Scary Machines*

*Examing the Effects of Logistic Regression and Multiple Layer Perceptron

1st Jinda Dong
Southern University of Science and Technology(SUSTC)
Systems Design and Intelligent Manufacturing(SDIM)
Shenzhen, China
Ethy9160.official@gmail.com

***Abstract*—This paper provides a concise exploration of two classical machine learning methodologies—logistic regression and Multi-layer Perceptron (MLP)—with the objective of assessing and comparing their effectiveness and performance across diverse application scenarios. Beginning with an overview of basic theory, the fundamental principles of logistic regression and multilayer perceptron are delineated. Subsequently, the specific implementations of these methods are introduced, followed by a testing and evaluation of their performance.**

The experimental phase involves a thorough analysis of multi-layer perceptron performance across various datasets. Emphasis is placed on the adjustment of learning rates and iteration numbers, addressing the vanishing gradient issue, selecting appropriate activation functions, and scrutinizing the influence of neural layer complexity on model efficacy. The exploration extends to the application of batch normalization techniques as a strategy to mitigate the vanishing gradient problem.

***Index Terms*—artificial intelligent, machine learning, logistic Regression, multi-layer perceptron (MLP), gradient problems**

I. Introduction

The course lessons have provided a foundational understanding of the fundamental theorems underlying logistic regression and multilayer perceptron, both extensively utilized tools in machine learning. Leveraging this knowledge, the researcher embarked on implementing the structures of logistic regression and multilayer perceptron. This paper is dedicated to elucidating the methodologies employed in constructing these structures, conducting tests to evaluate their performance, and engaging in discussions pertaining to observed phenomena and future prospects.

The experimental environment comprises Python 3.9, C++11, AMD Ryzen 5900H processor, and Windows 11 Home edition(version 10.0.22621).

II. Establishment on Logistic Regression and Neuron Networks

In fact, a logistic regression model can be regarded as a model with single-layer MLP, so details of which will be simplified in this report.

Identify applicable funding agency here. If none, delete this.

A. Realization of Logistic Regression

The forward calculating formula of a logistic regression model is considered as:

$$z = \boldsymbol{\omega}^T \boldsymbol{x} + b \tag{1}$$

Usually the activation function is chosen by the sigmoid function:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

Thus, the predict value $\hat{y}$ is:

$$\hat{y} = \sigma(z) \tag{3}$$

In that case, the formula of an logistic regression model can be written as:

$$y = \frac{1}{1 + e^{-(\boldsymbol{\omega}^T \boldsymbol{x} + b)}} \tag{4}$$

And:

$$e^{-(\boldsymbol{\omega}^T \boldsymbol{x} + b)} = 1 - \frac{1}{y} \tag{5}$$

Which is to say:

$$\ln \frac{y}{1 - y} = \boldsymbol{\omega}^T \boldsymbol{x} + b \tag{6}$$

In binary classification, y is usually the probability of "yes" or "no". Let $y = P(Y = 1|x) = p(x)$.

$$\ln \frac{P(Y = 1|x)}{1 - P(Y = 1|x)} = \boldsymbol{\omega}^T \boldsymbol{x} + b \tag{7}$$

In binary classification, output Y can only be "yes"(1) or "no"(0), and they're independent incidents, assume:

$$\begin{aligned} P(Y = 1|x) &= p(x) \\ P(Y = 0|x) &= 1 - p(x) \end{aligned} \tag{8}$$

