Tampereen yliopisto

Jukka Ahonen

ROBUSTNESS ANALYSIS OF HIGH-CONFIDENCE IMAGE PERCEPTION BY DEEP NEURAL NETWORKS

Faculty of Engineering and Natural Sciences
Bachelor's Thesis
January 2020

ABSTRACT

Jukka Ahonen: Robustness Analysis of High-Confidence Image Perception by Deep Neural Networks
Bachelor's Thesis
Tampere University
Automation Engineering
January 2020

Deep neural networks are nowadays state-of-the-art method for many pattern recognition problems. As the performance grows, the robustness of them cannot be ignored. Specifically, the lack of robustness against slightly perturbated inputs called adversarial examples has been a hot topic for the last years. The main reason behind this is safety because if one can easily generate an adversarial example that can efficiently "fool" a neural network, it cannot be trusted in a real-life system where safety is an issue.

The goal of this thesis is to better understand deep neural networks, their robustness and tools to test their robustness. One tool called genetic algorithm is further studied and implemented with python to fool an example neural network VGG16 to misclassify images. VGG16 and its prediction probabilities being the fitness function, the algorithm uses crossover, mutation and elitism among other things to find the best adversarial solution. Three types of tests are conducted. First, a false positive randomly generated adversarial example is created. Second, example image is evolved to a false negative adversarial example with perturbation L_∞ limited to 5. And finally, example image is evolved to a false positive adversarial example with perturbation L_∞ limited to 5. When the perturbation is L_∞ limited to 5, the difference between the original image and adversarial image is unnoticeable to the human eye. The tests show that even a rather simple genetic algorithm can make VGG16 misclassify images with high confidence. It seems that deep neural networks without any safety measures are not very robust against adversarial examples.

Keywords: adversarial examples, robustness, genetic algorithm, neural networks

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

GA	Genetic Algorithm
DNN	Deep Neural Network
CNN	Convolutional Neural Network
ILSVRC-2014	ImageNet Large Scale Visual Recognition Challenge 2014

1. INTRODUCTION

Deep neural networks (DNN) and convolutional neural networks (CNN) are nowadays a state-of-the-art method for many image recognition problems. As the performance grows, the robustness of them cannot be ignored. Specifically, the lack of robustness against slightly perturbated inputs called adversarial examples has been a hot topic for the last years [17, p. 1]. The main reason behind this is safety because if one can easily generate an adversarial example that can efficiently “fool” a neural network, it cannot be trusted in a real-life system where safety is an issue.

This thesis is divided into three different sections. In the first section, the basic principle and structure of DNN/CNN are introduced. The first section also deals with the concept of robustness and adversarial examples and how they are linked to neural networks. The second section is about genetic algorithms (GA), how they function and how can they be used as a tool for robustness testing of neural networks. The last section is about the actual robustness analysis for image recognizing CNN. CNN used in the analysis is called VGG16, which is introduced at the beginning of the section. Then, a GA for constructing adversarial examples is implemented with python. Finally, three tests are made towards the VGG16 and the results are analyzed to gain insight into the robustness of the example network.

2. DEEP NEURAL NETWORKS

To understand how to measure the robustness of a neural network, one must first know what a neural network is and how it works. This chapter introduces the fundamentals of feedforward deep neural networks and their subtype convolutional neural networks. After the basic have been gone through, it is discussed what the robustness of neural networks means and how it can be evaluated.

2.1 Structure of neural networks

A neural network is a statistical model which tries to find relationships between input data and output data. According to Abdi & al [1, p. 1], they are said to be adaptive, because they can learn to adjust their parameters with the help of known examples. The network consists of layers which can have multiple neurons. The neurons are building blocks of the network which are connected to the adjacent layers' neurons, see Figure 2.1.

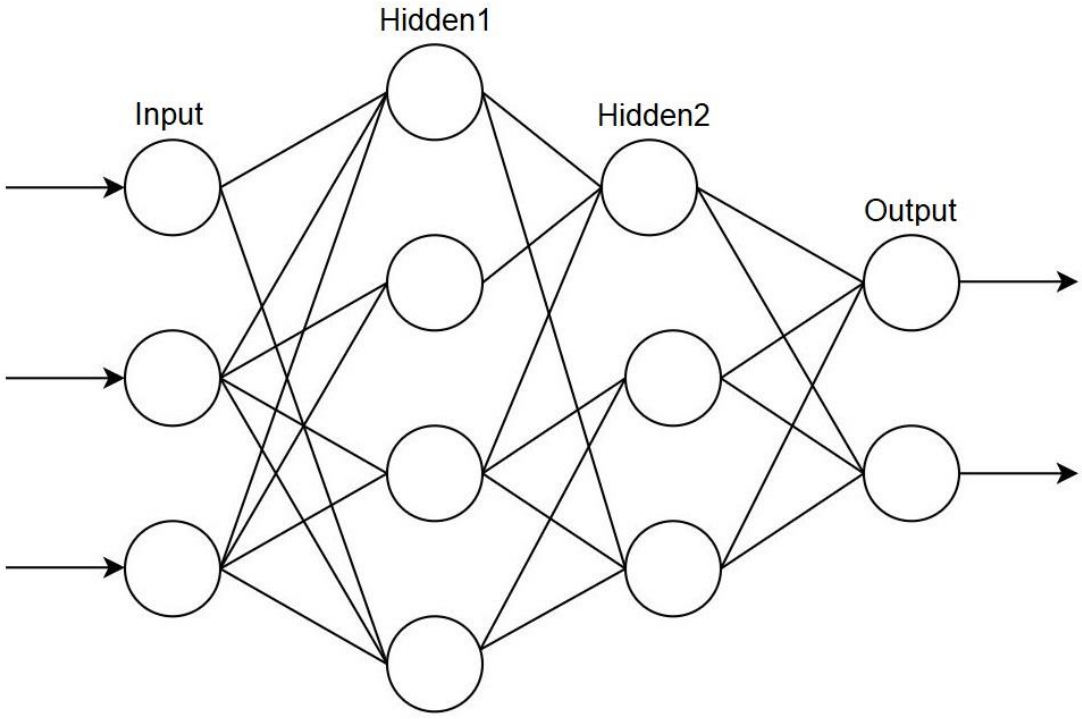


Figure 2.1 *Example structure of a feedforward neural network including an input layer, two hidden layers and an output layer. Circles represent neurons.*

From Figure 2.1 it can also be seen that the inputs are fed to the neurons of the first layer which is called an input layer. The information is then propagated forward to the next

layer which is called a hidden layer. This continues until the final layer called an output layer is reached, which produces some desired output for the information. [1, p. 2] The term deep neural network (DNN) is used to describe a neural network that contains many layers in it.

The connections of the neurons have weights which determine how powerful the connections between the two neurons are. Neurons also have a bias term and activation functions which determine if the neuron is being activated.

