Surrogate Based Modelling and Optimization

A report submitted in partial fulfilment of the requirements for the degree of B.Tech. Mechanical Smart Manufacturing

by

Asvataman V.
(Roll No: MSM17B029)

DEPARTMENT OF MECHANICAL ENGINEERING

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,

DESIGN AND MANUFACTURING, KANCHEEPURAM

October 2020

Certificate

I, **Asvataman V**, with Roll No: **MSM17B029** hereby declare that the material presented in the Internship Report titled **Surrogate Based Modelling and Optimization** represents original work carried out by me in the **Department of Mechanical Engineering** at the **Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram** during the months May 2020 to October 2020. With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

V. Asvataman

Date:October 25, 2020 Student's Signature

In my capacity as supervisor of the above-mentioned work, I certify that the work presented in this Report is carried out under my supervision, and is worthy of consideration for the requirements of B.Tech. Project work.

Advisor's Name: Advisor's Signature

Abstract

Most engineering design problems have an objective function and a set of constraints on parameters. One may solve the problem using Simulation models based on Numerical methods. Since a design problem involves experimentation, in case of change in the parameters the simulation needed to be generated again. Surrogate modelling is an engineering method in which a mathematical model, which is represents the true case as accurate as needed, is used to get the required result.

Surrogate modelling is crucial, as in a real life problem a single simulation may take hours or days to complete and any change in the design parameters requires a new simulation. Although generating or modelling a surrogate for the problem is also time demanding process, once modelled a problem will require almost no time to produce Output.

As almost all Physical problem can be represented as a Partial Differential Equation(PDE) the objective of the internship is to create a surrogate model for a PDE based system. The internship was begun by solving a simple Quadratic Equation using surrogate modelling by various methods. By the knowledge gained on studying surrogate modelling of a quadratic equation it is implemented to solve PDEs.

# *Acknowledgements*

I would extend my sincerest gratitude to would express my gratitude and appreciation to all those who gave me the possibility to complete this report. I am very thankful to Indian Institute of Information Technology, Design and Manufacturing, Kancheepuram to have given me this opportunity to undertake my five-month internship under the same. It was a significant experience for me to have worked under such a research topic consisting of interesting challenges.

I would also like to acknowledge with much appreciation the crucial role of my teammate D Manonmani whose help, stimulating suggestions and encouragement helped me in coordinating throughout the duration of the internship.

Last but not least, many thanks go to the guide of the project, Dr.Siva Prasad who has given his full effort in guiding the team in achieving the goal as well as his encouragement to maintain our progress in track. It was a joyful as well as a fruitful experience working under his guidance.

Further on I would also like to thank my friends, family and other interns who supported and made this demanding time joyful.

Contents

List of Figures

Abbreviations

PDE	**P**artial **D**ifferential **E**quation
GDA	**G**radient **D**escent **A**lgorithm
NN	**N**eural **N**etworks
QE	**Q**uadratic **E**quation
MLP	**M**ulti-**L**ayer **P**erceptron

Chapter 1

Introduction

Many real-life engineering design problems involve complex and tedious computation. These are often approached by complex computer simulation. For example, in modern aerospace design the computational power needed to support advanced decision making can be extraordinary even with the latest and most powerful computers. Designing process include a series of experiments, so simulation model of all possible case will directly affect the time and cost of the product.

Surrogate models are mathematical functions which act as a surrogate or substitute to the actual problem. Surrogate models represents the original models as accurate as needed by the designer and can give the output desired directly without simulation. The basic idea is for the **surrogate to act as a 'curve fit'** to the available data so that results may be predicted without recourse to use of the primary source (the expensive simulation code). This approach is considered advantages as once trained by the available data output can be obtained for any input in no time compared to simulation of the whole data.

In this Internship a surrogate model for a real life design problem which involves a system of PDEs is created. Since system of PDEs are too complex to begin with, a quadratic equation is used to analyze the various methods available such as *Polynomial Model, Kriging, Radial Basis Function and Artificial Neural Networks.* Consider the general quadratic equation :

$$ax^2 + bx + c = 0$$

$Let\ \ f\ \ be\ a\ function\ such\ that$

$$f(a, b, c) = (r_1, r_2)$$

$where\ \ r_1, r_2\ \ are\ roots\ of\ the\ quadratic\ equation$

Here f is surrogate to the roots of Quadratic Equation Formula

$$r_1, r_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Finding roots of a Quadratic Equation is chosen as it is simpler form where 3 parameters i.e. coefficients of the equation is to be mapped with the roots of the Equation. Then this idea is extrapolated to model a system of Partial Differential Equation like Navier - Stokes Equation.

Chapter 2

Statistical and Regression models

2.1 Models

Surrogates are blackbox which tries to mimic a complex mathematical model to produce the output at lower cost or time. Statistical and regression models involve a predefined mathematical function with arbitrary coefficients to begin with. The arbitrary coefficients are then optimized in such a way to make the function a surrogate to the original model.

Some of the popular model functions are as follows:

Polynomial Model :

$$\widehat{f}(x, m, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m = \sum_{i=0}^{m} w_i x^i$$

Here the vector **w** is the parameter that need to be optimized to get a required model.

Radial Basis Function:

$$\widehat{f}(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\boldsymbol{\psi} = \sum_{i=1}^{n_c} w_i \psi(\|\mathbf{x} - \mathbf{c}^{(i)}\|)$$

Kriging:

$$\mathrm{cor}[Y(\mathbf{x}^{(i)}), Y(\mathbf{x}^{(l)})] = \exp\left(-\sum_{j=1}^{k} \theta_j \mid x_j^{(i)} - x_j^{(l)} \mid^{p_j}\right).$$

These functions are readily available in MATLAB libraries.

2.1.1 Loss Function

The optimization of the parameters requires a measure of error with respect to the expected result. This measure is called a Loss function. A simple loss function would be just the difference of Output vs the Expected Output.

$$Loss = \hat{y} - y$$

In this Loss function, loss takes negative value when the expected output is more than the actual output and vice-versa. This lead to loss cancelling out due to the sign. This can be solved by use of modulus or square function.

$$Loss = |\hat{y} - y|$$

$$or$$

$$Loss = (\hat{y} - y)^2$$

The Squared Loss functions is used due to the fact that it is differentiable at all real values.