And generating the likelihood function:

$$L(\omega) = \Pi[p(x_i)]^{y_i}[1 - p(x_i)]^{1 - y_i} \tag{9}$$

And taking the logarithm:

$$\ln L(\omega) = \sum [y_i \ln p(x_i) + (1 - y_i) \ln(1 - p(x_i))] \tag{10}$$

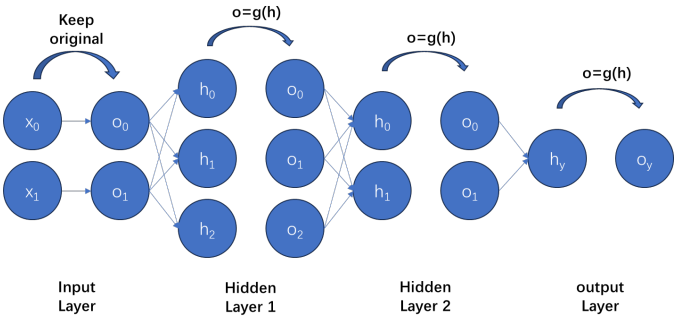
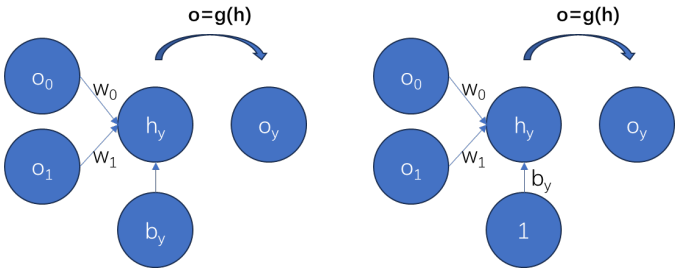


Fig. 1: Architecture of a Certain MLP(without bias)



(a) Decision boundary using sigmoid.

(b) Consider b_y a weight with input 1.

Fig. 2: Architecture of the output layer.

Apply Eq.4, Eq.6, Eq.7:

$$\ln L(\omega) = \sum [y_i(\omega \cdot x_i) - \ln(1 + e^{\omega \cdot x_i})] \tag{11}$$

We always want to maximize the likelihood of the current outcome, so we want to get the parameters that maximize the likelihood function ω. By taking the derivative of the parameter $\boldsymbol{\omega}$, we can gradually obtain the most appropriate parameter value $\boldsymbol{\omega}$.

A loss function is used to evaluate the degree of prediction error. Thus the likelihood function can be considered as a way to evaluate the prediction result.

$$loss = E = -\frac{1}{N} \ln L(\omega) \tag{12}$$

And the logistic model can be seen as a combination of linear regression with an activation function, recall the backward calculation of a linear regression model, the backward calculating formula in logistic regression can be written as:

$$\omega_i^* \leftarrow \omega_i - \alpha \frac{\partial E}{\partial \omega_i} \tag{13}$$

With:

$$\frac{\partial E}{\partial \omega_i} = (p(x_i) - y_i) * x_i^{(j)} \tag{14}$$

And y_i is the label of the data, with $p(x_i)$ as the prediction. Iterate to get better parameter $\boldsymbol{\omega}$. According to the formula, code in Appendix.1 provides an implementation method.

B. Realization of Multilayer Perceptron

Suppose the neural network architecture of an MLP is configured as shown in Fig. 1. The calculating process is similar to the logistic regression, and the loss function is the Mean-Square Error(MSE) as shown in 15.

$$loss_{MSE} = \frac{1}{N} \sum (\hat{y_i} - y_i)^2 \tag{15}$$

Divide the configuration into an output layer and hidden layer for detailed analysis.

- Backward Calculating in Output Layer

Assuming the neurons proximate to the output layer are depicted in Fig. 2, in the output layer, $E = \frac{1}{2}(y_{pred} - y_{label})^2$, and in the graph shown, $E = \frac{1}{2}(o_y - y_{label})^2$.

Consider the formula in 13, we're going to decompose the process of calculating the partial.

$$\frac{\partial E}{\partial o_y} = o_y - y_{label} \tag{16}$$

Activation function is $g(x)$, then $o_y = g(h_y)$, that is to say:

$$\frac{\partial o_y}{\partial h_y} = g'(h_y) \tag{17}$$

And the calculation of h_y is given by:

$$h_y = \sum_i \omega_i o_i + b_y \tag{18}$$

That is to say,

$$\frac{\partial h_y}{\partial \omega_i} = o_i \tag{19}$$

It seems uneasy to find the bias directly. But if we consider it a weight on a constant input 1, shown in Fig. 2b, apply 16, 17 and 19:

$$\frac{\partial E}{\partial b_y} = (o_y - y_{label}) \cdot g'(h_y) \cdot 1 \tag{20}$$

