Calibrating rough volatility models: a convolutional neural network approach

Henry Stone

Routledge
Taylor & Francis Group

Calibrating rough volatility models: a convolutional neural network approach

HENRY STONE*

Department of Mathematics, Imperial College London, London, UK

In this paper, we use convolutional neural networks to find the Hölder exponent of simulated sample paths of the rBergomi model, a recently proposed stock price model used in mathematical finance. We contextualise this as a calibration problem, thereby providing a very practical and useful application.

1. Introduction

The aim of this paper is to investigate whether a convolutional neural network can learn the Hölder exponent of a stochastic process, from a set of sample paths. Recall that a stochastic process $(X_t)_{t \geq 0}$ is Hölder continuous, with Hölder exponent $\gamma > 0$, if $|X_t - X_s| \leq C|t - s|^{\gamma}$ for all $s, t \geq 0$ and some constant C. Smaller values of the Hölder exponent γ correspond to 'rougher' sample paths. We shall use the terms Hölder continuity and Hölder regularity interchangeably.

Convolutional neural networks, referred to from here as CNNs, are known to be very powerful machine learning tools with a vast array of applications including (but of course not limited to) image classification (Back *et al.* 1997, Krizhevsky *et al.* 2012, Simonyan and Zisserman 2015); speech recognition (Deng *et al.* 2013, Abdel-Hamid *et al.* 2013) and self-driving cars (Chen *et al.* 2015, Iandola *et al.* 2017). Very recently, Bayer and Stemper (2018) used neural networks to learn implied volatility surfaces; the network is then used as part of a wider calibration scheme for options pricing. To the best of our knowledge, however, this paper is the first to explore the use of CNNs to predict the Hölder exponent of a given stochastic process.

There has been a resurgent interest in fractional Brownian motion and related processes within the mathematical finance community in recent years. In particular, Gatheral *et al.* (2018) carried out an empirical study that suggests that the log volatility behaves at short time scales in a manner similar to a fractional Brownian motion, with Hurst parameter

$H \approx 0.1$. This finding is corroborated by Bennedsen *et al.* (2017), who study over a thousand individual US equities and find that the Hurst parameter H lies in $(0, 1/2)$ for each equity. Both Bennedsen *et al.* (2017), Gatheral *et al.* (2018) use least square regression techniques to estimate the value of H. In this paper, we instead use a CNN to find the value of the Hölder exponent from simulated sample paths of the rBergomi model, with varying values for the Hölder exponent. The rBergomi model, which is introduced in section 3.1, has similar Hölder regularity properties as the fractional Brownian motion (Jacquier *et al.* 2018, Proposition 2.2).

The structure of the paper is the following. In section 2, we give a brief introduction to neural networks and the fractional Brownian motion. The methodology used in the paper is outlined in section 3. In section 4, we use CNNs to solve the regression problem of predicting the Hölder exponent, as a continuous value, for a given sample path. We hope to establish a robust means for calibrating rough volatility models; indeed, once the CNN has been trained we want it to perform well when making predictions on unseen data. Thus in section 5 we use the trained CNN to predict the value of the Hölder exponent on realised volatility data from financial markets; this provides a simple and accurate means of calibration. In section 6, we discuss our results and conclude the paper.

2. An introduction to neural networks and fractional Brownian motion

We begin with an introduction to neural networks, using (Kinh Gian Do *et al.* 2018) as our guide.
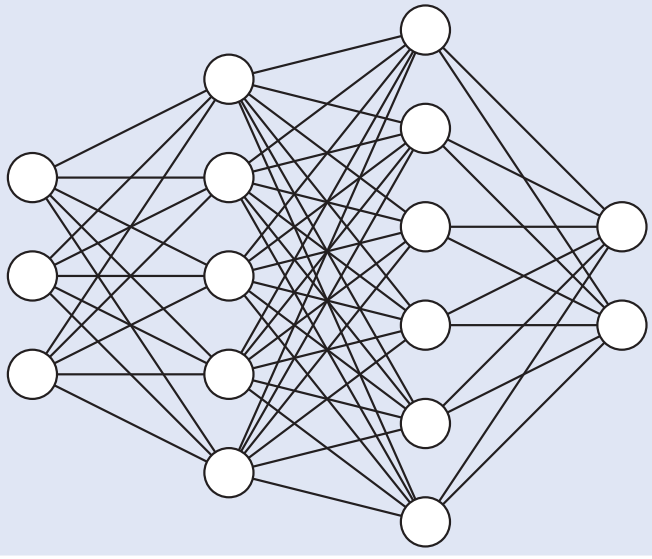
*Email: henry.stone15@imperial.ac.uk

Figure 1. An example of a neural network, with two hidden layers. The input layer has three nodes; the hidden layers have five and six nodes respectively; the output layer has two nodes.

2.1. Neural networks

An artificial neural network is a biologically inspired system of interconnected processing units, where each processing unit is called a layer. Inputs to each layer, apart from the first layer, are outputs from the previous layer. A layer is composed of a number of nodes, and each node in a given layer is connected to the nodes in a subsequent layer, thus forming a network; each edge in this network has a weight associated to it. The first processing unit is called the input layer, and the final processing unit is the output layer. The processing unit or units between the input layer and output layer are referred to as hidden layers; typically artificial neural networks have more than one hidden layer. Figure 1 illustrates the structure of a simple artificial neural network, drawn using Python, using code available on Github.† A formal, mathematical definition (Büler *et al.* 2018, Definition 4.1) of a neural network is given in Definition 2.1.

DEFINITION 2.1 Let $L \in \mathbb{N}$ denote the number of layers in the neural network. The dimension of each hidden layer is denoted by $N_1, \ldots, N_{L-1} \in \mathbb{N}$, and the respective dimension of the input and output layer is denoted by $N_0, N_L \in \mathbb{N}$. For $A^\ell \in \mathbb{R}^{N_{\ell-1} \times N_\ell}$ and $b \in \mathbb{R}^{N_\ell}$ let the affine function $W_\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$ be defined as $W_\ell(\mathrm{x}) := A^\ell \mathrm{x} + b^\ell$, for $\ell = 1, \ldots, L$. The entries of matrix A^ℓ are the weights connecting each node in layer $\ell - 1$ to layer ℓ. The neural network, with activation function σ, is then the function $\mathscr{N} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ defined as the composition

$$\mathscr{N}(\mathrm{x}) := W_L \circ (\sigma \circ W_{L-1}) \circ \cdots \circ (\sigma \circ W_1)(\mathrm{x}). \quad (1)$$

The learning process, also referred to as training, of an artificial neural network essentially boils down to finding the optimal weights in each matrix A^ℓ that minimise a given loss

† https://gist.github.com/craffel/2d727968c3aaebd10359

function, which depends on the task at hand: in our case, solving a regression problem. These optimal weights can then be used for predictions on the test set.

CNNs are a class of artificial neural networks, where the hidden layers can be grouped into different classes according to their purpose; one such class of hidden layer is the eponymous convolutional layer. Below we describe the classes of hidden layers used in our CNN. Of course, this list is not exhaustive, and there exist many classes of hidden layers that we do not describe for means of brevity. Note also that we describe a CNN in the context of the problem we are trying to solve, where the input data are one-dimensional vectors. CNNs can of course also be used on higher dimensional input data, but the fundamental structure and different roles of the hidden layers do not change.

- **Convolutional layer:** In deep learning, the convolution operation is a method used to assign relative value to entries of input data, in our case one-dimensional vectors of time series data, while simultaneously preserving spatial relationships between individual entries of input data. For a given kernel size k and an input vector of length m, the convolution operation takes entries $1, \ldots, k$ of the input vector and multiplies by the kernel element-wise, whose length is k. The sum of the entries of the resulting vector are then the first entry of the feature map. This operation is iterated $m + 1 - k$ times, thus incorporating every entry in the data vector into the convolution operation. The output of the convolutional layer is called the feature map.

- For example, let $(1, 2, 1, 0, 0, 3)$ be our input vector, and $(1, 0, 1)$ be our kernel; here the kernel size is 3. The first iteration of the convolution operation involves taking the element-wise multiple of $(1, 2, 1)$ and $(1, 0, 1)$: the resulting multiple is $(1, 0, 1)$ and the corresponding sum of entries is 2. This is therefore the first entry of the feature map. The resulting feature map in this example is then $(2, 2, 1, 3)$.

- Clearly, the centre of each kernel cannot overlap with the first and final entry of the input vector. Zero-padding, sometimes referred to as same-padding, preserves the dimensions of input vectors and allows more layers to be applied in the CNN: zero-padding is simply the extension of the input vector and the setting of the first and final entries 1 and to be zero, while leaving the other entries unchanged. In our example, the input vector becomes $(0, 1, 2, 1, 0, 0, 3, 0)$ after zero padding.

