UPPSALA
UNIVERSITET

Black Box Time Series Modeling

Simon Strömstedt Hallberg, Adam Lindell

Project in Computational Science: Report

February 2018

PROJECT REPORT

Institutionen för informationsteknologi

Contents

1 Popular Science Summary

Sometimes when looking at one or many graphs one can convince oneself that they know how the data is connected and therefore they can make a good prediction of how the graph will look in the future. This is done all the time in, for instance, the financial market whether it be stocks, options, bonds, currencies or any number of derivatives. This colossal industry is based on the assumption that such predictions can be made accurately enough to generate profits. This leads to the topic of this report: neural nets. A neural net is a collection of nodes (neurons), which given an input, will produce processed output. These nodes are very simple functions that link to other neurons in the network (see example here 2). This network can be trained on a collection of data, known as a training set. The data set containing patterns to be learned is fed into the network, and the training process tunes the architecture of the network to ensure that the network understands the underlying patterns in the data. It consist of these steps:

1) A data point is fed into the network (and the wanted output also has to be known) where every neurons at this point has a certain specification.
2) The network produces some output value that depends on the input and the specifications of the neurons.
3) The network compares the output it gave and the output it should ideally give and tweaks the specification in the necessary neurons to suit this wanted output.
4) The next data point is fed into the network and the process is repeated.

The network is now trained and can be put through testing where the performance is validated. After sufficient validation the network is ready for deployment. Essentially, a trained neural network is a model/machine that is trained to predict the output, given a certain input.

The goal of this project was to obtain fast models that learn the behavior of biochemical reactions that are popular in systems biology .

2 Abstract

This project is a study of how accurately stochastic time series can be modeled using recurrent neural networks instead of stochastic time series models based on the Gillespie algorithm [1], in order to decrease computation time. Dealing with stochasticity and temporal data is discussed. In order to achieve optimal accuracy, stochastic modeling approaches may be explored. Results indicate that the proposed methodology has great potential for achieving sufficient accuracy in the near future. A link to the project's Github page is provided in the references section.

3 Introduction and Background

Consider a biochemical system consisting of species undergoing certain reactions. This then results in a number of time series (as many as the number of different species) where the output of every series is the population count of the specific specie considered. In this project only one representative specie was modeled for simplicity.

However, at the current state of the field, these types of time series are simulated by various models based on the Gillespie algorithm. For the model used in this project, the Tyson oscillator[11], to simulate one time series on a personal laptop takes about 5 seconds. This amount of time is acceptable if not too many different time series have to be simulated and they are not needed too quickly.

Consider the following scenario. Simulation of a time series is done in a packaged function F. For applications such as parameter exploration, several repeated calls to the function F will need to be made. Each of these calls may have a different combination of parameters and time-step values. Now consider that this function call is within a large for-loop. It is obvious that the Gillespie algorithm is too slow for frequent usage.

The project explores neural networks as fast approximations of computationally expensive simulators. A trained NARX (Non-linear Autoregressive Exogenous input) or LSTM (Long Short Term Memory) network can be used for repeated calls many times faster than the computationally expensive Gillespie algorithm.

4 Project Description

The project was to model a stochastic time series using different machine learning approaches. The models used, described in section 6.3 of this report, are different kinds of recurrent neural networks. In particular, LSTM and NARX are considered. The following tasks were performed.

- Practically evaluating model types for time series encountered in systems biology.
- Studying the robustness and expressive power of each model type for such time series.
- Exploring model training methods that minimize training time while retaining model accuracy.
- Formulating pointers towards black box modeling of time series in reaction networks in systems biology.

The project was not to package a ready and trained network but more towards proof-of-concept of the general approach towards approximating stochastic time series using recurrent neural networks.

5 Scope

The underlying simulation model depends on seven parameters. To do a very rigorous exploration of the parameters one would ideally like to have at least ten different values for every parameter. However this would result in 10^7 possible combinations of parameter values. One combination of parameter values corresponds to one time series and the Tyson oscillator takes approximately five seconds to generate one time series. This means generating all the time series would take $5 \cdot 10^7$ seconds, which is almost 1.59 years. Also the computers used would have no way of dealing with such a large data set. Therefore we restricted us to varying three parameters with five values each resulting in 125 time series.

