Sequential Monte Carlo method for training Neural Networks on non-stationary time series

by

Jasper E. Hoogendoorn

to obtain the degree of Master of Science in Applied Mathematics with the specialization Financial Engineering at the Delft University of Technology, to be defended publicly on Friday July 5, 2019 at 11:00 AM.

June 28, 2019

Student number:	4253051	
Project duration:	October 1, 2018 – June 28, 2019	
Thesis committee:	Prof. dr. ir. C. W. Oosterlee,	TU Delft, CWI
	Dr. ir. J. Bierkens,	TU Delft
	Drs. L. Tegels,	KPMG
	Dr. A. Borovykh,	CWI

Abstract

In this thesis, we study the sequential Monte Carlo method for training neural networks in the context of time series forecasting. Sequential Monte Carlo can be particularly useful in problems in which the data is sequential, noisy and non-stationary. We compare this algorithm against a gradient-based method known as stochastic gradient descent (SGD), a commonly used method for training neural networks. The performance of SGD on forecasting non-stationary, noisy time series can be poor due to the possibility of overfitting on the data. The sequential Monte Carlo method may offer a solution for the problems that arise in forecasting non-stationary time series with SGD neural networks. At the same time, neural networks trained with SGD give deterministic predictions, and there is a need for quantification of the uncertainty in the prediction. Sequential Monte Carlo sequentially samples the weights of the neural network, providing a posterior distribution on the weights and thus the outcome. In this work, the sequential Monte Carlo algorithm is tested and analyzed, with different parameter settings, on four time series to give an overview of the behavior. Furthermore, we apply the SMC algorithm on a convolutional neural network known as WaveNet. We show that the SMC algorithm is very well-suited for forecasting non-stationary time series, and can significantly outperform the gradient-based SGD method. Additionally, we show that for specific time series the SMC algorithm on a convolutional neural network outperforms the SMC algorithm on a fully-connected neural network.

Keywords— Sequential Monte Carlo, neural network, deep learning, non-stationary time series, forecasting, Bayesian neural network, convolutional neural network

Preface

This thesis has been submitted for the degree Master of Science in Applied Mathematics, with the specialization Financial Engineering at the Delft University of Technology. The academic supervisor of this thesis was Prof.dr.ir. C.W. Oosterlee, professor at the Numerical Analysis group of the Delft Institute of Applied Mathematics. The close supervisor was Dr. A. Borovykh, staff member of the CWI Amsterdam. The most work for this thesis was done at KPMG, were L. Tegels was my supervisor.

Firstly, I would like to thank my supervisors Prof.dr.ir. C.W. Oosterlee, and especially Dr. A. Borovykh for their assistance and support in writing this thesis. I would also like to thank Dr. J. Bierkens for being part of the examination committee. Furthermore, I would like to thank L. Tegels, for being my supervisor at KPMG where I wrote this thesis at, providing a good and motivating working environment. Lastly, I would like to thank my family, friends and colleagues at KPMG, for their encouragement and support.

Jasper E. Hoogendoorn
The Hague, June 2019

Table of Contents

List of Figures

List of Tables

List of Symbols

The next list describes several symbols that will be later used within the body of the thesis

α_n	Incremental importance weight at time step n
$\bar{\mu}_l$	Initial variance of weights in layer l
$\bar{q}_\theta$	Parametric distribution to be optimized with variational inference
$\bar{w}_n$	Unnormalized importance weight
δ^l_j	The error in neuron j of the lth layer
δ_{x_0}	Dirac delta function
γ_n	Pointwise known density
$\hat{\mathbf{y}}_k$	Neural network output at time step k
$\mathbf{b}_k$	Biases of neural network at time step k
$\mathbf{w}_k$	Weights of neural network at time step k
μ	Prior density in state-space model
π_n	Sequence of n densities
σ	Activation function in neural network
θ	Parameter of a parametric distribution
a^l_j	Activation value of neuron j in layer l
b^l_j	Bias of neuron j in layer l
C	Cost function of neural network
f	First transition density in state-space model
g	Second transition density in state-space model
$I_n(\phi_n)$	Expectation of test function ϕ at n time steps
K_n	Markov transition kernel
N	The total number of Monte Carlo samples
n_l	Number of neurons in layer l
Q	The noise of the transition function
q_n	Importance density at time step n

R	The width of the likelihood
T	The threshold for the ESS
W_n^i	Normalized importance weights for Monte Carlo sample i
w_{jk}^l	Weight of connection between neuron k in layer $l-1$ and neuron j in layer l
z_j^l	Output of neuron j in layer l
Z_n	Normalizing constant

1 Introduction

Machine Learning (ML) has gained significant popularity in recent years with it being applied to solve a vast amount of different problems. One particularly popular machine learning algorithm is called a Neural Network (NN). These neural networks were inspired by the workings of the human brain, with interconnected neurons performing certain computations. The ability of modern computers to be able to train very deep neural nets has resulted in significant achievements in various applications. These nets are able to find specific patterns in a way that is similar to or even exceeds human performance. Spectacular breakthroughs have been achieved in the field, such as the ability to outperform professional players in the board game alphaGo Silver et al. [2016] or AI that learns to play videogames Mnih et al. [2013]. At the same time, many industries employ ML algorithms in their daily practices, e.g., image recognition used by Instagram and Facebook or reinforcement learning used by companies deploying self-driving cars. While these results are promising, recent work also indicates that machine learning algorithms do not yet achieve human performance; in particular, there is the vulnerability of the networks to specific adversarial examples Goodfellow et al. [2014] or changes in the underlying data distribution. Nevertheless, these methods provide new ways of modeling data which can be superior to existing methods. However, the development of algorithms which are stable and robust is still an active topic of research.

1.1 Uncertainty in neural networks

According to the National Institute of Standards and Technology (NIST), no measurement is complete without an accompanying statement of the associated amount of uncertainty. The same could be said about the uncertainty in model predictions or outcomes for the future. Uncertainty is critical to risk assessment and decision making. At the same time, people see neural networks and machine learning methods as black box algorithms where one does not know the internal processes that are used in obtaining the parameters of the method. These two issues can be a problem when adopting these algorithms in fields where risk measures and decisions are important. Organizations make decisions every day based on reports containing quantitative measurement data and predictive models. If model or prediction results are not accurate the decision risk increases. Giving a wrong diagnose using a machine learning model, for example, could result in wrong treatment and thus potentially harming a patient.

Obtaining the uncertainty of neural networks is becoming increasingly important nowadays due to the rising number of their applications. The predictions of neural networks are deterministic in the sense that the outputs are point estimates and do, in general, not include a measure of uncertainty. The work of Ghahramani [2001] shows that having a measure of uncertainty is crucial in modern neural networks as, when trained with the standard gradient descent algorithms, these networks output quantities which do not have an error bound and thus the predictions from the neural network could be used with false confidence. Such observations contributed to the recent increase of interest in Bayesian neural networks (BNN). These networks are trained in a probabilistic manner such that a measure of uncertainty can be computed by using the obtained posterior distribution over the outcome. Bayesian neural networks are not new and have been studied extensively in e.g., MacKay [1992] and Neal [1992]. Using Bayesian inference methods for large neural networks is complicated due to computational time. The modern neural networks architectures used in e.g., image recognition typically contain millions of parameters, resulting in the standard Bayesian methods to be computationally infeasible. Novel ways of obtaining posterior distributions in such deep neural networks have been proposed, the two main methods being variational inference (e.g., Graves [2011]) and Monte Carlo methods (e.g., Freitas et al. [2000]). Variational inference is based on an analytic approximation to the posterior distribution, while Monte Carlo methods are based on obtaining samples from the posterior and can be computationally expensive.

1.2 Non-stationary and sequential data for neural networks

Neural networks are typically trained with gradient descent algorithms, such as stochastic gradient descent (SGD), where, given a particular dataset, the network is optimized to find the weights that minimize the discrepancy between the dataset and the network outputs. Such algorithms are good in finding patterns, but these patterns should, in general, be stationary. Stationary data implies that the data has constant statistical properties over time, such as constant mean, homoscedasticity, and autocorrelation independent of time. Under such stationarity conditions, the predictive performance of the neural network on the training dataset should be close to the performance on the test dataset. However, the data used to train a neural network does not necessarily have to be stationary; in particular, the test data does not need to have the same statistical properties as the training data. In this setting, the predictive strength of the model decreases. This is particularly relevant in time series forecasting, where it is common that the time series is non-stationary. Furthermore, time series can have a low signal-to-noise ratio. These properties complicate the prediction problem. In particular, the standard algorithms used for training networks on i.i.d. data might not perform well, and it is of the essence to understand the limitations of the standard algorithms and to develop new algorithms for training neural networks on these non-i.i.d., sequential datasets.

Considering non-stationary time series, where the data is sequential, it is not straight forward to understand which data points should be used for the training set. Due to the non-stationarity of the data certain properties might change over time. Furthermore, the neural network might not capture the new patterns due to the limited amount of new data points. Also, it may be problematic to decide when a pattern has changed considering the sequentially arrived time steps. This so-called change point detection is an active field of research, which helps to quantify a potential change in the pattern. If a change point is detected, the neural network needs to be re-trained manually for the specific data points. Recent techniques like recurrent neural networks (RNN) have been shown to solve several of these problems, and have been successfully applied in time series forecasting settings. However, these networks are known to be difficult to train.

1.3 Research goals and outline of thesis

To overcome the possible problems described above, in Freitas et al. [2000], an alternative method for training neural networks was proposed, based on a sequential Monte Carlo (SMC) algorithm. This method can be useful for training a neural network on non-stationary and sequential data due to the sequential nature of the algorithm. The method is well-suited for problems consisting of non-linearity and non-Gaussian signals. The sequential Monte Carlo method does not require any assumptions on the data, making it a very flexible method. Furthermore, the SMC algorithm outputs a posterior distribution over the outcome and the underlying weights. This posterior distribution can be used as a measure of uncertainty, which is very useful, as stated above, in practical implementations. At the same time this method, in theory, is better suited to find a global optimum for the network than the gradient-based method as with such methods neural networks can get stuck in local optima. The downside of the SMC method is the increased computing time compared to these standard gradient descent algorithms. In particular, for the large networks used in applications such as in image recognition, the Monte Carlo methods are not feasible anymore. At the same time, in such image recognition tasks, the data is typically i.i.d., so that the SMC algorithm might not have significant added value for these problems. On the contrary, for time series forecasting typically relatively small neural networks are used, which makes the SMC algorithm well-suited for such networks. Furthermore, due to the sequential and non-stationary nature of the data, the SMC algorithm might be much better suited for training the network than the typical gradient methods.

The main scope of this thesis is to gain insight into the benefits and downsides of the SMC algorithm when applied to time series forecasting with deep neural networks. Monte Carlo methods, and in particular the SMC method, in combination with a neural network, has not been researched extensively. The recent emphasis has been on variational inference methods for obtaining the posterior distribution of the network outputs, due to the computational efficiency of this method on larger neural networks. For these reasons, it interesting to extend the work of Freitas et al. [2000], and to understand the benefits of training neural networks in the context of time series forecasting with the SMC algorithm compared to the widely used SGD method. We thoroughly test the SMC algorithm for neural networks on different time series and compare it to the SGD training method. We will study the effects of the various hyperparameters of SMC on its performance, and propose and test possible improvements to the SMC algorithm. We furthermore extend the SMC algorithm from Freitas et al. [2000] for fully-connected neural networks to a convolutional network structure and gain insight into the performance on such a network. Furthermore, we use the obtained posterior distribution over the weight and outcomes to see how these distributions behave for neural networks and if these give a proper measure of uncertainty.

The rest of this thesis is structured as follows: we start in Chapter 2 with a general explanation into neural networks and discuss the standard training algorithm, namely stochastic gradient descent. As the sequential Monte Carlo method is commonly used for state-space models, we show how a neural network translates to a state-space model and discuss how the sequential Monte Carlo method is derived for these models in Chapter 3. After the basic framework for neural networks and sequential Monte Carlo, several improvements for the algorithm on neural networks are explained in Chapter 4. In Chapter 5, the numerical results are presented for different time series.

2 The Basics of Neural Networks

In this section, neural networks are explained following Nielsen [2015]. After covering the basics, we explained the backpropagation algorithm used to train most of the neural networks.

2.1 Fully-connected neural network

We explain here how a fully-connected neural network (FNN) works, covering the structure of an FNN and the training method that is used most commonly for these types of neural networks.

2.1.1 Perceptron network

The simplest form of a neural network is known as a perceptron. A perceptron has binary inputs $\mathbf{x}_n = (x_1, \ldots, x_n)$ and the output is also binary i.e., zero or one. This output is dependent on a set of so called weights $\mathbf{w}_n = (w_1, \ldots, w_n)$. Figure 1 shows an example of a perceptron unit.

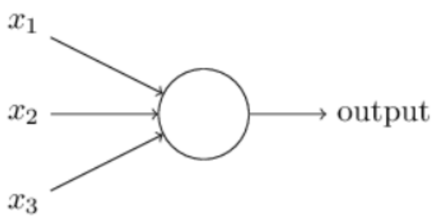


Figure 1: Example of one perceptron network

The combination of weights and inputs will determine the output of this perceptron neuron in the following way

$$\text{output} = \begin{cases} 0, & \text{if } \sum_i w_i x_i \leq \text{threshold} \\ 1, & \text{if } \sum_i w_i x_i > \text{threshold} \end{cases} \tag{1}$$

The weights measure how "important" the input is. The threshold defines when the output is zero or one, i.e., how high do we want the product of inputs and weights to be to "activate" the perceptron. This means that varying the weights, but also the threshold, can result in different models.

2.1.2 Deep fully-connected neural network

An extension of the basic perceptron algorithm is done by adding multiple layers of perceptrons. In the literature, this is known as a multilayer perceptron (MLP) and often FNN and MLP are used interchangeably. In Figure 2, an example is given. The inputs are given by an input layer, which can be seen as a vector of inputs $\mathbf{x}_n$, where every element is connected to all the neurons in the next layer, hence the name fully-connected neural network. These next layers are called hidden layers, of which there can be one or more.
With multiple layers of perceptrons we can rewrite Eq. (1) in a standard set of equations seen in the literature of neural networks. For a neural network with L layers and n_l

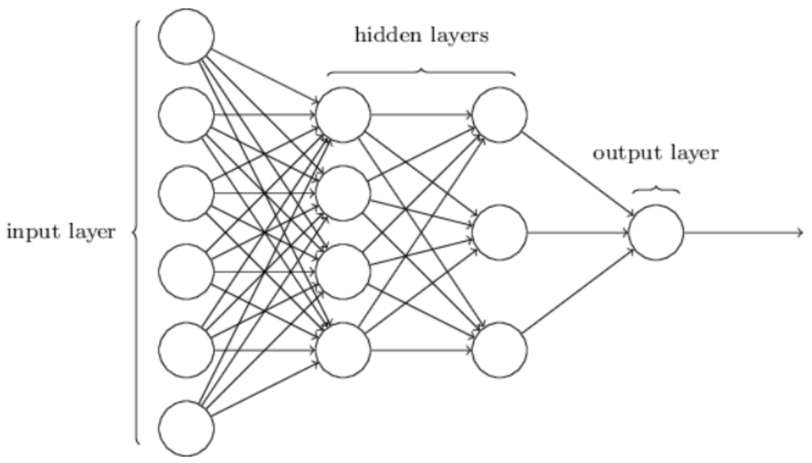


Figure 2: Multilayer Network

neurons per layer $l = 1, \ldots, L$, we have the following equation:

$$a_j^l = \begin{cases} 0, & \text{if } \sum_k w_{jk}^l a_k^{l-1} + b_j^l \leq 0 \\ 1, & \text{if } \sum_k w_{jk}^l a_k^{l-1} + b_j^l > 0 \end{cases} . \tag{2}$$

Here, we have $j = 1, \ldots, n_l$ for the neurons in layer l and $k = 1, \ldots, n_{l-1}$ for the neurons in layer $l - 1$. Furthermore, we have $b_j^l = -$threshold, which is called the bias and a_j^l is the activation value of a neuron.

One needs to find the values of w_{jk}^l and b_j^l so that when the input is passed through the network, the output of the network matches as closely as possible the given output. In other words, the parameters w_{jk}^l and b_j^l need to be "learned" by minimizing some kind of discrepancy function between given outputs and the networks' outputs. We use **w** and **b**, to denote all the different weights and biases respectively.
In essence, the way a neural network learns is to understand how a change in the weights and biases changes the output. In mathematical terms, we are interested in the influence of $\Delta\mathbf{w}$ on Δoutput. To obtain good performance, a small change in the weights should give a proportional change in the output. The output of a neuron is influenced by the *activation function* $\sigma(\cdot)$. One commonly used activation function is the sigmoid function, also known as the logistic function,

$$\sigma(a) = \frac{1}{1 + e^{-a}} .$$

With an activation function, the equation Eq. (2) for the activation values results in the following equations:

$$\begin{aligned} a_j^l &= \sum_k w_{jk}^l z_k^{l-1} + b_j^l , \\ z_j^l &= \sigma(a_j^l) . \end{aligned} \tag{3}$$

Here, z_j^l is the output of a neuron and calculating z_j^l is called *forward propagation* as the information flows forward through the FNN. Using the sigmoid function as the activation function the neuron has a value near zero for low values of a_j^l and a value near one for high values of a^l_j, similar to the perceptron. The smoothness of the sigmoid function results in the ability to differentiate the output of the neuron with respect to the network parameters **w** and **b**, so one can evaluate how changes in the weights or biases change the output.

2.1.3 Training a neural network

A neural network is trained using training data, where (x_i, y_i) for $i = 1, \dots, K$ is the sample set. In general $(x_i, y_i) \sim \mathcal{D}$, where $\mathcal{D}$ is the data distribution which is unknown. (x_i, y_i) consists of inputs x with the corresponding outputs y, and can be multi-dimensional denoted as $(\mathbf{x}_i, \mathbf{y}_i)$. The inputs, $\mathbf{x}_i$, of the training data are used as input to the neural network, and using the weights and biases this results in a certain predicted output $\hat{\mathbf{y}}_i$. The performance of the network is measured using a cost function (the term loss function is also used). There exists a large number of possible cost functions, and each has its own advantages and uses. The cost function defines a loss surface for all the possible values of the weights **w** and biases **b**, as the cost is calculated by measuring a type of distance between $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i(\mathbf{x}_i, \mathbf{w}, \mathbf{b})$. For deep neural networks, this loss surface typically consists of multiple local minima, and a global minimum might exist where the cost is the lowest. For this global minimum, this specific set of weights and biases, minimize the loss function and thus result in the network fitting the training data best. The idea of training a network is to find this minimum or at least a sufficiently good local minimum. The cost function that is considered here is the sum-of-squares error function

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2} \sum_{i=1}^{K} ||\mathbf{y}_i - \hat{\mathbf{y}}_i(\mathbf{x}_i, \mathbf{w}, \mathbf{b})||^2 . \tag{4}$$

Most optimization techniques start with an initial vector for the weights, $\mathbf{w^0}$, and biases, $\mathbf{b^0}$, and iteratively move through the loss surface by taking small steps in the direction of the negative gradients of the loss function. Generally speaking, for iterations $t = 1, ..., T$,

$$\begin{aligned} \mathbf{w}^{t+1} &= \mathbf{w}^t + \Delta \mathbf{w}^t , \\ \mathbf{b}^{t+1} &= \mathbf{b}^t + \Delta \mathbf{b}^t . \end{aligned} \tag{5}$$

With a change of $\Delta \mathbf{w}^t$ the error function changes with

$$\Delta C \approx \Delta \mathbf{w}^{tT} \nabla C(\mathbf{w}) .$$

Here, $\nabla C(\mathbf{w})$ is called the gradient and gives the direction in which ΔC changes the most. Particularly, when $\nabla C(\mathbf{w}) = 0$ the error does not change anymore in any of the directions. These points are called *stationary points* and can be classified as minima, maxima or saddle points. The training of a neural network consists of finding the weights **w** and biases **b** that result in the smallest error $C(\mathbf{w}, \mathbf{b})$.

To find this smallest error, a primary method is called *gradient descent.* Here we use Eq. (5) and update it each iteration with information of the gradient. In mathematical terms

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla C(\mathbf{w}^t) .$$

The same equation holds for $\mathbf{b}^t$ and the *learning parameter* η is introduced which denotes how fast the algorithm learns. The error equations depend on the training set $(\mathbf{x}_i, \mathbf{y}_i)$.

The gradient ∇C for the whole data set can be calculated by calculating the gradients of each data point ∇C_i. We then get the gradient by

$$\nabla C = \frac{1}{K} \sum_{i=1}^{K} \nabla C_i \, .$$

Calculating the gradient for one data point at a time is called *on-line* training. The training methods where the data set is split up in different batches are called *batch* methods. For batch methods the gradient is calculated over the loss function applied to a small subset of the data in each iteration t.
Considering the on-line training case, we have the following update equation:

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \nabla C_i(\mathbf{w}^{t}) \, .$$

The gradient is thus computed per data sample. We have equations propagating through the network Eq. (3) and the gradient of the cost function Eq. (4). We need to evaluate the gradient which means finding

$$\nabla C_i = \left(\frac{\partial C_i}{\partial w_{j1}}, \ldots, \frac{\partial C_i}{\partial w_{jn_l}} \right) ,$$

for one layer, which for a certain weight w_{jk}^l gives

$$\frac{\partial C_i}{\partial w_{jk}^l} = \frac{\partial C_i}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \, , \tag{6}$$

using the chain rule given that C_i depends on a_j^l, which itself depends on the weights w_{jk}^l. In the literature, *errors* are defined as

$$\delta_j^l \equiv \frac{\partial C_i}{\partial a_j^l} \, .$$

Considering Eq. (3) the other term can be written as

$$\frac{\partial a_j^l}{\partial w_{jk}^l} = z_k^{l-1} \, .$$

and for the bias we have

$$\frac{\partial a_j^l}{\partial b_j^l} = 1 \, ,$$

which only leaves δ_j^l. Now we can rewrite Eq. (6) as

$$\frac{\partial C_i}{\partial w_{jk}^l} = \delta_j^l z_k^{l-1} \, ,$$

and for the bias we have

$$\frac{\partial C_i}{\partial b_j^l} = \delta_j^l \, . \tag{7}$$

The consequence of applying the chain rule is that we need to evaluate the δ's for the hidden units and output. The *error* vector of the output unit is straight forward

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} = \frac{\partial C}{\partial z_j^L}\frac{\partial z_j^L}{\partial a_j^L}\,.$$

With $z_j^L = f(a_j^L)$, and the derivative of the cost function Eq. (4) we simplify the the above equation for

$$\delta_j^L = \sigma'(a_j^L)(\mathbf{y}_i - \hat{\mathbf{y}}_i(\mathbf{x}_i, \mathbf{w}, \mathbf{b}))\,.$$

For the hidden layers the chain rule is used again

$$\delta_k^{l-1} \equiv \frac{\partial C_i}{\partial a_k^{l-1}} = \sum_j \frac{\partial C_i}{\partial a_j^l}\frac{\partial a_j^l}{\partial a_k^{l-1}}\,.$$

Here, we sum over all the neurons in the lth layer. With $\sigma'(z)$ denoting the derivative of the activation function, this can be rewritten to

$$\delta_k^{l-1} = \sigma'(a_k^{l-1})\sum_j w_{jk}^l \delta_j^l\,. \tag{8}$$

As we have

$$\frac{\partial a_j^l}{\partial a_k^{l-1}} = \sigma'(a_j^{l-1})w_{jk}^l\,.$$

The formula Eq. (8) is called *back propagation* as the errors are propagated backwards through the network. This covers the algorithm behind the gradient descent method.