- **Activation layer:** The activation layer is a nonlinear function that is applied to the output of the convolutional layer, i.e. the feature map; the purpose of the activation layer is indeed to introduce non-linearity into the CNN. Such functions are called activation functions; examples include the sigmoid function and the hyperbolic tangent function. In our CNN, we use the 'LeakyReLU'

activation function, defined as

$$f_\alpha(x) := \begin{cases} x, & \text{if } x > 0, \\ \alpha x, & \text{otherwise.} \end{cases}$$

The LeakyReLU activation allows a small positive gradient when the unit is inactive.

- **Max pooling layer:** For a given pooling size p, the max pooling layer returns a vector whose entries are the maximum among the neighbouring p entries in the feature map. For example, for feature map $(1, 3, 8, 2, 1, 0, 0, 4, 6, 1)$ and $p = 3$ the max pooling output is $(8, 8, 8, 8, 8, 8, 2, 4, 6, 6)$.

- Other pooling techniques apply the same idea, but use different functions to evaluate the neighbouring p entries in the feature map. Examples include average pooling, and L2-norm pooling, which in fact uses the Euclidean norm in mathematical nomenclature.

- **Dropout layer:** Dropout is a well-known technique incorporated into CNNs in order to prevent overfitting. Without the addition of a dropout layer, each node in a given layer is connected to each node in the subsequent layer; dropout temporarily removes nodes from different layers in the network. The removal of nodes is random and is determined by the dropout rate d, which gives the fraction of nodes to be temporarily dropped. Note that dropout is only implemented during training; during testing the weights of each node are multiplied by the dropout rate d.

- An excellent overview of the technique is given by Hinton *et al.* (2014). The authors provide an extensive study to show how predictive performance of CNNs, in a number of different settings, is improved using dropout.

- **Dense layer:** Also referred to as the fully connected layer, each node in the input layer is connected to each node in the output layer as the name suggests. After being processed by the convolutional, activation, pooling and dropout layers, the extracted features are then mapped to the final outputs via a subset of the dense layer, an activation function is then applied subsequently.

REMARK 2.2 *The number of filters, the kernel size k, the pooling size p and the dropout rate d are all examples of CNN hyperparameters.*

Having described the structure, we now focus on the mechanics of training the CNN. As mentioned previously, training a CNN corresponds to finding weights in the fully connected layer and kernels in the convolutional layers that minimise a specific loss function. Forward propagation is the name for the process by which input data is translated to an output through layers of the CNN; it is used to give the value of the loss function, and therefore the predictive power of the CNN, for certain weights and kernels. The back propagation algorithm is used to compute the gradient of the loss function from the error values of the loss function computed via forward propagation; weights and kernels, depending on the

values of the loss function, are then updated iteratively. In the case of our CNN, the Adam optimiser is used. More details on the back-propagation algorithm and the Adam optimiser can be found in Bengio *et al.* (2016, section 6.5, pages 200–219) and Ba and Kingma (2017) respectively.

2.2. Fractional Brownian motion

The fractional Brownian motion, from here referred to as fBm, is a centred Gaussian process whose covariance function depends on a parameter $H \in (0, 1)$, called the Hurst parameter. The precise definition is given below, Biagini *et al.* (2008, Definition 1.1.1.); note that setting $H = 1/2$ recovers the standard Brownian motion.

DEFINITION 2.3 *Let $H \in (0, 1)$: a fractional Brownian motion $(W_t^H)_{t \geq 0}$ is a continuous, centred Gaussian process with the following covariance function:*

$$\mathbb{E}\left[W_t^H W_s^H\right] = \frac{1}{2}\left(|t|^{2H} + |s|^{2H} - |t - s|^{2H}\right),$$

$$\text{for all } s, t \geq 0.$$

The value of the Hurst parameter H completely determines the sample path Hölder regularity of an fBm. Indeed, W^H has a version whose sample paths are almost surely Hölder continuous, with Hölder exponent γ, for all $\gamma \in (0, H)$.

The value of H also determines how the increments of W^H are correlated: for $H > 1/2$, the increments of W^H are positively correlated, and in this case W^H is said to be persistent; for $H < 1/2$, the increments of W^H are negatively correlated, and in this case W^H is said to be antipersistent. Recall that the standard Brownian motion, in the case where $H = 1/2$, has independent increments. More details and proofs of the above statements can be found in Biagini *et al.* (2008, Chapter 1).

To visualise how the value of H affects these characteristics we plot two sample paths of an fBm in figure 2. Clearly the left plot, where $H = 0.1$, has a much 'rougher' sample path than the right plot; this corresponds to a lower Hölder regularity. The apparent mean reversion of the left plot, compared to the trend-following right plot, illustrates the antipersistence and persistence of $W^{0.1}$ and $W^{0.9}$ respectively.

REMARK 2.4 *Many different integral representations of fBm exist. Arguably the most well known is the characterisation by Mandelbrot and Van Ness (1968, Definition 2.1) as*

$$W_t^H = \frac{1}{\Gamma(H + 1/2)}\left(\int_{-\infty}^0 ((t - s)^{H-1/2} \right.$$

$$\left. -(-s)^{H-1/2})\mathrm{d}B_s + \int_0^t (t - s)^{H-1/2}\mathrm{d}B_s\right), \quad (2)$$

where B is a standard Brownian motion and Γ the standard Gamma function.

3. Methodology

We begin this section by introducing the rBergomi model, before moving on to describe the training and test data, and the CNN architecture.
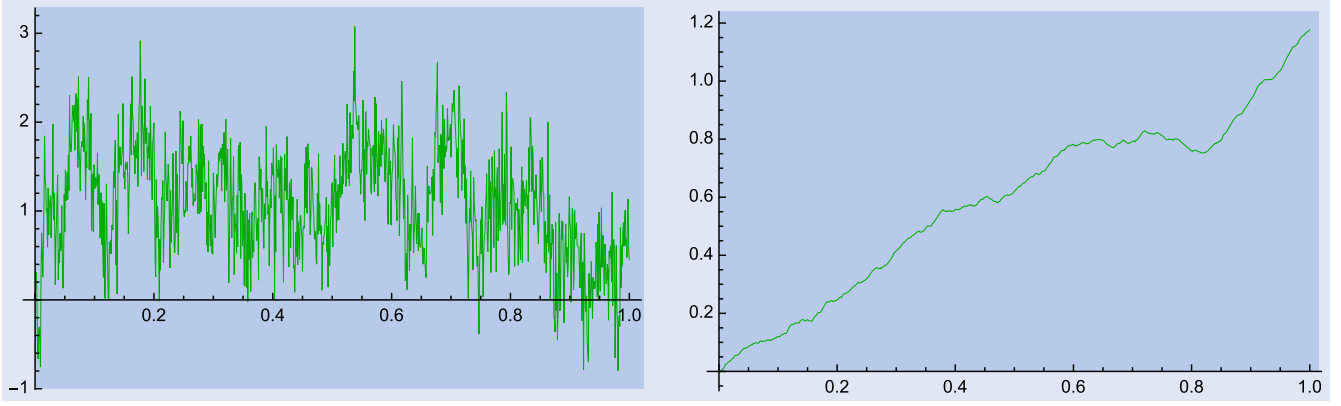
Figure 2. Two sample paths of a fractional Brownian motion W^H from $t = 0$ to $t = 1$ with 1001 sample points. In the left plot, we set $H = 0.1$ so that W^H is antipersistent; in the right plot, we set $H = 0.9$, hence W^H is persistent. Here we have used Mathematica's `'FractionalBrownianMotionProcess'` function to simulate each sample path.

3.1. The rBergomi model

Bayer *et al.* (2016) introduce a non-Markovian generalisation of Bergomi's 'second-generation' stochastic volatility model, which they dub the 'rBergomi' model. Let Z be the process defined pathwise as

$$Z_t := \int_0^t K_\alpha(s, t)\mathrm{d}W_s, \quad \text{for any } t \geq 0, \tag{3}$$

where $\alpha \in (-\frac{1}{2}, 0]$, W a standard Brownian motion, and where the kernel $K_\alpha : \mathbb{R}_+ \times \mathbb{R}_+ \to \mathbb{R}_+$ reads

$$K_\alpha(s, t) := \eta\sqrt{2\alpha + 1}(t - s)^\alpha, \quad \text{for all } 0 \leq s < t, \tag{4}$$

for some strictly positive constant η. Note that, for any $t \geq 0$, the map $s \mapsto K_\alpha(s, t)$ belongs to L^2, so that the stochastic integral (3) is well defined. The rBergomi model, where X is the log stock price process and v is the variance process, is then defined as

$$\begin{aligned} X_t &= -\frac{1}{2}\int_0^t v_s \mathrm{d}s + \int_0^t \sqrt{v_s}\mathrm{d}B_s, \quad X_0 = 0, \\ v_t &= v_0 \exp\left(Z_t - \frac{\eta^2}{2}t^{2\alpha+1}\right), \quad v_0 > 0, \end{aligned} \tag{5}$$

where the Brownian motion B is defined as $B := \rho W + \sqrt{1 - \rho^2}W^\perp$ for $\rho \in [-1, 1]$ and some standard Brownian motion $W^\perp$ independent of W. The filtration $(\mathscr{F}_t)_{t \geq 0}$ can here be taken as the one generated by the two-dimensional Brownian motion $(W, W^\perp)$.

