COMPUTATIONAL EFFICIENCY OF INFERENCE METHODS FOR STOCHASTIC DIFFERENTIAL EQUATION DYNAMICS

ALAN AMIN

1. Introduction. Stochastic differential equations with respect to a Wiener process (SDEs) are useful implicit descriptions of certain Markov processes. They have been applied to modelling financial markets [5, 1], projecting the paths of spacecraft [1], and describing biological phenomenon [8]. They also can be used to build stochasticity into state space models [9].

Inference of SDEs has remained a challenge - in particular, the task of computationally efficient calculation of gradients of a loss function. Early methods included Monte Carlo forward differentiation of parameters of the drift and noise functions, however, these scale poorly with the number of parameters, which can be very high when these functions are high dimensional or if a deeper or wider neural network is necessary for example. A more computationally efficient method in this case is Monte Carlo backpropogation through the SDE solver given a realization of the Wiener process [**?**]. However, this method requires one to save the forward path of the SDE which can be memory intensive for long time series. Adjoint methods instead define a backwards SDE that one may solve to calculate gradients. Early adjoint methods considered each step of an Euler-Maruyama SDE solver a constrained optimization problem and were able to define the backwards SDEs by taking derivatives: one effort defined the backwards SDE for use in calculating monte carlo estimates of the gradient [6], while another used quadrature to approximately integrate over all paths [2]. Recently, the adjoint method has been extended for the use of any Stratonovich SDE solver [9].

Here I explore these ideas by implementing a high performance version of both this later adjoint method, and backpropogation. To achieve low level control of the SDE solver, I implement my own solver using the Euler-Heun method. I compare the accuracy of the gradients of this adjoint method, and backpropogation as well as benchmark their computational cost and ability to fit data. I will demonstrate the convergence of my solver and gradient estimates to the correct values. Then I will show that my implementation of gradient estimation using backpropogation is more computationally efficient and results in higher quality inference than my implementation of the adjoint method for gradient calculation. This investigation represents my edification in the solving and inference of SDEs, as well as an investigation into the factors that effect the utility of these two gradient estimation methods.

2. Results.

2.1. Euler-Heun method solver.

2.1.1. Setup. I consider here the SDE, from time 0 to T,

$$dX_t = f(X_t, t)dt + g(X_t, t) \circ dW_t$$

where X_t has d dimensions and the Wiener process W has m dimensions. To investigate how the architecture of an SDE solver may affect the performance of gradient estimation techniques, and to easily implement backpropogation, I implement my own SDE solver.

The naive implementation of a solver for a Stratonovich integral is known as the Euler-Heun. In probability, as the mesh size of the grating $0 = t_0 < \cdots < t_n = T$

goes to 0,

$$\sum_{i=0}^{n-1} \frac{g(X_{t_{i+1}}, t_{i+1}) + g(X_{t_i}, t_i)}{2}(W_{t_{i+1}} - W_{t_i}) \to \int_0^T g(X_t, t) \circ dW_t.$$

The Euler-Heun (EH) method SDE solver approximates the integral for a time step δt via a second order Runga-Kutta (RK) method to approximate the average of the drift and diffusion evaluated at the start and stop time of each step:

$$\begin{aligned} H_{i+1} &= X_i + f(X_i, t_i)\delta t + g(X_i, t_i)\Delta W_i \\ X_{i+1} &= X_i + \frac{\delta t}{2}\left(f(X_i, t_i) + f(H_{i+1}, t_{i+1})\right) \\ &\quad + \frac{1}{2}\left(g(X_i, t_i) + g(H_{i+1}, t_{i+1})\right)\Delta W_i \\ \Delta W_i &\sim \sqrt{\delta t} N(0, 1). \end{aligned}$$

The EH method is strong order 0.5 - if $X_T^{\delta t}$ is approximated using the EH method with step size δt, then

$$\|X_T^{\delta t} - X_T\|_{L^2} \lesssim \delta t^{0.5}. \tag{2.1}$$

2.1.2. Results. Throughout, I will use one layer neural networks with tanh activation as the drift and diffusion functions f and g. As well, I will consider a two dimensional SDE with 2 dimensions of diffusion, i.e. $d = m = 2$. g will not be assumed diagonal or commutative. To allow for efficient evaluation of the neural networks, I used the @einsum macro, which avoids boundschecking by checking compatibility of the dimensions of tensors before the multiplication. I used function barriers when unpacking parameters to allow the function to specialize on the input type (functions f_NN! and g_NN! in the code). Finally, to avoid repeated memory allocation for evaluations of intermediate values in the evaluation of f or g, I preallocate the memory to store these values; I then pass the variables associated to this memory as a parameter to the functions f_NN! and g_NN! and mutate it to these intermediate values (variables $t1$ and $t2$ in the code).

I implement a high performance EH solver for SDEs that takes as input, a starting point, a parametrized drift function, a parametrized diffusion function, and parameters (function my_SDEsolve! in code) (Fig 1A). To minimize memory allocation per step, I preallocated all the memory necessary to store intermediate values for each step and mutate these variables (function EH! in code).

