Kernel Flows Demystified: Applications to Regression

Matthieu Darcy

September 2020

Supervisor: Dr. Boumediene Hamzi

CID: 01805975
Dept. of Mathematics
Imperial College London

Acknowledgements

I would like to thank my supervisor Dr. Boumediene Hamzi for helping me with this project. His continued aid was invaluable in helping me conduct the research necessary for this report and in writing it. I would like to thank Houman Owhadi and Gene Ryan Yoo for helping me understand their method and for sharing some of their programming code, which helped me write my own. I would also like to thank Dr. Kevin Webster for teaching me what I know of machine learning in his course at Imperial College London.

Finally, I would like to thank my friends and family for their moral support over the past few months.

Declaration

The work contained in this thesis is my own work unless otherwise stated. Matthieu Darcy (September 2020).

Abstract

In this report we present and investigate the Kernel Flows algorithm, both Parametric and Non-Parametric version. We explore their behavior and make suggestions to maximize their performance. We also propose a modified version of the RBF network which can be trained using the Kernel Flows Parametric algorithm. Finally we apply KF Non-Parametric, KF Parametric and the RBF network to standard datasets. We compare their performance and the impact of various hyper-parameter choices on the effectiveness of the algorithms.

Contents

Chapter 1

Introduction: Linear Regression and the Kernel trick

1.1 Introduction

Regression problems, the problem of predicting a continuous dependent variable from data, form an important class of problems in the field of data science and machine learning. One of the primary tools to solve these problems is linear regression. A popular extension of linear regression is Kernel Ridge regression which, thanks to the kernel trick, allows to apply linear regression to a transformation of the data without actually computing the said transformation. This technique however depends on the specification of a Kernel and its parameters. The choice of Kernel and associated parameters strongly impacts the performance of Kernel regression and is often difficult without experimentation. To tune the parameters of a Kernel, data scientists are often forced to perform a computationally costly grid search wherein many different values of the parameter are tested. This is often very impractical if the Kernel has multiple parameters since the search space will scale exponentially with the number of parameters.

In this report we will explore an algorithm, called Kernel Flows, recently proposed by Houman Owhadi and Gene Ryan Yoo in [12] which proposes to

optimize the Kernel and its parameters using a numerical algorithm. We will first expose in Chapter 1 the basic theory of Linear Regression and Kernel Regression, in particular the notion of Reproducing Kernel Hilbert Space upon which depends the algorithm. We will then in chapters 2 and 3 present the two versions of the algorithm (Parametric and Non-Parametric) and explore their basic properties. In 4 we will present the Radial Basis Function (RBF) network and show how Kernel-Flows Parametric can train an extended version of the basic RBF network. Finally, in Chapter 5 we will apply Kernel Flows to three standard datasets.

1.2 Linear Regression and the Kernel trick

We first remind the reader of some standard results in the theory of linear regression. Given a data set $(x_i, y_i)_{i=1}^N, x_i \in \mathbb{R}^d, y_i \in \mathbb{R}$, we assume that the data is generated according to

$$y_i = f(x_i, \theta) + \varepsilon_i \tag{1.1}$$

where θ is a set of parameters. The ε_i are independent, identically distributed Gaussian noise with mean 0:

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2). \tag{1.2}$$

The basic linear regression model ([2, p. 138]) can be written as

$$f(x_i, \theta) = \boldsymbol{\phi}(x_i)^T \theta.$$

Where $\boldsymbol{\phi}(x_i)$ is the vector of basis function $\boldsymbol{\phi}(x_i) = (\phi_1(x_i), \phi_2(x_i), ..., \phi_n(x_i))$ and θ is weight vector. More generally:

$$f(X, \theta) = \Phi\theta$$

where Φ is the feature matrix with entries $\Phi_i = \boldsymbol{\phi}(x_i)$. Given the above problem, we define the regularized mean squared error loss function to be minimized as follows ([2, p. 141]):

$$L(\theta) = \sum_{i=1}^{N} (y_i - f(x_i, \theta))^2 + \lambda ||\theta||_2^2 \tag{1.3}$$

The basic results of Kernel Ridge regression states that we may rewrite the solution to the above problem as

$$f(x) = \boldsymbol{K}(x, X)(\boldsymbol{K}(X, X) + \lambda \boldsymbol{I}_N)^{-1} Y \tag{1.4}$$

where $\boldsymbol{K}(x, X) = (K(x, x_1), ..., K(x, x_N)), Y = (y_1, ..., y_N)^T$ and $\boldsymbol{K}(X, X)$ is the $N \times N$ Gram matrix with i, j entries $K(x_i, x_j)$ [2, p. 293]. The function K is symmetric, positive definite kernel (see section 1.3). The quantity $K(x_i, x_j)$ corresponds to the inner product in feature space: $\langle \phi(x_i), \phi(x_j) \rangle$.

The term $\lambda \boldsymbol{I}_N$ is a regularization term derived from the regularized MSE loss function (1.3) and added to avoid degenerate matrices when inverting. This regularization parameter gives the name of "Ridge" to Kernel regression. Note that this is equivalent to redefining the kernel used as

$$K'(x, x') = K(x, x') + \delta(x - x')$$

where δ is 1 at 0 and 0 everywhere else.

For convenience, we will use the letter K to refer to both the kernel function, and to the vector or matrix with elements determined by the kernel functions. $K(x, x')$ will refer to a particular value, $K(x, X)$ to the vector and $K(X, X)$ to the matrix.

1.3 Reproducing Kernel Hilbert Space

In this section we present some of the basic results of theory of Reproducing Kernel Hilbert Space (RKHS).

We first define the positive definite kernel K mentioned above.

Definition 1.3.1. *Positive Definite Kernel [15, p. 417]. A symmetric function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive definite kernel if for any $x_1, x_2, ...x_N \in \mathcal{X}$ and any $c_1, c_2, ...c_N \in \mathbb{R}$:*

$$\sum_i^N \sum_j^N c_i, c_j K(x_i, x_j) \geq 0.$$

Equivalently, consider the matrix defined by $\boldsymbol{K}_{i,j} = K(x_i, x_j)$. Then for any $x_1, x_2, ...x_n \in \mathcal{X}$ the matrix $\boldsymbol{K}$ is positive definite: for all $\boldsymbol{c} \in \mathbb{R}^n$,

$$\boldsymbol{c}^T \boldsymbol{K} \boldsymbol{c} \geq 0.$$

We now define what a RKHS is.

Definition 1.3.2. *Reproducing Kernel Hilbert Space [8, p. 4]. Let H be Hilbert space of real functions and consider the linear functional*

$$L_x : f \mapsto f(x).$$

H is a RKHS if for any x, L_x is continuous over the functions f in H. In general, the Riesz representation theorem states that there is a K_x (which is an element of H) such that

$$f(x) = L_x(f) = \langle f, K_x \rangle_H.$$

This is the reproducing property.

Given a kernel function K, let $K_x = K(x, .)$ and define H_0 as the linear span of $\{K_x : x \in \mathcal{X}\}$. Then the Moore–Aronszajn tells us that a kernel defines a RKHS that is the closure of H_0.

Theorem 1.3.3. *Moore–Aronszajn, [8, p. 10]. Consider a symmetric, positive definite kernel K. Then there is a unique Hilbert space H of functions such that*

1. $H = \overline{span\{K_x : x \in \mathcal{X}\}}$
2. $K(x, y) = \langle K_x, K_y \rangle$

K is a reproducing kernel for the Hilbert Space H.

The space H above is the completion of H_0 with respect to the norm induced by the inner product

$$\langle \sum_i^n a_i K_{x_i}, \sum_j^m a_j K_{x_j} \rangle = \sum_i^n \sum_j^m a_i a_j K(x_i, x_j). \tag{1.5}$$

.

1.3.1 RKHS and Kernel regression

We now present the representer theorem, which plays an important role in theoretically justifying the use of Kernel regression.

Theorem 1.3.4. *Representer theorem, [15, p. 419]. Consider a symmetric, positive definite kernel and the associated RKHS H and*

1. *a training set $(X, Y), X \in \mathcal{X}^n, Y \in \mathbb{R}^n$*

2. *a strictly increasing function* $g : [0, \infty) \to \mathbb{R}$

3. *an arbitrary error function* $E(\mathcal{X} \times \mathbb{R}^2)^n \to \mathbb{R} \cup \{\infty\}$

These three elements define a loss function$f \mapsto E(X, Y, f(X)) + g(||f||)$ [1]. *Then any minimizer of the loss function*

$$f^* = arg\, min_{f_k \in H}\{E(X, Y, f(X)) + g(||f||)\}$$

admits the representation

$$f^* = \sum_i^n \alpha_i K_{x_i}. \tag{1.6}$$

Setting the coefficient α_i in (1.6) to c_i where $\boldsymbol{c} = (\boldsymbol{K}(X, X))^{-1}Y$ yields an function which interpolates the data point exactly, meaning that $f^*(x) = y$ for $x \in X, y \in Y$. This function is the minimizer of the loss function

$$\frac{1}{n}\sum_i^n (f(x_i) - y_i)^2$$

for $f \in H$ (the unregularized MSE). In the case where $\boldsymbol{c} = (\boldsymbol{K}(X, X) + \lambda \boldsymbol{I}_N)^{-1}Y$, the function f^* no longer exactly interpolates the points, but instead minimizes the loss function

$$\frac{1}{n}\sum_i^n (f(x_i) - y_i)^2 + \lambda ||f||_H^2.$$

Note that in this case, equation (1.6) is exactly equation (1.4).

Finally, we note that the interpolator function, by the definition of the inner product in H, equation (1.5), has norm:

$$||f^*||^2 = Y^T \boldsymbol{K}(X, X))^{-1}Y.$$

1.4 Universal Kernels

Kernel regression requires the prior specification of a kernel and a set of parameters for the kernel. In this section, we address the general form of the kernel through the notion of universal kernels, as presented in [10]. Intuitively, the

[1] In our case, the loss function is of the form (1.3)

RKHS of a universal Kernel can approximate any continuous function on any compact set.

Definition 1.4.1. *Consider a set X, an arbitrary compact subset Z of X and the set $C(Z)$ of continuous function on Z, equipped with the supremum norm. Let $K(Z) = \overline{span}\{K_y : y \in Z\}$ where $K_y(x) = K(y, x)$. We say that the kernel K is **universal** if for any compact set Z, for any $\varepsilon > 0$ and $f \in C(Z)$, there is a $g \in K(Z)$ such that $||f - g||_{\infty,K} < \varepsilon$.*

Two universal kernels, as identified in [10] are the Gaussian kernel and the Rational Quadratic kernel.

Definition 1.4.2. *The Gaussian Kernel.*

$$K(x, y) = \exp\Big(- \frac{||x - y||^2}{2\sigma^2}\Big) \tag{1.7}$$

where σ is the parameter to be optimized. Equivalently:

$$K(x, y) = \exp\Big(- \gamma||x - y||^2\Big), \quad \gamma > 0. \tag{1.8}$$

By properties of kernels (see [2], page 296), we may also define the n-linear gaussian kernel as

$$K(x, y) = \sum_{i=1}^{n} \beta_i^2 \exp\Big(- \frac{||x - y||^2}{2\sigma_i^2}\Big).$$

Note that the β_i are squared to only allow positive values.

Definition 1.4.3. *The Rational Quadratic Kernel.*

$$K(x, y) = (\beta^2 + \gamma||x - y||)^{-\alpha}, \quad \alpha, \beta, \gamma > 0. \tag{1.9}$$

Two particular cases are $\alpha = 1$, the inverse quadric kernel, and $\alpha = \frac{1}{2}$, the inverse multiquadric kernel.

Because of the universal property, these kernels are natural choices. Note also that these kernels are Radial Basis Functions because their output only depends on the distance between x and y. Hence they are both candidates for basis functions in a RBF network (see Chapter 4).

1.5 Kriging

An alternative perspective on the kernel regression problem is given by kriging, which is a probabilistic interpration of the regression problem. In this section we present its basic elements. .

1.5.1 Gaussian processes

The linear regression problem can be solved through a Gaussian process approach. We present here the main elements explained in [2, p. 303 onwards].

A Gaussian process is a stochastic process such that any finite collection of its random variables has a Gaussian distribution. We again consider the problem given by (1.1) and the associated linear model

$$f(X) = \Phi\theta. \tag{1.10}$$

We also place a prior on the weights

$$\theta \sim \mathcal{N}(\theta|0, \alpha^2 I_M). \tag{1.11}$$

Since $f(X)$ is a linear combination of Gaussian variables, $f(X)$ itself has a multivariate Gaussian distribution and hence defines a Gaussian process. Its distribution is entirely defined by its mean and covariance matrix, which, using (1.11), is

$$\begin{aligned} \mathbb{E}[f(X)] &= \Phi\mathbb{E}[\theta] \\ cov[f(X)] &= \mathbb{E}[f(X)f(X)^T] = \Phi E[\theta\theta^T]\Phi^T = \alpha^2\Phi\Phi^T = \Sigma \end{aligned}$$

Hence the covariance matrix matrix Σ has elements $\Sigma_{i,j} = \alpha^2\phi(x_i)\phi(x_j)$. As before, we may rewrite the feature maps as kernel functions: $\Sigma_{i,j} = K(x_i, x_j)$ where K is the chosen kernel function.

