Arjun Tapasvi

Scaling up Schrödinger Bridges to Image Generation

Computer Science Tripos – Part II

Christ's College

May 13, 2022

Declaration

I, Arjun Tapasvi of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed: *Arjun Tapasvi*

Date: *May 13, 2022*

Proforma

Candidate number:	**2437B**
Project Title:	**Scaling up Schrödinger Bridges to Image Generation**
Examination:	**Computer Science Tripos – Part II, 2022**
Word Count:	**11659**[1]
Line Count:	**1674**[2]
Project Originator:	**Francisco Vargas**
Supervisor:	**Francisco Vargas**

Original Aims of the Project

This project aimed to scale up a baseline Gaussian Process solution to the Schrödinger Bridge Problem (a widely-applicable ML task) to larger and higher-dimensionality datasets. The key aim was to implement randomised algorithms and subsampling approaches to reduce computational complexity, making computation on large datasets feasible. Extensions included implementing various techniques (like convolutional kernels) to improve result quality in high-dimensional datasets. The scaled-up algorithms would then be compared to the baseline in terms of quality, memory efficiency and runtime; the ultimate goal would be to generate novel MNIST images using learnt Schrödinger Bridges, a novel contribution to the field.

Work Completed

I successfully implemented multiple approximation algorithms, reducing the baseline implementation's time complexity from cubic to linear, and memory complexity from quadratic to linear. This resulted in a 1000× speedup (and 3× lower memory usage) on MNIST and toy datasets. I also implemented several improvements to algorithm *quality*, including exact and approximate convolutional kernels and a neural network approach. The end result improved significantly on the baseline, making GP Schrödinger Bridge computation feasible for large, high-dimensional datasets (like MNIST) for the first time. All core and most extension criteria were met, and I plan to co-author a paper publishing these results.

[1] Computed using `texcount`, including footnotes and tables. (`https://app.uio.no/ifi/texcount/`)
[2] Computed using `cloc` (`https://github.com/AlDanial/cloc`)

Special Difficulties

None.

Contents

List of Figures

Chapter 1

Introduction

Every day, our brains switch from one cognitive task to another hundreds of times. How can we model these transitions between states of the human brain, taking into account natural stochasticity of the dynamical neural system? It turns out that this fundamental problem in neuroscience– along with several others in statistics, medicine and the natural sciences (see section 1.1)– boils down to the Schrödinger Bridge Problem (SBP) [5]: finding the most probable stochastic evolution between two distributions with reference to a prior drift.

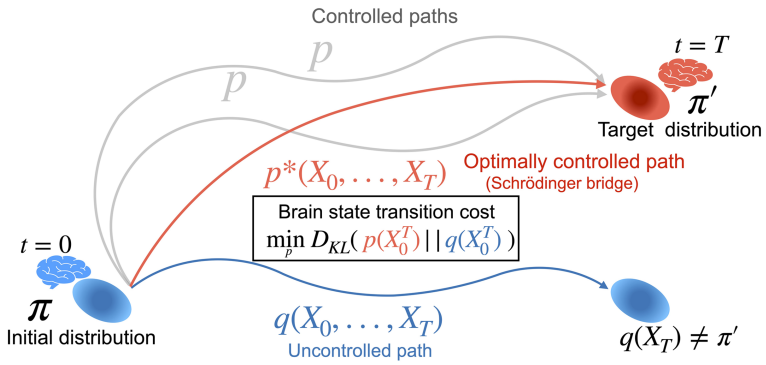


Figure 1.1: The Schrödinger Bridge Problem applied to neuroscience: finding the most likely transition between brain states [1]

The theoretical aspects of the SBP are well studied and understood, but the sample-based SBP (where we only have access to samples from the initial and final distributions) is an active area of research in the machine learning community [3, 6, 7, 8, 9, 10]. This sample-based approach is illustrated in figure 1.2: the Schrödinger Bridge algorithm finds the most likely evolution over timesteps from the `S` dataset to the `spiral` dataset [11], using a Brownian drift (random walk) prior.

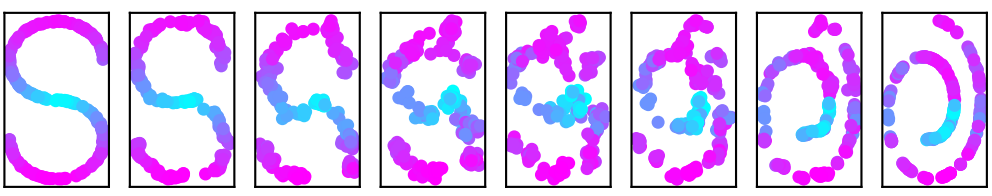


Figure 1.2: Evolution from `S` dataset to `spiral` dataset

This project builds on a specific approach [3] to solving the data-driven SBP, which uses Gaussian Processes (GPs) to fit the drift.[1] I scale up the baseline implementation– which suffers from computational infeasibility for large, high-dimensionality datasets– using randomised algorithms and sparse approximations, and implement various other optimizations (such as convolutional kernel structure) to enable high-quality Schrödinger Bridge computation for a dataset as large and high-dimensional as MNIST.

We can then learn the Schrödinger Bridge from the multivariate Gaussian distribution to the distribution of MNIST images: this allows us to generate novel MNIST-like images (i.e. images sampled from the MNIST distribution) from new Gaussian noise. This MNIST image generation is the key output of this project; MNIST provides a high-dimensional test bed for the scaling algorithms I implement, which can then naturally translate to the practical applications described in section 1.1.1.

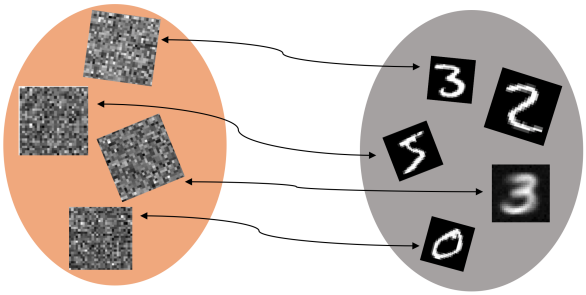


Figure 1.3: A sketch of the Schrödinger Bridge from noise to MNIST, and how we can use new noise to generate new images.

Overall, the aims (and achievements) of the project are two-fold:

1. *Scaling up*: the core aim is to scale up the baseline implementation in terms of runtime and memory usage (chiefly using random feature methods and sparse Gaussian processes, but also with performance optimizations like manual memory management), so computing Schrödinger Bridges for MNIST images is feasible;
2. *Improving quality*: extensions include various investigations to improve the quality of the computed bridge in high-dimensional use cases, such as introducing convolutional structure, implementing new kernels, and using heteroskedastic noise in the algorithm.

1.1 Motivation

This project focuses on a specific solution (scaling up the Gaussian process approach) to a general problem (the Schrödinger Bridge Problem). In this section, we first consider the applications of the SBP, then motivate the advantages of scaling up the GP approach to solving it.

[1] Details of the algorithm used in the baseline approach are given in section 2.4, because several technical preliminaries are required to formulate the problem and its algorithm. As a consequence, related work is also discussed in section 2.

1.1.1 Applications of SBP

A key application of the SBP is in high-dimensional optimal transport problems [12], where we find shortest paths between distributions subject to constraints. Optimal transport is useful in various sub-fields of statistics, for instance in aligning datasets and hypothesis testing [13, 14]; these techniques in turn find widespread applications in the natural sciences.

Another use case sets the initial and final distributions, π_0 and π_1, as two snapshots in time of a stochastic system: the Schrödinger Bridge can then be used to reason about the behaviour of the system in between these snapshots. This allows us to recover stochastic dynamics from sparse observations of a system, which can be useful for rapidly-evolving systems, or those where measurement is constrained (by cost or technology) to certain intervals. For instance, we can model the evolution over time of cell populations in biology [8], proteins folding in non-equilibrium thermodynamics [15, 16], and brain states transitioning [1] (see figure 1.1).

Finally, the SBP can be used for generative modelling: the bridge from random noise to a specific distribution (e.g. MNIST images) can be applied to perturbed noise to generate new samples from the target distribution. Such modelling has several uses in statistics– including density estimation, sampling in Bayesian inference, and the generation of anonymized synthetic datasets– and in inverse imaging problems, most notably in medical imaging [17, 18].

1.1.2 Gaussian processes vs neural network approach

There are two competing approaches to solving the SBP: one uses neural networks [6] to estimate the drift of a stochastic differential equation, while the other uses Gaussian processes [3]. The NN approach suffers from overfitting, a lack of uncertainty quantification, and a lack of theoretical guarantees: all traditional drawbacks of neural networks. For this reason, this project focuses on the Gaussian process (GP) approach, which leverages GPs' properties of robust and explainable results, theoretical guarantees, and the ability to interpolate sensibly between distributions [19].

The traditional issue with GPs [20] is the cubic complexity (in dataset size N) stemming from a matrix inversion, which limits the size of dataset GPs can be applied to. I mitigate this issue by implementing approximation methods that scale linearly in N: allowing the advantages of GP Schrödinger Bridges to manifest themselves in large and high-dimensional datasets. In doing so, I find that the approximation algorithms are significantly faster and less memory-intensive than the neural network approach as well, further strengthening the argument for the GP approach.

1.2 Overview of key contributions

The most notable result of this project is the generation of MNIST images using GP Schrödinger Bridges: this is a novel contribution to the field. The key implementation contributions which enable this include:

- A random features approximate kernel, with support for various kinds of features (including random Fourier features);
- A sparse GP framework for the SBP (using the Nyström method);

- Shallow and deep convolutional kernels, both exact and approximate, to capture patterns in high-dimensional data like images.

The first two of these are techniques for scaling up kernel methods to large datasets, while the last is a technique for improving bridge quality in high dimensions. This dissertation also describes various other implementation aspects, each of which improves either scaling performance or bridge quality.

Section 4 demonstrates practical speedups achieved by the approximation algorithms I implement ($\approx 1000\times$ faster than the baseline for $N = 1000$ datapoints), and confirms the asymptotic reduction in compute requirements (cubic to linear in time; quadratic to linear in memory).

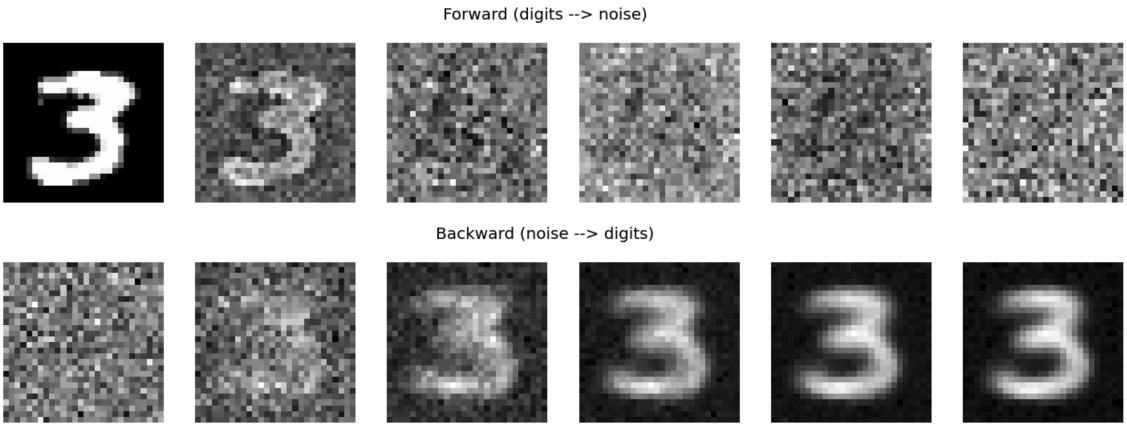


Figure 1.4: The generation of an MNIST-like 3 from noise using the learnt Schrödinger Bridge.

1.3 Report outline

Having introduced and motivated the problem in section 1, I move on in section 2 to detailing project requirements and software engineering tools, before reviewing the technical background (discussing relevant concepts, current approaches and related work). I describe the key algorithms and techniques implemented in section 3, and evaluate their performance in section 4. In section 5, I conclude and discuss future work.

Chapter 2

Preparation

In this section, I first describe the project requirements and some key planning and software engineering details. I then review some necessary mathematical, algorithmic and machine learning preliminaries to build the theory required for the approaches I implement. Finally, I define the Schrödinger Bridge problem and summarize the current state-of-the-art solutions.

2.1 Requirements analysis

This section details the project's evaluation criteria. These are fundamentally identical to the criteria stated in the proposal (appendix C), but build on them by adding extension criteria which were not initially considered.[1]

The core criteria are:

1. With sparse GP or random Fourier features approaches, obtain similar results to the exact GP approach on 2D toy examples– determined by a distributional distance metric;
2. Confirm that the approximation methods lead to faster Schrödinger Bridge computation;
3. Confirm that the approximation methods lead to lower memory usage during Schrödinger Bridge computation;
4. Evaluate whether MNIST images generated from the IPFP algorithm are close to original MNIST images;
5. Compare random Fourier features and sparse GP benchmarks to argue which is more effective.

Extension criteria include:

1. Consider more mathematically advanced algorithms like variational Fourier features;
2. Refactor baseline codebase to improve class hierarchy and code structure;
3. Compare the performance of different kernels;
4. Implement other techniques to improve bridge quality, and investigate how they help;

[1]Because the project has a significant research element, it was difficult to enumerate all of the extension criteria in October; the proposal acknowledged that extension criteria may be added as the project progressed.

5. Implement and investigate convolutional kernels for dealing with high-dimensional data;
6. Implement the neural network approach from [6] for comparison to the GP approach;
7. Generate more complex images, from fashion-MNIST and CIFAR-10.

Overall, the main aim is to scale up the GP approach to the SBP, so computation for large datasets is feasible. Extensions include more investigative, research-oriented questions regarding the *quality* of learnt bridges in high-dimensional space.

2.2 Starting point

2.2.1 Relevant courses and experience

Although part IB Data Science and AI provided familiarity with the basics of machine learning, I had no prior familiarity with stochastic differential equations, Gaussian processes, randomised algorithms or implementing neural networks. Understanding the theory– particularly around SDEs, which are usually covered in graduate-level courses– was therefore a significant and challenging part of my preparation. Part II courses ML+BI, Deep Neural Networks and Randomised Algorithms cover some aspects of these topics, but I had to learn the material before these courses began, and in more detail than the courses provide. IB Complexity Theory helped with deriving asymptotic complexities of algorithms; and IB Computation Theory helped in understanding low-level performance optimizations (e.g. caching and GPU memory layouts).

2.2.2 Existing codebase

This project builds on the codebase of Vargas et. al. [3], which had a working implementation of the IPFP algorithm (including the SDE solver, IPFP loop, and DriftFit function using GP regression).[2] I implemented everything else, including sparse GP and random features classes, a new kernel, code for heteroskedastic noise, convolutional kernels, the neural network approach, the evaluation framework, and performance and memory optimizations.

2.3 Software engineering tools and techniques

2.3.1 Programming language and libraries

Python is a natural choice for this project, due to its strong library support for numerical computing and visualisation (and the fact that the baseline implementation uses it). The main libraries I use are shown in table 2.1; I also use others like `scipy` [21] and `tqdm`, and several inbuilt modules including `gc` (for manual garbage collection) and `pickle` (for saving results of experiments). A lot of experimentation was done in Jupyter notebooks for rapid visualisation, iteration and development.

[2]These concepts are discussed in detail in section 2.5.1

Library	Familiarity	Purpose
`pytorch`	None	Numerical computing; machine learning; GPU interfacing; benchmarking and profiling
`pyro`	None	Gaussian processes
`numpy`	Moderate	Numerical computing
`matplotlib`	Moderate	Visualisation

Table 2.1: Key libraries used and their purposes

2.3.2 Backup and version control

Throughout the project, I maintained a Git repository, which I frequently committed changes to. I used pull requests to discuss code with my supervisor and for feature tracking, and created branches for significant feature additions. I backed up code and my notes/research to both Google Drive and a personal hard disk fortnightly to protect against unexpected hardware failures; I also had a copy of all my code both locally and on my supervisor's VM GPU.

2.3.3 Testing

This project is different to a regular software project, so required a different form of testing. Early on, I wrote a suite of metrics and plots (including both quality metrics and performance benchmarks) to evaluate a given setup; running these whenever I made a change acted as a form of regression testing to ensure nothing had broken.

2.3.4 Software engineering practices

Throughout the project, I tried to follow good software engineering practices. This included descriptive variable naming, thorough docstrings with type annotations (particularly useful for Python, where typing is not statically checked), and assertions that shapes of PyTorch objects were as expected. I created a virtual environment for tracking module dependencies, and wrote classes with extensibility in mind throughout the implementation. Finally, I invested time early on writing useful tools for logging and scripts supporting command-line experimentation, which allowed for rapid iteration and development.

2.3.5 Computational resources

I initially used my laptop for running experiments, but soon found it was infeasible (even small experiments took >10 hours to run). From January, I had access to my supervisor's VM GPU– an NVIDIA Quadro RTX 8000 with 48.6 GiB of memory.

2.4 Technical preliminaries

This project requires an understanding of stochastic differential equations (SDEs), GPs, convolutions, and random approximations: I now review the essentials of these concepts.

2.4.1 Stochastic processes and SDEs

A stochastic process is effectively a random variable with a time-dependent distribution. Stochastic processes are often denoted as $\{\boldsymbol{X}_t\}_{t\in T}$: each $\boldsymbol{X}_t$ is the random variable at time t, with a specific distribution. The precise definition 2.4.1 uses the probability space formalism; in practice, we regard the sample space ω argument as implicit.

Definition 2.4.1. [3] For a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, a stochastic process is a collection of time-dependent random variables $\boldsymbol{X}_t(\omega) : T \times \Omega \to \mathbb{R}$ (indexed by T), which we can write as

$$\{\boldsymbol{X}_t(\omega) : t \in T\}.$$

A stochastic process of particular interest in this problem is the Wiener Process, or Brownian motion. The Wiener Process is defined as the stochastic process $\boldsymbol{W}_s$ satisfying four key properties [22]:

1. $\boldsymbol{W}_0 = \mathbf{0}$;
2. $\boldsymbol{W}_{t_2} - \boldsymbol{W}_{t_1} \perp\!\!\!\perp \boldsymbol{W}_{t_1} - \boldsymbol{W}_{t_0} \forall t_0 < t_1 < t_2$ (increments are independent);
3. $\boldsymbol{W}_{t_2} - \boldsymbol{W}_{t_1} \sim \mathcal{N}(\mathbf{0}, (t_2 - t_1)\mathbb{I}_d) \quad \forall s < t$ (increments are Gaussian);
4. $\boldsymbol{W}_t$ is continuous in t.