6 Method

6.1 Tools

The tools used for this project were the Python Application Programming Interface (API) *Keras* [2], the MATLAB function *NeuralNetworkToolbox* and the Tyson oscillator example from the GillesPy library[3].

6.2 Data generation

The training data used for this project was generated from the Tyson oscillator simulator by varying three out of the seven parameters for five different parameter values each, resulting in a total of 125 time series of 199 time steps. The number of time steps was chosen to be sufficiently high enough to observe meaningful changes in the population count of the specie, but not so high as to increase the computational burden unnecessarily in proof-of-concept experiments. The test data was generated in a similar manner, but only four values were used for every parameter, resulting in 64 time series. Table 1 shows values used for the parameters of the training data.

Generating these 125 time series of size 199 time steps took approximately 11 minutes on a personal laptop with the following specifications: RAM: 7,7 GB
Processor: Intel® Core™ i7-4700HQ CPU @ 2.40GHz × 8
Operating system: Ubuntu 16.04 LTS

Table 1: Parameter values used to generate training data. The first four parameters were fixed while all the possible combinations of the last three were used. First row are the parameter names which are taken from the Tyson oscillator simulator.

P	kt	kd	a0	a1	a2	kdx
2	20	1	0.005	0.0350	0.0700	0.700
				0.0425	0.0850	0.850
				0.0500	0.1	1.00
				0.0575	0.115	1.15
				0.0650	0.130	1.30

6.3 Model Introduction

In this section the two different neural network types that were explored, LSTM and NARX, are introduced.

6.3.1 Long Short Term Memory neural network (LSTM)

An LSTM network is a recurrent neural network. Recurrent neural networks differ from feed-forward neural networks by being designed to use predicted values at time t in order to predict values at time t+1. This implies that an LSTM network has to remember values over time. The reason for wanting to be able to use values of previous predictions and having a memory implemented in a model is to be able to capture possibly repeating structures that appear over time, as in time series for example. Consider the weather. If one wants to predict the weather in Uppsala tomorrow, one would take the weather of the past week (or even longer) into consideration. If it's been snowing all week, it's not very likely that the temperature would be high tomorrow, the weather of the future is dependent on the weather of the past. This is why LSTM's are useful, they can remember past events and patterns over time. The technical details of LSTM networks are explained below.

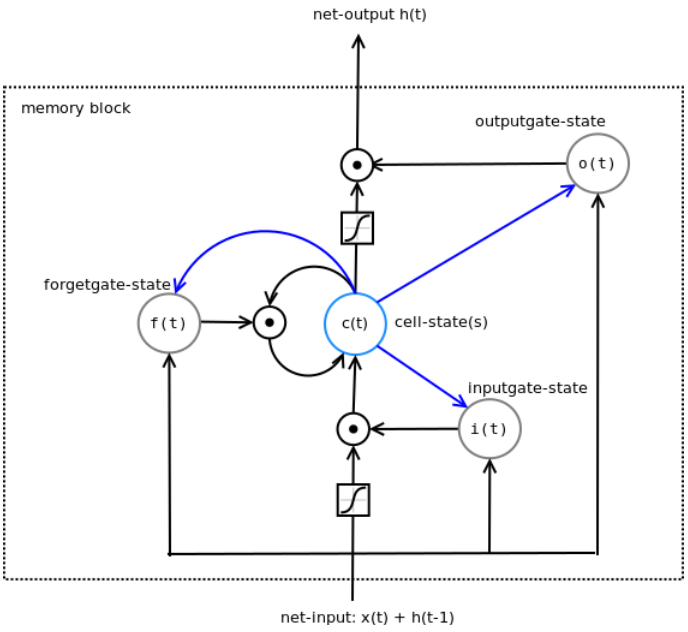

Figure 1: Illustration of an LSTM network.

Figure 1[4] shows a graphical representation of an LSTM. The different components of the LSTM network and its properties are explained below.

