TECHNISCHE UNIVERSITÄT DRESDEN

SCALE IT-Solutions for CAE

Fakultät Bauingenieurwesen Fakultätsrechenzentrum

Parameter estimation for spot weld design in automotive construction

Master Thesis

Student: Akhil Rajasekharan Pillai

Supervisor: Dr.-Ing. habil. Uwe Reuter

Consultant: Dipl.-Ing. Marko Thiele

Dresden, 19.03.2019

Acknowledgment

I wish to express my sincere gratitude to Prof. Dr. -Ing. habil. Uwe Reuter and Dipl. -Ing. Marko Thiele, for giving me an opportunity to carry out my Master Thesis under their guidance. I pay sincere gratitude for their time and efforts rendered, throughout my thesis work. Their excellent suggestions and advice have always helped me in achieving the objectives for my work.

I would also like to thank SCALE Gmbh managers for their invaluable guidance as well providing the data needed to achieve the objective of my work, for providing me a complete working station at their branch in Ingolstadt, for providing licensed softwares like Animator A4, ANSA and many others.

I would further like to express my profound gratitude to my parents for their wise counsel. I am grateful to my friends for their support and continuous encouragement throughout.

Contents

1 Introduction

1.1 Motivation

In automotive production, each automobile has approximately 7,000 to 12,000 spot welds along with other kinds of connections. The position of the spot weld with respect to the flange and the distance between the spot welds as well as various other parameters usually vary for each part combination (spot weld design). If these properties are known, they can be used for automatic generation of spot welds during the design phase of the product development which is otherwise a cumbersome manual process. The spot weld design to be determined by the engineer depends on many factors (input parameters) such as loads and forces that might be applied to the structure, material combination, geometry of the parts, connection technology and it's process parameters. Some of these parameters such as material combination and geometry of the parts are predefined by the designer or are results of the circumstances such as loads and forces applied at the connection. The remaining parameters such as connection technology, process parameters, spot weld distances and flange distance have to be chosen by the engineer. On the basis of existing designs and with help of machine learning techniques it may be possible to predict the spot weld design parameters like spot weld distance and flange distance. Fortunately the input parameters and the spot-weld design can supposedly be extracted from a vast amount of FEM simulation input data available in the Simulation Data Management (SDM) system LoCo of SCALE GmbH. This data can be the basis for training and benchmarking new methods for estimating spot weld parameters.

1.2 Objective

In this work, we will try to estimate the distances between spot welds for spot weld design using machine learning approaches. For a machine learning approach, the first step would be to collect relevant data required to predict the desired outcome. In our case the desired outcome is the spot weld design parameter of distance between the spot welds. In order to predict the desired outcome, we make use of simulation data used for crash analysis. From the simulation data for crash analysis, we extract the geometry of the parts and also the spot weld design. From the data of already developed car model, we can build a model which should be able to provide a good initial estimate for distance between spot welds. With this model the design engineer has a good initial estimate of distance between spot welds for different part combinations and hence the number of design iterations required to reach the optimum values decreases.

In order to address the above mentioned problem, in this work, we start with developing methods to extract significant input parameters of existing 3D geometries of individual parts. With this geometric data we address the problem of classification of 3D geometric data. In order to solve the classification problem, we need to develop methods to express part geometries in specified 3D geometric data which can then be consumed by a macine learning architecture to predict part identifications. Once we solve the problem of part identification we move to parameter estimation for spot weld design. Different approaches will be implemented for the same and evaluated. We will also perform error-estimation for the predicted spot weld parameter.

2 Mathematical basics

This chapter presents the mathematical concepts used in this work in a brief manner to provide basic understanding to the reader. The subsections cover the basics of Artificial Neural Networks (ANN's) and deep learning architectures. We also briefly present mathematical basis for the concepts applied in this work like Barycentric coordinates and Latin Hypercube Sampling technique.

2.1 Artificial neural networks

Basic explanations of Artificial Neural Networks (ANNs), on which this subsection is based, can be found in [1, 2, 3, 4]. The formation of ANNs is an attempt to mathematically model the performance and intuitive capabilities of the human brain. The functionality of the human brain is essentially based on the interaction between the brain's highly cross-linked nerve cells called natural neurons. The communication within a natural neural network takes place via signals. A neuron serves for receiving, processing and passing on incoming signals. The signals received from neighboring nerve cells are summed in the neuron and if this simulation exceeds a particular threshold value, then further signal is activated and transmitted to the adjoining neurons. This basic structure is imitated by ANNs. The ANNs are then used to realize complex mappings of input variables on output variables. ANNs mainly consist of cross-linked computational nodes or artificial neurons and communication takes place via numerical values. An artificial neuron receives numerical values from neighboring neurons (*input neurons*), which are combined to form a weighted sum. This determined sum is then compared with a threshold value (*bias*) and used as the argument for so called activation functions. The activation function yields a value (*output signal*) which is an input signal for further connected artificial neurons.

A large variety of artificial neural networks exist for widely varying fields of application. The rapid increase in computing capabilities has aided the increase of scientific activity in application of ANNs in different forms to solve diverse real world problems such as recognition of handwritten characters or autonomous driving to name a few. ANNs may be subdivided into methods for approximating functions, for classification purposes and as associative memory units. An important type of artificial neural network is the multilayer perceptron. Multilayer perceptrons are universally applicable and hence will be explained in this work to understand the mathematics behind ANNs.

2.1.1 Multilayer perceptron

A multilayer perceptron is a special type of artificial neural network. The artificial neurons are arranged in layers. Starting from an input layer, numerical values are transferred or propagated to an output layer via one or more hidden layers. The output layer provides the result for the corresponding input data. The input and output data are usually real numbers. In this work, the explanations for multilayer perceptron is given with an example of a network with one hidden layer. An extension to several hidden layers is possible, but for the ease of explaining the mathematics and for better basic understanding we use network with just one hidden layer. A so called two-layered (output and hidden layers are only counted) multilayered perceptron is shown in Figure (1).

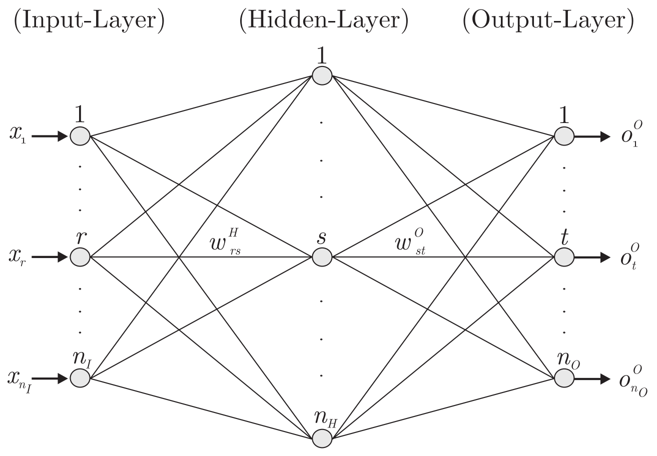


Figure 1: Schematic representation of a two-layer multilayer perceptron [4]

A variable x_r is assigned to each artificial neuron r of the input layer I. The counters $r = 1, 2, ..., n_I$ denote the number of neurons in the input layer. The task of the input layer is to receive the input data x_r and pass the data to the hidden layer as output variables o^I_r. Each neuron s of the hidden layer H lumps together the input variables x_r weighted by the value w^H_{rs} according to Eq.(2.1)

$$net^H_s = \sum_{r=1}^{n_I} w^H_{rs} \cdot x_r = \sum_{r=1}^{n_I} w^H_{rs} \cdot o^I_r \tag{2.1}$$

The counters $s = 1, 2, ..., n_H$ denotes the number of neurons in hidden layer. The intermediate result obtained is referred to as the net input net^H_s of the neuron s. The artificial neurons s maps the net input net^H_s onto the output variables o^H_s and transfers these to the output layer with the aid of an activation function $f_A(\cdot)$ according to Eq.(2.2)

$$o^H_s = f_A(net^H_s) \tag{2.2}$$

With the aid of the weights w^o_{st} the output variables o^H_s of the hidden layer H are lumped together in each case by the artificial neuron $t$of the output layer O to yield net input variables net_t^O according to Eq.(2.3)

$$net_t^O = \sum_{s=1}^{n_H} w_{st}^O \cdot o_s^H \tag{2.3}$$

The counters $t = 1, 2, ..., n_O$ denote the number of neurons in the output layer. Subsequently mapping the net input variables net_t^O onto the output variables o_t^O according to Eq.(2.4) provides the result data of the multilayer perceptron.

$$o_t^O = f_A(net_t^O) \tag{2.4}$$

In practice, threshold values are specified for individual neurons. These define the threshold above which the particular neuron becomes (highly) active. The threshold values θ_s^H and θ_t^O for the neurons $s = 1, 2, ..., n_H$ and $t = 1, 2, ..., n_O$ respectively, may be accounted for either directly in the activation functions according to Eq.(2.5)

$$o_s^H = f_A(net_s^H - \theta_s^H) \text{ and } \quad o_t^O = f_A(net_t^O - \theta_t^O) \tag{2.5}$$

or by an additional neuron in hidden or input layer. In this case, additional weights $w_{rs}^H(r = n_I + 1$ and $s = 1, 2, ..., n_H)$ and $w_{st}^O(s = n_H + 1$ and $t = 1, 2, ..., n_O)$ are used. The output variables o_I^r and $\tilde{o}$ of the additional neurons $r = n_I + 1$ and $s = n_H + 1$, respectively , are constant in accordance with Eq.(2.6)

$$o_r^I = 1 \text{ and } \quad o_s^H = 1 \tag{2.6}$$

The threshold values that are given by Eqs.2.7 and 2.8 accounts for the net input variables net_s^H and net_t^O of the hidden and output layers.

$$\theta_s^H = -w_{rs}^H \cdot o_r^I \text{with} \quad r = n_I + 1 \quad \text{and} \quad s = 1, 2, ..., n_H \tag{2.7}$$

$$\theta_t^O = -w_{st}^O \cdot o_s^H \text{with} \quad s = n_H + 1 \quad \text{and} \quad t = 1, 2, ..., n_O \tag{2.8}$$

The algorithm formulated for a two-layered multilayer perceptron can be generalized if several hidden layers are present. Processing of variables in an artificial neuron of hidden layer is show in Figure (2).

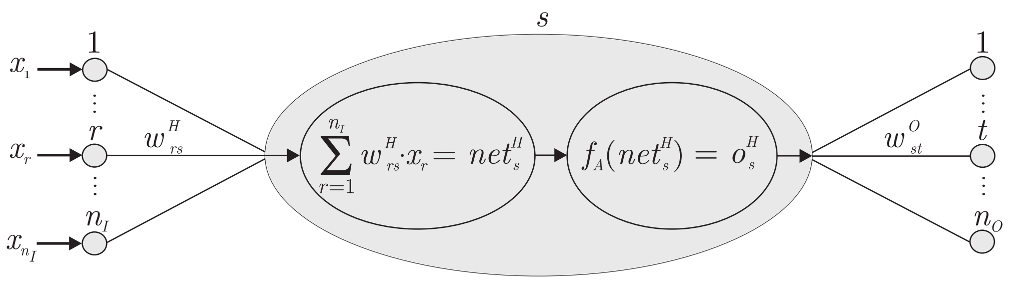


Figure 2: Processing of variables in an artificial neuron [4]

2.1.2 Learning algorithm of neural network

Multilayer preceptrons are used to approximate multidimensional non-linear functions. They are mainly used to map given input vectors on given target vectors as accurately as possible and to interpolate in between. Training patterns and training sets are constructed for this purpose.

A training pattern consists of an input vector $\underline{x} = (x_1, ..., x_r, ...x_{n_I})^T$ and a target vector $\underline{y} = (y_1, ..., y_t, ..., y_{n_O})^T$. A training set consists of m training patterns and is defined by the vectors $\underline{x}_k = (x_1(k), ..., x_r(k), ..., x_{n_I}(k))^T$ and $\underline{y}_k = (y_1(k), ..., y_t(k), ..., y_{n_O}(k))^T$ with $k = 1, 2, ..., m$.

For each input vector $\underline{x}_k = (x_1(k), ..., x_r(k), ..., x_{n_I}(k))^T$ of the training set, the multilayer perceptron yields an output vector $o_k^O = (o_1^O(k), ..., o_t^O(k), ..., o_{n_O}^O(k))^T$. The output error of the multilayer perceptron is determined by compairing the output vector $\underline{o}_k^O$ with the known target vector $\underline{y}_k$ with the aid of the square error E_k given by Eq.(2.9).

$$E_k = f_E(\underline{y}_k, \underline{o}_k^O) \tag{2.9}$$

$$= \frac{1}{2}\sum_{t=1}^{n_O}(y(t,k) - o^O(t,k))^2 \tag{2.10}$$

The output error of the multilayer perceptron for the complete training set is defined as the mean square error *MSE* according to Eq.2.11.

$$MSE = \frac{1}{m}\sum_{k=1}^{m} f_E(\underline{y}_k, \underline{o}_k^O) \tag{2.11}$$

With the aid of backpropagation algorithm the weights are determined in such a way as to minimize their mean square error *(MSE).* The process of optimization of the weights is referred to as training or learning of the multilayer perceptron. The backpropagation algorithm will be described briefly in this subsection.

The first step is the initialization of weights of the multilayer perceptron. This is accomplished usually by randomly assigning real values to them mostly from the interval $[-0.01, 0.01]$. In the second step a given input vector $\underline{x}_k$ is applied to the input layer of the multilayer perceptron, and the corresponding output from the network $\underline{o}_k^O$ is computed. This computed output vector $\underline{o}_k^O$ is compared with the corresponding given target vector $\underline{y}_k$ and the squared error is determined according to Eq.(2.9). The next step is the determination of correction weights $\Delta w_{st}^O(k)$ and $\Delta w_{st}^H(k)$ according to Eqs.(2.12) and (2.13)

$$\Delta w_{st}^O(k) = -\eta\frac{\partial E_k}{\partial w_{st}^O} \tag{2.12}$$

$$\Delta w_{rs}^H(k) = -\eta\frac{\partial E_k}{\partial w_{rs}^H} \tag{2.13}$$

The correction weights are defined in each case as being proportional to the partial derivatives of the errors with respect to weights. In broader sense, the backpropogation algorithm is equivalent to a gradient descent method. The factor η with $\eta > 0$ is called the learning rate.

The weights can be corrected at different points in the backpropogation algorithm. In online training the weights w_{st}^O and w_{rs}^H are modified according to Eqs. (2.14) and (2.15) immediately after the processing of a training pattern. This corresponds to a descent in the gradient direction of the error function given by Eq. (2.9).

$$w_{st}^O(\text{new}) = w_{st}^O(\text{old}) + \Delta w_{st}^O(k) \tag{2.14}$$

$$w_{rs}^H(\text{new}) = w_{rs}^H(\text{old}) + \Delta w_{rs}^H(k) \tag{2.15}$$

In offline training the weights w_{st}^O and w_{rs}^H are modified according to Eqs. (2.16) and (2.17) after taking in account of all given m training patterns.

$$w_{st}^O(\text{new}) = w_{st}^O(\text{old}) + \frac{1}{m}\sum_{k=1}^{m} \Delta w_{st}^O(k) \tag{2.16}$$

$$w_{rs}^H(\text{new}) = w_{rs}^H(\text{old}) + \frac{1}{m}\sum_{k=1}^{m} \Delta w_{rs}^H(k) \tag{2.17}$$

The determination of the partial derivatives in Eqs. (2.12) and (2.13) requires a distinction between the hidden layers and output layer [1, 2, 4].

The computation of partial derivatives $\frac{\partial E_k}{\partial w_{st}^O}$ and $\frac{\partial E_k}{\partial w_{rs}^H}$ will be described concisely in this work. The reader is refereed to [4] for detailed explanations of the calculation of these partial derivatives.

In order to compute the partial derivative $\frac{\partial E_k}{\partial w_{st}^O}$ of the error E_k with respect to the elements of the output layer weights, the chain rule is applied according to Eq.(2.18) must be applied.

$$\frac{\partial E_k}{\partial w_{st}^O} = \frac{\partial E}{\partial net^O(t)} \frac{\partial net^O(t)}{\partial w^O(st)} \tag{2.18}$$

Using Eq (2.3) , the term $\frac{\partial net^O(t)}{\partial w^O(st)}$ can be simplied into Eq. (2.19)

$$\frac{\partial net^O(t)}{\partial w^O(st)} = o^H(s) \tag{2.19}$$

and according to the usual notation adopted in literature, the abbreviation $\delta^O(t)$ is used for the term $\frac{\partial E}{\partial net^O(t)}$ and is given by Eq (2.20)

$$\delta^O(t) = -\frac{\partial E}{\partial net^O(t)} \quad = -(y(t) - o^O(t))f_A'(net^O(t) \tag{2.20}$$

Using the above equation the correction weights Δw_{st}^O can then be computed by means of Eq. (2.21)

$$\Delta w_{st}^O = \eta \cdot \delta^O(t) \cdot o^H(s) \tag{2.21}$$

Now similarly in order to compute the partial derivatives $\frac{\partial E_k}{\partial w^H_{rs}}$ of the errors E_k with respect to the elements of the weights of the hidden layer the chain rule according to Eq. (2.22)

$$\frac{\partial E_k}{\partial w^H(rs)} = \frac{\partial E}{\partial net^H(s)} \frac{\partial net^H(s)}{\partial w^H(rs)} \tag{2.22}$$

Similar to the procedure applied for the output layer but using Eq. (2.1), Eq (2.22) simplifies as :

$$\frac{\partial E_k}{\partial w^H(rs)} = \frac{\partial E}{\partial net^H(s)} o^I(r) \tag{2.23}$$

Analogous to the output layer, the abbreviated notation $\delta^H(s)$ is used for the term $\frac{\partial E}{\partial net^H(s)}$ in Eq (2.23), thereby resulting in Eq (2.24)

$$\delta^H(s) = -\frac{\partial E}{\partial net^H(s)} = f_A^{'}(net^H(s)) \sum_{t^{'}=1}^{n_O} \delta^O(t^{'}) w^O(st^{'}) \tag{2.24}$$

The correction weights Δw^H_{rs} for the hidden layer are thus given by Eq. (2.25)

$$\Delta w^H_{rs} = \eta \cdot \delta^H(s) \cdot o^I(r) \tag{2.25}$$

The $\delta^O(t)$ terms of the output layer are necessary in order to determine the $\delta^H(s)$ terms according to Eq. (2.24). For this reason the modification of the correction weights always begins with the output layer and proceeds in the direction of the input layer (backpropogation).

Since backpropogation algorithm is de facto equivalent to a gradient decent method, the problems associated with gradient descent methods must be avoided by applying suitable strategies. The greatest danger in gradient descent methods is that is not able to depart from local minimum and on the other hand it is possible to depart from a detected global minimum in favor of a suboptimum minimum. Moreover, flat plateaus or steep ravines in the error function given by Eq. (2.9) may lead to stagnation or oscillation of the learning process. When applying the backpropogation algorithm the damping or elimination of these problems can be achieved by a simple modification of the method. An proved possibility is the use of a momentum term γ. The corrected weights Δw_i given by Eqs. (2.12) and (2.13) in learning step i are thereby supplemented by the corresponding correction weights $\Delta w_{(i-1)}$ of the $(i-1)$-th learning step according to Eq. (2.26).

$$\Delta w_i = \eta \frac{\partial E}{\partial w} + \gamma \Delta w_{(i-1)} \tag{2.26}$$

Introduction of the momentum γ counteracts stagnations on flat plateaus as well as oscillations between steeply descending regions of the error function. Values between 0 and 1 are recommended for the momentum γ. A random assignment of values of learning rate η and the momentum γ is recommended in each learning step i. The choice of η is highly dependent on the given training data and the architecture of the artificial neural network.

2.1.3 Activation functions

An activation function sets the output behavior of each neuron in an artificial neural network according to Eq. 2.2. They basically decide whether a neuron should be activated or not i.e whether the information the neuron is receiving is relevant for the desired output. Activation functions are crucial to basic architectures of artificial neural networks because they introduce non-linear properties to the network. This enables the artificial neural network to learn from complex, non-linear mappings between input and response variables. In this work, some basic and commonly used activation functions will be described briefly.

Binary step function

The binary step function is extremely simple. It can be applied while creating a binary classifier as we would simply require to say yes or no for a single class. It would either activate the neuron or simply leave it to zero. The function is visualized in Figure (3) and expressed mathematically as

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x > 1 \end{cases} \tag{2.27}$$

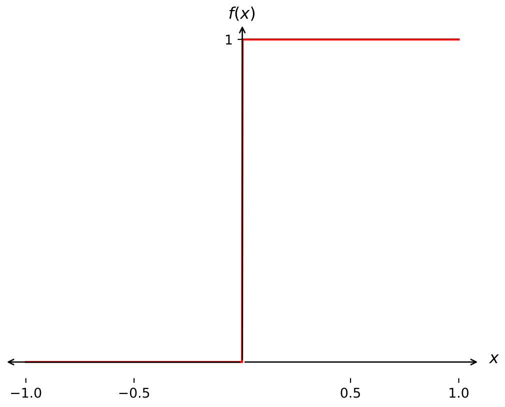


Figure 3: Binary step function

The function is more theoretical since in most cases one would be classifying the data into multiple classes than just a single class and this activation function will not be able to achieve it. Also the gradient of binary step function is zero

$$f'(x) = 0$$

which is not useful during back-propogation as the gradients of activation functions are used for error calculations to improve and optimize results.

Linear function

A linear step function is defined as

$$f(x) = ax \tag{2.28}$$

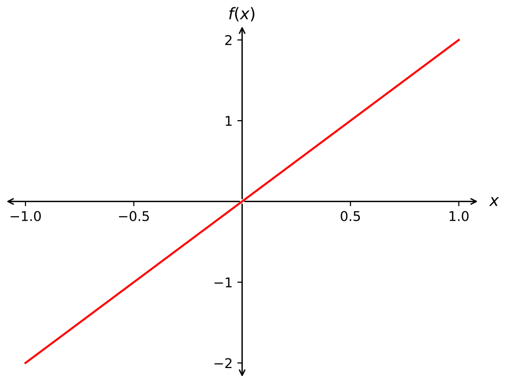


Figure 4: Linear function

We have taken $a = 2$ in Figure (4). Here the activation value is proportional to the input value. This can be applied to various neurons and multiple neurons can be activated at the same time. In case of multiple classes, we can choose from the one which has the maximum value. However the derivative of the linear function is a constant

$$f'(x) = a$$

This indicates that every time we execute back-propogation, the gradient would be the same and we would not be achieving any improvement in error since the gradient is pretty much the same. Thus irrespective of the number of hidden layers the final output will always be a linear transformation of the input. This is not desirable for classifying complex multi-class problems.

Sigmoid function

Sigmoid function is a widely used activation and is mathematically expressed as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.29}$$

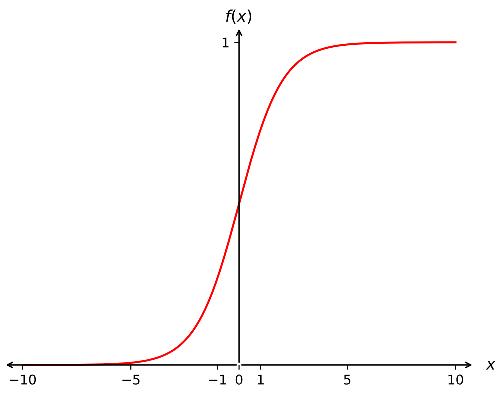


Figure 5: Sigmoid function

From Figure (5) we can see that the function is smooth and it is also continuously differentiable. The major advantage it has over both binary step and linear functions is that it is non-linear. This essentially means that when we have multiple neurons with sigmoid activation function, their output would be non-linear as well. The function values have domain of all real numbers, with return value monotonically increasing from 0 to 1. The gradient of the function is very high between the interval $[-3, 3]$ but gets much flatter in other regions. In this range small changes in x would bring about large changes in value of $f(x)$. So the function essentially tries to push $f(x)$ values towards extremes which is a very desirable quality when we are trying to classify the values to a particular class. On inspection of the gradient of the sigmoid function in Figure (6) we observe it's smooth and dependent on x. This means error can be back propagated easily and the weights can be updated accordingly.

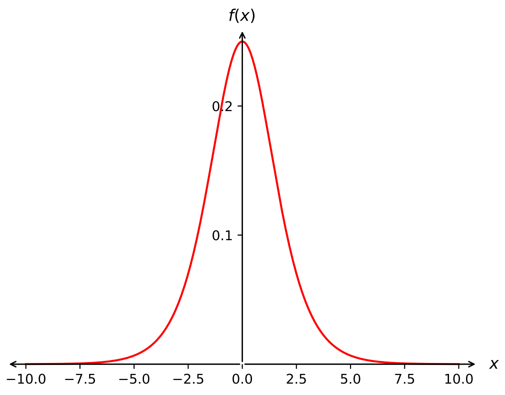


Figure 6: Gradient of sigmoid function

Sigmoid activation functions are widely used today for many problems but still have problems that we need to address. The function is pretty flat in $[-\propto, -3)$ and $(3,\propto]$ regions and once the functions falls in this regions, the gradients become very small. This means the gradient is approaching zero and the network is not really learning anything. Another problem that sigmoid function suffers is that the function values range is $[0, 1]$ which means it is not symmetric around the origin and the values received are all positive. It may not be desirable at all times that the values going to the next neuron to be all of the same sign. This can be addressed by scaling the sigmoid function which is explained in next part.

Tanh function

The tanh function is very similar to sigmoid function, it actually a scaled version of the sigmoid function. Tanh works similar to the sigmoid function but is symmetric over the origin as shown in Figure (7). It can be mathematically expressed as

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.30}$$

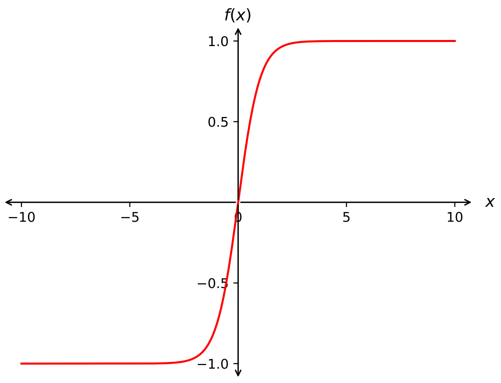


Figure 7: Tanh function

It is continuous and differentiable at all points, also it solves our problem of all values being in the same region in case of sigmoid function. The function as one can see is non-linear so we can easily propagate the errors. The gradient of the tanh function is steeper as compared to sigmoid function (Figure (8)). Hence our choice of using sigmoid or tanh would basically depend on the requirement of gradient in the problem statement. We still have the problem of vanishing gradient as in case of sigmoid function.

