DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Efficient Implementation of Deep Convolutional Gaussian Processes

Martin Meinel

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics

Efficient Implementation of Deep Convolutional Gaussian Processes

Effiziente Implementierung von Deep Convolutional Gaussian Processess

Author:	Martin Meinel
Supervisor:	Dr. Felix Dietrich
Advisor:	Prof. Dr. Christian Mendl
Submission Date:	15.10.2020

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, 15.10.2020 Martin Meinel

Abstract

Convolutional Neural Networks are used to obtain state of the art results in the field of image classification. However, they also come with certain drawbacks, such as their vulnerability to adversarial attacks and lacking uncertainty estimation.
These drawbacks can be tackled by making use of Bayesian inference, which is the main concept underlying Gaussian Processes for making predictions. A Gaussian Process is a distribution over functions defined by its mean and its covariance or kernel function. Deep Convolutional Networks can be represented as Gaussian Processes in the limit where the number of convolutional filters approaches infinity. The disadvantage of making predictions with Gaussian Processes is the big computational effort coming with each sample of the training set, because the kernel matrix has to be extended for each new sample. Additionally, this large kernel matrix for the training data has to be inverted.

In this thesis, it is explained in detail what a Gaussian Process is and how it can be used for Regression. Furthermore, a Gaussian Process is presented that is equivalent to a convolutional neural network in the limit of infinitely many filters.

To construct the kernel matrix from given training data, Iterative SVD, Soft Impute, Matrix Factorization and the Nyström method are investigated to increase the efficiency of Gaussian Processes by exploiting the low-rank structure of the kernel matrix.
Finally, it can be seen that it is sufficient to compute only 20% of the kernel matrix exactly and approximate the missing elements of it by making use of Iterative SVD or the Nyström method. By that the efficiency for large training sets can be improved significantly and the GP model obtains still good classification results.

Kurzfassung

Durch die Verwendung von Convolutional Neural Networks (Faltungsnetzwerke) ist es möglich bei der Klassifizierung von Bildern Spitzenergebnisse zu erzielen. Sie weisen jedoch auch einige Nachteile auf, wie beispielsweise ihre Anfälligkeit gegenüber feindlichen Angriffen und ihre fehlende Einschätzung der Unsicherheit.
Diese Nachteile können mithilfe der Bayesschen Inferenz behoben werden, die in den Gaußschen Prozessen als wichtigstes Konzept zum Erstellen von Vorhersagen verwendet wird. Ein Gaußscher Prozess ist eine Verteilung über Funktionen und ist durch einen Mittelwert und eine Kovarianz oder Kernelfunktion definiert. Deep Convolutional Network können als Gaußsche Prozesse innerhalb des Grenzwertes dargestellt werden, bei dem die Anzahl der Faltungsfilter gegen unendlich geht. Der Nachteil bei der Erstellung von Vorhersagen mithilfe Gaußscher Prozesse ist der große Rechenaufwand, der für jedes Sample des Trainingssatzes benötigt wird, weil die Kernelmatrix für jedes neue Sample erweitert werden muss. Außerdem muss diese große Kernel-Matrix für die Trainingsdaten invertiert werden.

Diese Masterarbeit erläutert genauer, was ein Gaußscher Prozess ist und wie er für Regression verwendet werden kann. Außerdem wird ein Gaußscher Prozess dargestellt, der im Grenzwert unendliche vieler Filter einem Convolutional Neural Network entspricht.

Zur Erstellung der Kernel-Matrix aus vorhandenen Trainingsdaten, werden Iterative SVD, Soft Impute, Matrix Factorization und die Nyström-Methode untersucht, um durch die Nutzung der niedrigrangigen Struktur der Kernelmatrix die Effizienz von Gaußschen Prozessen zu verbessern.
Die Schlussfolgerung lässt erkennen, dass es ausreichend ist, nur 20% der Kernelmatrix zu berechnen und den restlichen Teil mithilfe von Iterative SV oder der Nyström-Methode anzunähern. Dies führt zu einer Verbesserung der Effizienz bei dennoch guter Klassifikationsergebnissen des GP-Modells.

Contents

1. Introduction

Images are a central part of everyone's life nowadays. This can be seen from the fact that everyone has a smartphone capturing important or beautiful moments with their camera. Besides, facial recognition is often used in new cell phones for the unlock mechanism.
Pictures play an important role not only in daily life, but also in industry and science they have a central role. Doctors investigate x-ray images of patients to see if they have a broken bone. Another example are chest x-rays which are used to detect lung cancer.
With the progress in Machine Learning also Machine Learning models are used more and more to process image data. They are crucial for autonomous driving where they are responsible for detecting and classifying traffic signs. Another field is the medical domain, where they support doctors which check a huge number of scans every day to detect suspicious characteristics of diseases on these scans. One example for that is the application of these models for vessels extraction of the retina, where some diseases can be detected. Another current example is the usage of Convolutional Neural Networks (CNNs) to diagnose if a patient is suffering from Covid-19 using chest x-ray images [1].

Convolutional Neural Networks are used to achieve state of the art results for image data. As with any neural network the parameters of a CNN are optimized by minimizing the loss function and applying backpropagation to update the parameters stepwise. This boils down to be the most time consuming part in training a CNN. Another drawback of CNNs other than long training time is the vulnerability to adversarial attacks. An alternative to Convolutional Neural Networks are Gaussian Processes, which are distributions over functions and can also be used for regression and classification tasks. They make use of Bayesian Inference and can be less vulnerable to adversarial attacks if the additional uncertainty information is taken into account.

In chapter 2 Convolutional Neural Networks as well as their drawbacks are presented in detail. Furthermore, Gaussian Processes are introduced and explained regarding their use for regression and classification tasks. The equivalence of Gaussian Processes which approach an infinite number of convolutional filters with Convolutianal Neural Network is shown in the last section 2.3 of chapter 2.

Gaussian Processes are non-parametric models and therefore the training is different to Convolutional Neural Networks. This does not mean that GPs are efficient to train, because the covariance matrix of the GP has to be computed between every pair of training samples. There are many metrics which are used to evaluate Machine Learning models. Chapter 3 contains an introduction of some metrics which are used in this thesis to classify images of the

MNIST data set. Besides Iterative SVD, Soft Impute, Matrix Factorization and the Nyström method are investigated with the goal to improve the efficiency of Gaussian Processes and consequently reduce the time to obtain the mentioned covariance matrix. Several experiments show that Iterative SVD and the Nyström method achieve the best compromise of increasing the efficiency of Gaussian Processes and still obtaining good classification results. The presented metrics are used to show that computing only 20% of the covariance matrix is already sufficient to obtain good classification results and save significant time for large training data sets. Finally, in the last section 3.5 of chapter 3, several tables provide an overview of how the performance and the time differs between the original Gaussian Process and the Gaussian Process using only 20% of the exactly computed covariance matrix and Iterative SVD or the Nyström method respectively.

Finally, the last chapter 4 contains a summary of the results of this thesis. Furthermore, some thoughts are presented about how to improve the presented techniques.

2. State of the art methods for image classification

Image classification is a big part of Machine Learning and Deep Learning research. There are many real life applications making use of Machine Learning models used for image classification. One famous domain is medical imaging, where x-ray pictures of patients are processed through Machine Learning models to detect cancer. Another current example is the application of CNNs to detect COVID-19 on chest X-ray images. This is helpful since it is essential to identify patients very early and it is one of the fastest methods to test people [1]. As a consequence of the big importance of image classification models in real life, this topic is also very present in the research domain. This can be seen from the various existing labelled data sets such as MNIST [2], Cifar-10 [3] or ImageNet [4]. The MNIST data set is presented more in detail in chapter 3, because this thesis uses it for all presented experiments.
An overview over state of the art methods for image classification is given in this chapter. Convolutional Neural Networks are presented in section 2.1. They are used as state of the art models for image classification nowadays. Furthermore, a description of how Gaussian Processes are used for image classification and which advantages they provide in comparison to Convolutional Neural Networks is provided. This also explains why Gaussian Processes are a big topic of current research. Finally, the last section (sec. 2.3) contains a brief explanation how CNNs can also be represented as GPs in the limit of infinitely many convolutional filters.

2.1. Convolutional Neural Networks for image classification

Convolutional Neural Networks are a popular choice for classifying images. LeCun, Haffner, Bottou, and Bengio presented in 1999 in their paper "Object recognition with gradient-based learning" CNNs as a better alternative to the former most used multi-layer perceptron networks [5]. Images have certain characteristics which can be used to distinguish them from other input data. They are not suitable for feeding them directly into fully connected networks. One reason for this is that images possess many pixels, each containing at least one (gray-scale images) to many (RGB images) channels, resulting in a high ambient space dimension in general. Another characteristic is that images can be rotated, translated or augmented in a certain way, and humans still have no problems classifying them. This is also targeted to be learned by Machine Learning models [5]. Another characteristic of images is their clear local structure so that pixels that are close to each other are often highly correlated. For example neighbouring pixels often have a very similar color value [6].
All these characteristics lead to the fact that fully connected neural networks handle pictures poorly. In addition, already a reasonable amount of neurons would lead to a huge number

of parameters which have to be trained. This requires a large data set and leads to long training times. For processing an image with hundreds of pixels with only one value per pixel (assuming it is a gray-scale image) and a number of hundred neurons in the first layer the network would already have to train several ten thousand weights even though there is only one layer. This example illustrates why fully connected networks are hard to train on images.
The main drawback of fully connected networks is that they are not invariant to scaled, translated or distorted pictures. Fully connected layer neural networks might learn the same features in different neurons in case of training with augmented pictures. So there are redundant neurons at different positions of one layer learning the same features.
Besides, these networks cannot grasp the local structure within an image, because the order of the pixel values does not affect the output. This is an important drawback, since images have a clear topology with correlating pixel values for local neighborhoods as mentioned above. This characteristic can be used to first detect low level local features (horizontal and vertical lines) before combining them to detect more advanced features (corners) and spatial or temporal objects in the end.
These reasons show why a network consisting only of fully-connected layers is not suited for image classification. Convolutional Neural Networks follow a certain architecture to be able tackle these mentioned issues and this is why they are used to build state of the art models for image classification nowadays [5].

Convolutional Neural Networks are characterized by three architectural principles: local receptive fields, shared weights and spatial sub-sampling. These ideas are used to ensure shift, scale and distortion invariance, but also to decrease the amount of training images and training time. Furthermore these characteristics provide more structure for a CNN than a fully connected network has. One main idea is to extract local elementary features such as corners and end-points in small local neighborhoods, and this extraction is typically performed by convolution with a so-called "kernel" (not to be confused with the kernel used by Gaussian Processes, introduced in section 2.2). This kernel is usually smaller in width and depth than the original input image, but has the same number of channels. It is step-wise convolved over the image and generates an output for every local neighborhood. A convolution can be seen as a dot product between the kernel and the input image. All values of the input image are multiplied by their corresponding value of the kernel. Finally all products are summed up as in the normal dot product to obtain a scalar as output. The outputs of the application of one kernel is a so called feature map and is either smaller or the same size as the input data. There are several kernels convolved over the input image every layer, since every kernel creates one feature map, there are several feature maps for the next layer.
One kernel has the same fixed parameters all over the feature map over which it is convolved. This characteristic is called weight-sharing and leads to the fact that less parameters have to be optimized in comparison to a fully connected network. The result of that is that one kernel extracts only one specific type of feature such as corners for example. Another advantage of this kernel is that since it is applied all over the feature map it extracts the same features also for shifted images and is therefore invariant to translated pictures. At the first layers

low-level features such as corners are detected and extracted. The layers afterwards combine the already extracted features and therefore can extract high-level features [5].

In this section formal mathematical expressions for Convolutional Neural Networks are provided, which are used through the whole thesis. Matrices are capitalized while vectors are represented as lowercase letters. The input image which is fed into the CNN is represented as X with height $H^{(0)}$, width $D^{(0)}$ and channels $C^{(0)}$. All in all the matrix has a size of $C^{(0)} \times (H^{(0)} \times D^{(0)})$. Each channel is flattened to a one-dimensional array x_1, x_2 to $x_c^{(0)}$, where a gray-scale image has in general only one channel, while a RGB image has three channels. Figure 2.1 shows the vector x_j resulting from the flattened jth channel of the input image. The first layer linearly transforms the input and generates $C^{(1)}$ feature maps $a_i^1(X)$ which are processed in the next layer. For $i \in 1, ..., C^{(1)}$:

$$a_i^{(1)}(X) := b_i^{(1)} 1 + \sum_{j=1}^{C^{(0)}} W_{i,j}^{(1)} x_j. \tag{2.1}$$

It is important to notice that all parameters $\mathcal{W}$ belong to one layer and are denoted by the corresponding number of the layer: $\mathcal{W} = \{W^{(l)}, b^{(l)}\}_{l=1}^{L+1}$ [7]. Figure 2.1 illustrates formula 2.1 for one dimension of an input image. The parameters $U_{i,j}^{(0)}$ and $W_{i,j}^{(0)}$ of this figure are denoted in the formulas such as in equation 2.1 as $W_{i,j}^{(1)}$.
The filter $U_{i,j}^0$ is a 2×2 kernel and therefore can be convolved over the jth channel four times in total, where it is moved from the upper left corner to the bottom right corner. Therefore, the kernel $U_{i,j}^{(0)}$ can be transformed into a matrix $W_{i,j}^{(0)}$ with as many rows as patches and filled with zeros at the empty positions, where the filter does not apply. So the 2D convolution $U_{i,j}^{(0)} * x_j$ can be expressed as the dot product $W_{i,j}^{(0)} x_j$, where the μth row of $W_{i,j}^{(0)}$ corresponds to applying the kernel at one position of the input image and generates the μth convolutional patch of x_j [7].

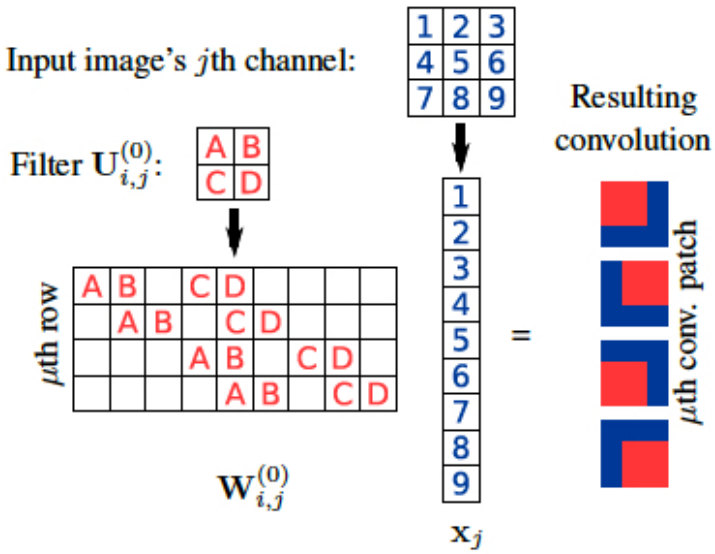


Figure 2.1.: Exemplary convolution of a kernel $U_{i,j}^{(0)}$ over one channel of an input image x_j[7].

A convolution can be expressed as a dot product if the convolution is only over one channel. In general images have often more than one channel and after the first layer convolutions are applied on the generated feature maps, so that convolutions are applied over several channels. So the results of the dot products have to be summed up over all the channels to obtain the result of a convolution over several channels. This is what is done in formula 2.1. There is also a bias term added in the end.
The first layer operates on the input image, while the layers afterwards are applied on the created feature maps of the previous layer. The first layer applies only linear transformations, while the other layers also apply non-linear transformations. So all feature maps after the first layer are defined as follows:

$$a_i^{(l+1)}(X) := b_i^{(l+1)}1 + \sum_{j=1}^{C^{(l)}} W_{i,j}^{(l+1)} \phi(a_j^{(l)}(X)). \tag{2.2}$$

The feature maps $a_i^{(l)}(x)$ are combined into feature maps $A^{(l)}(X)$ of shape $C^{(l)} \times (H^{(l)}D^{(l)})$, where each row a_i is the flattened vector of the jth channel of the image, which is the generated by the convolution of $\phi(A^{(l)}(X)$. The indices i and j are the number of feature maps of the next layer and the depth of the input image of the current layer. Formally $i \in 1, ..., C^{(l+1)}$ and $j \in 1, ..., C^{(l)}$.
The outputs of the Convolutional Neural Network correspond to the last feature maps $A^{(L+1)}(X)$ and have a width and height of one ($H^{(L+1)} = D^{(L+1)} = 1$) for regression or classification problems. The number of channels $C^{(L+1)}$ corresponds to the number of classes in the data set. Therefore, the weight matrix $W_{i,j}^{(L+1)}$ containing the kernel also has only one row and the generated activations $a_i^{(L+1)}$ are single element vectors. The initial parameters of

the CNN namely the weights and biases are sampled from Gaussian priors defined as follows

$$\begin{aligned} U_{i,j,x,y}^{(l)} &\sim \mathcal{N}\left(0, \frac{\sigma_w^2}{C^{(l)}}\right), \\ b_i^{(l)} &\sim \mathcal{N}\left(0, \sigma_b^2\right). \end{aligned} \tag{2.3}$$

It is important to understand this chapter and the notation of the convolutional network, since the section 2.3 builds up on that in order to show that there is a Gaussian Process equivalent to the the presented CNN [7].