An LSTM network is trained using back propagation through time and it solves the vanishing gradient problem. The vanishing gradient problem is a common problem when training artificial neural networks. During training the weights of the neural network are updated based on gradient descent of the error function. The error function is usually a function describing the difference

between predicted values and real values and the aim is to minimize it. Therefore the gradient of this function reveals the direction of which the weights should be adjusted in order to decrease the error the most. Sometimes these gradients are so small that the weights are not affected. The reason for this is that backpropagation uses the chain rule when computing gradients. This means that small values are multiplied with each other all the way from the current layer to the front layer of the network, resulting in an exponential decrease of the gradient. The front layers of the network then receive a vanishingly small gradient and train very slowly.

Instead of neurons, LSTM networks have memory blocks which are connected through its different layers. A memory block has different components that help it regulate how values flow through the system. A sequence is put through a block and gates inside it are triggered conditionally by sigmoid activation functions in order to regulate how the values of the sequence are used, while a cell component store values over time. A sigmoid activation function is a logistic function which, depending on input, returns a one or a zero, representing on or off. There are three types of gates.
The input gate controls the extent to which new values flow into memory.
The forget gate controls the extent to which a value remains in memory.
The output gate controls the extent to which the input and memory is used to compute the output. [5]

The following equations describe how an LSTM network takes an input sequence $x = (x_1, x_2, ..., x_T)$ and uses it in order to arrive at an output sequence $y = (y_1, y_2, ..., y_T)$. This is done by calculating the network unit activations using the equations below iteratively from t=1 to t=T.

$$\mathbf{i}_t = \sigma(W_{ix}\mathbf{x}_t + W_{im}\mathbf{m}_{t-1} + W_{ic}\mathbf{c}_{t-1} + \mathbf{b}_i) \tag{1}$$

$$\mathbf{f}_t = \sigma(W_{fx}\mathbf{x}_t + W_{fm}\mathbf{m}_{t-1} + W_{fc}\mathbf{c}_{t-1} + \mathbf{b}_f) \tag{2}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot g(W_{cx}\mathbf{x}_t + W_{cm}\mathbf{m}_{t-1} + \mathbf{b}_c) \tag{3}$$

$$\mathbf{o}_t = (W_{ox}\mathbf{x}_t + W_{om}\mathbf{m}_{t-1} + W_{oc}\mathbf{c}_t + \mathbf{b}_o) \tag{4}$$

$$\mathbf{m}_t = \mathbf{o}_t \odot h(\mathbf{c}_t) \tag{5}$$

$$\mathbf{y}_t = (W_{ym}\mathbf{m}_t + \mathbf{b}_y) \tag{6}$$

where $\mathbf{i}$, $\mathbf{f}$ and $\mathbf{o}$ represent the gates (input, forget, output) $\mathbf{c}$ represents the cell and $\mathbf{m}$ represents the output activation. The different W terms are the weight matrices, where for example W_{ix} is the weight matrix from the input gate to the input. W_{ic}, W_{fc}, W_{oc} are the diagonal weight matrices from the respective gates to the cell. These are called peephole connections and they exist so that the LSTM network can learn precise timing of the outputs. The b terms are the bias vectors, they can be viewed as adjustments of missing information. $\odot$ is the element-wise product of the vectors, g and h are the cell input and cell output activation functions. ϕ is the network output activation function. [6]

6.3.2 Non-linear Autoregressive Exogenous input neural network (NARX)

A NARX network is a very natural network for the problem considered since it is recurrent (uses previous value/values) and therefore captures the time dependence of the input data. A graphical overview of the network can be seen in Figure 2. This is what is known as a shallow neural network since the number of layers of neurons is small. The network has two distinct layers, as can be seen in Figure 2.

