Machine Learning Algorithms for the prediction of ASTRA Output

S. Kladov
Budker Institute of Nuclear Physics, Novosibirsk, 630090 Russia

September 07, 2021

Abstract

Particle tracking simulations play an important role in accelerator design and in the study of already operating systems. It is possible to significantly reduce the time of parameter scans using machine learning as replacement of simulation.

In this report we describe how the different machine learning algorithms were implemented to predict the output beam parameters of the Astra simulations based on the input numeric parameters – fields and initial beam properties.

The GUI program is also presented in this paper. It makes the process of training and using the machine learning algorithms convenient and simple for the user.

Contents

1. Introduction 3
2. Methods 3
2.1. Neural Network 4
2.2. Decision Tree 4
2.3. Tree Boosting (Gradient Tree Boosting) 5
2.4. Linear Models and Kernel Ridge 5
3. Datasets and Visualization 5
4. Fit Results 6
5. Automation Program 7
6. Summary 7
7. References 8

1. Introduction

Simulations are often needed during accelerator study when it is not possible to obtain an analytical solution. They are usually used to find boundaries of acceptable parameters (for example, when it is needed to investigate the dynamic aperture), and the optimization problem – searching parameters for the best solution available.

Often both of them require a plenty of time-consuming simulations, but in general, we don't need these precise results for every combination of simulation parameters. In most cases a decent estimation is enough. Therefore, it would be useful to create a software, that will be able to make these decent estimations while being faster. And for this report we do it using the Machine Learning (ML).

Our goal in this work is to make a fit of the simulation data. In the case when we know the function that will fit the data, we need only to perform usual fit with, say, squared error metrics. Otherwise, if we don't know the function, we have to use the ML. It is usually implemented in the tasks with dense datasets with significant errors. In this case Ml is trying to draw a curve somewhere in between dataset points (see Fig. 1 left).

When we have a sparse dataset every point matters. We would like our curve to go through all of these error bars. We have different possibilities to do that: we can just match all the points with lines (it will be named as 'linear fit' further) or try to draw some curve.

In our case the errors of the data points are almost zero because these are the simulations with huge number of particles, and we assume that the systematic errors are negligible as well. So the task for ML in this report is to draw a curve that go almost exactly through all of the dataset points with a decent behaviour in the gaps between them. The explaining example of such fit is shown in the Fig. 1 right.

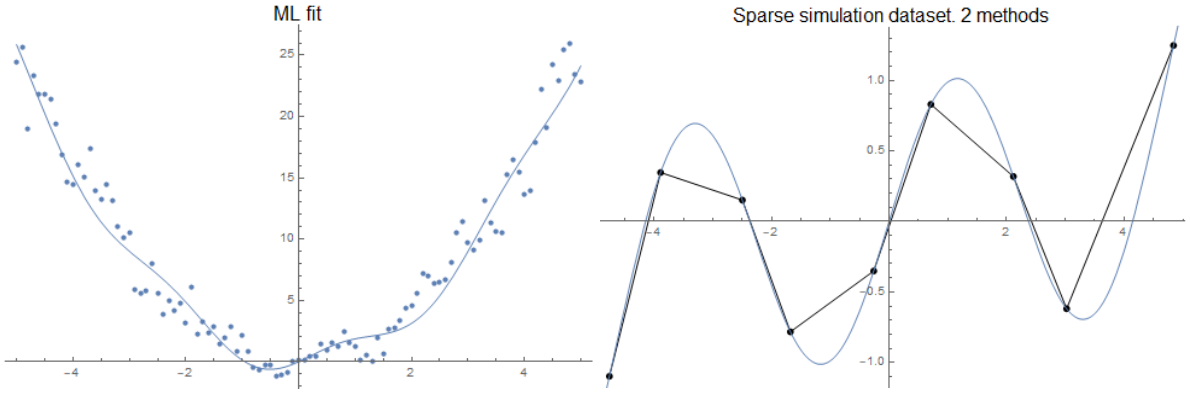

Figure 1: Usual Ml (left) and our-case (right) fit examples.