1.5.2 Gaussian Processes and Regression

As previously, we make the assumption (1.2) that the noise is normally distributed with mean 0. Then

$$p(Y) = \mathcal{N}(Y|\mathbf{0}, \boldsymbol{C}) \tag{1.12}$$

where $\boldsymbol{C}_{i,j} = K(x_i, x_j) + \sigma^2\delta(x_i - x_j)$. In other words, $\boldsymbol{C} = K(X, X) + \sigma^2\boldsymbol{I}_N$ The probability distribution of a new point, given all previous points (i.e. given the training data), $p(y_{N+1}|Y)$ is also Gaussian. Its mean and variance are given by

$$\mu(y_{N+1}) = K(x_{N+1}, X)^T\boldsymbol{C}^{-1}Y$$
$$\sigma(y_{N+1}) = (K(x_{N+1}, x_{N+1}) + \sigma^2) - K(x_{N+1}, X)^T\boldsymbol{C}^{-1}K(x_{N+1}, X).$$

The reader is invited to read [2, p. 306 onwards] for a more detailed derivation of the equations above. We note that the mean of probability distribution of y_{N+1} is the same as the predicted value of our kernel regressor.

Chapter 2

Kernel Flows: parametric version

In this chapter, we present the main ideas behind Kernel Flows, as were presented in [12]. We then explore some of its properties, notably the impact of the sample size and regularization techniques.

2.1 An introduction to Kernel Flows

Powerful as it may be, kernel regression requires the prior specification of both a Kernel function and a set of parameters for the Kernel. Kernel flows is an algorithm that allows for the numerical computation of the best kernel and comes in two forms: the parametric and non-parametric forms. Here we focus on the parametric version. The intuition behind Kernel Flows is that a good kernel should suffer a minimal loss when half of the data set is removed. Given a specified kernel and set of parameters θ for the kernel, this leads to the following algorithm:

Kernel Flows algorithm: parametric version

1. Select a batch (x_{i_b}, y_{i_b}) from the whole data set (x_i, y_i), the $i_b(1), ..., i_b(N_b)$ being a subset (which can be the whole set) of the indices $\{1, 2..., N\}$. The batch sub-vectors are denoted X_b and Y_b respectively.

2. Select a sample (x_{i_s}, y_{i_s}) from the batch (x_{i_b}, y_{i_b}), the $i_s(1), ..., i_s(N_s)$

being a subset of $\{1, 2, ..., N_b\}$ containing half of the indices[1]. The sample sub-vectors are denoted X_s and Y_s respectively.

3. Compute $\nabla_\theta \mathcal{L}(X_b, Y_b, X_s, Y_s, \theta)$ where θ is the set of parameters for our kernel and $\mathcal{L}$ is a loss function.

4. Adjust the parameters $\theta \leftarrow \theta - \delta \nabla_\theta \mathcal{L}(X_b, Y_b, X_s, Y_s, \theta)$.

There are two versions of the loss function $\mathcal{L}$:

1. The RKHS norm loss ρ:

$$\rho(X_b, Y_b, X_s, Y_s, \theta) = 1 - \frac{Y_s^T K(X_s, X_s)^{-1} Y_s}{Y_b^T K(X_b, X_b)^{-1} Y_b}. \tag{2.1}$$

   Let $\hat{y} = K(X_b, X_b)^{-1} Y_b$, $\hat{z} = \Pi^T K(X_s, X_s)^{-1} \Pi Y_b$, where $\Pi_{k,j} = \delta_{i_s(k),j}$ is the selection matrix. Then [12], equation (4.2) provides an explicit expression for the Fréchet derivative of ρ

$$\partial_{\theta_i} \rho(\theta) = -\frac{(1 - \rho(\theta))\hat{y}^T (\partial_{\theta_i} K(X_b, X_b))\hat{y} - \hat{z}^T (\partial_{\theta_i} K(X_b, X_b))\hat{z}}{Y_b^T K(X_b, X_b)^{-1} Y_b} \tag{2.2}$$

2. The l_2 norm between the batch and the prediction generated by the sample:

$$||Y_b - \boldsymbol{K}(X_b, X_s)\boldsymbol{K}(X_s, X_s)^{-1} Y_s||_2^2. \tag{2.3}$$

The RKHS loss ρ compares the performance of the interpolating function with half the points and the interpolating function with all the points. Letting v_b be the interpolation function which sees the full batch and v_s the interpolation function which sees the sample, then $\rho = \frac{||v_b - v_s||^2}{||v_b||^2}$ with $||.||$ the intrinsic RKHS norm generated by the kernel K. In essence, minimizing ρ entails choosing parameters such that v_b and v_s are close to each other in the RKHS space. See [12], section 3, specifically equations (3.2) and (3.4) for a derivation of the form presented here. In this report we will be primarily interested by the loss ρ.

2.1.1 Gradient descent optimizers

In the algorithm presented above, the parameter update is

$$\theta \leftarrow \theta - \delta \nabla_\theta \mathcal{L}(X_b, Y_b, X_s, Y_s, \theta)$$

[1] More generally, $p \times N$ indices, with $0 < p < 1$.

with δ being the learning rate parameter, typically a small value less than 1. We will refer to this update rule as Stochastic Gradient Descent (SGD).

We will also consider the Nesterov momentum update, [1]:

$$\begin{aligned} z_0 &= 0 \\ z_{k+1} &= \beta z_k + \nabla_\theta \mathcal{L}(\theta_k - \delta\beta z_k) \\ \theta_{k+1} &= \theta_k - \delta z_{k+1}. \end{aligned} \tag{2.4}$$

Nesterov momentum has an additional hyperparameter β which is typically set to 0.9. Nesterov momentum has the feature of "remembering" previous computed gradients, stored in the z_k term. These previous computed gradients, the momentum, push the parameter in the direction of *historical* gradient descent direction, not just the present gradient descent direction. This present several advantages:

- Faster learning of the parameters.
- Momentum can help avoid local minima: by remembering past gradients, the algorithm overshoots the local minima region.
- Momentum can help avoid some undesirable effects of the randomness of the batch/sample selection: the effect of "abnormal" gradients is diminished because past gradients are remembered.

However, these advantages can also be drawbacks as the randomness of the batch/sample is sometimes beneficial to learning the best value (see section 2.2.1). Therefore in practice which optimizer is best varies, but experiments show that SGD consistently performs well, even if Nesterov momentum does generally lead to increased performance.

2.1.2 Implementation

We implement the kernel flows algorithm using the Python programming language (version 3.5+). The algorithm is implemented in two different ways. The first uses the automatic differentiation package autograd [6]. The second directly computes the Fréchet derivative that was derived in [12]. The code is available at [5] and uses the Numpy package ([11], [17]). This code is in part possible thanks to Gene Ryan Yoo who shared his own code with me.

2.1.3 Computational cost

In this section, we compute the computational cost of every iteration of KF. We assume that the computational complexity of the matrix multiplication of matrices of dimension $n \times m$ and $m \times p$ is $\mathcal{O}(nmp)$. We also assume that the computational complexity of inverting a $n \times n$ matrix is $\mathcal{O}(n^3)$. We also assume that any element-wise operation on an $n \times m$ matrix (including the Hadamard product) is $\mathcal{O}(nm)$.

The complexity of computing the kernel matrix is $\mathcal{O}(N_b^3)$ (respectively $\mathcal{O}(N_s^3)$). Computing the derivative matrix $\partial_\theta K(X_b, X_b)$ has the same complexity.

Given that the kernel matrices are available, the complexity of computing ρ is $\mathcal{O}(N_b^3 + N_b^2 + N_b + N_s^3 + N_s^2 + N_s) = \mathcal{O}(N_b^3)$.

Given that the kernel matrices are available, computing $\hat{y}$ requires complexity $\mathcal{O}(N_b^2)$. The complexity of $\hat{z}$ is $\mathcal{O}(N_b N_s + N_s^2 + N_s) = \mathcal{O}(N_b^2)$

Finally, computing $\hat{y}^T(\partial_{\theta_i} K(X_b, X_b))\hat{y}$ has complexity $\mathcal{O}(N_b^2 + N_b) = \mathcal{O}(N_b^2)$. Computing $\hat{z}^T(\partial_{\theta_i} K(X_b, X_b))\hat{z}$ has the same complexity.

Therefore, computing the gradient of ρ (which includes ρ itself), has an overall complexity of $\mathcal{O}(N_b^3)$[2].

However, the above derivation does not take into account the complexity of sampling uniformly without replacement from the dataset, which will depend on the algorithm used.

2.2 Investigating of the Kernel Flows algorithm and its properties

2.2.1 Basic examples: synthetic data sets

We now consider a synthetic data set, generated according to the Gaussian kernel, with parameter $\sigma = 2.0$. We test the convergence of Kernel Flows with both SGD and Nesterov Momentum.

[2]However, this is reliant on our assumptions, which are not always true. Specifically, very large values within matrices can change the complexity of inversion and matrix multiplication.

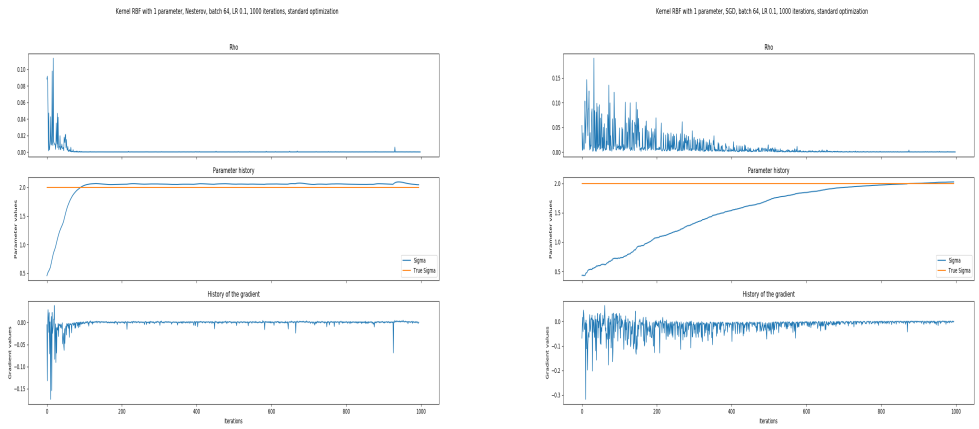


Figure 2.1: Convergence of KF for a synthetic data set with $\sigma = 2.0$, over 1000 iterations with update parameters $\delta = 0.1, \beta = 0.9$.

In both cases, KF converges to the true value, although Nesterov momentum does so much faster. We initiliazed both versions at the same value, uniformly chosen between 0 and 1.

Initialization and convexity

We now illustrate how the initialization of the algorithm affects the convergence of KF to the correct value. Again we consider a synthetic data set, generated by the Gaussian kernel with $\sigma = 10.0$ and plot the ρ function with the batch being the whole data set and a random sample of half the data set and note the local minima at values less than 4.

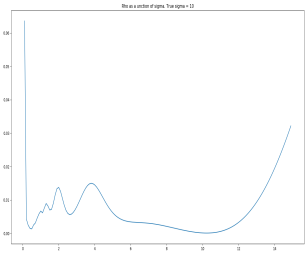


Figure 2.2: Rho as a function of σ. The batch is the whole data set.

Hence Kernel flows cannot attain the true minima at 10, if initialised below 4.

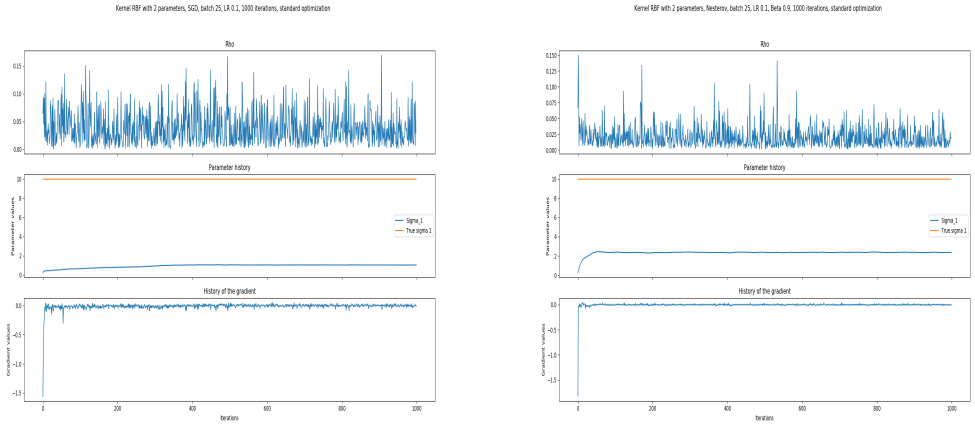


Figure 2.3: Non convergence of KF for $\sigma = 10$ when KF is initialized between 0 and 1.