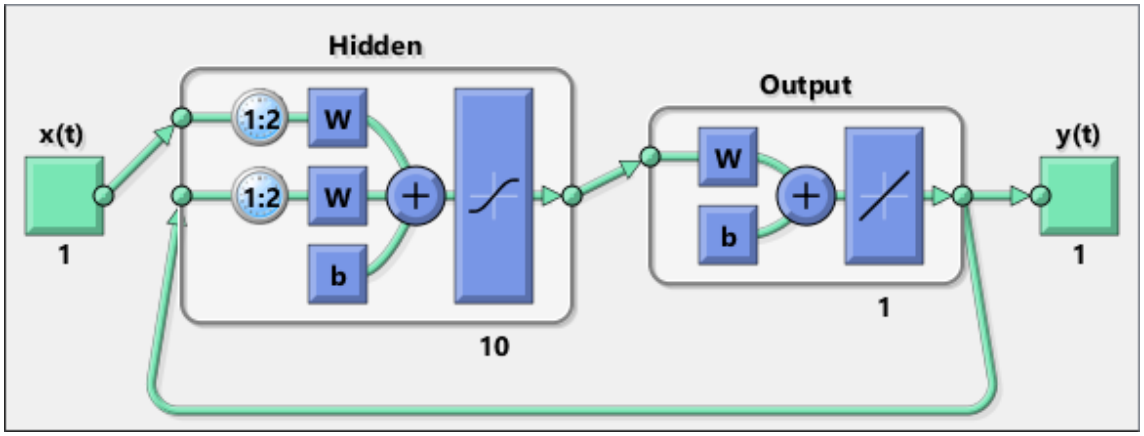

Figure 2: A graphical view of the NARX network with 10 neurons. In our case, the exogenous input was a vector of the parameters along with the time series y.

In Figure 2 [7], the first layer of neurons, labeled "Hidden", are sigmoid functions, an example being $\frac{1}{1+e^{-x}}$ which maps values on the real line to a value between zero and one. From input to these neurons there are weights w for each one which can act to steepen the sigmoid function curve. This is one of two ways of manipulating the activation function. The other is called a bias, which involves shifting the activation function so that it, for instance, gives values between negative one and zero. So, intuitively, when one wants to manipulate a sigmoid function in the context of a neural network one thinks of shifting and scaling, which these two (weights and bias) are. Then these neurons combine into a transfer function which is responsible for giving an output of the form

$$y(t) = f(y(t-1), u(t-1)) \tag{7}$$

where u is the exogenous input i.e. the parameters and $y(t-1)$ the previous output value.

The update algorithm used for the neural network during training was the default function (Levenberg-Marquardt) [8]. This back propagation algorithm is recommended although it uses more memory than its counterparts.

6.4 Implementation

6.4.1 NARX

The NARX network was implemented in MATLAB which has a library for such a network. This led to significant code reduction as opposed to if NARX was implemented in, for instance C. The script is approximately 120 lines. NARX exists in other languages as well, for instance Python in the library PyNeurgen. But since MATLAB has been used extensively though out our university education and a lot of resources regarding the matter existed on-line it was decided to use MATLAB.

For simply trying a NARX network in MATLAB there is an easy access to the NARX tool with the simple command "ntstool". There it is possible to use the graphic interface to train a network on either example data or given data. However this graphical interface did not suffice for the project due to format and other factors, so writing code was necessary.

6.4.2 LSTM

The LSTM model was implemented in Python, Keras, as mentioned in Section 6.1. The reason for this was primarily because the Tyson oscillator model was written in Python so the data had to be generated there, which made continuing in the same programming language a natural step. Also Python is a popular programming language for machine learning. There are a lot of out of the box tools to use and the online community, which consists of everything from young programming

enthusiasts to respected scientists, is a great asset.

The LSTM network was built with the Model Application Programming Interface (API) which is probably the most low level interface in the otherwise high level API Keras. This allowed for some custom layer implementations and the input and output could be modified to a greater extent. The input parameters were fed into an LSTM block which output a sequence of values which were then fed into another LSTM block which finally output the whole time series for that set of parameters. This was done for every set of parameters and their corresponding time series. The output time series from the model were then compared to the real time series corresponding to the same set of parameters.

One of the parameter sets was also chosen arbitrarily to compute the standard deviation of the values in every time step when generated with the Tyson oscillator ten times. This was done to get a better understanding of how much the stochastic element affects the output, and to be able to get more context for the test series prediction of the same parameter set.

6.5 Pre-processing

6.5.1 NARX

No pre-processing was done for and with the NARX network because the library functions did not require it.

6.5.2 LSTM

The data had to be normalized before it could be used for training the model. This was done by making all the values in the data set lie within the interval [0,1]. Normalizing data sets are standard practice when working with neural networks or any kind of machine learning algorithm because it reduces potential prior bias from specific variables. Some models have this inherently implemented, but this was not the case in this LSTM network. The formula for the normalization was $\frac{x-min}{max-min}$, where max was the largest value of the dataset, min was the smallest value of the dataset and x was a specific value in a time series.