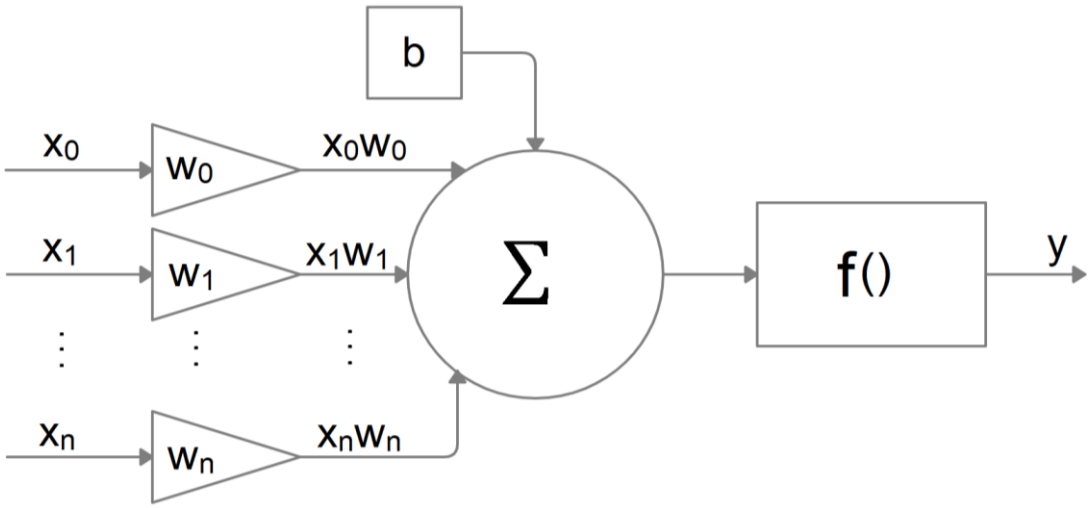


Figure 2.2 *Neuron of a basic neural network including inputs x, weights w, bias b, sum ∑, activation function f and output y*

In Figure 2.2, an example neuron can be seen. First, the inputs $x = [x_0, x_n]$ gain their weights via weight factors $w = [w_0, w_n]$, this happens by a dot product between inputs and weights. The weights exist to determine how powerful the connection is. After inputs have their weights, they are summed together with bias term *b*. This sum will go through the activation function *f*, which determines the output of the neuron. The output from the activation function has some lower and upper limit depending on its type. There exist many different types of activation functions, and the following ones are the most common today:

Rectified Linear Unit (ReLU),

$$f(x) = \max(0, x) \tag{2.1}$$

With an output range of [0, ∞), ReLu is the most commonly used activation function,

because it reduces the vanishing gradients problem and has a great performance. [9, pp. 8–9]

Logistic sigmoid function,

$$f(x) = \sigma(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

It has an output range of (0, 1) and is used mostly on the output layer of binary classification models. [9, p. 5]

Softmax,

$$x_i = \frac{e^{x_i}}{\sum_j^J x_j} \tag{2.3}$$

With an output range of (0, 1) for each probability and sum of them equal to 1. Softmax is used on output layers of multiclass problems. [9, p. 8] For comparison of different activation functions see the paper [9] from Nwankpa & al. They go through and test many different activation functions such as LReLU, PReLU and RReLU, which seem to have a great performance.

When considering the training of a DNN, there exists a forward pass and a backward pass. A forward pass is equivalent to the operations seen in Figure 2.1, where the training samples have passed to the end of the network. In the backward pass, the network output is fed to a cost function *C*, which gives the mean difference of all the actual and predicted values.

One example of the cost function is mean squared error (MSE), which is commonly known as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2, \tag{2.4}$$

Where n is the number of inputs, $\hat{y}$ is the predicted value and y is the true value. As the name states, it calculates the mean squared difference between predicted and true values.

Another commonly used cost function is called Cross-entropy which is used in classification problems. It measures how similar the probability distributions are.

$$Ls = -\sum_{i=1}^{m} \log \frac{e^{W_{y_i}^T x_i + b_{y_i}}}{\sum_{j=1}^{n} e^{W_j^T x_i + b_j}}, \tag{2.5}$$

Where x_i is a feature of ith sample, y_i is the class of x_i, W_j is the parameter vector of j and b_j is the bias. [3, p. 2028]

The goal of the whole training is to minimize the loss function, and this is done during the backward pass by adjusting the weights and biases. This adjustment is done by backpropagating the gradients of the cost function backwards on the net to adjust the weights and biases one by one. [1, p. 74–82] In other words, this process is an implementation of gradient descent into a DNN. Even a simple backpropagation can be quite laborious by hand, therefore today with DNNs where there can be hundreds of layers, backpropagation is done with the help of computers.

When the network has seen all the input data and used them to update the weights once, one Epoch has passed. This means that the network has now taken one step towards the negative gradient and another cycle can begin. After the network is trained with a suitable amount of training samples, it can be used to predict for example a class for the input. When predicting, the network similarly does the forward pass, and, in the end, activation functions of the output layer give the probability of input belonging into each class. The changes in the loss function are the key element to keep in mind when considering the robustness of neural networks, which will be discussed in later chapters.

Convolutional neural networks (CNN), see LeCun & al [6], are a subtype of DNN that are state of the art at image classification tasks. Typically, CNN consists of sets of convolutional layers followed by a pooling layer and at the end of the net a few dense layers. A convolutional layer has many filters or “convolutional kernels” just like basic feedforward DNN has neurons. However, they differ from each other in such manner that in the CNN the weights are convolutional operations. The kernels filter the data by performing a dot product between the kernel and corresponding kernel sized area while sliding through the whole input resulting in a feature map. The idea is to extract different features for the next layers so they can concentrate on more detailed features of the data than the first layers. Convolutional layers would shrink the size of the inputs if no padding is added.

Usually, zero-padding is added to preserve the input size for the next layers, so they don't miss any important areas of the input. Zero padding simply means that zeros are added to the input matrix. An example of convolution with zero padding can be seen in Figure 2.3.

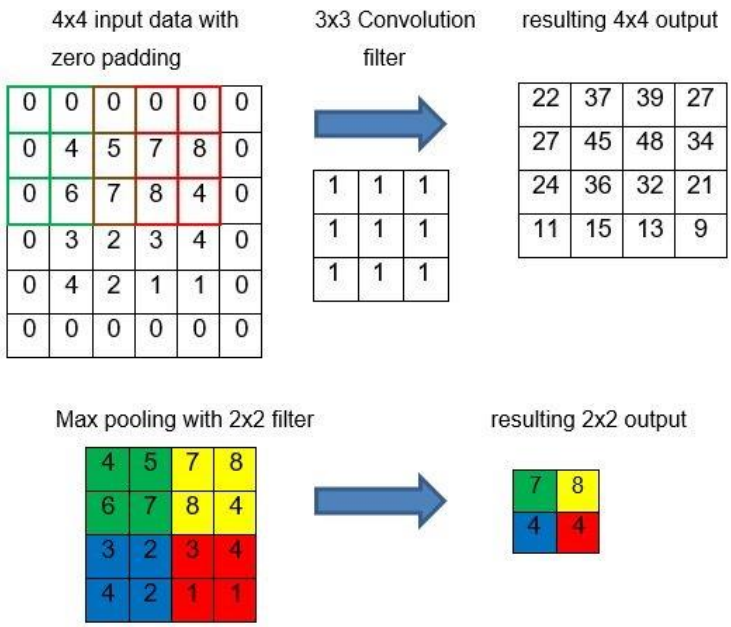


Figure 2.3 *2d convolution operation with 3x3 filter and max-pooling with 2x2 filter*

Pooling layers are another important layer type in CNNs, their function is to reduce the spatial size of the data which reduces the parameters and therefore reduces the needed computing power and the possibility of overfitting. Today mostly max pooling is used which returns the maximum value from each window sized block. Note, that the max-pooling layers always stay the same through the training since they don't have any type of weights to alter. An example of max pooling can be seen in Figure 2.3.

After convolutional layers have extracted the features, and max-pooling layers prevented overfitting, the feature maps are then fed to the dense layers at the end of the network. The dense layers calculate the actual predictions based on the feature maps the convolutional layers have generated.

2.2 Robustness of deep neural networks

The robustness of a DNN can be described as how well the trained model predicts new unseen data and how it reacts to small changes in the input data. A common way to test the robustness is to measure how well the model fairs against adversarial examples

while preserving the ability to handle the task it is designed for. Adversarial examples introduced by Szegedy & al [14] are model inputs that have been designed solely to make the model do a mistake while predicting the output.

It is hard to give any real value to the robustness of DNN. However, Weng & al [16] propose a method to measure the robustness of DNNs called CLEVER, short for Cross Lipschitz Extreme Value for nEtwork Robustness. In [16 p. 2] Weng & al state that: “CLEVER is the first attack-independent robustness metric that can be applied to any neural network classifier.” Since state-of-the-art robustness determination methods are quite complex, to keep this thesis simple and short enough, robustness is estimated only via the effectiveness of adversarial examples compared to the perturbation amount.

