Neural Networks for Experiment Predictions

Adwait N. Zarbade

Abstract

This report explores the idea of using deep neural networks with analytical simulations to solve a critical topic in experimentation. Getting results from a given experiment, through simulations takes time, and if experimentation system is complex enough time taken may increase significantly. Also, there is limitations on how much computation one may have. This often results in prolonged analysis period and increased cost.
This report studies the use case of deep neural networks in in-directly aiding to result exploration. Proposed method is to use simulation result data to predict the very same results on different parameters. Allowing users to drastically reduce simulation input. It is expected to see that input features from experiment design can lead to a remarkably accurate modeling of a far-fetched regression prediction. The performance is show to be as high as 80%.

Keywords: *Deep Neural Networks; Experiments; Predictions*

1. Introduction

The practice of data mining, in light of the availability of large-scale data, has been causing a paradigm change in scientific discovery, from empirical science to theoretical, computational science, and now to data science. For our purpose, we will be applying the medium to experimentation and simulation field.
In simulation study, the primary goal is concerned in getting experimental results without conducting experiments in real world. Using the computational power available to us, allows to us to replicate entire experimentation system, including environment, instruments, and focus object. A general idea for simulation - Initially desired experimental system has to be designed with all the appropriate environmental factors and also experiential functions. Now, users can feed simulation input parameters in order for the system to setup completely. Simulation is now setup and ready to start experiments. Users can tell simulation systems to output certain parameter and also track parameters over time. This allows users to later study and analyse the experiment. While this entire thing works with great accuracy it takes a lot of computational power, time, and right expertise in the given field.
What this report propose is, we also use Deep Neural Networks in conjunction with simulations. This will allow us to reduce the number of simulations and also reduce the variety of parameter runs, while having our neural network make predictions based on the limited simulation data.

2. Methodology

A general methodology flow has been shown in figure1 below.
Firstly, we have to get experiment data from simulations. Doing so will allow us to train our neural network for our subset of problems. After extraction of relevant data we need to process it.

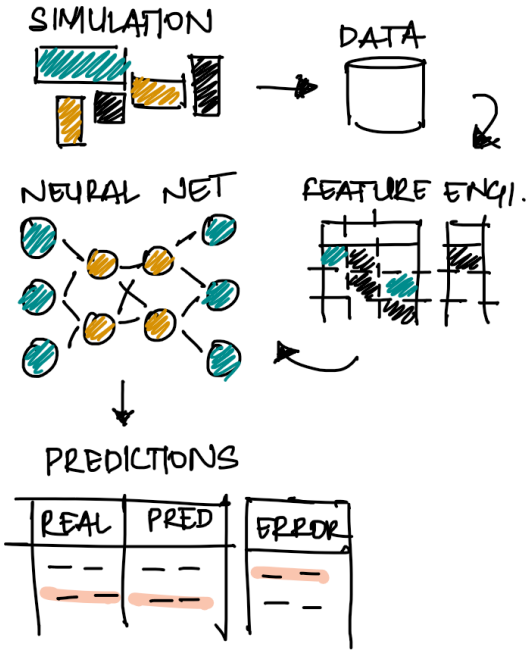
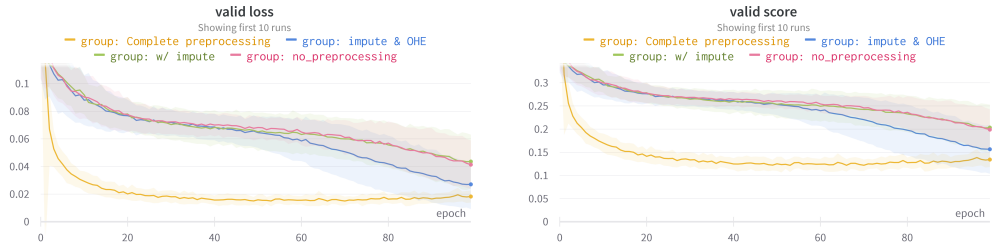


Figure 1. A general idea

While doing machine learning and deep learning in general we need to stabilize our data for neural net to work efficiently and give us better results. Processing data can affect neural net in a very significant way. The fig8 shows how each layer of data processing is impact neural network loss and score functions[1]. With this, we can justify that imposing a (pre)processing layer on our model helps neural net to reduce loss faster and drastically improve model score.



(a) Neural Network 'Loss' graph **(b)** Neural Network 'Score' graph

Figure 2. (pre)Processing impact on Loss and Score for Neural Network

2.1. Layers of (pre)Processing

On the topic of pre processing, while extracting data from simulation there is always a certainty that it will contain some form of noise. It may be deviations in computations, false results, or even missing values. For our neural net to perform efficiently we need to clean up these irregularities. Here, we have used generalized data processing methods, widely used by deep learning community. Some of which are,

- Imputation:
  This function deals with missing values in our data. Since we are trying to reduce amount of first hand simulation, we need to every bit of data we can have, imputation does just that. Rather that just dropping features because of missing target values, we calculate mean of

[1] explained later

the entire target column and substitute missing value with it. This technically creating a false simulation run, but for neural network it works just fine as they work towards finding patterns. Also when we do imputation on missing values we also add a new feature which keeps track of which exact index was imputed. This helps neural net even more.

