DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Adversarial Attacks on Gaussian Processes

Tim Waegemans

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Adversarial Attacks on Gaussian Processes

Adversariale Angriffe auf Gauß Prozesse

Author:	Tim Waegemans
Supervisor:	Prof. Dr. Christian B. Mendl
Advisor:	Dr. Felix Dietrich
Submission Date:	15.05.2021

I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 15.05.2021 Tim Waegemans

Abstract

Adversarial attacks can pose a security risk for any applied machine learning method. A lot of research has focused on attacking deep neural networks and also making them robust against such attacks. In this thesis we evaluate attacks on Gaussian processes. More specifically Gaussian processes that can be derived from a convolutional neural networks (CNN) in the limiting case of infinitely many channels. We apply the Fast Gradient Sign Method attack to this Gaussian process trained on the MNIST dataset. Our results indicate that this Gaussian process is more robust against this adversarial attack when compared to a regular CNN with a finite number of channels. However, the complexity of the Gaussian process is significantly worse.

Kurzfassung

Adversariale Angriffe können ein Sicherheitsrisiko für Anwendungen, welche Machine Learning benutzen, darstellen. Es wurde bereits einiges an Forschung betrieben, welche Angriffe auf tiefe neuronale Netze und Maßnahmen dagegen untersucht. In dieser Arbeit untersuchen wir Angriffe auf Gauß Prozessen, welche von konvolutionellen neuronalen Netzen (CNN) im Grenzfall von unendlich vielen Kanälen abgeleitet werden können. Wir wenden den Fast Gradient Sign Method Angriff auf solche Gauß Prozesse an, welche mit dem MNIST Datensatz trainiert wurden. Unsere Ergebnisse deuten darauf hin, dass die Gauß Prozesse robuster gegen einen adversarialen Angriff sind als gewöhnliche CNNs mit endlich vielen Kanälen. Allerdings ist die Komplexität der Gauß Prozesse deutlich schlechter.

Contents

1 Introduction

Machine learning has become more and more prominent in recent years. Many different methods have been published to deal with various types of data, for example convolutional neural networks for images and videos or transformers networks for text translation. Even in the recent COVID-19 pandemic machine learning can be applied. Researchers developed a machine learning model to predict if a person was infected or not by examining certain symptoms, age, sex and if there was contact with a positively tested person [1]. Machine learning models like this can be very useful for society.

Nevertheless, is important to know the limitations of the models we use. There are several questions we need to consider when using machine learning in real world applications. How can we ensure that a model gives us reasonable outputs? What happens when the input does not look like we expected it to? When does the model fail? Answering these questions is not easy. Especially for methods like deep neural networks, which have so many parameters that it becomes difficult to interpret what is happening in a model. Often methods are evaluated empirically by measuring their performance on certain datasets. However, since any training and testing dataset will most likely not encompass all possible inputs, there are regions where the model will probably produce incorrect outputs.

An adversarial attack will try to make use of the generalization problem, by constructing inputs maliciously to make the model fail. Deep Neural Networks have been shown to be vulnerable towards adversarial attacks. Small changes like changing a single pixel [2] or adding noise that is visually hard to spot [3] can lead to a completely different classification. This is a problem for critical applications that need robust models. An example of what such an attack does is shown in figure 1.1. A network can be fooled by adding some minor modification that looks like noise and is hard to perceive. The network is caused to misclassify an image of a panda as a gibbon with high confidence.

There are approaches to extend neural networks with uncertainty estimates like Monte Carlo dropout [4]. The hope is that regions in which the model does not generalize well on are identified and can be rejected.

Gaussian processes are a group of machine learning methods that provide a uncertainty estimate without any additional evaluations. Furthermore, Gaussian Process can be derived from an infinitely wide neural network with random weights [5]. This can even be extended to deep and convolutional neural networks [6][7]. In this thesis we will evaluate how robust against adversarial attacks such a Gaussian process derived from a convolutional neural network is.

In section 2 we will preset relevant work. It consists of two blocks. Firstly, we will present Gaussian processes, including their connection to neural network. Secondly, we will discuss adversarial attacks and introduce some specific ones.

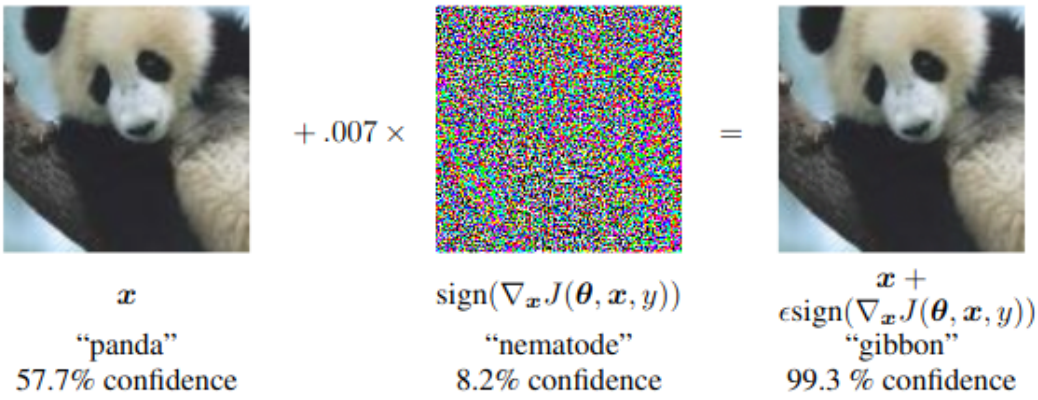


Figure 1.1: Example of an adversarial attack (image from [3])

The main part of out contribution is found in section 3. Firstly we will discuss how we apply an adversarial attack on Gaussian process. Then we will present the Gaussian process we used as well as some reference methods for comparison. We will discuss how to evaluate adversarial robustness, before evaluating the robustness of the Gaussian process and reference methods on the MNIST dataset.

In the final section 4 we will summarize our findings and present interesting topics for future work.

2 Related Work

As this thesis will focus on applying adversarial attacks to a Gaussian process we will introduce the necessary related work on these topic in this sections. We will start of with the Gaussian processes, their use supervised machine learning and their connection to neural networks. Finally we will present and discuss different adversarial attacks.

2.1 Gaussian Processes

A Gaussian process is a collection of random variables such that any finite subset follows a normal distribution [8]. The random variables are continuously indexed in a given domain. This domain often is time or space. We denote a Gaussian process as

$$f(\mathrm{x}) \sim \mathcal{GP}(m(\mathrm{x}), k(\mathrm{x}, \mathrm{x}')). \tag{2.1}$$

Where $m(\mathrm{x})$ denotes the mean function and $k(\mathrm{x}, \mathrm{x}')$ the covariance between two random variables. We will refer to $k(\mathrm{x}, \mathrm{x}')$ also as kernel or kernel function.

$$\begin{aligned} m(\mathrm{x}) &= \mathbb{E}\left[f(\mathrm{x})\right] \\ k(\mathrm{x}, \mathrm{x}') &= \mathbb{E}\left[\left(f(\mathrm{x}) - m(\mathrm{x})\right)\left(f(\mathrm{x}') - m(\mathrm{x}')\right)\right] \end{aligned} \tag{2.2}$$

For a finite subset of points in the domain $\{\mathrm{x}_1, ..., \mathrm{x}_\mathrm{n}\} \subset \mathcal{D}$ the distribution of random variables at these points follows the normal distribution with

$$\begin{bmatrix} f(\mathrm{x}_1) \\ \vdots \\ f(\mathrm{x}_\mathrm{n}) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(\mathrm{x}_1) \\ \vdots \\ m(\mathrm{x}_\mathrm{n}) \end{bmatrix}, \begin{bmatrix} k(\mathrm{x}_1, \mathrm{x}_1) & \dots & k(\mathrm{x}_1, \mathrm{x}_\mathrm{n}) \\ \vdots & \ddots & \vdots \\ k(\mathrm{x}_\mathrm{n}, \mathrm{x}_1) & \dots & k(\mathrm{x}_\mathrm{n}, \mathrm{x}_\mathrm{n}) \end{bmatrix}\right). \tag{2.3}$$

For many applications the mean function is fixed to zero. This allows us to express the Gaussian Process fully by only describing the covariance function k. We will use the following notation in this thesis to describe the covarinace matrix between to matrices.

$$\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}'} = K(\boldsymbol{X}, \boldsymbol{X}') = \begin{bmatrix} k(\mathrm{x}_1, \mathrm{x}'_1) & \dots & k(\mathrm{x}_1, \mathrm{x}'_\mathrm{m}) \\ \vdots & \ddots & \vdots \\ k(\mathrm{x}_\mathrm{n}, \mathrm{x}'_1) & \dots & k(\mathrm{x}_\mathrm{n}, \mathrm{x}'_\mathrm{m}) \end{bmatrix} \tag{2.4}$$

A Gaussian Process can be either derived from a specific model or explicitly be constructed from a covariance function. An example of a derived case is a Bayesian linear regression $f(\mathrm{x}) = \mathrm{x}^\top \mathrm{w}$ [8]. Where the weights follow a centered normal distribution $\mathrm{w} \sim \mathcal{N}(0, \boldsymbol{\Sigma})$. For

any finite subset of points $\boldsymbol{X} = \begin{pmatrix} \mathrm{x}_1 & \dots & \mathrm{x}_\mathrm{n} \end{pmatrix}^\top$ the affine transformation $f(\boldsymbol{X}) = \boldsymbol{X}\mathrm{w}$ is again distributed normally $f(\boldsymbol{X}) \sim \mathcal{N}\left(0, \boldsymbol{X}\boldsymbol{\Sigma}\boldsymbol{X}^\top\right)$. This gives us the following Gaussian process:

$$\begin{aligned} f(\mathrm{x}) &\sim \mathcal{GP}\left(0, k(\mathrm{x}, \mathrm{x}')\right) \\ k(\mathrm{x}, \mathrm{x}') &= \mathrm{x}^\top \boldsymbol{\Sigma} \mathrm{x}' \end{aligned} \tag{2.5}$$

A different example for a common kernel function is the squared exponential function. It is given by

$$k(\mathrm{x}, \mathrm{x}') = \exp\left(-\frac{1}{2}(\mathrm{x} - \mathrm{x}')^\top(\mathrm{x} - \mathrm{x}')\right) = \exp\left(-\frac{1}{2}\left\|\mathrm{x} - \mathrm{x}'\right\|_2^2\right). \tag{2.6}$$

This function is also referred to as radial basis function (RBF). The function results in a Gaussian Process where points that are close to each other have a high covariance. For applications where values tend to be similar when close to each other in space or time this kernel might provide a reasonable model.

2.1.1 Neural Networks as Gaussian Processes

There are many reasonable kernel functions. In this thesis we will focus on specific type of Gaussian processes that are derived from neural networks. This section will deal with this subject.

Bayesian Neural Network

A connection between neural networks and Gaussian processes was described by Neal [5]. To illustrate this behavior we will consider a simple neural network with one hidden layer. The output of the network $f(x)$ can be described by following equations:

$$\begin{aligned} a_j(x) &= b_j^{(0)} + \sum_i^D w_{j,i}^{(0)} x_i \\ f_j(x) &= b_j^{(1)} + \sum_i^H w_{j,i}^{(1)} \phi(a_i(x)) \end{aligned} \tag{2.7}$$

Here D is the number of input dimensions and H the number of hidden units. The weights and biases are given by w and b. ϕ denotes the activation function. The values of the hidden units before the activation are given by a.

Typically in machine learning the weights are initialized with random weights. The model then is trained by optimizing the weights using gradient descent. We however will consider a Bayesian approach. For this we set a normal prior on the weights

$$w_{i,j}^{(l)} \overset{iid.}{\sim} \mathcal{N}\left(0, \sigma_w^2\right) \tag{2.8}$$

$$b_j^{(l)} \overset{iid.}{\sim} \mathcal{N}\left(0, \sigma_b^2\right). \tag{2.9}$$

With this prior we can look at the distribution of $a_j(x)$. Since all the weights and biases are independently distributed $a_j(x)$ must also follow a normal distribution. The distribution is given by

$$a_j(x) \sim \mathcal{N}\left(0, \sigma_b^2 + \sigma_w^2 \sum_i^D x_i^2\right). \tag{2.10}$$

However the distribution of the neural network outputs $f_k(x)$ in general is not distributed normally. Firstly by applying the activation function ϕ to $a_j(x)$ the normal behaviour can be lost. Secondly even if $\phi(a_j(x))$ were distributed normally the product $w_{j,i}^{(1)}\phi(a_i(x))$ in general is not distributed normally. Nonetheless we still observe some interesting properties. The summands $w_{j,i}^{(1)}\phi(a_i(x))$ are independent and identically distributed. This allows us to make use of the central limit theorem. Let us consider the limit of $H \to \infty$. This results in the sum and by extension $f_j(x)$ being distributed normally. To show that f is a Gaussian process we also need the joint distribution of multiple points to be a multivariate normal random variable. For the joint distribution

$$\begin{bmatrix} f(x^{(1)}) \\ \vdots \\ f(x^{(n)}) \end{bmatrix} = b_j^{(1)} + \sum_i^H w_{j,i}^{(1)} \begin{bmatrix} \phi(a_i(x^{(1)})) \\ \vdots \\ \phi(a_i(x^{(n)})) \end{bmatrix} \tag{2.11}$$

we can again apply the central limit theorem for $H \to \infty$. Resulting in f being a Gaussian process.

Deep Neural Networks

As shown by Lee [6] even neural networks with multiple hidden layers can converge to a Gaussian process. We can describe a deep neural network by the following equations:

$$a_j^{(1)}(x) = b_j^{(0)} + \sum_i^D w_{j,i}^{(0)} x_i \tag{2.12}$$

$$a_j^{(i+1)}(x) = b_j^{(i)} + \sum_i^{H^{(i)}} w_{j,i}^{(i)} \phi(a_i^{(i)}(x)) \tag{2.13}$$

$$f_j(x) = a_j^{(n)}(x) \tag{2.14}$$

Since we now have multiple layers $a^{(l)}$ are the values of the hidden units at layer l before the activation. We use the same normal prior for the weights and biases as for the single layer case. The priors are given by equations (2.8) and (2.9). To show that this network converges to a Gaussian process we will take the limits $H^{(1)} \to \infty$, $H^{(2)} \to \infty$, ... consecutively. By induction we can then show the Gaussian process behaviour.

Induction Base We have already seen in the previous section that $a_j^{(1)}(x)$ follows a normal distribution. However we also need to add that $a_j^{(1)}(x)$ is identically and independently distributed over j. This is the case since the weights and biases are by definition iid. over j.

Induction Step To show that $a_j^{(l+1)}(x)$ is Gaussian and iid. over j we need to apply the induction criterion. The summands from (2.13) are all iid. since the weights and $\phi(a_i^{(i)}(x))$ are independent and identically distributed. Hence we can apply the central limit theorem for $H^{(i)} \to \infty$. Thus $a_j^{(l+1)}$ is distributed normally. The independence of $a_j^{(l+1)}$ follows from the fact, that the all random variables in (2.13) are iid.

This induction proof only ensures that the marginal distribution $f(x)$ of as single point is distributed normally. To extend this to a joint distribution of multiple input points we can apply the same proof with the multivariate central limit theorem.

Convolutional Neural Networks

The proof for deep neural networks can also be extended to convolutional neural networks. This was shown by Garriga-Alonso et. al. [9]. For the proof we will express the convolutional operations as matrix products. The convolution of the form

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \tag{2.15}$$

can be expressed as the matrix multiplication

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} & 0 & \mathbf{C} & \mathbf{D} & 0 & 0 & 0 & 0 \\ 0 & \mathbf{A} & \mathbf{B} & 0 & \mathbf{C} & \mathbf{D} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{A} & \mathbf{B} & 0 & \mathbf{C} & \mathbf{D} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{A} & \mathbf{B} & 0 & \mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}. \tag{2.16}$$

We will also later use the set P_μ to indicate all elements on the μth row of the matrix which are non zero. Hence, for this example we get $P_1 = \{1, 2, 4, 5\}$ and $P_3 = \{4, 5, 7, 8\}$. This way we can express the CNN as

$$A_j^{(1)}(x) = b_j^{(0)} + \sum_i^C W_{j,i}^{(0)} x_i \tag{2.17}$$

$$A_j^{(i+1)}(x) = b_j^{(i)} + \sum_i^{H^{(i)}} W_{j,i}^{(i)} \phi\left(A_i^{(i)}(x)\right) \tag{2.18}$$

$$f_j(x) = A_j^{(n)}(x). \tag{2.19}$$

The difference between this formulation and the one for fully connected neural networks is that we describe it in terms of feature maps instead of hidden units. $A_j^{(l)}$ is the feature map

of layer l before the activation. x_i is the i-th channel from the input image. C is the number of channels from the input image, whereas $H^{(l)}$ is the number of hidden channels at layer l. The weights $W_{j,i}^{(l)}$ are given by converting the convulutional filter to its corresponding matrix as seen in the equations (2.15) and (2.16). The activation function ϕ is applied element wise. We can prove the Gaussian behavior the same way as for the fully connected networks. For iid. normal weights in the convolutional filters and iid. normal biases, the CNN will converge to a Gaussian process for $H^{(l)} \to \infty$. We will not present the proof, as it is analogous to the fully connected case. The difference being that instead of each hidden unit of a single layer being normal iid. random variables, the feature maps at a single layer will be identical and independent multivariate normal random variables.

