## Abstract

In which we present ongoing discussions, present some autoregressive modeling, and begin studies of gaussian processes.

## 1 Introduction

Consider a space of variables $\mathcal{P} \subset \mathbb{R}^{|\mathcal{P}|}$ that are settings, $\mathcal{Q} \subset \mathbb{R}^{|\mathcal{Q}|}$ that are observations, and $\mathcal{E} = \{0,1\}^{256}$ that are indicators of event $\mathcal{E}_i, i \in \{1, 2, \dots, 256\}$ occuring. We are working on a problem with some input parameters $\vec{p} \in \mathbb{R}^{m_p}$, $\vec{q} \in \mathbb{R}^{m_q}$, and $\vec{E} \in \{0,1\}^{256}$, where $m_p, m_q \in \mathbb{N}$ may not necessarily be equal. Parameters $\vec{p} \in \mathbb{R}^m$ are real-valued, independent variables specifying settings for a machine; $\vec{q} \in \mathbb{R}^{m_q}$ are observable variables; and $\vec{E}$ is a binary vector mapping to events, where a value of 1 means an event is on and 0 off[1].We hope to learn a function $f$:

$$f : (\mathbb{R}^m \times \{0,1\}^{256})^T \to \mathbb{R},$$

where $m = m_p + m_q$, to predict some observable $q_i$, $i \in \{1, 2, \dots, m_q\}$ at future time steps from some $T$ observations. In particular, we let $q_0$ be the primary quantity of interest ($q_0 = \Delta V$).

For now, we have a large dataset over time of parameters $\vec{p}$ and $\vec{q}$. These samples are in discrete time, and for the remainder of this paper we let $t$ index the (discrete) time, so that $\vec{p}_t, \vec{q}_t, \vec{E}_t$ refer to the parameters at time $0 \leq t < T < \infty$.

Ultimately, we hope to determine the input space $\mathcal{X} \subset \mathcal{P} \cup \mathcal{Q} \cup \mathcal{E}$, so that we may refer to our samples as $\mathbf{x} \in \mathcal{X}$ and approximate some function $f$ where $f(\mathbf{x}) = y \in \mathbb{R}$. We have yet to specify precisely which variables should be chosen. We are in the process of obtaining the data $\vec{E}_t$, and we are still in the process of selecting parameters from $\mathcal{P}$ and $\mathcal{Q}$.

Note that parameters from $\mathcal{Q}$ are observations, so they may not be given in real-time to predict a parameter in it. Therefore, one would first specify that no $\vec{q}$ may enter the learning problem as an input. However, the parameters $\vec{p}$ are specified settings and are rarely changed, so that the data one generates over time of these inputs is only a rank 1 matrix. One can represent this matrix by a single row vector; it is therefore singular, and so we have problems with learning techniques that require full rank such as kernel methods.

To cut out kernel methods would eliminate a wide variety of robust methods for learning, so that is not a route we hope to take. Instead, we might predict other parameters in $\mathcal{Q}$ and use them to inform a model to predict the primary quantity of interest.

To understand if time series analysis is even required, or if the problem is merely static, we use `pandas`' `lag_plot` function to see the time dependence on a few of the variables of interest: two observation variables `B:IMINER` and `B:VIMIN`, and a setting `B_VIMIN`. We show the results in Figure 1. If the data were truly random, there would be no pattern showing dependence of the two. One can see clearly that the variables `B:IMINER` and `B:VIMIN` display linear time dependence; interestingly, the observation variables `B:VIMIN` seems to be reset and evolves linearly for some clusters of data. Therefore, the plots are compelling for studying different types of time series models.

[1]Let $i \in \{0, 1, m_p - 1\}$ and $j \in \{0, 1, \dots, m_q - 1\}$. We abuse notation and let $p_i \in \vec{p}$ or $q_j \in \vec{q}$ refer to both the real-valued variable it represents and what it represents. i.e. $q_0$ is the value $q_0$ takes and what it represents $\Delta V$.

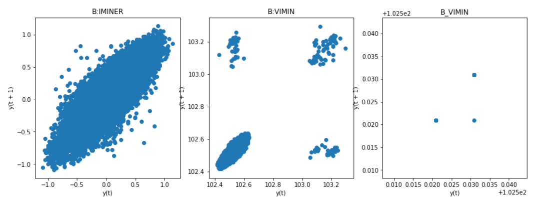


Figure 1: Lag plot of quantities of interest. The x-axis is the data sampled at time $t$ and the y-axis is the observed state at time $t + 1$.