2.1.4 Stochastic gradient descent method

The most standard method for training a neural network is known as stochastic gradient descent. Unlike the on-line learning algorithm where the gradient is calculated over a single data sample, in SGD at each iteration a random set of data points x_j is selected, the number of these points known as the batch size. The gradient is then computed over over these random points. The gradient of a batch is calculated as follows:

$$\nabla C = \frac{1}{m}\sum_{j=1}^{m}\nabla C_j,$$

where m is the number of data points in the batch. We remark that the gradient over such a batch of data can be seen as a noisy estimate of the full gradient, i.e. the gradient computed over the full dataset. One *epoch* is defined as the number of iterations needed for the whole data set to be used. Learning in batches can significantly improve the learning speed compared to full gradient descent, and thus result in faster convergence of the weights in the neural network. This SGD method is used as a training method in many state-of-the-art FNN's. Taking $m = 1$ translates this to the on-line learning method.

2.1.5 Overfitting

The overall performance measure of the model is in terms of the average of the errors on the training and test data sets. The Root Mean Squared Error (RMSE) defined in Eq. (9) is one of these measures that is widely used in assessing the performance of the trained neural network, i.e.,

$$RMSE = \sqrt{\frac{\sum_{i=1}^{K}(\hat{y}_i - y_i)^2}{K}}. \tag{9}$$

A problem that arises with neural networks is overfitting the data. In this setting, the training error is significantly lower than the test error. The aim of training a neural network is to obtain good generalization, i.e., make sure that the training error does not differ significantly from the test error. In particular, when increasing the number of layers and neurons of the neural network and making the number of weights and thus variables to be tuned arbitrarily large, overfitting can become a significant problem. In Hornik et al. [1989] it was shown that a sufficiently wide neural network could arbitrarily well fit any function at hand; this is known as the *universal approximation theorem*. Several techniques have been proposed to reduce the problem of overfitting, such as the regularization of the cost function, i.e., constraining the weight values. One could view this problem as a variance-bias trade-off, where under-fitting the data means low variance, but a high bias. An easy way to see if a model is overfitting can be as follows:

$$\text{RMSE over the training set} << \text{RMSE over the test set} \rightarrow \text{overfitting}$$

2.2 Convolutional neural network

In the last section, we discussed a fully-connected neural network. A different type of neural network is so-called a convolutional neural network (CNN). These networks were shown to be particularly powerful for image recognition Krizhevsky et al. [2012]. In this thesis, we will apply the SMC algorithm on a specific type of CNN suitable for time series called the Wavenet as first introduced in van den Oord et al. [2016]. To understand the structure of CNN, the basics of the convolutional neural network is discussed where Borovykh [2018] is followed in explaining the basic operations in CNN's.

2.2.1 Structure of a convolutional neural network

Within FNNs, the neurons are fully connected, and every connection has its own weight. The major differences between an FNN and a CNN are the usage of shared weights and local connectivity. Shared weights mean that multiple neurons have the same weights, and local connectivity means that a neuron is only connected to a specific sub-region of the input. These features of CNN's result in fewer weights to be trained. Since one of the major disadvantages of using Monte Carlo methods for neural networks is the large number of weights that need to be trained, having fewer weights makes CNN's potentially more suitable for the SMC algorithm as it is computationally less expensive. Figure 3 shows how an FNN is similar to a CNN.

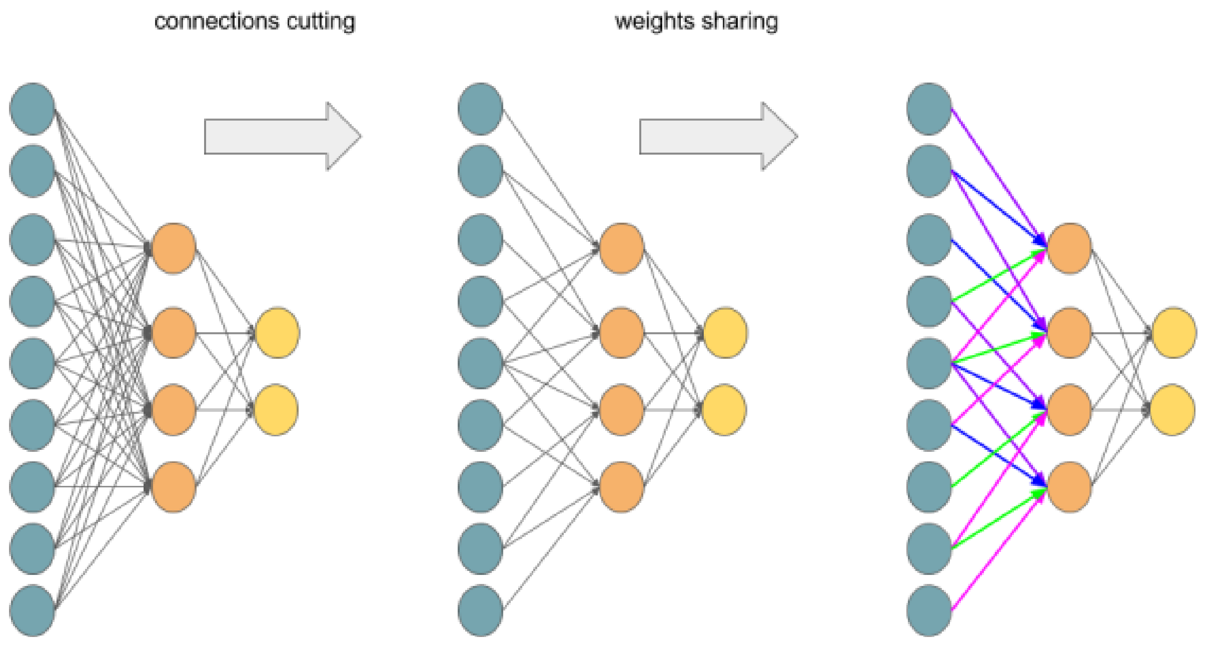


Figure 3: Transformation of neural network to convolutional neural network.

Instead of the forward propagation by matrix multiplication seen in Eq. (3) a convolutional operation is performed to calculate the activations in the next layer. A convolution is defined as follows, and denoted by an asterisk $*$:

$$(h * g)(t) = \int_{-\infty}^{\infty} h(\tau)g(t-\tau)d\tau\,.$$

Convolutions are commutative which means $(h * g) = (g * h)$. In the literature, CNN's are mostly 2D or 3D, but for time series, we only use a 1D approach, which means we have a discrete one-dimensional convolution. The convolution may be truncated to where the convolution has values different from zero, else one uses so-called *zero padding*, which puts zeros on the place where both samples are nonexistent. Given h has N values and g has M values we obtain:

$$(h * g) = \sum_{\tau=0}^{M-1} h(\tau)g(t-\tau) = \sum_{\tau=0}^{M-1} h_\tau g_{t-\tau}\,,$$

where the output dimension is $N-M+1$ when no zero padding is used, so $t = 0, \dots, N-M$. Translating this to neural networks, we have,

$$a_j^l = \sum_{r=-\infty}^{\infty} w_r^l z_{j-r}^{l-1} + b_j^l\,,$$
$$z_j^l = \sigma(a_j^l)\,.$$

Here, the infinite sum can also be truncated to where the convolution is not zero. Furthermore, r is the indicator for the weight in the 1D convolution of filter size κ, where for a 1D case we have the weights $\mathbf{w}_r^l = (w_1^l, \dots, w_\kappa^l)$ in layer $l = 1, \dots, L$. The number of neurons in the next layer depends on this filter size κ. There can be multiple channels that each can represent different time series of the same length. This thesis considers only one channel, although this can be extended to more, as in Borovykh [2018]. The number of neurons in each layer can be found by $n_l = n_{l-1} - \kappa + 1$.

2.2.2 WaveNet

Most traditional CNN's use structures as seen in Figure 3. However, in sequential forecasting tasks a specific structure for the CNN, the WaveNet van den Oord et al. [2016], has been particularly successful due to its ability to take into account longer histories of the input. WaveNet was initially introduced for modeling speech. WaveNet has fewer weights that need to be trained, which means it may be a more efficient structure to apply SMC on. Figure 4 shows how the weights propagate through the network. The input layer consists of the time series which is used for a 1D convolution of size two. The filter size $\kappa = 2$ means the receptive field, i.e., how many neurons of the previous layer each output node sees, is two, and that there are two trainable weights per layer.

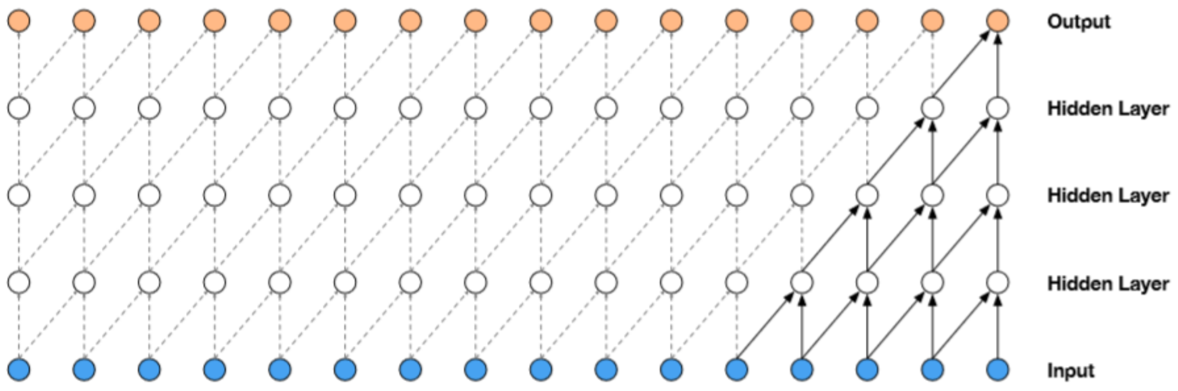


Figure 4: Architecture of WaveNet convolutional neural network.

2.2.3 Training for convolutional networks

The gradient step in the SMC algorithm for CNN's is slightly different from the gradient step in FNN's with SMC. We show here the backpropagation that is used for modeling time series with a CNN in a one-dimensional setting. We have the following equations for calculating the propagating errors δ_j^l in the CNN case, using the same notations as in Section 2.1.3,

$$\begin{aligned}
\delta_k^{l-1} \equiv \frac{\partial C}{\partial a_k^{l-1}} &= \sum_j \frac{\partial C}{\partial a_j^l} \frac{\partial a_j^l}{\partial a_k^{l-1}} \\
&= \sum_j \delta_j^l \frac{\partial (\sum_{r=-\infty}^{\infty} w_r^l \sigma(a_{j-r}^{l-1}) + b_j^l)}{\partial a_k^{l-1}} \\
&= \sum_j \delta_j^l w_r^l \sigma'(a_k^{l-1}) \text{ ,} with\ k = j - r\ \ as\ all\ other\ terms\ are\ zero \\
&= \sum_j \delta_j^l w_{j-k}^l \sigma'(a_k^{l-1}) \text{ ,} with\ r = j - k \\
&= \delta_j^l * w_{-r}^l \sigma'(a_k^{l-1})
\end{aligned} \tag{10}$$

Here, w^l_{-r} means a 180 degree rotation of the weight matrix, which in the 1D case is a vector. The backpropagation equation, Eq. (10), for CNN looks similar to Eq. (8) in the FNN case, only it has a convolution operation where the weights are rotated. The according changes for the biases are the same as in Eq. (7), but for the weights, we again need a convolution. Using Eq. (6) this leads to:

$$\begin{aligned}\frac{\partial C}{\partial w^l_r} &= \sum_j \delta^l_j \frac{\partial(\sum_{r'=-\infty}^{\infty} w^l_{r'}\sigma(a^{l-1}_{j-r'}) + b^l_j)}{\partial w^l_r} \\ &= \sum_j \delta^l_j \sigma(a^{l-1}_{j-r}) \quad \textit{,as we have zero for } r' \neq r \\ &= \delta^l_{-j} * z^{l-1}_k \,.\end{aligned}$$

Where $-j$ again stands for a rotation of 180 degrees on the error matrix.

3 Bayesian methods

This section gives an in-depth explanation of Bayesian inference. Firstly, the basics of Bayesian inference and problems that arise are discussed. Secondly, variational inference and sequential Monte Carlo are explained, and we provide the general framework for the SMC algorithm so that we can translate this to a neural network setting.

3.1 Bayesian inference

Bayesian inference is based on a prior and a likelihood, and the obtained posterior is a probability over the possible outcomes. Frequentists often reside on maximum likelihood estimation, which gives the most probable value of the parameter of interest. Bayesian inference provides a distribution over this parameter; in other words, this gives an uncertainty of the parameter. This can be useful in a setting in which data is scarce or when overfitting the data can be a problem. Another feature of Bayesian inference is that one can add a "prior" belief to the model with the argument that one often has an intuition about the problem at hand. Especially when data is limited, this can provide intuitively better estimates of parameters. The estimate of the parameter is computed as a posterior distribution that is given by Bayes' rule:

$$p(x|y) = \frac{p(x)p(y|x)}{p(y)}\,. \tag{11}$$

Here, $p(x)$ is the prior, $p(y|x)$ the likelihood and $p(y)$ the normalizing constant. The normalizing constant is calculated by integrating over the parameters of the joint probability $p(y, x)$. These give the posterior distribution $p(x|y)$ where the parameter of interest can be obtained through the posterior mean or the mode. Here, x indicates the parameter and y denotes the available data. In certain problems and with "uninformative" priors the frequentist and the Bayesian method may give the same outcome.

3.2 Problems with Bayesian inference

While Bayesian inference gives a distribution on the parameters of interest, this comes at a cost. The normalizing constant is often a hard to compute integral, meaning that often there is no closed form solution and this integral is in some cases high dimensional. In particular for neural networks the normalizing constant can be of high-dimension, making the full posterior intractable. The posterior can be approximated with numerical methods, but this is limited by the computing power available. Another problem for Bayesian inference is the choice of the prior as this can change the outcome of the posterior. In our setting, where we use Bayesian inference in a sequential manner, the influence of the prior will disappear after an increasing number of data points are used. Also, MacKay [1995] noted that the prior could introduce some regularization effect in the form of Occam's razor, where the prior regularizes the model in a way that it automatically tends to a simpler model, as by Occam's razor this should be favored over more complex ones. This is beneficial to the overfitting problem that neural networks have as more complex networks tend to overfit, therefore increasing the robustness, and predictive abilities for neural networks.

3.3 Different methods for Bayesian analysis

To derive the posterior distribution, one can use two different approaches. One approach is based on the class of Markov Chain Monte Carlo methods (MCMC), of which the SMC method is a subcategory. The other approach is called variational inference (VI). The difference between the two mainly relates to the bias-variance trade-off and computational complexity. The variational methods have a larger bias since a parametric density is used to approximate the underlying real density. This means that the sample variance is zero as one can sample directly from this density. For the MCMC methods, it is the other way around, the bias tends to be close to zero, and sometimes even equal to zero. The variance, however, is dependent on sample size and the underlying method used, which is related to the second difference: the computational time. Acceptable sample variances may require significant amounts of computation time. Variational methods are more efficient as the variance is not dependent on the sample size. Another significant advantage in using VI is that as a known parametric density is used, gradients can be computed. This leads to a back propagation algorithm similar to the gradient descent algorithm used for neural networks. This makes VI significantly faster than MCMC methods, and this advantage increases as the networks become bigger. The disadvantage arises when handling complex models that have distributions that do not fall into a parametric class of distributions from which the approximating distribution is chosen or have multi-modality, such as deep neural networks. Hybrid models possess the advantages of both methods. Recently, Gu et al. [2015] looked at SMC methods with a proposal density that was weighted with a Kullback-Leibler (KL) divergence. Future algorithms could implement a wake-sleep procedure, as seen in Hinton et al. [1995], which may improve results even further. In this thesis, we focus on the basic SMC method for Bayesian inference on the weights of neural networks.

3.4 Variational inference in Bayesian neural networks

This section explains the idea behind using variational inference for Bayesian neural networks. Algorithms based on variational inference are used in most recent papers on BNN's. As the algorithms are based on a complex method, a better understanding is provided in this section.

3.4.1 Variational inference

The basis of this method is to use a class of parametric distributions, that is "close" to the distribution of interest $\pi(x)$. To be more precise, we want to find a parametric distribution $\bar{q}_\theta(x)$, where θ denotes the parameters of $\bar{q}_\theta(x)$. θ is optimized such that $\bar{q}_\theta(x)$ is close to $\pi(x)$. In our setting $\pi(x)$ is the posterior distribution $p(x|y)$ in a Bayesian inference problem.

A measure to quantify this distance is the Kullback-Leibler divergence measures how close the distribution $\bar{q}_\theta(x)$ is to $\pi(x)$. A small value indicates a better approximation of $\pi(x)$. The KL-divergence is defined as follows given two distributions P and Q

$$\mathbb{KL}(P||Q) = \mathbb{E}_P\left(\log\left(\frac{P}{Q}\right)\right).$$

In our setting we aim to approximate the posterior distribution in a Bayesian framework so that,

$$\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)) = \mathbb{E}_{\bar{q}_\theta}(x)\left(\log\left(\frac{\bar{q}_\theta(x)}{p(x|y)}\right)\right).$$

Using Bayes' rule, Eq. (11), we can substitute for $p(x|y)$ obtaining

$$\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)) = \mathbb{E}_{\bar{q}_\theta}(x)\Big(\log\Big(\frac{\bar{q}_\theta(x)p(y)}{p(y|x)p(x)}\Big)\Big).$$

We then can derive the following equation

$$\begin{aligned}\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)) &= \mathbb{E}_{\bar{q}_\theta}(x)\Big(\log\Big(\frac{\bar{q}_\theta(x)}{p(y|x)p(x)}\Big)+\log p(y)\Big)\\ &= \mathbb{E}_{\bar{q}_\theta}(x)\Big(\log\Big(\frac{\bar{q}_\theta(x)}{p(y|x)p(x)}\Big)\Big)+\mathbb{E}_{\bar{q}_\theta}(x)(\log p(y))\\ &= \log p(y)+\mathbb{E}_{\bar{q}_\theta}(x)\Big(\log\Big(\frac{\bar{q}_\theta(x)}{p(y|x)p(x)}\Big)\Big).\end{aligned}$$

The last step follows as $p(y)$ is independent of $\bar{q}_\theta(x)$. We aim to minimize the KL divergence as a function of θ. Since $\log p(y)$ does not depend on θ, only the last term at the right-hand side needs to be minimized:

$$\mathbb{E}_{\bar{q}_\theta}(x)\Big(\log\Big(\frac{\bar{q}_\theta(x)}{p(y|x)p(x)}\Big)\Big) = \mathbb{E}_q[\log\bar{q}_\theta(x)-(\log p(y|x)+\log p(x))]$$
$$.$$

This is equivalent to maximizing the negation of this formula,

$$\begin{aligned}\mathcal{L} &:= \mathbb{E}_{\bar{q}_\theta}(x)\,[-\log\bar{q}_\theta(x)+\log p(y|x)+\log p(x)]\\ &:= \mathbb{E}_{\bar{q}_\theta}(x)\left[\log p(y|x)+\log\left(\frac{p(x)}{\bar{q}_\theta(x)}\right)\right].\end{aligned}$$

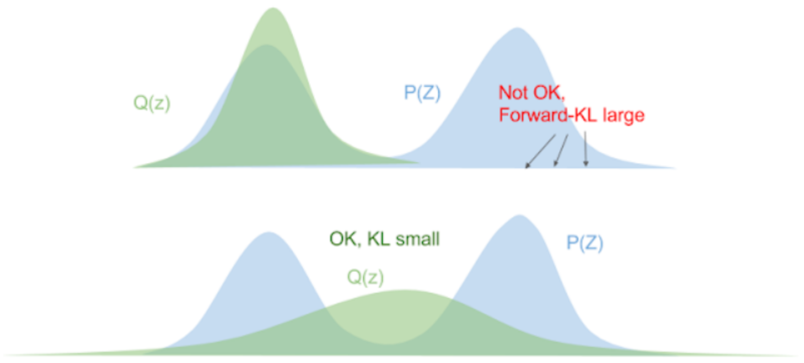
$\mathcal{L}$ is called the *variational lower bound.* This is computable as we can choose and calculate the densities in the expression. Note that,

$$\begin{aligned}\mathcal{L} &= \mathbb{E}_q[\log p(y|x)]+\mathbb{E}_{\bar{q}_\theta}(x)\left[\log\frac{p(x)}{\bar{q}_\theta(x)}\right]\\ &= \mathbb{E}_{\bar{q}_\theta}(x)\left[\log\frac{p(x|y)p(y)}{p(x)}+\log\frac{p(x)}{\bar{q}_\theta(x)}\right]\\ &= \log p(y)-\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)) \quad , \textit{as } p(y|x)=\frac{p(x|y)p(y)}{p(x)}.\end{aligned} \tag{12}$$

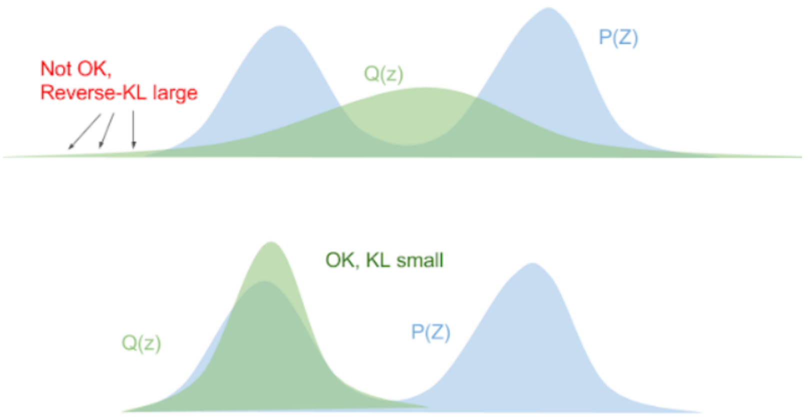
In other words,

$$\log p(y) = \mathcal{L}+\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)).$$

As we know that $\mathbb{KL}(\bar{q}_\theta(x)||p(x|y)) \geq 0$, we have that the log marginal likelihood $\log p(y) \geq \mathcal{L}$. This makes $\mathcal{L}$ a lower bound of the log marginal likelihood, also known as the *evidence lower bound* (ELBO). In other words, a good approximation is given by maximizing Eq. (12). It is important to note that the KL divergence is not a symmetric distance, i.e., it matters if we have $\pi(x)||\bar{q}_\theta(x)$ or $\bar{q}_\theta(x)||\pi(x)$. Therefore there are two ways of measuring the "distance" between two functions and these methods are called forward (zero-avoiding) and reverse KL (zero forcing). To shed some light on these terms, we first consider forward KL. With forward KL the KL is large if $\bar{q}_\theta(x)$ is small where $\pi(x)$ is large. On the other hand, when we consider reverse KL, we want to fit $\bar{q}_\theta(x)$ such that $\pi(x)$ is not small where $\bar{q}_\theta(x)$ is large. Figure 5, will give a visual explanation.