It can be regarded as the infinitesimal extension of a discrete random walk in d dimensions, where each increment in unit time is given by a multidimensional Gaussian:

$$\mathbf{w}_{t_{n+1}} = \mathbf{w}_{t_n} + \sqrt{(t_{n+1} - t_n)}\mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbb{I}_d).$$

Importantly for our use case, white noise $\boldsymbol{\epsilon}(t) \sim \mathcal{N}(\mathbf{0}, (t_2 - t_1)\mathbb{I}_d)$ can be seen (in some non-rigorous sense) as the derivative of the Wiener process.

We now move on to considering the Itô process, which forms our definition of a stochastic differential equation (SDE).

Definition 2.4.2. For stochastic processes $\mathbf{b}_t$ and $\boldsymbol{\sigma}_t$, the Itô process $\boldsymbol{X}_t$ is defined as

$$\boldsymbol{X}_t = \boldsymbol{X}_0 + \int_0^t \mathbf{b}_s ds + \int_0^t \boldsymbol{\sigma}_s \mathrm{d}\boldsymbol{W}_s. \tag{2.1}$$

Equation 2.1 is often notationally simplified to

$$\mathrm{d}\boldsymbol{X}_t = \mathbf{b}_t \mathrm{d}t + \boldsymbol{\sigma}_t \mathrm{d}\boldsymbol{W}_t. \tag{2.2}$$

The process $\mathbf{b}_t$ is often referred to as the drift, and $\boldsymbol{\sigma}_t$ as the volatility. It is common to refer to equation 2.2 as an SDE,[3] since "dividing" both sides by $\mathrm{d}t$ yields

$$\frac{\mathrm{d}\boldsymbol{X}_t}{\mathrm{d}t} = \mathbf{b}_t + \boldsymbol{\sigma}_t \boldsymbol{\epsilon}_t. \tag{2.3}$$

[3]Although this is a common notation, it is not rigorous, as most stochastic processes (including the Wiener Process) are not differentiable

Crucially, equation 2.2 describes a process with a drift $\mathbf{b}$ over time and some noise $\boldsymbol{\sigma}$: we will see that these are both parameters to the algorithm for solving the SBP.

2.4.2 Euler-Maruyama discretization

To deal numerically with SDE computations, we need to define a time-discretization; here, we use the Euler-Maruyama method. This is an extension of the Euler method for ordinary differential equations (ODEs); see algorithm 1 for details of the steps involved. Effectively, we partition the interval into equal sections, set the initial conditions, and then define the solution recursively by adding $\boldsymbol{W}_t \Delta t$ (the drift per timestep) and $\boldsymbol{\sigma}_t \Delta \boldsymbol{W}_t$ (the scaled white noise) at each timestep. This fits naturally with our understanding of equation 2.2.

Algorithm 1: Euler-Maruyama Discretization

input: $[t_0, t], \quad p(\boldsymbol{x}_{t_0}), \quad \Delta t, \quad \mathrm{d}\boldsymbol{X}_t = \mathbf{b}(\boldsymbol{X}_t, t)\mathrm{d}t + \boldsymbol{\sigma}(\boldsymbol{X}_t, t)\mathrm{d}\boldsymbol{W}_t$

Divide the interval$[t_0, t]$ into steps of size Δt: $t_0,\ t_0 + \Delta t,\ \ldots,\ t_0 + k\Delta t,\ \ldots,\ t$

$\boldsymbol{X}_{t_0} \sim p(\boldsymbol{x}_{t_0})$

for *each step* k **do**

  $\Delta \boldsymbol{W}_{t_k} \sim \mathcal{N}(\mathbf{0}, \Delta t \mathbb{I}_d)$

  $\boldsymbol{X}_{t_{k+1}} = \boldsymbol{X}_{t_k} + \boldsymbol{b}(\boldsymbol{X}_{t_k}, t_k)\Delta t + \boldsymbol{\sigma}(\boldsymbol{X}_{t_k}, t_k)\Delta \boldsymbol{W}_{t_k}$

end

return $\{\mathbf{x}(t_k)\}_k$

2.4.3 Neural networks and convolution

Neural networks are the standard workhorses of machine learning. They compute functions by propagating values through connected layers of "neurons" depending on weights and biases at each layer, and can be trained using backpropagation as described in part IB Artificial Intelligence.[4] Here, I briefly describe convolutions, which play a large part in both neural networks and some of the Gaussian process kernels I later implement.

The key idea of a convolution is to constrain a fully connected neural network by capturing spatially local information (particularly patterns in images). A *kernel*, or *filter*, is a patch of $n \times n$ weights $\mathbf{W}$ which slides over the image as shown in figure 2.1: at each location, the output is the dot product of the filter with the corresponding image patch, plus a bias term. We often add an activation function such as the Rectified Linear Unit (ReLU) as well, so $\mathbf{y}_i = \mathrm{ReLU}(\mathbf{W}\mathbf{x}_{\text{patch i}} + b)$. Note that the same kernel weights are used at every location of the images: this reduces the number of trainable parameters in an idea known as *parameter sharing*, which improves training efficiency and adds inductive bias (constraining the model to certain solution spaces).

A critical concept related to convolutions is that of *channels*, which form the "depth" dimension of images (for example, an RGB image has 3 channels). Convolutional kernels also have channels– imagine stacked 2D kernels with a different weight matrix per layer, as figure 2.1 illustrates– so the output of a convolution can have a different number of channels to the input. Typically, convolutions use the combination of channels and different filter sizes to capture various kinds of structure in the input.

[4] Covered in part IB of CST, so considered assumed knowledge here.

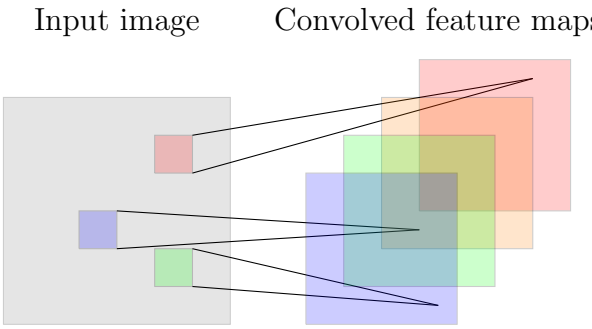


Figure 2.1: Illustration of a convolution with $\text{channels}_{\text{in}} = 1$, $\text{channels}_{\text{out}} = 4$

Other technicalities include *stride* (how many elements to slide the kernel along at each step) and *padding* (whether to add zeros on the borders to retain input dimension). Finally, *pooling* aggregates the input, reducing dimensionality: for instance, a 2×2 max-pooling takes the maximum of each 4-element patch in a sliding window.

Overall, convolutions are very useful for capturing local structure in high-dimensional data (particularly images); although most common in NNs, they can also help with Gaussian process kernels.

2.4.4 Gaussian Processes and GP regression

Gaussian process modelling is a clean and powerful non-parametric Bayesian technique for dealing with regression tasks. It involves taking priors over functions, and updating these in the light of data: this provides robust estimation along with uncertainty quantification. A Gaussian process is an infinite-dimensional generalization of the multivariate Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$;[5] specifically, a GP is a collection of random variables such that any finite subset of them is joint Gaussian distributed.

Definition 2.4.3. [23] A time-continuous stochastic process $\{X_t; t \in T\}$ is Gaussian if and only if for every finite set of indices $t_1, \ldots, t_k$ in the index set T

$$\mathbf{X}_{t_1,\ldots,t_k} = (X_{t_1}, \ldots, X_{t_k})$$

is a multivariate Gaussian random variable.

A Gaussian process is parameterized by a mean function $\mu : \mathbb{R}^n \to \mathbb{R}$ and a covariance function $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$: the mean function yields the expected value of the GP at an input point, while the covariance function yields the covariance between two input points (which typically decreases with distance). In practice, we often assume zero mean in GPs: $\mu = \lambda \mathbf{x}.0$; and we often refer to the *Gram matrix* $\mathbf{K}_{X,X}$ as the *kernel*.[6] The Gram (or covariance) matrix contains the value of the covariance function computed on all pairs of datapoints in the dataset $\mathbf{X}$: $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$.

[5] Recall that the density of the multivariate Gaussian distribution is given by

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k |\boldsymbol{\Sigma}|}}$$

[6] $\mathbf{K}_{X,X}$ is a notational shorthand for $\mathbf{K}(\mathbf{X}, \mathbf{X})$

Due to space constraints, the derivation of Gaussian process regression is omitted; in short, given training data of length N with features $\mathbf{X}$ and targets $\mathbf{y}$, the prediction for a new set of points $\mathbf{x}$ is given by equation 2.4. The computational bottleneck is the inversion of $\mathbf{K}_{\mathbf{X},\mathbf{X}}$, which has complexity $\mathcal{O}(N^3)$.

$$f(\mathbf{x}) = \mu(\mathbf{x}) + \mathbf{K}_{\mathbf{x},\mathbf{X}}(\mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma^2\mathbf{I})^{-1}\mathbf{y} \tag{2.4}$$

The choice of kernel k defines the prior imposed on functions: figure 2.2 visualises this effect for some common kernels. In this dissertation, we will mainly consider the exponential and radial basis function (RBF) kernels defined in equations 2.5 and 2.6 respectively. These kernels themselves have parameters: the variance σ^2 determines the average distance from the mean μ of the function, while the lengthscale l determines the reach of influence on neighbours [24].

$$k_{\text{RBF}}(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp\left(\frac{\|(\mathbf{x} - \mathbf{x}')^2\|}{2l^2}\right) \tag{2.5}$$

$$k_{\text{Exp}}(\mathbf{x}, \mathbf{x}') = \sigma^2 \exp\left(\frac{\|(\mathbf{x} - \mathbf{x}')\|}{l}\right) \tag{2.6}$$

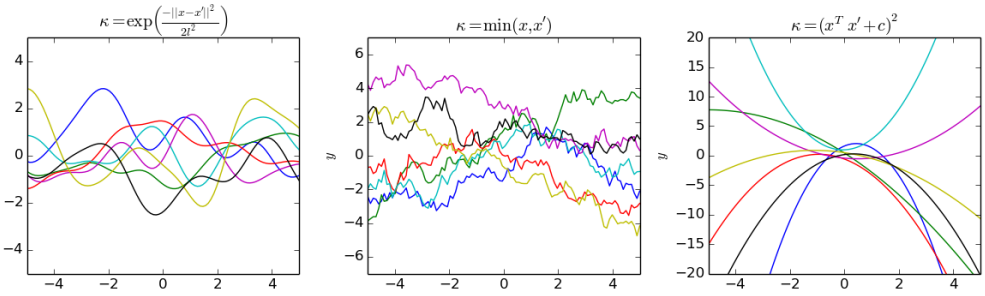


Figure 2.2: The effect RBF (left), Brownian (middle), Quadratic (right) kernels on priors [2]

2.4.5 Randomised approximation algorithms

Randomised approximation algorithms aim to approximate some potentially complicated function using randomly sampled features. We sample from specific distributions to compute an expectation (often an integral, like in Monte Carlo integration) which– in the limit of samples– tends to our desired function. These methods provide guarantees through mathematical proofs (e.g. using the Central Limit Theorem) of convergence behaviour as the number of samples F increases, and in practice can be effective with a surprisingly small number of features. Figure 2.3 visualises convergence of random features for a simple underlying function.

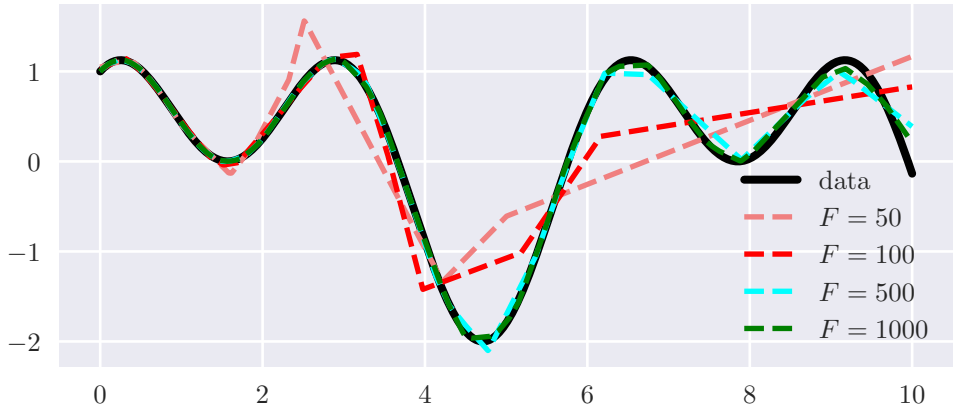


Figure 2.3: Convergence of random features approach to true function with increasing F

2.5 The Schrödinger Bridge Problem

Having covered these preliminaries, we are now able to formally define the SBP.

Definition 2.5.1 (Schrödinger Bridge Problem)**.** [3] Let $\mathbb{Q}^0$ be the distribution induced by the stochastic process

$$\mathrm{d}\mathbf{X}_t = \boldsymbol{b}(\mathbf{X}_t, t)\mathrm{d}t + \sqrt{\gamma}\mathrm{d}\mathbf{W}_t, \quad \mathbf{X}_0 \sim \pi_0 \tag{2.7}$$

The Schrödinger bridge distribution is given by

$$\mathbb{Q}^* = \underset{\mathbb{Q}\in\mathcal{D}(\pi_0,\pi_1)}{\arg\inf} \; D_{\mathrm{KL}}(\mathbb{Q}\|\mathbb{Q}_0) \tag{2.8}$$

where $\mathcal{D}(\pi_0, \pi_1) = \{\mathbb{Q} : (\mathbf{X}_0)_{\#}\mathbb{Q} = \pi_0,\ (\mathbf{X}_1)_{\#}\mathbb{Q} = \pi_1\}$ is the set of SDE distributions with π_0, π_1 as marginal distributions, $\mathbb{Q}_0$ is the prior distribution induced by the reference process, $(\mathbf{X}_t)_{\#}\mathbb{Q}$ denotes the marginal distribution of $\mathbb{Q}$ at time t, and $D_{\mathrm{KL}}(Q\|P) = \mathbb{E}_Q\left[\log \frac{dQ}{dP}\right]$ represents the Kullback-Leibler (KL) divergence [25] between two distributions.[7]

As described in section 1, there is an intuitive interpretation: the Schrödinger Bridge is the most likely evolution between two distributions, given a reference process (e.g. Brownian drift).

2.5.1 Current approaches

The main approach used to solve the SBP is called the Iterative Proportional Fitting Procedure (IPFP) [26]. The idea was originally developed by Fortet [27] and refined by Kullback [28]; the version given here is from [3]. It uses three subroutines:

- `SDESolve(drift, σ, π0, Δt)` uses the EM discretization of section 2.4.2 to solve an SDE. The result is a trajectory with the given `drift`, starting from distribution π_0, with noise σ and timestep Δt.

[7] Here, $\frac{dQ}{dP}$ is the Radon-Nikodym derivative, an idea from measure theory; it simply denotes the ratio between two distributions. If probability densities p_Q and p_p are defined, the RN derivative is given by $\frac{p_Q}{p_P}$.

- `DriftFit(timeseries_list, σ)` uses Gaussian processes to estimate the drift of the timeseries in `timeseries_list`. It uses the maximum likelihood autoregressive objective [29, 30] below to do this, which stems from dividing the SDE equation 2.3 by Δt.

$$\frac{\mathbf{X}_{t+1} - \mathbf{X}_t}{\Delta t} = f(\mathbf{X}_t) + \frac{\gamma_t \boldsymbol{\epsilon}}{\sqrt{\Delta t}} \tag{2.9}$$

- `Reverse(trajectory)` simply reverses a trajectory in the time dimension.

We can now understand the IPFP (algorithm 2). It takes as input the two terminal distributions and the prior drift, and initially sets the estimated drift to the prior. It then iterates until convergence, at each step setting one marginal constraint, solving an SDE from there, and estimating the drift of the reversed trajectory (for both the forward and backward process). Note that we symmetrically learn both a forward process ($\pi_0 \to \pi_1$) and a backward process ($\pi_1 \to \pi_0$). The IPFP algorithm is illustrated in figure 2.4: we start from the right hand side of the diagram, and each loop of the algorithm is a red then a blue dotted line.

Algorithm 2: IPML [3]

input: $\pi_0(\boldsymbol{x}), \pi_1(\boldsymbol{y}), \mathbb{Q}_0^\gamma$

1 Initialise: $i := 0$ $\mathbb{Q}_0^* := \mathbb{Q}_0^\gamma$

2 $\bar{\boldsymbol{b}}^-_{\mathbb{Q}_0}(\boldsymbol{x}, t) :=$ ObtainBackwardDrift$(\mathbb{Q}_0^*)$

3 **repeat**

4 $\quad i := i + 1$

5 $\quad \left\{\boldsymbol{x}^{(m)-}_{[1:T]}\right\}_m :=$SDESolve$\left(-\bar{\boldsymbol{b}}^-_{\mathbb{Q}_{i-1}}, \gamma, \pi_1, \Delta t\right)$

6 $\quad \bar{\boldsymbol{b}}^+_{\mathbb{P}_i} :=$ DriftFit$\left(\text{Reverse}\left(\left\{\boldsymbol{x}^{(m)-}_{[1:T]}\right\}_m\right), \frac{\gamma}{\Delta t}\right)$

7 $\quad \left\{\boldsymbol{x}^{(n)+}_{[1:T]}\right\}_n :=$ SDESolve$\left(\bar{\boldsymbol{b}}^+_{\mathbb{P}_i}, \gamma, \pi_0, \Delta t\right)$

8 $\quad \bar{\boldsymbol{b}}^-_{\mathbb{Q}_i} :=$ DriftFit$\left(\text{Reverse}\left(\left\{\boldsymbol{x}^{(n)+}_{[1:T]}\right\}_n\right), \frac{\gamma}{\Delta t}\right)$

9 **until** *convergence*;

10 **return** $\bar{\boldsymbol{b}}^-_{\mathbb{Q}_i}, \bar{\boldsymbol{b}}^+_{\mathbb{P}_i}$

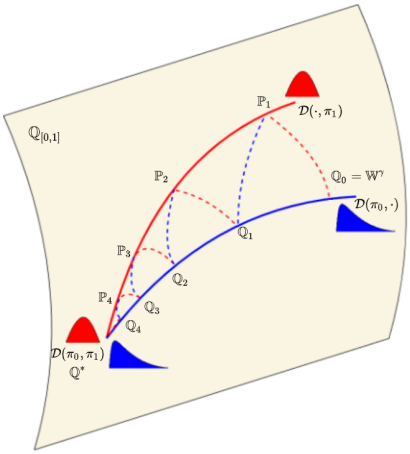


Figure 2.4: A visualisation of the IPFP algorithm from [3]

The bottleneck in this algorithm is `DriftFit` in lines 6 and 8, which takes as input timeseries of shape $N \times T \times d$ (N is the number of datapoints, T is the number of timesteps, d is the dimension of each datapoint). Currently, `DriftFit` flattens this into a $NT \times d$ array $\mathbf{X}$ and performs GP regression on this: the complexity is thus $\mathcal{O}((NT)^3)$. Speeding up this function is the main focus of this project, although I also consider changes to other parts of the algorithm (like `SDESolve`).