We find similar results with other values of σ and accounting for the randomness of the selection of both the mini-batch and the sample (as can be seen in the following plot).

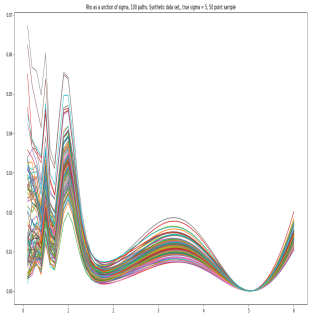


Figure 2.4: Rho function for 100 different batch/sample combinations. We note the greater irregularity when σ is close to 0.

These results suggest that the parameter initialisation is important to consider, and should be done depending on the kernel and the dataset. The reason

for the observed phenomena is the non-convexity of the ρ function: when the parameter is initialized near a local minima, the algorithm, like all gradient based learning algorithms, can find itself trapped in this area. There are however several measures that can help against this.

The Nesterov Momentum update rule, equation (2.4), can help avoid this problem thanks to the accumulated momentum pushing the parameter outside the are of local minima, as illustrated by figure 2.5

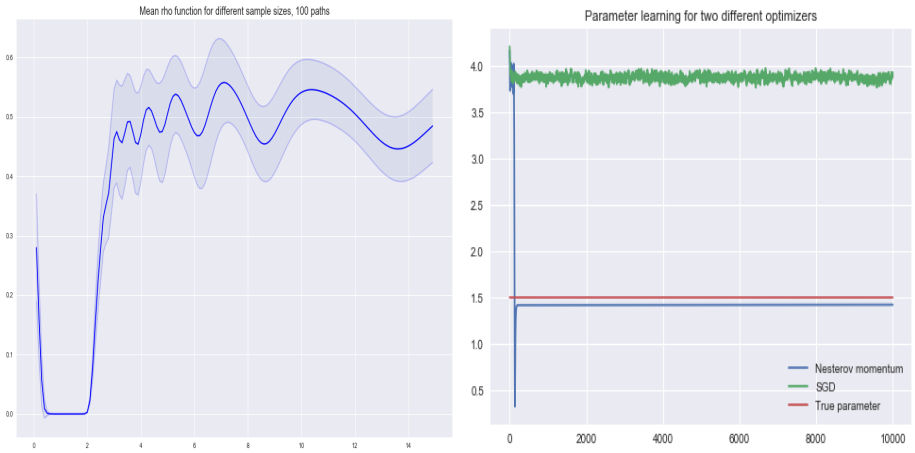


Figure 2.5: Convergence in the case where $\sigma = 1.5$, the Kernel is initialized at $\sigma = 4.2$. The ρ function (left) is highly non-convex. While SGD gets trapped in the nearby local minima, Nesterov momentum converges to the correct value, thanks to the accumulated momentum (right).

Similar results can be achieved by reducing the size of the batch size N_b: smaller batch sizes lead to increase variance in the ρ function and hence the chance of a batch/sample combination with different minima from the average. Figure 2.6 illustrates this phenomena.

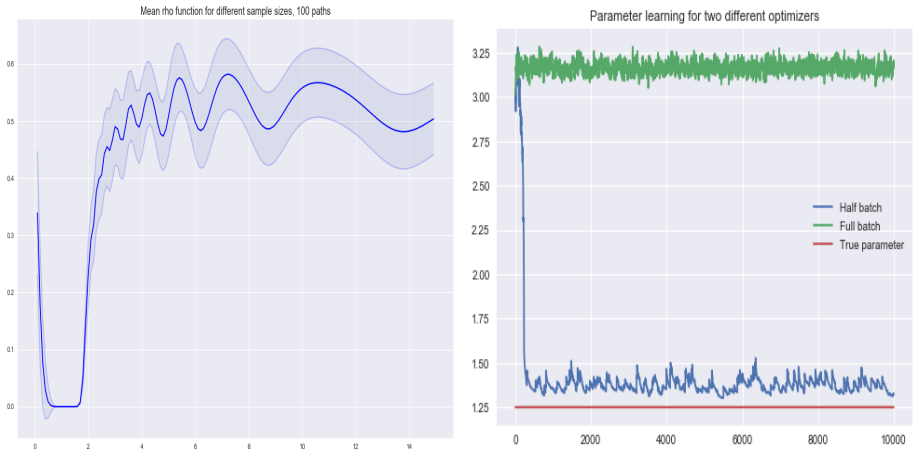


Figure 2.6: Convergence in the case where $\sigma = 1.25$, the Kernel is initialized at $\sigma = 4.2$. The ρ function (left) is highly non-convex. Using SGD with a full batch leads to being stuck in the local minima, whereas using a batch size of $N_b = \frac{1}{2}N$ allows to escape the local minima (right).

Finally, using a broad range of initialization values can help finding the best minima. These results are not systematically true, but practice shows that adopting these measures does increase performance.

Rational Quadratic kernel

We find similar results for the Rational Quadratic kernel when the parameter to be optimized is β.

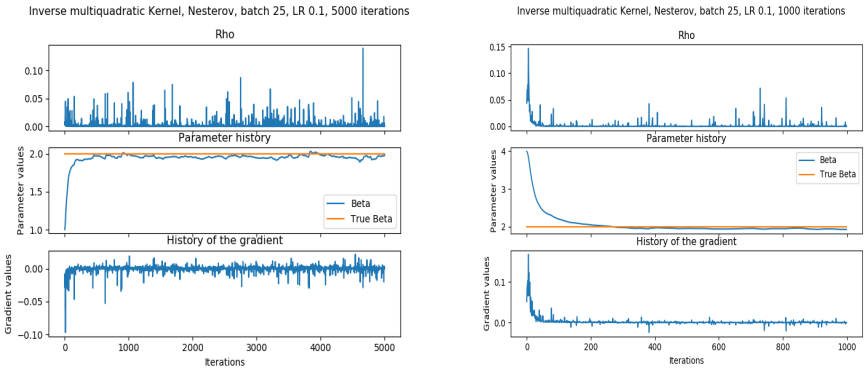


Figure 2.7: Convergence of β for the Rational Quadratic kernel. The true parameter is 2.0, the initializations are 1.0 and 4.0. KF is done over 5000 and 1000 iterations respectively.

We find similar results when the parameter to be optimised is α, although convergence is less stable.

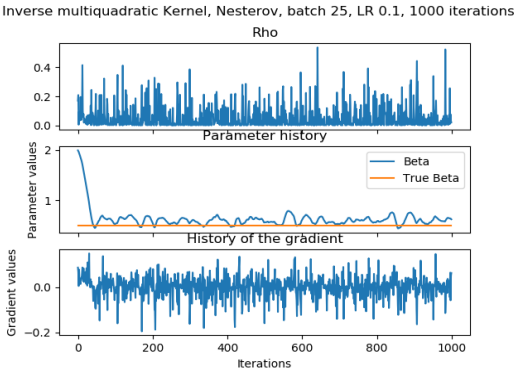


Figure 2.8: Convergence of α for the Rational Quadratic kernel. The true parameter is 0.5, the initializationis 2.0.

2.2.2 Convergence in the case of multiple parameters

Above we have shown that convergence is generally accurate when it comes to finding a single parameter. However, KF does not necessarily converge to the correct parameters when several parameters are involved.

As a first example, consider the linear combination of Gaussian Kernels

$$K(x,y) = \exp(-\frac{||x-y||^2}{\sigma_1^2}) + \exp(-\frac{||x-y||^2}{\sigma_2^2}).$$

In this case, Kernel Flows does not converge to the true parameters, but instead seems to find some equilibrium between the two values.

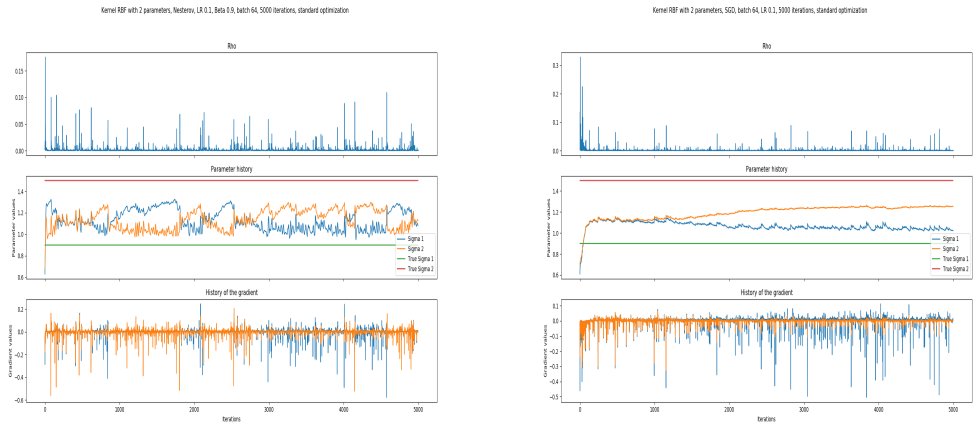


Figure 2.9: Non convergence to the real parameters in the case of the 2-linear combinations of Gaussian Kernels. The true parameters are 1.5 and 0.9 and the parameters were initialized uniformly between 0 and 1.

We observe similar results with the Rational Quadratic kernel.

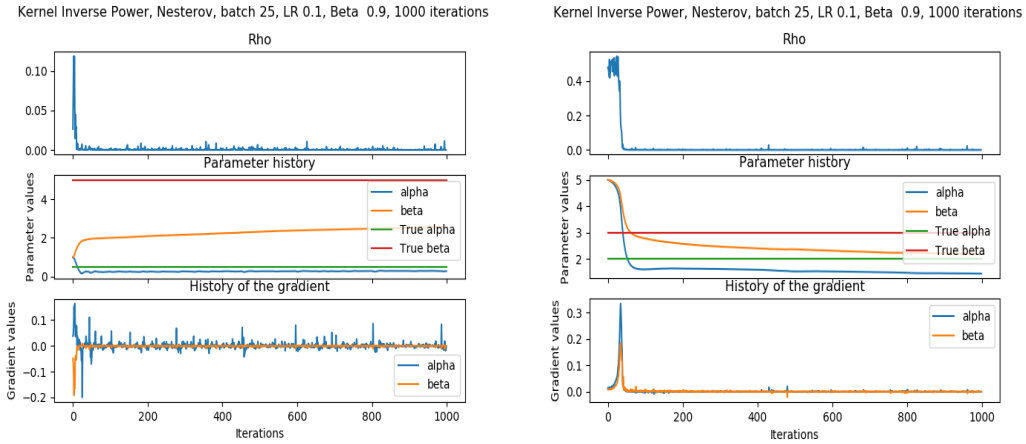


Figure 2.10: Non convergence to the real parameters in the case of the Rational Quadratic kernel. In the first case, the true values are $\alpha = 0.5, \beta = 5.0$, in the second case $\alpha = 2.0, \beta = 3.0$.

This suggests that convergence to the true parameters is initialisation dependent and value dependent. However, since KF seeks to minimize the RKHS loss (and not to find the exact parameter value), these results do not entail that KF should not be used with several parameters. The ultimate goal is extrapolation and regression to new data, which does not require learning exact parameters.

2.2.3 Kernel Flows and regularization

In equation (1.4), the regularization term λ serves two purposes. From a computational point of view it allows for the inversion of singular or near singular matrices. From a mathematical point of view, it adds a penalty term to the least squares error function. Hence the choice of λ is not only a a necessary computational tool, but an important hyperparameter which determines the performance of the regressor.

How does KF behave with different values of λ? We illustrate the answer through a toy data set, one generated by a sin function with added Gaussian noise: $y = \sin(x) + \varepsilon, x \in [0, 3\pi], \varepsilon \sim \mathcal{N}(0, 3)$.

Figure 2.11: Data set generated by a sin function, with added Gaussian noise.

The data set is sufficiently noisy that the regularization parameter should be fined tuned to prevent overfitting. We now use kernel flows to fit the RBF kernel to the training data, σ^2 is initialized to be the variance of the L^2 norms of the data points and is optimized over 10000 iterations using the Nesterov optimizer. We do so three times, with high, medium and low regularization: $\lambda = 10^{-2}, 10^{-5}, 10^{-10}$. Figure 3.1 illustrates how the different regularizations yield different values of σ^2

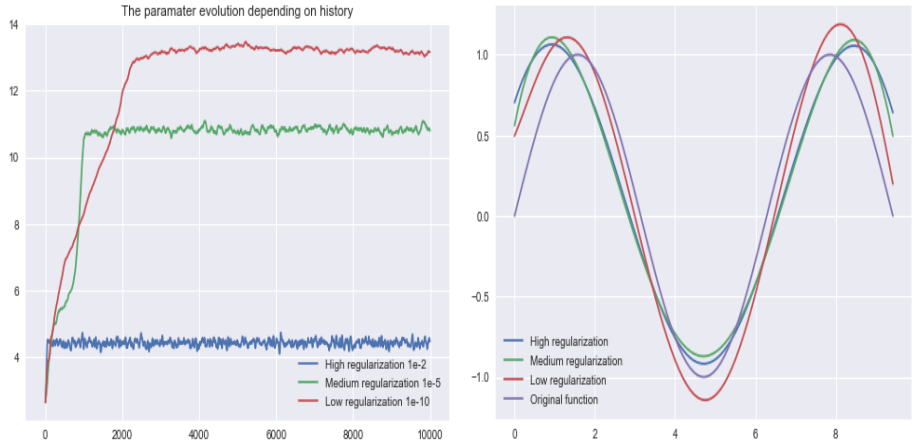


Figure 2.12: Evolution of σ and the predicted functions for different values of regularization.