### 1.1 Ongoing Discussions

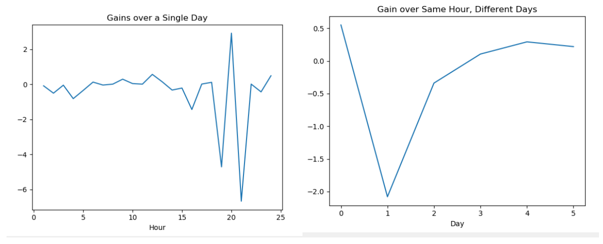
As shown in Figure 1, there is certainly time dependence on the data. One important aspect to learn is how the change in setting `B_VIMIN` relates to the change in `B:VIMIN` and `B:IMINER`. For example, the middle and right-most panel of Figure 1 are `B:VIMIN` and `B_VIMIN` respectively. Is there a correlation in time between the clusters seen in the middle plot and the change in the setting shown in `B_VIMIN`? Can we predict `B:VIMIN` based on `B_VIMIN`? Importantly, can we predict `B:IMINER` from `B_VIMIN`?

Originally, the relationship

$$\texttt{B:IMINER} = \gamma\ [\texttt{B\_VIMIN} - \texttt{B:VIMIN}], \qquad (1)$$

where $\gamma \in \mathbb{R}$ is some fixed constant (the "gain"). With that in mind, we experimented with a one-hour dataset and found $\gamma \approx 0.055$. Then, we tested 24 one-hour datasets over a single day and six one-hour data sets over multiple days. We obtained conflicting, oscillatory results; a plot of the two are shown in Figure 2

Figure 2: Hourly-binned Gain Plots for Function (1)



(a) Same Day (b) Multiple Days, Same Hour

After discussions with Bill Pellico, who controls this part of the process, we found the following. First, let $\vec{p}_t$

refer to independent parameters that are set and $\vec{q}_t$ refer to the parameters that are observations at time $t$. The settings B_VIMIN and B_VIMAX are stored in $\vec{p}$, and likewise B:VIMIN and B:VIMAX are stored in $\vec{q}$. The multipler $\gamma$ in (1) is dependent on the events in the Time Line Generator (TLG). In particular, at time $t$, $\gamma$ is dependent on the instanteous cost at some time $t$:

$$C_t^0(\vec{p}_t, \vec{q}_t) = \texttt{B\_VIMIN}_t - \texttt{B:VIMIN}_t, \tag{2}$$

and a longer-term cost:

$$C_t^1(\vec{p}_t, \vec{p}_{t-1}, \ldots, \vec{p}_{t-T}, \vec{q}_t, \vec{q}_{t-1}, \ldots, \vec{q}_{t-T}) = \sum_{\tau=t}^{t-T} \left[ \frac{\texttt{B\_VIMIN}_\tau - \texttt{B:VIMIN}_\tau}{2} + \frac{\texttt{B\_VIMAX}_\tau - \texttt{B:VIMAX}_\tau}{2} \right] \tag{3}$$

i.e.

$$\gamma \equiv \gamma_t^T(\vec{p}_t, \vec{p}_{t-1}, \ldots, \vec{p}_{t-T}, \vec{q}_t, \vec{q}_{t-1}, \ldots, \vec{q}_{t-T}),$$

where the dependence of the parameters is some function of the instantaneous cost (2) and a long-term cost 3. Can we learn this relationship using neural networks? Currently, the $T$ in (3) is unknown, but Bill could check the code for it. Once it is known, we can perhaps learn the relationship between the costs and B:IMINER by learning $\gamma$ as a function of the parameters (both the settings and the observations over time).