REMARK 3.1 The process $\log v$ has a modification whose sample paths are almost surely locally γ-Hölder continuous, for all $\gamma \in (0, \alpha + \frac{1}{2})$ (Jacquier *et al.* 2018, Proposition 2.2). As stated above, the fBm has sample paths that are γ-Hölder continuous for any $\gamma \in (0, H)$ (Biagini *et al.* 2008, Theorem 1.6.1), so that the rBergomi model also captures this 'roughness' by identification $\alpha = H - 1/2$.

3.2. Training and test set

We use simulated sample paths of the normalised log volatility process $(\log(v_t/v_0))_{t \geq 0}$ of the rBergomi model as our input

Table 1. rBergomi input data size description.

Data set	Number of samples
Training set	14,000
Test set	7500
Validation set	3500

data; with the corresponding Hölder regularity $H = \alpha + 1/2$ as the output data. To simulate rBergomi sample paths we use Cholesky decomposition; this very well-known simulation technique is recommended because the resulting sample paths have the exact distribution, rather than an approximate distribution, of the normalised log volatility process of the rBergomi model. The code used is publicly available on Github.†

By Jacquier *et al.* (2018, Proposition 2.2), the Hölder regularity of the normalised log volatility process is independent of the value of η; the same proposition proves that the process Z, defined in (3), and the normalised log volatility process has the same Hölder regularity. We therefore set $\eta = 1$ in the model (5) above when generating the sample paths in section 4.1, for simplicity, and we also ignore the deterministic drift term $t^{2\alpha+1}$. In section 4.2, we take $\eta \neq 1$ and verify that this does not affect the predictive power of the CNN.

Every member of the resulting input data set has the following form: a vector $\mathbf{x}_i$, which is an rBergomi sample path with a given $H = \alpha + 1/2$, and a label $\mathbf{y}_i$, which corresponds to that given H used to generate $\mathbf{x}_i$. For each H value, we generate 5000 rBergomi sample paths. We then split the input data into training and test sets; we subsequently create a validation set from part of the test set. The sizes of each training/test/validation set of the rBergomi data are given in table 1.

3.2.1. Selection of H values.
We begin by letting H, and hence the corresponding label $\mathbf{y}_i$, take values in the discrete grid $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. We also sample 5 H values from

---

two probability distributions: the Uniform distribution† on $(0, 0.5)$ and the Beta$(1, 9)$ distribution.‡ This not only allows us to avoid uniformity in the output of the network; it should also make the network more robust when it comes to calibration, as H values for historical volatility data will almost certainly not be on the discrete grid $\{0.1, 0.2, 0.3, 0.4, 0.5\}$. Furthermore, we are also able to add emphasis to the 'rough' values of H, i.e. $H \approx 0.1$, particularly in the case of the Beta distribution.

The probability density function of the Beta(α, β) distribution is given by $f_{\alpha, \beta}(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)} \mathbb{I}_{(0,1)}(x)$, where the function B is defined as $B(\alpha, \beta) := \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$ and Γ is the standard Gamma function. We set $\alpha = 1, \beta = 9$, so that the expected value of the Beta distribution is 0.1, in accordance with the existing empirical studies (Bennedsen *et al.* 2017, Gatheral *et al.* 2018). For each sampling method for H, we generate 5000 rBergomi sample paths for each of the five H values.

3.3. CNN architecture

We use a one-dimensional CNN, since our input $\mathbf{x}_i$ are (one-dimensional) vectors, with three layers of kernels, where the kernel size for each kernel is 20 and each layer is succeeded by the Leaky ReLU activation function with alpha = 0.1; we add max pooling layers, each of size 3, and dropout layers between each layer of kernels. We use zero-padding in each of the convolutional and max pooling layers. The values for kernel size, max pooling size, dropout rate, and rate for the Leaky ReLU activation function were chosen because they achieved the lowest mean square error among all values tested. By no means are these hyperparameters chosen in the most optimal way, but are indeed sufficiently optimal to achieve accurate predictions, see results in section 4. We clarify the specific structure of the hidden layers of the CNN below:

- the first layer, with 32 kernels;
- max pooling layer;
- a dropout layer, with rate = 0.25;
- the second layer, with 64 kernels;
- max pooling layer;
- a dropout layer, with rate = 0.25;
- the third layer, with 128 kernels;
- max pooling layer;
- a dropout layer, with rate = 0.4;
- a dense layer with 128 units;
- a dropout layer, with rate = 0.3.

The reason for choosing this structure for our CNN, which is fairly standard for image processing in the computer science discipline, is two fold. The first is that, heuristically, images are processed by considering the values of each entry of an image matrix together with the neighbouring entries and putting more emphasis on those neighbouring values than entries far from the entry being considered. To study the

Hölder regularity of a sample path of a stochastic process, the values of neighbouring points of each entry in a sample path vector will provide the most information about the Hölder regularity of that process; for this reason we employ an image processing-type architecture. The second is to avoid the task of choosing optimal hyperparameters for the number of filters in each layer.

If the CNN is able to accurately learn the value of the Hölder exponent, then we are making an important contribution to the field of mathematical finance. Indeed, existing calibration techniques for the rBergomi model are still in need of development. One calibration technique, suggested by Alós and Shiraya (2019), is based on their result that the difference between the at-the-money implied volatility for a European Call and the price of a volatility swap has power law behaviour of order $2H$ for short maturities, when the volatility process is driven by an fBm W^H. The method predicts H accurately on simulated data, but in practice volatility swaps tend to be illiquid for maturities less than 8 months, and therefore it is difficult to use this method for accurate calibration in practice. Another technique, proposed by Chang (2014), suggests using maximum likelihood estimation to estimate H. While the method accurately predicts H from simulated fBm data, the computational cost of this approach is too high for practical application in the quantitative finance industry. Lastly we consider the least squares method of Gatheral *et al.* (2018). Inspired by the qth moment formula§ for increments of an fBm, the authors suggest estimation of H via a linear regression of the log of lagged qth moments of the log volatility process against the log of the lags. The method, however, is sensitive to the choice of q; in particular the method does not perform well for higher order moments. Additionally, in section 4, we show that the least squares method yields erroneous H estimates for processes exhibiting mean reversion.

4. Solving the regression problem

We now move onto solving the regression problem, in order to find the Hölder exponent from sample path input data. In section 4.1, we use the rBergomi model described above to generate our input data, using Cholesky decomposition, which is the main focus of the paper. In section 4.2, we additionally train the CNN with the above rBergomi input data, except with $\eta \neq 1$, as well as with random η and H for each sample path. We also train with fBm sample paths as input data. The aim is to illustrate the robustness of this novel method using CNNs. In section 4.3, we briefly investigate if the CNN approach can be extended to additionally learn the parameter η, as well the parameter H.

4.1. rBergomi model

The sizes of each training/test/validation set of the rBergomi data are given in table 1. We train the CNN three times: for

---

† The corresponding set of possible H values is $\{0.05, 0.18, 0.29, 0.31, 0.44\}$.
‡ The corresponding set of possible H values is $\{0.02, 0.07, 0.06, 0.13, 0.22\}$.

§ For a fBm W^H, the following holds for all $q > 0$: $E[|W_{t+\Delta}^H - W_t^H|^q] = K_q \Delta^{qH}$, where K_q is the absolute q^{th} moment of a standard normal distribution.

Table 2. Test results for discretised H.

Input length	RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
100	1.041×10^{-2}	69.77	0.76	2.118×10^{-1}	591.76
200	8.196×10^{-3}	74.89	0.75	2.046×10^{-1}	622.00
300	1.096×10^{-2}	80.92	0.79	2.025×10^{-1}	635.65
400	8.263×10^{-3}	92.02	0.93	2.014×10^{-1}	634.22
500	1.232×10^{-2}	93.76	0.93	2.010×10^{-1}	627.62

Table 3. Test results for $H \sim \text{Uniform}(0.0, 0.5)$.

