Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Efficient Kernel Flows Optimization for Neural Network Induced Gaussian Process Kernels

Muhammad Waleed Bin Khalid

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Efficient Kernel Flows Optimization for Neural Network Induced Gaussian Process Kernels

Author:	Muhammad Waleed Bin Khalid
Examiner:	Dr. Felix Dietrich
Submission Date:	November 4th, 2022

I hereby declare that this thesis is entirely the result of my own work except where other-wise indicated. I have only used the resources given in the list of references.

November 4th, 2022 Muhammad Waleed Bin Khalid

Acknowledgments

First and most foremost, I would like to thank my family and friends, especially my parents for their never-ending support. Pursuing my studies at the Technical University of Munich would not have been possible without them.

I thank my advisor, Dr. Felix Dietrich, for his guidance throughout the course of this thesis. Our thorough and insightful discussions allowed me to gain knowledge regarding the topic and the research process in general. His persistent support pushed me to undertake complex tasks, incorporating a problem-solving mindset valuable for my future career. I sincerely hope my contributions help carry forward this research and assist those who venture to work on the topic.

Finally, I am grateful to Sunila Aftab for helping me review this thesis and providing valuable insights and feedback.

Abstract

A ubiquitous task in data-driven learning involves using available information to construct accurate models that can generalize to unseen input. Kernel-based learning methods are one such technique, but they rely on a good prior choice of the kernel. Often, knowledge is required regarding what type of kernel would be suitable for the available data and what kernel hyper-parameters would be needed for optimal performance. The Kernel Flows method is one learning technique that adapts the said kernel to provide the best performance for a given dataset without requiring as much expert knowledge. It works on the simple rationale that if a kernel is good, then there should not be a large change in the predictions even with reduced input data. The algorithm is available in two flavors, a parametric version where the kernel hyper-parameters are adapted, and a non-parametric one, where the input data itself is transformed to suit the base kernel.

On the same spectrum of learning techniques is the Gaussian process, more specifically the Gaussian process generated from Neural Networks. It can be shown that in the limit of infinite width, a fully connected (Dense) Neural Network (NN), and in the limit of infinite filters, a Convolutional Neural Network (CNN) is equivalent to a Gaussian process with a kernel that depends on the respective architecture. For such a Network, the kernel is parameterized by the variances of the learnable parameters, i.e., the variance of the weight and the bias, hence only two numbers per layer of the original architecture.

In this thesis, we will elaborate on both concepts and subsequently combine them by utilizing the Kernel Flows algorithm to optimize the NNGP kernels for kernel ridge regression tasks. We will explore the parametric version of the Kernel Flows algorithm with Neural Network induced Gaussian process (NNGP) kernels as the base kernels we wish to optimize. While the Kernel Flows algorithm can provide appreciable results with only a few data points, this is nevertheless computationally expensive when kernels from deep Neural Networks are involved. Hence, we will also provide efficient implementations of the proposed method which will lead us to variants of the parametric Kernel Flows algorithm that utilize different optimization techniques compared to the one used originally. Subsequently, we will compare the results of these optimized kernels along with the computational complexity involved in achieving them.

We will also explore the non-parametric version of the Kernel Flows algorithm. Particularly, we will explore the problem of unnatural perturbations of data points and poor convergence that have been highlighted in previous works to understand why such abnormalities exist, propose solutions that aim to remedy them, and test our proposed solutions.

Contents

1. Introduction

With the prominence of machine learning techniques, we have a range of methods at our disposal for data-driven learning that attempt to solve real-world problems ranging from image classification to tumor predictions. At its core, the idea is to use existing data and fit a function such that it generalizes well to unseen data.

Two such techniques are Kernel Ridge Regression and, closely related to it, Gaussian Process Regression. These techniques rely on a kernel, which can be interpreted as a similarity measure. The performance of kernel-based techniques depends heavily on the base kernel being used along with its hyper-parameters. This forces the user to either have enough prior knowledge about the underlying dataset, such that they can select the right kernel, or employ various search techniques to find the optimal setting. Both these approaches, however, have their limitations.

To tackle this issue, the Kernel Flows algorithm was developed [26]. This algorithm tries to improve the performance of the learning technique by adjusting the base kernel with the rationale that if the kernel performs well on a certain number of data points, the performance should not decrease drastically on a reduced number of data points. This is done in two ways. The first way is by changing the hyper-parameters of the kernel in question, which is referred to as the parametric version of the Kernel Flows algorithm. The second way is to perturb the data points themselves such that they adapt to the kernel. This is referred to as the non-parametric version of the algorithm. This algorithm was applied to some common yet simple kernels such as the Radial Basis Kernel in the works of [5] which demonstrated improved results, particularly in the parametric version of the algorithm.

While this improvement is beneficial, the question remains whether it is feasible to apply the algorithm to kernels that are more expensive to evaluate. A family of such kernels is those derived from a Neural Network Induced Gaussian Process (NNGP). Neural Networks (NNs) have established their importance in the field of machine learning as parametric models that can fit a wide range of data. Gaussian processes, on the other hand, are non-parametric data-driven learning tools. For a single-layer dense NN, it was shown that in the limit of infinite width of the hidden layer, the NN corresponds to a Gaussian process [24]. This result was further developed for deep neural networks [19] and Convolutional Neural Networks (CNNs) [9]. The Neural Network induced Gaussian Process kernels can be used to perform Kernel Regression tasks. These kernels, like many other widely known kernels, depend on some hyper-parameters.

In this thesis, we will explore both versions of the Kernel Flows algorithm. We will apply the parametric Kernel Flows algorithm to Gaussian Process kernels generated from both dense and Convolutional Neural Networks. We will see if significant performance improvements can be obtained at reasonable computational costs. To this end, three variations of the parametric Kernel Flows algorithm will be explained, implemented, and tested.

We will also study the non-parametric version of the Kernel Flows algorithm. We noticed that this version of the Kernel Flows algorithm leads to unnatural perturbations of data, which is an important reason for the poor performance of the algorithm on many datasets, as discussed in [5]. We build on top of the algorithm by suggesting key improvements and demonstrating how the results on synthetic datasets improve following our proposed changes.

In chapter 2 we dive into the theory surrounding the topics of Kernel Ridge Regression, and Gaussian Processes, particularly those generated from Neural Networks. We will also review the two versions of the Kernel Flows algorithm. In chapter 3 we will delve into concrete implementations and experiments and combine the theoretical concepts that we will have highlighted. Specifically, in section 3.5 we will look at three different implementations of the Parametric Kernel Flows algorithm on Neural Network Induced Gaussian process kernels assessing their comparative advantages in terms of performance, time, and space complexity. In chapter 4 we will explore the non-parametric version of the algorithm on synthetic datasets and the problems its present implementation suffers from. Subsequently, we will provide potential remedies. Finally, we will test our proposed algorithmic changes on the same datasets, following which we will make recommendations for future exploration directions.

2. Related Work

This chapter will cover the topics that are key to understanding the two versions of the Kernel Flows algorithm and the use of the parametric version on various kernels, including Neural Network induced Gaussian Process kernels. We will start with the idea of regression in section 2.1. Subsequently, we will see different types of kernels commonly used in practice and have a look at the theory of Kernel Ridge Regression.

We will then move our discussion to Gaussian Processes in section 2.2 and how they can be formulated using Dense Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). We will examine how the covariance matrix extracted from a Gaussian process can be used as a kernel in Kernel Regression. This kernel will then be the subject of our concern in the parametric version of the Kernel Flows algorithm, which will tie the two concepts together.

With a good understanding of the basics, we will move to the key algorithm that constitutes this thesis, the Kernel Flows algorithm, in section 2.3. First, we will look at the parametric version of the algorithm, which aims to find the optimal kernel hyper-parameters for a given dataset. Finally, we will study its non-parametric counterpart, which adjusts the data points to fit the base kernel.

2.1. Learning as an Interpolation Problem

Learning problems can be thought of as trying to fit a function to available data such that the function generalizes well to unseen data. We take the dataset in equation (2.1)

$$\mathcal{D} := \{(x_i, y_i) \mid \forall i = 1, \dots, N : (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}\}, \tag{2.1}$$

where $N \in \mathbb{N}$, $\mathcal{X} \in \mathbb{R}^d$ and $\mathcal{Y} \in \mathbb{R}^m$ and the target values also contain some Gaussian noise $\epsilon \sim N(0, \sigma^2)$ with σ as the standard deviation of the noise. The target values can be thought of as,

$$y_i = f(x_i) + \epsilon \text{ for } i \in \{1, \dots, N\}. \tag{2.2}$$

In our explanations, we will demonstrate the case of noise-free data. The case with noise can be explored in [29, Section 2.1.2].

2.1.1. Linear Regression

A simple model for this interpolating function would be to fit a linear function to the available data. Concretely, let us say $\mathbf{X} \in \mathbb{R}^{N \times d}$ is the design matrix containing all N number of input sample points and d being the dimensionality of each input point. The target values of the input points can be represented in the target matrix $\mathbf{Y} \in \mathbb{R}^{N\times m}$ such that m is the dimensionality of the target. We can then formulate a simple linear function $f_{linear}(\mathbf{X}) = \mathbf{XA}$ where $\mathbf{A} \in \mathbb{R}^{d\times m}$ is the matrix of coefficients. A bias weight is also included by augmenting the design matrix $\mathbf{X}$ with additional elements equal to one [29, Section 2.1.1]. Note that our representation looks slightly different from the representation in [29, Section 2.1.1 - 2.1.3] since we use the design matrix $\mathbf{X} \in \mathbb{R}^{N\times d}$ instead of its transpose. The matrix of coefficients can be found by minimizing the difference between the norm of the predictions and the targets, as shown in equation (2.3)

$$\min_{\mathbf{A}} \|\mathbf{Y} - \mathbf{XA}\|^2 + \lambda\|\mathbf{A}\,\|^2. \tag{2.3}$$

Since a regularization term, λ has been added, strictly speaking, we are performing ridge regression. While this way of interpolating data is simple to perform and easy to analyze, it is limited since a linear function can not approximate a complex dataset well.

2.1.2. Kernel Ridge Regression

Since the linear model has limited expressiveness, we add non-linearities with the help of some basis functions ϕ_i to allow us to map from the input space to a feature space. We can then apply the linear model in this feature space instead. Our non-linear approximation function becomes, $f_{\text{non-linear}}(\mathbf{X}) = \Phi(\mathbf{X})\mathbf{A}$ where $\Phi(\mathbf{X}) = (\phi_1(\mathbf{X}), \phi_2(\mathbf{X}), \ldots, \phi_L(\mathbf{X}))^T \in \mathbb{R}^{N\times L}$ projects our input data points from d dimensional input space to L dimensional feature space. The resulting minimization problem is very similar to 2.3, i.e. $\min_{\mathbf{A}} \|\mathbf{Y} - \Phi(\mathbf{X})\mathbf{A}\|^2 + \lambda\|\mathbf{A}\,\|^2$ where $\mathbf{A} \in \mathbb{R}^{L\times m}$.

This problem can be solved using the least squares method to obtain,

$$\mathbf{A} = (\Phi(\mathbf{X})^\top\Phi(\mathbf{X}) + \lambda\mathbf{I_N})^{-1}\Phi(\mathbf{X})^\top\mathbf{Y}, \tag{2.4}$$

where λ is the regularization term. The prediction for some test points, $\mathbf{X}_*$ becomes,

$$f_{\text{non-linear}} = \Phi(\mathbf{X}_*)(\Phi(\mathbf{X})^\top\Phi(\mathbf{X}) + \lambda\mathbf{I_N})^{-1}\Phi(\mathbf{X})^\top\mathbf{Y}. \tag{2.5}$$

However, we observe that the computation of the basis functions at each point can be prohibitively expensive. Thus, we look for an alternate way of projecting our data. Particularly we note that $(\Phi(\mathbf{X})^\top\Phi(\mathbf{X}))^{-1}\Phi(\mathbf{X})^\top = \Phi(\mathbf{X})^\top(\Phi(\mathbf{X})\Phi(\mathbf{X})^\top)^{-1}$ [1]. This gives us the alternate form [29]

$$f_{\text{non-linear}} = \Phi(\mathbf{X}_*)\Phi(\mathbf{X})^\top(\Phi(\mathbf{X})^\top\Phi(\mathbf{X}) + \lambda\mathbf{I_N})^{-1}\mathbf{Y}. \tag{2.6}$$

[1]This is because

$$\Phi^\top\Phi\Phi^\top = \Phi^\top\Phi\Phi^\top$$
$$(\Phi^\top\Phi)^{-1}\Phi^\top(\Phi\Phi^\top)(\Phi\Phi^\top)^{-1} = (\Phi^\top\Phi)^{-1}(\Phi^\top\Phi)\Phi^\top(\Phi\Phi^\top)^{-1}$$
$$(\Phi^\top\Phi)^{-1}\Phi^\top = \Phi^\top(\Phi\Phi^\top)^{-1}$$

With this alternate form at hand, we look at the result of an inner product of a basis function evaluated at two different input points $\Phi\left(x_i\right)\Phi\left(x_j\right)^T$. This collection of inner products is referred to as the Gram matrix $\Theta \in \mathbb{R}^{N\times N}$ defined as

$$\Theta_{i,j} := \Phi\left(x_i\right)\Phi\left(x_j\right)^T . \tag{2.7}$$

Provided two input points are not coincident, the matrix is positive definite and symmetric by construction of the dot product since $\mathbf{c}^\top\Theta\mathbf{c} \geq 0$ for all $\mathbf{c} \in \mathbb{R}^N$ [31].

We can define a kernel $\mathbf{K} \in \mathbb{R}^{N\times N}$ that is a similarity measure mapping two inputs to a real number such that $K(\mathbf{X}, \mathbf{X}') : \mathbb{R}^{N\times d} \times \mathbb{R}^{N\times d} \to \mathbb{R}^{N\times N}$ and $\mathbf{K}_{i,j} = K(x_i, x_j)$. Since $K(x, x') = K(x', x)$ for all x and x', the resulting kernel matrix is symmetric. Ensuring that the Kernel is also positive definite, the Gram matrix can be seen as a kernel where the dot product of the basis functions at two points correspond to an entry in the kernel matrix i.e. $K(x_i, x_j) = \langle\phi(x_i)\ , \phi(x_j)\ \rangle$. This substitution of the dot product evaluation with the kernel matrix is referred to as the Kernel trick [12]. This is very useful since now we do not have to compute the individual basis functions and can simply work with the kernel directly, saving us a lot of computational effort.

Given some training data with its target and some test data $\mathbf{X}_*$ we wish to predict the target values for, the resulting solution becomes

$$f(\mathbf{X}_*) = K(\mathbf{X}_*, \mathbf{X})\left(K(\mathbf{X}, \mathbf{X}) + \lambda \boldsymbol{I}_N\right)^{-1}\mathbf{Y}. \tag{2.8}$$

Note that a regularization term λ is added before inverting the kernel. This substitution and the final form of the equation can also be justified using the Representer theorem [31, p. 420].

There are a variety of kernels that can be applied for Kernel Ridge Regression that represent certain sets of feature maps. For certain kernels, the feature maps need not even project the data into a finite-dimensional space. Kernels often have parameters of their own which have a significant effect on the accuracy of the interpolated results. We call these parameters the hyper-parameters of the kernel. A common kernel that is used in machine learning is the Radial Basis Function (RBF) or the Gaussian kernel which is given by

$$K\left(x, x'\right) := \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right), \tag{2.9}$$

where $\sigma > 0$ is a hyper-parameter that has to be selected by the user. This kernel represents the feature map of monomials $\phi(x) = (1, x, x^2, x^3, ...)$ and shows how feature maps need not always map to finite dimensional spaces when using kernels.

2.2. Gaussian Process Kernels from Neural Networks

In this section, we will study the Gaussian processes that can be induced by neural networks. We will first look at Gaussian Processes in general, which is a different perspective on Kernel Ridge Regression that we looked at earlier. Then we will look at how Gaussian Processes can be generated from both dense NNs in section 2.2.2 and CNNs in section 2.2.3.

2.2.1. Gaussian Processes

In this section, we will elaborate on Gaussian processes. This is an alternate perspective on the kernel regression problem discussed in section 2.1.2. There are two views on Gaussian processes: the functional view and the weight-space view. We will discuss only the functional view in this explanation, as the weight space view is similar to the ideas presented in the previous section on kernel regression.

A Gaussian process is a collection of random variables, where any finite number of these variables has a joint Gaussian distribution [29, Definition 2.1]. In our case, the random variable will be the value of our target function $f(x)$ at the location x. We can define a Gaussian process by a mean and a covariance function. Consider a real process $f(x)$, then we define its mean and covariance functions as

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)], \\ k\left(x, x^{\prime}\right) &= \mathbb{E}\left[(f(x)-m(x))\left(f\left(x^{\prime}\right)-m\left(x^{\prime}\right)\right)\right], \end{aligned} \tag{2.10}$$

which generates the Gaussian process

$$f(x) \sim \mathcal{GP}\left(m(x), k\left(x, x^{\prime}\right)\right). \tag{2.11}$$

We will fix the mean function to zero to define the Gaussian Process only using the covariance for simplicity. We will now take the example mentioned in section 2.1.2 for a Gaussian process to illustrate the concept. This is the Bayesian linear regression model. The Bayesian linear regression model uses a prior on a set of weights $w \sim \mathcal{N}\left(0, \alpha^2 I\right)$ to predict

$$f(\mathbf{X}) = \Phi(\mathbf{X})^T w. \tag{2.12}$$

As $f(\mathbf{X})$ is a linear combination of Gaussian variables, the result has a multivariate Gaussian Distribution with the mean and covariance defined as

$$\mathbb{E}[f(\mathbf{X})] = \Phi(\mathbf{X})^{\top}\mathbb{E}[w] = 0. \tag{2.13}$$

$f(\mathbf{X})$ and $f(\mathbf{X}_*)$ are jointly Gaussian with zero mean and the given covariance. A covariance function needs to be defined so that individual basis functions need not be computed and thus we can make the substitution of $K(\mathbf{X}, \mathbf{X}_*) = \alpha^2\Phi(\mathbf{X})^{\top}\Phi\left(\mathbf{X}^{\prime}\right)$. As discussed before, one commonly used kernel is the radial basis function (RBF) or Gaussian Kernel function given by equation (2.9). This covariance function produces values close to 1 for

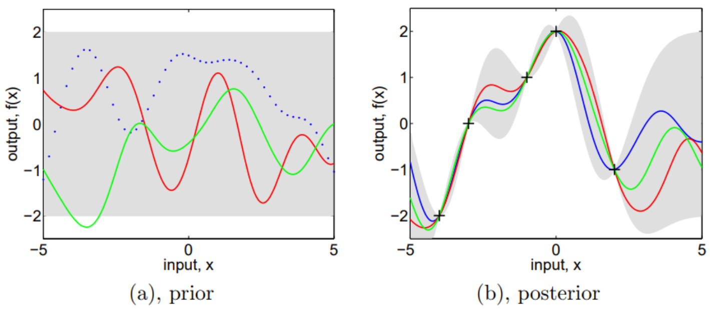


Figure 2.1.: a) Three functions are drawn randomly from a Gaussian Process prior. The dots indicate actual y values. b) Three random functions are drawn from the posterior, i.e. the prior conditioned on the five noise-free observations. The shaded area represents the point-wise mean plus and minus two times the standard deviation for each input value. Taken from [29].

inputs that are close and smaller values otherwise, giving a smoothing property in some sense.

We can now draw samples from our prior distribution of $\mathbf{f}_* \sim \mathcal{N}\left(\mathbf{0}, K_{RBF}\left(\mathbf{X}_*, \mathbf{X}_*\right)\right)$ for a set of N_* test points $\mathbf{X}_*$. This can be seen in panel a) of figure 2.1. Assuming noise-free observations we evaluate the value of our function at N training points which we denote by $\mathbf{X}$ and calculate the joint distribution of the training and test points as

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{X},\mathbf{X}) & K\left(\mathbf{X},\mathbf{X}_*\right) \\ K\left(\mathbf{X}_*,\mathbf{X}\right) & K\left(\mathbf{X}_*,\mathbf{X}_*\right) \end{bmatrix}\right), \tag{2.14}$$

where each of $K(.,.)$ is the kernel evaluation between the set of points. For the N training points $K(\mathbf{X}_*,\mathbf{X}) \in \mathbb{R}^{N_* \times N}$ and likewise for the others. Informally, we can think of calculating the posterior as selecting only those functions that pass through the training points. This can be done by conditioning the joint distribution on the observed training data points, giving us the posterior probability distribution

$$\begin{split} \mathbf{f}_* \mid \mathbf{X}_*, \mathbf{X}, \mathbf{f} \sim \mathcal{N}(& K\left(\mathbf{X}_*,\mathbf{X}\right) K(\mathbf{X},\mathbf{X})^{-1}\mathbf{f} \\ & K\left(\mathbf{X}_*,\mathbf{X}_*\right) - K\left(\mathbf{X}_*,\mathbf{X}\right) K(\mathbf{X},\mathbf{X})^{-1} K\left(\mathbf{X},\mathbf{X}_*\right)). \end{split} \tag{2.15}$$

We can now generate samples from this new posterior distribution, which is shown in panel b) of figure 2.1. We can see that the uncertainty drops to 0 at the observed training points and all the drawn functions pass through these points.