2.2.1 Threat model and perturbation

To define how the adversarial should work and what type of information it has access to, a *threat model* is needed. Yuan & al [17] introduce a way to construct an efficient threat model containing four aspects.

The first one is called *adversarial falsification*, which describes the type of adversarial example attack. It can be either *false positive attack* or *false negative attack*. [17, p. 4] False positive meaning the model gives high confidence for input that humans don’t recognize, and false negative meaning human can recognize the input, but the model gives low confidence.

The second aspect is the *adversary’s knowledge*. The adversary can know only the output of the model, having no other access to the model. This is called a *black-box attack*. On the other hand, when the adversary has access to everything in the model it is considered as a *white-box attack*. [17, p. 4] White-box attacks are usually more effective, but when the model structure or training data isn’t accessible it forces one to use a black-box attack.

The third aspect is called *Adversarial specificity.* The attack can either target a certain class or target any class except the original one. The final aspect is *attack frequency* describing one-time attacks and iterative attacks. One-time attacks can interact only once with the model. Iterative attacks can interact many times with the model, performing better, however with a bigger cost of computational power. [17, p. 4] These aspects de-

fined are a great tool when defining an adversarial example attack against a DNN because they help to further plan what type of an algorithm to choose and how to construct it.

Another important part of adversarial examples is the *perturbation*. Essentially, it means the alteration to the model input to make it an adversarial example. [14, p. 2] Perturbation can also be limited. While limiting the perturbation, it must be determined which is more important, minimize the perturbation to fool a human eye or have a more perturbated input which fools the model better [17, p. 5].

A common way to constrain or measure the perturbation of an adversarial image is with p-norm distance.

$$\|x - x_A\|_p = (\sum_{i=1}^{n} \|x_i - x_{Ai}\|^p)^{\frac{1}{p}} \quad (2.7)$$

Where x is the original input, x_A is adversarial input and p is defined by the Lp norm used. Norms mostly used are L0, L2 and L∞. Where L0 measures the numbers of modified pixels in the x_A, L2 measures the Euclidean distance of x and x_A, and L∞ measures the maximum change in any pixel of the x. Carlini & Wagner [4, p. 3–4] and Sharif & al [11, p. 2–3] In short, the concept of perturbation limitation is needed to define what type of modifications are allowed for the input. For example, when L0 is limited to one pixel, it means the adversarial example must be constructed by modifying only one pixel of the input image.

Adversarial examples can also be generated from scratch, with this method there exists no perturbation since there is no original input defined. Lp limitation enables one way to describe the actual robustness of a DNN by comparing the amount of Lp limitation against the effectiveness of the adversarial example.

2.2.2 Different ways to generate adversarial examples

Now that the concept of adversarial examples is familiar, this chapter introduces some of the common methods to generate those examples. It is noteworthy that there is a substantial number of different algorithms and styles to generate adversarial examples.

The first method introduced is *Fast gradient sign method* proposed by Goodfellow & al in [7]. It is a white box attack method that uses the signs of the gradients of a loss function respect to the input to modify the input data into an adversarial example. When this is stated as a formula we get $x_A = x + \varepsilon * sign(\nabla_x L(\theta, x, y))$, where x_A is the adversarial example, x is the original input, ε is the multiplier for the perturbation, θ is models parameters, and y is targets of the x. [7, p. 3] In short, this method adds perturbation to the input data which has the same direction that the gradient of the loss function has respect to the input data.

Another method called Genetic algorithm (GA) is also one great way to generate black-box adversarial examples. This method is discussed more thoroughly in chapter 3 since it has been chosen as the tool to use in this thesis implementation part. For more methods such as Carlini & Wagner attack, DeepFool, Zeroth Order Optimization and Feature Adversary see [4,5,17].

All the attacks should work at some level if there are no defensive measures in the targeted model, the attack method one would choose depends highly on the situation (threat model discussed in chapter 2.2.1). The attacks may seem different but the goal of them all is the same, they try to affect the models cost function in a way that the output layers activation functions yield desired confidences for desired classes.

2.2.3 Defence against adversarial examples

This chapter briefly introduces some defensive methods to enhance the robustness of DNNs. Defensive methods against adversarial examples have become an increasingly studied subject in the last years, and just as there are many different attack methods, there are defensive methods as well.

One defensive method against adversarial images is called *adversarial training* introduced by Huang & al in [8], which means that DNN is trained with adversarial images included in the training set. This teaches the model to ignore those types of adversarial examples. However, it seems to only defend well against adversarial images constructed with the algorithm it has also been trained on. [11, p. 3–4] If one would train the model with all different kinds of adversarial examples constructed with different algorithms, there would be too much fake data on the model, and it would become ineffective.

Another type of defence are *autoencoders* to DNN, they are in fact their own type of neural network which reduce the dimensions of the inputs and then expand them back to the original dimension size. While doing this, they should remove the perturbations before the inputs are fed to the other network. However, according to Gu & Rigazio [6, pp. 4–5], new adversarial examples can easily be constructed with a new model where an autoencoder is involved.

There are also number of different defensive methods such as network verification, reconstruction of input, network distillation, ensembling defences, etc. For more methods see [5,17]. Many of these defensive methods work only against adversarial examples generated in a certain way. Therefore, when choosing a defensive method, one should think about what kind of attacks could be introduced to the given model. There also exists python libraries for testing the robustness of the adversarial inputs, most popular ones seem to be CleverHans, IBM Adversarial Robustness Toolbox and Foolbox.

3. GENETIC ALGORITHMS TO HELP ROBUSTNESS TESTING

When it comes to analyzing the robustness of DNN and generating adversarial examples there are multiple ways to tackle the problem as discussed in the previous chapters. In their research, Nguyen & al [10] have used an approach called genetic algorithms. Inspired by their impressive results to “fool” a deep neural network with GA, it has been chosen as the method for this bachelor’s further analysis. This chapter introduces the basic principle behind the genetic algorithms and how they are used to generate adversarial inputs for a DNN.

3.1 Principle of genetic algorithms

A genetic algorithm can be described as a method to solve problems using similar principles as genetics, this also means there are similar concepts such as chromosome and gene involved. In this context, chromosome means a solution to the given problem e.g. an image, and gene is a building block of the chromosome, e.g. a pixel of an image. GAs have been widely used in many problems that include optimization or search.

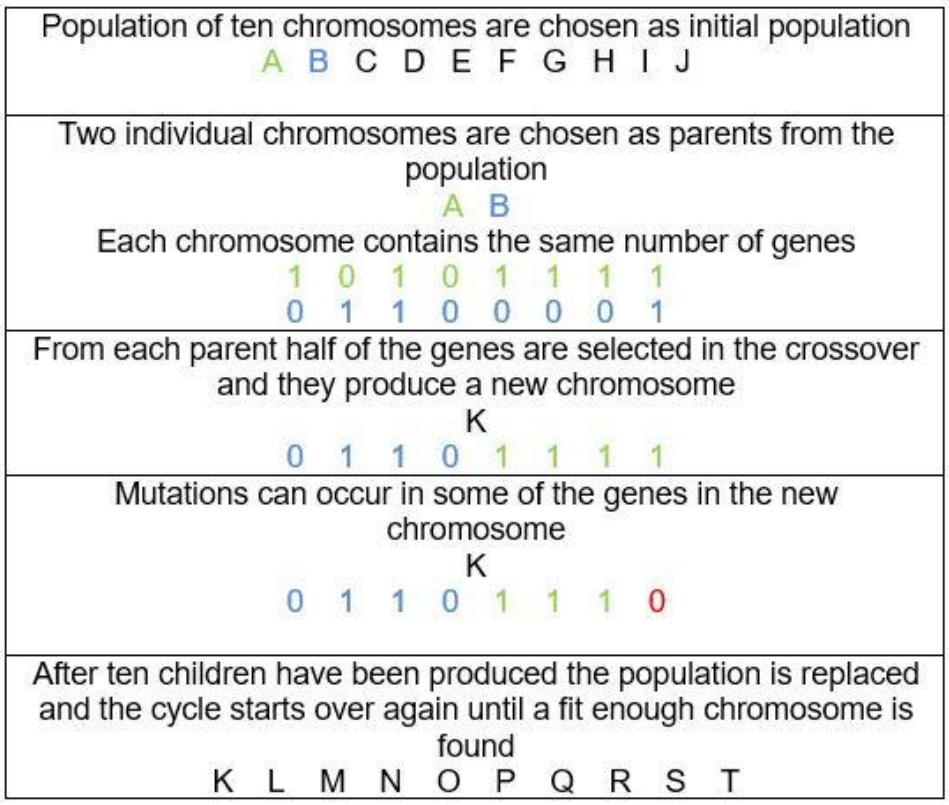


Figure 3.1 *Basic principle of genetic algorithm*

Figure 3.1 represents the basic principle of genetic algorithm. GA can be said to consist of four parts: initialization, selection, genetic operations and replacement. Each of them has an important role for the algorithm to be successful.