One step of the solver, given the step size of the Wiener process, was benchmarked as having 0 memory allocations and taking 1.790 μs on my machine (section "Time Euler-Heun" in the code). Much of this computational complexity came from the need to apply the costly tanh non-linearity: the time was almost halved to 1.080 μs when using a ReLu activation. Below however, I only consider the use of the tanh non-linearity unless stated otherwise. To evaluate the convergence of my solver, I used the Euler-Heun solver implemented in DifferentialEquations.jl with step size 2^{-16} to approximate X_T (in this case I used $T = 1$). Since the solver had a fixed step size, it was easy to save its noise and use it in my solver to calculate $X_T^{\delta t}$ with step sizes δt ranging from 2^{-7} to 2^{-12} (section "EH is order 0.5" in the code). Empirical estimation of the L^2 distance reveal that my method converges to the correct solution and a linear regression reveals that it is of approximate order 0.522, similar to the theoretical 0.5 described in equation 2.1.

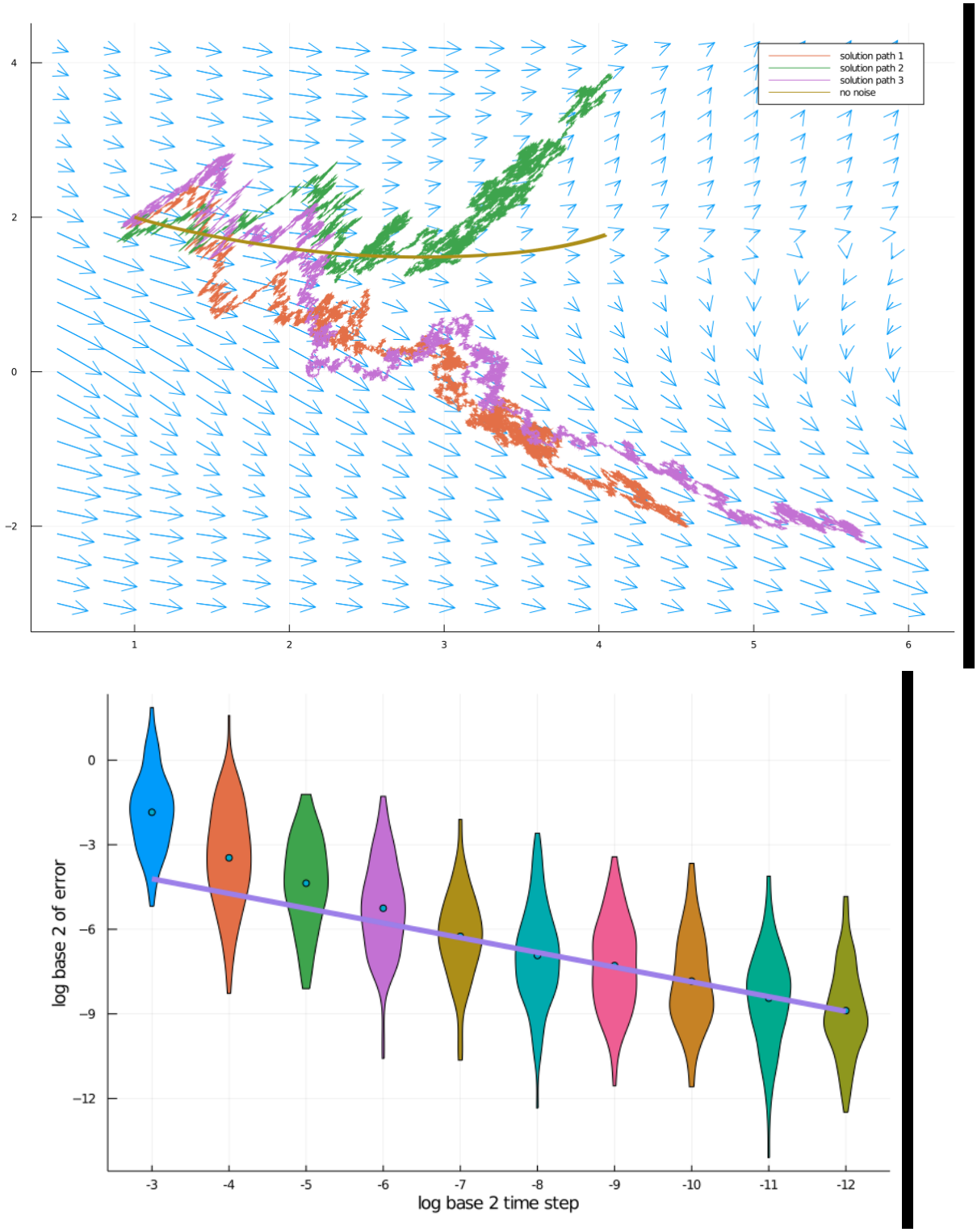

FIG. 1. *A: Example paths starting at the point (1, 2) given by my Euler-Heun solver given a instantiation of the drift (blue arrows) and diffusion functions. The solution to the ordinary differential equation without the drift is also shown. In this example, the stochasticity causes some paths to cross a separatrix and depart significantly from the ODE solution. B: Converge of the L^2 error defined in equation 2.1. A linear regression of the last 6 points gives a slope of -0.522 and an R^2 of 0.995, close to the theoretical -0.5. The violin plots contain 100 point each.*