Most covariance functions have parameters that can be varied. Similar to the kernel parameters, we refer to these parameters as the hyper-parameters of the covariance function or kernel. The values of these hyper-parameters play a crucial role in the functions that are sampled and the subsequent performance of the Gaussian Process regressor. In this thesis, we will soon discuss how we can extract a good kernel function by optimizing the hyper-parameters of a Neural Network-induced Gaussian process by combining it with the parametric Kernel Flows algorithm.

2.2.2. Deep Dense Neural Networks Induced Gaussian Process

Having understood the basic concepts behind Gaussian processes, we will extend our knowledge to Gaussian processes generated from Neural Networks. We will look at how an infinitely wide, fully connected neural network is in fact a Gaussian process. In the case of a single-layer neural network, it was proven that in the limit of infinite width, the network approaches a Gaussian process [24]. The result was extended to arbitrarily deep dense neural networks [22]. We will first look at the case for a single-layered NN and subsequently extend the line of thought to deep dense NNs.

Single Layer NN induced GP

We briefly go over the argument presented by [24] for single-layer neural networks as the explanation for deeper networks is built on the induction of this base case. Given some input $x \in \mathbb{R}^{d_{in}}$, a single layer neural network with hidden layer width of N_1, the i^{th} output of a single layer neural network, for some activation function (non-linearity) ϕ can be written as

$$\begin{aligned} x_j^1(x) &= \phi\left(b_j^0 + \sum_{k=1}^{d_{in}} W_{jk}^0 x_k\right), \\ z_i^1(x) &= b_i^1 + \sum_{j=1}^{N_1} W_{ij}^1 x_j^1(x), \end{aligned} \tag{2.16}$$

where the weights W and biases b are independent and identically distributed with zero mean and a variance of σ_w^2/N_1 and σ_b^2 respectively. As the post activations, x^1 are independent for non-coincidental points, and the output of the network is composed of a sum of these i.i.d terms, using the Central Limit Theorem by increasing the width of the hidden layer, the output of the neural network follows the Gaussian distribution $z_i^1 \sim \mathcal{GP}\left(\mu^1, K^1\right)$. Since the parameters of the network have zero mean, it follows that $\mu^1(x) = \mathbb{E}\left[z_i^1(x)\right] = 0$. The covariance can be represented as [24]

$$K^1\left(x, x'\right) \equiv \mathbb{E}\left[z_i^1(x) z_i^1\left(x'\right)\right] = \sigma_b^2 + \sigma_w^2 \mathbb{E}\left[x_i^1(x) x_i^1\left(x'\right)\right]. \tag{2.17}$$

Deep Neural Network induced Gaussian Process

Lee et al built on this base case to prove that the demonstration of generating a Gaussian process from a single-layer Neural Network can be extended to a deep neural network [19]. We will follow their arguments closely. The idea is to take the hidden layers to be of infinite widths in succession and apply proof by induction.

A result of some layer l of a dense neural network can be represented similarly to

$$z_i^l(x) = b_i^l + \sum_{j=1}^{N_l} W_{ij}^l x_j^l(x), \quad x_j^l(x) = \phi\left(z_j^{l-1}(x)\right). \tag{2.18}$$

All outputs z^l are again sums of i.i.d random terms so if we successively increase each layer to an infinite width then the output would be a joint multivariate Gaussian distribution, i.e. $z_i^l \sim \mathcal{GP}\left(0, K^l\right)$. It has been shown that the final covariance matrix depends on the covariance matrices of the previous layer namely on $K^{l-1}(x,x')$, $K^{l-1}(x',x')$ and $K^{l-1}(x,x)$ resulting in the shorthand [19]

$$K^l\left(x,x'\right) = \sigma_b^2 + \sigma_w^2 F_\phi\left(K^{l-1}\left(x,x'\right), K^{l-1}(x,x), K^{l-1}\left(x',x'\right)\right). \tag{2.19}$$

Here F_ϕ represents the covariance from the activation used in the neural network and for the first layer it is represented as

$$K^0\left(x,x'\right) = \mathbb{E}\left[z_j^0(x) z_j^0\left(x'\right)\right] = \sigma_b^2 + \sigma_w^2\left(\frac{x \cdot x'}{d_{\text{in}}}\right). \tag{2.20}$$

This base case has an important consequence as it means that uncentered data will result in a very large covariance, which will affect the result. This will be demonstrated in section 3.4.

2.2.3. Deep Convolutional Neural Networks Induced Gaussian Process

We have up till now seen the use of dense neural networks and the Gaussian process kernel derived from them. Fully connected neural networks have their limitations, particularly when it comes to tasks related to image processing and machine vision. They are unable to exploit the structures that exist in images. Moreover, they consume excessive memory since each neuron is connected to every other neuron in neighboring layers. For tasks such as image classification, Convolutional Neural Networks (CNNs) have proven to be significantly more efficient. We ask ourselves if these CNNs have a similar representation such that a Gaussian Process can be derived from a given CNN architecture. As we will see shortly, this is indeed possible in the limit that each hidden layer of the CNN has an infinite number of filters [10]. Before we understand how CNNs can be represented as Gaussian processes, we will first have a brief look into how a CNN functions.

Convolutional Neural Networks

For tasks that involve learning from datasets composed of images, Convolutional Neural Networks have proven to be a very efficient way to learn. They were presented as an alternative to dense neural networks that performed poorly on image datasets and were

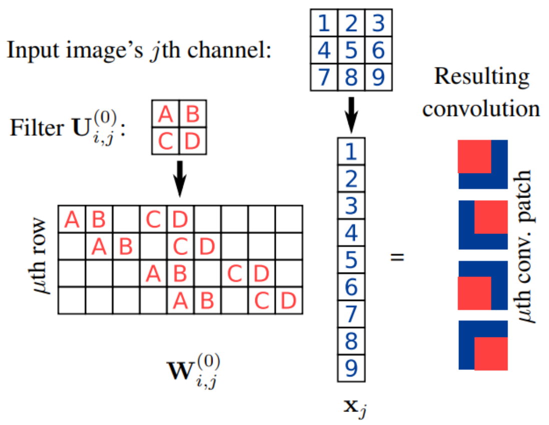


Figure 2.2.: Representation of a 2D convolution as a dot product where the blanks represent zeros. Taken from [9].

computationally more expensive. This is because convolutional neural networks can make use of inherent structures present in images [16, 18].

CNNs convolve the input image with a kernel (different from the kernels optimized by the Kernel Flows algorithm for kernel regression) that is smaller or equal to the image in width and height but with the same number of channels, to provide an output for the next layer of the network. This results in a feature map. Since the parameters of a kernel are fixed for a particular forward pass, fewer parameters must be optimized and stored, making the technique efficient. This is referred to as weight sharing. The feature map is passed through a non-linearity or an activation function similar to those in dense neural networks.

Let us have a look at CNNs more mathematically. A CNN takes as input an image $\mathbf{X} \in \mathbb{R}^{C^{(0)} \times (H^{(0)} D^{(0)})}$ where $H^{(0)}$ is the height of the image, $D^{(0)}$ the width and $C^{(0)}$ the number of color channels in the image (for example 3 for the case of RGB images). We take $\mathbf{U}^l$ to be the kernel of a particular layer l. The convolution can be transformed to an alternate representation of the kernel as $\mathbf{W}^l$ and using $\mathbf{x_i}$ as the flattened vector of the i^{th} channel of the image. This is represented clearly in figure 2.2.

The feature map provided by the first layers comes out as

$$\mathbf{a}_i^{(1)}(\mathbf{X}) := b_i^{(1)} \mathbf{1} + \sum_{j=1}^{C^{(0)}} \mathbf{W}_{i,j}^{(1)} \mathbf{x}_j. \tag{2.21}$$

An activation function is applied to the resulting feature map, and we get the recursion for all remaining layers as [9]

$$\mathbf{a}_i^{(\ell+1)}(\mathbf{X}) := b_i^{(\ell+1)}\mathbf{1} + \sum_{j=1}^{C^{(\ell)}} \mathbf{W}_{i,j}^{(\ell+1)} \phi\left(\mathbf{a}_j^{(\ell)}(\mathbf{X})\right). \tag{2.22}$$

$\mathbf{a}_i^{(\ell+1)}(\mathbf{X})$ represents a flattened version of the activation feature map $\mathbf{A}^{(\ell)}(\mathbf{X}) \in \mathbb{R}^{C^{(\ell)} \times \left(H^{(\ell)} D^{(\ell)}\right)}$. The weights and biases of this CNN are drawn from a Gaussian distribution as $U_{i,j,x,y}^{(l)} \sim \mathcal{N}\left(0, \frac{\sigma_w^2}{C^{(l)}}\right)$ and $b_i^{(l)} \sim \mathcal{N}\left(0, \sigma_b^2\right)$. In the case of classification or regression, the final layer can have an output such that the width and height are equal to one, so in effect, we have a dense final layer.

Convolutional Neural Networks Induced Gaussian Process

We will now discuss the equivalent Gaussian Process for Convolutional Neural Networks. This will be referred to as the Convolutional Neural Network Gaussian Process (CNNGP). Our explanation will closely follow that of Garriga-Alonso, Rasmussen, and Aitchison [9] who in turn build on top of [19, 6]. We saw how, for Dense NN the proof by induction follows by taking the width of the hidden layers to infinity. Similarly, here we take the number of channels of each layer to infinity. For the first layer, the result $\mathbf{a}_j^{(1)}(\mathbf{X}, \mathbf{X}')$ is a multivariate Gaussian joint distribution. This is because the parameters of different feature maps are themselves i.i.d as well. This base case can be extended to subsequent layers, as in equation (2.22) $b_i^{(\ell+1)}$ is i.i.d Gaussian and the second term also becomes Gaussian in the limit of $C^{(\ell)} \to \infty$ as the input from the previous layer is i.i.d by assumption and the weights are i.i.d by definition. The output feature maps after activation are also i.i.d as the results are jointly Gaussian and uncorrelated because the weights are independent with zero-mean, which removes correlations from the activation function [9, Section 2.2].

Let us now work towards obtaining the mean and the covariance obtained from the CNNGP. As suggested by Garriga-Alonso, Rasmussen, and Aitchison, we consider equation (2.22) in a more verbose fashion as

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

Here $i, j \in \left\{1, \ldots, C^{(l+1)}\right\}$ indicate the input and output channels and $\nu, \mu \in \left\{1, \ldots, H^{(l+1)} D^{(l+1)}\right\}$ denote the location in the feature map. Then, taking the expectation over equation (2.23) we get the mean function as

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

This is because both the biases and weights have a zero mean, and the weights are independent of the activation results from the previous layer. We now look at the covariance matrix of the CNNGP. Here the authors use an efficient method which allows us to avoid computing the covariance of the activations between all pairs of locations in a feature map ($\mathbb{C}\left[A_{i,\mu}^{(\ell+1)}(\mathbf{X}), A_{i,\mu'}^{(\ell+1)}(\mathbf{X}')\right]$ for $\mu, \mu' \in \left\{1, \ldots, H^{(\ell+1)} D^{(\ell+1)}\right\}$) which would be very high dimensional i.e. $N^2\left(H^{(\ell+1)} D^{(\ell+1)}\right)^2$. Instead, they consider only the diagonal covariance ($\mathbb{C}\left[A_{i,\mu}^{(\ell+1)}(\mathbf{X}), A_{i,\mu}^{(\ell+1)}(\mathbf{X}')\right]$ for $\mu \in \left\{1, \ldots, H^{(\ell+1)} D^{(\ell+1)}\right\}$) which is of size $N^2\left(H^{(\ell+1)} D^{(\ell+1)}\right)$. This holds true since the outputs for regression and classification tasks have a width and height of one, with the number of channels equal to the number of output classes. Hence, the covariance is only required at these locations, which in turn means we only require the covariance at the corresponding locations in the input as well [9, Section 3.3]. Taking the covariance of two inputs at a certain layer, we get

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

The first line is a consequence of the fact that for a sum of terms, the resulting covariance is the sum of individual covariances. The second step is valid since the terms in the covariance have a mean of zero and the weights and activations from the previous layer are independent. As the weights for different channels are independent, their joint expected value is zero ($\mathbb{E}\left[W_{i,j,\mu,\nu}^{(\ell+1)} W_{i,j',\mu,\nu'}^{(\ell+1)}\right] = 0$ for $j \neq j'$). Moreover, for each row μ the weights matrices have either independent variables or zeros. Due to this, the sums over v' and j' can be dropped, which gives us the last line [9, Section 3.1]. The sum over v can be restricted to the patch the kernel is applied to. The covariance of the first layer comes out as

$$K_\mu^{(1)}\left(\mathbf{X}, \mathbf{X}'\right) = \mathbb{C}\left[A_{i,\mu}^{(1)}(\mathbf{X}), A_{i,\mu}^{(1)}\left(\mathbf{X}'\right)\right] = \sigma_{\mathrm{b}}^2 + \frac{\sigma_{\mathrm{w}}^2}{C^{(0)}} \sum_{i=1}^{C^{(0)}} \sum_{\nu \in \mu \text{ th patch}} X_{i,\nu} X'_{i,\nu}, \tag{2.26}$$

and for subsequent layers,

$$K_\mu^{(\ell+1)}\left(\mathbf{X}, \mathbf{X}'\right) = \mathbb{C}\left[A_{i,\mu}^{(\ell+1)}(\mathbf{X}), A_{i,\mu}^{(\ell+1)}\left(\mathbf{X}'\right)\right] = \sigma_{\mathrm{b}}^2 + \sigma_{\mathrm{w}}^2 \sum_{\nu \in \mu \text{ th patch}} V_\nu^{(\ell)}\left(\mathbf{X}, \mathbf{X}'\right). \tag{2.27}$$

Note that here $V_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')$ is the covariance of the activation. The expression in equation (2.27) can be evaluated as a convolution [9, 33], i.e.

$$\begin{aligned}&\begin{bmatrix} K_1^{(l+1)}(\mathbf{X},\mathbf{X}') & \dots & K_n^{(l+1)}(\mathbf{X},\mathbf{X}') \\ \vdots & \ddots & \vdots \\ K_{(m-1)\cdot n+1}^{(l+1)}(\mathbf{X},\mathbf{X}') & \dots & K_{n\cdot m}^{(l+1)}(\mathbf{X},\mathbf{X}') \end{bmatrix} \\ &= \sigma_b^2 + \sigma_w^2 \begin{bmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{bmatrix} * \begin{bmatrix} V_1^{(l)}(\mathbf{X},\mathbf{X}') & \dots & V_k^{(l)}(\mathbf{X},\mathbf{X}') \\ \vdots & \ddots & \vdots \\ V_{(o-1)\cdot k+1}^{(l)}(\mathbf{X},\mathbf{X}') & \dots & V_{o\cdot k}^{(l)}(\mathbf{X},\mathbf{X}') \end{bmatrix}.\end{aligned} \tag{2.28}$$

This adds to the efficiency with which the kernel for the CNNGP can be evaluated.

Covariance of Activation Function ReLU

As mentioned in section 2.2.2, the covariance depends only on the activation function kernels $K_\mu^{(\ell+1)}(\mathbf{X},\mathbf{X})$, $K_\mu^{(\ell+1)}(\mathbf{X},\mathbf{X}')$ and $K_\mu^{(\ell+1)}(\mathbf{X}',\mathbf{X}')$. One of the most common activation functions used in both dense NN and CNNs is the ReLU activation. This is an element-wise one-to-one map if the input is positive and otherwise zero, and its covariance can be computed in closed form. This was derived by Youngmin Cho, Lawrence K Saul, and Matthews et al [4, 22] as

$$\begin{aligned}V_\nu^{(\ell)}(\mathbf{X},\mathbf{X}') &= \frac{\sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')}}{\pi}\left(\sin\theta_\nu^{(\ell)} + \left(\pi - \theta_\nu^{(\ell)}\right)\cos\theta_\nu^{(\ell)}\right), \\ \text{where } \theta_\nu^{(\ell)} &= \cos^{-1}\left(K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}') / \sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')}\right).\end{aligned} \tag{2.29}$$

The covariance of other activation functions can also be numerically computed when a closed form is not available [27]. Since the covariance of one layer depends on the output of the previous layer, the covariance kernel can be evaluated within the cost of the forward pass of the equivalent CNN.

2.3. Kernel Flows

In this section, we will discuss the two versions of the Kernel Flows algorithm. Before we dive into the intricacies of this algorithm, we reiterate that when performing Kernel Ridge Regression, the performance is highly contingent on the kernels used. For a particular parameterized kernel, the appropriate selection of hyper-parameters is critical. The selection of these hyper-parameters is usually done using search techniques such as random and grid searches. This, however, becomes increasingly infeasible when the cost of computing the kernel on large datasets is high and the kernel incorporates multiple parameters due to the curse of dimensionality. The parametric version of the Kernel Flows algorithm helps us tackle this issue by providing us with a good kernel based on the rationale that if the kernel performs well on a certain number of data points, its performance should also be good on a reduced number of data points measured using the intrinsic Reduced Kernel Hilbert Space (RKHS) norm associated with the kernel [26]. Alternatively, the non-parametric version of the Kernel Flows algorithm transforms the data points themselves such that they fit the given kernel well. This transformation is done based on the same rationale provided for the parametric version.

2.3.1. Measuring the Kernel Quality

The question remains as to how the quality of the kernel can be measured. We will follow the explanation provided by Owhadi and Yoo closely [26]. To understand this, let us say that we wish to find a function $u^\dagger$ such that it interpolates the data in (2.1) as

$$u^\dagger\left(x_i\right) = y_i \text{ for } i \in \{1, \dots, N\}. \tag{2.30}$$

We restrict the choice of u to a space of functions $\mathcal{B}$ with a norm $\| \ . \ \|$. This gives us the optimal recovery as the minimization of the relative error

$$\min_v \max_u \frac{\|u-v\|^2}{\|u\|^2}, \tag{2.31}$$

where $v \in \mathcal{B}$ are the candidates such that $v(x_i) = u(x_i)$. In order to satisfy the condition $u(x_i) = y_i$, the dual space $\mathcal{B}^*$ of $\mathcal{B}$ must contain the Dirac delta functions $\phi_i(.) = \delta(. - x_i)$. If we consider the norm $\| \ . \ \|$ to be quadratic, i.e. $\|u\|^2 = \left\langle Q^{-1}u, u\right\rangle$ where $Q : \mathcal{B}^* \to \mathcal{B}$ is a positive symmetric linear bijection, meaning $\langle \phi, Q\phi\rangle \geq 0$ and $\langle \phi, Q\psi\rangle = \langle \psi, Q\phi\rangle$ for $\psi, \phi \in \mathcal{B}^*$, then equation (2.31) has an explicit form

$$v^\dagger = \sum_{i,j=1}^{N} y_i A_{i,j} Q\phi_j. \tag{2.32}$$

Here $A = \Theta^{-1}$ and Θ is the $N \times N$ Gram matrix containing $\Theta_{i,j} = \langle \phi_i, Q\phi_j\rangle$.
Now if we define our kernel K as

$$K\left(x, x'\right) = \left\langle \delta(\cdot - x), Q\delta\left(\cdot - x'\right)\right\rangle, \tag{2.33}$$

then $(\mathcal{B}, \| \cdot \|)$ is equivalent to a RKHS with the norm

$$\|u\|^2 = \sup_{\phi \in \mathcal{B}^*} \frac{\left(\int \phi(x)u(x)dx\right)^2}{\int \phi(x)K(x,y)\phi(y)dxdy}. \tag{2.34}$$

With this, equation (2.32) corresponds to

$$v^{\dagger}(\cdot) = y^T A K(\mathbf{X}, \cdot). \tag{2.35}$$

Here, $\Theta_{i,j} = K(x_i, x_j)$. This corresponds to the solution following the representer theorem and also the expectation of the Gaussian process model [31].

Now we have all the tools to apply the rationale of the Kernel Flows algorithm in a concrete fashion. To this end, we construct an additional dataset that is a subset of the original dataset presented in (2.1). Let $\mathbf{S} = \{s_1, s_2, \dots, s_n\} \subset \{1, \dots, N\}$ so that $n < N$ then we can define $\mathcal{D}_s := \{(x_i, y_i) \mid \forall i \in \mathbf{S}\}$. Hence, $\mathcal{D}_s \subset \mathcal{D}$. The optimal recovery v^s of $u^{\dagger}$ upon seeing only the subset consisting of n datapoints gives us $v^s(\cdot) = \sum_{i=1}^{n} y_{s_i} \bar{A}_{i,j} K\left(x_{s_j}, .\right)$ and $\bar{A} = \bar{\Theta}^{-1}$ with $\bar{\Theta}_{i,j} = \Theta_{s_i,s_j}$. We can define a matrix $\pi \in \mathbb{R}^{n \times N}$ such that $\pi_{i,j} = \delta_{s_i,j}$ to give us the prediction as

$$v^s = y^T \tilde{A} K(\mathbf{X}_{\mathcal{D}_s}, \cdot), \tag{2.36}$$

where $\mathbf{X}_{\mathcal{D}_s}$ is the subset of data we discussed and $\tilde{A} = \pi^T \bar{A} \pi$ and $\bar{A} = \left(\pi \Theta \pi^T\right)^{-1}$. This is so that the Kernel is only evaluated once.