The first part of the initialization consists of defining the *search space*. The algorithm must have a search space, so it knows what type of solutions to look for and where to search them [13, p. 20].

The second part of the initialization is to define the *initial population*. The initial population consists of *chromosomes* which are just some solutions to the given problem. In their book, Sivanandam & Deepa [13, p. 30] tell that it is recommended for the initial population to have a good mixture of different types of chromosomes and they are often randomly generated. This allows the GA to have different types of solutions ready from the beginning.

Once the search space and initial population are defined, the next step is *selection*, meaning that some individuals from the population need to be selected as parents. For selection, a way to describe the fitness of each individual chromosome is needed, this is called a *fitness function*. A fitness function is always problem-specific, it gives a fit for each chromosome to the defined problem. Defining the fitness function is said to be the most difficult and important part when creating a GA. [13, p. 148] After the fitness function has evaluated the chromosomes, usually the ones with higher fit have a greater possibility to be selected as parents. There are multiple different methods to select the parents based on their fitnesses, see [13, pp. 46–50].

For GA to produce new solutions to the given problem, *genetic operations* must occur. The most important genetic operation is the *crossover*, which is applied by choosing half of the genes from two different parent chromosomes and combining those parts into a new child chromosome. These crossovers happen until a desired amount of child chromosomes have been produced. Usually, the number of children equals the size of the population [13, p. 6]. Without the crossover, the algorithm would just reproduce the same answers to the problem all over again and thus wouldn’t solve the problem. For a number of different crossover techniques see [13, pp. 51–56].

The second genetic operation is called *mutation* which happens in the child chromosomes. In the mutation, a small number of genes in child chromosomes will be changed. Mutations must occur to make sure that the variation in solutions is preserved over the

generations [13, p. 56]. In his book, Bäck [2, p. 197] tells that it is generally a good idea to keep the mutations rather small compared to the unmutated chromosome since bigger changes often seem to lower the fitness of the chromosome.

The last part of the GA is called *replacement*. Replacement means that the old population of chromosomes are replaced with the newly produced children. The replacement can be done in several ways, one option is to replace everyone in the old population. [13, pp. 58–59] Then there is a concept of *elitism*, which means that some of the best fitting chromosomes are preserved after the cycle despite them being parent chromosomes [13, p. 49]. This ensures that the fitness never goes downwards over the generation.

After one generation the process starts all over again with the newly generated population, these cycles or generations are continued until one of the chromosomes has a high enough fitness value, meaning a good enough solution has been found.

3.2 Adversarial examples for neural networks using genetic algorithms

According to Sivanandam & Deepa [13, p. 5], GAs have been used mostly on complex non-deterministic problems and machine-learning. This reflects nicely to the subject of generating adversarial examples for DNNs, which has all those features.

Taking the terms in use discussed in the chapter 2.2.1 following things can be said from GA as a tool to generate adversarial inputs. They can produce both false positive and false negative attacks and are used as black-box models. They must be used as iterative attacks since it is required by the iterative nature of the algorithm. The main drawback with genetic algorithms is that they require many queries to the model they are trying to produce the adversarial examples to.

As mentioned in chapter 2.1, DNN predicts the output probabilities from inputs with help of weights/convolutional filters, biases and activation functions. When creating an adversarial example with GA as well as with any adversarial example algorithm, the input must be perturbated in such a manner that it has an impact to the loss function resulting the output layer of the network to give some desired confidence.

GA tries to search for the input areas which affect the most to the loss function/output activations and once these areas are found, they are further evolved towards the best

fitting solution. The important thing to note is that now the fitness function of the GA is the whole DNN model, therefore the actual fitness of an individual is measured by the probabilities of the DNN predictions. There are many ways how a GA can be constructed to generate adversarial inputs to DNNs, but the threat model mentioned in chapter 2.2 ultimately determines what type of output it tries to generate. However, the basic principle of the GA described in chapter 3.1 still stays the same regardless of the threat model.

Genetic algorithm generated adversarial inputs should be hard to defend against, because of their random nature and the fact that the search space stays diverse because of the crossover and mutation. Every time an adversarial example is generated with a GA it is probably different from the next one. Following Figure 3.2 from Nguyen & al [10, p. 5] is a good example of the diversity of images GAs can produce.

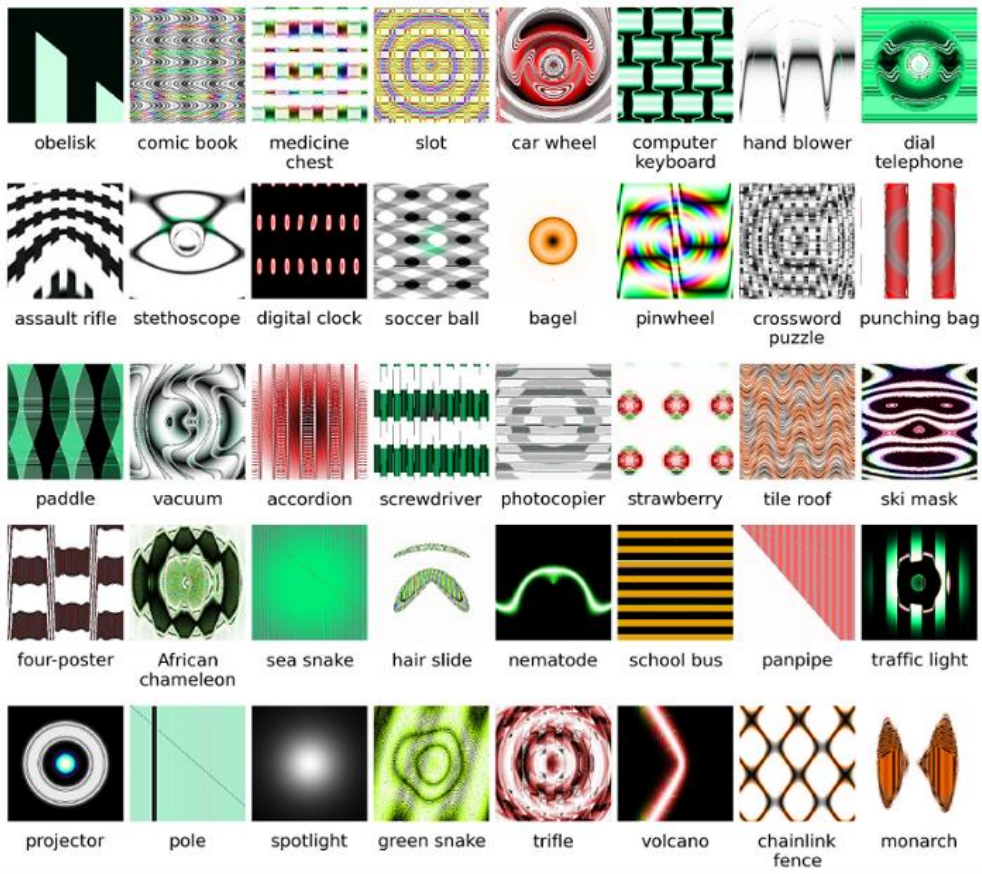


Figure 3.2 *Generated pictures showcasing the diversity GAs can produce, mean DNN confidence score of 99.12%.* [10, p. 5]

It can also be seen that the model confidence achieved by the GA is very high even though the initial population is generated randomly. This is a promising result towards

being able to implement a working GA that generates high confidence adversarial examples in this bachelor.