Finally, in the output layer, the formula of backward calculating is:

$$\begin{aligned} \frac{\partial E}{\partial \omega_i} &= (o_y - y_{label}) \cdot g'(h_y) \cdot o_i \\ \frac{\partial E}{\partial b_y} &= (o_y - y_{label}) \cdot g'(h_y) \end{aligned} \tag{21}$$

The calculation formula in the hidden layer is similar to the output layer, but if directly calculate, it's not easy. Recall 13, we can write the partial derivative in:

$$\begin{aligned} \frac{\partial E}{\partial \omega_{ki}} &= \frac{\partial E}{\partial o_k} \frac{\partial o_k}{\partial h_k} \frac{\partial h_k}{\partial w_{ki}} \\ &= \frac{\partial E}{\partial h_k} \frac{\partial h_k}{\partial \omega_{ki}} \end{aligned} \tag{22}$$

And let's consider a structure of the hidden layer shown in Fig. 3.

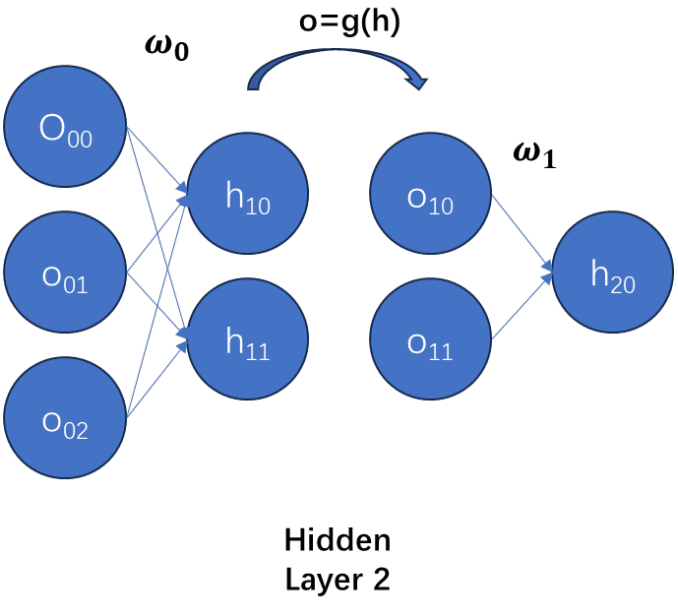


Fig. 3: A certain structure of hidden layer.

Assuming vectors $\boldsymbol{o_0}, \boldsymbol{\omega_0}, \boldsymbol{h_1}, \boldsymbol{o_1}, \boldsymbol{\omega_1}, \boldsymbol{h_2}$ are the matrix form of the elements in the shown structure. We get:

$$\begin{aligned} \boldsymbol{h_1} &= \boldsymbol{\omega_0} \cdot \boldsymbol{o_0} \\ \boldsymbol{h_2} &= \boldsymbol{\omega_1} \cdot \boldsymbol{o_1} \\ \frac{\partial E}{\partial \boldsymbol{\omega_0}} &= \frac{\partial E}{\partial \boldsymbol{h_1}} \frac{\partial \boldsymbol{h_1}}{\partial \boldsymbol{\omega_0}} \end{aligned} \tag{23}$$

Getting the partial, and writing in Jacobian form:

$$\frac{\partial E}{\partial \boldsymbol{h_1}} = \frac{\partial E}{\partial \boldsymbol{h_2}} \frac{\partial \boldsymbol{h_2}}{\partial \boldsymbol{o_1}} \frac{\partial \boldsymbol{o_1}}{\partial \boldsymbol{h_1}} \tag{24}$$

and $\boldsymbol{o_1} = g(\boldsymbol{h_1})$, $\frac{\partial \boldsymbol{h_2}}{\partial \boldsymbol{o_1}} = \boldsymbol{\omega_1}$, and $\frac{\partial \boldsymbol{o_1}}{\partial \boldsymbol{h_1}} = g'(\boldsymbol{h_1})$.