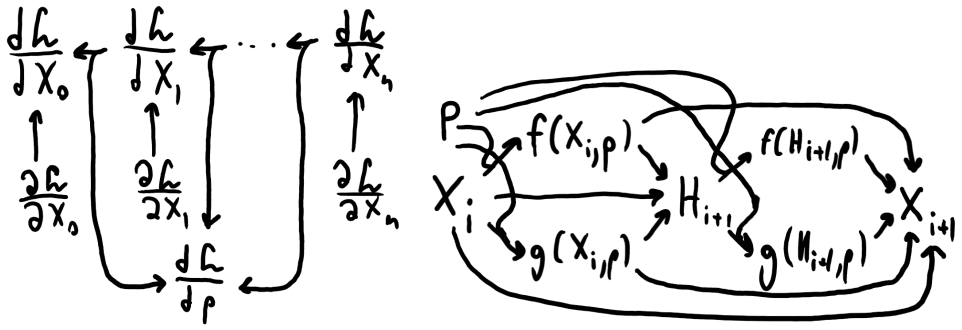

FIG. 2. *A: A schematic of the calculation of the gradient of the loss with respect to the parameters and starting point by backpropogation through he solver. B: The computation graph of a single Euler-Heun step. I reverse it to calculate the pullback.*

2.2. Backpropogation through the solver.

2.2.1. Setup. Here I consider the problem of approximating gradients of some loss function $\mathcal{L}$ by backpropogation through the solver. Say $(X_i)_{i=0}^n$ are the steps of the SDE iterator, i.e. $X_{i+1} = EH(X_i, p)$ where EH is one step of the solver. Call $\mathcal{B}^p_{EH(X,p)}$ the p component of the backpropogation of the Euler-Heun step and $\mathcal{B}^X_{EH(X,p)}$ the X component. Then

$$\frac{d\mathcal{L}}{dX_i} = \frac{\partial \mathcal{L}}{\partial X_i} + \mathcal{B}^X_{EH(X_i,p))}\left(\frac{d\mathcal{L}}{dX_{i+1}}\right) \tag{2.2}$$

$$\frac{d\mathcal{L}}{dp} = \sum_{i=1}^{n} \mathcal{B}^p_{EH(X_{i-1},p))}\left(\frac{d\mathcal{L}}{dX_i}\right) \tag{2.3}$$

(Fig 2A). To calculate $\mathcal{B}^X_{EH(X,p)}$ and $\mathcal{B}^p_{EH(X,p)}$ I consider the computation graph in fig 2B.

2.2.2. Results. To generate data to define a loss function, I initialize the parameters of f and g to random values and sample values from a path of the solution (section "Gen data" in the code). Bellow I consider the ℓ^2 loss with respect to this data.

When writing the pullback of the neural networks, I again used the @einsum macro (functions pullback_f! and pullback_g! in the code). When these functions were called, I again preallocate memory to store intermediate calculations and pass these variables as parameters. I also again use function barriers when unpacking parameters.

I implement a high performance pullback of the SDE solver that takes as input, the forward path, a parametrized drift function, a parametrized diffusion function, pullbacks of the drift and diffusion that add to the derivatives with respect to the parameters instead of overwrite them, and parameters (function my_SDEbackprop! in code).

One iteration of the Euler-Heun pullback, given the step size of the Wiener process, was benchmarked as having 0 memory allocations and taking 18.799 μs on my machine. The derivatives calculated by the backpropogation algorithm are validated by those calculated by finite differencing (section "Backprop" in code) (Fig 3). Small differences between the gradient calculated by backpropogation and that calculated

by finite differencing are likely due to the finite differencing algorithm only returning values up to a certain precision or precision loss when calculating gradients through a tanh function; in particular, the step size doesn't affect the agreement between the two methods.

2.3. Adjoint method.

2.3.1. Setup. I now implement the adjoint method of gradient estimation described in [9]. Call $\mathcal{B}_{f(X,p)}$ the pullback of the drift function and $\mathcal{B}_{g_i(X,p)}$ the pullback of the i-th column diffusion function. Consider the backwards SDE defined in [9]

$$d(A_t, B_t) = \mathcal{B}_{f(X,p)}(A_t) + \sum_{i=1}^{m} \mathcal{B}_{g_i(X,p)}(A_t) \circ d\widetilde{W}_{i,t}$$

With $A_T = \frac{\partial \mathcal{L}}{\partial X_T}$, $B_T = \vec{0}$ and A jumping at time t' by a magnitude of $\frac{\partial \mathcal{L}}{\partial X_{t'}}$. Then, by the results in [9], $A_0 = \frac{d\mathcal{L}}{dX_0}$ and $B_0 = \frac{d\mathcal{L}}{dp}$. It is also shown that the forward path X satisfies the backwards SDE

$$dX_t = -f(X_t, p)dt - g(X_t, p) \circ d\widetilde{W}_t. \tag{2.4}$$

Crucially, the stochastic integrals in these results are of the Stranovich interpretation. Thus, the EH method may be used to evaluate them.

