of [5,5,5]

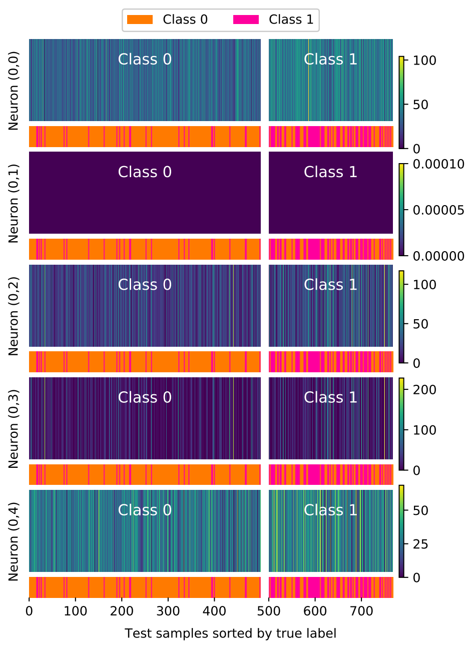
Thirdly, we look at a network with more hidden neurons, namely a network with 3 hidden layers which have 5 neurons each. The network is trained on the Indian Diabetes dataset, with a test split of 20%. With the same amount of epochs but more hidden neurons, an improvement in the performance is already visible as the prediction accuracy of this network is 78%. The neuron activation values for each neuron in each hidden layer is shown in figure 4.8.

As this is a large figure, it will not be discussed in detail for each neuron or layer. When looking at the figure overall, it is still visible that there are differences in activation values for the two different output classes. This does not hold for each neuron (e.g. neuron (2,2) where all values are 0), but about half of the neurons seem to contain meaningful information about the class labels of the data points.

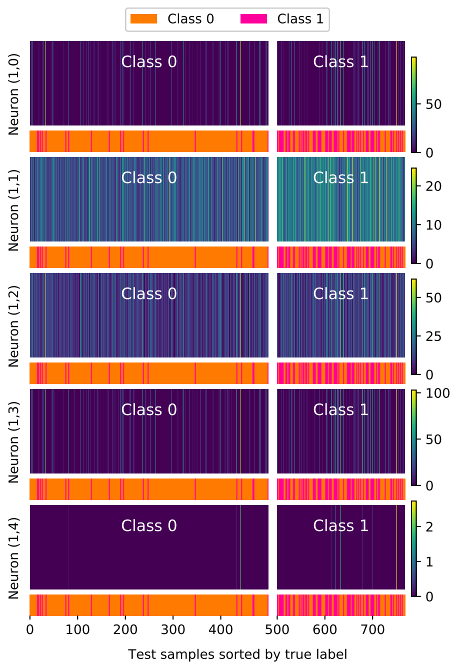
4.2.4. Conclusion and intuition

In the previous sections, neuron activation values were shown for three different network sizes and two different datasets. The corresponding figures all show similar results, and confirm the significance of the neuron activation values. Most importantly, the neuron activation values seem to hold information about the network's prediction, as the activation values can clearly differ in the same neuron for a data point of a different class. Also, the neuron activation values of data points that are incorrectly classified by the network seem to be distinguishable from the neuron activation values of correctly classified data points. These findings show that certain activation patterns are highly predictive of certain classes.

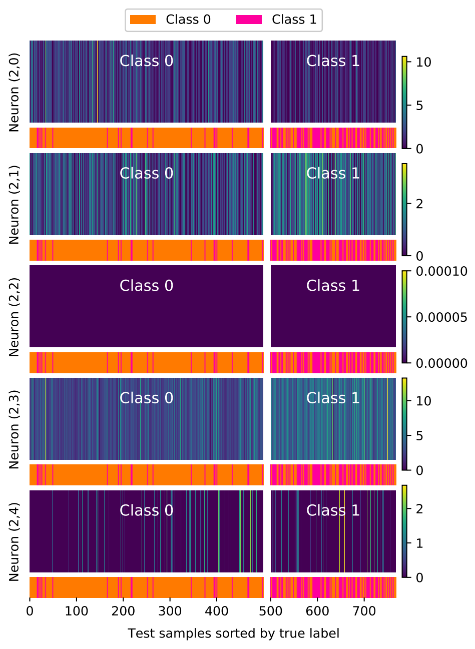
We are interested to see whether we can 'exploit' this finding when creating adversarial examples. Most techniques to create adversarial examples are based on a loss function, which maps the input values onto a



(a) Activation values of the first hidden layer, sorted by the true class label



(b) Activation values of the second hidden layer, sorted by the true class label



(c) Activation values of the third hidden layer, sorted by the true class label

Figure 4.8: Activation values for trained network on the Diabetes dataset with hidden neuron size [5,5,5]

scalar value that indicates how well the network did at predicting the output class. This loss function uses the network output (prediction) as well as the actual class labels. To create an adversarial example, small alterations need to be performed on the input vector. To this end, most techniques compute the gradient of the loss function with respect to the original input, and use this to alter the input in such a way that the loss increases. As the loss increases, one hopes that a different output label will become more likely for the altered data point, while still looking like the original data point. When this happens, the network will wrongly classify the altered data point, and a successful adversarial example is found.

Instead of using the gradient of the loss function with respect to the original input, we want to influence the network's prediction by manipulating the neuron activation values. A minor humanly unnoticeable change in the original data point can already change the activation values inside the hidden neurons. By substantially changing the neuron activation values or by mimicking the activation values of a different output class, new adversarial examples may be found.

4.3. Comparing neuron activations

As the goal is to change neuron activation values in a way that they either look more like the activations from a different class or less like the activations of the actual class, we need a method to compare the neuron activation values of a single data point to the activation patterns of the different output classes. This difference can be used in a loss function instead of the commonly used gradient.

4.3.1. Kernel Density Estimation

As was already shown in figures 4.4 and 4.7, the distribution of neuron activation values contains meaningful information about the classifications. We want to store these distributions, such that when in the process of creating an adversarial example the current neuron activation values can be compared to the distributions of known classes. Note that in order to do this, the training data or other data from which the output classes are known is needed. In this project we use the training data to estimate the distributions.

To estimate the distributions of neuron activation values for different output classes we use Kernel Density Estimation (KDE) [13, 34]. This is a non-parametric method to estimate the probability density function of a stochastic variable based on a finite data sample. The probability density function f of the data sample

$\{x_1, x_2, ...x_n\}$ is estimated as follows:

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right),$$

where K is a kernel and h is a smoothing parameter. The kernel K is a function which is symmetric around the y-axis. Some commonly used kernel functions are uniform, Epanechnikov, Biweight, Triweight and Gaussian. We use the Gaussian kernel function. The smoothing parameter h, also called the bandwidth, has a big influence on the results. With a (too) small bandwidth, the resulting function will be very spiky, whereas with a (too) big bandwidth the function becomes over-smoothed. We use Silverman's rule of thumb [39] to estimate the optimal value of h:

$$h = 0.9 \min\left(\sigma, \frac{IQR}{1.34}\right) n^{-\frac{1}{5}},$$

where σ is the standard deviation of the data sample, n is the sample size, and IQR is the interquartile range (difference between upper and lower quartiles).

4.3.2. Class probabilities

For each hidden neuron within the network we estimate the probability density function for each possible output class using KDE. When considering a new data point, the first step is to put it through the network. This allows us to retrieve the activation value for each hidden neuron. These activation values are inserted into the probability density functions of each different class, which were estimated before. For every neuron we can find the most likely class of the data point by choosing the output class with the highest function output. An example is shown in figure 4.9, where the same trained network was used as in figures 4.3 and 4.4 (Iris data with single hidden neuron).

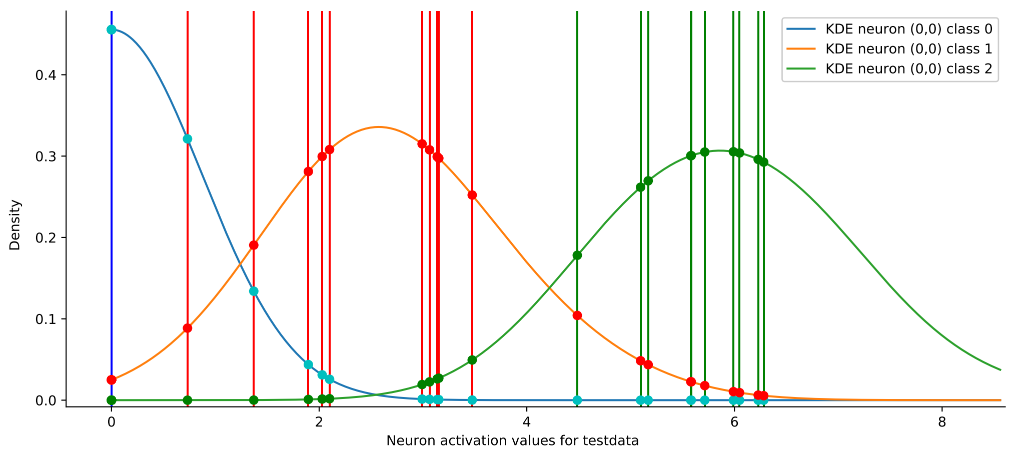
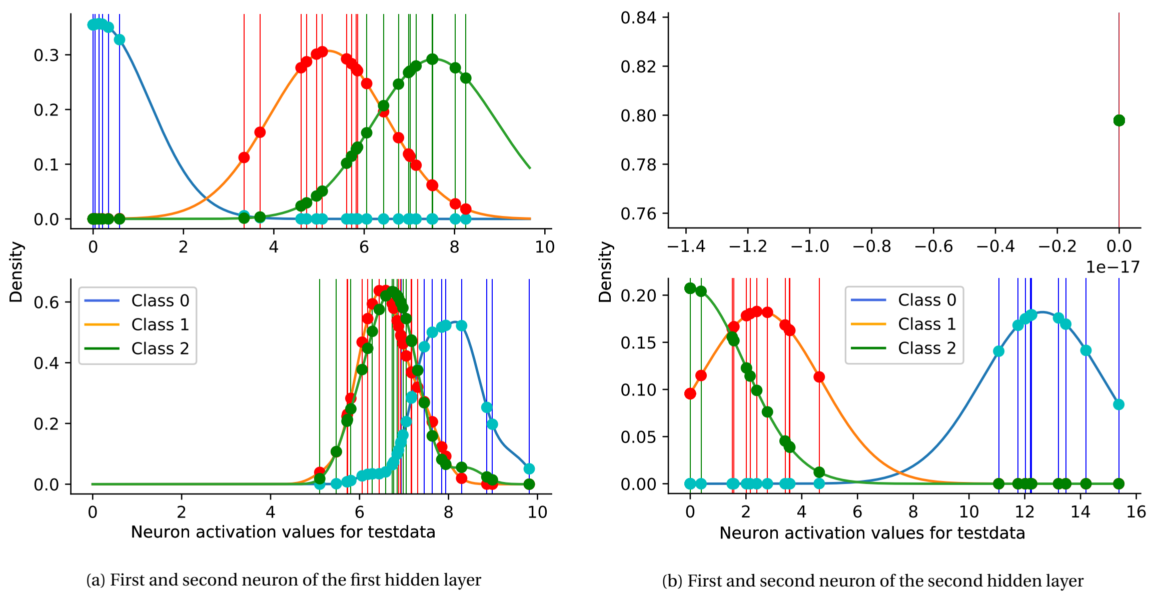


Figure 4.9: Probability density functions for network trained on Iris data with a single hidden neuron. The dots show the results of inserting the activations of the test data into the probability density functions. The color of the vertical line through each data point shows the actual class the data point belongs to.

The difference in neuron activation values of data points belonging to a different class is represented by the three probability density functions. Each data point in the test data is inserted into all three functions, resulting in the dots with the corresponding color. The color of the vertical line through each data point shows the actual class the data point belongs to. According to our probability density functions, the most likely class for the test data point is the one where the density is highest. The figure shows that this results in the correct class label almost all of the time.

The estimated class probabilities are normalized for each neuron and multiplied with the normalized class probabilities of all other neurons for the data point, such that we end up with one final probability for each possible output class of a data point (which is again normalized). To visualize this, we train a network with four hidden neurons on the Iris dataset. The resulting estimated probability density functions can be found in figure 4.10. This figure again shows that the kernel density estimation does a good job at distinguishing the neuron activation values of points with different class labels. The results of multiplying the normalized estimated probabilities is shown in figure 4.11. For most of the test data points, the highest estimated probability gets very close to 1, indicating that we are very confident about the class label of this



(a) First and second neuron of the first hidden layer

(b) First and second neuron of the second hidden layer

Figure 4.10: Probability density functions for network trained on Iris data with two hidden neurons. The dots show the results of inserting the activations of the test data into the probability density functions. The color of the vertical line through each data point shows the actual class the data point belongs to.

data point. For some data points, however, the estimated probabilities of two different classes both have substantial values. In these cases, there is more uncertainty and the highest estimated probability may not correspond to the actual output class. This is the case for data point 11 and 21 in figure 4.10, where we can see that the estimated probabilities for class 1 and class 2 lay somewhat close together, and the highest estimated probability does not match the actual output class.

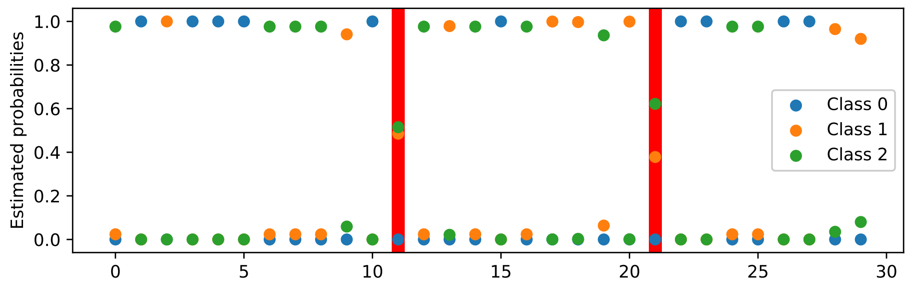


Figure 4.11: The estimated probabilities for each possible output class. A data point has a red vertical line through it if the class of the highest estimated probability does not match the actual output class.

In the example shown above, a small network is considered with only 4 hidden neurons. When considering bigger neural networks with hundreds of hidden neurons, the estimated class probabilities become very small due to the multiplication of every neuron. This is solved by taking the logarithm with base 2 after multiplication and before normalization.

4.4. Adversarial examples

To create an adversarial example, small alterations need to be performed on the input vector. To this end, most techniques compute the gradient of the loss function with respect to the original input, and use this to alter the input in such a way that the loss increases. As the loss increases, one hopes that a different output label will become more likely for the altered data point, while still looking like the original data point. When this happens, the network will wrongly classify the altered data point, and a successful adversarial example is found. This section will discuss a new way of computing losses and adversarial examples, with the use of the estimated class probabilities which are based on the neuron activation values.

4.4.1. Neuron activation loss

In the previous section a method was discussed to obtain estimated class probabilities for a data point. These probabilities can also be used to compute a loss value for the data point. A regular loss function, such as the negative log-likelihood loss or cross entropy loss, can be used for this. The loss function takes as input the probabilities for each class of a data point and the actual class label. It returns a single scalar value which represents the loss or cost of the estimated probabilities with respect to the actual class label of the data point.

In our loss function we want to consider two objectives: we want the neuron activation values of the data point (or adversarial example) to look less like its actual class, and we want the activation values to look more like another class to cause misclassification. We capture this by the following formula:

$$Loss_i = -p_i^l + \max p_i^{\neg l}$$

where l is the true output class of data point i. p represents the vector of probabilities for each class derived from the density functions. We thus take the probability of the true output class and subtract the probability of the most probable non-target class. The loss increases as the true class probability drops and/or if the probability of a different class rises.

Similar to other techniques, we can now try to maximize the loss for a data point to find an adversarial example. For our method this means that we try to make the neuron activation values of the data point less likely to be in the estimated probability density function of the actual class and more likely to be in the estimated probability density function of a different class.

4.4.2. Genetic algorithm

With the new loss function, we still need a way to make adjustments to the data point such that it can become an adversarial example (i.e. flip the network prediction). As there already exist many successful methods for this, we do not have to start from scratch. In 2016, Vidnerova and Neruda introduced the use of a genetic algorithm to create adversarial examples [45]. Later, in 2019, the method GenAttack was developed, also based on the genetic algorithm [2]. The authors claim that their method is the first demonstration of a black-box attack which can succeed against some state-of-the-art defenses. As was explained in section 2.2, a genetic algorithm uses a fitness function, which is similar to a loss function. As both methods mentioned above are black-box, their fitness function only makes use of the input and output values of the network. It would be interesting to see what happens when the fitness function is replaced by a white-box loss function, specifically the neuron activation loss. Before we dive into this, we first take a closer look at some of the important properties of a genetic algorithm.

