Learning First two moments from Model's parameters

February 9, 2023

Riccardo La Marca

Abstract

The goal of this project is to build a *Neural Network* that is able to predict the first two moments, *mean* and *variance*, of a model's outputs given just its fixed parameters. Essentially, learn the relationship between model parameters and the first two moments of its outputs. The code of this project is public on GitHub at https://github.com/lmriccardo/moments-learning

1. Introduction

In time series analysis and modeling knowing the underlying properties and behaviours of a time series is very important. Two main features are the **mean** and the **variance**, i.e. the first two moments. Whenever we already have the model that describe the time series and the input are fixes, we would like to know the relationship between the model's parameters and the first two moments of its outputs. To address this problem, *biological* models have been used as examples.

Biological models, in terms of SBML, are composed of: parameters, species, compartments, reactions, events and other types of rules like assignment and algebraic ones. In a general way, biological models can be considered without any doubt a continuous multivariate time series models by the fact that the evolution in time of each specie is described by a differential equation, and in total the entire model describe an ODE system.

Once we have the model, we can simulate it to a fixed horizon and so obtaining the trajectory, or a number of possible realizations in time, of the model. Using what we have previously produced it is possible to compute the mean and the variance of the species (or the outputs in general), thus obtaining the data that we will use to address our objective.

Email: <lamarca.1795030@studenti.uniroma1.it>.

Automatic Verification Of Intelligent Systems 2023, Sapienza University of Rome, 1st semester a.y. 2022/2023.

2. Addressing the problem

To achieve our goal we can either come up with a well formed formula, after gained a precise understanding of the underlying model, or we can use machine learning techniques, like *Neural Networks*, to learn this relationship between parameters and moments. In this case, the second approach has been used.

Before keep going with more technical details it is important to note that the relationship we would like to learn is not model-invariant, this means that the same neural network cannot be used simultaneously on different models, i.e., it should be trained and tested on data coming from the same model. However, we can indeed use the same neural network on n models separately, thus producing n different neural networks with different parameters for each layer.

2.1. The Neural Network

Whenever we have to setup a Neural Network to solve a task, first thing first we have to decide which problem we would like to address: *regression* or *classification*. In our case we want to predict the mean and the variance, i.e., real values, thus the problem that we would like to address is a *Regression* task. So, *linear regression* will be used.

There are different Neural Network models that can be used for linear regression task, however not all of them can be used to achieve our goal. For example, if we would like to predict the next value of a time series we could use *Recurrent Neural Netowrk* (RNN). This is not the case. To keep it simple I decided to implement a simple *Deep Feed-forward Neural Network* with only linear layers and the *ReLU* activation function to implement non-linearity.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{1}$$

2.2. The optimization problem

For this kind of tasks the optimization problem to setup is a classic minimization problem, where the function that needs to be minimized is the loss function, in this case the *Mean Squared Error*

$$\mathrm{MSE}(\bar{\mathbf{y}}, \mathbf{y}) = \frac{1}{n}\|\bar{\mathbf{y}} - \mathbf{y}\|^2 \tag{2}$$

To solve this optimization problems we could use technique like SGD (Stochastic Gradient Descent) and its derivatives. In this case I decided to use an alternative called Adam: an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments.

One problem arise when there is a large difference (in order of magnitude) between parameters and outputs in the dataset, that's because obviously the mean and variance of the output depends also on the input of the model. Talking about biological models, the inputs are the initial amounts (or concentrations) of the species. This huge difference can produce in the first stage of the training very big losses, and thus huge gradients resulting in the so-called *exploding gradient* problem. Since, in this kind of optimization algorithm, the weights of the neural network are updated using the computed gradient of the loss, then, they will be affected by this problem as well. To avoid this, a bunch of technique can be used like: *weight regularization* and *gradient clipping*. In this case, after some tests, I decided to use the first one, more precisely, the so-called L_2*-regularization*, which means applying an L_2 penalty to the loss function, in other words, adding a factor $\lambda\|\theta\|^2$, where θ are the weights of the network.

Finally, the last factor that we need to handle for the optimization is the *learning rate*, the one that decide the step size of the gradient to the minimum value. Instead of doing hyper-parameter tuning I decided to use an adaptive learning rate that changes depending on how much the network is learning or not. Whenever the scheduler detect that the network learning stagnates, then the learning rate is reduced by a certain factor, up to a minimum learning rate.

3. Experimental Results

Once having decided which the problem is and the way I decided to address it, it is time to show the experimental results that I obtained. The same neural network has been trained on the first 10 biological model taken from the BioModels Database, all of them having different number of parameters, parameters values and magnitudes, different reactions and species. These differences are important to show the effectiveness of the neural network learning. First of all we need to setup the dataset.