2.2. Gaussian Processes for image classification

CNNs build the foundation of state of the art Deep Learning models for image classification, but Gaussian Processes start to get a lot of attention in the research domain currently. The reason for that is that CNNs are vulnerable to adversarial attacks because they do not provide any uncertainty estimation about their provided results in the output [7].
Adversarial attacks are attacks on Machine Learning models where a slightly, sometimes not even for humans perceptible perturbation of the input data, a so called adversarial noise, leads to a wrong classification of the model. This can result into dangerous scenarios because Deep Learning systems are also applied in safety critical domains such as autonomous driving [8]. A famous example illustrating this risk is shown in figure 2.2, where on the left an original stop sign is shown which is polluted by some graffiti. On the right you can see a perturbed stop sign from [8] which created that adversarial example, which is misclassified as speed limit 45 sign in most of the cases by two different CNNs [8].

Figure 2.2.: On the left a real stop sign is shown which is covered by some graffiti. On the right you can see a perturbed image from [8], which was created to be misclassified by two different CNN classifiers [8].

Gaussian Processes are tackling the problem of adversarial attacks by making use of Bayesian Inference and therefore providing an uncertainty estimate for every classification result corresponding to an approximation how much the user can trust the result. Therefore, Gaussian Processes are less vulnerable to adversarial attacks. For example Bradshaw, Matthews, and Ghahramani show that Gaussian Processes are less susceptible to Fast Gradient Sign Method attacks [9]. The following sections present the foundations of Gaussian Processes and how they are applied to regression and classification tasks.

2.2.1. Introduction to Gaussian Processes

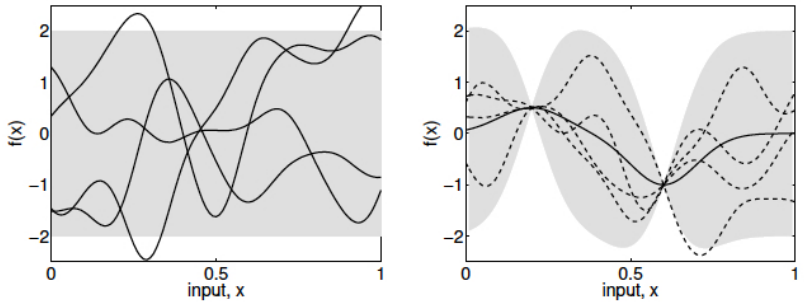
Definition 1. Gaussian Process "A Gaussian process is a generalization of the Gaussian probability distribution. Whereras a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a stochastic process governs the properties of functions."[10]

A Gaussian process is a distribution over functions specified by a mean function and a covariance function, determining the properties of the functions belonging to the distribution. All supervised Machine Learning tasks aim to find the function which describes best the given training data and also generalizes well to new unseen data. Shortly, the goal is to find the best function fitting the data. This task can be approached by choosing as a function the mean of a distribution over functions where a minimum prior probability is assigned to each function. Additionally, there are some higher probabilities assigned to functions seeming more likely, because of certain properties. One example for that is preferring smoother functions than less smooth ones. Gaussian processes are used to implement this approach, because they allow computing with infinite infinite-dimensional spaces of possible functions.
The following paragraph explains it in a less formal way. A function can be seen as a vector containing different function values evaluated for different points x. Since functions can be evaluated at an infinite number of points of the input domain these vectors are not finite. The advantage of using Gaussian processes is that taking only the properties of a function at a finite number of points into account and doing inference gives the same result as taking all points into account. This is how Gaussian processes make it computationally tractable to compute with functions.

Introduction to Bayesian modeling with Gaussian processes

Bayesian modeling with distributions over functions for a regression problem is explained more in detail in this section. Figure 2.3 a) presents four functions which are sampled from a prior distribution. This distribution is determined by a Gaussian process assigning more likelihood to smooth functions than to non-smooth ones. This prior distribution represents certain assumptions the user has met about the data before having observed any data points. The Gaussian process of a) is chosen with a mean of zero for every x of the input domain, even though that cannot be seen from the four drawn functions, but it would be visible by

drawing more functions. The shaded region shows the two times standard deviation for every data point. It can be seen that the deviation is constant over the input domain.



(a), prior (b), posterior

Figure 2.3.: a) Showing four functions drawn from a prior distribution. b) Showing four functions from posterior distribution after two data points have been observed with dashed lines. The solid line shows the mean prediction. Shaded region marks two times the standard deviation for each x [10].

Given a data set of two data points $\mathcal{D} = \{(x_1, y_1), (x_2, y_2)\}$, all functions which are not matching these data points are discarded. This scenario is shown in figure 2.3 b) where four exemplary functions going through both data points of $\mathcal{D}$ are represented with dashed lines. The solid line illustrates the mean value of these functions. The variance corresponding to the uncertainty marked by the grey area remarkably decreases around the given data points, since all functions of the distribution are consistent with these points. All in all the posterior, shown in 2.3 b) is obtained from the prior distribution of 2.3 a) and the observation of some data points. Adding data points to the set $\mathcal{D}$ leads to the fact that the mean function adjusts to pass through these points. Besides the uncertainty of the posterior also decreases around the observed data.
A Gaussian process is in contrast to neural networks a non-parametric model. This is why it is always possible to fit the model to the data, which is for example not the case for a linear regression model when it is used for non-linear data. The choice of the prior is important, because it specifies the functions which are used for inference. As already mentioned a Gaussian process is specified by its mean and its covariance function. The covariance function does not only determine the variance between different data points, it also specifies other properties of the distribution. This is why choosing a certain covariance function also specifies certain properties of the functions belonging to the corresponding Gaussian process. For example the smoothness and stationarity of the functions of a distribution can be determined by the covariance function. In Machine Learning the parameters of the model the are learned first and then used for inference. In the context of Gaussian Processes learning means to find suitable properties of the covariance function, which is usually performed through hyper-parameter optimization [10].

2.2.2. Gaussian Process Regression

As the previous section contains the introduction to the ideas about Gaussian Processes, the mathematical and explanation about how to do inference for Gaussian Process Regression is provided in this section.
There exist two different views on inference with Gaussian process regression. This thesis only presents the function view and not the weight space view, because both approaches end up with the same results, but the weight space view can be studied in [10].
Rasmussen and Williams also provide a second definition for Gaussian Processes.

Definition 2. "A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution." [10]

Random variables in this case represent function values $f(x)$ evaluated for some x of the input domain $\mathcal{X}$, which is in $\mathbb{R}^D$ for example.
A Gaussian process is defined by its mean function $m(x)$ and covariance or kernel function $k(x, x')$. Therefore, functions from a defined GP can be drawn as follows, where the GP might denote the prior of a GP regression model for example.

$$f(x) \sim GP(m(x), k(x, x')) \tag{2.4}$$

Any real process sampled from the Gaussian process in equation 2.4 is consistent with the mean and covariance function:

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)], \\ k(x, x') &= \mathbb{E}[f(x) - m(x))(f(x') - m(x'))^T]. \end{aligned} \tag{2.5}$$

In most cases the mean function is set to a constant, often zero:

$$m(x) = 0.$$

The reason for that is that it simplifies computations and notations with Gaussian Processes, which is shown later on in this section. Besides, Gaussian Processes are flexible enough to model the posterior mean arbitrary well, even if the prior mean is zero. The covariance function k, also called kernel, has to be positive definite, because the covariance between two data points is defined to be positive definite. Referring to definition 2 any finite set of points defines a joint Gaussian:

$$p(f|X) = \mathcal{N}(f|\mu, K), \tag{2.6}$$

where K denotes the kernel matrix containing the kernel function evaluated between every pair of points such that $K_{i,j} = k(x_i, x_j)$. Besides μ defines a vector containing the prior mean function applied to each data point: $\mu = (m(x_1), m(x_2), \ldots, m(x_N))$ [11].

Exemplary Gaussian Process

An example for a Gaussian process is the Bayesian linear regression model

$$f(x) = \phi(x)^T w,$$

with the following prior on the weights $w \sim \mathcal{N}(0, \Sigma_p)$. For this Gaussian process the mean and covariance are defined as follows:

$$\begin{aligned} \mathbb{E}[f(x)] &= \phi(x)^T \mathbb{E}[w] = 0, \\ \mathbb{E}[f(x)f(x')] &= \phi(x)^T \mathbb{E}[ww^T]\phi(x')^T = \phi(x)^T \Sigma_p \phi(x'). \end{aligned} \tag{2.7}$$

This shows that $f(x)$ and $f(x')$ are jointly Gaussian with a mean of zero and a covariance of $\phi(x)^T \Sigma_p \phi(x')$. More in general all function values $f(x_1), \ldots, f(x_n)$ are jointly Gaussian for any n number of input points.
A very popular covariance function is the Gaussian kernel function which is given by

$$\mathbb{C}(f(x_p), f(x_q)) = k(x_p, x_q) = \exp\left(-\frac{1}{2}|x_p - x_q|^2\right). \tag{2.8}$$

The important point shown by that function is that the covariance of the outputs can be computed by the inputs. This covariance function measures how close the two input points are with respect to each other. So in case both x_q and x_p are close to each other the exponent will be small and therefore the covariance will be close to one, while it will decrease the more different they are [10].

Mathematical foundations for multivariate joint Gaussian distributions

In this section some foundations regarding jointly Gaussians are provided, which are important for doing inference in the next section. It can be seen that for a jointly Gaussian the marginals as well as the posterior conditionals are Gaussian as well. For $x = (x_1, x_2)$ being a jointly multivariate Gaussian with the following parameters:

$$\begin{aligned} \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \\ p(x_1, x_2) = \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right), \end{aligned} \tag{2.9}$$

where μ_1 is the mean of x_1 and Σ_{11} is the covariance matrix of x_1. The covariance matrix Σ_{12} is just the Transpose of matrix Σ_{21} and is the covariance matrix between both Gaussian jointly random variables.
Then the marginal property of the joint Gaussian defines the marginals as follows:

$$\begin{aligned} p(x_1) &= \int p(x_1, x_2) dx_2 = \mathcal{N}(\mu_1, \Sigma_{11}), \\ p(x_2) &= \int p(x_1, x_2) dx_1 = \mathcal{N}(\mu_2, \Sigma_{22}). \end{aligned} \tag{2.10}$$

Given one Gaussian the posterior conditional is defined by:

$$\begin{aligned} p(x_1|x_2) &= \mathcal{N}(x_1|\mu_{1|2}, \Sigma_{1|2}) \text{ with} \\ \mu_{1|2} &= \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2) \text{ and} \\ \Sigma_{1|2} &= \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \end{aligned} \tag{2.11}$$

The posterior conditional of $p(x_2|x_1)$ can be computed analogues to equation 2.11. The derivations of these formulas are not provided here but are shown by Murphy in *Machine learning : a probabilistic perspective* [11].

Definition 2 presents Gaussian Processes as collection of random variables. Therefore, there has to be the marginalization property of equation 2.10 ensuring that the distribution of a small set of random variables does not change even if there are more variables observed [10].

Inference in Gaussian Process Regression

Supervised learning consists of learning the best parameters according to the training data and then using these parameters for generating predictions for unseen data points. This part explains how these predictions for unseen test data can be generated with a Gaussian Process regression model.
For doing inference with the Gaussian Process regression model the prior and the knowledge about the observed training data points $(x_i, f_i|i = 1, \ldots, n)$ are combined. The goal is to predict test outputs f_* for test inputs x_*. The training and test data points are subject to a Gaussian joint prior distribution such as in 2.9 given by:

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right). \tag{2.12}$$

Suppose there are n training points and n_* points belonging to the test set, the $n \times n_*$ $K(X, X_*)$ matrix contains all the element-wise covariances between them. This is analogous to the matrices $K(X, X)$ and $K(X_*, X_*)$ which contain the pairwise covariances between all training points respectively all test data points. In order to take the information from the observed training data into account the conditional posterior is computed as described in 2.11. Additionally, a constant mean function of zero is chosen as in equation 2.2.2, simplifying the conditional joined posterior to

$$\begin{aligned} f_*|X_*, X, f \sim \mathcal{N}(&K(X_*,X)K(X,X)^{-1}f, \\ &K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*)). \end{aligned} \tag{2.13}$$

In practice a fraction of the identity matrix is added to the kernel matrix to ensure invertibility. Consequently the Gaussian joined prior of equation 2.12 changes to

$$\begin{bmatrix} f \\ f_* \end{bmatrix} \sim \mathcal{N}\left(0, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right). \tag{2.14}$$

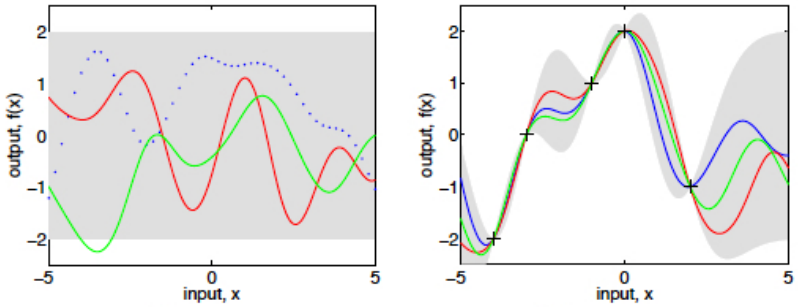
The conditional predictive distribution is directly affected by the modification of the joined prior and changes to

$$f_*|X, f, X_* \sim \mathcal{N}\left(\bar{f}_*, \mathbb{C}(f_*)\right), \text{ where} \tag{2.15}$$

$$\bar{f}_* = \mathbb{E}[f_*|X, f, X_*] = K(X_*,X)\left[K(X,X) + \sigma_n^2 I\right]^{-1} f, \tag{2.16}$$

$$\mathbb{C}\left(f_*\right) = K(X_*,X_*) - K(X_*,X)\left[K(X,X) + \sigma_n^2 I\right]^{-1} K(X,X_*). \tag{2.17}$$

Section 3.3.6 includes a more detailed discussion, about how the invertibility can be ensured. The predictions f_* for x_* can be generated by sampling from the given normal distribution in 2.13.



(a), prior (b), posterior

Figure 2.4.: a) Showing three functions sampled from the prior. b) Showing the joint posterior conditional. The knowledge of the five observed data points is incorporated with the prior [10].

This computational process can be thought about as sampling functions from the prior and only accepting the ones which are consistent regarding the training data points. This process is illustrated in figure 2.4, where the shaded grey region visualizes the mean plus minus two times the standard deviation for each point over the input domain. Panel a) visualizes three randomly sampled function from the prior distribution. The difference of the dotted function from the solid function is the way they were sampled. For the dotted function the function values were generated, while the solid functions were evaluated for a large amount of points and joined finally. Panel b shows five observed points each marked with a "+" sign, and three drawn functions from the posterior resulting from the prior conditioned on the observed points. The effect of the data points can be seen from the fact that the uncertainty decreases around the seen data points [10].

Generating multivariate Gaussian samples

As described in the previous section the posterior combining the knowledge from the prior and the observed data points can be easily computed given the joint Gaussian distribution of two random variables as in equation 2.9. The predictions f_* can be generated by sampling from the conditional joined posterior. This is why this section explains how to draw multivariate Gaussian samples. This explanation builds up on the usage of using a scalar Gaussian generator. The goal is to generate samples x from $\mathcal{N}(m, K)$ wit a given mean m and covariance matrix K.
The Cholesky decomposition is used to take the matrix square root of the positive definite covariance matrix K:

$$K = LL^T. \tag{2.18}$$

By making use several times of the scalar Gaussian generator we can sample $u \sim \mathcal{N}(0, I)$. Finally, the new function x can be computed through $x = m + Lu$, which is a sample from $\mathcal{N}(m, K)$ [10].