2.1.2 Neural Network Kernel

To use the Gaussian processes derived from neural networks in the previous section, we are interested in computing the covariance between two points and the mean at a single point. In this section we will firstly present the mean and covariance functions for convolutional neural networks. Secondly we will examine the covariance of the ReLU activation. All the equations we present here were derived from [9].

Convolutional Kernel

We will consider convolutional neural networks with following prior on its weights. The convolutional filters mapping the j-th feature map of layer l to the i-th feature map of layer $l+1$ have normal weights

$$\left(U_{i,j}^{(l)}\right)_{n,m} \overset{\text{iid.}}{\sim} \mathcal{N}\left(0, \frac{\sigma_w^2}{H^{(l)}}\right). \tag{2.20}$$

The biases applied at layer l to the channel j are also normaly distributed

$$b_j^{(l)} \overset{\text{iid.}}{\sim} \mathcal{N}\left(0, \sigma_b^2\right). \tag{2.21}$$

We will concentrate on a very specific type of convolutional neural networks. We require the output feature maps to be of size one by one. In this case the computation of the covariance becomes much simpler. This assumption can be made for classification and regression problems.

We will use μ and ν to index the pixels of feature maps. Whereas i and j index the channels. As all the channels are iid. we can compute the mean and covariance for an arbitrary channel j. We will firstly look at the mean of a feature map at a specific pixel μ. As the final feature map the single pixel will be the neural network output we do not need to consider the output

layer separately. The mean at layer $l+1$ before the activation will be zero,

$$\begin{aligned}\mathbb{E}\left[\left(A_j^{(l)}(x)\right)_\mu\right] &= \mathbb{E}\left[b_j^{(l-1)} + \sum_i^{H^{(l-1)}} \sum_\nu \left(W_{j,i}^{(l-1)}\right)_{\mu,\nu} \cdot \phi\left(A_i^{(l-1)}(x)\right)_\nu\right] \\ &= \mathbb{E}\left[b_j^{(l-1)}\right] + \sum_i^{H^{(l-1)}} \sum_\nu \mathbb{E}\left[\left(W_{j,i}^{(l-1)}\right)_{\mu,\nu} \cdot \phi\left(A_i^{(l-1)}(x)\right)_\nu\right] \\ &= 0\end{aligned} \tag{2.22}$$

since the weights have a zero mean and are independent to the activations of the previous layer.

The more interesting part is the covariance of two different inputs.

$$\begin{aligned}K_\mu^{(l+1)}(x,\hat{x}) &= \mathbb{E}\left[\left(A_j^{(l+1)}(x)\right)_\mu \cdot \left(A_j^{(l+1)}(\hat{x})\right)_\mu\right] \\ &= \mathbb{E}\left[b_j^{(l)} \cdot b_j^{(l)}\right] + \sum_i^{H^{(l)}} \sum_{\hat{i}}^{H^{(l)}} \sum_\nu \sum_{\hat{\nu}} \mathbb{E}\left[\left(W_{j,i}^{(l)}\right)_{\mu,\nu} \left(W_{j,\hat{i}}^{(l)}\right)_{\mu,\hat{\nu}} \cdot \phi\left(A_i^{(l)}(x)\right)_\nu \phi\left(A_{\hat{i}}^{(l)}(\hat{x})\right)_{\hat{\nu}}\right]\end{aligned} \tag{2.23}$$

$\left(W_{j,i}^{(l)}\right)_{\mu,\nu}$ and $\left(W_{j,\hat{i}}^{(l)}\right)_{\mu,\hat{\nu}}$ are independent for $i \neq \hat{i}$. We can also see in equation (2.16) that the values are independent when $\nu \neq \hat{\nu}$ for a given row μ. This allow us to drop two of the sums.

$$\begin{aligned}K_\mu^{(l+1)}(x,\hat{x}) &= \sigma_b^2 + \sum_i^{H^{(l)}} \sum_\nu \mathbb{E}\left[\left(W_{j,i}^{(l)}\right)_{\mu,\nu} \left(W_{j,i}^{(l)}\right)_{\mu,\nu}\right] \cdot \mathbb{E}\left[\phi\left(A_i^{(l)}(x)\right)_\nu \phi\left(A_i^{(l)}(\hat{x})\right)_\nu\right] \\ &= \sigma_b^2 + \sigma_w^2 \sum_{\nu \in P_\mu} \mathbb{E}\left[\phi\left(A_i^{(l)}(x)\right)_\nu \phi\left(A_i^{(l)}(\hat{x})\right)_\nu\right]\end{aligned} \tag{2.24}$$

The whole operation can also be expressed as the convolution

$$\begin{aligned}&\begin{bmatrix} K_1^{(l+1)}(x,\hat{x}) & \dots & K_n^{(l+1)}(x,\hat{x}) \\ \vdots & \ddots & \vdots \\ K_{(m-1)\cdot n+1}^{(l+1)}(x,\hat{x}) & \dots & K_{n\cdot m}^{(l+1)}(x,\hat{x}) \end{bmatrix} \\ &= \sigma_b^2 + \sigma_w^2 \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} * \begin{bmatrix} V_1^{(l)}(x,\hat{x}) & \dots & V_k^{(l)}(x,\hat{x}) \\ \vdots & \ddots & \vdots \\ V_{(o-1)\cdot k+1}^{(l)}(x,\hat{x}) & \dots & V_{o\cdot k}^{(l)}(x,\hat{x}) \end{bmatrix}\end{aligned} \tag{2.25}$$

with

$$V_\mu^{(l)}(x,\hat{x}) = \mathbb{E}\left[\phi\left(A_i^{(l)}(x)\right)_\mu \phi\left(A_i^{(l)}(\hat{x})\right)_\mu\right]. \tag{2.26}$$

In total we see that the kernel for the convolutional layers can be expressed by convolutions. This would already be sufficient to compute the covariance matrix for Gaussian process convolutional networks without activations. Next we will present the covariance between activations.

ReLU Activation

To fully express the convolutional neural network we also need to compute the equation (2.26). This value will depend on the choice of activation function. For the ReLU non linearity we get a closed form expression

$$V_{\mu}^{(l)}(x, \hat{x}) = \frac{\sqrt{K_{\mu}^{(l)}(x, x)\, K_{\mu}^{(l)}(\hat{x}, \hat{x})}}{\pi} (\sin\theta + (\pi - \theta)\cos\theta) \tag{2.27}$$

with

$$\theta = \arccos \frac{K_{\mu}^{(l)}(x, \hat{x})}{\sqrt{K_{\mu}^{(l)}(x, x)\, K_{\mu}^{(l)}(x, x)}}. \tag{2.28}$$

This closed form is reached by solving the double integral produced by the expectation in (2.26) using polar coordinates. The exact derivation is found in [10].

2.1.3 Gaussian Process Regression

Regression is a supervised machine learning problem. In the supervised setting we have a dataset consisting of some samples. Every samples consist of the input and target. The goal is to design a model that maps the input to the target as close as possible. In the special case of regression the target lies in a continuous domain. For example in $\mathbb{R}^D$. Such a model can be used to get a predicted target for a input where the true target is unknown.

Applications can be found in many different disciplines. For example in medicine models can be used to estimate the risk of a operation. The model might take inputs like weight, height, age, blood pressure and type of operation. An example from finance is the prediction of future stock development. Possible inputs are the past stock development or how often the company was mentioned in news articles.

Possible methods to tackle the regression problem include neural networks, k-nearest-neighbors and linear regression. We will look at how to make use of Gaussian processes for regression.

Exact Regression

For the Gaussian process we need the input domain to be continuous. We assume the inputs $\mathrm{x_i}$ to be elements from $\mathbb{R}^D$. Where D is the number of dimensions. For simplicity we will assume that the targets $y_i \in \mathbb{R}$ to be one dimensional values. For higher dimensional outputs the regression can be performed by using multiple models. Each regressing one single dimension.

We will use a dataset given by the pairs of inputs $\boldsymbol{X} = \begin{pmatrix} \mathrm{x_1} & \dots & \mathrm{x_n} \end{pmatrix}^\top$ and the targets $\mathrm{y} = \begin{pmatrix} y_1 & \dots & y_n \end{pmatrix}^\top$. For Gaussian process regression we assume the data to follow a Gaussian process $f(\mathrm{x}) \sim \mathcal{GP}(0, k(\mathrm{x}, \mathrm{x}'))$.

$$\mathrm{y} = f(\boldsymbol{X}) \sim \mathcal{N}(0, \boldsymbol{K}_{XX}) \tag{2.29}$$

The joint probability between two different sets of input again follows a normal distribution

$$f\left(\begin{bmatrix}\boldsymbol{X}\\ \boldsymbol{Z}\end{bmatrix}\right) \sim \mathcal{N}\left(0, \begin{bmatrix}\boldsymbol{K}_{XX} & \boldsymbol{K}_{XZ}\\ \boldsymbol{K}_{ZX} & \boldsymbol{K}_{ZZ}\end{bmatrix}\right). \tag{2.30}$$

However for inference we are interested in the distribution of $f(\boldsymbol{Z})$ conditioned on $f(\boldsymbol{X}) = \mathrm{y}$. The distribution is again normal with following parameters [8]:

$$\begin{aligned} f(\boldsymbol{Z}) \mid \boldsymbol{X}, \boldsymbol{Z}, f(\boldsymbol{X}) = \mathrm{y} &\sim \mathcal{N}(\mathrm{m}, \boldsymbol{C}) \\ \mathrm{m} &= \boldsymbol{K}_{ZX}{\boldsymbol{K}_{XX}}^{-1}\mathrm{y} \\ \boldsymbol{C} &= \boldsymbol{K}_{ZZ} - \boldsymbol{K}_{ZX}{\boldsymbol{K}_{XX}}^{-1}\boldsymbol{K}_{XZ} \end{aligned} \tag{2.31}$$

This conditional distribution effectively gives us the regression. By using all the data pairs of the dataset as $\boldsymbol{X}$ and y we can compute the mean and the covariance at every possible point. The mean effectively corresponds to the most likely value. For simple regression problems it might be enough to only look at the mean. Being able to calculate the covariance explicitly is noteworthy. Other methods like neural networks only provide a prediction. However we do not know how certain the prediction is. Using this information might be very useful to estimate the risk of acting on the prediction. Noteworthy is also that the covariance of the conditional distribution does not depend on the values of y. If the target values change only the mean needs to be recomputed.

An example of a Gaussian process regression can be seen in figure 2.1. We can clearly see how the variance increases as we move away from the given data points.

Regression with noisy targets

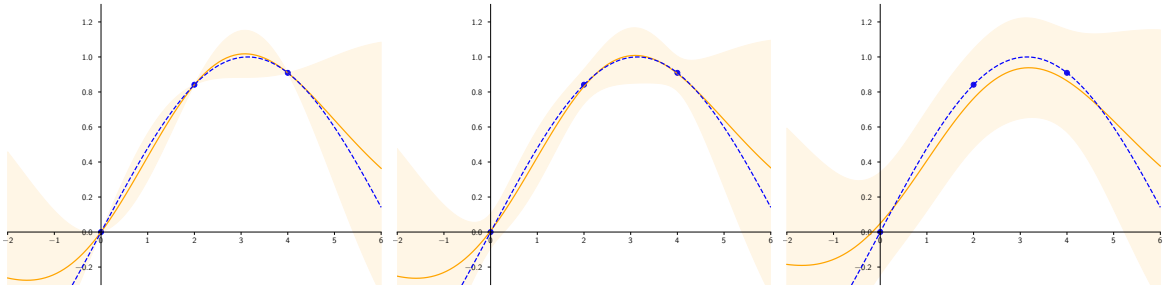
For most applications the exact target values are not known. Measuring devices might produce some error. Humans might mislabel the data. This can be modeled by assuming the target contains some added noise $\mathrm{y} = f(\boldsymbol{X}) + \epsilon$ [8]. Where the noise is distributed normally $\epsilon \sim \mathcal{N}\left(0, \sigma^2\boldsymbol{I}\right)$. We again can formulate the joint probability between two datasets. However this time the targets of the first dataset will have added Gaussian noise.

$$\begin{bmatrix}f(\boldsymbol{X}) + \epsilon\\ f(\boldsymbol{Z})\end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix}\boldsymbol{K}_{XX} + \sigma^2\boldsymbol{I} & \boldsymbol{K}_{XZ}\\ \boldsymbol{K}_{ZX} & \boldsymbol{K}_{ZZ}\end{bmatrix}\right) \tag{2.32}$$

Marginalizing out the points where the targets are given, we get the following equation:

$$\begin{aligned} f(\boldsymbol{Z}) \mid \boldsymbol{X}, \boldsymbol{Z}, f(\boldsymbol{X}) + \epsilon = \mathrm{y} &\sim \mathcal{N}(\mathrm{m}, \boldsymbol{C}) \\ \mathrm{m} &= \boldsymbol{K}_{ZX}\left(\boldsymbol{K}_{XX} + \sigma^2\boldsymbol{I}\right)^{-1}\mathrm{y} \\ \boldsymbol{C} &= \boldsymbol{K}_{ZZ} - \boldsymbol{K}_{ZX}\left(\boldsymbol{K}_{XX} + \sigma^2\boldsymbol{I}\right)^{-1}\boldsymbol{K}_{XZ} \end{aligned} \tag{2.33}$$

The effect of the noise can be seen in figure 2.1. If the target is assumed to be exact the mean will go through the given datapoints. The standard deviation at these points will be zero. Noisy labels will increase the standard deviation at and around the the points. Furthermore the mean might not follow the given samples.



(a) Exact Regression $\sigma = 0$ (b) $\sigma = 0.01$ (c) $\sigma = 0.01$

Figure 2.1: Gaussian process regression with different levels of noise added to the targets. The kernel is given by a radial basis function. Three points are used as training data. The blue points are sampled from a sine function, indicated by the blue dashed line. The orange line indicates the mean of the Gaussian process. The orange shaded area represents the standard deviation at each point.

2.1.4 Classification with Gaussian Process Regression

Classification problems have many parallels with regression problems. Both are trained in a supervised setting. For classification however the targets are not continuous values. Instead the target is a class label. Where each input is mapped to a specific class. An intuitive approach to bring this problem back to a regression setting might be to map each class to a numeric value. Then a regression could be performed. However this approach is very dangerous. We can illustrate this by looking at an example of classifying images of digits. The digits on the image are mapped to their corresponding value. The number 3,6,8 and 9 probably have a higher chance of being mistaken for each other. With a single continuous output however there is no decision boundary between each pair (except for 8 and 9). An output of 4.5 for a value on the decision border between 3 and 6 cannot be interpreted as such. It could also mean a number is on the border between 4 and 5.

A widely used approach in machine learning is to output a categorical distribution instead [11]. For each class a probability is outputted. Indicating if the input belongs to the class. For the output to be categorical all probabilities have to sum up to one. A regression model still can be used in this setting. For each class the model provides a latent value. All latent values are then normalized to probabilities. Many models including neural networks use the softmax function for this normalization [11].

$$\text{softmax}\left(\mathrm{x}\right)_i = \frac{\exp\left(\mathrm{x}_i\right)}{\sum_j \exp\left(\mathrm{x}_j\right)} \tag{2.34}$$

Where x is a vector containing the latent variables. We can see that the outputs of the softmax function will add up to one.