(a)



(b)

Figure 5: (a) Forward KL (b) Reverse KL

In variational Bayes for neural networks, the algorithms are based on reverse KL. This means that when the posterior is multimodal, which is often the case with neural networks, one gets false negatives when fitting a unimodal $\bar{q}_\theta(x)$.
We now needs to determine the form of $\bar{q}_\theta(x)$. One way is to use a mean-field approximation, which has its origins in physics. It is based on the assumption that there are multiple the hidden variables which are mutually independent, giving $\bar{q}_\theta(x) = \bar{q}_\theta(\mathbf{x})$, which can be factorized as

$$\bar{q}_\theta(\mathbf{x}) = \prod_{i=1}^{N} \bar{q}_\theta(x_i)\,.$$

A disadvantage is that this does not capture the correlation between hidden variables. On the other hand, this means that one can optimize the density per factor i.e., a local approximation for each variable.

3.4.2 Coordinate ascent variational inference

An algorithm that can be used to optimize the density per factor is called Coordinate Ascent Variational Inference (CAVI), first seen in Bishop [2007]. This is closely related to the *stochastic variational inference* that Hoffman et al. [2013] developed and is again similar to the gradient descent algorithm used for neural network training. In deriving the algorithm, we follow Murphy [2012].

Using the factorized form for $\bar{q}_\theta(x)$, we have the ELBO:

$$\begin{aligned}\mathcal{L} &= \mathbb{E}_{\bar{q}_\theta}(x)\left[\log \frac{p(x|y)p(y)}{p(x)} + \log \frac{p(x)}{\bar{q}_\theta(\mathbf{x})}\right], \\ &= \mathbb{E}_{\bar{q}_\theta(\mathbf{x})}[\log p(x,y) - \log \bar{q}_\theta(\mathbf{x})]\,.\end{aligned}$$

Factoring out a particular $\bar{q}_\theta(x_i)$,
defining $\mathbf{x}_{-i} := \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\}$ and $q_\theta(\mathbf{x}_{-i}) := \prod_{i\neq j} \bar{q}_\theta(x_i)$ we obtain:

$$\begin{aligned}\mathcal{L} &= \mathbb{E}_{\bar{q}_\theta(\mathbf{x})}[\log p(x,y) - \log \bar{q}_\theta(\mathbf{x})] \\ &= \int_{x_i} \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i})[\log p(x,y) - \log \bar{q}_\theta(\mathbf{x})] d\mathbf{x} \\ &= \int_{x_i} \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) \log p(x,y) d\mathbf{x} - \int_{x_i} \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) \log \bar{q}_\theta(\mathbf{x}) d\mathbf{x} \\ &= \int_{x_i} \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) \log p(x,y) d\mathbf{x} - \int_{x_i} \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) \sum_{i=1}^{N} \log \bar{q}_\theta(x_i) d\mathbf{x} \\ &= \int_{x_i} \bar{q}_\theta(x_i) \mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] dx_i \\ &\quad - \int_{x_i} \bar{q}_\theta(x_i) \log \bar{q}_\theta(x_i) \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) d\mathbf{x} \\ &\quad - \int_{x_i} \bar{q}_\theta(x_i) dx_i \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) \sum_{j\neq i} \log \bar{q}_\theta(\mathbf{x}_{-i}) d\mathbf{x}_{-i} \\ &= \int_{x_i} \bar{q}_\theta(x_i) \mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] dx_i \\ &\quad - \int_{x_i} \bar{q}_\theta(x_i) \log \bar{q}_\theta(x_i) dx_i \\ &\quad - \mathbb{E}_{\bar{q}_\theta(\mathbf{x}_{-i})}\left[\sum_{j\neq i} \log \bar{q}_\theta(\mathbf{x}_{-i})\right] \\ &= \int_{x_i} \bar{q}_\theta(x_i) \left[\mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] - \log \bar{q}_\theta(x_i)\right] dx_i \\ &\quad - \mathbb{E}_{\bar{q}_\theta(\mathbf{x}_{-i})}\left[\sum_{j\neq i} \log \bar{q}_\theta(\mathbf{x}_{-i})\right]\,. \end{aligned} \tag{13}$$

Here we used the following expression:

$$\mathbb{E}_{\bar{q}_\theta(\mathbf{x}_{-i})}[\cdot] := \int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i})(\cdot) d\mathbf{x}_{-i}\,,$$

which is the expectation over all the variables aside from $q_\theta(x_i)$.
Note that $\int_{\mathbf{x}_{-i}} \bar{q}_\theta(\mathbf{x}_{-i}) d\mathbf{x}_{-i} = 1$. Furthermore, the second term in Eq. (13) is a constant and does not depend on q_i. In order to maximize the ELBO Eq. (13), we apply the

Lagrangian technique giving

$$\mathbb{L} := \mathcal{L} - \sum_{i=1}^{N} \lambda_i \left(\int_{x_i} \bar{q}_\theta(x_i) dx_i - 1 \right) = 0\,.$$

We need to optimize this with respect to hidden variable $q_i = \bar{q}_\theta(x_i)$, so we get the following taking the functional derivative with respect to q_i, and equal it to zero:

$$\begin{aligned} \frac{\delta \mathbb{L}}{\delta q_i} &= \frac{\partial}{\partial q_i}[q_i[\mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] - \log q_i] - \lambda_i q_i] \\ &= \mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] - \log q_i - \lambda_i - 1\,. \end{aligned}$$

Now we can get an expression for q_i, as follows,

$$\begin{aligned} \log q_i &= \mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)] + C \\ q_i &= \frac{e^{\mathbb{E}_{\bar{q}_\theta(\mathbf{x_{-i}})}[\log p(x,y)]}}{Z_i}\,, \end{aligned} \tag{14}$$

where Z_i is a normalising constant that can be derived (as q_i is a known density), and is necessary to make it a density. It is often neglected in finding the q_i's.

Now that we have all the equations in place, we can explain the algorithm that will give us the q_i's. Note, that $\bar{q}_\theta(x_i)$ can be translated as the density of one of the weights in a neural network, such that each q_i is a weights density in the network. The goal was to maximize $\mathcal{L}$ such that we derive a $\bar{q}_\theta(\mathbf{x})$ which minimizes the KL divergence for the true posterior $p(x|y)$. The equations are derived as the mean field approximation, which means that the $\bar{q}_\theta(\mathbf{x})$ is split up in parts $\bar{q}_\theta(x_i)$ that are interdependent when they are minimized. To compute q_i, we need to know the values of the other q_{-i}'s, because of the numerator in Eq. (14) which contains the other factors. An iterative algorithm to get $\bar{q}_\theta(\mathbf{x})$ is given as follows:

1. As with most iterative algorithms we begin with random initializations for the parameters θ of $\bar{q}_\theta(\mathbf{x})$.
2. For each $\bar{q}_\theta(x_i)$, minimize the KL divergence with Eq. (14) by updating $\bar{q}_\theta(x_i)$, keeping the other $\bar{q}_\theta(\mathbf{x}_{-i})$ constant. Using algebra and writing out Eq. (14) one can see the update steps for the different parameters $\mathbf{x}$.
3. Repeat step 2 until a certain convergence criterion is met.

The disadvantage of this algorithm is that the update steps need to be derived each time algebraically for each different problem, which makes it cumbersome and prone to human error. There are recent developments which give a general algorithm for multiple problems, like Mnih and Gregor [2014]. This is a variational update algorithm based on gradients with respect to the different parameters.

3.5 Sequential Monte Carlo algorithm

Sequential Monte Carlo algorithms, also known as particle filters (Gordon et al. [1993]), became increasingly popular as computational power increased. For time series with time steps $k = 1, \dots, n$, we are interested in the posterior distribution to be computed recursively, and in particular one is interested in the marginal posterior density $p(x_n|\mathbf{y}_n)$, also called filtering density, where x_n is the parameter of interest at time n and $\mathbf{y}_n = (y_1, \dots, y_n)$ are the observations. The posterior distribution of the unknown quantities can be computed with the prior, likelihood, and Bayes' theorem Eq. (11). When the filtering distribution is found, one can make point estimates with the help of the posterior mean or mode. This is also known as the Bayesian filtering problem or optimal filtering problem. One of the biggest advantages of SMC is that it has no linearity or Gaussianity assumptions, which makes the method very flexible and makes it particularly relevant for non-stationary time series.

3.5.1 General framework for sequential Monte Carlo

We start by setting up a general framework according to Bernardo et al. [2007], which we use in deriving the sequential Monte Carlo method. The SMC method is in a class of Monte Carlo algorithms that sequentially sample from a target probability density. Denote by $\{\pi_n\}_{n\in\mathbb{T}}$, where $\mathbb{T} = \{1, \dots, K\}$, a sequence of probability measures on the n-dimensional measurable space $(E_n, \mathcal{E}_n)$. E_n is the set of values that $\mathbf{x}_n = \{x_n\}_{n\in\mathbf{T}}$ can take and $\mathcal{E}_n$ is the σ-algebra of this set. $\mathbf{x}_n$ is the sequence of variables that are unknown and need to be found. Each π_n has a density that is known up to a normalising constant:

$$\pi_n(\mathbf{x}_n) = \frac{\gamma_n(\mathbf{x}_n)}{Z_n}, \tag{15}$$

where $\gamma_n : \mathcal{E}_n \to \mathbb{R}^+$ is known pointwise, and $Z_n = \int \gamma_n(\mathbf{x}_n) d\mathbf{x}_n$ the normalising constant that can be unknown. In this framework $k = 1, \dots, n$ is used as a time index. The sequential Monte Carlo algorithm uses samples from the distributions $\{\pi_n\}_{n\in\mathbf{T}}$ and estimate their normalising constants $\{Z_n\}_{n\in\mathbb{T}}$ in a sequential way.

3.5.2 Basic Monte Carlo method

Prior to defining the SMC algorithm, we give an explanation of a basic Monte Carlo method. When approximating a probability density $\pi_n(\mathbf{x}_n)$ for a fixed n, one samples N independent random variables from this density function: $\mathbf{X}_n^i \sim \pi_n(\mathbf{x}_n)$. The Monte Carlo approximation is then given by

$$\widehat{\pi}_n(\mathbf{x}_n) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\mathbf{X}_n^i}(\mathbf{x}_n) \, .$$

Here, $\delta_{x_0}(x)$ is the Dirac delta function centered at x_0. Any marginal $\pi_n(x_k)$ can now be computed using

$$\widehat{\pi}_n(x_k) = \frac{1}{N}\sum_{i=1}^{N} \delta_{\mathbf{X}_k^i}(x_k) \, .$$

The expectation of any test function $\phi_n : \mathcal{E}^n \to \mathbb{R}$ is given by

$$I_n(\phi_n) := \int \phi_n(\mathbf{x}_n)\pi_n(\mathbf{x}_n) d\mathbf{x}_n \, .$$

This can be estimated in the following way:

$$\hat{I}_n(\phi_n) := \int \phi_n(\mathrm{x}_n)\widehat{\pi}_n(\mathrm{x}_n)d\mathrm{x}_n = \frac{1}{N}\sum_{i=1}^{N}\phi_n(\mathrm{X}_n^i)\,.$$

This Monte Carlo estimate is unbiased and its variance is given by

$$VAR[\hat{I}_n(\phi_n)] = \frac{1}{N}\Big(\int \phi_n^2(\mathrm{x}_n)\pi_n(\mathrm{x}_n)d\mathrm{x}_n - I_n^2(\phi_n)\Big)\,.$$

Due to the $1/N$ term, the variance will decrease with $\mathcal{O}(1/N)$. The problem with this method is that $\pi_n(\mathrm{x}_n)$ can be a complex high-dimensional probability density and sampling from this density can be unfeasible. This density is in many real-world problems non-standard, only known up to a proportionality constant and multivariate, as seen in Eq. (15). This makes the above method unsuitable for these kinds of problems.

3.5.3 Importance Sampling

To overcome this problem, one can use a method called Importance Sampling (IS). Importance sampling introduces a so-called importance density $q_n(\mathrm{x}_n)$, also known as proposal density in the literature. This importance density has the following property:

$$\pi_n(\mathrm{x}_n) > 0 \implies q_n(\mathrm{x}_n) > 0\,.$$

In other words, the support of the importance distribution $q_n(x_n)$ must include the support of the target distribution $\pi_n(\mathrm{x}_n)$. Importance sampling can also be used when the density $\pi_n(\mathrm{x}_n)$ is known, to reduce the variance of the estimate. Alternatively, it can be applied to problems in which this density is not available. In this setting, $\pi_n(\mathrm{x}_n)$ is known up to a normalising constant as $\gamma_n(\mathrm{x}_n)$, seen in Eq. (15).
Given an importance density $q_n(\mathrm{x}_n)$, we have the following identity for a test function ϕ_n

$$\begin{aligned} I_n(\phi_n) &= \int \phi_n(\mathrm{x}_n)\pi_n(\mathrm{x}_n)d\mathrm{x}_n \\ &= \int \phi_n(\mathrm{x}_n)\frac{\gamma_n(\mathrm{x}_n)}{\int \gamma_n(\mathrm{x}_n)d\mathrm{x}_n}d\mathrm{x}_n \\ &= \int \phi_n(\mathrm{x}_n)\frac{\bar{w}_n(\mathrm{x}_n)q_n(\mathrm{x}_n)}{\int \bar{w}_n(\mathrm{x}_n)q_n(\mathrm{x}_n)d\mathrm{x}_n}d\mathrm{x}_n\,. \end{aligned}$$

Here, $\bar{w}_n(\mathrm{x}_n)$ is called unnormalised importance weight and is defined as

$$\bar{w}_n(\mathrm{x}_n) = \frac{\gamma_n(\mathrm{x}_n)}{q_n(\mathrm{x}_n)}\,. \tag{16}$$

The target density can now be approximated as

$$\widehat{\pi}_n(\mathrm{x}_n) = \sum_{i=1}^{N} W_n^i\delta_{\mathrm{X}_n^i}(\mathrm{x}_n)\,,$$

where

$$W_n^i = \frac{\bar{w}_n(\mathrm{X}_n^i)}{\sum_{j=1}^{N}\bar{w}_n(\mathrm{X}_n^j)}\,, \tag{17}$$

are called the normalized weights and the normalizing constant Z_n can be approximated with

$$\widehat{Z}_n = \frac{1}{N}\sum_{i=1}^{N} \bar{w}_n(\mathbf{X}_n^i)\,.$$

Here, the independent samples $\mathbf{X}_n^i \sim q_n(\mathbf{x}_n)$, also called particles, are sampled from the known proposal density $q_n(\mathbf{x}_n)$. The Monte Carlo approximation for the expectation of $I_n(\phi_n)$ is calculated by

$$I_n^{IS}(\phi_n) = \frac{1}{N}\sum_{i=1}^{N} \phi(\mathbf{X}_n^i)\frac{\bar{w}_n(\mathbf{X}_n^i)}{\frac{1}{N}\sum_{i=1}^{N}\bar{w}_n(\mathbf{X}_n^i)} = \sum_{i=1}^{N} W_n^i \phi_n(\mathbf{X}_n^i)\,. \tag{18}$$

This means we only need to calculate the normalized importance weights and sample from $q_n(\curvearrowleft_n)$ to get the estimate of the function ϕ_n.

3.5.4 Sequential Importance Sampling

The Sequential Importance Sampling (SIS) method has a fixed computational complexity, which is not the case for IS derived above, that has is an increasing computational cost for an increasing number of observations and thus densities $\pi_n(\mathbf{x}_n)$. SIS uses an importance distribution which is sequential for all the time steps, which means $q_n(\mathbf{x}_n)$ is defined recursively based on $q_{n-1}(\mathbf{x}_{n-1})$. The recursive framework is defined using a Markov kernel, also known as transition kernel, $K_n : E_{n-1} \rightarrow \mathcal{P}(E_n)$. $\mathcal{P}(E_n)$ is the class of probability measures on E_n. In other terms, this gives: $\mathbf{X}_n^{(i)} \sim K_n\big(\mathbf{x}_{n-1}^{(i)}, \cdot\big)$ such that

$$q_n(\mathbf{x}_n) = q_{n-1}(\mathbf{x}_{n-1})K_n(\mathbf{x}_{n-1}, \mathbf{x}_n) = \int q_{n-1}(\mathbf{x}_{n-1})K_n(\mathbf{x}_{n-1}, \mathbf{x}_n)d\mathbf{x}_{n-1}\,.$$

This general representation of SMC with a Markov kernel is more precisely stated as a conditional probability by Doucet and Johansen [2009]. They used the general derivation above, from Bernardo et al. [2007], and give a more comprehensible framework for SIS:

$$\begin{aligned} q_n(\mathbf{x}_n) &= q_{n-1}(\mathbf{x}_{n-1})q_n(x_n|\mathbf{x}_{n-1}) \\ &= q_1(x_1)\prod_{k=2}^{n} q_k(x_k|\mathbf{x}_{k-1})\,. \end{aligned}$$

In other words, every time step $k = 1, \ldots, n$ we sample $X_k^i \sim q_k(x_k|\mathbf{x}_{k-1}^i)$, after initially sampling $X_1^i \sim q_1(x_1)$. Here, $\mathbf{x}_{k-1}^i$ indicates that each sample X_k^i depends on its earlier sample X_{k-1}^i. This way the unnormalized importance weights Eq. (16) can be calculated recursively:

$$\begin{aligned} \bar{w}_n(\mathbf{x}_n) &= \frac{\gamma_n(\mathbf{x}_n)}{q_n(\mathbf{x}_n)} \\ &= \frac{\gamma_{n-1}(\mathbf{x}_{n-1})}{q_{n-1}(\mathbf{x}_{n-1})}\frac{\gamma_n(\mathbf{x}_n)}{\gamma_{n-1}(\mathbf{x}_{n-1})q_n(x_n|\mathbf{x}_{n-1})} \\ &= \bar{w}_{n-1}(\mathbf{x}_{n-1}) \cdot \alpha_n(\mathbf{x}_n)\,. \end{aligned} \tag{19}$$

Here, α_n denotes the incremental importance weights.

$$\alpha_n(\mathbf{x}_n) = \frac{\gamma_n(\mathbf{x}_n)}{\gamma_{n-1}(\mathbf{x}_{n-1})q_n(x_n|\mathbf{x}_{n-1})}\,.$$

Eq. (19) gives us a recursive formula for the weights, starting with time step 1:

$$\bar{w}_n(\mathbf{x}_n) = \bar{w}_1(x_1) \prod_{k=2}^{n} \alpha_k(\mathbf{x}_k)\,,$$

We can now calculate the normalized importance weights as seen in Eq. (17).

In this setting, we have to specify the importance density. This is a difficulty for sequential Monte Carlo methods. The goal is to choose the importance density such that it minimizes the variance of the weights and this would be achieved by choosing

$$q_n(x_n|\mathbf{x}_{n-1}) = \pi_n(x_n|\mathbf{x}_{n-1})\,,$$

because then the variance of $\bar{w}_n(\mathbf{x_n})$ conditional on $\mathbf{x}_{n-1}$ equals zero. Unfortunately, sampling from $\pi_n(x_n|\mathbf{x}_{n-1})$ is not possible as this density is unknown and is the reason for deriving these methods in the first place. This is why we need to approximate this proposal density as close as possible to the true density $\pi_n(x_n|\mathbf{x}_{n-1})$.

3.5.5 Resampling

A disadvantage of IS and thus SIS is degeneracy of the weight vector, as shown in Doucet and Johansen [2009]. This is caused by the variance of the estimates that increases exponentially with n. After a few iterations, the variance of the importance weights is extremely large. This means that one of the particles gets all of the weight, and the rest goes to zero. Resampling is a method that can solve this problem and together with SIS forms the SMC method.