As mentioned, the quality of the Kernel can be measured by comparing the relative accuracies of the two datasets, one of which is strictly smaller than the other. Thus, we can define ρ as

$$\rho := \frac{\left\|v^{\dagger} - v^s\right\|^2}{\|v^{\dagger}\|^2}. \tag{2.37}$$

Following [26, Proposition 3.1], assuming that both $v^{\dagger}$ and v^s predict their own input dataset, the A norm becomes

$$||v^{\dagger}||^2 = (y^T A K(\mathbf{X}, \mathbf{X}))(y^T A K(\mathbf{X}, \mathbf{X}))^T = y^T A y, \tag{2.38}$$

and similarly, for v^s. We can use the orthogonal decomposition of $\left\|v^{\dagger}\right\|^2$ as $\left\|v^{\dagger}\right\|^2 = \|v^s\|^2 + \left\|v^{\dagger} - v^s\right\|^2$. We get an explicit equation to calculate ρ as

$$\rho = 1 - \frac{y^T \tilde{A} y}{y^T A y}. \tag{2.39}$$

We can see that if ρ is close to zero, it means that the evaluation using the subset of data was close to that using the entire data, suggesting that the kernel is good as it generalizes well even for smaller dataset sizes. Note that the above representation works only if $y \in \mathbb{R}^1$. Otherwise, we evaluate the trace of the matrices in the numerator and denominator following [26, Remark 6.1].

2.3.2. Parametric Kernel Flows

With a measure of kernel quality from the RKHS norm, we can now dive into the concrete algorithm of Kernel Flows. In this section, we will look at the parametric version of the algorithm. This takes the kernel to be parameterized by a set of hyper-parameters and subsequently tries to optimize the kernel by adjusting these parameters.

We choose a finite-dimensional linear space $\mathcal{W}$ and parameterize our kernel such that $K(x, x', W)$ where $W \in \mathcal{W}$. We again use the principal dataset mentioned in (2.1) but create two subsets in a similar way as we did in section 2.3.1. We take $N_c < N_f \leq N$ where N is the total dataset size, N_f is the batch size and N_c is the number of samples taken from the batch, (usually $N_c = N_f/2$). The subsets from the original dataset follow the indices $\mathbf{S_f} = \{s_{f,1}, s_{f,2}, \ldots, s_{f,N_f}\} \subset \{1, \ldots, N\}$ and $\mathbf{S_c} = \{s_{c,1}, s_{c,2}, \ldots, s_{c,N_c}\} \subset \mathbf{S_f}$. This gives us our datasets $\mathcal{D}_f := \{(x_i, y_i) \mid \forall i \in \mathbf{S_f}\}$ which is also referred to as the batch dataset and $\mathcal{D}_c := \{(x_i, y_i) \mid \forall i \in \mathbf{S_c}\}$ which is referred to as the sample dataset.

As before, we can define a sub-sampling matrix $\pi \in \mathbb{R}^{N_c \times N_f}$ such that $\pi_{i,j} = \delta_{s_{c,i},j}$ and the sub-sampled targets $y_f \in \mathbb{R}^{N_f}$ and $y_c \in \mathbb{R}^{N_c}$. We use the shorthand to represent our evaluated kernel $\Theta_{i,j}(W) = K\left(x_{s_{f,i}}, x_{s_{f,j}}, W\right)$. The Kernel for the sample data subset is then implicitly contained in the kernel evaluated for the batch dataset, which allows efficiency since we need not calculate the kernel twice and instead evaluate $\pi\Theta\pi^T$. Inserting all of this into equation (2.39) we get

$$\rho\left(W, \mathbf{X}, \mathbf{Y}, s_f, s_c\right) := 1 - \frac{y_c^T \left(\pi\Theta\pi^T\right)^{-1} y_c}{y_f^T \Theta^{-1} y_f}. \tag{2.40}$$

Note that for tasks such as classification where the second term would not be a scalar, we take the trace of the numerator and denominator as explained in [26, Remark 6.1].

ρ is now the objective function that we wish to minimize, and this can be done using gradient descent. [26] provide an explicit equation for the Fréchet derivative of ρ with respect to the parameters of the Kernel as

$$\partial_{W_i}\rho(W) = -\frac{(1-\rho(W))\hat{y}^T \left(\partial_{W_i}\Theta(W)\right)\hat{y} - \hat{z}^T \left(\partial_{W_i}\Theta(W)\right)\hat{z}}{y_f^T \Theta^{-1} y_f}, \tag{2.41}$$

where $\hat{y} := \Theta^{-1} y_f$, $\hat{z} := \pi^T \left(\pi\Theta\pi^T\right)^{-1} \pi y_f$ and W_i is a single entry in the set of parameters that define a kernel [26, Corollary 4.1]. The final algorithm becomes:

The founding paper uses gradient descent as the optimization method, so that the last step of algorithm 1 is

$$W_{i+1} = W_i - \eta \nabla_W \rho\left(W, s_f, s_c\right),$$

where η is the learning rate set by the user. Darcy improves this to utilize Gradient Descent With Nesterov Momentum to improve performance [5, 32]. We will soon see that other optimization methods can be applied and, depending on the kernel involved, might be significantly more performant and efficient than gradient descent.

Algorithm 1 Kernel Flows with a Parametric Family of Kernels

```
Initialize W
for $i = 1 \to MaxIter$ do
    Create batch $\mathcal{D}_{N_f}$ from the total dataset uniformly sampled without replacement.
    Create sample $\mathcal{D}_{N_c}$ from batch uniformly sampled without replacement.
    Calculate $\rho$ using (2.40)
    $W_{i+1} = \text{Optimization Method}(W_i, \rho, \mathcal{D}_{N_c}, \mathcal{D}_{N_f})$
end for
```

2.3.3. Non-Parametric Kernel Flows

In this section we will look at the second variant of the Kernel Flows algorithm, i.e., the non-parametric Kernel Flows algorithm. This perturbs the input data points using the same rationale as the parametric version of the algorithm. Thus, given a base kernel, with successive iterations, we adjust the base kernel such that after n iterations,

$$K_n\left(x, x'\right) = K\left(F_n(x), F_n\left(x'\right)\right), \tag{2.42}$$

with $F_1(x) = x$ and $F_{n+1}(x) = F_n(x) + \varepsilon_{n+1} G_{n+1}\left(F_n(x)\right)$. This is equivalent to integrating a stochastic data-driven dynamical system that allows for the training of very deep networks [26]. G_n is the perturbation added to the input data points and this determines the layers of a network. We once again use the batch and sample subsets $\mathcal{D}_f$ and $\mathcal{D}_c$ respectively obtained by uniform sampling without replacement along with the sub-sampling matrix π. This time we take the Fréchet derivative with respect to the input points, which can be represented as equation (2.43) as demonstrated in [26].

$$\hat{g}_{f,i}^{(n)} := 2\frac{(1-\rho(n))\hat{y}_{f,i}^{(n)T}\left(\nabla_x K_1\right)\left(x_{f,i}^{(n)}, x_{f,\cdot}^{(n)}\right)\hat{y}_f^{(n)} - \hat{z}_{f,i}^{(n)T}\left(\nabla_x K_1\right)\left(x_{f,i}^{(n)}, x_{f,\cdot}^{(n)}\right)\hat{z}_f^{(n)}}{y_f^T\left(\Theta^{(n)}\right)^{-1} y_f}, \tag{2.43}$$

where $\hat{y}_f^{(n)} := \left(\Theta^{(n)}\right)^{-1} y_f^{(n)}, \hat{z}_f^{(n)} := \left(\pi^{(n)}\right)^T \left(\pi^{(n)}\Theta^{(n)}\left(\pi^{(n)}\right)^T\right)^{-1}\pi^{(n)} y_f^{(n)}$. Note once again that this is valid for scalar targets only. For problems such as classification, we require the trace of both the numerator and denominator as explained in [26, Remark 6.1].

We still require the perturbations for the data points that were not in the batch $\mathcal{D}_f$ for which we interpolate

$$G_{n+1}(x) = \left(\hat{g}_{f,\cdot}^{(n)}\right)^T \left(K_1\left(x_{f,\cdot}^{(n)}, x_{f,\cdot}^{(n)}\right)\right)^{-1} K_1\left(x_{f,\cdot}^{(n)}, x\right). \tag{2.44}$$

Note that as per [26], $G^{(n+1)}\left(x_{f,i}^{(n)}\right) = \hat{g}_{f,i}^{(n)}$. For $i \in \{1, \ldots, N\}$ i.e. the points in the batch are not interpolated but added to the vector of perturbations, and only points that are not part of the batch are interpolated. This will prove to be consequential later.

Algorithm 2 Kernel Flows with a Non-Parametric Family of Kernels

```
Select base Kernel $K_1$
for $i = 1 \to Iter$ do
    Create batch $\mathcal{D}_{N_f}$ from the total dataset uniformly sampled without replacement.
    Create sample $\mathcal{D}_{N_c}$ from batch uniformly sampled without replacement.
    Calculate $\rho$ using equation (2.39)
    Calculate $\hat{g}_{f,i}^{(n)}$ using equation (2.43)
    Calculate $G_{n+1}(x)$ using equation (2.44)
    Update datapoints $x_i^{(n+1)} = x_i^{(n)} + \epsilon G_{n+1}\left(x_i^{(n)}\right)$
end for
```

The algorithm can then be presented as shown in algorithm 2.

The value of ϵ can be either fixed for the entire run or changed dynamically based on the number of iterations or the norm of the perturbations. We will, however, later see that this manner of updating the data points is not too effective.

3. Parametric Kernel Flows for Neural Network Induced Gaussian Processes

In this chapter, we will utilize the concepts discussed to assess the parametric version of the Kernel Flows algorithm. We will apply the parametric version of the Kernel Flows algorithm to Neural Network induced Gaussian Process kernels. We will look at the computational aspects of the algorithm, first theoretically in section 3.1. In section 3.3 we will have a look at the use of GPUs and the caveats attached to their use in numerical calculations involving large matrices in comparison to evaluations on the CPU. In subsequent sections, we will use the implementation of the Kernel Flows algorithm on kernels obtained from various NNs and compare them to randomly initialized NNGPs / CNNGPs and those mentioned in the literature. We will also look at the computational cost involved with the algorithm and how we worked to reduce the computational expense both in terms of memory and time to a minimum such that the algorithm can be applied to expensive NNGPs.

3.1. Theoretical Computational Cost of Kernel Flows

We will discuss the theoretical computational cost involved in performing the two variants of the Kernel Flows algorithm in this section. For this, we will use the insights presented by Darcy [5, Section 2.1.3]. We start by analyzing the parametric version of the algorithm. We assume that matrix multiplication of two matrices of size $n \times m$ and $m \times p$ is $\mathcal{O}(nmp)$ and that matrix inversion of an $n \times n$ matrix is $\mathcal{O}(n^3)$. We treat the cost of computing the kernel separately, which results from the NNGP / CNNGP kernel, or any other kernel used. There are various ways to improve the computational cost of evaluating the kernel. In particular, the discussion by Martin Meinel addresses, implements, and compares a variety of techniques to reduce the cost of kernel evaluation for CNNGP kernels. They conclude that by evaluating only a small part of the kernel matrix, the missing elements can be approximated sufficiently, which leads to reduced evaluation cost without deteriorating results too much [23]. In this thesis, we do not focus on improving the efficiency of the kernel evaluation. Rather our energies will be spent on making the Kernel Flows algorithm more efficient while using the kernel evaluation algorithm we have been provided with. For future work, it would be interesting to couple our implementation of the Kernel Flows algorithm with the efficient CNNGP kernel evaluations by Meinel.

Let us first look at the parametric version of the Kernel Flows algorithm. Keeping in mind Algorithm 1, we observe the key computational complexity comes from evaluating

Table 3.1.: Computational costs of evaluating the terms that constitute the evaluation of ρ in equation (2.39).

Evaluation	Computational Cost Breakup	Overall Computational Cost
$y_c^T \left(\pi\Theta\pi^T\right)^{-1} y_c$	$\mathcal{O}(N_c^3) + \mathcal{O}(N_c N_f^2) + \mathcal{O}(N_c^2 N_f) + \mathcal{O}(N_c^2) + \mathcal{O}(N_c)$	$\mathcal{O}(N_c^3)$
$y_f^T \Theta^{-1} y_f$	$\mathcal{O}(N_f^3) + \mathcal{O}(N_f^2) + \mathcal{O}(N_f)$	$\mathcal{O}(N_f^3)$

Table 3.2.: Computational costs of evaluating the derivative of ρ with respect to the kernel hyper-parameters as specified in equation (2.41).

Evaluation	Computational Cost Breakup	Overall Computational Cost
$\hat{y}$	$\mathcal{O}(N_f^3) + \mathcal{O}(N_f^2)$	$\mathcal{O}(N_f^3)$
$\hat{z}$	$\mathcal{O}(N_c^3) + \mathcal{O}(N_f^2) + 2\mathcal{O}(N_c N_f^2) + 2\mathcal{O}(N_c^2 N_f)$	$\mathcal{O}(N_c^3)$
$\hat{z}^T \left(\partial_{W_i}\Theta(W)\right)\hat{z}$ and $\hat{y}^T \left(\partial_{W_i}\Theta(W)\right)\hat{y}$	$\mathcal{O}(N_f^2)$	$\mathcal{O}(N_f^2)$
$y_f^T \Theta^{-1} y_f$	$\mathcal{O}(N_f^3) + \mathcal{O}(N_f^2) + \mathcal{O}(N_f)$	$\mathcal{O}(N_f^3)$

ρ and its Fréchet derivative with respect to each kernel parameter. Table 3.1 shows the costs of evaluation of each term in ρ based on equation (2.39) with slight notational abuse.

This gives the overall cost as $\mathcal{O}(N_f^3)$. Note that the details are slightly different from [5] since here we assume that the kernel is only evaluated once for the batch and the sample kernel is evaluated with the help of the π matrix. This prevents us from evaluating the kernel twice, which, as we will see in our experiments, is significantly more expensive than evaluating ρ. Nevertheless, the final computational cost of ρ is the same. Now we look at the cost of evaluating the gradient of $\partial_{W_i}\rho(W)$ based on equation (2.41) for a single kernel parameter. This is presented in table 3.2. Note, once again, we do not include the cost of differentiating the kernel with respect to its parameters and assume this matrix is given.

So the cost of computing the gradients is also $\mathcal{O}(N_f^3)$ for a single parameter. Although we do not discuss the cost of computing the kernel and its derivative, it is important to note that the analytical expression for evaluating $\partial_{W_i}\rho(W)$ is not used when using CNNGP or NNGP kernels. This is because a closed-form equation of $\partial_{W_i}\Theta(W)$ is not available and libraries and packages implementing these rely on automatic differentiation. Closed-form solutions do exist for simple kernels such as the RBF kernel, which can be evaluated with roughly the same computational complexity as the kernel evaluation. However, when using automatic differentiation, the results are much slower [5]. Our initial implementation involves using an automatic differentiation engine to get the derivative of ρ with respect to the kernel parameters, which has a separate cost compared to the analytical expression. This differentiation process will prove to be the most expensive part of the algorithm, as will be shown in section 3.6.3.

The case for the non-parametric Kernel Flows algorithm is similar except that it involves the derivative of the kernel matrix with respect to the inputs. This means the kernel derivative with respect to the input is of dimension $d \times N_b \times N_b$ where d is the dimensionality

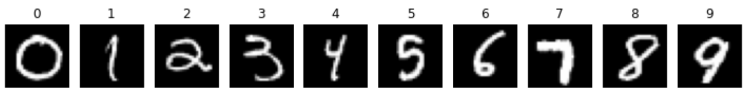


Figure 3.1.: Sample images from the MNIST Dataset [17]

Figure 3.2.: Sample images from the CIFAR-10 Dataset [15]

of a single input datapoint leading to a final result of $\mathcal{O}(dN_b^3)$ for $\hat{g}_{f,i}^{(n)}$ as given in equation (2.43) while the evaluation of ρ remains the same.

3.2. Datasets

3.2.1. MNIST Dataset

For the evaluation of the Kernel Flows method using various optimization techniques, one of the datasets we use is the MNIST dataset [17]. The dataset contains handwritten digits from 0 to 9 along with their labels, where each image is one digit. A sample of each digit can be seen in figure 3.1. Each image is a matrix of 28×28 and hence is in grayscale. We divide the dataset into training and testing subsets. The size of the test set is always restricted to 1000 in every experiment. The same dataset was used by [9] which allows us to directly compare our results with their results. We will not apply any transformations to the input dataset for MNIST since we wish to compare our results with previous literature and remain consistent.

3.2.2. CIFAR-10 Dataset

The second dataset that we utilize is the CIFAR-10 dataset [15]. This is a much more complicated dataset compared to the MNIST dataset. The CIFAR-10 dataset consists of 60,000 images each of size 32×32 pixels with three channels for red, blue, and green, making each input image a tensor of size $3 \times 32 \times 32$. Each image in the dataset corresponds to one of the ten classes shown in figure 3.2. Note that here we normalize the dataset with a mean and standard deviation of 0.5 across all channels to have centered data. Again, the dataset was divided into a training dataset and a testing dataset. The size of the test set was once again set to 1000 in every experiment.

3.3. Numerical Differences Between CPU and GPU Runs

Before diving into the implementation details of the Kernel Flows algorithm, we will digress slightly to understand the numerical differences that arise when performing numerical experiments on the CPU versus those on the GPU. It was observed during our experiments that the results extracted from a GPU were significantly different compared to those extracted from the CPU. We will explore these findings in this section.

We first present the results that motivated us to explore the behavior of GPU-based calculations compared to CPU-based calculations. We perform Kernel Ridge Regression for various numbers of input data points N_I using a CNNGP kernel as implemented by [9] [1] in `Pytorch` [28]. We essentially need to perform the computation of equation (3.1) to evaluate the predictions $\hat{\mathbf{Y}}_*$ for some test points $\mathbf{X}_*$ given training points $\mathbf{X}$ of sample size N_I using the MNIST dataset, i.e.,

$$\hat{\mathbf{Y}}_* = K(\mathbf{X}_*, \mathbf{X})(K(\mathbf{X}, \mathbf{X})^{-1} + \lambda \mathbf{I_N})\mathbf{Y}. \tag{3.1}$$

For this, we evaluated $K(\mathbf{X}, \mathbf{X})$ and $K(\mathbf{X}_*, \mathbf{X})$. The kernel evaluation was done on the GPU. In particular, we utilized an NVIDIA RTX-3080 GPU. The calculation of $K(\mathbf{X}, \mathbf{X})^{-1}\mathbf{Y}$ was done using least squares with a cut-off ratio of 10^{-8} on the CPU using 64-bit precision instead of using a linear solver since the kernel matrix is difficult to invert and can lead to numerical errors. It can be shown that the eigenvalues of the matrix decrease very fast, making it very likely to be singular [23, Section 3.3.6]. Finally, the multiplication of $K(\mathbf{X}_*, \mathbf{X})$ with $(K(\mathbf{X}, \mathbf{X})^{-1}\mathbf{Y})$ was done with four different settings mentioned in list 1.

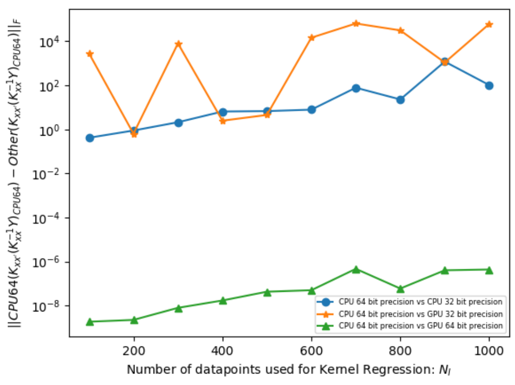
1. on a CPU with 32-bit precision,
2. on a CPU with 64-bit precision,
3. on a GPU with 32-bit precision,
4. on a GPU with 64-bit precision.

We use accuracy as a measure to test the performance of the method on the test dataset using,

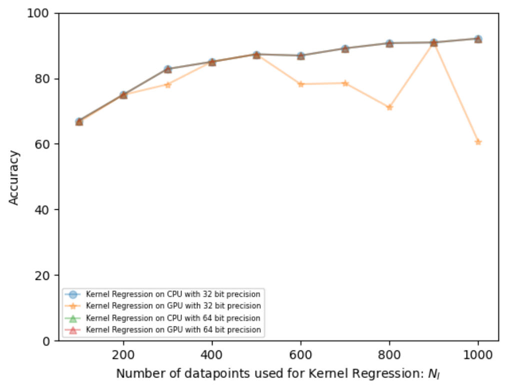
$$\text{Accuracy} = \frac{\text{Correct Classifications in DataSet}}{\text{Dataset Size}} \times 100. \tag{3.2}$$

We plot the accuracy of prediction against the number of training points used in the four different settings. The result is shown in figure 3.3b. We see that the results from the CPU with both 32 and 64-bit precision, as well as those from the GPU with 64-bit precision, are almost identical. However, the results from the GPU with 32-bit precision are very different. We repeat the experiment several times on the same GPU and each time

[1]Code from [9] available on the GitHub repository `https://github.com/cambridge-mlg/cnn-gp`



(a) Frobenius Norm of Difference between results of Matrix Multiplication evaluated on a CPU and on a GPU for various GPU bit precisions mentioned in list 1.