For a binary classification this approach can be reduced. Instead of fitting a categorical distribution with two values a single Bernoulli distribution can be fitted. Where the probability corresponds to one class. The probability of not being in that class corresponds to the

probability of being in the other class. Fitting the output of a regression model to the probability domain can be achieved by a squashing function. A single latent variable is transformed by the sigmoid function

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{2.35}$$

Fitting models that use these non linear transformations of latent variables cannot be directly transferred to a regression setting. Firstly we cannot fit on the latent variables directly, since we do not know the target latent values. Inverting the transformations would lead to infinite values. Secondly the transformation might make a closed expression of the fitted model impossible. For models that are fitted by gradient based optimization however these problems are not as applicable. Since the transformations are fully differentiable, the gradients can still be computed using the chain rule [11].

If we apply a non linear transformation π on a Gaussian process $f(\mathrm{x}) \sim \mathcal{GP}(0, k(\mathrm{x}, \mathrm{x}'))$ the resulting process $\pi(f(\mathrm{x}))$ in general is not Gaussian. Inference on this process is given by $\pi(f(\boldsymbol{Z})) \mid \boldsymbol{X}, \boldsymbol{Z}, \mathrm{y}$ assuming non noisy labels ($\mathrm{y} = \pi(f(\boldsymbol{X}))$). Inference can be divided into two steps [8]. Firstly the distribution of the latent function is computed.

$$p(f(\boldsymbol{Z}) \mid \boldsymbol{X}, \boldsymbol{Z}, \mathrm{y}) = \int p(f(\boldsymbol{Z}) | \boldsymbol{Z}, f) p(f | \boldsymbol{X}, \mathrm{y})\, df \tag{2.36}$$

Where $p(f|X, \mathrm{y})$ is the posterior of the latent function. Using the distribution of latent values we then can compute the class probabilities from the transformed values.

$$p(\boldsymbol{Z} \text{ belongs to class } c \mid \boldsymbol{X}, \boldsymbol{Z}, \mathrm{y}) = \int \pi_c(f(\boldsymbol{Z})) p(f(\boldsymbol{Z}) \mid \boldsymbol{X}, \boldsymbol{Z}, \mathrm{y})\, df(\boldsymbol{Z}) \tag{2.37}$$

Here π_c denotes the transformation function applied to the latent variables, assigning a probability value to class c. Analytically solving the integrals becomes intractable. The posterior of the latent function does not resolve in a Gaussian likelihood as in the regression case. There are methods to approximate solutions. The Laplace approximation [12] approximates the latent posterior by a Gaussian likelihood. Expectation Propagation [13] is different way to estimate the posterior in a tractable manner.

A more simple approach is done without the use of latent functions. Garriga-Alonso et. al. performed classification directly with exact regression [9]. The class labels were encoded in a one-hot vector. For each point the corresponding target is of dimension C. Where C is the number of classes. The resulting vector is filled with values $\mathrm{y}_i \in \{-1; +1\}^C$. Only the value at the position of the class is positive. All other values are negative. It is noteworthy that even though the labels are restricted to a domain $[-1, 1]$, the resulting mean and covariance function do not have these restrictions. Hence we cannot directly interpret the mean function as probabilities. However the model can still be used for classification. The authors assign the class corresponding to the highest value to the test sample.

2.2 Adversarial Attacks

The field of adversarial attacks is very broad and includes many different applications. It includes poisoning, oracle, evasion and more attacks. Poisoning attacks tamper with the training data. The goal can be to decrease a models accuracy [14] or to introduce backdoors [15]. Oracle attacks are applied using the model inference with the goal of gaining insights on the method. For example trying to recreate the training set [16] or using the methods predictions to train a new model without access to the original training data [17]. Evasion attacks are possibly the most prominent type of adversarial attacks. The goal is to trick the model during inference. As this is the attack we will be using in this thesis we will introduce it in more detail followed by several examples of evasion attacks.

2.2.1 Evasion attacks

Any non perfect model will have areas in which its predictions are incorrect. Evasion attacks exploit these areas. By modifying the input the attack will try to change the prediction. False predictions for small modifications can indicate that the model is not very robust. Especially if the input modification is indistinguishable for humans. Since this indicates that humans would have labeled the input differently than the model.

Optimization problem

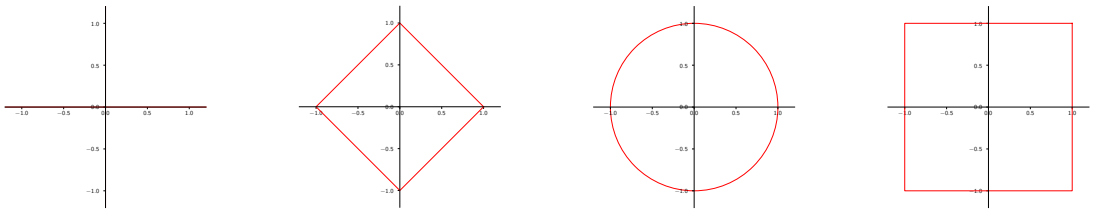
The attacker using an evasion attack will have to mostly opposing goals. Firstly, maximizing the confidence of the model towards a incorrect class. This is the malicious part of the attack. Secondly, minimizing the deviation from the original input to ensure that the adversarial samples can "fly under the radar". Meaning that the modification is hard to spot. To tackle these opposing optimization goals most attacks typically fix one while optimizing for the other. Restricting the modification while optimizing the model output can be formulated as the following optimization problem [2]:

$$\begin{aligned} \underset{e(\mathbf{x})}{\text{maximize}} \quad & f_{adv}(\mathbf{x} + e(\mathbf{x})) \\ \text{subject to} \quad & \|e(\mathbf{x})\| \leq L \end{aligned} \tag{2.38}$$

Where f_{adv} is the target we want the model to output, $\mathbf{x}$ is the input vector and $e(\mathbf{x})$ is the modification we add to the input. By restricting the norm of $e(\mathbf{x})$ to a threshold L we ensure that the modified input is close to the true input. An alternative formulation is to restrict the output of the adversarial target and minimize the offset:

$$\begin{aligned} \underset{e(\mathbf{x})}{\text{minimize}} \quad & \|e(\mathbf{x})\| \\ \text{subject to} \quad & f_{adv}(\mathbf{x} + e(\mathbf{x})) \geq c \end{aligned} \tag{2.39}$$

Where c is the minimum confidence we want the model to predict for the adversarial target. This attack does not allow control over the distance between the modified and the base input. Instead it finds the closest data point with a given label.



(a) L^0 norm (b) L^1 norm (c) L^2 norm (d) L^∞ norm

Figure 2.2: Unit balls of different norms ($\|x\| = 1$). The L^0 unit ball includes all point that lie on any of the axis excluding the origin.

The choice of the norm will have a significant impact on the attack. Possible norms include the L^0, L^1, L^2 and L^∞ norm. The unit balls of these norms are shown in figure 2.2. We can interpret these norms for images. The L^0 norm limits the number of pixels changed. The L^1 corresponds to the mean difference per pixel, while the L^2 norm reflects the mean squared difference per pixel. For the L^∞ norm we get the maximum difference per pixel. This norm is also referred to maximum norm. L^1 and L^2 losses are often used for image reconstruction task, especially if a model is trained by gradient descent. Both these losses can be differentiated and allow for a efficient optimization since the gradients flow through every pixel. However it is difficult to interpret the quantitative value of the L^1 and L^2 norms for images. The mean difference per pixel can lead to areas with very high differences that average out to low values over an entire image. In contrast the L^0 and L^∞ norms are easier to interpret for a image difference. As the L^0 norm limits the number of pixel changed this constrains modifications to be only local. The L^∞ norm allows all pixel to be changed while limiting the amount they are changed. The L^1 and L^2 norms lie somewhere between these two.

Evasion Goal

The formulation as an optimization problem above focused on making the model output a certain target. Following [18] we call this a targeted attack. In contrast an untargeted attack does not care about a specific output target. Instead the evasion goal is to make the model deviate from the ground truth. For a classifying model this boils down to making the model misclassify. We can easily reformulate the optimization problems. For the formulation shown in equation (2.38) we have to minimize the adversarial target instead of maximizing it. Analogously for equation (2.39) the limiting constraint has to be changed to an upper bound instead of a lower bound.

What attack is more useful is dependent on the application. If we consider real world examples this becomes clear. For example a malicious person might try to alter their appearance to be recognized as a political leader. Allowing him to spread propaganda and false information. This would be a targeted attack at the specific political leader. If a famous celebrity alters their appearance to not be recognized by paparazzi we have a untargeted attack. The celebrity does not care to be identified as any specific other person. It is only important to not be recognized as themselves.

For binary classification problems there essentially is no real difference between targeted and untargeted attacks. As moving away from one class is the equivalent to moving toward the only other class.

2.2.2 Fast Gradient Sign Method (FGSM)

Goodfellow, Shlens and Szegedy presented a method to generate adversarial examples without explicitly solving the presented optimization problems [19]. The examples are produced by following equations:

$$\begin{aligned} \mathbf{x}_{adv} &= \mathbf{x} + e(\mathbf{x}) \\ e(\mathbf{x}) &= \epsilon \cdot \text{sign}(\nabla_x J(\boldsymbol{\theta}, \mathbf{x}, t)) \end{aligned} \tag{2.40}$$

Where J is the loss function used to train the model, $\boldsymbol{\theta}$ are the model parameters, ϵ is a parameter to choose how close the adversarial example should be to the base. For this untargeted attack t is the target we want to avoid. Typical choices of t are either the ground truth or the models prediction with the goal of making the model misclassify. This method is similar to the optimization problem in equation (2.38). The norm $\|e(\mathbf{x})\|_\infty$ is limited by the choice of ϵ. However instead of finding the solution of the optimization problem, only a single step is taken. Similar to a gradient descent optimization the gradient is used. The direction of the step is given by the sign of the gradients to optimize the adversarial target function. Applying the sign function makes the gradient fall into the L^∞ unit ball. The size of the step is given by ϵ. This attack can be modified to use the L^1 and L^2 norms by normalizing the gradient instead of taking the sign.

The original authors showed that using this attack it is simple to produce adversarial examples. On the MNIST dataset this attack with $\epsilon = 0.25$ increased the error rate of an shallow softmax classifier to 99.9%. In this case $\epsilon = 0.25$ corresponds to a 25% change per pixel, since the pixel domain ranges from zero to one.

2.2.3 Basic Iterative Method (BIM)

An extension of the Fast Gradient Sign Method called Basic Iterative Method was introduced by Kurakin, Goodfellow and Bengio [20]. Examples are generated by repeatedly applying FGSM. After each step the values are clipped to fit inside a L^∞ unit ball. By following equation the adversarial examples are generated:

$$\begin{aligned} \mathbf{x}_{adv}^{n+1} &= \text{Clip}_{\mathbf{x},\epsilon}(\mathbf{x}_{adv}^{n+1} + e(\mathbf{x}_{adv}^{n+1})) \\ \mathbf{x}_{adv}^{0} &= \mathbf{x} \\ e(\mathbf{x}) &= \alpha \cdot \text{sign}(\nabla_x J(\boldsymbol{\theta}, \mathbf{x}, t)) \\ \text{Clip}_{\mathbf{x},\epsilon}(\mathbf{y})_i &= \max\{x_i - \epsilon, \min\{x_i + \epsilon, y_i\}\} \end{aligned} \tag{2.41}$$

Where $\mathbf{x}_{adv}^n$ is the example after n iterations of FGSM, α is the stepsize used in FGSM and ϵ defines the maximum L^∞ distance between the sample and the adversarial example. $\text{Clip}_{\mathbf{x},\epsilon}$ ensures that the element wise difference is smaller then ϵ. This corresponds to the L^∞ distance. Their results show that this attack causes a misclassification more frequently then by applying

the FGSM attack for small values of ϵ. For larger values the FGSM was a more malicious attack. The reasoning for this given by the author is that the FGSM will in general be at the border of the L^∞ limit, since it takes only a single step of that size. In contrast the BIM will be closer to the original image as the iterative approach does not force the adversarial image to this border. Another interesting result of the authors was that the attack even works for printouts of the image. They showed that printing out the adversarial examples and photographing the printouts causes misclassification significantly more often than for photographs of the base image printouts.

2.2.4 Pixel Attacks

An attack not based on the L^∞ norm was presented by Su, Vargas and Sakurai [2]. Their optimization target was given by:

$$\begin{aligned} \underset{e(\mathbf{x})}{\text{maximize}} \quad & f_{adv}(\mathbf{x} + e(\mathbf{x})) \\ \text{subject to} \quad & \|e(\mathbf{x})\|_0 \leq d \end{aligned} \tag{2.42}$$

Where d is the number of features where the input can be changed. For images this corresponds to the number of pixels that are modified.

In contrast to the previous attacks the gradient was not used for optimization. By using differential evolution (DE) the authors tackled the optimization problem. The modification $e(\mathbf{x})$ is encoded in a array of size d. Each element is a tuple containing the index of the feature being changed and the changed value. For DE a number of these modifications are randomly sampled. In multiple steps the samples are combined to new samples. Keeping only those with a high optimization score. Since DE does not require any information on gradients this method can even work on black box models.

In their experiments on the CIFAR-10 dataset the authors show how effective this attack is. For several convolutional neural networks a single pixel change if enough to misclassify over 60% of all samples. For tree and five pixels this rate even goes up to around 86%.

2.2.5 High Confidence Low Uncertainty Attack (HCLU)

Attacks become more complexity for machine leaning models that provide an uncertainty estimate in addition to the prediction. The uncertainty provides an additional safety against predictions that lie outside of the models capacity. Hence, an attack must also ensure that the uncertainty stays in a reasonable bound. This was formulated by Grosse et al. as High Confidence Low Uncertainty (HCLU) optimization problem [21]:

$$\begin{aligned} \underset{e(\mathbf{x})}{\text{minimize}} \quad & \|e(\mathbf{x})\|_2 \\ \text{subject to} \quad & f_{adv}(\mathbf{x} + e(\mathbf{x})) > 0.95 \\ \text{and} \quad & f_{unc}(\mathbf{x} + e(\mathbf{x})) \leq f_{unc}(\mathbf{x}) \end{aligned} \tag{2.43}$$

Where f_{adv} is the confidence of the adversarial output class and f_{unc} measures the uncertainty of the model at the given point. The constraint on f_{adv} ensures high confidence. Low

uncertainty comes from the second constraint which limits the uncertainty for the adversarial example to not be higher then for the base image. This way if the adversarial example is rejected for a low uncertainty the base image is also rejected by the model.

The authors showed that is is possible to construct adversarial examples for a Gaussian process. They generated the examples by solving the optimization problem using a L-BFGS-B solver.

3 Adversarial Attacks on Gaussian Processes

In this section we combine the topics introduced in the previous section, by applying adversarial attacks to a Gaussian process. In section 3.1 we will describe the attack we are using for our experiments. The dataset we use is described in section 3.2. Details about the implementation are in section 3.3. In section 3.4 we will present the Gaussian process and how we use it for classification. Additionally we will introduce multiple reference methods for comparison and evaluate their performance on a classification task. Before we run experiments in 3.6 and 3.7 we will discuss how to evaluate the robustness against adversarial attacks in section 3.5. In the final section 3.8 the runtime of different methods is analyzed.

3.1 Adversarial Attack

In this section will give the reasoning how we chose the attack used for our experiments and how we apply it. Furthermore we will also present the cross entropy loss, which is plays a large role in our attack and also the training of reference methods.

3.1.1 Fast Gradient Sign Method

There are many possible adversarial attacks that we could perform. Some are briefly shown in 2.2. In this thesis we will only focus on the Fast Gradient Sign Method. This has multiple reasons. The attack only requires a single gradient computation. From this gradient we can compute adversarial examples for all choices of the limiting norm $\|e(\mathbf{x})\|_\infty < \epsilon$. Methods like the Basic Iterative Method or Pixel Attacks require us to resolve the attack for every choice of ϵ. Hence, we can easily evaluate the attack for a wide range of ϵ using the FGSM.

Another reasons for using this attack is that is has only a few requirement for the model. The attack only needs a loss function and the ability to compute the gradient of this loss. This allow for a comparison between different models. In contrast the High Confidence Low Uncertainty Attack only is defined for models that provide a confidence and an uncertainty estimate. This would prohibit a comparison between conventional neural networks and Gaussian processes. We would like to note that there are neural networks that can provide an uncertainty estimate (e.g. with Monte Carlo Dropout [4]). However, we will not be using these neural networks in this thesis.