The term resampling comes from the fact that we sample from a distribution that was itself sampled. To be precise, with SIS we have an approximation of $\pi_n(\mathbf{x}_n)$ given by $\widehat{\pi}_n(\mathbf{x}_n)$. This approximation is based on weighted samples from the importance density $q_n(\mathbf{x}_n)$. To improve the approximation, we sample from this SIS approximated sample by selecting particle $\mathbf{X}_n^i$ with probability W_n^i. This is then repeated N times to get a full sample. N_n^i copies of each particle in the sample set are created and are given a new importance weight of $1/N$. For the resample approximation $\bar{\pi}_n$ we now have:

$$\bar{\pi}_n(\mathbf{x}_n) = \sum_{i=1}^{N} \frac{N_n^i}{N} \delta_{\mathbf{x}_n^i}(\mathbf{x}_n)\,.$$

This results in an unbiased approximation of $\widehat{\pi}_n(\mathbf{x}_n)$ since

$$\mathbb{E}[N_n^i | W_n^1, \ldots, W_n^N] = N W_n^i\,.$$

Essentially, the weights and the variance of the weights are reset. The problem of the variance increasing exponentially is now eliminated, while the estimate is still unbiased. Doucet and Johansen [2009] states that when estimating $I_n(\phi_n)$ an estimate with lower variance is obtained with $\hat{\pi}_n$ than that would be obtained with $\bar{\pi}_n$, as resampling removes the particles with low weight and multiplies the particles with higher weights. This comes at the cost of additional variance in the estimate. If the importance weights of the particles do not have high variance, meaning that the samples have not degenerated, a resampling step may not be needed. Therefore there should be a threshold that indicates if the variance is high or not to decide if the particles need to be resampled. In the literature, this additional threshold is assessed with the *Effective Sample Size* (ESS). There are multiple types of formulas for the ESS. One that is most used is

$$ESS = \left(\sum_{i=1}^{N} (W_n^i)^2 \right)^{-1} .$$

The ESS takes values between 1 and N, and resampling takes place when the ESS is below a defined threshold.

The specific method of resampling influences the variance of the number of copies for each particle. There are multiple methods that all try to optimize this variance while preserving the fact that this estimator is unbiased. The most used ones are described below

Systematic resampling Sample $U_1 \sim \mathcal{U}[0, 1/N]$ and with $U_i = U_1 + \frac{i-1}{N}$ for $i = 2, \ldots, N$. The number of copies N_n^i are now chosen as

$$N_n^i = \left| \left\{ U_i : \sum_{k=1}^{i-1} W_n^k \leq U_i \leq \sum_{k=1}^{i} W_n^k \right\} \right| .$$

This method outperforms other methods shown by Hol et al. [2006], which is why this method will be used as resampling method in our SMC algorithm. *Stratified resampling* is another method, done the same way as *systematic resampling* only then U_1 will be sampled each iteration of i.

Residual Resampling Set $\widetilde{N}_n^i = \lfloor N W_n^i \rfloor$, whereafter one samples additional copies $\bar{N}_n^i$ from a multinomial distribution with parameters $\left(M, (\overline{W}_n^1, \ldots, \overline{W}_n^N)\right)$. Here, $\overline{W}_n^i \propto W_n^i - N^{-1}\widetilde{N}_n^i$ and $M = N - \sum_i \widetilde{N}_n^i$. Then the copies count $N_n^i = \widetilde{N}_n^i + \overline{N}_n^i$. This method ensures that each weight (that has a high enough importance weight) has copies in the next iteration.

Multinomial Resampling This method just samples $N_n^1, \ldots, N_n^N$ from the multinomial distribution with parameters $\left(N, (W_n^1, \ldots, W_n^N)\right)$. In other words, every sample will be copied with the probability of its normalized weight. Hol et al. [2006] showed that this was the most computationally complex method and had the largest variance, which is why other methods are preferred.

4 Sequential Monte Carlo for neural networks

The goal of this thesis is to sequentially train an FNN as new data in a time series becomes available. One can see such a neural network as a dynamical model, as it produces a sequence of outputs over time. We assume that the underlying process for creating this sequence evolves in time, and we will consider a discrete time setting here. The reason we want the neural network to be represented as a state space problem is that SMC methods can solve these kinds of problems. As we want to use SMC to train the weights of the neural network, it is necessary to precisely formulate this as a state space problem so we can use all relevant literature on SMC. Understanding the literature can help in expanding certain available algorithms for SMC already in use for these problems. Using the concepts introduced in Chapters 2 and 3, we can eventually formulate the algorithm of Freitas et al. [2000].

4.1 General state space models

In this thesis, SMC is used for the solution of the *optimal filtering* problem. To define this problem, first, we need to understand general state space models. Then, we can see the general structure on which SMC is used and translate this to its application to a neural network.

4.1.1 General state space model

General state space models are also known as hidden Markov models (HMM). The formulation can be used for optimal filtering, control theory, and parameter estimation. In this section, we define a general state space model according to Kantas [2009].

Definition 1 *Let $\{X_n\}_{n\geq 0}$ be a Markov chain with initial density μ defined on $(\mathcal{X}, \mathcal{E}_{\mathcal{X}}, \mathbb{P})$ and $\{Y_n\}_{n\geq 0}$ on $(\mathcal{Y}, \mathcal{E}_{\mathcal{Y}}, \mathbb{P})$ and M and G denote, respectively, a Markov transition kernel from $(\mathcal{X}, \mathcal{E}_{\mathcal{X}})$ to $(\mathcal{X}, \mathcal{E}_{\mathcal{X}})$ and a transition kernel from $(\mathcal{X}, \mathcal{E}_{\mathcal{X}})$ to $(\mathcal{Y}, \mathcal{E}_{\mathcal{Y}})$. The bivariate process $\{(X_n, Y_n)\}_{n\geq 0}$ is called a Hidden Markov Model (HMM) with state x_n and observation y_n, if for any sets $B_X \in \mathcal{E}_{\mathcal{X}}$ and $B_Y \in \mathcal{E}_{\mathcal{Y}}$, we have for any n*

$$\begin{aligned}\mathbb{P}(X_n \in B_X | \mathbf{X}_{n-1} = x_{n-1}, \mathbf{Y}_{n-1} = y_{n-1}) &= \mathbb{P}(X_n \in B_X | X_{n-1} = x_{n-1}) \\ &= \int_{B_X} M(x_{n-1}, dx_n)\,, \\ \mathbb{P}(Y_n \in B_Y | \mathbf{X}_{n-1} = \mathbf{x}_{n-1}, \mathbf{Y}_{n-1} = \mathbf{y}_{n-1}) &= \mathbb{P}(Y_n \in B_Y | X_n = x_n) \\ &= \int_{B_Y} G(x_n, dy_n)\,.\end{aligned}$$

Next, we consider a less general definition, the case of a fully dominated HMM:

Definition 2 *Let there exist a dominating probability measure ρ on $(\mathcal{Y}, \mathcal{E}_{\mathcal{Y}})$ such that for all $x \in \mathcal{X}$, $G(x, \cdot)$ is absolutely continuous with respect to ρ, i.e., $G(x, \cdot) \ll \rho(\cdot)$, with the transition density function being $g(\cdot|x) = \frac{dG(x,\cdot)}{d\rho}$. Also, let there exist a dominating probability measure λ on $(\mathcal{X}, \mathcal{E}_{\mathcal{X}})$ such that for all $x \in \mathcal{X}$, $\mu(\cdot)$ and $M(x, \cdot)$ are absolutely continuous with respect to λ, i.e., $\mu(\cdot) \ll \lambda(\cdot)$ and $M(x, \cdot) \ll \lambda(\cdot)$, with the transition density function being $f(\cdot|x) = \frac{dM(x,\cdot)}{d\lambda}$. The hidden Markov Model $\{(X_n, Y_n)\}_{n\geq 0}$ is then called fully dominated and the joint Markov transition kernel $M(x', x)G(x, y)$ is dominated by the product measure $\lambda \otimes \rho$ and admits the transition density $f(x|x')g(y|x)$.*

An HMM is used in a large range of problems that require the estimation of unobserved, time-varying states of Markov chains using a sequence of noisy observations. This includes non-linear and non-Gaussian time series models, taking $k = 1, \dots, n$:

$$\begin{aligned} x_{k+1} &= \Psi(x_k, d_k)\,, \\ y_k &= \Phi(x_k, v_k)\,. \end{aligned}$$

Here, d_k and v_k are independent sequences of random variables and the functions Ψ and Φ are non-linear, defining the evolution of the state and observations. This model is called a *general state space model.* Most of the literature considers models where $\mathcal{X}$ is finite, which translates to a finite time horizon. However, the way in which the problem is stated also allows for an infinite time horizon, which is in the scope of this thesis. In this case, we call the above-stated set of equations a state space model.

4.1.2 General state space for parameter estimation

Let $\{X_n\}_{n\geq 0}$ and $\{Y_n\}_{n\geq 0}$ be $\mathcal{X}$ and $\mathcal{Y}$ valued stochastic processes defined on a measurable space $(\Omega, \mathcal{F})$ and suppose that $\theta \in \Theta$ is the parameter vector, where Θ is an open subset of $\mathbb{R}^n$. Here, θ denotes a static parameter, e.g., the dynamic noise variance. This parameter can be either known or unknown depending on the problem. In this framework, we assume θ to be known. A state space model uses the unobserved state $\{X_n\}_{n\geq 0}$ as a Markov process of initial density $X_0 \sim \mu$ and Markov transition density $f_\theta(x'|x)$. As the process $\{X_n\}_{n\geq 0}$ is hidden we get indirect information using the observations $\{Y_n\}_{n\geq 0}$. The state space model assumes the observations $\{Y_n\}_{n\geq 0}$ to be conditionally independent given the state $\{X_n\}_{n\geq 0}$, and its behavior is modeled by a conditional marginal density $g_\theta(y|x)$. With $k = 1, \dots, n$, the model is summarised as

$$\begin{aligned} X_k|X_{k-1} = x_{k-1} &\sim f_\theta(\cdot|x_{k-1})\,, \\ Y_k|X_k = x_k &\sim g_\theta(\cdot|x_k)\,. \end{aligned} \tag{20}$$

4.1.3 SMC for Optimal filtering

In Chapters 2 and 3, we formulated the tools on which we can build the optimal filtering problem. The goal is to find the hidden states of the state space model. Having Eq. (20) as our state space model we define,

$$\{X_n\}_{n\geq 0} = \{\pi_n\}_{n\geq 0}\,, \tag{21}$$

which gives us a sequence of unknown densities. This can be the same sequence that is described in Eq. (15). This distribution is a complete solution of the state inference problem as it summarises all that is known about the hidden states given the observations. Translating this to a Bayesian setting, optimal filtering denotes the hidden state sequence Eq. (21) as posterior densities $\{p(\mathbf{x}_n|\mathbf{y}_n)\}_{n\geq 0}$; which gives $\pi_n(\mathbf{x}_n) = p(\mathbf{x}_n|\mathbf{y}_n)$. In many applications, we are interested in estimating these posteriors recursively in time, especially the marginal of this density $p(x_n|\mathbf{y}_n)$. This marginal is called the filtering density, hence the name *optimal filtering.* Compared to SMC methods for optimal filtering, general MCMC methods tend to be computationally too expensive, as they re-assess the whole data set every iteration. This especially holds for large n. Referring to Eq. (15) as the

density, we have $\gamma_n(\mathbf{x}_n) = p(\mathbf{x}_n, \mathbf{y}_n)$ for optimal filtering or written in terms of the state space functions:

$$\gamma_n(\mathbf{x}_n) = \mu(x_1)g(y_1|x_1)\left\{\prod_{k=2}^{n} f(x_k|x_{k-1})g(y_k|x_k)\right\} . \tag{22}$$

Where μ, f and g are defined in Section 4.1.1. We cannot sample from the state posterior $p(\mathbf{x}_n|\mathbf{y}_n)$ directly, so a suitable importance density needs to be chosen. The key for choosing this density is to use as much information as possible while keeping computational costs low. The importance density must be as close to the real distribution as possible in order to minimize the variance of the importance weights (Doucet and Johansen [2009], Doucet et al. [2000]). Using $\gamma_n(\mathbf{x}_n)$ in Eq. (22), and Eq. (15) we get:

$$\begin{aligned}\pi_n(x_n|\mathbf{x}_{n-1}) &= p(x_n|y_n, x_{n-1}) \\ &= \frac{g(y_n|x_n)f(x_n|x_{n-1})}{p(y_n|x_{n-1})} .\end{aligned}$$

This means we should use the optimal importance density of the form

$$q_n(x_n|\mathbf{x}_{n-1}) = p(x_n|y_n, x_{n-1}) ,$$

first seen in Zaritskii et al. [1975]. For the incremental importance weight this gives us

$$\begin{aligned}\alpha_n(\mathbf{x}_n) = \alpha_n(x_{n-1}, x_n) &= \frac{g(y_n|x_n)f(x_n|x_{n-1})}{p(x_n|y_n, x_{n-1})} \\ &= p(y_n|x_{n-1}) .\end{aligned}$$

The importance density shown here has a few drawbacks as it requires samples from $p(x_n|y_n, x_{n-1})$ and evaluation of $p(y_n|x_{n-1})$, which has no analytic form in the non-linear and non-Gaussian case. Furthermore, if we use an recursive importance density for the posterior $p(\mathbf{x}_n|\mathbf{y}_n)$ of the form

$$q_n(\mathbf{x}_n|\mathbf{y}_n) = q_1(x_1|y_1)\prod_{k=2}^{n} q_i(x_k|\mathbf{x}_{k-1}, \mathbf{y}_k) ,$$

Doucet et al. [2000] states that the unconditional variance of the importance weights increases over time. This is why often the importance density in state space models is chosen in a more simple form, i.e.,

$$q_n(x_n|\mathbf{x}_{n-1}) = f(x_n|x_{n-1}) . \tag{23}$$

This gives an incremental weight

$$\alpha_n(\mathbf{x}_n) = g(y_n|x_n) . \tag{24}$$

The equations Eq. (23) and Eq. (24) will be used in the SMC algorithm for neural networks as state space model and is also used in the bootstrap filter by Gordon et al. [1993]. Doucet et al. [2000] states that the importance density Eq. (23) is often inefficient in simulations as the state space is explored without any knowledge of the observations and is especially sensitive for outliers. Nevertheless, this importance density is often used in literature because it is easy to implement.

4.2 Neural Network in state space representation

The work of de Freitas et al. [1998] was the first paper where a state space representation of a neural network was proposed to capture its evolution in time. The evolution was presented in the following way, having time steps $k = 1, \dots, n$:

$$\begin{aligned} \mathbf{w}_{k+1} &= \mathbf{w}_k + \mathbf{d}_k \, , \\ \mathbf{y}_k &= \hat{\mathbf{y}}_k(\mathbf{w}_k, \mathbf{x}_k) + \mathbf{v}_k \, . \end{aligned} \tag{25}$$

The bold notation is used, as these variables may be in vector form. For convenience, we assume that $\mathbf{w}_{k+1}$ also denotes the biases of the neural network. Furthermore, we have $(\mathbf{y}_k \in \mathbb{R}^o)$ as the output measurements, $(\mathbf{x}_k \in \mathbb{R}^d)$ as the input measurements and $(\mathbf{w}_k \in \mathbb{R}^m)$ for the weights of the neural network. The non-linear function, denoted with $\hat{\mathbf{y}}(\cdot)$, is the mapping of a neural network itself. We assume some measurement noise $\mathbf{v}_k$, which is one of the free parameters in the SMC algorithm for neural networks. The weights propagate through time being only dependent on the previous state and some noise component $\mathbf{d}_k$, which is used to define the transition function. These noise components may be different from a normal Gaussian distribution.

Using this formulation of the problem the evolution of the weights can be described by a hidden Markov process, with an initial probability $\mu(\mathbf{w}_1)$ and a transition probability $f(x_k|x_{k-1}) = p(\mathbf{w}_k|\mathbf{w}_{k-1})$. Here we also assume that the observations are conditionally independent given the states. This formulates the neural network in the general state space framework from Section 4.1.1, which means we are now equipped to use SMC for analyzing the (hidden) states which are the weights of the NN. The SMC method that is used is called *optimal filtering* and is described in Chapter 4.1.3.

4.3 Sequential Monte Carlo method for Neural networks

We can formulate this problem as a Bayesian neural network as we are interested in the posterior distribution of the weights of the neural network $p(W_k|Y_k)$, where $W_k = \{\mathbf{w}_1, \dots, \mathbf{w}_k\}$ and $Y_k = \{\mathbf{y}_1, \dots, \mathbf{y}_k\}$. Here, k denotes the time steps. We are interested in the marginal density $p(\mathbf{w}_k|Y_k)$, the filtering density. In the literature, a prediction and update step is used to determine the marginal density. For the prediction step we have the following equation:

$$p(\mathbf{w}_k|Y_{k-1}) = \int p(\mathbf{w}_k|\mathbf{w}_{k-1}) p(\mathbf{w}_{k-1}|Y_{k-1}) d\,\mathbf{w}_{k-1} \, . \tag{26}$$

and the transition density, also presented in the general state space model chapter, is described by $f(x|x_{k-1}) = p(\mathbf{w}_k|\mathbf{w}_{k-1})$. This transition density is modeled by Eq. (25), using the noise $\mathbf{d}_k$. The solution for the filtering density is given by combining the prediction step Eq. (26) and Bayes' rule to get the update step:

$$p(\mathbf{w}_k|Y_k) = \frac{p(\mathbf{y}_k|\mathbf{w}_k) p(\mathbf{w}_k|Y_{k-1})}{p(\mathbf{y}_k|Y_{k-1})} \tag{27}$$

The normalizing constant in Eq. (27) can lead to computational challenges regarding high-dimensional integration especially when the number of parameters are high, which is the case for neural networks. As Chapter 3 highlights, this can be solved with the SMC method, which approximates the posterior distribution of interest.

4.3.1 Calculating the posterior distribution of the weights

We take the transition density as the proposal density as seen in Eq. (23),

$$f(x_k|x_{k-1}) = p(\mathbf{w}_k|\mathbf{w}_{k-1})\,.$$

This means that the incremental weights are proportional to the likelihood as in Eq. (24),

$$p(\mathbf{y}_k|\mathbf{w}_k) \propto \exp -\frac{1}{2}\left((\mathbf{y}_k - \hat{\mathbf{y}}_k(\mathbf{w}_k, \mathbf{x}_k))^T R^{-1} (\mathbf{y}_k - \hat{\mathbf{y}}_k(\mathbf{w}_k, \mathbf{x}_k))\right). \tag{28}$$

The noise d_k is assumed to be Gaussian $d_k \sim \mathcal{N}(0, Q)$. In other words, each time step the weights at time step $k+1$ are the weights at time step k with added noise with variance Q. Furthermore, the observation noise v_k is also assumed to be Gaussian $v_k \sim \mathcal{N}(0, R)$, where R is the variance of the observations. To make the one step ahead predictions, we use Eq. (18). In other words, the predictions are a weighted average of the predictions by the neural network for the different weight samples.

This gives us the tools to formulate the pseudo code seen in Algorithm 1 and the code used for the numerical tests can be found in Appendix A.

4.3.2 The SMC algorithm for a convolutional neural network

The WaveNet structure is already able to efficiently model time series with the SGD method. One of the advantages of the SMC algorithm for an FNN is that FNN's are not able to efficiently train sequential time series and the SMC algorithm makes this possible. This advantage is lost for WaveNet because it already trains sequentially. Nevertheless, the SMC algorithm may improve time series modeling with WaveNet, as it may be able to capture changing patterns by adjusting the weights in the network. As explained in Chapter 2, a convolutional neural network is similar to a fully-connected neural network, so that the equations above can also be applied on convolutional neural networks. The only difference is that the architecture of the network depends on the number of previous time steps that are taken into account.The filter size κ then accounts for the number of total layers. For example, a CNN that depends on the last five time steps has three hidden layers for $\kappa = 2$, but only one hidden layer when $\kappa = 3$.
Considering the initialization of the weights, the number of weights for a convolutional neural network is the same for each layer and is equal to the filter size. This means that the number of weights plus the bias is

$$\#\text{parameters} = \kappa \times (L-1) + (\sum_l n_l + 1).$$

The dimensions of the weights matrices are all the same, where with multiple channels (other time series), the filters become multi-dimensional. The noise on the weights is the same as for fully-connected neural networks. The script for the SMC algorithm for CNN is given in Appendix C.

Algorithm 1: SMC for neural networks

```
Data: Dataset $\mathcal{D} = (\mathbf{x}, \mathbf{y})$
Parameter: number of layers $l = 1, \ldots, L$
           noise of weights: $Q$
           noise of data: $R$
           Threshold: $T$
           number of samples: $N$
Output: Neural network one step ahead predictions
for $k = 0$ do
    for $i = 1, \ldots, N$ do
        Draw weights $w_0^i$ for each layer from prior: $\mu_l(w_0)$
        Evaluate importance ratio's
            $$\bar{w}_0^i = p(\mathbf{y}_0 | \mathbf{w}_0^i)$$
        Normalize importance ratio's
            $$W_0^i = \frac{\bar{w}_0^i}{\sum_{j=1}^N \bar{w}_0^j}$$
for $k > 0$ do
    for $i = 1, \ldots, N$ do
        $\hat{\mathbf{w}}_{k+1}^i = \mathbf{w}_k^i + \mathbf{d}_k$        ▷ where $d_k$ is a sample from $p(d_k) \sim \mathcal{N}(0, Q)$
        Evaluate importance ratio's:
            $$\bar{w}_{k+1}^i = \bar{w}_k^i p(\mathbf{y}_{k+1} | \mathbf{w}_{k+1}^i)$$
        Normalize importance ratio's:
            $$W_{k+1}^i = \frac{\bar{w}_{k+1}^i}{\sum_{j=1}^N \bar{w}_{k+1}^j}$$
        if $N_{eff} \geq$ *Treshold* then
            $\mathbf{w}_{k+1}^i = \hat{\mathbf{w}}_{k+1}^i$
        else
            Resample new index $j$ from discrete set $\{\hat{\mathbf{w}}_{k+1}^i, W_{k+1}^i\}$
            $\mathbf{w}_{k+1}^i = \hat{\mathbf{w}}_{k+1}^j$
            $\bar{w}_{k+1}^i = 1/N$
```

4.4 Improvements for the SMC algorithm on neural networks

In this section, we describe several ways to improve the basic SMC algorithm and the neural network. In the literature, multiple options are proposed.