2.2.3. Gaussian Process Classification as Multi Target Regression

All supervised Machine Learning problems can be classified into regression or classification problems. This thesis aims for improving the efficiency of Gaussian Processes and investigates the classification problem of digits of the MNIST data set as Garriga-Alonso, Rasmussen, and Aitchison did [7].
Gaussian Processes represent real-valued functions, and thus the classification problem has to be turned into a multi-target regression problem. This is a very common approach, which can be seen for example in [12] and [7] and this is why multi-target regression is used here as well. The advantage of multi-target regression is that the prior and likelihood can be chosen to be Gaussian. This results also in a Gaussian posterior and therefore in a Gaussian Process. In case the problem can be described as a multi-target regression problem the output domain is continuous and therefore a Gaussian Process can be used. For the case of a classification problem there is the problem that the likelihood is not Gaussian since the classes are not continuous. There are ways to apply Gaussian Processes for classification problems directly, too, which are explained in [10]. In this thesis a multi-target regression problem is solved instead of a classification problem.
A multi-target regression distinguishes itself from single target regression by the fact that by feeding an input of the input domain into the model, the model will return several outputs and not just one. In the case of a classification problem the number of outputs for one data point corresponds to the number of classes of the classification problem.

2.3. Convolutional Neural Networks as Convolutional Gaussian Processes

CNNs are state of the art models due to its characteristics fitting the properties of image data. Still, they are also vulnerable to adversarial attacks since they are not using Bayesian Inference and therefore do not incorporate uncertainty. Gaussian Processes as non-parametric models are making use of Bayesian inference and therefore also provide an uncertainty estimate for every prediction. Intuitively these two models seem to be very different.
However Neal showed that the function corresponding to a one layer hidden deep neural network in the limit of infinitely many neurons is equivalent to a Gaussian process [13].

2.3.1. Equivalence of single layer neural networks and Gaussian Processes

This section repeats the arguments from Neal about the equivalence between a single hidden layer neural network and a Gaussian Process. Given a fully connected single hidden layer neural network the input is denoted with x and the ith neuron of the output is represented

by z_i^1. A specific input α is expressed as x^α. The parameters which are the weights and biases are i.i.d drawn from the following normal distributions, where N_1 denotes the width of the first and only hidden layer.

$$\begin{aligned} W_{ij} &\sim \mathcal{N}(0, \frac{{\sigma_w}^2}{N_1}) \\ b_i &\sim \mathcal{N}(0, {\sigma_b}^2) \end{aligned} \tag{2.19}$$

Therefore, z_i^1 (the pre-activation) and x_j^1 (the post-activation) can be computed as follows, dependent on the input x:

$$\begin{aligned} z_i^1(x) &= b_i^1 + \sum_{j=1}^{N_1} W_{ij}^1 x_j^1(x), \\ x_j^1(x) &= \phi\left(b_j^0 + \sum_{k=1}^{d_{in}} W_{jk}^0 x_k\right). \end{aligned} \tag{2.20}$$

Due to the fact that the parameters are drawn i.i.d, the outputs x_i, x_j after applying the nonlinearity ϕ are independent for any pair i, j where $i \neq j$. Furthermore, $z_i^1(x)$ is composed of a sum of i.i.d terms. Applying the Central Limit Theorem for $N_1 \to \infty$ the post affine transformations $z_1^1(x)$ are Gaussian distributed.
Analogous the multivariate Central Limit Theorem states that any finite collection of $\{z_i^1(x^{\alpha=1}), \ldots, z_i^1(x^{\alpha=k})\}$ share a joint multivariate Gaussian distribution. This corresponds to the definition of Gaussian processes (Def. 2). It can be easily seen that all outputs z_i^1 come from one common Gaussian process with mean μ_1 and covariance K^1.

$$z_i^1 \sim GP(\mu_1, K^1) \tag{2.21}$$

From equation 2.19 it can be seen that all parameters have a mean and therefore an expectation value of zero. As a consequence of that the mean function of the GP is zero as well [12]:

$$\mu_1(x) = \mathbb{E}(z_i^1(x)) = 0,$$

$$K^1(x, x') = \mathbb{E}(z_i^1(x), z_i^1(x')) = \sigma_b^2 + \sigma_w^2 \mathbb{E}\left(x_i^1(x), x_i^1(x')\right) = \sigma_b^2 + \sigma_w^2 C(x, x'). \tag{2.22}$$

The computation of the covariance K^1 follows Neal introducing $C(x, x') = \mathbb{E}[x_i^1(x), x_i^1(x')]$ [13].

2.3.2. Gaussian Processes representing deep neural networks

After the presentation that a single fully connected layer in the limit of infinity many neurons corresponds to a GP, the researchers [12] and [14] went further and showed that even deep neural networks can be represented by a Gaussian Process, where the depth of the network is finite but all alyers approach infinite width. The proof builds up on the results of Neal[13] and can be studied in "Deep Neural Networks as Gaussian Processes" or "Gaussian

Process Behaviour in Wide Deep Neural Networks" [13, 12, 14]. As these proofs showed the equivalence between Gaussian Processes and fully connected deep neural networks, Garriga-Alonso, Rasmussen, and Aitchison proofed that there also exist an equivalence between Convolutional Neural Networks and Gaussian Processes in the limit of infinitely many convolutional kernels [7].

2.3.3. ConvNet Kernel

Methods to improve the efficiency of Gaussian Processes are presented in the following chapter. An explanation of the GP which is used as baseline in the following chapter for classifying images of the MNIST data set is given here.
This section presents how Garriga-Alonso, Rasmussen, and Aitchison retrieve an equivalent Gaussian Process for a Convolutional Neural Network. The proof that GPs can represent CNNs can be studied in the paper [7].
A Gaussian Process is defined by its mean $m(X)$ and covariance function $k(X, X')$ for any two inputs X and X'. Therefore, it is necessary to retrieve the mean and covariance function corresponding to the convolutional network, presented in section 2.1 with figure 2.1, to find an equivalent GP.

In order to find the right mean and covariance function the entire CNN is denoted with the following mathematical formulation:

$$A_{i,j}^{l+1}(X) = b_i^{l+1} + \sum_{j=1}^{C^{(l)}} \sum_{\nu=1}^{H^{(l)}D^{(l)}} W_{i,j,\mu,\nu}^{(l+1)} \phi(A_{j,\nu}^{(l)}(X)). \tag{2.23}$$

In this equation l and $l+1$ represent the input and output layers, the output channels are denoted by j and $i \in \{1, \ldots, C^{(l+1)}\}$, and ν and $\mu \in \{1, \ldots, H^{(l+1)}D^{(l+1)}\}$ express the location within the feature maps or the input and output channel [7].

Mean function of the ConvNet Kernel

The mean function is obtained by taking the mean of equation 2.23

$$\mathbb{E}[A_{i,j}^{l+1}(X)] = \mathbb{E}[b_i^{(l+1)}] + \sum_{j=1}^{C^{(l)}} \sum_{\nu=1}^{H^{(l)}D^{(l)}} \mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)} \phi(A_{j,\nu}^{(l)}(X))] = 0. \tag{2.24}$$

It can be seen that the mean function becomes zero, because the expectation of a sum can be expressed as its sum of expectations (eq. A.1), with a mean of zero for the bias $b_i^{(l+1)}$ and the weights $W_{i,j,\mu,\nu}^{(l+1)}$ according to their prior distributions defined in equation 2.3. Besides, the fact is used that the weights are independent of the results of the previous layer $\phi(A_{j,\nu}^{(l)}(X))$ so that the expectation of a product equals the product of its expectations (eq. A.2). This is zero, since the weights have an expectation of zero.
The fact that the mean function of the ConvNet Kernel is zero simplifies the mean of the Gaussian joint posterior for creating predictions from equation 2.11 to equation 2.13 [7].

Covariance function of the ConvNet Kernel

For most networks the covariance has to be evaluated between all pairs of locations in the feature map $\left(\mathbb{C}[A_{i,\mu}^{(l+1)}(X), A_{i,\mu'}^{(l+1)}(X')]\text{for } \mu, \mu' \in \{1, \ldots, H^{(l+1)}D^{(l+1)}\}\right)$, which would be extremely computational expensive since there are $N^2\left(H^{(l+1)}D^{(l+1)}\right)^2$ many different pairs of locations. In this case it is easier however, because it shows that it is sufficient to compute only the "diagonal" covariance: $\left(\mathbb{C}[A_{i,\mu}^{(l+1)}(X), A_{i,\mu}^{(l+1)}(X')] \text{ for } \mu \in \{1, \ldots, H^{(l+1)}D^{(l+1)}\}\right)$. This is less computational work, since there are only $N^2\left(H^{(l+1)}D^{(l+1)}\right)$ pairs left for which the covariance has to be computed.
This applies for the output layer $L+1$, where the dimensions fit to the principle of multi-target regression. This means that $H^{(L+1)} = D^{(L+1)} = 1$ and the depth corresponds to the number of classes of the classification problem. Consequently the covariance can be only computed at one location. The following derivations show that since the output is only needed at some locations it is sufficient to evaluate the covariance at the corresponding locations of the input. This argument applies for the whole network until the first layer is reached.
The covariance function can be computed by taking the covariance of the CNN of any two different inputs X and X':

$$\begin{aligned}
\mathbb{C}[A_{i,j}^{l+1}(X), A_{i,j}^{l+1}(X')] &= \mathbb{V}[b_i^{(l)}] + \sum_{j=1}^{C^{(l)}}\sum_{j'=1}^{C^{(l)}}\sum_{\nu=1}^{H^{(l)}D^{(l)}}\sum_{\nu'=1}^{H^{(l)}D^{(l)}} \mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)}\phi(A_{j,\nu}^{(l)}(X))W_{i,j',\mu,\nu'}^{(l+1)}\phi(A_{j',\nu'}^{(l)}(X'))] \\
&= \sigma_b^2 + \sum_{j=1}^{C^{(l)}}\sum_{j'=1}^{C^{(l)}}\sum_{\nu=1}^{H^{(l)}D^{(l)}}\sum_{\nu'=1}^{H^{(l)}D^{(l)}} \mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)}W_{i,j',\mu,\nu'}^{(l+1)}]\mathbb{E}[\phi(A_{j,\nu}^{(l)}(X))\phi(A_{j',\nu'}^{(l)}(X'))] \\
&= \sigma_b^2 + \sum_{j=1}^{C^{l}}\sum_{\nu=1}^{H^{(l)}D^{(l)}} \mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)}W_{i,j,\mu,\nu}^{(l+1)}]\mathbb{E}[\phi(A_{j,\nu}^{(l)}(X))\phi(A_{j,\nu}^{(l)}(X'))].
\end{aligned} \tag{2.25}$$

The first line can be obtained by using the definition of the covariance as defined in equation A.3. Besides the derived property that the mean is zero of equation 2.24 is applied and the fact that the expectation of a sum can be expressed as sum of expectations (eq. A.1). Additionally, it is also made use of the properties of the weights and bias according to equation 2.3. The second line is retrieved by separating the expectations of the independent weights and activations from the previous layer according to equation A.2. This is analogous to equation 2.24. The weights $W_{i,j}^{(l+1)}$ and $W_{i,j'}^{(l+1)}$ are independent because they are sampled i.i.d for different channels even though they belong to the same feature map. Therefore, $\mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)}W_{i,j',\mu,\nu'}^{(l+1)}] = 0$ for $j \neq j'$. Besides, every weight matrix $W_{i,j}^{(l+1)}$ possesses only zeros or independent variables for every row μ (Fig. 2.1), so that $\mathbb{E}[W_{i,j,\mu,\nu}^{(l+1)}W_{i,j',\mu,\nu'}^{(l+1)}] = 0$ for $\nu \neq \nu'$. Consequently the sums over j' and ν' can be discarded ending up with the result of equation 2.25.

Figure 2.1 shows that for any given patch μ weights $W_{i,j}^{(l+1)}$ are zero at the locations ν which

do not belong to μ. Therefore, the sum over ν can be limited over the non zero values. The first layer variance can be expressed as follows:

$$K_{\mu}^{(1)}(X, X') = \mathbb{C}[A_{i,\mu}^{(1)}(X), A_{i,\mu}^{(1)}(X')] = \sigma_b^2 + \frac{\sigma_w^2}{C^{(0)}} \sum_{i=1}^{C^{(0)}} \sum_{\nu \in \mu \text{th patch}} X_{i,\nu} X'_{i,\nu}. \tag{2.26}$$

The same applies for any other layer, but with the difference that the input are the activations from the previous layer:

$$K_{\mu}^{(l+1)} = \mathbb{C}[A_{i,\mu}^{(l+1)}(X), A_{i,\mu}^{(l+1)}(X')] = \sigma_b^2 + \sigma_w^2 \sum_{\nu \in \mu \text{th patch}} V_{\nu}^{(l)}(X, X'), \tag{2.27}$$

with the covariance of the activations, which are independent of the channel j [7]

$$V_{\nu}^{(l)}(X, X') = \mathbb{E}[\phi(A_{j,\nu}^{(l)}(X))\phi(A_{j,\nu}^{(l)}(X'))]. \tag{2.28}$$

Covariance of the activations

The advantage of Gaussian Processes is that there exist closed form solutions for the conditional joint posterior from which the predictions are generated. Therefore, it is essential that also the elementwise covariance of the activations (eq. 2.27) can be computed in closed form. This is the case for some ϕ if the activations come from a Gaussian distribution.
Several researchers provide some closed form solutions of different choices of ϕ. Nowadays most networks and also ConvNet make use of the ReLU nonlinearity, this is why this thesis only presents the closed form solution of the ReLU nonlinearity derived by Cho and Saul and G. Matthews, Hron, Rowland, et al. [15, 14]. Following their research the ReLU nonlinearity can be expressed as follows:

$$\begin{aligned} V_{\nu}^{(l)}(X, X') &= \frac{\sqrt{K_{\nu}^{(l)}(X, X) K_{\nu}^{(l)}(X', X')}}{\pi} \left(\sin \theta_{\nu}^{(l)} + \left(\pi - \theta_{\nu}^{(l)} \right) \cos \theta_{\nu}^{(l)} \right) \\ &\text{with} \\ \theta_{\nu}^{(l)} &= \arccos \left(\frac{K_{\nu}^{(l)}(X, X')}{\sqrt{K_{\nu}^{(l)}(X, X) K_{\nu}^{(l)}(X', X')}} \right). \end{aligned} \tag{2.29}$$

All the different derivations are are assembled in Algorithm 1 showing how to compute the complete ConvNet kernel from scratch.

Efficiency of the ConvNet kernel

Optimization of software is applied in all fields of computer science not only in Machine Learning, since the user is benefiting from a software which needs less time to run or less space. This is why developers in general aim for finding an efficient solution for a given problem.

Garriga-Alonso, Rasmussen, and Aitchison also identified some opportunities to improve the kernel computation of the ConvNet regarding efficiency. At first it can be seen from the equations 2.27 and 2.29 that the covariance of the activations at one layer only depends on the covariance of the activation from the previous layer. This is really important, because this leads to the fact that the computational cost is within the cost of the forward pass of the equivalent CNN with only one kernel in every layer regardless some constant.
The covariance matrix can be computed easier because it does not need to be computed for all possible pairs of points $a_j^{(l)}(x)$ and $a_j^{(l)}(x')$ resulting in $2H^{(l)}D^{(l)} \times 2H^{(l)}D^{(l)}$. It is sufficient to compute a $H^{(l)}D^{(l)}$ vector per layer and points. Another point that simplifies the computation of the required variances and covariances for the ConvNet Kernel is the specific form of the kernel (eq. 2.1 and 2.2) leading to the fact that they can be efficiently computed as convolutions [7].

The two different Machine Learning Models Convolutional Neural Networks and Gaussian Processes were presented in this chapter. They might seem very different from the first sight, however there are Gaussian Processes which are equivalent to CNNs and therefore bring the advantage of Bayesian Inference, which provides uncertainty estimation and are therefore less vulnerable to adversarial attacks.
The importance of the equivalence of fully connected neural networks and Convolutional Neural Networks to GPs can be explained by being able to apply Bayesian inference on these networks since there exist a Gaussian Process representation. Gaussian Processes offer precise output prediction distributions with a closed form solution. But the challenge of that is to retrieve before mathematically the mean and covariance function of the GP corresponding to a specific neural network as seen for the ConvNet GP in section 2.3.3. When this mean and covariance function is known, GPs offer a closed form solution for training and inference. This is an advantage to normal neural networks where the training often takes very much time in order to optimize the gradients where no closed form solution exists, because optimizing the loss of a neural network with respect to its parameters is not a convex problem. But also GPs have a computationally expensive problem to deal with since the covariance function $K()$ has to be evaluated between all training points n to obtain the matrix $K(X, X)$ and all training points and new points n_* to obtain the matrix $K(X, X_*)$. Equation 2.13 shows that these two matrices are necessary for generating predictions. Therefore, the next chapter deals with the topic of efficiency of Gaussian Processes and several methods to save time by computing these matrices.