Solving such a backwards SDE is identical to solving the forwards SDE

$$dY_t = -f(Y_t, p)dt - g(Y_t, p) \circ dV_t$$

where $Y_t = X_{T-t}$ and $V_t = -W_{T-t}$. The major difficulty in implementing this model is that the forwards and backwards noise must be the same. One solution for fixed step-size solvers is to save the path of the noise and reuse it when going backwards. [9] avoids this by implementing using what they call a Brownian tree. Here, a single random seed determines the value of the Wiener process path at all points $(i2^{-m})_{i=0}^{2^m}$ for a given m. The computation cost of querying the Brownian tree scales linearly with m.

2.3.2. Results. I use the implementation of the Brownian tree at http://github.com/SciML/DiffEqNoiseProcess.jl/pull/65.

To implement a high performance version of the adjoint method, I simply plugged in the pullback function for the neural networks f and g into my SDE solver (section "Backwards SDE" in the code). I show that the forward and reverse solutions to the SDE are the same as described by equation 2.4 (Fig 4). Furthermore, I show that the gradients of the adjoint method agree with those calculated by finite differencing, with the agreement becoming better as the step size used in the solver decreases, and thus the solution becomes more accurate (Fig 3).

At a step size of 2^{-12}, the Brownian tree algorithm takes 5.750 μs and makes 100 allocations of 10 kB. This is almost an order of magnitude more resources than a step of the solver. Since the Euler-Heun method is a fixed step size solver and saving the path does not incur a significant memory cost, hereafter, I opt instead to simply save the noise from the forward pass for use in the backwards pass. As well, instead of using the forward solution as calculated in the backwards pass, I simply save the forwards pass, incurring minimal memory cost, as pass this path to the backwards path; I expect this to increase the stability of the solver.