Thus, we have inferred an important formula. In the backward calculation, **the derivative of the loss function corresponds to the derivative of the loss of the neighbor layer**, and eventually this relationship will be inferred to relate to the loss of the output layer (e.g., $\frac{\partial E}{\partial h_y} = (o_y - y_{label}) \cdot g'(h_y)$ in this example in Fig. 2b).. Therefore, the partial derivative of the loss function with respect to the output layer in the previous step ($\frac{\partial E}{\partial \boldsymbol{h_k}}$) can be saved and passed on from layer to layer. The code implementation is shown in Appendix.6.

III. Simple Examine on Logistic Regression and Neuron Networks

A. Evaluation Criteria

In the following article, the experimenter's evaluation of the model mainly has the following criteria.

The Recall is calculated using the formula:

$$\text{Recall(R)} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{Precision(P)} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Accuracy(ACC)} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\text{F}_\alpha = \frac{(\alpha^2 + 1)\text{P} \cdot \text{R}}{\alpha^2(\text{P} + \text{R})}$$

Choose $\alpha = 1$ as the coefficient, the experimenter decided to use F_1 as the judging score.

$$\text{F}_1 = \frac{\text{P} \cdot \text{R}}{\text{P} + \text{R}}$$

TP, FN, FP, TN respectively means:

TABLE I: Evaluation Criteria of Models

Real Situation	Prediction	
	Positive	***Negative***
Positive	TP	FN
Negative	FP	TN

In conjunction with the loss function, these performance metrics will be employed for the evaluation of the model.

B. Basic Examination on Logistic Regression

The logistic regression is a basic model for binary prediction. In this part, the loss function is considered as the cross entropy:

$$loss = -[ylog(y_{pred}) + (1 - y)log(1 - y_{pred})]$$

Consider simple training data generating by `skylearn`, generating a dataset containing 60 samples, split it into training data and test data.

Basic Criteria:

```
Recall = 0.857
Precision = 1.00
F1 Score = 0.923
Accuracy = 0.917
Final loss = 0.155
```

With input learning rate of 0.01 and epoches of 2000, and the decision boundary is shown in Fig.4.

Hyperparameters have a profound impact on machine learning. [6] The researcher examined the effect of choice of learning rate and number of iterations on the logistic regression. TABLE.II and Fig. 5 show the result of different learning rate, and TABLE.III and Fig. 6 show the result of different epochs. The size of dataset is 600, and split into train data and test data randomly by skylearn.

The results indicate that the choice of the learning rate is crucial, as opting for an excessively large learning rate can easily result in overshooting or divergence, **impeding convergence** toward optimal solutions. Conversely, a learning

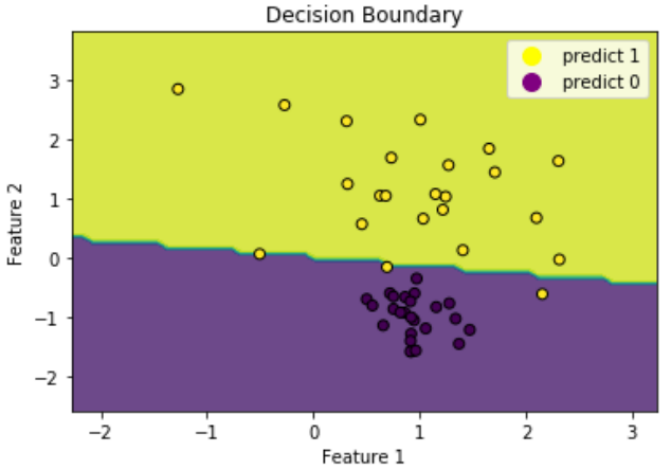


Fig. 4: Loss function of logistic regression.