## 2 AR Models

### 2.1 Introduction

Autoregressive (AR) models are a form of linear regression models where the time series is regressed on previous time steps. Suppose we have a discrete time series $\{\mathbf{x}_t\}_{t=0}^T$, and we hope to predict $\mathbf{x}_T$. Consider the model

$$\mathbf{x}_T = \alpha_0 + \alpha_1 \mathbf{x}_{T-1}.$$

This model is an AR(1), model, where the argument 1, the *order* of the model, indicates the most distant previous time step used in the model. Generally speaking, for $0 < T' < T$

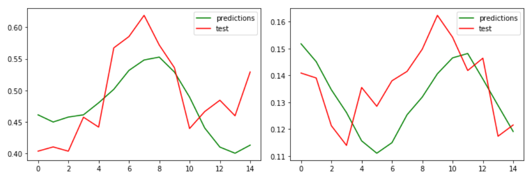
$$\mathbf{x}_T = \alpha_0 + \sum_{k=1}^{T'} \alpha_j \mathbf{x}_{T-k}$$

is an AR(T') model of order T'.

There are several advantages to using AR models. The coefficient of correlation between data of two different time steps, known as the autocorrelation function, gives a quick measure of the linear relationship between observations at different times. The difference in the time steps is known as the *lag*. One can use the lag in the autocorrelation function to deduce the optimal order of an AR model.

In addition to the nice, though simple, properties of the model, there are many computer programming (coding) packages available to implement an AR model and

Figure 3: AR Models for B:IMINER and B:VIMIN



(a) B:IMINER 1 second Prediction MSE: 3.2e-3 (b) B:VIMIN 1 second Prediction MSE 2.0e-4

optimize an AR model for the dataset. The python packages pandas and scipy may be used to build models and to predict a future time series over a specified time window. In the following section, we show some results over a one-hour time period of data.

### 2.2 Results

In this section, we show the results of some experiments with a 60-minute window of samples. We fit an AR model using scipy's AR package. Qualitative results are shown in Figure 3.

Our first goal is to predict a cycle – one second, or 15 time steps – in advance. We have a total of 48108 time steps of data to train it and, as a test, save the final 15 pieces of data to serve as a test. The prediction for the final 15 time steps is plotted alongside the observed values. For both models, the optimal lag computed and used was 56. The training for B:VIMIN was an order of magnitude closer to the true result compared with B:IMINER, which is a derived quantity and dependent on B:VIMIN.

An important aspect of this experimentation is to note the smoothness of the predictions versus the true observation. The autoregressive model fails to capture the noise and instead plots a smoothed curve. Since the data we deal with is highly noisy, these models might not be the best choice for modeling.

## 3 Gaussian Processes

### 3.1 Introduction

Gaussian Processes are a Bayesian method that combines the nice properties of Gaussian functions and the concreteness of regression with the richness of kernel methods. In Bayesan linear regression, one assumes a prior distribution over parameters and outputs a probability distribution over possible outputs. Gaussian processes generalize the output to a random functions– an infinite dimensional space.

Let $\{f(\mathbf{x}) : \mathbf{x} \in \mathcal{X}\}$ be a collection of random variables. A collection of such variables is said to be drawn from a Gaussian process with mean $m : \mathcal{X} \to \mathbb{R}$ and covariance $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ if for any finite input set $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m \in$

$\mathcal{X}$, the associated set $f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_m)$ satisfies:

$$\begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_m) \end{bmatrix} \sim \mathcal{N}\left( \begin{bmatrix} m(\mathbf{x}_1) \\ \vdots \\ m(\mathbf{x}_m) \end{bmatrix}, \begin{bmatrix} \boldsymbol{K}(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \boldsymbol{K}(\mathbf{x}_1, \mathbf{x}_m) \\ \vdots & \ddots & \vdots \\ \boldsymbol{K}(\mathbf{x}_m, \mathbf{x}_1) & \cdots & \boldsymbol{K}(\mathbf{x}_m, \mathbf{x}_m) \end{bmatrix} \right)$$

The covariance function may be any function that satisfies Mercer's conditions. In the discrete case, if we let $K$ refer to the matrix of $k(\cdot, \cdot)$ values specified above, any kernel function statisfying Mercer's condition is positive semidefinite.

A well-known, popular kernel is the gaussian kernel:

$$\boldsymbol{K}(\mathbf{x}, \mathbf{x}'; \sigma) = \exp\left( -\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2} \right),$$

where $\sigma \in \mathbb{R}$. We will use this for now in our preliminary experiments.

Python has several packages for Gaussian Processes. We use a nice package `GPFlow`, with a backend in Tensorflow, that we use for our experiments.