(b) Accuracy on the test set after Kernel Right Regression with the same Kernel by performing evaluations on the CPU vs on the GPU under different precision settings mentioned in list 1.

Figure 3.3.: Numerical Differences between evaluations on CPU vs GPU with various bit precisions mentioned in list 1.

the results from the GPU with 32-bit precision appeared to be the worst out of the four settings. Thus, we observe that the problem appears in the matrix multiplication step. We also plot the norm of the difference between the result of $K(\mathbf{X}, \mathbf{X}_*)K(\mathbf{X}, \mathbf{X})^{-1}\mathbf{Y}$ under the different settings compared to the evaluation on the CPU with 64-bit precision which we take to be the most accurate. This is shown in figure 3.3a. We can observe that the norm of the differences is the largest for the GPU with 32-bit precision, followed by the calculation from the CPU with 32-bit precision. The calculation from the GPU with 64-bit precision is almost identical to the evaluation on the CPU with 64-bit precision. We conclude that the evaluation of matrix multiplication on the GPU is of limited accuracy.

To confirm that this behavior is indeed due to just matrix multiplication and not an artifact of the kernel evaluation, we explore the problem in a more general setting. For our evaluation we instantiate two random matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{N_i \times N_i}$ where $N_i = 100i$ for $i \in [1 \ldots 10]$ and perform the matrix multiplication $\mathbf{AB}$ on all four precision settings discussed previously. We then evaluate the Frobenius norm of the differences of the evaluations compared to the evaluation on the CPU with 64-bit precision which we again take to be the most accurate evaluation, effectively calculating $||(\mathbf{AB})_{\text{cpu 64 bit}} - (\mathbf{AB})_{\text{other settings}}||_F$. The results are shown in figure 3.4. We see that even under this general setting, the results from the GPU on 32-bit precision are much worse compared to results from the CPU with 32-bit precision. The norm of the differences between the results from the GPU with 64-bit precision and the CPU with 64-bit precision is extremely small such that it is lower than the precision shown on the y-axis of the graph, which is why this line is not visible in the

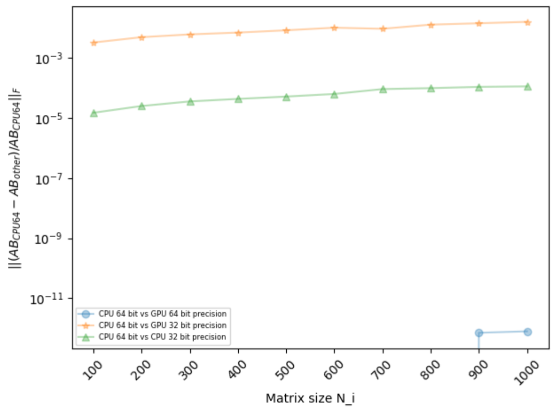


Figure 3.4.: Frobenius Norm of Difference between results of Matrix Multiplication of random matrices under various settings mentioned in list 1. Note that the norm of differences between the results from GPU with 64-bit precision and CPU with 64-bit precision is extremely small such that it is lower than the precision shown on the y-axis of the graph which is why this line is not visible in the graph.

graph. We hypothesize that the reason for the discrepancy in the evaluation of the GPU with 32-bit precision is that in order to exploit the parallel behavior of the GPU, the order of arithmetic operations is changed. However, floating point arithmetic is not associative and the change in order leads to numerical differences. Moreover, it is possible that the CPU is computing the results internally at a higher precision than the one we set using the `Pytorch` API. A detailed explanation of this can be found in a white paper published by Whitehead et al [34].

The sub-optimal accuracy of the GPU is clear and consistent with our experiments, but to check reproducibility on different hardware, we run the general experiment of the multiplication of two matrices and evaluate the Frobenius norm of the difference between the evaluation on the CPU with 64-bit precision and other settings on different hardware. For this, we utilize a GPU available via Google Colaboratory and run the experiments on the CPU, a 64-bit x86 architecture, and GPU, a Tesla T4, provided there [2]. The result of the experiment on the GPU available from Google Colaboratory is shown in figure 3.5. The results are surprising as we note that the poor accuracy resulting from the GPU with 32-bit precision seems to have disappeared and that this setting is performing roughly identically to the evaluations on the CPU with 32-bit precision. It is likely that the resulting numerical differences, while possibly coming from the reordering of floating-point operations, only appear in some hardware and not all due to either faulty or outdated driver software.

Keeping these findings in mind, we opted for a different GPU, an RTX 3070 with up-to-date driver software for our evaluations. We ensure that this GPU does not result in the inaccuracies seen on the first machine by rerunning the above-mentioned experiments

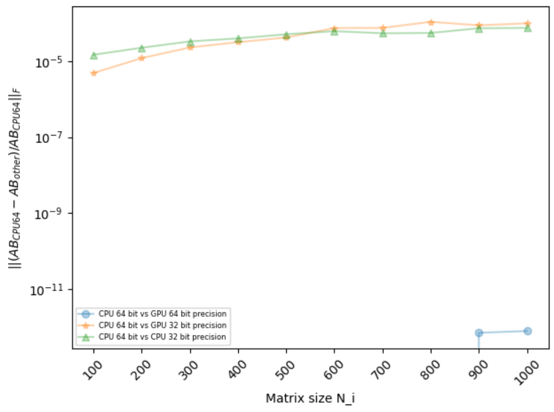


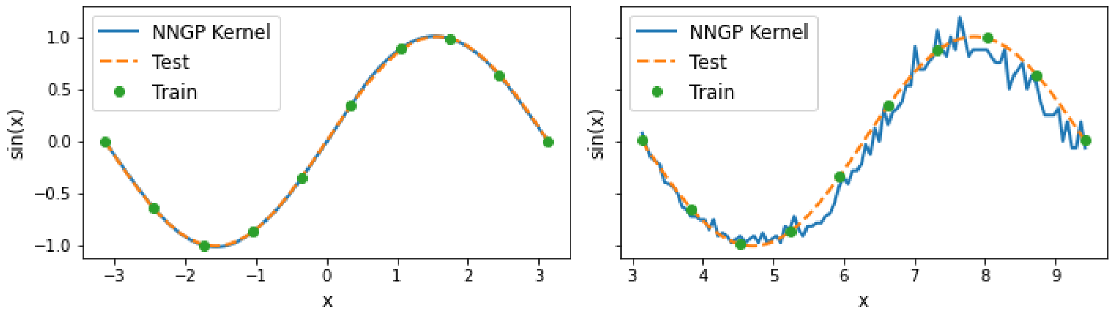
Figure 3.5.: Frobenius Norm of Difference between results of Matrix Multiplication of random matrices under various settings mentioned in list 1 on GPU available from Google Colaboratory [2].

on it and verifying that the accuracy of the GPU is the same as that of the CPU. Thus, we conclude that the problem indeed was hardware dependent and suggest that before carrying out numerical experiments on the GPU, some numerical tests should be carried out to confirm the absence of erroneous behavior like the one noted here.

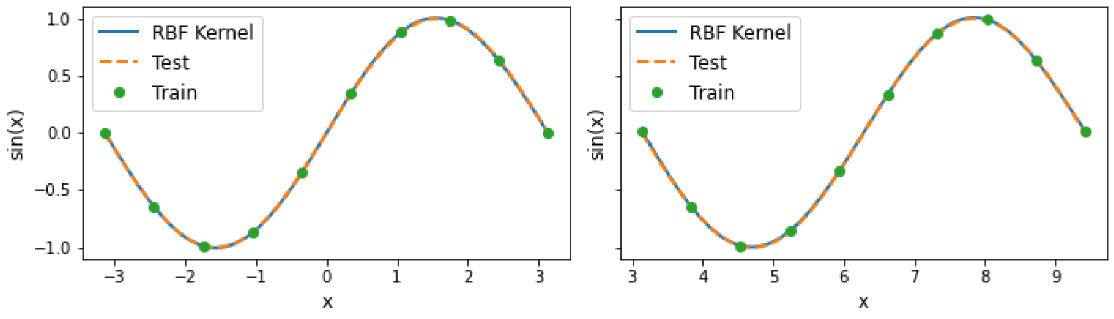
3.4. Extracting Kernels from Neural Network Induced Gaussian Process

In this section, we will go into the details of how we applied the parametric version of the Kernel Flows algorithm on kernels extracted from Neural Networks to improve the kernel quality. In this case, the weight and bias variances σ_w^2 and σ_b^2 form the trainable parameters of the kernel. A single training step of the Kernel Flows algorithm then boils down to evaluating the kernel on a randomly selected batch from the entire dataset without replacement, calculating ρ and performing the backward pass on ρ to update the values of σ_w^2 and σ_b^2 until the stopping condition is reached. We will soon see that performing this backward pass in the true sense of Algorithm 1 comes with restrictions due to which alternates will be presented.

It is important to note that before using dense NNGP (and CNNGP kernels), the training data points should be centered around zero. This is because equation (2.20) starts off by multiplying the training points with each other, and if they are not centered, then the kernel results are also off-center, leading to poor evaluations during the regression step. This is shown in figure 3.6. Figure 3.6a shows how, with uncentered data, the predictions using an NNGP kernel are very poor. However, this is not something the RBF kernel suffers from, as shown in figure 3.6b.



(a) Performance of NNGP Kernel on centered vs uncentered data.



(b) Performance of RBF Kernel on centered vs uncentered data.

Figure 3.6.: Difference in performance on centered vs uncentered data for an NNGP kernel and an RBF kernel.

3.4.1. Kernel Flows on Dense Neural Network induced Gaussian Process

To obtain Dense Neural Network induced Gaussian Process kernels, we used the Neural Tangents library developed by Google [25]. This Python API allows users to build neural networks by specifying the individual layers that constitute a network, both of finite and infinite widths. The infinite width networks correspond to Gaussian processes, and in this case, their kernel function is also obtained. The API to generate an NNGP kernel from a given architecture is straightforward and similar to many common libraries that allow users to create neural networks with ease. An example of making a very simple dense Neural Network along with its corresponding Gaussian process kernel is shown in the code snippit 3.1.

```
init_fn, apply_fn, kernel_fn = stax.serial(
    stax.Dense(32),
    stax.Relu(),
    stax.Dense(1))
```

Source Code 3.1.: Example construction of a Dense NNGP kernel using the Neural Tangents library.

Here `init_fn` is used to initialize a network, `apply_fn` represents the finite width neural network, and `kernel_fn` represents the NNGP kernel. This kernel function is our target kernel, which will be utilized in the parametric Kernel Flows. Note that the Neural Tangents library also allows the construction of CNNGP kernels, but we do not rely on this library for CNNGP kernels. This is because the API does not allow the user to modify the variance of the weights and biases with the help of its automatic differentiation engine. Due to this, the Kernel Flows algorithm in its original form cannot be applied to kernels obtained from the Neural Tangents library. Nevertheless, as we will see in section 3.5, we can use some alternate techniques to optimize the said kernels without requiring automatic differentiation. In all future experiments, we will stick to ReLU as our activation function since this activation function has an analytical form as shown in equation (2.29).

3.4.2. Kernel Flows on Convolutional Neural Network induced Gaussian Process

In order to extract Convolutional Neural Network Gaussian Process kernels, we use the implementation from Garriga-Alonso et al with some changes, which will be discussed shortly [9]. This implementation of CNNGP kernels is built on top of `Pytorch` and the user can define a CNNGP kernel in the same way as a standard neural network would be defined [28]. The structure of the neural network should be such that the final layer is equivalent to a dense layer after the final convolution is performed. Subsequently, the extracted kernel is simply a callable function that accepts two datasets. This callable function can be used as an argument in our Kernel Flows algorithm and optimized accordingly. Again, we will stick to ReLU as our activation function as we have its closed-form implementation available.

In order to utilize the CNNGP implementation from Garriga-Alonso et al alongside the Kernel Flows algorithm, we had to introduce a few changes. The original implementation did not develop the CNNGP kernel as a trainable module. We changed this such that the weight and bias variances become trainable parameters of the CNNGP kernel, and their gradients can be tracked during the calculation of ρ using the automatic differentiation engine of `Pytorch`. Secondly, the authors allowed each convolution layer to have a different variance for weight and bias. However, this would incur excessive memory costs due to which we altered the kernel to have a universal weight (σ_w^2) and bias (σ_b^2) variance for each convolution layer. This changed the API slightly as now the weights and bias variances had to be declared in the sequential block and not for each convolution layer. Nevertheless, the general structure is still similar to the pattern defined by `Pytorch`. A simple example for a CNN to model the MNIST dataset can be defined by code snippet 3.2.

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

Source Code 3.2.: Example construction of a Convolutional NNGP kernel using the code from [9].

Notice how the above architecture forms a dense output layer for the MNIST dataset, which is important as noted earlier. The third change we implemented was for numerical stability. The authors only implemented the ReLU activation, which involves trigonometric functions which are not differentiable everywhere. As the authors did not require any backpropagation in their implementation they used equation (2.29) with slight changes where the evaluation of $cos(\theta)$ was clamped to $[-1, 1]$. In order to allow for backpropagation without having undefined gradients, we clipped the result to $[-1 + 10^{-6}, 1 - 10^{-6}]$ similar to [33].

With this, we have a differentiable ρ function. Note that our implementation of parametric Kernel Flows is in fact agnostic to the kernel type as long as it is a `torch.nn.Module` which is callable and has its hyper-parameters as trainable `Pytorch` variables. Hence, it can be applied to all sorts of kernels.

3.5. Variations of Parametric Kernel Flows on NNGP Kernels

In this thesis we implemented and experimented with 3 variations of the original parametric Kernel Flows algorithm 1 as presented by [26] and implemented by [5]. In this section, we will discuss the concepts that tweak the algorithm for specific benefits compared to the original implementation. We remind ourselves that the original implementation worked using a gradient-based optimization step and had a closed form representing both the parameter ρ which we wished to optimize and its gradient with respect to the kernel parameters. The closed-form expression of the gradient is not useful when it is not possible or difficult to differentiate the kernel explicitly with respect to its parameters. This is the case with NNGP kernels. In such instances, we use automatic differentiation, a technique ubiquitous in packages that provide neural network frameworks such as `Pytorch` [1]. This will be the first variation we explore. This approach was already employed by Darcy, where the Autograd library was used for concrete implementation [21]. We will highlight the problems that arise with automatic differentiation and then shift our focus to gradient approximation methods, particularly finite differences. We will also have a look at gradient-free optimization with Bayesian Optimization as the final variation of the Kernel Flows algorithm.

3.5.1. Implementation Details

We will briefly discuss the high-level implementation details of the Kernel Flows algorithm [2]. We used Python as the programming language for our implementation. Darcy implemented the parametric Kernel Flows algorithm using the closed form Fréchet derivative given in equation (2.41) and also using the automatic differentiation engine Autograd [21]. However, as we do not know the closed-form derivative of an NNGP kernel of arbitrary architecture, we could not use the closed-form implementation. As discussed in section 3.4, we utilize the NNGP implementations from various libraries which rely on their own automatic differentiation engines namely JAX and Pytorch. Hence, an overarching implementation was not possible and we divided our implementation into one that works with Dense NNGP kernels implemented using the Neural Tangents and JAX library [3, 25] and one that works with CNNGP kernels using `Pytorch` and the implementation from [9]. Darcy's implementation is useful when the kernel function relies only on `numpy` and the use of a GPU is not necessary [5]. We also utilize the `matplotlib` library for graphing purposes. With regards to the hardware used, all our experiments were performed on an NVIDIA RTX 3070 GPU, which had a memory of 8GB whereas the CPU computations were performed on an AMD Ryzen 7. Darcy also implemented adaptive sampling during batch and sample dataset formation steps of Algorithm 2, but we stick to fixed sizes for both of these datasets for all our experiments.

[2]The code is available on the GitHub repository `https://github.com/waleedbinkhalid74/cnn-gp/tree/develop`

3.5.2. Gradient Descent with Automatic Differentiation

We will first look at the most intuitive way of implementing the parametric Kernel Flows algorithm on NNGP kernels. This is done by using an automatic differentiation engine. As mentioned in section 3.4.1, our Dense NNGP kernels utilize the JAX library, which does allow us to modify the standard deviation of the weights and biases and use them as trainable parameters. Hence, this version of the Kernel Flows algorithm is only applied to the Convolutional NNGP kernels where we made modifications to the implementation by [9] which were discussed in section 3.4.2. The central idea, nevertheless, remains the same. The variances, σ_w^2, and σ_b^2 are initialized and neural network architecture is defined to construct a CNNGP kernel function. We then select the hyperparameters of the Kernel Flows algorithm, including the regularization coefficient, learning rate, gradient descent method, batch size N_f, and sample proportion. Subsequently, a forward pass is conducted with the chosen batch and sample from the entire dataset to calculate ρ. This is where the automatic differentiation engine comes in handy. Since it is very difficult to have a closed-form expression of the derivative of the CNNGP kernel with respect to the weight and bias variances, the automatic differentiation engine maintains a computational graph of all the calculations involved in evaluating ρ. Once the forward pass is finished, we can perform the backward pass, which uses this computational graph to evaluate the gradient of ρ with respect to all the parameters of the kernel. After this, only the gradient update step needs to be performed, which in our case is the stochastic gradient descent update [30]. The algorithm, however, suffers from the problem of instability given a very poor initial kernel since this can result in undefined values. To remedy this, we add a re-initialization step where the values of the variances are reset if the randomly selected starting kernel produces undefined values of ρ.

This routine is similar to training a neural network. This way of performing parametric Kernel Flows worked well for [5] where simple kernels like the radial basis functions shown in equation (2.9) were utilized, which are cheap to evaluate. However, we soon run into memory problems when using CNNGP kernels. Since the CNNGP kernels (and also the Dense NNGP kernels) involve a large number of computations, we observed that running the algorithm for large CNNs or with large batch sizes results in the GPU running out of memory. We ran some memory experiments where we evaluated the GPU memory consumed in evaluating the CNNGP kernel for different batch sizes N_f while the automatic differentiation engine maintained its computational graph for the learnable parameters. For our demonstrations, we use the MNIST dataset mentioned previously with a test CNNGP architecture with two convolution layers and one ReLU layer in between, shown in figure 3.7. The results of this memory experiment can be seen in figure 3.8 in the orange plot. We observe that the memory usage growth is superlinear and that with just 450 image samples, the GPU memory usage was reaching 8 GBs. This makes sense since, in the case of a CNNGP kernel, the initial tensor that is formed before the data is fed to any of the convolution layers is a result of equation (2.26) of size $(N_f \times N_f, 1, 28, 28)$ for the MNIST dataset. We can see that this quickly blows up for large values of N_f. While having

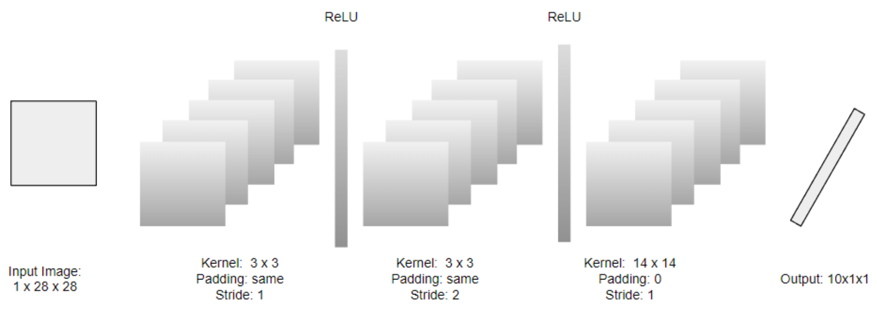


Figure 3.7.: Three-Layered CNN architecture used to obtain equivalent Gaussian Process Kernel.

large tensors in and of itself is not necessarily a problem, we must note that in our modified CNNGP kernel evaluation, the weight and bias variances, σ_w^2 and σ_b^2 are learnable parameters. This means that there is a computational graph attached to the elements of the tensors which will be utilized by the automatic differentiation engine when performing the backward pass. This extra memory attached to the elements of this large tensor is what is responsible for the exploding memory.