2.4. Comparing the adjoint method and backpropogation.

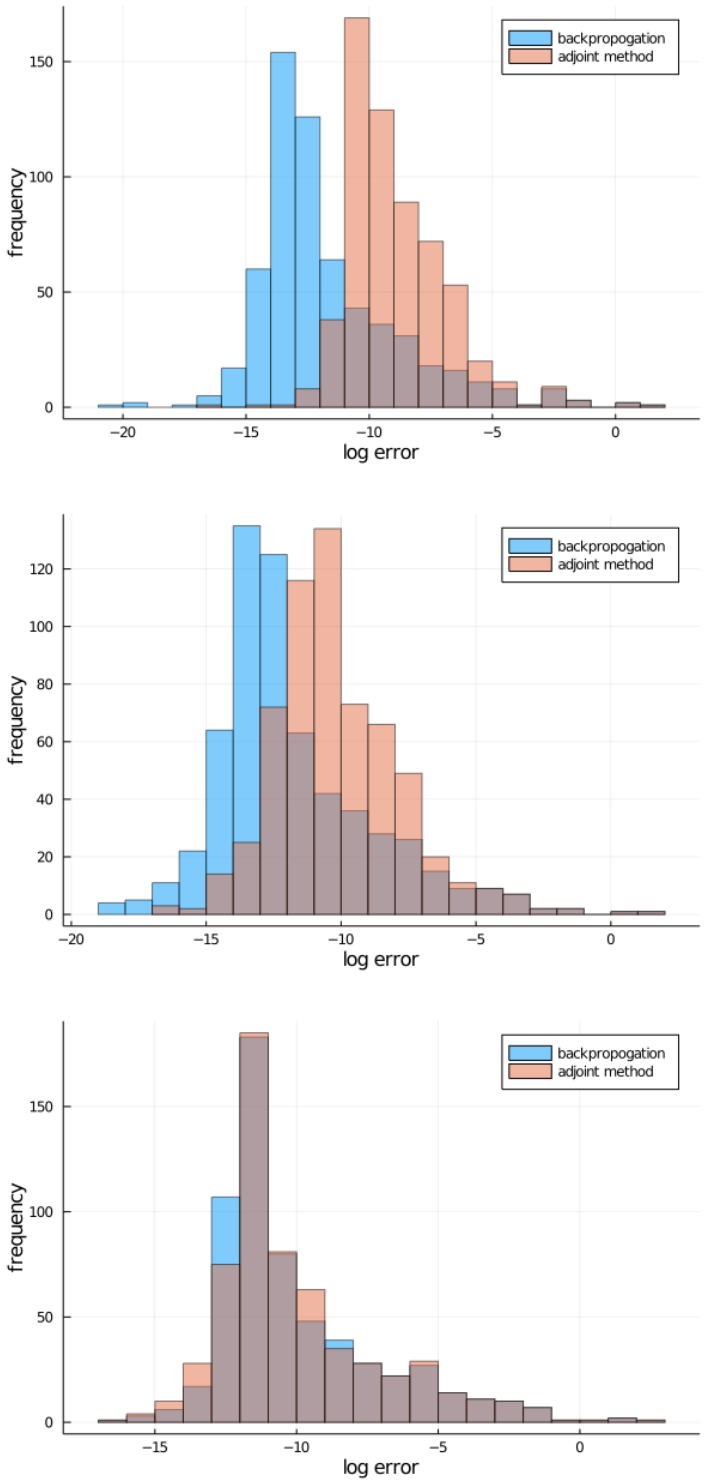

FIG. 3. *Fractional error of parameter gradients calculated using back propagation or the adjoint method in comparison to those calculated by finite differencing for step sizes A:* 2^{-10}*, B:* 2^{-12}*, C:* 2^{-14}*. These plots are each for a single realization of the Wiener process and plot all parameters of* f*,* g *and the starting point. Notice the* x *axis scaling is not the same between plots (sorry!).*

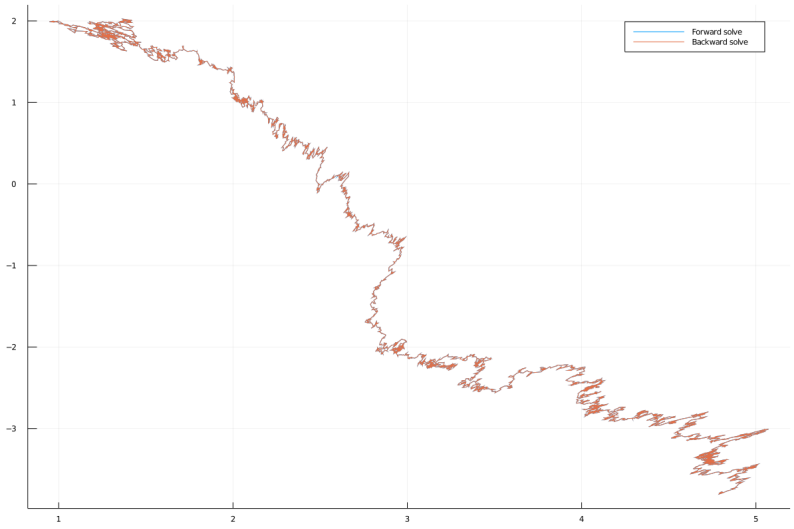

FIG. 4. *Shown here is the forwards and backwards solution to an SDE for a given realization of the Wiener process. The two paths are superimposed and thus visually identical.*

2.4.1. Performance. Here I will compare the computational complexity and accuracy of the gradient calculation by the adjoint method and by backpropogation through the solver.

For a step size of 2^{-12} I was able to optimize the backpropogation method to take 86.573 ms while the adjoint method took 554.085 ms with an order of magnitude more allocations. The most time consuming step in the later case is the calculation of the pullback of the diffusion function. In particular, my implementation updates the diffusion matrix column by column requiring one to take slices of the diffusion matrix. These views are allocated to the heap. This represents an opportunity for optimization by predefining the views and reusing them step to step.

As was shown above, both methods agree with parameter gradient estimated by finite differencing with the agreement increasing for the adjoint method as the step size decreased. The result is similar when comparing the gradients of the two methods directly (Fig 5A). In principle, the gradients calculated by finite differencing and backpropogation are not exactly the gradients of the loss function as the forward solve is inexact. However, there is no reason *a priori* to believe the adjoint method should have more accurate gradient estimates; in fact, one would expect that the error from the backwards solve add to that of the forwards solve to give an even less accurate estimate. Thus, those values in Fig 5A may be interpreted as lower bound for the error of the gradients. Clearly, due to the increased computational efficiency and more accurate gradient estimates, my implementation of backpropogation through the solver is superior to my implementation of the adjoint method.

As a curiosity, it is worth mentioning that path of $(A_t)_t$ as calculated by the adjoint method is very similar to $(\frac{d\mathcal{L}}{dX_{T-t}})_t$ (Fig 5B), although it is expected that these two paths have similar start and end points $(\frac{d\mathcal{L}}{dX_T})_t$ and $(\frac{d\mathcal{L}}{dX_0})_t$ respectively.