## 3.2 Results

We use the same data set as before, except now we only hope to train the model to predict values of `B:IMINER`. We use the gaussian kernel, as described above, and use `gpflow` to build a model [3]. A qualitative view of the results on this preliminary experiment are shown in Figure 4.

## 3.3 Discussion for Time Series

With Bayesian inference, we're building a probability distribution over functions

$$p(\boldsymbol{f}(\mathbf{x})) = \mathcal{N}(\boldsymbol{m}(\mathbf{x}), \boldsymbol{K}(\mathbf{x}, \mathbf{x})),$$

where $\boldsymbol{f}(\mathbf{x}) = \{f(\mathbf{x}_1), f(\mathbf{x}_2), \ldots, f(\mathbf{x}_m)\}$ are dependent function values evaluated at $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$. $m(\mathbf{x})$ is a mean function and $\boldsymbol{K}(\mathbf{x}, \mathbf{x})$ is our covariance matrix. If we assume that the data has some random, uncorrelated noise from sample to sample, then the variance:

$$\mathbb{V}(\mathbf{x}) = \boldsymbol{K}(\mathbf{x}, \mathbf{x}) + \sigma^2 \boldsymbol{I},$$

where $\boldsymbol{I}$ is the identity matrix and $\sigma^2$ is a hyperparameter of the variance of the added noise.

Now, if we seek to use a GP over some test sample $(\mathbf{x}_*, y_*)$, then we begin with the training data and then combine with the test set:

$$p\left( \begin{bmatrix} \mathbf{y} \\ y_* \end{bmatrix} \right) = \mathcal{N}\left( \begin{bmatrix} \boldsymbol{m}(\mathbf{x}) \\ m(x_*) \end{bmatrix} \begin{bmatrix} \boldsymbol{K}(\mathbf{x}, \mathbf{x}) & \boldsymbol{K}(\mathbf{x}, x_*) \\ \boldsymbol{K}(x_*, \mathbf{x}) & k(x_*, x_*) \end{bmatrix} \right)$$

Then, we may deduce the posterior distribution over $y_*$ is Gaussian with mean and covariance given by

$$\begin{aligned} m_* &= m(x_*) + K(x_*, \mathbf{x}) \boldsymbol{K}(\mathbf{x}, \mathbf{x})^{-1} (\boldsymbol{f}(\mathbf{x}) - m(\mathbf{x})) \\ \sigma_*^2 &= K(x_*, x_*) - \boldsymbol{K}(x_*, \mathbf{x}) \boldsymbol{K}(\mathbf{x}, \mathbf{x})^{-1} K(x_*, \mathbf{x})^T, \end{aligned}$$

where the superscript $T$ denotes the transpose.

We then extend this for a set of input outside our observations $\mathbf{x}_*$ to obtain a posterior distribution over $\boldsymbol{f}_* \equiv \boldsymbol{f}(\mathbf{x}_*)$ described by:

$$p(\boldsymbol{f}_*) = \mathcal{N}(\boldsymbol{m}_*, \boldsymbol{C}_*),$$

where

$$\begin{aligned} \boldsymbol{m}_* &= \boldsymbol{m}(\mathbf{x}_*) + \boldsymbol{K}(\mathbf{x}_*, \mathbf{x}) \boldsymbol{K}(\mathbf{x}, \mathbf{x})^{-1} (\boldsymbol{f}(\mathbf{x}) - m(\mathbf{x})) \\ \boldsymbol{C}_* &= \boldsymbol{K}(\mathbf{x}_*, \mathbf{x}_*) - \boldsymbol{K}(\mathbf{x}_*, \mathbf{x}) \boldsymbol{K}(\mathbf{x}, \mathbf{x})^{-1} \boldsymbol{K}(\mathbf{x}, x_*)^T. \end{aligned}$$

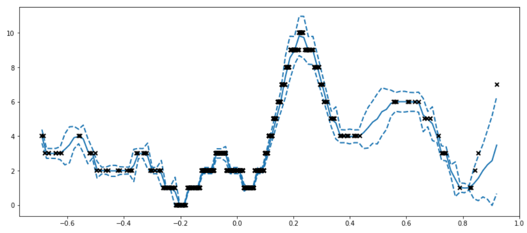


Figure 4: Gaussian Process Model of Predicted Values (black x) within a window of two standard deviations from the expected value.