However, the mean squared error between the true function and the predicted function is lowest with the value predicted by the low regularization version of the algorithm (MSE = 0.082, 0.081, 0.041 for high, medium, low regularizations). Moreover, while the medium and high regularization kernels can improve in performance by subsequently tuning λ parameter (typically by lowering), the low regularization kernel seems to perform best using the original λ.

This suggests that Kernel Flows acts as a natural regularizer: while low regularization implies that the kernel can exactly interpolate the training points, the rho function and the randomness of the batch/sample pair encourages the kernel to generalize.

2.3 The ρ function

The above considerations warrant an investigation into the ρ function. Note that ρ is a deterministic function of the random variables X_b, Y_b, X_s, Y_s. We first make the obvious observation that ρ is non-convex (see Figure 3.1). Hence we cannot hope to systematically find the true minima. There are two standard measures which we adopt to mitigate this problem: first the randomness of the batch/sample selection, second the use of momentum in our gradient descent optimizer (such as Nesterov momentum). In general, increase randomness in batch/sample selection could help avoid local minima through the higher variance of ρ evaluated at any specific batch/sample combination. This suggests the use of smaller batch and samples.

We now examine how ρ changes depending on the sample size and the intrinsic noise of the dependent variable.

2.3.1 ρ and noise

For these tests, we use the same data set as in Section 2.2.3:

$$y_i = \sin(x_i) + \varepsilon_i, \quad \varepsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{2.5}$$

We consider three values of $\sigma = 0, 0.01, 0.1, 0.3$ and compute ρ for 100 samples (the batch is the whole data set). The figures below illustrate the results:

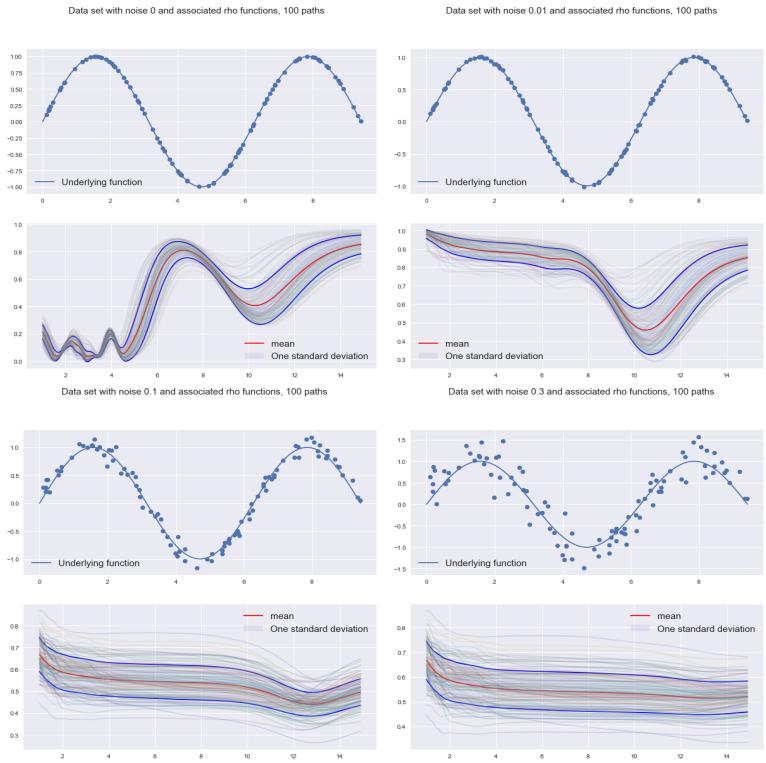


Figure 2.13: The data set and associated rho function for different values of the noise parameter, $= 0, 0.01, 0.1, 0.3$. The shaded area is the one standard deviation.

Noise does not only change the minimum of the ρ function, but also its shape: greater noise flattens out the function. Thus, noisier data means a smaller difference between kernels and slower learning of the optimal parameters θ.

2.3.2 ρ and sample size

The ρ function also depends on the size of both the batch and the sample. In the original paper [12], the size of the sample is set to the arbitrary quantity $N_s = N_b/2$, with N_b left as a hyper parameter. We wish to consider the generalized case where $N_s = p \times N_b, 0 < p < 1$. This warrants an investigation into behavior of ρ depending on the sample size.

In this section we consider the case where $N_b = N$ and the Gaussian kernel $K(x, x') = \exp(-\frac{||x-x'||^2}{2\sigma^2})$. The data set is generated by $Y = \sin(X), X \sim$

$\mathcal{U}(0, 3\pi)$, for 100 data points. We do not add noise. To investigate the relation between ρ and N_s, we compute ρ over the interval $[4, 15]$, with a spacing of 0.1. We do so for $N_s = p \times N_b, p = 0.25, 0.5, 0.75$, each for 100 sample selections and compute the mean value over the interval, as well as the one standard deviation. The results are presented below.

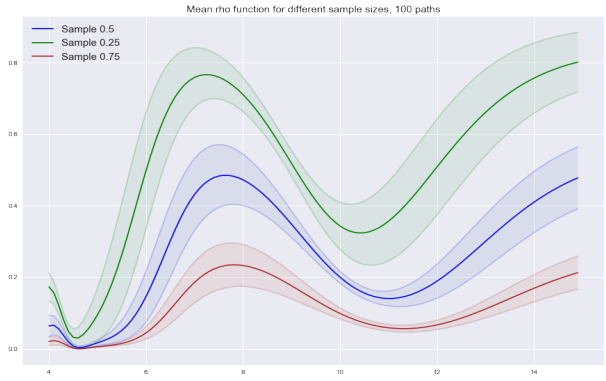


Figure 2.14: The rho function for different values of p. The shaded area is the one standard deviation.

We make three observations, one trivial and two informative. First, observe that smaller values of p lead to larger values of ρ. This is obvious since fewer points leads to a worse interpolating function. Second, ρ appears to have similar, albeit slightly different, minima and maxima. Finally, notice how bigger values of p lead to a smoother function, while smaller values lead to greater change: this seems to be because smaller values of p help accentuate the difference between Kernels.

These observations support the following hypotheses. First, there is no *a priori* best value for p: choosing any of the three values $0.25, 0.50.75$ would lead to a similar optimized parameter. Second, speed of training may be impacted by the choice of p: since smaller choices of p lead to greater changes in the values of ρ, this will affect its gradient. This second hypotheses is supported by figure (2.3.2): the size of the gradient varies depending on the choice of p. Moreover, near local minima/maxima, $\partial_\theta \rho$ seems to take smaller values.

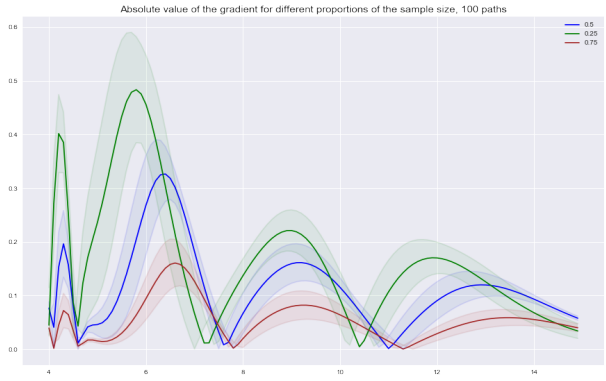


Figure 2.15: The absolute value of the gradient of the rho function for different values of p. The shaded area is the one standard deviation.

However, if the sample proportion is too small, this can have the opposite effect: differences between Kernels become smaller and ρ flattens out, as illustrated by 2.16

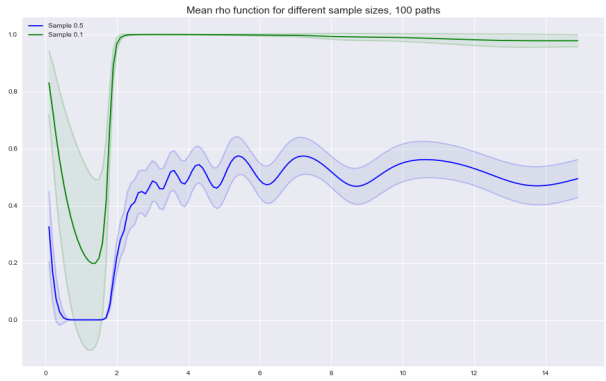


Figure 2.16: The rho function for different values of p. The data is generated by a Gaussian Kernel with $\sigma = 1.25$.

These observations suggests that the use different values of p for better performance and for a scheme to select a good value p, which we will discuss in the next section.

2.3.3 Sample size adjustments

In the original paper, the sample size is chosen to be $N_s = 0.5 \times N_b$. The choice of 0.5 is mostly arbitrary, but reflects the need for a kernel to generalize. In this section, we propose a modification of the original algorithm, based on changing the sample size N_s throughout training.

As was seen in the previous sections, both p and the intrinsic noise of the data affect the shape of ρ. Smaller values of p can increase $|\partial\rho|$ in regions far away from local minima/maxima. Conversely, in the region of a local minima/maxima $|\partial\rho|$ seems to increase with p. This suggests that when our kernel is bad we should select a small value of p, but when our kernel is good we should select a large value for p. We therefore propose to start with small values of p, such as 0.1 or 0.2, and to gradually increase p through training, up to the original 0.5. There are two versions of the proposed adaptive rate.

An additional benefit is that if ρ becomes smoother when the intrinsic noice of the data set is high, then using smaller values of p can help alleviate slow training with noisy datasets.

Finally, smaller values of p encourage greater generalization.

Linear increase

The first version we refer to as linear increase. It consists simply of choosing an interval $[p_{\min}, p_{\max}]$, where $0 < p_{\min} < p_{\max} \leq 1$. The proportion p of the sample size is chosen at each step so as to increase linearly throughout training.

Dynamic sampling

The second version is to adapt the p value based on ρ:

$$p = \frac{1}{2}(1 - \mathbb{E}[\rho_{0.5}])$$

where $\rho_{0.5}$ is the ρ function computed using a the standard half batch for the sample. The reasoning behind this choice is that $\mathbb{E}[\rho_{0.5}]$ provides the measure of performance of our Kernel. Note that p increases when ρ decreases and that $0 \leq p \leq 0.5$. Large values of $\mathbb{E}[\rho_{0.5}]$ indicate a bad Kernel which should use a small proportion to accelerate training and push it quickly towards values that promote generalization. Smaller values of $\mathbb{E}[\rho_{0.5}]$ indicate a good kernel which should use a proportion close to 0.5, to conform to the original proposed proportion. In practice, we would also impose a minimal value of p.

Moreover, since $\mathbb{E}[\rho]$ isn't available to us and changes throughout training, we will use at each iteration n

$$p_n = \frac{1}{2}(1 - \frac{1}{k}\sum_{i=n}^{n-k}\rho_i)$$

where k is some hyperparameter and ρ_i is the measured value of $\rho_{0.5}$ at iteration i. This has the disadvantage of incurring a higher computational cost, however since $\rho_{0.5}$ and ρ_p have common elements (specifically the term $||v_b||^2$), in practice this change does not incur a significant computational cost. Note that we do not use ρ_p to compute p_n because this can easily leads to no adaptive behavior in the sample size: a small sample size leads to a high ρ which leads to small sample size and so on.

Finally, it is also possible to set p to some static quantity less than 0.5.

Chapter 3

Kernel Flows: non-parametric version

3.1 Introduction: presentation of the algorithm

In this section we quickly summarize the non-parametric version of Kernel Flows derived in [12].

The non-parametric version of Kernel Flows is based on the same premise as the parametric version: to measure the efficacy of the kernel through a loss function $\mathcal{L}$ depending on half of the points. However, instead of perturbating the kernel parameters in the direction of gradient descent of $\mathcal{L}$, the non-parametric version perturbs the points in the direction of gradient descents of $\mathcal{L}$.

More precisely, given a prior kernel K, the goal of KF non-parametric version is to learn kernels of the form:

$$K_n(x, x') = K(F_n(x), F_n(x')) \tag{3.1}$$

where

$$F_1(x) = x \tag{3.2}$$

$$F_{n+1}(x) = F_n(x) + \varepsilon_{n+1} G_{n+1}(F_n(x)). \tag{3.3}$$

Note that this is still a kernel by basic properties of kernels ([2, p. 296]). This

is equivalent to learning a hierarchy of kernels defined by

$$K_{n+1}(x,x') = K_n(x + \varepsilon_n G_{n+1}(x), x + \varepsilon_n G_{n+1}(x')). \tag{3.4}$$

The functions F_n are referred to as the *flow*. The non-parametric versions of kernel flows therefore learns a new kernel K_n, from the base kernel K, by iteratively perturbing the points by some value $G_n(x)$ which depends on the position of the points at iteration n. The value $G_n(x)$ is in fact chosen at iteration n to be the direction of gradient descent of $\mathcal{L}(K, X_b, Y_b, X_s, Y_s, \theta)$, which depends on the base kernel K and will be ρ defined by (2.1). Note that ε_n will in general depend on n.