TABLE II: Results of Different Learning Rate

Learning Rate	R	P	F1	ACC	Loss
0.001	0.932	0.821	0.873	0.867	0.442
0.01	0.966	0.950	0.958	0.958	0.110
1	0.949	0.949	0.949	0.950	0.094
50	0.864	0.962	0.910	0.917	0.234

[a]'Loss' represents the final loss, epochs is 200

TABLE III: Results of Different Epochs

Epochs	R	P	F1	ACC	Loss
100	0.949	0.903	0.926	0.925	0.532
500	0.949	0.933	0.941	0.942	0.343
2000	0.966	0.934	0.950	0.950	0.218
10000	0.966	0.950	0.958	0.958	0.129

[a]'Loss' represents the final loss, learning rate is 0.01

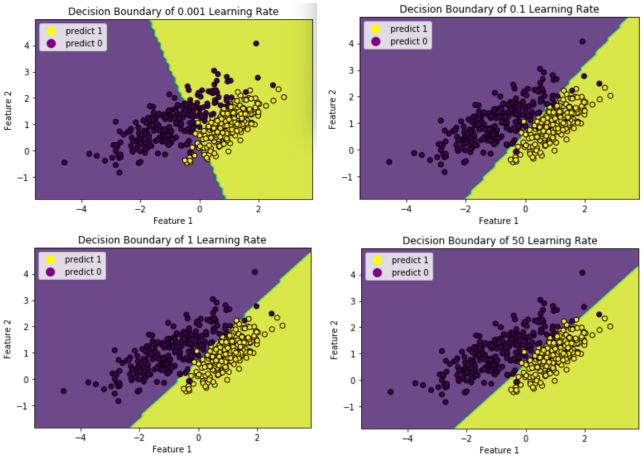


Fig. 5: Visualize results of different learning rates.

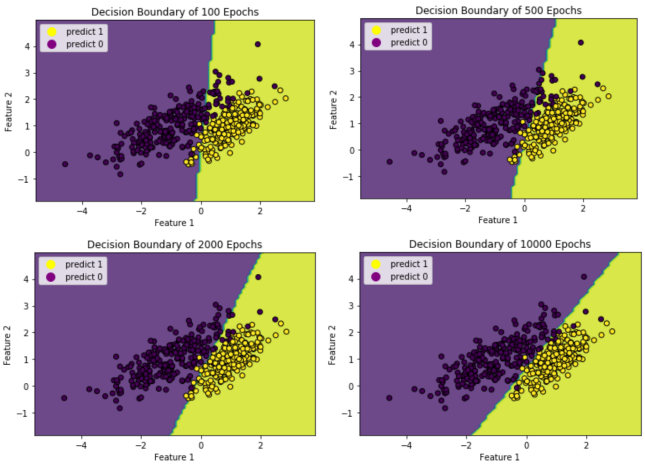


Fig. 6: Visualize results of different epochs.

rate that is too small increases the risk of converging slowly and getting **stuck in local optima**, leading to sub-optimal learning outcomes.

Simultaneously, prolonged iterations generally contribute to an improved learning effect. This is attributed to the continual reduction of the loss function, validating the hypothesis that an extended training duration enhances the model's capacity to refine its predictive performance. This observation is expected to be substantiated in subsequent experiments.

C. Basic Examination on MLPs

Within the same hidden layer, the variation in the number of neurons significantly impacts the learning efficacy. The experimenter assessed the learning performance of the Multilayer Perceptron (MLP) with a single hidden layer, employing a learning rate of 0.01 and conducting 8000 iterations. Neuron quantities of 3, 6, 10, and 20 in the hidden layer were systematically evaluated for their influence on the learning process.

```
3 Neurons:          6 Neurons:
Recall = 0.888      Recall = 0.912
Precision = 0.836   Precision = 0.912
F1 Score = 0.861    Precision = 0.912
Accuracy = 0.806    Accuracy = 0.880

10 Neurons:         20 Neurons:
Recall = 0.929      Recall = 0.976
Precision = 0.932   Precision = 0.962
F1 Score = 0.931    F1 Score = 0.969
Accuracy = 0.906    Accuracy = 0.958
```

Fig. 7 and Fig. 8 show the result.