Search space

As was already explained in section 3.3.1, it is important to evaluate the search space of an algorithm. If only a small part of the input space is considered when creating adversarial examples, it becomes less likely that a successful adversarial example is found. A major downside of all methods based on basic gradient descent on some loss function, is that the bounds of the search space are very tight. For each iteration, gradient descent will change the values with a predetermined learning rate in the direction of the gradient. As a consequence, the algorithm can get stuck in a local optimum and does not further explore the input space.

Fortunately, there are also other methods that consider a larger part of the input space, such as the genetic algorithm. This algorithm is gradient-free and uses randomization to avoid getting stuck in local optima and cover a large part of the input space. Figure 4.12 shows the intermediate results of a method using basic gradient descent (FGSM) and a genetic algorithm based on the same loss function. The FGSM slowly converges to a (possibly local) optimum, and stays there. The genetic algorithm more often takes a big step, favoring exploration. As shown by the (lack of) red background, the gradient descent method crosses the decision boundary after 27 iterations and does not consider moving back in the direction of the decision boundary. The genetic algorithm shows that the larger, randomized steps can cause the algorithm to cross back over the decision boundary, causing the white gaps in some iterations. As we keep track of past successful input values, the algorithm can always recover after a 'misstep', but it might also find a new optimum to exploit.

Figure 4.12 only showed the intermediate results for a single data point. This can also be visualized in a different way for more data points. Figure 4.13 shows the development of adversarial examples for 5 different data points, using the FGSM and a genetic algorithm. The scatter matrix shows the development throughout the iterations for each pair of attributes. Looking at the FGSM, it is visible that the adjustments to the adversarial examples follow a single line, limiting a wider search. As for the genetic algorithm the intermediate

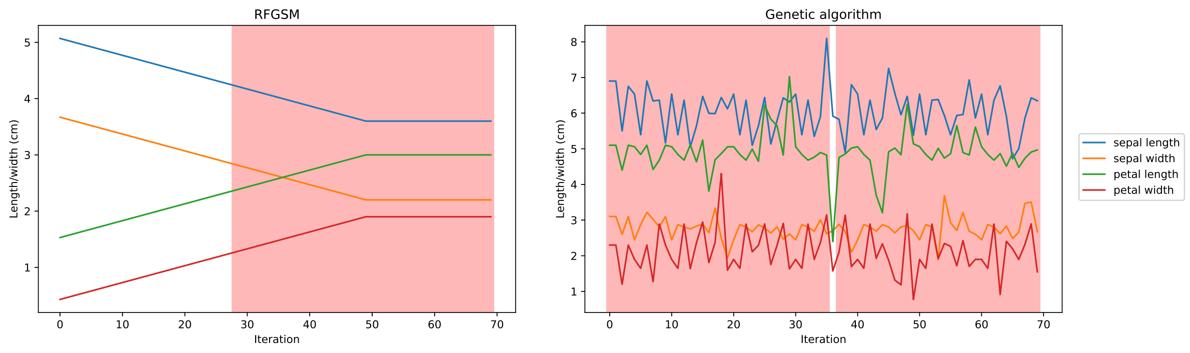


Figure 4.12: The intermediate results for searching for an adversarial example for the FGSM and Genetic algorithm respectively. If the background is red, the adversarial example was successful.

adversarial examples are more scattered around the search space, the search space is better covered by this method. The genetic algorithm also finds an adversarial example sooner than the FGSM.

Parameters

A genetic algorithm has several parameters that influence its results in some way. First of all, a population size needs to be determined. With a larger population, the chance of finding an optimal solution increases. However, for every member of the population calculations need to be made, causing large populations to also be computationally costly. The experiments in this work use a population size of 6.

The next parameter is the number of parents, which needs to be smaller than the population size. The number of parents determine how many data points are used to derive offspring from. As the parents represent the currently best solutions, they are also kept in the population for the next generation. A higher number of parents can yield better convergence [36] at the cost of exploration. The experiments in this work consider 2 parents.

The parents are used to create offspring, using a method called crossover. For non-binary data there are a lot of ways to do crossover. In our experiments we use BLX-a, or Blend Crossover. This is a crossover method for which it has been shown that it has a good search ability [16]. For each gene, the offspring value is chosen randomly from the interval $[X_i^{min}, X_i^{max}]$ following the uniform distribution, where $X^{min} = min(P1_i, P2_i) - \alpha d_i$, $X^{max} = max(P1_i, P2_i) + \alpha d_i$, and $d_i = |P1_i - P2_i|$. $P1$ and $P2$ represent the two parents and i indicates the index of the gene. α is a positive parameter, which we set at 0.366 as this is a good value according to previous works [43].

After the offspring is generated, the mutation phase takes place. This phase uses two parameters: the mutation rate and the mutation size. The mutation rate is set at 0.1, meaning that each gene has a 10% chance of being mutated. When a gene is selected for mutation the mutation size parameter is needed. For creating the perturbations a random sample is drawn from the Gaussian distribution, with a standard deviation determined by our mutation size parameter. In our experiments this parameter is set to 0.05, to ensure minor perturbations.

Finally, we need to decide how many times we want to carry out the reproduction process: the number of generations. Intuitively, the success rate rises as the number of generations is increased. However, each generation requires some costly calculations, so for time efficiency we do not want this number to be too high. In our experiments we run the algorithm for 100 generations.

The population size is rather small, considering we run the algorithm for 100 generations. These parameters show a trade-off between doing many tiny steps on fewer data points and fewer steps on many data points (if we do many tiny steps on many data points the algorithm becomes too costly). By trial and error we found that a small population is already capable of retaining information, and a higher number of generations helps in exploring the promising areas of the search space.

4.4.3. Combining genetic algorithms with neuron loss

We show some preliminary results of the incorporation of the neuron loss in the genetic algorithm. The difference in adversarial examples for the new technique and the fast gradient sign method is shown in figure 4.14. This figure only shows the first thirty test data points, as with all data points the figure becomes too

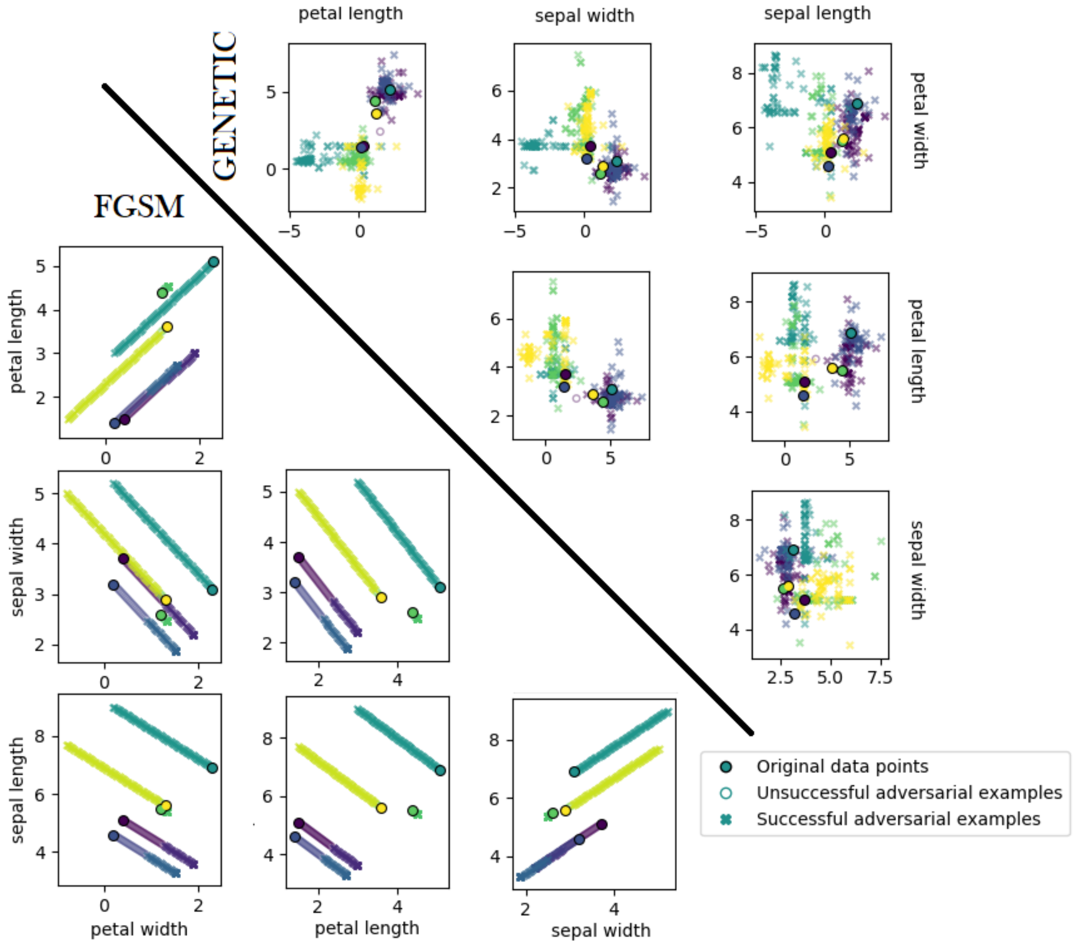


Figure 4.13: Scatter matrix of the intermediate results when developing an adversarial example. Five different data points are considered, each shown in a different color. On the bottom-left the results for the FGSM are shown, and in the top-right the results for the genetic algorithm are shown.

full. The results from the methods to create adversarial examples are only shown in the plot if the adversarial example was successful.

The figure shows that the methods each find different adversarial examples. As one may have noticed, there are more orange crosses than blue crosses. The neuron activation method found adversarial examples for 18 of the 30 data points, whereas the fast gradient sign method found 12.

4.4.4. Accelerated alternative

For neural networks with a large number of hidden neurons, the proposed method can be very time consuming. In each generation of the genetic algorithm the data points are slightly modified, after which their neuron activation values have to be recalculated. For each data point, for each hidden neuron, and for each output class, the new neuron activation value has to be scored against the learned density estimations. The scoring process is expensive, hence we also introduce an alternative method with a simplified scoring mechanism.

In the alternative method, the creation of the kernel density estimations is skipped. Instead, for each output class and each neuron, the corresponding neuron activation values of the training data is put into a limited number of bins, similar to the histograms that were shown in figure 4.7. A probability is assigned to each bin, corresponding to the number of neuron activation values from the training set that belong to this bin. Instead of scoring activation values against the learned kernel density estimations, this method takes the probabilities from the from the learned histogram. We determine in which bin the new neuron activation value falls, and the corresponding probability is used. After the probability calculations we return to the original method and execute the remaining steps as explained earlier. The accelerated alternative is used for the two larger datasets in the experiments.

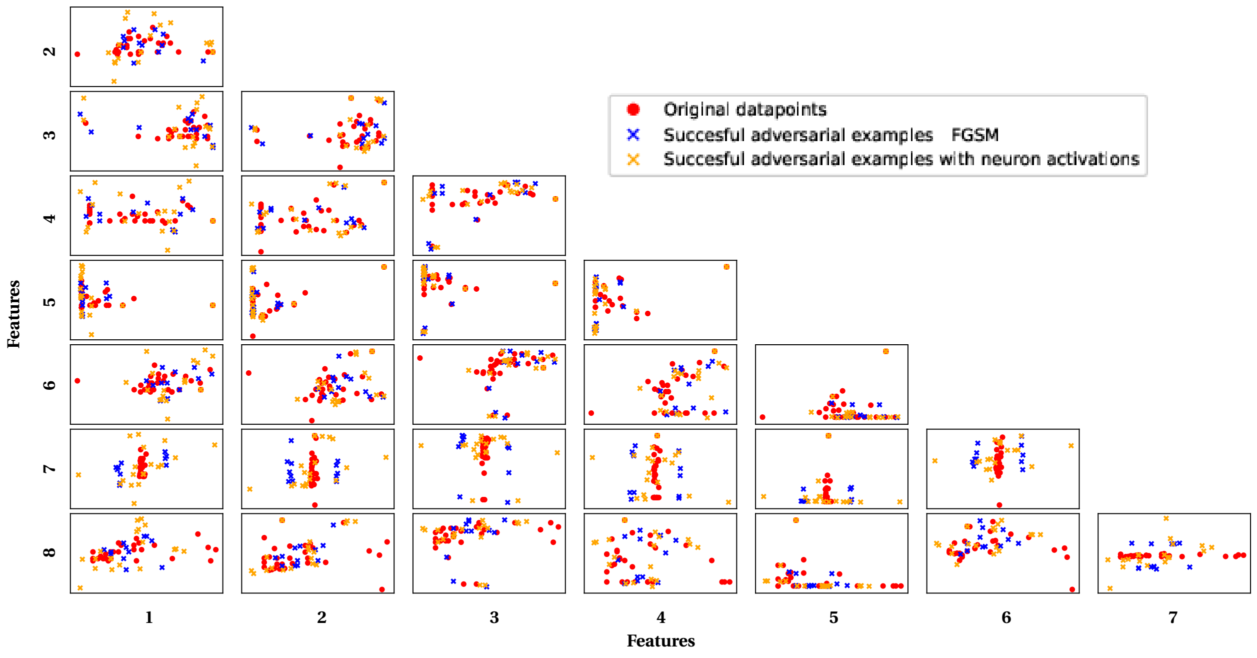


Figure 4.14: Adversarial examples using FGSM and the neuron activation technique on the diabetes dataset

4.5. Summary & conclusions

At the end of this chapter we are able to answer two of our research questions. By visualizing neuron activation values, we found that neurons have different distributions of neuron activation values for different classes. This distinction between neuron activation values of different classes seems to hold for different network sizes. The distributions seem to have a different range for each class, for which almost all activation values fall into this range. The ranges for a single neuron are often partly overlapping, making it mostly infeasible to draw conclusions based on a single hidden neuron. Taking the partly overlapping distributions, we can derive an approximate probability of a (new) neuron activation value belonging to a particular class. These probabilities are calculated for each hidden neuron and each class, after which the probabilities of each neuron are combined into a single probability for each class. Experiments show that this final probability is a good predictor for the class label of the data point. We use these class probabilities for our proposed new attack method, called GenNeuAct, to create adversarial examples. The method uses a genetic algorithm to find small perturbations to the original data point, such that the neuron activation values change. Consequently, we expect the class prediction of the network to also change, generating an adversarial example. The performance of the new method will be evaluated in the next chapter.

5 Experiments

In the previous chapter a new method for creating adversarial examples was introduced, from now on referred to as GenNeuAct. Initial results and comparisons to the FGSM showed a promising research direction. In this chapter, further experiments will be presented to show the performance of the new algorithm in comparison to the state-of-the-art methods. Different datasets of different sizes will be considered, as well as difference in the number of hidden neurons in the neural network. This chapter will first discuss the method that was used for the experiments. Thereafter the results are presented for each dataset. In section 3 the results are discussed and analysed.

5.1. Method

For reproducability, this section contains a detailed description of the method that was used to carry out the experiments. We discuss the implementation of the state-of-the-art methods, consider natural and adversarial training, and look at the four datasets that are used in the experiments. Furthermore, the approach for measuring performance is explained.

5.1.1. State-of-the-art comparison

The new technique will be compared to five state-of-the-art white-box methods for creating adversarial examples: C&W, FGSM, BIM, DeepFool, and RFGSM. For details about the inner workings of these methods, we refer back to section 3.1 where all methods are individually discussed. For all methods except C&W, the implementation from the PyTorch library Torchattacks [23] is used. The library also contains an implementation for the C&W attack, but the implementation does not include the binary search for the parameter c (due to the corresponding time restraints). Since the binary search is important for the high performance of the method, we decided to include this and use the implementation of Kaiwen Wu [46] instead of the implementation from Torchattacks. All methods use the default values of their parameters.

5.1.2. Natural and adversarial training

Neural networks and its development has gotten a lot of attention the last decade. Researchers try to improve the predictions and accuracy of a neural network, and find new applications for it. As neural networks are more widely employed, adversaries have also become more interested in neural networks, especially in finding ways to deceive the neural network: adversarial examples. A common method for making a neural network more robust against these adversarial examples is adversarial training. This concept was first introduced by Szegedy et al. in 2013 [42]. When doing adversarial training, the network is not (only) trained on data points of the actual dataset, but (also) on adversarial examples. When the network is only trained on the original data points, the training type is called natural.