2. Methods

Machine learning includes many different algorithms suitable for different tasks. The task for ML in our case is to fit the simulation data – nonlinear dependencies of the output beam parameters on the input ones. So, the general type of ML for this problem is nonlinear regression.

The ML algorithm training performance has two extreme cases – so-called over-fitting and under-fitting. The general aim of the training is to find the gold mean between these extremes, which is very dependent of the problem. In our case this mean is close to the over-fitting case, because we usually have small datasets, where the curve (multidimensional surface), drawn exactly through all of the dataset points, is still pretty simple. We would like our ML to draw this simple surface with only requirement of considerable estimations in the gaps between and beyond the dataset points.

Among the nonlinear regression methods, we choose here the NN, decision trees, boosting algorithms, linear models with modified input vectors, kernel ridge and their combinations.

In this paper we realize these algorithms and compare their performances on the typical simulation datasets.

2.1. Neural Network

Neural network is a common choice for ML algorithm. It was used by many other groups with similar problem [1,2,3]. This algorithm has a decent variety of parameters, so the model can be adjusted to almost any problem and dataset. Also, the fitted by NN curve is smooth, but it requires some time to train, can get stuck in a local minimum, and the architecture should be adjusted to the lengths of the current input and output parameters.

General scheme of a simple NN is shown on the Fig. 2. The input vector is multiplied by the layer matrix and an activation nonlinear function is applied. Then this process is repeated for each layer. Further information can be found in [4,5].

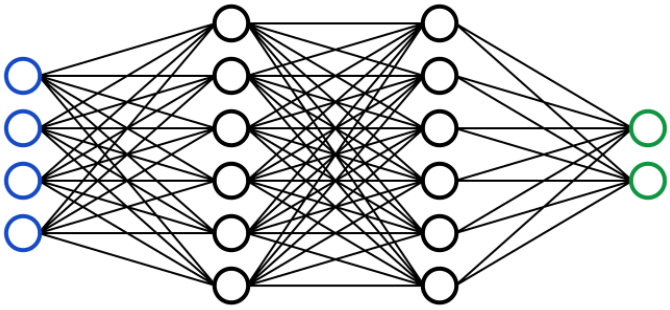

Figure 2: General block-scheme of a NN.

The architecture and the optimizer for our case were chosen based on the 3 facts:

- the surface is simple, so the number of layers should be small. More layers will significantly reduce the training speed, and the model will likely not train well at all, stuck in a local minimum;
- the number of input and output parameters can be $\gtrsim 10$, so the layer sizes should be significant. If these sizes are huge enough, the model can train each output parameter separately, but the training speed is reduced;
- we do not care about over-fitting, so the regularization is dumped to 0, and the batch normalization is turned off. As for the optimizer, we use Adam as the most common choice.

As a result, the NN architecture consists of 3 fully-connected layers with the shapes of $(input, 256), (256,1024), (1024, output)$, where the $input$ and $output$ are the dimensions of the parameters spaces, and ReLU as an activation function.

2.2. Decision Tree

The principle of the trees is quite straightforward: as always, we start from the input values. Then, on each step, we have a simple rule: if the incoming value is less than x (training parameter), then choose one of the two possibilities. As a result, we have a simple, fast-trained method, but the fitting curve is rigid. More detailed explanation of the tree models can be found in [6,7]. The method block-scheme example is shown in the Fig. 3.