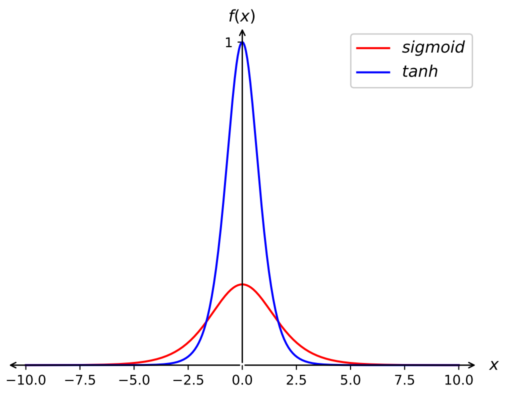


Figure 8: Gradient of Tanh function compared to sigmoid function

ReLU function

The ReLU function is the Rectified Linear Unit and is the most widely used activation function. It is mathematically defined as

$$f(x) = \max(0, x) \tag{2.31}$$

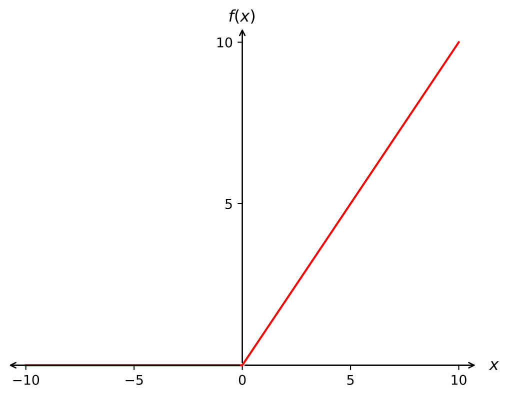


Figure 9: ReLU function

It can be graphically represented as in Figure (9). ReLU function is non-linear and we can easily back propagate the errors and have multiple layers of neurons being activated by the ReLU function. The main advantage of using the ReLU function over other activation function is that it does not activate all the neurons at the same time. If we look closely at the ReLU function, when the input is negative it will transform it to zero and thus the neuron will not get activated. This means that at a time only a few neurons are activated making the network sparse, which is efficient and easy for numerical computation. Figure (10) shows the gradient of ReLU function. We observe ReLU also falls prey to the gradients moving towards zero. If we look at the negative side of the graph, the gradient is zero, which means for activation's in that region would result in zero gradients and consequently weights would not be updated during backpropogation. This can create dead neurons which are never activated.

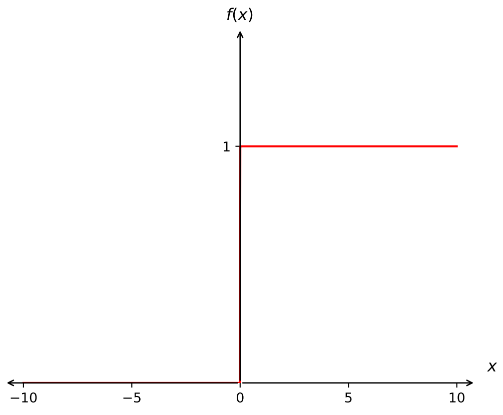


Figure 10: Gradient of ReLU function

In order to circumvent this problem an improved version called Leaky ReLU function can be used. In this instead of defining ReLU function as zero for $x < 0$, we define it as a small linear component of x. It is expressed mathematically as

$$f(x) = \begin{cases} ax & \text{for } x < 0 \\ x & \text{for } x > 0 \end{cases} \tag{2.32}$$

In Leaky ReLU we have simply replaced the horizontal line in Figure (9) with a non-zero, non-horizontal line. Here a is a small value like 0.01 or so. The main advantage of replacing the horizontal line is to remove the zero gradient. In leaky ReLU, the gradient on negative side of the graph is non-zero and hence we no longer encounter dead neurons. The gradient of leaky ReLU would look like in Figure .

Similar to leaky ReLU function, we also have Parameterised ReLU function. It is defined similar to leaky ReLU, but the difference is that the parameter a in Eq (2.32) is a trainable parameter. The artificial neural network as learns the value of a for faster and optimum convergence. The Parameterised ReLU function is used when leaky ReLU functions fails to solve the problem of dead neurons which leads to relevant information not being passed to next layer in a meaningful manner.

2.1.4 Loss functions

A loss function is a measure of "how good" a neural network did with respect to it's given training samples and the expected output. It is a single value, not a vector, because it rates how good the neural network did as a whole. In Subsection 2.1.2 we have seen how artificial neural networks use loss functions for learning. In this subsection, mathematical expressions of several commonly used loss functions will be explained briefly. In order to improve transparency and for ease of understanding, $\underline{\hat{y}}$ is the predicted value by the artificial neural network, $\underline{y}$ is the target vector and the loss function is represented by E. For detailed mathematical explanation, the reader is referred to [5] . This subsection gives comprehensive explanation about several commonly used loss functions.

Mean squared error

Mean Squared Error (MSE), or quadratic loss, is widely used in linear regression as a performance measure, and the method of minimizing MSE is called Ordinary Least Square (OLS). The basic principle of OLS is that the optimized fitting line should be a line which minimizes the sum of distance of each point to the regression line. The standard form of MSE loss function is defined as

$$E = \frac{1}{m}\sum_{i=1}^{m}(y_i - \widehat{y}_i)^2 \tag{2.33}$$

where $(y_i - \widehat{y}_i)$ is named as residual, and the target of MSE loss function is to minimize the residual sum of squares. When we use Sigmoid activation fucntion, the quadratic loss function would suffer from the problem of slow convergence (learning speed).

Mean squared logarithmic error

Mean Squared logarithmic Error (MSLE) loss function is a variant of MSE, which is defined as

$$E = \frac{1}{m}\sum_{i=1}^{m}(\log(y_i + 1) - \log(\hat{y}_i + 1))^2 \tag{2.34}$$

MSLE actually measures the change in variance. It is usually used when we do not want to penalize huge differences in the predicted and the actual values when both predicted and true values are huge numbers. It penalizes under-estimates more than over estimates.

L2

L2 loss function is the square of the L2 norm of difference between actual value and predicted value. It is mathematically similar to MSE, but we do not have division by m, it is computed by

$$E = \sum_{i=1}^{m} (y_i - \widehat{y}_i)^2 \tag{2.35}$$

Mean absolute error

Mean Absolute Error (MAE) is a quantity to measure how close predictions are to the actual targets, which is computed by

$$E = \frac{1}{m} \sum_{i=1}^{m} \mid y_i - \widehat{y}_i \mid \tag{2.36}$$

where $\mid \cdot \mid$ denotes the absolute value. Even though both MSE and MAE are used in predictive modeling, there are several differences between them. MSE has nice mathematical properties which makes it easier to compute gradient unlike MAE, which requires complicated tools like linear programming for gradient computation. Large errors have relatively greater influence on MSE than smaller errors because of square function. MAE is robust to outliers since it does not make use of square function.

L1

L1 loss function is sum of absolute errors of the difference between actual value and predicted value. Similar to relation between MSE and L2, L1 is mathematically similar to MAE, only we don not have division with m. It is defined as

$$E = \sum_{i=1}^{m} \mid y_i - \widehat{y}_i \mid \tag{2.37}$$

Kullback Leibler (KL) divergence

KL Divergence, also known as relative entropy, is a measure of how one probability distribution diverges from second expected probability distribution. KL divergence loss functions is computed by

$$\begin{aligned} E &= \frac{1}{m} \sum_{i=1}^{m} \mathcal{D}_{KL}(y_i \,\|\, \widehat{y}_i) \\ &= \frac{1}{m} \sum_{i=1}^{m} \left[y_i \cdot \log \left(\frac{y_i}{\widehat{y}_i} \right) \right] \\ &= \frac{1}{m} \sum_{i=1}^{m} \left(y_i \cdot \log \left(y_i \right) \right) - \frac{1}{m} \sum_{i=1}^{m} \left(y_i \cdot \log \left(\widehat{y}_i \right) \right) \end{aligned} \tag{2.38}$$

where the first term is entropy and second term is cross entropy (another kind of loss function which will be explained later). KL divergence is a distribution-wise asymmetric measure and thus does not qualify as a statistical metric of spread. In a simple case, a KL divergence of 0 indicates that we can expect similar, if not the same, behavior of two different distributions, while a KL divergence of 1 indicates that the two distributions behave in such a different manner that the expectation given the first distribution approaches zero.

Cross entropy

Cross Entropy is commonly used in binary classification (labels are assumed to take values 0 or 1) as loss function (For multi-classification, we use Multi-Class Cross Entropy), which is computed by

$$E = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log\left(\widehat{y}_i\right) + \left(1 - y_i\right)\log\left(1 - \widehat{y}_i\right)\right] \tag{2.39}$$

Cross entropy measures the divergence between two probability distribution. If cross entropy is large, it means that the difference between two distribution is large, while if cross entropy is small, then the two distributions are similar to each other. We have already seen that MSE suffers from slow convergence when using sigmoid activation function but cross entropy does not have such a problem with sigmoid activation function.

Negative logarithmic likelihood

Negative Log Likelihood loss function is widely used in classifiers, when the model outputs a probability for each class rather than just the most likely class. It is “soft” measurement of accuracy that incorporates the idea of probabilistic confidence. Negative log likelihood is computed by

$$E = -\frac{1}{m}\sum_{i=1}^{m}\log\left(\widehat{y}_i\right) \tag{2.40}$$

Poisson

Poisson loss function is a measure of how the predicted distribution diverges from the expected distribution. The Poisson as a loss function is a variant from Poisson Distribution, which is widely used for modeling count data. The Poisson loss function is computed by

$$E = \frac{1}{m}\sum_{i=1}^{m}\left(\widehat{y}_i - y_i \cdot \log\left(\widehat{y}_i\right)\right) \tag{2.41}$$

Cosine proximity

Cosine Proximity loss function computes the cosine proximity between predicted value and actual value, which is defined as

$$E = -\frac{\mathbf{y} \cdot \hat{\mathbf{y}}}{\parallel \mathbf{y} \parallel^{\mathbf{2}} \cdot \parallel \hat{\mathbf{y}} \parallel^{\mathbf{2}}} = -\frac{\sum_{i=1}^{m} y_i \cdot \widehat{y}_i}{\sqrt{\sum_{i=1}^{m} (y_i)^2} \cdot \sqrt{\sum_{i=1}^{m} (\widehat{y}_i)^2}} \tag{2.42}$$

where $\mathbf{y} = \{y_1, y_2, \dots, y_m\} \in \mathbb{R}^m$, and $\hat{\mathbf{y}} = \{\widehat{y}_1, \widehat{y}_2, \dots, \widehat{y}_m\} \in \mathbb{R}^m$. It is same as Cosine Similarity, which is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them. In this case, the unit vectors are maximally “similar” if they are parallel and maximally “dissimilar” if they are orthogonal.

Hinge

Hinge Loss, also known as max-margin objective, is a loss function mainly used for training classifiers. The hinge loss is used for “maximum-margin” classification, most notably for Support Vector Machines (SVMs). It is defined as

$$E = \frac{1}{m} \sum_{i=1}^{m} \max(0, 1 - y_i \cdot \widehat{y}_i) \tag{2.43}$$

Here $\widehat{y}_i$ is the “raw” output of the classifier’s decision function and not the predicted class label.

2.2 Convolutional neural networks

Convolutional networks [6], also known as convolutional neural networks, or CNN's, are a specialized kind of artificial neural network for processing data that have a known grid like topology. Examples of data on which CNN's can be employed include time series data, which can be thought of as 1-D taking samples at regular time intervals, and image data which can be thought of as a 2-d grid of pixels. The name "convolutional neural networks" indicates that the network employs a mathematical operation called convolution. Convolutional neural networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of it's layer. In this subsection, we first describe what convolution is. Next, we describe an operation called pooling, which almost all convolutional networks employ. Research into convolutional network architecture proceeds so rapidly that we have a new best architecture for a given benchmark announced every few months, making it impractical to to describe the best architecture. However, all the architectures have mainly been composed of the building blocks described in this section. A typical layer of a convolutional network consists of three stages as shown in Figure (11). In the first step, a layer performs several **convolutions** in parallel to produce a set of linear activation's. In the second step, each linear activation is run through a nonlinear activation function, mostly the rectified linear activation function. This step is sometimes called the **detector layer**. In the third step, we use a **pooling function** to modify the output of the layer further.

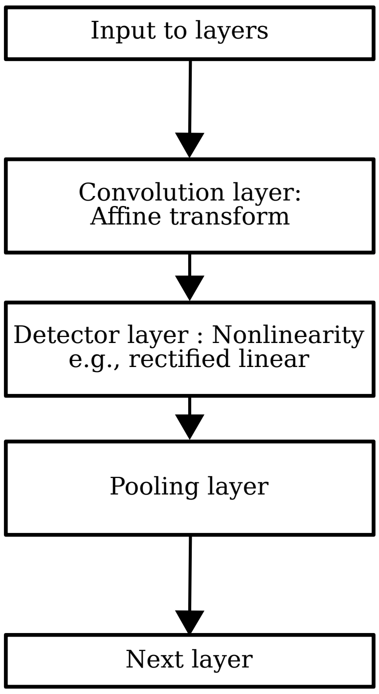

Input to layers
Convolution layer: Affine transform
Detector layer : Nonlinearity e.g., rectified linear
Pooling layer
Next layer

Figure 11: Components of typical convolutional neural network layer

2.2.1 Convolution operation

Convolution is an operation on two functions of a real-valued argument. To better understand the definition of convolution, we make use of an example of two functions.

Suppose we are tracking the location of a body with a laser sensor. The sensor provides a single output $x(t)$, the position of the body at time t, where both x and t are real valued. Assume that our sensor is somewhat noisy, so to obtain a less noisy estimate of the body's position, we can average several measurements. To make recent measurements more relevant we will employ a weighted average that gives more weight to recent measurements. We can also do this with a weighting function $w(a)$, where a is the weight of the measurement. When we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the body.

$$s(t) = \int x(a)\, w(t-a)\, da \tag{2.44}$$

This operation is called convolution. The convolution operation is typically denoted with asterisk:

$$s(t) = (x * w)(t) \tag{2.45}$$

In our example, w has certain limitations which are particular for the example only. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages. In convolutional neural network terminology, the first argument (in our example, the function x) to the convolution is often referred to as the **input**, and the second argument (in our example, the function w) as the **kernal**. The output is often referred to as the **feature maps**.

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant representations** [7]. We will describe each of these ideas briefly.

Traditional neural networks employ matrix multiplication of a matrix of parameters with a separate parameter describing the interaction between each input and output unit. This indicates that every output unit interacts with every input unit. Convolutional networks, however, typically employ **sparse interactions** (also referred to as **sparse connectivity** or **sparse weights**). This is accomplished by making the kernel smaller than the input. To understand this concept let us take an example of processing an image. The image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges or curves with kernels that occupy only tens or hundreds of pixels. This would mean that we only need to store fewer parameters, which reduces the memory requirements of the model and improves it's statistical efficiency. It would also mean that computing the output requires fewer operations. Usually this improvements in efficiency is quite large. Suppose if there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ run time. If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ run time. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m.

Parameter sharing refers to using same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is exactly used once when computing the output of the layer. It is multiplied by one element of the input and never used again. Another term one might encounter of parameter sharing in various literature's is that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a Convolutional neural network, each member of the kernel is used in every position of the input. This parameter sharing used by convolution operation means that we learn only one set of parameters for every location rather than learning a separate set of parameters for every location. This does not affect the run-time of forward propagation - it is still $O(k \times n)$ - but it reduces the storage requirements of the model from n to k parameters. k is usually several orders of magnitude smaller than m. In most cases, m and n are roughly the same size, hence $m \times k$ is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of memory requirements and statistical efficiency.

Also in case of convolution, the particular form of parameter sharing causes the layers to have a property called **equivariance** to translation. A function is said to be equivariant when it's output changes in the same way as the input changes. For example, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g. Convolution is not naturally equivariant to some other transformations, such as changes in scale or rotation of the input data. Other mechanisms are necessary to tackle these kinds of transformations. Finally, some kinds of data cannot be evaluated by neural networks defined by matrix multiplication with a fixed shaped matrix. Convolution enables processing of some of these kinds of data.

2.2.2 Pooling

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. The **max pooling** [8] operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L^2 norm of a rectangular neighborhood, or a weighted average based on a distance from a central position. In all cases, pooling helps to make the representation approximately **invariant** to small translations of the input. The use of pooling can be viewed as adding an infinitely strong prior that the function the layers learn must be invariant to small translations. When this assumption is correct, it can greatly improve that statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the network can learn which transformations to be invariant to. Since pooling summarizes the responses over the whole neighborhood, it is possible to use fewer pooling units than detector units. This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of it's input (when the next layer is fully connected and based on matrix multiplication), this reduction in input size can also result in statistical efficiency and reduced memory requirements for storing parameters.

For many tasks, pooling is essential for handling inputs of varying size. Pooling can complicate some kinds of network architectures that use top-down information, such as Boltzmann machines and autoencoders. Some theoretical works gives us guidance as which kinds of pooling one should use in various situations [9].

With section 2.1 and 2.2, we have tried to provide a comprehensive explanation about basics of ANNs and the terminologies used in modern deep learning architectures. In further sections, we will try to provide a short mathematical explanation of the concepts used while generating required data to be used within this work.

2.3 Barycentric coordinates

In geometry, the barycentric coordinate system is a coordinate system in which the location of a point of a simplex (a triangle, tetrahedron, etc.) is specified as the center of mass, or barycenter, of usually unequal masses placed at it's vertices. The system was introduced in 1827 by August Ferdinand Möbius [10].

Consider a set of points $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ and consider the set of all affine combinations taken from these points, i.e , all points P can be written as

$$\alpha_0\mathbf{P}_0 + \alpha_1\mathbf{P}_1 + \cdots + \alpha_n\mathbf{P}_n \tag{2.46}$$

for some

$$\alpha_0 + \alpha_1 + \cdots + \alpha_n = 1 \tag{2.47}$$

Then this set of points forms an affine space, and the coordinates

$$(\alpha_0, \alpha_1, \ldots, \alpha_n)$$

are called the barycentric coordinates of the points of the space [11]. These coordinates system is frequently useful and extensively used in working with triangles. This barycentric parameterization is exactly the parameterization that is usually used in many cases and in our case for generating 3D geometric data.

In context of this work, we will make use of barycentric coordinates to generate points in a triangle. Consider three points P_1, P_2, P_3 in the plane. If $\alpha_1, \alpha_2, \alpha_3$ are scalars such that $\alpha_1 + \alpha_2 + \alpha_3 = 1$, then the point P defined by

$$P = \alpha_1 \mathbf{P}_1 + \alpha_2 \mathbf{P}_2 + \alpha_3 \mathbf{P}_3 \tag{2.48}$$

is a point on the plane of the triangle formed by P_1, P_2, P_3. The point is within the triangle $\Delta P_1 P_2 P_3$ if

$$0 \leq \alpha_1, \alpha_2, \alpha_3 \leq 1 \tag{2.49}$$

If any of the α's is less than zero or greater than one, then the point P is outside the triangle. If any of the α's is zero then P is on one of the lines joining the vertices of the triangle. Figure (12) shows an example of such a triangle and three points P, Q and R were calculated using the following α's :

- $P : \alpha_1 = \alpha_2 = \frac{1}{4}, \alpha_3 = \frac{1}{2}$
- $Q : \alpha_1 = \frac{1}{2}, \alpha_2 = \frac{3}{4}, \alpha_3 = -\frac{1}{4}$
- $R : \alpha_1 = 0, \alpha_2 = \frac{3}{4}, \alpha_3 = \frac{1}{4}$

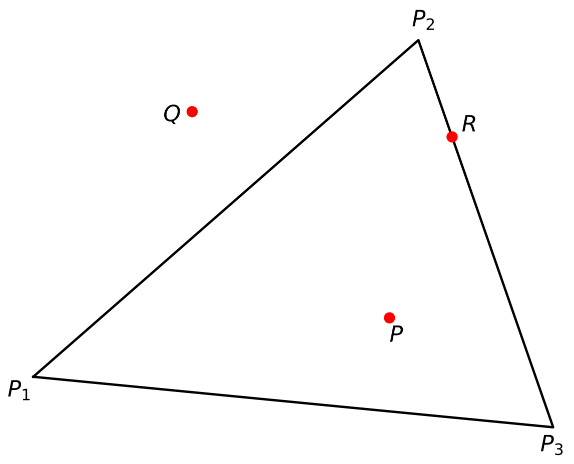


Figure 12: Barycentric coordinates

In this work, we would utilize the concept of Barycentric coordinates to generate points inside elements of a part. In order to achieve this we would need to sample combinations of $\alpha_1, \alpha_2, \alpha_3$ so that we can generate points inside elements that are uniformly distributed. In order to do, we use the concept of Latin Hypercube sampling.

2.4 Latin hypercube sampling

Latin Hypercube Sampling (LHS) is a method of sampling a model input space, usually for obtaining data for training meta models. In the context of statistical sampling, a square grid containing sample positions is a Latin square if and only if there is only one sample in each row and each column. A Latin hypercube is the generalization of this concept to arbitrary number of dimensions.

When sampling a function of N variables, the range of each variable is divided into M equally probable intervals. M sample points are then placed to satisfy the Latin hypercube requirments. By representing each variable as its Cumulative Distribution Function (CDF) and partitioning the CDF into M regions and taking a simple sample from each region., increases the likelihood that entire range of the function is sampled. A requirement for LHS is that each region of the CDF can only be sampled once for each parameter. This is best visualized in a 2D space with Figure (13).

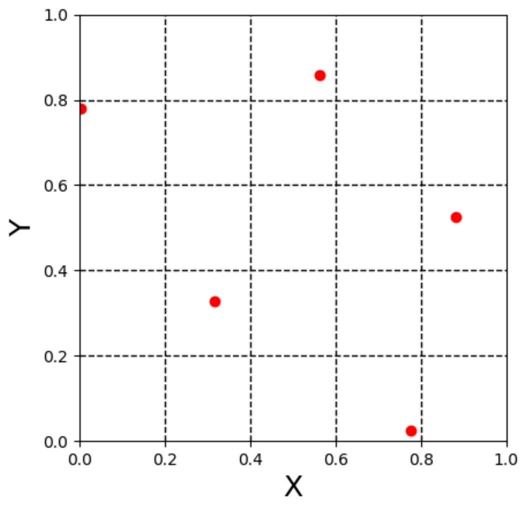


Figure 13: LHS 2D sample

As seen in Figure (13), there is only one sample in each row and column in (X, Y) space. Using LHS technique, we sample α's and then using these α's we generate points inside elements using Eq. (2.48). When we generate more than one point inside an element, we do not want clustering of points in certain regions. In this case we employ LHS sampling to produce fairly equally distributed points inside the element.

3 Machine learning approaches for classification of geometric data

3.1 Theoretical aspects of 3D geometric data

The raw 3D data that are captured by different methods come in forms that vary in both, the structure and properties. This section presents a comprehensive information on different representations of 3D data by categorizing them into two main families: Euclidean data and non-Euclidean data [12].

3.1.1 Euclidean-structured data

Certain 3D representations have an underlying Euclidean-structure where the properties of the grid-structured are preserved such as having a global parameterization and a common system of coordinates. The main 3D representations that fall under this category are:

Descriptors

A shape descriptor is a simplified representation of the 3D object to describe geometric or topological characteristics of a 3D object. Shape descriptors are usually obtained from the object's surface,texture, depth or any other characteristic or a combination of all [13]. Kamzi et al.[13] provides comprehensive surveys about 3D shape descriptors. Shape descriptors can be considered as a signature of the 3D shape, which is presented in numeric values to ease processing and computations. Figure shows an example of descriptor,

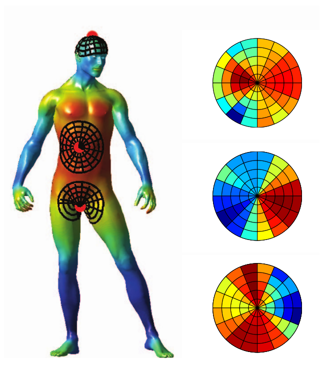


Figure 14: Descriptor, which is constructed as a histogram of some field, defined on an intrinsic local polar coordinate system (left, shown at three different points) [14]

3D data projections

Projecting 3D data into 2D space is another representation for raw 3D data, in which the projected data captures some of the key properties of the original 3D shape. The type of features captured is dependent on the type of projection. Projecting 3D data into spherical and cylindrical domains [15] have been a common practice for representing such format. However, such representations are not optimal for complicated 3D computer vision tasks due to information loss in projection [16].

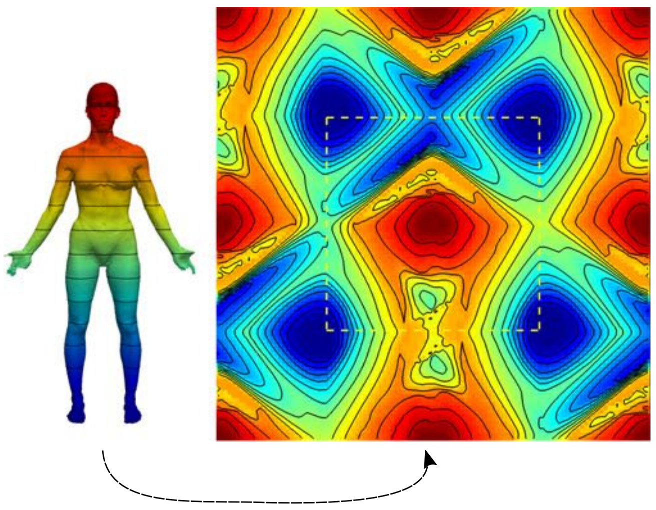


Figure 15: Projections [14]

RGB-D data

Due to the popularity of RGB-D sensors such as Microsoft's Kinect [17], representing 3D data as RGB-D images have become popular in recent years. RGB-D data provides information about the captured 3D object using a depth map (D) along with 2D color information (RGB). Besides being inexpensive, RGB-D data are simple yet effective representations for 3D objects to be used for different tasks such as identity recognition [18], pose regression [19] etc. Also the number of available RGB-D datasets is huge when compared to other 3D datasets [20].

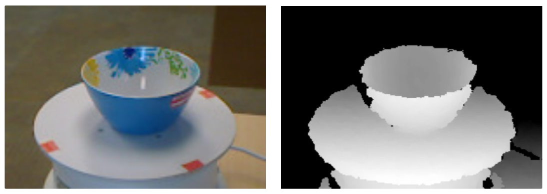


Figure 16: Each RGB-D fame consist of and RGB image (left) and corresponding depth image (right) [21]

Volumetric data

3D data can be represented as a regular grid in the 3D space. Voxels are used to model 3D data by describing how the 3D object is distributed through the three dimensions of space. Viewpoint information about the 3D shape can be encoded by classifying the occupied voxels into visible, occluded or self-occluded. Despite the simplicity, the voxel based representation is not always efficient and not suitable for representing high resolution data [22].

A more efficient 3D volumetric representation is octree based [22], which is essentially varying sized voxels. It models 3D objects as a hierarchical data structure that divides the 3D scene into cubes that are either inside or outside of the object. Despite the simplicity of forming 3D octrees, they are powerful in representing the fine details of 3D objects as compared to voxels. However, both voxels and octree representations do not preserve the geometry of 3D objects in terms of the intrinsic properties of the shape and the smoothness of the surface.

