Training Optimization of Feedforward Neural Network

Omkar Thawakar (Author)
SGGSIE&T
Nanded, India
omee0805@gmail.com

Pranav Gajjewar(Author)
SGGSIE&T
Nanded, India
2015BCS025@sggs.ac.in

Abstract—**In this paper, we present a novel technique to reduce the training time of a feedforward neural network by intuiting some of the parameters involved in construction and initialization of the network. These estimated parameters include the number and size of the hidden layers along with the weights related to the neurons. The weights and network architecture is estimated before training by formulating an approximation of the \emph{decision boundary} we want the network to learn. This specific configuration will allow the network to learn the optimum weights in less iterations than in the case of random initialization of weights.**

Keywords-component; formatting; style; styling; insert (key words)

I. INTRODUCTION (HEADING 1)

We briefly discuss the basic concepts required for the proper understanding of the optimization method.

A. **Multi-Layer Perceptron**

Feedforward Neural Network or Multi layer perceptron (MLP) model is a fundamental machine learning tool used to solve many different types of problems. It can be used for classification, regression, function approximation, etc. The network learns its behaviour by modeling the weights of the neural connections contained in it. It typically consists of an input layer, an output layer and one or more hidden layers. Each neural connection is associated with a weight value. One feature of this network is that the neurons are all fully connected which means that each neuron is connected to all the neurons in the previous layer as well as all the neurons in the next layer. Hence it is also known as fully connected neural network. The output of the network is calculated by passing the value of input neurons to the next. Thus the activation value of any one neuron is calculated using the activation values of all the neurons in the previous layer.
This activation is given by -

$$A^l = \sigma(W^l . A^{l-1} + B^l) \quad (1)$$

Where A^l = Activation in Layer l
A^{l-1} = Activation in Layer l-1
W^l = Weights matrix of layer l
B^l = Bias matrix of layer l
σ = Activation function

The activation value of each neuron is dependant on the activation values of the neurons in the previous layer and the weights of the connections. When a specific neuron is activated (has a relatively high activation value), it activates some specific set of neurons in the next layer and so on until the output layer where a specific set of activations give the final output. This property allows the network to map a wide range of input patterns to the required outputs at the final layer. One can imagine the intermediate set of neurons and connections as a function Φ mapping input pattern space to output.

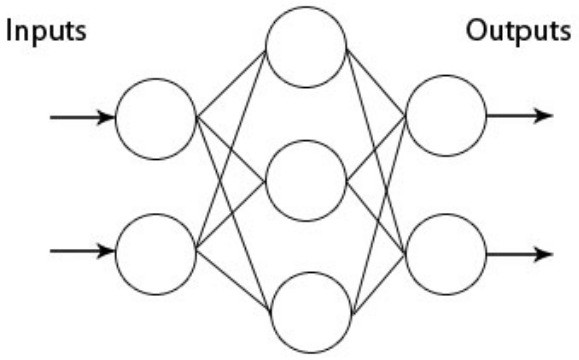


Fig. 1. A small multi-layer perception network

By adding enough number of neurons and layers, a network can emulate any function Φ. Hence MLP is known as universal function approximator [1]. This property is widely documented in research literature, see [1] [2] [3]. and others. The important thing needed is the configuration of weights that describe a specific mapping function.

B. **Backpropogation**

Learning is the task of finding the correct weight values. Given a set of training samples, the network by itself determines the best weights to map the input patterns to labels during the training phase. A common learning algorithm that is used for training a neural network is *Backpropagation*.

978-1-5386-8260-9/19/$31.00 ©2019 IEEE

The training phase of a network involves following steps-

- Initialize the weights of the network based on a weight initialization scheme.
- Define a loss function based on the network's output and the original output.
- Do a forward pass of the network with input data to obtain network's output.
- Find the error between original output and the network's output for all input data.
- Make changes to the weights based on this error such that the network's output becomes closer to the original output.

Generally, the loss function used for training, referred as square loss, is defined as-

$$E = \sum (Y - Y')^2 \quad (2)$$

Where Y = desired output for the input pattern
Y' = network's predicted output

In the backpropagation algorithm, the change in the weight values is calculated using gradient descent. We take the derivative of the loss function with respect to each weight parameter in the network. This means we are calculating how much contribution a specific weight had in the final output of the loss function. This derivative is denoted by ΔW. The weight change is given by-

$$W_{new} = W_{old} - \Delta W \quad (3)$$

$$\Delta W = -\eta(d_i - F(W^t * X))F^{|}(W^t * X)X \quad (4)$$

Where η = learning rate

Thus after each iteration over data, we obtain weights that decrease the loss value and make the network's output closer to the original output.

Here learning rate (η) plays a significant role in learning the final appropriate weights of network. Higher the value of η, greater the probability of missing the global minimum of loss function. So in standard practice, value of η is conventionally minimum. But problem with a small value of η is that the rate of approach towards the desired final weights becomes slow and it takes many training cycles for the network to get fully trained. To avoid this, it is better to initialize the weights of network close to the desired value so that our network trains faster and global minimum value of loss is achieved in less iterations.