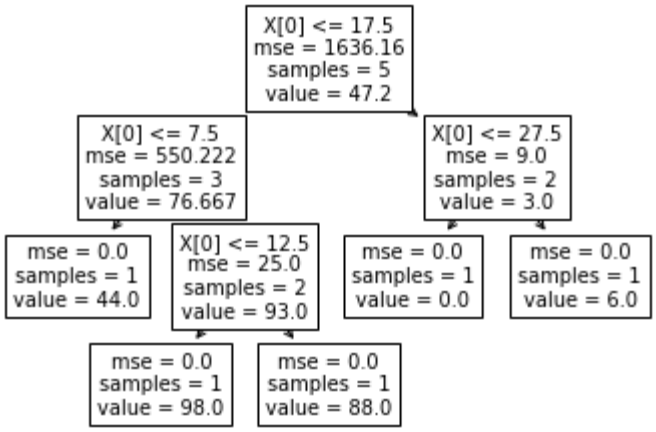

Figure 3: Block-scheme example of a Tree.

The algorithm parameter that we can change here is the deepness of the tree. As we don't care about overfitting, we can set this parameter large.

2.3. Tree Boosting (Gradient Tree Boosting)

The method is based on the trees model, but instead of training only one massive tree, this method trains the sequence of 'weak learners' – simple decision trees, and then makes a prediction through the voting system [8,9]. As a result, if the number of the weak learners is large enough, the method's resulted curve will almost coincide with the tree's one, except the case of dependencies where the output does not vary much (for example, if it is 1 ± 0.001, the boosting fit will be just 1). Complexity (training time) is larger than in the single tree case, but is still much less than the NN's one.

2.4. Linear Models and Kernel Ridge

In this report we also tried to use the Lasso, SGD and Kernel Ridge algorithms. The idea of using the linear models in nonlinear problems is following: as the first step we will modify the input vector: to the n-dimension vector we add squares of the values, cubes, multiplications of i and j input parameters and so on. Kernel ridge with the polynomial kernel, which was used in the paper, has the same work principle. These methods work perfectly fine in the case of 1-dimensional datasets, but in the 2d (and more) case, when the correlation of input parameters is significant, we need also the cross terms. This requirement significantly complicates the use of these models, and, as we have other nonlinear models, in this paper we haven't implement linear and kernel ridge models.

3. Datasets and Visualization

In this paper different simulation scans were used as datasets. The 1-, 2- and many-dimension input scans were made using two methods with grid-like and randomized sets of points. For our tests we used a model with electron gun with solenoid and booster. We mainly scanned over the initial beam charge, solenoid and electron gun electric field. The beam current scans are often useful on practice, and, in the combination with the rotating beam solenoid field variation they result in interesting dependencies, that we would like to fit.

Currently we fit 13 beam properties from the result of the tracking simulation. These are the beam energy, position, energy and transverse momenta spreads, emittances, bunch length and transverse sizes. Usually we need only the part of them: emittances – the phase space areas occupied by the beam, bunch sizes and transverse momenta spreads named alphas.

In the case of 1-dimension input space problem it is easy to plot the fit results – we can draw line plots for each output parameter.

In the case of grid-like parameter scan we can expand this idea and draw multiple sets of such pictures, where only one input parameter is variate for each set.

In the case of random 2d scan it is possible to draw a surface where z axis is the model prediction (one of the output parameters), and the x-y plane is the input parameters space.

4. Fit Results

Figure 4 shows a general NN performance on a common 2d dataset.