For our experiment we applied the FGSM attack as shown in algorithm 1. In this algorithm we can clearly see how we can get the adversarial examples for multiple choices of ϵ without any costly computation. The attack we applied is untargeted and uses the ground truth as labels to avoid. We also clip the adversarial examples to the image domain. This ensures that the adversarial examples could be stored as valid image files. Furthermore, any pixel that

lies outside of the valid range can be caught during prepossessing. This would allow for an easy detection of an adversarial attack. The loss function we used for the attack is the cross entropy loss which will be presented in the next section.

Algorithm 1: Evaluating the Fast Gradient Sign Method for multiple choices of ϵ

```
for x, y ∈ Dataset do
    prediction ← model(x);
    loss ← loss-function(prediction,y);
    gradient ← ∇_x loss;
    for ε ∈ {ε_1, ..., ε_n} do
        x_adv ← x + ε · sign(gradient);
        evaluate(model(x_adv)));
    end
end
```

3.1.2 Cross Entropy Loss

To evaluate how good a model fits to the given data we can have a look at the likelihood function [11]

$$p(y|d) = \prod_i \prod_c d_{i,c}^{y_{i,c}}. \tag{3.1}$$

Where $d_{i,c}$ gives us the probability output by the model for the sample i belonging to class c. $y_{i,c}$ is the ground truth probability. Typically for classification y_i is a one hot encoded vector. The only element set to one in the vector corresponds to the class. All other elements are set to zero. To simplify the optimization of this function typically the logarithm is applied, giving us the log likelihood

$$\log p(y|d) = \sum_i \sum_c y_{i,c} \log(d_{i,c}). \tag{3.2}$$

Since the logarithm is a monotonously increasing function a maximizer of the log likelihood will also maximize the likelihood. As loss function are typically minimized we can flip the sign. This gives us the Cross Entropy Loss

$$\text{cross-entropy}(y, d) = -\log p(y|d) = -\sum_i \sum_c y_{i,c} \log(d_{i,c}). \tag{3.3}$$

This function is sometimes also referred to as negative log likelihood. We use this function for the adversarial attacks in this thesis. Additionally, we will make use of this loss function for training reference methods.

3.2 MNIST Dataset

For all our experiments we use the MNIST dataset [22]. It contains images of handwritten digits and their corresponding label. Each image shows exactly one digit (0 to 9). The images

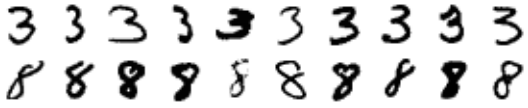
are of the size 28 by 28 pixels with only a single grayscale channel. The whole dataset consists of roughly 60 000 samples for training and 10 000 samples for testing.

Since most experiments would require long computation times using the entire dataset, we use specific subsets of the training dataset. Firstly we will use only the digits 3 and 8. We will call this dataset MNIST38. The resulting problem is a binary classification. Since we only have the classes 3 and 8 in the training data. We also reduce the test data to these classes. This dataset in total contains 11 982 images for training and 1 984 images for testing. The classes are roughly distributed evenly with 5 851 eights in the training set. The test set has 974 eights respectively. A sample of this dataset can be seen in figure 3.1a.

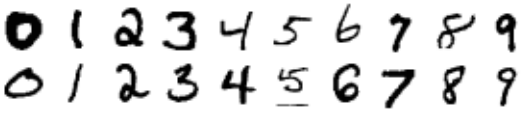
The second subset we will use contains all ten digits. However, to limit the number of training points, we reduce the training set to 1 000 images per digits. This gives us a training set with 10 000 images. We will refer to this dataset as MNIST1000. The subset is sampled randomly. However for every experiment we will use the same sampled subset to avoid a sampling bias between experiments. Some sample images from this dataset can be seen in figure 3.1b. As this subset still contains all classes, we use the whole test dataset.

For the adversarial attack we will use the training data to fit the models. Since some reference methods require an additional validation set, we will split the training set. Twenty percent of the training set are used for validation. The remaining eighty percent are used for actual training. This split is done randomly. However the same split is used for every method we fit. The test data is then used to evaluate the performance of the classification itself. Furthermore we evaluate the attacks on the test data.

The only preprocessing we apply to the data is scaling the range of the grayscale channel. It originally ranges from 0 to 255. We scale it to the range $[0, 1]$. We do not apply any other transformations, since the digits are already centered and fitted to the image size.



(a) MNIST38



(b) MNIST1000

Figure 3.1: Example digits from the MNIST subsets used for training.

3.3 Implementation

For the implementation of our attack we use python. We make heavy use of the PyTorch [23] package. PyTorch is a deep learning framework that supports the automatic calculation of gradients by backpropagation. This allowed us to easily compute the fast gradient sign method attack. Furthermore PyTorch was used for the training of the reference methods. For

the computation of the convolutional Gaussian process kernel we use the implementation from [9]. Since their implementation was also using PyTorch the adversarial attack could be implemented similarly to the attack on the reference methods. However we needed to make some slight adjustments in the authors original package to allow for backpropagation. These changes are discussed in section 3.4.3. Our modified version of their code is found at `https://github.com/waegemans/cnn-gp/tree/stable-backprop`. This package has similar API as the PyTorch package. Models can be defined the same way:

```
model = Sequential(
        Conv2d(kernel_size=7,
            padding="same",
            var_weight=var_weight,
            var_bias=var_bias),
        ReLU(),
        Conv2d(kernel_size=28,
            padding=0,
            var_weight=var_weight,
            var_bias=var_bias)
    )
```

The difference is that for the Gaussian processes we do not need to define the number of channels. Instead we initialize each layer with the variance of the weights and the biases. The forward pass

```
Kxz = model(X,Z)
```

does not give the model outputs for the inputs. Instead the forward pass is used to build the covariance matrix from two datasets. This matrix we can then use for the regression.

For data processing and evaluations we additionally used the packages NumPy [24] and scikit-learn [25]. The graphs in this thesis were generated using matplotlib [26].

We also would like to highlight the Adversarial Robustness Toolbox [27]. Although we did not use it in the experiments showed in this thesis it was helpful in exploring adversarial attacks. The toolbox contains implementations of various different adversarial attacks. Not even limited to evasion attacks. These attacks are implemented for PyTorch, Tensorflow, GPy and various other machine leaning libraries. The reason we did not use this toolkit for our experiments was the runtime. As the gaussian process we used is very slow we need to run the FGSM attacks without recomputing the gradients for every ϵ we want to examine. The attack implementation in the toolbox require a fixed ϵ. So instead of using the toolbox we chose to implement the attack using the algorithm 1 ourselves.

3.4 Classification Methods

Before we can evaluate any adversarial attacks we need machine learning models to attack. In 3.4.1 we will give a detailed explanation of the Gaussian process derived from an CNN we will use for classification. The section 3.4.2 will present how we used the Gaussian process

regression for classification. We discuss the differentiability of the Gaussian process in 3.4.3. As we evaluate the Gaussian process not without an comparison we also have some reference methods. This are shown in 3.4.4. In the final part of this section we evaluate how good the Gaussian process and the reference methods are at a classification task.

3.4.1 ConvNet-GP

We follow Garriga-Alonso et. al. [9]. Like in their publication we will call this method ConvNet-GP. Firstly the kernel we chose is derived from a CNN. The neural network consists of seven convolutional layers. Each convolution uses a 7x7 filter. The feature maps are padded with zeros to keep the dimensions equal for all layers. Every convolution is followed by a ReLU non linearity. The final layer has uses a 28x28 filter with no padding and no non linearity following. This represents a fully connected layer.

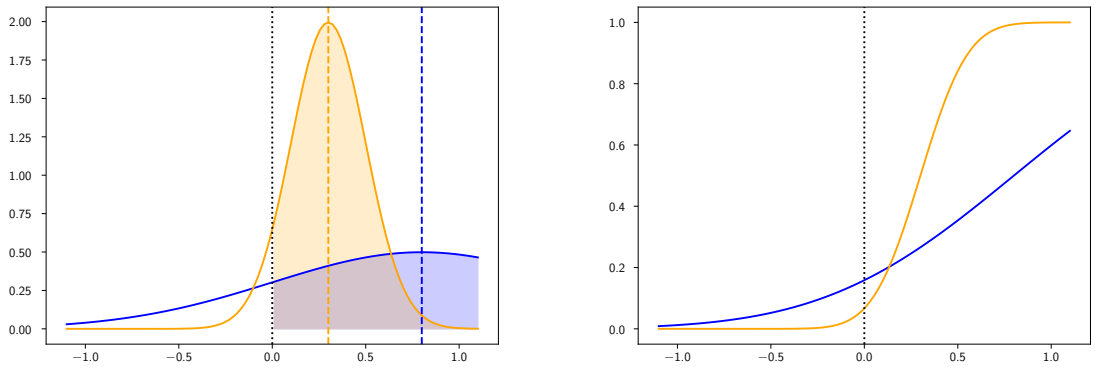
The actual classification is done as described in section 2.1.3. For multiple class classification we transform the labels to a one hot encoding. We then use this encoding for an exact regression without any transformations like the softmax function.

For binary classification we slightly modify the approach of the authors. Instead of a one hot encoding we transform the label into a on dimensional scalar. With a value of -1 for one class and value 1 for the other class. These labels we again use for an exact regression. However we only need to regress to one dimension which makes the problem slightly easier.

As an extension of the authors method, we also include the variance information provides by the model. Instead of solely looking at the class predicted by the mean, we calculate probabilities for each class based on the mean an variance. This step is explained in 3.4.2. With these probabilities we get comparable results to the output of neural networks. Where neural networks get the probabilities by a softmax of sigmoid function. Having these comparable outputs allows us to use the same attacks with the same loss functions.

3.4.2 Probabilities from Gaussian Process Regression

The approach we are following from [9] does not output any class probabilities directly. Training is done with $\{-1, +1\}$ labels for a binary classification. Linearly transforming the range given by the labels to a probability range can have some unwanted properties. Firstly the mean function for new points is not restricted to the range $[-1, 1]$. Meaning that the transformation might output values larger then one or smaller then zero. These values cannot be interpreted as probabilities. Secondly the Gaussian process does not output a single scalar value. Instead a normal distribution is given for every sample. This again means that a fixed confidence interval might not lie entirely in the probability range when transformed. In the paper [9] the authors did not require any probabilities, since they were only interested in the classification result. They choose the class where the normal distribution had the highest mean. However only considering the mean function will lose some information. We can illustrate this in the binary case. For labels of $\{-1, +1\}$ for the classes the decision boundary lies at zero. In figure 3.2 we can see that a higher mean does not imply a higher probability of the value being above the decision boundary. The variance also can impact this probability.



(a) probability density function

(b) cumulative distribution function

Figure 3.2: Two different normal distributions. The blue distribution has a larger mean. However since the orange one has a smaller variance, the probability of a positive value is higher for the orange distribution. This is illustrated by the shaded area in the probability density plot. The cumulative distribution function also reflect this. The orange CDF has a lower value at zero.

This probability for a normal variable $X \sim \mathcal{N}(\mu, \sigma^2)$ being larger smaller then a decision boundary d is given by

$$P(X < d) = \int_{-\infty}^{d} f_X(t)\, dt = F_X(d) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{d-\mu}{\sigma\sqrt{2}}\right)\right]. \tag{3.4}$$

f_X is the probability density function of X and F_X is the cumulative distribution function. The error function is given by

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt. \tag{3.5}$$

In this thesis we will convert the output of the Gaussian processes to the probability of being larger then the decision boundary. For a classification with multiple classes we calculate this probability for each class. Resulting in multiple Bernoulli distributions instead of a single categorical distribution. We transform these distributions to a categorical one by normalizing them to sum up to one. Using these categorical scores we can calculate the cross entropy loss for the Gaussian process. The loss we will then use for the FGSM adversarial attack.

Furthermore, having a meaningful probability score assigned to each class allows for some interpretation. Classifications with lower probabilities could be rejected. Setting a threshold for rejection carefully allows us balance the false positive and the false negative rates of our model. Only using the mean function for rejecting samples would still have the same problem seen in figure 3.2. As many other machine learning models also produce probabilities we can compare the Gaussian processes with these models directly, especially for adversarial attacks.

3.4.3 Backpropagarion of Convolutional Gaussian Processes

Since we compute the gradients used for the attack by backpropagation, we need to ensure that all operations are differentiable. The operations for inference from the covariance matrix (2.31) are differentiable, since they are linear operations. The functions we apply after the inference (erf, cross-entropy) are also differentiable. For the covariance matrix itself it is not as clear. For the ConvNet-GP convolutional layers are differentiable, since again only linear operations are applied. However the ReLU non-linearity requires trigonometric functions. The sine and cosine function needed in (2.27) are fully differentiable. The arccosine in (2.28) is only defined for the range $[-1, 1]$. The authors of [9] only clipped the input to that range for cases where numerical errors caused it to lie outside of that range. However, at the borders -1 and 1 this function is not differentiable. In our work, we clipped the range slightly tighter to $\left[-1 + 10^{-6}, 1 - 10^{-6}\right]$. This modification allows us to calculate the gradient by backpropagation.

3.4.4 Reference Methods

We want our experiments to give insights on the advantages and disadvantages of the Gaussian process. To do this we will set them in comparison with other machine learning methods used for classification. The methods we will compare the ConvNet-GP to are shown in this section. Furthermore, we present how we train these methods.

ConvNet-CNN

Instead of choosing arbitrary convolutional neural network architectures for comparison, we will use the finite channel CNNs from which the Gaussian process was derived from. For this we choose a number of channels C we want the hidden layers to have. The architecture of the CNN will then be the same as described above. With the difference that all hidden layers will have C number of channels. We will refer to this network as ConvNet-CNN-C. However for this network we will not use Bayesian weights. Instead the weights are randomly initialized and then optimized during training. For the binary case the output dimension will be one. This value will be squashed by a sigmoid function to map it to the range $[0, 1]$. For multiple classes the output will have number classes dimensions. The output is normalized by applying a softmax function. The normalized values can be interpreted as probabilities for a categorical distribution.

Lenet5

As second comparison we use the convolutional neural network published by LeCun [28]. This network called Lenet5 was specifically designed for the MNIST dataset. The network architecture consists of three convolutional layers followed by two fully connected layers. Each has a convolutional kernel of size 5x5. The first layer has six hidden channels and uses a zero padding of 2 pixels on each size. The second and third layer have no padding with 16 and 120 hidden channels. After the first two convolutional layers the feature maps

are sampled down using maximum pooling with a stride and filter size of two. After the three convolutional layers the feature maps are of size 1x1 pixels. The first fully connected layer uses 84 hidden units. The final output layer has a one unit for the binary classification. For multiple classes the output dimension is the number of classes. This method also uses sigmoid and softmax functions to squash the outputs to probabilities. The activation function tanh is applied after every layer, except for the last one.

Logistic Regression

The third reference method will be logistic regression. This is a non convolutional method. We will use the definition from [11]. In the binary case logistic regression is given by following function:

$$f(x) = \sigma(w^\top x + b) \tag{3.6}$$

Here σ is the sigmoid function. The weights are given by w and a scalar bias is given by b. For many problems the input is often times transformed using basis functions. For example by using the basis functions $\phi(x)_n = x^n$. This would allow a polynomial relationship to be described. We will not apply any basis functions and use the pixel values instead. The images are transformed to a vector by simply flattening the image into a single domain. While the output of the model has no directly linear relationship to the inputs, it has still some linear aspects. The decision boundary between the two classes is linear. The decision boundary lies at $f(x) = 0.5$ which is the case for $w^\top x + b = 0$.

The multi class logistic regression is very similar to the binary case. Following equations describe the multi class logistic regression:

$$f(x) = \text{softmax}(W^\top x + b) \tag{3.7}$$

In this case the weights are given by the matrix W and the bias term b is a vector. We can see that the logistic regression is equivalent to a neural network with no hidden layers.

The simple nature of this method allows for very well developed optimization methods. For example a second order optimization by the Newton method can be used. This typically is not possible for neural networks. However, in this work we will not use these methods. Instead we will use the same training as for the neural networks which is described in the next section. We do this since the speedup from second order optimization does not play a huge role. In comparison to the neural networks or the Gaussian process, logistic regression is trained a lot faster while the classification accuracy is lower.