Input length	RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
100	1.137×10^{-2}	66.66	0.68	1.989×10^{-1}	611.61
200	7.910×10^{-3}	72.20	0.73	1.927×10^{-1}	620.72
300	5.115×10^{-3}	79.80	0.78	1.907×10^{-1}	630.75
400	9.409×10^{-3}	86.35	0.82	1.895×10^{-1}	634.24
500	1.282×10^{-2}	96.79	0.93	1.892×10^{-1}	628.26

Table 4. Test results for $H \sim \text{Beta}(1, 9)$.

Input length	RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
100	6.672×10^{-3}	70.71	0.72	1.040×10^{-1}	616.74
200	7.193×10^{-3}	73.26	0.70	9.962×10^{-2}	626.77
300	1.207×10^{-1}	80.63	0.74	9.791×10^{-2}	637.45
400	1.171×10^{-2}	87.09	0.75	9.699×10^{-2}	637.33
500	1.207×10^{-1}	94.20	0.77	9.663×10^{-2}	644.60

discretely sampled H, for Uniformly sampled H, and for Beta sampled H.

The CNN calibration method should also be robust to the dimensions of the input data, and training the CNN on vectors of length 100 should produce similar predictive performance to the CNN trained on vectors of length 500. Consequently we train the CNN with the length of the input vector taking values in $\{100, 200, 300, 400, 500\}$.

We present the test results for the CNN in tables 2– 4. We use the mean square error as the loss function in the CNN, and report the predictive performance of the CNN using the root mean square error (RMSE), so that the predicted value and true value of H are of the same unit of measurement. We also give the time taken, in seconds, to complete the training and testing of the CNN.† The Python code is available here: https://github.com/henrymstone/CNN-repository.

We train the network with the architecture described above, setting `batch size` $= 64$ as is fairly standard, and `epochs` $= 30$ as this value gave the lowest mean square error. As a comparison, we also use the least square (LS) calibration approach suggested by Gatheral *et al.* (2018, section 2.1) on the test set used for the CNN, and compute the loss as the root mean square error between the predicted and true values for H.

For each input length, the predictive performances of the CNNs trained on $H \sim \text{Uniform}(0.0, 0.5)$ and discretised H are similar, and in each case the CNN approach clearly outperforms the least square approach in terms of predictive power,

by 1 or 2 orders of magnitude. For $H \sim \text{Beta}(1, 9)$, the CNN also outperforms the least square approach, again by 1 or 2 orders of magnitude, when the input vector length is 100, 200 or 400; the accuracy of the CNN is slightly poorer than the least square approach for the other input vector lengths.

As one would expect for both calibration approaches, the time taken in both cases is, in general, an increasing function of the length of the input vector for each method of sampling H. Since we are able to train the CNN using Google Colaboratory's GPUs, and Python's Keras module has been optimised for execution on GPU, the time taken for training and testing is approximately eight times less than the least square approach. The training and test times for each sampling method are all very similar.

The above analysis indicates that decreasing the length of the input vector does not significantly worsen the predictive performance of the CNN; in fact when $H \sim \text{Beta}(1, 9)$ the performance is improved, when comparing the performance of the length 100 input vector and the length 500 input vector.

4.2. Robustness test

Following the analysis in section 4.1, we set the input vector length to be 100. We then generate our input data by letting η take values in $\{0.25, 0.8, 1.3, 2.5\}$ in the rBergomi model, and use discretised sampling for H to generate 5000 sample paths for each H. We give the results in table 5; as before we also include the root mean square error (RMSE) and time taken for the least square (LS) approach of Gatheral *et al.* (2018), applied to the test set as a comparison. Plots of the training error and validation error are given in Appendix 2.

We can see that the CNN approach vastly outperforms the least square approach in each case, both in terms of the

---

† All computations were executed in Python, using the Keras module to build and train the network, on a Macbook pro with a 2.6 GHz Intel Core i5 processor and 8 GB 1600 MHz DDR3 memory. The code was run on Google Colaboratory, using the platform's GPU.

Table 5. rBergomi regression results for $\eta \neq 1$ and input vector length $= 100$.

η	RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
0.25	8.206×10^{-3}	66.81	0.63	2.122×10^{-1}	666.79
0.8	1.137×10^{-2}	66.93	0.60	2.122×10^{-1}	665.68
1.3	1.473×10^{-2}	67.22	0.66	2.122×10^{-1}	671.11
2.5	9.003×10^{-3}	67.27	0.63	2.122×10^{-1}	667.16

Table 6. rBergomi regression results for $\eta \sim \text{Uniform}(0, 3)$, $H \sim \text{Beta}(1, 9)$, and input vector length $= 100$.

RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
1.382×10^{-2}	66.52	0.61	1.499×10^{-1}	598.51

Table 7. fBm regression results for discretised H, $H \sim \text{Uniform}(0.0, 0.5)$, and $H \sim \text{Beta}(1, 9)$.

Sampling	RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
Discretised	2.483×10^{-2}	72.79	0.65	2.346×10^{-1}	635.18
Uniform	2.001×10^{-2}	71.95	0.64	2.090×10^{-1}	615.74
Beta	1.945×10^{-2}	72.20	0.67	9.785×10^{-2}	622.36

accuracy of the predictions and the time taken. The CNN's performance for $\eta = 0.8, 1.3$ is slightly worse than for the other two values of η, but is still superior to the least square approach. Note that values for the root mean square error for the least square approach are only equal when rounded to three decimal places.

We further extend the robustness test on rBergomi data as follows: we begin by generating 25,000 $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$, and then use these values to simulate 25,000 rBergomi sample paths of length 100, each with its own unique and random η and H. The corresponding training, test and validation sets thus remain the same size. The results are presented in table 6; as above we include the root mean square error and time taken for the least square approach as a comparison. Plots of the training error and validation error are given in Appendix 3.

Interestingly, when $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$ for each sample path, the predictive power of the CNN is only inferior, in the worst case, by 1 order of magnitude compared to when the values for either η or H are fixed. Furthermore, the CNN approach maintains its significant advantage over the least square approach, both in terms of predictive power and speed.

We conclude this robustness test by using fBm sample paths, generated using Cholesky decomposition, as input data to train and test the CNN. For each H we simulate 5000 sample paths of length 100; we employ the discretised H, $H \sim \text{Uniform}(0.0, 0.5)$ and $H \sim \text{Beta}(1, 9)$ sampling methods. The results are given in table 7 and the plots of the training error and validation error are given in Appendix 4.

The CNN maintains its speed advantage over the least square method for each sampling method for H, as well as maintaining superior predictive performance, by an order of magnitude. The results from this final part of the robustness test allow us to conclude that CNNs can indeed identify Hölder regularity from a set of sample paths, thus answering the question posed in the introduction of the paper.

4.3. Extension to learning η

In this section, we briefly explore whether the CNN can learn the value of η, as well as H. We use the rBergomi model, with $\eta \sim \text{Uniform}(0, 3)$, $H \sim \text{Beta}(1, 9)$ for each sample path, and input vectors of length 100 as our input data, as above; the corresponding output variable then becomes a two-dimensional vector $\mathbf{y}_i = (H_i, \eta_i)$. We continue to use the root mean square error as a measure of predictive power and compare to the least square approach as above. The least square (LS) approach can indeed be used to estimate the value of η (Gatheral *et al.* 2018, section 3.4), although the authors use the notation ν instead of η. The results are given in table 8, with loss plots given in Appendix 5.

Encouragingly, we see that the CNN approach still outperforms the least square approach for both accuracy and time; however, the CNN approach is approximately twice as accurate as the least square approach, compared to the orders of magnitude in the cases above. Note that we kept all hyperparameter values unchanged; it is likely that superior predictive power could be achieved with some tuning of the hyperparameter values in the CNN. This is not the aim of this paper, however, and we leave this once again to further research.

REMARK 4.1 We conclude the above results with a remark on the speed and accuracy advantages of the CNN approach, from a theoretical perspective. The above results show the CNN method to be more accurate, by orders of magnitude and significantly faster than the existing method suggested in Gatheral *et al.* (2018) when estimating H on simulated rBergomi and fBm data. The explanation for the speed advantage of the CNN method is that once the network has been trained, estimations are made by the straightforward computation of the composition (1) given in Definition 2.1; on the other hand, the LS approach requires a number of successive regressions to be executed in order to estimate H. Clearly then the CNN method will be faster than the LS approach. We now recall our motivation that justifies the use of a CNN to estimate H: the predictive power of the CNN lies in the convolutional

Table 8. Regression results for learning H and η, with input vector length $= 100$.