Adversarial training is also useful for our experiments. On a neural network that was trained naturally, it can be (too) easy to find adversarial examples. This makes it hard to see whether one method is more effective than another. With adversarial training the objective to find adversarial examples is harder to realize, thus better showing the search capabilities of the different methods.

Adversarial training is often very expensive, as adversarial examples continuously have to be created throughout the training process. Therefore it is always a trade-off between speed and performance, making it hard to select a single state-of-the-art method that is best in all cases. Projected Gradient Descent (PGD) is a standard first-order optimization method and is known as a universal adversary [30]. This method is widely used as it is effective and the speed is acceptable. In further experiments PGD will be used when training a network adversarially. As we want the network to learn the adversarial inputs, but also still recognize the normal inputs, the adversarially trained networks in the experiments will be trained on both. Each batch in each epoch has a 75% chance of being converted to adversarial examples, thus 25% of the training is based on original data points.

5.1.3. Datasets

The performance of the six different methods will be measured on several datasets with diverse properties. The following datasets are considered: Pima Indian Diabetes [38], Breast Cancer Wisconsin (Diagnostics) [14], Optical Recognition of Handwritten Digits [14], and Fashion-MNIST [47] . For each dataset, the combination of the input size, output size, dataset size and learnability is different. An overview of the datasets is shown in table 5.1. For each dataset a neural network is learned. The properties of the neural network, such as the number and size of hidden layers, the learning rate, and the size of the test set, differ for each dataset. An overview of network properties for each dataset is shown in table 5.2.

Some methods, such as the C&W attack, require the inputs to only contain values between 0 and 1. Therefore, all datasets that did not meet this requirement beforehand are normalized for the experiments.

Dataset	**Description**	**Input**	**Output**	**Size (# datapoints)**
Pima Indian Diabetes	Dataset with diagnostic measurements to diagnostically predict whether or not a patient has diabetes,	8 medical predictor values	Diagnosis: 0 or 1	768
Breast Cancer Wisconsin (Diagnostics)	Dataset with features computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.	30 breast cell nucleus measurements	Diagnosis: 0 or 1	569
Optical Recognition of Handwritten Digits	Dataset of extracted normalized bitmaps of handwritten digits from a preprinted form. 32x32 bitmaps are divided into nonoverlapping blocks of 4x4 and the number of on pixels are counted in each block.	Bitmap of counted activated pixels of 8x8 (64 input values)	Digit 0-9	5620
Fashion-MNIST	Dataset with images of 10 different article types from Zalando.	Grayscale images of 28x28 pixels (784 input values)	Article class 0-9	60000

Table 5.1: Overview of the datasets used in the performance analysis, including important properties of the dataset.

Dataset	**Hidden layers**	**Size test set**	**Learning rate**	**Batch size**
Pima Indian Diabetes	(20, 20)	77 (10%)	0.05	10
Breast Cancer Wisconsin (Diagnostics)	(15, 15)	100 (18%)	0.005	10
Optical Recognition of Handwritten Digits	(25, 25)	1000 (18%)	0.005	10
Fashion-MNIST	(100, 100)	10000 (17%)	0.05	100

Table 5.2: Overview of the training parameters for each dataset.

5.1.4. Measuring performance

With all methods, data, and network settings in place, the experiments are ready to be carried out. For each dataset two neural networks are learned (naturally and adversarially trained), whereupon each method is tasked to create adversarial examples for all data points in the test set. The test set is identical for both neural networks, such that the results can be conveniently compared. For each set of created adversarial examples two performance measures are calculated: the success rate and the mean distance. The success rate is easily calculated by dividing the number of successful adversarial examples by the number of data points in the test set. A higher success rate thus shows that the method was capable of finding an adversarial example for more data points. For calculating the mean distance, only successful adversarial examples are taken into account. For each of these points the distance to the original data point is calculated, using the Euclidean distance measure. The distances are added and divided by the number of successful adversarial examples to get the mean distance. The mean distance is a measure for how good the adversarial example is. A good adversarial example is very similar to the original data point, and therefore the mean distance should be small. As the datasets contain multidimensional data, the measuring of the distance may not feel straightforward, hence an example is shown in figure 5.1.

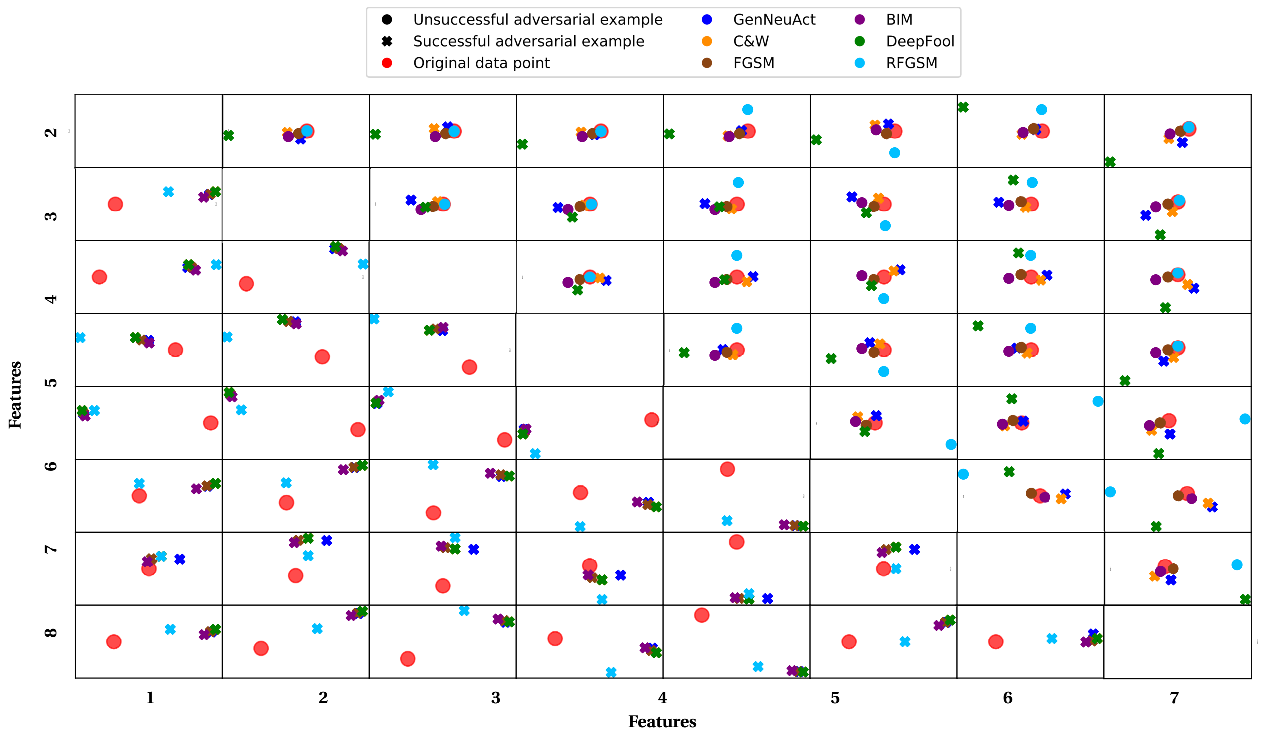


Figure 5.1: Visualizing the distance measure for multidimensional data. The figure shows two scatter matrices (bottom-left and top-right), with each one data point from the Pima Indian Diabetes dataset. The original data point is represented by the large red dot, and is shown for all combinations of dimensions. Each method is tasked to create an adversarial example for these data points, resulting in either a cross (successful adversarial example) or dot (unsuccessful adversarial example) with the corresponding color. For each dimension the distance to the original data point can be calculated straightforwardly. The matrix in the bottom left shows that all methods successfully found an adversarial example. Most methods have found a similar example, as the crosses of these points lay closely together in all dimensions. The RFGSM clearly found a different adversarial example, as the light blue crosses mostly isolated from the others. As the distance to the original data point differs in each dimension, the Cartesian coordinates are used to compute the Euclidean distance, covering all dimensions. The matrix in the top right shows that not all methods were able to find an adversarial example.

5.2. Results

In this section the results are presented. The results are grouped per dataset, resulting in the four subsections. In each subsection the performance of the corresponding networks are shown, followed by the analysis of the results and the performance of GenNeuAct compared to state-of-the-art methods.

5.2.1. Pima Indian Diabetes

The Pima Indian Diabetes dataset was already introduced and used in the previous chapter. The dataset consists of 768 data points, which each contain 8 human features such as age and blood pressure. Each data point is labeled with the corresponding diagnosis: diabetes or no diabetes.

For this dataset, neural networks are trained with two hidden layers of size 20. 10% of the data is withheld to use for testing. The test set data points are given to the different attack methods, which will create adversarial examples for these data points. For reliability we repeat the experiment 5 times, in which the networks are trained with identical settings and data points. The accuracy scores for each network can be found in table 5.3. An overview of successful adversarial examples for each method can be found in figure 5.2 and figure 5.3 for natural training and adversarial training respectively. Details about the success rate and mean distance of the adversarial examples can be found in figures 5.4 and 5.5.

	Natural training		**Adversarial training (PGD)**	
	Train accuracy	**Test accuracy**	**Train accuracy**	**Test accuracy**
Net 1	0.747	0.714	0.683	0.675
Net 2	0.804	0.662	0.730	0.662
Net 3	0.774	0.597	0.730	0.701
Net 4	0.766	0.675	0.744	0.701
Net 5	0.776	0.610	0.717	0.701

Table 5.3: Overview of the accuracy scores of the 10 models trained on this dataset.

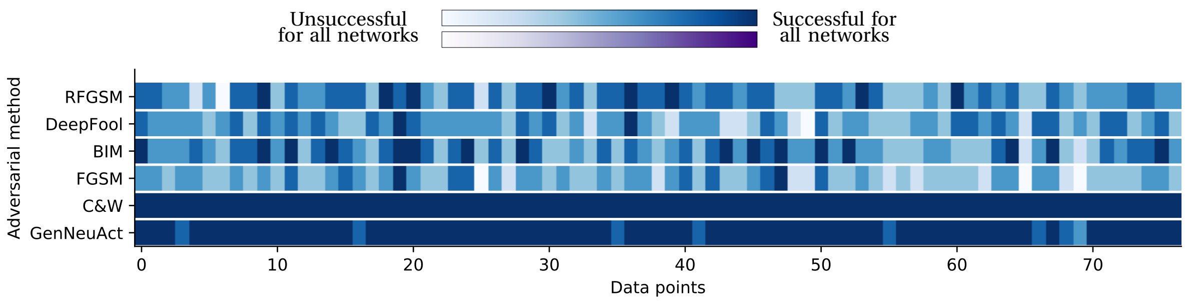


Figure 5.2: Overview of successful adversarial examples for a naturally trained neural network on the Pima Indian Diabetes dataset.

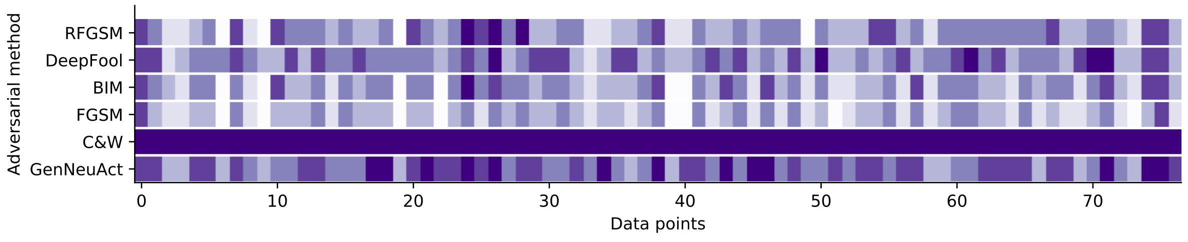


Figure 5.3: Overview of successful adversarial examples for an adversarially trained neural network on the Pima Indian Diabetes dataset.

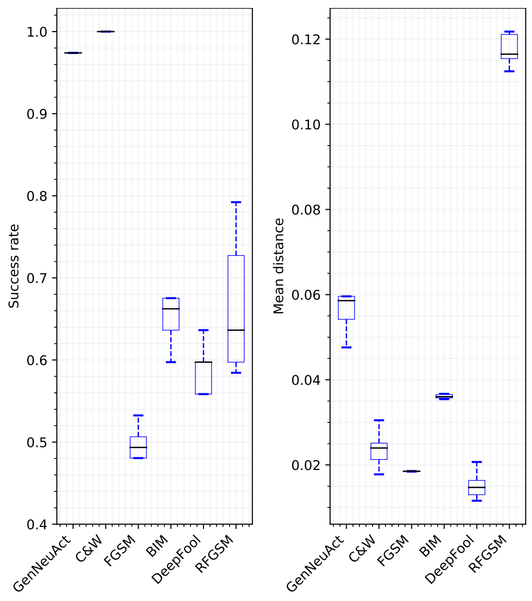


Figure 5.4: Overview of the success rates and mean distances for the five networks that were trained naturally on the Pima Indian Diabetes dataset.

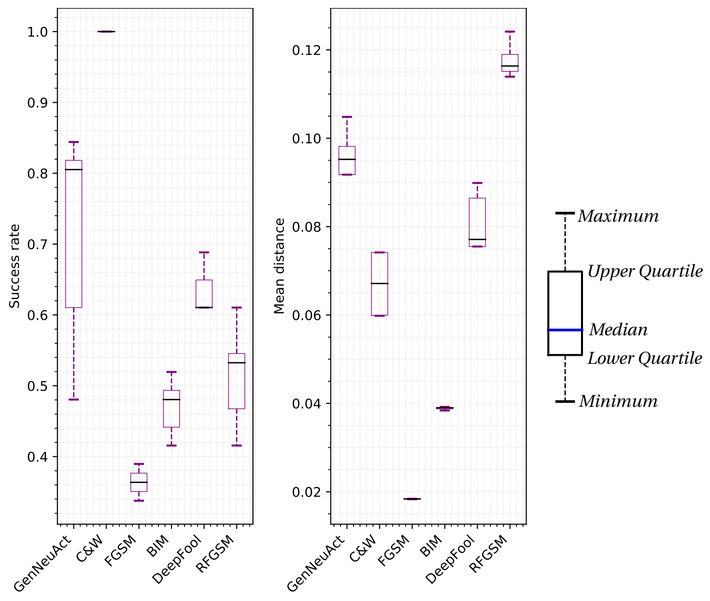


Figure 5.5: Overview of the success rates and mean distances for the five networks that were trained adversarially on the Pima Indian Diabetes dataset.

5.2.2. Breast Cancer Wisconsin (Diagnostic)

The Breast Cancer Wisconsin (Diagnostics) dataset was created by the University of Wisconsin to improve predicting whether a cancer is benign or malignant. For each data sample a fine needle aspirate was taken from the breast mass, and digitized into an image. The image is analyzed, and 10 cytological features are extracted from it (such as radius and smoothness). For each cytological feature the mean, the extreme value, and the standard error is calculated, resulting in 30 input values per data point. All data is labeled with the corresponding diagnosis: benign or malignant.

For this dataset, neural networks are trained with two hidden layers of size 15. About 18% of the data (100 data points) is withheld to use for testing. The test set data points are given to the different attack methods, which will create adversarial examples for these data points. For reliability we repeat the experiment 5 times, in which the networks are trained with identical settings and data points. The accuracy scores for each network can be found in table 5.4. An overview of successful adversarial examples for each method can be found in figure 5.6 and figure 5.7 for natural training and adversarial training respectively. Details about the success rate and mean distance of the adversarial examples can be found in figures 5.8 and 5.9.

	Natural training		**Adversarial training (PGD)**	
	Train accuracy	**Test accuracy**	**Train accuracy**	**Test accuracy**
Net 1	0.934	0.930	0.928	0.920
Net 2	0.921	0.920	0.928	0.920
Net 3	0.940	0.920	0.932	0.920
Net 4	0.930	0.920	0.919	0.920
Net 5	0.934	0.920	0.923	0.910

Table 5.4: Overview of the accuracy scores of the 10 models trained on this dataset.