7 Results

7.1 NARX

The NARX network was trained with four neurons on a dataset with 125 time series of 199 time steps each. The network was tested on 40 time series and one of the best performing networks in terms of the sum of the error at every time step is shown below. The training of the network took about two and a half minutes although this varies from approximately one to six minutes. We suspect this is due to the validation data. The script is written so that MATLAB chooses a certain percentage of the data points for validation, but the data points are chosen randomly. If these were to fall in a especially volatile region of the curve this could affect the validation error which would tell the network it needs further training. However the average training time was approximately two minutes.

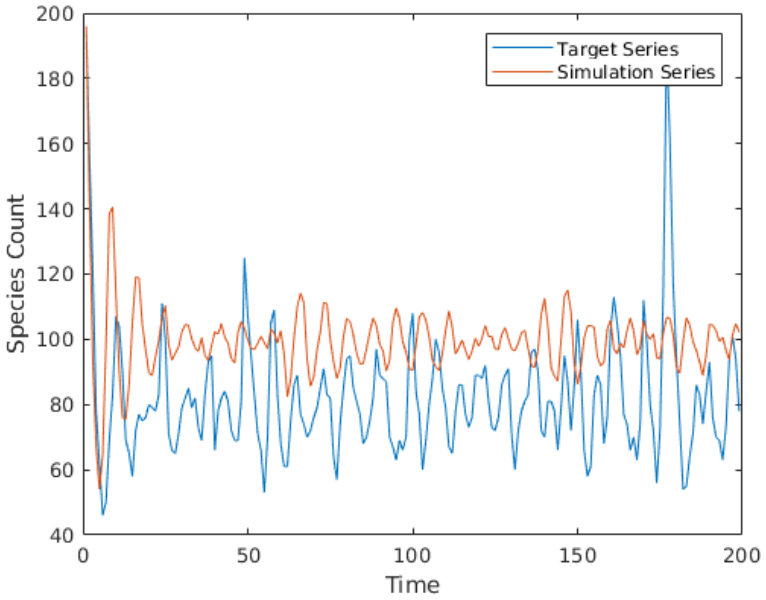

Figure 3: Plot of the test series with the lowest error. The Relative Error (RE): 0.3741. The Mean Squared Error (MSE): 42.34.

7.2 LSTM

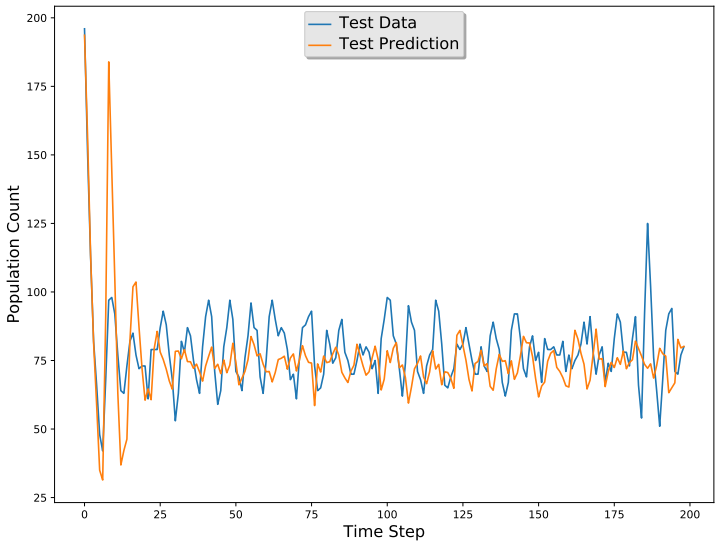

Figure 4: Plot of the test series with the lowest error, Mean Absolute Error (MAE) was 10.80 for this specific series, the average MAE for all the test series of this run was 34.97, the average MAE for all training series of this run was 33.95

Figure 4 illustrates how well the predictions fit the test data generated from the Tyson oscillator in the best case. It can be seen that the orange line (the predictions) follows the general structure of the blue line (the test data), but it often fails to predict perfectly for every time step. This can be attributed to the stochasticity of the problem.