The algorithm can be summarized as follows. For each data point x_i denote $x_i^n = F_n(x_i)$, the point at iteration n of the flow, and let X^n be the whole data set at iteration n of the flow. For each point x_i denote $g_i^n = G_n(x_i^n)$, the perturbation computed at iteration n, and let G_b^n be the vector of perturbations of the batch.

Kernel Flows non-parametric version:

1. Choose a base kernel K.
2. Select a mini-batch (X_b^n, Y_b^n) (which can be the whole data set) and associated sample (X_s^n, Y_s^n).
3. Compute $g_i^n = -\partial_{x_i^n}\mathcal{L}(K, X_b^n, Y_b^n, X_s^n, Y_s^n, \theta)$ for each data point in X_b^n.
4. Compute $G^n(x_i) = \boldsymbol{K}(x_i^n, X_b^n)(\boldsymbol{K}(X_b^n, X_b^n))^{-1}G_b^n$ for the data points not in X_b^n. In other words compute the perturbations of the data points not in the batch by interpolation through the base kernel K.
5. Adjust the points by $F_{n+1}(x_i) = x_i^{n+1} = x_i^n + \varepsilon_n g_i^n$
6. Repeat steps 2-5.
7. To generate a prediction, use the kernel defined by $K_N(x_i, x_j) = K(x_i^N, x_j^N) == K(F_N(x_i), F_N(x_j))$.

Since $\mathcal{L} = \rho$, equation (6.5) of [12] gives an explicit formula for $-\partial_{x_i^n}\mathcal{L}(X_b^n, Y_b^n, X_s^n, Y_s^n, \theta)$. Letting $\hat{y_b^n} = K(X_b^n, X_b^n)^{-1}Y_b$, $\hat{z_b^n} = (\Pi^n)^T(K(X_s^n, X_s^n)^{-1})(\Pi^n)Y_b$, we have

$$g_i^n = 2\frac{(1-\rho)\hat{y_{b,i}^n}(\nabla_{x_i}K(x_i^n, X_b^n)\hat{y_b^n} - \hat{z_{b,i}^n}(\nabla_{x_i}K(x_i^n, X_b^n)\hat{z_b^n}}{Y_b^T K(X_b^n, X_b^n)^{-1}Y_b} \tag{3.5}$$

Note that Kernel Flows depends on the base kernel K through the flow in two important ways.

- $\mathcal{L}$ (and the G_n) depend on K. Therefore the hierarchy of kernels (3.4) can only modify the original kernel through linear perturbations dependent on K at every iteration.
- The perturbations g_i^n of the points not in X_b^n depend on interpolation by the kernel K. Notably, the perturbations of test set are only done through interpolation by the kernel K.

Hence choosing a good kernel K (and good parameters θ) greatly affects the final kernel K_N. One possible solution to choosing good parameters θ is to use Kernel Flows parametric to optimize θ.

3.2 The choice of ε

Unlike the parametric version of Kernel Flows (and many gradient based algorithms), ε cannot be simply set to some small value. Instead we choose the perturbations depending on the data set. Two common choices (originally proposed in [12]) are

- Choose ε_n at every step n such that the absolute translation is less than some specified learning rate α: $\max_i ||\varepsilon_n g_i^n|| \leq \alpha$.
- Choose ε_n at every step n such that the relative translation is less than some specified learning rate α: $\max_i \frac{||\varepsilon_n g_i^n||}{||x_i^n||} \leq \alpha$.

At inference time, we propose three choices for the ε_n of the test set:

1. Use ε_n^{train}, the same as the training set at every step.
2. Calculate ε_n^{test} at every step, based on the test set but using the same rule as the training set.
3. Use $\min\{\varepsilon_n^{train}, \varepsilon_n^{test}\}$.

On synthetic datasets options 2 and 3 perform the best, with option 3 performing slightly better. On real datasets, option 3 significantly outperforms the other two.

3.2.1 Computational cost and implementation

Under the same assumptions as the ones presented in the KF Parametric version, the computational complexity of computing the gradient is similar, but now depends on the dimension of each point within the dataset $x \in \mathbb{R}^d$. This follows from the fact that $\nabla_{x_i} K(x_i^n, X_b^n)$ has dimension $d \times N_b$. All other quantities being identical, using what was derived in section 2.1.3, we deduce that the overall complexity of the algorithm is $\mathcal{O}(dN_b^3)$.

The implementation is done using the same numpy package as in the Parametric case ([11], [17]) and is also available at [5].

3.3 Non parametric Kernel Flows and brittleness

One of the properties of the non-parametric version of Kernel Flows is its apparent sensitivity to small changes in the training set and hyper-parameters. We illustrate this property through the Swiss roll cheesecake data set, first presented in [12], pictured below. The points are labelled $y_i = \pm 1$, depending on the spiral.

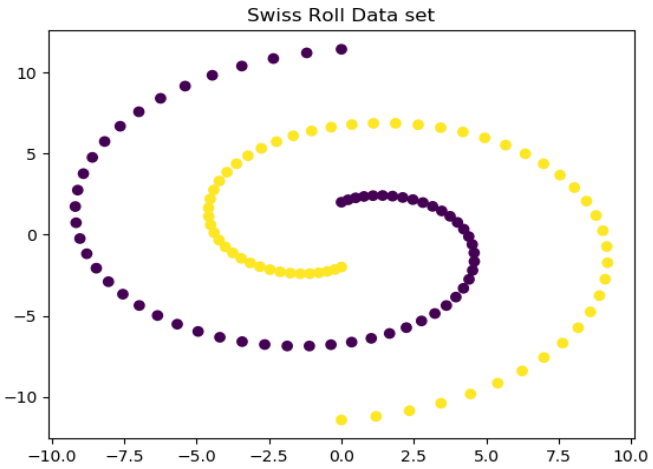


Figure 3.1: Swiss roll cheesecake, with 120 points.

We first train kernel flows with the Swiss roll 120 points. The base kernel

K is the RBF kernel with parameter $\sigma = 2.0$ and $\varepsilon = 20\%$ (a large value). We train for 10000 iterations.

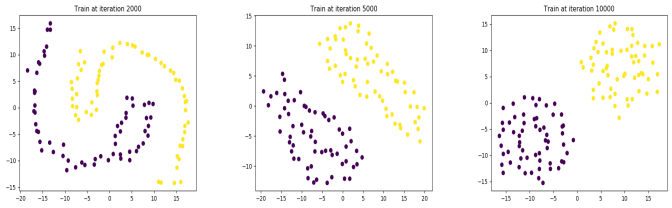


Figure 3.2: The flow transformation with 120 points, $\varepsilon = 20\%$.

Kernel Flows quickly linearly separates the data set. We repeat the procedure, but this time with 80 points (40 points in each spiral instead of 60).

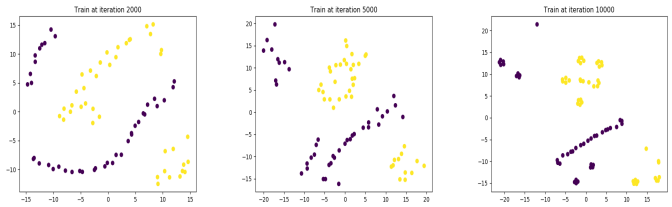


Figure 3.3: The flow transformation with 80 points, $\varepsilon = 20\%$.

In this case, the data set is not linearly separated. This can be solved by reducing the hyperparameter $\varepsilon = 10\%$. However, since the learning rate is smaller, this also requires to increase the number of iterations.

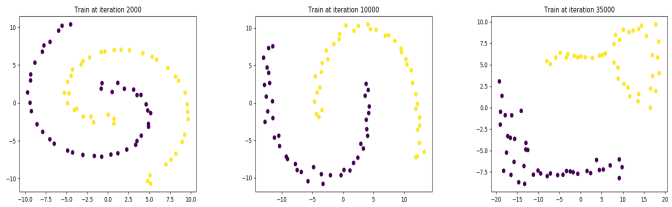


Figure 3.4: The flow transformation with 80 points, $\varepsilon = 10\%$.

In this case, the data is linearly separated after 35000 iterations. Moreover, the final configurations is different from the 120 points version.

Finally, we go back to the 120 point version, $\varepsilon = 10\%$, but with $\sigma = 5.0$. Training is done over 30000 iterations.

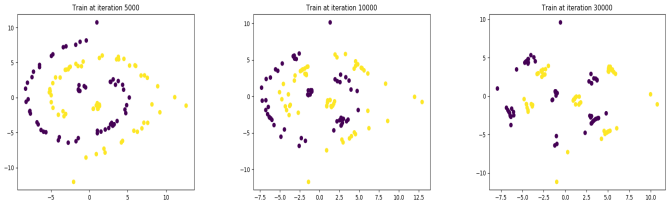


Figure 3.5: The flow transformation with 120 points, $\varepsilon = 10\%$ $\sigma = 5.0$.

In this case, the data is never linearly separated. Further tests show that linear separability does occur with $\sigma = 3.0$, but not with $\sigma = 4.0$.

These tests illustrate how the initial choice of kernel can have an enormous impact on the performance of the algorithm. It is therefore advisable to first optimize the parameters of the base kernel through Kernel Flows parametric version. Moreover, small changes in the dataset may require careful adjustments of the learning rate to achieve the desired result. It was noted in [12], section 8.1, that KF can use the britelness of deep learning to its advantage. The above examples suggests that Kernel Flows is also vulnerable to small changes in the original dataset.

3.3.1 On the importance of regularization

Recall that when discussing the non-parametric version of Kernel flows, we observed that different values of the regularization parameter λ lead to different optimized kernels. In this section, we show how regularization also greatly impacts the effectiveness of KF Non-parametric.

We first consider the case of a linear data set and the linear kernel. We expect that KF would not modify the data set since it already is perfectly interpolated by the base kernel.

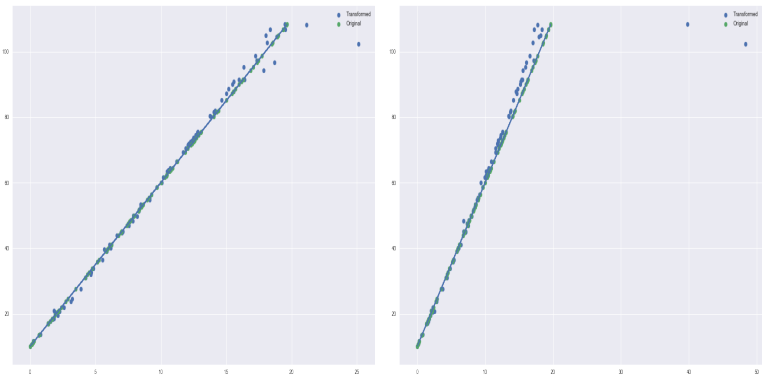


Figure 3.6: The flow transformation with a line and linear kernel for the base kernel. Left is $\lambda = 0.01$, right is $\lambda = 0.00001$. Some points on the right are "ejected".

We do get the expected behavior when we use $\lambda = 0.01$, but with smaller values, some points are occasionally ejected from the line. This behavior was first noted in [12] for the swiss roll dataset.

However, the effect of λ is not simply limited to the behavior of a few data points. We reconsider the swiss roll data set of the previous section and KF with a Gaussian base kernel, with $\sigma = 4.0$. Such a value does not linearly separate the data set and leads to behavior similar figure (3.3), when $\lambda = 10^{-5}$. Increasing λ to 0.01 can lead to the data being linearly separated, but using a higher value 0.1 can lead to the algorithm reverting its behavior. The dataset does not contain any intrinsic measurement noise, which is generally the reason to utilize regularization.

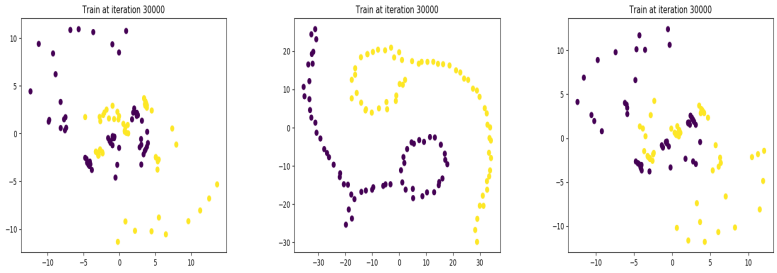


Figure 3.7: The swiss roll transformed for $\lambda = 10^{-5}$ (left), $\lambda = 0.01$ (middle), $\lambda = 0.1$ (right).

Note however that this behavior is inconsistent: even with $\lambda = 0.01$, the data will sometimes exhibit behavior similar to low regularization.