Training of Reference Methods

All reference methods are trained on the data by optimizing the weights using Adam. Adam is a variant of stochastic gradient descent [29]. The optimizer uses a fixed learning rate of 10^{-3}. As loss function we use binary cross entropy for the binary case and analogous cross entropy for more classes. The training is done in batches of 64 samples. After each epoch the we evaluate the loss function on a the validation set. If the validation loss does not improve

model	accuracy[%]			loss (binary cross entropy)		
	training	validation	test	training	validation	test
ConvNet-GP	**100.0**	99.6	99.8	**0.000**	0.012	0.009
ConvNet-CNN-16	**100.0**	99.7	**99.9**	**0.000**	**0.006**	**0.003**
ConvNet-CNN-32	99.9	**99.9**	99.7	0.005	0.008	0.008
ConvNet-CNN-64	99.9	99.8	99.8	0.004	0.008	0.008
Lenet5	99.7	99.4	99.5	0.010	0.028	0.015
Logistic Regression	96.8	97.4	96.8	0.098	0.095	0.096

Table 3.1: MNIST38 Classification Results. Accuracy and loss are rounded to 1 and 3 decimals. The training loss of ConvNet-GP is truly zero whereas for the ConvNet-CNN-16 the loss is rounded.

for three consecutive epochs we finish the training and select the model from the epoch with the best validation loss. The weights of the networks are initialized using the Xavier method introduced in [30]. The goal of this initialization is to avoid exploding or vanishing gradients. This is especially useful for methods with saturating activation functions.

3.4.5 Classification Results

In order to discuss adversarial robustness we need to ensure that the models produce meaningful full results when not attacked. We evaluate the predictions of the models based on accuracy and the loss. The accuracy is given by the fraction of correctly classified samples. The loss function we evaluated is the cross entropy loss, since we use this loss for training of the neural networks and for the adversarial attack. We evaluate the model separately on the three disjunct datasets for training, validation and testing. The evaluation on the test data is the most important measure. As the training and validation sets were used to fit the models or optimize hyperparameters. Thus the model could be overfitted on these samples and not generalize well to new datapoints.

The results for the methods trained on the binary MNIST38 dataset are shown in table 3.1. We can see the methods ConvNet-GP and ConvNet-CNN-16 having a perfect accuracy and zero loss on the training data. The other convolutional neural networks are marginally worse on the training set. On the validation set the Gaussian process and the CNNs have very close performance with accuracies between 99.4% and 99.9%. The highest validation accuracy we observe for the ConvNet-CNN-32. Like for the validation set these models have a very similar accuracy on the test set. The accuracy ranges from 99.5% to 99.9%. ConvNet-CNN-16 having the best accuracy on the test sets. Overall we can see these method are comparable. Only the logistic regression trails them with accuracies of 96.8%, 97.4% and 96.8% on the training validation and test sets. However these values are still very high. Especially if we consider the simplicity of the logistic regression.

For the multi class dataset with all ten digits the results are shown in table 3.2. We will first look at the methods trained on the reduced MNIST1000 dataset. Here we again observe

model	accuracy[%]			loss (cross entropy)		
	training	validation	test	training	validation	test
ConvNet-GP	**100.0**	**97.9**	98.2	0.000	0.110	0.092
ConvNet-CNN-16	99.5	**97.9**	98.2	0.016	0.092	0.065
ConvNet-CNN-32	99.1	97.7	97.8	0.028	0.113	0.085
ConvNet-CNN-64	99.2	97.5	**98.3**	0.024	0.103	0.058
ConvNet-CNN-128	98.0	96.6	97.0	0.065	0.111	0.104
Lenet5	**100.0**	97.6	98.0	0.006	0.099	0.068
Logistic Regression	93.7	91.0	91.6	0.239	0.333	0.299
ConvNet-CNN-16 (full MNIST)	99.5	**99.0**	**99.0**	0.015	0.041	0.034
ConvNet-CNN-32 (full MNIST)	99.1	98.8	98.8	0.031	0.043	0.037
ConvNet-CNN-64 (full MNIST)	99.6	98.7	98.8	0.014	0.043	0.040

Table 3.2: MNIST Classification Results. Accuracy and loss are rounded to 1 and 3 decimals. The top method are trained on the MNIST1000 dataset. The bottom three are trained on the entire MNIST dataset. The test set is the entire test set and the same for all methods.

similar results as for the binary dataset. All of the convolutional methods have very similar accuracies. However, we do observe that the accuracies have dropped in comparison to the simpler MNIST38 data. We find the best test accuracy for the ConvNet-CNN-64 at 98.3%. The equivalent Gaussian process just slightly does not match that with an test accuracy of 98.2%. Overall we see very high accuracy scores. Again logistic regression does worse then the convolutional approaches. Since we reduced the training set only to manage the complexity of the Gaussian process, we will also evaluate the neural networks trained on the full dataset. The ConvNet-CNNs trained on the entire training data produce better results. The test accuracies for ConvNet-CNNs with 16, 32, and 64 channels range are all higher then 98.8%. This is better then the best model trained on the reduced set. This shows us that the complexity of the Gaussian process comes at an cost for large datasets. The faster CNNs can use more training data. However, for the same data set we get similar results between the GP and the CNN.

3.5 Evaluating Adversarial Robustness

To evaluate how robust methods are against an attack we need some metrics. Ideally we then can measure this in a scalar value. This allows us to evaluate the relationship of ϵ and the metric. In this section we will present the metrics used to evaluate the experiments we have performed.

3.5.1 Classification Results

The straightforward way of evaluating the robustness against the attack is to look at the classification result. Inherently the attack tries to manipulate the output labels. Monitoring the accuracy will tell us how successful this attack was. Furthermore the loss function can also be considered. An increased loss indicates that the attack is going in the correct direction, even if it was not enough to flip the output label. As we used both these metrics to evaluate the performance of the models, we already have reference points.

3.5.2 Receiver Operating Characteristics (ROC)

By evaluating the accuracy we reduce the output of the models. For the accuracy only the class with the highest probability output is considered. We do not evaluate the probability itself. However the probability could be used as additional information. If the models outputs a low probability score for the highest class we might want to reject that sample. Especially for real world application this could be a valuable information. To evaluate how good the models perform we will look at receiver operating characteristics curves (ROC curves). We will present a brief explanation about the concepts needed for the evaluation. For more detail on the ROC curves we refer to [31].

The main idea behind ROC curves is finding a threshold at which we reject our classification result. For a given rejection threshold we can divide every sample into one of four groups. True positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). Here positives mean that the result is kept i.e. not rejected. While negatives means that we reject the classification result for that sample. From these groups we can define the true positive rate and the false positive rate

$$\begin{aligned} TPR &= \frac{|TP|}{|P|} \\ FPR &= \frac{|FP|}{|N|}. \end{aligned} \tag{3.8}$$

The true positive rate is given by the fraction of correctly classified samples where the result is kept. The highest true positive rate is achieved by keeping all results. In contrast the false positive rate is given by the fraction of falsely classified samples where the result is kept. We are looking to minimize this value. The lowest false positive rate is given by rejecting every result. While the extreme cases of rejecting all or rejecting none are already known for every model. They are not useful for any applications. Instead we are more interested in balancing the maximization of TPR and the minimization of FPR. They are two opposing targets. The exact balance is heavily dependant on the application. For example in the medical field a quick test for a viral disease may require a higher TPR to stop further contamination as soon as possible. The FPR might not be as relevant, as the consequences of individuals being quarantined may not be as relevant. In contrast before performing a highly risky medical procedure like chemotherapy it should be ensured that an actual cancer threat is present. Hence, here a lower FPR might be reasonable.

Since we are less interested in finding a threshold for a certain FPR or TPR we will look at the relationship of these two values. To do this we can plot them against each other. This way we obtain the ROC curve. An example is shown in figure 3.3.

We are interested in maximizing the true positive rate while minimizing the false positive rate. Visually we can see if this is the case by looking at the plot. However we can also express this quantitatively. We want the ROC curve to follow the left top border as close as possible. This is the case if the area under the curve (AUC) is large. As the optimal model will follow the unit box the maximal value for the area under the curve is one. A random model with the diagonal ROC curve will have an AUC of 0.5. With the scalar AUC value we can compare different methods quantitatively. This is especially useful if the ROC curves intersect or there are many different ROC curves we want to compare.

3.5.3 Visual Evaluation

Metrics allow us to compare the robustness of different method quantitatively. We can compare certain metrics to the limit of the norm L, or in our case ϵ for the FGSM attack. This however might not measure the adversarial robustness completely. For any choice of ϵ two cases can occur. Either the adversarial example still be considered by humans to be of the original class or humans would choose a different label for the adversarial example. For every sample the border between the original class and a different one will lie at an individual choice of ϵ. In general we cannot say that an adversarial example with e.g. $\epsilon = 0.1$ will correspond to the original class. Even if this is very likely to be the case for small choices of ϵ. To examine if the model is fooled by an adversarial attack, we will also inspect the adversarial examples visually. That way we can ensure that the model is actually misclassifying samples instead of the attack producing samples from a different class.

3.6 FGSM Attack on MNIST38

In this section we will look at the fast gradient sign method attack applied to the MNIST38 dataset. In our experiments we performed the attack using the training set. For multiple choices of ϵ we ran the attack. The results of the attack are shown in figure 3.4. The choice of ϵ is plotted on the x axis. Accuracy, loss and ROC AUC scores are plotted on the y axis. Since the grayscale channel of images are scaled to the range $[0, 1]$ a value of $\epsilon = 0.15$ means that every pixel can change by a maximum of 15% of the channel range. We can see that generally for all models the accuracy and the loss worsens with a increase of the L^∞ limit ϵ. When looking at the accuracy in more detail, we see that the up to $\epsilon = 0.1$ the models ConvNet-GP and ConvNet-CNN-32 maintain high accuaracies of above 90%. ConvNet-CNN-16 is slightly worse with 81.2% at $\epsilon = 0.1$. However the other three models have already dropped significantly at this point. ConvNet-CNN-64 has an accuracy of 54.0%. Logistic Regression and Lenet5 only reach 29.2% and 38.2% accuracy. As ϵ increases further the accuracy rapidly approaches zero for almost all methods. Only for the ConvNet-CNNs with 16 and 32 channels the accuracy stagnates at around 37% and 25%. We will explain the

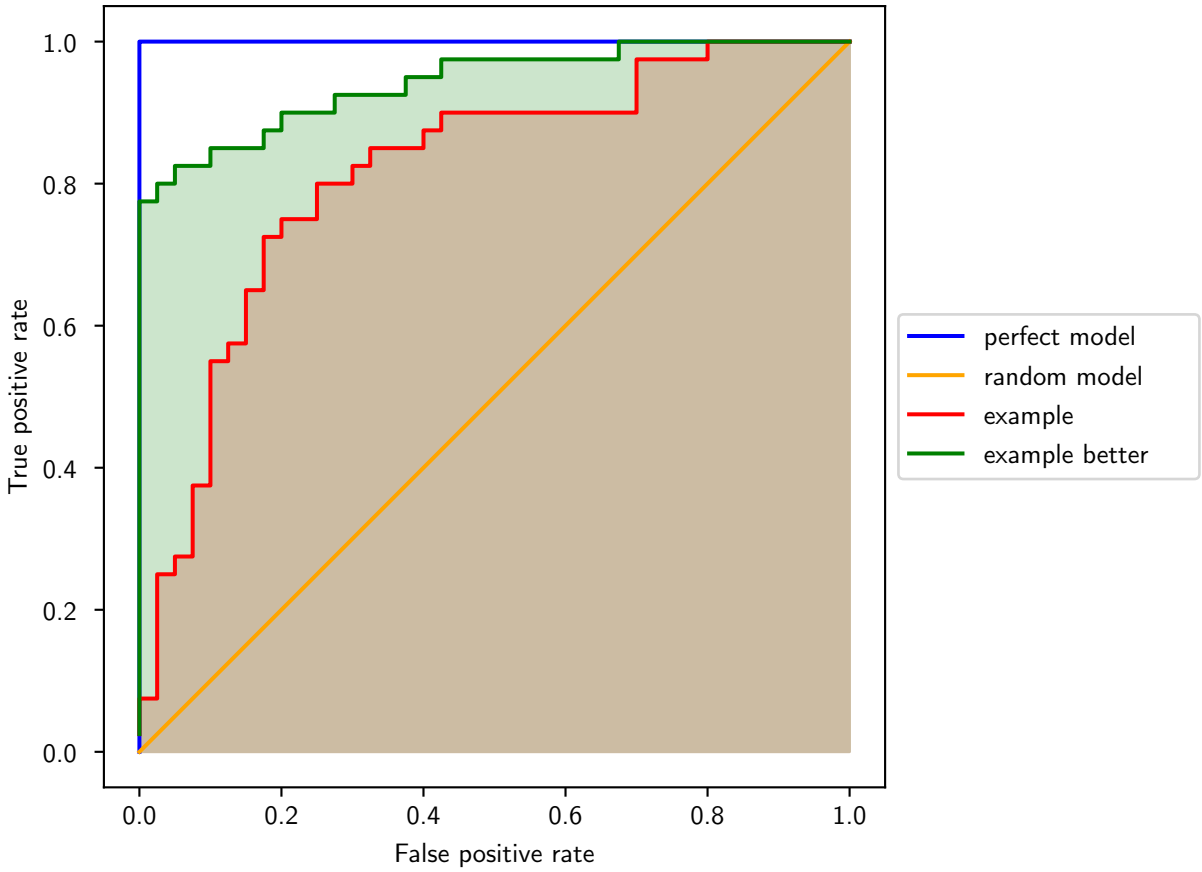


Figure 3.3: Receiver operating characteristics curve is generated by plotting the false positive rate against the true positive rate. A random model will follow the diagonal. Every model that is above the diagonal is better then random. An optimal model will follow the left top outside of the unit box. Models closer to the top and left border are better. They have a higher true positive rate at the same false positive rate and a lower false positive rate for the same true positive rate. For two example models the area under the curve is indicated by the shaded area.

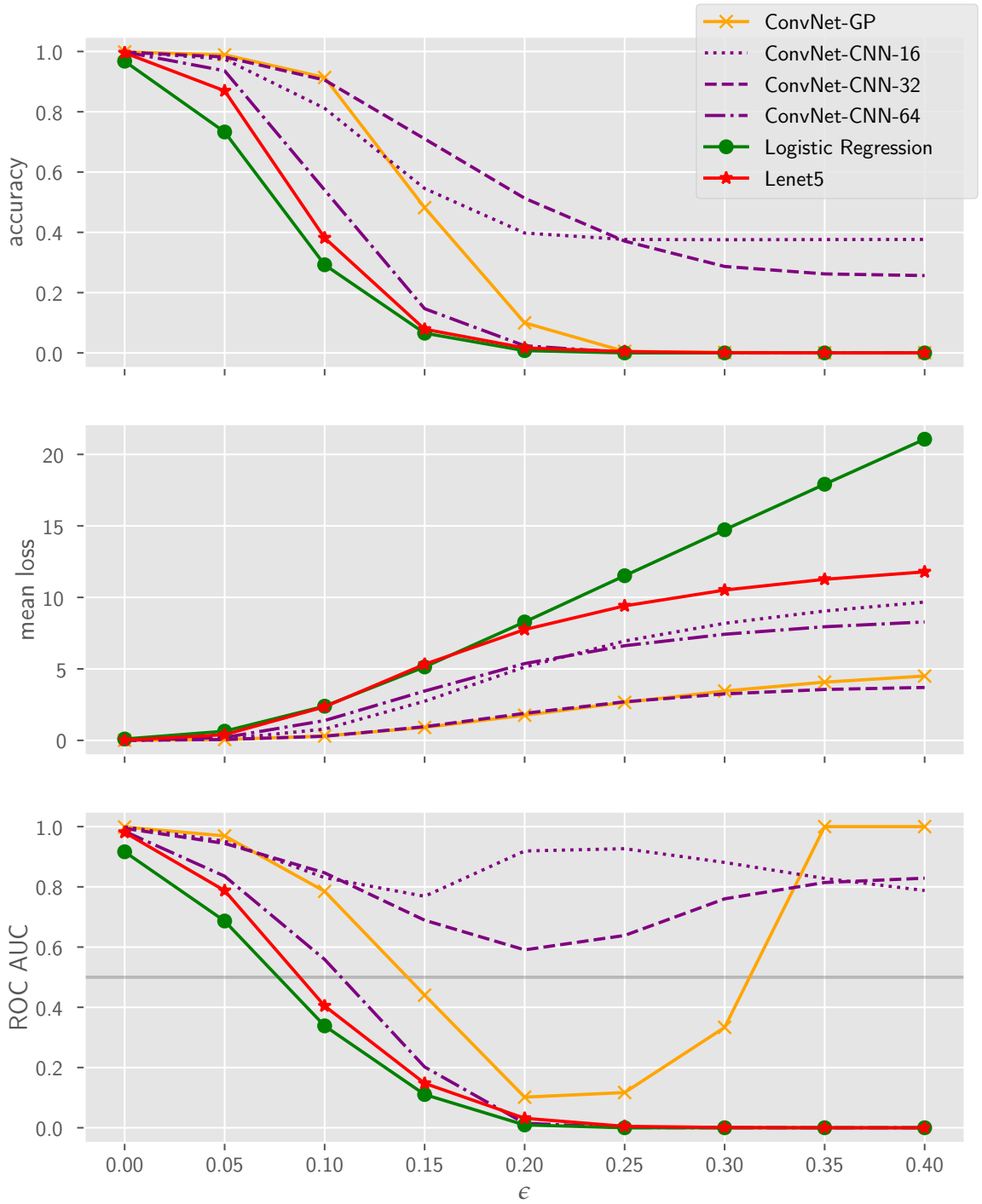


Figure 3.4: Quantitative results of the FGSM adversarial attack. Methods are trained on the binary MNIST38 dataset. Attack is performed on the respective subset from the MNIST test set.

reason for this later while discussing the ROC AUC curve.