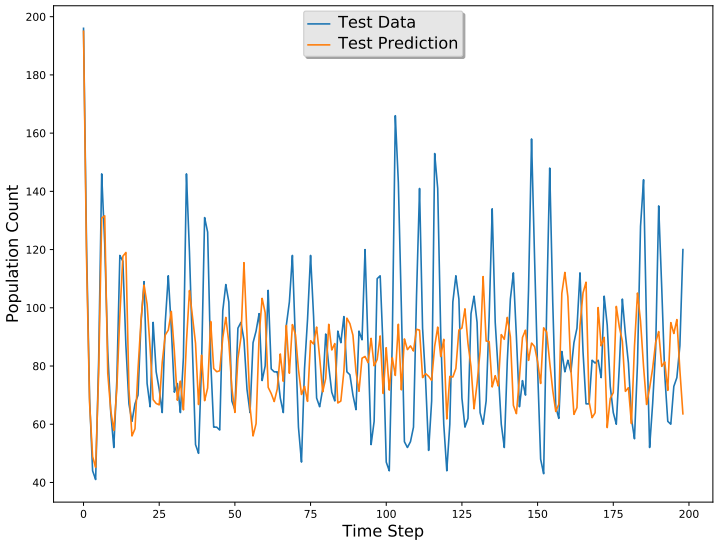

Figure 5: Plot of the last test series

Figure 5 shows the same result as Figure 4, but this is an arbitrarily chosen time series, namely the last of all the test series (number 64), selected to compute standard deviation. This figure complements the figure shown below, which includes the standard deviation.

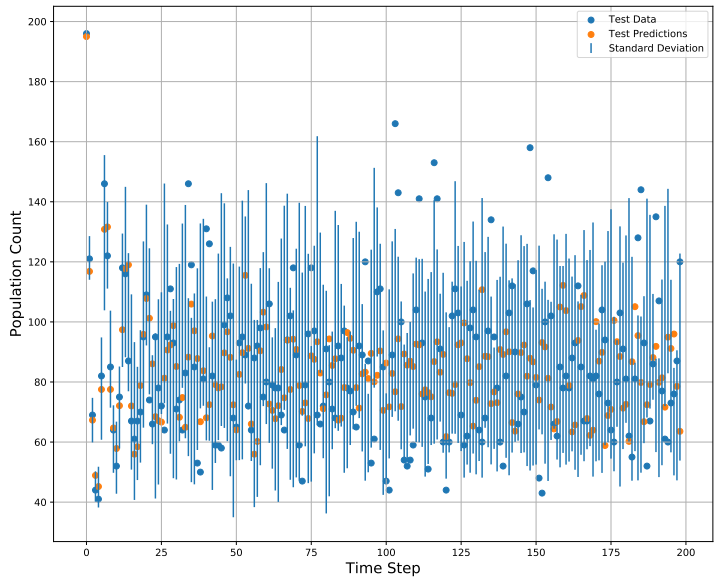

Figure 6: Plot of the last test series, blue lines are standard deviation, blue dots are test data and orange dots are test predictions

Figure 6 shows the plot of the same time series as Figure 5, but as a scatter plot and with standard deviation bars added. It's shown that most of the predictions (orange dots) are within the standard deviation bars which means that the model usually predicts the population count within the standard deviation of true data. Some of the real data points (blue dots) are outside the

standard deviation bars which illustrates the high variance between different runs with the same set of parameters.

Training the model for 300 epochs took about 24 seconds, one epoch is one iteration through the training data. Generating the 64 time series after training took less than a second. This was using a laptop with the following specifications:
RAM: 7,7 GB
Processor: Intel® Core™ i7-4700HQ CPU @ 2.40GHz × 8
Operating system: Ubuntu 16.04 LTS

8 Discussion

8.1 General discussion

There are a lot of improvements to be made in general and it mostly concerns the data. Since only three out of the seven parameters that affect the outcome were varied, only part of the structure could be learned by the models. This means that using all of the variables when training the models would be interesting to explore as future work. Another substantial improvement would be to implement a Gaussian process instead of, or in complement to, the other models, in order to deal with the stochastic behavior of the time series. Since the same set of parameter values generates a slightly different time series every time the Tyson oscillator is called, and the models are fed only one time series per parameter set there is no way for them to take the stochasticity into consideration when trained.