4. ROBUSTNESS ANALYSIS OF VGG16

This chapter is the actual implementation part of the thesis. First, an example network VGG16 is introduced, then a genetic algorithm is implemented with python to generate adversarial inputs to this network. Next, different kinds of adversarial based tests are made with this algorithm towards the example network. Finally, the results are analyzed to gain information about the robustness of the example network.

4.1 VGG16

For robustness analysis, an example neural network is required. A deep CNN model called VGG16 introduced by Simonyan & Zisserman in [12] has been chosen for this. VGG16 was chosen because of its good merits in ImageNet Large Scale Visual Recognition Challenge 2014 (ILSVRC-2014) achieving top-5 test error (%) of 7.3 with 1000 different classes [12, p. 8]. Top-5 test error is measured in such a way that the target label must be in one of the top 5 predictions.

As can be seen from Figure 4.1, VGG16 is designed to take 224x224x3 RGB pictures as it's input. It consists of five blocks each having a couple of convolutional layers followed by a max pooling layer. The size of a single convolutional kernel is 3x3. This was the main difference between VGG16 and other CNNs at the time of ILSVRC-2014 since before this the kernels had been significantly larger [12, p. 2]. The max pooling layers have a window size of 2x2 with a stride of 2. The filter amount starts from 64 and is doubled after every max pooling layer. After the fifth block, there is a flattening layer, which flattens the feature maps into a 1-dimensional vector. In the end-part of the net, there are three dense layers, first two having 4096 neurons each and the last one 1000. This means the VGG16 gives a probability for 1000 different classes. The activation functions of the hidden layers are all ReLU (2.1) and the output layer is softmax (2.3). The cost function used in training is cross-entropy (2.5).

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000

Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0

Figure 4.1 *Layer types, output shapes and parameters of the VGG16*

4.2 Implementation of the algorithm

The tests were made with Python3 Anaconda distribution, which contains all the needed tools to work with neural networks and manipulate the data. The editor used was Spyder and computer specs were: Intel i5 6600k, Nvidia gtx980 and 16gb of ram. Python libraries used can be seen in *Program 1.*

Rather than using some predetermined library for robustness testing, the idea was to generate a GA from scratch. Also, there actually wasn't an existing implementation of GA in the most common robustness testing libraries. The goal of the GA was to generate adversarial examples to the VGG16. Data was represented as 224x224x3 RGB-pictures and the initial population was defined by either generating or downloading a set of images that size. VGG16 model and its predictions worked as the fitness function and the objective was to evolve the pictures in such manner that the network would be "fooled" to classify the picture differently than humans.

4.2.1 Creating the genetic algorithm

First part of the algorithm was to import the VGG16 to python with Keras library and save it as a model, no further modifications to the network was needed since by default the network is pre-trained with ImageNet weights and it takes 224x224x3 RGB-images with extra dimension as an input. After the VGG16 was saved as a model it looked just like Figure 4.1 when shown on the Spyder interface.

The implementation of the actual genetic algorithm was started by defining the initial population, this was done by creating a function "generate_initial_population", which downloads a desired number of images into an array from a given path and returns the array as the initial population. See *Program 2.* Depending on the desired result, these pictures can be either randomly generated or real images downloaded from some path. The number of images, path and randomness are given as parameters.

After the initial population function, a fitness function was needed to tell the fitness of the population. A function called "fitness" was created to achieve this. See *Program 3*. The function goes through the given population (224x224x3) RGB-images and reshapes the images into a format that VGG16 can read by adding an extra dimension into the images resulting (1x224x224x3) images. Then, the images are fed to the VGG16 model which predicts the probability of the images to belong to each class. I.e. It works as the real

fitness function. Probability of each image to belong in predetermined ImageNet class is saved as fitness of that image. After everyone in the population is evaluated, the fitness's of them are returned as numpy array.

After evaluating the fitness of the population, the next step was to choose the fittest amongst the population. This was achieved by a function "select_fittest", which searches the desired amount of minimum or maximum fitness's from the fitness's array and returns their indices. See *Program 4.* The minimum or maximum choice is given as parameter and depends on whether one needs to minimize or maximize the ImageNet class fitness.

Then, the next step was to create a "crossover" function to produce children for the fittest population. See *Program 7*. First, the function modifies the fittest population images into 1d-form, by in order, taking each of the colour channels values from left to right, top to bottom, resulting in a one-dimensional array. This modification is done with a helper function called "image_to_chromosome", this helper function can also modify 1d array back to an image form. *See Program 5.* Next, the crossover function selects a predefined number of parent pairs amongst all the possible permutations of two parents from the fittest population. After all the parent pairs are selected, half of each parent pairs genes are concatenated randomly resulting in a new child chromosome from each of the pairs.

To achieve efficient results, mutations must occur in the newly created child chromosomes. This was achieved by "mutation" function. See *Program 6*. It takes the newly created children and inflects mutation to every child by modifying a given factor amount of their genome by a random value from zero to ten to a random direction. If mutation limit (L^∞ limitation) is given as parameter, the mutations will stay in the range of that limitation.

After the needed functions were defined, they could be called from the main program. See *Program 8.* One key thing to notice is that the algorithm uses elitism to always carry the defined number of fittest individuals over to the next generation.

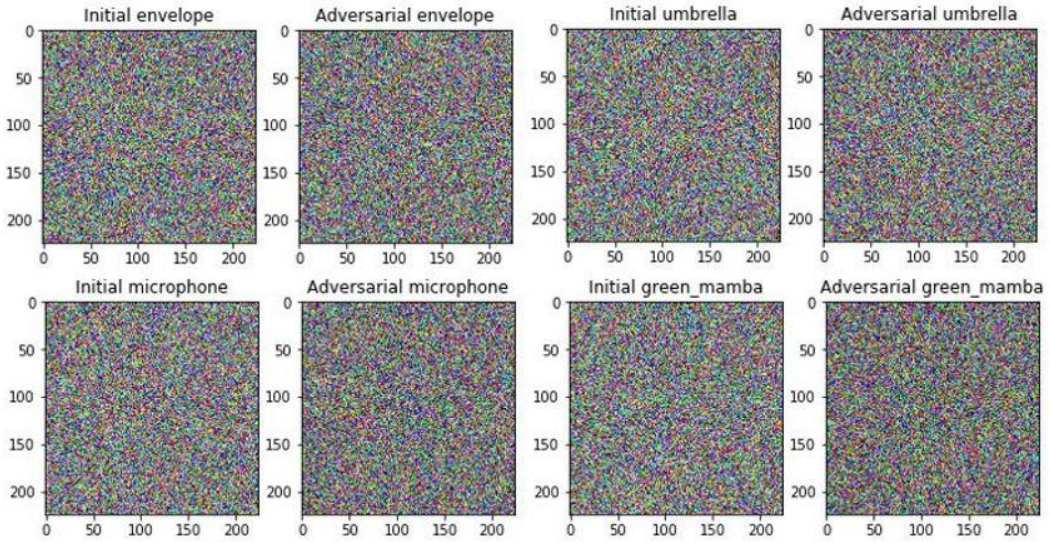
4.3 Test results

In the first test, the idea was to generate the initial population images from random pixels and try to get a high fit for a predetermined ImageNet class. The parameters were set to produce a false positive result and the iterations to stop after the probability of 0.99 is achieved. Parameters used in the test can be seen from Table 1 in the section "Test1".