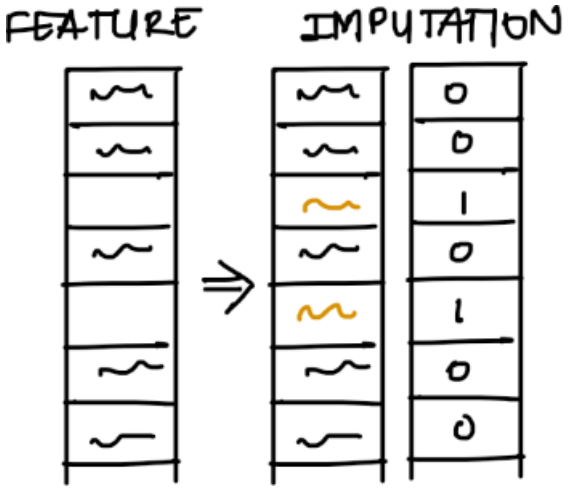


Figure 3. Imputation Layer

- One Hot Encoding:
  For categorical features, we need to represent them in a way that neural network can understand. We do this by one hot encoding the feature column. This creates a binary column for each category. This enables neural net to use these features as categories rather than plain numbers.

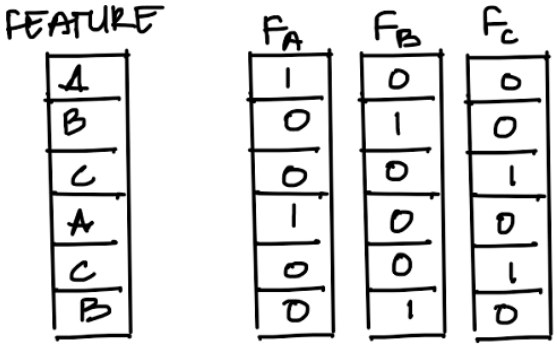


Figure 4. OneHotEncoding Layer

- Normalized:
  This layer fits all the given features into range of either [0, 1] or [-1, 1]. This helps neural net work inside a generalized probability domain rather than to in random space. This also helps to reduce feature skewing. Formula used to normalized is,

$$z = (x - \mu)/\sigma \tag{1}$$

  here,
  $z \rightarrow Standardization$
  $\mu \rightarrow Mean$
  $\sigma \rightarrow Standarddeviation$

2.2. Neural Network Model

Now that our input data is processed and ready to inject into neural network, first we need to make one. Goal is clear and simple. We need to build a model, which can predict the simulation results given experiment input parameters.
To achieve this we will be using python and some pre built deep learning libraries - PyTorch.
Code available at: https://github.com/AZarbade/HVIS_redesign/tree/master

The process of building a neural networks is faily straight forward and their fore left out of this report. However, all the important parameters and configs are mentioned below. After building a neural network

- Model Configs:
  - random_seed = 1024: used to set reproduciblity.
  - num_neurons = 64: number of perceptrons per hidden layer
  - num_layers = 3: number of hidden layers
  - layer_dropout = 0.2: percentage of droput after each layer, excep last layer
  - activation_function = Relu: perceptron activation function
- Trainer Configs:
  - epochs = 100: number of iterations model was trained
  - batch_size = 32
  - loss_function = mean_squared_error: loss funtion used to penalize model
  - optimizer = AdamW: optimizer used
  - learning_rate = 0.0003

2.3. K-Fold method

In deep learning, K-Fold method is used for cross-validation of neural networks. Learning the parameters of a prediction function and testing it on the same data is a methodological mistake: a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet-unseen data. This situation is called **overfitting**. To avoid it, it is common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set X_test, y_test. Note that the word "experiment" is not intended to denote academic use only, because even in commercial settings machine learning usually starts out experimentally. Here is a flowchart of typical cross validation workflow in model training.

When evaluating the model there is still a risk of overfitting on the test set because the parameters can be tweaked until the estimator performs optimally. This way, knowledge about the test set can "leak" into the model and evaluation metrics no longer report on generalization performance. To solve this problem, yet another part of the dataset can be held out as a so-called "validation set": training proceeds on the training set, after which evaluation is done on the validation set, and when the experiment seems to be successful, final evaluation can be done on the test set.