The ROC AUC curves has some interesting properties. Firstly we see that for Logistic Regression, Lenet5 and ConvNet-CNN-64 the ROC AUC decreases steadily as the accuracy does. This behavior is what we generally expect. A worse model should have lower accuracy and lower ROC AUC. However the other three methods have an increase in ROC AUC after a initial decrease. At the point $\epsilon = 0.4$ the accuracy for ConvNet-GP is about 0.1%. More precisely exactly a single sample is classified correctly. The perfect ROC curve with an AUC of one can be explained by this sample having the highest score. Hence we can separate the correctly classified samples from the incorrectly sampled ones perfectly. A similar situation arises for the ConvNet-CNNs with 16 and 32 hidden channels. However here the border between correctly and incorrectly classified samples cannot be drawn as clearly. This behavior show us that the ROC AUC cannot be interpreted independently in our use case. Nonetheless we still can explain this behaviour. The three methods ConvNet-GP, ConvNet-CNN-16 and ConvNet-CNN-32 correctly predict one, 747 and 509 adversarial samples respectively at $\epsilon = 0.4$. If we look at the number of test samples where the computed gradients are zero we get a similar distribution of one, 714 and 417. The prediction for these samples does not change with an increase of ϵ. Hence, it makes sense that the accuracy is lower bound by these samples. For the other methods there is not a single sample where the gradient is entirely zero. These missing gradients arise from the squashing function we apply to the models. The sigmoid function used for the neural networks has an very low gradient for high absolute values. Which means the model prediction is very certain. Analogous for the Gaussian process we use the cumulative distribution function. This function has very low gradients either for a large difference between the mean and the decision boundary or for a low variance. This again reflects predictions with a high certainty.

We can also investigate the effect of the hidden channels for the ConvNet-CNNs. We have already seen that there are more samples with zero gradients for the lower channeled CNNs. This might be caused by overfitting. As the wider models are more likely to overfit. However we did not see this behavior for the more simple Lenet5 and the Logistic Regression. We cannot detect a direct relationship between the adversarial robustness and the number of channels. We even see the accuracy of the 16 and 32 channeled CNNs cross at about $\epsilon = 0.25$. Furthermore we cannot see any relationship from the CNNs to the derived Gaussian process.

Adversarial examples for the Gaussian process are shown in figure 3.5. The misclassified samples are marked with a red border, the correct ones with a green border. Under each image the confidence of the model is given. For every sample in the figure we can see that the smallest ϵ at which it is incorrectly classified still is recognizable. This tells us that the attack is successful. We can produce images where the prediction of the model diverges from the human label. However, in the last column it is not as clear what the ground truth of the image is. Especially for the eights. We can see that the lines on the left side of the number slowly vanish, making them look more like the number three. For comparison the adversarial examples for the ConvNet-CNN-16 are shown in figure 3.6. Here we firstly see that most of the eights have a zero gradient and thus do not change. For the samples with a gradient we again see that already get classified incorrectly at a point where the image is still

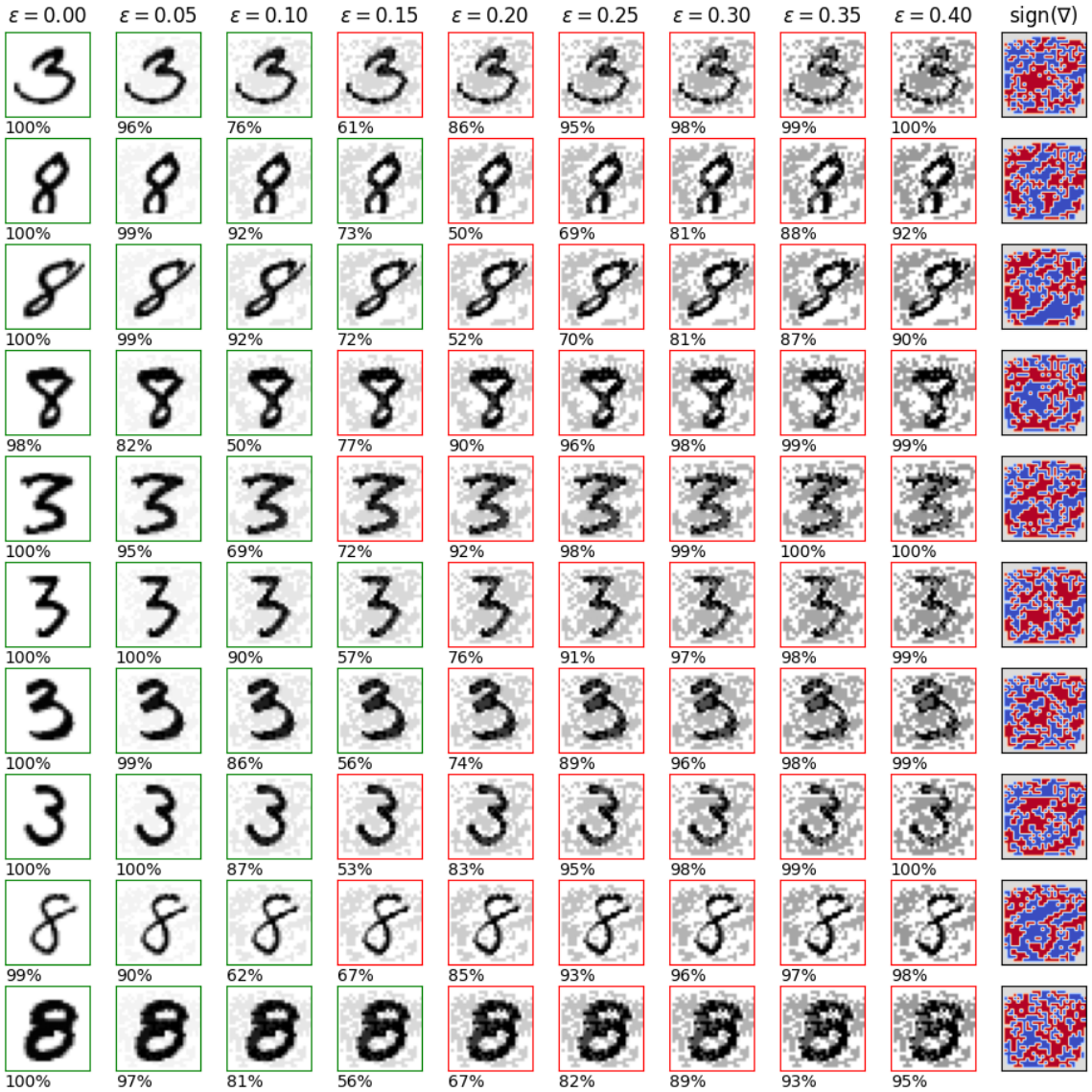


Figure 3.5: MNIST38 adversarial examples for ConvNet-GP

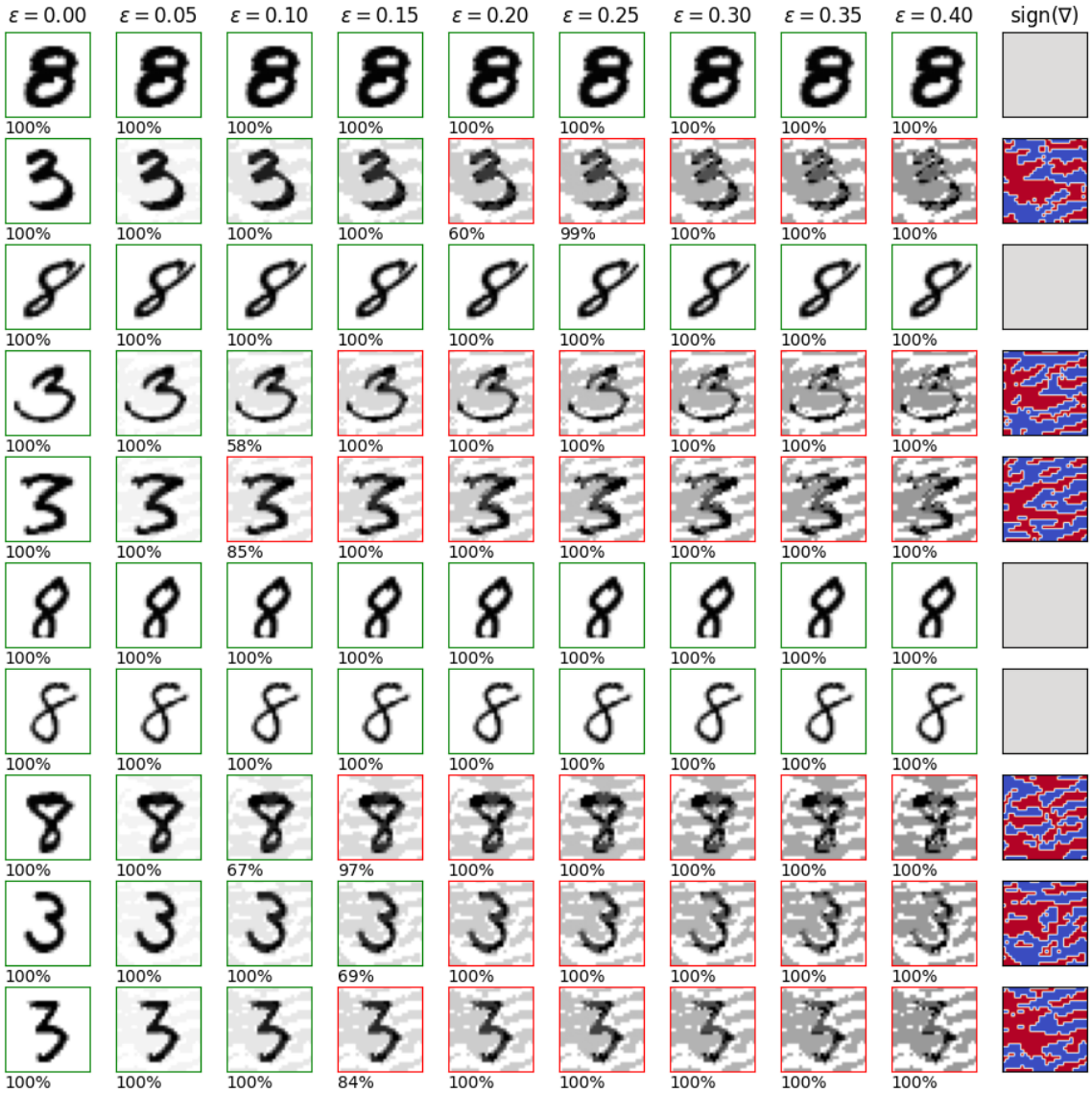


Figure 3.6: MNIST38 adversarial examples for ConvNet-CNN-16

recognizable for humans. An interesting difference we can see between the two methods is that the ConvNet-CNN-16 has higher confidence values even at the border where it is being misclassified. For the ConvNet-GP the confidence tends to decrease more slowly towards the misclassification border and then also increases at a slower rate. The more sharp boundary may be the cause of optimization by gradient descent in contrast to the normal prior on the weights.

Overall we can see a similar performance of the three models ConvNet-GP and its finitely channeled counterparts with 16 and 32 channels for a small ϵ. For attacks with a higher limit ϵ the missing gradients become very visible, with better performances of ConvNet-CNN-16 and ConvNet-CNN-32. Logistic Regression and Lenet5 were the most prone to adversarial attacks.

3.7 FGSM Attack on MNIST

We can run the same experiment as in the previous section for the MNIST dataset with all classes. We apply the fast gradient sign method attack to different models. Since we use the same MNIST test set for the attack, we can compare the results from the models that were trained on the entire MNIST dataset against the reduced MNIST1000. We see the results of the models trained on the reduced dataset MNIST1000 in figure 3.7. We see very clearly that the ConvNet-GP maintains a higher accuracy for larger values of ϵ, then any of the other methods. This becomes especially clear when looking at $\epsilon = 0.1$ and $\epsilon = 0.15$. Here we see a large gap in accuracy between the the Gaussian process and the reference methods. At $\epsilon = 0.1$ ConvNet-GP still has an accuracy above 80% whereas the other methods have dropped below 60% accuracy. Again the accuracy of the logistic regression declines the fastest. Lenet5 is slightly worse then the ConvNet-CNNs. We do not see any converging behaviour for with an increasing of the hidden channel size for the ConvNet-CNNs. The loss functions also indicate the ConvNet-GP being more robust, as the loss stays lower. However, for the loss function we do not see as clear of a gap between the methods as for the accuracy. The area under the receiver operating characteristic also shows that the ConvNet-GP is slightly more robust then the trained convolutional neural networks. Up to $\epsilon = 0.2$ the ROC AUC score is the highest for the ConvNet-GP. For larger values of ϵ it decreases below all other convolutional methods. However, at for $\epsilon > 0.2$ the accuracy is so low, that the ROC AUC metric might not be meaningful anymore. Only a few samples are classified correctly and it might by just by chance that these have relatively high confidence values, which would boost up the ROC AUC score. Additionally with ROC AUC scores lower then 0.5 the confidence of the models is worse then random. We do not see the same increase in ROC AUC that we observed in the binary dataset. As we discussed before this was caused by some samples having a vanishing gradient. For the multiple class dataset we do not observe samples with zero gradients, except for a single one with the ConvNet-GP. The reason for this may be, that the problem itself is harder. With more classes it is more difficult to give a high confidence output. There are more classes where the sample could fall into. For the softmax function to have vanishing gradients a single class must have a confidence while all others are very low.