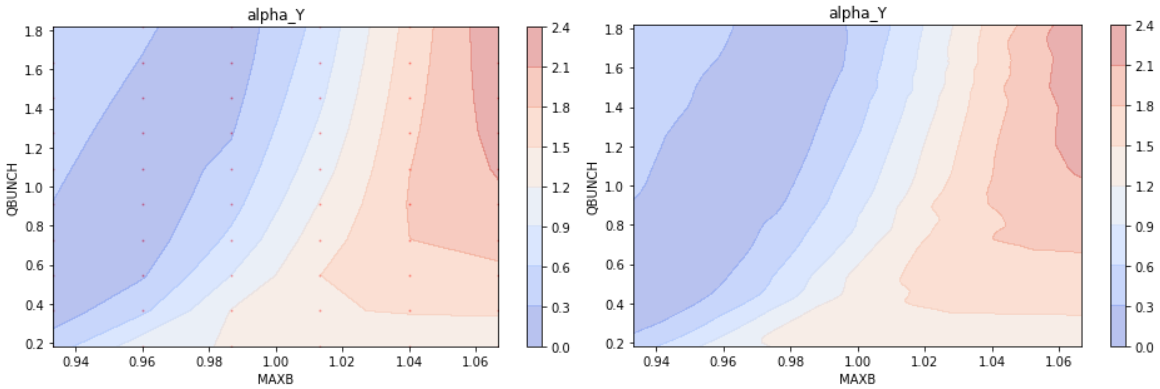

Figure 4: dependencies of alpha-y on the initial beam charge for 2d grid-like dataset. Simulation data is on the left and the NN model performance is on the right. Red dots on the first picture are the simulation data points.

The qualitative coincidence of the simulation and fit maps is clearly seen from these pictures. In order to obtain the quantitative fit characteristics the usual 1d dependencies of one of the output parameters on one of the inputs (when all other input parameters are frozen) should be plotted.

Figure 5 represents such plots as well as nonlinear models comparison. It is convenient to compare them on a sparse dataset, so that they can be easy divided in the gaps between dataset points. Because of that the number of beam charge points is 6 for the Fig. 5, while it is10 for the Fig. 4.

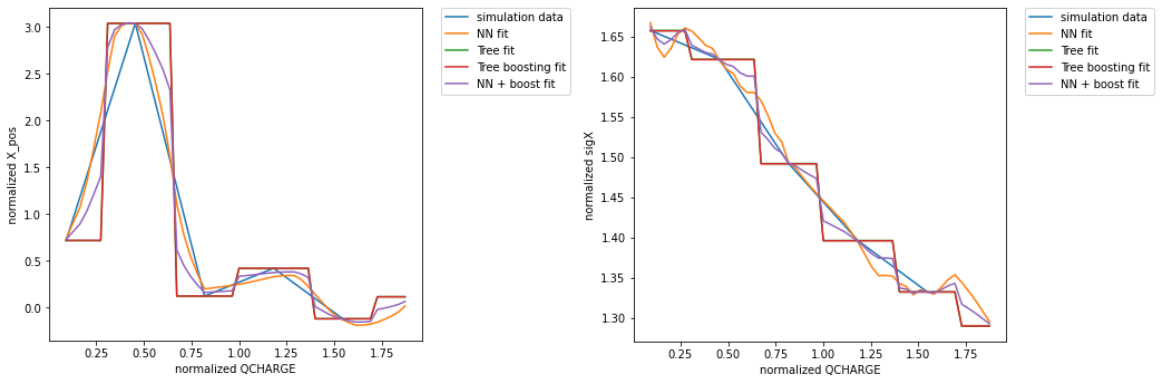

Figure 5: dependencies of the final emittances on the initial beam charge for a 3d grid-like sparse dataset. NN + Boosting is an averaging of these methods.

All the nonlinear fit curves go almost exactly through the dataset points, and it can be seen from the Fig. 5, so we haven't lost accuracy at least on these points and one of the tasks for ML is fulfilled, but the behaviour of the models in the gaps between the points and in the outer region vary much from model to model.

The differences of the models, as well as an overall performance is better seen from the "relative difference" plots. These plots show the deviations of ML fits from the defined above 'linear fit' – simple matching of dataset points with straight lines (blue curves on Fig. 5). These plots are shown on the Fig. 6. They correspond to the ones from Fig. 5.