3. Efficient implementation of deep convolutional Gaussian Processes

In this chapter the current efficiency bottlenecks of the computational expensive parts of Gaussian Processes are explained. Several of these bottlenecks are removed, improving the efficiency for some computations by a factor of 40%. To make the term "efficiency" clear and assessable, evaluation metrics for speed and performance of Machine Learning models are introduced in section 3.1 and applied later on for GPs.
The section 3.2 presents the base line implementation and evaluation of ConvNet from Garriga-Alonso, Rasmussen, and Aitchison, who also worked on the computational efficiency of the ConvNet GP as explained in section 2.3.3. Building up on that implementation this thesis presents several techniques about how to improve the efficiency for the computation in section 3.3. Afterwards section 3.5 evaluates these methods with the previously presented metrics and compares it to the original results of [7] presented in section 3.2.

3.1. Definition of performance and efficiency measurements for Gaussian Processes

There usually exist several models to tackle a given Machine Learning problem. Therefore, it is essential to have some metrics to evaluate each model on its own. The results can then be used to compare all the models among each other in order to meet a reasonable and comprehensive decision about which model to use. The metrics which are used also depend on the problem to be solved.
This thesis aims at developing methods for computing Gaussian Processes more efficiently than they are used at the moment. Therefore, there have to be metrics presented in advance to be able to evaluate whether different methods help to improve the efficiency of GPs or not. In the following subsections metrics are presented which are used afterwards to evaluate different approaches to make Gaussian Processes more efficient. One may assume that efficiency is only about time and memory space. But in this context it also matters how the performance of a model is affected by using methods which aim for saving computational time. This is why classification accuracy on the MNIST data set is also used to assess the performances.

It is important to remark that improving the efficiency is often hard to achieve (sometimes it is, e.g by changing from python to c with the same code) without sacrificing the best classification accuracy. The goal is to increase the efficiency by computing only a part of the

$K(X, X)$ matrix exactly and approximating the other elements. So, an approximated kernel matrix is used to compute the predictions for each test sample. Obviously it is impossible that the approximated matrix achieves the same accuracy as the exactly computed kernel matrix.

3.1.1. Time measurement

One metric which is very important to assess whether a technique can be used successfully to increase the efficiency is time. The obvious reason for that is that this thesis aims for improving the efficiency and therefore is looking for methods which can reduce the run time of Gaussian Processes, specifically the ConvNet GP. For approaches taking longer than the actual implementation of [7] it is not useful to replace it by a longer method, since these researchers already worked on developing an efficient implementation.
Nowadays most researchers in the domain of all kinds of neural networks and also Gaussian Processes make use of Graphical Processing Units (GPUs) which make it possible to train the models with lots of data in less time than using CPU only. By using GPUs there also comes a challenge regarding time measurement. The challenge of time measurement is that often the time is measured on CPU, but if the GPU is used as well the GPU can work in parallel to the CPU. The risk that arises from that is that the CPU might stop the time measurement, even though the GPU is still working in parallel at the same time. This behaviour is called asynchronous execution. Therefore, the time measurement has to be synchronized between GPU and CPU so that the CPU stops measuring the time, right after the GPU has finished its jobs. This thesis pays attention to that problem, even though there are no jobs running on GPU and CPU, it is ensured by using the synchronization that it is sufficient to measure on CPU only.
Another problem coming by the usage of the GPU is that the GPU is initializing before it is used for computing. Therefore, the first run of a process on GPU is delayed by the time the GPU takes to initialize. Consequently it makes sense to warm up the GPU before running the experiments which are supposed to be measured. In that way the initialization time of the GPU is not part of the time measurement.
Finally it is not sufficient and significantly meaningful to measure only one run. This is why we run every process in the same setting several times and take the average value in the end as measured run time [16].

3.1.2. Accuracy

The other important part which is essential to assess is the performance of the model. In this thesis the accuracy is used for that. Table 3.1 shows the confusion matrix of a regular binary classification problem. The columns represent the ground truth while the rows represent the predictions coming from the model. Consequently the correctly classified samples are on the diagonal of the matrix. They are either positive and correctly classified as positive (true positive) or negative and correctly classified as negative (true negatives). On the right upper corner there are the samples which are actually negative but classified as positive and therefore called false positives. On the left bottom corner there are the samples which

are actually positive and wrongly labelled to be negative. These samples are called false negatives.

		Ground truth	
		1	0
Predicted Labels	1	TP	FP
	0	FN	TN

Table 3.1.: Confusion matrix of a binary classification problem with classes 0 and 1

For a binary classification problem the accuracy can be computed by taking the number of correctly classified samples TP and TN and dividing them by the total number of samples as the following equation shows:

$$\text{Accuracy} = \frac{TN + TP}{TN + FN + FP + TP}.$$

This can be extended to a classification problem with any number of classes by just summing the diagonal elements of the extended confusion matrix and dividing by the total number of samples. Shortly the accuracy denotes how many samples in relation to the whole data set were classified correctly also shown by equation 3.1.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of predictions in total}}. \tag{3.1}$$

The drawback of accuracy is that it is not really meaningful in case of an imbalanced classification problem. This can easily be seen from the fact that it favours classes with a lot of samples, meaning that a model with a lot of samples belonging to one class and much less samples belonging to the other class can still obtain a good accuracy score if it barely classifies the samples from the underrepresented class correctly, but knows how to classify the samples of the major class. In an extreme case a model only predicting the major class could obtain a high accuracy score. This example shows that it is important to think about which metrics are used for evaluation [17].
This problem does not apply to the MNIST classification task of this thesis, because MNIST is a balanced data set. Furthermore, it is always paid attention to the fact that the model is trained and evaluated with balanced data even if only a subset of the MNIST data set was used for experiments.

Other popular metrics assessing the performance of Machine Learning models are Recall and Precision, with their following definitions:

$$\text{Recall} = \frac{TP}{TP + FN'},$$
$$\text{Precision} = \frac{TP}{TP + FP}.$$

Precision tries to measure which fraction of positive classified samples were actually positive. This evaluation metrics is useful to see if the model is able to return only the relevant instances. For example precision is useful for a disease test, where it measures how many people actually had the disease which were labelled as positive. Consequently it returns how much one can believe the model when it classifies a sample as positive. If that value is low the test or in other contexts the model looses its trustworthiness. Recall measures how many samples from all positives were correctly classified as positive. This is the counterpart to precision and to stay with the example it measures if someone has the disease and makes the test how likely it is that the test is also positive. In general recall and precision are used in case of an imbalanced data set [18].

This thesis aims for classifying the digits of the MNIST data set. Since this data set is balanced and there is also no use case to apply recall nor precision, this thesis makes only use of accuracy in order to evaluate the model performance.

3.1.3. Root-Mean-Square-Error

In section 3.3 the goal of this thesis is to generate a close approximation of a matrix with missing values to its original version with no missing values. These algorithms can only be evaluated if the error between the original matrix and its approximation can be assessed. In general there exists a lot of different errors who are used assessing that. The following is made use of in this chapter, where the original matrix is denoted with X and its approximation with $\tilde{X}$

$$\text{RMSE} = \sqrt{\frac{\sum \left(X - \tilde{X}\right)^2}{N}} = \frac{||X - \tilde{X}||_2}{\sqrt{N}}. \tag{3.2}$$

N in equation 3.2 denotes the number of values which was deleted of X and is therefore approximated in $\tilde{X}$. The nominator contains the square root of the sum of the squared differences between both matrices and therefore can be expressed by the $L2$-norm.
This error measures just an absolute value and is therefore hard to interpret without knowing the scale of the data. Therefore, the techniques in section 3.3 are evaluated by using the relative form which is obtained by taking the error of equation 3.2 and dividing it by the difference of the maximum and minimum value of the matrix X. The entire relative error can be seen here:

$$\text{Relative RMSE} = \frac{\text{RMSE}}{\max X - \min X} = \frac{||X - \tilde{X}||_2}{(\max X - \min X)\sqrt{N}}. \tag{3.3}$$

3.2. Evaluation of the ConvNet GP on MNIST

The presented metrics of the last section make it possible now to evaluate the ConvNet GP of [7]. This is essential since the goal of this thesis is to improve the efficiency of GPs in

general but applied on the use case of this ConvNet GP. Therefore, this section presents the architecture, the evaluation of the original model and investigates where the computation of the GP takes the most time.

MNIST

This thesis follows Garriga-Alonso, Rasmussen, and Aitchison and makes also use of the famous MNIST data set for training and doing inference with the GP. MNIST contains 70 000 gray-scale images showing handwritten digits from 0 to 9 and is balanced, so that every number occurs approximately equally often in the data set. Therefore, it is a famous classification data set, but as explained in section 2.2.3 it is tackled by multi-target regression here due to several advantages with Gaussian Processes.

3.2.1. Architecture of ConvNet and ConvNet GP

Algorithm 1 and the equations 2.26, 2.27 and 2.29 show how to compute the covariance function of the ConvNet GP for two input points X and X'. However it can be seen that there are several hyperparameters which have to be determined. The hyperparameters of ConvNet GP are the following ones: σ_b^2, σ_w^2, the number of layers, the stride and the padding of the convolution, the filter size and the activation function. Garriga-Alonso, Rasmussen, and Aitchison optimized these hyperparameters by random search and this thesis uses the same ones in order to obtain comparable results. Table 3.2 shows the optimized used hyperparameters.

Hyperparameter	ConvNet GP
σ_w^2	2.79
σ_b^2	7.86
# layers	7
Stride	1
Kernel size	1
Padding	SAME
Activation function	ReLU

Table 3.2.: Hyperparameter of ConvNet GP [7]

By determining the hyperparameters of the ConvNet GP the structure of the CNN, which is equivalent to it in the limit of infinitely many kernels per layer, is also determined. This corresponding CNN is visualized in figure 3.1.

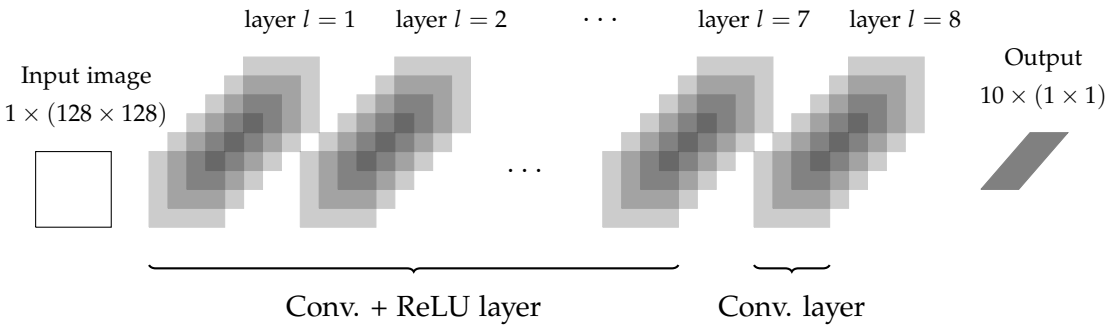


Figure 3.1.: The ConvNet is taking a 128×128 grayscale image as input. Feeding it through seven convolutional and ReLU layers. The last layer is only a convolutional layer returning a 1×1 output of depth 10 corresponding to the 10 classes of MNIST. In the limit the number of feature maps per layer goes to infinity so that it can be represented as a GP (ConvNet GP).

3.2.2. Evaluation of the ConvNet GP

Since the hyperparameters are fixed from the previous section, the metrics of section 3.1 can be used to evaluate the ConvNet GP. The GP is evaluated on the MNIST data set, but on less pictures trained in comparison to [7] due to the limitation of computation resources. Run time analysis and efficiency measurements strongly depend on the machine they are executed. The machine used for this thesis is an g4dn.2xlarge machine from AWS. The specifications can be seen in the appendix A.1. There are three different settings of the original ConvNet GP, which are evaluated with this machine.

1. Setting one using 5000 images for training and 1000 images each for validation and testing
2. Setting two using 20 000 images for training and 2500 images each for validation and testing
3. Setting 3 using 25 000 images for training and 2500 images each for validation and testing

There are a lot of different ways of data pre-processing and augmentation especially in the domain of images. This aims for training the model in a way it is still able to classify images correctly even if they might be scaled or rotated. This thesis follows [7] regarding data pre-processing as well and this is why it does not apply any form of it. The images are directly fed into the model.
In general all of the data sets used for training, validation or testing are balanced and disjoint within the setting, so that it is not validated nor tested on points used for training before. Table 3.3 shows the accuracy on the validation and test sets for all three settings. Furthermore, the time it takes to run for all settings is measured.
This time includes the computation of the kernel matrices $K(X, X)$, containing the covariance

evaluated between all points of the training set, $K(X, X_V)$ and $K(X, X_T)$, containing the covariance between all training data points and points of the validation set respectively test set. Additionally, the measured time includes the time it takes to generate the predictions from these kernel matrices according to equation 2.13. The predictions are obtained by computing the mean of equation 2.13 and taking the class with the maximum value. The uncertainty estimate to that prediction is obtained by evaluating the corresponding covariance of the GP, which requires the computation of the matrices $K(X_V, X_V)$ and $K(X_T, X_T)$. These matrices contain the point wise covariances between all points in the validation set respectively test set. The time which is measured and denoted in table 3.3 does not include the computation of the uncertainty estimation.

	Validation accuracy	Test accuracy	Time in minutes
Setting 1	96.4%	97.5%	4
Setting 2	98.68%	98.72%	64
Setting 3	98.24%	98.52%	94

Table 3.3.: Evaluation of the ConvNet GP

3.2.3. Efficiency Problems of Gaussian Processes

This thesis aims for improving the efficiency of Gaussian Processes. This goal can only be achieved by identifying the parts of GPs which have the most negative impact on the total run time. When these parts are identified it can be aimed for optimizing the efficiency finally. Therefore, it makes sense to see how the predictions are generated. Equation 2.13 show that the predictions are obtained by computing $K(X_*, X)K(X, X)^{-1}f$ if the assumption that the mean function is zero holds and f denotes the labels of the training data.
From that equation it can be seen that this equation contains two challenging parts.
One problem is computing the matrix $K(X, X)$, because this one is in comparison to all other matrices the biggest one, since there are much more data points in the training set than in the validation or test set. Therefore, the matrix $K(X, X)$ can be very large for large training data sets. Consequently computing the kernel matrix $K(X, X)$ slows down the process of generating predictions. Asymptotically it can be determined that computing this matrix is in $\mathcal{O}(N^2LD)$ where L denotes the number of layers, D the dimension of the training points and N the number of training points.

After having computed all the matrices the inverse has to be computed for the $K(X, X)$ matrix, which is the second part taking more computational effort. Computing the inverse is in $\mathcal{O}(N^3)$. For the MNIST data set it only takes around a factor of ten more to compute the inverse than computing the the kernel matrix. The reason for that are the low dimensional images of MNIST having only one channel, because they are gray-scale images, meaning that $LD \approx \frac{N}{10}$ [7].

From an asymptotic analysis it can be seen which parts take the most time, but since the kernel computation and the computation of the matrix are close to each other it makes sense

to investigate further which part is more affecting the run time of a GP.
For this purpose there exist profilers, which can be run together with the software and provide information about the number of calls of each module in the code, the time each module takes excluded any calls to other modules and the accumulated time including the time of any sub-calls of other modules.
This thesis uses cProfile from the standard python library to investigate the source code more in detail and consequently to identify the software parts which take the most time in order to optimize them. In general the software consists of three main parts to compute the predictions in the end. The first module is responsible for computing the kernel matrices $K(X, X)$, $K(X, X_V)$ and $K(X, X_T)$ where the first matrix is the biggest also the most important one.
In case there are several GPUs available the load is split between both GPUs where one worker on each GPU is computing parts of the matrices. This is why the second step merges the results of the workers. Finally the last part uses the computed kernel matrices to compute the predictions for the validation and test set by using equation 2.13 with $K(X, X_*)$ corresponding to $K(X, X_V)$ or $K(X, X_T)$ respectively. Besides it always computes the accuracy as described in section 3.1.
The cProfiler is used with setting 3 of the previous defined settings. It clearly shows that the computation of the kernel matrices dominates the run time as shown in table 3.4.

	Kernel Matrices	Merging	Predictions & Accuracy	Total
Time in minutes	93	0	1	94
Proportion in %	99	0	1	100

Table 3.4.: Run time of the different modules measured by cProfiler

From this result it can be seen that even though the theoretical rum time of the inversion of a matrix should be bigger than the one computing the kernel matrices, the time computing the kernel matrices clearly dominates the whole run time. The reason for this may be that matrix inversion is implemented very efficiently already, and is performed with an iterative algorithm (see section 3.3.6. Besides, the inversion of the kernel matrix can be executed quite fast in comparison to that, since it takes only 1% of the total time. The merging is not executed because the machine which is used has only one GPU.
As a consequence of that the goal has to be to reduce the time it takes to compute the kernel matrices in order to improve the efficiency of the GP. Improving any other module does not improve the efficiency since the total time and is not significantly affected by anything else than computing the kernel matrix.
The module computing the kernel matrices computes the kernels $K(X, X)$, $K(X, X_V)$ and $K(X, X_T)$. The biggest matrix of these ones is the $K(X, X)$ containing the point wise covariance between every pair of points of the training set. Therefore, computing this matrix affects the run time much more than computing $K(X, X_V)$ or $K(X, X_T)$, since the validation and test set are in the most cases much smaller than the training set. Therefore, this thesis focuses on how to improve the efficiency of the computation of the kernel matrix $K(X, X)$.