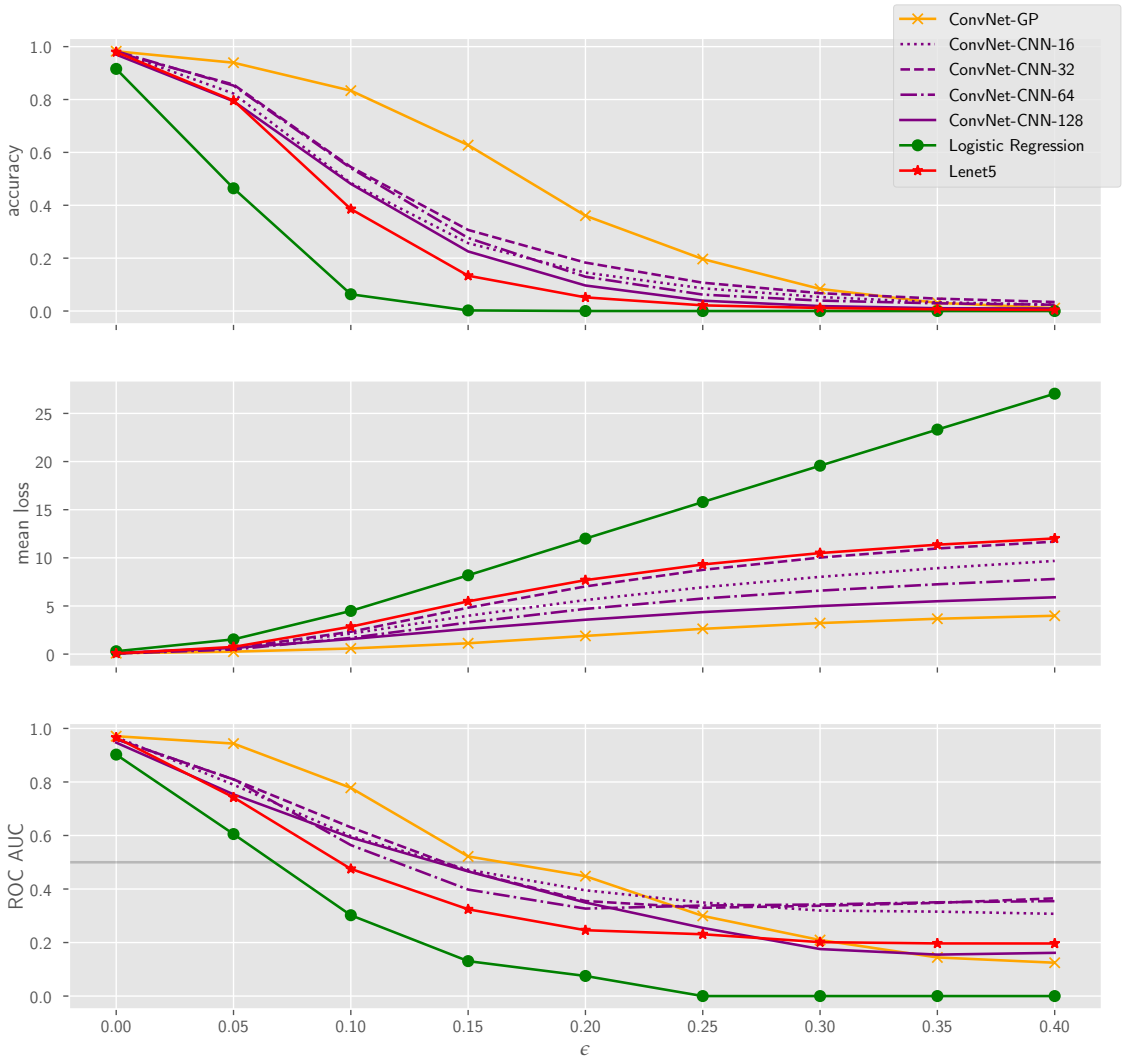


Figure 3.7: Quantitative results of the FGSM adversarial attack. Methods are trained on the MNIST1000 dataset. Attack is performed on the MNIST test set.

Which is definitely less likely with more classes. However we still have a single sample with zero gradients. The reason this was not reflected in the ROC AUC scores as for the binary case is that not all other samples were misclassified. If we compare the accuracy development for the fast gradient sign method attack on the MNIST38 and the MNIST dataset, we observe that for the binary data the accuracy initially stays higher. However, it then falls of steeper and reaches close to zero accuracy much earlier. This seems like a reasonable behavior. More classes mean that samples are more likely to lie close to decision boundaries. Which explains less robustness for a very small ϵ. For more classes however also more distinct features need to be learned, which could make the model more robust for larger values of ϵ.

Overall we have seen the convolutional Gaussian process being slightly more robust towards the fast gradient sign method attack than its finitely channeled counterpart when both are trained on the MNIST1000 dataset. However, as mentioned previously, the only reason for us to use this reduced dataset is the complexity of the Gaussian process. We have already seen in section 3.4.5 that more data increased the test accuracy of the ConvNet-CNN models. To see if this also increases the adversarial robustness we have applied the fast gradient sign method to these models trained on the entire training set. The results are shown in figure 3.8. We observe, that more data tends to make the models more resistant towards the adversarial attacks. The models ConvNet-CNN-16 and 64 maintain a higher accuracy for the same ϵ. Especially the model with 16 hidden channels gains a huge increase in adversarial robustness. However, in comparison to the ConvNet-GP trained on the reduced dataset the conventional convolutional nets still have slightly worse accuracy up until $\epsilon = 0.2$. The gap between the Gaussian process and the CNN however is not at big. For values of $\epsilon > 0.2$ we see that the ConvNet-CNN-16 and 64 have higher accuracies. In the ROC AUC metric we can see that the two CNNs with 16 and 64 benefit from more training data. The ROC AUC is higher then for the reduced training set and the two method even surpass the ROC AUC of the Gaussian process at about $\epsilon \geq 0.15$. The ConvNet-CNN-32 seems to have no change in adversarial robustness, except for the loss, which stays lower. In total we have seen that more training data can increase the adversarial robustness for CNNs.

We will now inspect the adversarial examples. The examples for the ConvNet-GP trained on the MNIST1000 data are shown in figure 3.9. Almost every digit is still recognizable, especially at the lowest ϵ where the attack forced a wrong label. The digit five in the figure slowly morphs into a three. However, for this samples the gaussian process already switched the label at $\epsilon = 0.15$, where it still looks like a five. The adversarial examples in figures 3.10 and 3.11 for the ConvNet-CNN-16 trained on the reduced MNIST1000 and the full MNIST dataset behave similarly. We see a misclassification before the image cannot be mapped back to its original class by humans. For the digits five and eights we can see them slowly changing into a tree and five respectively. So overall we see that the fast gradient sign method attack is able to produce adversarial examples, where the models fail.

We can also combine the visual and the quantitative evaluations. The samples in figures 3.9, 3.10 and 3.11 look like the original image with some added background up until around $\epsilon = 0.2$. For larger values of ϵ some of the lines are thinned out so much, that it becomes harder to distinguish between foreground and background. We could consider this

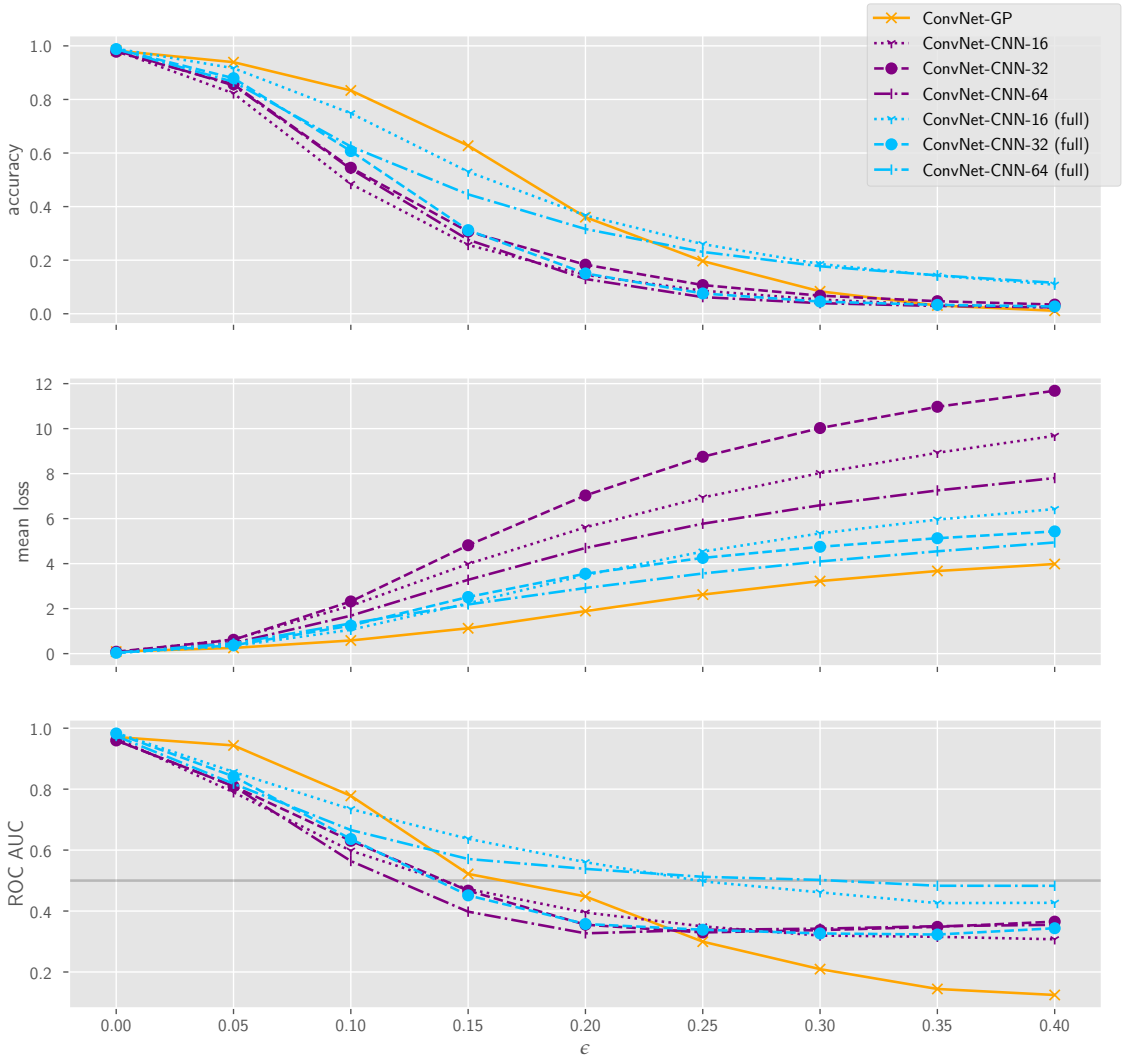


Figure 3.8: Quantitative Results of the FGSM attack. Blue methods are trained on the entire MNIST training data. All other methods are trained on the reduced MNIST1000 data. The attack is perfomed on the entire MNIST test dataset.

as the limiting ϵ on which we evaluate adversarial robustness, as for $\epsilon \leq 0.2$ is seems very likely the image still resembles the original input. Since this is also the range where the ConvNet-GP has the highest accuracy, this indicates that the ConvNet-GP is more robust towards the fast gradient sign method then the other models we considered.

3.8 Runtime Analysis

As we have mentioned previously, we had to reduce the dataset for our experiments because of the ConvNet-GP. In this section we will investigate the complexity of this method, more specifically the time complexity. To evaluate this empirically we ran an experiment. For each model trained on the MNIST1000 dataset we timed how long it takes to evaluate 100 samples. We consider two cases. Firstly we only calculate the predictions of the model with a forward pass. Secondly we calculated the predictions and also the gradient of a loss function by backwards propagation. We repeated every experiment five times and only take the median value. The results are shown in figure 3.12. We can see that the Gaussian process is roughly 1.7 order of magnitude slower then the large ConvNet-CNN-128. Compared to the ConvNet-CNN-16 it is almost tree orders of magnitude slower. This is a huge constraint for using the Gaussian process, when we consider that the ConvNet-CNN-16 has very similar classification results as the ConvNet-GP. We see roughly the same runtime differences for the forward and the backward pass. One could make an argument, that a slower calculation of the gradients makes an adversarial attack more difficult. However this does not make the model more robust. A malicious attacker might not have a time constraint, especially if an attack has severe real world implications.

The reason for the difference between the Gaussian process and the neural networks is caused by the complexity. If we consider a training set with M samples, a evaluation set with N samples and convolutional neural networks with C hidden channels and L layers we can analyse the complexity of the different methods. The Gaussian process requires us to evaluate the covariance for every combination of training and evaluation sample. In total this leads to $\mathcal{O}(NML)$ convolutions we need to calculate. For the conventional CNNs the complexity does not scale with the number of training points. However it does scale quadratic with the number hidden channels, resulting in $\mathcal{O}(NC^2L)$ convolutions needed. One could argue that C is a constant and could be removed from the complexity. Most of the time this is not true. The number of channels has to be chosen according to the data. More datapoints allow for larger models with better results to be trained. So there is some relationship between the number of datapoints and the number of channels. For applications with a lot of datapoints and a small manifold convolutional neural networks are probably better suited. As the number of channels nodes not need to be very high to learn the manifold. For smaller more complex datasets the Gaussian process equivalent of a neural network might be more applicable.