2.4.2. Inference. Here I will investigate how these two method perform in an inference task. I will attempt to minimize the ℓ^2 error to the data generated above via (stochastic) gradient descent (section "Training loop" in the code). I had better

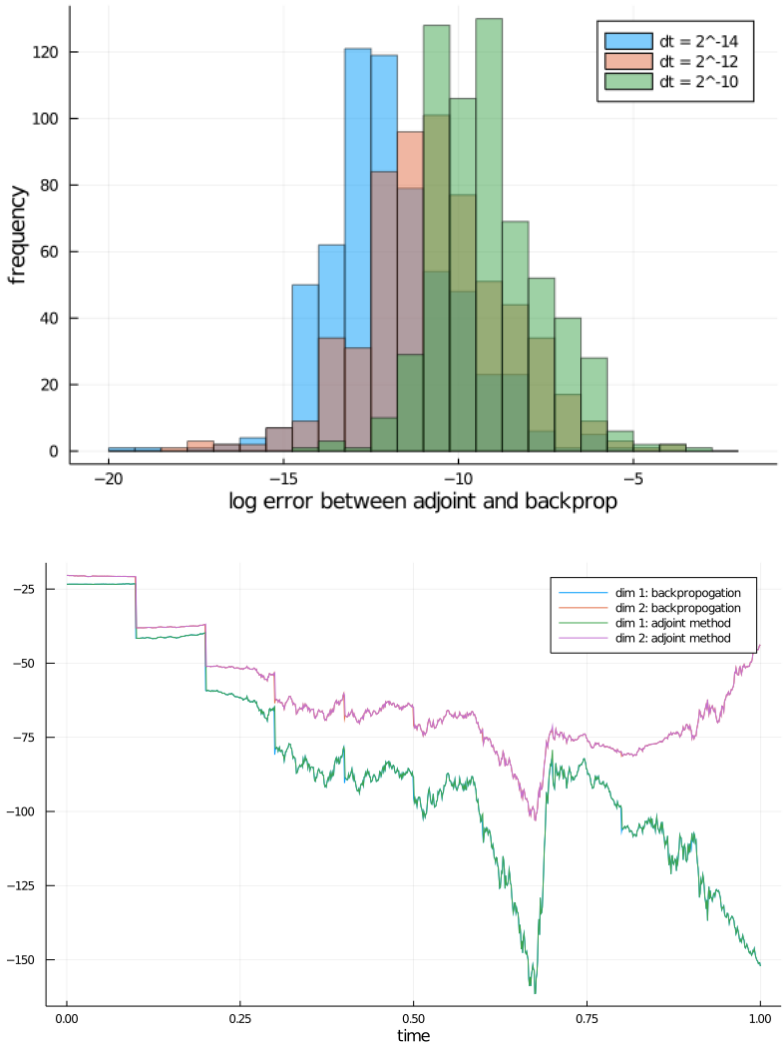

FIG. 5. *A: Fractional error of parameter gradients calculated using back propagation in comparison to those calculated by adjoint method for step sizes* 2^{-10}, 2^{-12}, 2^{-14}. *B: Two dimensions of the paths of the adjoints* A_t *and* $(\frac{d\mathcal{L}}{dX_{T-t}})_t$ *for a particular realization of the Wiener process. The two paths are superimposed and visually identical.*

results using a ReLu activation in the neural networks than tanh so below I will consider fitting f and g with ReLu activations. As well, inference of the starting point made the problem much more difficult, so in this setting, I considered the starting point known.

Because of the stochasticity of the gradients, inference was very difficult. Parameters would explode or the magnitude of g would stabilize at a large value if the learning rate was too high. To avoid divergence, it was necessary to average the gradients of the parameters over large batches of realizations of the Wiener process. Here I used a batch size of 100 and a learning rate of 0.0005. I also observed better results when initializing with f and g near 0 but with the values of the first hidden layer positive,