RMSE (CNN)	Training time (s)	Test time (s)	RMSE (LS)	Time (s)
0.666	71.82	0.62	1.170	613.62

Table 9. Mean H estimates for a (mean-reverting) Ornstein–Uhlenbeck process X.

c	CNN (Discretised)	CNN (Uniform)	LS
3	0.46	0.42	0.22
0.3	0.49	0.44	0.67

Table 10. Calibration results

Sampling method	Root mean square error	Standard deviation
Discretised H	5.558×10^{-2}	2.900×10^{-3}
$H \sim \text{Uniform}(0.0, 0.5)$	2.444×10^{-1}	1.141×10^{-2}
$H \sim \text{Beta}(1, 9)$	4.253×10^{-2}	1.098×10^{-3}

operator, which assigns a value to each entry $\mathbf{x}_i^j$ of an input vector $\mathbf{x}_i$; this value is determined by the relative values of the entries neighbouring $\mathbf{x}_i^j$. To estimate the Hölder regularity of a stochastic process, the values of neighbouring points of each entry in a trajectory vector will provide the most information about the Hölder regularity of that trajectory, and thus a CNN is indeed a valid means of estimating H. The CNN is able to detect very subtle path regularity properties, via integrated applications of the convolutional operator, allowing for very accurate predictions of H. Furthermore, the LS estimation is based on an approximation of the qth moment of an increment of log volatility, whose accuracy is highly dependent on the choice of q, while the CNN approach is completely independent of the choice of q. Together, this explains why the CNN approach gives more accurate H estimations than the LS approach.

4.4. *Robustness test on a (mean-reverting) Ornstein–Uhlenbeck process*

We finish section 4 with a comparison of the performance of the CNN and LS approaches on a (mean-reverting) Ornstein–Uhlenbeck process. Recall that a (mean-reverting) Ornstein–Uhlenbeck process X satisfies the following SDE:

$$\mathrm{d}X_t = (a - bX_t)\mathrm{d}t + cX_t\mathrm{d}W_t, \qquad X_0 = x_0 \in \mathbb{R},$$

where W is a standard Brownian motion, and that X is γ-Hölder continuous for all $\gamma \in (0, 1/2)$; therefore we expect both approaches to estimate $H \approx 0.5$.

We set the input length to be 100 for the CNN, using the Discretised and Uniform H sampling methods detailed above.† Each trained network is then used to estimate H on trajectories of a (mean-reverting) Ornstein–Uhlenbeck process, which are simulated using the Euler–Maruyama method. We simulate 1000 trajectories of length 100, setting $(x_0, a, b) = (0.1, 1., 2.1)$, and report the mean estimated H values for the CNN and LS approaches in table 9, for the 'high volatility' ($c = 3$) and 'low volatility' ($c = 0.3$) regimes.

The results in table 9 provide compelling evidence that the CNN is indeed learning the Hölder regularity of the

sample path, as the mean H estimates for the CNN are sufficiently close to 0.5. Note that the mean LS H estimates are significantly further from 0.5. The results provide further evidence that the CNN approach is preferable to the LS approach, which is not only slower and less accurate than the CNN approach but may also incorrectly identify roughness in data.

5. Calibration using CNNs

We begin by using the trained CNNs from section 4 to predict the Hölder exponent of historic realised volatility data from the Oxford–Man Institute of Quantitative Finance, which is free and publicly available.‡ We choose the length of the input vectors to be 100 from the analysis given in section 4. We took a sample of 10 different indices§ from the 31 available; for each index we then used a time series of 200 sequential data points to create 11 vectors of length 100 (entries 0 to 100, 10 to 110 and so on) to predict the Hölder exponent for each index. We compute the root mean square error between the CNN prediction and the least square prediction, and the standard deviation of the difference between the two predictions , see table 10.

Indeed, we can see that this set of results is very promising. In each case, both the root mean square error and the standard deviation are small; note that both root mean squared error values and the standard deviations of discretised H and $H \sim \text{Beta}(1, 9)$ are an order of magnitude greater than for $H \sim \text{Uniform}(0.0, 0.5)$. This therefore indicates that the calibration values attained by the network are very close to those attained by the least squares approach. Note that this in turn provides further evidence that $H \approx 0.1$, further corroborating the findings of Gatheral *et al.* (2018) and Bennedsen *et al.* (2017).

We can state, therefore, that this calibration scheme is precise enough to be used in practice, where we recommend using $H \sim \text{Beta}(1, 9)$ to train the network with input length 100, due smaller root mean squared error and standard deviation in testing, the nondiscretised network output, and emphasis of this distribution on 'rough' values of H i.e. $H \approx 0.1$.

---

† We do not experiment with the Beta sampling method, since those H values are concentrated around 0.1, and so will produce poor estimations when we expect $H \approx 0.5$.

‡ https://realized.oxford-man.ox.ac.uk/data/download
§ AEX, All Ordinary, DAX, FTSE 100, Hang Seng, NIFTY 50, Nasdaq 100, Nikkei 225, S&P500, Shanghai Composite.

The practical implementation of our calibration scheme is a simple two-step process. The first step is to train the CNN, with the H sampling method and input length n chosen by the practitioner; this can be done once offline, with the weights of the trained network saved to avoid unnecessary repetition for each calibration task. The second step is to input the most recent n volatility observations into the CNN, which will return the corresponding H value for those n observations. This value of $H = \alpha + 1/2$ can then be inputted into the rBergomi model and used for, say, pricing. We note that some testing for the optimal choice of H sampling method and input length n is required on the part of the practitioner implementing our calibration scheme.

REMARK 5.1 We now take this opportunity to discuss the calibration methodology presented in this paper. We are treating calibration as a *supervised* learning problem when in practice it is an *unsupervised* learning problem, strictly speaking. While each vector in the input data in the regression problem given in section 4 was indeed labelled with the corresponding H value, the data from the Oxford–Man Institute 'realised' library has no such labels, thus we use the least square calibration values as 'true' values.

6. Discussion of results and conclusion

The results presented in the paper are very promising: CNNs can indeed be used to give highly accurate predictions for the Hölder exponent of rBergomi and fBm sample paths. In fact, we conjecture that even higher accuracy could be attained if we optimised the hyperparameter values in our CNN. Additionally, invoking a k-fold cross validation could further improve the predictive power of the CNN. These two techniques could additionally be used to attain more accurate predictions for the parameter η in the rBergomi model.

The results in the paper invite a number of intriguing questions for future research. First, is it possible to use the unlabelled volatility data, together with known forecasting formula, to train the network and thus predict 'roughness', potentially improving this calibration method further and addressing Remark 5.1? Second, can we achieve more accurate predictions in the case where the input data are fBm sample paths or when we are also trying to learn the value of η? Lastly, we propose the following question: is it possible to achieve similar predictive results using more traditional machine learning techniques, such as ensemble and boosting methods?

To conclude, in this paper we have shown for the first time that CNNs can indeed learn the 'roughness' (i.e. the Hölder exponent) of time series data with a high degree of accuracy. We have additionally provided an efficient and accurate means of calibrating the rBergomi model. We have also shown our method to be far more accurate, by orders of magnitude and significantly faster than the existing method suggested in Gatheral *et al.* (2018). We have shown our method to correctly estimate H on mean-reverting trajectories, where the least squares approach fails and incorrectly identifies roughness. Finally, the paper has also opened up a number of interesting avenues for future research.

References

Abdel-Hamid, O., Deng, L. and Yu, D., Exploring convolutional neural network structures and optimization techniques for speech recognition. *Interspeech*, 2013, **11**, 73–75.

Alós, E. and Shiraya, K., Estimating the Hurst parameter from short term volatility swaps: A Malliavin calculus approach. *Finance Stochastics*, 2019, **23**(2), 423–447.

Büler, H., Gonon, L., Teichmann, J. and Wood, B., Deep hedging. Preprint available at arXiv:1802.03042v1, 2018.

Ba, J. and Kingma, D.P., Adam: A method for stochastic optimization. Preprint available at arXiv:1412.6980v9 (accessed 12 February 2019), 2017.

Back, A.D., Chung-Tsoi, A., Giles, C.L. and Lawrence, S., Face recognition: A convolutional neural-network approach. *IEEE Trans. Neural Netw.*, 1997, **8**(1), 98–113.

Bayer, C., Friz, P.K. and Gatheral, J., Pricing under rough volatility. *Quant. Finance*, 2016, **16**(6), 887–904.

Bayer, C. and Stemper, B., Deep calibration of rough stochastic volatility models. Preprint available at arXiv:1810.03399v1 (accessed 12 February 2019), 2018.

Bengio, Y., Courville, Y. and Goodfellow, I., *Deep Learning*, 2016 (MIT Press: Massachusetts).