2.1.2 Optimization Algorithm

After finding the Loss the next step is to minimize the loss by changing the weights or parameter **w.** This is done by the Optimization Algorithm. The most basic of this is called Gradient Descent Algorithm.

In Gradient Descent Algorithm(GDA) the slope of the Loss function with respect to **w** is found. If this slope is zero it means the optimal point is reached or else the **w** is updated by a constant value for which the slope tends to zero and the loss is calculated again till the slope becomes zero.

2.2 Solving a Quadratic Equation

2.2.1 Data Preparation

Data collection is the most import step in modelling. In most case these are data collected and recorded from actual experiments and sometimes these are collected from sophisticated simulations.

In this example we take a quadratic equation with coefficients a=1, b= 5, c=6. i.e.

$$y = x^2 + 5x + 6$$

Random values for x is taken to generate the y. As roots of quadratic equation is system of equations, data is prepared such that it is present only in one half of the parabola.

2.2.2 Model

A polynomial model with degree 4 is used as model. Therefore, there are 5 weight values need to be optimized. Using MATLAB the following weights are obtained to be the optimal value and thus the Polynomial is

$$x = 0.0675y^4 - 0.2101y^3 - 2007y^2 - 0.582y - 3.0261$$

$$y = 0 \quad as\ we\ need\ to\ find\ the\ root$$

$$For\ \ y = 0\ \ x = -3.0261$$

Whereas the actual root is **x = -3**

2.3 Limitations

As of now the accuracy in the roots of the quadratic equation is pretty good. But this result is made by considering only one root of the Quadratic Equation and imaginary roots omitted by choosing a, b, c accordingly.

Moreover, data points fed into the model are close to the actual roots. Even considering only one root generalized model for finding roots with any coefficient is not possible with Statistical Models as square root is complex to model in them.

This can be solved by the incorporation of Neural Network model as it can handle complex functions and system of equations in parallel. Modelling both roots is important as many real-life problems involving PDEs are often present as system of Equations.

Chapter 3

ANNs and Quadratic Equation

3.1 MP Neuron and Perceptron

3.1.1 MP Neuron

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943. MP Neuron is simple function which returns 1 (one) when the sum of the inputs is greater than a threshold **b.** Though this is a simple function this is the base for Neural Networks.

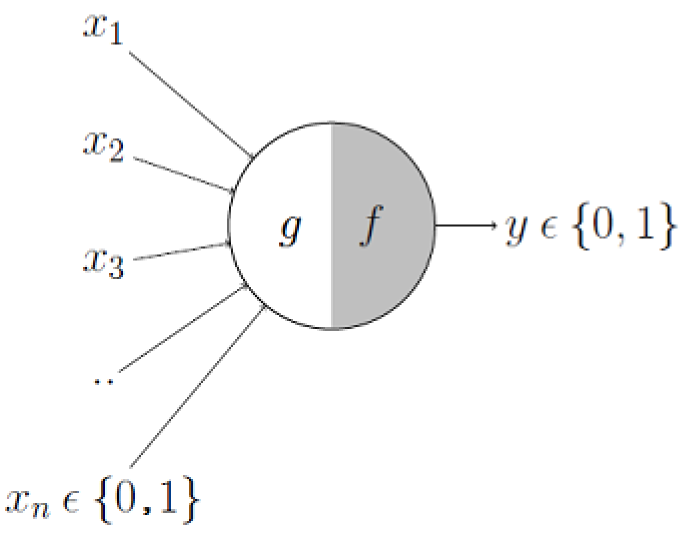


Figure 1: MP Neuron

The model was based on how a neuron works in a human body. So it was defined as two functions f and g representing **synapse and Dendrites**. The function g collects all inputs values and sums them up while the function f makes a decision whether the output is 1 or 0. Here each input is a Boolean value.

$$g(x_1, x_2, x_3, \ldots.., x_n) = g(X) = \sum_{i=1}^{n} x_i \quad , x \; \epsilon \; \{0, 1\}$$

$$\begin{aligned} f(g(X)) &= 1 \quad if \quad g(X) \geq b \\ &= 0 \quad if \quad g(X) < b \end{aligned}$$

$$where \;\; b \; \epsilon \; [0, n]$$

As we can see the major drawback with MP neuron is that :

- Every input has equal contribution to the final result.
- Both input and output are Boolean.
- It will not work for data which is not linearly separable.

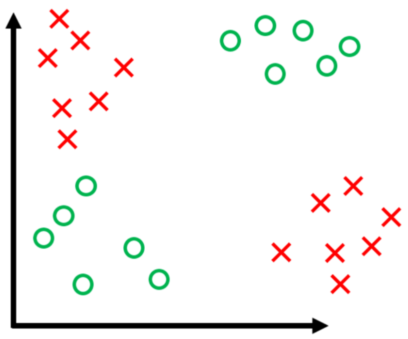


Figure 2: Linearly Inseparable data

3.1.2 Perceptron

A perceptron which is derivative of a MP neuron is the building block of a Neural Network. Unlike MP Neuron it takes real numbers as input and incorporates weights for each input.