We look at the computational graph resulting from the evaluation of the CNNGP kernel and subsequent calculation of ρ for the CNNGP architecture shown in figure 3.7. This graph can be seen in the appendix in figure A.1. We observe that this is excessively large as it is composed of many computations, and we know that each node maintains a cache of the calculated tensors required to calculate the gradients. This is a rather naive way of using the automatic differentiation engine because only individual operations are presented for which a derivative calculating function exists by default in `Pytorch`. A lot of these steps could be avoided if we group a set of calculations and provide the engine with its equivalent differentiation in closed form. This is something that is already done in commercial automatic differentiation packages where common but larger operations have predefined functions that calculate the gradient instead of relying on basic arithmetic operations. In particular, in the case of the ReLU covariance evaluation, we see that it consists of many operations that can be grouped together. Thus, instead of forcing the automatic differentiation engine to differentiate each expression separately, we provide a custom function that implements the ReLU covariance function the same way as is done by [9] along with the differentiation of this function with respect to all its inputs, namely $K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})$, $K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')$, and $K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')$. From equation (2.29) we know the form of the covariance of the ReLU activation, which can equivalently be written as

$$V_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}'\right) = \frac{\sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}') - K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')^2} + (\pi-\theta)K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')}{2\pi}. \tag{3.3}$$

Note the division by 2 is done to avoid multiplying the ReLU by $\sqrt{2}$ as suggested by [9]. Now differentiating with respect to all inputs.

$$\frac{\partial V_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}'\right)}{\partial K_\nu^{(\ell)}\left(\mathbf{X}',\mathbf{X}'\right)} = \frac{-\cos^{-1}\left(\frac{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')}{\sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')}}\right)+\pi}{2\pi}. \tag{3.4}$$

Then

$$\frac{\partial V_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}'\right)}{\partial K_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}\right)} = K_\nu^{(\ell)}\left(\mathbf{X}',\mathbf{X}'\right)P, \tag{3.5}$$

and similarly

$$\frac{\partial V_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}'\right)}{\partial K_\nu^{(\ell)}\left(\mathbf{X}',\mathbf{X}'\right)} = K_\nu^{(\ell)}\left(\mathbf{X},\mathbf{X}\right)P, \tag{3.6}$$

where

$$P = \frac{\frac{1}{2\sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')-K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')^2}} - \frac{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')^2}{2(K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}'))\sqrt{K_\nu^{(\ell)}(\mathbf{X},\mathbf{X})K_\nu^{(\ell)}(\mathbf{X}',\mathbf{X}')-K_\nu^{(\ell)}(\mathbf{X},\mathbf{X}')^2}}}{2\pi}.$$

With the help of this, we can drastically reduce memory consumption since the automatic differentiation engine now only needs to maintain the inputs to the ReLU covariance function in its cache instead of having to maintain a node for each operation in the evaluation of the ReLU covariance. The computational graph was also significantly reduced, which can be seen in the appendix in figure A.2. The memory consumption results can be seen with the blue plot in figure 3.8. This shows that with our custom implementation of the ReLU activation kernel backward pass, we were able to reduce memory consumption substantially. This subsequently allowed us to perform Kernel Flows on this simple CNNGP network with reasonable batch sizes.

3.5.3. Gradient Descent with Finite Difference Approximation

We will now look at a gradient approximation method instead of trying to compute the exact gradient of ρ with respect to the kernel parameters. This is motivated by the fact

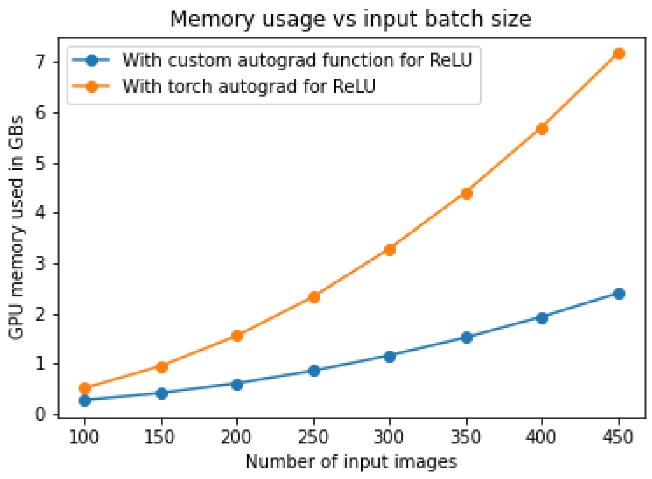


Figure 3.8.: Comparison of the memory usage when evaluating ρ using the three-layered CNNGP kernel from figure 3.7 with both the default implementation of the ReLU covariance and with a custom implementation of the ReLU covariance alongside its derivative with respect to the inputs.

that even with our memory-efficient implementation in section 3.5.2, automatic differentiation is still restrictive if the depth of the neural network that induces the GP kernel grows. Since the bottleneck is the excessive memory consumed by the computational graphs from the automatic differentiation engine, we took the approach of avoiding the storage of gradients altogether. For this purpose, we choose to approximate the gradients using finite differences. We use the forward difference as our gradient approximation method. As our CNNGP kernels, irrespective of their depths, have only two parameters, the forward difference evaluation would require three kernel evaluations for each iteration. Specifically, one without any perturbation, one with perturbation to σ_w^2, and one with perturbation to σ_b^2. The resulting optimization method step in Algorithm 1 is as shown as

$$\begin{aligned} \sigma_w^2 &= \sigma_w^2 - \eta \frac{\rho_{\sigma_w^2+h,\sigma_b^2} - \rho_{\sigma_w^2,\sigma_b^2}}{h}, \\ \sigma_b^2 &= \sigma_b^2 - \eta \frac{\rho_{\sigma_w^2,\sigma_b^2+h} - \rho_{\sigma_w^2,\sigma_b^2}}{h}. \end{aligned} \tag{3.7}$$

We set h as 10^{-4} for all experiments when using finite differences to approximate gradients. For small kernels, this is efficient, but as the kernel depth grows, performing more kernel evaluations for each iteration of the Kernel Flows algorithm becomes computationally expensive. Nevertheless, the good news is that no memory needs to be reserved, meaning that, given enough time, one could perform the parametric Kernel Flows algorithm with an arbitrary batch size and with an NNGP kernel stemming from an NN of arbitrary depth. This is because the kernel evaluation, when done without tracking gradients, can be performed using a block-wise approach, which is also used by [9, 23, 33]. This block-wise approach was not applicable to the automatic differentiation variation of the Kernel Flows algorithm since, despite having small blocks, as the larger tensor was formed, the computational graph had to be maintained, resulting in the same memory consumption. However, the gradient approximation variant of the algorithm also suffers from the problem of instability given a very poor initial kernel, and similar to the automatic differentiation version of the algorithm, we add a re-initialization step if the randomly selected starting kernel produces undefined values of ρ.

3.5.4. Bayesian Optimization

Up till now, we have only seen gradient-based, local optimization methods and observed how they had their limitations with respect to computational cost, memory expense, and overall achievable improvement. We wish to avoid calculating the gradients of ρ with respect to the parameters of the kernel altogether, as we saw that this can be unstable, memory-intensive, and computationally demanding. Hence, we shift to Bayesian optimization, which is a global optimization strategy that does not require the gradient of the objective function at all. We will not discuss the theory behind Bayesian optimization in too much detail, but rather provide a brief overview. For a more detailed explanation, we refer the reader to Peter I. Frazier's tutorial on Bayesian Optimization [8].

Bayesian optimization is particularly useful when we wish to optimize the objective function and the following conditions are met.

$$\min_{x \in A} f(x),$$

- f is expensive to evaluate.
- f lacks known structure, meaning techniques that exploit these structures cannot be used.
- f is continuous.
- Input argument $x \in \mathbb{R}^d$ with $d \leq 20$.

This works using two key ingredients. First, a method for statistical inference, in our case a Gaussian process, which was discussed in section 2.2.1, and secondly, an acquisition function to decide where to sample the next evaluation point from. Initially, a Gaussian prior is fitted over the target function, which is evaluated at some randomly chosen initial points. Thereafter, the posterior probability is updated based on the results. Subsequently, an acquisition function is used to determine the next evaluation point, and this new evaluation further updates the posterior. The process is repeated until the termination condition is reached. We summarize the routine in Algorithm 3. Note that this algorithm tries to maximize the target function as we wish to be consistent with the reference used, but it can easily be turned into a minimization problem by multiplication with -1 [8].

Algorithm 3 Bayesian Optimization Pseudo-Code [8]

```
Place Gaussian Prior on $f(x)$.
Evaluate $f(x)$ at randomly selected $n_0$ points.
for $i = n_0 \to MaxIter$ do
    Update the posterior probability distribution on $f(x)$ using available observations.
    Evaluate the maximum of the acquisition function over $x$ based on the current posterior distribution.
    Evaluate $f$ at the point selected by maximizing the posterior distribution.
end for
Return the point where $f(x)$ was found to be the largest.
```

For our choice of acquisition function, we stick to the Expected Improvement (EI). Other activation functions can also be utilized based on the applicability of these acquisition functions to the specific problem. Let us briefly discuss the EI acquisition function. Suppose, based on the previous evaluations, we have a candidate maximum $f_n^* = \max_{m \leq n} f(x_m)$ where n is the number of points we have evaluated our objective function on up till now. For our next evaluation, the value of the optimal point can either remain the same, if $f(x) \leq f_n^*$, or it can become the newly evaluated point if $f(x) \geq f_n^*$. The improvement in our result is then $\max(0, f_n^* - f(x))$. We wish to choose x such that this improvement is large, but we cannot determine this as $f(x)$ is not known until after x is determined. We thus take the expected value of the improvement. This is defined as

$$\mathrm{EI}_n(x) := E_n[\max(0, f_n^* - f(x))], \tag{3.8}$$

where the expectation is taken under the posterior probability distribution defined by equation (2.15) at the points we have already evaluated the target function on.

We can then calculate the expected improvement using integration by parts [14].

$$\mathrm{EI}_n(x) = [\Delta_n(x)]^+ + \sigma_n(x)\varphi\left(\frac{\Delta_n(x)}{\sigma_n(x)}\right) - |\Delta_n(x)|\,\Phi\left(\frac{\Delta_n(x)}{\sigma_n(x)}\right), \tag{3.9}$$

where $\Delta_n(x) := \mu_n(x) - f_n^*$ is the expected difference in the quality of the result between the last optimum point and the new point. $\phi(.)$ and $\Phi(.)$ are the standard normal density and distribution functions. Subsequently, the next point to evaluate the objective function on can be selected using

$$x_{n+1} = \operatorname{argmax} \mathrm{EI}_n(x). \tag{3.10}$$

Various techniques can be applied to solve (3.10) as this is inexpensive to evaluate. We utilize LBFGS, which is a quasi-Newton method [20]. For our implementation, we utilize the skopt library in Python, which has Bayesian Optimization using Gaussian Processes [11]. This minimizes the target function by default. We stick to the EI acquisition function without any noise for our implementation of this variant in the parametric Kernel Flows algorithm.

During optimization, similar to the effect seen in both finite difference and automatic differentiation-based optimization, we found that for a kernel with very poor parameters it was possible for ρ to take on values beyond the range $[0, 1]$. Values larger than 1 are not a problem since they are simply seen as poor values for a minimization problem, but values lower than 0 are an issue since they are seen as optimal values when minimizing ρ whereas in reality, they are unstable values. In order to tackle this, we clamp negative values to 1 so that values resulting from unstable kernels are not considered minimum values. Moreover, it was also observed that ρ took on non-negative values that were very close to zero, but the performance of such kernels was nevertheless poor. We added further robustness to the optimization. Whenever Bayesian Optimization resulted in a new minimum, we evaluated ρ with the same parameters but on a different batch. If the new value was such that $|\rho_{new} - \rho_{old}| \leq \epsilon$ for some threshold ϵ, then the change was accepted, otherwise this value of ρ was also clamped to 1 in order to remove the acceptance of unstable kernels. This further meant, that the Bayesian optimization often evaluated the kernel more than the total iteration count provided by the user when it had to check for robustness. This was indeed an extra computational cost, but we did not have to perform the same number of iterations as we were performing in finite difference and automatic differentiation-based optimization. Hence, this was a cost that we could accept given that we got the advantage of having a very robust method. Through our experiments, we will show how these extra evaluations were indeed not a significant overhead in light of the benefits they resulted in.

3.6. Parametric Kernel Flows Results

We have looked at how the Parametric Kernel Flows algorithm can be adapted with respect to the optimization method used, providing us with three distinct variations of implementing it. We discussed the limitations and benefits of the methods generally. In this section, we will run concrete experiments using the datasets discussed in section 3.2 to analyze the resulting improvements and the computational aspects of the implementations on NNGP kernels.

3.6.1. Parametric Kernel Flows on Convolutional NNGP Kernels

We will first look at the Convolutional NNGP kernels, utilizing the implementation from Garriga-Alonso et al. This is because we have a CNNGP kernel implementation in Pytorch which makes it easy for us to exploit the automatic differentiation engine for the parameter update step of algorithm 1 by making the minor changes to the implementation from Garriga-Alonso et al as discussed in section 3.4.2. In particular, we will look at the

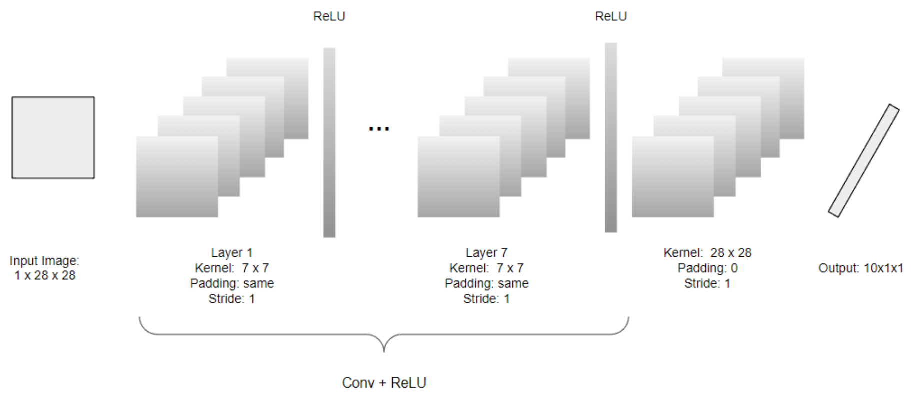
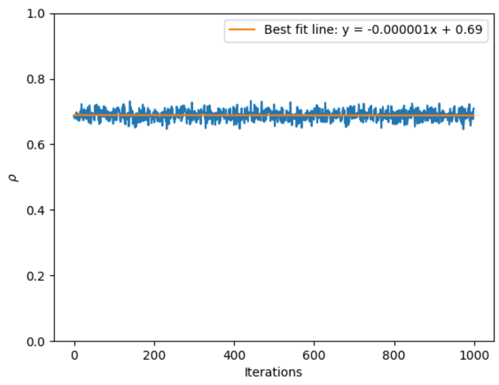


Figure 3.9.: ConvNet-GP Architecture used by [9] for MNIST dataset.

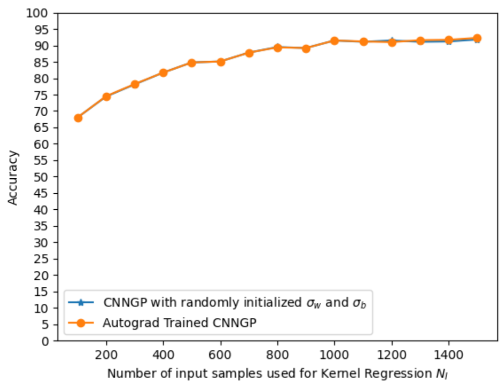
simple-three-layer CNN architecture presented in figure 3.7 and the ConvNet-GP, which is utilized in [9, 23]. The ConvNet-GP utilizes 8 convolution layers where each layer except the last is succeeded by a ReLU activation. For testing with the MNIST dataset, each layer except the last has a kernel size of 7×7 with 'same' padding, while the last layer has a kernel size of 28×28 with zero padding, making the final layer equivalent to a dense layer. Again, the limit of the number of feature maps per layer go to infinity so that this CNN can be represented as a Gaussian Process. The architecture is summarized in figure 3.9. Good candidates for the variance of weight and bias for this network for the MNIST dataset have been found using a random grid search, allowing us to make a nice comparison to see if our training using parametric Kernel Flows resulted in improved performance than finding these parameters via random grid search [9]. We also utilized this architecture for the CIFAR-10 dataset. However, a few changes needed to be made to adapt to the different image sizes in this dataset. Hence, the first seven convolution layers were changed to have a kernel of size 8×8 and the final layer was changed to have a kernel of size 32×32. The rest remained the same.

Gradient Descent with Automatic Differentiation

We start our experiments with the parametric Kernel Flows algorithm on the three-layer convolutional neural network equivalent GP kernel for the MNIST dataset. In section 3.5.2 we saw how with a batch size $N_f = 450$ we ran out of memory on the GPU without having a custom automatic differentiation function for the ReLU activation covariance matrix. Hence, we only use the altered CNNGP kernel implementation which implements the backward pass of the ReLU activation covariance. In order to evaluate the quality of a kernel on the classification tasks at hand, we used accuracy as our testing measure, as shown in equation (3.2).



(a) Progression of ρ for Kernel Flows using automatic differentiation.

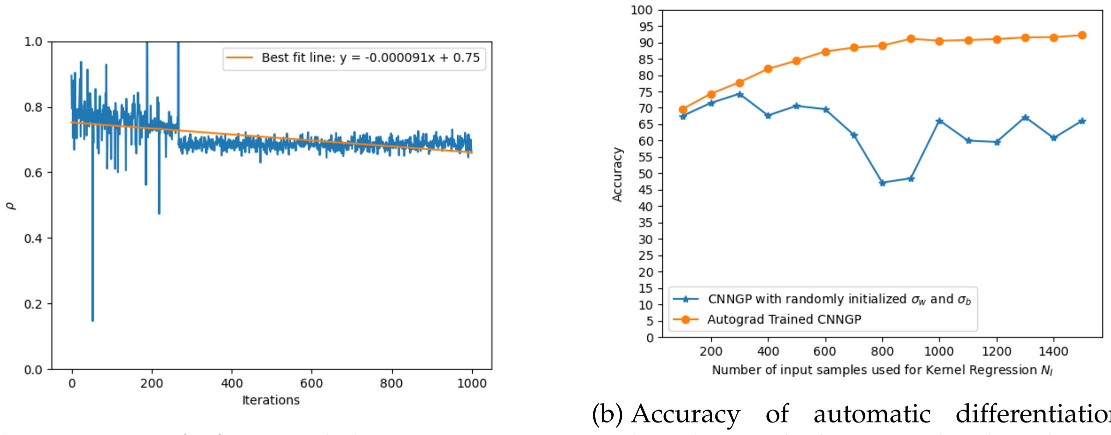


(b) Accuracy of automatic differentiation on the three-layered network with good initialization of parameters.

Figure 3.10.: Training of the three-layered CNN GP Kernel using parametric kernel flows algorithm with the `Pytorch` automatic differentiation engine (autograd) on the MNIST dataset with good initialization of parameters.

We perform Kernel Flows with a batch size of $N_f = 600$ with randomly initialized values for σ_w^2 and σ_b^2 and a sample proportion of 0.5, i.e., $N_c = 300$. The learning rate is set at 0.1, regularization at 0.0001 and we perform 1000 iterations. The optimization step is done using stochastic gradient descent. The progress of ρ versus the iteration count can be seen in figure 3.10a. The accuracy of Kernel Regression performed with the untrained randomly initialized kernel and the same kernel trained using Kernel Flows can be seen in figure 3.10b.

We see only a very minor improvement in results when more than 1000 sample images are used in the Kernel Regression step, and the value of ρ also remains stable. This is because the randomly initialized kernel was already well-performing. However, this is not always the case. We repeat the same experiment but this time with a purposefully poor choice of parameters σ_w^2 and σ_b^2. The results are shown in figure 3.11 and we see how this time the accuracy from performing kernel regression with the trained kernel is much better. Notice how the values of ρ are unstable in the beginning due to the poor choice of the parameters, as shown in figure 3.11a. In fact, it was observed for some evaluations the absolute value of ρ became very large resulting in an undefined gradient leading to a failure of the automatic differentiation-based Kernel Flows algorithm. This makes the method fragile as one has to take care of the initial state of the kernel with respect to its parameters. One possible approach to making the optimization more stable, which we implemented and discussed in section 3.5.2, was to observe if ρ attains an undefined value and, if so, to reinitialize the kernel and then optimize this new kernel. However,



(a) Progression of ρ for Kernel Flows using automatic differentiation.

(b) Accuracy of automatic differentiation-based Kernel Flows on the three-layered network with poor initialization of parameters.

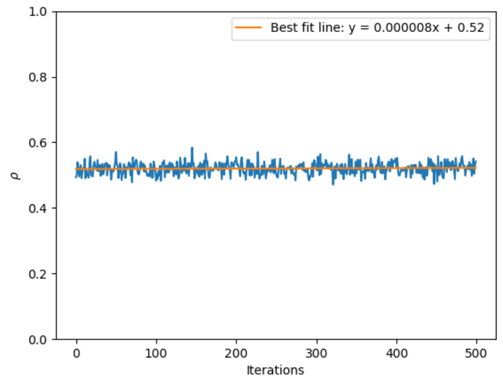
Figure 3.11.: Training of a poorly initialized three-layered CNN GP Kernel using parametric kernel flows algorithm with the `Pytorch` automatic differentiation engine (autograd) engine on the MNIST dataset with poor initialization of parameters.

this leads to a waste of computational resources as all evaluations prior to this would not have contributed to improving the kernel. Moreover, despite our custom implementation of the ReLU kernel function, we still ran into memory issues when using architectures of greater depths such as the ConvNet-GP. Indeed, it was not possible to run this variation of the parametric Kernel Flows algorithm on the ConvNet-GP architecture with batch sizes larger than 100 as we once again ran out of memory on the GPU. Thus, we could not present experimental results of using the automatic differentiation-based gradient descent optimization for the parametric Kernel Flows algorithm on the ConvNet-GP kernel.