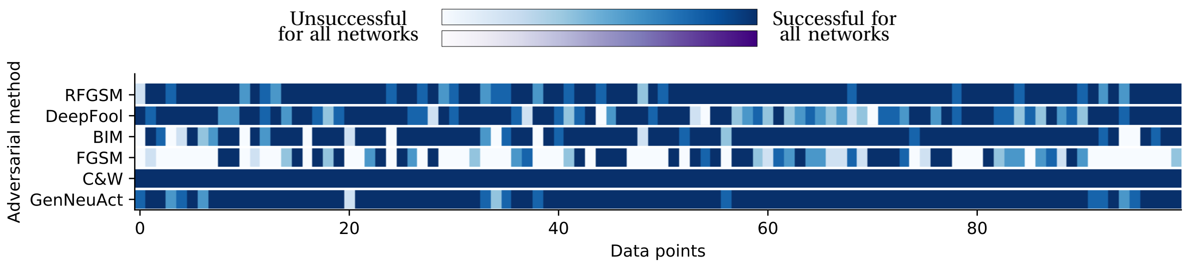


Figure 5.6: Overview of successful adversarial examples for an naturally trained neural network on the Breast Cancer Wisconsin (Diagnostic) dataset.

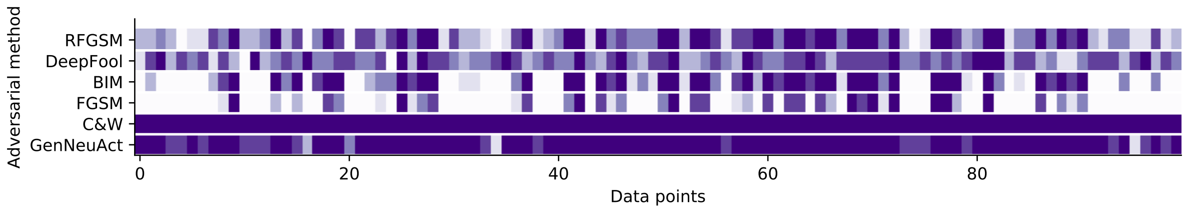


Figure 5.7: Overview of successful adversarial examples for an adversarially trained neural network on the Breast Cancer Wisconsin (Diagnostic) dataset

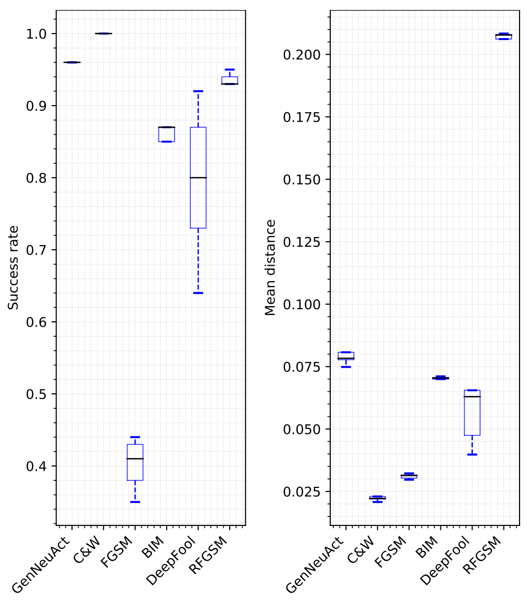


Figure 5.8: Overview of the success rates and mean distances for the five networks that were trained naturally on the Breast Cancer Wisconsin (Diagnostic) dataset.

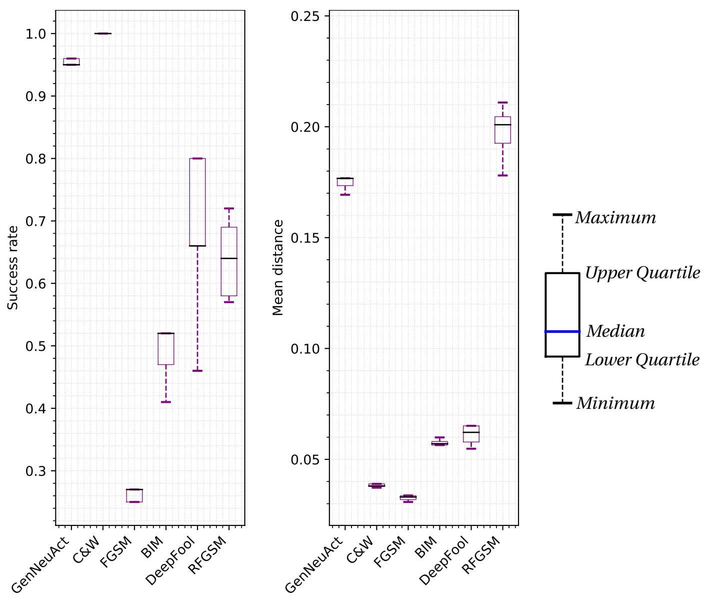


Figure 5.9: Overview of the success rates and mean distances for the five networks that were trained adversarially on the Breast Cancer Wisconsin (Diagnostic) dataset.

5.2.3. Optical Recognition of Handwritten Digits

The Optical Recognition of Handwritten Digits dataset contains handwritten digits from a total of 43 people; 30 people contributed to the training set and 13 different people to the test set. The original bitmap of 32x32 is divided into non-overlapping blocks of 4x4, whereupon the number of 'on' pixels is summed in each block. This generates an input of 8x8 where each value is an integer in the range 0 to 16. The summation reduces the dimensionality of the data, and small distortions are less noticeable. By visualizing the resulting input, the different digits are still clearly visible, as shown in figure 5.10.

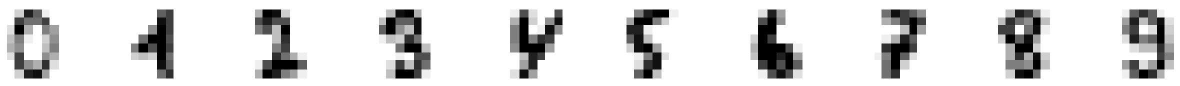


Figure 5.10: Visualization of an example from each output class taken from the Optical Recognition of Handwritten Digits dataset.

For this dataset, neural networks are trained with two hidden layers of size 25. About 18% of the data (1000 data points) is withheld to use for testing. The test set data points are given to the different attack methods, which will create adversarial examples for these data points. For reliability we repeat the experiment 5 times, in which the networks are trained with identical settings and data points. The accuracy scores for each network can be found in table 5.5. An overview of successful adversarial examples for each method can be found in figure 5.11 and figure 5.12 for natural training and adversarial training respectively. Details about the success rate and mean distance of the adversarial examples can be found in figures 5.13 and 5.14. For these experiments we used the accelerated alternative presented at the end of chapter 4.

	Natural training		**Adversarial training (PGD)**	
	Train accuracy	**Test accuracy**	**Train accuracy**	**Test accuracy**
Net 1	0.996	0.970	0.996	0.987
Net 2	0.996	0.975	0.996	0.986
Net 3	0.995	0.966	0.996	0.987
Net 4	0.992	0.963	0.994	0.983
Net 5	0.996	0.970	0.996	0.981

Table 5.5: Overview of the accuracy scores of the 10 models trained on this dataset.

As the dataset represents simplified handwritten digits, the adversarial examples can be nicely visualized. The results for two data points are shown, in figure 5.16 for the naturally trained network and figure 5.15 for the adversarially trained network. These figures also show the differences in pixel values for each adversarial example. In both given examples, DeepFool failed to create a successful adversarial example, and FGSM failed only for the second data point. The images for the failed methods show their best attempt. Looking at both figures, we can deduce some differences and similarities between the different methods. FGSM and BIM show a similar pattern in changing pixel values, and most of their alterations are equal in size. The changes made by BIM are a bit larger, causing this method to have a higher success rate than FGSM. Surprisingly, C&W and DeepFool also show similar alteration patterns, despite their contrasting methods. DeepFool, however, often fails to tweak the pixels precisely right to create an adversarial example, whereas C&W is almost always successful. GenNeuAct makes some small alterations as well as large alterations. For the RFGSM it even becomes hard to determine the correct digit visually.

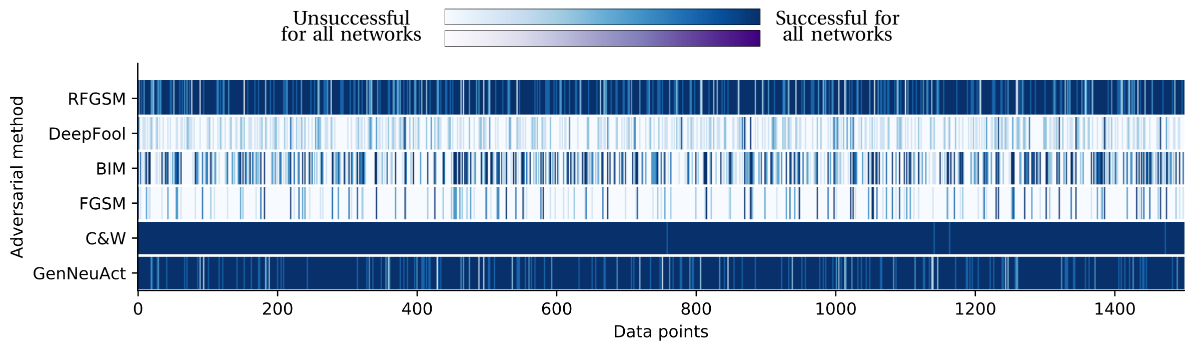


Figure 5.11: Overview of successful adversarial examples for a naturally trained neural network on the Optical Recognition of Handwritten Digits dataset.

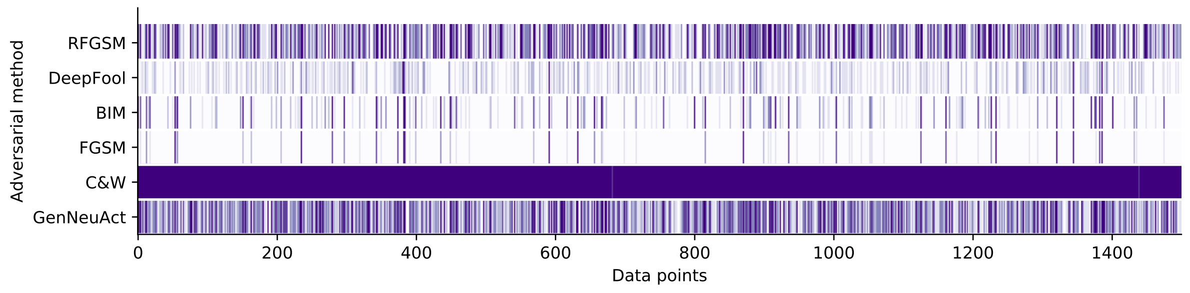


Figure 5.12: Overview of successful adversarial examples for an adversarially trained neural network on the Optical Recognition of Handwritten Digits dataset

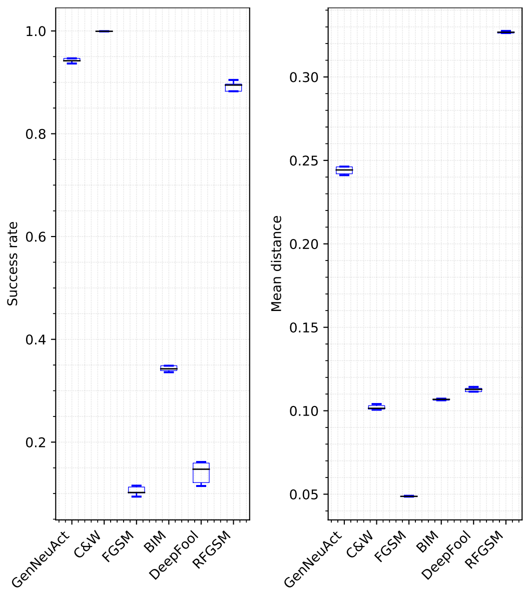


Figure 5.13: Overview of the success rates and mean distances for the five networks that were trained naturally on the Handwritten Digits dataset.

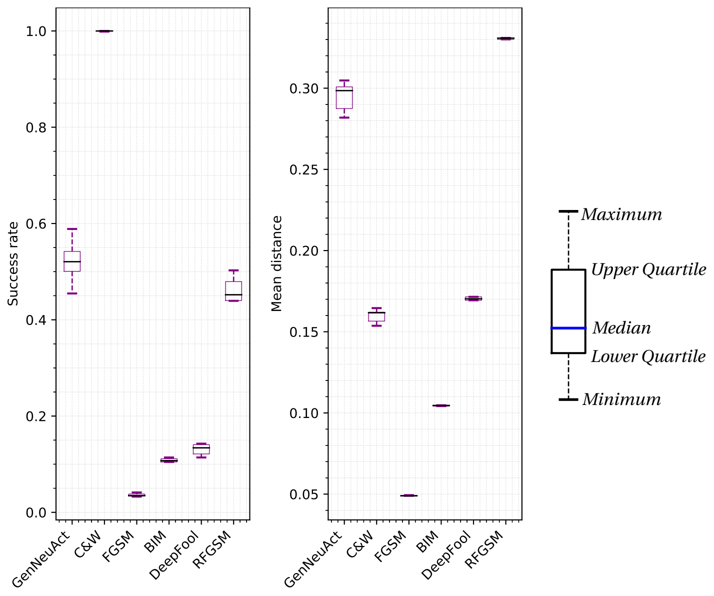


Figure 5.14: Overview of the success rates and mean distances for the five networks that were trained adversarially on the Handwritten Digits dataset.

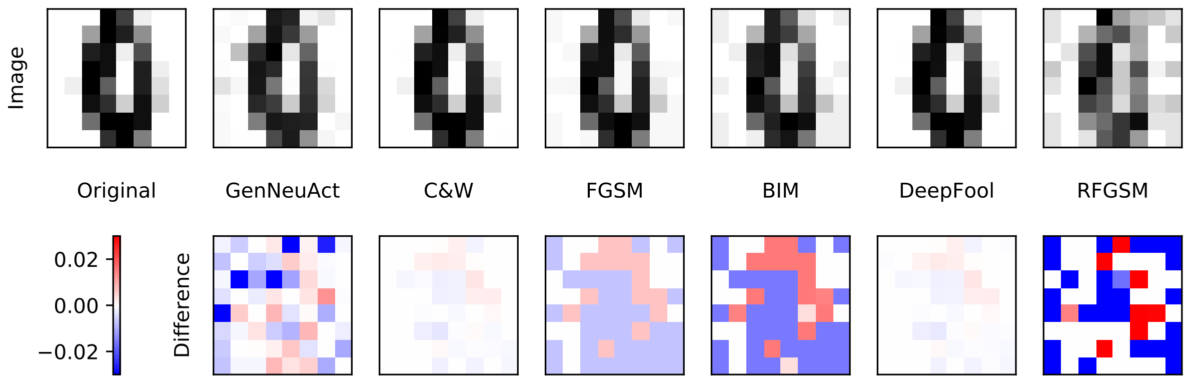


Figure 5.15: Visualization of the adversarial examples created by the different methods for one data point, using a naturally trained network. The top row shows the adversarial example, and the bottom row shows a magnified colormap of the difference between the original picture and the adversarial example. GenNeuAct, C&W, FGSM, BIM, and RFGSM created a successful adversarial example. Only DeepFool failed to produce one.

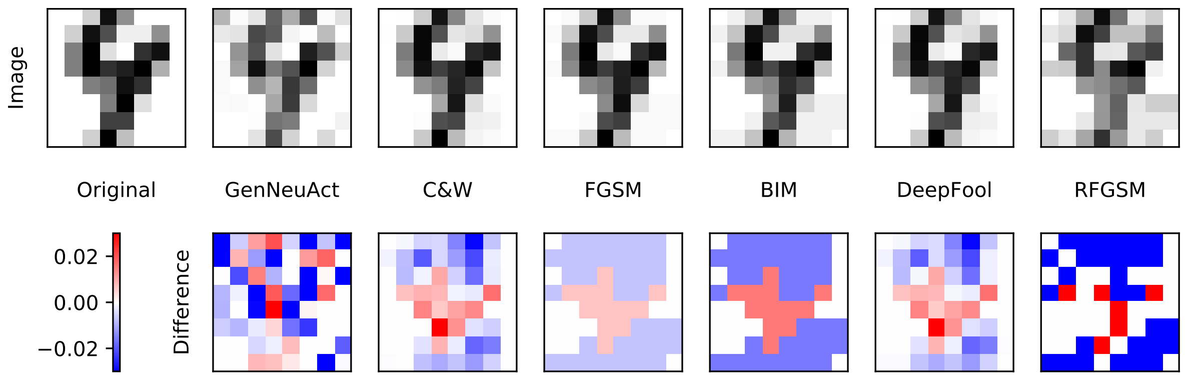


Figure 5.16: Visualization of the adversarial examples created by the different methods for one data point, using an adversarially trained network. The top row shows the adversarial example, and the bottom row shows a magnified colormap of the difference between the original picture and the adversarial example. GenNeuAct, C&W, BIM, and RFGSM created a successful adversarial example. FGSM and DeepFool failed to produce one.