4.4.1 ReLU as activation function

The current state-of-the art neural networks use Rectfied Linear Units (ReLUs) as activation function. The ReLU activation function is as follows:

$$\sigma(a) = max\left\{0, a\right\} .$$

Krizhevsky et al. [2012] showed that ReLUs converge faster than a hyperbolic tangent or sigmoid activation function. At the same time, the calculations to be done in backpropagation are faster with a ReLU activation function. Also, the SMC algorithm may lose important information of earlier points if the weights did not converge fast enough, resulting in a resampling as the sample space became too sparse. This happens more with sigmoid activation functions because they can become saturated and learn even slower. A saturated neuron means that the derivative, $\sigma'(z)$, of the activation function, is near zero. For the sigmoid activation function, this is the case when $|z| >> 0$. The ReLU activation functions make neurons less likely to be saturated and therefore we expect that the SMC algorithm on ReLU networks will be able to converge faster while keeping the relevant information.

4.4.2 Adding gradient descent into the algorithm

Another possible improvement is to add a gradient descent step into the SMC algorithm Freitas et al. [2000]. Adding a gradient descent step makes the algorithm more efficient since the prior can include the information of the current data point y_k. The backpropagation algorithm explained in Chapter 2.1.3, on which the gradient descent is based, uses the cost function, which in our case is the likelihood Eq. (28). This likelihood gives the gradients based on the data point y_k. The more information is provided to the prior, the more accurate the weights sampled from this prior will become. The gradient step is used after the prediction step of the SMC algorithm, where the predicted weights are propagated from the previous weights with the noise $d_k \sim N(0, Q)$. This gradient step is a normal backpropagation step, but instead of using batch learning, it only trains on-line, which means the weights get updated each time step. When adding the gradient descent step in the SMC algorithm, a learning parameter η has to be defined. This parameter influences the step that is taken in the direction the gradients give. A small learning parameter means that a small step is taken into the direction of the gradient. This means slow convergence. On the other hand, if a large learning parameter is taken, the neural network may become unstable as the step taken into the direction of the gradient is too big, which means no convergence takes place. The gradients have to be calculated for each individual weight sample $\hat{\mathbf{w}}_k^i$ and bias $\hat{\mathbf{b}}_k^i$. The calculation of the gradient can be done using the backpropagation from Section 2.1.3. The algorithm including the gradient step is defined in Algorithm 2. The code for the SMC with gradient step and ReLU activation function can be found in Appendix B.

Algorithm 2: SMC for neural networks with gradient step

```
Data: Dataset $\mathcal{D} = (\mathbf{x}, \mathbf{y})$
Parameter: number of layers $l = 1, \ldots, L$
           noise of weights: $Q$
           noise of data: $R$
           Threshold: $T$
           number of samples: $N$
Output: Neural network one step ahead predictions
for $k = 0$ do
    for $i = 1, \ldots, N$ do
        Draw weights $w_0^i$ for each layer from prior: $\mu_l(w_0)$
        Evaluate importance ratio's
            $$\bar{w}_0^i = p(\mathbf{y}_0 | \mathbf{w}_0^i)$$
        Normalize importance ratio's
            $$W_0^i = \frac{\bar{w}_0^i}{\sum_{j=1}^N \bar{w}_0^j}$$
for $k > 0$ do
    for $i = 1, \ldots, N$ do
        $\hat{\mathbf{w}}_{k+1}^i = \mathbf{w}_k^i + \mathbf{d}_k$        ▷ $d_k$ is a sample from $p(d_k) \sim \mathcal{N}(0, Q)$
        Update each $\hat{\mathbf{w}}_{k+1}^i$ with corresponding gradient step
        Evaluate importance ratio's:
            $$\bar{w}_{k+1}^i = \bar{w}_k^i p(\mathbf{y}_{k+1} | \mathbf{w}_{k+1}^i)$$
        Normalize importance ratio's:
            $$W_{k+1}^i = \frac{\bar{w}_{k+1}^i}{\sum_{j=1}^N \bar{w}_{k+1}^j}$$
        if $N_{eff} \geq$ Treshold then
            $\mathbf{w}_{k+1}^i = \hat{\mathbf{w}}_{k+1}^i$
        else
            Resample new index $j$ from discrete set $\{\hat{\mathbf{w}}_{k+1}^i, W_{k+1}^i\}$
            $\mathbf{w}_{k+1}^i = \hat{\mathbf{w}}_{k+1}^j$
            $\bar{w}_{k+1}^i = 1/N$
```

4.4.3 Initializing weights with He initialization

The initialization of the weights can be very important for the convergence of the neural network. The initialization can improve performance significantly and has been shown already in early papers like LeCun et al. [1998]. Glorot and Bengio [2010] found the "Xavier" method, which was made and derived specifically for symmetric activations in the neurons, like sigmoid and hyperbolic tangent activations. When using ReLU as the activation function in deep neural networks, He et al. [2015] found the most efficient way of initializing the values of the weights.

The He initialization is defined as follows,

$$\mathbf{w} \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l}}),$$

where n_l here is the number of neurons in the layer a specific weight is in. At the same time, the biases are initialized as zeros. In the numerical results, we study whether He initialization can improve the SMC algorithm.

5 Numerical results

In this section, the results of testing the SMC algorithm on different time series are presented. The time series used are a composite function similar to the one used in Freitas et al. [2000], a sine with noise, the daily stock returns of the S&P 500, and global mean temperatures. The main goal of this thesis is to gain insight into when the SMC method on an FNN outperforms the standard way of training networks with the SGD algorithm. By testing the performance of SMC and SGD on a wide variety of time series we hope to gain insight into the added value of SMC. Furthermore, we thoroughly study the influence of the hyperparameters of SMC on the network performance. We then study the effects of the improvements on SMC as proposed in Section 4.4, and in particular apply the SMC algorithm on a novel network structure, the CNN.
The SMC algorithm has several hyperparameters which are summarized in Table 1:

Parameter	Definition
N	# of samples per time step
Q	The noise of the transition function
R	The width of the likelihood
T	The threshold for the ESS
$\bar{\mu}_l$	Variance on the initialization of weights in layer l
n_l	# of neurons in layer l

Table 1: Parameters for SMC algorithm

5.1 Defining the time series

Throughout the numerical results, we use four different time series, which we here describe in detail. We have $k = 1, \dots, K$ to indicate the time steps.

5.1.1 Time series 1: Composite function

We define the composite function as:

$$y_k(x_1(k), x_2(k)) = 4\sin(x_1(k) - 2) + 2x_2^2(k) + 5\cos(0.02k) + 5 + v\,. \tag{29}$$

Here, $v \sim \mathcal{N}(0, 0.1)$ is Gaussian noise with a zero mean and variance of 0.1. The input $(x_1(k), x_2(k))$ is simulated from a Gaussian distribution $\mathcal{N}(0, 1)$. We generate $K = 400$ time steps of $y_k(x_1(k), x_2(k))$. As input for the neural network, we use $x_1(k)$ and $x_2(k)$ to model $y_k(x_1(k), x_2(k))$, which gives us $n_1 = 2$ input neurons.

5.1.2 Time series 2: The sine function with noise

We define the sine function with noise as:

$$y_k = \sin 0.02k + v\,,$$

Here, $v \sim \mathcal{N}(0, 1)$ is Gaussian noise with a zero mean and variance of 1. We generate $K = 400$ time steps of y_k. As input for the neural network we use the five previous time steps $\mathbf{y}_{k-5} = (y_{k-5}, y_{k-4}, y_{k-3}, y_{k-2}, y_{k-1})$ to model y_k, which gives us $n_1 = 5$ input neurons. This also means we forecast only $K = 395$ time steps, starting from y_6. This time series gives insight into how the algorithm performs with noise. The optimal RMSE would be equal to the noise variance so that a larger RMSE means that the neural network is overfitting.

5.1.3 Time series 3: The daily returns of S&P 500

Financial data is known to be non-stationary and with a low signal-to-noise ratio. As mentioned before, SMC may be well-capable of handling these properties, which is why this is an interesting real-world time series to test SMC on. We use the daily returns $\tilde{R}_k$ of the daily adjusted closing price $\tilde{P}_k$. The date we start the time series is 01-01-2016 and take $K = 400$ daily adjusted closing prices, where the daily returns are calculated with adjusted closing price $\tilde{P}_k$ as follows:

$$\tilde{R}_k = \frac{\tilde{P}_k - \tilde{P}_{k-1}}{\tilde{P}_{k-1}}.$$

As input for the neural network we use the four previous returns
$\mathbf{y}_{k-4} = (\tilde{R}_{k-4}, \tilde{R}_{k-3}, \tilde{R}_{k-2}, \tilde{R}_{k-1})$ to model $y_k = \tilde{R}_k$, which gives $n_1 = 4$. This means we forecast $K = 396$ time steps and start with forecasting $y_5 = \tilde{R}_5$.

5.1.4 Time series 4: The global mean temperature data

For the last time series, we use the yearly global mean temperature data $\tilde{T}_k$ from the GISS NASA website. The data start from the year 1880 till 2018, which means we have $K = 138$. Here we use again the five previous global temperature means $\mathbf{y}_{k-5} = (\tilde{T}_{k-5}, \tilde{T}_{k-4}, \tilde{T}_{k-3}, \tilde{T}_{k-2}, \tilde{T}_{k-1})$ to model $y_k = \tilde{T}_k$, which gives $n_1 = 5$. This means we start forecasting $y_6 = \tilde{T}_6$, and forecast a total of $K = 133$ time steps. This time series has a clear trend, similar to the artificial noisy sine function, but the presence of noisy outliers can make the forecasting task more challenging. This time series is used to compare the performance of the SMC algorithm on a CNN.

5.2 The SMC algorithm on FNN

5.2.1 SMC compared to SGD

In this section, we compare the SMC algorithm, as seen in Algorithm 1, with the SGD method on FNN's using different network architectures. The different architectures are denoted as structure in the Tables below, and give the number of neurons each layer, starting with the input layer. When we add more neurons and layers, we express this as the neural network getting more complex. To quantify the performance, we use two measures: the RMSE and MASE. We compute the RMSE as follows:

$$RMSE = \sqrt{\frac{\sum_{k=1}^{K} (\hat{y}_k - y_k)^2}{K}},$$

and for the MASE we have:

$$MASE = \frac{1}{K} \sum_{k=1}^{T} \left(\frac{|\hat{y}_k - y_k|}{\frac{1}{K-1} \sum_{k=2}^{K} |y_{k-1} - y_k|} \right), \tag{30}$$

where the denominator is the mean absolute error for the naïve prediction. The naïve prediction takes the last seen time step as one step ahead prediction. Therefore this measure gives a relative value, where values smaller than one indicate that the model is outperforming a naïve prediction. The parameters we choose each time for the different tests are the ones that in initial tests gave the best performance. To measure the consistency of the algorithms, we run the SMC algorithm 50 times and the SGD for 10 runs and calculate the standard deviation between the runs. Furthermore, SMC is a sequential algorithm, and SGD is not. To obtain a fair comparison, SGD is trained with the first 350 data points, whereafter the last 50 one step ahead predictions are predicted with the set of weights SGD obtained after the training phase of 10.000 epochs. SMC gives one step ahead predictions after every time step, but the algorithm has a small convergence period after initializing the weights finding the correct weight samples. The last 50 one step ahead predictions of the SMC algorithm are used to compare to the last 50 one step ahead predictions of the SGD. We study the performance on different FNN architectures, in order to fully understand the SMC and SGD performance.

The composite function The parameters values we used are as follows: $Q = 0.1$, $R = 1$, $N = 200$, $T = N/3$ and $\bar{\mu}_l = [5, 2, 1, 1]$ for respectively $n_2 = [5, 10, 20]$ and when adding $n_3 = 20$.

In Table 2 we compare the SMC algorithm to the SGD on FNN's with different architectures. SGD appears to be performing worse than SMC when the structure gets more complex. In other words, this indicates that SGD starts to overfit with more trainable parameters, a problem that is well-known to arise in SGD. Furthermore, for all the architectures SMC has a similar or better RMSE than SGD. At the same time, the variance of the SMC algorithm decreases as a more complex architecture is used. These results indicate that the SMC method is better at forecasting non-stationary time series than SGD given these architectures.

Structure	SMC RMSE	SGD RMSE	SMC MASE	SGD MASE
$[2, 5, 1]$	2.36 ± 0.37	2.30 ± 0.055	0.49 ± 0.076	0.45 ± 0.012
$[2, 10, 1]$	1.90 ± 0.34	2.17 ± 0.015	0.38 ± 0.067	0.43 ± 0.003
$[2, 20, 1]$	1.80 ± 0.31	2.20 ± 0.019	0.35 ± 0.055	0.43 ± 0.004
$[2, 20, 20, 1]$	1.93 ± 0.29	2.55 ± 0.057	0.37 ± 0.056	0.50 ± 0.014

Table 2: Test values for the composite function trained with SMC and SGD *We make the architectures more complex each time resulting in better results until we add the second hidden layer. The SMC has better results for almost all the structures for both RMSE and MASE.*

The sine function with noise The parameters used for SMC are $R = 1$, $N = 200$, $T = N/3$, and $Q \in \{0.1, 0.05, 0.025, 0.01\}$ changes respectively for each more complex architecture, as initial tests showed that more complex architectures need a lower Q-value to converge. Furthermore, $\bar{\mu}_2 = 4$ for the hidden layer and $\bar{\mu}_L = 0.5$ for the output layer. When adding the second hidden layer $\bar{\mu}_3 = 4$ is used for this layer again.

In Table 3, we see that SGD is overfitting when the network size increases, which we expected. The overfitting is indicated by the RMSE which is larger than the noise $v =$

$\mathcal{N}(0,1)$. SMC is better at avoiding overfitting on the data compared with SGD for the last two architectures. In this data set previous points are the input for the neural networks, opposite to the first data set where the input was the generated $x_1(k)$ and $x_2(k)$ for simulating the data set. Using previous time steps as input makes it prone to a naïve prediction, where the last seen data point becomes too important. This is why the MASE is essential here as this gives the relative error to the naïve prediction, see Eq. (30). A MASE smaller than one indicates that the neural network finds a better pattern in the previous points than solely giving the last point a significant weight.

Structure	SMC RMSE	SGD RMSE	SMC MASE	SGD MASE
$[5,5,1]$	1.08 ± 0.077	1.043 ± 0.031	0.80 ± 0.064	0.78 ± 0.036
$[5,10,1]$	1.09 ± 0.067	1.09 ± 0.038	0.81 ± 0.059	0.83 ± 0.034
$[5,20,1]$	1.09 ± 0.076	1.29 ± 0.091	0.81 ± 0.055	0.95 ± 0.073
$[5,20,20,1]$	1.18 ± 0.150	1.69 ± 0.143	0.86 ± 0.112	1.22 ± 0.134

Table 3: Test values for the sine with noise with trained with SMC and SGD *The architecture gets more complex each time resulting in overfitting for SGD, while SMC is keeping a similar RMSE and MASE in the first three architectures*

The daily returns of the S&P 500 The parameters are as follows: $N = 200$, $R = 0.02$, $T = N/2$. The $Q \in \{0.01, 0.005, 0.001, 0.001\}$ changes for each structure respectively. For initialization, we have: $\bar{\mu}_2 = 1$ and $\bar{\mu}_L = 0.25$ is used for respectively the hidden layer and the output layer. When the second hidden layer is added we use $\bar{\mu}_3 = 1$ for this layer.

In Table 4 it is shown that the more complex the network gets, the better the RMSE of SMC is, while the opposite is true for SGD. The bad performance of SGD is related to the algorithm overfitting on the train data and it not being able to work with the highly non-stationary and noisy stock data. For the less complex architecture, it seems that SMC cannot correctly capture the patterns in the data needed to give a good one step ahead prediction. Nevertheless, the SMC algorithm, on the largest architecture, gives the best performance in terms of RMSE and MASE. This verifies our initial claim that the SMC algorithm is a suitable technique for neural network training to model with non-stationary time series as it avoids overfitting and can capture non-stationary patterns.

Structure	SMC RMSE	SGD RMSE	SMC MASE	SGD MASE
$[4,5,1]$	$0.00670 \pm 2.6\text{e}{-4}$	$0.00598 \pm 4.3\text{e}{-4}$	0.91 ± 0.05	0.78 ± 0.08
$[4,10,1]$	$0.00649 \pm 4.4\text{e}{-4}$	$0.00606 \pm 5.3\text{e}{-4}$	0.89 ± 0.07	0.79 ± 0.10
$[4,20,1]$	$0.00596 \pm 4.8\text{e}{-4}$	$0.00640 \pm 8.8\text{e}{-4}$	0.82 ± 0.08	0.86 ± 0.17
$[4,20,20,1]$	$0.00571 \pm 2.9\text{e}{-4}$	$0.00648 \pm 8.7\text{e}{-4}$	0.76 ± 0.06	0.88 ± 0.17

Table 4: Test values for the daily returns of the S&P 500 trained with SMC and SGD starting from 01-01-2016 and using 400 adjusted closing prices, resulting in a modeling of 396 daily returns, as we use 4 previous returns as input.

For the global mean temperature we observed the same good performance of SMC compared to SGD, as for the other three time series, which is why we omit these results.

5.2.2 Sensitivity of the SMC algorithm to the parameters

In this section, we study how the SMC algorithm behaves for different parameter settings. This gives an overview of how to set the parameter values when using the SMC algorithm in practice. In particular, we present here the results on the composite function, as for the other time series we saw similar sensitivity of the SMC algorithm for the parameters, and we, therefore, omit these tables.

We generate $K = 200$ time steps with Eq. (29) and use the RMSE over all the 200 points. Furthermore, we used the architecture of the network that had the smallest RMSE in Table 2, which consists of one hidden layer of 20 neurons.

N	25	50	100	200	400	800	1600
mean RMSE	3.38	3.22	3.05	3.06	3.00	2.98	2.97
SD RMSE	0.33	0.27	0.35	0.28	0.26	0.33	0.28

Table 5: Results for the sensitivity of hyperparameter N, showing the mean RMSE values for 50 simulations changing the parameter N for the SMC algorithm. *Here we changed the parameter N, keeping the other parameters fixed*

In Table 5, one can see that after more than 100 generated samples each time step, the RMSE does not improve significantly. The authors of Crisan and Doucet [2000] showed that as the number of samples at initialization goes to infinity, one can obtain convergence in time with the SMC algorithm. This means we expected a decrease in the RMSE as we increase the sample size. The rate of decrease slows down the more samples we generate, which is expected for Monte Carlo methods. One possible explanation could be that the architecture of the neural network used here is not complex enough to capture the structure of the time series, meaning that this is the best error the neural network of this complexity can give.

T	N	N/2	N/3	N/5	N/10	N/20
mean RMSE	3.21	3.1	3.06	3.00	2.97	2.95
SD RMSE	0.27	0.33	0.28	0.26	0.25	0.25

Table 6: Results for the sensitivity of hyperparameter T, showing the mean RMSE values for 50 simulations changing the parameter T of the SMC algorithm. *Here we changed the parameter T, keeping the other parameters fixed. When $T = N$ every iteration a resampling takes place. In Chapter 3, it was explained that with resampling the algorithm makes a reset. With lower T the amount of resampling decreases.*

In Table 6 it shows that when decreasing the amount of resampling the RMSE improves. The composite function benefits from not resetting the samples every iteration. This means that the properties of the historical data points are essential to take into account when making one step ahead predictions using SMC.

In Table 7, one can see that when the parameter R becomes too large, the algorithm is not able to make good predictions. This is most likely because SMC is not discriminating enough between the samples, as we have a wide likelihood because the measurement noise v is assumed to be high. The large R also gives samples that are not capturing the

R	1	1.5	2	4	8	16
mean RMSE	3.06	3.03	3.08	3.04	3.29	3.83
SD RMSE	0.31	0.26	0.28	0.27	0.19	0.21

Table 7: Results for the sensitivity of hyperparameter R, showing the mean RMSE values for 50 simulations changing the parameter R of the SMC algorithm. *Here we changed the parameter R, keeping the other parameters fixed. The parameter R influences the width of the likelihood. A large R, means a larger likelihood for more points and thus less discriminaton between the different weight samples.*

underlying properties of the data set a more substantial likelihood. This larger likelihood gives large normalized importance weights for these poor samples, and the prediction of the neural network depends on these normalized weights. As poor weight samples are given a higher probability, the predictions are too distorted by these samples.

Q	0.05	0.1	0.25	0.5	1	2
mean RMSE	3.69	3.46	3.05	3.06	3.85	5.46
SD RMSE	0.15	0.17	0.24	0.28	0.65	1.16

Table 8: Results for the sensitivity of hyperparameter Q, showing the mean RMSE values for 50 simulations changing the parameter Q of the SMC algorithm. *Here we changed the parameter Q, keeping the other parameters fixed. A larger Q means that a larger weights space is being considered. A small Q may lead to insufficient variety in the weight samples and not being able to find the optimal weights. On the other hand, a too large Q leads to distortion of possible optimal weight samples.*

In Table 8 we can see how SMC behaves for different values of Q. For $Q = 0.05$, the RMSE is quite large, and this may be because the properties of the time series change faster than the transition function, based on $d_k \sim N(0, Q)$, can capture this change. In other words, the weight samples need to change fast, but a small Q only explores a small amount of the weight space each iteration. The larger Q-values give even worse results. This is because when the variance of the transition function becomes too big, the weights are changing too fast and are not able to converge to a specific value, causing the algorithm to become unstable.

5.2.3 Posterior distributions

In this section, we study the posterior distributions that the SMC algorithm generates over both the weights and the outcome. We again consider the SMC algorithm on the composite function. We expect to see some multi-modality in the posterior distribution of the weights as neural networks are known to have this multi-modality.

Posterior distribution of the weights The SMC method provides a posterior distribution over the weights. This posterior can be used to obtain insight into the weight distribution. In this section, we visualize these distributions. We show the posterior distribution of the weights for the composite function tested in Table 2. A 3D plot is made to visualize the movement of the distribution better.
In Figure 6 the distributions of the weight, w^L_{13}, between the third neuron in the hidden layer and the output layer neuron is shown. The posterior distribution oscillates over time, similar to the cosine function that is in the composite function. In some time steps, the distribution is more peaked than others. This may indicate more confidence in the value of the weight. Some multi-modality can be seen as well, especially at time step 380. Both these modes may have a similar likelihood when interacting with the other weights of the neural network, giving both modes a similar probability in the posterior distribution of the weight.