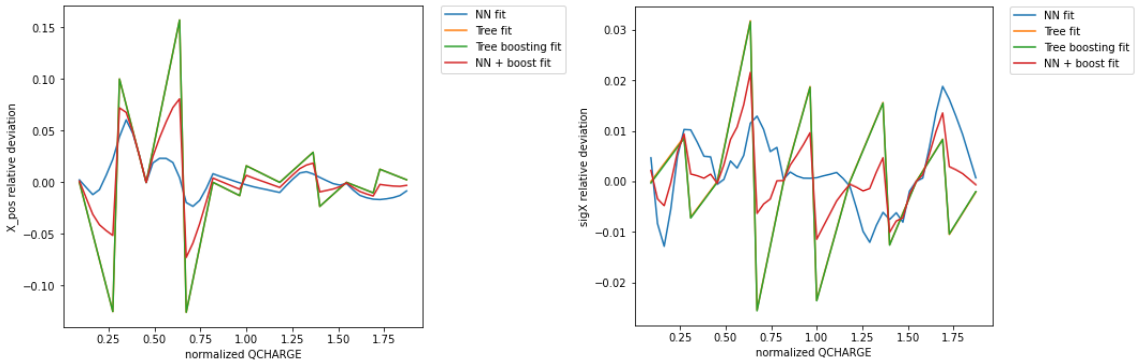

Figure 6: relative divergence between fit and simulation. Error is plotted on z axis.

The rigidity of the tree models reveals itself on the Fig. 5 and 6. Also they show the biggest relative errors up to 30%. The average errors of the best model are about 5% and are generally achieved by NN + Boosting fit (averaging of NN and Gradient Tree Boosting methods).

Anyway all the models give decent estimations in the inner region of the dataset, So the model for the particular task should be chosen manually depending on the dataset. Here we provide two most general considerations as example. In the outer region NN predictions can sharply change its direction to the wrong one, so in this case the user should use the tree models, predicting the horizontal lines (see Fig. 5). If the simulation dataset is tiny, it would be better to have a line between two points, than a step, so in this situation one would prefer to use the NN as it gives a smooth curve.

The training time of the NN model is ~1 minute, and the trainings of the tree models are almost instant (compared to the simulation process). Hence, the scan times are greatly reduced without huge losses in accuracy.

5. Automation Program

For ease of use, the GUI Java program has been written. It has two modes: one for the simulation scans to generate the training dataset, and another for the ML training, predictions and visualizing results.

The first mode allows user to choose input parameters, that he wants to scan and ranges of the scan. The scan is available in two possibilities: grid-like and random.

The second mode gives the possibility to calculate dataset from the scan ASTRA output, train different models on the dataset conveniently, and then predict the simulation results for the desired input parameters and visualize the training results as surface plots.

The details of the program and the instructions how to use it can be found in the Appendix A.

6. Summary

Though the usage of the ML algorithms requires a dataset, it can be moderately sparse, so the input parameters scan times can be significantly reduced by using the trained model instead of new simulations.

Several machine learning methods were implemented and tested on various training datasets with different dimensions, including initial beam charge, solenoid and cavity fields scans.

All the method performances were quite good and fitted the datasets well (an error in the worst case ~40% and an average performance is ~5%). Though, the peculiarities of the methods results were investigated and the model choice based on the dataset was described.

Almost all the models were packed into the Java GUI program, that, among the easy train and prediction, gives a convenient way to scan an input parameters space and prepare a dataset.

7. References

[1] F. Wang et al., “Machine learning for design optimization of storage ring nonlinear dynamics”, *arXiv:1910.14220*, 2019.

[2] J. Wan et al., “Neural network-based multiobjective optimization algorithm for nonlinear beam dynamics”, Physical Review Accelerators and Beams 23, 081601, China (2020).

[3] Y. Lu et al., “Enhancing the MOGA Optimization Process at ALS-U with Machine learning”, Lawrence Berkeley National Laboratory, CA 94720, USA (2021).

[4] Geoffrey E. Hinton, “Connectionist learning procedures”, Artificial intelligence 40.1 (1989): 185-234.

[5] https://scikit-learn.org/stable/modules/neural_networks_supervised.html

[6] L. Breiman, et al., “Classification and Regression Trees”, Wadsworth, Belmont, CA, 1984.

[7] https://scikit-learn.org/stable/modules/tree.html#tree

[8] H. Drucker, “Improving Regressors Using Boosting Techniques”. Proceedings of the 14th International Conference on Machine Learning (1997).