C. **Visualizing network's learning**

A practical example will demonstrate the above discussed concepts in action. Suppose we have a binary classification problem in which we want the network to predict the class of the input data point. The data for the classification task is as follows-

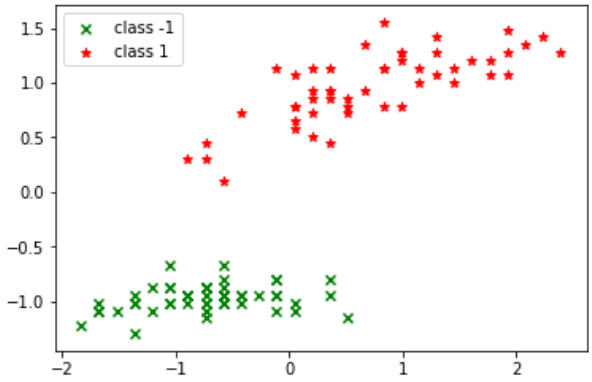


Fig. 2. An example data distribution for binary classification task

A single neuron network is used for this task as we can see from Figure 2 that the data is linearly classifiable. We define the loss function as given in Eqn (2). The networks should learn some general logic using which it can predict whether a certain input data point is class -1 or class 1.

Geometrically, we want the network to find a boundary in the input pattern space which will distinguish between the two classes. This is called a decision boundary or classification curve.

Looking at Figure 2, we see that a linear boundary can be set between the two class points. All points on one side of this boundary are class -1 and others class 1. We use backpropagation to train our network on input data.

The network is trained for 20 iterations over all the data while updating weights after every data point evaluation. When the loss value has decreased sufficiently, we say that the network has learned the weights to classify the points. The decision boundary learned by the network is visualized in Figure 3.

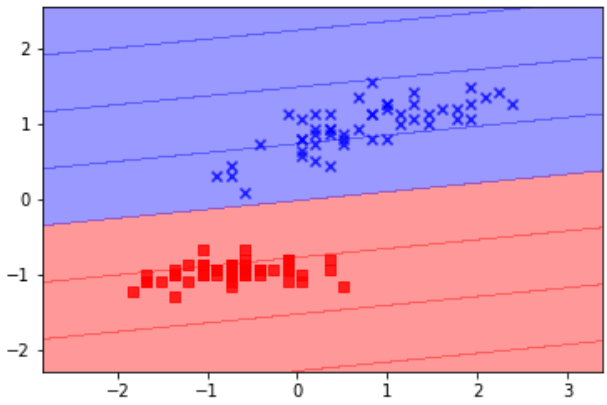


Fig. 3. Decision boundary for classifying the points.

The parameters that we initialized randomly before were adjusted using gradient descent to arrive at weights that define the obtained decision boundary. At the time of

initialization, instead of choosing the parameters randomly, if we had chosen the parameters such that they would form a decision boundary close to what the network learned on its own then the parameters would have taken less adjustment to arrive at the same decision boundary that it did in the above case. The importance of an optimum weight initialization is quite evident both in achieving an optimum training and in achieving very fast learning. Very good and fast results can obviously be obtained when a starting point of the optimization process is very close to an optimum solution [4].

II. RELATED WORK

We take a brief look at some of the previous studies done regarding training optimization of a FNN. There have been many studies regarding optimization of the backpropagation algorithm, originally given by *Rumelhart et. al* in 1980s, to reduce the training time of MLP, see [5], [6] & [7]. Relatively less attention has been given to the task of weight initialization even though the weight initialization scheme used is pertinent to network learning and directly impacts the training performance of the network.

If weights initialized are close to the global optimum, any descent algorithm can take the network weights toward the optimum fast and reliably. Thus the goal of our weight initialization method is to answer the question-

``Given labelled data, how to derive parameter initializations that are close to the parameter values that define the optimum solution for our task?"

In a multi-layer network of n arbitrary depth, there is no method to find the exact parameters of the final output curve learned by the network. Such an analysis has been attempted by backpropagating the desired response through the layers of a multilayer perceptron (MLP) using the analytic linear least squares solution [8]. Another solution for single-hidden layer FNNs-- by application of singular value decomposition (SVD) on the input data matrix considering the Taylor expansion of the mapping problem and the nonlinearity of sigmoids, an approximation of mapping function Φ is derived [9]. Both of these solution are limited in scope and do not present a method to derive weights for multi-hidden layer networks.

There is a lack of generalized method to initialize the parameter values of a network of any depth such that it will improve upon the random weight initialization scheme at the task of learning. In this paper, we attempt an intuitive approach to determine a loose approximation of the mapping function Φ and derive the weight initialization based on it. It is not an analytical solution so a good understanding and intuition regarding data separation and curve behaviours is requisite for deriving good results.