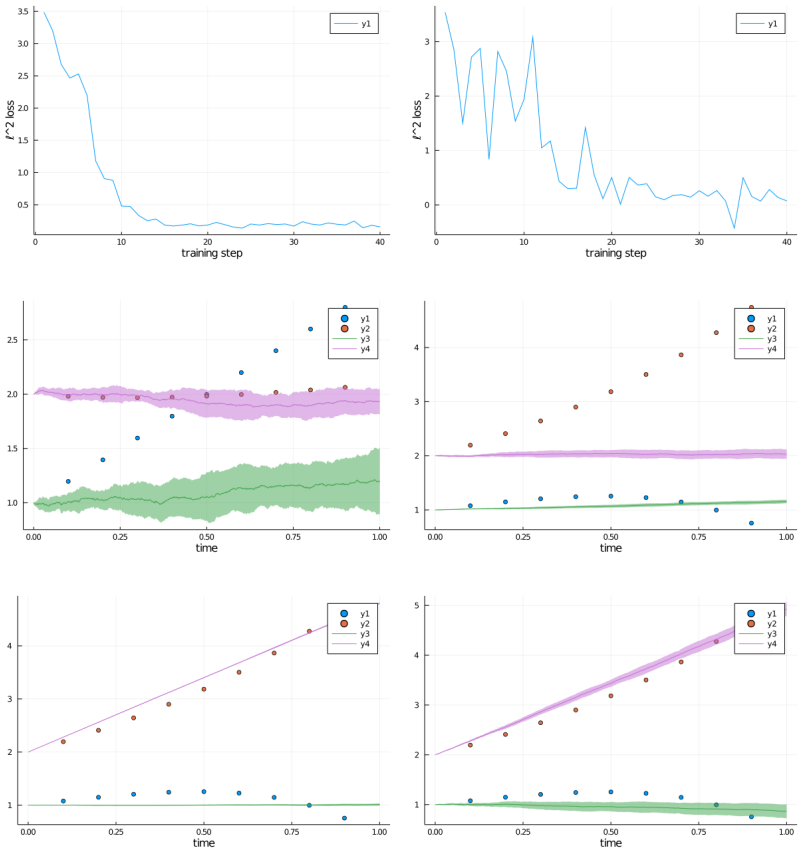

FIG. 6. *The first column shows inference using backpropogation through the solver to estimate gradients while the second shows the use of the adjoint method to estimate gradients. For each set of data, both models were trained for 40 steps, the first column at a learning rate of 0.0005 and the second at 0.0003. The first row shows the loss decreasing stochastically with increased training. The middle and last row show the data (points) and the average and standard deviation of 10 paths of the SDE; the middle row shows paths from the SDE with untrained f and g, and the last row uses the trained f and g.*

so as not to be set to 0 by the activation function.

I thorough comparison of the two methods would require tuning of the inference procedure which is outside the scope of this report. However, I noticed that when using gradients calculated by backpropogation, inference with the above parameters would usually converge while when using the adjoint method the sometimes diverged. Decreasing the learning rate slightly to 0.0003 usually resulted in convergence for the adjoint method. However, both methods were eventually able to converge to solutions (Fig 6). The solution arrived at when using the adjoint method had a higher magnitude g when stochasticity is not necessary to explain the data and had a much more stochastic loss curve; this is possibly a by-product of less accurate gradient estimates. Neither solution fit the data satisfactorily despite the loss plateauing, likely because of many of the nodes of the neural network being turned off by the ReLu activation. Further tuning of the architectures of f and g would likely lead to higher quality fits.

3. Discussion. Here I have implemented the EH SDE solver, backpropogation through the solver, and implemented the adjoint method of calculating integrals described in [9]; I was able to use these techniques to fit an SDE to data. My preliminary results show that the use of the Brownian tree represents a significant computational cost in the evaluation of gradients using the adjoint method, and that the more accurate gradient estimates of the backpropogation method resulted in higher quality inference.

While backpropogation through an EH step should in principle cost more function evaluations (one extra pullback of f) the particulars of my implementation meant that the backpropogation method was significantly more computationally efficient than the adjoint method for gradient calculation. To evaluate accurately the computational complexities of these models, a more careful implementation of the adjoint method is necessary.

Multiple ideas important to the comparison of the adjoint method and backpropogation were not explored in this report.

1) Most important is the selection of the SDE solver.

a) The EH method has fixed timesteps. This allowed me to save the forward path and noise at the time-points visited in the forwards pass for use in the backwards path, significantly decreasing the computational complexity of the adjoint method while likely increasing its stability. Many solvers use adaptive step sizes however to increase solution stability. This negates the benefit in stability and speed I saw in my analysis.

b) When efficiency is measured as number of calculations required to reach a certain error rate, the stochastic Runga-Kutta (SRK) method described in [12] leads to very efficient solvers; in particular, methods that are more efficient than the EH method. By running these SRK methods with two tableau, one may calculate estimated error to create adaptive SDE solvers as well [11]. These family of methods are more efficient given more stringent requirements on the form of the diffusion term g: the SRA methods are the most efficient, assuming additive noise (i.e. $g(x,t) = g(t)$); the SRID methods are less efficient, only assuming the diffusion function is diagonal, i.e. the noise in each dimension is independent; the SRIC methods are less efficient assuming a commutativity property; and finally, the general SRI methods, which make no assumption about the form of the noise, are least efficient as they must calculate iterated stochastic integrals [13]. The form of the noise for the backwards SDE may differ from that of the forwards, requiring a less efficient solver. In particular, when the forward SDE has diagonal noise, the backwards SDE is only guaranteed to have commutative noise. However, when the forwards SDE has additive noise, the pull back of the noise does as well. Thus it is possible that the adjoint method may only be optimal for certain SDEs.

c) Here I only considered simple neural SDEs without stiffness. Stiff SDEs require the application of solvers with large stability regions. To achieve this, implicit methods have often been used. Backpropogation through implicit methods is extremely costly. This fact has previously been described as an advantage of the adjoint method for gradient estimation [3]. However, recent work has also developed stability optimised SRK methods SOSRI and SOSRI2 [10]; these methods are stable enough to solve stiff SDEs, are significantly more computationally efficient than implicit methods, and are much easier to backpropogate through. As well, these methods are restricted to SDEs with diagonal noise however and thus cannot be used for the backwards pass of the adjoint method. This potentially puts the adjoint method at a disadvantage when handling stiff SDEs.

d) Here I considered an SDE that did not need to be run with small time steps for the solver to get accurate estimates of the solution. SDEs where smaller step sizes are necessary would increase the memory burden, making the adjoint method more appealing. However, adaptive methods that decrease the step size of the solver commensurately with the estimated error of the solver may decrease the number of steps in the path and thus incur a smaller burden on memory. As well, more computationally efficient methods, with fewer calls to the Wiener process and function evaluations would incur a smaller memory cost to the solver. Finally, not every intermediate calculation need be saved to implement backpropogation: some may be recalculated, such as H_{i+1} in my implementation of backpropogation.