Table 1 *Parameters used in the tests*

Parameter name	Test1	Test2	Test3
false_positive	True	False	True
imagenet_object	See Figure 4.2	“sea_lion”	“chimpanzee”
generation_limit	5000	5000	10000
population_amount	25	25	25
parent_amount	10	10	10
elite_amount	4	4	4
children_amount	21	21	21
mutation_factor	0.02	0.02	0.02
perturbation_limit	“none”	5	5
randomImages	True	False	False
stopprobability	0.99	0.01	0.95

Initial and resulting adversarial images for a couple of different classes can be seen from Figure 4.2. It also includes the generation amounts and initial probabilities.



ImageNet class	Initial accuracy to ImageNet Class	Generations until 0.99 probability
envelope	0.10818934	297
umbrella	0.01499798	394
microphone	0.00038530072	1056
green_mamba	6.4896885e-05	1674

Figure 4.2 *Randomly generated initial images and evolved adversarial images for different ImageNet classes including their initial accuracies and generation amounts*

As could be expected from a random input, these images don't really resemble anything. Note that the generation amount needed changes drastically if the initial fit is different. The first test was successful enough to confirm that the algorithm works and gives interesting high-confidence results.

In the second test, the purpose was to generate false negative adversarial example (Low accuracy for positive class). The perturbation limitation used was L^{∞} and it was limited to 5, meaning each pixel's colour channel value could change only by a maximum of ±5. This limitation should result in an adversarial picture that seems basically unmodified to a human eye compared to the initial image. For the initial image, a sea lion was chosen. The vgg16 gives this example sea lion a probability of 0.893 for belonging into the class "sea_lion". The parameters were set to produce a false negative result and the iterations to stop after the probability of 0.01 was reached for the adversarial image to belong in a "sea_lion" class. Parameters used in the second test can be seen from Table 1 in the section "Test2".

The script stopped after 672 generations when the "sea_lion" class probability had dropped to 0.01 and the highest accuracy was for class "toilet_tissue" which was roughly 0.57. It is noteworthy that after 100 and 300 generations the probabilities for "sea_lion" class were already at 0.28 and 0.05.

Figure 4.3 *Initial sea-lion with a probability of 0.893 and adversarial image with a probability of 0.01 for belonging into ImageNet class "sea_lion". Initial image from [18].*

The same test with same parameters was also performed with different images, most notable one was the class "hatchet", which had a quite high 0.999 initial probability to

belong into an ImageNet “hatchet” class. The generations needed for probability to drop under 0.01 was roughly 1000.

The final and the hardest test for the algorithm was about creating false positive adversarial images to a predetermined class. I.e. Try to maximize the confidence into some predetermined class other than the initial maximum class. The hardness comes from the fact that now it was needed to maximize just one other chosen class which had quite low initial probability while minimizing the probability of all the other classes. The L∞ limit was 5 which was same as in the third test. Basically, this test was the same as the first one, but with initial images not being random and having the L∞ limitation. Image of a lion was chosen for this test, for which VGG16 gave an accuracy of 0.75 to belong to ImageNet class “lion”. The object class was “chimpanzee” and the other parameters for the test can be found from Table 1 in the section “Test3”.

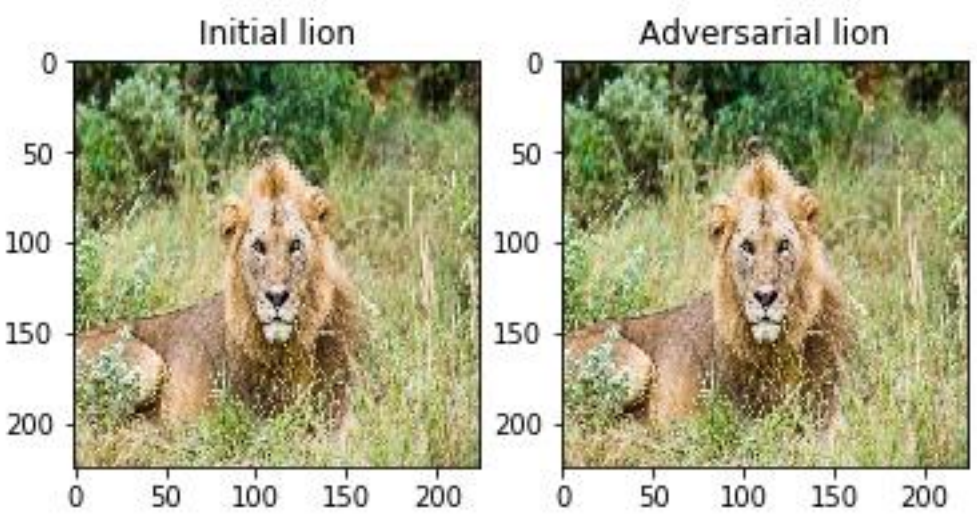


Figure 4.4 *Initial lion with a probability of 0.75 for belonging into ImageNet class “lion” and adversarial lion with a probability of 0.95 for belonging into ImageNet class “chimpanzee” Initial image from [19]*

This time the script took 6000 generations to reach the desired accuracy of 0.95 for “chimpanzee class”, which is quite high as expected. Second highest accuracy of 0.013 was for the class “siamang”.

4.4 Analysis of the results

The first test was about proofing that the algorithm works and can produce random adversarial images from scratch. This was proven right since the algorithm was able to

generate random high confidence false positive images. Creating a false positive image, in this case, is a much harder task than false negative, because one must maximize certain predetermined class instead of minimizing just one. Now, because the initial population of the GA was generated randomly, there is no concept of perturbation involved, thus it cannot be restricted. However, it can be said that the robustness of VGG16 against GA generated false positive images with a random initial population is not very high, because the generated images eventually achieved high confidence. Although, from Figure 4.2 can be seen that with very low initial accuracy the generations needed are quite high. The parameters could be better optimized to achieve smaller generation number needed. The main factor to reduce the number of generations is the size of the initial population which was just 25 in this test. On the other hand, increasing the initial population increases the computational cost.

The second test was the false negative test with L^∞ norm of 5. During the testing, it was notable that increasing the restriction of L^∞ norm would lower the generations needed drastically. Another factor was the initial accuracy of the object. For objects that had initial accuracy of roughly 1, it took a lot more generations than for the sea lion which had an initial accuracy of 0.893. Overall, the false negative test was a success and it can be said that against false negative adversarial examples the VGG16 is quite vulnerable without any defences, even when the L^∞ limit is as low as 5.

The final test was the hardest one for the algorithm, maximize predetermined class while having an L^∞ limit of 5. Even though it took 6000 generations to achieve the accuracy of 0.95, the algorithm still converged and didn't stop at some probability. Again, the resulting image seen in Figure 4.4 looks unmodified to the human eye and increasing the L^∞ limit and population size reduced the needed iterations. Especially in this test the increase in L^∞ was very effective in decreasing the generation amount.

All the tests made clear that against VGG16 it takes much higher generation amount from GA to achieve the desired accuracy for certain class if the initial accuracy of that class is very low. However, the GA is still able to produce a high confidence adversarial example no matter the initial accuracy. The results of second and third tests are interesting in a sense that GA could evolve working adversarial examples even with low L^∞ limit, which resulted in images that had small enough perturbation to not to be noticed by the human eye. With these results can be said that an image classifying DNN is not very robust against adversarial examples without any defensive methods, even though the iteration amount needed is sometimes rather high.

5. CONCLUSION AND FURTHER WORK

In this thesis the main goal was to gain insight about the robustness of image classifying DNNs, this was achieved with the help of theory and adversarial examples generated by the GA. From the test results, it can be said that the example network VGG16 is relatively easy to fool with genetic algorithm generated adversarial examples which are L_∞ limited. Especially false positive adversarial examples seem to be easy to generate. Thus, can be said that a high confidence image classifying DNN without any defences is not very robust against adversarial examples. To enhance the robustness of a DNN against adversarial examples one would need to use some defensive methods discussed in 2.2.3. The need for measuring and enhancing the robustness of neural networks is a crucial subject of neural networks research. If one can easily introduce a working adversarial example into a DNN, the network cannot be used in a place where safety is an issue.