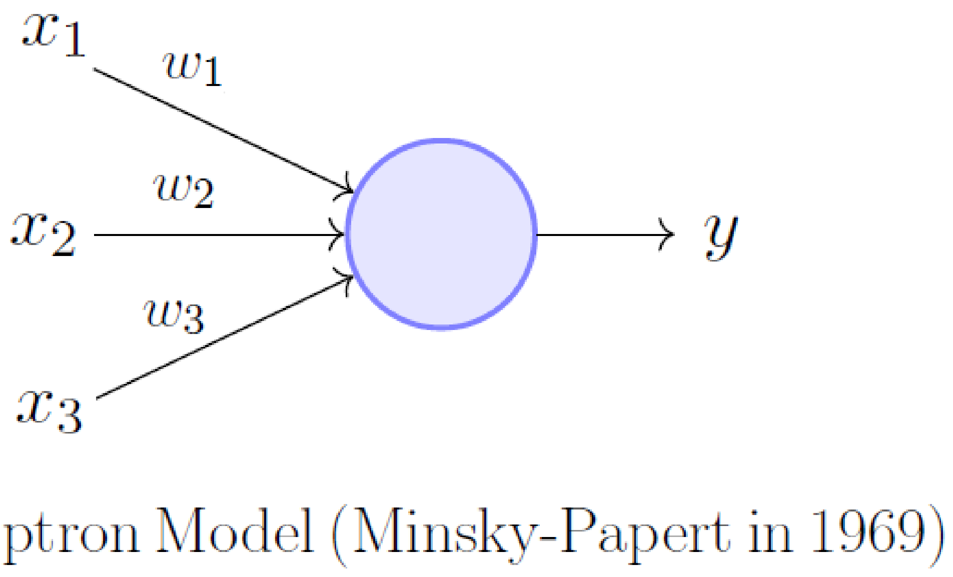


Figure 3: Perceptron Model

The perceptron model can be represented as follows

$$y = 1, \text{if} \sum_{i=1}^{n} w_i x_i \geq b$$

$$y = 0, \text{if} \sum_{i=1}^{n} w_i x_i < b$$

where ***w, x, b*** belongs to Real numbers.

The perceptron model solves the problem with MP Neuron partially. The inputs are real numbers, each input now have weights and the threshold is also real but still the output is Boolean. Thus perceptron requires the data to be linearly separable to be functional. As for as the problem in hand, which is finding roots of a Quadratic Equation it is illogical to use MP Neuron or Perceptron.

3.2 Multi-Layer Perceptron(MLP)

3.2.1 Architecture of MLP

Multi-Layer Perceptron is a network of Perceptron except that each node uses a non-linear activation function to overcome the problem of output being Boolean. After summation of inputs with their respective weights instead of a threshold function an activation function whose output is a Real Number is used.

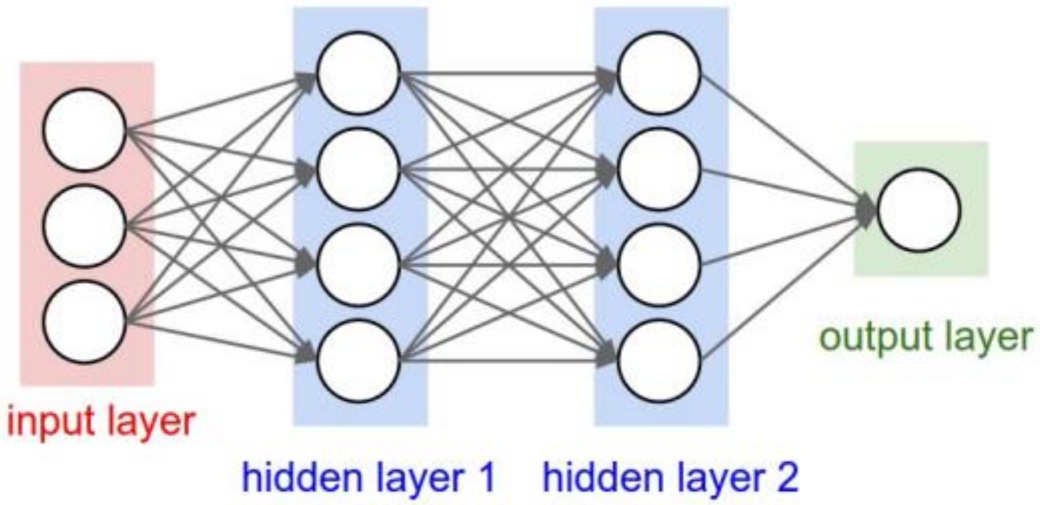
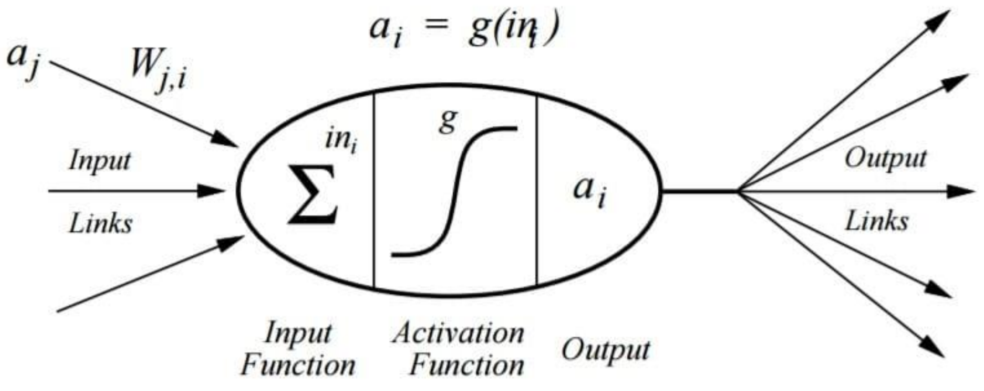


Figure 4; Multi-Layer Perceptron

The hidden layers are collection of a Neuron whose output is not used directly for analysis instead they are connected to the other hidden layers or to Output Layer. Each neuron in a MLP take real inputs and gives a single output which is passed on to all neurons in the next layer. An individual neuron looks as given below.



$$a_i = g(\sum_j W_{j,i} a_j)$$

Figure 5: A single Neuron in a MLP

3.2.2 Weights and Biases

In a MLP for finding the appropriate weights and biases we need to implement the following steps:

- Feed Forward
- Back Propagation

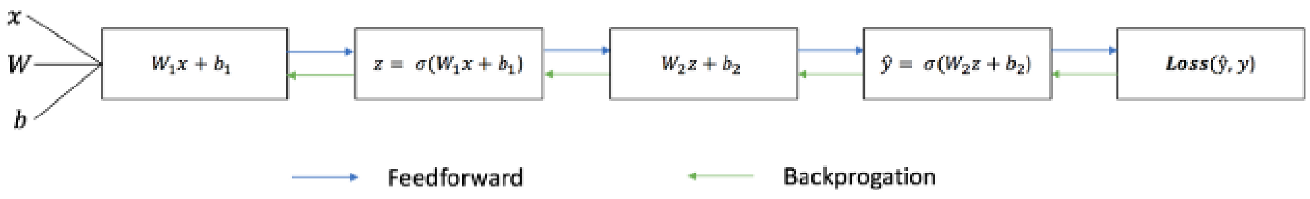
In Feed Forward we have a set of inputs and we assign the weights and biases randomly starting from input layer till the last hidden layer. The output is calculated using the assigned weights and biases. During Back Propagation the error in the output with respect to the expected output from the data is calculated and an optimization algorithm is used to update the weights in the direction which results in zero loss.