This behavior seems to be a case of the underfitting/overfitting problem. When we optimized σ through kernel flows non parametric, our base kernel fit the data very well. Moreover, as was seen in the section discussing KF and regularization, the parameter σ is optimized so that the kernel was intrinsically regularized. In the case where $\sigma = 4.0$ we have a bad base kernel which lends itself to overfitting the data when regularization is too low, but underfitting the data when regularization is too high.

We also note that in all these tests, we have used the full data set for the batch and therefore do not need to find the perturbation for the data points outside the batch. The problems exhibited by a bad kernel on the flow are magnified when this is the case.

3.4 Bad kernels and kernel flows

The previous observations highlight the need for a good base kernel for Kernel Flows to work well. In this section we layout strategies to ensure a good base kernel and to limit the effects of the fragility of Kernel Flows.

The first two are straightforward from the previous sections:

- Optimize the base kernel through Kernel Flows Parametric.
- Optimize the regularization parameter.

The next will be discussed in the next section and is a hybrid approach between the parameteric and non-parametric versions of kernel flows, which adapts both the flow and the base kernel parameters θ

3.4.1 Kernel Flows: a hybrid approach

In this section we present the hybrid approach, which combines both versions of Kernel Flows. The motivation behind this version is the simple observations that bad base kernel K is bad on two fronts:

- In determining $g_i^n = \partial_{x_i^n} \rho$
- In computing $G^n(x_i)$ through interpolation of the perturbations g_i^n

This suggests that we should use the parametric version of Kernel Flows to optimize the parameters θ. We propose that not only should we do this before using KF non-parametric, but to also adapt the parameters of the base kernel K throughout the flow transformation. This leads to a hybrid approach between the parametric and non-parametric versions:

1. Select (X_b^n, Y_b^n) (which can be the whole data set) and (X_s^n, Y_s^n).
2. Compute $g_i^n = -\partial_{x_i^n} \mathcal{L}(K, X_b^n, Y_b^n, X_s^n, Y_s^n, \theta)$ for each data point in X_b^n.
3. Compute $\nabla_\theta \mathcal{L}(X_b, Y_b, X_s, Y_s, \theta)$.
4. Adjust the points by $x_i^{n+1} = x_i^n + \varepsilon_n g_i^n$.
5. Adjust the parameters $\theta \leftarrow \theta - \delta \nabla_\theta \mathcal{L}(X_b, Y_b, X_s, Y_s, \theta)$.

The proposed approach leads to learning kernels of the form

$$K_n(x, x') = K_{\theta_n}(F_n(x), F_n(x')). \tag{3.6}$$

We now illustrate the benefits of such an approach, through the swiss roll data set.

First we consider the Gaussian kernel with $\sigma = 4.0$ and low regularization. Recall that this is a bad kernel. While the data is not linearly separated, it is arranged in lines following the classes, figure 3.8. These results can be replicated with larger values of σ or when using half of the data set for each batch, figure 3.9.

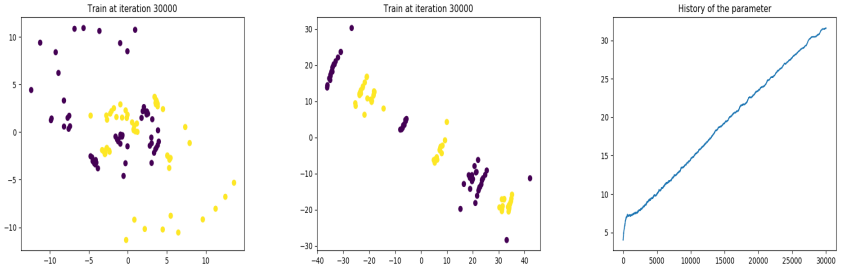


Figure 3.8: Swiss roll transformed with $\sigma = 4.0$, non-parametric (left), hybrid version (middle) and the parameter history (right).

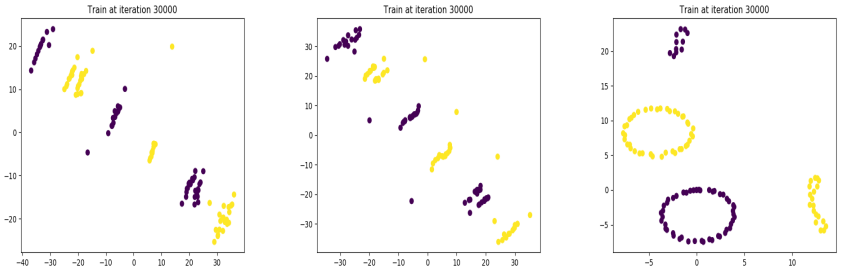


Figure 3.9: Swiss roll transformed with $\sigma = 5.0$ (left), $\sigma = 8.0$ (middle) and $\sigma = 4.0$ with half the points in a batch (right).

But is the hybrid version simply correcting a bad kernel which could've been better selected to begin with? This time we use kernel flows with a good kernel where σ is optimized first through the parametric version of Kernel Flows. While both versions exhibit similar behavior, the hybrid version seems to linearly separate the data slightly faster (Figure 3.10. Moreover, the parameter σ never stops evolving, despite being optimized *a priori*. This suggests that the hybrid version is doing more than simply correcting a bad base kernel and instead is adapting σ to the flow functions F_n.

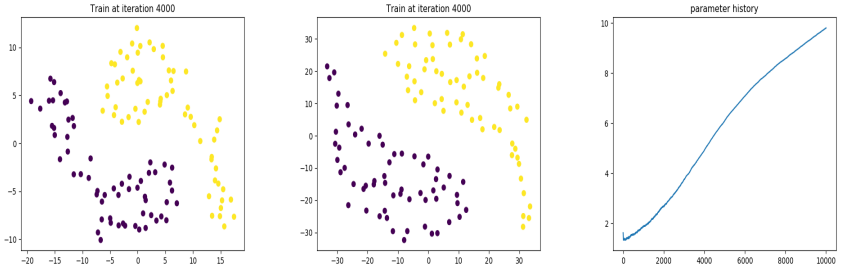


Figure 3.10: Swiss roll transformed with $\sigma^2 = 2.0$ (left), $\sigma^2 = 2.0$ hybrid version (middle) and the parameter evolution for the hybrid version (right).

Chapter 4

Radial Basis Networks and Kernel Flows

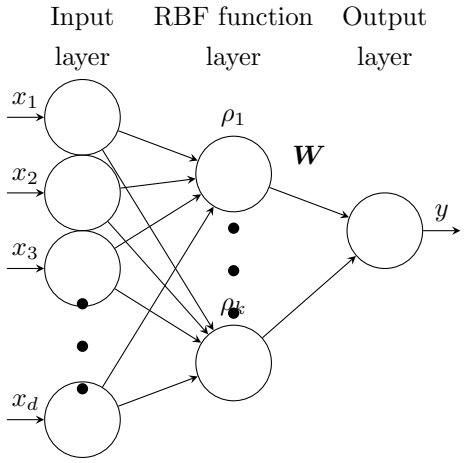
In this chapter we present the Radial Basis Function network. We then propose a modification of the original RBF network which is a Kernel Regression model using a linear combination of kernels. This modified network can be trained using Kernel Flows parametric version.

4.1 Introduction to Radial Basis Netwroks

The standard Radial Basis Function (RBF) network can be represented through the function:

$$\phi(x) = \sum_{i=1}^{k} w_i \rho(||x - \mu_i||, \boldsymbol{\sigma}_i) \tag{4.1}$$

where the $\boldsymbol{\mu}_i$ represent the centroids of the RBF function and the $\boldsymbol{\sigma}_i$ represent any other parameter of the RBF functions [3, p. 2]. These can be constant throughout all RBF functions, $\boldsymbol{\sigma}_i = \boldsymbol{\sigma}$. We will consider functions ρ of the form of the Gaussian kernel (1.7) and the Rational Quadratic Kernel (1.9). This can be schematized as follows:



Where $\boldsymbol{W}$ represents the weights w_i.

A particular case of this architecture is when we wish to interpolate exactly each point in the training data set $(x_i, y_i)_{i=1}^N$. In this case, the network architecture contains N RBF functions and $\mu_i = x_i$: each RBF function is centered on one training data point. In such a case the weights w_i are uniquely determined and the equation for a new data point takes the familiar form (writing $K(x, x')$ for $\rho(||x - x'||)$):

$$\phi(x) = \boldsymbol{K}(x, X)(\boldsymbol{K}(X, X))^{-1}Y$$

where $\boldsymbol{K}(x, X) = (K(x, x_1), ..., K(x, x_N)), Y = (y_1, ..., y_N)^T$ and $\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x})$ is the $N \times N$ Gram matrix with i, j entries $K(x_i, x_j)$, [3, p. 3]. Therefore our RBF network is simply kernel regression with kernel $K = \rho$. This can be regularized by:

$$\phi(x) = \boldsymbol{K}(x, X)(\boldsymbol{K}(X, X) + \lambda \boldsymbol{I}_N)^{-1}Y.$$

4.2 Properties of RBF networks

One of the many useful properties of the RBF network is that it possesses properties of universal approximation.

Theorem 4.2.1. *Universal $L^p(\mathbb{R}^d)$ approximation of RBF networks, [13]. The family of networks defined by equation* (4.1)*, with constant parameter σ, is dense in $L^p(\mathbb{R}^d)$, for all $\in [1, \infty)$.*

Note that the above theorem requires the specification of an arbitrary number of centroids μ_i. Generally, these centroids are determined through some

unsupervised learning method such as clustering. The weights are then trained through some supervised learning method such as gradient descent [16]. Hence it may be more practical to use every point in the data set as a centroid and to optimize a single smoothing factor σ. In this case, the RBF network reduces to the usual kernel regression and the weights can be determined explicitly by the standard theory of Kernel Regression.

4.3 A multilayer RBF network

4.3.1 Equations and architecture

We now propose a multilayer version of the above architecture. The equation for this network is:

$$\phi(x) = \sum_{i=1}^{N} w_i \Big(\sum_{j=1}^{l} \alpha_j \rho_j(||x - x_i||, \boldsymbol{\sigma}_j) \Big) \tag{4.2}$$

where $\alpha_j \geq 0$. The network architecture is based on the following property of kernels: if K_1 and K_2 are kernels, then $K(x, x') = c_1 K_1(x, x') + c_2 K_2(x, x')$ is also a kernel, where $c_1, c_2 > 0$. Hence (4.2) is equivalent to

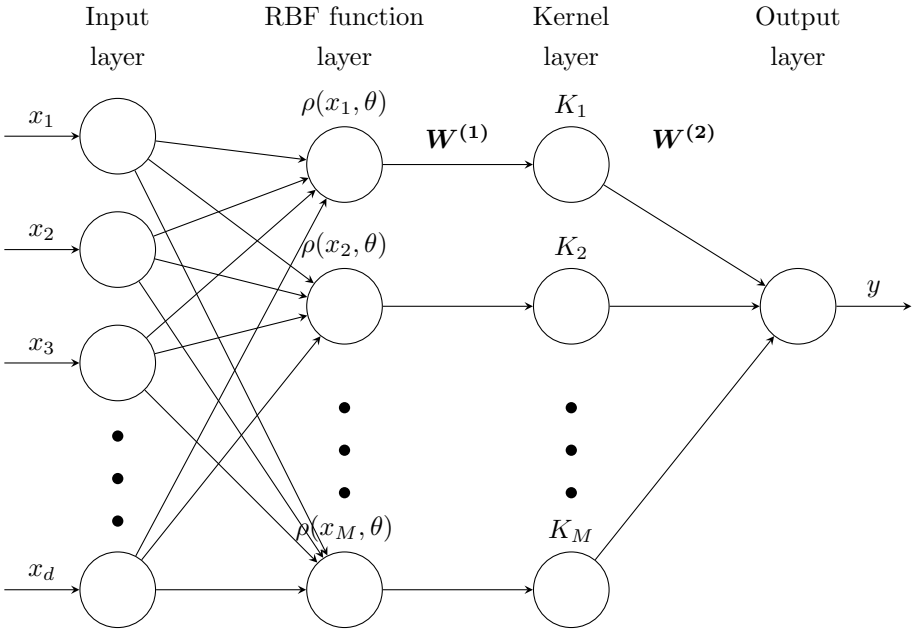
$$\phi(x) = \sum_{i=1}^{N} w_i K(x, x_i) \tag{4.3}$$

with

$$K(x, x_i) = \sum_{j=1}^{l} \alpha_j \rho_j(||x - x_i||, \boldsymbol{\sigma}_j) \tag{4.4}$$

Note that the ρ_j can be of the same or different forms (e.g. they can all be Gaussian kernels or Rational Quadratic kernels or a mix of both). Equation (4.3) can be solved in the usual manner with kernel regression using the kernel defined above.