We mention Drago and Ridella's *Statistically controlled activation weight initialization* study [4]. They do not attempt to derive the weight initialization based on estimation of mapping function but instead explore the amplitude window in maximum magnitude of the initial weights such that the neuron doesn't enter a *paralyzed* or *saturated* state. Small initial values lead to poor local minima, while large ones lead to *saturated* state. We incorporate this property of initial weights in the results section when we use a *scaling factor* to reduce the weights to a range compatible with the range derived from the above study predicting optimum learning performance.

Finally, an overview and discussion of various weight initialization schemes can be found here [10].

III. METHODOLOGY

In feedforward neural networks, the output of the final layer is dependant on initial layer outputs as a function $f(*)$. If $f_n(*)$ and $f_{n-1}(*)$ are the activations of final and initial layer then the output of the network can be written as $f_n(f_{n-1}(*))$. Thus for a network of n layers represented as a directed acyclic graph, the output is given by-

$$f_n(f_{n-1}(\ldots.(f_2(f_1(*))))) \tag{5}$$

Each neuron in the subsequent layer maps the outputs of the neurons in previous layer to a new region in pattern space. This new region assumes a boundary that is better able to classify the points than the regions defined by previous neurons. In this way, by combining the regions defined by previous neurons the final classifcation region and \emph{decision boundary} is derived. This phenomenon is shown in Figure 5.

Our approach is to initialize the weights of the neural network by visualizing the *decision boundary* of final layer and splitting the curve in terms of lines such that the final curve can be said to be a combination of these lines represented by the first layer neurons of the network.

Observe Figure 6, we can easily determine the shape of the curve which can effectively classify all the points in the input data set. A *decision boundary* learned by a neural network at Tensorflow Playground [11] is shown in Figure 4. We can approximately tell what *classification curve* the network intends to emulate to attain the global minimum of loss function. It is not possible for us to give an analytical solution for this *curve*. But we know that a FNN has the ability to approximate any curve including this one. So we can create a network that can approximate this *curve* after we train it.

Fig. 4. An example of a non-linear decision boundary.

A. **Reasoning behind pre-estimation of weights**

During training, feedforward neural network weights are adjusted using backpropagation as shown in Eqn 3 and Eqn 4.

In Eqn 4, η represents the learning rate which gives the proportion of each error signal to be considered for a weight update. It is important to choose a small value for $ \eta$ because using a large value will not allow the network to converge to the global minimum of the loss function. Hence $ \eta$ is generally initialized as 0.001. But the problem with this η is that the weight updates at each step are a small proportion of the error signal. If the weights that we initialized randomly at the start need large adjustment to arrive at the optimum weights, then using this η value will take a large number of update steps. Thus the training process can be made faster by pre-estimating weights such that they need only little adjustment to become the optimum weights needed for the network.

B. **Deriving the network architecture**

The method we present to estimate the network architecture is based on the intuitive understanding of the *decision boundary* that we want to emulate using the network.

A single neuron in the first layer (after input layer) derives a linear *boundary* (in case of 2 input dimensions, a line) to classify the input points. The forward pass of this neuron is given by Eqn 1. The weights in W^1 and B^1 for this neuron form 3 parameters. These 3 parameters also give the equation of the line in 2D space which is the *decision boundary* for this neuron. As we move deeper into the network, the outputs of the first layer are passed to the next layer. The neurons in this next layer combine the outputs of the previous layer to derive its own version of the *classification curve*. At this point, we get a non-linear *boundary*. This concept is represented in Figure 5.

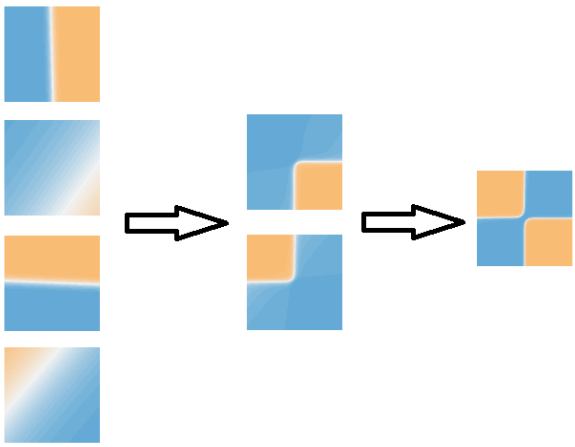


Fig. 5. Decision boundaries as we move deeper into the network

If we had two neurons in the first layer and one neuron in the output layer, then the output neuron can approximate a non-linear curve by combining the linear boundaries of the previous neurons with its own weights and bias. Using this understanding of how neurons learn the *decision boundaries*, we can approximate the arrangement of neurons which can be used to achieve the required *classification curve*.

Suppose we have linearly separable 2D data, then a single neuron can learn the linear *boundary*. So for this task, we would use no hidden layer and one output neuron. Similarly, if we want to achieve a *decision boundary* as shown in Figure 4, then we would need to combine at least 3 linear boundaries together to get the resultant *classification curve* approximating the required *curve*. This type of reasoning is further explained in the results section with use of multiple examples. This method will not give the most minimalistic architecture but provides a framework for constructing a network that will work well with our estimated weights.