Bennedsen, M., Lunde, A. and Pakkanen, M.S., Decoupling the short- and long-term behavior of stochastic volatility. Preprint available at arXiv:1610.00332v2 (accessed 12 February 2019), 2017.

Biagini, F., Hu, Y., Øksendal, B. and Zhang, T., *Stochastic Calculus for Fractional Brownian Motion and Applications*, 2008 (Springer: London).

Chang, Y.C., Efficiently implementing the maximum likelihood estimator for hurst exponent. *Math. Probl. Eng.*, 2014, **2014**, Article ID 490568, 10 pages.

Chen, C., Kornhauser, A., Seff, A. and Xiao, J., DeepDriving: Learning affordance for direct perception in autonomous driving. IEEE International Conference on Computer Vision (ICCV), pp. 2722–2730, 2015.

Deng, L., Hinton, G. and Kingsbury, B., New types of deep neural network learning for speech recognition and related applications: An overview. IEEE International Conference on Acoustics, Speech and Signal Processing, 2013.

Gatheral, J., Jaisson, T. and Rosenbaum, M., Volatility is rough. *Quant. Finance*, 2018, **18**(6), 933–949.

Hinton, G.E., Krizhevsky, A., Salakhutdinov, R., Srivastava, N. and Sutskever, I., Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 2014, **15**, 1929–195.

Iandola, F.N., Jin, P.H., Keutzer, K. and Wu, B., SqueezeDet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. IEEE

Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 446–454, 2017.

Jacquier, A., Pakkanen, M.S. and Stone, H., Pathwise large deviations for the rough Bergomi model. *J. App. Prob.*, 2018, **55**(4), 1078–1092.

Kinh Gian Do, R., Nishio, M., Togashi, K. and Yamashita, R., Convolutional neural networks: An overview and application in radiology. *Insights. Imaging.*, 2018, **9**(4), 611–629.

Krizhevsky, A., Sutskever, I. and Hinton, G.E., Image net classification with deep convolutional neural networks. *Adv. Neural. Inf. Process. Syst.*, 2012, **25**, 1–9.

Mandelbrot, B.B. and Van Ness, J.W., Fractional Brownian motions, fractional noises, and applications. *SIAM Rev.*, 1968, **10**, 422–437.

Simonyan, K. and Zisserman, A., Very deep convolutional networks for large-scale image recognition, 2015. Preprint available at arXiv:1409.1556v6 (accessed 12 February 2019).

Appendix 1. Loss plots for rBergomi with different *H* sampling

Here we plot the training loss and validation loss (MSE) at each epoch, for discretised H, $H \sim$ Uniform$(0.0, 0.5)$ and $H \sim$ Beta$(1, 9)$ in figures 1, 2 and A1 respectively. Note that the training loss and validation loss both tend to decrease as the number of epochs increases; this is a good indication that predictive performance improves as the number of epochs increases, without overfitting (Kinh Gian Do *et al.* 2018, Overfitting Section, Page 619). The only exception is figure A1, in the case where input length is either 300 or 500; note that these two cases also correspond to larger root mean squared error values. Recall, however, that our calibration scheme uses input vectors of length 100 so this does not pose any real problems for practical use.
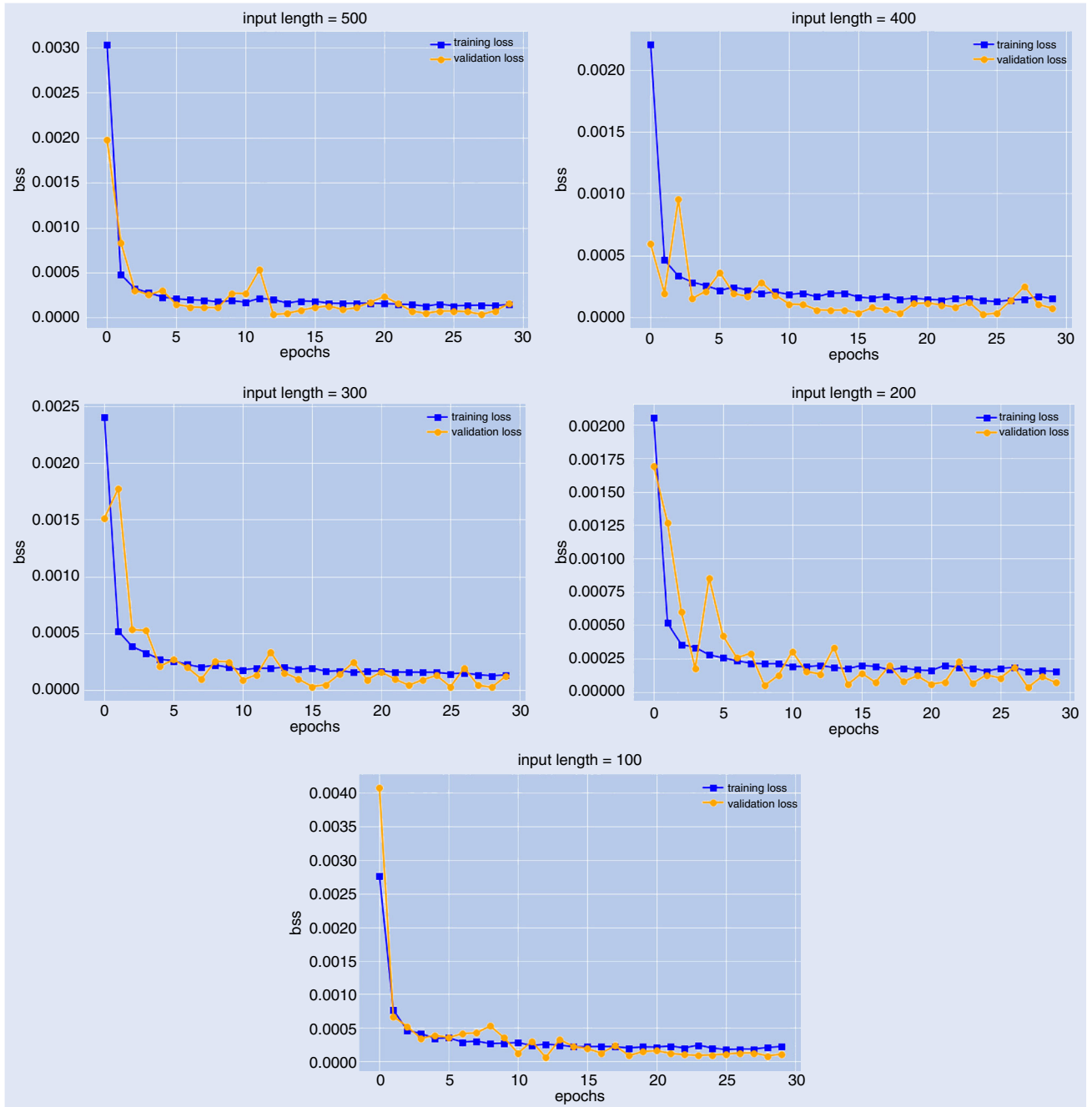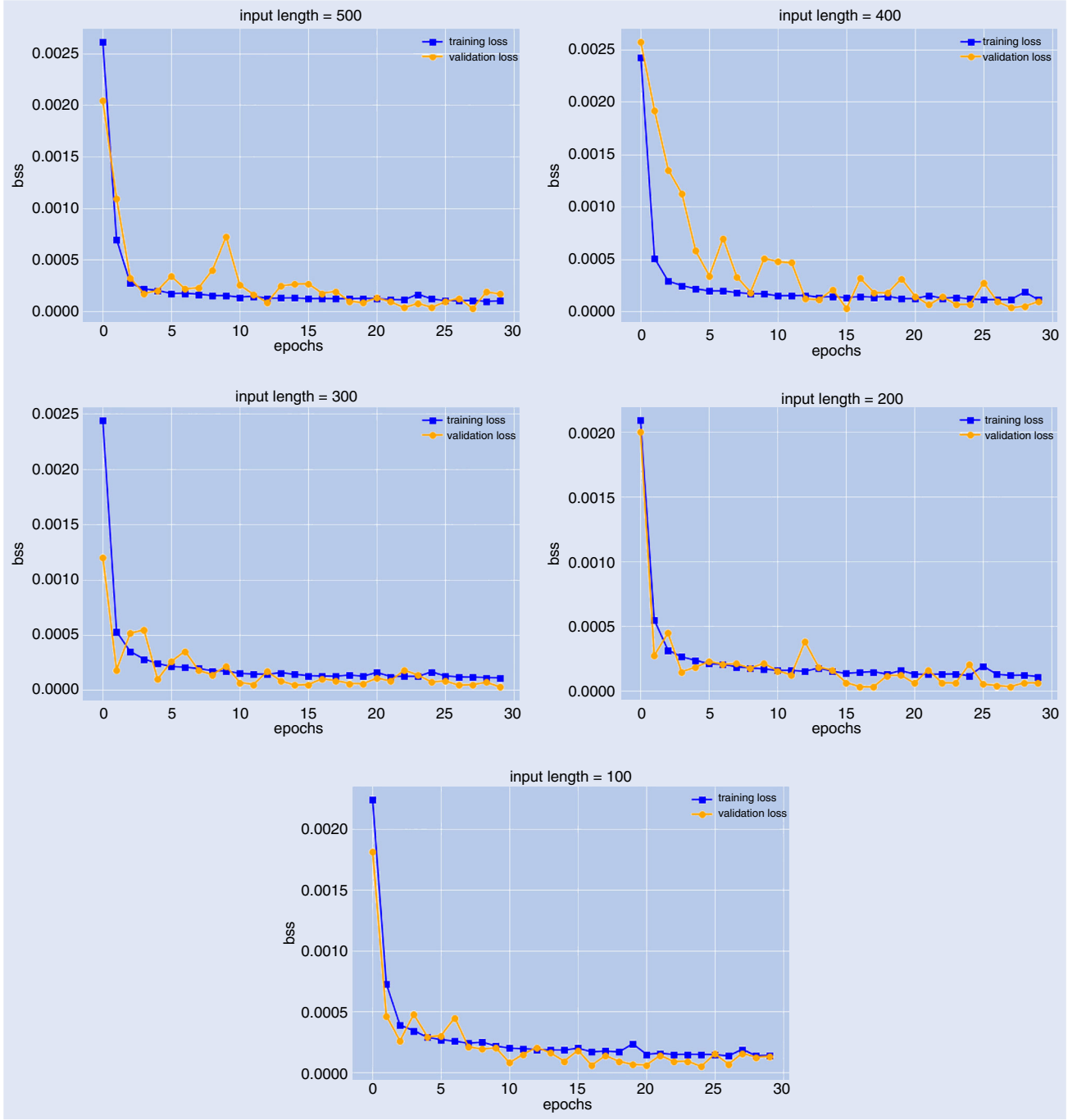