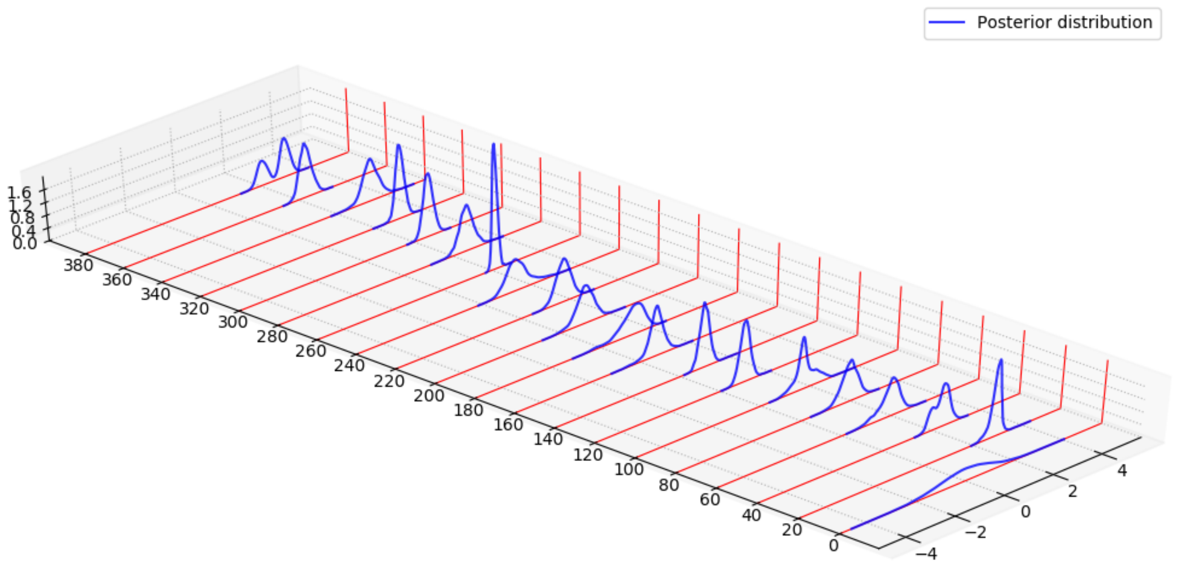


Figure 6: Posterior distribution of weight w^L_{13}, with $N = 5000$, for the composite function in the first test

Distribution on the outcome Here, we show the distribution on the outcome. The plotted distributions give a view of where the different outcome values lie. As the algorithm uses the normalized importance weights to give a weighted average of these outcomes, the predictions are not equal to the posterior mean. Figure 7 shows the predicted and true values with the distribution of the outcome. We observe that initially, the predicted values are not well-aligned with the true values, however, as the algorithm continues, the predicted distribution on the outcome contains the true values more often. This means that this distribution could be an indication of the uncertainty the neural network has.

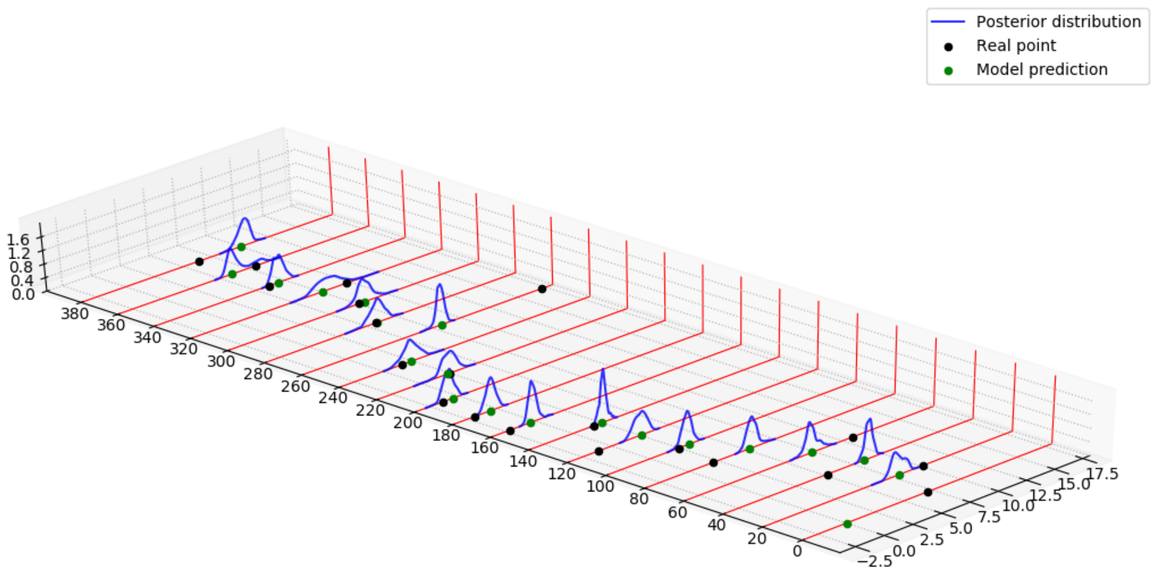


Figure 7: Posterior distribution on the outcome, with $N = 500$, for the composite function

5.2.4 FNN with ReLU activation function

In Section 4.4, we introduced possible improvements for the SMC algorithm on neural networks, and in this section, we test if using ReLU activation functions improve the performance of the SMC algorithm on FNN's.

The composite function In this section, we replicate the tests done for Table 2 to see how the ReLU activation compares to the sigmoid as activation function when training with the SMC algorithm. Here the RMSE over the whole data set is included. The values of the parameters used are almost the same as the tests for Table 2, except that we set $\bar{\mu}_l = 0.5$, for all layers and use $Q = 0.05$ for the one hidden layer network and $Q = 0.025$ for the network with two hidden layers.

Structure	SMC RMSE50	SMC RMSE all	SMC MASE50
$[2, 5, 1]$	1.62 ± 0.42	$2.579 \pm 0.0.426$	0.32 ± 0.0876
$[2, 10, 1]$	1.33 ± 0.34	2.151 ± 0.344	0.252 ± 0.0526
$[2, 20, 1]$	1.207 ± 0.189	2.271 ± 0.208	0.233 ± 0.0376
$[2, 20, 20, 1]$	1.467 ± 0.338	2.038 ± 0.24	0.276 ± 0.057

Table 9: Test values for the composite function with ReLU activation functions. *Here instead of using sigmoid activation functions in the neural network, ReLU activation function are used. RMSE50 denotes the RMSE over the last 50 points as done in the previous tests. RMSE all denotes the RMSE over the whole data set of 400 time steps. The MASE50 is the MASE over the last 50 points.*

Table 9 shows that using a ReLU activation function for the neural network significantly improves the RMSE and MASE compared to the results in 2. Again the most complex network is overfitting the data more, resulting in a larger RMSE than the less complex networks with $n_2 = 10$ and $n_3 = 20$. The improvement can be explained by the faster convergence of the ReLU activation function, as explained in Chapter 4.4.1.

The daily returns of S&P500 Here, we introduce the hit rate as an additional performance measure, specifically designed for assessing the forecasts of one step ahead returns. The hit rate, $\tilde{H}$, indicates how well the neural network predicts the direction of the stock. In other words, the hit rate shows how well the neural network captures the underlying patterns. The hit rate is calculated as follows:

$$\tilde{H} = \frac{1}{K} \sum_{k=1}^{K} H_k \,,$$

where H_k is defined as:

$$H_k = \begin{cases} 1, & \text{if sign}\left((y_{k+1} - y_k)(\hat{y}_{k+1} - \hat{y}_k)\right) > 0 \\ 0, & \text{otherwise} \end{cases}$$

The parameters used for testing are the same as in the tests for Table 4. The difference is we use $\bar{\mu}_l = 0.3$ for all layers and $Q = 0.0005$.

Structure	RMSE50	MASE50	HIT
$[4, 5, 1]$	0.00529 ± 0.000303	0.674 ± 0.0549	0.479 ± 0.0698
$[4, 10, 1]$	0.00543 ± 0.000391	0.707 ± 0.0725	0.4664 ± 0.0652
$[4, 20, 1]$	0.00571 ± 0.000598	0.763 ± 0.0955	0.4668 ± 0.0565
$[4, 20, 20, 1]$	0.00552 ± 0.000401	0.723 ± 0.0705	0.468 ± 0.0684

Table 10: Test values for the S&P 500 daily returns with ReLU activation functions. *Here instead of using sigmoid activation functions in the neural network, ReLU activation functions are used. RMSE50 the RMSE over the last 50 points as done in the previous tests. MASE50 denotes the MASE over the last 50 points. HIT denotes the hit rate over the last 50 points of the data set. The 4 previous returns are used for the input neurons*

In Table 10 the ReLU again gives a significant improvement for the test measures compared to the sigmoid activation functions in Table 4, showing a higher RMSE for all architectures. We observe however that unlike with the sigmoid, the RMSE actually gets worse as the architecture becomes more complex. Also, the hit rate becomes lower as the architecture gets more complex, meaning that the neural network is not finding the patterns in the data, or cannot capture specific changes in the patterns fast enough. Nevertheless, the performance of the ReLU activation is better than that of the sigmoid.

For the sine function with noise, we got near-optimal performance for the basic SMC algorithm, and the same performance was observed when training an FNN with ReLU activation functions for the sine with noise. Also, for the global mean temperatures these observations where made. This is why we omit these results here.

5.3 Gradient descent step

In this section, SMC is tested with the gradient descent step added to the general SMC algorithm, as seen in Algorithm 2. The gradient step is tested for both sigmoid and ReLU activations for the neural network to see if the gradient descent step is more efficient for one of these activations.

Composite function For the test in Table 11 the learning parameter $\eta = 0.01$ is chosen. Furthermore, the same parameters are used as in the tests for Table 9 to make a fair comparison.

Structure	SMC RMSE50	SMC RMSE all	SMC MASE50
$[2, 5, 1]$	1.267 ± 0.385	2.829 ± 0.221	0.235 ± 0.0689
$[2, 10, 1]$	1.046 ± 0.275	2.435 ± 0.215	0.185 ± 0.0553
$[2, 20, 1]$	0.998 ± 0.184	2.141 ± 0.199	0.175 ± 0.0337
$[2, 20, 20, 1]$	1.040 ± 0.234	1.902 ± 0.266	0.193 ± 0.0386

Table 11: Test values for the composite function with gradient step and ReLU activation functions. *Here, a gradient step is added to the SMC algorithm and ReLU activation functions are used. RMSE50 denotes the RMSE over the last 50 points as done in the previous tests and MASE50 the MASE over the last 50 points. RMSE all denotes the RMSE over the whole data set of 400 time steps.*

Table 11 shows that the RMSE for the whole data set steadily decreases as the network structure becomes more complex, indicating faster convergence. Adding the gradient descent step gives significantly better results than only changing the activation function to ReLU. It seems that for the architecture with two hidden layers the gradient step provides significantly faster convergence, as it is now comparable to the less complex architectures for the RMSE50.

For the sigmoid activation functions, we take the same parameters as for the initial test of Table 2, and use $\eta = 0.01$ as learning parameter. This way, we can see the influence of the gradient step for the sigmoid activation.

Structure	SMC RMSE50	SMC RMSE all	SMC MASE
$[2, 5, 1]$	2.229 ± 0.374	3.075 ± 0.418	0.464 ± 0.0826
$[2, 10, 1]$	1.847 ± 0.318	2.735 ± 0.264	0.376 ± 0.0656
$[2, 20, 1]$	1.679 ± 0.344	2.644 ± 0.184	0.328 ± 0.0664
$[2, 20, 20, 1]$	1.865 ± 0.300	2.642 ± 0.167	0.360 ± 0.0632

Table 12: Test values for the composite function with gradient step and sigmoid activation functions. *RMSE50 denotes the RMSE over the last 50 points as done in the previous tests and MASE50 the MASE over the last 50 points. RMSE all denotes the RMSE over all 400 one step ahead predictions.*

Table 12 shows a decrease in RMSE for all architectures compared to Table 2. Compared to the ReLU activation function with the gradient step in Table 11, the gradient step improves the RMSE significantly less when using the sigmoid function. The reason may be that the gradient step benefits more from the faster convergence of ReLU, and has issues with saturated neurons that arise with sigmoid activation functions, as explained in Section 4.4.1. It concludes that changing the activation function from sigmoid to ReLU significantly improves SMC training for neural networks with and without a gradient step.

Structure	RMSE50	HIT	MASE50
$[4, 5, 1]$	0.00525 ± 0.00026	0.4664 ± 0.0664	0.6477 ± 0.0435
$[4, 10, 1]$	0.005829 ± 0.002013	0.4648 ± 0.0488	0.74 ± 0.029
$[4, 20, 1]$	0.005955 ± 0.001310	0.4712 ± 0.0600	0.770 ± 0.218
$[4, 20, 20, 1]$	0.005569 ± 0.000446	0.4852 ± 0.0647	0.710 ± 0.075

Table 13: Test values for the daily returns of the S&P500 with gradient step and ReLU activation functions. *RMSE50 denotes the RMSE over the last 50 points as done in the previous tests. MASE50 denotes the MASE for the last 50 points. HIT is the hit ratio over all the time steps.*

Financial data The parameters used are the same as for in Table 10, only changing $Q = 0.00025$ and taking $\eta = 0.00001$. It is observed, in Table 13, that adding the gradient makes the RMSE worse for the last three architectures, compared to Table 10. The hit ratio seems to be increasing as the network becomes more complex, which would make it better for practical uses than without the gradient step. However, it is interesting to see that the RMSE is higher than without a gradient step, seen in Table 10, meaning the gradient step here potentially leads to more overfitting of the data. The gradient step evaluates the observation y_k, and we use previous time steps as input for the neural network. This means for the prediction $\hat{y}_{k+1}$ this is the last seen time step. It could be that with taking previous time steps as input, the gradient step directs the neural network too much to this observation y_k, for example giving the last seen time step a too high weight, indicated by the slightly higher MASE50 than seen in Table 10, which badly influences the prediction for time step $k + 1$ as it tends toward a naïve prediction.

For the sine with noise, we saw similar behavior as for the S&P 500 time series. This indicates that the gradient step is, in some cases, not beneficial when the inputs are previous time steps.

Structure	SMC RMSE50	SMC RMSE all	SMC MASE
$[5, 5, 1]$	0.135 ± 0.0387	0.314 ± 0.00905	1.0386 ± 0.115
$[5, 10, 1]$	0.135 ± 0.0369	0.314 ± 0.00871	1.040 ± 0.107
$[5, 20, 1]$	0.135 ± 0.0364	0.316 ± 0.0105	1.0285 ± 0.106
$[5, 20, 20, 1]$	0.136 ± 0.0374	0.314 ± 0.00891	1.0465 ± 0.123

Table 14: Test values for the temperature data set with gradient step and ReLU activation functions. *RMSE50 denotes the RMSE over the last 50 points as done in the previous tests. RMSE all denotes the RMSE over the whole data set of 134 temperature time steps.*

Global mean temperature data Here we used parameters: $Q \in \{0.1, 0.05, 0.025, 0.01\}$, $R = 0.3$, $N = 200$, $\eta = 0.001$, $T = N/2$ and $\bar{\mu}_l = 0.3$ for all layers. Surprisingly, Table 14 shows that all architecture have similar performance on the temperature data set. There are 133 time steps, which is not that much, but the algorithm can still model this non-stationary time series. Although, it is outperformed by a naïve prediction, indicated by the MASE50 value above one. These values will be used to compare the results in the next section on CNN.

5.3.1 Using automatically scaled initialization

Now we test if He initialization, discussed in Section 4.4.3, improves the performance even further for the SMC algorithm with gradient descent. This allows us to reduce the number of parameters that need to be chosen manually. We test the composite function with $K = 600$ generated time steps and use the SMC algorithm with gradient step and ReLU activation functions on the architecture with two hidden layers. Furthermore, we use same parameters as for the tests seen in Table 11, only then we used $Q = 0.01$. We use the moving average of the RMSE and standard deviation of the distribution on the outcome, with a window of 100 time steps.

Figure 8 shows that with a He initialization we get a significantly better initialization. The RMSE over the first 100 time steps is approximately 7.5 for He initialization, against 32.5 for the manual initialization in three runs. This is an improvement in two ways. As now, we also have less hyperparameters to choose, which is beneficial for the robustness of the SMC algorithm in general.

The problem with the He initialization is that for certain time series it results in a too high initial weight variance and then the He initialization performs worse. For the composite function, the He initialization helped the initial convergence significantly, but for other data sets, this was not the case. The initialization method should depend on the structure of the time series. In any case, we can conclude that proper initialization is essential for a good initial set of weight samples in the SMC algorithm.

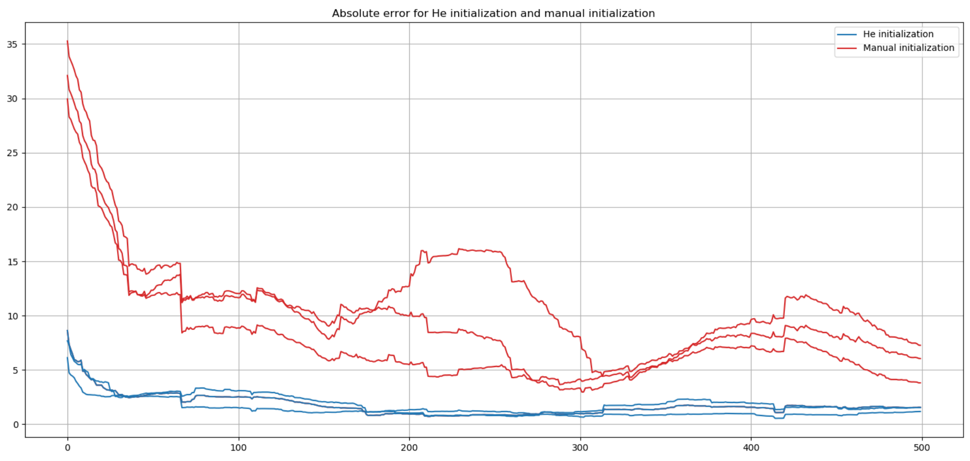


Figure 8: Three simulations for He initialization and manual initialization. For the manual initialization we have $\bar{\mu}_l = 0.3$ for all layers. The architecture used is $[2, 20, 20, 1]$

5.4 SMC on convolutional neural networks

In this section we present the results of the SMC algorithm on the convolutional network structure. The SMC algorithm has not been tested before on this kind of network, and our results are the first to do so. Table 15 shows that the RMSE and MASE are approximately equal, or better, compared to the SMC algorithm for a fully-connected neural network. The number of previous time steps is equal to the experiments with SMC applied the FNN, to keep the information that the networks train on the same. Furthermore, we used the same parameters as for the tests done in Section 5.3 for the specific time series, where for the sine function with noise we chose $\eta = 0.0001$. Only the Q was different, which we changed to $Q \in \{0.02, 0.002, 0.1\}$ for respectively time series 2,3 and 4.

Data set	RMSE50	MASE	HIT
Temperature	0.118 ± 0.00418	0.955 ± 0.028	-
Sine with noise	1.049 ± 0.0675	0.735 ± 0.0457	-
Stock returns	0.005654 ± 0.00118	0.738 ± 0.148	0.4908 ± 0.072

Table 15: Test values for three different time series using the SMC algorithm on a CNN. *RMSE50 denotes the RMSE over the last 50 points as done in the previous tests. MASE here is on the last 50 points as well. HIT here is the hit ratio and is only applicable for the stock returns.*

For the temperature data and the sine with noise, the CNN performs significantly better. For the stock returns, it performs approximately equal. The structure of WaveNet can capture the dependence of previous time steps to the current time step better than an FNN. The specific structure of WaveNet is beneficial when modeling the time series in an autoregressive way with previous time steps as input. These results are promising, especially as the number of previous points was matched with the tests for FNN. It could be that more previous time steps as input improve the results even more.

5.5 Uncertainty with the SMC algorithm

We discussed the importance of having a measure of uncertainty in a model and that one of the benefits of the SMC algorithm was that the output was given by samples from this posterior. In this section, we look at the distribution on the outcome and the spread of this distribution as measured by the standard deviation. The distribution of the outcome is what eventually determines the prediction and it captures the uncertainty of the network. Theoretically, a small spread in the distribution should mean that the algorithm is more confident about the value of the weight. A high spread means the opposite and means that the algorithm is not sure about the value of the weight.

For this section we used the composite function to generate $K = 12.000$ time steps, which we model with the SMC algorithm seen in Algorithm 2. The network architecture used consists of two layers of 20 hidden neurons, and the same parameters used in Section 5.3 for the composite function. Only here we used $Q = 0.01$ and $Q = 0.005$, to see the difference in long term behavior. Furthermore, we use moving averages for the absolute errors and standard deviations of the distribution on the outcome, for each time step. For the moving averages, a window of 300 time steps is used.