At first the connection between the number of training data points and the time it takes to compute the $K(X, X)$ matrix is investigated. Figure 3.2 shows, that by increasing the data set linearly the time increases quadratically. The number of training data points is increased from 200 to 5000, and from there to 25 000 in steps of 5000. All evaluations are marked with a blue dot. The orange function is a fitted quadratic function successfully proving that the growth is quadratic. Intuitively this also makes sense since the corresponding kernel matrix also grows quadratically for adding new points to the training set.

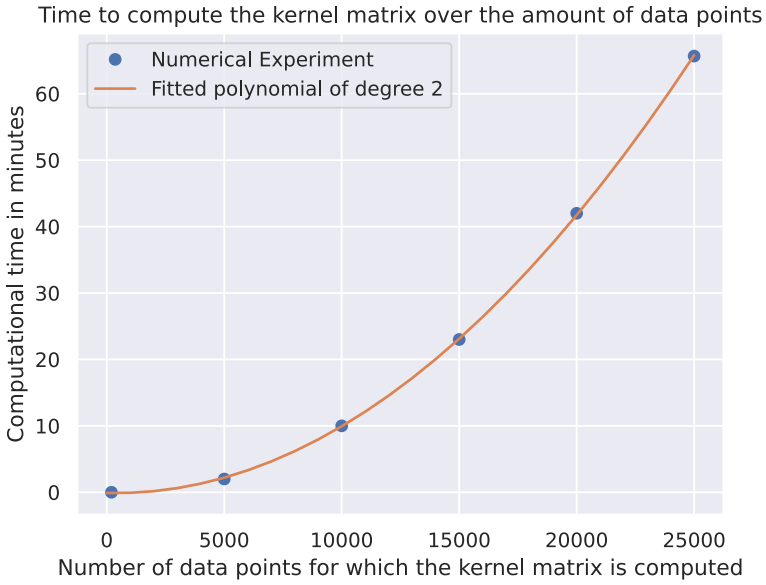


Figure 3.2.: The time for computing the matrix $K(X, X)$ grows quadratically with the size of the training data set.

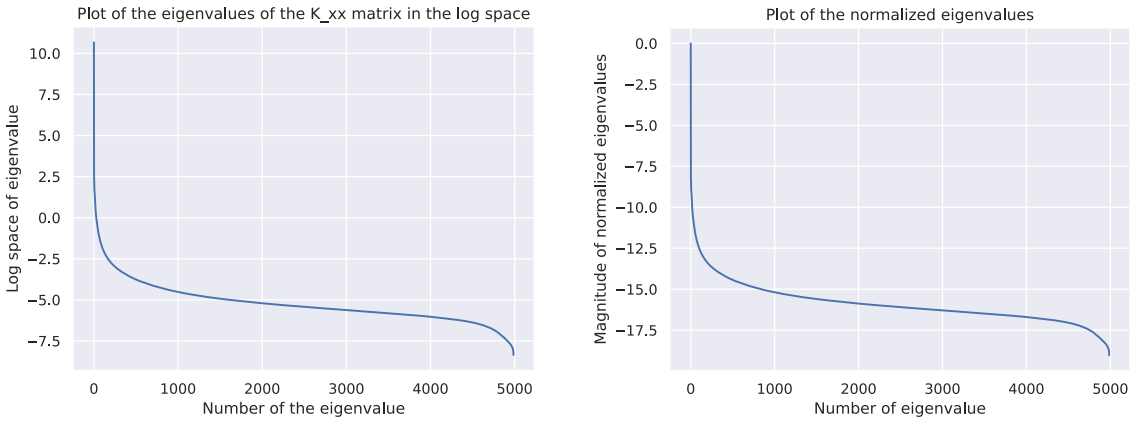
The quadratic growth shows that it makes sense to do research about increasing the efficiency of the $K(X, X)$ matrix, since the run time for computing rises very fast for larger training data sets. Therefore, the investigation of how the computation of the $K(X, X)$ matrix can be improved is presented in the next section.

3.3. Methods to improve Gaussian Proccesses' performance

It can be seen easily that the best choice of optimizing this Gaussian Process Regression task is to optimize the computation of the kernel matrices.
This is a typical problem since kernels are used for several applications in the Machine Learning domain such as for Support Vector Machines, Kernel PCA or Kernel Regression. The kernel trick transfers the original data into a higher dimensional space where it might be possible to separate the transformed data linearly. This transformation is applied by computing the kernel between each data point of the data set and every other one. This $n \times n$

kernel matrix is usually dense. It takes $\mathcal{O}(n^2)$ space to store the corresponding kernel matrix and $\mathcal{O}(n^2 d)$ operations to compute it, where n is the amount of data points with dimension d. Therefore, there exist approaches trying to approximate the kernel matrix using less memory space and consequently also reducing the computation time, which normally also decreases, because the computation time usually correlates to the used memory space [19].

This is why this section aims for improving the efficiency for computing the $K(X, X)$ matrix. Therefore, it makes sense to extract the structure of the kernel matrix to see which techniques might be helpful to reach this goal. So the kernel matrix of setting 1 was computed and the eigenvalues are plotted in figure 3.3. In the figure on the left the eigenvalues are plotted in the log space, while on the right they are additionally normalized by the biggest eigenvalue λ_1.



(a) (b)

Figure 3.3.: The plots show the eigenvalues of the 5000×5000 $K(X, X)$ matrix. In 3.3a the eigenvalues are plotted in the log space. In 3.3b the eigenvalues in the log space of 3.3a are normalized by the biggest eigenvalue λ_1.

Both plots clearly show that the eigenvalues drop very fast and as a consequence, the full kernel matrix can be well approximated by a low rank matrix. This shows that the data in the matrix is not independent and can therefore be represented by using a subset of eigenvectors. Both plots show that the first 500 eigenvalues are much larger than the other ones, leading to the consequence that the first 500 eigenvectors are sufficient to describe most of the information contained in the $K(X, X)$ matrix. Therefore, it shows that already 10% of the eigenvectors cover the information of most of the data.
This is why this section tries to exploit this knowledge by trying several techniques building up on computing only a subset of the kernel matrix exactly and approximating the missing values. The challenge of that is that the $K(X, X)$ matrix is dense. The following subsections introduce these methods.

3.3.1. Iterative Singular Value Decomposition (SVD)

The first method introduced here is Iterative SVD, presented first by Troyanskaya, Cantor, Sherlock, et al. to approximate missing values of gene microarray data to be able to execute experiments successfully [20]. They also make use of the fact that the data in general is correlated such as the data in the kernel matrix $K(X, X)$ shown by figure 3.3. Therefore, it makes sense to also apply this technique for the problem of estimating the kernel matrix.

This method applies Singular Value Decomposition (SVD) iterative several times. Before it can be applied the first time, all missing values are approximated by replacing them with the average of the row they belong to. This step is essential, because SVD can only be applied to complete matrices. This new generated matrix is called $\tilde{K}(X, X)$ from now on. The SVD of that matrix can be seen in equation 3.4, where Σ is a diagonal matrix containing the singular vectors in descending order on its diagonal denoting how much of the variance of the total data can be explained by the corresponding right eigenvector. The right eigenvectors are contained in the matrix V^T, while the left ones are contained in U. They are orthonormal to each other and can therefore span a lower-dimensional subspace.

$$\tilde{K}(X, X) = U\Sigma V^T \tag{3.4}$$

As shown in figure 3.3 the data can be explained by taking just a few eigenvectors. This is why here only the k most significant right singular vectors are used to approximate the missing values, where k is determined empirically. The k selected singular vectors are fitted in a regression way for every row of $\tilde{K}(X, X)$, where only the values are taken into account which were set from the beginning on and not have been approximated by taking the row mean. After the parameters are determined by fitting the regression, the values are approximated by generating predictions from the fitted regression model. This procedure is applied to every row. After having fitted a regression model using the singular vectors and generated predictions with it for the missing values for every row of $\tilde{K}(X, X)$ the SVD is computed for the new generated matrix replacing $\tilde{K}(X, X)$. This is applied in an iterative fashion as long as the total difference of the new obtained $\tilde{K}(X, X)$ and the old one is below a specific threshold [20].

Performance of Iterative SVD

In general computing the SVD for a $m \times n$ matrix is in $\mathcal{O}(n^2 m)$. For this method the SVD is executed several times due to the expectation maximization algorithm which is used. Therefore, the complexity is in $\mathcal{O}(n^2 mi)$ for i iterations until the threshold criterion is fulfilled. Consequently the complexity for the $K(X, X)$ matrix is in $\mathcal{O}(n^3 i)$, where n is the number of training data points [20]. This is also one drawback of this method, because the SVD is applied on the complete matrix, which is still computationally expensive. The Nyström method presented in section 3.3.4 decreases the complexity by applying SVD only on a part of the kernel matrix.

3.3.2. Soft Impute

Mazumder, Hastie, and Tibshirani developed an algorithm motivated by the "Netflix competition" where the goal is to generate movie recommendations for each user. This problem boils down to a matrix completion problem, where from a huge user-movie matrix only some values are given and the missing ones are aimed to be predicted. This thesis introduces this method because figure 3.3 shows that the data lies in a lower-dimensional manifold and also the researchers of Soft Impute assume this behaviour for their matrix completion problem.

In the following the matrix $\tilde{K}(X,X) \in \mathbb{R}^{n\times n}$ denotes the matrix which is not complete and for which the missing elements are aimed to be approximated. $\Omega \subset \{1,\dots,n\} \times \{1,\dots,n\}$ denotes the indices of the given elements of $\tilde{K}(X,X)$. The goal of Soft Impute is to learn a lower rank representation $Z \approx V_{n\times k} G_{k\times n}$ of $\tilde{K}(X,X)$ where $k \ll \min(n)$.
This goal can be noted as following optimization problem:

$$\begin{aligned} &\text{minimize } \operatorname{rank}(Z) \\ &\text{subject to } \sum_{(i,j)\in\Omega} \left(\tilde{K}_{i,j}(X,X) - Z_{i,j}\right)^2 \leq \delta, \end{aligned} \tag{3.5}$$

where δ is a regularization parameter responsible for the tolerance in the training error. The problem can be solved for a complete matrix by using the k most significant singular vectors and singular values of the SVD of $\tilde{K}(X,X)$ (eq. 3.4). [21] The rank problem of equation 3.5 makes it very hard to solve, because it is not convex. [22] This is why convex relaxation is used to simplify the optimization problem. So, instead of minimizing the rank of Z, the nuclear norm of Z is minimized, which is equivalent to minimize the sum of singular values of Z. [23] Therefore, the problem can be expressed as follows, where the nuclear norm is denoted by $||Z||_*$.

$$\begin{aligned} &\text{minimize } ||Z||_* \\ &\text{subject to } \sum_{(i,j)\in\Omega} \left(\tilde{K}_{ij}(X,X) - Z_{ij}\right)^2 \leq \delta. \end{aligned} \tag{3.6}$$

The problem of 3.6 can be solved by semi-definite programming algorithms making use of second order methods. [24] This leads to the problem that second order methods are computationally expensive for large matrices. [25] Therefore, 3.6 can be formulated as a Lagrangian optimization problem:

$$\min_Z \frac{1}{2} \sum_{(i,j)\in\Omega} \left(\tilde{K}_{ij}(X,X) - Z_{ij}\right)^2 + \lambda ||Z||_*, \tag{3.7}$$

where λ has to fulfil $\lambda \geq 0$ and is a direct mapping of $\delta > 0$ controlling the nuclear norm.
The Soft Impute algorithm therefore minimizes the derived equation 3.7. For doing that a new matrix is introduced projecting missing values to zero and keeping observed elements. Y is just used as an exemplary matrix in this equation.

$$P_\Omega(Y)(i,j) = \begin{cases} Y_{ij} & \text{if } (i,j) \in \Omega \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

The complementary projection $P_{\Omega}^{\perp}(Y)$ is defined so that $P_{\Omega}^{\perp}(Y) + P_{\Omega}(Y) = Y$.
By making use of equation 3.8 $\sum_{(i,j)\in\Omega} \left(\tilde{K}_{ij} - Z_{ij}\right)^2$ can be expressed as
$||P_{\Omega}(\tilde{K}(X,X)) - P_{\Omega}(Z)||_F^2$. Therefore, the minimization problem of 3.7 can be denoted as:

$$\min_Z f_\lambda(Z) := \frac{1}{2}||P_{\Omega}(\tilde{K}(X,X)) - P_{\Omega}(Z)||_F^2 + \lambda||Z||_*. \tag{3.9}$$

Finally a lemma of [25] is shown which is used by the Soft Impute algorithm.

Lemma 3.3.1. Given a matrix $W \in \mathbb{R}^{m\times n}$ with rank r. The solution of the optimization problem

$$\min_Z \frac{1}{2}||W - Z||_F^2 + \lambda||Z||_*.$$

is given by

$\hat{Z} = S_\lambda(W)$ where
$S_\lambda(W) = UD_\lambda V^T$ with $D_\lambda = diag[(d_1 - \lambda)_+, \ldots, (d_r - \lambda)_+]$
UDV^T denotes the SVD of W analogous to equation 3.4,
$D = diag[d1, \ldots, d_r]$ and $t_+ = \max(t, 0)$

The Soft Impute algorithm solves equation 3.9 by making use of lemma 3.3.1 for different values of λ and is shown in the appendix 2. It is similar to the previous explained Iterative SVD algorithm. The algorithm updates the missing elements of the matrix in an iterative fashion by repeatedly solving equation 3.9 of the previous solution. This is executed until the solution converges [21].

3.3.3. UV Decomposition

The UV decomposition is a form of Singular Value Decomposition presenting one matrix as a product of two thin matrices under the assumption that the underlying data structure of the original matrix can be described by a low number of features. This is also the fact for the $K(X,X)$ matrix which is aimed to be approximated. This kernel matrix $K(X,X)$ with n rows and n columns is estimated with a matrix U with n columns and d rows and a matrix V with d columns and n rows such that $K(X,X)$ is roughly $U * V$.

The RMSE used for the UV decomposition is very similar to the RMSE presented in equation 3.2. The only difference is that for the UV decomposition only the observed values are taken into account, not the estimated ones.

The UV decomposition starts with randomly initialized matrices U and V which are updated alternating to minimize the RMSE. This means that one matrix is fixed while the other one is updated and this changes in the next step. The challenge of that problem is that there are several local minima. This is why this algorithm is executed several times

with different initial matrices U and V, even though there is no guarantee to find the global minimum of the RMSE. All in all the UV decomposition consists of the following four steps:

1. Normalization of the original matrix this case $K(X, X)$
2. Random initialization of the matrices U and V
3. Optimizing U and V
4. Stop optimization.

It often makes sense to normalize the original matrix in order to execute matrix completion, because the values of the matrix might be scaled differently depending on the meaning of the column and the rows and it is easier to fill the blank spaces of a matrix if its entries have the same scale. In the case of this thesis it is not necessary to normalize the matrix $K(X, X)$ due to the fact that $K(X, X)$ is symmetric and contains the kernel between all data points of the training set. So the values of $K(X, X)$ have already the same scale [26].

There exist several local minima, therefore the algorithm is ran several times with different random initializations for the matrices U and V to get several different solutions and increase the chance to find the global minimum of the RMSE among them.
An easy approach is to start with U and V filled by the same value $\sqrt{\frac{a}{d}}$, where a is the mean value of the non empty values of $K(X, X)$ and d is the chosen number of features explaining the data. In case the original matrix was normalized then this fraction is zero. In order to obtain several starting points this fraction can be perturbed randomly and independently for every element of U and V [26].