[9] https://scikit-learn.org/stable/modules/ensemble.html#adaboost

Appendix A: How to use the java program

The instructions for the complete input ASTRA parameters scan process are given here.

This instruction is valid only for the version that was available 08.09.2021. Please consider checking the github for updates: https://github.com/KirikaUeno/simulationMachineLearning/tree/main/runAstra

The names of files are important!

Preparing the dataset (sparse scan)

The first thing you need is to prepare the main 'run.in' file manually: it should be the usual ASTRA input file with the information about the initial beam distribution, aperture and the lattice elements. Make sure, that all the paths, specified in the 'DISTRIBUTION', 'FILE_APERTURE', 'FILE_BFIELD' and 'FILE_EFIELD' are valid, and set the values of all the static parameters (that you don't want to scan) to the desired ones (you can do the latter from the program UI later though).

Then place the 'batch.sh' file into the directory with jar file (the directory is named 'main folder' further). It should contain the 'Astra' word in the last line and have 'o_out' and 'e-out' parts in the corresponding output flow files.

In order to program work you need Java. If you do not have it install the newest version.

After that you can run the program via the command 'java -jar runAstra.jar' (or use your local Java), and window with the first mode of the program should appear. Specify the path to your main 'run.in' file in the dedicated text field (with the name of the file) – it can be the full path in your system, or the path relative to the main folder. Click load file. If all is done correctly, you will see the two tables – empty left (with parameters to scan) and filled with parameters right (with static parameters) (see Fig. A.1 left).

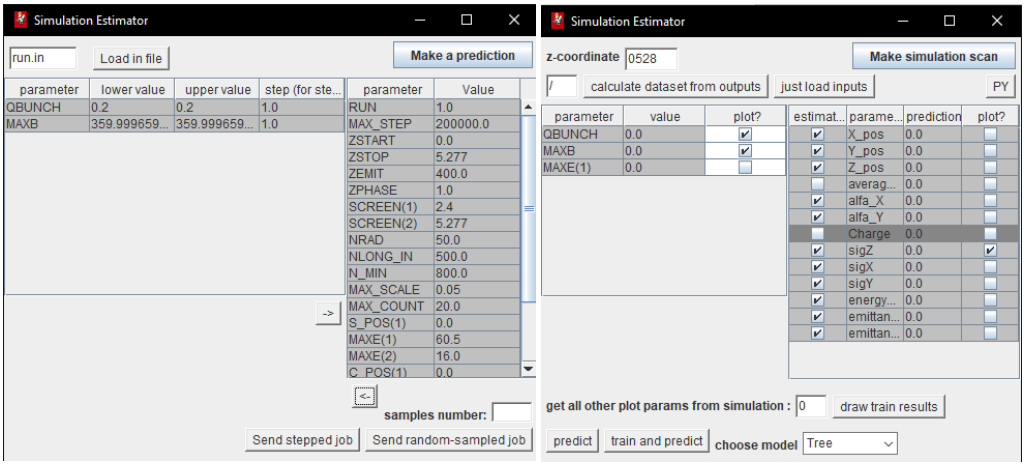

Figure A.1: The first (left) and the second (right) modes of the program.

By clicking on the '<–' button you can move selected parameters from 'static' to 'dynamic' and vice-versa with the '–>' button.

You can change the values of the static parameters if you want. Specify the ranges of the dynamic parameters variations, and the scan steps if you want to make the grid-like scan, or the number of samples (in the lower-right corner) if you want to make the random scan. Keep in mind, that 'MAXB' parameters are displayed in Amperes with the formulae of conversion valid for the electron gun (that was used for the tests in this paper) solenoid. Also make sure, that the number of simulations is not large – the DESY cluster recommends to not store over 1000 output files in one directory, and the dataset calculation from simulation output, that is performed in the second program mode can get stuck if the dataset is large (will be fixed in further updates).