To summarize, the number of lines that we need to sufficiently represent the behaviour of the curve is the number of neurons in the first hidden layer of our network. The size of the subsequent layer depends on how you intend to combine the linear curves of the first layer. A rule of thumb being that you use 1 neuron for every 2 linear curves in the first layer.

C.**Pre-estimating network parameters**

Just as we derive the network architecture, we obtain the parameters that are used for network initialization using the *classification curve* we want the network to learn. Note that in the case of multiple hidden layers, only neurons in the first layer are initialized with pre-estimated weights. Any subsequent layers will have its parameters initialized randomly. In the results section, we show that pre-estimating only the first layer parameters is enough to improve upon the *learning* of network with random parameter initialization.

The forward pass of a single neuron is given by-

$$A = \sigma(W.I + B) \tag{6}$$

Where A= output of neuron

W= weights of the connections

I= input points

B= bias for the neuron

σ= activation function

In case of 2D input data (i.e I is a 2D vector), the neuron has 2 weight parameters in the weight vector W and one bias parameter B associated with it. The *decision boundary* represented by this neuron will be a line. The standard equation of a line requires 3 parameters as seen in Eqn 7.

$$a\,x + b\,y = c \tag{7}$$

From Eqn 6 & Eqn 7, we give the equivalence as seen in Eqn 8. Using this we can visualize the *decision boundary* learned by a neuron by the values of its weights and bias.

$$W = \begin{bmatrix} a \\ b \end{bmatrix} \qquad B = c \tag{8}$$

Once we determine the lines to use for pre-estimating the parameters of the network, we can derive the weight and bias values from the parametric equation of these lines. Practically, we can derive the lines by plotting any two points lying on the line on the graph and using the *2-point* equation as seen in Eqn 9 to get the parameter values.

$$(y_1 - y_2)x + (x_2 - x1)y = y_1(x_2 - x_1) - x_1(y_2 - y_1) \quad (9)$$

where (x_1, y_1) & (x_2, y_2) are the two points.

Thus the network architecture and the network parameters are derived from the estimated *classification curve*. This technique is applied to example data distributions to show the training optimization of the network.

IV. RESULTS

In this section, we demonstrate the optimization method on multiple examples. A network with randomly initialized weights and pre-estimated weights is trained over the sample data. Its training performance is measured in terms of decrease in loss value.

A. 2D pattern space

I. *Circular Decision Boundary*

In this example, we use a feed-forward neural network to classify the binary labelled distribution as shown in Figure 6. This data distribution is taken from one of the classification examples at Tensorflow Playground [11].

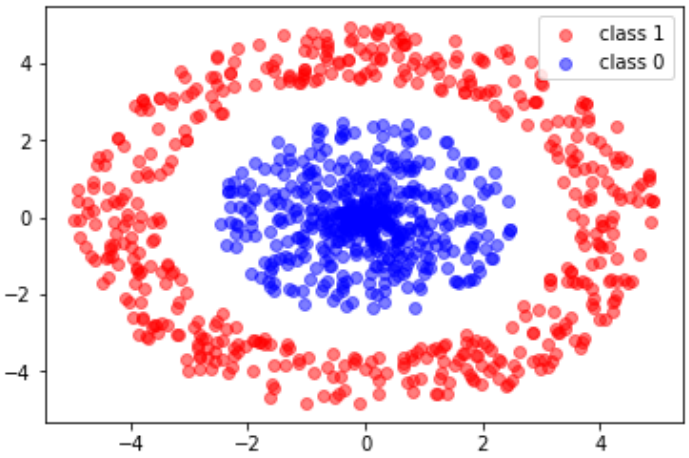


Fig. 6. Binary labelled input data distribution

From the graph, it is visually evident to us what *decision boundary* the network should approximate to classify the points correctly. Based on this knowledge, we will construct and initialize our network. We want a circular classification curve enclosing the *class 0* points. We can imagine three separate lines combining to form a circle like curve in the center. Utilizing this intuition, we can determine that our network's first hidden layer should have 3 neurons (each classifying the data linearly) and then an output neuron which will add the necessary non-linearity to get the lines to converge in the required shape.

We can derive a line by determining any two points in the plot that lie on the line. We assume the three lines for the first layer classification as seen in Figure 7.

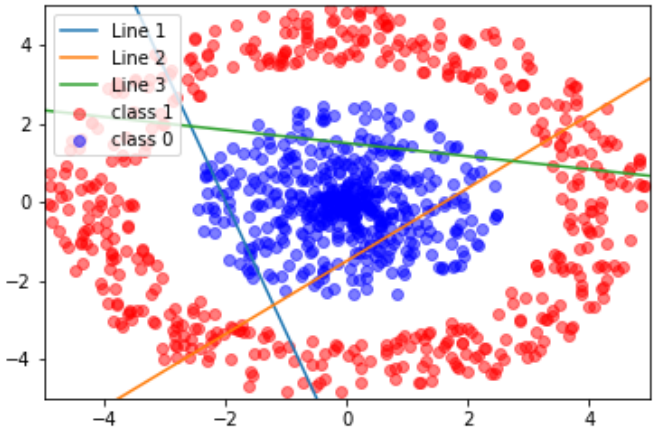


Fig. 7. Binary labelled input data distribution

As we see from Figure 7, the lines bounding the classification curve do not need to be exact because during training the weights and bias of the output neuron will determine the best way to combine these lines and to derive the final curve.