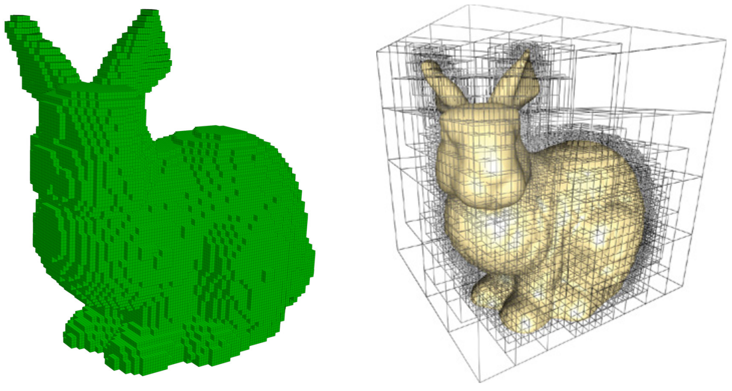

(a) Voxels[23] (b) Octree [24]

Figure 17: Volumetric data

Multiview data

3D data can be represented as a combination of multi-view 2D images for the same object from different point of views [25]. Representing 3D data in this manner allows learning multiple feature sets to reduce the effect of noise, incompleteness and illumination problems on the captured data. However, the question of how many views are sufficient to model the 3D shape is still open. Representing the 3D object with an insufficiently small number of views might not capture the intrinsic properties of the whole 3D shape. Also too many views would cause an unneeded computational overhead. Multiview data is also not capable to capturing intrinsic shape properties. However, learning well represented multi-view data has proved better performance over learning 3D volumetric data [26].

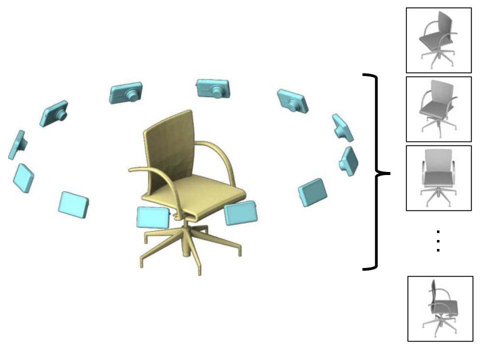


Figure 18: Multiview data [27]

3.1.2 Non-Euclidean data

An another type of 3D data representations is the non-Euclidean data. This type of data representations suffers from the absence of global parametrization and the non-existence of a common system of coordinates or vector space structures [28], which makes processing of such data representations a challenging task. Considerable efforts are directed towards learning from such data and applying machine learning techniques on it. The main type of non-Euclidean data is point clouds, 3D meshes and graphs. Usually processing such data types happens on a global scale to learn the whole 3D object's feature which is convenient for complex tasks such as recognition and correspondence.

3D Point clouds

A point cloud can be seen as a set of unstructured 3D points that represents the geometry of the 3D object. Such a realization makes it a non-Euclidean geometric data representation. However, point clouds can also be realized as a set of small Euclidean subsets that have a global parameterization and a common system of coordinates and invariant to transformations such as translation or rotation. So the definition of the point cloud's structure depends on whether one is considering the global or local structure of the object. Since in most machine learning techniques strive for capturing the whole features of the object to perform complex tasks of recognition, correspondence, matching or retrieval; we classify point clouds as non-Euclidean data

Despite the ease of capturing point clouds using available technologies such as Kinect [17] and structured light scanners [29], processing them presents a challenging task due to some problems related to their data structure. The data structure related problems mainly arise due to lack of connectivity information in the point clouds, which makes the surface information ambiguous. Furthermore, environment related problems are usually present in real acquisition setup, i.e, the raw captured point clouds suffer from noise, incompleteness, irregular sampling density and sometimes holes and missing parts.

Motivated by the use of point clouds in multiple computer vision tasks e.g. 3D reconstruction [30], object recognition [31] and vehicle detection [32], a lot of work has been done on processing point clouds for noise reduction.

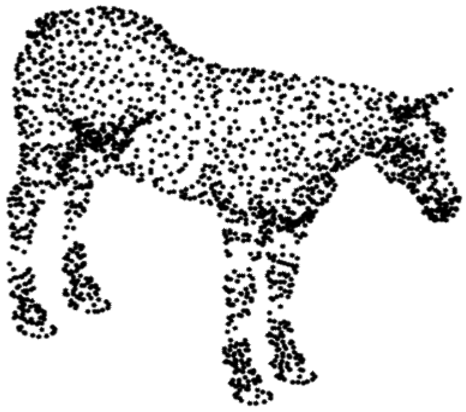
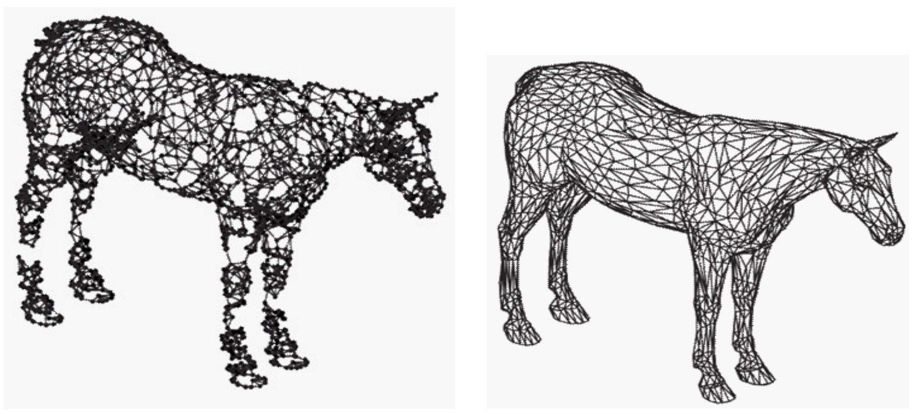


Figure 19: Point cloud[33]

3D meshes and graphs

3D meshes are one of the most popular representations for 3D shapes. A 3D mesh structure consists of a set of polygons called faces described in terms of a set of vertices that describe how the mesh coordinates exist in the 3D space. These vertices are associated with a connectivity list which describes how these vertices are connected to each other. Learning 3D meshes is a challenging task because of two main reasons: DeepLearning methods haven't been readily extended to such irregular representations besides such data also suffers from noise, missing data and resolution problems [34].



(a) 3D graph (b) 3D mesh

Figure 20: 3D meshes and graphs [33]

3.2 Deep learning on 3D data

3D data has multiple popular representations as described briefly in Section (3.1), leading to various approaches for learning. In this section we will briefly discuss some of the significant machine learning approaches applied to them.

Volumetric CNNs: [35, 36, 37] are the pioneers applying 3D convolutional neural networks on voxelized shapes. But as we know, volumetric representation is constrained by it's resolution due to data sparsity and computation cost of 3D convolution. FPNN [38] and Vote3D [39] has proposed methods to deal with the sparsity problem; however, it's challenging for them to process very large point clouds.

Multi-view CNNs: [40, 37] have tried to render 3D shapes into 2D images and then apply 2D convolutional networks to classify them. This approach has achieved dominating performance on shape classification and retrieval tasks due to the fact that image CNN's are well engineered for classification tasks [41]. However the question of how many views are required to model the 3D shapes are still open.

Spectral CNNs: Some of the latest approaches [42, 43] use spectral CNNs on meshes. However, these methods are currently constrained on manifold meshes such as organics objects and it's not obvious how to extend them to non-isometric shapes.

Feature-based DNNs: [44, 45] extracts traditional shape features from 3D data and converts it into a vector. Then they use a fully connected network to classify the shape. This approach is constrained by the representation power of the features extracted.

3.3 PointNet

In this work, we explore deep learning architectures capable of reasoning about 3D geometric data of point clouds. Point clouds are simple and unified structures that avoid the combinatorial irregularities and complexities of meshes and are thus easier to learn from. The network named PointNet [46], provides a unified architecture for applications ranging from object classification, part segmentation, to scene parsing. Though simple, PointNet is highly efficient and effective. Empirically, it shows strong performance on par or even better than state of the art. In this section we will discuss about the approach of using this deep learning architecture for geometry classification.

PointNet is a unified architecture that directly takes point clouds as input and outputs either class labels for the entire input or per point segment/part labels for each point of the input. The basic architecture of the network is surprisingly simple as in the initial stages each point is processed identically and independently. A point cloud is represented as a set of 3D points $\{P_i \mid i = 1, \ldots, n\}$ where each point P_i is a vector of it's (x, y, z) coordinate plus extra feature channels such as color, normal etc. For simplicity and clarity, we only use the (x, y, z) coordinates as our input.

The architecture of the network is inspired by three main properties of point sets in $\mathbb{R}^n$:

- *Unordered.* Unlike pixel arrays in images or voxel arrays in volumetric grids, point cloud is a set of points without specific order. In other words, a network that consumes N 3D points sets need to be invariant to $N!$ permutations of the input set in data feeding order.
- *Interaction among points.* The points are from a space with a distance metric. It means that points are not isolated, and neighboring points form a meaningful subset. Therefore, the model should be able to capture local structures from nearby points, and the combinatorial interactions among local structures.
- *Invariance under transformation.* The learned representation of the point set should be invariant to certain transformations. For example, rotating or translating points all together should not modify the global point cloud category.

The network architecture, visualized in Figure (21), outputs k scores for all the k candidate classes. Please read the caption of Figure (21) for the pipeline.

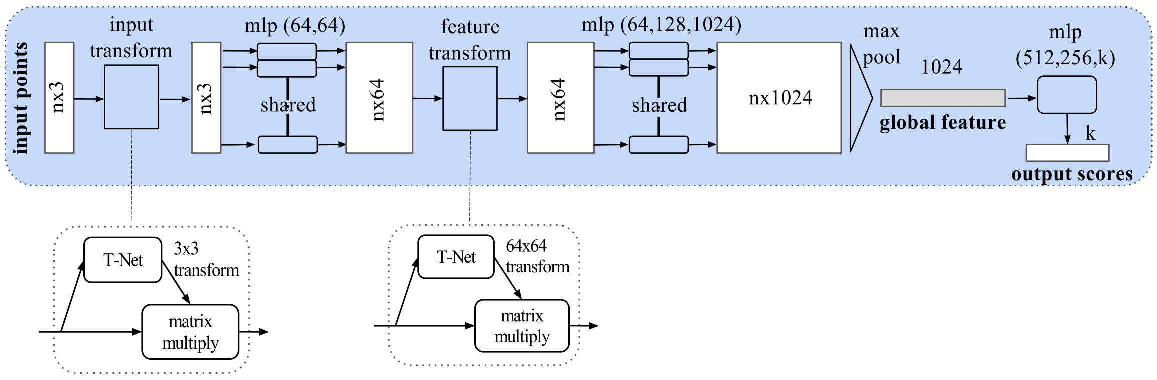
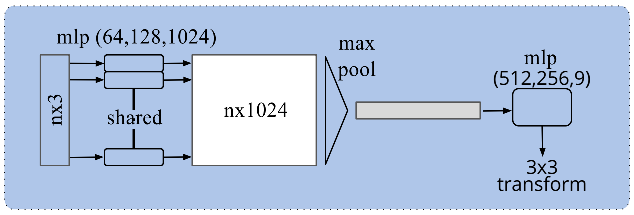
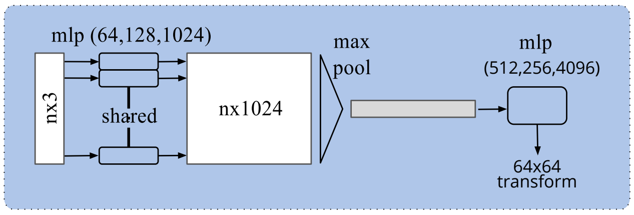


Figure 21: **PointNet Architecture** [46]. The classification takes n points as input, applies input and feature transformations, and then aggregates point features by max pooling. The output is classification scores for k classes. The input transform and feature transform network are visualized in Figure (22)

The network has three key modules: the max pooling layer as a symmetric function to aggregate information from all the points, a local and global information combination structure, and two joint alignment networks that align both input points and point features. The reason behind this design choices are discussed as follows:



(a) input transform network



(b) feature transform network

Figure 22: Transform Networks

Symmetry function for unordered input. In order to make a model invarient to input permutation, three strategies exist:

1. sort input into a canonical order;
2. treat the input as a sequence to train a Recurrent Neural Network, but augment the training data by all kind of permutations;
3. use a simple symmetric function to aggregate the information from each point.

Here, a symmetric function takes n vectors as inputs and outputs a new vector that is invariant to the input order. $+$ and $*$ operators are symmetric binary functions. While sorting sounds like a simple solution, but in high dimensional space there does not exist an ordering that is stable w.r.t point perturbations in the general sense. The idea to use RNN considers the point set as a sequential signal and hopes that by training the RNN with randomly permuted sequences, the RNN will become invariant to input order. However in “OrderMatters” [47] the authors have shown that order does matter and cannot be totally omitted. In PointNet, the idea is to approximate a general function defined on a point set by applying a symmetric function on transformed elements in the set:

$$f\left(\{x_1, \dots, x_n\}\right) \approx g\left(h\left(x_1\right), \dots, h\left(x_n\right)\right) \tag{3.1}$$

where $f : 2^{\mathbb{R}^n} \rightarrow \mathbb{R}$, $h : \mathbb{R}^n \rightarrow \mathbb{R}^K$ and $g : \underbrace{\mathbb{R}^{\mathbb{K}} \times \dots \times \mathbb{R}^{\mathbb{K}}}_{n} \rightarrow \mathbb{R}$ is a symmetric function. Empirically, the basic module is very simple. It approximates h by a multi-layer perceptron network and g by a composition of a single variable function and a max pooling function. This is found to work well by experiments in [46]. Through a collection of h, the architecture can learn a number of f's to capture different properties of the set.

Local and Global information aggregation The output from the above section forms a vector $[f_1, \ldots, f_k]$ which is a global signature of the input set. It can easily be used to train a Support Vector Machine or multi-layer perceptron classifier on the shape global features for classification.

Joint alignment network The semantic labeling of a point cloud has to be invariant if the point cloud undergoes certain geometric transformations, such as rotation or translation. The PointNet architecture predicts an affine transformation matrix by a mini-network (T-net in Figure (22)) and directly apply this transformation to the coordinates of input points. The mini-network itself resembles the big network and is composed by basic modules of point independent feature extraction, max pooling and fully connected layers. This idea is further extended to the alignment of feature space, as well. It inserts another alignment network on point features from different input point clouds. The transformation matrix in the feature space has much higher dimension than the spatial transform matrix which greatly increases the difficulty of optimization. To overcome this a regularization term is added to the softmax training loss. It constrains the feature transformation matrix to close to orthogonal matrix:

$$L_{reg} = \| I - AA^T \|_F^2 \tag{3.2}$$

where A is the feature alignment matrix predicted by the mini-network. An orthogonal transformation will not lose information in the input.

The PointNet Architecture is implemented in TensorFlow, which is a open source machine learning frame work. In this work we used Python 2.7 to generate training data and Python 3.5 to train the PointNet architecture [48]. In order to speed up the training cycle we used a GPU (Graphics Processor Unit). TensorFlow allows us to use GPU's to train machine learning algorithms with the aid of CUDA and cuDNN. CUDA is parallel computing platform and programming model invented by NVIDIA. It enables dramatic increase in computing performance by harnessing the power of GPU [49]. The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU- accelerated library of primitives for deep neural networks [50]. cuDNN provides highly tuned implementations for standard routines such as forward and back ward convolution, pooling, normalization , and activation layers. In this work we we utilized CUDA 9.0 and cuDNN 7.0 for training purpose.

The training data for PointNet was stored in HDF5 binary data format. Python package "h5py" was used to achieve this. HDF5 is a high-performance data management and storage suite. It supports n-dimensional datasets and each element in the data set can itself be complex object. It comes with a set of integrated performance features that allow for access time and storage space optimization's. Also there is no limit on the number or size of data objects in the collection, giving great flexibility for big data. In this work we would be working with storing huge amount of geometric data as point clouds and hence HDF5 was chosen.

Once we have generated training data, we train a model [48] to classify point clouds:

```
$ python train.py
```

Log files and network parameters will be saved to log folder in default. We can use TensorBoard to visualize the network architecture and monitor the training progress.

```
$ tensorboard --logdir log
```

Once the training is complete, the trained model can be evaluated and we can check visualizations of the error case

```
$ python evaluate.py --visu
```

3.4 Generation of 3D geometric data from FEM data

In the previous, a deep learning architecture is explained briefly which consumes point clouds and can be used to classify 3D geometry. In this section ideas and algorithms used to generate point clouds of 3D geometry for parts of car. Firstly we will have a look at how part geometry is expressed in FEM data. Then we will discuss methods to extract part geometry for individual parts and then using this part geometry data we will generate point clouds. The approaches presented were developed using libraries developed by SCALE Gmbh.

3.4.1 Extraction of geometry data from FEM data

In this subsection, we will discuss the ideas used to extract geometry data from FEM data. Nodes are the base for expressing geometric data in FEM data. Elements are formed with nodes and using elements we express part geometry. So when we want to extract geometric data, it indicates that we are extracting the coordinates of nodes for elements in a part. Now one can argue that we can just extract data of nodes present in a part and use it as point cloud of a part. This would be a simple process but we know that for feeding this point cloud into PointNet we would require the number of points in point cloud for each part to be the same. In Body in white (BIW) of a car not all parts are of same size and hence the number of nodes used to express the geometries of different parts will be obviously different. Hence it becomes necessary to extract data associated with elements that make up the parts and then use this data to generate points on surface of the parts according to our need. In this work we will only encounter with planar shell elements and thus the further discussions on elements will consider only the properties of shell elements.

Figure (24) shows the example of include deck for LS-Dyna. Each part is associated with a unique part identification number called the **PID**. Every node present in the model is also identified using a unique node identification number called **NID**. As seen in the Figure the ***NODE** card holds the geometric data i.e (x, y, z) coordinates. The elements are defined using NID and PID. Using this relationship we can easily extract the geometric data for each part from the whole model. SCALE Gmbh has developed a python library called **femparser** with which one could parse the include decks of FEM data and extract geometric data. Figure 23 depicts the work flow for geometric data extraction process which was implemented for this work. The output of femparser is a python object which can be used to extract the geometric data of individual parts using combination of part numbers, elements ID's and node ID's. The extraction algorithm is presented in Algorithm (1). The output is stored in **JSON** format files for the ease of reading the outputs and for futher processing steps (i.e generation of point clouds)

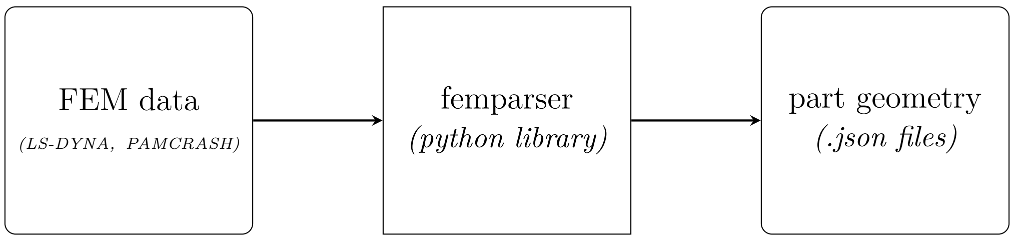


Figure 23: Geometric data extraction

Algorithm 1 Part geometry extraction

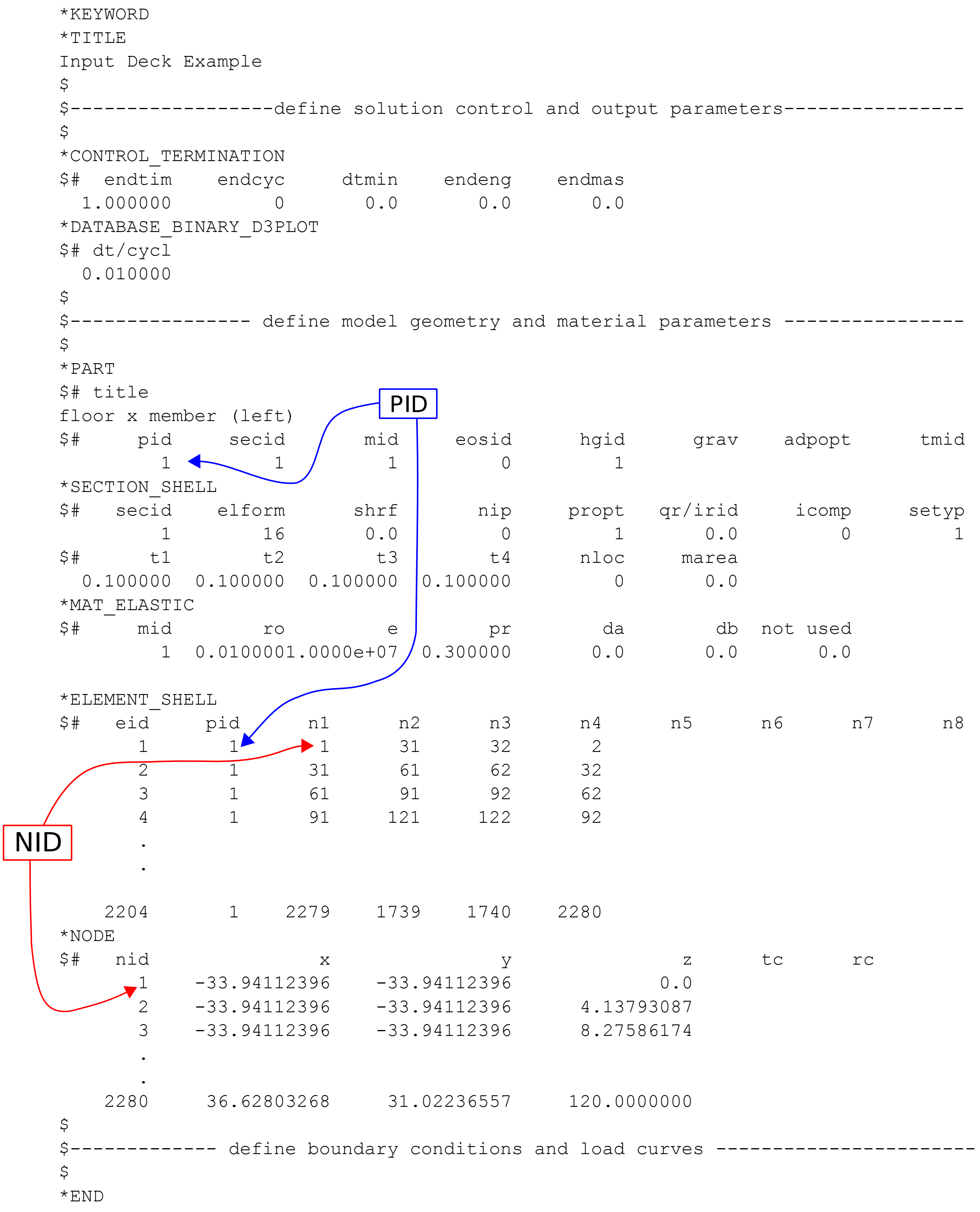
```
 Input: FEM include file of car

 Apply femparser → mesh(python object)

 procedure 1.EXRACT ALL PART ID'S
     for part in mesh do
         extract part id's
     end for
 end procedure

 procedure 2.EXTRACT GEOMETRY OF EACH PART
     for id in parts id's do
         Extract element id from mesh
         for element id do
             extract nodal coordinates from mesh
         end for
     end for
 end procedure

 Output: part json's
```



```
*KEYWORD
*TITLE
Input Deck Example
$
$------------------define solution control and output parameters----------------
$
*CONTROL_TERMINATION
$#  endtim    endcyc     dtmin    endeng    endmas
  1.000000         0       0.0       0.0       0.0
*DATABASE_BINARY_D3PLOT
$# dt/cycl
  0.010000
$
$---------------- define model geometry and material parameters ----------------
$
*PART
$# title
floor x member (left)
$#     pid     secid       mid     eosid      hgid      grav    adpopt      tmid
         1         1         1         0         1
*SECTION_SHELL
$#   secid    elform      shrf       nip     propt   qr/irid     icomp     setyp
         1        16       0.0         0         1       0.0         0         1
$#      t1        t2        t3        t4      nloc     marea
  0.100000  0.100000  0.100000  0.100000         0       0.0
*MAT_ELASTIC
$#     mid        ro         e        pr        da        db  not used
         1  0.0100001.0000e+07  0.300000       0.0       0.0       0.0

*ELEMENT_SHELL
$#   eid     pid      n1      n2      n3      n4      n5      n6      n7      n8
       1       1       1      31      32       2
       2       1      31      61      62      32
       3       1      61      91      92      62
       4       1      91     121     122      92
       .
       .

    2204       1    2279    1739    1740    2280
*NODE
$#   nid               x               y               z      tc      rc
       1    -33.94112396    -33.94112396             0.0
       2    -33.94112396    -33.94112396      4.13793087
       3    -33.94112396    -33.94112396      8.27586174
       .
       .
    2280     36.62803268     31.02236557     120.0000000
$
$------------- define boundary conditions and load curves -----------------------
$
*END
```

Figure 24: Input deck for LS-Dyna

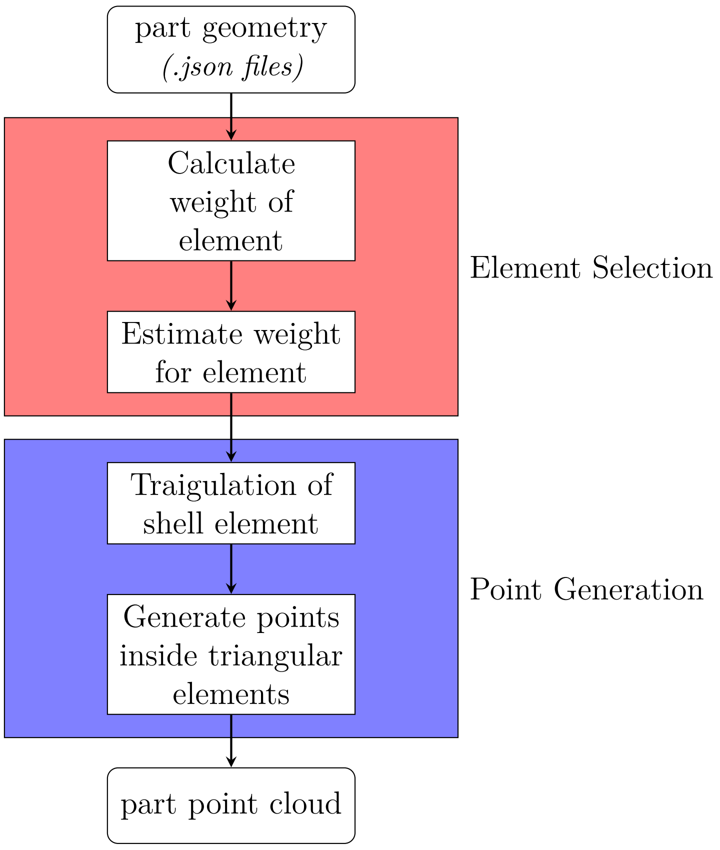

part geometry
(.json files)
Calculate weight of element
Estimate weight for element
Element Selection
Traigulation of shell element
Generate points inside triangular elements
Point Generation
part point cloud

Figure 25: Process of point cloud generation

3.4.2 Generation of point clouds

In the previous subsection, we extracted the geometric data for individual parts and saved in JSON format. In this subsection we will discuss the approaches and algorithms used to generate point clouds for parts which will be used to train the PointNet. We have the coordinates of shell and triangular elements which makes up the parts. To generate the point cloud for the complete geometry of the part we would have to generate points on the surface of elements. It is necessary to have a roughly equal distribution of points on the surface of the part so that concentration of points in certain regions do not affect the quality of results which we would obtain in further sections. We also have to take into consideration the fact that the number of elements which describe a part are not all equal. Also the size of parts vary according to their function. So we need to develop methods to generate point clouds which has points uniformly distributed across the part surface and capture the complete geometric information. In this work, two step approach is used. First we select elements so that we can capture the complete geometric information and then we triangulate the elements to generate points on surface using the concept of barycentric coordinates

Element selection

First we have to decide the number of points n_{points} to be generated in the point cloud. Once we know how many points to generate, we can employ different approaches to select elements, on the surface of which points will be generated. The simplest approach would be to generate one point, in random position, per element. Even though it sounds logical, it fails to generate equal number of points for different parts because they are not defined by the same number of elements. The next approach is called the Naive random selection technique, in which we can select randomly n_{points} elements and generate one point, in a random position, inside the corresponding element. This approach looks fairly simple and would provide us a different point clouds for same part. The problem with this approach is that we cannot guarantee a fairly equal distribution of points over the complete part since elements are chosen randomly. Also when the number of points n is greater than the number of elements available for point generation, some elements can be chosen multiple times and this would lead to concentration of points at certain areas which are not under our control. In certain cases this would also lead to loss of geometric information.