A simple Neural Network with 2 hidden layer is used with a sigmoid activation is used to solve the quadratic equation. The Neural Network has the following features:

- An input layer, $\mathbf{X}$
- 2 Hidden layers
- An Output Layer, $\mathbf{\hat{Y}}$
- A set of weights and biases at each layer, $\mathbf{W}$ and $\mathbf{b}$
- Sigmoid activation function for each layer, $\boldsymbol{\sigma}$

Here X, $\hat{Y}$, W and b are vectors

So the Neural Network would look like the following



Thus the output layer of would be

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

3.2.3 Loss Function

For optimizing the value of W and b Gradient Descent Algorithm is used. It is a simple algorithm Where gradient of the loss function is used to minimize the loss. The weights are updated such that Gradient of the loss function decreases which ultimately leads to the decrease in the value of loss. The loss function here taken is sum of square.

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \qquad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * \text{z(1-z)} * x$$

For simplicity b is assumed to be 0. The data for the problem is generated by assuming random coefficients and their roots are obtained by the quadratic equation formula.

Since the number of data points available for modelling is limited in a real life problem the data points are passed to the neural network in multiple iterations. Each iteration is called an **Epoch**.

After modelling, the model is validated by passing new set of inputs and compared with the predicted output. The following graph shows the train and test loss for the above procedure.

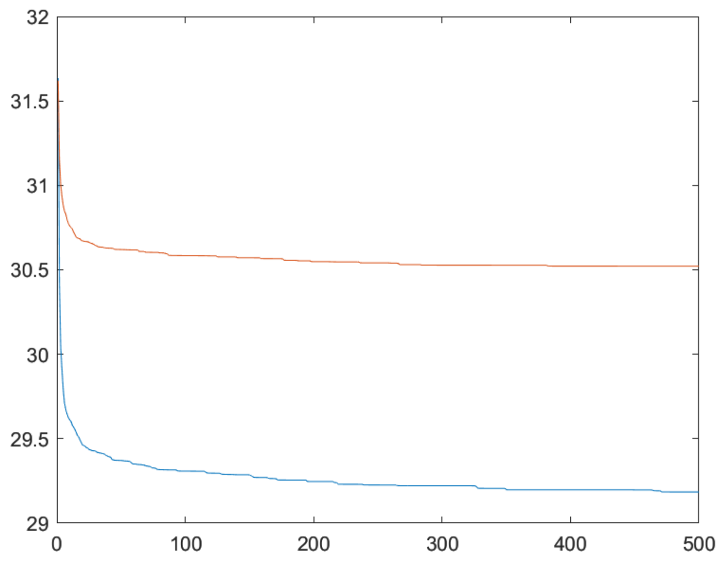


Figure 6: Epochs vs Loss graph with 2 hidden layers

The training loss is 29.1947 which is very high. It is solved by choosing appropriate Algorithm, Epochs, Layers and Nodes.

3.3 Algorithms and Hyper-parameters

This Section covers the effect of the result on the output by Optimization Algorithm, number of Neurons per layer and number of Hidden Layers and Activation Function.

3.3.1 Activation Function.

In the previous NN sigmoid function is used as activation function for which the output is in range (0,1). The loss is very high because roots are real number and negative are mapped to positive values. To solve this ***tanh*** activation is used for which the output is in range (-1,1).

3.3.2 Optimization Algorithm

As mentioned earlier GDA is basic algorithm where the weights and biases are updated in the direction where gradient of the loss function is decreasing. The value with which the weights are updated is called as learning rate. If learning rate is increased the weights will oscillate by skipping the optimum point.

RMSprop is a gradient based optimization technique used in training neural networks. RMSprop deals with the above issue by using a moving average of squared gradients to normalize the gradient.

This normalization balances the step size(momentum), decreasing the step for large gradients to avoid exploding, and increasing the step for small gradients to avoid vanishing.

In both RMSprop and GDA the learning rate is same for all layers. This is improved in ADAM optimizer by maintaining a separate learning rate for each layer. Now the following graphs shows the effect of Optimization Algorithm on the model.