If considering the actual GA implemented, the main drawback of it was the iteration amount needed, especially with small L_∞ limitation and low initial accuracy to the target class. The algorithm could be better optimized, moreover the crossover and mutation part. However, the iteration amount could also be lowered just by increasing the size of the population, which would also increase the computational cost. Considering the time in hand and the fact that this is a bachelor thesis, the current solution was sufficient.

This thesis has many aspects which could be taken further. First, some actual descriptive robustness metric system could be taken into use e.g. CLEVER briefly mentioned in chapter 2.2, instead of relying just on the insight gained from GAs effectiveness. Second, it could be analyzed how different structures of networks affect to their robustness and compare their robustness against each other. Third, different methods than GA could be used for generating adversarial examples (some of them mentioned in 2.2.2) to find which types of attacks work best against certain types of neural networks. Finally, the most important part. Different types of defensive methods could be tested and evaluated against adversarial examples generated in different styles, and ultimately try to find some new way which could boost the robustness against adversarial examples.

BIBLIOGRAPHY

[1] H. Abdi, D. Valentin & B. Edelman, Quantitative Applications in the Social Sciences: Neural networks. Thousand Oaks, CA: SAGE Publications, 1999, pp. 1–2, 74–82, Available from: http://methods.sagepub.com/book/neural-networks [cited 20.12.2019].

[2] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, 1996, p. 197.

[3] J. Cao, Z. Su, L. Yu, D. Chang, X. Li and Z. Ma, Softmax Cross Entropy Loss with Unbiased Decision Boundary for Image Classification, Chinese Automation Congress (CAC), 2018, pp. 2028-2032, DOI 10.1109/CAC.2018.8623242, [cited 20.12.2019].

[4] N. Carlini, D. Wagner, Towards Evaluating the Robustness of Neural Networks, Cornell University, 2017, pp. 4–5, Available from: https://arxiv.org/abs/1608.04644 [cited 20.12.2019].

[5] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, D. Mukhopadhyay, Adversarial Attacks and Defences: A Survey, Cornell University, 2018, Available from: https://arxiv.org/abs/1810.00069 [cited 20.12.2019].

[6] S. Gu, L. Rigazio, Towards deep neural network architectures robust to adversarial examples, Cornell University, 2015, pp. 1–9, Available from: https://arxiv.org/abs/1412.5068 [cited 20.12.2019].

[7] I.J. Goodfellow, J. Shlens, Christian Szegedy, Explaining and Harnessing Adversarial Examples, Cornell University, 2015, Available from https://arxiv.org/abs/1412.6572 [cited 20.12.2019].

[8] R. Huang, B. Xu, D. Schuurmans, C. Szepesvári, Learning with a strong adversary, Cornell university, 2015, Available from: https://arxiv.org/abs/1511.03034 [cited 20.12.2019].

[9] C. Nwankpa, W. Ijomah, A. Gachagan & Stephen Marshall, Activation Functions: Comparison of trends in Practice and Research for Deep Learning, Cornell University, 2018, p. 5–9. Available from: https://arxiv.org/abs/1811.03378 [cited 20.12.2019].

[10] A. Nguyen, J. Yosinski, J. Clune, Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In Computer Vision and Pattern Recognition (CVPR ’15), IEEE, 2015, Available from: https://arxiv.org/abs/1412.1897, [cited 20.12.2019].

[11] M. Sharif, L. Bauer, M. K. Reiter, On the Suitability of Lp-norms for Creating and Preventing Adversarial Examples, Cornell University, 2018, p. 2–4, Available from: https://arxiv.org/abs/1802.09653, [cited 20.12.2019].

[12] K. Simonyan & A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, ICLR, 2015, Available from: https://arxiv.org/abs/1409.1556, [cited 20.12.2019].

[13] S.N. Sivanandam and S.N. Deepa, Introduction to genetic algorithms, Springer, 2007, pp. 5–6, 20, 30, 46–56, 58–59, 148

[14] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, Intriguing properties of neural networks, International Conference on Learning Representations, 2014, Available from: https://arxiv.org/abs/1312.6199 [Cited 20.12.2019].

[15] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, L. Wang, G. Wang, J. Cai and T. Chen, Recent advances in convolutional neural networks, v6, Cornell University, 2017, pp. 354–377.

[16] T. Weng, H. Zhang, P. Chen, J. Yi, D. Su, Y. Gao, C. Hsieh, L. Daniel, Evaluating the Robustness of Neural Networks: An Extreme Value Theory Approach, ICLR, 2018, Available from: https://arxiv.org/abs/1801.10578 [cited 20.12.2019].

[17] X. Yuan, P. He, Q. Zhu and X. Li, Adversarial Examples: Attacks and Defenses for Deep Learning, IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 9, 2019, pp. 2805–2824. Available from: https://arxiv.org/abs/1712.07107 [Cited 20.12.2019].

[18] S. Seitamaa, Black sea lion, Unsplash, https://unsplash.com/photos/BqX2laVn2cQ [Cited 20.12.2019].

[19] F. Berger, Lion lying on a grass, Unsplash, https://unsplash.com/photos/PYBiF96syK8 [Cited 20.12.2019].

APPENDIX A: CODE SNIPPETS

```
1.  import warnings
2.  warnings.filterwarnings("ignore")
3.  from keras.applications.vgg16 import decode_predictions
4.  from keras.applications.vgg16 import VGG16
5.  from copy import deepcopy
6.  import itertools
7.  import random as stdlib_random
8.  from PIL import Image
9.  import numpy as np
10. import matplotlib.pyplot as plt
11. from keras.preprocessing.image import load_img
12. from keras.preprocessing.image import img_to_array
13. import time
```

Program 1 *Imports of required python libraries*

```
1.  def generate_initial_population(amount, path, random=True):
2.      """
3.      Generates "amount" number of initial population (224x224x3)RGB array.
4.      If "random"==True generates random images
5.      If "random"==False downloads the image from "path"
6.      returns the population as numpy array
7.      """
8.
9.      population = []
10.     for i in range(0,amount):
11.         if random:
12.             img = (np.random.standard_normal([224, 224, 3]) * 255).astype(
13.                    np.uint8)
14.         else:
15.             img = load_img(path, target_size=(224, 224))
16.             img = img_to_array(img)
17.         population.append(img)
18.     return np.asarray(population)
```

Program 2 *"generate_initial_population" -function*

```
1.  def fitness(population, model, class_name):
2.      """
3.      Uses the neural network classifier "model" to predict the
4.      fitness of each individual ( (224x224x3)RGB array ) in
5.      "population" to belong in the class "class_name"
6.      returns the fitnesses as numpy array
7.      """
8.
9.      labels = [decode_predictions(model.predict(member.reshape(
10.                 (1,224,224,3))),1000) for member in population]
11.
12.     fitnesses = np.asarray([obj[2] for label in labels for obj in label[0]
13.                 if obj[1]==class_name ])
14.     return fitnesses
```

Program 3 *"fitness" -function*

```
def select_fittest(fitnesses, amount, maximum=True):
    """
    If "maximum"==True, finds the "amount" number of maximum fitnesses
    from "fitnesses" array.
    If "maximum"==False, finds the minimum fitnesses
    returns the indexes of the found fitnesses as list.
    """

    # Takes a deepcopy of fitnesses so the original values won't be altered
    tempfit = deepcopy(fitnesses)
    elite_indexes = []
    for i in range(0,amount): # Loops for the amount of elites desired
        if maximum == True:
            index = np.argmax(tempfit) # Finds the maximum fitness
            #Replaces the found max value as zero so the next iteration
            #gives second highest value etc..
            tempfit[index] = 0
        else:
            index = np.argmin(tempfit)
            # Replaces the found min value as two so the next iteration
            # gives second lowest value etc..
            tempfit[index] = 2
        elite_indexes.append(index)

    return elite_indexes
```