5.2.4. Fashion-MNIST

The Fashion-MNIST dataset contains grayscale images of 28x28 pixels, showing ten different article types of Zalando. This dataset was created by Zalando SE as an alternative for the MNIST dataset [26]. MNIST is a well-known dataset of handwritten digits, consisting of 70000 images of 28x28 pixels, which is very popular and used as a benchmark among scientists exploring pattern recognition and machine learning. Despite the popularity, the flaws and drawbacks of the dataset are discussed more often. As classic machine learning algorithms can already easily achieve 97% accuracy, MNIST is thought to be too easy. The dataset also cannot represent modern computer vision tasks. Fashion-MNIST was created to serve as a direct drop-in-replacement for the original MNIST dataset, without the drawbacks. The amount and size of the images as well as the number of target classes are identical to the MNIST dataset. A sample for each output class, taken from the training set, is shown in figure 5.17.

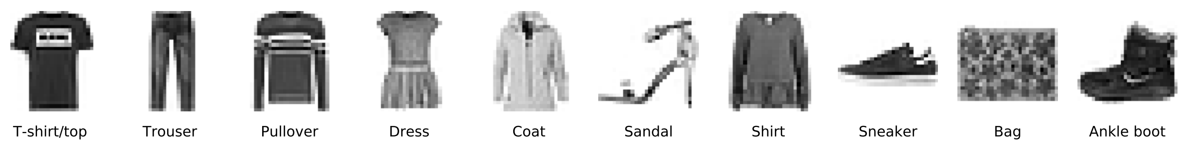


Figure 5.17: Visualization of an example from each output class taken from the Fashion-MNIST dataset.

For this dataset, neural networks are trained with two hidden layers of size 100. The dataset has a predetermined test set of about 17% of the data (10000 data points). The test set data points are given to the different attack methods, which will create adversarial examples for these data points. For reliability we repeat the experiment 5 times, in which the networks are trained with identical settings and data points. The accuracy scores for each network can be found in table 5.6. An overview of successful adversarial examples for each method can be found in figure 5.11 and figure 5.19 for natural training and adversarial training respectively. Details about the success rate and mean distance of the adversarial examples can be found in figures 5.20 and 5.21. For these experiments we used the accelerated alternative presented at the end of chapter 4.

	Natural training		**Adversarial training (PGD)**	
	Train accuracy	**Test accuracy**	**Train accuracy**	**Test accuracy**
Net 1	0.941	0.880	0.939	0.889
Net 2	0.941	0.883	0.936	0.879
Net 3	0.942	0.883	0.943	0.886
Net 4	0.942	0.879	0.940	0.887
Net 5	0.939	0.878	0.933	0.882

Table 5.6: Overview of the accuracy scores of the 10 models trained on this dataset.

As this dataset contains images, the adversarial examples can be nicely visualized. The results for two data points are shown, in figure 5.22 for the naturally trained network and figure 5.23 for the adversarially trained network. These figures also show the differences in pixel values for each adversarial example. In both given examples, (only) DeepFool failed to create a successful adversarial example. The images for the failed methods show their best attempt.

Looking at both figures, we can deduce some differences and similarities between the different methods. Much like the results from the Digits dataset, FGSM and BIM show a similar pattern in changing pixel values, and most of their alterations are equal in size. RFGSM makes large alterations, GenNeuAct has some larger and smaller alterations and C&W and DeepFool only have small alterations, as was also seen in the previous results. Noteworthy is the distinctive power of C&W to only alter important pixels (i.e. not the background), all other methods have perturbations all over the place.

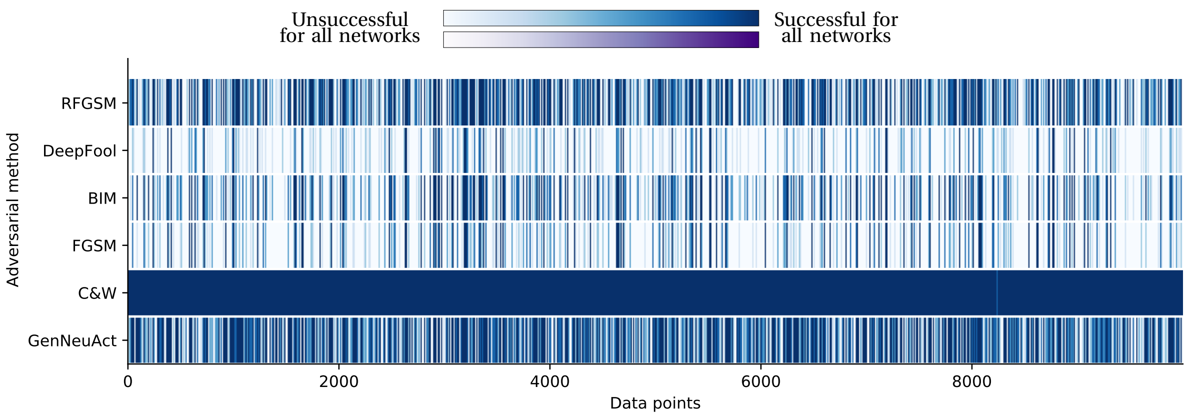


Figure 5.18: Overview of successful adversarial examples for a naturally trained neural network on the Fashion-MNIST dataset.

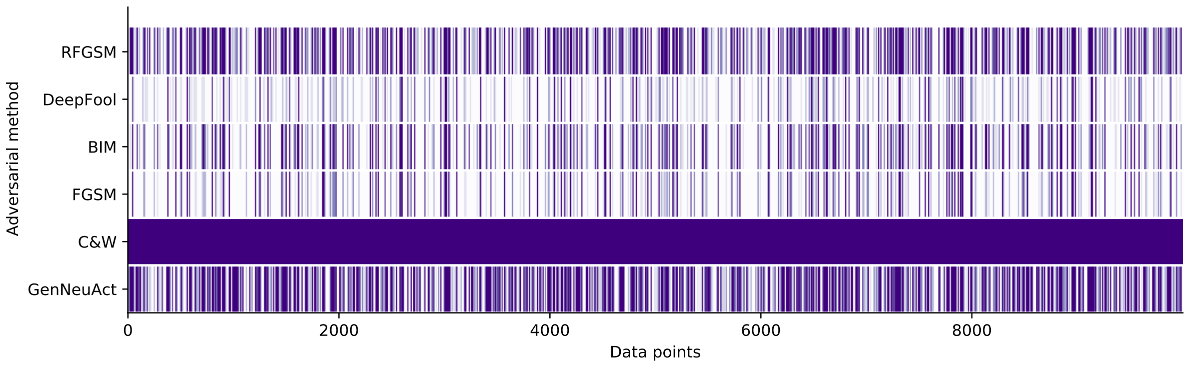


Figure 5.19: Overview of successful adversarial examples for an adversarially trained neural network on the Fashion-MNIST dataset

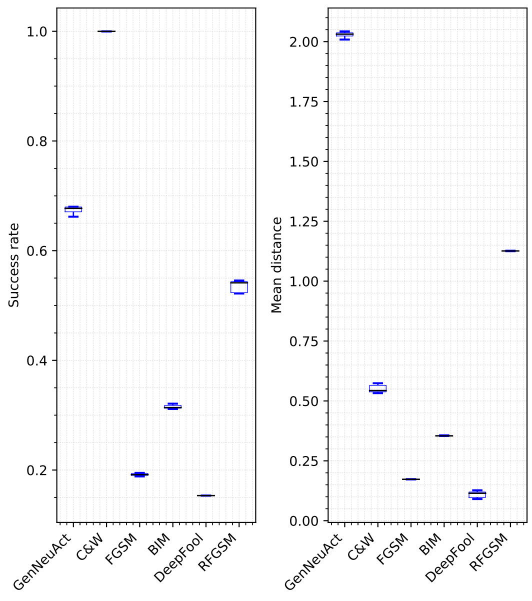


Figure 5.20: Overview of the success rates and mean distances for the five networks that were trained naturally on the Fashion-MNIST dataset.

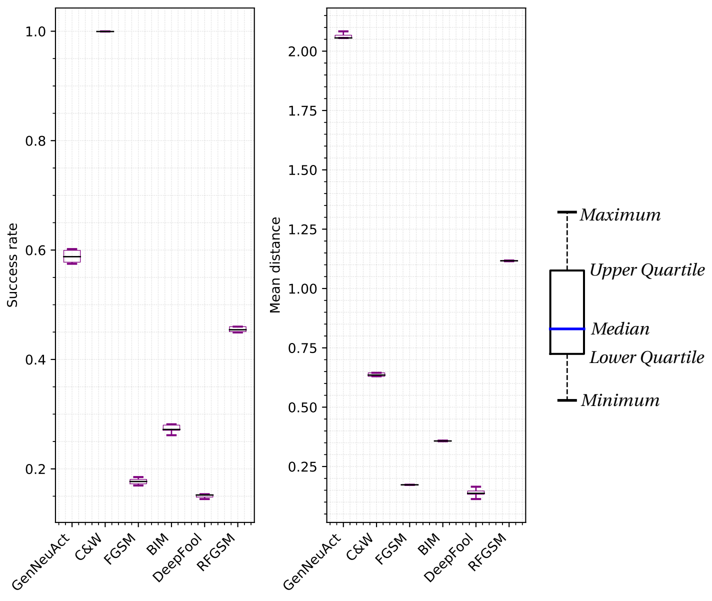


Figure 5.21: Overview of the success rates and mean distances for the five networks that were trained adversarially on the Fashion-MNIST dataset.

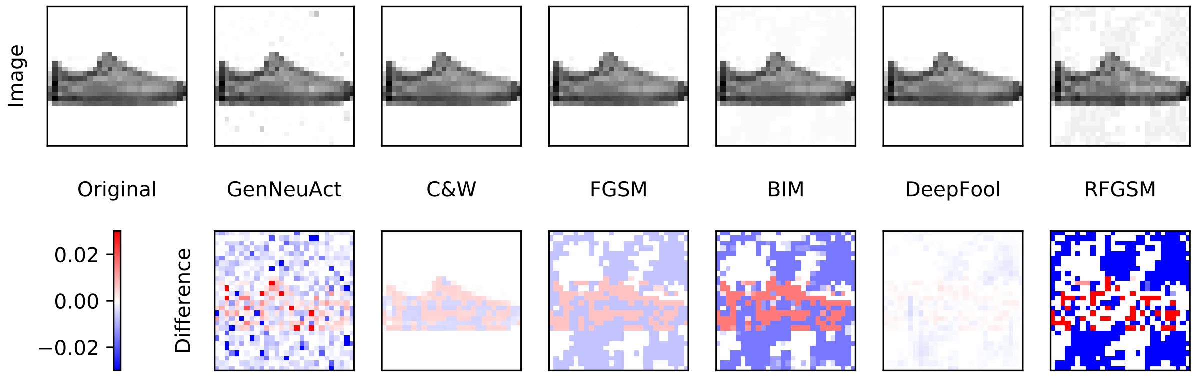


Figure 5.22: Visualization of the adversarial examples created by the different methods for one data point, using a naturally trained network. The top row shows the adversarial example, and the bottom row shows a magnified colormap of the difference between the original picture and the adversarial example. GenNeuAct, C&W, FGSM, BIM, and RFGSM created a successful adversarial example. Only DeepFool failed to produce one.

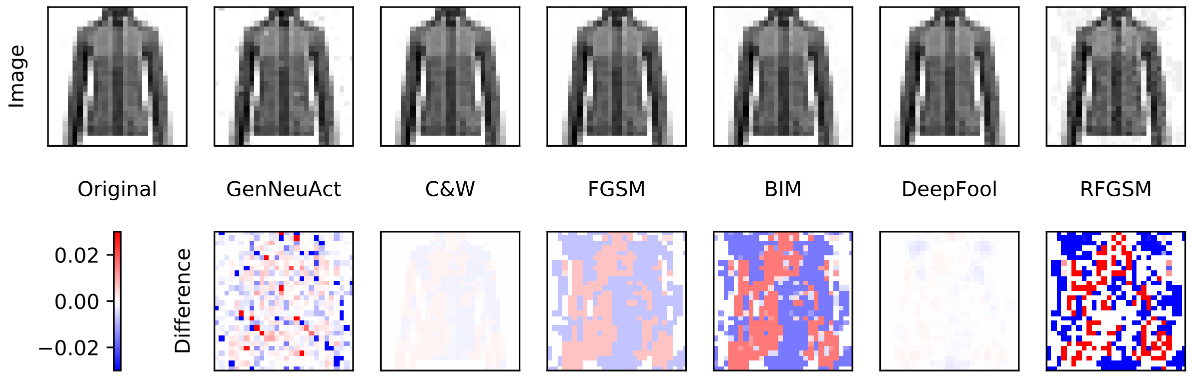


Figure 5.23: Visualization of the adversarial examples created by the different methods for one data point, using an adversarially trained network. The top row shows the adversarial example, and the bottom row shows a magnified colormap of the difference between the original picture and the adversarial example. GenNeuAct, C&W, FGSM, BIM, and RFGSM created a successful adversarial example. Only DeepFool failed to produce one.

5.3. Analysis

In this section we discuss the results that were presented in the previous section. Different performance measures are evaluated, and for striking results an explanation is sought.

5.3.1. Success rate

In the results from the Indian Pima Diabetes dataset, C&W and GenNeuAct clearly distinguish themselves from the other methods, leaving a large gap in success rate between them and the next-best method. The results for natural and adversarial training are similar, but the performance of GenNeuAct on the adversarially trained network varied, giving a large standard deviation. FGSM performed very badly on this dataset.

For the Breast Cancer Wisconsin dataset we see comparable results. For natural training, C&W and GenNeuAct are closely followed by RFGSM, BIM and DeepFool. On the adversarially trained networks, however, C&W and GenNeuAct remain steady while the success rate of RFGSM, BIM, and DeepFool dropped significantly. Again, the success rate of FGSM is very low.

On the Handwritten Digits dataset we see a somewhat different behavior. For natural training, C&W, GenNeuAct and RFGSM have excellent success rates, while BIM, DeepFool and FGSM fail to get even half the success rate of RFGSM. For adversarial training, both GenNeuAct and RFGSM drop in success rate, but keeping them well above BIM, DeepFool and FGSM.

On the final dataset, Fashion-MNIST, C&W comes out on top as it again reaches success rates close to 100%. C&W is followed by GenNeuAct, RFGSM, BIM, FGSM and DeepFool in that order. The ranking order is the same for natural and adversarial training.

Overall, the success rate for each method on the different datasets is quite similar. C&W has the highest success rates (close to 100%), followed by GenNeuAct on all datasets. The other methods vary a bit in rank throughout the results, but in most cases GenNeuAct is followed by RFGSM, BIM, DeepFool, and FGSM in that order. Despite the seemingly unbeatable C&W attack, GenNeuAct has a very good success rate, outperforming several state-of-the-art white-box attack methods. The combined success rates for all experiments are shown in figure 5.24. The median of the success rate of GenNeuAct lies over 25% higher than the next-best method RFGSM.

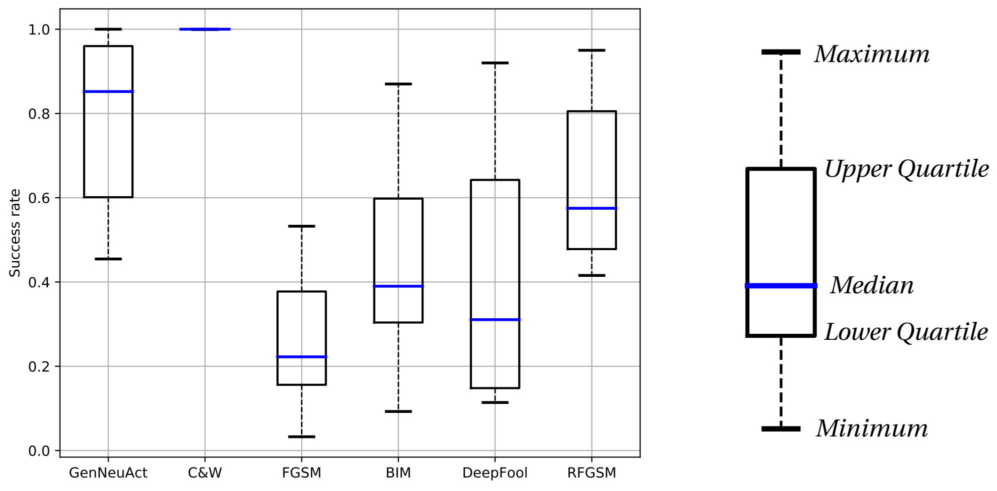


Figure 5.24: Success rate per method, combining all results (combining all four datasets and natural as well as adversarial training).