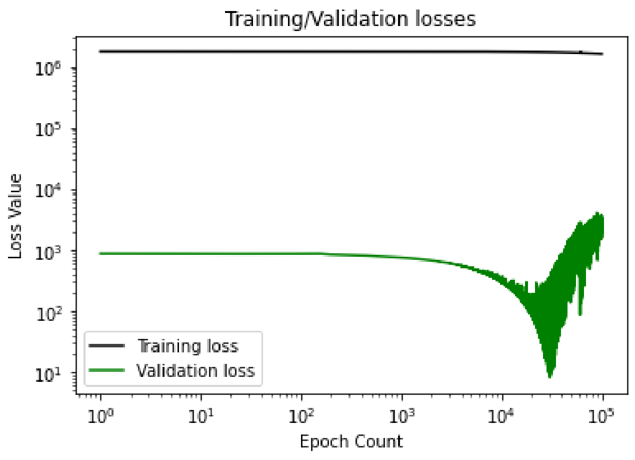


Figure 7: RMSprop Loss vs Epochs

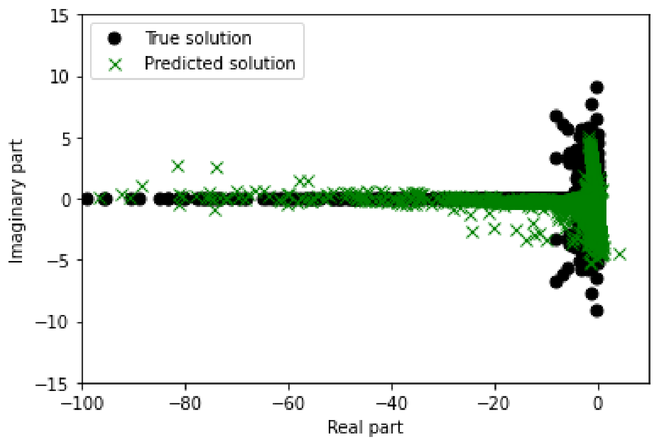


Figure 8: RMSprop True vs Predicted

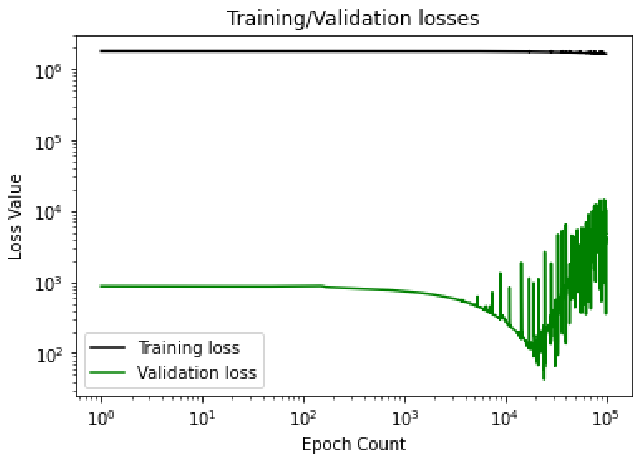


Figure 9: ADAM Loss vs Epochs

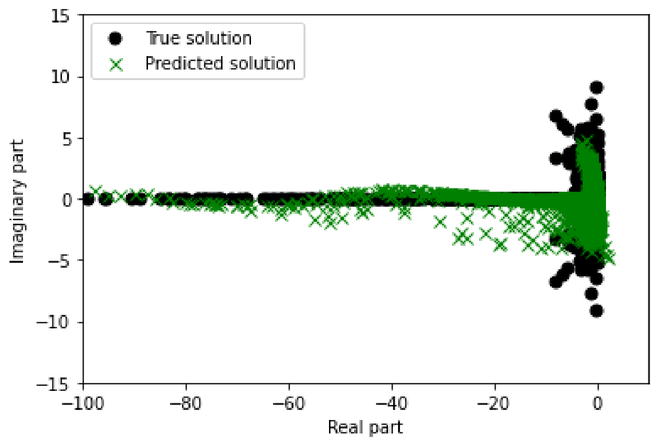


Figure 10: ADAM True vs Predicted

It is observed that the error for RMSprop is in order of 10^1 and in case of ADAM optimizer it in order of 10^2. For further analysis, RMSprop is used.

3.3.3 Number of Neurons

Now the number of neurons is varied with a 3 hidden layer NN to see the effect on the solution.