8.2 NARX

Below we present and discuss the implementation and the results from the NARX model.

8.2.1 Capturing the Stochastic Behavior

A Gillespie algorithm, and of course by extension a biochemical reaction system for which it was originally developed, is stochastic in nature. But since a trained network is deterministic, i.e. a certain input will always give the same output, which contradicts with the stochastic nature of the Gillespie algorithm. Thus the stochasticity cannot be captured by a deterministic model alone. In future, stochastic variants of recurrent neural networks can be explored, along with Gaussian kernels to account for stochasticity in the target problems.

8.2.2 RAM Usage and Toolbox Use

One problem encountered during the project was that MATLAB's NARX tool seem to deal with the input data all at once and not one after another which makes the RAM usage a large problem.

When training the network on the time series which were 101 time steps long the RAM on a computer (8 GB plus some virtual memory), the NARX tool could use up to 13. However, when using the time series 200 elements long the tool could use four neurons only. This RAM restriction is significant since we could not analyze the performance when using more neurons. Even though the results obtained were reasonable it would be interesting to investigate the performance of the network for more neurons.

8.2.3 Incorrect Training and Input Form

Another problem encountered during the project was the format of the data required by MATLAB. Observe the matrices below.

$$Targets = \begin{pmatrix} \tau_{1,1} & \tau_{1,2} & ...\tau_{1,T} \\ \tau_{2,1} & \tau_{2,2} & ...\tau_{2,T} \\ \tau_{3,1} & \tau_{3,2} & ...\tau_{3,T} \\ . & & . \\ . & & . \\ . & & . \\ \tau_{n,1} & \tau_{n,2} & \tau_{n,T} \end{pmatrix}, \quad Parameters = \begin{pmatrix} p_{1,1} & p_{1,2} & ...p_{1,7} \\ p_{2,1} & p_{2,2} & ...p_{2,7} \\ . & & . \\ . & & . \\ . & & . \\ p_{n,1} & p_{n,2} & ...p_{n,7} \end{pmatrix}$$

In the Target's matrix the data is ordered by a certain time series on a line and every column is a value in time. For the Parameters matrix every line is a certain subset of parameter values. The data then had to be compressed into what MATLAB calls a cell array.
$Targets$: $([1xT]_1, [1xT]_2, ..., [1xT]_n)$
$Parameters$: $([1x7]_1, [1x7]_2, ..., [1x7]_n)$

The tool now views these two vectors as the time series, i.e. the time series has T values per time step and a vector of the parameter subset of length n. This is a bit reversed because we would rather have that the tool interprets the data as n values with a time series T long. This is no problem for the $Targets$ matrix, but for the $Parameters$ matrix it certainly is. If we were to format our $Targets$ cell array this way i.e. $Targets_{cell} = ([1xn]_1, [1xn]_2, ..., [1xn]_T)$ the following problem is encountered. There is no way to format the $Parameters$ matrix so that it is a cell array of the same length and format. As future work, better architectures of handling data can be explored.

8.3 LSTM

This section includes a discussion of the LSTM model, its implementation and results.

8.3.1 Bulding the model in Python, Keras

There were some difficulties with building the network in Keras. The shape of the input data had to be in a specific format. The parameters that were given as input weren't in time series format, but rather initial values of the system. Therefore they had to be converted into a whole time series somewhere in the network. This is why two LSTM blocks were implemented. When trying to use only one LSTM block the model expected the output to be in the same format as the input. This is generally not a problem, a layer shouldn't have to have its output in the same format as the input, but for some reason this was a problem in this case, not unlikely due to the authors' limited knowledge of Python and Keras.

8.3.2 Feeding data into the network

The process of feeding data into the built model was somewhat difficult. Since the data had to be in a specific format in order for the model to be able to interpret it, there had to be some preprocessing. As mentioned in Section 6 the preprocessing did not just consist of normalizing the dataset, but the data also had to be reshaped in order to fit into the model. The shape of the data had to fit the expected format of the specific input and output layers of the model. For example, an LSTM block needs to be fed data in the format of [samples, time steps, features] while other types of layers might require different input shapes. This means that the process of trying many different models or layers means having to also change the shape of the data which can be inefficient.