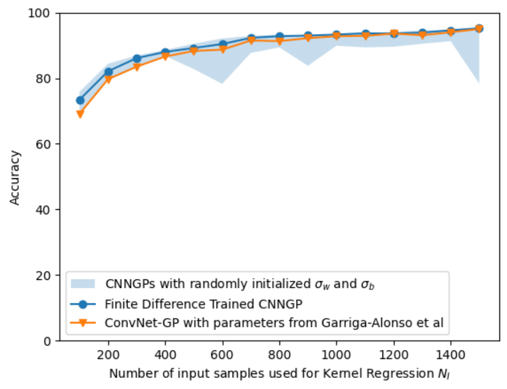
Due to the prohibitive memory limitations, we were also unable to test any kernel with the CIFAR-10 dataset. Hence, we will move on to other optimization techniques that are not memory-bound.

Finite Differences

We will now look at the results of optimizing a CNNGP kernel with gradient approximation instead of exact gradient evaluation for the gradient-based optimization step. As we saw previously, since the memory bottleneck is resolved by the finite difference approximation of the gradient as no computational graph is maintained, we can perform Kernel Flows on deeper CNNGP kernels with larger batch sizes. We work with the ConvNet-GP for the MNIST dataset and initialize five ConvNet-GP kernels with parameters σ_w^2 and σ_b^2



(a) Progression of ρ using finite difference-based Kernel Flows.

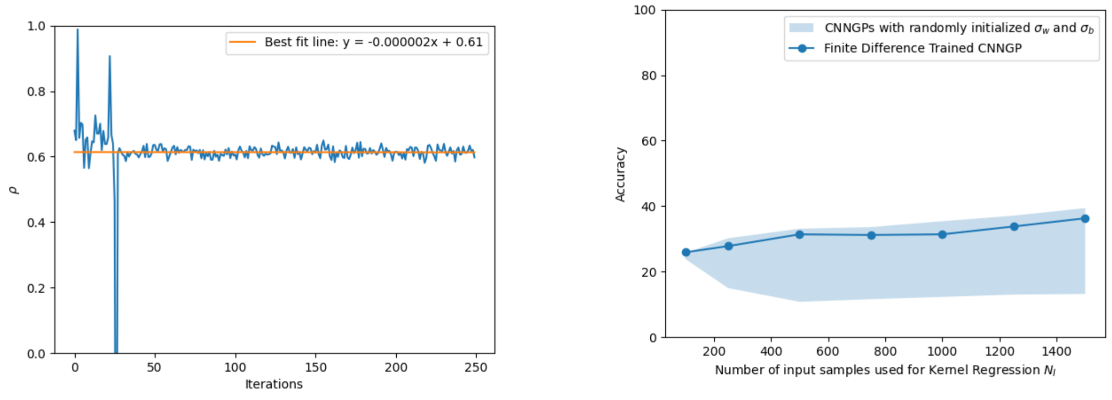


(b) Accuracy obtained on the test set using kernel regression with kernel trained using finite difference based Kernel Flows.

Figure 3.12.: Training of a poorly initialized ConvNet-GP Kernel using parametric kernel flows algorithm with finite differences on the MNIST dataset.

selected randomly. We first evaluate their accuracy using kernel ridge regression on the MNIST dataset for different training dataset sizes N_I with regularization of 0.0001. Subsequently, we select the worst-performing ConvNet-GP and apply the finite difference-based Kernel Flows algorithm to it for 500 iterations with a batch size of 600 and a learning rate of 0.1. We note that this is a small number of iterations because evaluating the ConvNet-GP on 600 images three times on each iteration proved to be expensive. Moreover, we did not observe the value of ρ to show a meaningful decline as can be seen in figure 3.12a. Nevertheless, the evaluation of the ConvNet-GP with parameters optimized after training proved to be better than the worst initialized kernel and even better than the ConvNet-GP that utilized hyperparameters from [9]. The solution, however, did not outperform the best-performing randomly initialized ConvNet-GP.

We perform the same experiment with the CIFAR-10 dataset and use the ConvNet-GP kernel adapted to cater to the CIFAR-10 dataset. Again, we initialize five random ConvNet-GP kernels and evaluate their performance in kernel ridge regression on the test set using various training images. The range of accuracy is shown in the blue shaded area in figure 3.13b. Once again, we select the worst performing kernel and perform the parametric Kernel Flows algorithm with the same parameters as before, except this time we only perform 250 iterations as the CIFAR-10 dataset contains images that are much larger, meaning each iteration step is expensive. We soon find that the poor initialization of the kernel leads to very erratic values of ρ. This can be seen in the initial spikes in figure 3.13a. Eventually, ρ becomes undefined and our robustness check re-initializes the kernel and optimizes the newly initialized kernel. The new kernel is much more stable as we can see



(a) Progression of ρ using finite difference-based Kernel Flows.

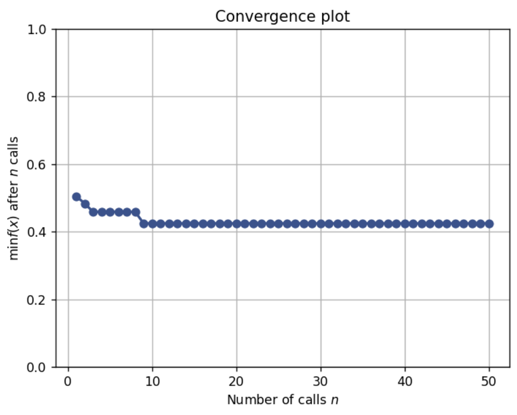
(b) Accuracy obtained on the test set using kernel regression with kernel trained using finite difference based Kernel Flows.

Figure 3.13.: Training of a poorly initialized ConvNet-GP Kernel using parametric kernel flows algorithm with finite differences on the CIFAR Dataset

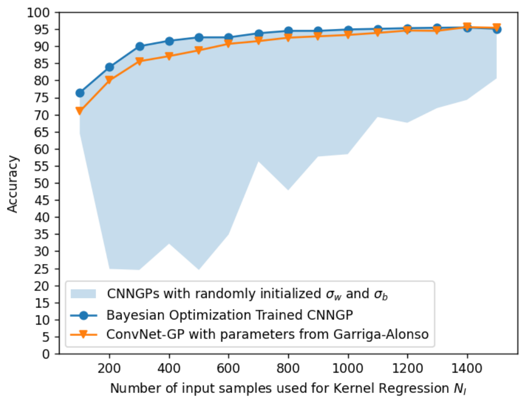
that the remaining values of ρ do not fluctuate. Nevertheless, we once again do not see a significant decline in the value of ρ. The kernel after optimization is tested, and we observe that while it is better than most of the randomly initialized kernels, it does not outperform the best-performing kernel. This is similar to the effect we saw in our experiment with the MNIST dataset. There are many possible reasons why this might happen. One possibility is that, since we use gradient approximation instead of exact gradients, our approximate gradients might not be accurate enough. Another possibility is that our gradient-based optimization gets stuck in a local minimum. Finally, it could simply be the case that the optimization does not converge since we only perform a limited number of iterations in both cases.

Bayesian Optimization

We can now finally move to the experiments with Bayesian Optimization as the optimization routine in the parametric Kernel Flows algorithm. We will first test the method on the MNIST dataset with the ConvNet-GP we have been using. Again, we initialize 5 random kernels and evaluate the accuracies from each with regularization of 10^{-4} with a different number of data points for training N_I on 1000 test points. Subsequently, we apply the Bayesian Optimization variant of the parametric Kernel Flows algorithm. We train using a batch size $N_f = 1200$ as we are not limited by memory constraints, and perform 50 iterations with 15 random initial starts. The results are shown in figure 3.14b. It can be seen that Bayesian optimization takes the accuracy right up to the best-performing kernel. Moreover, it is interesting to note that after just 9 iterations, we have reached our current



(a) Progression of ρ after Kernel Flows training of ConvNet-GP with Bayesian Optimization.

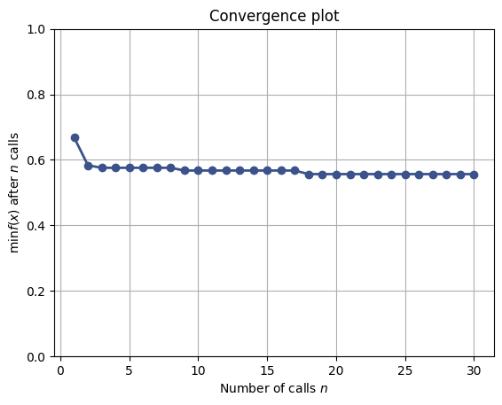


(b) Accuracy results after Kernel Flows training of ConvNet-GP Architecture with Bayesian optimization.

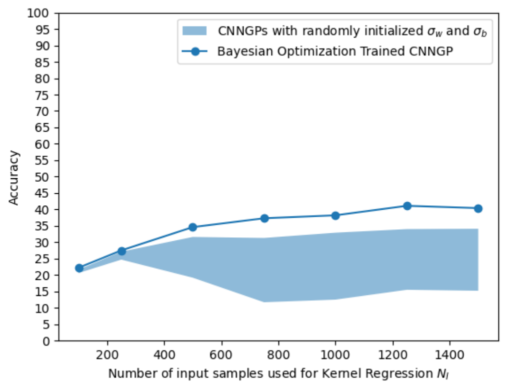
Figure 3.14.: Training of the ConvNet-GP Architecture using parametric kernel flows algorithm with Bayesian optimization on the MNIST Dataset.

optimum and we do not see any improvements beyond this, as seen in figure 3.14a. We also note that our optimized kernel performs even better than the ConvNet-GP used by [9] which was selected via random search.

We repeat the same experiment with the CIFAR-10 dataset and the adjusted ConvNet-GP kernel for the dataset. All other parameters were kept the same as with the MNIST experiment except this time we only perform a total of 30 iterations. Once again, we test 5 random kernels and plot the band of their accuracies. The results in figure 3.15b show that the kernels trained via Bayesian Optimization based Kernel Flows performs much better than all the randomly initialized kernels, including the best one. Comparing these results to the ones where we performed optimization with automatic differentiation and finite differences, we observe that the overall kernel improvement using Bayesian Optimization was the best and required the least number of iterations, meaning the least number of kernel evaluations as well, making it the most efficient in terms of computational complexity. Indeed, the accuracy even after performing Kernel Flows with Bayesian Optimization reached only close to 45% whereas the state-of-the-art learning methods have been able to achieve accuracies of 99% on the CIFAR-10 dataset [13]. These methods utilize highly specialized convolutional neural networks, which are well designed for image processing tasks. Since we perform kernel ridge regression, we do not exploit the structures that are embedded in images but only interpolate based on the available data using the kernel at hand, which hinders us from achieving state-of-the-art accuracy with this dataset.



(a) Progression of ρ after Kernel Flows training of ConvNet-GP with Bayesian optimization.



(b) Accuracy results after Kernel Flows training of ConvNet-GP Architecture with Bayesian optimization.

Figure 3.15.: Training of a ConvNet Kernel using parametric kernel flows algorithm with Bayesian optimization on the CIFAR-10 Dataset.

Performance Comparison

Let us summarize the performance improvement of the kernels on unseen data across the three optimization method variants of the parametric Kernel Flows method. We try to optimize the same CNNGP kernel using the three methods. Since the ConvNet-GP cannot be optimized using the automatic differentiation variant, we stick to the three-layered architecture shown in figure 3.7 and work with the MNIST dataset. The same kernel is provided for the three variants. All three variants work with a regularization of 0.0001. The finite difference and automatic differentiation variants share the same learning rate of 0.1, an iteration count of 500, and a batch size of $N_f = 600$ while the Bayesian Optimization variant uses a larger batch size of $N_f = 1200$ and an iteration count of 30. We plot the accuracy of the kernel on the test dataset for each of the optimizations and for the untrained kernel as well. This is plotted in figure 3.16.

We see that while all the variants of the algorithm improve the performance of the initial kernel, the Bayesian Optimization variant is the best performing. This is underscored even more when we realize that this provided the shown results with a drastically reduced number of iterations compared to its gradient-based optimization counterparts.

3.6.2. Parametric Kernel Flows on Dense NNGP Kernels

We have till now seen the implementation and results of the parametric Kernel Flows algorithm on Convolutional NNGP kernels. In this section, we will extend the algorithm to dense NNGP kernels. The intrinsic idea of the algorithm with all three variants of opti-

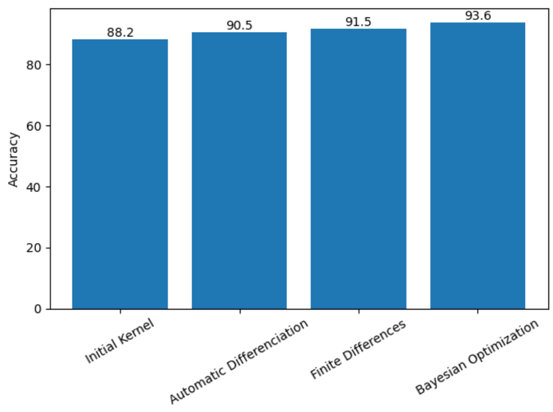
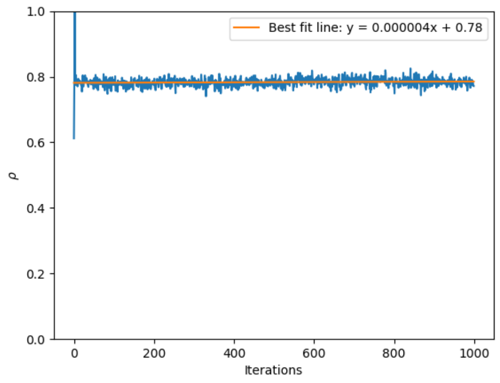


Figure 3.16.: Comparison of the accuracy of the CNNGP kernel trained on the MNIST dataset using the three variants of the parametric Kernel Flows algorithm.

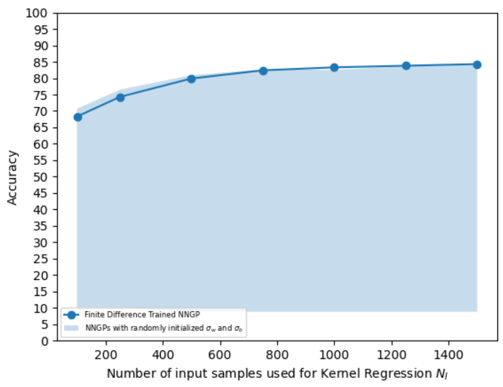
mization remains exactly the same. The only difference is the specifics of the implementation. As discussed in section 3.5.1, we used the Neural Tangents library for Dense NNGP kernels [25]. We had the limitation here that unlike the implementation of CNNGP kernels in `Pytorch`, we could not convert the weight and bias variances σ_w^2 and σ_b^2 into trainable parameters as the library did not allow for this. Hence, we were unable to use the JAX library in this case, which is the automatic differentiation engine used by Neural Tangents [3]. However, this is not a loss since we have seen that Kernel Flows on NNGP kernels with automatic differentiation is a very costly and unstable procedure. Hence, we stick to applying the finite difference and Bayesian optimization variants to dense NNGP kernels. Moreover, in this case, we perform evaluations on only the MNIST dataset to demonstrate compliance with prior results from CNNGP kernel optimization. In both techniques, we work with the same architecture, an equivalent GP kernel of a dense NN with 2 hidden layers each with 25 neurons, and an output layer with 10 neurons so as to produce the ten classes of the MNIST dataset. Note that we will only show experiments with the MNIST dataset since the overall results are to demonstrate the extensibility of the parametric Kernel Flows algorithm and its variants.

Finite Difference

We conduct the same experiment we did with the CNNGP kernels for the dense kernels for the finite differences variant of the parametric Kernel Flows algorithm on the MNIST dataset. We initialize 5 random NNGP kernels and measure their accuracy on the MNIST dataset with various numbers of input data points for kernel ridge regression N_i. Like the experiments with CNNGP kernels, this is shown with the translucent blue band. We select the worst-performing kernel and use it as the initial kernel for finite difference-based optimization. We use a batch size $N_f = 600$ and perform 1000 iterations, keeping reg-



(a) Progression of ρ after Kernel Flows training of a Dense NNGP with Kernel Flows using finite difference.



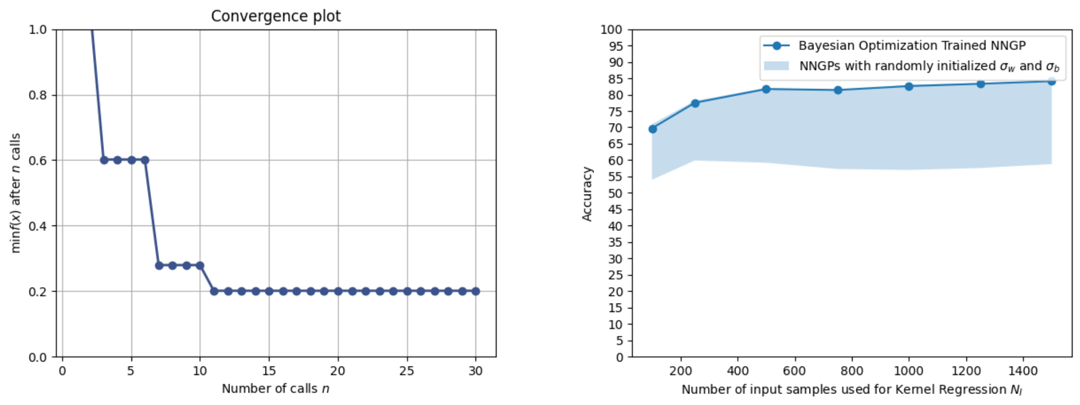
(b) Accuracy results from performing kernel regression after training the of a Dense NNGP kernel using Kernel Flows with Finite Difference.

Figure 3.17.: Training of a Dense NNGP Kernel with Finite Difference-based Kernel Flows on the MNIST Dataset.

ularization at 0.00001 and the learning rate at 0.1. The results in figure 3.17b show that after optimization, the worst-performing kernel improves to roughly the best-performing kernel. The initial parameters were particularly bad, which is reflected in the initial spikes shown in figure 3.17a and indeed resulted in undefined values, which forced our implementation to reinitialize the kernel.

Bayesian Optimization

We also perform the parametric kernel flows algorithm using Bayesian Optimization as our optimization step on the MNIST dataset. The initial procedure is again similar, where we initialize five random kernels and measure their accuracies. Subsequently, we apply the Kernel Flows algorithm for 30 iterations with a batch size $N_f = 1200$ keeping regularization at 0.0001. We observe from the convergence graph shown in figure 3.18a that just after 11 iterations we have our optimal result. We observe in figure 3.18b that the optimal result performs as well as the randomly selected kernel that performed the best. Once again, the Bayesian Optimization variant of the parametric Kernel Flows algorithm provided very valuable results with the minimal computational expense. We will discuss the computational aspects of these variants more quantitatively now in the next section.



(a) Progression of ρ after Bayesian Optimization based Kernel Flows training of the Dense NNGP kernel.

(b) Accuracy results from performing kernel regression after Bayesian Optimization based Kernel Flows training of the Dense NNGP.

Figure 3.18.: Training of a Dense NNGP Kernel with Bayesian Optimization-based Kernel Flows on the MNIST Dataset.

3.6.3. Computational Expense

Profiling Results

To check what part of the execution of Kernel Flows with CNNGP Kernels became the computational bottleneck, we utilized profiling tools that can provide information regarding the number of calls to particular functions and the time taken to execute various parts of the code. We used `cProfile` which is a standard Python library to profile our code. We profile the execution for all three variants presented previously with a batch size N_f of 300, executing for 10 iterations each for the MNIST dataset. We utilize the three-layered CNN architecture shown in figure 3.7 since otherwise, we will not be able to profile the automatic differentiation variant due to the memory bottleneck. Other parameters are not important as they do not impact the computational cost which we are studying. Since our evaluations get executed on a GPU, we also run into the problem of asynchronous execution, where the CPU might stop measuring the time execution. At the same time, the GPU continues to process data asynchronously. To avoid this, we synchronized time measurements between the CPU and the GPU and subsequently measured the CPU time only [23]. The results are summarized in table 3.3.

We note that the most expensive variant is the one utilizing automatic differentiation for gradient evaluations despite having the least number of kernel evaluations. Most of the evaluation time here is spent in the backpropagation step. This, coupled with the memory bottlenecks, makes this variant of the parametric Kernel Flows algorithm infeasible for NNGP kernels, particularly when the underlying Neural Network is deep. For the

Table 3.3.: Results from Profiling the implementation of the 3 variants of the Parametric Kernel Flows algorithm.

Optimization Technique	Kernel Evaluation		% Time in Backpropogation	Total Time (s)
	No of Evaluations	% Time Consumed		
Auto Differentiation	10	10.48 %	88.47 %	85.81
Finite Difference	30	89.12 %	-	30.30
Bayesian Optimization	13/10	88.10 %	-	12.42

Table 3.4.: Time taken in seconds to perform a single iteration of the different variations of the Parametric Kernel Flows algorithm.