In order to improve the sampling of elements, we could use the area of element as a parameter. Using area of element as a parameter we can estimate the significance of an element. When the number of elements is greater than n_{points}, we would want to generate more points in elements with larger area. With this we can also mitigate the problem of loss of geometric information when the number of points n_{points} are less than number of elements in a part. In this work, algorithms were developed considering the above mentioned aspects. The area of elements were used as a parameter to select the significant elements and weights were generated to estimate the number of points to be generated within the element.

In this work the area of triangular elements is calculated using Eq (3.3) .

$$\text{area}\triangle ABC = \frac{1}{2} \mid \vec{AB} \times \vec{AC} \mid \tag{3.3}$$

where $\vec{AB} = B - A$,$\vec{AC} = C - A$ and A, B, C are the vertices of the triangle. In order to calculate the area of shell elements, we triangulate them and sum up the areas of the triangle. Once we calculate the area of all the elements in a part we generate a weighing factor w for each element which is the measure of number of points to be generated within the element. The weighing factor w_i for i^{th} element is calculated using the Eq (3.4)

$$w_i = \frac{A_i}{\sum_{i=1}^{n} A_i} \times n_{points} \tag{3.4}$$

where A_i is the area of the i^{th} element and n is the number of elements which defines the part. An important aspect to be noted during generation of w_i is that they should be whole numbers and hence the calculation needs to be refined accordingly. In order to do so the first idea would be to round w_i's to nearest integer. When we round the w_i to nearest integer two cases arise which needs to be addressed.

Case I : $\sum_{i=1}^{n} w_i < n_{points}$

We are required to generate n_{points} for each part and during the operation of rounding w_i, it might occur that $\sum_{i=1}^{n} w_i < n_{points}$. In such cases we would have a deficit of $n_{points} - \sum_{i=1}^{n} w_i$ points. We would need to account for this difference of points. Algorithm (2) presents the methods employed in this work.

Algorithm 2 Refinement of weighing factors for the case $\sum_{i=1}^{n} w_i < n_{points}$

```
Calculate deficit $d = n_{points} - \sum_{i=1}^{n} w_i$
Calculate $n_z$ = number of elements with $w_i = 0$
if $n_z > d$ then
    Randomly choose elements with $w_i = 0$ and make their $w_i = 1$
end if
if $n_z < d$ then
    Choose elements with $w_i = 0$ and make their $w_i = 1$
    Find difference $d_z = d - n_z$
    for $d_z$ do
        Choose random element and do $w_i = w_i + 1$
    end for
end if
```

Case II : $\sum_{i=1}^{n} w_i > n_{points}$

When $\sum_{i=1}^{n} w_i$ is greater than n_{points}, then we will be generating excess points which is not required. Algorithm presents methods used to reduce the these excess points in this work.

Algorithm 3 Refinement of weighing factors for the case $\sum_{i=1}^{n} w_i > n_{points}$

```
Calculate excess $e = \sum_{i=1}^{n} w_i - n_{points}$
Find $e$ number of elements with largest $w_i$
for selected $e$ elements do
    Perform $w_i = w_i - 1$
end for
```

Point generation

Now that we know the number of points to generate on surface of each element of a part, the next step is to discuss the method used for generating point inside the element. In Chapter 2, we discussed the concept of barycentric coordinates and Latin hyper cube sampling technique. These concepts will be used for generating points on the surface of elements of the parts. The concept of barycentric coordinates can be applied directly to triangular elements, but in our case we also have rectangular elements. The easiest way to generate point for rectangular elements would be triangulate them and then use the concept of barycentric coordinates to generate points on their surface. This method is employed in this work for generating points on surface for rectangular elements.

With the aid of a simple example, the process of generation of points on surface of elements will be explained. The example considers a simple rectangular element as with this example the process for triangular elements becomes self explanatory. Figure (26) shows the example for rectangular element $ABCD$.

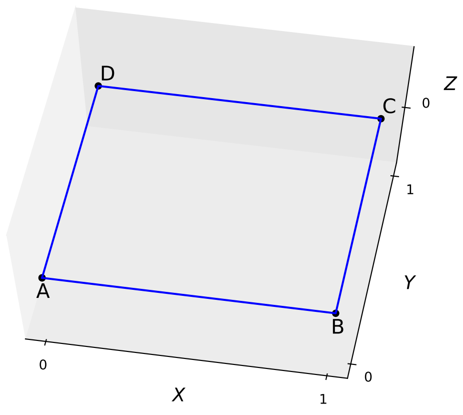
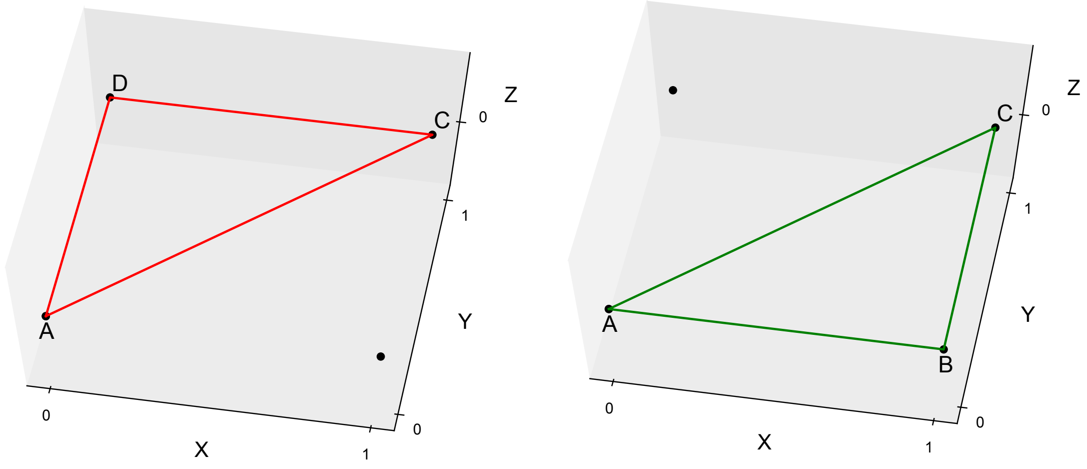


Figure 26: Element $ABCD$

Step 1

Divide the rectangular element into two triangular elements as shown in Figure (27)



(a) $\triangle ACD$ (b) $\triangle ABC$

Figure 27: Triangulation of rectangular element

In general case we can triangulate elements along any one of it's diagonal. Ideally we should expect similar results irrespective of the diagonal used for triangulation, but in case with non planar elements this is not the case. In this work, we have not explored methods to overcome this inconsistency.

Step 2

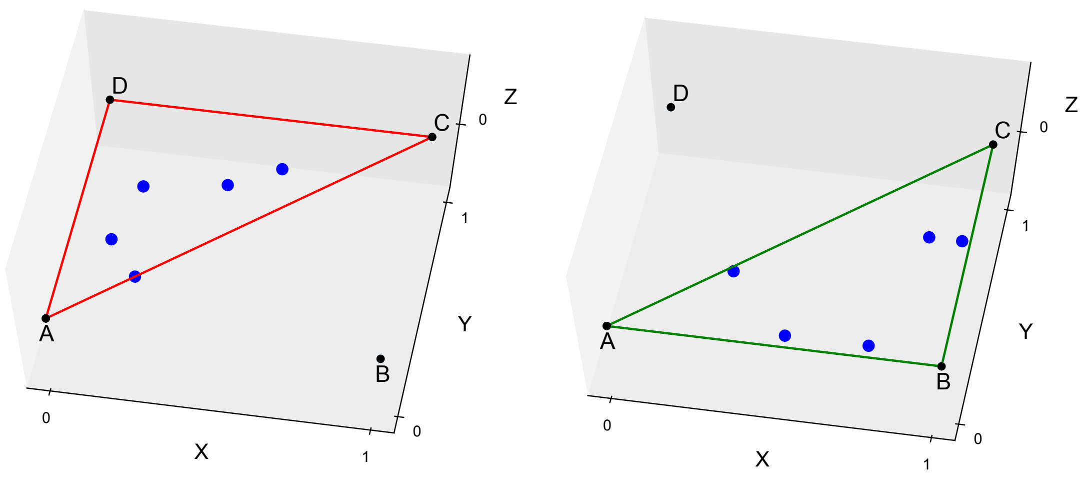
For this example let us assume we need to generate $n_{points} = 10$ for the element. As this is a simple example the triangulation of the elements leads to triangle's of equal area and hence the number of points to be generated for each triangle is $n_{points}/2$. We already have the geometric coordinate data of the vertices of the triangle and hence we can easily generate points on surface of triangle using the concept of barycentric coordinates. In order to do so we would require combinations of u, v, w's such that they sample the space uniformly and there is no clustering of points. This is achieved with the help of Latin Hyper-cube Sampling (LHS) technique. In order to generate point inside a triangle we generate sample weights u, v, w such that

$$u + v + w \leq 1 \tag{3.5}$$

A point p on the surface of the $\triangle ABC$ can be calculated using the coordinates of the vertices A, B and C using the Eq (3.6)

$$\begin{aligned} p_x &= \quad u \cdot A_x + v \cdot B_x + w \cdot C_x \\ p_y &= \quad u \cdot A_y + v \cdot B_y + w \cdot C_y \\ p_z &= \quad u \cdot A_z + v \cdot B_z + w \cdot C_z \end{aligned} \tag{3.6}$$

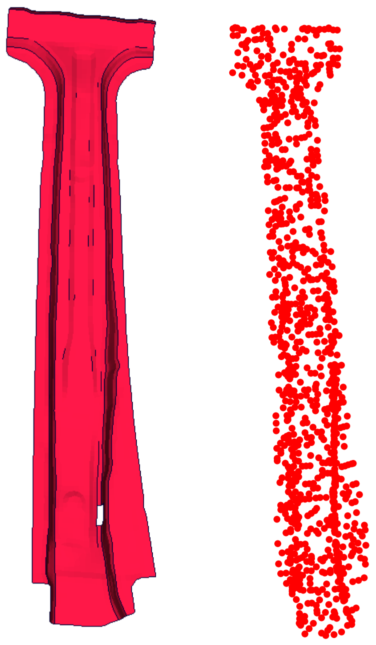
Using Eq (3.5) and (3.6) we generate points on surface of $\triangle ABC$ and $\triangle ACD$ (Figure (27)) which we can see in Figure (28). The combination of points generated on surface of triangle's gives points on surface of shell elements.

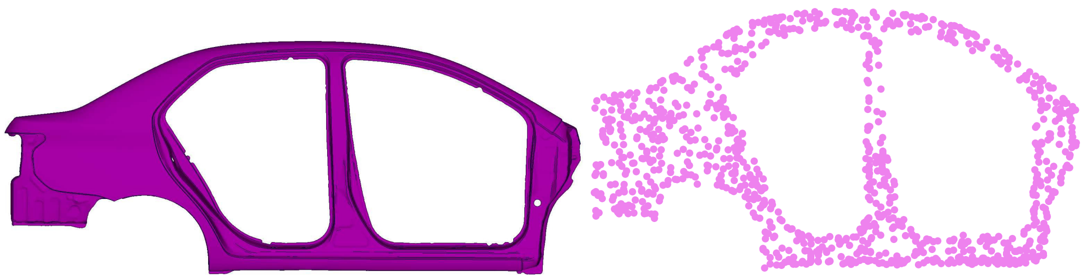


(a) $\triangle ACD$ (b) $\triangle ABC$

Figure 28: Points generated on surface of triangular elements

We extend this concept to all elements of the part and generate n_{points} number of points on the surface of the part. Figure shows the examples of point clouds of some of the parts of 2010 Toyota Yaris model which is detailed Finite Element Model available in public domain [51].


(a) *b-pillar*


(b) *side panel*

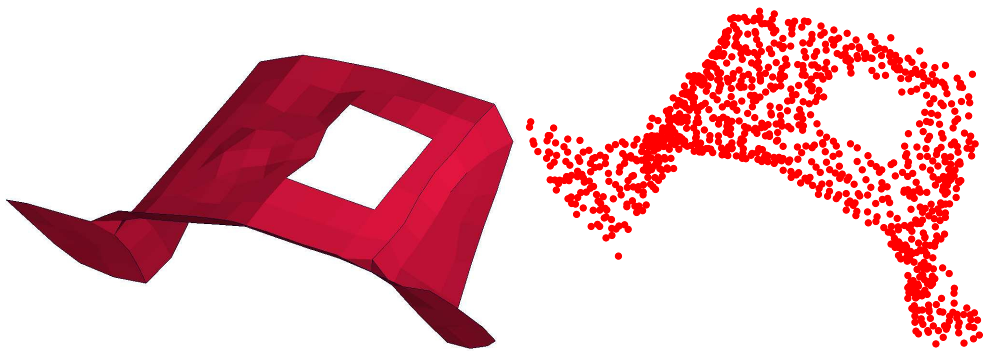

(c) *floor x member*

Figure 29: Point clouds of parts

3.5 Geometry classification experiments

This section presents the investigations conducted to identify different parts of Toyota Yaris Model using point clouds and PointNet. The results presented in [46] were on objects across 16 object categories divided into 40 classes. We want to investigate if we can extend the PointNet model to consume point cloud of part and provide an accurate identification of the part. The basic idea is to give each part an unique class label and check if the architecture is able to provide an accurate prediction of the class label for all parts. The position of the part in the complete body is also an important geometric information which can be used to identify the part, hence we do not normalize our input data for PointNet. Numerous experiments were conducted and the results obtained from them are outlined in this section.

3.5.1 Influence of number of points in point cloud

We tried to investigate the influence of number of points in point cloud to the classification capacity of PointNet. It was necessary to estimate the optimum minimum number of points per point cloud required to obtain accurate classification capability. It would also help to improve the performance in terms of training time and complexity of the PointNet. In order to do the same, 10 different parts were taken from Toyota Yaris Model and PointNet was trained for different number of points per point cloud. Training data i.e point clouds were generated for the 10 parts with $256, 400, 1024$ and 2048 points per part. The training data consisted for 400 samples for 10 parts. The PointNet was trained with the following input parameters.

- batch size = 10
- decay rate = 0.7
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$

The above mentioned input parameters were chosen in recommendation with the results from [46]. The training was done on Nvidia GeForce GTX 750 Ti. The results obtained from training the PointNet architecture for different number of points in point cloud is shown in Figure (30).

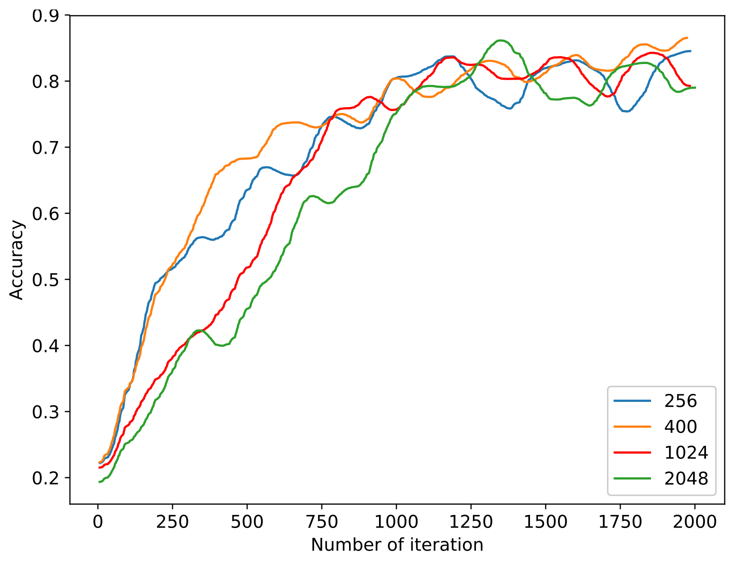


Figure 30: Training accuracy for 10 parts

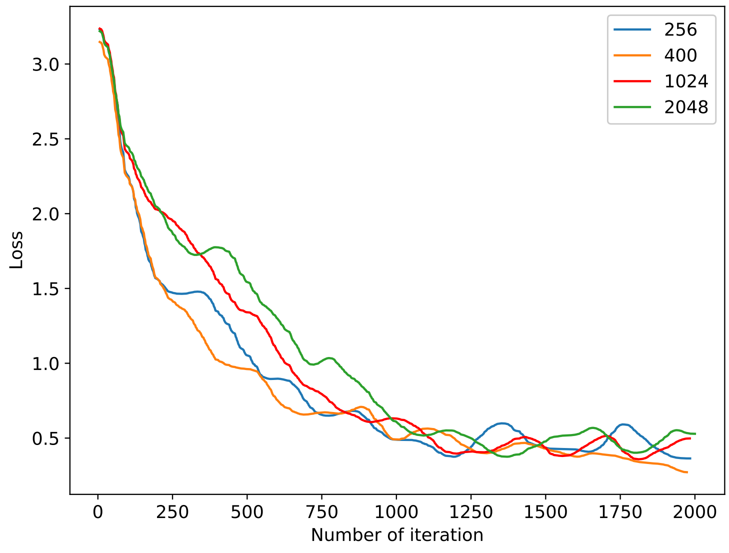


Figure 31: Training loss for 10 parts

From Figure (30) we can observe that the number of points in point cloud does not have a drastic effect on the results of classification for 10 parts. Also the training accuracy is nearly 90% which is expected in accordance with the results in [46]. It can inferred that the number of points in point cloud does not have significant effects in classification problem, but the number of points in point cloud has a significance when it comes to capturing the geometric information of the part. In order to mitigate the loss of geometric information it would be considered safe to take at least $n_{points} = 1024$ and proceed to check the classification capability of PointNet architecture for more that 40 classes.

3.5.2 Influence of number of training samples

In this work, the input data for PointNet architecture was generated from FEM models and unlike in [46] for each class label we only have one unique part. In order to have a positive outcome for the classification problem, we need to determine the influence of number of training samples used to train PointNet. We also need to have a look at the influence of sampling technique used to sample parts in the training data set. For this we chose 30 different parts from Toyota YARIS model and trained the model for 500 training samples and 1000 training samples with test set consisting of 100 samples. The parts were randomly sampled to generate the training set. We employ randomization of weights in LHS sampling due to which we have different point clouds for same part .The PointNet was trained with the following input parameters on Nvidia GeForce GTX 750 Ti for $n_{points} = 1024$.

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$

The performance plots are shown in Figure (32). It can be clearly noted that the performance of PointNet is better when we have more samples in training data. The evaluation performance was estimated using 100 newly generated samples. The model trained with 500 samples was able to correctly predict labels for 26 out of the 30 parts whereas the model trained with 1000 samples was able to correctly predict the labels for all 30 parts.

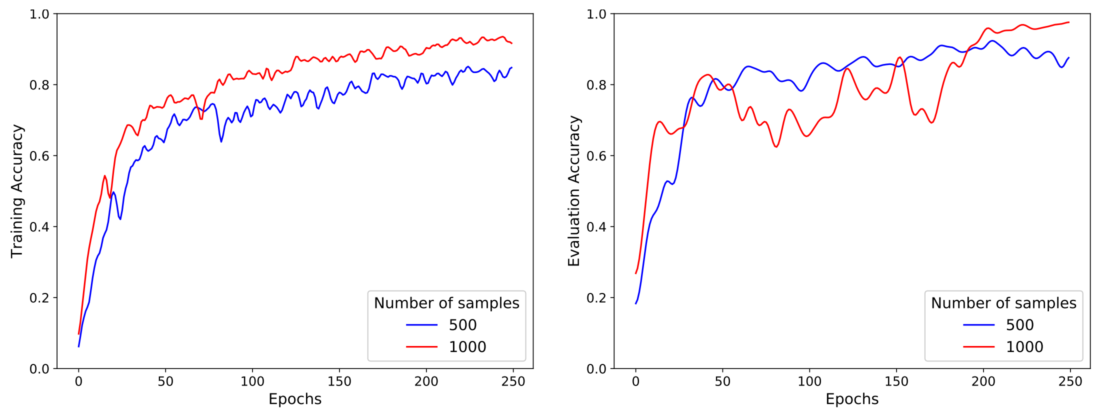


Figure 32: Performace of PointNet for 30 parts

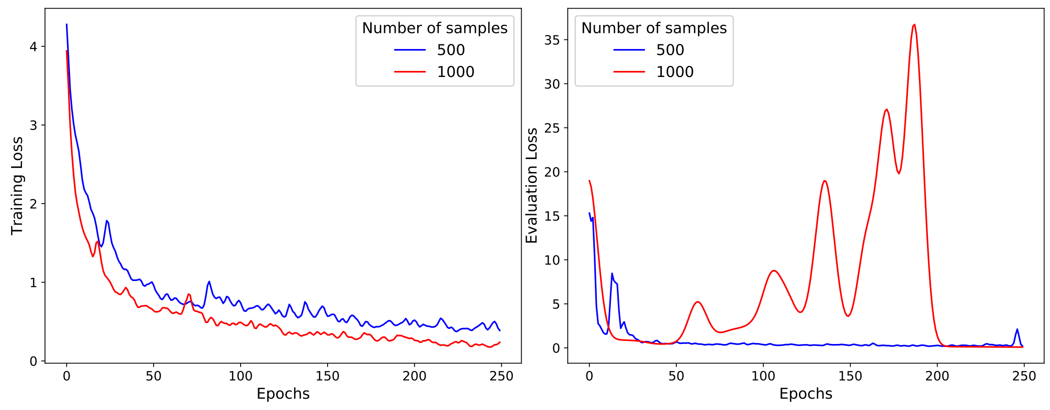


Figure 32: (Cont.) Performace of PointNet for 30 parts

3.5.3 Classification capability of PointNet

In this subsection we try to investigate the classification capability of the PointNet architecture. The classification capability for up to 40 unique classes was achieved in [46]. Our target would be to classify all the parts of a car or at least BIW of the car. In order to do so 100 unique different parts were chosen from the Toyota YARIS model. Point clouds were generated for these 100 parts and the PointNet architecture was trained with the following input parameters on Nvidia GeForce GTX 750 Ti for $n_{points} = 1024$.

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = 3500
- number of samples in test set = 500
- number of epochs = 250

The performance plots are shown in Figure (33). The training time for 100 parts was $\sim$ 8 hours. The performance of PointNet on 100 newly generated samples was quite impressive with an accuracy of 0.95. An in depth investigation was conducted on wrong classifications. It was found out that parts with similar shapes in close proximity were wrongly predicted.

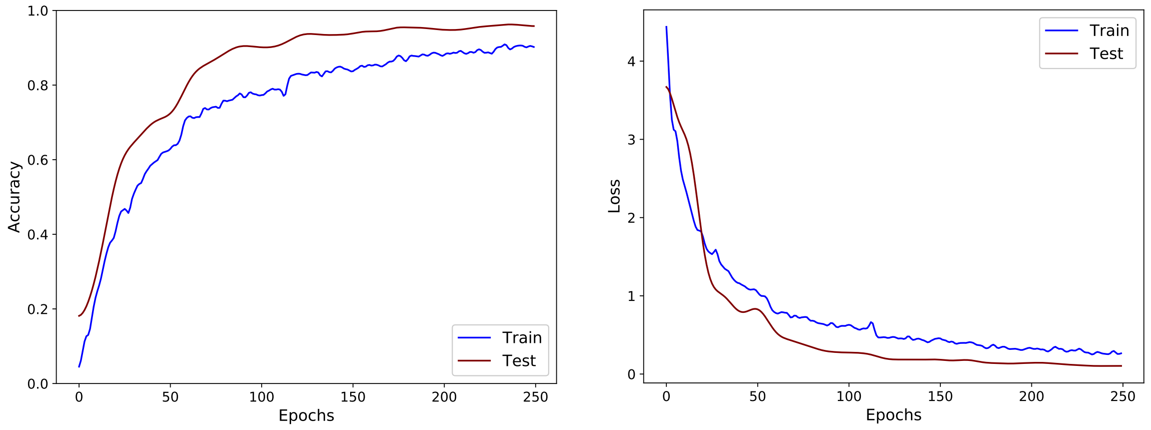
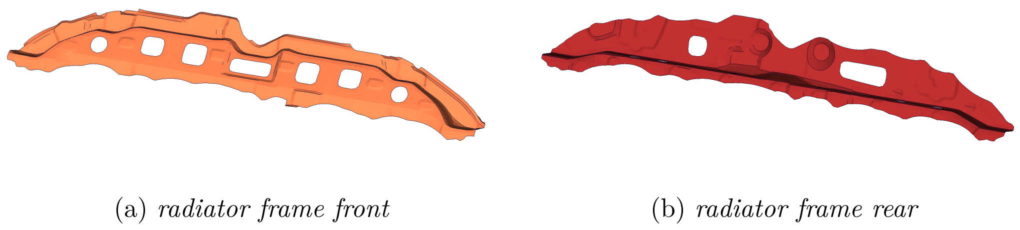


Figure 33: Performance of PointNet for 100 parts


(a) *radiator frame front* (b) *radiator frame rear*

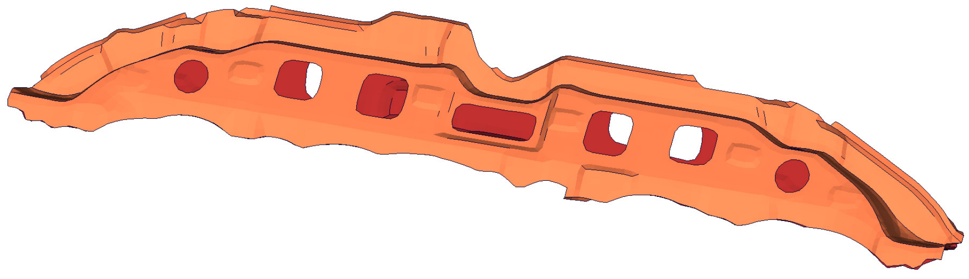


Figure 34: *radiator frame*

As an example, PointNet identifies *radiator front frame* (Figure (34a)) as *radiator rear frame* (Figure (34b)). The radiator frame front and radiator frame rear are closely positioned as seen in Figure (34) and also it would be difficult for the human eye to also distinguish them. Also the frequency with which the part appears in the training data set also affects the results. This problem can be solved by generating equal number of samples for all the parts instead of sampling them randomly.

3.5.4 Classification of parts for TOYOTA YARIS model

From the previous subsection it is clear that the PointNet architecture was able to classify 100 parts with 95% accuracy. In this subsection we will try to classify all the parts present in BIW of the TOYOTA YARIS model (Figure 35).