8.3.3 Performance

The results shown in Figure 4 are satisfactory, given the problem, as the model is able to learn the underlying patterns within the time series. However that predicted time series was only one out of 64 and it was the best one. Different time series of the same run varied a lot in accuracy and in general the results were not as consistent. This points towards training data being insufficient to capture all representative patterns. However, considering that there are already a lot of known improvements to make, results should improve in future with larger training sets. Also since the training was relatively fast a lot of different approaches can be tried without having to invest too much time.

Figure 5 is an example of one of the time series that the model didn't do so well on (not the worst one though). However, when taking the stochasticity into account and viewing the results from a different perspective as in Figure 6 the results does not look as bad. Considering even some of the real data points were outside the standard deviation of those values in time series generated with the same set of parameters it's not surprising that the model had a hard time predicting those values. The fact that it usually predicts the values inside the standard deviation is promising. Some values being outside std dev bars also points towards the fact that the model is not overfitting. Certainly ten runs might not be sufficient to get an accurate measure of the standard deviation and it was only done for one of the time series, but it gives a pointer of the challenges that the model faced.

9 Contributions

- First effort towards training recurrent networks in the context of stochastic reaction networks, to the best of authors' knowledge.
- The usual time series learning framework includes a single time series to be learned, given training points at different time-steps. In this problem, each training sample was a distinct time series. Therefore, the training setting was not the general time series formulation. This necessitated the detailed pre-processing performed in order to transform training data into a convenient learning format, and also fine tune network architectures for the considered setting.
- The prediction times achieved were very fast, and met the goal of obtaining a very fast approximation of the simulator based on the Gillespie algorithm.

10 References and Links

References

[1] Daniel T. Gillespie "Exact stochastic simulation of coupled chemical reactions" The Journal of Physical Chemistry 1977 81 (25), 2340-2361
DOI: 10.1021/j100540a008

[2] Keras documentation https://keras.io (2018-01-22)

[3] Tyson oscillator example https://github.com/JohnAbel/gillespy/blob/master/examples/tyson_oscillator.py (2018-01-22)

[4] Figure 1 Illustration of an LSTM network
ChristianHerta "LSTM with forget gate and peephole connections"
http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/LSTM.php (2018-01-27)

[5] Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/ (2018-01-22)

[6] Haşim Sak, Andrew Senior, Françoise Beaufays "Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling" Google, USA

[7] Figure 2: https://se.mathworks.com/help/nnet/ref/narxnet.html (30-12-17 14:50)

[8] M.T. Hagan, M.B. Menhaj "Training feedforward networks with the Marquardt algorithm" IEEE Transactions on Neural Networks 1994 5 (6) p. 989-993

[9] NARX Guide https://se.mathworks.com/help/nnet/ug/design-time-series-narx-feedback-neural-networks.html

[10] The project Github page https://github.com/simonsh94/BBTSM (2018-01-22)

[11] Novák, Béla, and John J. Tyson. "Design principles of biochemical oscillators." Nature reviews Molecular cell biology9.12 (2008): 981.
APA

11 Appendix

The code for the LSTM and NARX models and the slightly modified Tyson oscillator is available on Github. [10]

11.1 Tyson Oscillator

Mathematical formula for the degradation of Y in a Tyson oscillator:
$\frac{dY}{dt} = k_1 S \frac{K_d^p}{K_d^p + Y^p} - k_2 E_T \frac{Y}{K_m + Y}$,
where the first term is protein synthesis rate and the second is its rate of degradation. The S is a signal to which the rate of synthesis is proportional, multiplied by a factor, $K_d^p/(K_d^p + Y^p)$, expressing the extent to which Y down-regulates gene transcription. K_d is the dissociation constant for binding of Y to the up-stream regulatory sequence of the gene, and p is an integer indicating the length of the oligomer Y when binded to the DNA sequence. The rate constant k_1 is the rate of synthesis of Y when the concentration of Y is small and the gene is fully expressed. E is a protease that degrades Y, its turnover rate is k_2 and its Michaelis constant is K_m. [11]