TABLE I
Points used for estimating lines that describe the required classification curve.

Line	Points
1	(1,-10),(-5,10)
2	(0,-1.5),(7,5)
3	(-3,2),(3,1)

We use a neural network whose configuration is shown in Figure 8. *ReLU* activation is used for hidden layer neurons and *sigmoid* activation is used for output layer neuron. We initialize two instances of the network. One where weights are initialized randomly from a uniform distribution and another in which the first layer's weights are pre-estimated using the parameters derived from lines seen in Figure 7.

For training, we use RMSProp mini-batch gradient descent optimization algorithm with a learning rate of 0.001 and a batch size of 16. We train both instances of the network for 30 epochs measuring the value of loss after each epoch. The results of this training can be found in the graph shown in Figure 9.

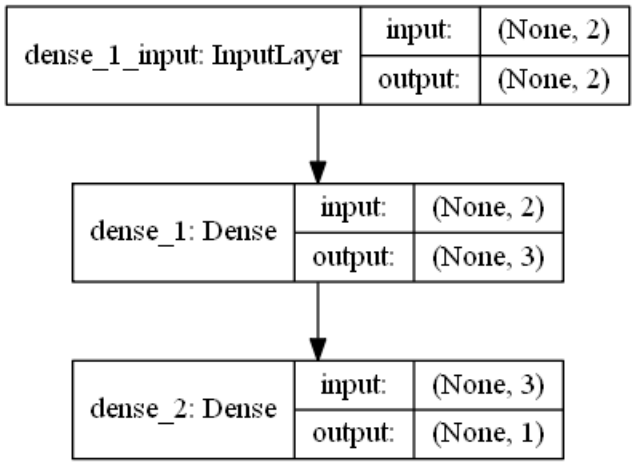
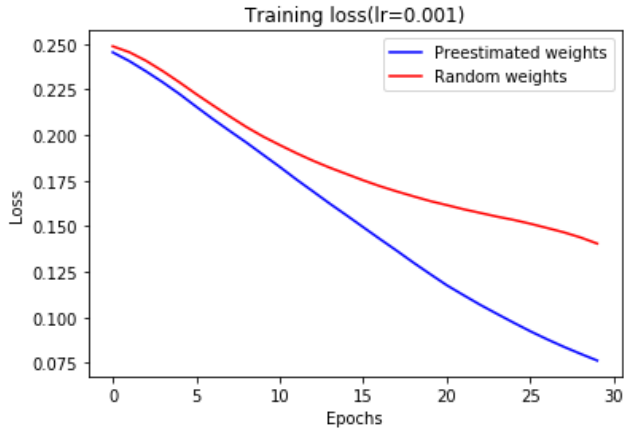


Fig. 8. Neural network with one hidden layer.

Figure 9 shows the loss curve for the two instances of our network. We can observe that the network with pre-estimated weights for the first layer converges to a lower loss value quicker than the network with all the weights initialized randomly. After performing multiple different initializations for the random weights in both the networks, we observe that



the network with pre-estimated weights consistently outperforms the other network in terms of converging to a lower loss value in less epochs

Fig. 9. Value of loss at each epoch for two networks.

II. Spiral decision boundary

This is another 2D classification task taken from Tensorflow Playground [11] which shows a more complex decision boundary than the previous example. The distribution of labelled data points is observed in the Figure 10.

This data distribution will require a spiral classification curve to classify the points. An example of a classification curve as learned by a neural network on the Tensorflow Playground [11] is present in Figure 11.

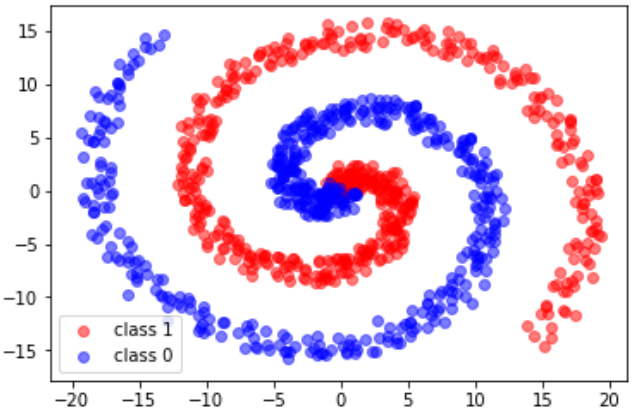


Fig. 10. Spiral data distribution.