2.5.2 Related work

This project builds directly on the GP approach in Vargas et. al [3], which uses a maximum likelihood drift fitting objective [29, 30]. De Bortoli et. al [6] solve the same problem using a neural network to fit drift instead of GP, using ideas from score matching [31] instead of maximum likelihood. Both build upon sample-based SBP ideas from [32].

Chapter 3

Implementation

In this chapter, I provide a repository overview sketching the high-level functionality of the codebase, before describing the key implementation aspects of the project. These implementation aspects– along with an explanation of which key project aim they address– are summarised in table 3. *(Recall that there are two key aims: to computationally scale up the baseline approach, and to improve quality of computed bridges.)*

Concept	Section	Purpose
Performance optimization	§3.2	*Scaling up:* how I used profiling and memory debugging throughout the project to write performant code.
Quality improvements to baseline	§3.3	*Improving quality:* some isolated changes, including an important kernel implementation, which improve quality of results.
Sparse GPs and random features	§3.4 – 3.5	*Scaling up:* The key algorithms used to achieve the $\mathcal{O}(N^3) \rightarrow \mathcal{O}(N)$ speedup and $\mathcal{O}(N^2) \rightarrow \mathcal{O}(N)$ memory improvement.
Convolutional kernels	§3.6	*Improving quality and scaling up:* convolutional kernels improve quality for high-dimensional data like images; combining these with the random features algorithms then improves performance. I also implement a CNN formulation of the GP approach.

Table 3.1: Summary of implementation

A significant component of the project was developing and applying relevant theories in a way that was well-suited to this problem; for this reason, some of the *Implementation* subsections contain theory I had to develop along the way.

3.1 Repository overview

A subset of my directory tree is shown below, with a brief explanation of each module and some important scripts. The only scripts I used from the original codebase are `MLE_drift.py`,

`sde_solver.py` and `GP.py` (all in `sbp_ipfp/`), and I ended up significantly modifying/adding to each of these; all other scripts are entirely my own work. The repository is organised for clear modularity: for instance, the `sbp_ipfp` module contains a fully working IPFP implementation, which can then be run using kernels from the `kernels/` module or plotted using the `eval` module. Such modularity precludes import cycle errors, enforces independence of functionality, and yields an extensible API – e.g. to run IPFP with a new kernel, one simply has to add a script to the `kernels` folder (without needing to understand any other code, or worry about disrupting the internal consistency of the codebase).

```
GP_schrodinger_bridge/
├─ sbp_ipfp/ ..........  Self-contained SBP solution using IPFP
│  ├─ MLE_drift.py.....  Main IPFP loop, calling SDE solver and DriftFit
│  ├─ random_features.py..  RandomFeatures abstract class + implementations
│  ├─ unet.py..........  Neural network definition + training loop
│  └─ GP.py............  Exact and sparse GP regression classes
├─ kernels/..........  New kernels implementations, all subclassing pyro's Kernel
│  ├─ arccos.py........  Arc-cosine kernel implementation
│  └─ conv_kernel.py...  Exact + approximate convolutional kernels
├─ eval/.............  Plotting, metrics, and benchmarking scripts
├─ utils/............  Profiling and memory debugging utilities
└─ examples/.........  Sample usage of some implementation aspects, e.g. IPFP
```

3.2 Performance optimization

Some of the key technical difficulty of this project related to debugging and improving memory- and compute-intensive computations on a GPU in Python. In this section, I briefly outline some of the tools and techniques I used throughout the project– particularly when dealing with high-dimensional MNIST images– to ensure efficiency.

3.2.1 Memory management

Out-of-memory errors were a common occurrence, particularly when dealing with the MNIST dataset. It is crucial to understand the PyTorch computation graph model to debug these: the graph keeps track of all tensors[1] and their dependencies, in case they are needed for backpropagation. Calling `.detach()` manually removes tensors from the computation graph, and the `with no_grad()` context manager disables gradient computation altogether (which is what we want in all cases except the CNN approach in section 3.6.4). Memory is further complicated by the fact that tensor objects live in CPU memory, and are stubs pointing to data on the

[1]Tensors are effectively n-dimensional matrices; they are the fundamental data structures in PyTorch.

GPU. To debug memory issues, I implemented several functions, such as the one whose output is shown in figure 3.1 (which shows the amount of memory occupied by tensors of each shape). Implementing these required interfacing with PyTorch's GPU memory primitives and `gc`'s CPU memory primitives, to inspect currently-tracked variables and their sizes.

Mem / GiB	Shape	NumTensors
46.055	(1000, 680)	8466
37.0	(1000, 1000)	4625
0.082	(680, 1000)	15
0.081	(20, 1000)	505
0.049	(1000, 1)	6114

Figure 3.1: The output of a memory-debugging function I wrote

As an example of the kind of memory optimizations that were common, consider adding a scalar σ to each element of square matrix $\boldsymbol{A}$ of size $n \times n$. Instead of using the naive `A = A + σ * torch.identity(n, n)`, wastefully allocating another n^2 elements, we can use `A.view(-1)[::n+1] += σ`: this accesses the underlying contiguous memory of $\boldsymbol{A}$ and adds a constant to every $(n+1)^{\text{th}}$ element, and is much more time and memory efficient than other approaches. Another example of memory optimization involved inspecting and deleting variables[2] when they were no longer needed, and calling Python's garbage collector using `gc.collect()` to force cleaning up of redundant values.

3.2.2 Runtime profiling

I also used runtime profiling extensively to find bottlenecks: these could then often be solved by manipulating the PyTorch computational graph or parallelising computation. For instance, the profile result below confirms that in the baseline implementation, the key bottleneck is the matrix inversion (the third line), which takes 1.732s on average.

Name	CPU total %	CPU time avg	Self CUDA %	CUDA total	CUDA time avg	# of Calls	Input Shapes
aten::mm	0.05%	56.199us	52.31%	5.859s	43.083ms	136	[[200, 6800], [6800, 6800]]
volta_dgemm_64x64_tt	0.00%	0.000us	52.31%	5.859s	43.082ms	136	[]
aten::_linalg_inv_out_helper_	9.65%	520.061ms	46.33%	5.195s	1.732s	3	[[6800, 6800], [], []]
volta_dgemm_128x64_nn	0.00%	0.000us	30.83%	3.453s	67.696ms	51	[]
volta_dgemm_64x64_nn	0.00%	0.000us	10.61%	1.189s	4.953ms	240	[]

Finally, I used PyTorch's built-in benchmarking facilities to robustly evaluate each individual change I made. If a change negatively impacted performance, it could then be analysed with the profiler as above.

3.3 Improvements to baseline

This section describes some of the isolated changes I made to the repository to improve the quality of learnt Schrödinger bridges: each was a combination of developing the relevant theory and implementing it in an efficient manner.

[2]Python uses reference counting, which deallocates memory when a variable is no longer referenced

3.3.1 Time-dependent heteroskedastic noise

Recall that in the IPFP algorithm, we iteratively solve an SDE (with `SDESolve`) and fit a reverse drift (with `DriftFit`). The noise parameter σ is used in both stages: to determine the magnitude of white noise in the EM algorithm in the former, and in the GP regression formula in the latter. There is a clear tradeoff in choice of σ: too high a σ leads to noisy results at the endpoints, while too low a σ prevents the SDE from effectively exploring the search space to find the target distribution. These cases are illustrated in the first two plots of figure 3.2. Ideally, we want to hone in on precise distributions in the final timesteps, while still searching a large state-space in the intermediate timesteps: this suggests the natural algorithm of varying σ with timestep. I implement a "wedge" as shown in the third plot (recall that we reverse the drift, so we need a small σ on both extremes of timestep); at t_0 and t_T, σ is small, while at $t_{T/2}$ it is large. This results in extremely high quality fits, as shown in the fourth plot below.

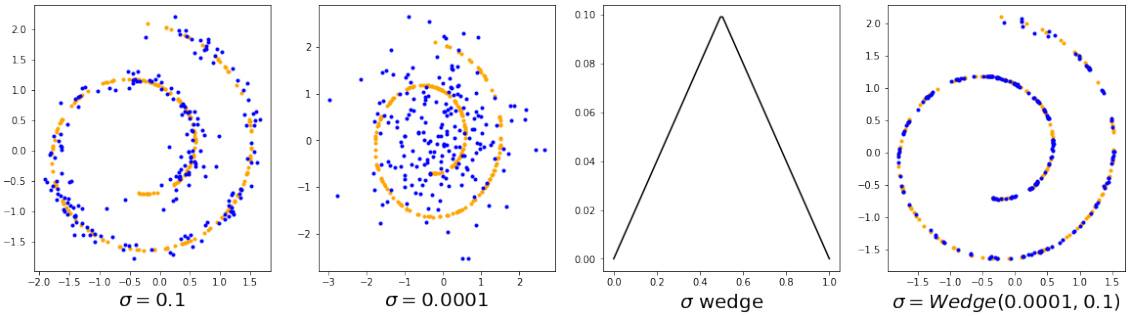


Figure 3.2: The effect of σ on bridge quality

This time-varying noise now violates the GP regression assumption of homoskedasticity (constant noise); fortunately, we can make a simple alteration to the GP regression formula to account for heteroskedastic noise. We replace the scalar matrix $\sigma^2\mathbf{I}$ by a matrix $\mathbf{M}$ whose diagonal elements give the noise for each datapoint; (as implemented in section 3.2.1).

$$f(\mathbf{x}) = \mu(\mathbf{x}) + \mathbf{K}_{\mathbf{x},\mathbf{X}}(\mathbf{K}_{\mathbf{X},\mathbf{X}} + \mathbf{M})^{-1}\mathbf{y}, \text{ where } \mathbf{M}_{ij} = \begin{cases} \sigma_i^2, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

3.3.2 OU-process prior

So far, we have only considered Brownian drift (i.e. the Wiener process) as the SBP prior. However, this zero-drift process can struggle with the curse of dimensionality: searching a state space becomes increasingly difficult in higher dimensions. Can we instead use a stochastic process with certain properties that help us to converge quickly to a specific distribution? One such process is the Ornstein-Uhlenbeck process [33], a canonical example of a mean-reverting stochastic process with equation

$$\mathrm{d}\boldsymbol{X}_t = -\theta \boldsymbol{X}_t \mathrm{d}t + \sigma \mathrm{d}\boldsymbol{W}_t. \tag{3.2}$$

Analysis of the OU process [34] shows that whatever the initial distribution π_0, it tends to the Gaussian distribution $\mathcal{N}(0, \frac{\sigma^2}{2\theta})$ as $T \to \infty$. We can use this to our advantage by "flipping"

the standard way of looking at the SBP: setting π_1 as Gaussian noise and π_0 as the dataset of interest (instead of vice versa).[3] We can now leverage the mean-reverting property of the OU process (informing our choice of θ by the distribution we want to converge to); we will see (in the evaluation section) that this property gives us significantly better results than Brownian motion can for MNIST.

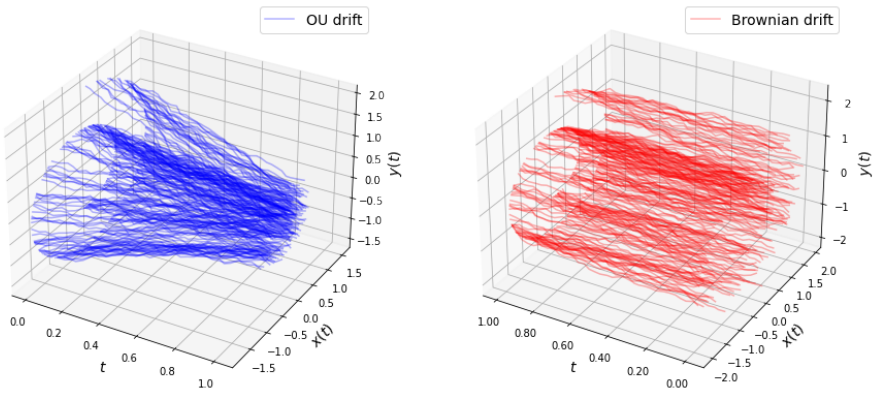


Figure 3.3: Comparing OU (left) and Brownian (right) drift

3.3.3 Arc-cosine kernel

As we saw in section 2.4.4, we are free to choose a kernel to fit the GP with; we now consider a family of kernels introduced in [35], called the arc-cosine kernels. These are closely related to neural network computation, and are useful in implementing the convolutional kernels in [36] that we will discuss in section 3.6. The family of kernels (one for each positive integer n) defines similarity between two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ as:

$$k_n(\mathbf{x}, \mathbf{y}) = 2 \int \frac{e^{-\frac{\|\mathbf{w}\|^2}{2}}}{(2\pi)^{d/2}} \Theta(\mathbf{w} \cdot \mathbf{x}) \Theta(\mathbf{w} \cdot \mathbf{y}) (\mathbf{w} \cdot \mathbf{x})^n (\mathbf{w} \cdot \mathbf{y})^n \mathbf{dw} \tag{3.3}$$

This integral can be analytically solved for different values of n; for brevity, I omit the derivation here. In this project, we choose the kernel for $n = 1$, which is given by

$$k_n(\mathbf{x}, \mathbf{y}) = \frac{1}{\pi} \|\mathbf{x}\| \|\mathbf{y}\| \left(\sin\theta \left(\pi - \theta\right) \cos\theta\right) \tag{3.4}$$

where θ is the angle between the inputs $\mathbf{x}$ and $\mathbf{y}$ in d dimensions, satisfying $\cos\theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$.

The `pyro` interface for implementing a `Kernel` object is to write `init` and `forward` methods, where `init` initialises any necessary parameters, and `forward` computes the covariance matrix $\mathbf{K}$ between $\mathbf{X}$ (of size $N \times D$: i.e. N datapoints of dimension D each) and $\mathbf{Z}$ (of size $M \times D$). Specifically, `forward` computes the kernel value between every pair of points $\mathbf{x} \in \mathbf{X}$ and $\mathbf{z} \in \mathbf{Z}$: so the output will be the $N \times M$ Gram matrix $\mathbf{K}$, where $\mathbf{K}_{ij} = k_n(\mathbf{X}_i \mathbf{Z}_j)$.

We can use matrix operations and trigonometry to compute the kernel in a very elegant and efficient[4] manner, without iterating over elements, as shown in listing 3.1 (which is a simplified

[3] By the symmetry of forward and backward bridges, this is an identical problem.

[4] GPUs are optimized for linear algebra, so matrix multiplications are far better than for loops

version of the `forward` function I implemented). Note that in the listing, `sqrt` and other arithmetic operations are element-wise; `mm` refers to matrix multiplication and `.T` to transpose; and `norm(X, dim=1)` gives the norm of each d-dimensional element in `X`.

```
def arccos_forward(X, Z):
    XZ = X.mm(Z.T)                                          # shape: (n, m)
    X_norm = linalg.norm(X, dim=1)                          # shape: n
    Z_norm = linalg.norm(Z, dim=1)                          # shape: m
    X_norm_Z_norm = outer_product(X_norm, Z_norm)           # shape: (n, m)
    cos_theta = clip(XZ / X_norm_Z_norm, -1, 1)             # shape: (n, m)
    sin_theta = sqrt(1 - cos_theta ** 2)
    J = sin_theta + (pi - acos(cos_theta)) * cos_theta
    return  J * X_norm_Z_norm / pi                          # shape: (n, m)
```

Listing 3.1: ArcCos kernel implementation

Connection to neural networks

In practice, the simple kernel implementation struggles with GP regression; we need to transform the inputs appropriately with a weight and a bias, like in the original paper [35]. This is reminiscent of neural networks, and in fact the ArcCos kernel is also known as the neural network kernel due to a close connection to NNs. I will refer back to this later, so I outline the key ideas here.

Consider a simple, single-layer feedforward NN with weights $\mathbf{W}$, input $\mathbf{x}$ and output $\mathbf{f}(\mathbf{x}) = \Theta(\mathbf{W}\mathbf{x})$, where Θ is a ReLU nonlinearity. The inner product between two outputs $\mathbf{f}(\mathbf{x})$ and $\mathbf{f}(\mathbf{y})$ is then 3.5, where m is the number of output units and $\mathbf{w}_i$ is the i^{th} row of the weight matrix $\mathbf{W}$.

$$\mathbf{f}(\mathbf{x}) \cdot \mathbf{f}(\mathbf{y}) = \sum_{i=1}^{m} \Theta(\mathbf{w}_i \cdot \mathbf{x})\Theta(\mathbf{w}_i \cdot \mathbf{y})(\mathbf{w}_i \cdot \mathbf{x})^n(\mathbf{w}_i \cdot \mathbf{y})^n \tag{3.5}$$

If we now assume the weights are Gaussian initialised ($\mathbf{W}_{ij} \sim \mathcal{N}(0, 1)$), as m tends to infinity, this tends to the kernel equation 3.3 in expectation:[5] $\lim_{m\to\infty} \frac{2}{m}\mathbf{f}(\mathbf{x}) \cdot \mathbf{f}(\mathbf{y}) = k(\mathbf{x}, \mathbf{y})$ [35]. Thus, the ArcCos kernel is the inner product between the outputs of an infinite-width single-layer NN: this is an elegant result which we will refer back to, and which allows us to consider deep NN kernels (as analogous to *multilayer*, infinite-width NNs).

3.4 Sparse GPs

We now move on to considering sparse Gaussian processes, the first of the two methods for scaling up: I provide the intuition behind the method, derive the complexity, and then discuss implementation details.