The goal of UV decomposition is to find $U \in \mathbb{R}^{n \times d}$ and $V \in \mathbb{R}^{d \times n}$ such that:

$$\min_{U,V} \sum_{i,j \in \mathbb{R}} (K(X, X)_{rj} - u_r * v_j)^2,$$

where $K(X, X)_{rj}$ corresponds to the element in row r and column j of the kernel matrix $K(X, X)$.
In the following paragraph the matrix P is obtained by multiplying U by V.

$$P = UV$$

This paragraph shows how an arbitrary element u_{rs} of matrix U is optimized to decrease the Root-Mean-Squared Error between $K(X, X)$ and UV. The element u_{rs} only affects the elements in row r of matrix P. The RMSE excludes all elements from $K(X, X)_{rj}$ and p_{rj} from the calculation where the value is blank in the kernel matrix. Consequently only the elements $p_{r,j}$ are important for which $K(X, X)_{rj}$ is not blank for any j.

$$p_{r,j} = \sum_{k=1}^{d} u_{rk} v_{kj} = \sum_{k \neq s} u_{rk} v_{kj} + x v_{sj} \tag{3.10}$$

u_{rs} is from now on replaced by x as shown in equation 3.10 in order to simplify the notation for the optimization example. The notation $\sum_{k \neq s}$ means that k is iterating from 1 to d skipping only the case $k = s$. By using equation 3.10 the following equation shows how x contributes to sum of the squares of RMSE for any j where $K(X, X)_{rj}$ is not blank.

$$(K(X, X)_{rj} - p_{r,j})^2 = (K(X, X)_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2$$

The sum notation $\sum_j$ in the following equation means it sums over all j such that $K(X, X)_{rj}$ is not blank. So the sum of squares of the RMSE affected by the change of x is given by:

$$(K(X, X)_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2 = \sum_j (K(X, X)_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2.$$

From this equation the derivative has to be taken with respect to x and set to zero, since x should be updated to minimize the RMSE. The following equation 3.11 shows the derivative set to zero.

$$\sum_j -2 v_{sj} (K(X, X)_{rj} - \sum_{k \neq s} u_{rk} v_{kj} - x v_{sj})^2 \stackrel{!}{=} 0 \tag{3.11}$$

The last step is to solve for x so that the new value for x can be computed. The solution is presented in the following equation:

$$x = \frac{\sum_j v_{sj} (K(X, X)_{rj} - \sum_{k \neq s} u_{rk} v_{kj})}{\sum_j v_{sj}^2}. \tag{3.12}$$

Updating an arbitrary value $v_{rs} = y$ of the matrix V is analogous to the explained optimization of an arbitrary value $u_{rs} = x$. Equation 3.13 shows the analog solution how to optimize a $v_{rs} = y$ to minimize the RMSE. Analogous to $\sum_j$, $\sum_i$ is the sum over all i such that $K(X, X)_{is}$ is not blank. Besides $\sum_{k \neq r}$ is the sum from k is one to d excluded the case that x is r.

$$y = \frac{\sum_i u_{ir} K(X, X)_{is} - \sum_{k \neq r} u_{ik} v_{ks})}{\sum_i u_{ir}^2} \tag{3.13}$$

The important thing to notice is to apply alternating optimization. This means that only one entry of one matrix is optimized while the other matrix is fixed. In this way all elements of U and V can be updated after each other. This whole process has to be executed as long as the values for U and V do not converge, because updating an element once does not correspond to obtaining the optimal value for that element to minimize the RMSE. After updating another element of one of the matrices the element could be updated again. This is why that optimization strategy of U and V is executed until U and V converge or the stopping criterion is fulfilled. The elements of U and V can either be optimized one after the other in a Round-Robin fashion, but they can also be updated in a random order if it is ensured that every element is optimized once per round [26].

Theoretically the goal is to optimize the approximation of U and V until the RMSE gets zero. In practice this is often not possible due to the fact that there might be more elements missing in $K(X, X)$ than U and V have elements together. So it is important to define a criterion when to stop optimizing. A good approach is to stop optimizing when the RMSE could not be decreased by at least a certain threshold for one entire round of optimizing U and V. This stopping criterion is also used for Iterative SVD and Soft Impute [26].

All the previously presented methods are based on Singular Value Decomposition, providing the best low rank approximation to any matrix. But the computation of a SVD might be still too expensive for some tasks. This is why there also exist some other approaches such as Greedy basis selection techniques, incomplete Cholesky decomposition, and Nyström methods. The next section introduces the Nyström method.

3.3.4. The Standard Nyström method

The Nyström method is motivated by estimating large kernel matrices as $K(X, X)$ is. As already mentioned this is not only useful for Gaussian Processes, but also for Support Vector Machines, Kernel PCA or Kernel Regression.

The standard Nyström method from Williams and Seeger suggests to sample $m \ll n$ columns from the Kernel matrix $K(X, X) \in \mathcal{R}^{n \times n}$ [27]. These m columns build the matrix $C \in R^{n \times m}$. The matrix $M \in \mathcal{R}^{m \times m}$ is the much smaller kernel matrix between the sampled m data points. Therefore, C and K can be rewritten as follows without any loss of generality:

$$\begin{aligned} C &= \begin{bmatrix} M \\ E \end{bmatrix}, \\ K(X, X) &= \begin{bmatrix} M & E^T \\ E & F \end{bmatrix}. \end{aligned} \tag{3.14}$$

The standard Nyström method generates a k-rank approximation $\tilde{K}$

$$K(X, X) \approx \tilde{K} = C M_k^+ C^T. \tag{3.15}$$

M_k^+ is the pseudo inverse of the k rank approximation of M. The k rank approximation of M is obtained by applying Singular Value Decomposition on the matrix M. In contrast to the previous methods the Singular Value Decomposition is more efficient since it is only applied on the reduced kernel matrix M of the sampled m data points and not on the original kernel matrix K(X,X).
The matrix M is symmetric and can therefore be written as $M = U \Lambda U^T$, where U is an orthonormal matrix and Λ is a diagonal matrix with all singular values σ of M in decreasing order on its diagonal. Therefore, M_k^+ can be written as

$$M_k^+ = \sum_{i=1}^{k} \sigma_i^{-1} U_i U_i^T \tag{3.16}$$

by applying SVD [28].

Efficiency of the Nyström method

The SVD of M is in $\mathcal{O}(m^3)$, while the other matrix multiplications of equation 3.15 are in $\mathcal{O}(nmk)$. It can be seen from that the standard Nyström method takes $\mathcal{O}(m^3 + mnk)$ in time. The SVD of the kernel matrix directly is in $\mathcal{O}(n^3)$ and therefore much slower than the standard Nyström method since $m \ll n$.
This is why this method is faster and better suited to approximate K than the matrix completion algorithms presented in the previous subsections [28].

3.3.5. Normalization Layer

Figure 3.4 shows that Iterative SVD, Soft Impute and Matrix Factorization can impute missing data successfully if the scale of the matrix is not too large. A normalization layer is added after every convolution and before the ReLU layer. This layer normalizes all three distinct entries of the bivariate covariance matrix between the inputs, namely $K_\mu^{(l+1)}(X, X), K_\mu^{(l+1)}(X, X')$ and $K_\mu^{(l+1)}(X', X')$. The normalization corresponds to a division by the maximum value. This normalization layer does not affect the process of generating predictions. This can be seen from the equation 2.13, where the predictions correspond to $K(X_*, X)K(X, X)^{-1}f$. The normalization layer is used for computing all kernel matrices not only $K(X, X)$. Therefore, the normalization applied to $K(X_*, X)$ and the normalization of $K(X, X)$ cancel out, since $K(X_*, X)$ is multiplied by the inverse of $K(X, X)$. The next sections show that Iterative SVD, Soft Impute and Matrix Factorization can now be used with the normalization layer after every convolution.

3.3.6. Linear Solver

Additional to the normalization layer this thesis contributes also in another way to the way Garriga-Alonso, Rasmussen, and Aitchison compute the predictions. A linear solver is used to compute:

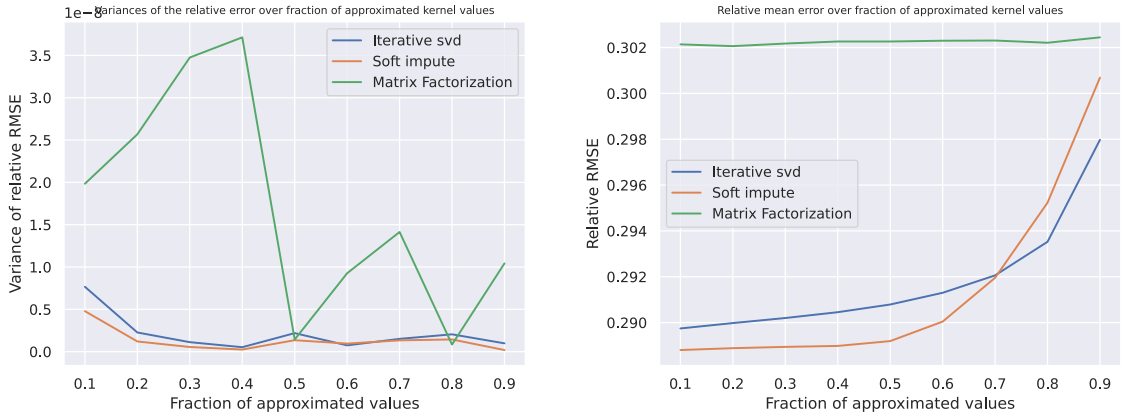
$$K(X, X)^{-1}f$$

for computing equation 2.13 and consequently the predictions [7]. The problem of the previous used solver is that it is not able to solve this equation for many cases because $K(X, X)$ is singular. A matrix is singular and cannot be solved in theory if its determinant is zero. This is equivalent to the case that the product of its eigenvalues is zero and therefore it is sufficient that one eigenvalue is zero. Figure 3.3 shows that the eigenvalues are decreasing very fast and therefore it is likely that the matrix is close to singular. This thesis uses a solver which is able to solve this kind of equations and does that also in a very fast way. The scipy.linalg.solve method is replaced by the scipy.sparse.linalg.lsqr method, an iterative approximation method if a least squares solution where a tolerance of $1e^{-3}$ was set.

3.3.7. Evaluation of the presented methods

The previous sections contain the explanation of four different techniques to compute only a part of the big and dense $K(X, X)$ matrix and approximate the missing values.
At first the performance of these methods is evaluated by generating a random 5000×5000 matrix and deleting randomly values, beginning with omitting only 10% of the data and ending up with discarding and approximating 90% of the matrix data. For this experiment the relative RMSE (eq. 3.3) is used. This makes sense because it is also important that the error reflects the scale of the data which is estimated. For example an error of 1.0 is much better if every element x of the approximated matrix: $x \in [0, 100]$, than having $x \in [0, 1]$. The randomly generated values are in a range of zero to one.

At first only the methods of Iterative SVD, Soft Impute and Matrix Factorization (UV decomposition) are evaluated. The reason for that is that these ones are also applicable to general problems of matrix completion, while the Nyström method is specialized to approximate kernel matrices. Therefore, it does not make sense to evaluate the Nyström method on a randomly generated 5000×5000 kernel matrix, where random values are omitted. The fact that the Nyström method is a specific method for the use case of this thesis explains that it is also reasonable that this first approximation experiment is only executed with the more general methods.
Figure 3.4 shows the variance of the relative RMSE on the left 3.4a and the relative Error on the right. Iterative SVD and Soft Impute behave very similar, because both have very low variance in the error and also have a similar relative RMSE starting below 0.29 where only 10% of the matrix has to be approximated until reaching an error of 0.3 where only 10% of the data is given and 90% has to be estimated. It can be seen that the error slightly increases the more data has to be approximated. This makes intuitively sense and is also supported by figure 3.5 which illustrates the maximum error between the original and the approximated matrix over the fraction of missing elements. It shows that the maximum error increases the more values are approximated. As a consequence of that also the expected relative error of figure 3.4 increases the more values have to be imputed, even though also the denominator of the relative RMSE (eq. 3.3) increases since N corresponds to the number of approximated elements.
In comparison to Iterative SVD and Soft Impute Matrix Factorization achieves worse errors and also a higher variance, even though it is almost constant over the fraction of approximated elements. In general the errors for all three techniques are low and seem promising to be used successfully for the original goal approximating the kernel matrix.



(a) Expectation of the variance of the relative error

(b) Expectation of the relative error

Figure 3.4.: A random generated 5000×5000 matrix is approximated with the previously presented methods. From all the values 10% until 90% are omitted and therefore approximated by the methods. Figure 3.4a shows how the variance of the Relative RMSE (eq. 3.3) evolves over the fraction of values which is omitted. Figure 3.4b shows how the expected relative RMSE of equation 3.3 is affected by the fraction of missing values.

Figure 3.5.: The figure shows how the maximum error between a random 5000×5000 and its approximation changes depending on how many elements are estimated.

Additionally to the relative RMSE and its variance also the time it takes to approximate

a specific fraction of matrix elements is plotted in figure 3.6. The exactly measured times are shown in table 3.5. In contrast to the error plots where all the approximation algorithms and especially Iterative SVD and Soft Impute seem very similar to each other regarding their performance there are differences regarding the time it takes for every method to estimate. Especially the Soft Impute algorithm takes much longer the more values are omitted and have to be approximated. The SVD Iteration method and the Matrix factorization algorithm take only slightly more time for an increasing number of missing elements of the matrix, specifically Iterative SVD takes not even two minutes to approximate the random 5000×5000 matrix.

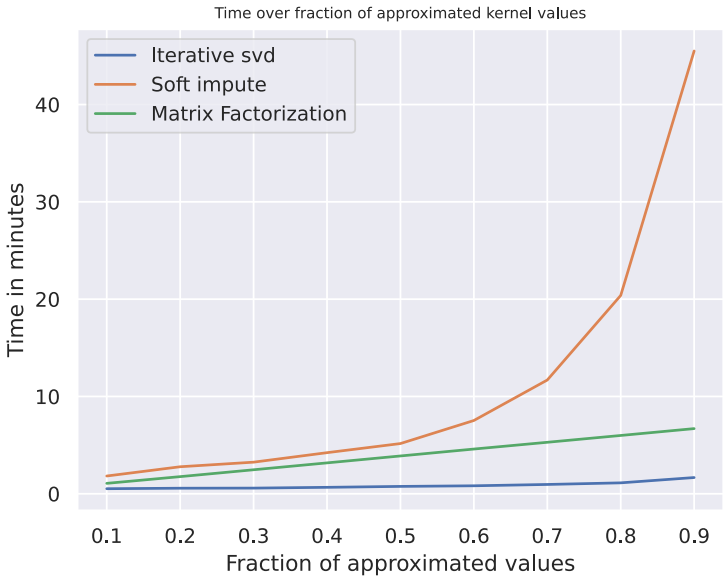


Figure 3.6.: Plot of the expected time it takes to approximate a random 5000×5000 matrix over the fraction of missing elements by the previously presented methods. The time only includes the approximation process, but not the creation of the matrix.

	Fraction of approximated values								
	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Iterative SVD	0.52	0.57	0.58	0.66	0.76	0.82	0.96	1.12	1.67
Soft Impute	1.83	2.78	3.25	4.22	5.16	7.53	11.70	20.38	45.5
Matrix Factorization	1.07	1.76	2.47	3.18	3.89	4.6	5.29	5.99	6.69

Table 3.5.: This table shows how long each method takes exactly to approximate the matrix of figure 3.6. The time is given in minutes.

Summarizing, the plots show that these methods are promising to be able to approximate a kernel matrix successfully. Therefore, the same experiment is set up but instead of taking a

random matrix with the values being in an interval of $[0, 1]$, the $K(X, X)$ matrix is computed exactly and then random elements are omitted from there as before. This experiment shows that all of these methods are not able to impute the missing values, because the measured error is huge. The reason for that is that all kernel matrices are computed by convolutions as stated in 2.3.3. As a consequence of that the values of the kernel matrices are summed up repeatedly and consequently explode at some point. This leads to the fact that the elements of $K(X, X)$ have values up to e^{14}. Therefore, it is impossible for the presented methods to estimate the missing values. This is why the original kernel derived in 2.3.3 is adjusted so that it can be approximated better. In Section 3.3.5 the normalization layer is explained, which prevents kernel values from exploding. This normalization layer makes it possible to approximate the kernel matrix with the presented methods.

3.4. Evaluation of the methods

Figure 3.4 shows that the previously presented methods can be used to approximate a random matrix, with randomly omitted elements. Additionally, the ConvNet kernel is adjusted by adding the normalization layer of subsection 3.3.5 after every convolution to avoid exploding values of the kernel matrix. Now these methods and also the Nyström method are used to approximate the ConvNet kernel matrix for setting 1. This means the goal is to approximate the 5000×5000 kernel matrix $K(X, X)$ and evaluate every method. The challenge of approximating that matrix is that before the values were omitted at random positions of the matrix, but this time the matrix is step wise filled and the other values are estimated. The kernel matrix is symmetric and therefore filled block wise, so that the challenge now is to approximate $K(X, X)$ with having blocks of non observed elements. This phenomenon occurs especially in the bottom right corner of the matrix. The methods are evaluated according to the Relative RMSE between the original exactly computed $K(X, X)$ matrix and its approximation, the time it takes to estimate the missing matrix elements and the accuracy on the validation set of setting 1. On the one hand side it is important to assess the time, since the goal is to improve the efficiency of the kernel computation. But on the other side it is also essential to measure the accuracy, because the goal is still to obtain good classification results with the estimated matrix. In general this thesis aims for a good trade-off between decreasing the computation time and generating good classification results with the GP model. Therefore, the following plots are used to find the best compromise. All figures represent the fraction of the matrix which is occupied on the x-axis. As an example 0.1 means that 10% of the $K(X, X)$ matrix is occupied. This is equivalent to use around $32\% = \sqrt{0.1}$ of the training data.