5.3.2. Mean distance

The success rate is not the only important measure for an adversarial attack method. Besides finding an adversarial example, it is also important that the adversarial example is useful, meaning that it is very similar to the original data point. This is evaluated by calculating the mean distance of the adversarial examples to the original data points. To attain even more insight into the distances, the distances for all successful adversarial examples are combined and summarized in figure 5.25.

It is not straightforward how to compare the distances, as higher distances occur more often for methods

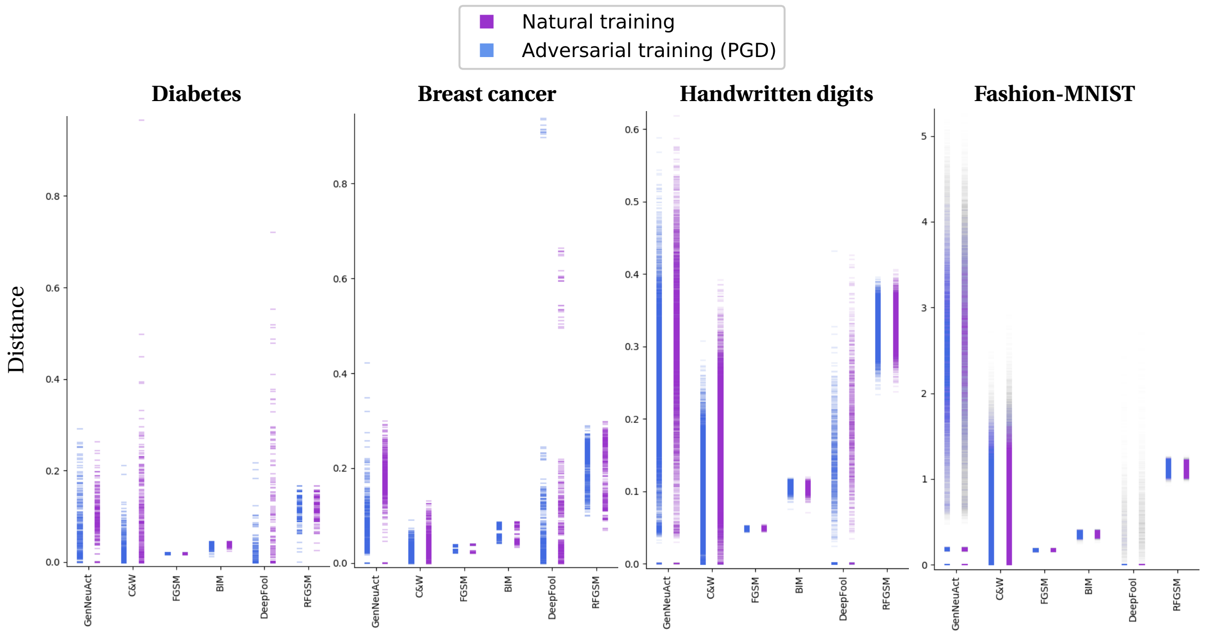


Figure 5.25: Overview of the distances between successful adversarial examples and the original data points for all methods and datasets.

with higher success rates. On all datasets it is visible that C&W can create close adversarial examples, as the color bars in the figure start close to zero. This is not necessarily reflected in the mean distance, as C&W is has a very high success rate, with also adversarial examples with higher distances. Looking at the mean distances, FGSM appears to perform best. For the adversarial examples found by this methods, the distances are promising, but the success rate is very low. Comparing to the other methods in the figure, we see that the other methods find adversarial examples with similar distances; the other methods mostly have a higher mean distance as they find more adversarial examples. Similar reasoning can be applied to BIM and RFGSM, noting that their distances are already larger than FGSM. Even though DeepFool's success rate is similar to BIM and FGSM, the distances for this method vary much more, resulting in better as well as worse adversarial examples than FGSM and BIM. GenNeuAct has a relatively high mean distance on all datasets. The distances of the adversarial examples are, however, spread widely, meaning there is a high diversity in the quality of the adversarial examples. Comparing to C&W (both methods have high success rates), the adversarial examples of C&W seem to be superior to the adversarial examples of GenNeuAct, since the distances of GenNeuAct are more widely spread and the spread mostly starts at a larger distance than C&W.

5.3.3. GenNeuAct

Overall, GenNeuAct seems to be a promising method with unique adversarial examples. It has a high success rate, but relatively large perturbations. To explain our findings, this section will elaborate some more on the inner workings of the method, and how it gets to a certain adversarial example. This section also presents some advantages and disadvantages of the method.

Neuron activations of adversarial examples

As GenNeuAct is focused on changing neuron activation values, it is useful for understanding to show the effect of the method. Hence, we show the difference in neuron activation values for some original data points and the corresponding adversarial examples. For clarity, we first look at a data point from the simpler Iris dataset with a smaller neural network. Next, a data point from the Handwritten Digits dataset will be analysed.

Simpler example for Iris dataset

In figure 5.26 the effect of GenNeuAct on the neuron activations is shown for the simpler Iris dataset. The neural network was naturally trained with three hidden layers of size 2, 3, and 2 respectively. The original data point that was used for this visualization belongs to class 2 (shown in green in the figure), and the adversarial example created by GenNeuAct was misclassified as class 1 (shown in red in the figure). For the neurons that are not shown in the figure, either the neuron was dead or the neuron activation values did not change significantly for the adversarial example.

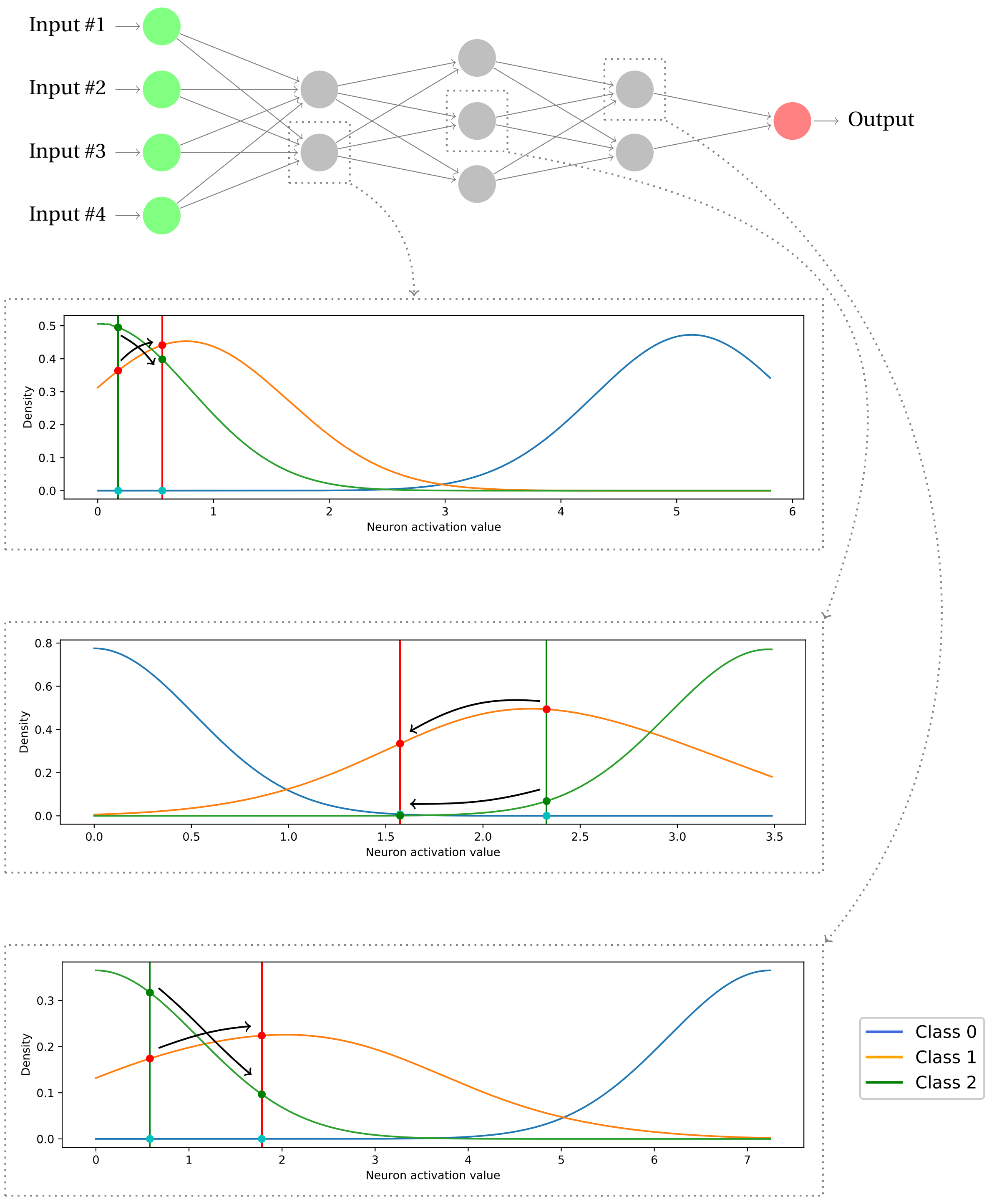


Figure 5.26: Visualization of the change in neuron activation values when creating adversarial examples. At the top of the figure the neural network structure is shown, highlighting three nodes for which an explanatory plot is shown below. Each plot shows the neuron activation values of the original data point (class 2) and the generated adversarial example (class 1). The vertical colored bars show the classification of the network for the data point.

The three graphs show how GenNeuAct works. The distribution of the neuron activation values of the training set for each class label is shown for each highlighted neuron. The neuron activation value of the original data point is represented by the green vertical line, where the intersections with the density functions are also highlighted. GenNeuAct tries to find an adversarial example for which the activation value of the correct class lowers, and highers for other classes. This is clearly visible in the first graph, as the density of class 2 falls and the density of class 1 even rises above the density of class 2 for the adversarial example. The hidden neuron in the third layer, displayed in the third graph, shows similar findings. In the second graph, the situation is different. For the original data point, class 1 already had the highest density value. For the adversarial example the density of the original class drops, while at the same time the density of class 1 also drops a bit. As the final decision of the neural network is created by a combination of these neurons, not all neurons have to agree on the most likely class (as with the original data point). GenNeuAct tries to find the smallest perturbation to the input, such that the altered neuron activation values lead to a misclassification.

Example Handwritten Digits dataset
In figure 5.27 the effect of GenNeuAct on the neuron activations is shown for the Handwritten Digits dataset. The neural network was naturally trained with two hidden layers of size 25. The original data point that was used for this visualization belongs to class 0 (shown in blue in the figure), and the adversarial example created by GenNeuAct was misclassified as class 3 (shown in green in the figure). The four selected neurons for which a graph is shown are randomly chosen, excluding dead neurons and neurons where the activation values did not change significantly.

The graphs show that GenNeuAct tries to find an adversarial example for which the activation value of the correct class lowers, and highers for other classes. For all four graphs the density of the correct class (blue line) drops significantly for the adversarial example. Since there are 50 hidden neurons, it is not unlikely that there are a few neurons for which this is not the case, but these will be a minority. The first graph shows a neuron for which the neuron activation value drops to a value where class 6 (pink), class 7 (gray) and class 3 (green) are the three classes with the highest density value, in that order. In the second and third graph, however, the new activation values make classes 6 and 7 highly unlikely. Throughout the figures, the density of class 3 (green) for the adversarial example is decent, and on a whole sufficient to cause the misclassification.

The development of the adversarial example corresponding to figure 5.27 is shown in figure 5.28. The progression is shown for both GenNeuAct and RFGSM. The leftmost picture shows the original data point, and the rightmost picture shows the final adversarial example. The four pictures in between show intermediate results from the methods. RFGSM, which uses the gradient of the loss function with respect to the input to find adversarial examples, fails to create an adversarial example for this data point as all its attempts are classified as the correct class ’0’. The final image of GenNeuAct is a successful adversarial example, and is classified by the network as ’3’. The final image shows that the desired changes in the neuron activation values cause some important pixels to alter their values to be more resemblant of a three, but also quite some noise is generated as several pixels are activated which have a value of zero for all digits.

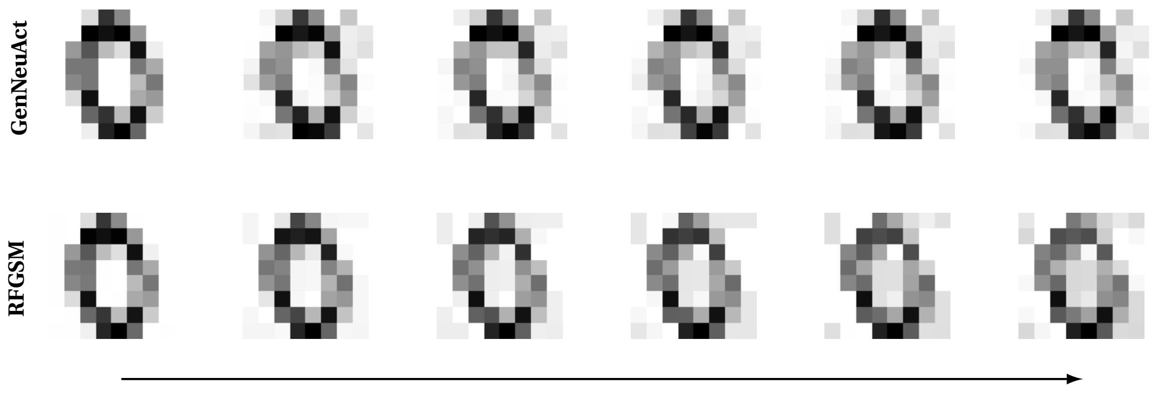


Figure 5.28: Development of an adversarial example for GenNeuAct and RFGSM. For RFGSM all images are classified as the correct class label ’0’. For GenNeuAct the last image represents a successful adversarial example as the network classifies it as ’3’.

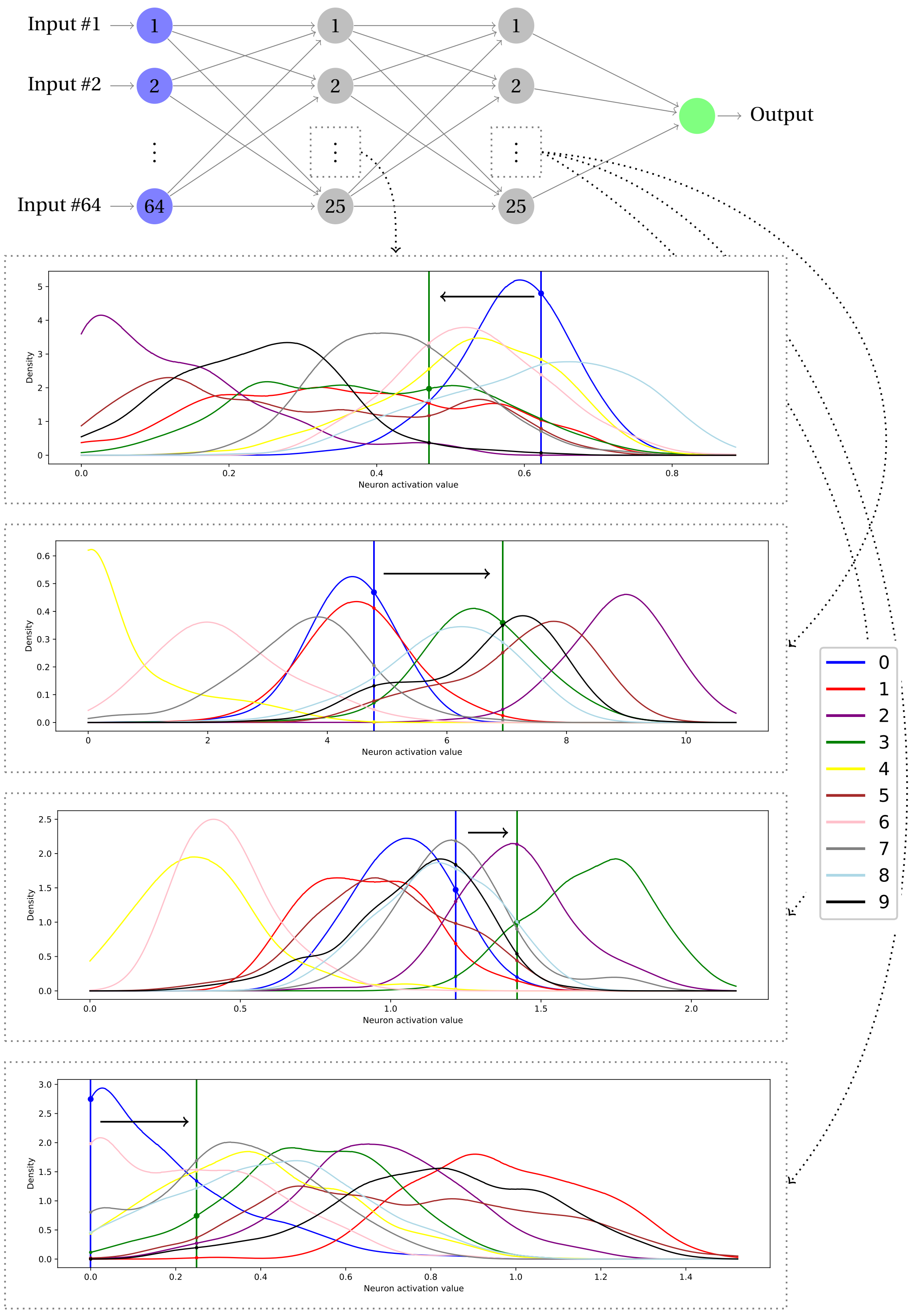


Figure 5.27: Visualization of the change in neuron activation values when creating adversarial examples. At the top of the figure the neural network structure is shown, highlighting four hidden nodes for which an explanatory plot is shown below. Each plot shows the neuron activation values of the original data point (class 0) and the generated adversarial example (class 3). The vertical colored bars show the classification of the network for the data point.