3.4.1 Sparse low-rank approximations

The idea behind the sparse GP approximation is to find a smaller collection of M points, known as *inducing variables*, which represent the N training data points $\mathbf{X}$. We can then run

[5] Up to a constant scaling factor

a modification of the full GP regression formula on this subset. This is illustrated in figure 3.4: the inducing variables capture most of the variation in the underlying function, so are an effective representation of the whole dataset.

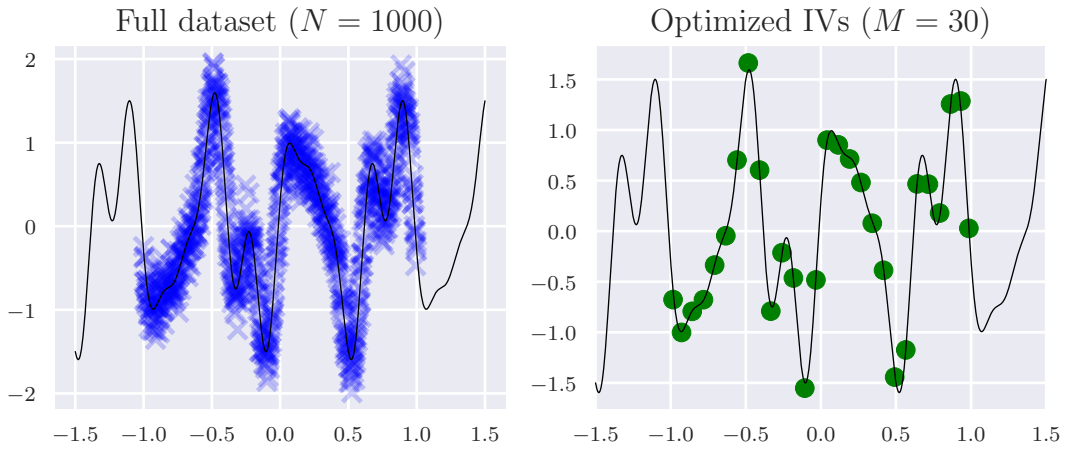


Figure 3.4: Illustration of sparse GPs: 30 inducing variables represent all 1000 datapoints

The complexity of regular GP regression is $\mathcal{O}(N^3)$ where N is the number of training points, due to the inversion of a $N \times N$ matrix $\mathbf{K} + \sigma^2\mathbf{I}$. In the sparse GP approach, this is reduced to $\mathcal{O}(NM^2)$: a significant speedup when $M \ll N$. Specifically, in our case we will fix M (as even small M values work surprisingly well with large N), meaning that time complexity is linear as we scale up the size of dataset.

3.4.2 Nyström method complexity analysis

The Nyström method [37] is one of the sparse GP approaches: it involves subsampling points from the input to create a low-rank approximation, and then using an approximation to the kernel to reduce the complexity of the inversion. Specifically, we sample without replacement a subset $\mathbf{Z}$ (of M points) from the training data $\mathbf{X}$ (of N points) such that $M \ll N$. We then approximate our training kernel $\mathbf{K}_{XX}$ by $\tilde{\mathbf{K}}$, where $\tilde{\mathbf{K}} = \mathbf{K}_{XZ}\mathbf{K}_{ZZ}^{-1}\mathbf{K}_{XZ}^{\top}$. Note the shapes, as this will provide our speedup: $\tilde{\mathbf{K}} \in \mathbb{R}^{N\times N}$, $\mathbf{K}_{XZ} \in \mathbb{R}^{N\times M}$, $\mathbf{K}_{ZZ} \in \mathbb{R}^{M\times M}$.

Substituting $\tilde{\mathbf{K}}$ for $\mathbf{K}$ in our GP regression equation doesn't directly bring any speedup, as $\tilde{\mathbf{K}}$ also has shape $N \times N$. We thus need to appeal to the Woodbury inversion formula [38]:

$$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}. \tag{3.6}$$

Recalling the GP regression equation, the problematic term is the $N \times N$ matrix inversion $(\mathbf{K}_{XX} + \sigma^2\mathbf{I})^{-1}$; substituting $\tilde{\mathbf{K}}$ for $\mathbf{K}$ here and using the Woodbury formula directly gives

$$\begin{aligned}(\tilde{\mathbf{K}} + \sigma^2\mathbf{I})^{-1} &= (\mathbf{K}_{XZ}\mathbf{K}_{XX}^{-1}\mathbf{K}_{XZ}^{\top} + \sigma^2\mathbf{I})^{-1} \\ &= \sigma^{-2}\mathbf{I}_N - \sigma^{-4}\mathbf{K}_{XZ}(\mathbf{K}_{ZZ} + \sigma^{-2}\mathbf{K}_{XZ}^{\top}\mathbf{K}_{XZ})^{-1}\mathbf{K}_{XZ}^{\top}\end{aligned}$$

We can then plug this into the full GP regression formula to get the sparse GP formula; note that we are now inverting $(\mathbf{K}_{ZZ} + \sigma^{-2}\mathbf{K}_{XZ}^\top \mathbf{K}_{XZ}) \in \mathbb{R}^{M\times M}$, so we invert an $M \times M$ matrix instead of an $N \times N$ one. Due to the multiplications, this approach has complexity $\mathcal{O}(NM^2)$: a significant speedup. Moreover, since we do not have to store the entire Gram matrix, the memory complexity reduces from $\mathcal{O}(N^2)$ to $\mathcal{O}(NM)$. As a final sanity check, note that if we sample the entire dataset $\mathbf{X}$, our approximation kernel $\tilde{\mathbf{K}} = \mathbf{K}\mathbf{K}^{-1}\mathbf{K}^\top = \mathbf{K}$ so the approximation is exact as expected.

More sophisticated algorithms for Sparse GPs also exist, such as variational inference [39] (which runs a learning process to optimize choice of inducing variables). In practice, due to the Nyström method's excellent performance and the prohibitive compute requirements of other methods like VI and FITC[6] [40], this project only considered the Nyström approach.[7]

3.4.3 Implementing the Nyström method

Although `pyro` comes with an inbuilt sparse GP implementation, I needed to implement some key modifications for its application to the SBP. Firstly, I had to implement the subsampling steps of the Nyström method (as `pyro` only supports VI and FITC) in a way that made sense for timeseries data. I chose the API of specifying `num_time_points` N_t and `num_data_points` N_d (so $M = N_t N_d$), and implemented sampling using the multinomial distribution,

Secondly, I had to optimize the GP regression object. In the `pyro` implementation, certain matrices (e.g. the Cholesky decomposition) relating to $\mathbf{K}_{XZ}$ and $\mathbf{K}_{ZZ}$ are recomputed for each dimension: we instead want to cache them and reuse them across dimensions. This is particularly useful for high-dimensional data like MNIST (784 dimensions): we avoid significant wasted computation by reducing the complexity from the naive implementation's $\mathcal{O}(NM^2d)$ to $\mathcal{O}(NM^2)$.

Finally, the inbuilt implementation required some reshaping to work with timeseries data. Overall, I ended up re-implementing most of the key methods provided by the native `pyro` implementation, and the results were excellent: for instance, Nyström with parameters ($N_d = 50$, $N_t = 30$) achieved a $> 500\times$ speedup over the baseline for $N = 1000$, $T = 34$.

3.5 Random features

I now move on to describing the second main approximation algorithm: the random features approach. The aim is to find a distribution of features ϕ such that their inner product approximates the kernel[8]: $k(\mathbf{x}, \mathbf{y}) \approx \langle \phi_{\mathbf{x}}, \phi_{\mathbf{y}} \rangle$. We can then compute the Gram matrix $\mathbf{K}_{XZ}$ by mapping each element of our inputs $\mathbf{X}$ and $\mathbf{Z}$ into a feature space using these features, and computing the matrix product: $\mathbf{K}_{XZ} \approx \Phi_X \Phi_Z^\top$.

[6] Fully Independent Training Conditional

[7] I hereby use *Nyström* and *sparse GP* interchangeably; technically, the Nyström method is a particular instance of the more general class of sparse GPs

[8] This has ties to Mercer's Theorem [41], the formalism behind the kernel trick.

3.5.1 Complexity analysis

Once we have this approximation, we can use a similar trick to that in the Sparse GP case. First, we rewrite the original GP formula to use the random features approach, so

$$f(\mathbf{x}) = \mu(\mathbf{x}) + \mathbf{K}_{\mathbf{x},\mathbf{X}}(\mathbf{K}_{\mathbf{X},\mathbf{X}} + \sigma^2\mathbf{I})^{-1}\mathbf{y} \tag{3.7}$$

becomes (once we approximate $\mathbf{K}_{XZ} \approx \Phi_X\Phi_Z^\top$)

$$f(\mathbf{x}) = \mu(\mathbf{x}) + \Phi_x\Phi_X^\top(\Phi_X\Phi_X^\top + \sigma^2\mathbf{I})^{-1}\mathbf{y}. \tag{3.8}$$

Note that Φ_X has shape $N \times F$, where F is the number of random features. We then use the push-through Wooodbury identity [42] $(I_F + UV)^{-1}U = U(I_N + VU)^{-1}$ (subscripting the identity matrix to show its size) to see that

$$\Phi_X^\top(\Phi_X\Phi_X^\top + \sigma^2\mathbf{I}_N)^{-1} = (\Phi_X^\top\Phi_X + \sigma^2\mathbf{I}_F)^{-1}\Phi_X^T. \tag{3.9}$$

Finally, we can substitute this back into equation 3.8 to get the prediction for a point $\mathbf{x}$ given training data $\mathbf{X}$:

$$f(\mathbf{x}) = \mu(\mathbf{x}) + \Phi_x(\Phi_X^\top\Phi_X + \sigma^2\mathbf{I}_F)^{-1}\Phi_X^T\mathbf{y} \tag{3.10}$$

We are now inverting the $F \times F$ matrix $(\Phi_X^\top\Phi_X + \sigma^2 I_f)$ instead of the original $N \times N$ matrix: this provides our speedup, reducing the complexity to $\mathcal{O}(NF^2)$.[9] Moreover, we don't have to store the Gram matrix– we just have to store the $N \times F$ matrix Φ, reducing our memory complexity to $\mathcal{O}(NF)$. Like in the sparse GP case, we will often fix F: meaning that the approach scales linearly in both time and memory with N.

3.5.2 Algorithm and abstract base class

We have not yet seen how to generate the Φ matrix; this is kernel-specific, and discussed in later sections. However, from the equation, we can implement an interface to return a prediction function for $\mathbf{x}$ given training data $(\mathbf{X}, \mathbf{y})$: this is sketched in algorithm 3. Input F is the number of random features: the higher, the better the approximation, but the slower the computation.

This interface requires an implementation of the `init_params` and `feature_mapping` functions, but these depend on the kernel and techniques involved: for instance, the RBF and Exponential kernels use random Fourier features for feature mapping, while the ArcCos (and later convolutional) kernels use a different technique. This suggests a natural structure: an abstract base class implementing algorithm 3, with subclasses being required to implement `init_params` and `feature_mapping`. This design allows for easy extensibility: to implement a new random features kernel, one simply has to implement the feature mapping function which defines their random features.

[9] Interestingly, equation 3.10 is actually just a feature-mapped linear regression: so we can use a pre-optimized function e.g. `torch.linalg.lstsq` to compute it.

Algorithm 3: Random features approximation

input: $\mathbf{X}$, $\mathbf{y}$, F, kernel, σ

params = init_params(kernel)
$\Phi_X = \Phi(\mathbf{X}, \text{params})$ (apply feature mapping)
weights matrix $\mathbf{W} = []$
for *each dimension in* $\mathbf{y}$ **do**
    $\mathbf{w}$ = solution to eq. 3.10 $((\Phi_X^\top \Phi_X + \sigma^2 \mathbf{I}_F)^{-1} \Phi_X^T \mathbf{y})$
    $\mathbf{W}$.append($\mathbf{w}$)
end
predict_fn = $\lambda x_{\text{test}} \to \Phi(x_{\text{test}}, \text{params})\ \mathbf{W}$
return *predict_fn*

The UML diagram is shown in figure 9; I implemented it in Python using `abc.ABC` (the abstract base class implementation), which allows me to mark abstract methods with the `@abstractmethod` decorator.

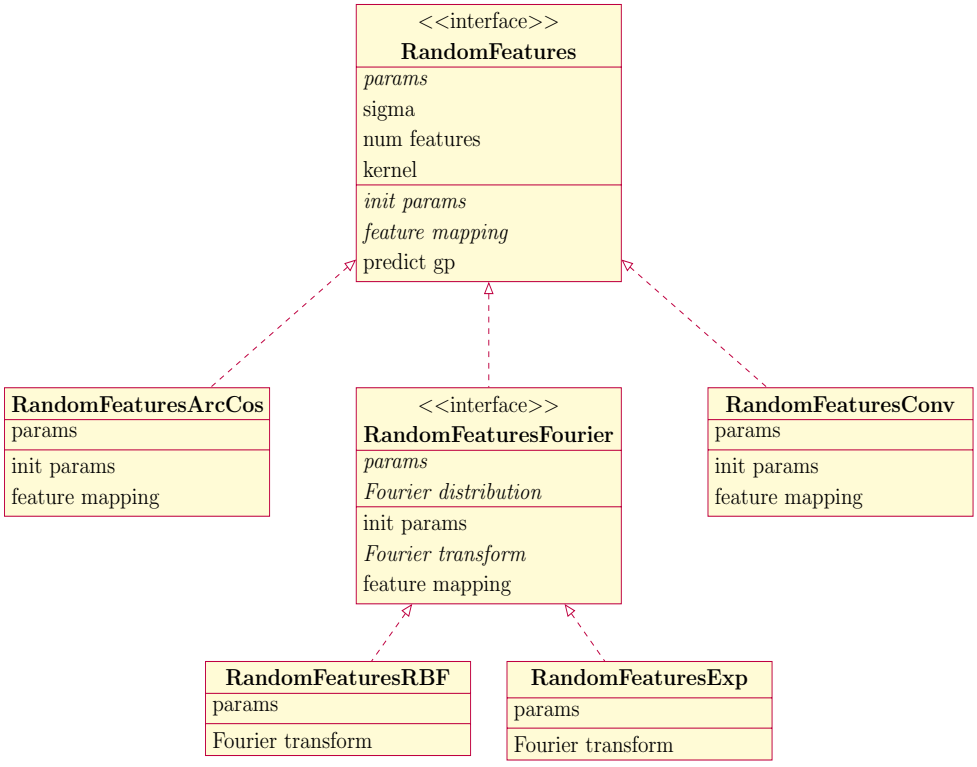


Figure 3.5: UML diagram of RandomFeatures implementation

3.5.3 Exponential/RBF kernel (Fourier features)

Within the random features approach, both the RBF and Exponential kernels use a technique known as random Fourier features [43]. This involves the following steps:

1. `fourier_transform(kernel)` transforms the kernel density into a Fourier space, and saves the Fourier distribution's sampling function in the variable `fourier_distribution`;
2. `init_params()` samples parameters ω from this Fourier distribution and $\mathbf{b}$ from $\mathcal{U}(0, 2\pi)$;
3. `feature_mapping(X)` returns $\Phi_X = \cos(\mathbf{X}\omega + \mathbf{b})$

Hence RandomFeaturesFourier itself forms another abstract base class: it implements `init_params` and `feature_mapping`, and requires subclasses to specify the kernel Fourier transform to sample from. I had to derive the Fourier transforms for these distributions by hand, using the formula $f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \exp(-i\omega x) f(\omega) d\omega$ and taking care to propagate kernel parameters and scalings appropriately. The details are in appendix A.3; here, I just show the Fourier distributions of the two kernels. A key implementation detail is the data structure used to store the Fourier distribution: the sampling function λ `shape` $\to$ `samples_of_shape` was the most pragmatic choice.

Algorithm 4: Exp Fourier transform

input: `kernel`

$\mathbf{\Sigma} = \frac{1}{\texttt{kernel.lengthscale}} \times \mathbf{I}_d$

`Normal` $= \lambda$ `dim` $\to \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$`.sample(dim)`

`Gamma_dist` $= \Gamma(0.5, 0.5)$`.stack(depth=d)`

`Gamma` $= \lambda$ `dim` $\to$ `Gamma_dist.sample(dim)`

return $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$`.sample`

Algorithm 5: RBF Fourier transform

input: `kernel`

$\mathbf{\Sigma} = \frac{1}{\texttt{kernel.lengthscale}^2} \times \mathbf{I}_d$

return $\mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$`.sample`

3.5.4 Arc-cosine kernel

The random features for the ArcCos kernel are significantly simpler: we can just Monte Carlo sample the defining integral we saw in section 3.3.3. That is, we sample the expectation

$$k(\mathbf{x}, \mathbf{y}) = 2\mathbb{E}_{\mathbf{b}\sim\mathcal{N}(0,\sigma_b^2)}\mathbb{E}_{\omega\sim\mathcal{N}(0,\sigma_\omega^2)}[\mathrm{ReLU}(\omega \cdot \mathbf{x} + \mathbf{b})\mathrm{ReLU}(\omega \cdot \mathbf{y} + \mathbf{b})]. \tag{3.11}$$

This leads to the following implementation: sample ω and $\mathbf{b}$ of the appropriate shapes from univariate Gaussian distributions in `init_params`, and return $\Phi_X = \mathrm{ReLU}(\mathbf{X}\omega + \mathbf{b})$ in `feature_mapping`. This implementation ties in closely to the concept of the ArcCos kernel approximating a randomly initialised neural network, which we will return to in section 3.6.5.

3.6 Convolutional kernels

Having successfully scaled up the GP approach using the Nyström and random features approaches, we may hope to have high-quality Schrödinger bridge computations with large, high-dimensional datasets. However, as we will see in section 4, although the techniques discussed up to now scale computationally to large datasets and perform well on toy examples, they do not compute effective Schrödinger bridges for MNIST. Thinking back to section 2.4.3 suggests why: recall that MNIST images are 28×28 pixels, so each image is a 784-dimensional datapoint (each pixel is a dimension). The simple approach of flattening each image into a 784-vector doesn't work because in such high-dimension spaces, we need some regularisation and structure to guide the learning process. We now consider convolutional GPs as a solution to this.