Figure A1. Loss plots for discretised *H*.

Figure A2. Loss plots for $H \sim \text{Uniform}(0.0, 0.5)$.

Appendix 2. Loss plots for rBergomi with $\eta \neq 1$

We now plot the training loss and validation loss (MSE) at each epoch, as above, for the rBergomi model with discretised H and $\eta \in \{0.25, 0.8, 1.3, 2.5\}$ in figure A2. We fix the input vector length to be 100. For each η value, the training loss and validation loss both decrease with each epoch.
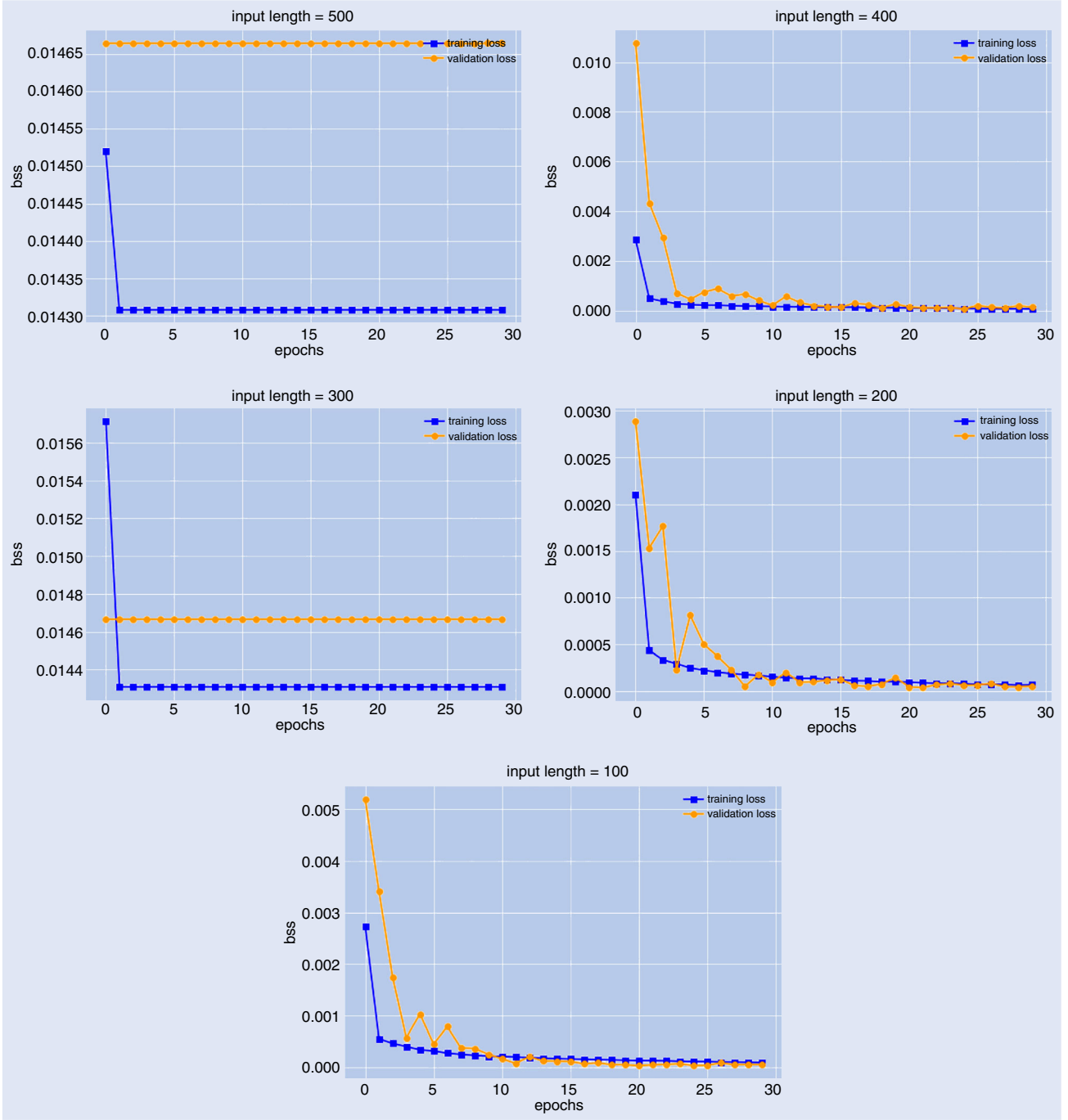
Appendix 3. Loss plots for rBergomi with $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$

We now plot the training loss and validation loss (MSE) at each epoch, as above, for the rBergomi model with $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$ for each sample path, in figure A3. We fix the input

vector length to be 100. Note that the training loss and validation loss both decrease with each epoch.

## Appendix 4. Loss plots for fBm with different H sampling

We plot the training and validation loss (MSE) at each epoch, for fBm with discretised H, $H \sim \text{Uniform}(0.0, 0.5)$, and $H \sim \text{Beta}(1, 9)$ in figures A4– A6 respectively. As in Appendix 2, we fix the input vector length to be 100. For discretised H and $H \sim \text{Uniform}(0.0, 0.5)$ the training loss and validation loss both decrease with each epoch. For $H \sim \text{Beta}(1, 9)$, however, the training loss decreases but the validation loss remains flat. This could possibly suggest overfitting, and poor predictive performance; note that the root mean square error was indeed higher than for discretised H and $H \sim \text{Uniform}(0.0, 0.5)$.
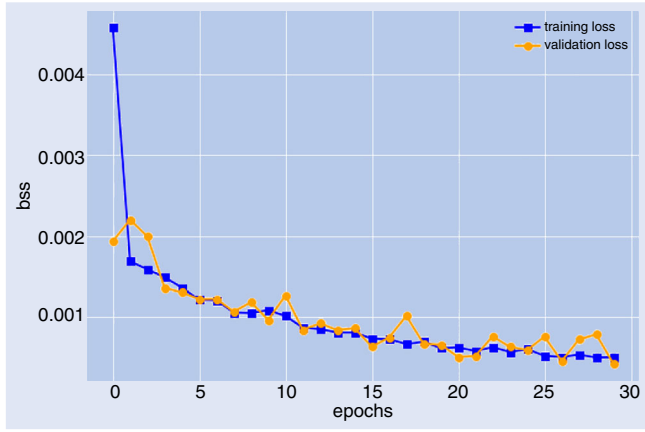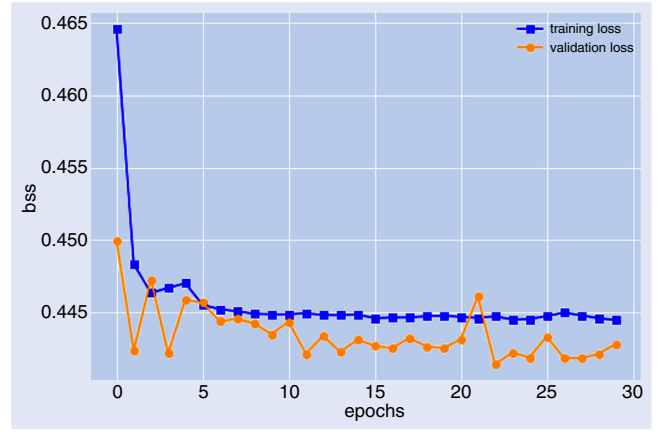
Figure A3. Loss plots for $H \sim \text{Beta}(1, 9)$.