5.3.4. Dominance of C&W

The results discussed before clearly show that C&W is superior to the other methods, with an impeccable success rate and acceptable distances. It is, of course, interesting to investigate why this method is so successful, as these aspects can be used to increase the effectiveness of future methods.

Every adversarial attack method is designed to optimize a specific formula. For L-BFGS, FGSM, BIM and RFGSM, this formula is based on the gradient of the loss function with respect to the input. DeepFool has a different approach, where a step is taken in the direction of a simplified, linearized solution. To get to this solution, the differences in the gradient of the model output of the actual class with respect to the input and the gradient of the output for other classes with respect to the input are used. C&W distinguishes itself from the other methods by having two different components in their optimization function: a loss component and a distance component. The loss component influences the success rate and the distance component influences the effectiveness (or similarity) of the attack. These components form a trade-off which is measured by the magnitude of a variable c. Hence, this method can determine a favourable trade-off value for each different network. Its importance is visible when the optimization formula is rephrased to simplified version:

$$\min \left[\begin{array}{c}\text{Distance}\\ \text{component}\end{array} + c * \begin{array}{c}\text{Loss}\\ \text{component}\end{array} \right]$$

The value of c is determined by a binary search. Possibly, the distinguished trade-off method of C&W is the reason for the superiority of its adversarial examples. The superiority does, however, come with a cost, as this method is much more computationally expensive than FGSM, BIM, RFGSM and DeepFool.

Let us further investigate the importance of the trade-off method of C&W by skipping over the binary search, and run the attack with different values of c. The results for a network trained on the Fashion-MNIST dataset is shown in figure 5.29. The trade-off is clearly visible, as both the success rate and average distance increase when c increases. With low values for c, for example the lower bound of the binary search (10^{-3}), the attack is barely successful. With high values for c the attack favors success over distance, and as such it is very successful but the average distances keep rising. This emphasizes the importance of this variable, as the performance of this method fully depends on the value of c.

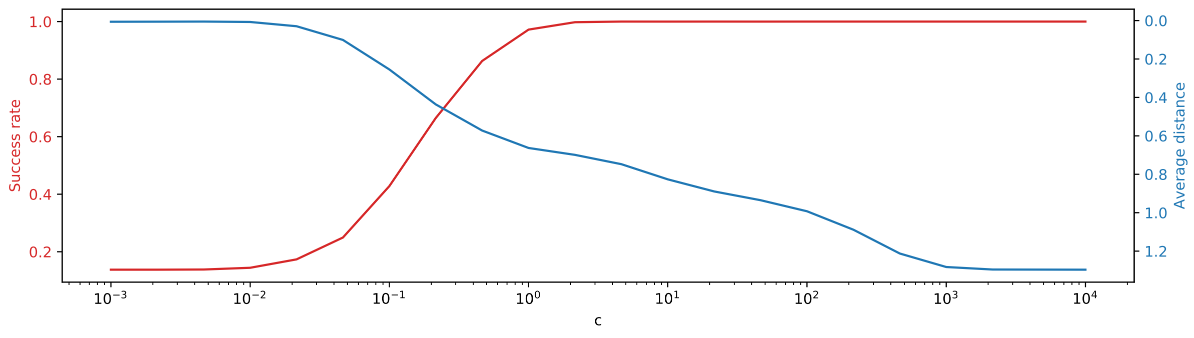


Figure 5.29: Success rates and mean distance of the C&W attack for different c values.

5.3.5. Natural versus adversarial training

The results for the adversarially trained networks compared to the naturally trained networks are similar for each dataset. When the network is trained with adversarial examples, it should become harder for the methods to find good adversarial examples. This is clearly reflected in the results, as the success rate decreases and the mean distance increases in most cases. The increase in distance was already shown in figure 5.25, where the adversarial distances are located higher and are more widely spread. The differences between the success rate and mean distances of the different training methods are shown in table 5.7, where the mean was taken for each set of five experiments.

As expected, the success rates decrease as the adversarially trained networks are more robust and thus it is harder for the methods to find adversarial examples. In some cases, such as for C&W, the success rate barely changes but the mean distance does increase. As a good adversarial example has a small distance, the increase in distance shows that C&W also has more difficulties with finding good adversarial examples.

Similar to the other methods, the performance of GenNeuAct decreases when adversarial training is applied. As such, we investigate the difference in neuron activation values for naturally and adversarially trained

Method	Pima Indian Diabetes		Breast Cancer Wisconsin	
	Difference success rate	**Difference mean distance**	**Difference success rate**	**Difference mean distance**
GenNeuAct	-0.265	0.033	-0.016	0.095
C&W	0.000	0.050	0.000	0.016
FGSM	-0.125	0.000	-0.128	0.002
BIM	-0.200	0.003	-0.344	**-0.014**
DeepFool	**0.034**	0.062	-0.116	0.023
RFGSM	-0.153	0.000	-0.290	**-0.009**

Method	Handwritten Digits		Fashion-MNIST	
	Difference success rate	**Difference mean distance**	**Difference success rate**	**Difference mean distance**
GenNeuAct	-0.424	0.051	-0.086	0.030
C&W	0.000	0.058	0.000	0.082
FGSM	-0.069	0.000	-0.015	0.000
BIM	-0.241	**-0.002**	-0.042	0.003
DeepFool	-0.010	0.061	-0.002	0.029
RFGSM	-0.422	0.004	-0.077	**-0.010**

Table 5.7: Overview of the change in value of the success rate and mean distance for natural and adversarial training. If the success rate has risen or the mean distance is reduced, the value is printed in bold. The difference is calculated by taking the mean of the results from the five experiments with adversarial training, minus the mean of the five experiments with natural training.

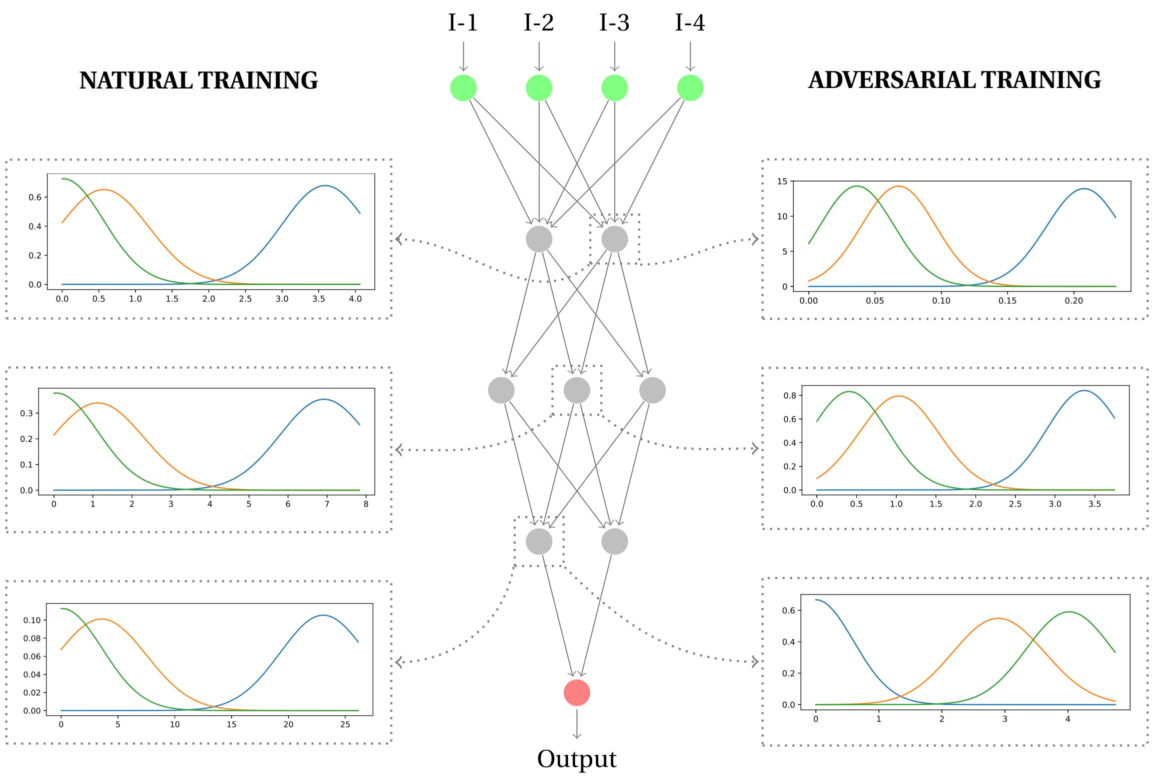


Figure 5.30: Difference in neuron activation distributions for a naturally and an adversarially trained network on the Iris dataset.

networks. Comparing the density functions of neuron activation values is not straightforward, as training a network on identical data (even without changing to adversarial training) can lead to completely different neuron activation value distributions. Hence, we use the small Iris dataset with a small number of hidden neurons to investigate this. Our findings are shown in figure 5.30.

For the naturally trained network, the density functions look very similar throughout the network, only the scales are different. In the first two hidden layers we see similar behavior for the adversarially trained network: the density functions look very similar to the corresponding density function of the naturally trained network, with a different scale. Despite the similar shape, there is a very noteworthy change for the adversarially trained network: a larger part of the curves is covered. In the first two hidden layers this difference is visible in the left of the density functions, as the peak of the green curve has shifted to the right. In the third hidden layer we see something different, as the three classes have a completely different density function compared to the naturally trained network. The significance of the change in activation distributions becomes clear when looking at an example, shown in figure 5.31

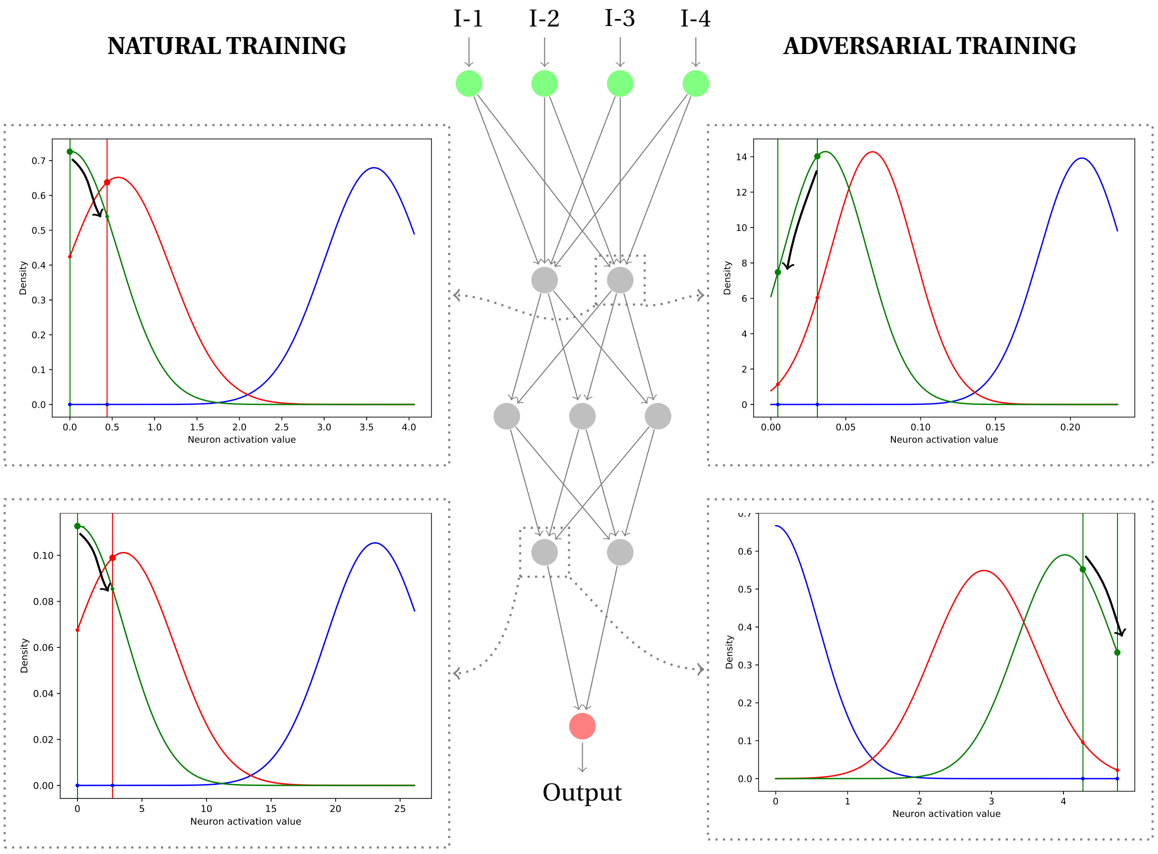


Figure 5.31: Visualization of the change in neuron activation values when creating adversarial examples, showing the differences in natural and adversarial training. Each plot shows the neuron activation values of the original data point (green dot) and the generated adversarial example (red dot if succeeded, else green dot). The vertical colored bars show the classification of the network for the data point.

In the figure on the left we see the change in neuron activation values of a data point and an adversarial example created by GenNeuAct. The methods finds inputs with neuron activation values such that we descend the green curve, and climb curves of other classes, making misclassification more likely. The plots on the right show the process of finding an adversarial example on an adversarially trained network. Due to the adversarial training, the neuron activation value of the original data point has moved further away from the other density curves, to the other side of the peak of the green curve. When the method descends its curve now, we have similar problems to gradient descent where we only find local optima and fail to find an adversarial example. As GenNeuAct uses a genetic algorithm with random mutations, there are chances to escape the local optima. As other curves are further away, the mutations are less likely big enough to escape the local optima, resulting in lower success rates on adversarial training. Future work can consider tackling this challenge by altering the loss function such that getting higher on the curves of a different class is more heavily weighted than descending the curve of its own class. In this work we only considered PGD for adversarial training; other methods may have different effects on the activation density functions.

We further investigate the difference in neuron activation values for natural and adversarial training with the Mann-Whitney *U* test. This is a nonparametric statistical test to determine if two independent samples come from the same distribution. We perform this test for every neuron and every combination of output classes, where the two samples are the neuron activation values in that specific neuron for the two selected classes. When the *pvalue* resulting from the test is lower than 0.05, the samples differ significantly and we conclude that the samples came from different distributions. We are interested in whether the percentage of significantly different neuron activations per output class pair drops when adversarial training is used. The results are shows in table 5.8.

	Natural training	**Adversarial training**
Pima Indian Diabetes	0.724	0.681
Handwritten Digits	0.941	0.919
Wisconsin Breast Cancer	0.968	0.949
Fashion-MNIST	0.962	0.961

Table 5.8: Difference in percentages of significantly different activation values for each combination of output classes in every neuron (dead neurons excluded). This table shows the average percentage over the five networks that were trained in the corresponding setting.

The table shows a slight drop in significantly different neuron activation values for each dataset. This could explain the increasing difficulty for GenNeuAct in finding adversarial examples when adversarial training is used. However, the decreases are small and therefore it is hard to say how big its role is in increasing the difficulty of finding adversarial examples.