Even without the *decision boundary* visualization provided here, it should be apparent the type of curve that can classify the concerned labelled data distribution. According to our method, we derive a general network architecture to approximate this *classification curve*. We define 8 lines that bound this curve and effectively describe its behaviour. These lines are presented in Figure 12. The choice of 8 lines might seem arbitrary but by observing the bends of the classification curve, it is seen that these lines sufficiently describe the curve for our purposes. Using the idea that the two linear boundaries combine with a neuron in next layer to give a non-linear boundary as explained in the methodology section, we use a rule of thumb to derive the size of the second hidden layer. We use 4 neurons in the second hidden layer with the notion that each neuron will focus on two linear curves from previous layer to give some non-linear curve. Although at the time of training, this is not necessarily the way these neurons will work but this intuition allows us to derive a general size that will work effectively. The question might arise to some as to why a second a layer was needed. From the complexity of the *curve*, it can be said that to achieve a *curve* like that we need a combination of different non-linear curves instead of just a combination of linear curves. Hence we find it necessary to add a hidden layer which will output 4 non-linear *decision boundaries* which will combine with the neuron in the last layer to approximate the required *classification curve*.

The resultant network architecture is shown in Figure 13. The same network configuration as previous example is used with *ReLU* activation for the hidden layer neurons and \emph{sigmoid} activation for the output layer neuron. The points used to derive the lines seen in Figure 12 are shown in Table 2. Based on these lines, we derive the parameters for the first layer neurons of our network in the preestimation case. Note that using the equation given in methodology section, our choice of points for this task will yield parameters relatively large in value. We want our parameters to be small

so that our neurons don't start out saturated. A scaling factor is used to scale back the parameters by dividing each parameter with it. In this case, a scaling factor of 40 is used with the consideration that resultant parameters are not too small or too large effectively in the range [-2, 2].

TABLE 2
Points used for estimating lines that describe the required classification curve.

Line	Points
1	(15,3),(-10,15)
2	(7, 13), (15, -10)
3	(0, -16), (10, -12)
4	(-20, -5), (-5, -20)
5	(0, -12), (7, 0)
6	(0, -12), (-10, -6)
7	(-13, -2), (-9, 15)
8	(-5, 16), (10, 14)

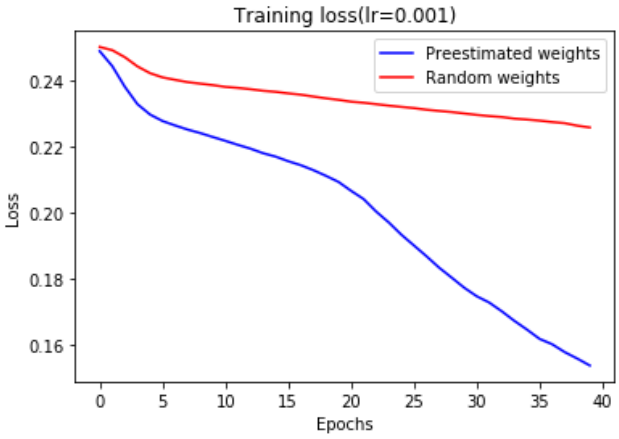


Fig. 14. Loss curve for the two network instances

For training, we use RMSProp mini-batch gradient descent optimization algorithm with a learning rate of 0.001$and a batch size of 16. We train two instances of the network, one with pre-estimated parameters in the first layer and other with all parameters initialized randomly from a uniform distribution. We train for 40 epochs measuring the value of loss after each epoch. The results of this training can be found in the graph shown in Figure 14.

The results show that the loss value for the network instance with pre-estimated weights using our method decreases significantly quicker than the network instance with random weights. We can explain this disparity between the performance of two instances with the fact that randomly initialized parameters at the time of initialization are a large distance away from the optimum configuration that the network should learn. In the other network instance the first layer neurons are configured to be already close to the optimum configuration and hence require less adjustment.

III. **3D pattern space**

In this section, we look at how this method can be applied to 3D pattern space i.e 3D input data. All the concepts explained in the methodology are applicable to 3D input data as well and we show this with examples. Only notable change is that neurons with 3D input learn a linear curve not as a line in 2 dimensions but as a 3D plane. The parameteric standard equation of this plane gives us the weights and bias associated with the neuron.

A. 3D linear decision boundary

The dataset in this example was taken from here [12]. It measures variance, curtosis and skewness. The training set contains 1372 data points labelled as class 0 or class 1. The distribution of the data points is shown in Figure 15.

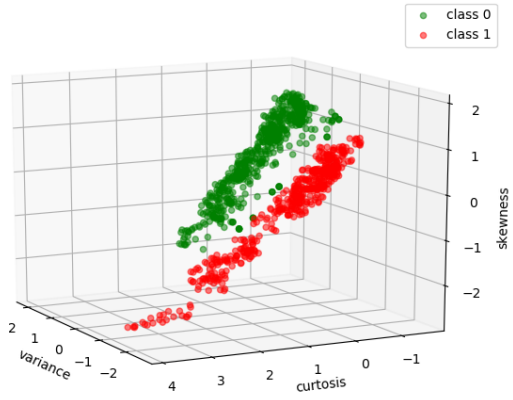


Fig. 15. 3D input labelled data distribution

The data points are linearly separable. A plane passing between the points will correctly classify all the points. Hence we need a network to approximate a linear *decision boundary*. Since we are learning a linear *classification curve*, we will only need a single neuron in our network. As a single neuron in 2D input space approximates a line as *decision boundary*, a single neuron approximates a planar *decision boundary*. Our network consists of one output neuron with no activation applied and 4 parameters consisting of 3 weights of the connections from input layer to the neuron and a bias associated with the neuron.