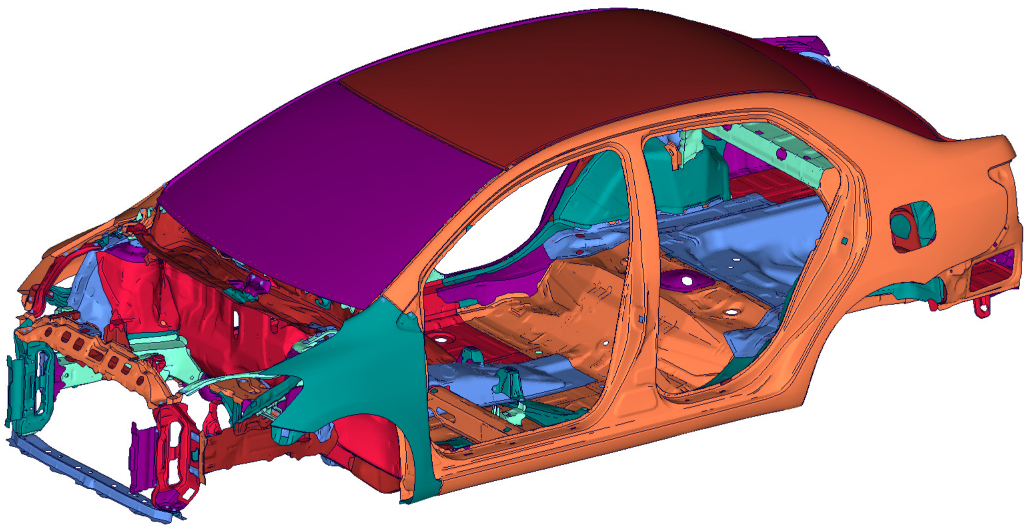
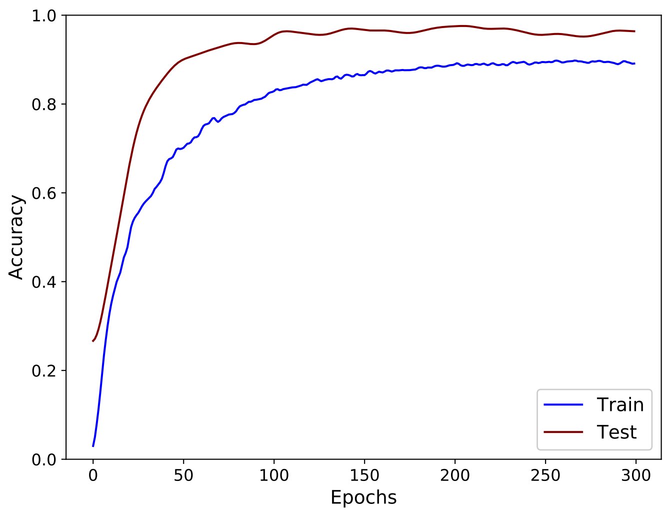


Figure 35: Toyoto YARIS model

250 different parts were identified and extracted from the FEM model. We have not combined mirrored parts i,e part with identical geometry on left and right side of the car. It is necessary for us to know if PointNet is able to identify them with respect to their positions. The PointNet was trained on Nvidia GeForce GTX 750 Ti for $n_{points} = 1024$ with the following input parameters for the architecture.

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = $10,000$
- number of samples in test set = 2500
- number of epochs = 300

Figure (36) shows the performance plots of PointNet when trained for 250 parts of the YARIS model. The training time for 300 epochs is $\sim$ 30 hours.



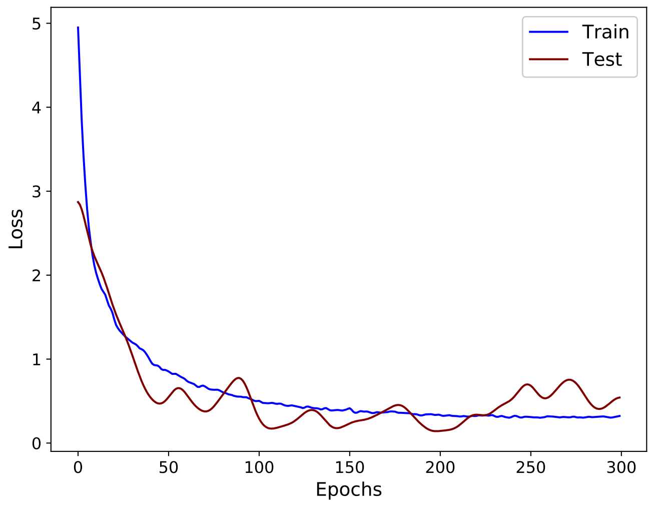


Figure 36: Performance of PointNet for 250 parts

PointNet was able to achieve 88% training accuracy. The trained model was then used to predict the class labels for unseen sample of point cloud for 250 parts. The model was able to predict 237 class labels correctly for 250 parts. With this results it becomes evident that 95% of part labels are predicted accurately by PointNet for the YARIS model. It is also interesting to note that the model can accurately differentiate similar geometries located on either side of axis of symmetry of the car. The only downside here is we had only one version for each part. It would be interesting to see the performance of PointNet when we have different versions of same part. In order to do so we would require information of different versions of same part.

Now that we are able to identify parts using PointNet we move on to extending the idea of using PointNet to predict spot weld parameters. Before moving on to that we also conducted a small experiment to check if we have performance improvements when we increase the number of points in point cloud drastically.

The PointNet was trained on Nvidia GeForce GTX 750 Ti for $n_{points} = 4096$ with the following input parameters for the architecture.

- Number of parts $= 100$
- batch size $= 10$
- learning rate $= 0.001$
- momentum $= 0.9$
- optimizer $= ADAM$
- number of samples in training set $= 8,000$
- number of samples in test set $= 200$
- number of epochs $= 300$

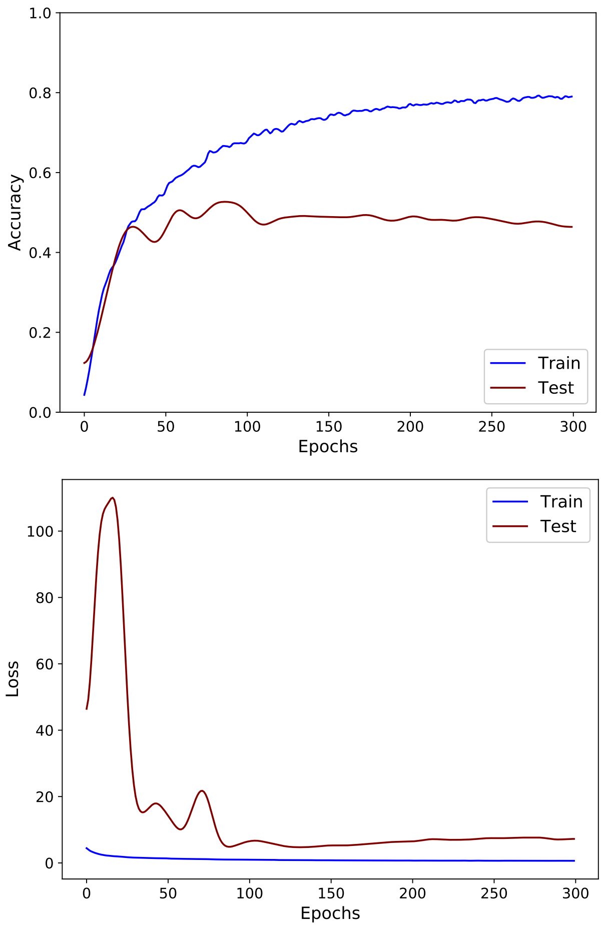


Figure 37: Performance for $n_{points} = 4096$

It is quite evident from the performance plot that increasing the points in point cloud to capture more detailed information does not improve results but provides far inferior results as the test accuracy is only 50% and also the training time was ~ 73 hours.

We can also attribute this to the fact that PointNet extracts only 1024 feature dimensions for classification after the feature transformation network and hence increasing the points in point cloud drastically won't help our case. In future works, we can experiment with the architecture of PointNet to extract more feature dimensions and check for improvement in classification capability.

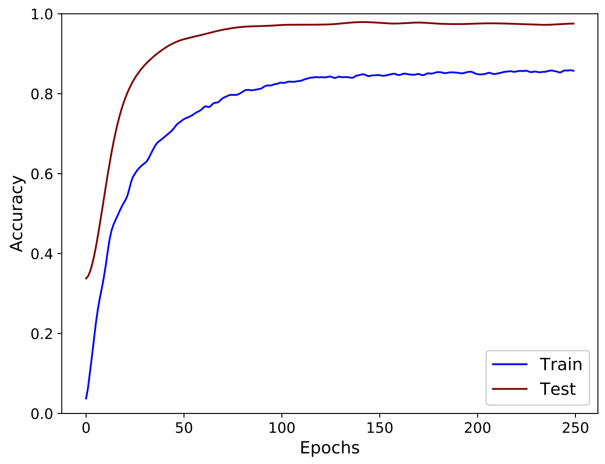
3.5.5 Classification of parts for AUDI model

The Simulation Data Management System LoCo provided an example of AUDI model where we had the opportunity to test the performance of PointNet when different versions of same parts exists. LoCo had information of a FEM model of an AUDI car for a whole development cycle of roughly 5 years. From this data we used 2 include files to check if some parts have different versions and we found 13 parts with different geometries. The model consisted of 350 parts and PointNet was trained for $n_{points} = 1024$ with the following input parameters for the architecture.

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = $17,500$
- number of samples in test set = 5000
- number of epochs = 250

Figure (38) shows the performance plots of PointNet for the AUDI model. PointNet was able to achieve 80% training accuracy and the training time was ~ 48 hours. On evaluation of the trained model with new set of point clouds, the model was able to identify 339 out of 350 parts correctly. This means that the trained model predicted 96% of the part labels accurately. This results are quite promising.

The trained model was then used to predict the part label for new versions of parts used for training the model. Figure (39) show an example of part version where the part was extended. The model was trained to identify the geometry shown in Figure (39a) The trained model was used predict the part label of a version of the same part where the part was extended as shown in Figure (39b). In reality the model has not seen this new version in it's training data but the trained model is able to predict the correct part label. With this example we could see that the trained network is able to predict correct class labels even when the part geometry changes but still maintains positional similarity and some geometrical similarity to the part used for training the model.



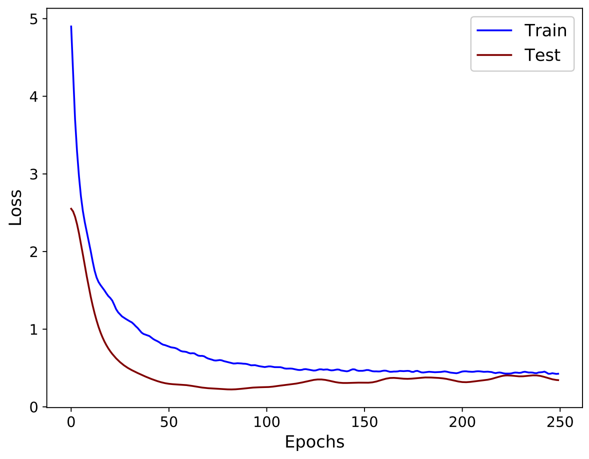


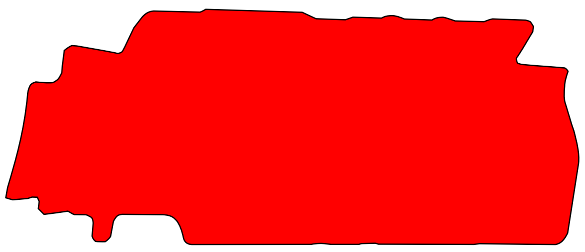
Figure 38: Performance of PointNet on AUDI model

Figure (40) shows an example of part version where the part has some internal changes. The model was trained to identify the geometry shown in Figure (40a) The trained model was used predict the part label of a version of the same part where the part has a small internal difference as shown in Figure (40b). In reality the model has not seen this new version in it's training data but the trained model is able to predict the correct part label. One could argue that the sampling methods used to generate the point clouds would not have captured such a small change and thus for the trained model it doesn't make any difference. In reality this would not be the case as the PointNet uses overall geometry and the position of the part also as information for predicting class label. The trained model was also able to predict accurate class labels for remaining 11 parts with versions.

In this section we have seen that we are able to identify the parts of BIW using point clouds and the prediction accuracy is very good for a machine learning algorithm. It was able to classify different versions of same part with identical part labels which is excellent for us.

When we scale our PointNet for a general case, we would need to test the classification capability of PointNet by training with parts of one car and then checking the label prediction of the model for parts of a different car. This experiment can be conducted and verified by an expert in automotive industry in future works. The identification capability of parts by PointNet leads to applying this method for many use cases, for example to identify parts of car to assign material properties to the part, to name the parts initially etc.

With the idea that we can identify the parts of BIW of car, we will try to extend this idea for estimation of spot weld parameters for part combinations in BIW of a car.


(a) Model trained with geometry

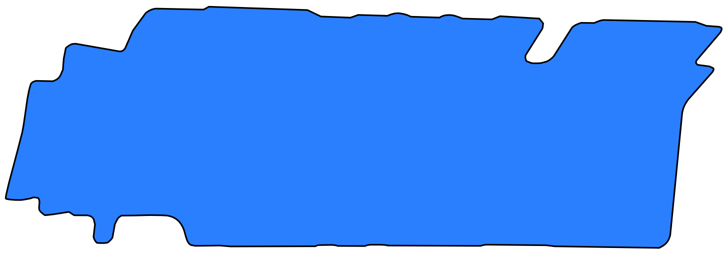
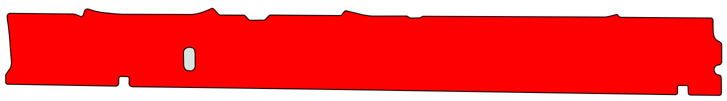
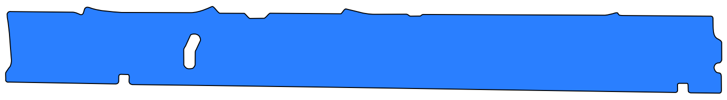

(b) New version

Figure 39: Part version example 1


(a) Model trained with geometry


(b) New version

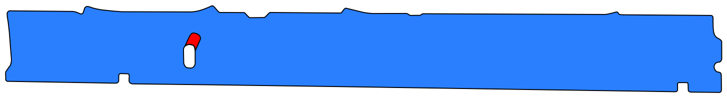

(c) Over lapped images to display internal difference

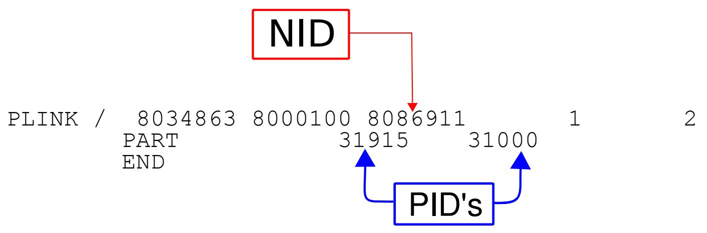
Figure 40: Part version example 2

4 Machine learning approaches for estimation of spot weld design

In this chapter we will discuss the approaches used to estimate the parameters of spot weld design for automotive construction of BIW of a car. In this work we tried to apply machine learning approaches to estimate elementary data for automatic generation of spot weld. This would provide us a fair idea if the possibility of automatic generation of spot weld design is an feasible idea to pursue. We will first have a look at extracting rudimentary data of spot weld design from FEM data which can then be used to estimate parameters of spot weld design. We then try to apply machine learning approaches to estimate these parameters and evaluate their performance for further studies.

4.1 Extraction of spot weld data from FEM data

In this section, we will discuss the ideas used to extract rudimentary data of spot welds. By rudimentary data, we mean the position of a spot weld (x, y, z), the combination of parts being connected and the number of spot welds connecting combinations of parts. In this work we extracted spot weld data from **PAM-CRASH** include files. Pam-Crash is a software package from ESI Group used for crash simulation and design of occupant safety systems in automotive industry. In Pam-Crash include files, PART DEFINITION cards are used to define various parameters of a part. For us the PLINK Element card is of importance. This card provides us information of spot welds in the model. Figure (41) show an example of PLINK card, which gives us the information of the coordinates of the weld point and the parts connected with this weld point.



```
PLINK /   8034863 8000100 8086911       1       2
         PART              31915    31000
         END
```

Figure 41: PLINK card

From the collection of data of spot welds, we can find information about parts which are connected to each other. Once we know the part combinations which are connected to each other, we can easily extract the spot weld data just for a particular combination. This approach was employed in this work. We first found out the combination of parts which are connected to each other using spot welds. Then for each part combination we extracted the position of spot welds i.e, (x, y, z) coordinates. In order to do the above mentioned ideas **femparser** was used. femparser provides PLINK as a python object from which we can easily access the connected part names and coordinates of the spot weld. For the example PLINK shown in Figure (41), using femparser we extract the coordinates of NID 8086911 and this spot weld connects PID's 31915 and 31000. This method was adopted to extracted the complete information about the spot welds present in the model. Now we also have information of part combinations which are connected using spot weld.

4.2 Identification of part combination's using PointNet

This section presents the investigations conducted to identify various part combinations of the AUDI model. Using the ideas presented in Section 4.1, we identified 873 part combinations in AUDI model connected using spot weld. We want to investigate if we can extend the PointNet model to consume point cloud of part combinations and provide an accurate identification of the part combination. The basic idea is to give each part combination a unique label and evaluate if the architecture is able to provide accurate prediction of class label for all the part combinations. In this case also the position of the part combination is an important geometric information for identification of the part combination. Numerous experiments were conducted and the results obtained are outlined in this section.

4.2.1 Classification capability for 10 part combinations

In this subsection, we try to investigate the classification capability of PointNet architecture for 10 part combinations. Due to the computational effort and time involved, a small problem of classifying 10 part combinations was conducted. It would help us to understand if the PointNet architecture is able to provide good results or do we need to make some modifications to architecture for classification of part combinations. First we try to use the same architecture used for identification of individual parts to classify part combinations. Training data i.e, point clouds were generated for part combinations with 1024 points per part and thus each part combination would be made up of $n_{points} = 2048$ points. The PointNet was trained with following input parameters on Nvidia GeForce GTX 750 Ti.

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = 400

- number of samples in test set = 100
- number of epochs = 250

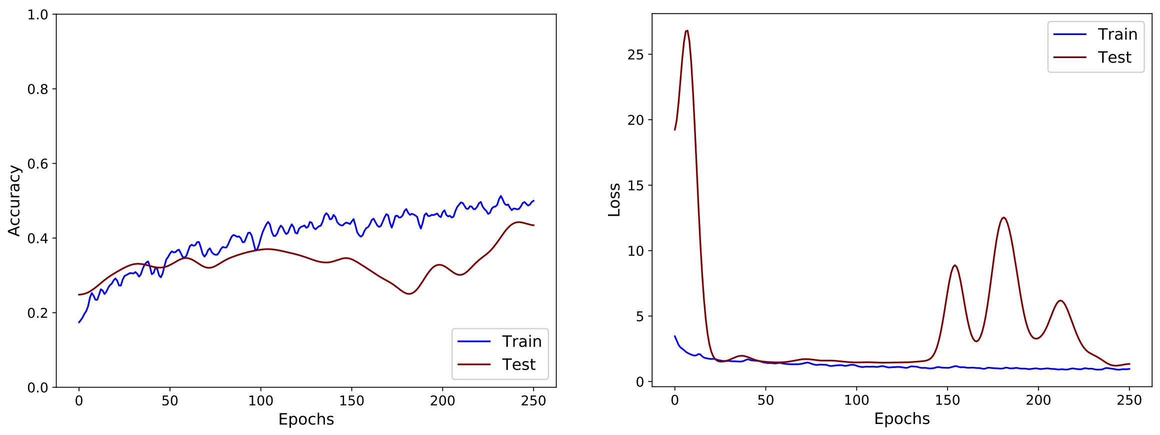


Figure 42: PointNet performance for 10 part combinations

The performance plots are shown in Figure (42). It can be clearly noted that the performance of PointNet for classifying part combinations is poor. The training accuracy is a mere 45% only. In order to understand the reason for this poor performance the architecture was analyzed. Using just individual parts as test samples, we found out that the input transform just rotates and scales the input training data to standard scale. This can be avoided and hence we decided to remove the input transform. The architecture after removing input transform is as shown in Figure (43). In this work, this modified architecture of PointNet will be referred to PointNet-Combi.

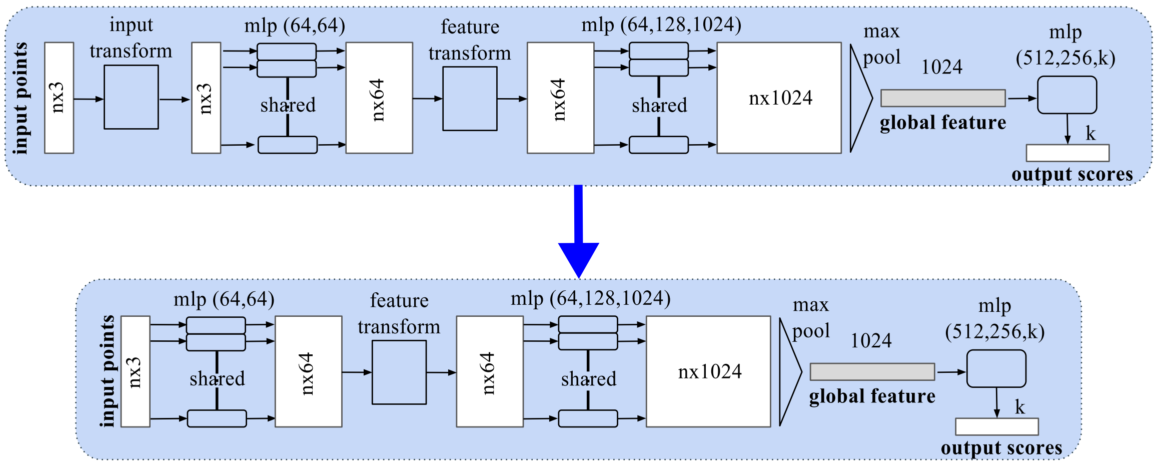


Figure 43: PointNet architecture modification for part combinations

PointNet-Combi was trained with the same training data with identical parameters as mentioned in the previous experiment. The performance plots are visualized in Figure (44). We can see that there is a considerable improvement in performance of PointNet-Combi as compared to the performance of PointNet for classifying 10 part combinations. PointNet-Combi was able to identify 7 out of 10 parts correctly. With this modified architecture of PointNet-Combi, we next try to classify the 873 part combinations of AUDI model.

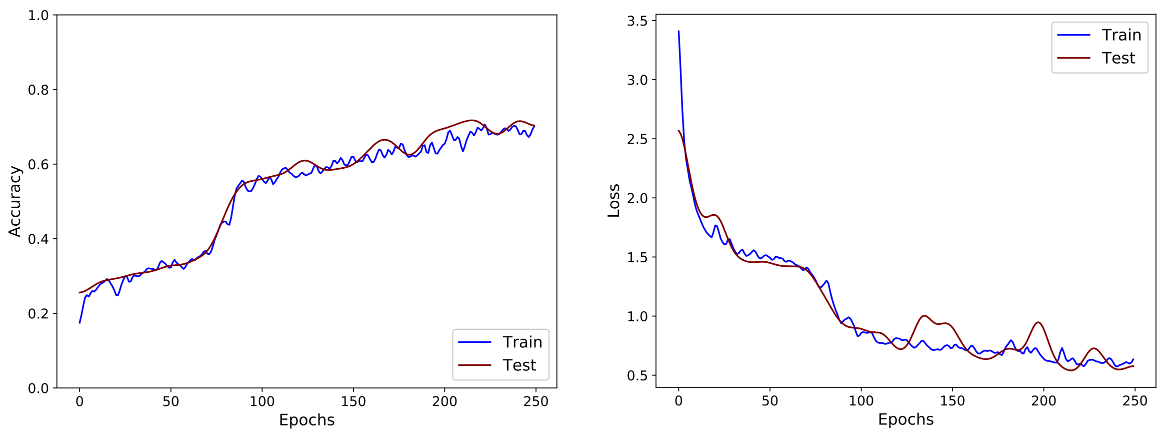
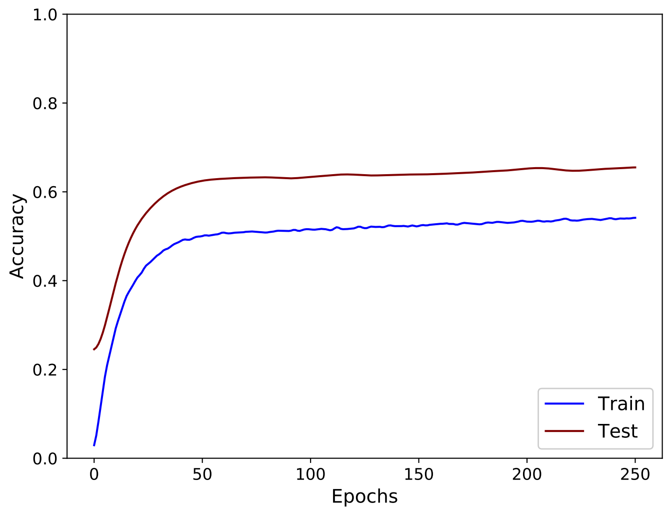


Figure 44: PointNet-Combi performance for 10 part combinations

4.2.2 Identification of part combinations using PointNet-Combi

In previous subsection, modification of PointNet architecture to PointNet-Combi architecture produced improved performance. In this subsection we use this modified architecture to classify 873 part combinations from the AUDI model. The PointNet-Combi architecture was trained for $n_{points} = 1024$ with the following input parameters on Nvidia GeForec GTX 750 Ti :

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = $43,650$
- number of samples in test set = 5000
- number of epochs = 300



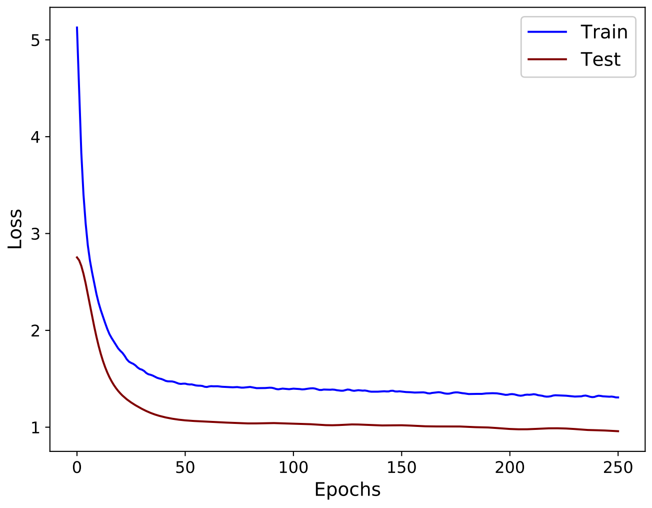


Figure 45: PointNet-Combi performance on 873 part combinations

The performance plots are show in Figure (45). The evaluation accuracy is just 60% i.e, the trained network is able to predict only 524 part combination labels correctly. This kind of performance is not useful for our further works. In an attempt to improve the performance of PointNet-Combi architecture, one more modification was done to the network. We removed the feature transform network. This architecture without input and feature transforms will be referred to as PointNet-Basic in this work. The modified architecture of PointNet-Basic is visualized in Figure (46).