In simpler terms, under normal conditions without issues such as gradient vanishing or exploding, increasing the number of neurons in the same hidden layer tends to improve the learning effectiveness.

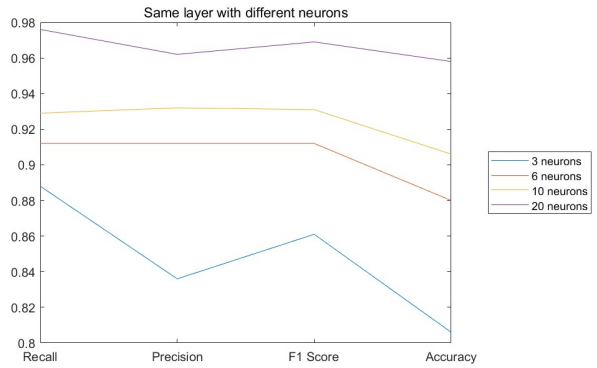
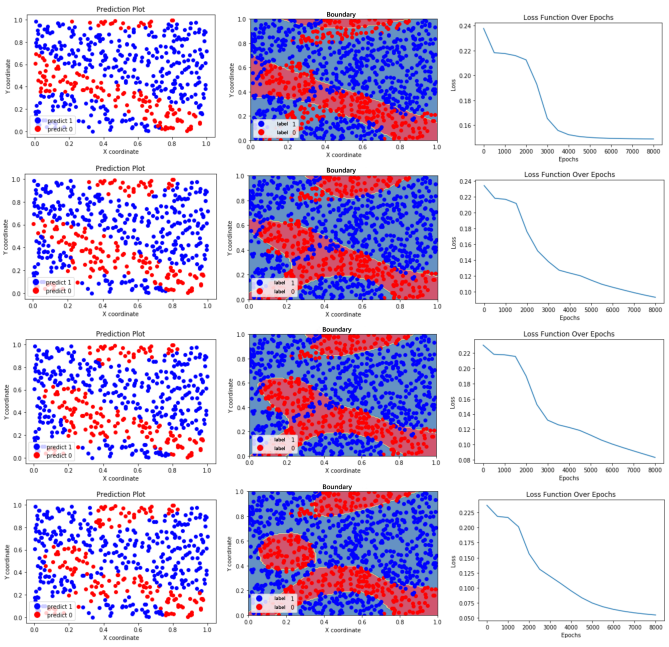


Fig. 7: Same layer with different neurons



Prediction Decision boundary Loss

Fig. 8: Same layers, different neurons

The number of neurons in a single hidden layer is 3, 6, 10, 20 in each row from top to bottom.

IV. Discussion on Question Occurred

A. Loss Function of Steady State

The loss function and iteration number of the normal working MLP basically show a decreasing trend, and tend to converge with the increase of the iteration epochs. The experimenter used the training data to examine a MLP with 5 hidden layers, and got a bad result:

```
Recall = 0.00
Precision = 0.00
F1 Score = 0.00
Accuracy = 0.606
Final loss = 0.238
```

With inputs:

TABLE IV: Parameters of the Experiment

Learning Rate	Epochs	Hidden Layers
0.01	8000	[3,5,11,2,4]

Notice the final loss is that huge, vanishing gradient might happen.

The values of the loss function are stored after every 400 iterations, and visualization is:

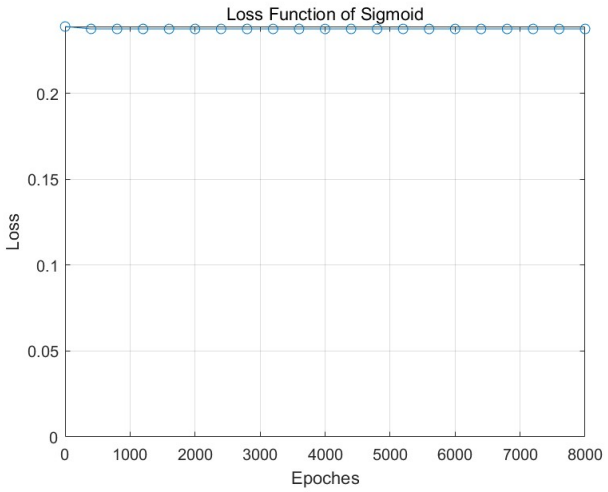


Fig. 9: Loss function change curve for this experiment.