Program 4 *"select_fittest" -function*

```
def image_to_chromosome(images, reverse=False):
    """
    Reshapes all the individuals from array "images"
    if "reverse"==True reshapes 1d array to (224,224,3)
    if "reverse"==False reshapes (224,224,3) array to 1d
    returns the reshaped individuals as list of numpy arrays
    """

    if reverse: # If reverse is set to True makes images from chromosomes
        chromosomes=[image.reshape(224,224,3) for image in images]
    else: # If reverse is set to False makes chromosomes from images
        chromosomes=[image.reshape(-1) for image in images]
    return chromosomes
```

Program 5 *"image_to_chromosome -function"*

```
def mutation(chromosomes, mutation_factor, limit):
    """
    Mutates the 1d numpy array "chromosomes" by changing the
    "mutation_factor" determined amount of genes from each chromosome.
    If "limit"==0, changes the genes value by 10 to a random direction
    If "limit" > 0, changes the genes value by limit to a random direction
    """

    if limit==0:
        for chromosome in chromosomes:
            indecies = [np.random.randint(0, chromosome.size)
                for p in range(0, int(mutation_factor*chromosome.size))]

            for index in indecies:
                value = chromosome[index]
                if value > 9 and value < 246:
```

```
                    color = range(int(value)-10, int(value)+10)
                elif value < 10:
                    color = range(int(value), int(value)+10)
                else:
                    color = range(int(value)-10, int(value))
                chromosome[index] = stdlib_random.choice(color)
    else:
        for chromosome in chromosomes:
            indecies = [np.random.randint(0, chromosome.size)
                for p in range(0, int(mutation_factor*chromosome.size))]

            for index in indecies:
                initvalue = initchromosome[index]
                low=0
                high=0
                if initvalue-limit<0:
                    low=0
                else:
                    low=initvalue-limit
                if initvalue+limit > 255:
                    high = 255
                else:
                    high = initvalue+limit
                color = range(int(low),int(high))
                chromosome[index] = stdlib_random.choice(color)
```

Program 6 *"mutation" -function*

```
def crossover(parent_candidates, how_many_childs):
    """
    Chooses "children_amount" of parent pairs amongst "parent_candidates"
    and for every parent pair produces a new child
    by randomly combining genes from both parents.
    returns the newly produced children as list of numpy arrays
    """

    # Modify each image in parent_candidates to 1d array
    parent_candidates = image_to_chromosome(parent_candidates)
    # Chooses k number of r=2 permutation from 1d elites as parents
    selected_parents = stdlib_random.choices(list(itertools.permutations(
                        parent_candidates, r=2)), k=how_many_childs)
    # Loops through each of the parent pairs
    # takes half of genes from each parent and concatenates them
    # resulting an array of new children
    arraylen = int(parent_candidates[0].shape[0])
    n = int(arraylen/2)
    children = []
    temparr = np.zeros(arraylen,int)
    for parents in selected_parents:
        arrayman= deepcopy(temparr)
        indecies1 = np.random.choice(parents[0].shape[0], n, replace=False)
        indecies2 = np.setxor1d(np.indices(parents[1].shape), indecies1)
        arrayman[indecies1] = parents[0][indecies1]
        arrayman[indecies2] = parents[1][indecies2]
        children.append(arrayman)
    return children
```

Program 7 *"crossover" -function*

```
if __name__ == "__main__":
    start_time = time.time()    # start time for the script
    model = VGG16()             # neural network model used

    # if true -> false positive, if False -> false negative
    false_positive = True
    # imagenet object, whose fitness we are interested
    imagenet_object = "some_imagenet_class"
    generation_amount = 10000
    population_amount = 25
    parent_amount = 10
    # if other than 0, uses elitism
    elite_amount = 4
    children_amount = population_amount-elite_amount
    # Factor that determines how many genes from children are mutated.
    mutation_factor = 0.02
     # Amount of how much each gene is allowed to change
    perturbation_limit = 5
     # True -> generate random images. False -> download image from path
    randomImages = False
    # desired probability of the adversarial image to the imagenet_object
    stopprobability = 0.95
    # path to the initial image, only used if randomImages = False
    imagepath = "some/path"


    population = generate_initial_population(population_amount,
                                             imagepath, randomImages)

    stop=False # Loop stopper
    for i in range(0,generation_amount):
        fitnesses = fitness(population, model, imagenet_object)
        fittest_indexes = select_fittest(fitnesses,
                                         parent_amount, false_positive)
        fittest_population = population[fittest_indexes]

        # save the best fitting initial image, its fitness and 1d form
        if i==0:
            initpic = population[0].astype(np.uint8)
            initchromosome = image_to_chromosome(population)[0]
            initfitness=str(fitnesses[fittest_indexes[0]])
            print("Fitness of the initial image: {}".format(initfitness))
            labels = decode_predictions(model.predict(initpic.reshape(
                    (1,224,224,3))),3)
            print("\nTop 3 initial accuracies:\n   {} \n   {} \n   {} \n"
          .format(': '.join(map(str, labels[0][0][1:3])),
          ': '.join(map(str, labels[0][1][1:3])),
          ': '.join(map(str, labels[0][2][1:3]))))

        # make children with the crossover function
        children = crossover(fittest_population,children_amount)
        # Mutates the newly created children
        mutation(children,mutation_factor,perturbation_limit)
        # Children back to image shape
        children = image_to_chromosome(children,True)
        # replace the old population with children and elites
        population = np.concatenate((fittest_population[0:elite_amount],
                                     children), axis=0)
        # false positive case
        if false_positive and fitnesses[fittest_indexes[0]
                                        ] > stopprobability:
            stop=True
        # false negative case
        if not false_positive and fitnesses[fittest_indexes[0]
                                            ] < stopprobability:
            stop=True
```

```
67.
68.         # Plot the situation after every 100 generations
69.         # and when the desired probability is achieved
70.         if (i+1)%100==0 or stop:
71.             print("\n\n{} generations have passed in {} seconds.".format(
72.                     i+1,time.time() - start_time))
73.             fig = plt.figure()
74.             ax1 = fig.add_subplot(1,2,1)
75.             plt.imshow(initpic)
76.             ax2 = fig.add_subplot(1,2,2)
77.             plt.imshow(fittest_population[0].astype(np.uint8))
78.             ax1.title.set_text('Initial lion')
79.             ax2.title.set_text('Adversarial lion')
80.             plt.show(block=True)
81.             print("Fitness to a '{}' class for the best generated image: "
82.             .format(imagenet_object) + str(fitnesses[fittest_indexes[0]]))
83.             # break the loop when desired probability achieved
84.             if stop:
85.                 break
86.
87.     # Print the three best initial accuracies
88.     labels = decode_predictions(model.predict(initpic.reshape(
89.             (1,224,224,3))),1000)
90.     print("\nTop 3 initial accuracies:\n   {} \n   {} \n   {} \n"
91.           .format(': '.join(map(str, labels[0][0][1:3])),
92.           ': '.join(map(str, labels[0][1][1:3])),
93.           ': '.join(map(str, labels[0][2][1:3]))))
94.
95.     # Print the three best adversarial accuracies
96.     labels = decode_predictions(model.predict(
97.             fittest_population[0].reshape((1,224,224,3))),3)
98.     print("\nTop 3 adversarial accuracies:\n   {} \n   {} \n   {} \n"
99.           .format(': '.join(map(str, labels[0][0][1:3])),
100.                ': '.join(map(str, labels[0][1][1:3])),
101.                ': '.join(map(str, labels[0][2][1:3]))))
102.
103.
104.         # save the adversarial image
105.         best_answer = fittest_population[0].astype(np.uint8)
106.         im = Image.fromarray(best_answer)
107.         im.save("fooling_{}.jpg".format(imagenet_object))
```

Program 8 *The main program*