Appendix 5. Loss plots for learning η

In figure A7, we plot the training loss and validation loss (MSE) at each epoch, as in Appendix 3, for the rBergomi model with $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$ for each sample path. Recall that in this case the CNN is learning the value of η, as well as the value of H. We fix the input vector length to be 100. Note that the training loss and validation loss both tend to decrease with each epoch.
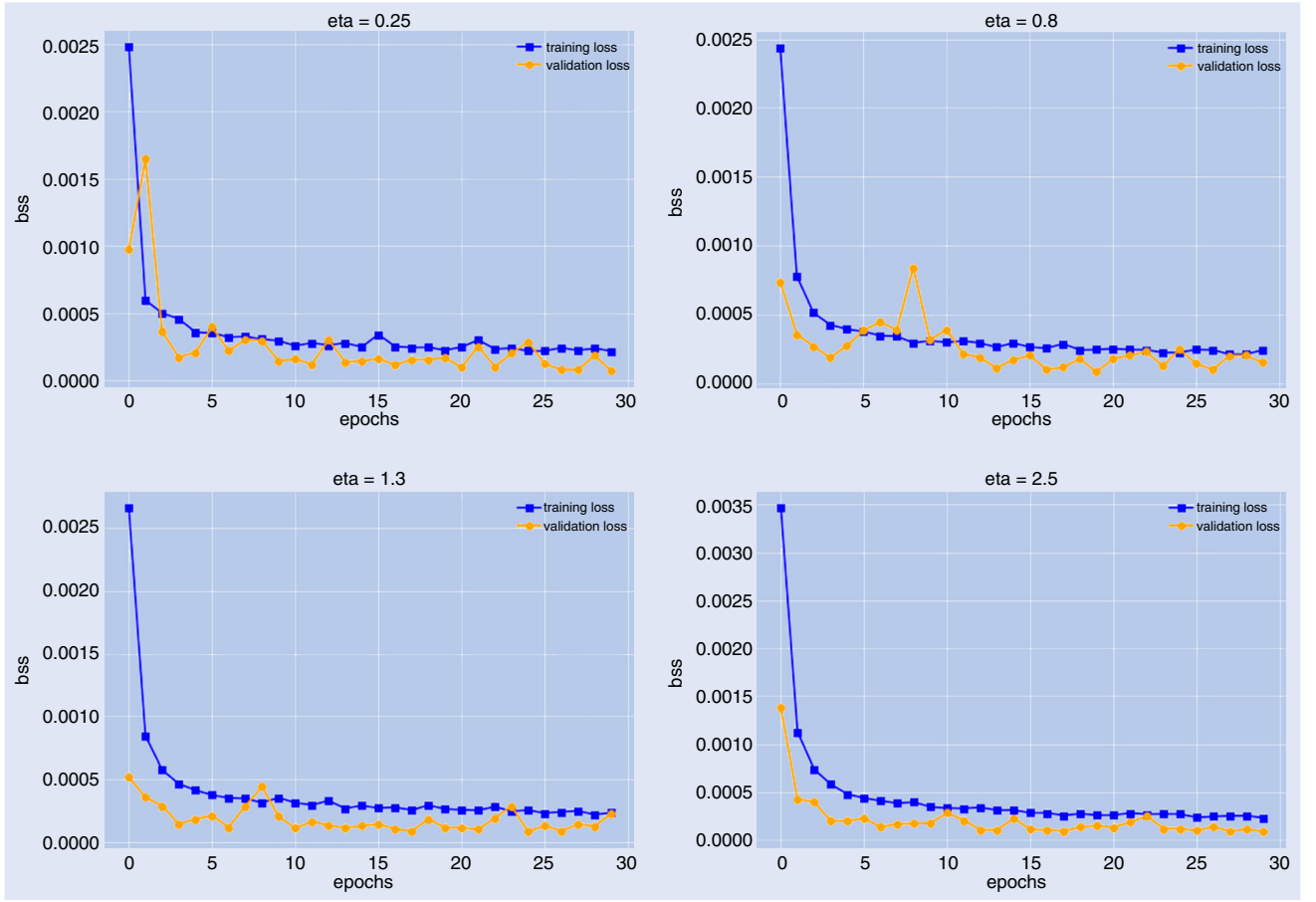
Figure A4. Loss plots for discretised H and $\eta \in \{0.25, 0.8, 1.3, 2.5\}$.
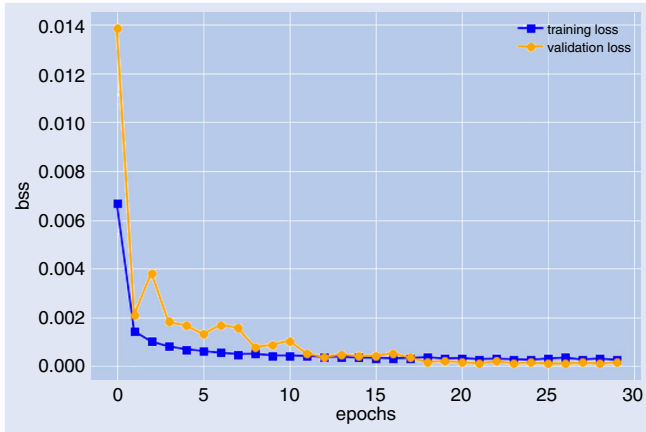
Figure A5. Loss plots for $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$.
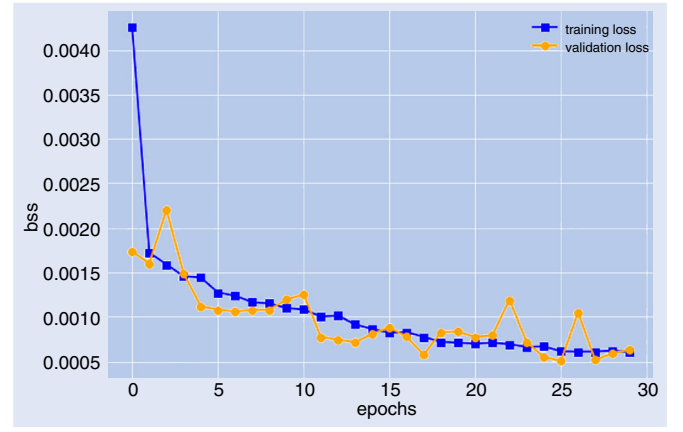
Figure A6. Loss plot for fBm with discretised H.

Figure A7. Loss plot for fBm with $H \sim \text{Uniform}(0.0, 0.5)$.

Figure A9. Loss plots for $\eta \sim \text{Uniform}(0, 3)$ and $H \sim \text{Beta}(1, 9)$.
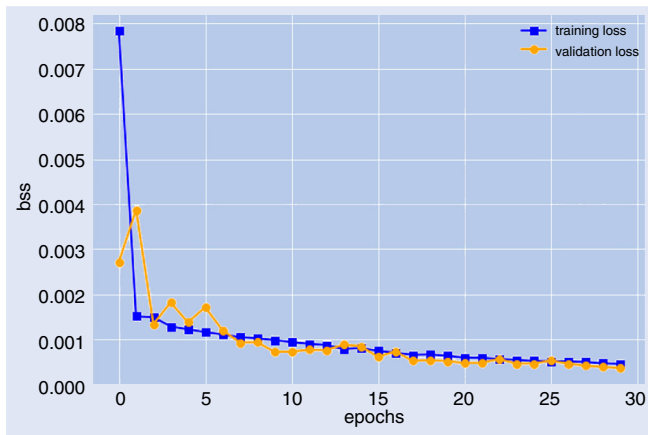
Figure A8. Loss plot for fBm with $H \sim \text{Beta}(1, 9)$.