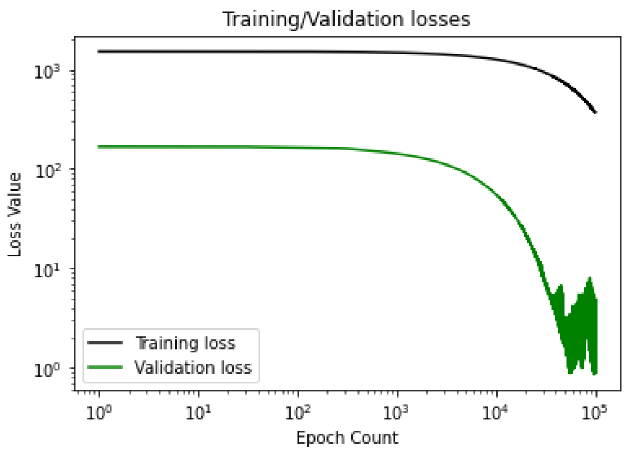


Figure 11: 5 Neurons Loss vs Epochs

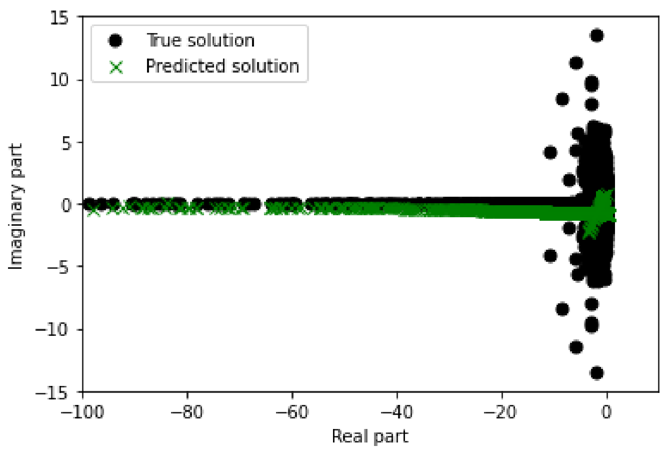


Figure 12: 5 Neurons True vs Predicted

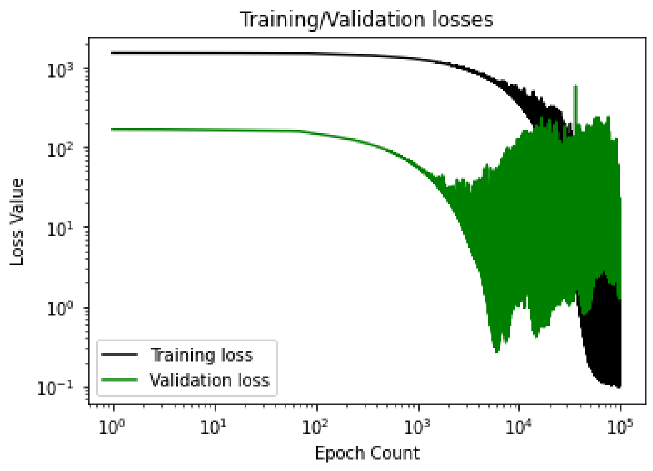


Figure 13: 50 Neurons Loss vs Epoch

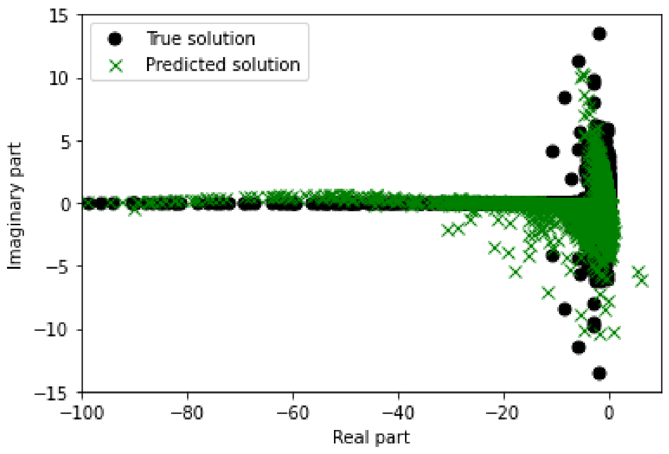


Figure 14: 50 Neurons True vs Predicted

It is observed that the error for 5 Neurons per layer is in order of 10^0 and in case of 50 Neurons it in order of 10^{-1}.

3.3.4 Number of Hidden Layers

For studying the effect of number of layers on the output a NN with 50 Neurons at each layer is used. For 7-layer the loss is in order of 10^{-2} whereas for 3-layer it is 10^{-1}.

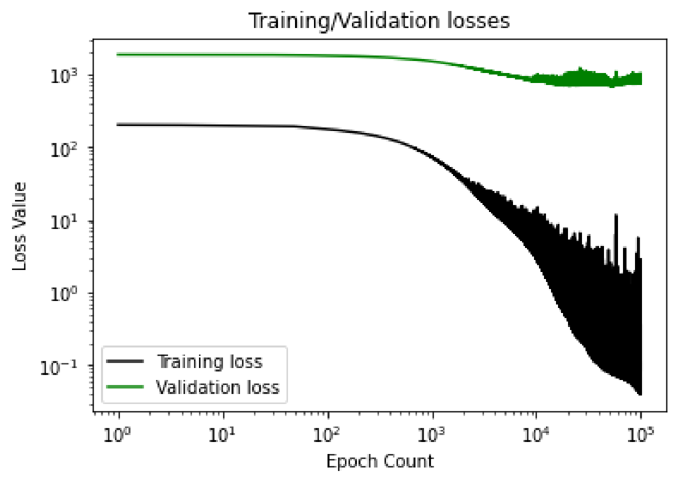


Figure 15: 3-Layer Loss vs Epochs

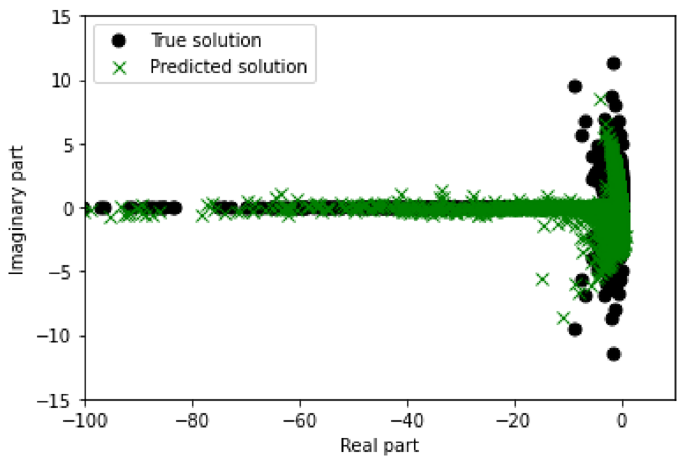


Figure 16: 3-Layer True vs Predicted

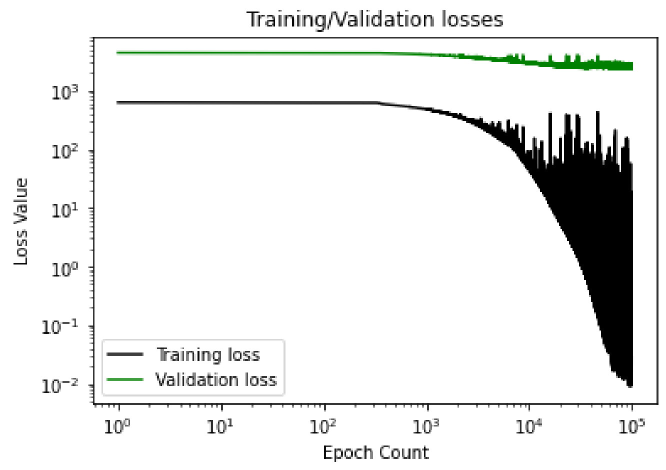


Figure 17: 7-Layer Loss vs Epochs

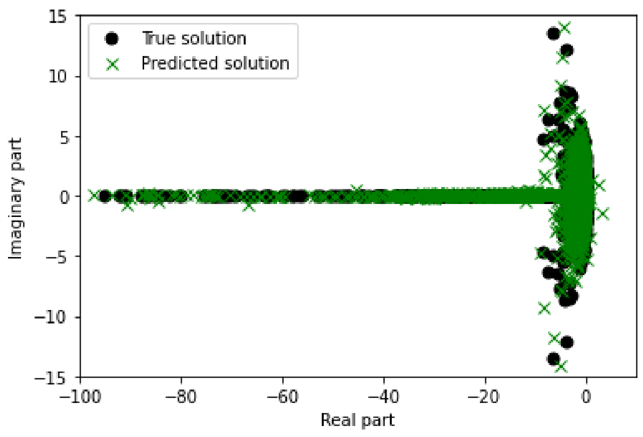


Figure 18: 7-Layer True vs Predicted

Chapter 4

Modelling a PDE

4.1 Basic Scheme

Solving a PDE is much more complex than a quadratic equation. In a quadratic equation the co-efficients are independent of the variable. But in case of PDEs the variables are interdependent. Thus passing the input parameters direct will result in poor accuracy. Moreover, finding the partial differential of the data which need to be passed as input is not practical. We solve this by making a tweak in the loss function.