However, by partitioning the available data into three sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV for short). A test set should still be held out for final evaluation, but the validation set is no longer needed when doing CV. In the basic approach, called k-fold CV, the training set is split into k smaller sets (other approaches are described below, but generally follow the same principles). The following procedure is followed for each of the k "folds":

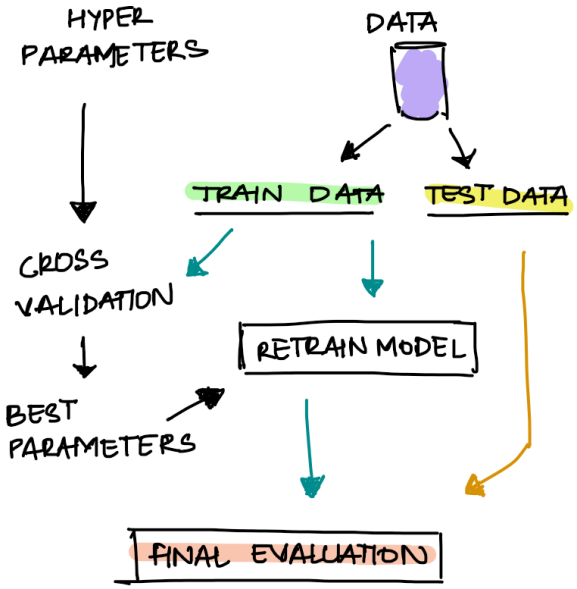


Figure 5. Cross-Validation workflow

- A model is trained using $k-1$ of the folds as training data.
- the resulting model is validated on the remaining part of the data.

This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set), which is a major advantage in problems such as this where the number of samples is very small.

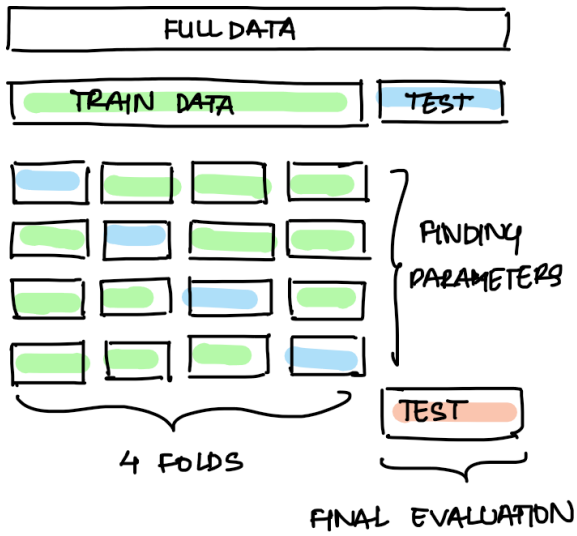


Figure 6. Cross-Validation data structure

3. Results and Analysis

Now that we have created our neural net, we are ready to inject our data into it and let the model train. Fig7 shows the loss curve of the model over 100 epochs. To get this curve, we used K-Fold method to train our model and logged loss curve across all the intermediate runs.
What we have in fig7 is mean of validation loss and their standard deviations.

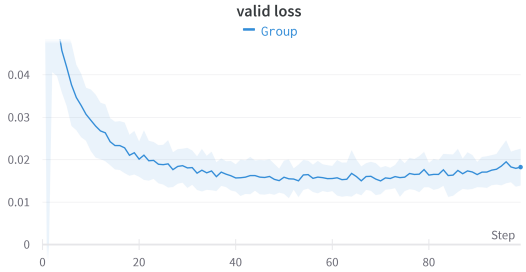


Figure 7. Neural Net Validation loss over 100 epocs

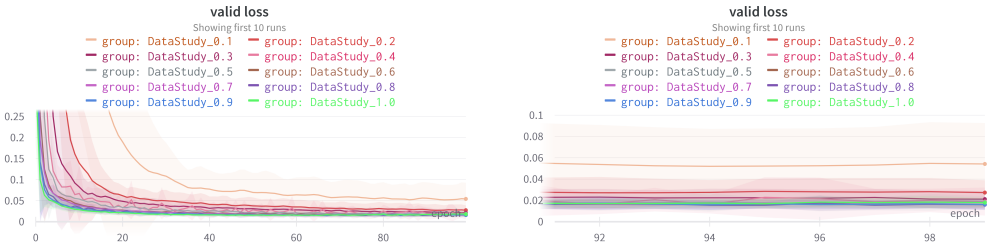
If studied closely we can infer couple of things:

- Standard deviation is on the higher side, ie. model predictions are fluctuating a lot.
- After about 70 epochs the curve starts to lean upwards, indicating overfitting.

We can conclude couple things here. firstly, we need to tune our hyper parameters more. Even though the current model was somewhat tuned even more tuning is needed. Secondly, we still haven't compared our model to existing statistical methods or models.

3.1. Data threshold study

In this section we will look at the impact of amount of data on our model.
Since we are trying to optimize for amount of data needed, we also need to evaluate our model for the same.



(a) Percentage of data used **(b)** Zoomed on couple of epochs

Figure 8. DataStudy over 10% steps

What can we infer:

- We can immediately see that our overfitting problem was solved by using a better training loop.
- However, model still shows great amount of standard deviations across all the runs.
- In fig8b we can see that we can reduce our data size to about 20% and still have a significantly better performance than going to 10%.

4. Conclusion

In conclusion, we can say that the Neural networks is indeed capabale of handling such problems. Use of neural networks for predicting results is possible. However, current implementation of neural networks seems to be unreliable in providing confidence in its results. There are other methods present which are much more suited to solve such tabular datasets. Which I will have in the next report.

For now though, current solution seems to be correct way forward for our problem statement.