PointNet-Basic was trained with the same training data with identical parameters as mentioned in the previous experiment. The performance plots are visualized in Figure (47). We can clearly see that there is no significant improvement in performance with PointNet-Basic architecture.

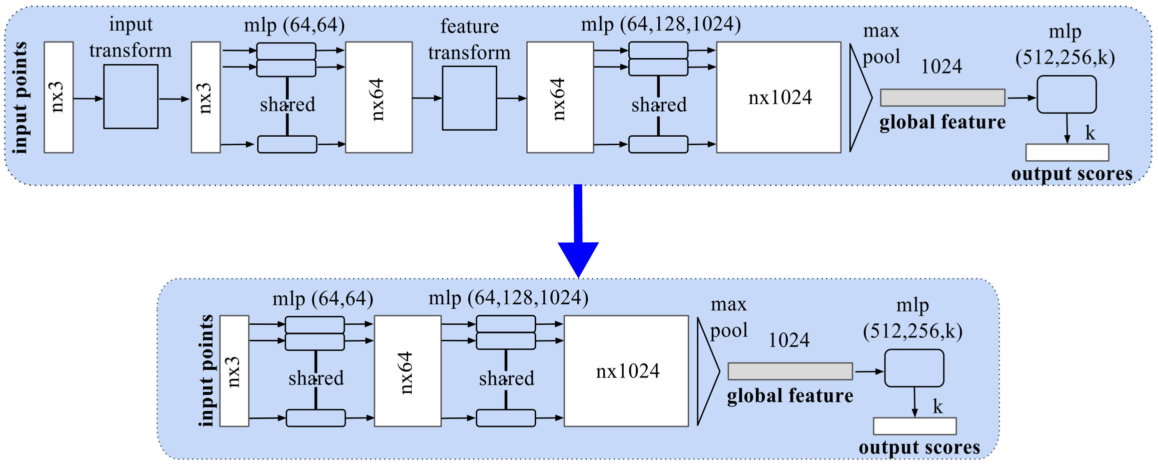


Figure 46: PointNet-Basic architecture

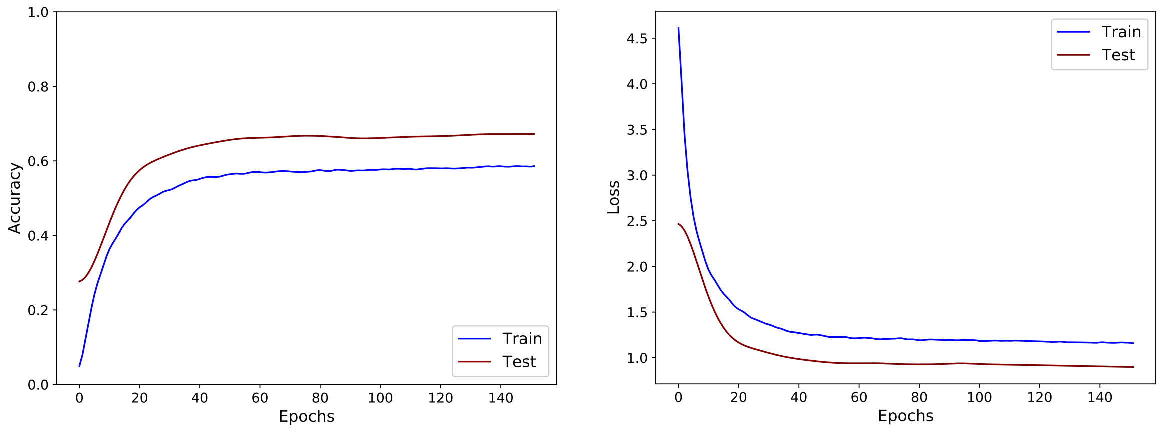


Figure 47: PointNet-Basic performance on 873 parts

Another attempt was conducted to improve the performance of PointNet-Basic architecture. Since we already know that there are two parts in a combination, we tried to label the points and then train the PointNet-Basic architecture. In doing so the input data dimension changes from nx3 to nx4, as shown in Figure (48), where the fourth dimension is the label. We labeled points belonging to one part with 0 and the label for points belonging to second was set to 1. This new architecture was trained to identify the part combinations for $n_{points} = 1024$ with similar network parameters as in previous examples and

- number of samples in training set $= 26,190$
- number of samples in test set $= 873$
- number of epochs $= 300$

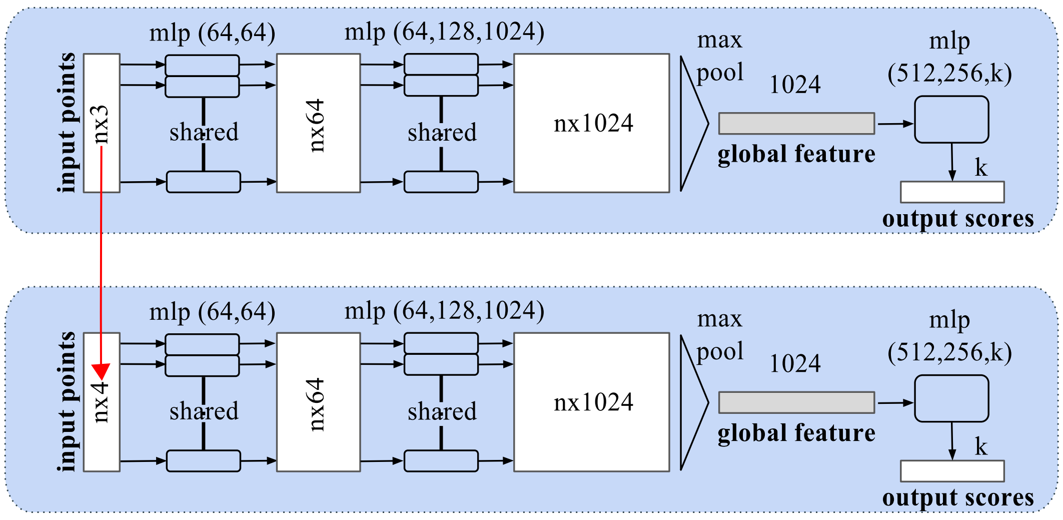


Figure 48: PointNet-Basic architecture with point labelling

The performance plots are shown in Figure (49). On evaluation with the trained model, it is only able to identify 596 part combinations correctly out of 873.

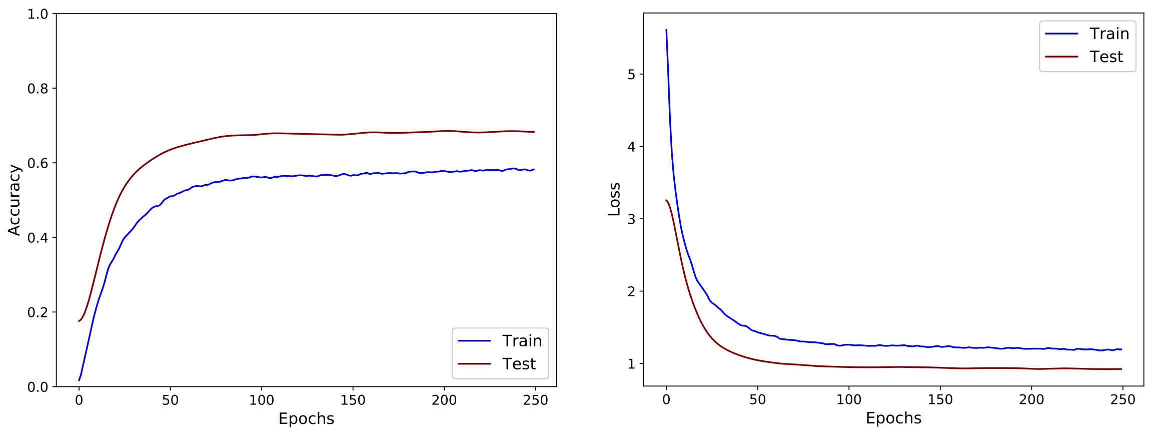


Figure 49: Performance plot for PointNet-Basic with point labeling

On inspection of predicted class label for part combinations, the trained model is unable to differentiate part combinations when one part in the combination is significantly larger in size than the other part. This leads to masking of the smaller geometry by the bigger geometry which PointNet is not able to distinguish.

With the results from various experiments conducted to identify part combinations, we can conclude that PointNet architecture cannot be effectively used to identify part combinations.

4.3 Spot weld design

The spot weld design, determined by the design engineer depends on many factors, i.e, input parameters like loads and forces that might be applied to the structure, material combination and geometry of the parts, the connection technology and it's process parameters. Our focus is mainly on parameters like position of the spot weld with respect to the flange and the distance between the spot welds. Both these parameters usually vary for each part combination. In this section we will describe the methods use to calculate these parameters from FEM model

4.3.1 Minimum distance between spot welds

Figure (50) shows an example of *bpillar* of a car with spot weld design. The spot weld design obtained for this example is a final result of numerous design and engineering iterations. The design varies with respect to materials used and the sheet thickness of the part. As an engineer our target would be to use machine learning for prediction of this spot weld design without the effort and cost of numerous engineering simulations. As this is a "first of it's kind" work, initially we would be working towards estimation of minimum distance between spot welds so that we a have a good starting design which would need to go through minimum number of design iterations. In doing so we would be at least saving a significant amount of cost and computational efforts required to reach optimum spot weld design.

The spot welds are not always placed at equal distances to each other as shown in Figure (51). The distances between the spot welds vary depending on many factors. So in order to have a good initial estimate, the minimum distance between the spot welds would be a good starting point. This subsection describes the methods used to estimate the minimum distance between the spot welds "d_s".

In Section 4.1 we have extracted the coordinates of all spot welds present a particular part combination. We now have a list of (x, y, z) coordinates which is the position of spot welds for a particular part combination. Using this information we try to estimate the minimum distance between the spot welds. In order to do so, we compute distance to the nearest point for each point in the list and then compute the average of all the minimum distances to get d_s.

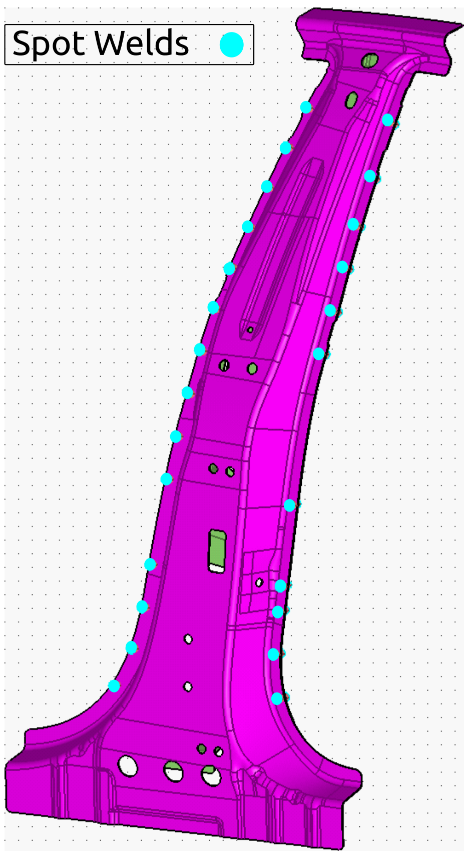


Figure 50: *bpillar* with spot welds

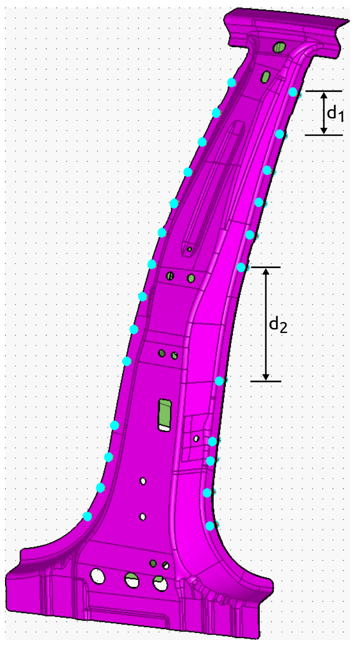


Figure 51: Distance between spot welds

Consider we have n spot weld points for a part combination. Algorithm (4) presents the method employed to estimate d_s for a part combination. There exists many methods to find the closest point for a chosen point from a given set of points. In this work we employed a linear search solution, in which for a given point we computed the euclidean distance to all other points and the closest point is the one which is at the shortest distance to the given point.

Algorithm 4 Calculate d_s for a part combination

```
    **Input:** $P_1, \dots, P_n$

    for $i = 1, \dots, n$ do
        Find closest point $P_i^c$ for $P_i$
        Calculate $d_i^c = \sqrt{\sum_{j=1}^{3}(P_{i,j}^c - P_{i,j})^2}$
    end for

        $d_s = \frac{\sum_{i=1}^{n} d_i^c}{n}$
```

Now that we have calculated d_s, we have a parameter associated for each part combination. We could apply machine learning approaches to estimate this parameter.

4.3.2 Contact surface of joining parts

In spot weld design, in order to connect two parts, it is necessary to determine the area where actual contact between the parts occurs. Figure (52) shows an example of a part with spot welds placed at surfaces of contact with the joined part.

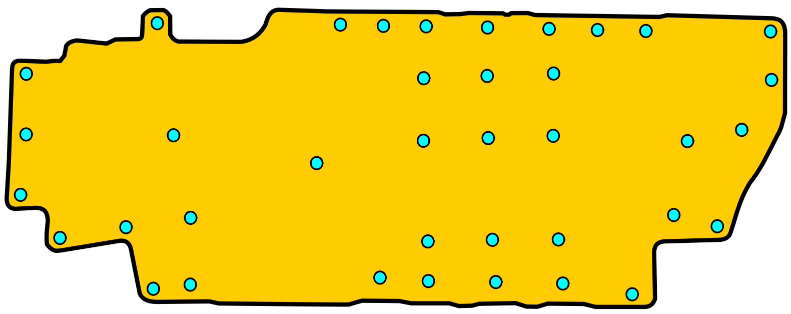


Figure 52: Part example with spot welds

We observe that for a part combination we can identify clusters which are isolated contact surfaces with respective design properties i,e. for each contact surface the distance between the spot welds can be different. In this case, we need to develop methods to identify this clusters. The clusters are shown in Figure (53)

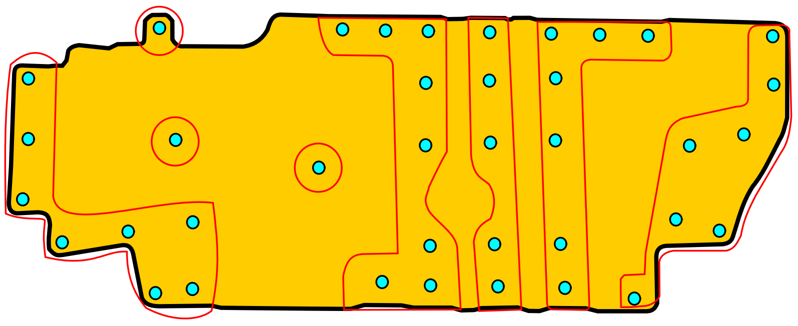


Figure 53: Example of clusters on a part

In our case, the problem of finding clusters, leads to first finding elements of one part that are in contact with another part and then using element connectivity to isolate clusters. This subsection will describe and out lie the methods use to solve the above mentioned two problems.

Finding contact elements

The basic idea used in this work to estimate contact elements was to find the nodes of parts that are in contact proximity with each other. With this information of nodes which are in contact proximity, we can easily find the elements that are in contact for the two parts. Let us consider two parts $P1$ and $P2$. For each part we have the coordinates of the element. The first step would be to identify unique nodes for both $P1$ and $P2$. We identify the nodes of different parts that are in close proximity using a parameter called '*search radius*'. In FEM models, the parts in contact have a small distance separating the parts as shown in Figure (54).

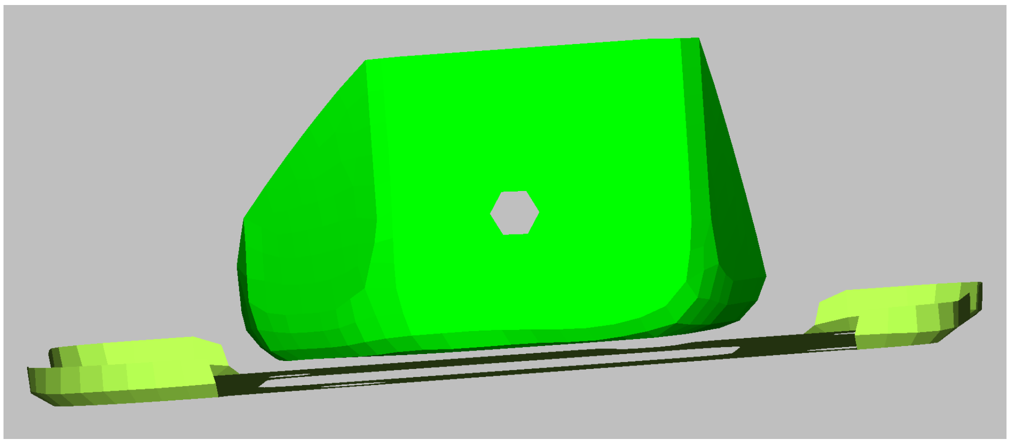


Figure 54: Example showing distance between connected parts

Using advanced CAE pre–processing software ANSA, we found out distance between closest nodes of two parts and this distance will be our *search radius*. The method used to estimate nodes in contact proximity is that, for nodes in $P1$ we find nodes in $P2$ such that the euclidean distance between them is less than or equal to search radius. Now with these nodes we perform a reverse look up to find the elements that are in contact. Thus we will have information of elements that are in contact for two parts.

One numerical aspect we encounter with the above mentioned method is that when the element size is larger than search radius. When this is the case, our methods would not consider the nodes in close proximity and thus we would have loss of information. In order to overcome this aspect, we first find elements of $P1$ which are in contact and then using this elements we again perform a contact search with element size as search radius to find contact elements on $P2$. On this contact elements of $P2$, either we can do a refinement with default search radius or we can find the closest nodes of $P1$ and thereby estimate the contact elements in $P1$. Figure (55) shows the nodes of contact elements for part in Figure (52)

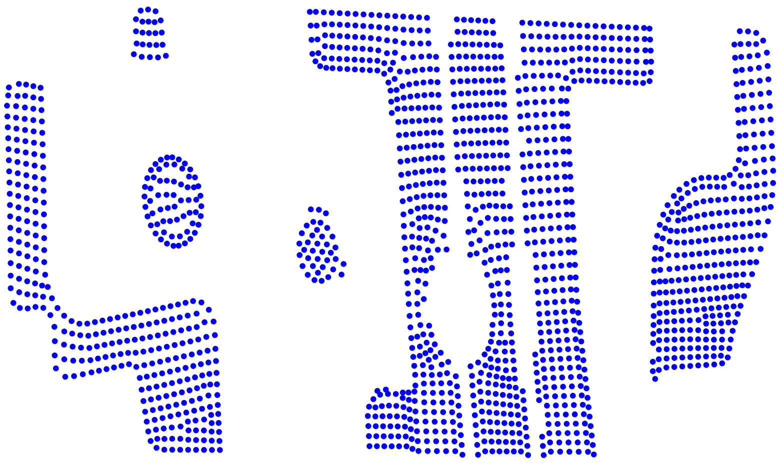


Figure 55: Contact elements displayed with help on nodes for part in Figure (52)

Estimating clusters in contact elements

Once we have the contact elements, our next step is to find clusters within this contact elements. In order to achieve this, we utilize the idea of element connectivity. From Figure (55) we see that, one can find clusters by grouping elements that are connected together. Algorithm (5) explains the methods employed in this work to estimate the clusters given that we have contact elements.

Algorithm 5 Algorithm to find cluster from contact elements

```
1. Choose a random element from contact elements and add to Cluster list
2. Find elements in contact with elements present in Cluster list
3. Add these new elements to Cluster list.
4. Repeat Step 2 and Step 3 until no new elements are found to add to Cluster list
5. Elements in Cluster list represents a cluster.
6. Remove the elements found in Step 5 from contact elements
7. Repeat Step 1 to Step 6 until there are no elements left in contact elements
```

The method used to find clusters in this work is explained with an aid of an simple example for the ease of understanding. Consider a simple example of contact elements as shown in Figure (56).

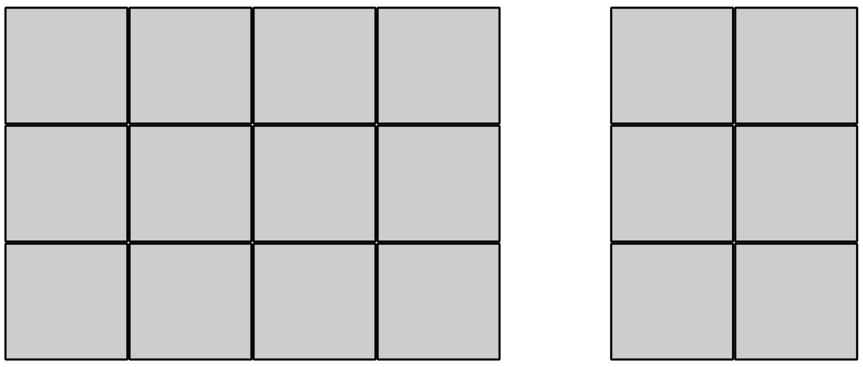
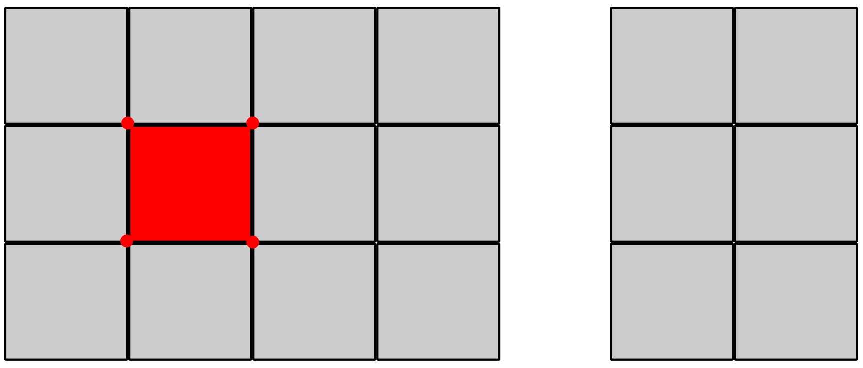
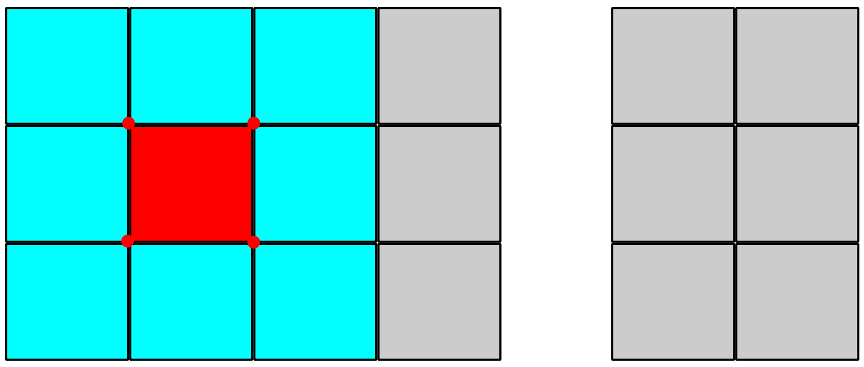


Figure 56: Example of contact elements

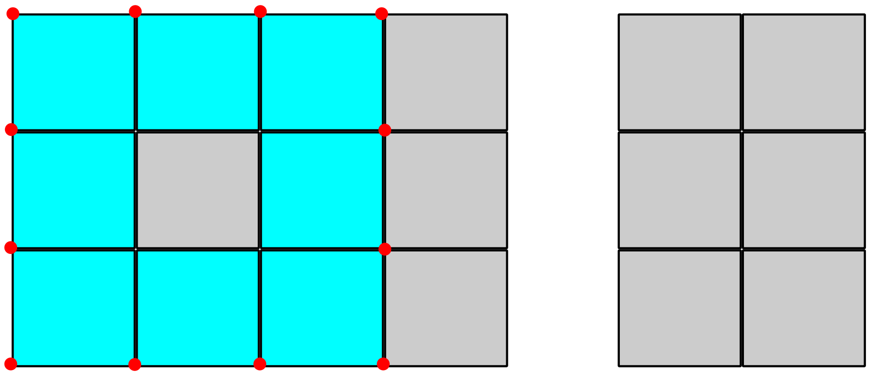
Step 1 Choose a random element and find the outer nodes for this element as shown



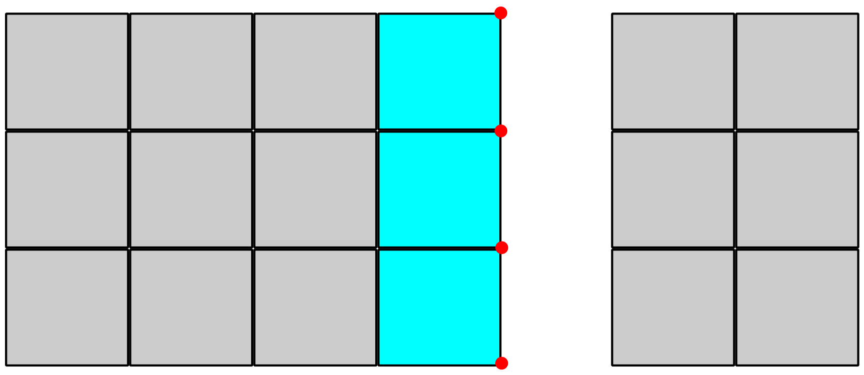
Step 2 Find elements in contact using the outer nodes



Step 3 Find new outer nodes for elements found in Step 2



Step 4 Repeat Step 2 and Step 3 until the outer nodes in Step 2 finds no new elements. This indicates we have found a cluster.



Step 5 We remove the elements that belong to the cluster found in Step 4 from contact elements and repeat Step 1 to Step 4 for remaining elements.

When we apply Algorithm (5) to find clusters for contact elements found in Figure (55) we obtain the the results presented in Figure (57), in which each cluster is visualized using a unique color.

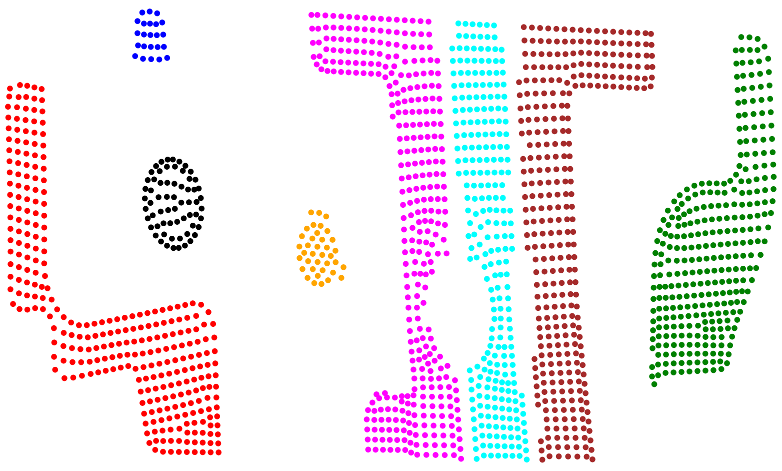


Figure 57: Different clusters for contact elements in (55)

4.4 Estimation of minimum distance between spot welds

In this section we will discuss the machine learning approaches employed in this work for estimating minimum distance between spot welds d_s. In Section 4.3 we discussed methods to calculate minimum spot weld distance for part combinations. Two approaches were explored to predict this parameter.