This can also be thought of as multilayer version of the RBF network, where we denote $K_i = K(x, x_i)$:



We note that $\boldsymbol{W^1}$ contains the α_j weights, which act on the outputs of each ρ_i neurons, and $\boldsymbol{W^2}$ contains the w_j weights, which act on the K_i neurons as a whole. Each ρ neuron can be through of as further containing l sub-neurons, $\rho_i^{(j)}$. Each $\rho_i^{(j)}$ sub-neuron is associated with the same RBF function across all ρ_i neurons, i.e. ρ_i^j and ρ_k^j share the same activation with the same parameters, save for the centroid which is dependent on ρ_i.

4.3.2 Advantages of the proposed approach

We highlight two advantages of the proposed architecture. These follow from the fact that the network is not a network in itself, but a form of kernel regression using the combination of kernels.

First, the network contains few learned parameters. Because the centroids and the linear weights are determined from the data set, the number of parameters only depends on the number of basis functions chosen for the kernel. In the case of the Gaussian the number of learned parameters is $2 \times n$ where n is the number of chosen basis functions. In the case of the Rational Quadratic kernel, the number of parameters is $3 \times n$.

The second is that the Kernel Flows algorithm can be used to train the network.

4.3.3 Training

The above architecture presents significant benefits: the number of basis functions K, the weights $W^{(2)}$ and the centroids $c_i = x_i$ are entirely determined by the training data (hence the kernel layer is not a hidden layer in the usual neural network sense). The parameters to be trained are σ_j and $W^{(1)}$. The number of RBF functions ρ is a hyperparameter.

For training, we use the parametric version of the Kernel Flows algorithm, with the kernel defined by (4.4).

The initialization of the parameter can be random. However in the case where ρ is the Gaussian kernel, σ can be initialized to the variance of the norms of the training points $(x_i)_{i=1}^N$. The α_j are initialized randomly between 0 and 1.

Chapter 5

Applications to real data sets

In this section we implement KF for applications to three standard data sets. Two are available from the Scikit Learn Python library [14]: the boston housing dataset [9] and the diabetes dataset. The last one is a dataset available from the UCI Machine Learning Repository [7]: the Wine quality dataset [4].

5.0.1 Setup

To test the performance of the Kernel Flows arlgorithms, we will use a 5-fold cross validation, meaning that the data set is split into 5 segments and KF is run with 4 segments as training set and the last segment as test set. This is repeated 5 times so that each segment is used once as a test set. The performance is recorded for each run and averaged. We do this for several hyper parameter choices.

In each case we train the kernel flows algorithm over 10 000 iterations, with the Gaussian kernel. We initialize the σ parameter to the values $1, 100, 500, 1000$, the "natural variance" $\hat{\sigma} = Var(X_{norm})$ where X_{norm} is the squared norm of each data point, and, as in [12], σ_{MSD} such that $\frac{1}{2\sigma^2}$ is equal to the mean squared distance between data points. In each case, regularization λ is first tuned using the initialization value. We do the same for the Rational quadratic kernel, with initialization values $\alpha = 0.5, \beta = 1.0$. In all cases presented we used the Nesterov momentum update rule, unless specified, as it generally performed the best.

For KF non-parametric version, we use the σ parameter with the lowest MSE, based on the KF parametric version, which we will denote as σ_{opt}.

For the parametric version ρ and its gradient is computed using a batch of size of 100. For the non-parametric version we also train the algorithm with larger batch sizes, but for a reduced number of iterations. We first present the setup of the experiments, then provide tables with the best results. The best result is highlighted in bold and the errors on the training data is in parenthesis. Finally we discuss the results and provide interpretations.

In all cases we do **not** pre-process the data.

Performance metrics

We will use two main performance metrics, the mean squared error and the mean absolute error. The Mean Absolute Error (MAE) is defined as

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{5.1}$$

where y_i is true dependent value of the ith sample and $\hat{y}_i$ the predicted dependent value. The Mean Squared Error (MSE) is defined as

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{5.2}$$

Compared to the MAE, the MSE punishes large errors, but reduces errors less than one. Hence, while the MAE is easier to interpret, the MSE can indicate if very large errors occur.

We also run the same experiments for the RBF network/kernel regression with a linear combination of Gaussian kernels which was proposed in chapter 4. We train the network for 10000 iterations with the same 5-fold cross validation as the Kernel Flows Parametric and Non-Parametric. We will use three basis functions which we initialize to parameters $\sigma = 1.0, 5.0, 10.0$ respectively.

5.0.2 Boston Housing data set

Description

The data set consists of 506 data points, with dimension 13: $X \in \mathbb{R}^{506\times 13}$. The features are attributes of houses in an area, such as the per capita crime rate of

the town or the average number of rooms per house. The target values range are the median price (in thousand of dollars) of the homes in the are. The target values range from 5 to 50, with a mean of 22.53 and a variance of 84.4.

Results

Table 5.1: KF Parametric version

Method	MSE	MAE
Gaussian, $\sigma = \hat{\sigma}$	34.363 (7.564)	4.009 (1.881)
KF Gaussian, $\sigma = \hat{\sigma}$	32.876 (11.59)	3.901 ((2.356)
KF Gaussian, $\sigma = \hat{\sigma}$ with dynamic sampling	**30.738** (10.264)	**3.763** (2.214)
Rational quadratic, $\alpha = 0.5, \beta = 1.0$	126.024 (0.0)	7.942 (0.0)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$	72.407 (0.0)	5.662 (0.0)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$ with dynamic sampling	72.019 (0.0)	5.717 (0.0)

Table 5.2: KF Non-Parametric version. In all cases, $\varepsilon = 0.1\%$.

Method	MSE	MAE
Gaussian Kernel, $\sigma = 501.4$	47.830 (28.406)	4.978 (3.825)
Gaussian Kernel, $\sigma = 501.4$, hybrid version	48.187 (28.230)	5.005 (3.823)
Gaussian Kernel, $\sigma = 501.4$, full dataset	**39.603** (26.02)	**4.501** (3.642)
Gaussian Kernel, $\sigma = 501.4$, full dataset, hybrid version	40.083 (26.07)	4.523 (3.651)

5.0.3 Diabetes data set

Description

The data set consists of 442 data points, each with 10 features: $X \in \mathbb{R}^{442\times 10}$. The features are characteristics of patients such as sex and age. The target values, which correspond to the disease progression, range from 25 to 346, with a mean of 152.13 and a variance of 5929.88.

Results

Table 5.3: KF Parametric version.

Method	MSE	MAE
Gaussian Kernel, $\sigma = 10$	2936.088 (2519.247)	**43.367** (40.228)
KF Gaussian Kernel, $\sigma = 10$	2911.321 (2664.77)	43.559 (41.593)
KF Gaussian Kernel, $\sigma = 10$ with dynamic sampling	**2909.619** (2656.92)	43.551 (41.53)

Table 5.4: KF Non-Parametric version. In all cases $\varepsilon = 1\%$.

Method	MSE	MAE
Gaussian Kernel, $\sigma = \sigma_{opt}$	3063.7876 (1560.291)	45.4709 (32.699)
Gaussian Kernel, $\sigma = \sigma_{opt}$, hybrid version	**3057.3552** (1561.337)	**45.4198** (32.69)

5.0.4 Wine quality data set

Description

The Wine quality data set was first used in [4]. The features are 11 characteristics of wine such as pH and sugar. The target values are grades corresponding to the perceived quality of the wine by wine tasting expert, ranging from 1 to 10 (10 being the highest grade). There are two sub-datasets, one for red wine and the other for white wine. Denote (X_r, Y_r) the data set of the red wine and (X_w, Y_w) the data set of the white wine. We have $X_r \in \mathbb{R}^{1599\times 11}$, $X_w \in \mathbb{R}^{4898\times 11}$. Y_r has mean 5.63 and variance 0.65. Y_w has mean 5.88 and variance 0.78.

Because the range of Y is restricted to $[0, 10]$ we will also restrict any prediction to this range, meaning that any value outside will be rounded down to 10 or up to 1.

Results

Red wine

Table 5.5: KF Parametric version.

Method	MSE	MAE
Gaussian, $\sigma = 500.0$	0.442 (0.424)	0.512 (0.503)
KF Gaussian, $\sigma = 500.0$	**0.4202** (0.3917)	**0.503** (0.487)
KF Gaussian, $\sigma = 500.0$, with dynamic sampling	**0.4202** (0.3917)	**0.503** (0.487)
Rational quadratic, $\alpha = 0.5, \beta = 1.0$	0.717 (0.0)	0.643 (0.0)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$	0.6412 (0.0)	0.609 (0.001)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$, with dynamic sampling	0.577 (0.0)	0.587 (0.002)

Table 5.6: KF Non-Parametric version. In all cases $\varepsilon = 1\%$.

Method	MSE	MAE
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 100$, 10000 iterations	0.554 (0.238)	0.592, (0.375)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 300, 1000$ iterations	**0.487** (0.28)	**0.542** (0.41)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 100$, 10000 iterations, hybrid version	0.55, (0.236)	0.589 (0.374)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 300$, 1000 iterations, hybrid version	0.489, (0.283)	0.541, (0.412)

White wine

Table 5.7: KF Parametric version.

Method	MSE	MAE
Gaussian, $\sigma = 500.0$	0.571 (0.54)	0.594 (0.578)
KF Gaussian, $\sigma = 500.0$	**0.5631** (0.523)	**0.588** (0.567)
KF Gaussian, $\sigma = 500.0$, with dynamic sampling	**0.5631** (0.523)	**0.588** (0.567)
Rational quadratic, $\alpha = 0.5, \beta = 1.0$	0.728 (0.0)	0.668 (0.0)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$	0.682 (0.0)	0.618 (0.0)
KF Rational quadratic, $\alpha = 0.5, \beta = 1.0$, with dynamic sampling	0.692 (0.0)	0.656 (0.002)

Table 5.8: KF Non-Parametric version. In all cases $\varepsilon = 1\%$.

Method	MSE	MAE
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 100$, 10000 iterations	0.668 (0.518)	0.631 (0.552)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 300$, 1000 iterations	0.641 (0.532)	0.624 (0.552)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 100$, 10000 iterations, hybrid version	0.676, (0.517)	0.635 (0.55)
Gaussian Kernel, $\sigma = \sigma_{opt}$, $N_b = 300$, 1000 iterations, hybrid version	**0.62** (0.602)	**0.619** (0.609)

Cortez et al.in [4] reported the best results using the MAE of 0.46 and 0.45 using SVM on these data sets. Moreover they reported errors of 0.50/0.59 and 0.51/0.58 using polynomial regression and neural networks respectively. These results suggest that KF Parametric performs as well as polynomial regression.

5.1 RBF network results

Table 5.9: RBF Network.

Dataset	MSE	MAE
Boston Housing Dataset, untrained network	403.054 (0.0)	17.014 (0.0)
Boston Housing Dataset, trained network	110.94 (0.0)	7.631 (0.0)
Diabetes untrained network	2912.837, (0.0)	43.356 (0.007)
Diabetes, trained network	2885.161, (2639.445)	43.050 (40.120)
Red wine, untrained network	0.848 (0.0)	0.668 (0.007)
Red wine, trained network	0.66, (0.0)	0.625 (0.0)
White wine, untrained network	0.8855, (0.0)	0.702 (0.005)
White wine, trained network	0.642, (0.0)	0.630 (0.0)

5.2 Result interpretation

In this section we interpret the results of our numerical experimentations.

5.2.1 Kernel Flows Parametric

5.2.2 Parameter initialization

We first observe that the parametric version of Kernel Flows generally performs well and improves on Kernel Linear Regression with same initial parameter. However we note that the effectiveness of KF parametric depends on the initial set of parameters. For example, for the Boston Housing dataset, an initial choice of $\sigma = \hat{\sigma}$ improves kernel regression from 34.363 to 32.876 (an improvement of 4.3 %). On the other hand, when the parameter is initialized at $\sigma = 500$, the MSE goes from 31.108 to 31.112 (no change). Hence the parameters should be carefully initialized, depending on the specific problem. In general it is recommended to choose a broad number of values from which to initialize the algorithm.

5.2.3 Batch size

In our presentation of Kernel Flows Parametric in Chapter 2, we explained that reducing the batch size could potentially increase performance by helping avoiding local minima by increasing the variance of ρ. Here we provide evidence that this is indeed the case. The table below compares some values obtained between batch sizes. In all cases we used KF with the Gaussian kernel and Nesterov momentum update.

Table 5.10: KF Parametric version for different batch sizes.

Method	MSE	MAE
KF diabetes, $\sigma = 10$, batch size 353 (full dataset)	2919.366 (2700.84)	43.603 (41.914)
KF diabetes, $\sigma = 10$, batch size 100	2911.321 (2664.77)	43.559 (41.593)
KF Red wine, $\sigma = \hat{\sigma}$ batch size 500	0.537 (0.303)	0.552, (0.424)
KF Red wine, $\sigma = \hat{\sigma}$, batch size 100	0.499 (0.322)	0.5421 (0.439)
KF White wine, $\sigma = \hat{\sigma}$ batch size 500	0.662 (0.426)	0.616 (0.509)
KF White wine, $\sigma = \hat{\sigma}$, batch size 100	0.603, (0.444)	0.596, (0.521)

In some cases performance improved only slightly for smaller batches, but in other cases, such as the wine data sets, the MSE decreases by 7-8% and the MAE by 1.6-1.8%. Hence, recalling that increasing the batch size scales the

computational costs cubically, it is strongly advisable to use small batch sizes.