3.1. The Dataset

Given a biological model in the SBML form I simulated it using COPASI with an horizon of 100 and step size 0.01 obtaining in this way a trajectory. Given the dense output I computed the means and the variances of the output species and saved them into a CSV along with the parameters of the model, creating a kind of mapping. Note that, whenever I use the term "parameter" I include only those model's parameters that are fixed, essentially the kinetic constants of all the reactions.

This process has been repeated 2500 times changing every time the values of the model parameters but maintaining the initial amount (or concentration) of the species fixed. In order to maintain a kind of "sense" in the values of the parameters, they have not been chosen completely at random. Instead, starting from their original value p, each new parameter has been selected at randomly from a ball $\mathcal{B}_r(p)$ centered at p with radius $r = p/2$.

At the end of the process I obtained a dataset $(\mathbf{X}, \mathbf{Y})$ where $\mathbf{X} \in \mathbb{R}^{2500\times N}$ are the parameters (the input values) and $\mathbf{Y} \in \mathbb{R}^{2500\times 2\times M}$ are the first two moments (the output values, M is the number of species). In the last step, the matrix $\mathbf{X}$ has been standardized to have 0 mean and standard deviation 1 to improve the performance during the learning process. Finally, the dataset has been splitted into **train** and **test** set with a respective ratio of 0.8 and 0.2, i.e., 2000 sample for train and 500 for testing.

3.2. Neural Network, Optimizer and LR Scheduler

Once we have the dataset, we need to implement the structure of the Neural Network. As I already said, in this case I decided to implement a simple *Deep Feed-forward Neural Network* with: 1 input linear layer, 4 hidden layer with 50 neurons each, 2 hidden layer with 30 neurons each and 1 output layer, for a total of

$$\begin{aligned} i_{size} \times 50 + 4 \times 50^2 + 2 \times 30^2 + 30 \times o_{size} \\ + 5 \times 50 + 2 \times 30 + o_{size} \end{aligned}$$

learnable parameters, where i_{size} is the number of input of features and o_{size} is the number of output of features. As already being said, the activation function for each layer, except for the last one, is the ReLU.

For the optimization algorithm I choose Adam with an initial learning rate of 1.0e-4 and weight decay of 1e-04 as well (the weight decay is the the L_2 penalty factor λ).

Regarding the LR scheduler, I decided to use the so-called ReduceLROnPlateau which reduces the learning rate by a certain factor, I used 0.5, after a number of stagnation epoch, in this case 5 epochs, up to a minimum value, I chose 1.0e-6.

3.3. Training

Given the dataset, the neural network has been trained for 250 epochs. During each epoch *K-fold Cross Validation* has been used to increase the generalization abilities of the neural network and reduce the risk of overfitting. A total

of 5 cross fold validation iterations have been ran during each epoch. At the end of the training process both very low losses, in the order of 10^{-5} or 10^{-4}, and very high accuracy, $90 - 99$ % have been reached. The picture below shows an example of improvement during the training stage.

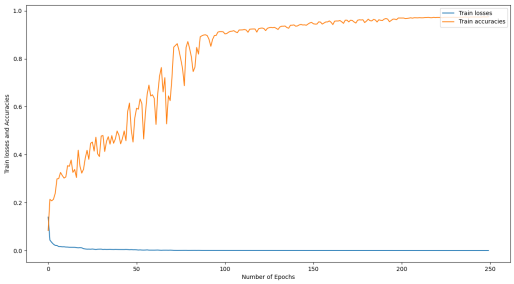

Figure 1. Train loss (blue) and accuracy (orange) during epochs of the biomodel BIOMD00004 (Goldbeter1991 - Min Mit Oscil, Expl Inact)

In this case the final train accuracy ends to be 99.08 with a train loss of 7.48e-5. Moreover, we can also see the effectiveness of the LR scheduler: at some point during the training the learning rate has been reduced in way such that the accuracy and the loss became less oscillating thus improving the learning of the neural network. The side effect is that the learning is actually slower, but more precise. All the images for training losses and accuracies over epochs can be found in the repository given in the abstract.

3.4. Testing

In the test stage the neural network is feed with new samples that it didn't see during the training phase. Once obtained the output we can compute the accuracy. The table 1 shows the results for all the 10 models.