We use 3 non-collinear points lying in the plane to determine the parametric equation of the plane. The points $ (-3,-5,5), (0,5,-10), (-5,-2,0)$ are used for this example. We

get the equation of the plane as- $5x - 45y - 29z = 65$. A scaling factor of 100 is used to derive parameters as-

$$W = \begin{bmatrix} 0.05 \\ -0.45 \\ -0.29 \end{bmatrix} \quad B = [0.65] \tag{10}$$

We use RMSProp mini-batch gradient descent optimization algorithm with a learning rate of 0.001 and a batch size of 16. We train two instances of the network, one with pre-estimated parameters and other with randomly initialized parameters. We train for 50 epochs measuring the value of loss after each epoch. The results of this training can be found in the graph shown in Figure 16.

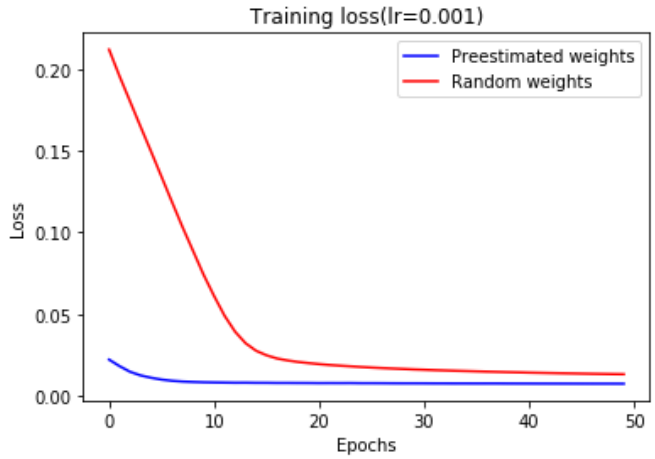


Fig. 16. Loss curve for 3D input dataset

In this case, our initial weight approximation was already very close to the optimum value of weights and hence it converged to the global minimum far more quickly than in the case of randomly initialized weights. We can observe the learned decision boundary as plotted in 3D in Figure 17. A linearly separable distribution is a trivial case. We take a look at a more complex *decision boundary* in 3D space in the next example.

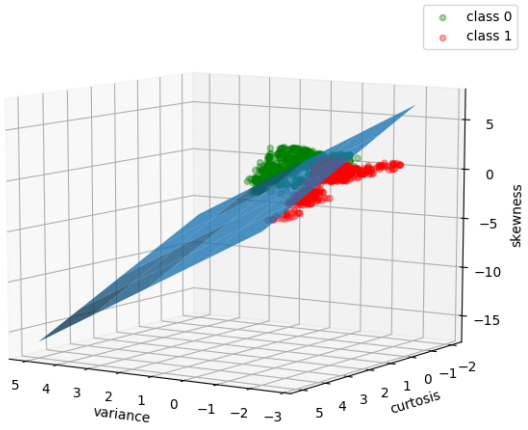


Fig17. Planar decision boundary in linearly separable distribution

B. 3D Non linear decision boundary

In this example, we use Skin Segmentation dataset which is constructed over B, G, R color space. Skin and Nonskin dataset is generated using skin textures from face images of diversity of age, gender, and race people. This dataset contains ~250000 labelled points. Due to the high quantity of points, a full distribution is not possible. This is an extreme example which highlights the limitations in our approach.

After carefully analysing the various visually feasible subsets of data points, we derive an estimation of *decision boundary* as 3D cylinderical curve (imagine a circle extended in 3rd dimension) with some inclination such that points lying inside this region belong to one class and points outside the region belong to another class. This is similar to the circular *decision boundary* we solved in 2D examples. We can derive a network architecture for emulating this \emph{decision boundary}. Considering that the cylinderical \emph{curve} can be obtained by combination of 3 planes through an output neuron for added non-linearity, we arrive at a single hidden-layer architecture with 3 neurons in the hidden layer and 1 output neuron. We use \emph{ReLU} activation for hidden layer neurons and \emph{sigmoid} activation for the output neuron.

We use RMSProp mini-batch gradient descent optimization algorithm with a learning rate of 0.001 and a batch size of 16. We train two instances of the network, one with pre-estimated parameters and other with randomly initialized parameters. We train for 50 epochs measuring the value of loss after each epoch. The results of this training can be found in the graph shown in Figure 18.

If we look at the loss curves, the one with pre-estimated weights shows a slight advantage over the curve for randomly initialized weights. But this advantage could be attributed to the variance in randomly initialized parameters. This does not represent a significant increase in performance from MSE loss' perspective. But if we plot the classification accuracy for both the networks, as seen in Figure 19, we observe that the pre-estimated weighted network converges to a higher accuracy value more rapidly than the other network. This remains consistent across multiple parameter initializations thus accounting for the variance from random initializations.