Figure 3.7 shows the relative RMSE error between the exactly computed kernel matrix $K(X, X)$ in the log space, where 5000 samples of the training set are used and its approximations generated from Iterative SVD, Soft Impute, Matrix Factorization and the Nyström method. It can be seen that Iterative SVD, Soft Impute and Matrix Factorization achieve constantly low relative errors, even if only 10% of the data is observed. The Nyström method

is the exception for that having a much higher error at first when less than 50% of the data is given, but improves a lot afterwards, so that it achieves even lower errors when at least 50% of all elements are observed. The reason for the good performance of these methods is the normalization layer scaling down the kernel matrix, because without this layer it is not possible to achieve these good results.

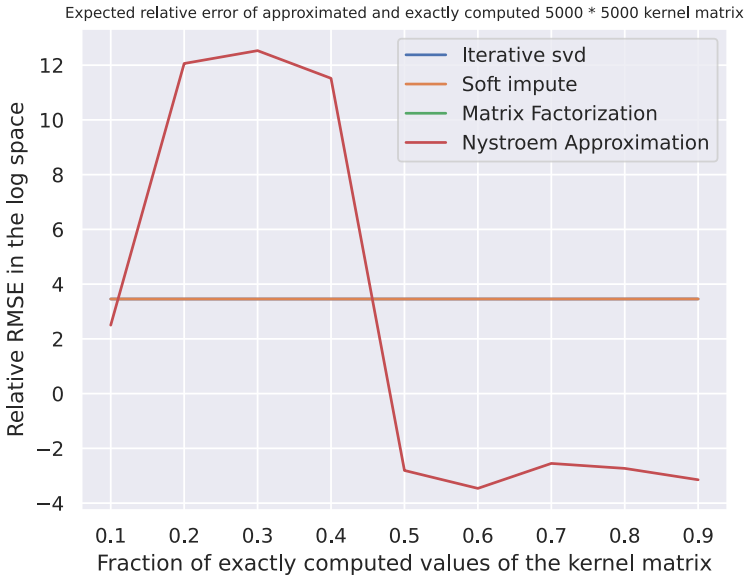


Figure 3.7.: The plot shows how the relative RMSE of the kernel matrix depends on the fraction of data which can be used by the single methods to approximate the missing elements. It can be seen that all methods approximate the kernel matrix very well, even though the Nyström method requires more exactly computed data than the other ones.

The more important metric to evaluate these methods is the accuracy, since it matters the most which method can achieve good classification results with the generated approximation. Figure 3.8 visualizes how the accuracy evolves over the fraction of given data in the kernel matrix. The phenomenon of figure 3.7 where the Nyström method cannot approximate the kernel matrix well for less than 50% of the data given, leads to the fact that it also cannot perform a good classification if less than 50% of the matrix data is given. The reason is obviously that since the matrix cannot be approximated well also the classification suffers from that and is therefore bad. But given at least 50% of the matrix the Nyström method achieves the best accuracy results for all methods, even slightly more than Iterative SVD and Soft Impute can achieve with an exception for 70% of observed values.
Besides, figure 3.8 shows that Iterative SVD and Soft Impute estimate the kernel matrix even with having only 10% of the matrix observed very well, so that classification results from over 80% can be achieved and slightly improve for observing more data.

The Matrix Factorization approach does not obtain any good accuracy results for the generated kernel matrix used on the validation set, even though the relative RMSE in figure 3.7 is low and does not show any bad results there as the Nyström method does. A reason for that could be that for figure 3.4b and figure 3.4a the algorithm showed some instability and was also there good, but still worse then the successfully working Iterative SVD and Soft Impute.

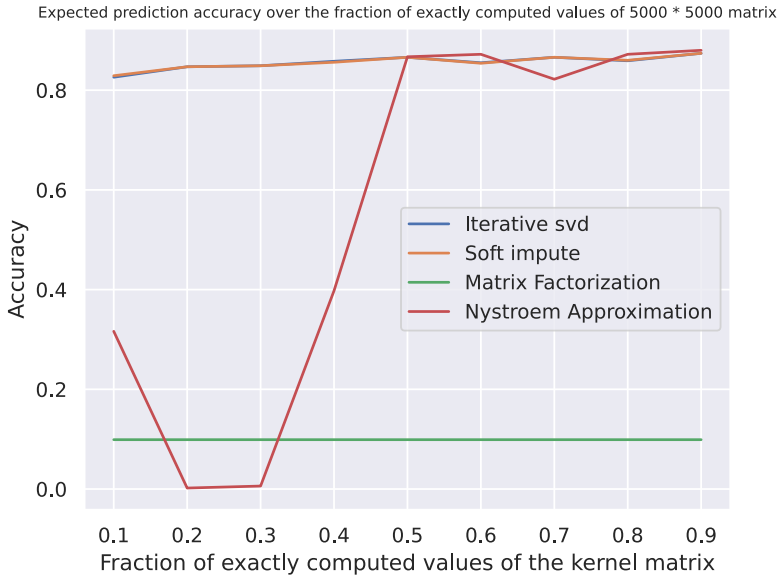


Figure 3.8.: The plot shows how the classification accuracy of the validation set of setting 1 evolves depending on how many values of the matrix $K(X, X)$ are computed exactly. It shows that already 10% of all matrix elements are sufficient to achieve an accuracy over 80% using the Iterative SVD or Soft Impute. Matrix Factorization performs poorly even if more than half of the data of $K(X, X)$ is observed.

The other important factor which has to be evaluated is the time the methods take to approximate the kernel matrix. Figure 3.9 visualizes all the methods and how much time they take to impute the missing values depending on how much data is observed. It clearly shows that there are big differences between the single the techniques. While the Matrix Factorization, the Nyström method and the Iterative SVD take only under 10 minutes, the Nyström method and the Iterative SVD being very fast, the Soft Impute algorithm takes between 35 and 45 minutes. Taking into account that the original implementation of Garriga-Alonso, Rasmussen, and Aitchison takes only a few minutes according to figure 3.2 and setting 1 Soft Impute cannot be considered as an efficient solution.

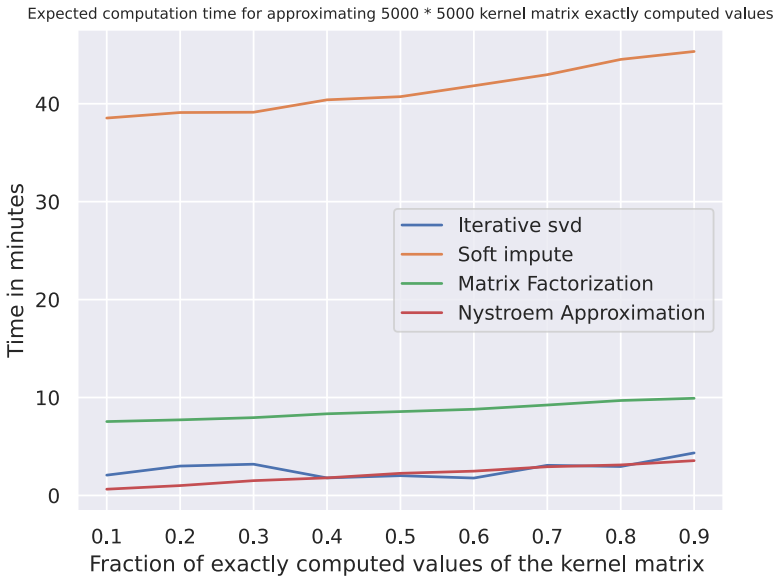


Figure 3.9.: The time is plotted over the fraction of values present in the matrix to approximate. It can be seen that every method is almost constant or increases slightly, but every method is clearly separated from each other. Matrix Factorization, the Nyström method and Iterative SVD take under 10 minutes, while Soft Impute is with at least 35 minutes far away from that.

As a consequence of the previous shown figures 3.8 and 3.9 Soft Impute and Matrix Factorization are no solutions to the goal to improve the efficiency of Gaussian Processes. The Soft Impute algorithm takes by far too long to estimate the 5000×5000 kernel matrix, which is not a big matrix regarding normal sizes of training sets. Furthermore, the performance of the Matrix Factorization approach is poorly, so that it cannot be used in the domain of estimating kernel matrices.

The plots also show that Iterative SVD and the Nyström method can be used for improving the efficiency of bigger kernel matrices. Therefore, there has to be taken the decision of how many training data samples are used and how many values are imputed. In order to take a reasonable decision the same experiment as before is run with only making use of Iterative SVD and the Nyström method, where the goal is to approximate a 12500×12500 matrix.

Efficiency-Performance Trade-off

The previous figures show that it does not make any sense to use Soft Impute and Matrix Factorization for approximating the kernel matrix, because the Soft Impute algorithm takes long even though approximating the matrix and doing inference with it leads to good results. The results obtained with Matrix Factorization are too poorly to be used.
Iterative SVD and the Nyström method show good results and also do not take much time.

There is a trade-off between minimizing the computational time and maximizing the classification accuracy. The following plots help to meet a good decision of how many values should be computed exactly to improve the efficiency. Figure 3.10 illustrates the relative RMSE between the estimated 12500×12500 kernel matrix $K(X, X)$ and the exactly computed one. The Nyström method achieves a very low Relative RMSE which is even not dependent on how much data is observed. This is very different to the previously observed figure 3.4, where this method needed 40% of the data to minimize the Relative RMSE. An explanation for this behaviour might be that 10% of the smaller 5000×5000 matrix is too less data to be used to approximate the missing elements, while it is here sufficient. The Relative RMSE of the Iterative SVD is also constant over the added elements but significantly worse than the Relative RMSE of the Nyström method.

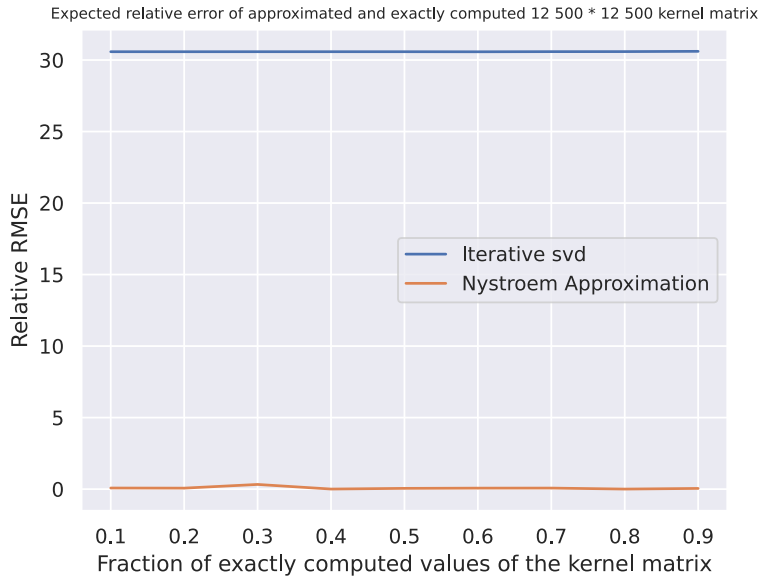


Figure 3.10.: Relative RMSE between approximated and exactly computed 12500×12500 kernel matrix.

Figure 3.11 plots the classification accuracy of the estimated kernel matrix against the fraction of given elements of $K(X, X)$. Similarly to figure 3.8 the Nyström method obtains better overall classification results than using the Iterative SVD method, even though for some fractions the performance is worse than the one of Iterative SVD.
However, the Nyström method obtains a classification accuracy of 82% already if only 10% of the data is observed. This is very different to figure 3.8 where 50% of the data was required to achieve good results. From that it can be seen that the Nyström method has difficulties with approximating a small matrix with having only a small fraction given. However, this improves a lot if the matrix is bigger and therefore also small fractions correspond to more given values. So it can be concluded that the total number of elements is important for the

Nyström method and not the fraction.
The Iterative SVD also already achieves a classification accuracy of over 80% for only given 10% of the data. The performance increases the more elements are observed until obtaining a result of 84% given 90% of the kernel matrix. This makes intuitively sense since less data has to be imputed.

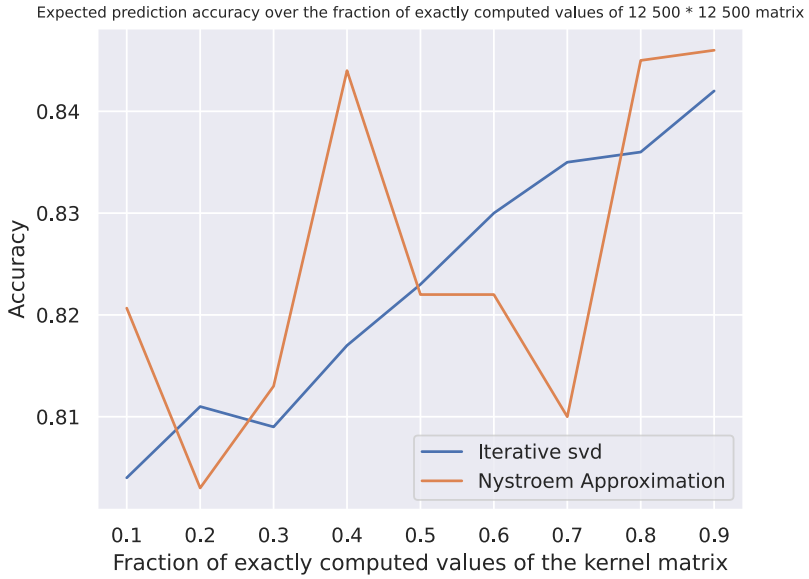


Figure 3.11.: Classification accuracy for using the approximated kernel matrices over observed data in 12500 $\times$ 12500 the kernel matrix.

Finally, the time it takes to approximate the 12500 $\times$ 12500 $K(X, X)$ matrix over the fraction of observed values is visualized in figure 3.12. In contrast to the accuracy plot (fig. 3.11), where the Nyström method obtains the better overall result than the Iterative SVD, here the Iterative SVD takes always less time than the Nyström method. So while the Nyström method obtains better results regarding the overall accuracy, Iterative SVD takes less time. The time increases linearly for both methods the more values are observed. The reason for that is the fact that the values have to be computed exactly before and this takes longer the more values are computed, which can be seen in figure 3.2. This figure also shows that it takes around 15 minutes to compute the kernel matrix exactly. Therefore, the Iterative SVD can be applied to any fraction of observed elements of $K(X, X)$, while for the Nyström method it makes only sense to compute at most 40% of the kernel matrix exactly. Otherwise the method takes longer than computing the entire matrix exactly.

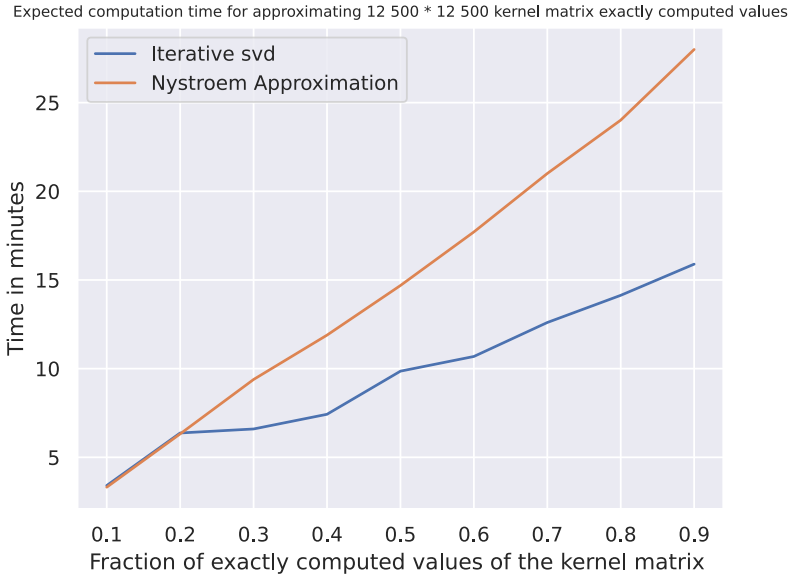


Figure 3.12.: Time it takes to approximate 12500 $\times$ 12500 kernel matrix over the fraction of observed elements.