Optimization Method	**3 Layer CNN GP Kernel**		**ConvNet-GP Kernel**	
	Batch size $N_f = 600$	Batch size $N_f = 1200$	Batch size $N_f = 600$	Batch size $N_f = 1200$
Automatic Differentiation	0.736	Out of Memory	Out of Memory	Out of Memory
Finite Difference	0.667	2.715	8.667	34.916
Bayesian Optimization	0.385	1.243	3.969	15.778

remaining two methods, we notice that the bulk of the time was spent on the kernel evaluations. We recall that for the finite differences variant, we had to evaluate a kernel thrice in each iteration, and hence we see 30 kernel evaluations for 10 iterations. Subsequently, the time taken for the finite difference is also roughly thrice that of the Bayesian Optimization variant. Note for the Bayesian Optimization variant: 13/10 refers to a total of 13 kernel evaluations, of which 10 were direct calls and 3 were recursive calls. These recursive calls happened due to the robustness checks added to the Bayesian optimization algorithm. We conclude that for any given number of iterations of the Kernel Flows algorithm, the Bayesian Optimization variant proves to be the most efficient in terms of time consumption. This is further underscored by the fact that this variant required orders of magnitude fewer iterations than its gradient-based optimization counterparts.

Execution Time For Various Optimization Methods

In this section, we will compare the run times of the different optimization techniques we have utilized. Our aim is not only to get the version of the algorithm that simply gets the best kernel, but also one that does so in a reasonable time without requiring excessive resources.

For this, we run all our optimization methods multiple times and calculate the total time taken using the MNIST dataset. We perform this for batch sizes of 600 and 1200 on both the three-layered CNNGP kernel and the ConvNet-GP kernel. Subsequently, we find the time taken per iteration by dividing the average of the total time taken across the many runs by the number of iterations. The results are presented in table 3.4.

As noted in the profiling results, the Bayesian Optimization variant takes the minimum time. Moreover, we observed that in our experiments with the ConvNet-GP Kernel, we were able to achieve significant improvement compared to the initial kernel after 50 iterations across many evaluations. This means that to get meaningful results from Bayesian optimization, we require an execution time of about 13 minutes. This is, of course, not the case for the finite differences, where we have to conduct hundreds of iterations. In the experiment with ConvNet-GP using finite difference, we worked with a batch size of 600 and performed 500 iterations. This alone took us 72 minutes of execution time. We see a vast difference between the computational expense of the two methods. Moreover, not only is Bayesian Optimization more economical computationally, but it also delivers much better results and is more robust to misbehaving values of ρ induced via a poor kernel. We can thus conclude that for NNGP kernels, out of the three variants, the one utilizing the Bayesian Optimization step is the best choice for the parametric Kernel Flows algorithm.

4. Non-Parametric Kernel Flows

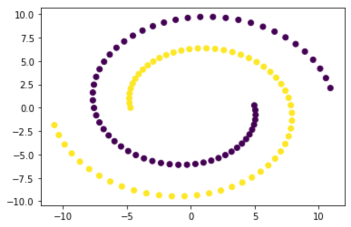
We now shift our focus to the non-parametric version of the Kernel Flows algorithm. This version of the algorithm perturbs the input data instead of the kernel hyper-parameters to improve the performance of the kernel. As a warm-up, we repeat the experiment done by both [26, 5] on the Swiss roll cheesecake dataset. This dataset consists of two Swiss rolls where each has an assigned target of either 1 or -1. This is shown in figure 4.1a. We found the non-parametric Kernel Flows method separates the dataset such that they can be classified linearly. The experiment was performed with RBF kernel parameter σ as 2, an adaptive ϵ with a base value of 0.01, and regularization of 0.01 for 35,000 iterations with a batch size N_f of 32 and sample N_c half the size of the batch. The results are shown in figure 4.1b.

4.1. Flow on 1D Dataset

To investigate the transformation of the data points in more detail, we simplified the experiment to a one-dimensional scenario. We shift our focus to a very simple regression task of fitting a curve. The function we try to fit is

$$f(x) = sin(2x) + 3sin(3x) + 2sin(4x). \tag{4.1}$$

We refer to this as the three bumps dataset, which is shown in figure 4.2. All the perturbations only happen in the x dimension now, as y is the target, so we can observe the


(a) The Swiss roll Cheesecake Dataset with 120 data points.

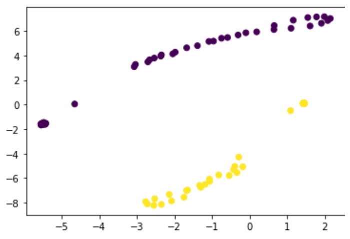

(b) The transformed Swissroll Cheesecake dataset after performing Non-Parametric Kernel Flows.

Figure 4.1.: Performing Non-Parametric Kernel Flows on the Swiss roll Cheesecake dataset using the RBF kernel.

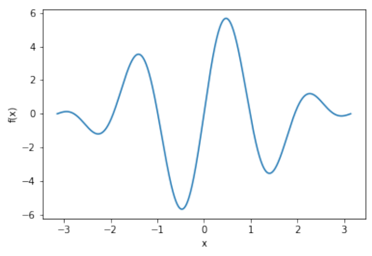


Figure 4.2.: The three bumps dataset for 1D regression task with function as shown in equation (4.1).

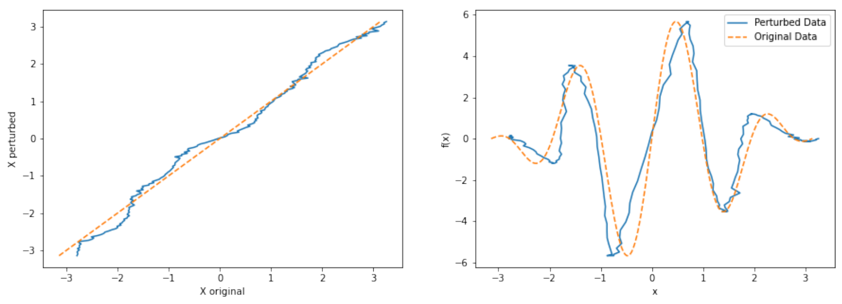


Figure 4.3.: Performing Non-Parametric Kernel Flows on the Three Bumps dataset.

flow in more detail. We perform the non-parametric kernel flows on this as defined in Algorithm 2 and as implemented by [5].

The entire dataset has 200 points. We perform the non-parametric Kernel Flows with the RBF kernel as shown in equation (2.9) with parameter σ as 4, an adaptive ϵ with a base value of 0.1, and regularization of 0.0001 for 20,000 iterations with a batch size N_f of 64 and sample N_c half the size of the batch. The results are shown in figure 4.3. The left panel shows the original x points plotted against the perturbed points and the right panel shows the original dataset vs the perturbed dataset after executing the algorithm.

Since we are using an RBF kernel with a large parameter value, we would have expected the dataset to smooth out so that our kernel fits the dataset well. However, we see that this does not happen. Secondly, we notice that the perturbed points are changed such that they cross each other in the form of horizontal spikes at specific points in the plot. This is highly irregular because if we refer to Algorithm 2, the vector g is in fact the vector field and the final step is essentially a forward Euler step. Given that we perturb our data points by a vector field, we do not expect the points to cross each other. We postulate that the update step, using the forward Euler method (with a varying time step), is perhaps not of sufficient accuracy, which results in the crossings we observe.

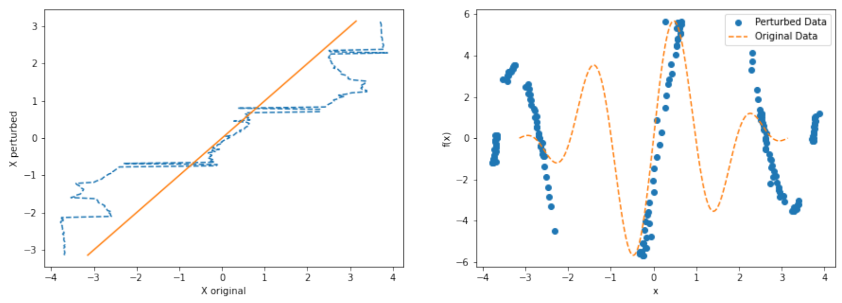


Figure 4.4.: Performing Non-Parametric Kernel Flows using RK45 to solve an initial value problem in each iteration.

4.2. ODE Solver Based Non-Parametric Kernel Flows

With this hypothesis in mind, we set out to change the final step in Algorithm 2 to the Runge Kutta 45 (RK45) update step [7]. The crux of the algorithm remains the same, i.e. we still calculate $\hat{g}_{f,i}^{(n)}$ using equation (2.43) and then interpolate for all data points not in the batch using equation (2.44). However, now the perturbations are not simply added to our data points using the forward Euler step, but rather we solve an initial value problem using the RK45 method. It is important to note that the batch and sample datasets remain the same during one iteration and hence also during one IVP solve call. We use `scipy` for the purposes of solving this initial value problem. The parameters are kept the same as the previous run except that this time we only executed 1000 iterations. Moreover, since we use the RK45 ODE solver, we do not require the ϵ parameter at all, instead we simply integrate the equation for a time span from 0 to 1. The results can be seen in the figure 4.4. We observe that even though the data is now more spread out, the plot of the perturbed data against the original data shows that there are crossings in the vector field [1]. We understand that by using a more accurate ordinary differential equation (ODE) solver, the data was indeed perturbed so as to fit the kernel in fewer iterations. However, we must still explore why the crossings happen.

4.3. Complete Interpolation of Vector Field

In order to investigate why crossings happen when we solve an IVP for the given vector field $\hat{g}_{f,i}^{(n)}$, we must have a closer look at Algorithm 2 and particularly at [26, Equation 6.6] or equation (2.43) alongside the note that follows. We observe that this equation consists of two parts in its numerator. The first part is $(1-\rho(n))\hat{y}_{f,i}^{(n)T}\left(\nabla_x K_1\right)\left(x_{f,i}^{(n)}, x_{f,\cdot}^{(n)}\right)\hat{y}_f^{(n)}$. $(1-$

[1]A video of how the points are perturbed using this method can be viewed on this link: https://youtu.be/U6A1DUqdHXg

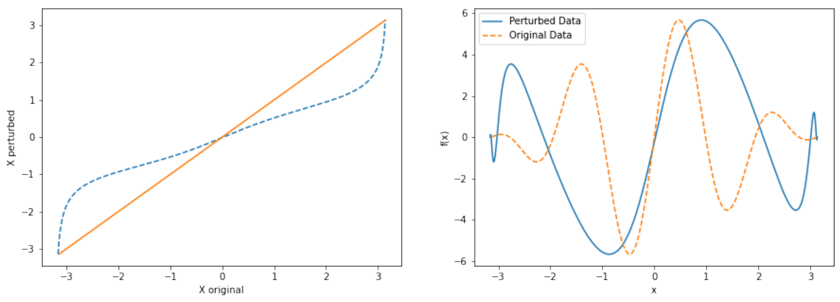


Figure 4.5.: Performing Non-Parametric Kernel Flows using RK45 to solve an initial value problem in each iteration and interpolating the acquired vector field to all input data points instead of just to the points not in the batch.

$\rho(n))$ is simply a scalar and $(\nabla_x K_1)\left(x_{f,i}^{(n)}, x_{f,\cdot}^{(n)}\right)\hat{y}_f^{(n)}$ is a matrix-vector multiplication. The result of this matrix-vector multiplication goes through an element-wise multiplication with $\hat{y}_{f,i}^{(n)}$. This way we get a vector in $\mathbb{R}^{N_f}$. However, more interesting is the second term in the calculation of the vector field namely $\hat{z}_{f,i}^{(n)T}(\nabla_x K_1)\left(x_{f,i}^{(n)}, x_{f,\cdot}^{(n)}\right)\hat{z}_f^{(n)}$ and more specifically $\hat{z}_f^{(n)} := \left(\pi^{(n)}\right)^T\left(\pi^{(n)}\Theta^{(n)}\left(\pi^{(n)}\right)^T\right)^{-1}\pi^{(n)}y_f^{(n)}$. This is essentially multiplying the kernel matrix resulting only from the N_c sample points with the sample targets resulting in a vector in $\mathbb{R}^{N_c}$. However, the final multiplication with $\left(\pi^{(n)}\right)^T$ from the left projects the resulting vector field back to $\mathbb{R}^{N_f}$ setting the elements that are not in the sample set but in the batch set to 0. This means that we essentially extract two vector fields. One for the batch and one for the sample set and once these get to the ODE solver, two separate problems are solved, which is why we observe the crossings. Moreover, the acquired vector field is then interpolated only for the points that are not part of the batch (and hence also not the sample), while the vector fields for the batch and sample data points remain the same. This ultimately results in the final vector field $G_{n+1}(x)$ not being smooth. We need to convert this vector field into a smooth function so that a proper vector field is obtained.

We tackle this problem by interpolating the entire dataset instead of just the points that are not in the batch as was done originally. This changes equation (2.44) to

$$G_{n+1}(x) = \left(\hat{g}_f^{(n)}\right)^T\left(K_1\left(x_f^{(n)}, x_f^{(n)}\right)\right)^{-1}K_1\left(x_f^{(n)}, x\right). \tag{4.2}$$

With this change to the original algorithm, along with an IVP solver using RK45, we rerun the test with the three bumps dataset again. All parameters were kept the same as before. We can see the results in figure 4.5 [2].

[2]A video showing how the points are perturbed using this method can be viewed on this link: https://youtu.be/1mBDTSJ9y6U

This time we observe that the perturbations happen without crossings, and indeed the original function is more spread out such that it fits better with our RBF kernel, which has a high parameter value.

4.4. Evaluation of Improved Non-Parametric Kernel Flows on Regression Tasks

In order to make the predictions for some test data points, we first need to perturb the test points. Of course, we cannot evaluate the perturbation vector for these due to the absence of the target values, so instead, we store the batches used during training. Subsequently, in the transformation phase of the test dataset, we use these saved batches to evaluate the vector field and thereafter interpolate the vector field for both the batch and the test set to get the entire vector field that is used by the IVP solver. It is important to note that the integration time span, iteration count, interpolation method, and regularization constant should be the same during the training and testing phases. Once the test points are transformed, we can use kernel ridge regression by utilizing the perturbed training and test points instead of the original ones. Let us see this in action with the three bumps dataset.

In order to have a baseline to compare with, we first perform normal Kernel Ridge Regression using 80 training points and 200 test points with an RBF kernel with parameter value 4. The regularization is again at 0.0001. The results can be seen in figure 4.6a. We use the mean squared error to calculate the effectiveness of the prediction on the test dataset, which is given by

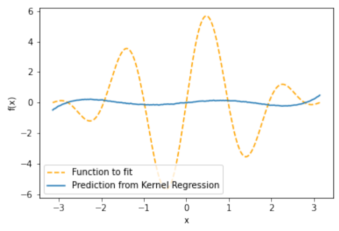
$$\text{MSE} = \frac{1}{N_*}\Sigma_{i=1}^{N_*}(\hat{y}_i - y_i)^2, \tag{4.3}$$

where N_* is the number of test points we have, $\hat{y}_i$ are the predicted target values, and y_i the actual target values.

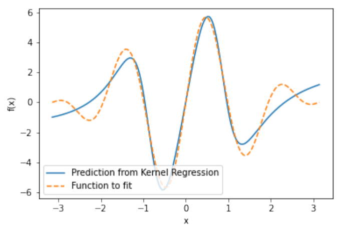
The resulting mean squared error was 6.865. We then perform the experiment again but this time train the kernel with the modified Non-Parametric Kernel Flows algorithm on a dataset of size 80 with a batch size $N_f = 64$, sample proportion half that of the batch size, and regularization of 0.0001 for 1000 iterations. For test points, we use the same 200 data points as before. The resulting prediction is shown in figure 4.6b which results in an MSE of 0.504.

We can see that the prediction has considerably improved, and the mean squared error has dropped significantly after modifying the kernel using Non-Parametric Kernel Flows. We can also observe in figure 4.7 that the value of ρ decreases as the iterations proceed.

Despite these nice results, our problems are not entirely solved yet. Since the vector field, $\hat{g}_f^{(n)}$ consists of two separate vector fields as explained above, if the interpolation technique is such that it is able to recover the difference to a great extent, then the problem we had been encountering until now will re-emerge. We demonstrate this with another



(a) Applying Kernel Ridge Regression with an RBF Kernel using $\sigma = 4.0$ resulting in an MSE of 6.865.



(b) Applying kernel ridge regression using an RBF Kernel using $\sigma = 4.0$ that is trained using Non-Parametric Kernel Flows, resulting in an MSE of 0.504.

Figure 4.6.: Kernel Ridge Regression with RBF Kernel using $\sigma = 4.0$ with and without Non-Parametric Kernel Flows to improve the base Kernel.

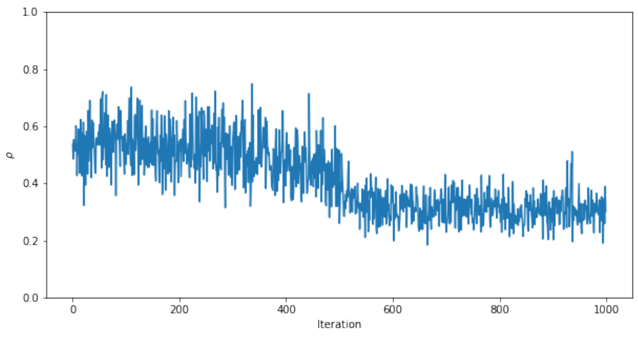


Figure 4.7.: Progression of ρ when training the RBF kernel on the three bumps data set using non-parametric Kernel Flows.

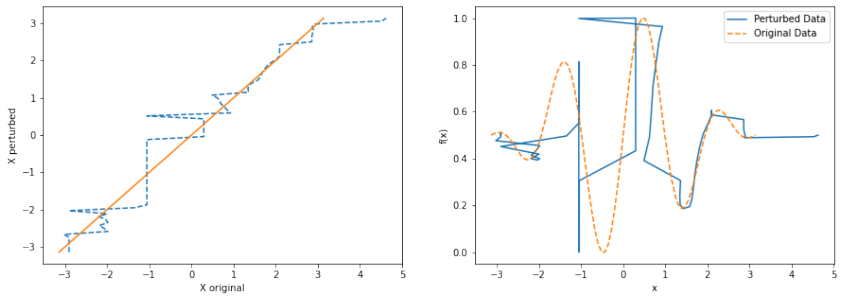


Figure 4.8.: Performing Non-Parametric Kernel Flows using RK45 to solve an initial value problem in each iteration and interpolating the acquired vector field to all input data points with an RBF kernel with $\sigma = 0.5$.

experiment with the same synthetic dataset. This time we perform the non-parametric Kernel Flows algorithm for an RBF kernel with a parameter $\sigma = 0.5$. Note that this same kernel is also used in the interpolation step of Algorithm 2. We perform 100 iterations and keep the remaining parameters the same as in the previous experiment. The results of the perturbed dataset are shown in figure 4.8.

We see that the data is perturbed in a very strange way once again because the narrow kernel used for the interpolation step can reproduce the two different vector fields that result from the batch data points and the sample data points. It is necessary to obtain a smooth vector field in order to avoid this unnatural behavior. It would be insightful to look into the evaluation of ρ itself in more mathematical detail and study how it is possible to achieve a smooth vector field from the get-go.

5. On the Formulation of ρ

We will now go back to the basics of the Kernel Flows algorithm and examine how equation (2.39) is formulated. For this, we need to revisit [26, Proposition 3.1]. As noted in section 2.3.1 this evaluation assumes that the predictions $v^\dagger$ and v^s predict their own training points, otherwise the inverse of the kernel matrix of the training points A would not cancel the kernel matrix of the test points as required by the reduced form in equation (2.39). This, however, leads to a different vector size for v^s and $v^\dagger$. Concretely, if we refer to our notation for the batch and sample size, then the predicting vector for the batch will be of size N_f, and that for the sample will be of size N_c. Since $N_f > N_c$, the prediction vector for the sample will always be smaller. Because of this, the norm of v^s will always be smaller than that of $v^\dagger$. This means, even when the prediction from the sample set is close to the batch set, the value of ρ will be high. Instead, the two sets should be predicting the same set of data points so that their predicted vectors are of equal length for a fair comparison of the norm. We will demonstrate this with the three bumps dataset here.

We try to predict the three bumps dataset with an RBF kernel with $\sigma = 3.0$ and calculate ρ based on the predictions as implemented by [5, 26] first using 32 training points and then using 64 training points. The resulting predictions are shown in figure 5.1.