3.6.1 Patch-sum kernel

The first attempt at incorporating convolutions into a GP kernel was in [44]. The authors describe a "patch-sum" kernel, given by equation 3.12 and illustrated in figure 3.6 below. To

compute $k(\mathbf{x}, \mathbf{y})$ (where datapoints $\mathbf{x}$ and $\mathbf{y}$ are now images), we require an "underlying" kernel k_{base} which is applied to patches of our images. Assume we have patch size d, resulting in P patches.[10] We take a double sum over all patches in $\mathbf{x}$ and all patches in $\mathbf{y}$: for each pair of patches $(\mathbf{x}^{[p]}, \mathbf{y}^{[p']})$ we compute $k_{\text{base}}(\mathbf{x}^{[p]}, \mathbf{y}^{[p']})$ (having flattened the two patches of size $d \times d$ into d^2-length vectors before taking the k_{base} value).

$$k(\mathbf{x}, \mathbf{y}) = \sum_{p=1}^{P} \sum_{p'=1}^{P} k_{\text{base}}(\mathbf{x}^{[p]}, \mathbf{y}^{[p']}) \tag{3.12}$$

A key computational issue with this kernel is that the double sum makes computation of the Gram matrix unwieldy: we cannot use any efficient matrix multiplications, and have to simply loop over every element of the inputs $\mathbf{X}$ (of length N) and $\mathbf{Z}$ (of length M). The complexity of this approach is readily analysed: assuming the base kernel takes linear time $\mathcal{O}(dim)$ for single inputs of dimension dim (which is true for RBF, Exponential and ArcCos) the complexity is $\mathcal{O}(MNP^2d^2)$ (MN from the loop over datapoints; P^2 from the double sum over patches; d^2 from the kernel on a single pair of patches).

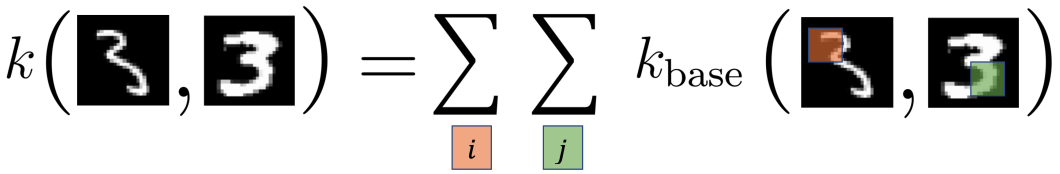


Figure 3.6: Illustration of double patch-sum kernel

I implemented this kernel, but (despite trying various optimizations) found that it took prohibitively long to experiment with. A simple modification, inspired by [36], is to use a kernel with a single sum over patches instead. One could argue that when judging similarity between images (recall that this is what a kernel does), patch i of image $\mathbf{x}$ should only be compared to patch i of image $\mathbf{x}'$, and not to patches j ($j \neq i$). This approach is shown in equation 3.13, and reduces the complexity to $\mathcal{O}(MNPd^2)$. I implemented this kernel, and although it runs faster and shows some promise, it is still too slow to experiment with (i.e. we cannot run several IPFP iterations feasibly).

$$k(\mathbf{x}, \mathbf{y}) = \sum_{p=1}^{P} k_{\text{base}}(\mathbf{x}^{[p]}, \mathbf{y}^{[p]}) \tag{3.13}$$

A final consideration is whether to include multiple channels in these convolutions: recall that different patch sizes can capture different patterns, and multiple channels can aggregate the information from these patches. In this case, the implementation (and equation) is simple: we just add another sum over channels to the front of the single/double sum over patches (and normalise by P to ensure the same scale is the same across channels). Computationally, there is an obvious linear effect of the number of channels C on the complexity,[11] which becomes $\mathcal{O}(CMNPd^2)$

[10] P and d are simply related, e.g. in the case of MNIST $P = (29 - d)^2$

[11] although this computation can be parallelized

The implementation for the convolutional kernels was trickier than expected, due to a bug in `pyro` which precluded defining base kernels as member variables within the `ConvKernel` class. After alerting the `pyro` devs to this, I worked around it by passing pre-instantiated base kernels' forward functions to the `ConvKernel` class, instead of the base kernels themselves.

3.6.2 Random-featurized patch-sum kernel

Having noticed some potential in convolutional kernels, but being unable to realise it due to computational complexity issues, we can try to scale them up using random feature techniques. First, recall that we can approximate $k_{\text{base}}(\mathbf{x}^{[p]}, \mathbf{y}^{[p]})$ with $\phi_x \phi_y^\top$ (where we represent the patches as flat vectors of length d^2). We can then flatten *the entire sum* over patches into the dot product of two vectors ϕ'_x and ϕ'_y of length Pd^2 each: we then have $k(\mathbf{x}, \mathbf{y}) \approx \langle \phi'_x, \phi'_y \rangle$ as required by the random features interface.

My `init_params` implementation thus samples $\mathbf{W}$ of shape $Pd^2 \times F$ (F = number of features) from $\mathcal{N}(\mathbf{0}, \sigma_w^2)$ and $\mathbf{b}$ of length F from $\mathcal{N}(\mathbf{0}, \sigma_b^2)$. `feature_mapping(X)` computes and flattens the convolutional patches of each point of the input, generating a matrix $\mathbf{X}'$ of shape $N \times Pd^2$, and returns $\Phi_X = \mathbf{X}'\mathbf{W} + \mathbf{b}$. In terms of complexity, we once again have only a linear dependence on N: we get our Φ by multiplying an $N \times Pd^2$ matrix with a $Pd^2 \times F$ one.[12] Like the non-convolutional case, this approach is also linear in N in memory.

This method immediately produces excellent results, as we will see in the evaluation; although I now describe a few other concepts I implemented (which may form future directions of research), this approach was by far the most successful in generating MNIST images.

3.6.3 Deep convolutional kernels

We saw in section 3.3.3 some parallels between neural networks and kernel machines, and touched on the possibility of deep kernels. Inspired by the ideas in [36], which uses layers of channels to learn different patterns in the data, I implemented a deep convolutional kernel to see whether it added quality to the bridge computation. For the exact approach, this effectively builds kernels inductively: $k^{(1)}$ uses k_{base} as before, and $k^{(l+1)}$ uses $k^{(l)}$ as its "base" kernel. Unfortunately, the complexity of this approach makes it infeasible: the complexity of a deep kernel of depth L is $\mathcal{O}((MNPd^2)^L)$, exponential in the depth. Given that the single-layer convolutional kernel was already infeasible, it is not worth experimenting with this kernel.

However, the random features case is more amenable to depth: with L layers, we can simply sample L weight matrices $\mathbf{W}^l$ and $\mathbf{b}^l$ for $l = 1, ..., L$ and compute

$$\Phi'_X = \left(\dots \left(\left(\mathbf{X}'\mathbf{W}^{(1)} + \mathbf{b}^{(1)} \right) \mathbf{W}^{(2)} \right) + \mathbf{b}^{(2)} \dots \right) \mathbf{W}^{(L)} + \mathbf{b}^{(L)}. \tag{3.14}$$

Although this increases the complexity of the random features approach by a factor of L, it is still feasible for relatively shallow kernels; the implementation follows from the equation above.

[12] Although this looks computationally intensive, remember that P and d are small compared to N

3.6.4 The UNet approach

As we delve deeper into convolutional kernels, it is worth taking a step back to take stock of the problem we are trying to solve. We are still trying to fit the drift of the SDE (effectively estimating the drift at each timestep) using Gaussian processes; the convolutional structure is an attempt to regularize the problem so our solutions work in high-dimensional spaces. As hinted at in previous sections, there is a fundamental connection between NN and GP computation; implementing an NN approach to fitting the drift can provide an alternative computational perspective of the GP approach. In this section, I outline my implementation of the neural network approach (specifically, the UNet).

The UNet architecture was first proposed in [45], and is used by de Bortoli et. al. [6] in their IPFP implementation. It consists of iterated blocks of convolution and pooling (resulting in a small but deep representation) followed by upconvolutions with a residual connection. This allows for output of the same shape as the input, which is useful for our purpose of predicting images. The architecture is illustrated in figure 3.7: downward arrows represent max-pooling (which halves the spatial dimensions and doubles the number of channels), upward arrows represent upconvolutions[13] (which double the spatial dimensions and halve the number of channels), and adjacent blocks represent convolutional layers. Residual connections from the convolutions are also added at the deconvolution stages, ensuring that information is passed effectively across the layers. I detail the exact architecture I used in appendix A.

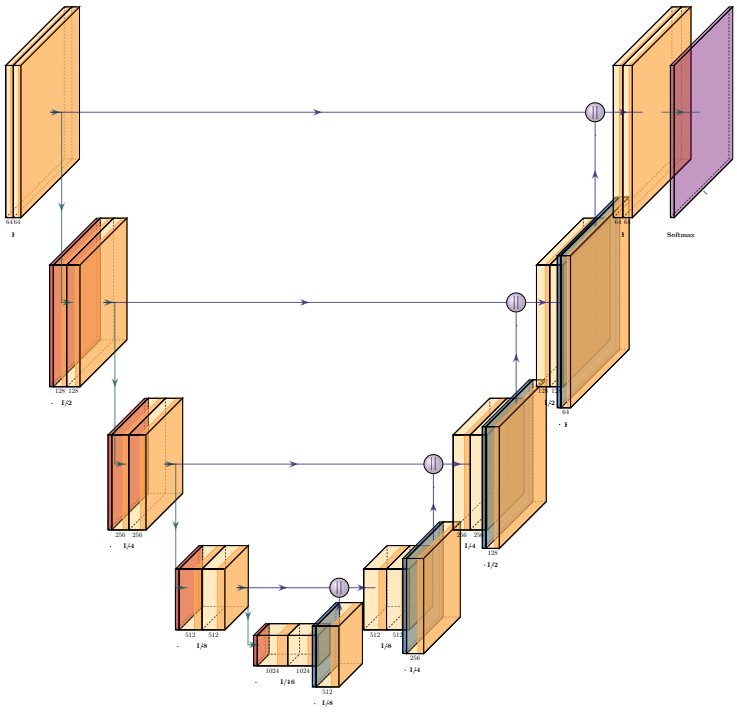


Figure 3.7: Illustration of the UNet architecture [4]

I trained this model with gradient descent on mean squared error loss, using the Adam optimizer for its fast convergence properties. I tried several variants of the architecture and ran grid-searches for hyperparameter tuning, testing on simple noise-digit regression tasks: in general,

[13]The implementation here is a PyTorch `ConvTranspose2D`, which transposes the convolution matrix and applies it. It is effectively (although not technically) the inverse of a convolution, so can be used to increase the height and width of inputs.

the model performed well. However, it took a long time to train, which made it infeasible to run on the full IPFP algorithm. I analysed the characteristics of training error vs epoch, and profiled the code, but the long training time seemed an unavoidable consequence of using this deep network to predict $\mathcal{O}(NT)$ datapoints several times in a loop. The only real results I achieved were from a $\sim$ 12-hour run, which showed some promise; it is left to future work to refine the approach– by using more compute power or further fine-tuning the architecture– and gain better insight into results.

3.6.5 Reconciling NN and GP approaches

The main reason for implementing the UNet approach is the realisation that with certain alterations, the NN computes exactly as a GP. The concept of infinite-width, single layer, randomly-initialised neural networks tending to GP behaviour has been well studied [46, 47]; but recent results have extended this to deep neural networks as well [48, 49]. This is related to the deep kernels from the previous section: in particular, if we take a UNet, initialise weights and biases according to a specific prior, and train *only* the final layer [50], this NN tends to a deep, convolutional GP with random features.

In this case, we are training a model of the form $\mathbf{y} = \mathbf{W}\Phi(\mathbf{x})$, where $\Phi(\mathbf{x})$ is the (randomly-initialised, fixed-weight) UNet up to the penultimate layer; since this is a linear model, we are effectively approximating a GP with kernel $\langle \Phi_{\mathbf{x}}, \Phi_{\mathbf{y}} \rangle$. This equivalence can be practically implemented as follows:

1. Initialise UNet weights with $\mathcal{N}(0, \frac{\sigma_w^2}{\texttt{num\_channels}})$
2. Detach all weights except for those in the last layer from the PyTorch computational graph
3. Run the optimization, where the optimizer only optimizes the last layer's weights

This would give us a scalable way of computing deep random features, as the last layer can even be trained with the closed form linear regression equation. I implemented this approach, making use of PyTorch's flexibility in dynamically modifying the computational graph, and used Kaiming parameter initialisation [51] to appropriately scale weights (taking into account the ReLU activation); but I did not have time to tune to model carefully (as this investigation was a distant extension), so this is left as an avenue of future work.

Chapter 4

Evaluation

In this section, I systematically evaluate the performance of the implementation components described in chapter 3, demonstrating that all core objectives and several extensions– including some which were not initially planned– were achieved and exceeded.

4.1 Evaluation methodology

Before diving into results, I briefly overview the datasets used, some key recurring metrics and plots, and evaluation methodologies.

4.1.1 Datasets

Most of my experimentation and evaluation was done on 2-dimensional toy examples from the `sklearn` library [11], which are sampled from specific toy distributions according to parameters like number of datapoints. I used these datasets because they are easy to visualise and compute metrics for, while still allowing us to push the limits of time and memory performance by increasing the number of datapoints. These datasets have an element of randomness (due to sampling), which I control using seeds for reproducible evaluation. The specific datasets used are shown in figure 4.1 below;[1] for the plots and metrics in this evaluation section, I largely use the spiral dataset as it is easy to visualise.

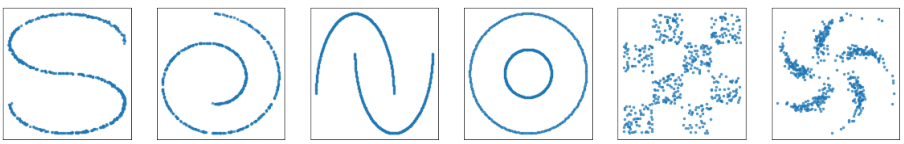


Figure 4.1: Toy datasets used for evaluation

To test whether the scaling up and quality improvements (specifically convolutional kernels, which are are only useful for images) extend to high-dimensional datasets, I then move to the MNIST dataset [52]. MNIST is a canonical ML dataset of 60,000 handwritten digits, often used for classification tasks. It consists of 28x28 images, so each image is a 784-dimensional

[1] the first four are from `sklearn`; the last two I implemented, with inspiration from [6]

datapoint; in the context of the Schrödinger Bridge problem, I sample 784-dimensional Gaussian noise, arranged in a 28x28 grid, and then compute the forward and reverse bridges to MNIST from noise. I implemented stratified subsampling (taking a subset with equal proportions of each class) to reduce the number of images for some experiments, and normalised the dataset before use for stability.

Figure 4.2: Sample MNIST images

4.1.2 Key metrics and plots

Although different evaluation methods are suitable for assessing different parts of the implementation, we need some core metrics to evaluate bridge quality and code performance.

Firstly, to evaluate bridge quality (i.e. compare the distribution generated by our Schrödinger Bridge to the target distribution), we need a sample-based measure of distributional distance. Here, I use the Earth mover's distance, or EMD: see appendix A.2 for the definition and an elegant implementation using the weighted bipartite matching problem. The key plots used to inspect bridge quality include 3D plots of Schrödinger bridge trajectory over time and 2D plots overlaying the target and learnt distributions on the same axes. I evaluate how well different approaches scale by plotting EMD, time and memory against number of samples (for toy examples): this clearly demonstrates the complexity reductions in the approximate methods.

For MNIST, EMD is no longer meaningful. In fact, comparing the generated distribution to the target is difficult; instead, I consider single generated images (datapoints), and compare them to all images of the class they are trying to generate. I use mean squared error loss for this, aggregated over the target class: for instance, if we are trying to generate a 3, I compute the average MSE between the generated image and all 3s in the MNIST dataset.

4.1.3 Ensuring robust evaluation

I took care during evaluation to ensure that results are rigorous and reproducible. Firstly, I used PyTorch's built-in benchmarking and memory-inspection facilities: the time benchmarking uses optimizations like disabling the garbage collector to reduce random variation, while the memory interface retrieves statistics directly from the GPU. I ran each experiment at least 5 times (often more, for faster-running experiments), and reported the mean result with error bars given by the standard deviation.

There were several changes to evaluate, and it was important to isolate them: for instance, running the sparse GP approach with time-dependent noise against the exact GP baseline with constant noise would not be a fair test. Each experiment in this evaluation section tests exactly one change (although I did also check combinations of changes to ensure there were no dependencies).

4.2 Sparse GPs

We first consider the sparse GP approach. Before comparing it to the exact GP approach (hereby referred to as the "baseline"), it is worth getting a feel for how the number of time points N_t and number of data points N_d (whose product is the number of inducing variables M) affects quality and performance characteristics. The tables below show results for $N = 200$, i.e. 200 datapoints:[2] as expected, as number of inducing variables increases, bridge quality generally improves but time and memory usage increase.

	10	30	60	100	150
10	0.34	0.26	0.29	0.16	0.20
20	0.31	0.22	0.17	0.23	0.16
30	0.28	0.21	0.18	0.17	0.16

Table 4.1: EMD

	10	30	60	100	150
10	0.59	0.64	0.84	1.59	2.54
20	0.72	0.92	2.04	2.11	3.67
30	0.73	1.55	3.29	3.66	7.05

Table 4.2: Time / s

	10	30	60	100	150
10	0.62	1.05	1.71	2.59	3.74
20	0.83	1.71	3.05	4.89	7.31
30	1.05	2.37	4.43	7.31	11.13

Table 4.3: Memory / GiB

Table 4.4: How performance scales with increasing N_t (down rows) and N_d (across columns)

We can now compare the scaling behaviour of the sparse GP approach to the baseline. The three plots in figure 4.3 show that as N increases, the sparse approach (here with $N_t = 30, N_d = 30$) gives similar EMD scores to the baseline; but the time taken is significantly lower, and memory scales better. This ties in with the theory: the exact approach uses quadratic memory (to store the Gram matrix) while the sparse approach uses $\mathcal{O}(NM^2)$ memory, which is linear in N (since M is constant in these experiments). Similarly, the exact approach takes cubic time in N for the matrix inversion; while the sparse approach takes $\mathcal{O}(NM^2)$, which– since M is fixed– is simply linear in N.