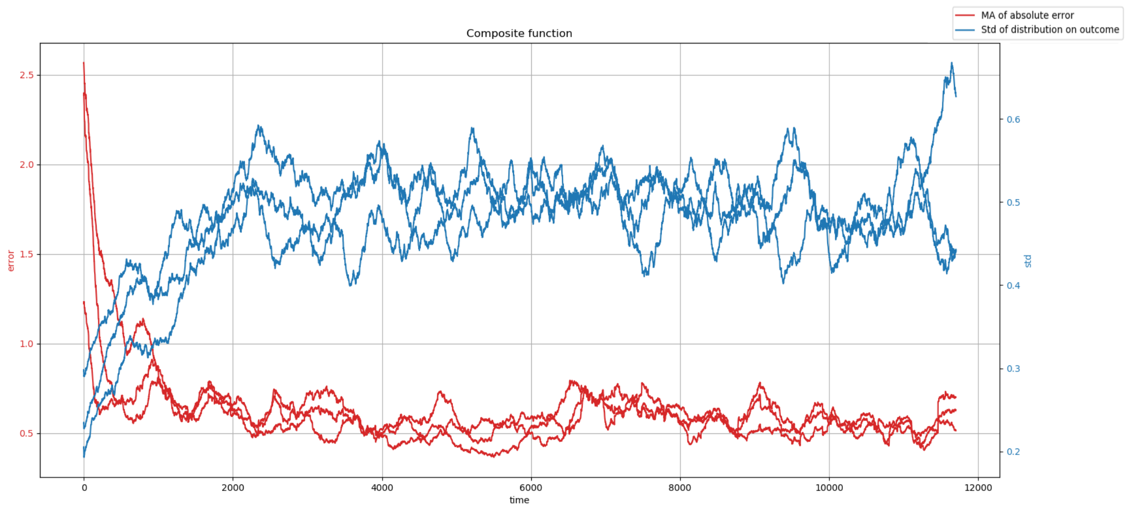


Figure 9: Three simulations of the moving average of the error and standard deviation of the outcome distribution for the composite function data set for parameter $Q = 0.01$

We see that after a certain number of iterations, the error converges to slightly above 0.5 in Figure 9, while this is below 0.5 for the smaller Q-value in Figure 10. For a high Q-value, the algorithm showed less consistency between runs. However, a higher Q-value lets the algorithm converge faster, but may hit a plateau in terms of decreasing error. Whereas a lower Q converges less fast but can get a smaller error over time and is more stable. At the same time, we see that with a low Q the algorithm moves away slower from a wrong initialization. What the value of Q should be, is different for each time series, and should be carefully selected.

For the spread of the distribution on the outcome, we saw similar behavior. The standard deviation converged to a specific region, and this happened faster for the larger Q-value. We calculated the correlation between the spread on the outcome and the absolute error, and the correlation was positive and around a value of 0.35 depending on

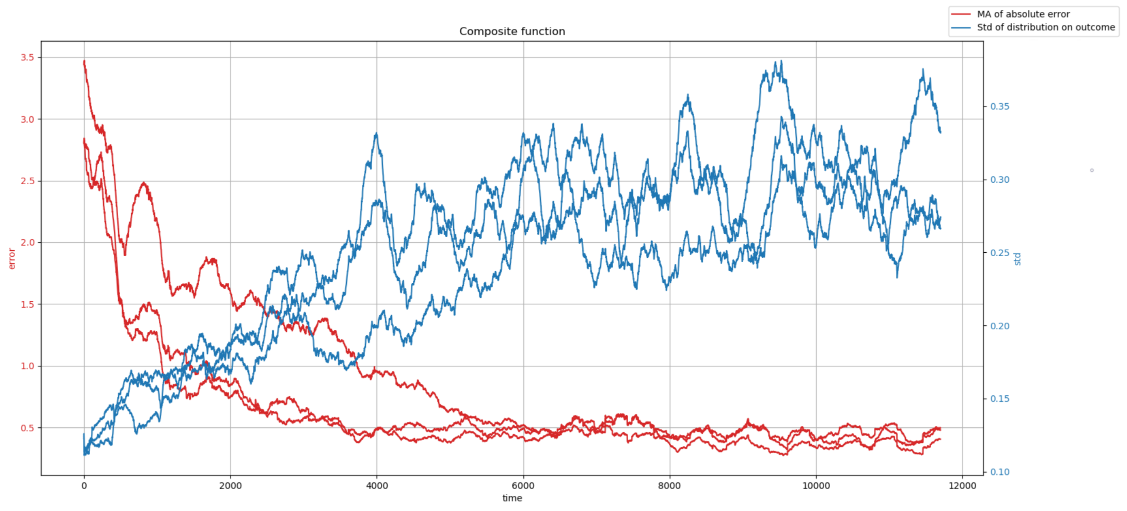


Figure 10: Three simulations of the moving average of the error and standard deviation of the outcome distribution for the composite function data set for parameter $Q = 0.005$

the run. This means that higher spreads on the outcome tend to result in higher errors, as we expect to see.

Unfortunately, when we did the same tests for the sine function with noise, and the daily returns, the theoretical behavior that a lower error has higher confidence did not hold for these time series, indicated by a correlation around zero for these time series. Understanding this is something that we leave for future investigation.

5.6 Capturing change points with the SMC algorithm

In this section, we simulate a change in the noise to see how the algorithm reacts to this. We use the sine function with noise to see if the SMC algorithm can capture this change, and can keep predicting well. We use the standard deviation of the distribution on the outcome, to indicate if the uncertainty becomes smaller when the noise is lower. We talked about that SGD is not able to capture sudden changes in the data, and that it is useful to detect when these changes present themselves. We generate 4000 time steps, where at time step 2000, the noise changes from a high noise, $\mathcal{N}(0, 1.5)$, to low noise, $\mathcal{N}(0, 0.3)$. We use a moving average with a window of 200 time steps, for the absolute error and standard deviation.

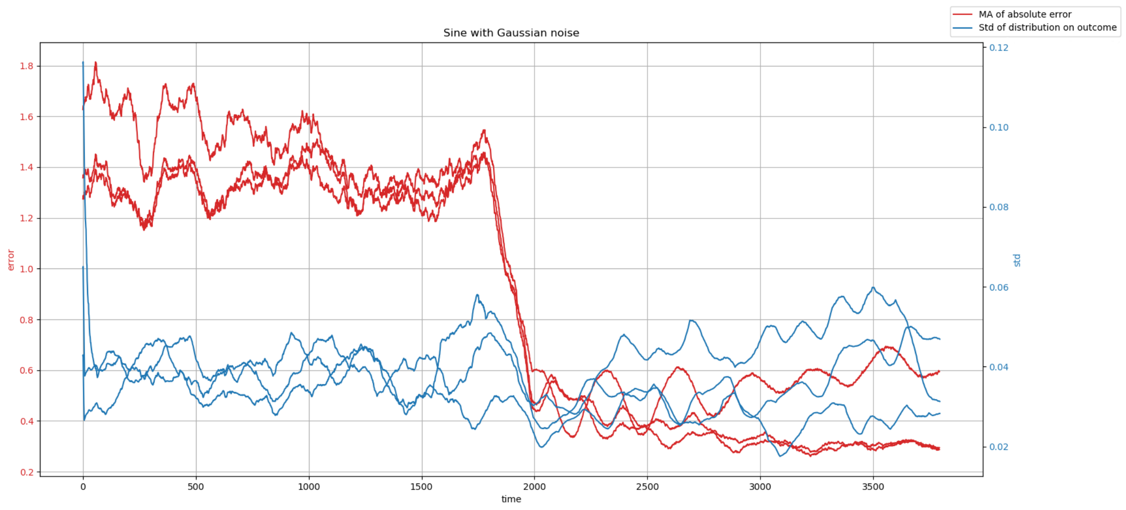


Figure 11: Three simulations of the moving average of the error and standard deviation of the outcome distribution for the sine function with noise, changing the noise at $k = 2000$

In Figure 11 we see that initially the spread drops when the noise is changed. Indicating more confidence in the prediction, which is expected as the noise becomes lower. The initial drop in the spread can be used as change point detection as it decreases fast. However, after this drop, only one of the runs stays around these lower values. The other runs increase over time, and one of the runs seems to become unstable. The SMC algorithm is not able to consistently show a decrease in uncertainty when the noise is smaller. On the other hand, the SMC algorithm does have a smaller error for two of the runs when the noise becomes smaller. This proves that the SMC can adapt to changing patterns of the data providing consistent performance for most of the runs, but is unable to consistently capture this lower uncertainty in the spread of the distribution.

The same behavior was observed for the other time series, giving an error that decreased or increased relative to high or low noise, but not able to consistently capture this change in terms of uncertainty in the distribution of the outcome. Understanding how SMC can be used for this change-point detection is something that is also left for future work.

6 Conclusion and discussion

The main goal of this thesis was to understand whether using the SMC algorithm to train an FNN can outperform the standard SGD training method when applied to a non-stationary time series forecasting task. In this thesis, we showed by extensive numerical tests on a diverse set of time series that the SMC can indeed provide a significant benefit over the SGD method, showing the SMC algorithm for neural networks can be a powerful method for forecasting non-stationary time series. Up to the best of our knowledge, this is one of the first works extensively testing the SMC method for neural networks. We gained further insight into the effects of the hyperparameters of SMC on the performance, showing that in particular the parameters governing the trade-off the algorithm makes between exploring the state-space more and providing faster convergence, can impact the performance and convergence of the model.

Furthermore, we proposed several improvements on the standard SMC algorithm by incorporating novelties from the neural network research, such as using a more robust initialization strategy, using a different activation function known as the ReLU or including a gradient step in the SMC algorithm. These adaptations were able to improve the results for the different time series even further. The SMC algorithm with the ReLU activation significantly outperformed the SGD method. After incorporating the gradient step in the SMC algorithm, we saw even further improvement, although this was not the case for all time series. When using an autoregressive type of input, taking previous time steps, the SMC algorithm seemed to benefit less of the gradient step. Moreover, we used the SMC algorithm on a novel architecture applied to time series forecasting known as the WaveNet, a particular kind of convolutional neural network. The SMC algorithm applied to the WaveNet structure performed approximately equal or better for the time series, and an added benefit of this structure is that it has significantly fewer weights compared to the FNN. WaveNet, in combination with the SMC algorithm, was shown to be better for autoregressive modeling of time series, compared to the SMC algorithm on FNN's.

The SMC algorithm performance depends on the hyperparameters chosen, and choosing these hyperparameters optimally can be time-consuming. A possible improvement could be to incorporate some form of automatic parameter tuning into the SMC algorithm instead of manually finding the right parameters. Using a grid search to find optimal parameters would be beneficial as the values of the parameters differ for each data set. Alternatively, as noted in Bergstra and Bengio [2012], a random grid search could be even better. The authors of Snoek et al. [2012] provide a Bayesian optimization scheme for the tuning of these parameters, and it may be possible to incorporate this optimization in the SMC algorithm itself as this technique is sequential as well.

In terms of further improvements on the SMC algorithm, we saw that a larger Q-value is useful for fast convergence, but can hit a plateau in terms of error. The authors of Sbarufatti et al. [2017] used an adaptive Q that started with higher values and gradually decreased in value. This way, it has initial fast convergence and can reduce the error further due to the lower Q-value improving convergence, as seen in the results of Section 5.5. This method might again be dependent on the underlying time series. It could then be better to adaptively change Q according to the error and the variance in the data. The authors of Freitas et al. [2000] used an extended Kalman filter (EKF) to make this possible. The EKF method to calculate the gradients is computationally more expensive, so depending on the practical use it would not be able to give a full and smooth posterior distribution on the weights, and therefore on the outcome, because it is not able to handle many samples as computation times are slow for this method. If the computational efficiency could be improved, then this could be a good improvement for the SMC algorithm, while keeping a full and smooth posterior.

As the SMC algorithm is used broadly in practice for navigation, robotics and solving state-space problems, a lot of literature is written about specific problems and how to improve these. One of these is the auxiliary particle filter Carpenter et al. [1999]. According to de Freitas [2003], this auxiliary particle filter performs better when the likelihood is high in the tail of the prior. On the contrary, the basic SMC behaves better when the prior coincides with the likelihood. Given this fact, it may be useful to incorporate both in an algorithm and use them adaptively. Additionally, because an EKF gradient step is less effective at finding a full posterior, some smoothing algorithms could be of use. Smoothing in the resampling step, or between the few points that are propagated may give better results and better posteriors. The authors of Doucet and Johansen [2009] describes different techniques for smoothing in the SMC algorithm.

To conclude, in this work we showed that the SMC algorithm could be a powerful alternative to SGD for forecasting time series. One of the main disadvantages, and reason for SMC not being used widely, is the high computational time. However, with the recent increase in computing power, the computational disadvantage of Monte Carlo methods become less of an issue. Furthermore, the need for a posterior distribution on the outcome to have a measure of uncertainty is increasingly urgent as well. Again emphasizing the need for Bayesian neural networks, of which the SMC algorithm is an example, making it a very useful and relevant method. Nevertheless, the SMC algorithm still has a lot of room for further study and improvement, some of which we proposed in this section.

Appendices

A Python code for SMC algorithm

Listing 1: SMC algorithm

```
"""
Algorithm for performing SMC training on a neural network for time series.
The output is a one-step ahead prediction for data that is fed sequentially
     to the algorithm.
You can import this python file in another file, and then fill in the
    parameters:
sizes(structure of neural network), N(# monte carlo weight samples),
    sigma_initial_w(initial sigma's of weights),
sigma_initial_b(initial sigma's of biases). As seen in the __init__
    statement this creates an instance of
a neural net with these parameters. Then with the SMC function, one can
    obtain one step
ahead predictions on data. NOTE: The initial sigma's for the weights are
    given in an array that should have the same
length as the length of the sizes vector.

"""
import numpy as np
import time
import sys
from tqdm import tqdm
from multiprocessing import Pool
from joblib import Parallel, delayed
import multiprocessing

from numpy.random import random

class SMC_algorithm(object):

    def __init__(self,sizes, N, layer_sigmas_w, layer_sigmas_b, posterior=
        False, epoch=False, weights=None, biases=None):
         self.sizes=sizes
         self.output_dim=sizes[-1]
         self.num_layers = len(sizes)-1
         self.L0=np.zeros((N,1))
         self.L=np.zeros((N,1))
         self.sigma_initial_w=layer_sigmas_w
         self.sigma_initial_b=layer_sigmas_b
         self.forward_output=np.zeros((N,sizes[-1]))
         self.N=N
         self.posterior=posterior
         self.epoch=epoch
         if self.epoch:
              self.BiasStore=biases
              self.WeightsStore=weights
         else:
              self.BiasStore=[]
              self.WeightsStore=[]
    def store_weights(self,w):
         self.WeightsStore.append(w)

    def store_bias(self,b):
         self.BiasStore.append(b)

    def initial_weights(self,N):
         """
         Creating the initial weight samples.
         """
         if self.num_layers == len(self.sigma_initial_b) and not self.epoch:
              for i in range(0,N):
                  biases = [l*np.random.randn(y, 1) for y, l in zip(self.
                      sizes[1:], self.sigma_initial_b)]
                  weights = [l*np.random.randn(y, x)
                                       for x, y, l in zip(self.sizes[:-1],
                                           self.sizes[1:],self.sigma_initial_w)
                                           ]

                  self.store_weights(weights)
                  self.store_bias(biases)
              # print self.WeightsStore
              self.WeightsStore=np.array(self.WeightsStore)
              self.BiasStore=np.array(self.BiasStore)

         elif self.epoch:
              self.WeightsStore=self.WeightsStore
              self.BiasStore=self.BiasStore

         else:
              print 'ERROR: initial weights sigma vector size is not equal to
                  number of layers'

         return self.WeightsStore, self.BiasStore

    def feedforward(self,w,b,u):
         """
         Feedforward algorithm for N different samples, for arbitrary size
              and layers of neural networks.
         """
         u=u[np.newaxis]

         for i in range(0,self.N):
              l=0
              while l < self.num_layers-1:
                  if l==0:
                      a = np.dot(u,w[i][l].T)+b[i][l].T
                      a=sigmoid(a)
                      l+=1

                  if l < self.num_layers-1:
                      a = np.dot(a,w[i][l].T)+b[i][l].T
                      a = sigmoid(a)
                      l+=1

              self.forward_output[i]=np.dot(a, w[i][l].T)+b[i][l].T

         return self.forward_output
```

```
def update_weights(self,w,b,Q):
    """
    Updating the weights, meaning giving them a noise with sigma Q.
    """

    for i in range(0,self.N):

        for l in range(0,self.num_layers):

            self.BiasStore[i][l] = np.array(b[i][l] + np.random.normal
                (0,Q,b[i][l].shape))
            self.WeightsStore[i][l] = np.array(w[i][l] + np.random.
                normal(0,Q,w[i][l].shape))

    return self.WeightsStore, self.BiasStore


def SMC(self,data,real,Q, R,threshold,indweight=None):
    """
    This function gives a one step ahead prediction. The first element
        is 0 as one cannot give a prediction for
    the first data point. So the error should start with the second
        element of the prediction array compared with
    the second data point of the real vector.
    """
    K=len(real)
    print self.sizes
    resample_index=np.zeros((K,1))
    if self.posterior:
        posterior=np.zeros((self.N,1))
        posterior_timestep=np.zeros((K,self.N,1))
    outcome_posterior=np.zeros((K+1,self.N,self.output_dim))
    prediction_next_datapoint=np.zeros((K+1,1))
    iter=0
    for k in tqdm(range(0,K)):

        if k==0:
            w,b = self.initial_weights(self.N) # initialize Weights
            output_model=np.array(self.feedforward(w,b,data[k]))    #
                get output from every weight sample i
            prediction=np.array(self.feedforward(w,b,data[k+1]))# get
                output from every weight sample, with input of next data
                 point.
            L0=self.Likelihood(real[k],output_model,R) # get likelihood
                 of every weight sample i
            q=L0
            q_norm=normalize(q)
            outcome_posterior[k+1]=prediction
            prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)
            if self.posterior:
                for p in range(0,self.N):
                    weightlayer=np.array(w[p][-1])
                    weightlayer=np.array(weightlayer[0])
                    posterior[p]=weightlayer[indweight]
                posterior_timestep[k]=posterior
        else:
            w,b=self.update_weights(self.WeightsStore,self.BiasStore,Q)
```

```
                output_model=np.array(self.feedforward(w, b,data[k])) # get
                    output from every weight sample i
                if self.posterior:
                    for p in range(0,self.N):
                        weightlayer=np.array(w[p][-1])
                        weightlayer=np.array(weightlayer[-1])
                        posterior[p]=weightlayer[indweight]
                    posterior_timestep[k]=posterior
                L=self.Likelihood(real[k],output_model,R) # get likelihood
                    of every weight sample i
                q=q*L
                q_norm=normalize(q)
                prediction=np.array(self.feedforward(w,b,data[k+1]))
                prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)
                outcome_posterior[k+1]=prediction
                Neff=self.Neff(q_norm)
                if Neff<threshold:
                    resample_index[k]=1
                    iter += 1
                    w,b,q=self.resampling(w,b,q_norm,q)
                    q_norm=q

        print 'Number␣of␣resamples', iter+1

        return prediction_next_datapoint, posterior_timestep,
            outcome_posterior, w, b,resample_index

    def resampling(self,w,b,q_norm,q):
        """
        Resampling of the weight samples based on systematic resampling
        """
        N = len(q_norm)
        positions = (random() + np.arange(N)) / N
        indx = np.zeros(N, 'i')
        cumulative_sum = np.cumsum(q_norm)
        i, j = 0, 0
        while i < N:
            if positions[i] < cumulative_sum[j]:
                indx[i] = j
                i += 1
            else:
                j += 1
        for i, j in zip(range(0,self.N),indx):
            for l in range(0,self.num_layers):
                w[i][l]= np.array(w[j][l])
                b[i][l]= np.array(b[j][l])
        for l in range(0, self.N):
            q[l] = 1. / self.N

        return self.WeightsStore, self.BiasStore, q

    def Neff(self,q_norm):
        """
        Calculating the effective sample size of the q_norm
        """
        x = 1. / np.sum(np.power(q_norm,2))
        return x
    def Likelihood(self,output, a, R):
        """
        Calculating the likelihood of the outcome.
        """
        for i in range(0,self.N-1):
            self.L[i]=np.sqrt(1/(2*np.pi*R**2))*np.exp(-0.5 * ((output-a[i
                ]).dot(2*(1./R**2)).dot(output-a[i])))

        return self.L

def relu(z):
    return z * (z > 0)

def leaky_relu(z,epsilon=0.1):
    return np.maximum(epsilon*z,z)

def sigmoid(z):
    """The sigmoid function."""

    return 1.0/(1.0+np.exp(-z.astype(float)))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

def normalize(probs):
    """Normalizing the input, such that the sum equals one"""
    prob_factor = 1 / sum(probs)
    return np.array([prob_factor * p for p in probs])
```