One would expect a good value of rho since the predictions from both the larger training set and the smaller training set are very close. This is not the case, however, since the evaluation of rho as shown in equation (2.39) comes out to be close to 0.5. Moreover, even if the value of ρ was close to 0 since the prediction from both the large and small batches is close, this would be a problematic result because we can see that the actual prediction

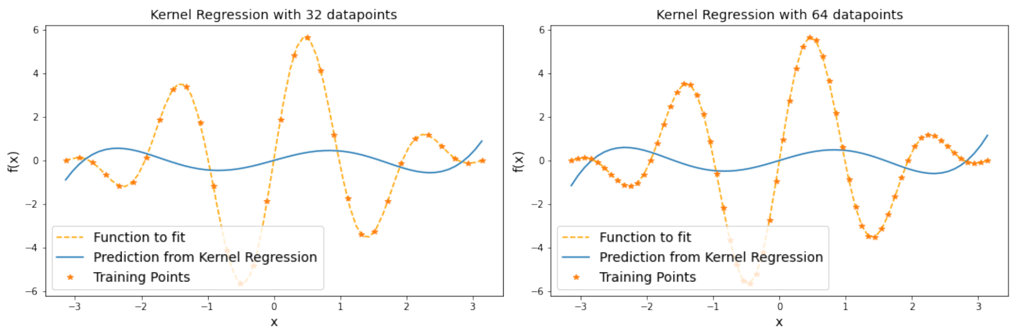
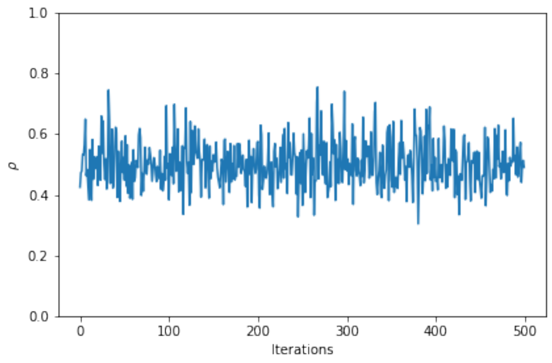
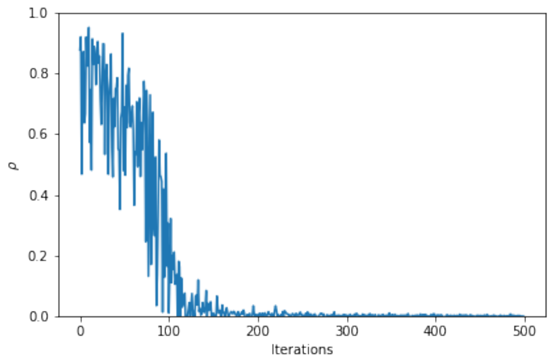


Figure 5.1.: Results from predicting the three bumps dataset first with 64 training data points and then with 32 training data points.



(a) Convergence of ρ using equation (2.39).



(b) Convergence of ρ using algorithm 4.

Figure 5.2.: Comparision of the convergence of ρ.

is inaccurate and so with a low value of ρ our optimization method would not be able to improve it even though we know the kernel requires much improvement.

To this end, we propose an amendment to the way we calculate ρ. Firstly, both the batch and the sample predictions need to predict the same number of data points so that the norm is calculated for the same vector length. Secondly, since we already know the target values for all our points, we can set the predictions from the batch to these target values instead of predicting them via kernel regression. The sample set can subsequently predict these. This way, the norm comparison will be between the target values and the target values predicted via the sample points as training data points. We summarize these changes in the algorithm 4.

Algorithm 4 Altered evaluation of ρ

```
    Create batch $\mathcal{D}_{N_f}$ from the total dataset uniformly sampled without replacement.
    Create sample $\mathcal{D}_{N_c}$ from batch uniformly sampled without replacement.
    Calculate $||v^s||^2 = (y_{N_c} A_{\mathcal{D}_s} K(\mathbf{X}_{\mathcal{D}_\mathbf{s}}, \mathbf{X}_{\mathcal{D}_\mathbf{f}}))(y_{N_c} A_{\mathcal{D}_s} K(\mathbf{X}_{\mathcal{D}_\mathbf{s}}, \mathbf{X}_{\mathcal{D}_\mathbf{f}}))^T$.
    Calculate $||v^\dagger||^2 = y_{N_f}^T A_{\mathcal{D}_f} y_{N_f}$.
    Calculate $\rho = 1 - \frac{||v^s||^2}{||v^\dagger||^2}$.
```

We now perform parametric Kernel Flows with the same three bumps dataset with an RBF kernel with $\sigma = 2.5$ using both the formulation by [5, 26] and then with our altered version of ρ. The results for the convergence of ρ can be seen in figure 5.2.

We can see how the convergence is much better when we use algorithm 4 to evaluate ρ and the value of our kernel parameter settles at a good value which is not the case when using the implementation of ρ from equation (2.39). We, however, noticed that the convergence did not happen for all starting values of σ. To this end, we evaluate ρ and its gradient with respect to the kernel parameter across various values of the kernel parame-

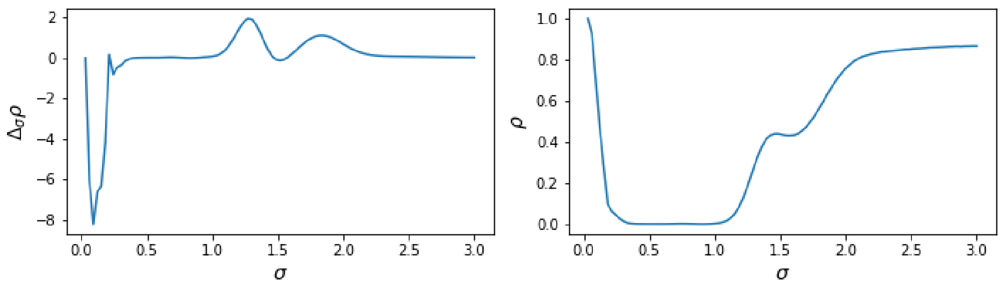


Figure 5.3.: Value of ρ and its gradient with respect to the RBF kernel parameter σ across various values of σ.

ter for a fixed batch and sample set. The results are shown in figure 5.3. We can see from the gradient that beyond values of 2 it is mostly a plateau and hence our gradient-based optimization method does not work too well, which is why we were not able to achieve convergence for values of σ that were farther away from the optimal value. This provides a lot of insights regarding why gradient-based methods were difficult to use with the algorithm and why often convergence did not take place. The formulation of ρ and its subsequent evaluation across various parameters would make for an interesting investigation as future work.

6. Conclusion

The choice of the kernel and its hyper-parameters is consequential to the performance of a kernel-based learning technique. In this work, we explore the two versions of the Kernel Flows algorithm, which attempt to modify the kernel such that its performance improves on the underlying dataset without having the user make manual optimizations.

6.1. Discussion

We first explored the idea of optimizing kernels obtained from a Neural Network induced Gaussian Process via the parametric version of Kernel Flows. This had previously been done on easy-to-evaluate kernels such as the radial basis function. However, things get complicated when the kernel evaluation becomes expensive, which is the case with NNGP kernels. To this end, we presented three different variations of the original algorithm by modifying the optimization method used to select the kernel hyper-parameters. Namely, we explored gradient-based optimization using both automatic differentiation and gradient approximation along with derivative-free optimization using Bayesian optimization. We compared the three variations across performance, time complexity, and space complexity and found that the variation that involved derivative-free optimization proved to outperform others in all three categories while using automatic differentiation was not feasible when working with expensive kernels. We also compared this more guided kernel hyper-parameter search to the random grid search performed in [9] for the ConvNet-GP kernel and saw how we were able to achieve improved performance while investing fewer computational resources. We went a step further and evaluated the performance of the kernel on a more complicated dataset than just the MNIST dataset, i.e., the CIFAR-10 dataset, and saw that the results were indeed consistent.

We then moved to the non-parametric version of the Kernel Flows algorithm, which also aims to improve the kernel but instead of tweaking the hyper-parameters, changes the underlying dataset to fit the kernel. We attempt to resolve some of the poor performance issues of the algorithm mentioned in [5] by exploring the flow on simple one-dimensional synthetic datasets. We observed poor convergence even on simple datasets and also the flow of points crossing other points. The latter is unnatural since the points move as a dynamic system based on the evaluated vector field, which should not result in point trajectories intersecting. We hypothesized that the reason for this was two separate vector fields being generated due to the sampling of the batch and sample datasets from the original dataset and subsequently interpolating only to the points that are not part of the batch dataset. We made the modification of performing the interpolation on the entire dataset and used an initial value problem solver to transform the data points in each step, which remedied the issue to a degree. We saw an improved performance on our synthetic datasets. However, the performance was still highly dependent on how the interpolation of the perturbation vector field was done, and while the flow of the system made more sense, it did not work for every kernel hyper-parameter value. Nevertheless, this highlighted how the points were transformed to fit the base kernel. Finally, we looked at the formulation of ρ itself and highlighted some concerns regarding the assumptions that go into its evaluation. We also presented an alternate way to evaluate ρ and illustrated why gradient descent fails even on simple datasets.

6.2. Future Work

We provided insights into the two versions of the Kernel Flows algorithm. For the parametric version of the algorithm, we introduced modifications that made it possible to perform the algorithm on the NNGP kernel. It would be interesting to see how the algorithm works on NNGP kernels that include other layers of a neural network apart from dense or convolutional layers, such as dropout, batch normalization, and pooling. Moreover, the Bayesian optimization variant works mostly when the number of hyper-parameters that parameterize the kernel is under a certain limit. It would be interesting to see how the performance and computational cost vary when the number of parameters increases, which is possible for deep NNGP kernels, for example by allowing each layer of the architecture to have a unique weight and bias variance.

For the non-parametric version of the algorithm, while we add robustness to the algorithm, there is still significant work left. We observed how, for poor interpolations of the vector field, results were deteriorating even when the base kernel was performing well. This indicates that the base kernel that is being optimized might not be the best choice for interpolation of the vector field and that alternates might be worth exploring. The algorithm also does not perform if the base kernel is too far off from the optimal kernel, as the kernel is essentially parameterized by all the data points, and adjusting all of them, especially for more complicated datasets, is not an inexpensive or trivial task. Moreover, the algorithm also runs into trouble when the starting RBF kernel is too narrow. It would be worth exploring alternate ways of transforming the dataset so that this anomaly can be understood and overcome. Finally, while we started our exploration in the formulation of the optimization target function ρ, there is still much to be discovered there. Importantly, the numerical aspects of calculating ρ along with the assumptions that go into simplifying the expression from its definition should be studied thoroughly in future work. This will most likely have a direct impact on both versions of the Kernel Flows algorithm particularly tackling the problem of unnatural perturbation in the latter.

Bibliography

[1] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018.

[2] Ekaba Bisong. *Google Colaboratory*, pages 59–64. Apress, Berkeley, CA, 2019.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[4] Youngmin Cho and Lawrence Saul. Kernel methods for deep learning. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[5] Matthieu Darcy. Kernel flows demystified: Applications to regression. Master's thesis, Imperial College London, 2020.

[6] Alexander G. de G. Matthews, Jiri Hron, Mark Rowland, Richard E. Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. In *International Conference on Learning Representations*, 2018.

[7] J.R. Dormand and P.J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[8] Peter I. Frazier. A tutorial on bayesian optimization, 2018.

[9] Adrià Garriga-Alonso, Carl Edward Rasmussen, and Laurence Aitchison. Deep convolutional networks as shallow gaussian processes, 2018.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks, 2016.

[11] Tim Head, Manoj Kumar, Holger Nahrstaedt, Gilles Louppe, and Iaroslav Shcherbatyi. scikit-optimize/scikit-optimize, October 2021.

[12] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *The Annals of Statistics*, 36(3):1171 – 1220, 2008.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2018.

[14] Donald R. Jones, Matthias Schonlau, and William J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13(4):455–492, Dec 1998.

[15] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60:84 – 90, 2012.

[17] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[18] Yann Lecun, Patrick Haffner, and Y. Bengio. Object recognition with gradient-based learning. 08 2000.

[19] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S. Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes, 2017.

[20] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45(1):503–528, Aug 1989.

[21] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, volume 238, page 5, 2015.

[22] Alexander G. de G. Matthews, Mark Rowland, Jiri Hron, Richard E. Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks, 2018.

[23] Martin Meinel. Efficient implementation of deep convolutional gaussian processes. Master's thesis, Technical University of Munich, 2020.

[24] Radford M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag, Berlin, Heidelberg, 1996.

[25] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A. Alemi, Jascha Sohl-Dickstein, and Samuel S. Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*, 2020.

[26] Houman Owhadi and Gene Ryan Yoo. Kernel flows: From learning kernels from data into the abyss. *Journal of Computational Physics*, 389:22–47, jul 2019.

[27] Guofei Pang, Liu Yang, and George Em Karniadakis. Neural-net-induced gaussian process regression for function approximation and PDE solution. *Journal of Computational Physics*, 384:270–288, may 2019.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[29] CE. Rasmussen and CKI. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, USA, January 2006.

[30] Herbert E. Robbins. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 2007.

[31] Bernhard Schölkopf, Ralf Herbrich, and Alex J. Smola. A generalized representer theorem. In David Helmbold and Bob Williamson, editors, *Computational Learning Theory*, pages 416–426, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[32] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[33] Tim Waegemans. Adversarial attacks on gaussian processes. Master's thesis, Technical University of Munich, 2021.

[34] Nathan Whitehead and Alex Fit-Florea. Precision and performance: Floating point and ieee 754 compliance for nvidia gpus. 2011.

A. Appendix

A.1. Computational Graphs of CNN GP with Parametric Kernel Flows

Here we present the computational graphs obtained from performing the parametric Kernel Flows algorithm on the three-layer Convolutional Neural Network Gaussian Process kernel shown in figure 3.7 with the help of the automatic differentiation engine of Pytorch.

The first graph shown in figure A.1 is the one without any modification such that the automatic differentiation engine needs to keep track of each operation resulting in a lengthy graph. This takes a lot of memory on the GPU limiting the size of NNGPs and batch sizes that are feasible to train with.

The second computational graph shown in figure A.2 shows the results after combining the operations involved in the calculation of the ReLU activation covariance kernel into a single function by providing a closed-form derivative. We can see how this significantly reduces the size of the computational graph and subsequently memory consumption on the GPU as fewer operations need to be tracked.

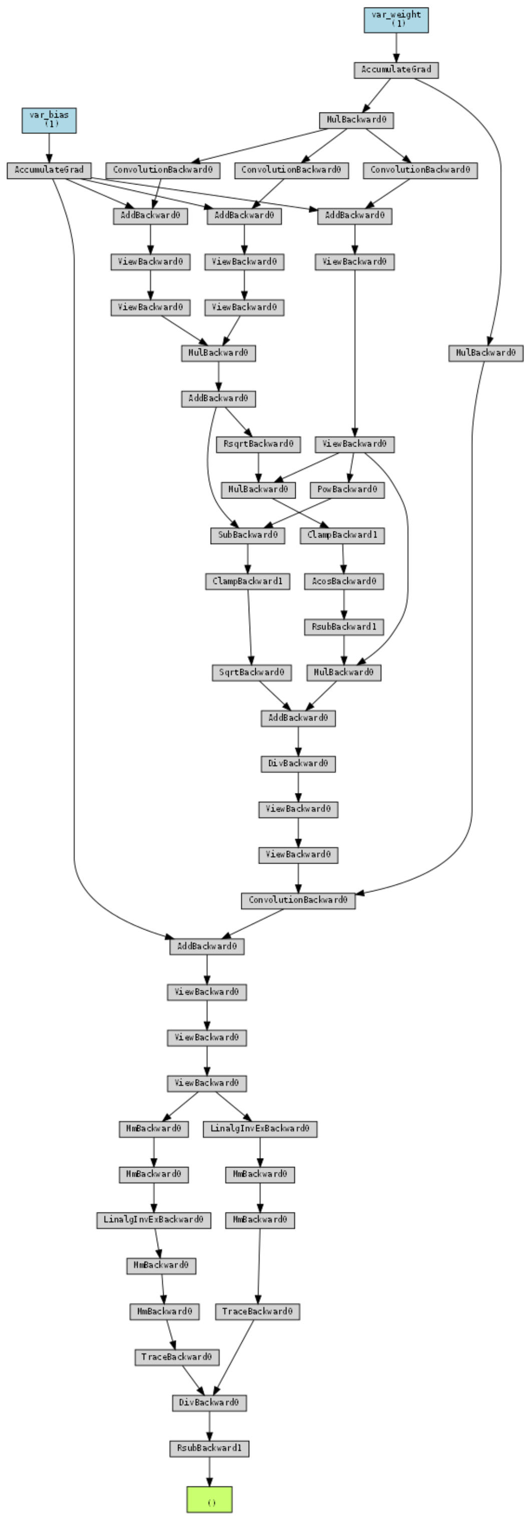


Figure A.1.: Backpropogation Graph with Pytorch Autograd.

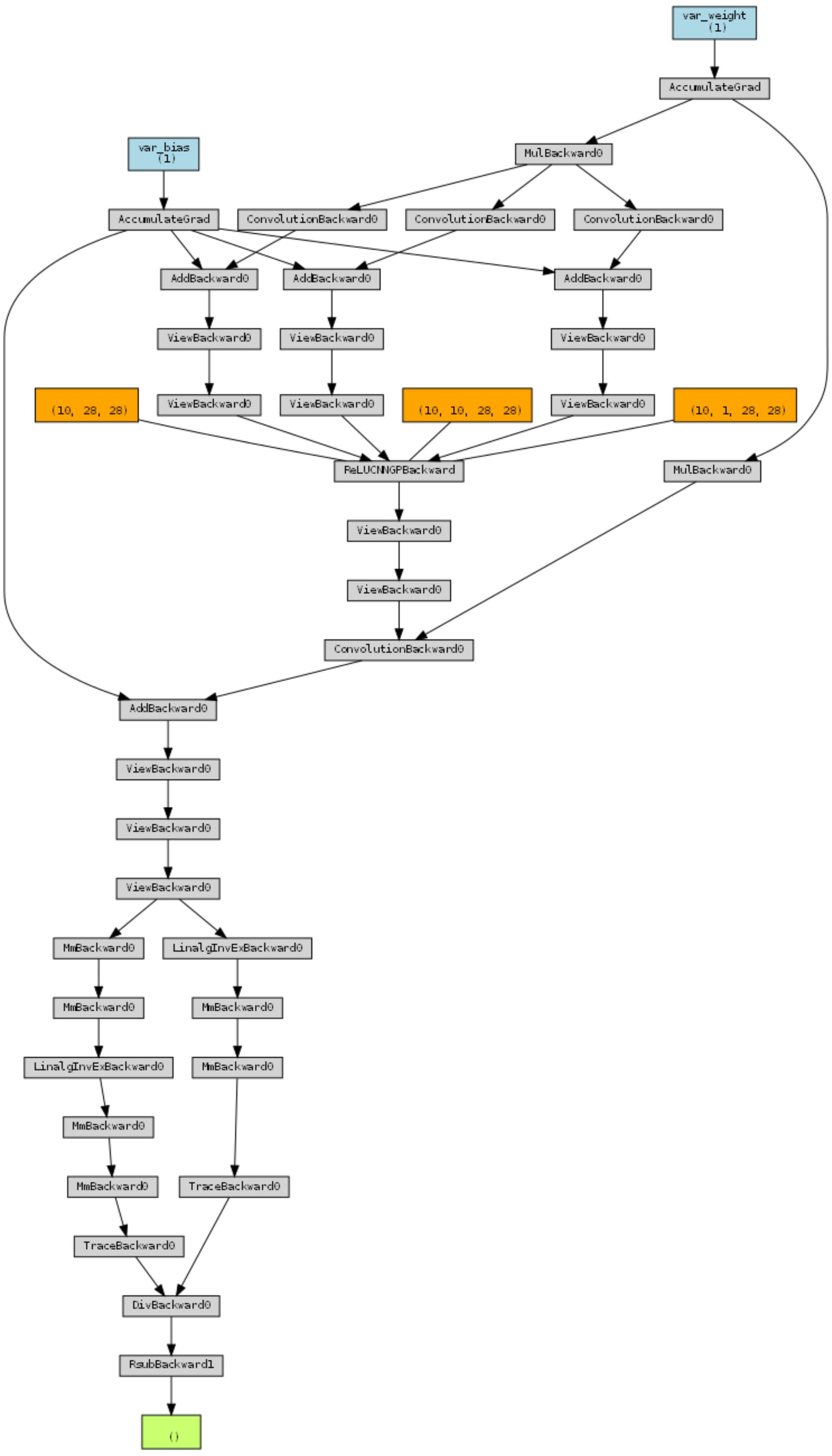


Figure A.2.: Backpropogation Graph with Custom ReLU Covariance implementation along with its closed-form derivative.

A.2. Hyper-parameter Settings for Various Experiment

This section lists the hyper-parameters set for the various experiments to evaluate the different variations of the Parametric Kernel Flows algorithm. The hyperparameters used in the experiments with the MNIST dataset are listed in table A.1. The ones used in the experiments with the CIFAR-10 datasets are listed in table A.2.

Table A.1.: Hyper-parameter settings for the variations of the Parametric Kernel Flows experiments on the MNIST dataset.

Optimization Technique	NN Architecture	Regularization Constant	Batch Size	Sample Size	Learning Rate	Iteration Count
Auto Differentiation	Three-Layer CNN-GP	0.0001	600	300	0.1	1500
Finite Difference	ConvNet-GP	0.0001	600	300	0.1	500
Bayesian Optimization	ConvNet-GP	0.0001	1200	600	-	50

Table A.2.: Hyper-parameter settings for the variations of the Parametric Kernel Flows experiments on the CIFAR-10 dataset.

Optimization Technique	NN Architecture	Regularization Constant	Batch Size	Sample Size	Learning Rate	Iteration Count
Auto Differentiation	-	-	-	-	-	-
Finite Difference	ConvNet-GP	0.0001	600	300	0.1	250
Bayesian Optimization	ConvNet-GP	0.0001	1200	600	-	30