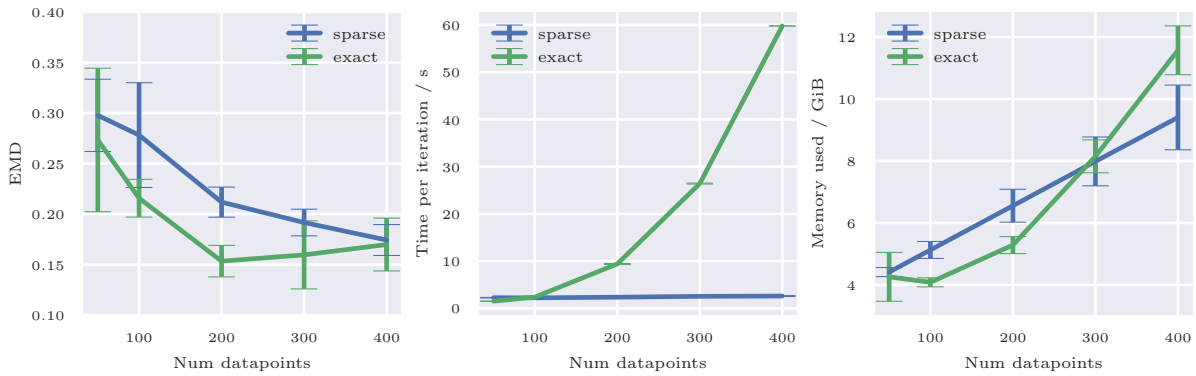


Figure 4.3: Sparse GPs vs baseline: EMD, time and memory scaling with N

Finally, we can consider pushing the sparse approach to its limits. The exact approach runs out of memory for $N > 1000$, so this is the limit it can be tested on: it takes 1174 ± 21 seconds per iteration, and 46.25GiB of memory to achieve an EMD score of 0.134 ± 0.018. The sparse GP approach with $N_t = 20, N_d = 50$ achieves a similar EMD (0.136 ± 0.035) with just 2.21 ± 0.04

[2] A note on timesteps T: remember that the "N" value in the GP regression is actually NT, number of datapoints $\times$ number of discretization timesteps. For instance, in the $N = 200$ example above, the exact approach inverts a Gram matrix of size $200 \times 34 = 6800$. Throughout the evaluation, I fix $T = 34$; as this is a constant, I omit it from the complexity and simply refer to $O(N^3)$ as the complexity.

seconds per iteration and 25GiB of memory: i.e. an identical result with a 530× speedup and $\sim 2\times$ lower memory consumption.

Moreover, the sparse GP approach can scale up to far larger datasets, e.g. $N = 10,000$, which the exact approach would require several terabytes of memory for. Consider $n = 5000$: taking $N_t = 30, N_d = 30$ gives an EMD of 0.131 ± 0.051, with just 4.15 ± 0.16s per iteration and 35GiB of memory. Extrapolating the exact approach (based on its known complexity) suggests it would take roughly a week (and 3.5TB), to compute: this reaffirms the superiority of the sparse GP approach.

This section meets core criteria 1, 2 and 3: I have obtained similar results to the exact approach with sparse GPs, while taking significantly less time and memory.

4.3 Random features

To evaluate random features, we can use similar plots as those used to evaluate sparse GPs.[3] In the random features case, we have a single parameter F (the number of features); there is again a tradeoff implicit in the choice of this parameter (higher F means better quality but slower computation). Figure 4.4 demonstrates the random features approach (here with $F = 2000$) yields similar EMD scores to the baseline; but the time taken is significantly lower, and memory scales much better. Again, with N datapoints, we have reduced memory complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(NF)$, and time complexity from $\mathcal{O}(N^3)$ to $\mathcal{O}(NF^2)$ (both linear in N).

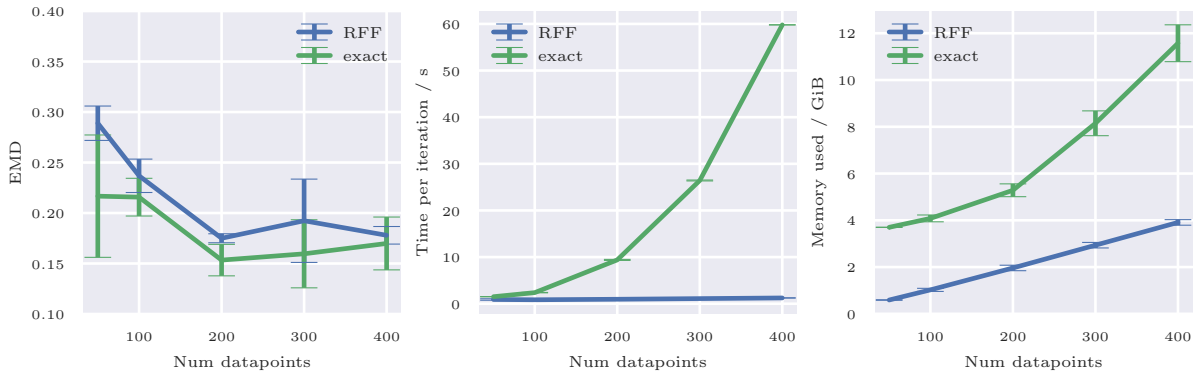


Figure 4.4: Random features vs baseline: EMD, time and memory scaling with N

Like we did for sparse GPs, we can push the RFF approach to the limit to see how it performs. First we compare it to the largest N the exact approach can manage: recall that this is $N = 1000$, and the exact approach takes 1174 ± 21 seconds per iteration (and 46.25GiB of memory) to achieve an EMD score of 0.134 ± 0.018. With random Fourier features, we achieve a similar EMD of 0.135 ± 0.07, with 1.12 ± 0.08s per iteration and 15GiB of memory. This is a $> 1000\times$ speedup and $3\times$ reduction in memory for an identical quality Schrödinger bridge.

[3] I use the Exponential kernel implementation of random Fourier features for most of the analyses in this chapter, for consistency with the sparse GP and exact kernel approaches. I thus sometimes refer to this as the RFF approach.

As in the sparse GP case, we can also push the random features approach to the extreme of 5000 datapoints:[4] it performs extremely well, achieving an EMD of 0.103 ± 0.06 with just 7.53 ± 0.24s time and 39GiB memory used.

This section meets core criteria 1, 2 and 3: I have obtained similar results to the exact approach with random features, while taking significantly less time and memory.

4.3.1 Convergence to true kernel

Given that the choice of F plays such a crucial role in determining a quality-speed tradeoff, we may wonder how many features are required for a good approximation to the true kernel. I investigate this with two approaches below. Firstly, I run the IPFP algorithm with different numbers of features F, and compare the EMD scores among runs: this is plotted in figure 4.5(a), and suggests that convergence is fairly rapid (adding more features doesn't significantly improve Schrödinger Bridge quality).

We can also remove a layer of abstraction, and consider directly the convergence of the approximate kernel $\Phi\Phi^\top$ to the exact kernel. As hoped, as F increases, the error tends to 0 rapidly: figure 4.5(b) shows the maximum absolute error $\max(\mathbf{K} - \Phi\Phi^\top)$, while figure 4.5(c) shows the mean squared error $\mathbb{E}((\mathbf{K} - \Phi\Phi^\top)^2)$.

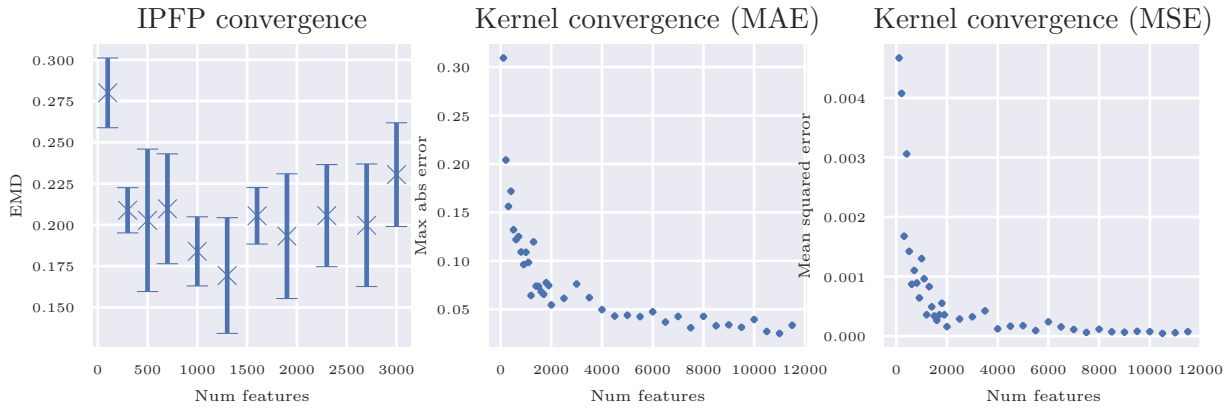


Figure 4.5: Convergence behaviour of random features with increasing F

4.4 Comparing sparse GPs and random Fourier features

Core criterion 5 was to compare sparse GP and RFF approaches: we have already developed the tools and plots to do so. Firstly, we can consider how they scale with N using similar plots to in the previous sections. A natural question is which parameters to use: there is a cubic search space (F, N_t, N_d) to choose from at each point. For these experiments, I choose $F = 2000$ for the random features approach, and ($N_t = 20$, $N_d = 50$) for the sparse GP approach: a pragmatic choice which puts both methods on a similar runtime scale.

Figure 4.6 demonstrates that the approaches have similar EMD scores, and have runtimes and memory usages on the same scale. RFF is twice as fast for small values of N, and remains faster

[4] A low enough F can handle 10,000 or 20,000 features as well, but EMD computation becomes the bottleneck

until $N = 1000$; it also uses less memory by a clear constant factor, which means it can handle a greater N when the bottleneck is memory (which it most often is). Note that these plots confirm the linear (in N) runtime and memory of both approaches as well: it is encouraging to see that this trend continues as expected for greater N.

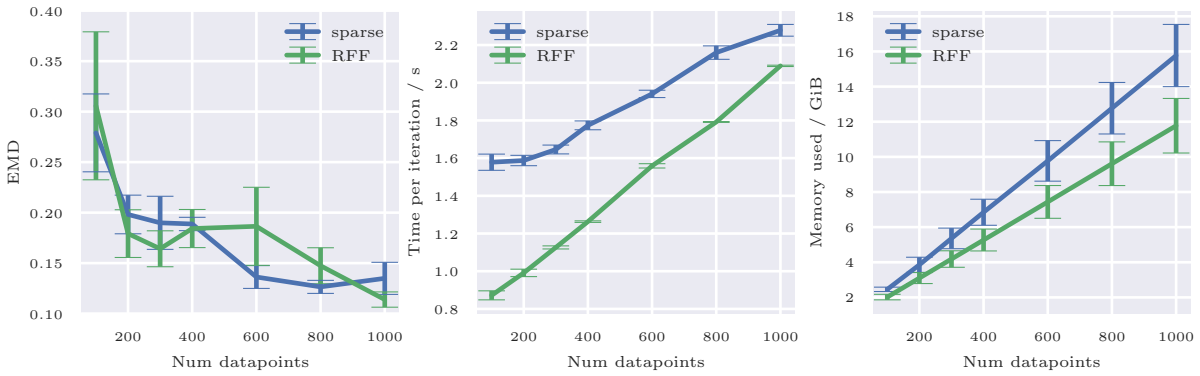


Figure 4.6: Sparse GPs vs random Fourier features: EMD, time and memory scaling with N

Although the plot above is informative, it only gives the picture for a single set of parameters. To gain more insight, we can fix $N = 200$, and vary the parameters for the two approaches; we can then plot EMD against runtime for both sparse and RFF approaches, with one datapoint per parameter setting. Figure 4.7(a) plots this approach, while figure 4.7(b) plots the same for memory. It seems that RFF is consistently slightly faster and more memory-efficient than the sparse GP approach, while achieving similar EMDs: suggesting it is the superior method.

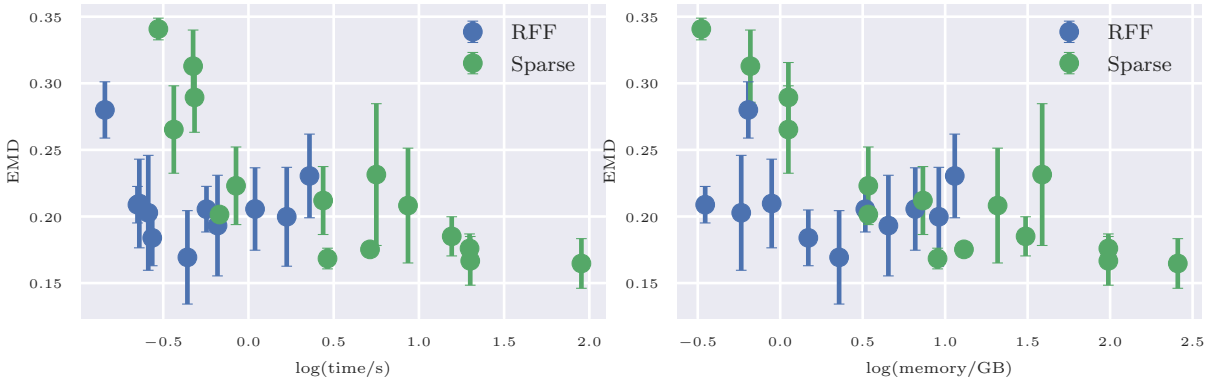
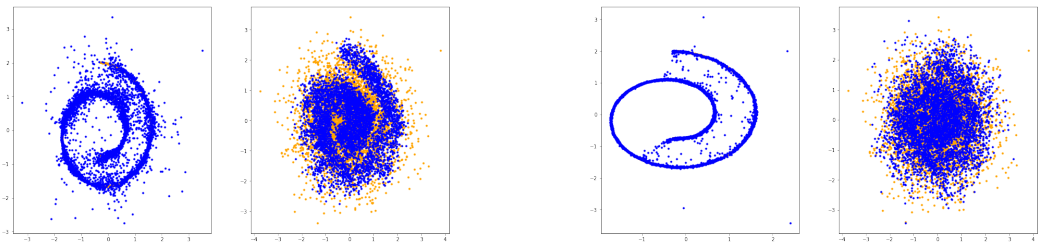


Figure 4.7: Sparse GPs vs random Fourier features: quality against memory and time taken

Finally, we can investigate how the methods compare in the extreme case of $N = 5000$. Figure 4.8 demonstrates the result qualitatively, suggesting RFF is significantly better in these high-data regimes. The values for EMD, time and memory usage are shown in table 4.5; they confirm that with a similar memory budget, RFF performs much better than sparse GPs. Overall, I find that sparse GPs may scale as well in terms of time, but the RFF approach is significantly more memory efficient and achieves better EMDs in large dataset regimes – making it more suitable for our most of our use cases.

	Time/s	Mem/GiB	EMD
Sparse GP	4.15 ± 0.16	35.1	0.131 ± 0.051
Random features	7.53 ± 0.24	39.0	0.103 ± 0.063

Table 4.5: Comparing sparse GP and random Fourier feature approaches for $N = 5000$



(a) Sparse (b) RFF

Figure 4.8: Performance of approximation methods for $N = 5000$

This meets core criterion 5: I have compared sparse GP and RFF approaches, and argued which is better.

4.5 MNIST experiments

Having covered most of the core evaluation criteria, I now evaluate the extensions of convolutional kernels applied to the MNIST dataset.

4.5.1 Exact convolutional kernels

As mentioned in section 3.6.1, the exact convolutional kernels were very slow to run: a few runtime values are given in table 4.6, which confirm the linear growth of runtime with number of channels.[5] Due to space constraints, and since the next section discusses convolutional kernels' efficacy, no further evaluation is provided.

	Filter sizes (channels)		
Runtime / iteration (s)	[3]	[3, 10]	[3, 10, 16]
Double-patch-sum kernel	1103	2037	2971
Single-patch-sum kernel	642	1184	1650

Table 4.6: Runtime per iteration (in seconds) of the two exact convolutional kernel approaches

4.5.2 Random features convolutional kernel

The random-featurised convolutional kernel is responsible for the effective generation of MNIST images, the headline result of this project. Plots of time and memory taken for different dataset sizes N (using $F = 1000$) are given in figure 4.9: they demonstrate that this approach is

[5]Error estimates are unnecessary here, as we are not drawing any meaningful conclusions.

significantly faster than other techniques for image generation (e.g. Generative Adversarial Networks [53], which take several minutes to train to any reasonable quality).

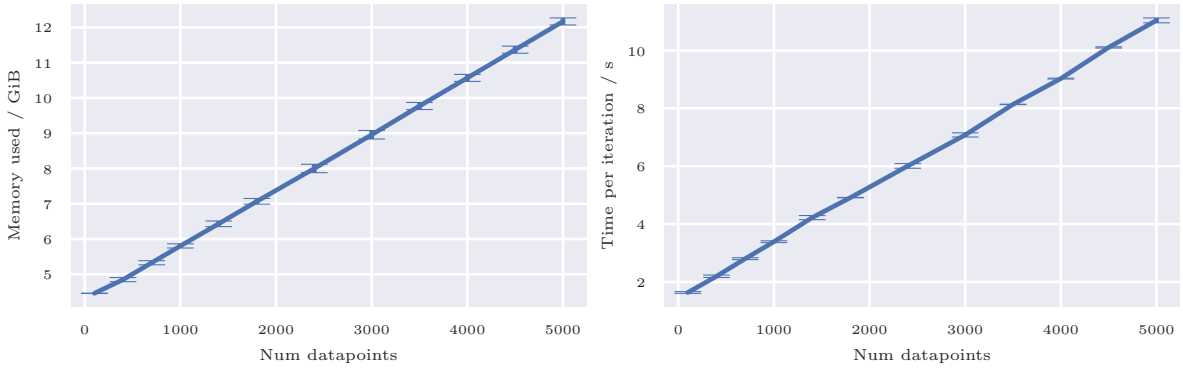


Figure 4.9: Random-featurised convolutional kernel, time and memory scaling with N

We can evaluate the quality of generated images both qualitatively and quantitatively: the former is perhaps more appropriate. An instance of the learnt Schrödinger Bridge is shown in figs 4.10 and 4.11: the algorithm is able to effectively transport the 2 to noise, and new Gaussian noise into a novel generated 2.

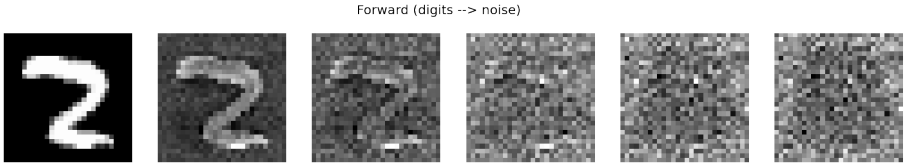


Figure 4.10: Learnt Schrödinger bridge from digits to noise

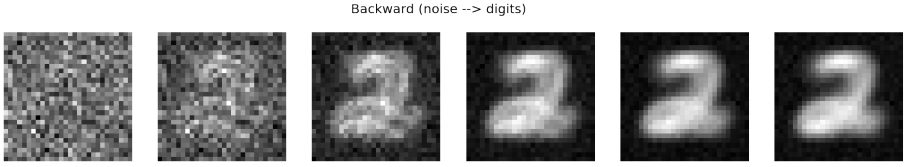


Figure 4.11: Learnt Schrödinger bridge from noise to digits

Figure 4.12 shows generated images for each digit: these clearly demonstrate that the approach is working as expected.