Models	Test Accuracies
BIOMD00001	95.67 %
BIOMD00002	95.17 %
BIOMD00003	88.26 %
BIOMD00004	87.59 %
BIOMD00005	92.25 %
BIOMD00006	95.16 %
BIOMD00007	84.66 %
BIOMD00008	86.05 %
BIOMD00009	88.49 %
BIOMD00010	90.09 %

Table 1. Test accuracies for the 10 models

4. Reversing The Problem

Until now I have shown a simple method to build a Neural Network from scratch that is actually able to learn the relationship between model parameters and the first two moments. One could be asking: *Is this a One-Way Relationship*? It depends on the underlying structure of the model and on the nature of it (deterministic or stochastic). Although it depends on these two factors, all the 10 models that I have been using to develop the project didn't shown any type of randomness, and thus it seems very hard to me that, given the same initial condition (initial values for species), with different parameters we could get same results. For this reason, let's dive more in the details on how I decide to reverse the problem.

First of all, the dataset is the same presented in 3.1, with the only difference that in this case $\mathbf{X}$ are the mean and variances, while $\mathbf{Y}$ are the parameters. Second, the neural network previously constructed does not work at all for this problem. This is not a case of course, but there are important considerations that we need to take care of, before deciding how to attack the problem.

One of the main reasons why the previous Neural Network does not work anymore, at least in this case, is due to the presence of *outliers*, that "invalidate" the assumptions at the bases of the linear regression. These outliers, in my case, are represented by some parameters with very high values, i.e., in order of magnitude of 1.0e8, which produces very high loss during the training stage, causing a very slow training or a complete stagnation. In this case there are two paths that one can follow: (1) removing outliers by applying techniques like *Inter-Quartile Range* or *Z-score*; (2) uses learning technique more robust to outliers, like *Random Forest*. I decided to use the second path.

Random forests came with a set of hyper-parameters that needs to be tuned like the number of *estimators*, i.e. the number of *decision trees*, and the *maximum depth* that each decision tree can reach. To perform hyper-parameter tuning I decided to use a Grid Search approach with Cross Validation to avoid the usual overfitting problem. Finally, after obtained the best estimator from Grid Search, I applied it to the test set obtaining the following results.

5. Conclusions

The goal of the first part of the project was to build a learning model able to predict the mean and the variance, so the first two moments, of the output of a model given its fixed parameters. To achieve this objective a simple *Deep Feedforward Neural Network* has been developed, trained and tested on 10 different datasets separately, each of them with 2500 samples, obtained from the simulations of biological models.

Models	Accuracies	Estimators	Depth
BIOMD00001	73.31 %	1200	100
BIOMD00002	73.09 %	400	150
BIOMD00003	75.25 %	1200	150
BIOMD00004	75.20 %	800	100
BIOMD00005	82.40 %	800	100
BIOMD00006	79.04 %	1200	150
BIOMD00007	74.22 %	1200	150
BIOMD00008	75.17 %	1200	50
BIOMD00009	74.07 %	1200	150
BIOMD00010	75.00 %	1200	100

Table 2. Results for the reversed problem

Despite the simplicity of the adopted approach, overall, I have obtained very good results also due to some tricks that improved the learning stage: adaptive learning rate, regularization techniques and data pre-processing. This shows how effectively the neural network was learning this relationship. During the training the model achieve very good performance reaching also the 99 % of accuracy with very small losses. This could be a sign of overfitting, but fortunately it is not, since also in the testing stage the NN shows that it actually learned how to predict.

Although the neural network didn't achieve the same results for each of the used model, this is quite normal for several reasons:

- **Complicated models**. Some models are more complicated then others, and since the relationship we would like the network to learn is indeed model-dependent, this means that also this kind of "mapping" between parameters and moments is not very simple as well
- **Same network**. In this case I decided to use the same Neural Network to train and test all the models, just changing the number of neurons on the input layer and output layer to match the dimension of the input data and the output data. Obviously, this is not the best option.

Nevertheless, these results can be improved individually in a significant way by: augmenting the dataset (more simulations); fine-tune the structure of the Neural Network depending on the size and the difficulty of the problem, or adopt strategies of *Dynamic Deep Neural Network*, in which the structure of the NN is dynamically updated depending on the results of each train step.

For the second part of the project (the reversed problem) a different approach has been used. In this case, I found out that following a neural network path was not a great choice overall, due to the problems that I have already largely discussed above. So, instead of changing the dataset, by applying some sort of transformation on the output data (for example a log transformation[1]), I decided to take this as an opportunity to try a different learning model, i.e., Random Forests. Despite the difficulty of this problem, with respect to the previous one, Random Forests have reached very good results.

[1]I actually tried to use Neural Network after applying log transformation on the output data. I have obtained good results overall, but some models had zero-valued parameters resulting in mathematical errors when log-transform.