All in all after taking into account all the plots regarding accuracy and time of the both methods Iterative SVD and the Nyström method for the 5000 $\times$ 5000 and the 12500 $\times$ 12500 kernel matrix it makes sense to compute only 20% of the matrix exactly. The reason for that is according to the accuracy plots it is sufficient for both methods to achieve good results already with just a small percentage of data. This even more applies if the data set is big, so that even a small percentage equals to big number of data points.
Another reason is the time which has to be taken into account, because the more values are computed exactly the more time it takes and since the goal is to improve the efficiency and therefore minimizing the time, it is reasonable to compute only 20% of the kernel matrix exactly. This decision is evaluated for both methods on bigger matrices and compared to the exact computation of the ConvNet GP in the next section in order to see if it makes sense to approximate the matrices or if it is better to compute it exactly.

3.5. Performance of the ConvNet GP using approximation methods

The goal of this thesis is to increase the efficiency. After observing the plots in section 3.4 with approximating a 5000 $\times$ 5000 matrix and a 12500 $\times$ 12500 matrix it could be seen that much time can be saved by computing only 20% of the elements in the kernel matrix exactly and approximate the others. In order to see if this is a good trade-off the settings 1, 2 and 3 are evaluated with the Iterative SVD and the Nyström method for computing only 20% of the data exactly. The accuracy for a validation and test set such as the time it takes to compute all kernel matrices and generate the predictions is measured. These measurements are compared

to the original implementation of [7] where each element is computed exactly.

The results of the evaluation of the first setting, where a 5000×5000 kernel matrix is approximated, can be seen in table 3.6. In general it does not make sense to approximate such small matrices, since the matrix can be computed exactly in just four minutes achieving very high accuracy results with 96.6% and 97.5% for the validation and test set. Iterative SVD and the Nyström method take both less time than the original computation, but the difference is so low, that the kernel matrix for a small number of training points should always be computed exactly. This applies even more when the performance of the Iterative SVD and the Nyström method are evaluated. Analogous to figure 3.8 the Nyström method cannot be used to generate good predictions with only a small fraction of a small training data set, because it performs very poorly. In comparison to that the Iterative SVD performs well achieving 80.2% and 77.5% for the validation and test set and is also faster than the Nyström method, but still far away from the very high results of the exact computation.

	Validation Accuracy	Test Accuracy	Time in minutes
Original ConvNet GP	96.6%	97.5%	4
ConvNet GP with Nyström	0.2%	0.4%	3.22
ConvNet GP with Iterative SVD	80.2%	77.5%	2.69

Table 3.6.: Evaluation of the exact computation, the Nyström method and Iterative SVD regarding time and accuracy of setting 1. Clearly the data set is too small for the Nyström method to achieve good accuracy.

Table 3.7 shows the evaluation results of setting 2, where the training set consists of 20 000 MNIST images and the validation and test set contain 2500 images each. Since the training set of the second setting is much bigger than the one of the first setting it also takes much more time to compute $K(X, X)$ completely, with 64 minutes in comparison to 4 minutes in table 3.6. For this use case it makes much more sense not to compute the entire kernel matrix completely, but only a part of it and approximate the missing elements. This can be seen from the fact that the ConvNet GP with the Nyström method takes only around 38 minutes and the Iterative SVD approach takes even less than with 27 minutes. Regarding the performance the original ConvNet GP achieves excellent results with an accuracy of 98.62% and 98.72% for the validation set respectively test set, but also the Nyström method performs well with accuracies of 87.04% and 85.84% for the validation and test set. Iterative SVD is the fastest technique but also shows the worst performance among these three evaluated methods. 81.8% of accuracy for the validation set and 79.88% of accuracy for the test can be achieved by using the Iterative SVD method, which takes even less than half of the time the original model takes.

	Validation Accuracy	Test Accuracy	Time in minutes
Original ConvNet GP	98.62%	98.72%	64
ConvNet GP with Nyström	87.04%	85.84%	38.89
ConvNet GP with Iterative SVD	81.8%	79.88%	27.44

Table 3.7.: Evaluation of the exact computation, the Nyström method and Iterative SVD regarding time and accuracy of setting 2.

The results of the measured times and accuracy of the validation and test set of setting 3 are presented in table 3.8. The original GP model achieves excellent classification results again with between 98.24% and 98.52% for validation and test set, but it also takes 94 minutes to obtain these results.Iterative SVD and the Nyström method behave similarly to their results in table 3.7. While the Nyström method achieves the higher performance with 87.86% and 86.64% for training respectively test set, Iterative SVD takes the least time to generate predictions. It olny takes 35 minutes in comparison to 54 minutes for the Nyström method and 94 minutes to the original ConvNet GP and still achieves an accuracy of 81.16% for the validation set and 81.68% for the test set.

	Validation Accuracy	Test Accuracy	Time in minutes
Original ConvNet GP	98.24%	98.52%	94
ConvNet GP with Nyström	87.86%	86.64%	54.22
ConvNet GP with Iterative SVD	81.16%	81.68%	35.58

Table 3.8.: Evaluation of the exact computation, the Nyström method and Iterative SVD regarding time and accuracy of setting 3.

From the previously seen evaluations for the different settings it can be seen that it does not make any sense to approximate small matrices, because not a lot of time can be saved inc comparison to the original ConvNet GP implementation. Besides, the accuracy is also worse and the Nyström method performs so poorly that it cannot be used for small matrices. However, for bigger matrices a lot of time can be saved by applying the Nyström method or Iterative SVD. While the Nyström method achieves better classification results, Iterative SVD is faster in approximating the missing kernel matrix elements. The important fact that was shown here is that it is sufficient to use only 20% of the training set to increase the efficiency but still obtain good classification results.

4. Conclusion

The computational expensive part of Gaussian Processes is the computation of the kernel matrix $K(X, X)$, which contains the pairwise covariance between all data points of the training set. In this thesis it is shown that the efficiency can be increased by not computing the whole kernel matrix exactly, but only computing a fraction of it and approximating the missing matrix elements. It can be seen that it is sufficient to evaluate the covariance between only 20% of the training data set and apply Iterative SVD or the Nyström method afterwards to impute the missing values.
This does not make sense if the kernel matrix is small in general and can be computed in less time, because the space for improving the efficiency is small and the performance is bad.
It makes sense to apply it to bigger matrices where a lot of computational time can be saved by approximating the kernel matrix. The Nyström method achieves the better performance results, but Iterative SVD takes less time to impute the missing elements. Therefore, each use case has to be investigated by itself to determine if the performance or the run time is more important and consequently to decide for making use of the Nyström method or Iterative SVD.
In general it seems to make sense for most use cases to apply the Nyström method, since a high classification accuracy is provided and the efficiency still increases significantly.

There are several Machine Learning models that make use of kernel matrices such as Support Vector Machines or Kernel Regression models, where also the Nyström method is used. Therefore, there exist several variants building up one the classic Nyström method explained in this thesis. An example for that is the ensemble Nyström method presented by Kumar, Mohri, and Talwalkar [29]. As the Nyström method can be used to increase the efficiency of GPs it also makes sense to investigate more variants of this method and apply it to Gaussian Processes to see if the efficiency can be improved even more.

In this thesis a normalization layer is used preventing kernel values from exploding (sec. 3.3.5). This layer is rather simple and uses the largest value for normalization. Therefore, there is still space for investigation if more complex ways of normalization like batch normalization for example can lead to better performances.

This thesis uses the ConvNet GP from [7] to increase the efficiency of Gaussian Processes. Therefore, it is necessary to apply Iterative SVD and the Nyström method also on different Gaussian Processes to see if these methods can be used in general to increase the efficiency. Furthermore, this thesis follows [7] and makes also use of the MNIST data set to evaluate the ConvNet GP kernel matrix with all presented methods. There are a lot of different data sets,

such as Cifar-10 distinguishing itself from MNIST by the fact that several color values belong to each pixel. Consequently, it is worth to investigate Iterative SVD and the Nyström method also on different data sets. There is even more potential to increase the efficiency because computing the kernel exactly may take more time due to the higher number of values per pixel.

A. Appendix

A.1. ConvNet kernel

A.1.1. Mathematical Foundations

This section contains some mathematical foundations which are used to retrieve the ConvNet kernel. In the following equations X and Y are independent random variables.
The expectation of a sum is equivalent of the sum of expectations:

$$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y] \tag{A.1}$$

The expectation of a product of independent random variables can be expressed as its product of expectations:

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y] \tag{A.2}$$

The covariance of two random variables is defined as follows:

$$\begin{aligned} \mathbb{C}[X,Y] &= \mathbb{E}[(X - \mathbb{E}[X])\,(Y - \mathbb{E}[Y])] \\ &= \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \end{aligned} \tag{A.3}$$

A.1.2. Algorithm to compute the ConvNet kernel

Algorithm 1 is used to compute the ConvNet kernel.

Algorithm 1: The ConvNet kernel k(X,X′) [7]

input: Two images X and X′ $\in \mathbb{R}^{C^{(0)} \times (H^{(0)} W^{(0)})}$

Compute $K_\mu^{(1)}(X,X)$, $K_\mu^{(1)}(X,X')$, and $K_\mu^{(1)}(X',X')$ for $\{\mu \in \{1,\dots,H^{(1)}D^{(1)}\}$; using Eq. 2.26

for $l = 1,2,\dots,L$ **do**

  Compute $V_\mu^{(l)}(X,X')$, $V_\mu^{(l)}(X,X')$ and $V_\mu^{(l)}(X,X')$ for $\mu \in \{1,\dots,H^{(l)}D^{(l)}\}$; using Eq. 2.29, or different nonlinearity ϕ

  Compute $K_\mu^{(l+1)}(X,X)$, $K_\mu^{(l+1)}(X,X')$ and $K_\mu^{(l+1)}(X',X')$ for $\mu \in \{1,\dots,H^{(l+1)}D^{(l+1)}\}$; using Eq. 2.27

end

output: The scalar $K_1^{(l+1)}(X,X')$

A.2. Specifications of g4dn.2xlarge machine

The machine used for all the evaluations and experiments of this thesis were run on a g4dn.2xlarge machine from AWS. The specifications can be seen in table A.1.

#GPUs	1
GPU	NVIDIA T4 Tensor Core GPU
Processor	Intel Xeon Cascade Lake
GPU memory	16 Gib
RAM	32Gib

Table A.1.: Specifications of g4dn.2xlarge machine.

A.3. The Soft Impute algorithm

The Soft Impute algorithm, which is used to approximate the kernel matrix $K(X, X)$ is shown in algorithm 2.

Algorithm 2: SOFT-IMPUTE [21]

1. Initialize $Z^{old} = 0$
2. Do for $\lambda_1 > \lambda_2 > \cdots > \lambda_k$:
   a) Repeat:
      i. Compute $Z^{new} \longleftarrow S_{\lambda_k}\left(P_\Omega(X) + P_\Omega^\perp(Z^{old})\right)$
      ii. If $\frac{||Z^{new} - Z^{old}||_F^2}{||Z^{old}||_F^2} < \epsilon$ exit.
      iii. . Assign $Z^{old} \longleftarrow Znew$.
   b) Assign $\tilde{Z}_{\lambda_k} \longleftarrow Z^{new}$.
3. Output the sequence of solutions $Z_{\lambda_1}, \ldots, Z_{\lambda_K}$.

List of Figures

List of Tables

Bibliography

[1] S. Minaee, R. Kafieh, M. Sonka, S. Yazdani, and G. Jamalipour Soufi. *Deep-COVID: Predicting COVID-19 From Chest X-Ray Images Using Deep Transfer Learning*. Apr. 2020.

[2] Y. LeCun and C. Cortes. *MNIST handwritten digit database*. 2010. URL: http://yann.lecun.com/exdb/mnist/.

[3] A. Krizhevsky and G. Hinton. *Learning multiple layers of features from tiny images*. 2009.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. IEEE. 2009, pp. 248–255.

[5] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. "Object recognition with gradient-based learning". English (US). In: *Shape, Contour and Grouping in Computer Vision*. Ed. by J. Mundy, R. Cipolla, D. Forsyth, and V. di Gesu. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). International Workshop on Shape, Contour and Grouping in Computer Vision ; Conference date: 26-05-1998 Through 29-05-1998. Springer Verlag, Jan. 1999, pp. 319–345. ISBN: 3540667229.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[7] A. Garriga-Alonso, C. E. Rasmussen, and L. Aitchison. "Deep Convolutional Networks as shallow Gaussian Processes". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=Bklfsi0cKm.

[8] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song. "Robust Physical-World Attacks on Deep Learning Visual Classification". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1625–1634.

[9] J. Bradshaw, A. Matthews, and Z. Ghahramani. "Adversarial Examples, Uncertainty, and Transfer Testing Robustness in Gaussian Process Hybrid Deep Networks". In: *arXiv: Machine Learning* (2017).

[10] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005. ISBN: 026218253X.

[11] K. P. Murphy. *Machine learning : a probabilistic perspective*. Cambridge, Mass. [u.a.]: MIT Press, 2013. URL: https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020/ref=sr_1_2?ie=UTF8&qid=1336857747&sr=8-2.

[12] J. Lee, J. Sohl-dickstein, J. Pennington, R. Novak, S. Schoenholz, and Y. Bahri. "Deep Neural Networks as Gaussian Processes". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=B1EA-M-0Z.

[13] R. M. Neal. *Bayesian Learning for Neural Networks*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 0387947248.

[14] A. G. de G. Matthews, J. Hron, M. Rowland, R. E. Turner, and Z. Ghahramani. "Gaussian Process Behaviour in Wide Deep Neural Networks". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=H1-nGgWC-.

[15] Y. Cho and L. K. Saul. "Kernel Methods for Deep Learning". In: *Advances in Neural Information Processing Systems 22*. Ed. by Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta. Curran Associates, Inc., 2009, pp. 342–350. URL: http://papers.nips.cc/paper/3628-kernel-methods-for-deep-learning.pdf.

[16] A. Greifman. *The Correct Way to Measure Inference Time of Deep Neural Networks*. URL: https://towardsdatascience.com/the-correct-way-to-measure-inference-time-of-deep-neural-networks-304a54e5187f. (accessed: 26.09.2020).

[17] G. Developers. *Classification: Accuracy*. URL: https://developers.google.com/machine-learning/crash-course/classification/accuracy. (accessed: 25.09.2020).

[18] G. Developers. *Classification: Precision and Recall*. URL: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall. (accessed: 25.09.2020).

[19] S. Si, C.-J. Hsieh, and I. S. Dhillon. "Memory Efficient Kernel Approximation". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 682–713. ISSN: 1532-4435.

[20] O. Troyanskaya, M. Cantor, G. Sherlock, T. Hastie, R. Tibshirani, D. Botstein, and R. Altman. "Missing Value Estimation Methods for DNA Microarrays". In: *Bioinformatics* 17 (July 2001), pp. 520–525. DOI: 10.1093/bioinformatics/17.6.520.

[21] R. Mazumder, T. Hastie, and R. Tibshirani. "Spectral Regularization Algorithms for Learning Large Incomplete Matrices". In: *Journal of machine learning research : JMLR* 11 (Mar. 2010), pp. 2287–2322.

[22] N. S. Nati and T. Jaakkola. "Weighted Low-Rank Approximations". In: *In 20th International Conference on Machine Learning*. AAAI Press, 2003, pp. 720–727.

[23] M. Fazel. "Matrix Rank Minimization with Applications." PhD thesis. Stanford University, 2002.

[24] S. Boyd and L. Vandenberghe. *Convex Optimization*. USA: Cambridge University Press, 2004. ISBN: 0521833787.

[25] J.-F. Cai, E. J. Candès, and Z. Shen. "A Singular Value Thresholding Algorithm for Matrix Completion". In: *SIAM Journal on Optimization* 20.4 (2010), pp. 1956–1982. DOI: 10.1137/080738970. eprint: https://doi.org/10.1137/080738970. URL: https://doi.org/10.1137/080738970.

[26] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. 2nd. USA: Cambridge University Press, 2014. ISBN: 1107077230.

[27] C. K. I. Williams and M. Seeger. "Using the Nyström Method to Speed Up Kernel Machines". In: *Advances in Neural Information Processing Systems 13*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. MIT Press, 2001, pp. 682–688. URL: http://papers.nips.cc/paper/1866-using-the-nystrom-method-to-speed-up-kernel-machines.pdf.

[28] M. Li, W. Bi, J. T. Kwok, and B. Lu. "Large-Scale Nyström Kernel Matrix Approximation Using Randomized SVD". In: *IEEE Transactions on Neural Networks and Learning Systems* 26.1 (2015), pp. 152–164.

[29] S. Kumar, M. Mohri, and A. Talwalkar. "Ensemble Nystrom Method". In: *Advances in Neural Information Processing Systems 22*. Ed. by Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta. Curran Associates, Inc., 2009, pp. 1060–1068. URL: http://papers.nips.cc/paper/3850-ensemble-nystrom-method.pdf.