Note, however, as it stands only observational data changes. The variables that are settings remain constant, and so we don't have varying input to be able to learn observational output. We can make predictions using observational data to train, but at test time we cannot rely on parameters that are observations. In the expression for the probability of the new output $\boldsymbol{m}_*$, there is explicit dependence on the input vector $\mathbf{x}_*$– requiring only independent variables, not observations. One possible method to overcome this problem is to forecast the observations, if they are more stable, and then use them as input into the Gaussian Process model to forecast the quantity of interest.

# 4 Recurrent Neural Networks

## 4.1 Introduction

Recurrent Neural Networks (RNNs) are a technology first introduced in the 80s that is a form of neural network for processing a sequence of values. Two main distinctions between typical feedforward networks and RNNs are parameter sharing and cycles. For example, consider the dynamical system

$$\mathbf{x}_t = f(\mathbf{x}_{t-1}; \boldsymbol{\theta}), \tag{4}$$

where $\mathbf{x}_t$ is the state at time $t$, $\boldsymbol{\theta}$ a set of parameters, and $f$ is a fixed state-transition function dependent on $\boldsymbol{\theta}$. Equation (4) is an example of a constraint used to define an RNN. Usually, for every input $\mathbf{x}_t$, one has a "hidden unit" $\boldsymbol{h}_t$ which is passed forward in time so that (4) becomes:

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \mathbf{x}_t; \boldsymbol{\theta}). \tag{5}$$

In this way, one can understand the network as cycling through units $\boldsymbol{h}_t$ for $t = 1, 2, \ldots,$ according to $f$. Importantly, by this framing of the problem, the learned parameters $\boldsymbol{\theta}$ is of fixed size. The function $f$ may change with time, but the primary feature of RNNs is that these state-transition functions always operate on a fixed domain and with the same parameters $\boldsymbol{\theta}$.

Note that this model is known as a latent function modeling; $f$ can be thought of as summarizing the information about previous states, since $\boldsymbol{h}_{t-1}$ is a function of all previous states in the same way that $\boldsymbol{h}_t$ is.

While RNNs are defined in such a way to handle sequential data, there are several limitations of RNNs. In particular, as more data is used (i.e. longer time series), the network becomes deeper, and the problem of vanishing gradients rears its head. In some cases, exploding gradients arises. One way to manage this would be to bin the sampled data for a fixed, shorter length and train the model over batches of the time series. However, for longer series dependence, other models such as Echo State Networks and tools such as Long Short Term Memory (LSTM) have been developed to deal with the issue of vanishing gradients [2].

One can understand the recurrence equation (4) that characterize RNNs as a power method. If $f$ admits a discretization, for example, $f = \boldsymbol{A}_{\boldsymbol{\theta}}$, then one can write (5) as

$$\boldsymbol{h}_t = \boldsymbol{A}_{\boldsymbol{\theta}}^t \boldsymbol{h}_0,$$

and, furthermore, if $\boldsymbol{A}_{\boldsymbol{\theta}}$ is a full rank matrix, then it admits an eigendecomposition

$$\boldsymbol{A}_{\boldsymbol{\theta}} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T,$$

where $\boldsymbol{Q}$ is a matrix of eigenvectors and $\Lambda$ is a diagonal matrix with the eigenvalues along the diagonal. Then, raising $\boldsymbol{A}_{\boldsymbol{\theta}}$ to the $t^{th}$ power is equivalent to raising the eigenvalues to the $t^{th}$ power; for any eigenvlaue of magnitude greater than one, the states grow exponentially. Likewise, for any eigenvalue of magnitude less than one, the states decay exponentially. Moreover, if one uses gradient descent to train the system parameters $\boldsymbol{\theta}$, this eigenvalue issue with the states translates to the gradients. This problem of stabilty is not unique to RNNs but certainly very visible.

People have tried several methods to overcome the problems of vanishing and exploding gradients, and one implemented in tensorflow and used quite broadly is LSTM, Long Short-term Memory. Using a series of gates in the form of "forget" neurons and output neurons, LSTM stores information to be used later on for quick access. This method addresses the vanishing gradient problem in many cases, and is quite powerful.