4.4.1 Classification based approach

Classification based approach considers the use of PointNet to estimate minimum distance between spot welds. In Section 4.2 we discussed the idea of using PointNet to identify part combinations and results obtained were poor. So instead of using PointNet to identify parts, we would try to generate class labels for d_s and train the PointNet-Combi to identify these class labels for part combinations. Experiments were conducted based on this idea and the the results obtained are outlined in this subsection.

Generation of class labels for d_s

In previous geometry identification experiments, we labeled each part combination with a unique label. However in this experiment we label the part combinations according to a criteria, which their corresponding d_s values satisfies. The basic idea is to use classifier to predict values for d_s based on part combinations. This approach was an attempt to use the geometry as an input to predict the parameter d_s. We conducted two experiments for this approach.

Experiment 1

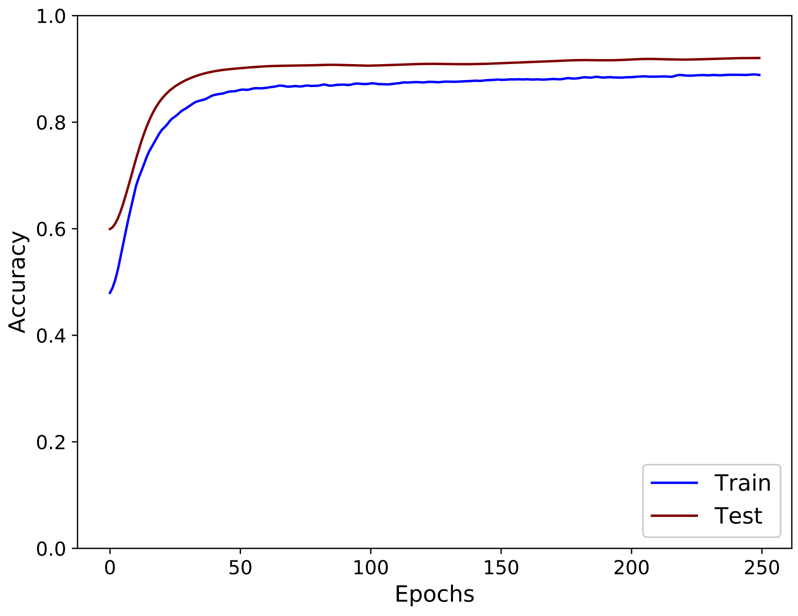
In this experiment, d_s values were computed for all part combinations. For certain part combinations, the spot weld design is quite simple. They just have a single connection at the point of contact between the connecting parts. We labeled this part combinations with a class label of numerical value zero. Also in order to simply the first experiment, part combinations with with $d_s > 100$ mm were labeled with same class label. For rest of the part combinations, labels were assigned values between 1 and 10 according to the criteria shown in Table (1) and thus we have 12 class labels.

Table 1: Labeling criteria for Experiment 1

criteria (mm)	label
$d_s = 0$	0
$0 < d_s < 10$	1
$10 < d_s < 20$	2
$20 < d_s < 30$	3
$30 < d_s < 40$	4
$40 < d_s < 50$	5
$50 < d_s < 60$	6
$60 < d_s < 70$	7
$70 < d_s < 80$	8
$80 < d_s < 90$	9
$90 < d_s < 100$	10
$d_s > 100$	11

The PointNet-Combi was trained with the newly labeled training data of part combinations on Nvidia GeForce GTX 750 Ti for $n_{points} = 1024$ with the following parameters

- batch size = 10
- learning rate = 0.001
- momentum = 0.9
- optimizer = $ADAM$
- number of samples in training set = $52,380$
- number of samples in test set = 873
- number of epochs = 250



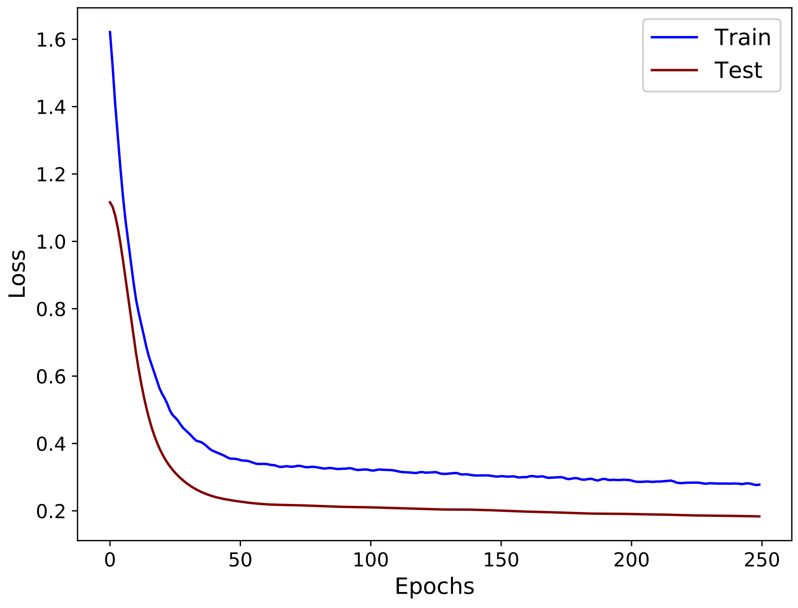


Figure 58: Performance plot for experiment 1

Figure (58) shows the performance plots for experiment 1. From the performance plots we can see that PointNet is able to predict 90% class labels accurately. On evaluation with new set of point clouds for part combinations, the trained model was able to predict class labels for 803 part combinations accurately out of 873 part combinations. This results are quite promising.

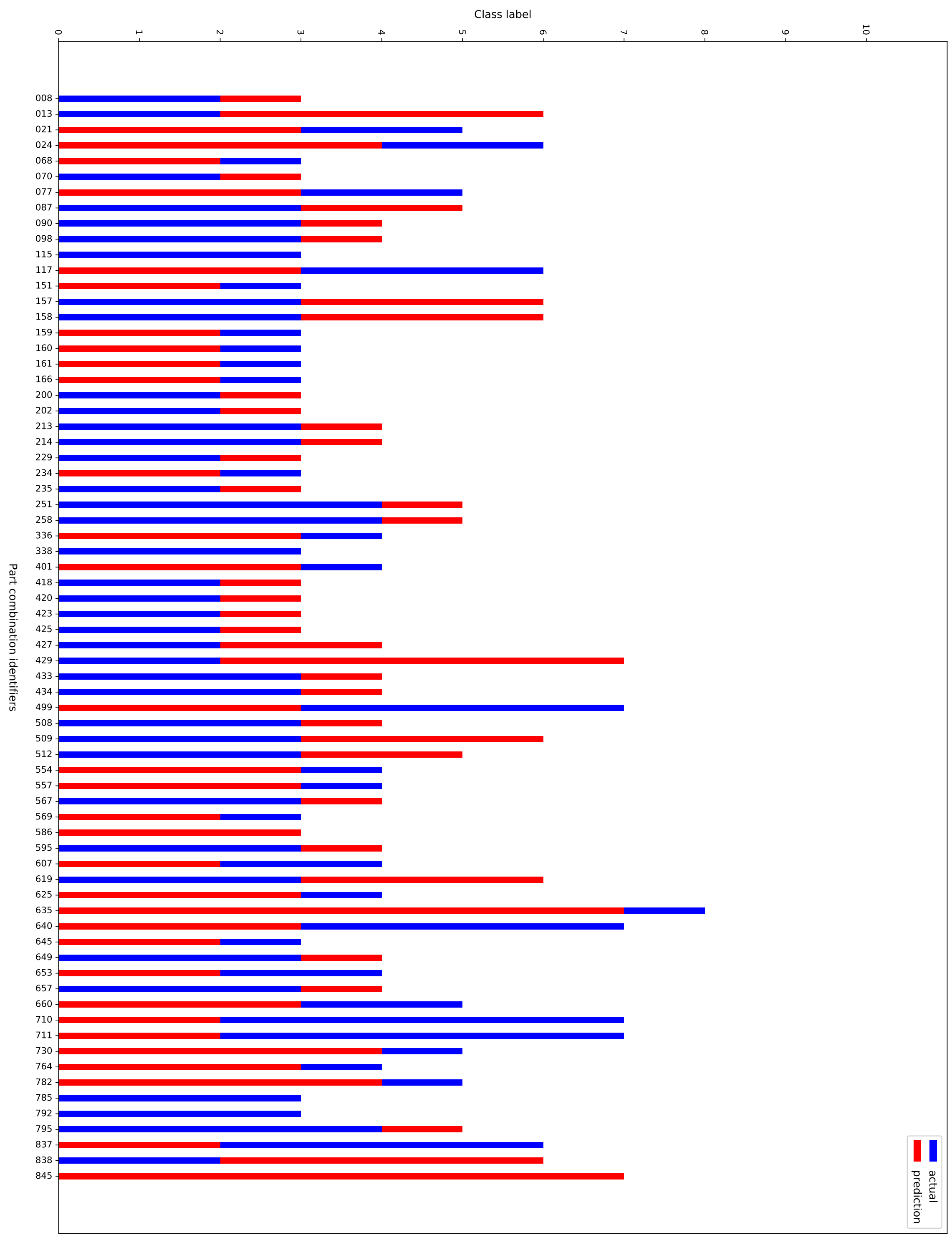


Figure 59: Prediction analysis for experiment 1

Now that were a able to predict class labels for part combinations which gives us an initial estimate of d_s, we can analyze the wrong predictions of the trained model to check deviation from the accurate results. For the spot weld design, it would be really inaccurate to use the prediction of the trained model as an initial estimate for generation of spot welds when prediction is greater than d_s. In cases when the prediction is less than d_s, we would have an increased number of spot welds which can then be reduced to optimum number using design iterations. With this basic idea, we analyze the wrong predictions of the trained PointNet model. 803/873 class label predictions were accurate. For 36 part combinations, the class label predictions were greater than actual labels. For 34 part combinations, the class label predictions were less than actual labels. Figure (59) shows a plot of prediction versus actual label for wrongly labeled part combinations. We can see that out of 36 cases, when the prediction is greater than actual label, in 24 instances the prediction deviated by just one class label from the actual class. With these results we can see that this approach can be refined further and we can predict the minimum distance between spot welds using classification models.

Experiment 2

In previous experiment the class interval for which we labeled d_s values was 10 mm. In order to have more meaningful results we reduced this class interval to 2 mm. Like in previous case, we limited our predictions for part combinations with $d_s < 100$ mm. In this experiment also we labeled part combinations with single spot weld connection with label zero. We have 50 labels for values of d_s lying between $0 - 100$ mm as shown in Table (2).

Table 2: Labeling criteria for Experiment 2

criteria (mm)	label
$d_s = 0$	0
$0 < d_s < 2$	1
$2 < d_s < 4$	2
$\vdots$	$\vdots$
$98 < d_s < 100$	51
$d_s > 100$	52

The PointNet-Combi was trained with the newly labeled training data of part combinations on Nvidia GeForce GTX 750 Ti for $n_{points} = 1024$ with the following parameters

- batch size = 10
- learning rate = 0.001
- momentum = 0.9

- optimizer = $ADAM$
- number of samples in training set = $52,380$
- number of samples in test set = 873
- number of epochs = 200

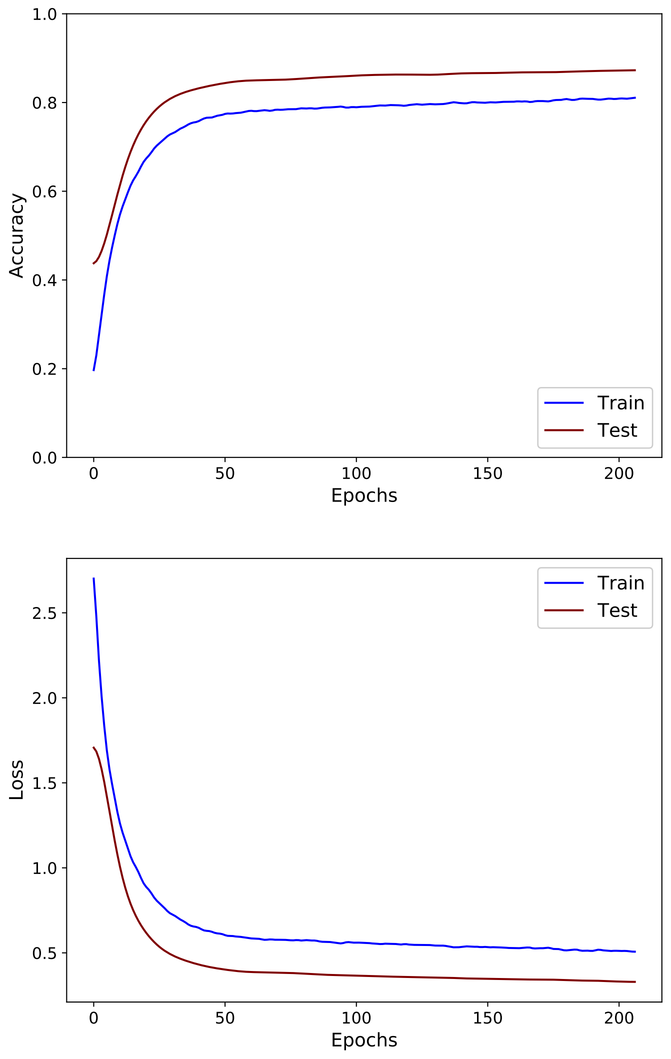


Figure 60: Performance plots for Experiment 2

Figure (60) shows the performance plots for Experiment 2. From the performance plots, we can see that we see that the evaluation accuracy of the model reached nearly 90% during training. The trained model was evaluated using a new set of point cloud for each part combination and the prediction accuracy was 86%. The trained model was able to predict the class labels of 758 part combinations correctly. The 115 wrong predictions by the trained model was analyzed. For 56 part combinations the predicted class label was less than the actual class label and for 59 class labels the predictions were greater than actual class label.

Thus with this trained model we are able to predict the minimum spot weld distance parameter d_s for 86% of the the available part combinations with an accuracy of ± 2 mm.

4.4.2 Prediction based approach

The basic idea of this approach was to use the identification capability of PointNet to identify the parts, and then use another neural network to estimate spot weld parameters. The approach is outlined in Figure (61). Here we first use PointNet to identify the parts from their point clouds and then use the part identification to predict spot weld parameters.PointNet is able to identify 96% part labels of AUDI model accurately. So in this approach we try to build a model which consumes part labels for two parts and predicts spot weld parameter d_s.

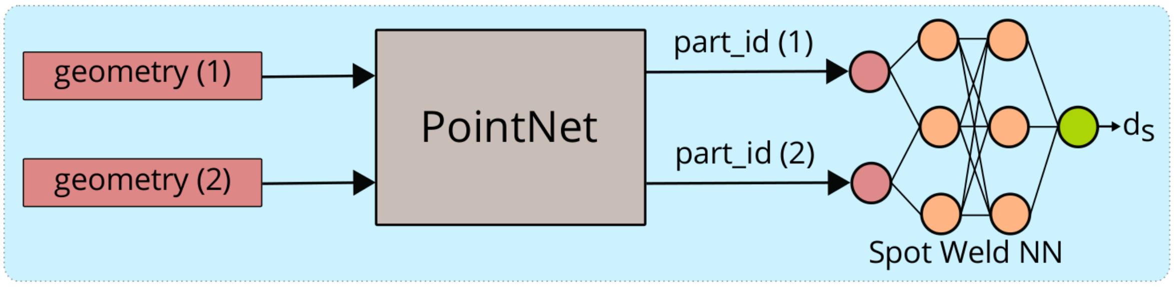


Figure 61: Prediction based approach

The model used which consumes part labels and predicts the spot weld parameters is a simple feed forward neural network with two or more hidden layers. In this work we utilized a feed forward neural network with two hidden layers. The parameter prediction model is a non linear model and we would require at least two hidden layers to have meaningful results. The parameter prediction model was implemented using TensorFlow. The important aspect in this model was data preparation and the model architecture.

The input data for parameter prediction model are two part labels which are numeric values. The output from the model is also a numeric value which is the prediction for a spot weld parameter. In this work we tried to estimate the minimum distance between the spot welds using this approach.

Training spot weld parameter predictor

In order to train the spot weld parameter predictor model, a training data set was generated from the AUDI model. The part labels and the spot weld parameter d_s were normalized in the range $[0, 1]$ for training the feed forward neural network. Since we did not have a bench marked architecture for this type of problem, many architecture were experimented upon and only the architecture which provided meaningful results has been presented in this work. A feed forward neural network with 150 neurons in the first hidden layer and 50 neurons in the second hidden layer was trained to predict the spot weld parameter d_s and the results obtained are outlined in this work. Back-propogation algorithm was used to train the model and gradient descent optimization was utilized with Mean Squared Error as the cost function. The model was trained with the following parameters

- batch size $= 10$
- learning rate $= 0.1$
- number of samples in training set $= 846$
- number of epochs $= 500,000$

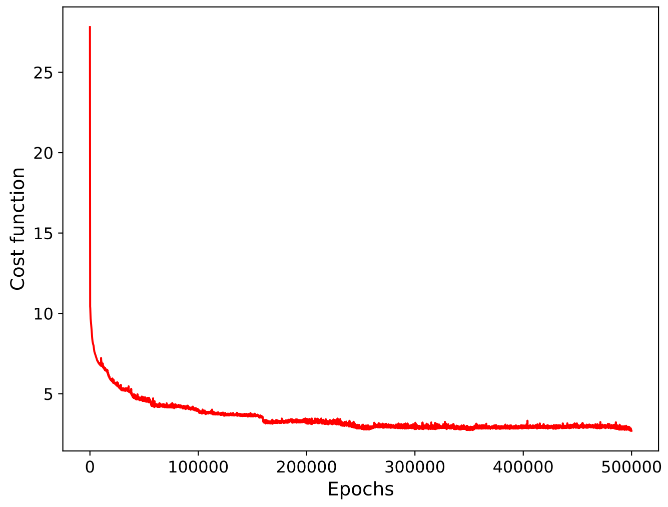


Figure 62: Training spot weld parameter predictor model

Figure (62) shows cost vs epochs plot during model training. The trained model was then used to predict spot weld parameter for the AUDI model. The results obtained are presented below. Figure (63) shows the predictions of the trained spot weld predictor model against the actual value of d_s for the part combinations of AUDI model. From the plot we can see that model performance is fairly good and in some cases the predicted value is exactly equal to the actual value. This model also provides us fairly meaningful results.

We calculated the deviation of the predicted values by the trained model so that we can have an estimate of the overall performance of the model. The deviation d_e was estimated using Eq. (4.1)

$$d_e = \text{Actual } d_s - \text{Predicted } d_s \tag{4.1}$$

Table (3) shows the performance of the trained model. The performance accuracy is calculated as the percentage of number of part combinations for which the d_e value satisfies the mentioned deviation criteria.

Table 3: Performance of spot weld predictor model

Deviation in mm	performance accuracy
$d_e = 0$	30.02%
$d_e < 2$	65.85%
$d_e < 5$	80.73%
$d_e < 10$	86%

Form the results in Table (3), we can say that when we correctly identify the part labels the spot weld predictor model is able to predict the minimum spot weld distance parameter d_s for 65% of the available part combinations with an accuracy of ± 2 mm.

4.4.3 Comparison of approaches

In order to evaluate the performance of the approaches mentioned in previous subsections, a comparison study was performed. The methodologies used to predict spot weld parameter d_s is different for Classification based approach and Prediction based approach.

Classification based approach directly consumes the point cloud of part combinations and provides a prediction of a interval of width 2 mm. For a part combination, visualized in Figure (64) , with d_s value equal to 10 mm, the model in Classification based approach consumes the point cloud of part combination as shown in Figure (65a) and provides a prediction that the d_s value for this particular part combination lies in the interval of width 2 mm. Thus when the model predicts correct class label, the accuracy of the prediction is within 2 mm of the actual value of d_s. Thus we can evaluate the accuracy of predictions and compare it with Prediction based approach.

Prediction based approach consumes point cloud of individual part in a part combination and provides a numeric estimation for d_s. It achieves this by using the point cloud of individual parts (Figure (65b)), first to identify them and then use a second Neural network to predict the value of d_s. In this approach, the model predicts a numeric value for d_s which can be directly compared to actual value of d_s for respective part combinations.

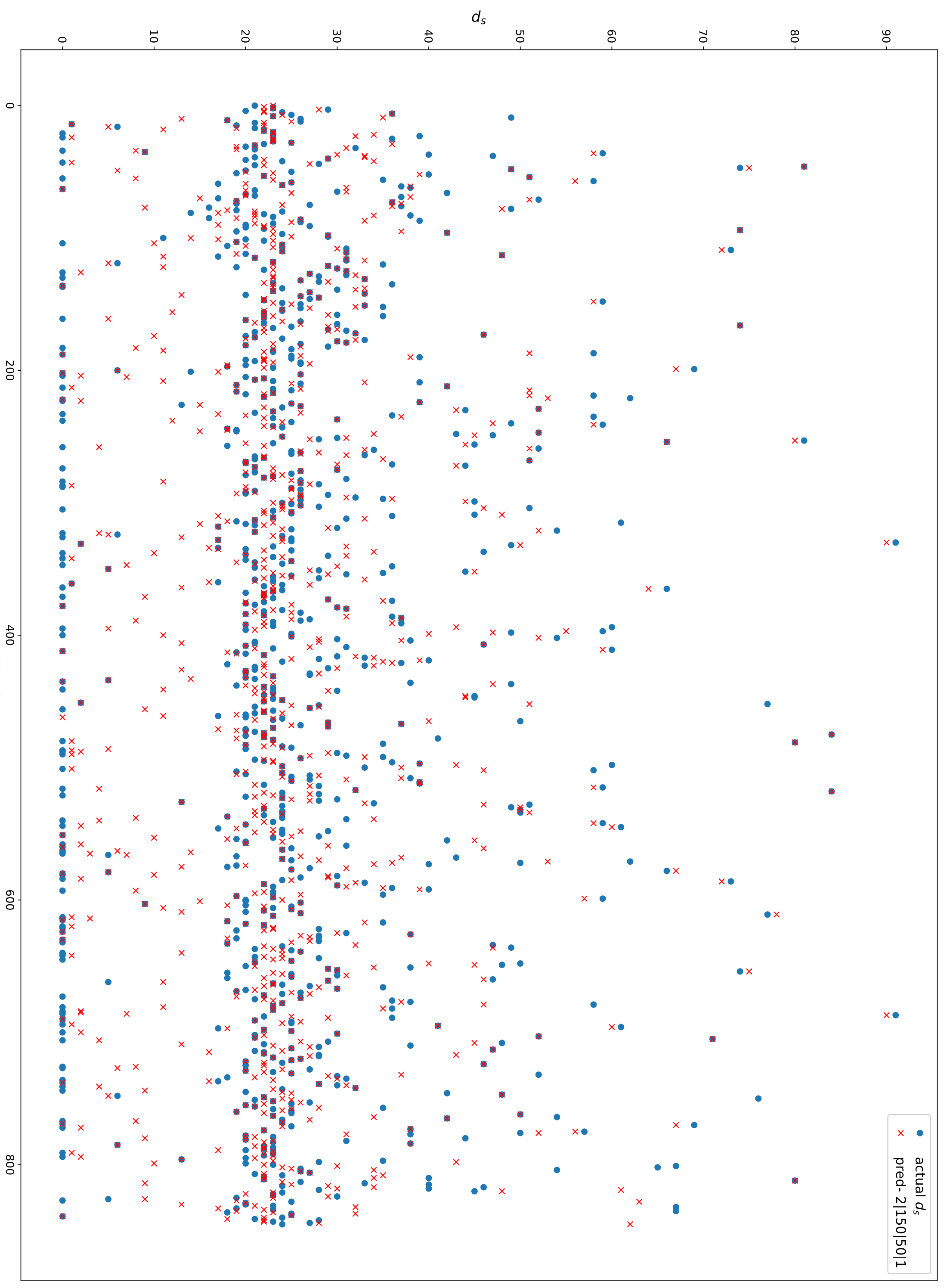


Figure 63: Predictions of spot weld predictor model

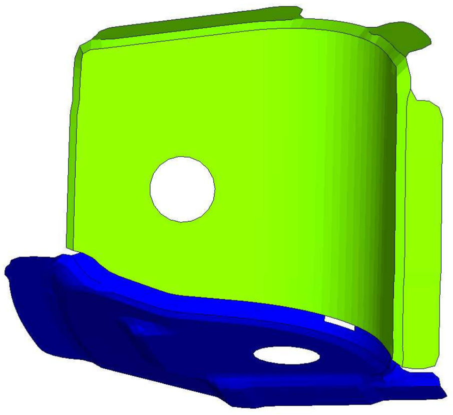
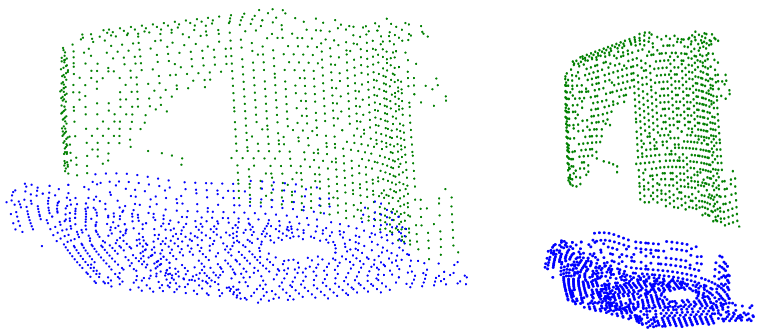


Figure 64: Visualization of example part combination



(a) Input for Classification based approach (b) Input for Prediction based approach

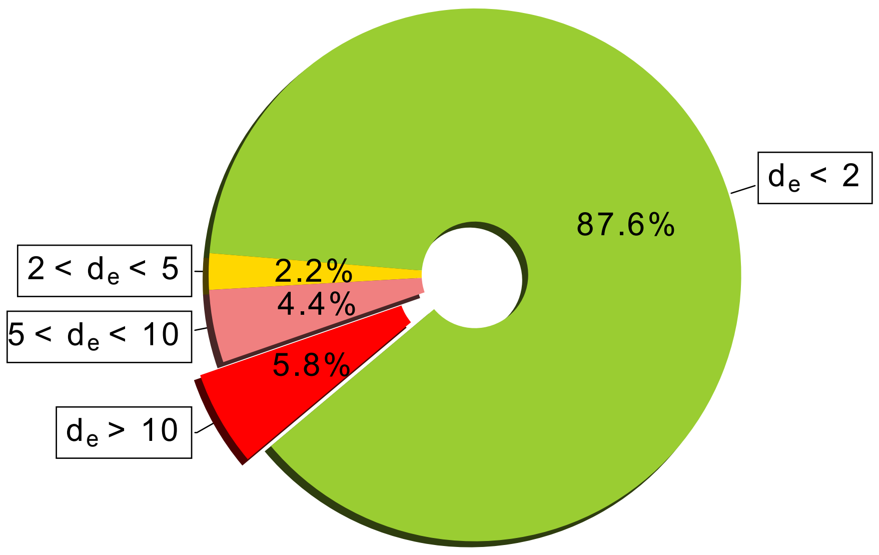
Figure 65: Visualization of input for different approaches

In order to evaluate the performance of both approaches we calculate the deviation of predicted value of d_s. We calculate the deviation d_e for both approaches differently. d_e for Classification based approach us estimated using Eq (4.2)

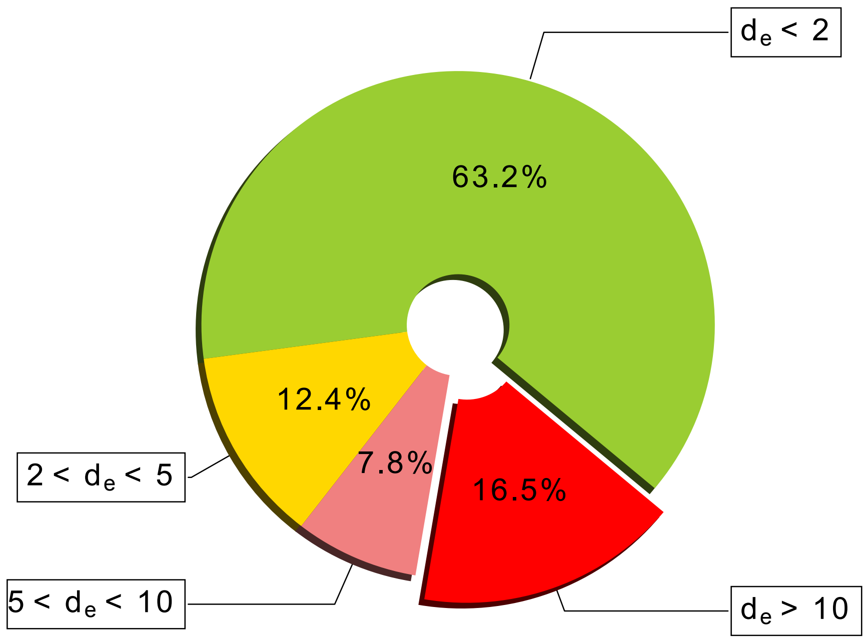
$$d_e^{Approach1} = (\text{Predicted } c_L - \text{Actual } c_L) * \text{Interval size} \tag{4.2}$$

where c_L is the class label and Interval size is the difference between upper limit and lower limit of the criteria defining the class labels. For Prediction based approach, we estimate d_e using Eq (4.1).