At last, send the job to the cluster using one of the 2 buttons as you prefer. The command ‘qsub batch.sh’ is executed by doing this, so your main directory should have an access to the cluster.

Making predictions with ML

If your simulations were finished successfully (the output files with the shape of run1.mid-FileName.001 should appear in the min directory), launch the Java program again with the same command. Click on ‘make a prediction’ button – it will open the second mode of the program (see Fig. A.2).

This mode requires Python. If you don’t have it, just download the latest version. If you have multiple version, or just want to use your own python, you can specify the path to it by clicking on the ‘Py’ button in the upper-right corner of the program. This python cersion should have installed libraries: pytorch, scikit-learn and matplotlib (and numpy obviously).

The program needs 4 python files in the main directory, which are distributed with the main program in archive.

Specify the directory of your output files and the z-coordinate where you want to fit you data (‘midFileName’ in your output files) and click on ‘load simulation results’. The input parameters that you variated should appear in the left table. Keep your time, the program should evaluate all the output beam properties from the output files. If you have already pushed this button and have a valid ‘information.txt’ file with dataset in your main directory, you can click on the ‘just load inputs’ button: it will load the input parameters that you variated, but will not create dataset (will use ‘information.txt’).

After that you have a dataset and can use the ML.

In order to make a single prediction change the values of the scanned input parameters to desired and choose which output parameters you want to estimate.

If you use the desired type of the ML on this dataset and with these output parameters for the first time, you should train the model first, so click on ‘train and predict’ button. The predicted values should appear in the right table. If you have already trained the ML model, you can use ‘predict’ button.

If you have multidimensional (>1d) dataset, the program can visualize for you the trained model as a 3d surface plot, that you can rotate. It is helpful for ease of the best parameters searching and validation of the training. In order to do that, firstly choose 2 input parameters and 1 output parameter that you want to draw, then specify the number of simulation from which you want to take all the rest input parameters and click on draw train results (make sure that the model is chosen correctly). Note that the ‘other input parameters’ scans should be grid-like: otherwise the program will plot only one point.

The surface plot of the NN model prediction for the 2d randomized scan is shown in the Fig. A.1.

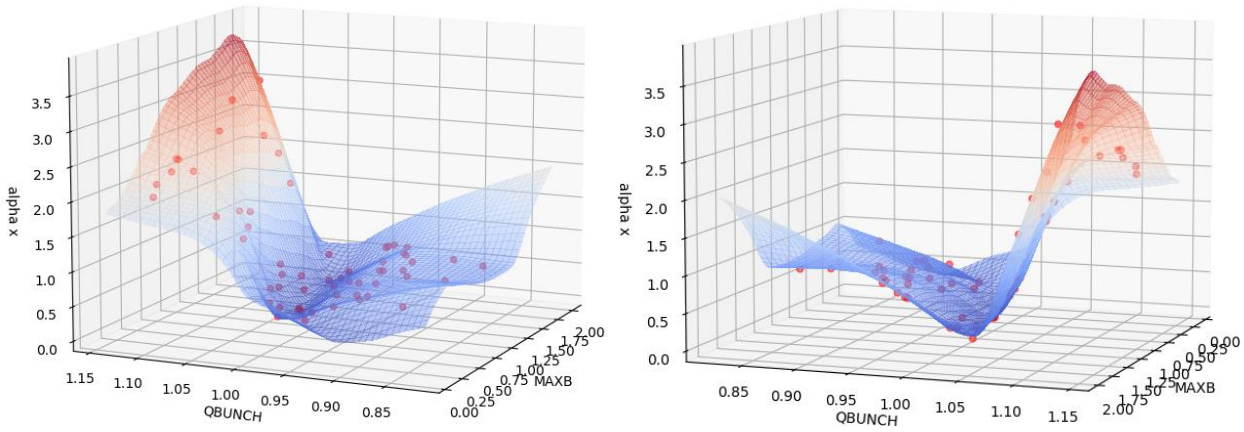

Figure A.1: NN model fit result on a 2d randomized dataset.