2) One potential solution to the instability of inference of SDEs is regularization by penalizing the magnitude of higher order derivatives of the solutions or the derivatives of the dynamics as suggested in [4, 7]. While the former solution is not easily translatable to the SDE case where solutions are a.s. nowhere differentiable, the latter may lead to more stable inference. This could make the instability of the adjoint method less of a detriment during inference.

3) In [9] it was recommended that latent SDEs be trained using the evidence lower bound (ELBO) as an objective. This objective avoids the collapse of the noise during training as seen in Fig 6. To make sense of the ELBO, one must be able to compute the KL divergence to a prior. Using Girsanov's theorem, one may find the KL divergence between a prior SDE with the same diffusion, but possibly different drift, and the learned SDE (when the diffusions are different, it is possible that the supports of the laws of the solutions are non-overlapping and thus have divergence infinity - for example solutions for SDEs with different additive diffusions a.s. have paths with different quadratic variation). The Bayesian setup would normally require that one fix the prior, this would result in fixing the diffusion. Optimizing the ELBO with respect to the diffusion is, however, still mathematically possible ([9] section 9.7). How the treatment of the diffusion as a learnable function or hyper-parameter affects the quality of the latent representation is unexplored; it is thus possible that the adjoint method may be superior in the situation where one must only learn the drift.

Both methods of gradient calculation explored in this report have their theoretical advantages. However, an enumeration of use case and class of SDE is complex and results are subject to change as new SDE solvers are implemented. However, such challenges must be overcome to progress towards scalable inference of SDEs.

REFERENCES

[1] R. F. Bass, *Stochastic Processes*, Cambridge University Press, 2011.

[2] H. S. Bhat and R. W. Madushani, *Nonparametric adjoint-based inference for stochastic differential equations*, Proceedings - 3rd IEEE International Conference on Data Science and Advanced Analytics, DSAA 2016, (2016), pp. 798–807, https://doi.org/10.1109/DSAA.2016.69.

[3] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, *Neural Ordinary Differential Equations*, in NeurIPS, 2018, https://doi.org/10.1007/978-3-662-55774-7_3, https://arxiv.org/abs/arXiv:1806.07366v5.

[4] C. Finlay, J.-H. Jacobsen, L. Nurbekyan, and A. M. Oberman, *How to train your neural ODE: the world of Jacobian and kinetic regularization*, (2020), http://arxiv.org/abs/2002.02798, https://arxiv.org/abs/2002.02798.

[5] M. Giles and P. Glasserman, *Smoking adjoints: fast monte carlo greeks*, Risk, (2006), pp. 88–92, http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Smoking+adjoints:+fast+Monte+Carlo+Greeks{#}0.

[6] B. P. Gross, *Applications of the Adjoint Method in Stochastic Financial Modelling*, (2015).
[7] J. Kelly, J. Bettencourt, M. J. Johnson, and D. Duvenaud, *Learning differential equations that are easy to solve*, arXiv, 3 (2020), https://arxiv.org/abs/2007.04504.
[8] M. Kimura, *On the probability of fixation of mutant genes in a population.*, Genetics, 47 (1962), pp. 713–719.
[9] X. Li, T.-K. L. Wong, R. T. Q. Chen, and D. Duvenaud, *Scalable Gradients for Stochastic Differential Equations*, in Proceedings of the 23rdInternational Conference on Artificial Intelligence and Statistics (AISTATS), vol. 108, 2020, http://arxiv.org/abs/2001.01328, https://arxiv.org/abs/2001.01328.
[10] C. Rackauckas, *Stability-Optimized High Order Methods and Stiffness Detection for Pathwise Stiff Stochastic Differential Equations ods for Diagonal Noise SDEs*.
[11] C. Rackauckas and Q. Nie, *Adaptive methods for stochastic differential equations via natural embeddings and rejection sampling with memory*, Discrete and Continuous Dynamical Systems - Series B, 22 (2017), pp. 2731–2761, https://doi.org/10.3934/dcdsb.2017133.
[12] A. Rössler, *Runge-Kutta methods for Stratonovich stochastic differential equation systems with commutative noise*, Journal of Computational and Applied Mathematics, 164-165 (2004), pp. 613–627, https://doi.org/10.1016/j.cam.2003.09.009.
[13] M. Wiktorsson, *Joint characteristic function and simultaneous simulation of iterated Itô integrals for multiple independent Brownian motions*, Annals of Applied Probability, 11 (2001), pp. 470–487, https://doi.org/10.1214/aoap/1015345301.