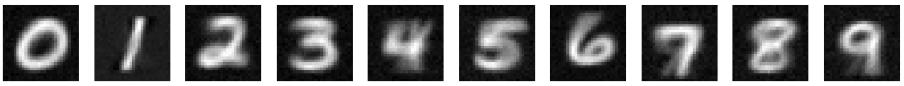


Figure 4.12: Generated MNIST images using featurised convolutional kernel

As a final quantification of success, we can compare the images to the original MNIST dataset. The metric I use here is total MSE: the average MSE to all MNIST images in the target class. This value can then be compared to another generative network; I implemented a GAN [53] architecture for this purpose. I normalised the images generated from my approach and the GAN, and then computed total MSE for each digit: table 4.7 shows the results (where

lower numbers indicate greater similarity). Clearly, the Schrödinger Bridge approach with my convolutional kernel is significantly better at generating MNIST images than the GAN. Moreover, while the SBP takes a few seconds, the GAN takes several minutes to train. *This meets core criterion 4 and extension criterion 5: generating MNIST images using convolutions.*

	Digit									
	0	1	2	3	4	5	6	7	8	9
SBP	0.629	0.635	0.804	0.747	0.821	0.883	0.781	0.802	0.714	0.701
GAN	1.497	0.883	1.178	1.135	0.997	1.309	0.859	0.965	1.012	1.014

Table 4.7: Total MSE scores of SBP vs GAN for generated digits

4.5.3 UNet approach

I implemented a UNet approach to the Schrödinger Bridge problem, *meeting extension criterion 6*. The UNet was extremely slow to train and very memory-intensive, so was infeasible for using in the IPFP loop (the only working runs took > 1 hour per iteration); it is left to future work to further evaluate and develop the NN approach.

4.6 OU prior

I now evaluate some of the smaller-scale changes I implemented to improve quality, starting with the OU vs Brownian prior. To compare the effect of the prior, I applied the IPFP algorithm to MNIST images, once with Brownian drift (figure 4.13(a)) and once with OU drift (figure 4.13(a)). We can see that the Brownian drift fails to transport the 3 to true noise, meaning that the reverse process cannot generate images from new noise. Meanwhile, the OU drift converges faster to a true Gaussian distribution, so is more successful in image generation from new noise. This effect is more pronounced in high dimensions where the search space is larger: due to the curse of dimensionality, Brownian drift is inadequate. *This section meets extension criterion 4 of improving bridge quality.*

Figure 4.13: MNIST IPFP with Brownian (left) vs OU (right) prior

4.7 Heteroskedastic noise

Recall the need for heteroskedastic noise (that a high σ explores the state space, while a low σ hones in on the target distribution); we can observe this dichotomy in diagram 4.14. On the left, with too low a constant σ, the SDESolver cannot drift to the target distribution; while in the middle (with a higher σ) it does drift to the right location, but fails to capture the detail

of the distribution. The heteroskedastic case is shown on the right: we manage to drift to the correct area when σ is high (around the middle timesteps), *and* successfully capture some detail of the distribution when σ is low (in the last timesteps). The EMD values are 8.13, 0.34 and 0.25 respectively; these confirm our qualitative assessment that heteroskedastic noise improves results. *This section meets extension criterion 4 of improving bridge quality.*

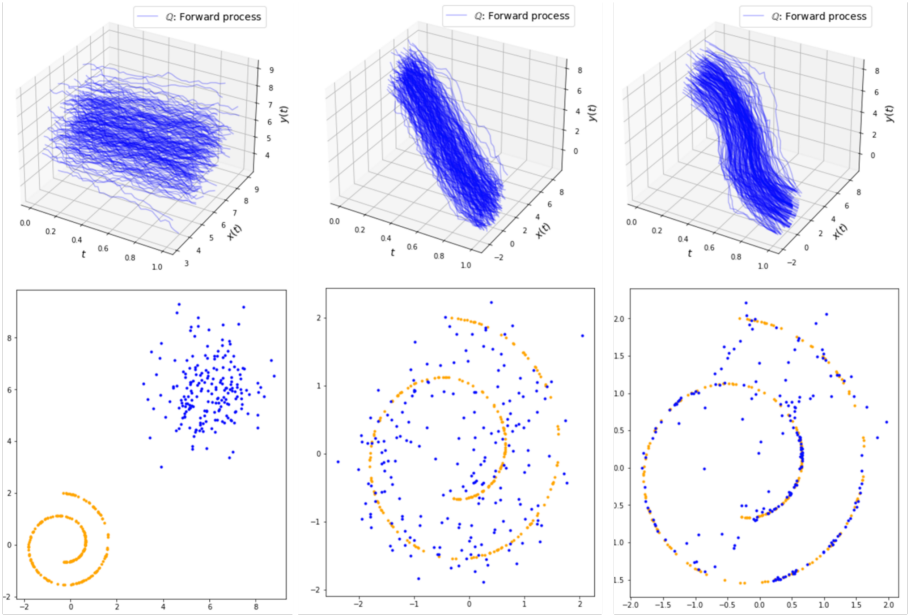


Figure 4.14: Effect of time-varying σ

4.8 Comparing kernels

Finally, I evaluate the performance characteristics of the exact ArcCos kernel I implemented, with respect to pre-implemented kernels. Figure 4.15 demonstrates that my ArcCos implementation performs as well as `pyro`'s Exponential, and better than RBF, in EMD; the time taken is similar across kernels (as kernel `forward` computation is not the bottleneck); and my ArcCos uses slightly more memory than the `pyro` Exp and RBF implementations. The results back up the theory: that Exponential and ArcCos are the optimal kernels for this task.

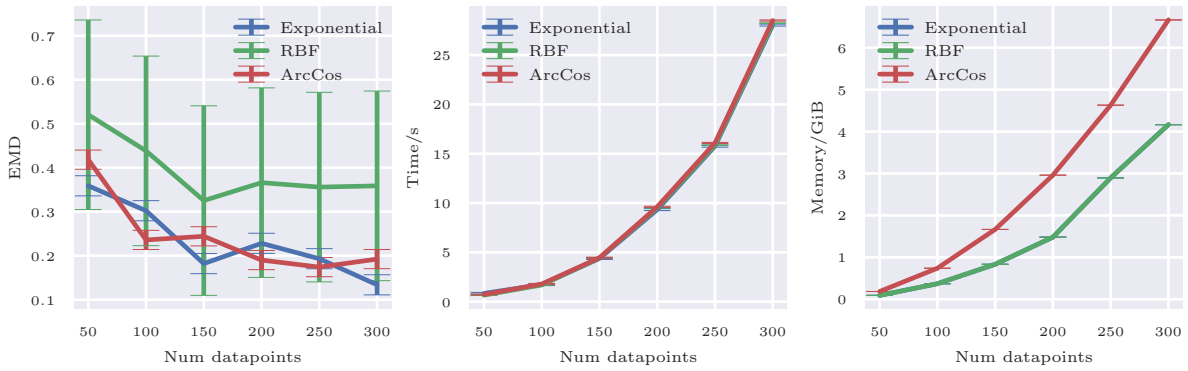


Figure 4.15: Comparing kernels: time, memory and quality scaling with N

This meets extension criterion 3 of implementing and comparing kernels.

Chapter 5

Conclusion

5.1 Achievements

In this project, I implemented two different techniques (sparse GPs and random features) for reducing time and memory complexity of the Gaussian process solution to the Schrödinger Bridge Problem, and demonstrated a significant speedup and reduction in memory usage over the baseline implementation. Even for small datasets ($N = 1000$), my random features implementation is $1000\times$ faster than the baseline and $3\times$ more memory efficient, without compromising in quality. This approach also significantly outperforms[1] the neural network approach [6] to the SBP, the other state-of-the-art technique: computing results in a few seconds rather than hours, and scaling up to large datasets without memory issues.

These scaling-up algorithms enabled GP Schrödinger Bridge computation on the MNIST dataset for the first time, the core aim of the project. To improve the quality of results in high dimensions, I then researched and implemented convolutional Gaussian process kernels; combining these with the random features approach yielded high-quality bridges that were capable of generating MNIST images. This is a novel contribution to the field, which I plan to publish in conjunction with my supervisor; it has the potential to advance various applications of the Schrödinger Bridge Problem, like medical imaging.

Overall, the project was a success: I achieved and exceeded all core and several extension objectives, including many which were not initially planned. Beside the achievements already discussed, I implemented performance optimizations, various quality improvements, a new kernel, and a robust evaluation framework: each of these played a part in the headline results of image generation. I also explored the connection between Gaussian processes and neural networks, providing an implementation of a UNet architecture for fitting the SDE drift and discussing equivalences between NN and GP computation.

5.2 Future work

There are several promising avenues of future work to build on the results presented in this project.

[1] In computational terms; I have not compared quality

- **NN kernel architectures** — I discussed in section 3.6.5 how a randomly initialised neural network where we only train the last layer acts as a GP. Investigating different architectures for this, which correspond to different kinds of random features, will likely to lead to improvements in quality.
- **Larger and higher-dimensionality datasets** — With greater compute power, the approaches in this project approaches can be tested on larger datasets (using, for instance, a batched approach). Although larger dataset size should improve results, higher dimensionality (like in CelebA and CIFAR-10) may require refining the convolutional features and considering new approaches.
- **Exploring new features and kernels** — Although I implemented a deep convolutional random features approach, I did not have time to test or tune it thoroughly. Adding more channels and refining the deep approach may offer significant improvements on the single-layer random feature approach.

5.3 Reflections on challenges and learnings

Overall, this project was challenging and enjoyable in equal measure, and I learnt a great deal. On the technical side, it was rewarding to study advanced concepts like SDEs and deep kernel methods: I had no experience with such topics before, so had to spend a significant amount of time reading textbooks and watching lectures. It was also satisfying to practically implement some concepts which we only cover theoretically in the CST, like randomised algorithms – I gained an appreciation of how these ideas can be applied to solving real engineering issues. Lastly, the project familiarised me with several practical software engineering techniques like memory optimization, neural network tuning, and extensible codebase design, all of which will help me in future projects.

The project also had its challenges, some of which stemmed from the large amount of theory I had to understand and apply: I often had to grapple for hours with advanced concepts just to implement a few crucial lines of code. Another key challenge was the significant research element of my project, which meant that I often worked on implementation aspects for several hours without promising results. It was difficult to organise research and identify potential avenues of improvement, and I sometimes found myself at dead ends; in retrospect, I would have spent more time considering my approach at a high level, rather than getting caught up in the minutiae of experimental tweaks.

All in all, I am very satisfied with how the project went. I learnt a lot, and advanced the state of the art in GP Schrödinger Bridge computation; I hope these techniques are taken forward and implemented in real-world applications like medicine and the sciences.

Bibliography

[1] Genji Kawakita, Shunsuke Kamiya, Shuntaro Sasai, Jun Kitazono, and Masafumi Oizumi. Quantifying brain state transition cost via schrödinger bridge. *Network Neuroscience*, 6(1):118–134, 2022.

[2] Cdipaolo96. Gaussian process draws from prior distribution. `https://commons.wikimedia.org/wiki/File:Gaussian_process_draws_from_prior_distribution.png`, 2016.

[3] Francisco Vargas, Pierre Thodoroff, Austen Lamacraft, and Neil Lawrence. Solving Schrödinger bridges via maximum likelihood. *Entropy*, 23(9), 2021.

[4] HarisIqbal88. Plotneuralnet. `https://github.com/HarisIqbal88/PlotNeuralNet`, 2018.

[5] Erwin Schrödinger. Uber die Umkehrung der Naturgesetze. *Sitzungsberichte der Preuss Akad. Wissen. Berlin, Phys. Math. Klasse*, pages 144–153, 1931.

[6] Valentin De Bortoli, James Thornton, Jeremy Heng, and Arnaud Doucet. Diffusion Schrödinger bridge with applications to score-based generative modeling. *arXiv preprint arXiv:2106.01357*, 2021.

[7] Tianrong Chen, Guan-Horng Liu, and Evangelos A Theodorou. Likelihood training of Schrödinger bridge using forward-backward SDEs theory. *arXiv preprint arXiv:2110.11291*, 2021.

[8] Charlotte Bunne, Ya-Ping Hsieh, Marco Cuturi, and Andreas Krause. Recovering stochastic dynamics via gaussian schr\" odinger bridges. *arXiv preprint arXiv:2202.05722*, 2022.

[9] Jian Huang, Yuling Jiao, Lican Kang, Xu Liao, Jin Liu, and Yanyan Liu. Schrödinger-Föllmer sampler: Sampling without ergodicity. *arXiv preprint arXiv:2106.10880*, 2021.

[10] Gefei Wang, Yuling Jiao, Qian Xu, Yang Wang, and Can Yang. Deep generative learning via Schrödinger bridge. *arXiv preprint arXiv:2106.10410*, 2021.

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

[12] Johannes Klicpera, Marten Lienen, and Stephan Günnemann. Scalable optimal transport in high dimensions for graph distances, embedding alignment, and more. In *International Conference on Machine Learning*, pages 5616–5627. PMLR, 2021.

[13] Arthur Gretton, Karsten M Borgwardt, Malte J Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

[14] Aaditya Ramdas, Nicolás García Trillos, and Marco Cuturi. On Wasserstein two-sample testing and related families of nonparametric tests. *Entropy*, 19(2):47, 2017.

[15] John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*, 2018.

[16] Carsten Hartmann, Ralf Banisch, Marco Sarich, Tomasz Badowski, and Christof Schütte. Characterization of rare events in molecular dynamics. *Entropy*, 16(1):350–376, 2013.

[17] Margaret Duff, Neill DF Campbell, and Matthias J Ehrhardt. Regularising inverse problems with generative machine learning models. *arXiv preprint arXiv:2107.11191*, 2021.

[18] Yang Song, Liyue Shen, Lei Xing, and Stefano Ermon. Solving inverse problems in medical imaging with score-based generative models. *arXiv preprint arXiv:2111.08005*, 2021.

[19] Christian Wachinger, Polina Golland, Martin Reuter, and William Wells. Gaussian process interpolation for uncertainty estimation in image registration. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 267–274. Springer, 2014.

[20] Haitao Liu, Yew-Soon Ong, Xiaobo Shen, and Jianfei Cai. When gaussian process meets big data: A review of scalable gps. *IEEE transactions on neural networks and learning systems*, 31(11):4405–4423, 2020.

[21] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[22] Richard Durrett. *Probability: Theory and examples*. Cambridge University Press, 2020.

[23] David J. C. Mackay. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, 2003.

[24] Jochen Gortler, Rebecca Kehlbeck, and Oliver Deussen. A visual exploration of gaussian processes. *Distill*, 2019. https://distill.pub/2019/visual-exploration-gaussian-processes.

[25] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[26] Imre Csiszár. I-divergence geometry of probability distributions and minimization problems. *The annals of probability*, pages 146–158, 1975.

[27] Robert Fortet. Résolution d'un systeme d'équations de M. Schrödinger. *J. Math. Pure Appl. IX*, 1:83–105, 1940.

[28] Solomon Kullback. Probability densities with given marginals. *The Annals of Mathematical Statistics*, 39(4):1236–1243, 1968.

[29] Andreas Ruttor, Philipp Batz, and Manfred Opper. Approximate Gaussian process inference for the drift function in stochastic differential equations. In *Advances in Neural Information Processing Systems*, pages 2040–2048, 2013.

[30] Omiros Papaspiliopoulos, Yvo Pokern, Gareth O Roberts, and Andrew M Stuart. Nonparametric estimation of diffusions: a differential equations approach. *Biometrika*, 99(3):511–531, 2012.

[31] Xitong Yang. Understanding the variational lower bound, 2017.

[32] Michele Pavon, Esteban G. Tabak, and Giulio Trigila. The data-driven Schrödinger bridge. *arXiv preprint*, 2018.

[33] Joseph L Doob. The Brownian movement and stochastic equations. *Annals of Mathematics*, pages 351–369, 1942.

[34] Jonathan Goodman. Lecture notes in sdes, 2007.

[35] Youngmin Cho and Lawrence Saul. Kernel methods for deep learning. *Advances in neural information processing systems*, 22, 2009.

[36] Adrià Garriga-Alonso, Carl Edward Rasmussen, and Laurence Aitchison. Deep convolutional networks as shallow gaussian processes. *arXiv preprint arXiv:1808.05587*, 2018.

[37] Christopher Williams and Matthias Seeger. Using the nyström method to speed up kernel machines. *Advances in neural information processing systems*, 13, 2000.

[38] Nicholas J Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002.

[39] Michalis Titsias. Variational learning of inducing variables in sparse gaussian processes. In *Artificial intelligence and statistics*, pages 567–574. PMLR, 2009.

[40] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. *Advances in neural information processing systems*, 18, 2005.

[41] James Mercer. Xvi. functions of positive and negative type, and their connection the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character*, 209(441-458):415–446, 1909.

[42] Harold V Henderson and Shayle R Searle. On deriving the inverse of a sum of matrices. *Siam Review*, 23(1):53–60, 1981.

[43] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20, 2007.

[44] Mark Van der Wilk, Carl Edward Rasmussen, and James Hensman. Convolutional gaussian processes. *Advances in Neural Information Processing Systems*, 30, 2017.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[46] Radford M Neal. Priors for infinite networks. In *Bayesian Learning for Neural Networks*, pages 29–53. Springer, 1996.

[47] Christopher Williams. Computing with infinite networks. *Advances in neural information processing systems*, 9, 1996.

[48] Alexander G de G Matthews, Mark Rowland, Jiri Hron, Richard E Turner, and Zoubin Ghahramani. Gaussian process behaviour in wide deep neural networks. *arXiv preprint arXiv:1804.11271*, 2018.

[49] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.

[50] John Wieting and Douwe Kiela. No training required: Exploring random encoders for sentence classification. *arXiv preprint arXiv:1901.10444*, 2019.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[52] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[53] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[54] Krzysztof Choromanski, Mark Rowland, Tamás Sarlós, Vikas Sindhwani, Richard Turner, and Adrian Weller. The geometry of random features. In *International Conference on Artificial Intelligence and Statistics*, pages 1–9. PMLR, 2018.