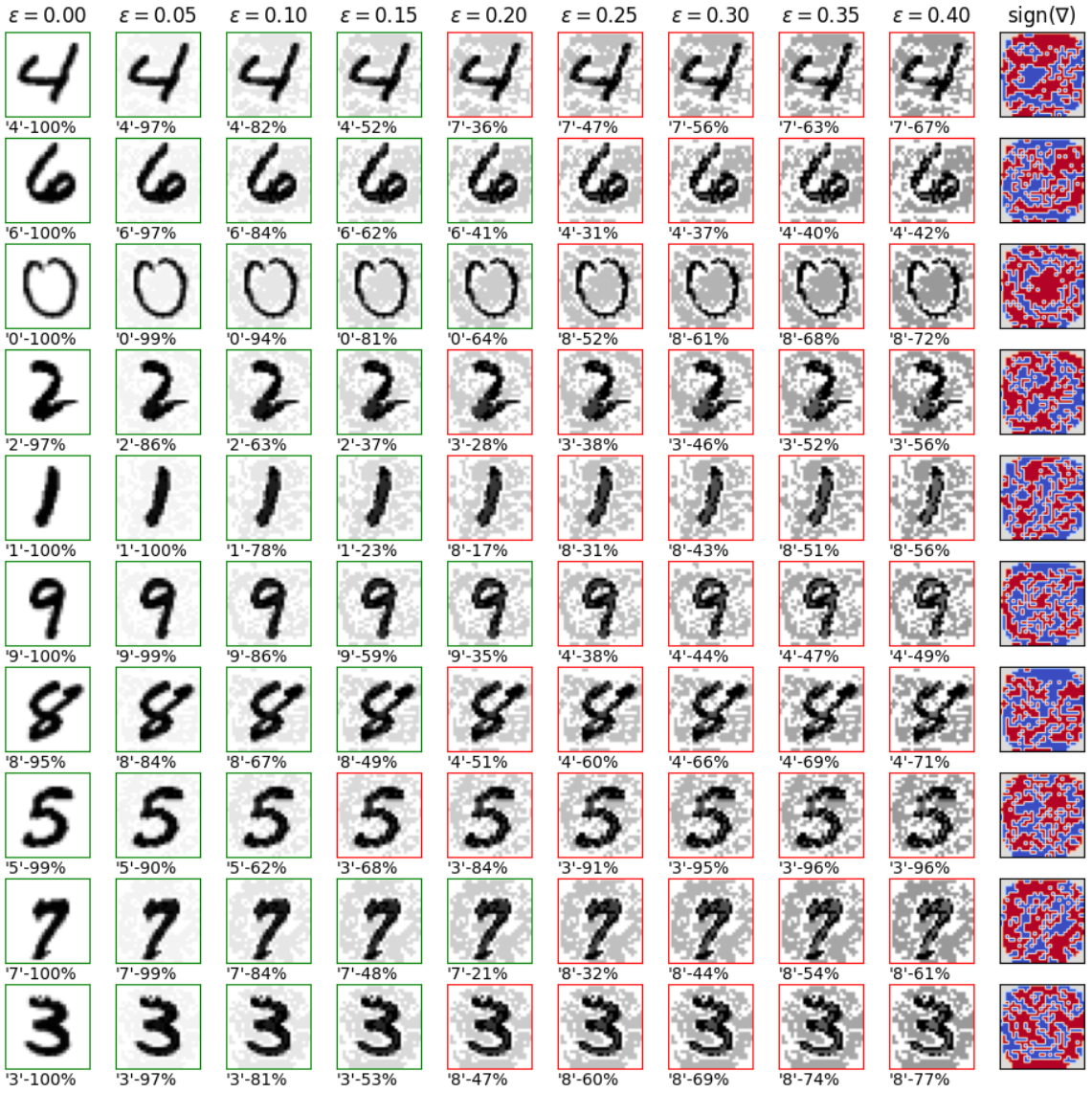


Figure 3.9: Adversarial examples for ConvNet-GP trained on MNIST1000

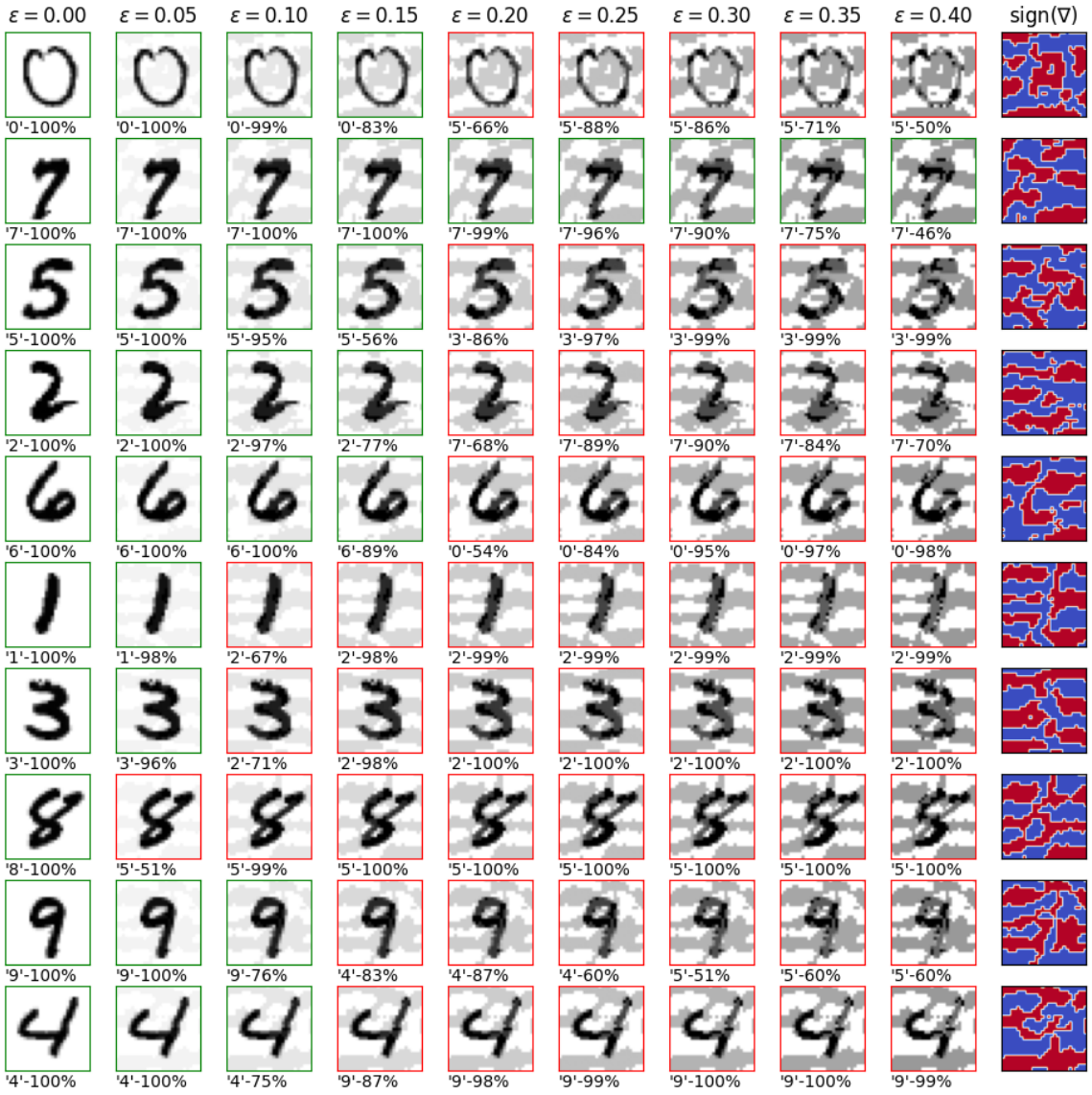


Figure 3.10: Adversarial examples for ConvNet-CNN-16 trained on MNIST1000

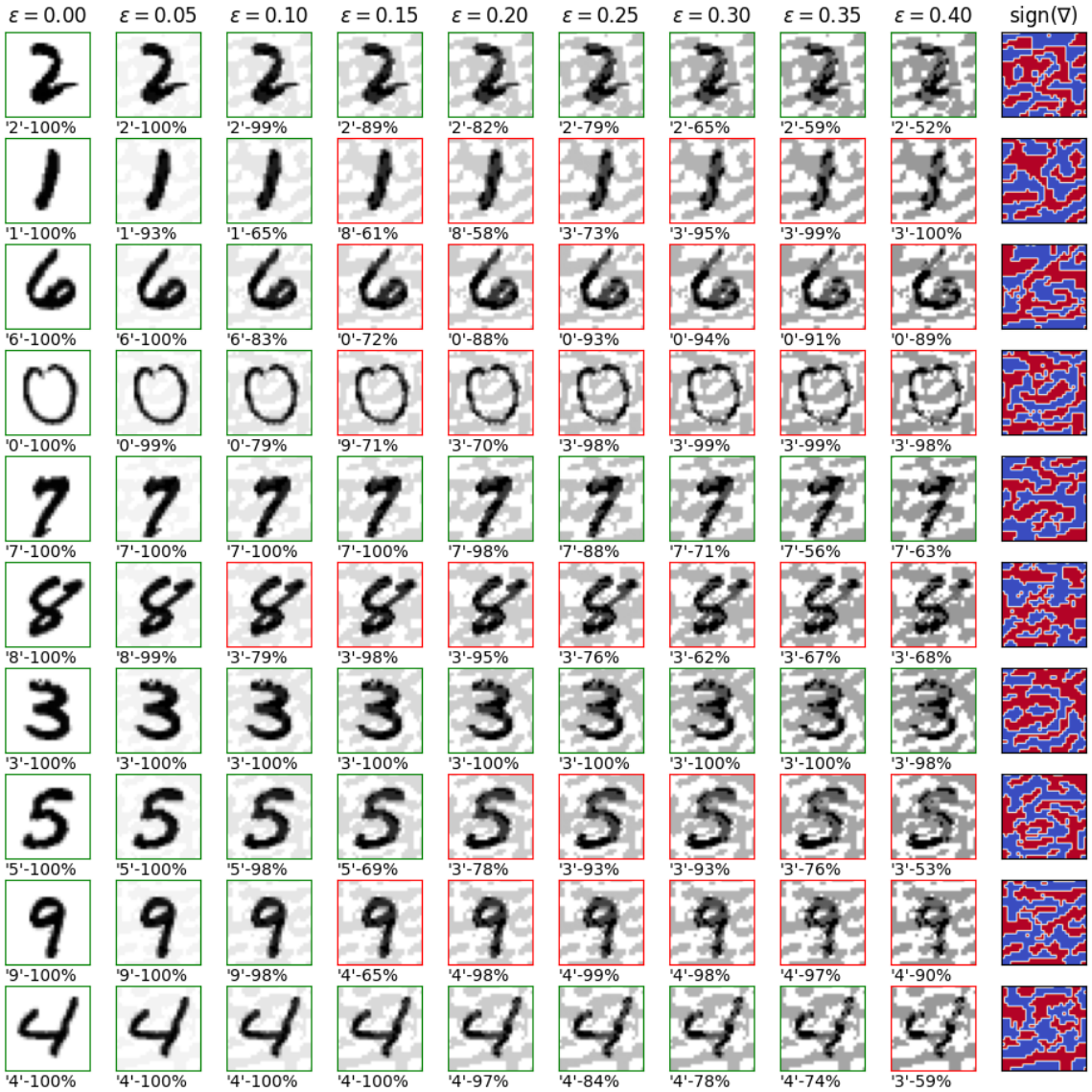
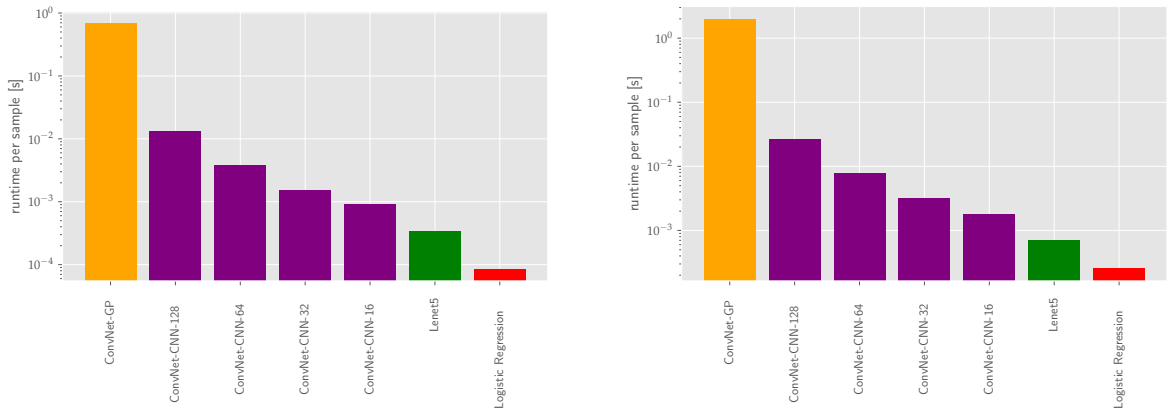


Figure 3.11: Adversarial examples for ConvNet-CNN-16 trained on full MNIST dataset



(a) Forward pass

(b) Forward+Backward pass

Figure 3.12: Runtime comparison of different methods. All methods are trained on the MNIST1000 dataset. Median runtime from five runs per method are shown.

4 Conclusion and Future Work

In this section we will sum up our findings and then discuss future work.

Chapter 2 we showed the equivalence of neural networks and Gaussian processes in the limit case of infinitely wide networks. The Gaussian processes can be used for a classification task. In the same chapter we also presented relevant adversarial attacks.

In chapter 3 we applied an adversarial attack to a Gaussian process derived from a neural network. We discussed how to apply and evaluate the attack. We used the Fast Gradient Sign Method as an adversarial attack. Overall our results on the MNIST dataset show that the Gaussian processes are more robust then conventional neural networks. On the binary MNIST38 dataset we could not confirm the same behaviour entirely. Due to numerical issues in the gradient computation the neural networks were more robust for a larger modifications. The reasoning for the Gaussian process being more robust is probably very complex. We will give some possible causes for this behaviour. Firstly the Gaussian process represents the full posterior of a neural network with distributed weights. Regular neural networks only provide a single point estimate at the networks weights. The posterior distribution will generally carry more information then a MAP estimate. A second reason for more adversarial robustness of the Gaussian process is that the it uses the information of all datapoints at inference time. In contrast a regular neural network will approximate a representative function to fit to the datapoints. This approximation is the only thing used at inference time. The added adversarial robustness of the Gaussian process neural network does not come for free. We have seen that the convolutional Gaussian process is significantly slower then regular CNNs. Furthermore the gap in the adversarial robustness becomes much closer if the CNNs is trained on more data. This can be done easily, since the complexity for evaluating does not depend directly on the size of the training dataset. For applications the choice between a conventional CNN and the Gaussian process version will depend on a variety of factors. The dataset size, time constraints and expected adversarial robustness are relevant aspects for this decision.

A point we have not discussed is how to increase the adversarial robustness of a model. For neural networks there are some methods to do this. A typical way of doing this is by data augmentation. In [19] adversarial examples were generated during training and also included in the loss function to increase the robustness. The authors also showed that adding random noise to the samples while training increased the robustness. Since these methods add more datapoints to the model, they are not directly suited for Gaussian processes.

To be make use of these methods the Gaussian process has to be more efficient. There are approaches like sparse Gaussian processes and methods using inducing points that scale less with an increasing training size. Using such an approach to include adversarial examples into the training set would be an interesting research.

In our work we have only focused on the fast gradient sign method applied to models

trained on the MNIST dataset. There is still a lot of room for exploration of different attacks and other datasets. Especially a problem like spam recognition in the natural language domain would be interesting. Maneuvering around spam detectors is basically a real world adversarial attack. Finding models with a good adversarial robustness is essential for this problem.

It would also be very interesting to examine the cause of the adversarial robustness in more detail. For example by comparing the robustness of Gaussian process neural networks to other neural network approaches that simulate the posterior of a Bayesian neural network (e.g. bagging or monte carlo dropout). This would give deep insights if using posterior distributions is more robust then point estimates.

List of Figures

List of Tables

Bibliography

[1] Y. Zoabi, S. Deri-Rozov, and N. Shomron. "Machine learning-based prediction of COVID-19 diagnosis based on symptoms". In: *npj digital medicine* 4.1 (2021), pp. 1–5.

[2] J. Su, D. V. Vargas, and K. Sakurai. "One Pixel Attack for Fooling Deep Neural Networks". In: *IEEE Transactions on Evolutionary Computation* 23.5 (Oct. 2019), pp. 828–841. ISSN: 1941-0026. DOI: 10.1109/tevc.2019.2890858. URL: http://dx.doi.org/10.1109/TEVC.2019.2890858.

[3] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. *Intriguing properties of neural networks*. 2014. arXiv: 1312.6199 [cs.CV].

[4] Y. Gal and Z. Ghahramani. *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning*. 2016. arXiv: 1506.02142 [stat.ML].

[5] R. M. Neal. *Bayesian learning for neural networks*. Vol. 118. Springer Science & Business Media, 2012.

[6] J. Lee, Y. Bahri, R. Novak, S. S. Schoenholz, J. Pennington, and J. Sohl-Dickstein. *Deep Neural Networks as Gaussian Processes*. 2018. arXiv: 1711.00165 [stat.ML].

[7] M. van der Wilk, C. E. Rasmussen, and J. Hensman. *Convolutional Gaussian Processes*. 2017. arXiv: 1709.01894 [stat.ML].

[8] C. Rasmussen, C. Williams, M. Press, F. Bach, and P. (Firm). *Gaussian Processes for Machine Learning*. Adaptive computation and machine learning. MIT Press, 2006. ISBN: 9780262182539. URL: https://books.google.de/books?id=Tr34DwAAQBAJ.

[9] A. Garriga-Alonso, C. E. Rasmussen, and L. Aitchison. *Deep Convolutional Networks as shallow Gaussian Processes*. 2019. arXiv: 1808.05587 [stat.ML].

[10] Y. Cho and L. Saul. "Kernel Methods for Deep Learning". In: *Advances in Neural Information Processing Systems*. Ed. by Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta. Vol. 22. Curran Associates, Inc., 2009. URL: https://proceedings.neurips.cc/paper/2009/file/5751ec3e9a4feab575962e78e006250d-Paper.pdf.

[11] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[12] C. K. Williams and D. Barber. "Bayesian Classification With Gaussian Processes". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 20 (1998), pp. 1342–1351.

[13] T. P. Minka. "A family of algorithms for approximate Bayesian inference". PhD thesis. Massachusetts Institute of Technology, 2001.

[14] C. Yang, Q. Wu, H. Li, and Y. Chen. "Generative poisoning attack method against neural networks". In: *arXiv preprint arXiv:1703.01340* (2017).

[15] T. J. L. Tan and R. Shokri. "Bypassing backdoor detection algorithms in deep learning". In: *arXiv preprint arXiv:1905.13409* (2019).

[16] S. Gambs, A. Gmati, and M. Hurfin. "Reconstruction attack through classifier analysis". In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2012, pp. 274–281.

[17] J. R. Correia-Silva, R. F. Berriel, C. Badue, A. F. de Souza, and T. Oliveira-Santos. "Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2018, pp. 1–8.

[18] J. Wu and R. Fu. "Universal, transferable and targeted adversarial attacks". In: *arXiv preprint arXiv:1908.11332* (2019).

[19] I. J. Goodfellow, J. Shlens, and C. Szegedy. *Explaining and Harnessing Adversarial Examples*. 2015. arXiv: 1412.6572 [stat.ML].

[20] A. Kurakin, I. Goodfellow, and S. Bengio. *Adversarial examples in the physical world*. 2017. arXiv: 1607.02533 [cs.CV].

[21] K. Grosse, D. Pfaff, M. T. Smith, and M. Backes. *The Limitations of Model Uncertainty in Adversarial Settings*. 2019. arXiv: 1812.02606 [cs.CR].

[22] Y. LeCun and C. Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[24] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[26] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[27] M.-I. Nicolae, M. Sinn, M. N. Tran, B. Buesser, A. Rawat, M. Wistuba, V. Zantedeschi, N. Baracaldo, B. Chen, H. Ludwig, I. Molloy, and B. Edwards. "Adversarial Robustness Toolbox v1.2.0". In: *CoRR* 1807.01069 (2018). URL: https://arxiv.org/pdf/1807.01069.

[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[29] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[30] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[31] T. Fawcett. "An introduction to ROC analysis". In: *Pattern recognition letters* 27.8 (2006), pp. 861–874.