## 4.2 Discussion for GMPS

As of now, I have implemented a very basic, vanilla RNN with LTSTM to train a network to forecast the quantities B_VIMIN - `B:VIMIN` and `B:IMINER`. It is still a work in progress, and there's a lot more I'd like to address.

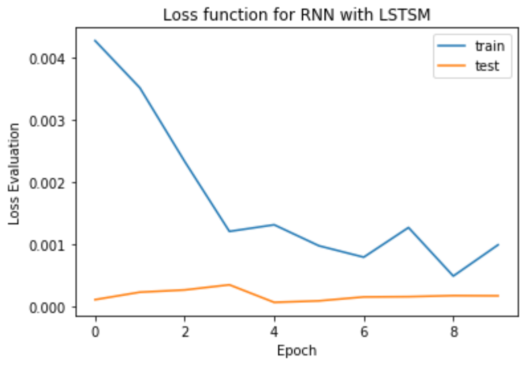


Figure 5: Example Loss for RNN with LSTM.

Figure 5 is an example of beginning, basic implementation of an RNN for B_VIMIN - `B:VIMIN`. The model was trained over 10 epochs of 1000 samples from a dataset of approximately 250,000 data points. Interestingly, the model performs better on the test than the training data.

For now we're experimenting with the loss used. Since the data is small (1e-1 to 1e-4), mean squared logarithmic error would yield more accurate results than mean squared error, as the losses would be magnified. Indeed, preliminary experiments are showing a gain in mean-squared-error on a test prediction window an order of magnitude greater with the mean squared logarithmic loss. These results, though, are dependent on a number of variables in the model and optimization framework.

Regarding the training process, one can see larger errors from increasing the number of batch sizes; this is a common problem of overfitting. One interesting aspect of the RNNs is that increasing the number of epochs past a certain amount doesn't seem to improve or impair the model; the error oscillates between a small and smaller amount, an indicator of convergence. It's unclear as of yet to me though, if the mathematics exists to prove convergence of these models.

## 4.3 RNN or ResNet?

RNNs were first introduced by Rumelhart, Hinton, and Williams in 1985[6], and since then a large variety of learning techniques and generalizations have developed. One in particular is called Residual Neural Networks, or ResNet, which may be viewed as a class of RNNs. ResNets adopt a state-transition function, as any neural network, except it allows a "skip" in the sense that an identity mapping is applied to the previous state in addition to some nonlinear function $f$. That is, ResNet has the dynamics:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + f(\mathbf{x}_t; \boldsymbol{\theta}_t), \tag{6}$$

where here we note that the parameters $\boldsymbol{\theta}$ are relaxed to allow different parameters to be learned with each step, hence $\boldsymbol{\theta}_t$. Wonderfully, this expression (6) is a scaled version of Euler's step. In particular, if we let $\eta = \Delta t$, then

(6) may be written:

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \Delta t \frac{f(\mathbf{x}_t; \boldsymbol{\theta}_t)}{\eta}. \tag{7}$$

One can understand (7) in continous time $t \in [0, \infty)$ as a first-order solution to the differential equation

$$\begin{aligned} \dot{\mathbf{x}} &= f(\mathbf{x})/\eta \\ \mathbf{x}(0) &= \mathbf{x}_0. \end{aligned} \tag{8}$$

That is, if one were to take the first-order Taylor expansion of $\mathbf{x}$ for time steps $\Delta t$ apart, and assuming $f$ differentiable, one would obtain (7) with error in big-O notation: $\mathcal{O}(\max\{(\Delta t)^2, f'\})$.

## References

[1] Do, Chuong and Lee, Honglak. *Gaussian processes.* Course Lecture Notes. Stanford University, Spring 2019. `http://cs229.stanford.edu/section/cs229-gaussian_processes.pdf`.

[2] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.

[3] GPFlow. `https://gpflow.readthedocs.io/en/latest/notebooks/ordinal.html`

[4] Rasmussen, Carl Edward. "Gaussian processes in machine learning." Summer School on Machine Learning. Springer, Berlin, Heidelberg, 2003.

[5] Roberts, Stephen, et al. "Gaussian processes for time-series modelling." Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 371.1984 (2013): 20110550.

[6] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. No. ICS-8506. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.