[55] Peter Guttorp and Tilmann Gneiting. Studies in the history of probability and statistics xlix on the matérn correlation family. *Biometrika*, 93(4):989–995, 2006.

Appendix A

Additional implementation concerns

A.1 Numerically stable drift fitting

Numerical stability is a common issue in machine learning tasks: the nature of certain algorithms can lead to floating point errors. GP regression is no different: with certain kernels, the features and targets can get extremely large. To mitigate this somewhat, we can consider solving a different objective.

$$\mathbf{X}_{t+1} - \mathbf{X}_t = f(\mathbf{X}_t)\Delta t + \gamma_t \sqrt{\Delta t}\boldsymbol{\epsilon} \tag{A.1}$$

That is, we do not divide the targets by Δt: since Δt is often small ($O(10^{-2})$), this can help with numerical stability, particularly when kernel values are unbounded (e.g. for the arc-cosine kernel, which scales with magnitude of the input vectors).

A.2 EMD implementation

As discussed in section 4.1.2, I use EMD as a sample-based distributional distance metric. KL divergence is unsuitable because we only have samples, and not the probability density at each point; but EMD lends itself to the samples approach through its formulation. Informally, EMD is the minimum cost of transporting a pile of material from one distribution to the other (where "cost" is defined as amount of material times distance moved). The mathematical definition is unwieldy, but the samples case is simpler: we effectively want the optimal way of mapping each sample in the first distribution to a unique sample in the second. This can be framed elegantly as the weighted bipartite matching problem: we treat the two distributions of samples as partitions in a bipartite graph, and assign edges between $p \in \pi_0$ and $p' \in \pi_1$ the cost $\|p - p'\|$ (Euclidean distance). We can then use a pre-implemented linear assignment algorithm (implemented in `scipy` [21]) to compute the optimal cost, which gives us the EMD.

A.3 Fourier transforms of RBF and Exponential kernels

For the random Fourier features approach, I had to derive the Fourier transforms of the exponential and RBF kernels. For the RBF kernel, which has a Gaussian density, this is fairly

simple: the Fourier transform of a Gaussian distribution is just another Gaussian, with an inverted scale parameter. This is reflected in the implementation I use: the original kernel's lengthscale ℓ is the "σ" we want to invert, so we return a new Gaussian with covariance $\frac{1}{\ell^2}\mathbf{I}$.

The exponential kernel has a more interesting Fourier transform [54, 55]: it maps to the multivariate Cauchy distribution. The multivariate Cauchy distribution is analogous to the multivariate Student's t-distribution for the case degrees of freedom = 1; and this t-distribution can in turn be expressed either in terms of a chi-squared and normal or a gamma and normal distribution. I used the gamma and normal approach: sampling g from a $\Gamma(\frac{1}{2}, 2)$ distribution and g from a $\mathcal{N}(0, \sigma)$, and returning $\mu + \frac{z}{\sqrt{g}}$.

A.4 UNet architecture used

After experimenting with the UNet, aiming to balance sufficient depth (for predictive power) with fast training times, I settled on the architecture below.

- Input dimension 28×28
- 3×3 Conv (channels $1 \to 128$); ReLU; 3×3 Conv (channels $128 \to 128$)
- 2×2 MaxPool
- 3×3 Conv (channels $128 \to 256$); ReLU; 3×3 Conv (channels $256 \to 256$)
- 2×2 UpConv (channels $256 \to 128$)
- Residual concatenation (channels $128 \to 256$)
- 2×2 UpConv (channels $256 \to 128$); ReLU; 2×2 UpConv (channels $128 \to 128$)
- 1×1 Conv (channels $128 \to 1$)

For the NN-as-GP experiments, I replaced the last 1×1 Conv with:

- 1×1 Conv (channels $128 \to 1000$)
- 1×1 Conv (channels $1000 \to 1$)

Appendix B

Additional image generation results

For completeness, I include a sample of generated MNIST images using a lower σ to that used in the Evaluation section. Lower σ leads to more variance in the predictions: generating more unique MNIST images, but of arguably lower quality. This was not discussed in the dissertation (due to space constraints, and since the extension objectives had already been met); but it is an interesting trade-off for future work to investigate.

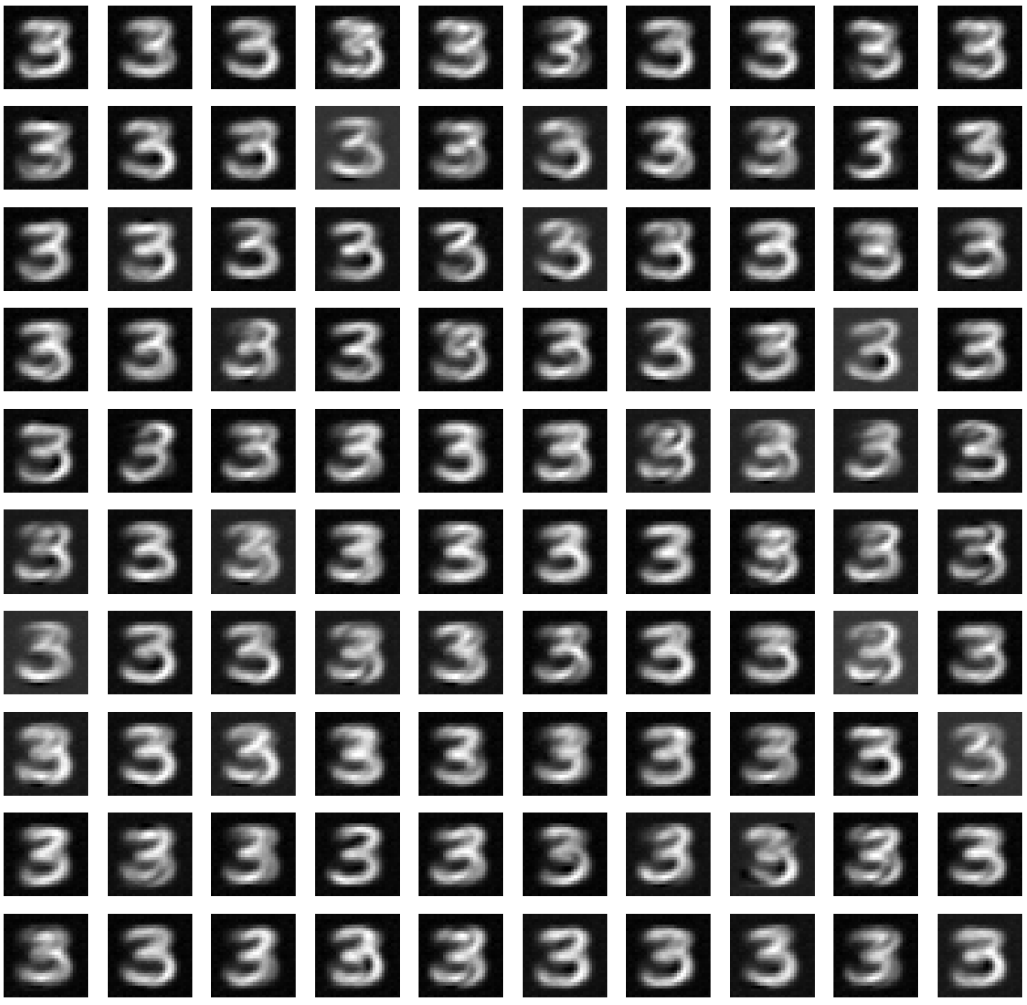


Figure B.1: MNIST images generated by random-features convolutional kernel

Appendix C

Project Proposal

Introduction, background, description of work

This project aims to scale up kernel methods and Gaussian Processes (GPs) to solving the Schrödinger Bridge problem for larger datasets. The current GP implementation (by my supervisor's group) works on low-dimensionality, small datasets, but is infeasible for larger ones; this project aims to use techniques like random Fourier features (RFFs) or sparse GPs (SGPs) to scale the GP methods up to image datasets, like MNIST.

The Schrödinger Bridge problem is that of finding the most likely evolution between two probability distributions, given a reference prior evolution (for instance, finding the most likely drift function given a Brownian motion prior). It has many uses, such as in image generation – where a bridge between multivariate normal noise and images can be used to generate similar images from new random noise. The trajectory of the bridge can be obtained, which is often more useful than the "black box" generative adversarial network and variational auto-encoder approaches. More useful applications include bioinformatics and molecular dynamics (specifically, protein folding).

Although the problem dates back to 1932, only very recently have computational methods been developed. Most such methods are diffusion methods, which aim to predict a function $f*$ from the drift. The standard way of doing this is using a neural network; a group at Oxford University (Bortoli, Thornton, Heng, & Doucet, 2021) researches these techniques. However, the neural network approach has some disadvantages, including instability and overfitting, a lack of theoretical guarantees, and lack of uncertainty quantification. Gaussian Processes ameliorate all of these, at the cost of greater computational complexity; this project aims to use approximate methods to reduce this cost at higher dimensions and with larger datasets, to allow GPs' advantages to fully manifest themselves. Despite GPs' success in classification and dimension reduction tasks on MNIST before, they have yet to be successfully applied to generating MNIST images: this would be a novel contribution in the field.

The aim is to scale the current GP techniques up to a level where they can compute bridges from random noise to MNIST images (and vice-versa) in reasonable time; they can then be used to generate MNIST-like images from random noise. We can test generated images for similarity to MNIST, using, for instance, inception score – the proportion of generated images which are correctly classified by a well-established, well-trained classifier.

Note that several other methods exist for generating images, and (aforementioned) neural network implementations exist for solving the Schrödinger Bridge problem; we do not aim to outperform these methods on image generation. We are instead motivated by more far-reaching contributions, like applications in the physical sciences, which scaling up the GP approach for Schrödinger Bridges to this size of dataset offers.

The main techniques used to scale up GPs will be sparse GPs (Titsias, 2009) and random Fourier features (Rahimi & Recht, 2007). A lot of the project will involve speeding up the `driftFit` function, which is called on each iteration of the "iterative proportional fitting procedure" (IPFP) used to compute Schrödinger Bridges. The role of the GP is to estimate the drift (vector field) of the optimal diffusion that characterizes the Schrödinger Bridge. There will be a lot of experimentation with different techniques, such as using baseline SGP and RFF implementations and choosing which to pursue further; as a result, quantitative compute requirement estimates are hard to judge at this point.

Most development will be in Python. Tools and techniques used will likely include memory and time profiling, optimization using PyTorch, running models on a GPU for speedups, exploring manual memory management, visualisation and animation-handling infrastructure, machine learning modelling, loggers, numpy, perhaps a neural network baseline, and many others. Specifics of tooling will become clear at relevant points, but initial consideration has taken place: for instance, naïve benchmarking may be done with general tools like pytest benchmark, with Torch profiling used for specific machine learning aspects.

Starting point

I will be building on a codebase written by my supervisor (Vargas, 2021), containing:

1) A naïve implementation of the `driftFit` function I will optimize;
2) An example notebook visualizing drift trajectories.

I will also be using the python `pyro` library for Gaussian processes (which is already used in my supervisor's codebase, including for the current `driftFit` function) – this library provides a naïve sparse GP implementation, but nothing for Random Fourier Features.

Evaluation criteria

The aim of the project is to scale up certain algorithms, in terms of both memory and time requirements; and the output of the project will be generated MNIST-like images. Consequently, the evaluation will be based on both time/memory benchmarking and quality of generated images. Specifically, the base evaluation criteria are:

1) With the approximate method chosen (SGPs or RFFs), obtain similar results to the full GP implementation on toy examples (naturally, performance will degrade slightly due to the approximation)
    a. Ensure performance is not excessively degraded using a distributional distance metric, e.g. KL divergence (a quantitative estimate for this is meaningless, as these metrics have no fixed scale)
2) Train time will likely increase using the approximate methods, but test time (time taken to compute the Schrödinger Bridge) should decrease
    a. For a few different sizes of dataset, from toy examples up to MNIST, check that test time is reduced by using the approximate methods
3) For a given memory budget, we want to perform better than the baseline full GP implementation
    a. Obtain better distributional difference, HSIC (Sejdinovic, Sriperumbudur, Gretton, & Fukumizu, 2013) and CKA (Kornblith, Norouzi, Lee, & Hinton, 2019) scores using the sparse GP method than using a discretization dt (timestep) which gives the same memory reduction
4) Check that the MNIST generated images are close to real MNIST images
    a. Compute inception score and HSIC/CKA metric for the generated images
    b. These scores should be higher than for the following two baseline models (early generative models):

   i. Variational auto-encoders (Kingma & Welling, 2013)
   ii. Generative adversarial nets (Goodfellow, et al., 2014)
5) Compare benchmarks between naïve RFF and SGP approaches to argue which is more effective

Extension criteria include:

1) Generating more complex images: such as those from fashion-MNIST (as a first step) and then CIFAR-10
   a. Be able to generate images of any quality (computational leap)
   b. Obtain inception score above that of baseline models
2) Consider variational Fourier features as a blend of SGPs and RFFs
3) Consider whether JAX provides additional implementation speedups (I would have to implement GP support)
4) Refactor supervisor's codebase to improve class hierarchy and code structure

Plan of work, specifying a timetable and milestones

The below has been planned taking into account my course load from lectures and units of assessment. For instance, my Lent lecture timetable is far more packed than Michaelmas, so I plan to do most implementation in Michaelmas; and my Michaelmas unit of assessment starts in week 5 and has one big deadline in week 8, so I leave some flexibility around then.

Dates	**Work to be done**
18 Oct – 7 Nov	Learn about relevant theory: probability theory, stochastic differential equations, Gaussian Processes, programming in Torch, etc. Research memory and computational complexities (in terms of inducing points) of RFFs and SGPs. Familiarise myself with the existing codebase, and start running some simple baseline approaches e.g. using SGPs from pyro on the notebook to see the computed trajectories and trying toy examples.
8 Nov – 28 Nov	Experiment with `driftFit` function provided (which is in 1 dimension): try to scale up the method provided to greater # points and check memory and time requirements. Implement the baseline SGP and make improvements to scale them up. This is where a lot of the key algorithmic work comes in – finding ways to scale up using e.g. inherent structure of timeseries, sampling, etc. In this step, SGPs are not applied directly to the Schrödinger Bridge problem, but to the sub-problem of fitting the drift for a single iteration in IPFP. **Milestone: SGP/RGG implementation for driftFit.**

29 Nov – 5 Dec	Plug the successful SGP/RFF implementation for drift fit into the full Schrödinger Bridge problem – still on low dimensionality toy examples, to debug and verify. **Milestone: solve full Schrödinger Bridge problem on toy examples.**
6 Dec – 19 Dec	*Holiday – unable to work*
20 Dec – 16 Jan	Vacation – assess progress so far and sort out any loose ends. Read about efficient programming techniques in preparation for extending to MNIST. Read about metrics like HSIC and CKA, and how they can be implemented. Slack time for bring project up to date: ambitious Michaelmas timetable, so one of the earlier steps may take longer than planned. Start writing infrastructure for extending to MNIST. Start dissertation: learn LaTeX, write section headers and start introduction section. **Milestone: be fully prepared for next sections; complete all loose ends from Michaelmas (solve full Schrödinger Bridge problem on toy examples.)**
17 Jan – 13 Feb	Extend results to MNIST: a much larger dataset with high dimensionality. Several issues will likely crop up, which will require debugging and speeding up through various techniques. Firstly, this will need to be run on a GPU, which will bring its own technical challenges; we might then run into memory issues requiring manual memory management intervention, bugs to fix, grid-searches to find optimal parameters, tuning of results, etc. Write progress report and make presentation. Continue writing dissertation: preparation section. Adapt timeline as appropriate, given progress up to this point. **Milestone: dissertation section 2 draft; progress report; generate MNIST images**
14 Feb – 27 Feb	Implement metrics; write benchmarking infrastructure; conduct experiments as described in the evaluation section. Continue writing dissertation, now having a sketch of the entire document. **Milestone: dissertation sketch, benchmark times for MNIST images**
28 Feb – 13 Mar	On the basis of benchmarking experiments, make further speedups to image generation framework; benchmark each attempt. If ahead of schedule, start trying some extension objectives.

	Continue dissertation. **Milestone: all core objectives completed; implementation section on dissertation sketched.**
14 Mar – 27 Mar	Finish off term lecture + unit of assessment work – might be little time to work on project. Discuss overall progress with supervisor and consider direction of project so far; adapt timeline if necessary, reconsider extension objectives and judge which to attempt.
28 Mar – 24 Apr	(Vacation.) Fill in details of the dissertation – flesh out each section to completion, write final evaluation section, review initial goals and judge success. Try to implement further extension objectives. Continual feedback loop with supervisor; also send dissertation to DoS for feedback. **Milestone: some extensions tried; dissertation completed, feedback received.**
25 Apr – 12 May	Slack time for finishing loose ends, implementation and dissertation. **Milestone: submit dissertation by 13 May!**

Resource declaration

I will primarily use my own laptop and the MCS Linux machines for my project. I will require some GPU computation to run ML models, preferably with around 16GB of memory; if the department cannot directly provide me with this, my supervisor is happy to ask for me to be added to his machine (so I can share his GPU resources).

I accept full responsibility for my machine and I have made contingency plans to protect myself against hardware and/or software failure, including periodic backups to Google Drive and Git.

References

Bortoli, V. D., Thornton, J., Heng, J., & Doucet, A. (2021). Diffusion Schrödinger Bridge with Applications to Score-Based Generative Modeling.

Goodfellow, I. J., Jean Pouget-Abadie, M. M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Networks .

Kingma, D. P., & Welling, M. (2013). Auto-Encoding Variational Bayes .

Kornblith, S., Norouzi, M., Lee, H., & Hinton, G. (2019). *Similarity of Neural Network Representations Revisited*.

Nieman, D., Szabo, B., & Zanten, H. v. (2021). Contraction rates for sparse variational approximations in Gaussian process regression.

Rahimi, A., & Recht, B. (2007). Random Features for Large-Scale Kernel Machines.

Sejdinovic, D., Sriperumbudur, B., Gretton, A., & Fukumizu, K. (2013). Equivalence of Distance-based and RKHS-based Statistics in Hypothesis Testing.

Titsias, M. (2009). Variational Learning of Inducing Variables in Sparse Gaussian Processes.

Vargas, F. (2021). *GP_Sinkhorn*. Retrieved from https://github.com/franciscovargas/GP_Sinkhorn