B Python code for SMC algorithm with gradient step and ReLU activation function

Listing 2: SMC algorithm

```python
"""
Algorithm for performing SMC training with gradient step on a neural
    network with ReLU for time series.
The output is a one-step ahead prediction for data that is fed sequentially
     to the algorithm.
You can import this python file in another file, and then fill in the
    parameters:
sizes(structure of neural network), N(# monte carlo weight samples),
    sigma_initial_w(initial sigma's of weights),
sigma_initial_b(initial sigma's of biases). As seen in the __init__
    statement this creates an instance of
a neural net with these parameters. Then with the SMC function, one can
    obtain one step
ahead predictions on data. NOTE: The initial sigma's for the weights are
    given in an array that should have the same
length as the length of the sizes vector.

"""
import numpy as np
import time
import sys
from tqdm import tqdm
from multiprocessing import Pool
from joblib import Parallel, delayed
import multiprocessing


from numpy.random import random

class SMC_algorithm(object):

    def __init__(self,sizes, N, eta, layer_sigmas_w, layer_sigmas_b,
        posterior=False, epoch=False, weights=None, biases=None):
         self.sizes=sizes
         self.eta=eta
         self.output_dim=sizes[-1]
         self.num_layers = len(sizes)-1
         self.L0=np.zeros((N,1))
         self.L=np.zeros((N,1))
         self.sigma_initial_w=layer_sigmas_w
         self.sigma_initial_b=layer_sigmas_b
         self.forward_output=np.zeros((N,sizes[-1]))
         self.N=N
         self.posterior=posterior
         self.epoch=epoch
         if self.epoch:
              self.BiasStore=biases
              self.WeightsStore=weights
         else:
              self.BiasStore=[]
              self.WeightsStore=[]
```

```
def store_weights(self,w):
    self.WeightsStore.append(w)

def store_bias(self,b):
    self.BiasStore.append(b)

def initial_weights(self,N):
    """
    Creating the initial weight samples.
    """
    if self.num_layers == len(self.sigma_initial_b) and not self.epoch:
        for i in range(0,N):
            biases = [l*np.random.randn(y, 1) for y, l in zip(self.
                sizes[1:], self.sigma_initial_b)]
            weights = [l*np.random.randn(y, x)
                            for x, y, l in zip(self.sizes[:-1],
                                self.sizes[1:],self.sigma_initial_w)
                                ]

            self.store_weights(weights)
            self.store_bias(biases)
        # print self.WeightsStore
        self.WeightsStore=np.array(self.WeightsStore)
        self.BiasStore=np.array(self.BiasStore)

    elif self.epoch:
        self.WeightsStore=self.WeightsStore
        self.BiasStore=self.BiasStore

    else:
        print 'ERROR: initial weights sigma vector size is not equal to
            number of layers'

    return self.WeightsStore, self.BiasStore

def feedforward(self,w,b,u):
    """
    Feedforward algorithm for N different samples, for arbitrary size
        and layers of neural networks.
    """
    u=u[np.newaxis]

    for i in range(0,self.N):
        l=0

        while l < self.num_layers-1:
            if l==0:
                a = np.dot(u,w[i][l].T)+b[i][l].T
                a=relu(a)
                l+=1

            if l < self.num_layers-1:
                a = np.dot(a,w[i][l].T)+b[i][l].T
                a = relu(a)
                l+=1

        self.forward_output[i]=np.dot(a, w[i][l].T)+b[i][l].T
        return self.forward_output



    def update_weights(self,w,b,Q,R,x,y):
        """
        Updating the weights, meaning giving them a noise with sigma Q.
        """

        for i in range(0,self.N):
            for l in range(0,self.num_layers):

                self.BiasStore[i][l] = np.array(b[i][l] + np.random.normal
                    (0,Q,b[i][l].shape))
                self.WeightsStore[i][l] = np.array(w[i][l] + np.random.
                    normal(0,Q,w[i][l].shape))

        for i in range(0,self.N):

            bias=self.BiasStore[i]
            weight=self.WeightsStore[i]

            nabla_b = [np.zeros(b.shape) for b in bias]
            nabla_w = [np.zeros(w.shape) for w in weight]

            delta_nabla_b, delta_nabla_w = self.backprop(bias, weight, x, y
                , R)

            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

            self.WeightsStore[i] = [w-self.eta*nw
                        for w, nw in zip(self.WeightsStore[i], nabla_w)]
            self.BiasStore[i] = [b-self.eta*nb
                        for b, nb in zip(self.BiasStore[i], nabla_b)]

        return self.WeightsStore, self.BiasStore



    def backprop(self, bias, weights, x, y, R):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""

        nabla_b = [np.zeros(b.shape) for b in bias]
        nabla_w = [np.zeros(w.shape) for w in weights]

        activation = x[np.newaxis]
        activations = [activation] # list to store all the activations,
            layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(bias[:-1], weights[:-1]):
            z = np.dot(activation,w.T)+b.T

            zs.append(z)
            activation = relu(z)
            activations.append(activation)

        z = np.dot(activation,weights[-1].T)+b[-1].T
        zs.append(z)
        activation = z
        activations.append(activation)

        delta = cost_derivative(activations[-1], y, R)
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2])
        for l in range(2, self.num_layers+1):

            z = zs[-l]
            sp = relu_prime(z)
            delta = np.dot(delta, weights[-l+1]) * sp
            nabla_b[-l] = delta.T
            nabla_w[-l] = np.dot(delta.T, activations[-l-1])

        return (nabla_b, nabla_w)


    def SMC(self,data,real,Q, R,threshold,indweight=None):
        """
        This function gives a one step ahead prediction. The first element
            is 0 as one cannot give a prediction for
        the first data point. So the error should start with the second
            element of the prediction array compared with
        the second data point of the real vector.
        """
        K=len(real)
        print self.sizes
        resample_index=np.zeros((K,1))
        max_q=np.zeros((K,1))
        if self.posterior:
            posterior=np.zeros((self.N,1))
            posterior_timestep=np.zeros((K,self.N,1))
        outcome_posterior=np.zeros((K+1,self.N,self.output_dim))
        prediction_next_datapoint=np.zeros((K+1,1))
        iter=0
        for k in tqdm(range(0,K)):

            if k==0:
                w,b = self.initial_weights(self.N) # initialize Weights

                output_model=np.array(self.feedforward(w,b,data[k]))     #
                    get output from every weight sample i
                prediction=np.array(self.feedforward(w,b,data[k+1]))# get
                    output from every weight sample, with input of next data
                     point.

                L0=self.Likelihood(real[k],output_model,R) # get likelihood
                     of every weight sample i
                q=L0
                q_norm=normalize(q)
                max_q[k]=np.max(q_norm)
                outcome_posterior[k+1]=prediction
                prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)
                if self.posterior:
                    for p in range(0,self.N):
                        weightlayer=np.array(w[p][-1])
                        weightlayer=np.array(weightlayer[0])
                        posterior[p]=weightlayer[indweight]
                    posterior_timestep[k]=posterior
            else:
                w,b=self.update_weights(self.WeightsStore, self.BiasStore,Q,
                    R,data[k],real[k])
                output_model=np.array(self.feedforward(w, b,data[k]))
                if self.posterior:
                    for p in range(0,self.N):
                        weightlayer=np.array(w[p][-1])
                        weightlayer=np.array(weightlayer[-1])
                        posterior[p]=weightlayer[indweight]
                    posterior_timestep[k]=posterior

                L=self.Likelihood(real[k],output_model,R) # get likelihood
                    of every weight sample i
                q=q*L
                q_norm=normalize(q)
                prediction=np.array(self.feedforward(w,b,data[k+1]))
                prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)
                outcome_posterior[k+1]=prediction
                Neff=self.Neff(q_norm)
                max_q[k]=np.max(q_norm)

                if Neff<threshold or np.max(q)<1e-300:
                    resample_index[k]=1
                    iter += 1
                    w,b,q=self.resampling(w,b,q_norm,q)
                    q_norm=q

        print 'Number␣of␣resamples', iter+1
        return prediction_next_datapoint, posterior_timestep,
            outcome_posterior, w, b,resample_index,max_q



    def resampling(self,w,b,q_norm,q):
        """
        Resampling of the weight samples based on CDF.
        """
        N = len(q_norm)
        # make N subdivisions, and choose positions with a consistent
            random offset
        positions = (random() + np.arange(N)) / N
        indx = np.zeros(N, 'i')
        cumulative_sum = np.cumsum(q_norm)
        i, j = 0, 0

        while i < N:
            if positions[i] < cumulative_sum[j]:
                indx[i] = j
                i += 1
            else:
                j += 1
        for i, j in zip(range(0,self.N),indx):
            for l in range(0,self.num_layers):
                w[i][l]= np.array(w[j][l])
                b[i][l]= np.array(b[j][l])
        for l in range(0, self.N):
            q[l] = 1. / self.N

        return self.WeightsStore, self.BiasStore, q

    def Neff(self,q_norm):
        """
        Calculating the effective sample size of the q_norm
        """
        x = 1. / np.sum(np.power(q_norm,2))
        return x

    def Likelihood(self,output, a, R):
        """
        Calculating the likelihood of the outcome.
        """
        for i in range(0,self.N):
            self.L[i]=np.sqrt(1/(2*np.pi*R**2))*np.exp(-0.5 * ((output-a[i
               ]).dot(2*(1./R**2)).dot(output-a[i])))
        return self.L


def relu(z):
    return z * (z > 0)

def relu_prime(z):
    return (z>0)

def leaky_relu(z,epsilon=0.1):
    return np.maximum(epsilon*z,z)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z.astype(float)))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

def normalize(probs):
    """Normalizing the input, such that the sum equals one"""
    # print np.max(probs)
    prob_factor = 1 / np.sum(probs)
    return np.array([prob_factor * p for p in probs])

def cost_derivative(output_activations, y, R):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (1./R**2)*(output_activations-y)*(np.exp(-0.5 * ((
       output_activations-y)*(1./(R**2)))*(output_activations-y)))
```

C Python code for SMC algorithm with gradient step and ReLU activation function for a CNN

Listing 3: SMC algorithm

```
"""
Algorithm for performing SMC training on a convolutional neural network
    called WaveNet for time series.
The output is a one-step ahead prediction for data that is fed sequentially
    to the algorithm.
You can import this python file in another file, and then fill in the
    parameters:
sizes(structure of neural network), N(# monte carlo weight samples),
    sigma_initial_w(initial sigma's of weights),
sigma_initial_b(initial sigma's of biases). As seen in the __init__
    statement this creates an instance of
a neural net with these parameters. Then with the SMC function, one can
    obtain one step
ahead predictions on data. NOTE: The initial sigma's for the weights are
    given in an array that should have the same
length as the length of the sizes vector.

"""
import numpy as np
import time
import sys
from tqdm import tqdm
from multiprocessing import Pool
from joblib import Parallel, delayed
import multiprocessing

from numpy.random import random

class SMC_algorithm(object):

    def __init__(self,previous_points,filter_size, N,eta, posterior=False,
        epoch=False, weights=None, biases=None):
        self.filter_size=filter_size
        if filter_size % 2 ==0:
            self.num_layers=previous_points/(filter_size -1)-1
            print self.num_layers
            self.sizes=list(reversed([x for x in range(1,previous_points+1,
                filter_size -1)]))
        else:
            self.num_layers=int(previous_points/(filter_size -1.)-(1/(
                filter_size -1)))
            print self.num_layers
            self.sizes=list(reversed([x for x in range(1,previous_points+1,
                filter_size -1)]))
        self.output_dim=self.sizes[-1]
        self.L0=np.zeros((N,1))
        self.L=np.zeros((N,1))
        self.eta=eta
        self.forward_output=np.zeros((N,self.sizes[-1]))
        self.N=N
        self.posterior=posterior
        self.epoch=epoch
        if self.epoch:
            self.BiasStore=biases
            self.WeightsStore=weights
        else:
            self.BiasStore=[]
            self.WeightsStore=[]

    def store_weights(self,w):
        self.WeightsStore.append(w)

    def store_bias(self,b):
        self.BiasStore.append(b)

    def initial_weights(self,N):
        """
        Creating the initial weight samples.
        """
        if not self.epoch:

            sigmas=[1]*self.num_layers
            for i in range(0,self.num_layers):
                sigmas[i]=np.sqrt(2./(self.sizes[i+1]))

            for i in range(0,N):
                biases = [l*np.random.randn(y, 1) for y, l in zip(self.
                    sizes[1:], sigmas)]
                weights = [l*np.random.randn(self.filter_size,1)
                                    for x, l in zip(range(0,self.num_layers
                                        ), sigmas)]
                self.store_weights(weights)
                self.store_bias(biases)

            # print self.WeightsStore
            # self.WeightsStore=np.array(self.WeightsStore)
            self.BiasStore=np.array(self.BiasStore)

        elif self.epoch:
            self.WeightsStore=self.WeightsStore
            self.BiasStore=self.BiasStore

        else:
            print 'ERROR: initial weights sigma vector size is not equal to
                number of layers'

        return self.WeightsStore, self.BiasStore

    def feedforward(self,w,b,u):
        """
        Feedforward algorithm for N different samples, for arbitrary size
            and layers of neural networks.
        """
        u=u[np.newaxis]
        for i in range(0,self.N):
            l=0

            while l < self.num_layers-1:
                if l==0:
                    a = np.convolve(u.flatten(),np.rot90(w[i][l],2).flatten
                        (),'valid')+b[i][l].T
                    a=relu(a).T
                    l+=1
                if l < self.num_layers-1:
                    a = np.convolve(a.flatten(),np.rot90(w[i][l],2).flatten
                        (),'valid')+b[i][l].T
                    a = relu(a).T
                    l+=1
            self.forward_output[i]= np.convolve(a.flatten(),np.rot90(w[i][l
                ],2).flatten(),'valid')+b[i][l].T

        return self.forward_output

    def backprop(self, bias, weights, x, y,R):
        """Return a tuple ``(nabla_b, nabla_w)`` representing the
        gradient for the cost function C_x.  ``nabla_b`` and
        ``nabla_w`` are layer-by-layer lists of numpy arrays, similar
        to ``self.biases`` and ``self.weights``."""

        nabla_b = [np.zeros(b.shape) for b in bias]
        nabla_w = [np.zeros(w.shape) for w in weights]

        activation = x[np.newaxis]
        activations = [activation] # list to store all the activations,
            layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(bias[:-1], weights[:-1]):
            z = np.convolve(activation.flatten(),np.rot90(w,2).flatten(),'
                valid')+b.T

            zs.append(z)
            activation = relu(z)
            activations.append(activation)

        z = np.convolve(activation.flatten(),np.rot90(weights[-1],2).
            flatten(),'valid')+bias[-1].T
        zs.append(z)
        activation = z
        activations.append(activation)
        # backward pass

        delta = cost_derivative(activations[-1], y,R)
        nabla_b[-1] = delta
        nabla_w[-1] = np.convolve(np.rot90(delta,2).flatten(),activations
            [-2].flatten(), 'valid')
        for l in range(2, self.num_layers+1):

            z = zs[-l]
            sp = relu_prime(z)
            delta = np.convolve(delta.flatten(), weights[-l+1].flatten()) *
                 sp
            nabla_b[-l] = delta.T
            nabla_w[-l] = np.convolve(np.rot90(delta.T,2).flatten(),
                activations[-l-1].flatten(),'valid')

        return (nabla_b, nabla_w)
    def update_weights(self,w,b,Q,R,x,y):
        """
        Updating the weights, meaning giving them a noise with sigma Q.
        """

        for i in range(0,self.N):
            for l in range(0,self.num_layers):
                self.BiasStore[i][l] = np.array(b[i][l] + np.random.normal
                    (0,Q,b[i][l].shape))
                self.WeightsStore[i][l] = np.array(w[i][l] + np.random.
                    normal(0,Q,w[i][l].shape))

        for i in range(0,self.N):

            bias=self.BiasStore[i]
            weight=self.WeightsStore[i]

            nabla_b = [np.zeros(b.shape) for b in bias]
            nabla_w = [np.zeros(w.shape) for w in weight]

            delta_nabla_b, delta_nabla_w = self.backprop(bias, weight, x, y
                ,R)

            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw[np.newaxis].T for nw, dnw in zip(nabla_w,
                delta_nabla_w)]

            self.WeightsStore[i] = [w-self.eta*nw
                        for w, nw in zip(self.WeightsStore[i], nabla_w)]
            self.BiasStore[i] = [b-self.eta*nb
                        for b, nb in zip(self.BiasStore[i], nabla_b)]

        return self.WeightsStore, self.BiasStore

    def SMC(self,data,real,Q, R,threshold,indweight=None):
        """
        This function gives a one step ahead prediction. The first element
            is 0 as one cannot give a prediction for
        the first data point. So the error should start with the second
            element of the prediction array compared with
        the second data point of the real vector.
        """
        K=len(real)
        print self.sizes
        resample_index=np.zeros((K,1))
        if self.posterior:
            posterior=np.zeros((self.N,1))
            posterior_timestep=np.zeros((K,self.N,1))
        outcome_posterior=np.zeros((K+1,self.N,self.output_dim))
        prediction_next_datapoint=np.zeros((K+1,1))
        iter=0

        for k in tqdm(range(0,K)):

            if k==0:
                w,b = self.initial_weights(self.N) # initialize Weights
                output_model=np.array(self.feedforward(w,b,data[k]))    #
                    get output from every weight sample i
                prediction=np.array(self.feedforward(w,b,data[k+1]))# get
                    output from every weight sample, with input of next data
                     point.
                L0=self.Likelihood(real[k],output_model,R) # get likelihood
                     of every weight sample i
                q=L0
                q_norm=normalize(q)
                outcome_posterior[k+1]=prediction
                prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)

                if self.posterior:
                    for p in range(0,self.N):
                        weightlayer=np.array(w[p][-1])
                        posterior[p]=weightlayer[indweight]
                    posterior_timestep[k]=posterior

            else:

                w,b=self.update_weights(self.WeightsStore,self.BiasStore,Q,
                    R,data[k],real[k])
                output_model=np.array(self.feedforward(w, b,data[k])) # get
                     output from every weight sample i

                if self.posterior:
                    for p in range(0,self.N):
                        weightlayer=np.array(w[p][-1])
                        posterior[p]=weightlayer[indweight]
                    posterior_timestep[k]=posterior

                L=self.Likelihood(real[k],output_model,R) # get likelihood
                    of every weight sample i
                q=q*L
                q_norm=normalize(q)
                prediction=np.array(self.feedforward(w,b,data[k+1]))
                prediction_next_datapoint[k+1]=np.nansum(q_norm*prediction)
                outcome_posterior[k+1]=prediction
                Neff=self.Neff(q_norm)

                if Neff<threshold or np.max(q)<1e-200: #cutoff to reduce
                    chance of overflow
                    resample_index[k]=1
                    iter += 1
                    w,b,q=self.resampling(w,b,q_norm,q)
                    q_norm=q

        print 'Number␣of␣resamples', iter+1

        return prediction_next_datapoint, posterior_timestep,
            outcome_posterior, w, b,resample_index

    def resampling(self,w,b,q_norm,q):
        """
        Resampling of the weight samples based on CDF.
        """
        N = len(q_norm)
        # make N subdivisions, and choose positions with a consistent
            random offset
        positions = (random() + np.arange(N)) / N
        indx = np.zeros(N, 'i')
        cumulative_sum = np.cumsum(q_norm)
        i, j = 0, 0

        while i < N:
            if positions[i] < cumulative_sum[j]:
                indx[i] = j
                i += 1
            else:
                j += 1

        for i, j in zip(range(0,self.N),indx):
            for l in range(0,self.num_layers):
                w[i][l]= np.array(w[j][l])
                b[i][l]= np.array(b[j][l])
        for l in range(0, self.N):
            q[l] = 1. / self.N

        return self.WeightsStore, self.BiasStore, q

    def Neff(self,q_norm):
        """
        Calculating the effective sample size of the q_norm
        """
        x = 1. / np.sum(np.power(q_norm,2))
        return x

    def Likelihood(self,output, a, R):
        """
        Calculating the likelihood of the outcome.
        """
        for i in range(0,self.N):
            self.L[i]=np.sqrt(1/(2*np.pi*R**2))*np.exp(-0.5 * ((output-a[i
                ]).dot(2*(1./R**2)).dot(output-a[i])))
        return self.L

def relu(z):
    return z * (z > 0)

def relu_prime(z):
    return (z>0)

def leaky_relu(z,epsilon=0.1):
    return np.maximum(epsilon*z,z)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z.astype(float)))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))

def normalize(probs):
    """Normalizing the input, such that the sum equals one"""
    prob_factor = 1 / sum(probs)
    return np.array([prob_factor * p for p in probs])
```

```
def cost_derivative(output_activations, y, R):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (1./R**2)*(output_activations-y)*(np.exp(-0.5 * ((
        output_activations-y)*(1./(R**2)))*(output_activations-y)))
```

References

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

JM Bernardo, MJ Bayarri, JO Berger, AP Dawid, D Heckerman, AFM Smith, and M West. Sequential monte carlo for bayesian computation. In *Bayesian statistics 8: proceedings of the eighth Valencia International Meeting, June 2-6, 2006*, volume 8, page 115. Oxford University Press, USA, 2007.

C Bishop. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*, 2007.

Anastasia Borovykh. A Gaussian process perspective on convolutional neural networks. *Research Gate*, 2018.

James Carpenter, Peter Clifford, and Paul Fearnhead. Improved particle filter for nonlinear problems. *IEE Proceedings-Radar, Sonar and Navigation*, 146(1):2–7, 1999.

Dan Crisan and Arnaud Doucet. Convergence of sequential monte carlo methods. *Signal Processing Group, Department of Engineering, University of Cambridge, Technical Report CUEDIF-INFENGrrR38*, 1, 2000.

J. F. G. de Freitas, M. Niranjan, and A. H. Gee. Hierarchical bayesian-kalman for regularisation and ard in sequential learning. Technical Report CUED/F-INFENG/TR 307, Cambridge: Cambridge University, 10 1998. Available at http://mi.eng.cam.ac.uk/reports/index-full.html.

João Ferdinando Gomes de Freitas. *Bayesian methods for neural networks*. PhD thesis, Citeseer, 2003.

Arnaud Doucet and Adam M Johansen. A tutorial on particle filtering and smoothing: Fifteen years later. *Handbook of nonlinear filtering*, 12(656-704):3, 2009.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and computing*, 10(3):197–208, 2000.

JFG de Freitas, Mahesan Niranjan, Andrew H. Gee, and Arnaud Doucet. Sequential monte carlo methods to train neural network models. *Neural computation*, 12(4):955–993, 2000.

Zoubin Ghahramani. An introduction to hidden markov models and bayesian networks. *International journal of pattern recognition and artificial intelligence*, 15(01):9–42, 2001.

Xavier Glorot and Yoshua Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, 2010.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE Proceedings F-radar and signal processing*, volume 140, pages 107–113. IET, 1993.

Alex Graves. Practical variational inference for neural networks. In *Advances in neural information processing systems*, pages 2348–2356, 2011.

Shixiang Gu, Zoubin Ghahramani, and Richard E Turner. Neural adaptive sequential monte carlo. In *Advances in Neural Information Processing Systems*, pages 2629–2637, 2015.

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.

Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The" wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

Jeroen D Hol, Thomas B Schon, and Fredrik Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop*, pages 79–82. IEEE, 2006.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Nikolas Kantas. *Sequential Decision Making in General State Space models*. PhD thesis, University of Cambridge, 2 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

David JC MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

David JC MacKay. Probable networks and plausible predictions—a review of practical bayesian methods for supervised neural networks. *Network: computation in neural systems*, 6(3):469–505, 1995.

Andriy Mnih and Karol Gregor. Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030*, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Kevin P Murphy. Machine learning: a probabilistic perspective, 2012.

Radford M Neal. Bayesian training of backpropagation networks by the hybrid monte carlo method. Technical report, Citeseer, 1992.

Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. Available at http://neuralnetworksanddeeplearning.com/.

Claudio Sbarufatti, Matteo Corbetta, Marco Giglio, and Francesco Cadini. Adaptive prognosis of lithium-ion batteries based on the combination of particle filters and radial basis function neural networks. *Journal of Power Sources*, 344:128–140, 2017.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. *ArXiv e-prints*, 2016.

VS Zaritskii, VB Svetnik, and LI Šimelevič. Monte-carlo technique in problems of optimal information processing. *Avtomatika i Telemekhanika*, (12):95–103, 1975.