Let $\lambda(x, y, z) = 0$ be a Partial Differential Equation and **L** be the squared error loss of the Output layer to the true value.

$$Loss \ = \ L + [\lambda(x, y, z)]^2$$

Now by this tweak when the optimization algorithm tries to make the Loss as zero consequently the PDE condition is also satisfied.

As an example the Burgers' equation is considered. In one space dimension, the Burger's equation along with Dirichlet boundary conditions reads as.

$$\begin{aligned}
&u_t + uu_x - (0.01/\pi)u_{xx} = 0, \quad x \in [-1,1], \quad t \in [0,1],\\
&u(0,x) = -\sin(\pi x),\\
&u(t,-1) = u(t,1) = 0.
\end{aligned}$$

Now a function f is defined as

$$f := u_t + uu_x - (0.01/\pi)u_{xx},$$

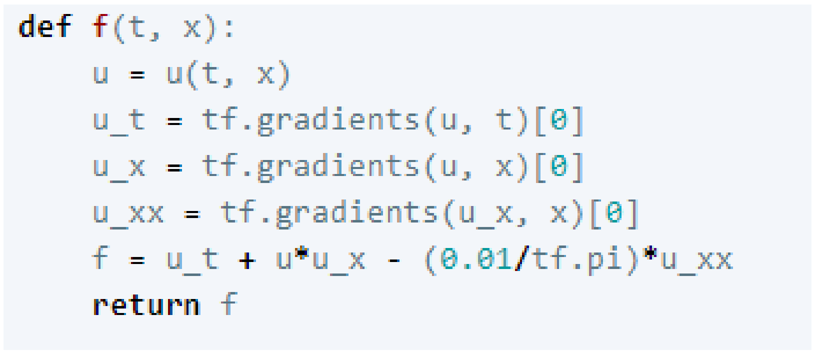
Where,

$$u_x = \frac{\partial u}{\partial x}, \quad u_{xx} = \frac{\partial^2 u}{\partial x^2}$$

A Neural Network is defined such that it takes ***x*** and ***t*** as input layer and ***u(t, x)*** as output layer. The following code snippet shows how it done using Tensorflow

```
def u(t, x):
    u = neural_net(tf.concat([t,x],1), weights, biases)
    return u
```

For calculating the loss, we need to define the function *f*. For this the predefined function in Tensorflow is used.



```python
def f(t, x):
    u = u(t, x)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]
    f = u_t + u*u_x - (0.01/tf.pi)*u_xx
    return f
```

The shared parameters between the neural networks ***u(t ,x)*** and ***f(t, x)*** can be learned by minimizing the mean squared error loss.

MSE = MSE $_U$ + MSE $_f$

Where

$$MSE_u = \frac{1}{N_u} \sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2,$$

and

$$MSE_f = \frac{1}{N_f} \sum_{i=1}^{N_f} |f(t_f^i, x_f^i)|^2.$$

The loss MSE$_U$ corresponds to the initial and boundary data while MSE $_f$ enforces the structure imposed by the Burgers' equation at a finite set of collocation points.

4.2 Navier-Stokes Equation

Navier-Stokes equations describe the physics of many phenomena of scientific and engineering interest. They may be used to model the weather, ocean currents, water flow in a pipe and air flow around a wing. The Navier-Stokes equations in their full and simplified forms help with the design of aircraft and cars.

Considering the Navier-Stokes equation in 2-D in explicit form as below to model a Neural Network with Continuity equation for incompressible flow,

Navier-Stokes Equation :

$$u_t + \lambda_1(uu_x + vu_y) = -p_x + \lambda_2(u_{xx} + u_{yy}),$$
$$v_t + \lambda_1(uv_x + vv_y) = -p_y + \lambda_2(v_{xx} + v_{yy}),$$

where $\lambda = (\lambda_1, \lambda_2)$ is an unknown parameter

Continuity Equation :

$$u_x + v_y = 0.$$

A latent function $\psi(t, x, y)$ is assumed such that

$$u = \psi_y, \quad v = -\psi_x$$

From the assumptions made, the Neural Network is defined with

Input Layer : Consisting of 3 Neurons representing *x, y, t* from the dataset.

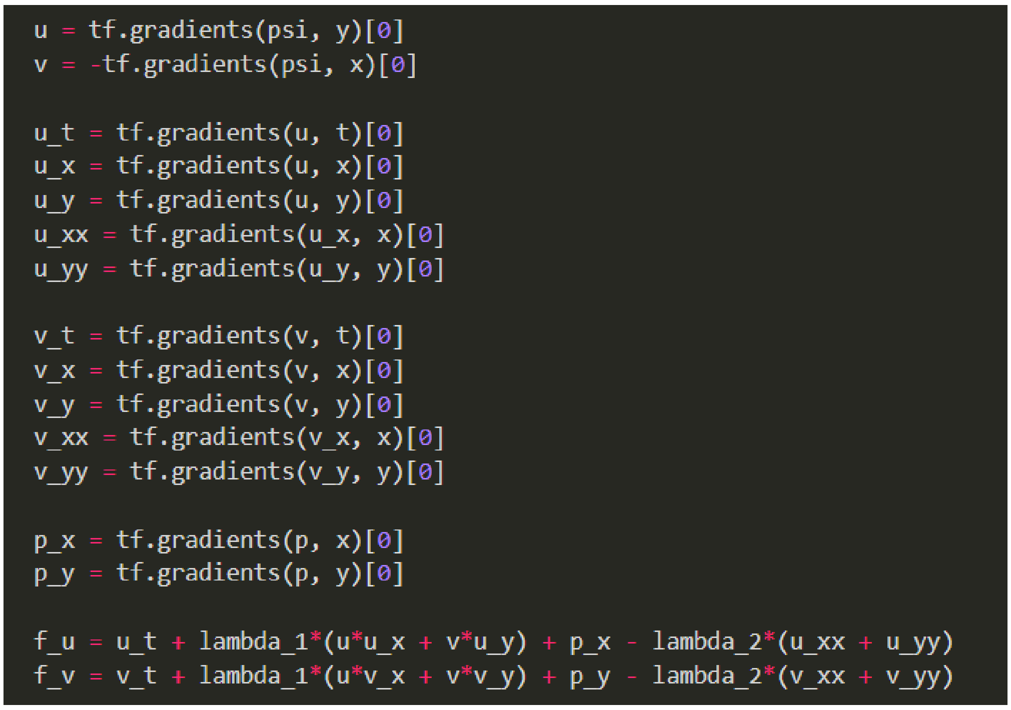
Output Layer : Consisting 2 Neurons representing $\psi(x, y, t)$ and $p(x, y, t)$.