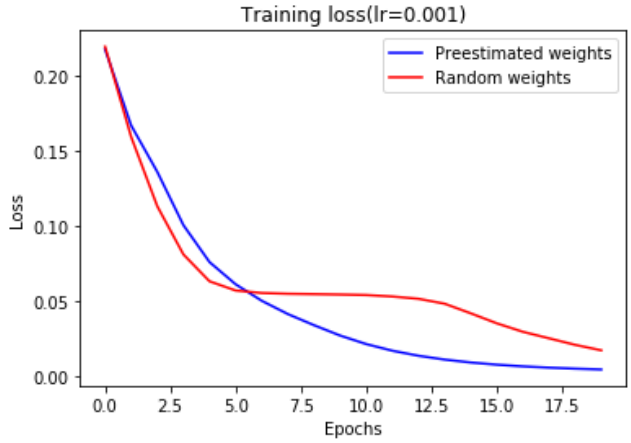


Fig. 18. Loss curve for 3D input

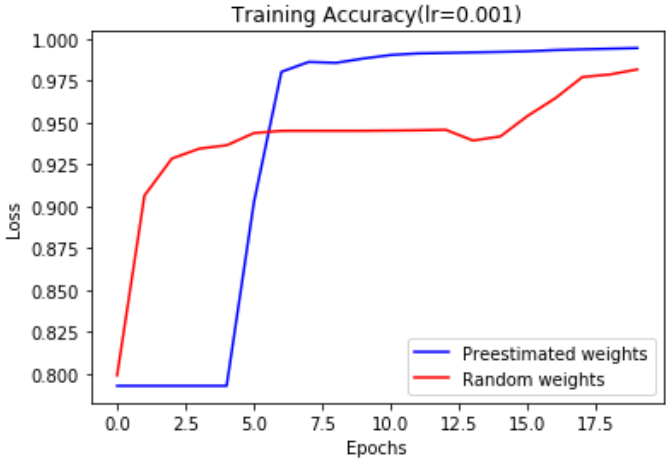


Fig. 19. Accuracy curve for 3D input

V. **Conclusion**

In this paper, we introduce a new approach to initialize the weights of neural network, an idea of using pre-estimated weights to reduce the training cycles. With pre-estimated weights our network converge rapidly towards global minimal solution. Our method has strong empirical performance over challenging benchmark and presents a new research direction for training the neural network with best suitable weights for convergence. Our method is flexible so that it can applied on multi-layer neural architectures. With further work, it can also be extended to n dimensional pattern spaces.

Acknowledgments

The authors would like to thank SGGSIE\&T, Nanded for supporting this paper work. Gratitude is extended towards Dr. U.V.Kulkarni for support in transit of helping this paper.

REFERENCES

[1] Kurt Hornik, Maxwell Stinchcombe and Halbert White, "Multilayer feedforward networks are universal approximators" Neural Networks Volume 2, Issue 5, 1989, Pages 359-366

[2] S. Geva and J. Sitte, "A constructive method for multivariate function approximation by multilayer perceptrons," in IEEE Transactions on Neural Networks, vol. 3, no. 4, pp. 621-624, Jul 1992.

[3] Pinkus, Allan. 1999. “Approximation Theory of the MLP Model in Neural Networks.” Acta Numerica 8. Cambridge University Press: 143–95

[4] G. P. Drago and S. Ridella, “Statistically controlled activation weight initialization (SCAWI)", IEEE Transactions on Neural Networks, vol. 3pp. 627–631, 1992

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations of backpropagation errors” Nature, vol. 323, pp. 533–536, 1986

[6] R. A. Jacobs, “Increased rates of convergence through learning rate adaptation,” Neural Netw., vol. 1, no. 4, pp. 295–308, 1988.

[7] S. Amari, “Natural gradient works efficiently in learning,” Neural Computat., vol. 10, pp. 251–276, 1998.

[8] D. Erdogmus, O. Fontenla-Romero, J. C. Principe, A. Alonso-Betanzos, and E. Castillo, “Linear-least-squares initialization of multilayer perceptrons through backpropagation of the desired response,” IEEE Transactions on Neural Networks, vol. 16, no. 2, pp. 325–337, 2005.

[9] P. Costa and P. Larzabal, “Initialization of supervised training for parametric estimation,” Neural Processing Letters, vol. 9, pp. 53–61, 1999.

[10] C. A. R. de Sousa, "An overview on weight initialization methods for feedforward neural networks," 2016 International Joint Conference on Neural Networks (IJCNN), Vancouver, BC, 2016, pp. 52-59.

[11] A Neural Network Playground https://playground.tensorflow.org/

[12] Banknote authentication Data Set by Volker Lohweg (University of Applied Sciences, Ostwestfalen-Lippe, volker.lohweg@hs-owl.de) \& Helene DÃ¶rksen (University of Applied Sciences, Ostwestfalen-Lippe, helene.doerksen@hs-owl.de)