During evaluation of performance for Prediction based approach, we also factor in the fact that identification of PointNet for all parts of AUDI model is not 100%. We identified the wrongly labeled parts and propagated this error in identification to the final prediction of d_s. The results obtained are visualized in Figure (66).



(a) Classification based approach



(b) Prediction based approach

Figure 66: Comparison of approaches

In Figure (66), the results displayed are percentage of total number of part combinations analyzed. From the results, we can see that Classification based approach provides better estimation for larger number of part combinations as compared to Prediction based approach. But Classification based approach doesn't provide us the possibility to include parameters like sheet metal thickness or material properties in prediction of spot weld parameters. Prediction based approach provides good results with the possibility of extending the model to include the parameters mentioned above in order to have more realistic results.

Classification based approach and Prediction based approach have their own strengths and shortcomings. In Classification based approach, we are directly making use of geometric information of the parts to predict spot weld parameters whereas in Prediction based approach, the prediction of spot weld parameter has no relation to geometric information. Since this work is first of it's kind, we are looking for methods which can incorporate more input parameters for predicting spot weld design, thereby increasing the credibility of prediction, and thus Prediction based approach looks promising.

In future works, the performance of Prediction based approach can be improved by experimenting with different input parameters of neural network architecture. We can also conduct experiments on different architectures which can provide better performance when we consider more input parameters for prediction of spot weld design.

4.5 Further outlooks

In previous section, we evaluated different approaches to estimate parameters of spot weld design. With this work as base, we can develop methods to extract other spot weld design parameters like distance of the spot weld from part edge, spot weld diameter etc. Machine learning models can be trained to estimate these parameters as well. Furthermore, we can also develop algorithms to generate spot weld designs automatically using the predicted parameters.

Also in this work, only minor variations in architecture of PointNet was experimented with. We can also probe with making architectural changes to PointNet which will enable us to incorporate more input parameters for prediction of spot weld design. In order to do so, we will need to investigate methods which will enable us to encode geometric data and input parameters together as input for a machine learning architecture.

With the idea that PointNet has proved to identify parts using geometric data, we can develop other methods to parametrize the spot weld design. For example, paramterize the spot weld design as a geometry and then develop algorithms which can use this to generate spot welds directly on part geometries. In simple terms, we generate a map of spot weld design for each part combinations and then try to develop algorithms which will try to fit this map to part combinations when we have a newer version for one or both parts in the part combination.

5 Conclusion

In this work, we have developed and defined methods to extract significant input parameters of existing 3D geometry from FEM data. The extraction of significant parameters was completed with aid the library femparser and Python scripts. We also defined methods to generate point clouds of individual geometries from significant input parameters of existing 3D geometry. The identification of individual part using point clouds was achieved using the deep learning architecture called PointNet. The results of this classification problem were analyzed and we also tried to reason for wrong predictions of classification problem. Due to the complexity of modeling parts in FEM, for some parts extraction of individual geometries were not feasible and this has contributed to minor errors in identification problem. In further works, we should be able to solve this problem.

In order to estimate parameters for spot weld design, we first extracted significant input parameter of minimum distance between the spot welds for existing FEM data. This process was also achieved with aid of femparser and python scripts. In this work, we have outlined two different approaches which were used to estimate spot weld parameter. Evaluation of performances of these two approaches was conducted and their results are also compared.

Classification based approach dealt with the idea of directly using geometric data to predict parameters. It involved the transformation of prediction problem into a classification problem for PointNet. Prediction based approach used PointNet first to identify the parts from their geometries and then used this information to provide a prediction for spot weld parameter using a separate neural network. It was found out that, in order to improve the credibility of estimation of parameters for spot weld design, Prediction based approach provided suitable base for further research.

With this work, we are able to provide credible results for identification of parts using point cloud of parts and PointNet. We also have outlined approaches with which we estimated minimum distance between spot welds for part combinations. These approaches can further be developed for estimation of further parameters of spot weld design.

References

[1] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1998.

[2] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[3] D. E. Rumelhart, J. L. McClelland, and C. PDP Research Group, Eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations*. Cambridge, MA, USA: MIT Press, 1986.

[4] U. Reuter, *Module BIWO-04 "Numerical Methods"-Lecture notes.* Technische Universität Dresden, Fakultät Bauingenieurwesen, 2016.

[5] K. Janocha and W. M. Czarnecki, "On loss functions for deep neural networks in classification," *CoRR*, vol. abs/1702.05659, 2017. [Online]. Available: http://arxiv.org/abs/1702.05659

[6] Y. Lecun, *Generalization and network design strategies.* Elsevier, 1989.

[7] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[8] Y. T. Zhou and R. Chellappa, "Computation of optical flow using a neural network," *IEEE 1988 International Conference on Neural Networks*, pp. 71–78 vol.2, 1988.

[9] Y. Boureau, N. Le Roux, F. Bach, J. Ponce, and Y. LeCun, "Ask the locals: multi-way local pooling for image recognition," in *Proc. International Conference on Computer Vision (ICCV'11).* IEEE, 2011.

[10] E. Hille, *Analytic Function Theory*, 2nd ed. New York, NY, USA: Chelsea Publishing Company, 1982.

[11] T. DeRose, "Coordinate-free geometric programming," Seattle, Washingtion, Tech. Rep., 1994.

[12] E. Ahmed, A. Saint, A. E. R. Shabayek, K. Cherenkova, R. Das, G. Gusev, D. Aouada, and B. E. Ottersten, "Deep learning advances on different 3d data representations: A survey," *CoRR*, vol. abs/1808.01462, 2018. [Online]. Available: http://arxiv.org/abs/1808.01462

[13] I. K. Kazmi, L. You, and J. J. Zhang, "A survey of 2d and 3d shape descriptors," in *2013 10th International Conference Computer Graphics, Imaging and Visualization(CGIV)*, vol. 00, Aug. 2013, pp. 1–10. [Online]. Available: doi.ieeecomputersociety.org/10.1109/CGIV.2013.11

[14] H. Maron, M. Galun, N. Aigerman, M. Trope, N. Dym, E. Yumer, V. G. Kim, and Y. Lipman, "Convolutional neural networks on surfaces via seamless toric covers," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 71:1–71:10, Jul. 2017. [Online]. Available: http://doi.acm.org/10.1145/3072959.3073616

[15] Z. Cao, Q. Huang, and K. Ramani, “3d object classification via spherical projections,” *CoRR*, vol. abs/1712.04426, 2017. [Online]. Available: http://arxiv.org/abs/1712.04426

[16] A. Sinha, J. Bai, and K. Ramani, “Deep learning 3d shape surfaces using geometry images,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 223–240.

[17] J. Han, L. Shao, D. Xu, and J. Shotton, “Enhanced computer vision with microsoft kinect sensor: A review,” *IEEE Transactions on Cybernetics*, vol. 43, no. 5, pp. 1318–1334, Oct 2013.

[18] N. Erdogmus and S. Marcel, “Spoofing 2d face recognition systems with 3d masks,” in *2013 International Conference of the BIOSIG Special Interest Group (BIOSIG)*, Sep. 2013, pp. 1–8.

[19] G. Fanelli, T. Weise, J. Gall, and L. Van Gool, “Real time head pose estimation from consumer depth cameras,” in *Pattern Recognition*, R. Mester and M. Felsberg, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 101–110.

[20] M. Firman, “Rgbd datasets: Past, present and future,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2016.

[21] K. Lai, L. Bo, X. Ren, and D. Fox, “A large-scale hierarchical multi-view rgb-d object dataset,” in *2011 IEEE International Conference on Robotics and Automation*, May 2011, pp. 1817–1824.

[22] M. Tatarchenko, A. Dosovitskiy, and T. Brox, “Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017, pp. 2107–2115.

[23] J. Javanshir Hasbestan and I. Senocak, “Binarized octree generation for cartesian adaptive mesh refinement around immersed geometries,” *Journal of Computational Physics*, vol. 368, 12 2017.

[24] S. Lefebvre and H. Hoppe, “Parallel controllable texture synthesis,” in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH ’05. New York, NY, USA: ACM, 2005, pp. 777–786. [Online]. Available: http://doi.acm.org/10.1145/1186822.1073261

[25] J. Zhao, X. Xie, X. Xu, and S. Sun, “Multi-view learning overview: Recent progress and new challenges,” *Information Fusion*, vol. 38, pp. 43 – 54, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1566253516302032

[26] Su, Hang, Maji, Subhransu, Kalogerakis, Evangelos, Learned-Miller, and Erik, “Multi-view convolutional neural networks for 3d shape recognition,” in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[27] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, “Multi-view convolutional

neural networks for 3d shape recognition," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[28] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, July 2017.

[29] B.-Q. Shi, J. Liang, and Q. Liu, "Adaptive simplification of point cloud using k-means clustering," *Computer-Aided Design*, vol. 43, no. 8, pp. 910 – 922, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010448511000844

[30] J. Park, H. Kim, Y.-W. Tai, M. S. Brown, and I. Kweon, "High quality depth map upsampling for 3d-tof cameras," in *2011 International Conference on Computer Vision*, Nov 2011, pp. 1623–1630.

[31] J. C. Rangel, V. Morell, M. Cazorla, S. Orts-Escolano, and J. García-Rodríguez, "Object recognition in noisy rgb-d data using gng," *Pattern Analysis and Applications*, vol. 20, no. 4, pp. 1061–1076, Nov 2017. [Online]. Available: https://doi.org/10.1007/s10044-016-0546-y

[32] C. Yan, H. Xie, D. Yang, J. Yin, Y. Zhang, and Q. Dai, "Supervised hash coding with deep neural network for environment perception of intelligent vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 1, pp. 284–295, Jan 2018.

[33] M. B. Alexander Bronstein, "Discrete geometry tutorial," 2008. [Online]. Available: tosca.cs.technion.ac.il

[34] L. Cosmo, E. Rodolà, M. M. Bronstein, A. Torsello, D. Cremers, and Y. Sahillioğlu, "Partial matching of deformable shapes," in *Proceedings of the Eurographics 2016 Workshop on 3D Object Retrieval*, ser. 3DOR '16. Goslar Germany, Germany: Eurographics Association, 2016, pp. 61–67. [Online]. Available: https://doi.org/10.2312/3dor.20161089

[35] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao, "3d shapenets: A deep representation for volumetric shapes," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[36] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep. 2015, pp. 922–928.

[37] C. R. Qi, H. Su, M. Niessner, A. Dai, M. Yan, and L. J. Guibas, "Volumetric and multi-view cnns for object classification on 3d data," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[38] Y. Li, S. Pirk, H. Su, C. R. Qi, and L. J. Guibas, "Fpnn: Field probing neural networks for 3d data," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds.

Curran Associates, Inc., 2016, pp. 307–315. [Online]. Available: http://papers.nips.cc/paper/6416-fpnn-field-probing-neural-networks-for-3d-data.pdf

[39] D. Zeng Wang and I. Posner, “Voting for voting in online point cloud object detection,” in *Proceedings of the Robotics: Science and Systems, Rome, Italy*, 07 2015.

[40] H. Su, S. Maji, E. Kalogerakis, and E. Learned-Miller, “Multi-view convolutional neural networks for 3d shape recognition,” in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[41] M. Savva, F. Yu, H. Su, A. Kanezaki, T. Furuya, R. Ohbuchi, Z. Zhou, R. Yu, S. Bai, X. Bai, M. Aono, A. Tatsuma, S. Thermos, A. Axenopoulos, G. T. Papadopoulos, P. Daras, X. Deng, Z. Lian, B. Li, H. Johan, Y. Lu, and S. Mk, “Large-scale 3d shape retrieval from shapenet core55: Shrec’17 track,” in *Proceedings of the Workshop on 3D Object Retrieval*, ser. 3Dor ’17. Goslar Germany, Germany: Eurographics Association, 2017, pp. 39–50. [Online]. Available: https://doi.org/10.2312/3dor.20171050

[42] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, “Spectral networks and locally connected networks on graphs,” *CoRR*, vol. abs/1312.6203, 2013. [Online]. Available: http://arxiv.org/abs/1312.6203

[43] J. Masci, D. Boscaini, M. M. Bronstein, and P. Vandergheynst, “Geodesic convolutional neural networks on riemannian manifolds,” in *The IEEE International Conference on Computer Vision (ICCV) Workshops*, December 2015.

[44] Y. Fang, J. Xie, G. Dai, M. Wang, F. Zhu, T. Xu, and E. Wong, “3d deep shape descriptor,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[45] K. Guo, D. Zou, and X. Chen, “3d mesh labeling via deep convolutional neural networks,” *ACM Trans. Graph.*, vol. 35, no. 1, pp. 3:1–3:12, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2835487

[46] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “Pointnet: Deep learning on point sets for 3d classification and segmentation,” *arXiv preprint arXiv:1612.00593*, 2016.

[47] O. Vinyals, S. Bengio, and M. Kudlur, “Order Matters: Sequence to sequence for sets,” *arXiv e-prints*, p. arXiv:1511.06391, Nov 2015.

[48] Q. Charles, “pointnet,” Available at https://github.com/charlesq34/pointnet (2019/02/22), 2017.

[49] NVIDIA, “Cuda,” Available at https://www.geforce.com/hardware/technology/cuda (2019/02/22).

[50] “cudnn,” Available at https://developer.nvidia.com/cudnn (2019/02/22).

[51] “2010 toyota yaris detailed finite element model,” Available at https://www.ccsa.gmu.edu/models/2010-toyota-yaris/ (2019/02/22), 2016.

A FEM data extraction source code

A.1 *geometry_extraction.py*

```
# -*- coding: utf-8 -*-
from femparser import dynaparser
import json

filename = 'BIW.key'

# Defining parser

parser = dynaparser.DYNAParser()

with open(filename, 'r') as fem_model:
    mesh = parser.parse(fem_model)


# Extract PID's

part_ids = []
part_names = []
for part in mesh.parts:
    ids.append(mesh.parts[part].id)
    part_names.append(mesh.parts[part].name)


# Extract geometric data and save in
# json format


for id in part_ids:
    mesh_nodes = []
    for element in mesh.parts[id].elements:
        temp = []
        for nodeid in element.node_ids:
            temp.append(mesh.nodes[mesh.node_ids[nodeid]].
               ↪ tolist())
        mesh_nodes.append(temp)

    with open('part_geometries/{}_coordinates.json'.format(
       ↪ part_names.index(id)), "w") as fh:
        json.dump(mesh_nodes, fh, indent=4)
```

A.2 *spotweld_data_extraction.py*

```python
# -*- coding: utf-8 -*-

from femparser import dynaparser, pamparser
import json


def dyna_to_mesh(filename):
    parser = dynaparser.DYNAParser()

    with open(filename, 'r') as fem_model:
        return parser.parse(fem_model)

def pam_to_mesh(filename):
    return pamparser.Mesh(filename)


filename = 'audi_model.inc'

mesh = pam_to_mesh(filename)

plinks = [e for e in iter(mesh.get_containing_elements()) if
   ↪ isinstance(e, pamparser.Plink)]

connected_parts = []
spot_welds = []
for plink in plinks:
    if len(plink.pre_selection.parts) > 1:
        new_connection = {}
        p1, p2 = plink.pre_selection.parts
        connected_parts.append([p1, p2])
        new_connection['Part1'] = p1
        new_connection['Part2'] = p2
        new_connection['coordinates'] = mesh.get_node_by_id(
           ↪ plink.node_ids[0])
        spot_welds.append(new_connection)

with open('spot_weld_data.json', 'w') as fh:
    json.dump(spot_welds, fh, indent=4)
```

B Point cloud generation source code

B.1 *build_dataset.py*

```
#! -*- coding:utf-8 -*-

from pointcloud_generator import *
import os
import natsort

# Load part goemetries

path = 'Yaris/part_geometries/'
files = natsort.natsorted(os.listdir(path))
print(files)

# Generate part names for evaluation in PointNet

generate_shapenames_txt(files)

print('Number of parts = {}'.format(len(files)))

# Define number of samples to be generated
# for training PointNet

number_of_samples = 1000

# Define number of points to be generated in
# pointcloud

number_of_points = 1024


# Generate point clouds

labels = []
dataset = []
counter = 0

for i in range(number_of_samples):
    if counter == 0:
        index = i
    else:
        index = i % counter

    filename = files[index]
    labels.append(files.index(filename))
    points = point_cloud_combi(path, filename, number_of_points
       ↪ )

    print("processing {}/{}".format(i, number_of_samples))

    dataset.append(points)

    if i != 0:
        if i % len(files) == len(files) - 1:
            counter += len(files)


# Save point clouds in HDF5 format

with h5py.File("yaris_parts_train_1024.hdf5", "w") as f:
    dset = f.create_dataset("coordinates", data=dataset,
                            shape=(number_of_samples,
                               ↪ number_of_points, 4))
    dset2 = f.create_dataset("labels", data=labels,
                             shape=(number_of_samples, 1))
```

B.2 *pointcloud_generator.py*

```
#! -*- coding:utf-8 -*-

import json
import numpy as np
import random
from scipy import optimize
from pyDOE import *
import h5py
import glob
from random import choice
import os


def area_triangle(data):
    '''
    Calculates area of triangle
    '''
    AB = data[1] - data[0]
    AC = data[2] - data[0]
    area = 0.5 * np.linalg.norm(np.cross(AB, AC))
    return area


def surface_area(coor):
    '''
    Calculates area of elements
    '''
    data = np.array(coor)
    s = 0
    if len(data) == 4:
        s = area_triangle(data[[0, 1, 2], :])
            + area_triangle(data[[0, 2, 3], :])
    else:
        s = area_triangle(data)
    return s


def quality_check_weights(weights, n):
    '''
    Function to correct errors generated during
    rounding of weights
    '''
    if weights.sum() > n:
        extra = int(weights.sum() - n)
        n_largest_ind = np.argpartition(weights, -extra)[-extra
            ↪ :]
        weights[n_largest_ind] = weights[n_largest_ind] - 1
    else:
        diff = n - weights.sum()
        pos_zeros = np.where(weights == 0)
        if len(pos_zeros[0]) < diff:
            for index in pos_zeros[0]:
                weights[index] = 1
            for i in range(int(diff - len(pos_zeros[0]))):
                weights[i] = weights[i] + 1
        else:
            while n - weights.sum() != 0:
                index = random.choice(pos_zeros[0])
                weights[index] = 1
    return weights


def barycentric(data, n):
    '''
    Generates point inside a triangle
    based on barycentric coordinates
    & LHS sampling
    '''
    lhd = lhs(2, samples=int(n), criterion='c')
    a = np.sum(lhd, axis=1)
    t = []
    for l in lhd:
        t.append((l[0] * data[0] + l[1] * data[1] + data[2] *
           ↪ (1 - (l[0] + l[1]))).tolist())
    return t


def generate_point(data, n):
    '''
    Function to generate points inside element
    '''
    data = np.array(data)
    if len(data) == 4:
        a1 = surface_area(data[[0, 2, 3], :])
        a2 = surface_area(data[[0, 1, 2], :])
        if n > 1:
            w1 = round(n * (a1 / (a1 + a2)))
            w2 = n - w1
            t1 = barycentric(data[[0, 2, 3], :], w1)
            t2 = barycentric(data[[0, 1, 2], :], w2)
            t = t1 + t2
        elif n == 1:
            if a1 > a2:
                t = barycentric(data[[0, 2, 3], :], 1)
            else:
                t = barycentric(data[[0, 1, 2], :], 1)
    else:
        t = barycentric(data, n)
    return t


def point_cloud(path, filename, n):
    """
    Function to generate point cloud for
    individual parts
    'path' = directory to part jsons
    'filename' = part name
    'n' = number of points in point cloud
    """

    with open(path + filename) as f:
        data = json.load(f)

    input_data = np.array(data)
    areas = []
    for data in input_data:
        areas.append(surface_area(data))

    areas = np.array(areas)
    weights = np.around((np.asarray((areas / areas.sum()).
        ↪ astype(float)) * n),
                        decimals=0)

    weights = quality_check_weights(weights, n)  # required
        ↪ because of rounding

    points = []
    for i, data in enumerate(input_data):
        if weights[i] != 0:
            gp = generate_point(data, weights[i])
            for coordinates in gp:
                points.append(coordinates)

    return points


def point_cloud_combi(path, filename, n):
    '''
    Function to generate point cloud for
    part combinations
    '''
    with open(path + filename) as f:
        data = json.load(f)

    n_new = n / len(data)

    points = []
    for part in data:
        input_data = np.array(part)
        areas = []
        for data in input_data:
            areas.append(surface_area(data))

        areas = np.array(areas)

        weights = np.around((np.asarray((areas / areas.sum()).
            ↪ astype(float)) * n_new),
                            decimals=0)

        weights = quality_check_weights(weights, n_new)

        for i, data in enumerate(input_data):
            if weights[i] != 0:
                gp = generate_point(data, weights[i])
                for coordinates in gp:
                    # coordinates.append(part_num)   #uncomment
                        ↪  to insert part0 / part1
                        points.append(coordinates)

    points = np.array(points)
    rotation_angle = np.random.uniform() * 2 * np.pi
    cosval = np.cos(rotation_angle)
    sinval = np.sin(rotation_angle)
    rotation_matrix = np.array([[cosval, 0, sinval],
                                [0, 1, 0],
                                [-sinval, 0, cosval]])

    rotated_points = np.dot(points, rotation_matrix)
    points = rotated_points.tolist()

    count = 0
    part_num = 0
    for row in points:
        row.append(part_num)
        count += 1
        if count == n_new:
            part_num = 1
    return points


def generate_shapenames_txt(files):
    with open('shape_names.txt', 'w') as f:
        for filename in files:
            basename, ext = os.path.splitext(filename)
            pid, name, coordinates = basename.split('_')
            f.write('{}\n'.format(pid + '_' + name))
```

C Neural network for prediction based approach

```
# -*- code: utf-8 -*-

import tensorflow as tf
import pickle
import time


def data_provider(filename):
    with open(filename, 'rb') as fh:
        input_data = pickle.load(fh)
    return input_data


# Import training data

training_data = data_provider('input_data_100.pickle')
training_value = data_provider('values_100.pickle')


# Define model architecture

batch_size = 100
x_ = tf.placeholder(tf.float32, shape=[None, 2], name='x-input'
    ↪ )
y_ = tf.placeholder(tf.float32, shape=[None, 1], name='y-input'
    ↪ )

Theta1 = tf.Variable(tf.random_uniform([2, 150], -1, 1), name="
    ↪ Theta1")
Thetat = tf.Variable(tf.random_uniform([150, 50], -1, 1), name=
    ↪ "Theta-t")
Theta2 = tf.Variable(tf.random_uniform([50, 1], -1, 1), name="
    ↪ Theta2")

Bias1 = tf.Variable(tf.zeros([150]), name="Bias1")
Biast = tf.Variable(tf.zeros([50]), name="Biast")
Bias2 = tf.Variable(tf.zeros([1]), name="Bias2")

with tf.name_scope("layer2") as scope:
    A2 = tf.sigmoid(tf.matmul(x_, Theta1) + Bias1)

with tf.name_scope("extra-layer") as scope:
    extra = tf.sigmoid(tf.matmul(A2, Thetat) + Biast)

with tf.name_scope("layer3") as scope:
    Hypothesis = tf.sigmoid(tf.matmul(extra, Theta2) + Bias2)

with tf.name_scope("cost") as scope:
    cost = tf.reduce_sum(0.5 * (y_ - Hypothesis) ** 2)

with tf.name_scope("train") as scope:
    train_step = tf.train.AdamOptimizer(0.01).minimize(cost)


# Train the model

init = tf.global_variables_initializer()
sess = tf.Session()

sess.run(init)

num_batches = int(len(training_data) / batch_size)

logger = open('log.txt', 'w')
t_start = time.clock()
iter_no = 500000
for i in range(iter_no):
    for batch_idx in range(num_batches):
        start_idx = batch_idx * batch_size
        end_idx = (batch_idx + 1) * batch_size
        current_input = training_data[start_idx:end_idx]
        current_y = training_value[start_idx:end_idx]
        sess.run(train_step, feed_dict={x_: current_input, y_:
           ↪ current_y})
    if i % 100 == 0:
        print('Epoch ', str(i).zfill(len(str(iter_no))), '----'
           ↪ , 'cost ', sess.run(cost, feed_dict={x_:
           ↪ training_data, y_: training_value}))
        logger.write('{} {}\n'.format(i, sess.run(cost,
           ↪ feed_dict={x_: training_data, y_: training_value
           ↪ })))
t_end = time.clock()
print('Elapsed time ', t_end - t_start)


# Test the model and write out predictions

test_data = training_data
test_value = training_value

with open('predictions_spwd_150_50.txt', 'w') as fh:
    for i, value in enumerate(test_data):
        pred = sess.run(Hypothesis, feed_dict={x_: [value]})
            ↪ [0][0]
        print('Actual value = {} --- Prediction = {}'.format(
            ↪ test_value[i][0], pred))
        fh.write('{} {:.2f}\n'.format(test_value[i][0], round(
            ↪ pred, 2)))
```