Hidden Layers : 8-Hidden layers with 20 Neurons on each layer.

Activation Function: tanh

Optimizer Algorithm: ADAM optimizer.

u and v are obtained by finding the gradient of ψ with respect to *y* and *x* respectively. Refer the code snippet.



```
u = tf.gradients(psi, y)[0]
v = -tf.gradients(psi, x)[0]

u_t = tf.gradients(u, t)[0]
u_x = tf.gradients(u, x)[0]
u_y = tf.gradients(u, y)[0]
u_xx = tf.gradients(u_x, x)[0]
u_yy = tf.gradients(u_y, y)[0]

v_t = tf.gradients(v, t)[0]
v_x = tf.gradients(v, x)[0]
v_y = tf.gradients(v, y)[0]
v_xx = tf.gradients(v_x, x)[0]
v_yy = tf.gradients(v_y, y)[0]

p_x = tf.gradients(p, x)[0]
p_y = tf.gradients(p, y)[0]

f_u = u_t + lambda_1*(u*u_x + v*u_y) + p_x - lambda_2*(u_xx + u_yy)
f_v = v_t + lambda_1*(u*v_x + v*v_y) + p_y - lambda_2*(v_xx + v_yy)
```

The following graph show the Loss function of the Neural Network for 10^5 epochs.

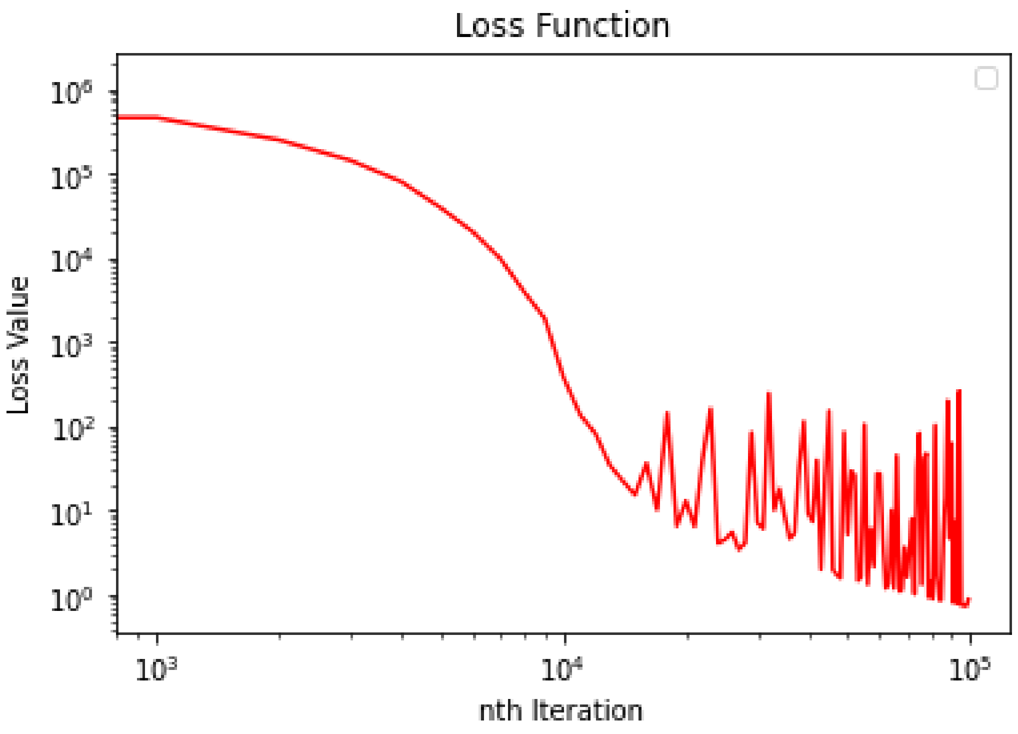


Figure 19: Loss vs Epoch in Navier-Stokes equation.

The loss function is in order of 10^0. The error value can be reduced if the number of data points are large. Due to the limitation in computing time available (4 hours) the number of epochs is limited.

Chapter 5

Summary and Conclusion

The internship was started with goal to create a surrogate model for an Engineering Design problem. As most problems involve PDEs the mission in hand was to create a surrogate model for solving PDEs. The problem was approached by solving a simple Quadratic Equation using basic statistical model. This method is proved to be ineffective so a neural network based solution was implemented.

Since using a Neural Network is more effective for solving a QE it is used to model a PDE. For this Navier-Stokes Equation which is widely used in many design problems is taken as an instance. But using a NN directly to solve PDE is not possible as it requires all variables (including their gradients) as either input or output for which generating data is nearly impossible. So a simple tweak of adding the squares of the PDEs to the loss function does the job.

In conclusion solving PDE using Neural Network maybe computationally equally or more expensive than simulation model. But in a long repetitive usage, which is usually in the case of designing, using a Neural Network is crucial.

Appendix A

Program Codes

All program codes are included in the drive link below for reference.

https://drive.google.com/drive/folders/1Dvtd1ehss5TeO6MFmsLeIJjQDedGEAZE?usp=s
haring

Bibliography

[1] Machine learning-based surrogate modelling for data-driven optimization: a comparison of subset selection regression techniques,Sun Hye Kim, Fani Boukouvala (2019).

[2] Variable Reduction for Surrogate modelling, J. Straus and S. Skogestad.

[3] Data driven approximation of parametrized PDEs by Reduced Basis and Neural Networks, Niccolò Dal Santo, Simone Deparis, Luca Pegolotti (2019).

[4] Artificial Neural Networks for solving Ordinary and Partial Differential Equation, I.E. Lagaris, A. Likas and D.I. Fotiadis

[5] Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations, Maziar Raissi, Paris Perdikaris, George Em Karniadakis (2017).