5.2.4 Sample size

Overall, Kernel Flows parametric performs as well or better with dynamic sampling: on the Boston Housing dataset, dynamic sampling improves the MSE of standard KF by 6.5% for the Gaussian kernel. On the red wine dataset dynamic sampling improves the MSE of the rational quadratic kernel by 10%.

However it is unclear if the good performance of the dynamic sampling is due to the adaptive nature of the sample size or to the use of a smaller sample size in general. Figure 5.1 illustrates how sometimes ρ (and the associated sample size) does not radically evolve over time. Hence most of the success of this method could stem from using a sample size smaller than 0.5. Nonetheless, the value of ρ can provide a good means to an appropriate sample size, regardless of any adaptation over time.

In some cases simply reducing the proportion of the batch used for the sample leads to increase performance. In one run of the red wine dataset, using a kernel defined as a linear combination of Gaussian kernels and with SGD as the optimizer, reducing the proportion from 0.5 to 0.3 improved the MAE from 0.494 to 0.485, a reduction of 1.8%. Further reducing from 0.5 to 0.1 improved the MAE to 0.468, a reduction of 5.3%. In general, using smaller proportion performed the best when the chosen Kernel has poor initial performance or when the Kernel has many different parameters to optimize.

The fact that the dynamic sample size method can improve on the standard version of Kernel Flows indicates that the sample size is an important hyper-parameter which should be tuned to the specific problem to be solved.

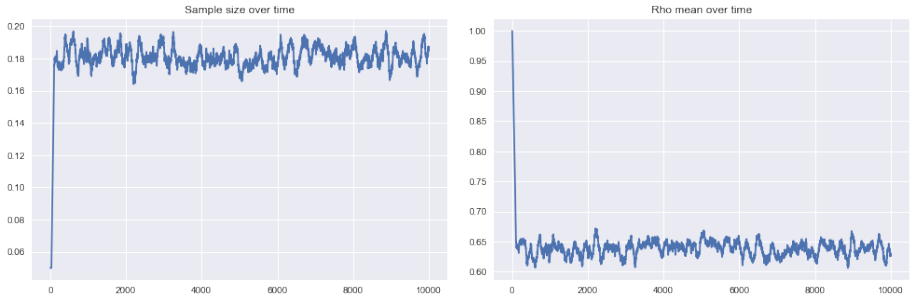


Figure 5.1: The sample size (left) and rho mean (right) for the Boston Housing dataset

5.2.5 Kernel Flows Non-Parametric

Overall performance

Overall, Kernel Flows non-parametric version performs more poorly than both Kernel Flows parametric and regular kernel regression, even when the base kernel is optimized through Kernel Flows parametric. This is despite the fact that rho does get minimized through the algorithm (see Figure 5.2). There are several possible explanations.

First, we might be using batches which are too small: for the wine dataset, due to computational constraints, we used either batches of size 100 for 10000 iterations or batches of size 300 for 1000 iterations. We note that in all cases, the larger batch always outperformed the smaller batch, despite fewer iterations.

Second, we might be using a number of iterations too small for the algorithm to perform well. However, this seems unlikely since ρ significantly decreases in our numerical experiments.

Third, the ε and regularization hyper-parameters might be incorrectly tuned. We opted for a high level of regularization (10^{-2} in general) based on the performance of the base kernel. We also used ε to be 1% or 0.1%. Changing these values might improve performance, as was seen for the swiss roll dataset.

Finally, Kernel Flows Non-Parametric might not be well adapted to regression problems.

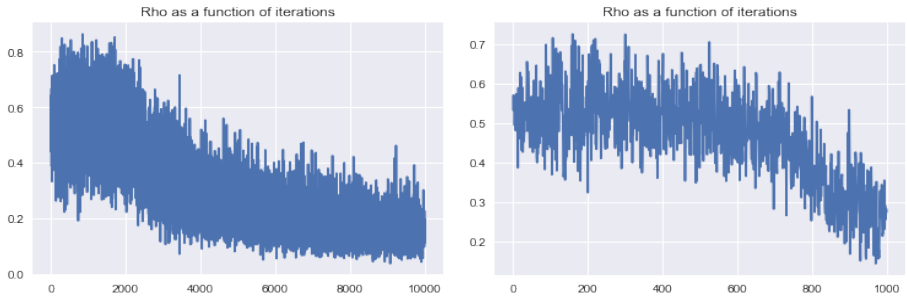


Figure 5.2: The value of ρ over time, for 10000 iteration with batch size 100 (left) and 1000 iterations with batch size 300 (right).

Kernel Flows Hybrid version

The hybrid version of Kernel Flows generally performs about as well as Kernel Flows Non-Parametric, and sometimes better. Notably, for the white wine

dataset, the MSE is improved by 8.3% and the MAE by 2.5%. In all other cases, the performance is similar. This indicates that the hybrid version should be considered when using Kernel Flows as it presents a small increase in computational cost, but can significantly improve the performance of the algorithm.

This is especially true if the parameters are not optimized through Kernel Flows Parametric first. On the Boston Housing dataset, setting $\sigma = \hat{\sigma}$, KF non-parametric leads to a large errors: the MSE is 331.641 and the MAE is 14.833. Training the hybrid version of KF with the same parameters improves the MSE by 18 % to 270.833 and the MAE by 12% to 13.036.

5.2.6 RBF network

In all cases, training using the Kernel Flows algorithm improves the performance of the network. In the case of the Boston Housing Dataset, the MSE is reduced by 72% and the MAE is improved by 55%. For the wine dataset, the MSE are reduced by 22% / 27.5% and the MAE are improved the 6.4%/10%. For the diabetes dataset however, the improvement is negligible. This shows that Kernel Flows is an adequate training method for these types of networks.

However, overall the RBF networks trained in these experiments perform worse than the simpler kernel regression with a single Gaussian kernel and Kernel Flows Non-Parametric. These networks seem to overfit the data, as can be seen from the very low training data errors, often 0. A method of regularization is therefor required to make these networks work. The usual regularization parameter λ does not significantly change the test accuracy, even with high values. One exception is the diabetes dataset where it performs better than all other methods, perhaps because the relation is hard to capture through Kernel Regression hence the added complexity of the RBF network improves performance without overfitting. The improvement is small however.

5.2.7 Kernel Flows: limitations and shortcomings

While Kernel Flows does often improve the performance of the algorithms presented here, this is not always the case. The first example is in the case of Kernel Flows Non-parametric, where despite ρ being minimized (see Figure 5.2), the MSE and MAE are **larger** than with kernel regression with the original kernel.

However, this is not limited to the Non-parametric version of Kernel Flows. Similar increase in MSE and MAE can be observed with the parametric versions of Kernel Flows. We consider the Boston Housing dataset, and apply kernel

flows with the Gaussian and Rational Quadratic kernels with the same setup as described above. The Gaussian kernel is initilazed to $\sigma = \sigma_{MSD}$ and the rational quadratic kernel is initialized with $\alpha = 0.3, \beta = 1.0$. The next table and figure illustrate the results.

Table 5.11: KF Parametric version for the Boston Housing dataset.

Method	MSE	MAE
Gaussian Kernel, $\sigma = \sigma_{MSD}$	30.892 (11.78)	4.978 (3.825)
KF Gaussian Kernel, $\sigma = \sigma_{MSD}$	31.565 (12.056)	3.817 (2.398)
Rational quadratic Kernel, $\alpha = 0.3, \beta = 1.0$	71.98 (0.0)	5.574 (0.0)
KF Rational quadratic Kernel, $\alpha = 0.3, \beta = 1.0$	74.363 (0.0)	5.743 (0.0)

The MSE and MAE go up, albeit by small amounts and ρ does not get minimized (see figures 5.4 and 5.3). A possible explanation is that both these kernels are already highly optimal and hence that Kernel Flows cannot meaningfully improve the kernel. This explanation seems to fit rational quadratic kernel (figure 5.3), where due to randomization of the batch/sample, the parameters oscillate. This is another example of the importance of the initialization: either the Kernel is already highly optimized (and hence cannot improve) or we are in a local minima where the algorithm is stuck.

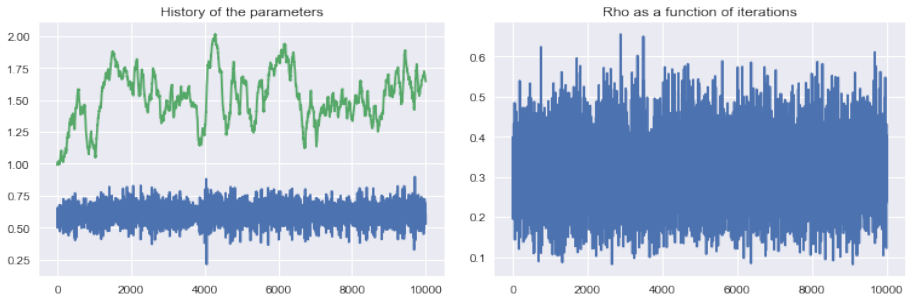


Figure 5.3: The parameter history (right) and ρ values (left) for the rational quadratic kernel.

However, for the Gaussian kernel (figure 5.4), the σ parameter continually increases from 144 to 151. It appears that the algorithm is continually moving the parameter in a direction, without improving ρ or the MSE/MAE. This

could be due to the flatness of the ρ function observed in Section 2.3: the algorithm pushes the parameters in direction of gradient descent without meaningful improvement to either ρ or the MSE/MAE.

Figure 5.4: The parameter history (right) and ρ values (left) for the Gaussianc kernel.

Chapter 6

Conclusion

In this report we presented the Kernel Flows algorithm and explored its properties. In particular, we showed how the performance of the algorithm heavily depends on ρ in the case of the Parametric version. We showed that this function was non-convex and that noisy observations flatten the function. To help remedy this problem we proposed to use a smaller batches and smaller sample sizes. We also showed how in the case of the Non-parametric version, the performance of the algorithm is heavily dependent on the chosen hyper-parameters. We then presented the theory of the RBF network and a proposed extension which can be trained using Kernel Flows Parametric.

Finally we applied the algorithm to standard datasets. On these datasets Kernel Flows Parametric generally performed the best, with the the proposals formulated in Chapter 2 generally improving performance. There were some notable exceptions where the performance worsened. We also showed that Kernel Flows was effective in training our modified RBF network, even though the overall performance of the network was in general poorer than the simpler Kernel Regression. Kernel Flows Non-Parametric however did not perform as well as Kernel Flows Parametric, which indicates the need for further research for its applicability to Regression problems. It is important to note that a slightly modified version of KF Non-parametric (presented in [12]) seems to be effective in *classification* problems.

Bibliography

[1] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. *Advances in Optimizing Recurrent Networks*. 2012. arXiv: 1212.0901 [cs.LG].

[2] Christopher Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006. ISBN: 0-387-31073-8.

[3] D.S. Broomhead and David Lowe. *Radial basis functions, multi-variable functional interpolation and adaptive networks*. RSRE, 1988.

[4] Paulo Cortez et al. "Modeling wine preferences by data mining from physicochemical properties". In: *Decision Support Systems* 47 (Nov. 2009), pp. 547–553. DOI: 10.1016/j.dss.2009.05.016.

[5] Matthieu Darcy. *Kernel Flows*. https://github.com/MatthieuDarcy/KernelFlows. 2020.

[6] Maclaurin Dougal et al. *Autograd*. https://github.com/HIPS/autograd. 2015.

[7] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[8] Szafraniec F.H. "The Reproducing Kernel Property and Its Space: The Basics". In: *Operator Theory*. Ed. by Alpay D. Basel: Springer, 2015.

[9] D. Harrison and D.L Rubinfeld. "Hedonic prices and the demand for clean air'". In: *J. Environ. Economics Management* 5 (1978), pp. 81–102.

[10] Charles Micchelli, Yuesheng Xu, and Haizhang Zhang. "Universal Kernels". In: *Mathematics* 7 (Dec. 2006).

[11] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.

[12] Houman Owhadi and Gene Ryan Yoo. "Kernel Flows: From learning kernels from data into the abyss". In: *Journal of Computational Physics* 389 (July 2019), pp. 22–47. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2019.03.040. URL: http://dx.doi.org/10.1016/j.jcp.2019.03.040.

[13] J. Park and I. W. Sandberg. "Universal Approximation Using Radial-Basis-Function". In: *Neural Computation* 3 (1991), pp. 246–257.

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[15] Bernhardand Schölkopf, Ralf Herbrich, and Alex J. Smola. "A Generalized Representer Theorem. Computational Learning Theory". In: *Lecture Notes in Computer Science* 2111 (2001), pp. 416–426.

[16] Friedhelm Schwenker, Hans A. Kestler, and Gunther Palm. "Three learning phases for radial-basis-function networks". In: *Neural Networks* 14 (2001), pp. 439–458.

[17] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2 (2011), p. 22.