5.3.6. Time efficiency

The run time of GenNeuAct depends on the size of the dataset and the number of hidden neurons. At the start of the method, the estimated density function needs to be calculated for every neuron and every output class. For large networks, this task can become time consuming. However, the calculation of the estimated density functions only needs to happen once and as such the time consequences are reasonable.

The basic steps of the genetic algorithm that are used for creating adversarial examples are fairly efficient. These steps, of course, take longer when the population size or number of generations is increased. The bottleneck in the genetic algorithm for GenNeuAct is the fitness calculation. To determine the fitness of a data point the data point first needs to be put through the network to retrieve its neuron activation values. Thereafter the neuron activation value for each neuron is scored against the estimated density functions of all different output classes. As one can expect, the more hidden neurons the network has and the more output classes the dataset has, the longer this process takes. The accelerated alternative speeds up this process by replacing the density functions with bins, making the scoring procedure less expensive.

Currently the time efficiency of GenNeuAct is undoubtebly worse than some of the state-of-the-art methods (FGSM, BIM, RFGSM and DeepFool). However, the time efficiency of the methods also depend a lot on the implementation. In our experiments the implementation of the *Torchattack* library is used for most state-of-the-art methods. These implementations have been designed and reviewed to be as fast as possible. The focus of this work was to explore the possibilities around neuron activation values, wherefore the speed of the algorithm was not a main priority. A re-implementation of GenNeuAct with a focus on efficiency will likely accelerate the process.

That the implementation is important is also shown by the C&W attack that was used in our experiments. We did not use the *Torchattack* library for the C&W attack, as its implementation does not include the binary search for the parameter c (due to the corresponding time restraints). Since the binary search is important for the high performance of the method, we decided to include this and use the implementation of Kaiwen Wu [37]. Even though very successful, the implementation is not efficient. The C&W attack is by nature more expensive than methods such as FGSM, but this implementation is unnecessarily slow. As our implementation of GenNeuAct is not (yet) optimized for time efficiency, GenNeuAct may also be unnecessarily slow.

The choice of machine learning library may also have a positive impact on the run time of GenNeuAct. The current implementation uses the *PyTorch* library, which is in hindsight probably not the most lucrative choice for our method. This library does not store the intermediate neuron values, only giving us the network output. By registering a forward hook in the network, we were able to keep track of the neuron activation values of a single data point. Because of the inner workings of the *PyTorch* library, however, it is not possible to get all neuron activation values from a batch; with our hook we can only extract the neuron values from the last data point in the batch. Consequently, when we want to get the neuron activations from a set of points, we need to put these points through the network one by one, which is more expensive than putting a batch through the network. By using a different library or framework which does not have this limitation, GenNeuAct's time efficiency may significantly increase as the method needs the intermediate neuron values a lot.

On a whole, GenNeuAct is currently not time efficient enough to compete with most state-of-the-art methods. There are, however, still many aspects of the implementation that can be improved for time efficiency. This may result in a speed-up large enough to compete with the state-of-the-art methods concerning time efficiency.

6 Discussion

In this work we introduced and evaluated a new white-box attack method which uses neuron activation values to create adversarial examples. This chapter points out some limitations concerning the generalizeability of the obtained results. Furthermore, recommendations for future work are presented.

6.1. Limitations

As the scope of this research was limited, we identify some limitations concerning the generalizeability of the results. As the limitations can be overcome by more extensive research, the limitations also already show some possibilities for future work.

Neural network type

Throughout the research, only relatively simple neural networks are considered. All networks used in experiments consist of fully connected layers and use the ReLU activation function. The findings therefore only apply to this type of neural network. Future work could explore the use of different activation functions and the use of different layer types, such as convolution layers.

Representativeness of datasets

Another limitation of this research concerns the use of different datasets for testing the performance of GenNeuAct. In the final experiment only four datasets were considered. Even though the datasets were selected because of their variety and recommendations by other works, the datasets may not be representative of all datasets. As such, it is unsure to which extent the results are generalizable. Future work could further investigate this.

Parameter tuning

Each attack method has its own set of parameters, which can be changed to alter the attack in a desired way. Examples of some method specific parameters are the maximum perturbation (ϵ), step size (α) and the number of steps. As we already saw for the c parameter in the C&W attack, the parameter settings can have a big influence on the method's performance. In this work, all experiments are carried out with the default parameter settings of the method. No parameter tuning was done, and as such the result could be different and perhaps even better when (dataset specific) parameter tuning is done. The same goes for GenNeuAct, as the method could also benefit from more parameter tuning.

6.2. Recommendations for future work

During the research we noticed several potential improvements and extensions to the proposed method, which were not explored due to time constraints. Hence, we present these ideas here as a recommendation for future work.

Acceleration

Currently, GenNeuAct is quite computationally expensive, causing the method to be rather slow. The focus of this work was to explore the possibilities around neuron activation values, wherefore the speed of the algorithm was not a priority. A re-implementation of the proposed method with a focus on efficiency will likely accelerate the process.

The choice of machine learning library may also have a positive impact on the run time of the method. The current implementation uses the *PyTorch* library, which is in hindsight probably not the most lucrative choice for our method. This library does not store the intermediate neuron values, only giving us the network output. By registering a forward hook in the network, we were able to keep track of the neuron activation values of a single data point. Because of the inner workings of the *PyTorch* library, however, it is not possible to get all neuron activation values from a batch; with our hook we can only extract the neuron values from the last data point in the batch. Consequently, when we want to get the neuron activations from a set of points, we need to put these points through the network one by one, which is more expensive than putting a batch through the network. By using a different library or framework which does not have this limitation, GenNeuAct's time efficiency may significantly increase as the method needs the intermediate neuron values a lot.

Distance-loss trade-off

Most well-known and state-of-the-art white-box methods use the gradient of the loss function to find adversarial examples. These methods limit the step-size to assure small perturbations. With a step-size that is small enough, the methods aim to find the closest adversarial example. C&W has a different approach, which is very successful according to our results. The optimization formula of C&W consist of two components instead of one. Similar to the previously mentioned methods and GenNeuAct, one of these components is the loss component, which focuses on the success rate of the adversarial examples. C&W adds another component to their optimization formula: a distance component. The importance of the distance component over the loss component is determined by a variable. This variable showed to be of high importance for the performance of the method, and experiments showed that this variable captures the trade-off between success and distance exquisitely. Our intuition is that more methods can benefit by incorporating this trade-off. As such, for future work we recommend to extend the implementation of GenNeuAct by including a separate distance component in the optimization formula.

Classification with neuron activation values

In our proposed method we use neuron activation values to predict class labels. In chapter 4 we showed how we get class probabilities from the combination of the distributions of neuron activation values of all neurons. We noticed that the final class probabilities point to one class with high confidence, as the probability for this class approaches 1 and the probabilities for all other classes approach 0. GenNeuAct uses this to attack the network, but our intuition is that this could also be very useful for improving the robustness of a neural network. Currently, the classification of a data point is only based on the output layer of a neural network. For future work we propose to use the neuron activation values to classify a data point instead. As this method takes more network information into account and makes predictions with more certainty than the regular method, we expect that a network using activation based classification will be more robust against adversarial attacks.

7 Conclusions

In this research we investigated the behaviour of neuron activation values with the intention of finding patterns from which a class prediction can be extracted, and in turn adversarial examples can be generated in a new fashion. We proudly conclude that neuron activation values contain meaningful information with which we can generate new adversarial examples. As such, the main research question was answered. Below we summarize the answers to the sub-questions of the research:

- **What differences are visible in neuron activation values for different classes, and does this hold for different network sizes?**
  By visualizing neuron activation values, we found that neurons have different distributions of neuron activation values for different classes. The distributions seem to have a different range for each class, for which almost all activation values fall into this range. The ranges for a single neuron are often partly overlapping, making it mostly infeasible to draw conclusions based on a single hidden neuron. We investigated the difference in distributions for different network sizes. With more hidden neurons, the distributions of neuron activation values for different classes are still distinguishable in a single neuron. We found that on average in 89% of the cases the neuron activation values of two different output classes are significantly different, considering 40 networks trained naturally as well as adversarially on four different datasets.

- **How can we deduce the class label from neuron activation distributions, and how can we use this to create adversarial examples?**
  Taking the partly overlapping distributions from the previous sub-question, we can derive an approximate probability of a (new) neuron activation value belonging to a particular class. These probabilities are calculated for each hidden neuron and each class, after which the probabilities of each neuron are combined into a single probability for each class. Experiments show that this final probability is a good predictor for the class label of the data point.

  We presented a method, GenNeuAct, which uses the probabilities following from the neuron activation values to create adversarial examples. The method uses a genetic algorithm to find small perturbations to the original data point, such that the neuron activation values change. Consequently, we expect the class prediction of the network to also change, generating an adversarial example. The intuition and process of finding an adversarial example with GenNeuAct are visualized in figure 5.26, where the change in neuron activation values causes misclassification by the network. With this technique we find different adversarial examples than state-of-the-art methods.

- **What is the performance of the new method using neuron activation values, and how does this compare to the state-of-the-art?**
  The performance of GenNeuAct is evaluated and compared by running experiments on four datasets: the Pima Indian Diabetes dataset, the Wisconsin Breast Cancer dataset, the Handwritten Digits dataset, and Fashion-MNIST. For each dataset, 5 networks are trained naturally and 5 networks are trained adversarially (with PGD), after which the performance of GenNeuAct is compared to five state-of-the-art methods (C&W, FGSM, BIM, DeepFool, and RFGSM). The performance of the adversarial examples are

evaluated using different measures, the most important measures being success rate and quality (distance). With an average success rate of 79%, GenNeuAct beats most state-of-the-art methods, such as FGSM (25%), BIM (44%), DeepFool (40%) and RFGSM (64%). Only C&W achieved superior success rates, approaching an average success rate of 100%. The quality of the adversarial examples created by GenNeuAct is comparable to the state-of-the-art.

A big strength of GenNeuAct is the network coverage, as the method explores a larger part of the search space. Also, to our knowledge, this is the first work to (successfully) utilize intermediate network information for guiding the search to adversarial examples.

This work has proven the value of (intermediate) network information, by proposing a method for creating adversarial examples solely based on neuron activation values. GenNeuAct is a promising method with unique adversarial examples, compatible with the state-of-the-art. It has a high success rate, but relatively large perturbations. For further development of the method we recommend to lower the size of the perturbations by balancing the loss-distance trade-off, investigate the effect of different neuron activation loss functions, and improve the speed of the method by efficient reimplementation.

Besides introducing a new algorithm and evaluating its performance compared to state-of-the-art methods, we also investigated *why* these results were obtained, providing a deeper understanding of the inner workings of the methods and developing new insights. For deeper understanding, we visualized how neuron activation values and their density functions change for different input classes, and we also visualized how these density functions are used in the process of creating adversarial examples. During analysis we found that the trade-off between success rate and distance has a huge impact on the results of a method. Our findings on the C&W attack show that this trade-off can be carefully balanced by formulating an optimization formula with a separate loss and distance component. Furthermore we gained insight into the effect of adversarial training on neuron activation values. The adversarial training impacts the neuron activation values such that the estimated density functions move in a way that makes it harder to find a close adversarial example. Future work can utilize these insights to develop even better attacks and defenses for neural networks.

Bibliography

[1] Hussein A Abbass. An evolutionary artificial neural networks approach for breast cancer diagnosis. *Artificial intelligence in Medicine*, 25(3):265–281, 2002.

[2] Moustafa Alzantot, Yash Sharma, Supriyo Chakraborty, Huan Zhang, Cho-Jui Hsieh, and Mani B Srivastava. Genattack: Practical black-box attacks with gradient-free optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1111–1119, 2019.

[3] Maral Amir and Tony Givargis. Priority neuron: A resource-aware neural network for cyber-physical systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2732–2742, 2018.

[4] Edgar Anderson. The species problem in iris. *Annals of the Missouri Botanical Garden*, 23(3):457–509, 1936.

[5] Rob Ashmore and Elizabeth Lennon. Progress towards the assurance of non-traditional software. In *Safety-critical Systems Symposium*, 2017.

[6] Safial Islam Ayon, Md Islam, et al. Diabetes prediction: A deep learning approach. *International Journal of Information Engineering & Electronic Business*, 11(2), 2019.

[7] Thomas Back, Ulrich Hammel, and H-P Schwefel. Evolutionary computation: Comments on the history and current state. *IEEE transactions on Evolutionary Computation*, 1(1):3–17, 1997.

[8] Jonathan S Barlow. Data-based predictive control with multirate prediction step. In *Proceedings of the 2010 American Control Conference*, pages 5513–5519. IEEE, 2010.

[9] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[10] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*, pages 39–57. IEEE, 2017.

[11] Po-Hung Chen and Hong-Chan Chang. Large-scale economic dispatch by genetic algorithm. *IEEE transactions on power systems*, 10(4):1919–1926, 1995.

[12] William A Crossley and David H Laananen. Conceptual design of helicopters via genetic algorithm. *Journal of Aircraft*, 33(6):1062–1070, 1996.

[13] Richard A Davis, Keh-Shin Lii, and Dimitris N Politis. Remarks on some nonparametric estimates of a density function. In *Selected Works of Murray Rosenblatt*, pages 95–100. Springer, 2011.

[14] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL `http://archive.ics.uci.edu/ml`.

[15] Gamaleldin Elsayed, Dilip Krishnan, Hossein Mobahi, Kevin Regan, and Samy Bengio. Large margin deep networks for classification. In *Advances in neural information processing systems*, pages 842–852, 2018.

[16] Larry J Eshelman and J David Schaffer. Real-coded genetic algorithms and interval-schemata. In *Foundations of genetic algorithms*, volume 2, pages 187–202. Elsevier, 1993.

[17] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2): 179–188, 1936.

[18] Sushmito Ghosh and Douglas L Reilly. Credit card fraud detection with a neural-network. In *System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on*, volume 3, pages 621–630. IEEE, 1994.

[19] David E Goldberg and John Henry Holland. Genetic algorithms and machine learning. 1988.

[20] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[21] Xiaojun Jia, Xingxing Wei, Xiaochun Cao, and Hassan Foroosh. Comdefend: An efficient image compression model to defend adversarial examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6084–6092, 2019.

[22] Yiding Jiang, Dilip Krishnan, Hossein Mobahi, and Samy Bengio. Predicting the generalization gap in deep networks with margin distributions. *arXiv preprint arXiv:1810.00113*, 2018.

[23] Hoki Kim. Torchattacks: A pytorch repository for adversarial attacks. *arXiv preprint arXiv:2010.01950*, 2020.

[24] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1039–1049. IEEE, 2019.

[25] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[26] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*.

[27] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 165–176, 2020.

[28] Po-Ling Loh and Martin J Wainwright. Regularized m-estimators with nonconvexity: Statistical and algorithmic theory for local optima. *The Journal of Machine Learning Research*, 16(1):559–616, 2015.

[29] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 120–131, 2018.

[30] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[31] Deepak Mane and U.V. Kulkarni. Modified fuzzy hypersphere neural network for pattern classification using supervised clustering. *Procedia Computer Science*, 143:295–302, 11 2018. doi: 10.1016/j.procs.2018.10.399.

[32] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[33] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*, pages 372–387. IEEE, 2016.

[34] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[35] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[36] Seng Pan That Pann Phyu and Gun Srijuntongsiri. Effect of the number of parents on the performance of multi-parent genetic algorithm. In *2016 11th International Conference on Knowledge, Information and Creativity Support Systems (KICSS)*, pages 1–6. IEEE, 2016.

[37] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[38] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015. URL `http://networkrepository.com`.

[39] Bernard W Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[40] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.

[41] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Structural test coverage criteria for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–23, 2019.

[42] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[43] Masato Takahashi and Hajime Kita. A crossover operator using independent component analysis for real-coded genetic algorithms. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 643–649. IEEE, 2001.

[44] Florian Tramèr, Alexey Kurakin, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. Ensemble adversarial training: Attacks and defenses. *arXiv preprint arXiv:1705.07204*, 2017.

[45] Petra Vidnerová and Roman Neruda. Evolutionary generation of adversarial examples for deep and shallow machine learning models. In *Proceedings of the The 3rd Multidisciplinary International Social Networks Conference on SocialInformatics 2016, Data Science 2016*, pages 1–7, 2016.

[46] Kaiwen Wu. A rich-documented pytorch implementation of carlini-wagner's l2 attack. `https://https://github.com/kkew3/pytorch-cw2`, 2018.

[47] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.