The loss function is almost unchanged in this experiment.

Previous studies tell that the vanishing gradient will cause the loss function to remain basically unchanged, resulting in poor learning effect. And one cause for this has to do with the activation function. [2]

The activation function used in this experiment is called the sigmoid function $g(x) = \frac{1}{1+e^x}$, whose derivative is:

$$\frac{d}{dx}g(x) = \frac{e^{-x}}{(1+e^{-x})^2} \tag{25}$$

And second order derivative :

$$\frac{d^2}{dx^2}g(x) = \frac{-e^{-x}(1-e^{-x})}{(1+e^{-x})^3} \tag{26}$$

According to the mathematical analyse, the increasing and decreasing properties of the derivative of the sigmoid function are shown in TABLE V.

TABLE V: Monotonicity of the Derivative of Sigmoid

	x<0	x=0	x>0
g'(x)	increasing	maximum=0.25	decreasing
g"(x)	>0	=0	<0

Visualization of $g'(x)$ shown in Fig. 10.

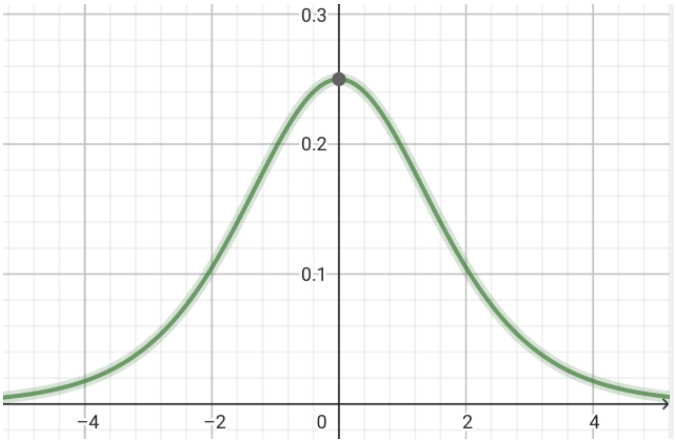


Fig. 10: Loss function change curve for this experiment.

This is to say, any $x \in \mathbb{R}$, satisfy $g'(x) \leq 0.25$. Recall 24, as the forward computation of neurons progresses, each update to $\frac{\partial E}{\partial \boldsymbol{h_k}}$ undergoes successive multiplications by a factor no greater than 0.25. This leads to the final learning gradient diminishing to exceedingly small values [2], even potentially descending below the precision limit of a computer's floating-point calculations.

Indeed, sigmoid functions have been demonstrated to be unsuitable for deep neural networks due to the tendency of upper hidden layers to saturate. [7]

Consider another function which might perform better. The Rectified Linear Units function [1] - `ReLU` is another activation function:

$$ReLU(z) = max(z : 0) \tag{27}$$

This means, when the input value exceeds 0, the derivative function consistently evaluates to 1; conversely, when the input value is less than or equal to 0, the derivative reaches a value of 0.

The activation function $g(x) = ReLU(x)$, and note that in instances where the input surpasses 0, the phenomenon of gradient vanishing is ameliorated:

```
Recall = 0.995
Precision = 0.942
F1 Score = 0.968
Accuracy = 0.974
Final loss = 0.00438
```

The experimental results indicate that in contrast to the outcomes observed in the preceding experiment utilizing the sigmoid activation function, the adoption of the Rectified Linear Unit (ReLU) activation function successfully addresses the issue of vanishing gradients.

B. Initial State to Training Result

In the experiment, the researcher found that the setting of the initial values also had an impact on the experimental results.

The experimental results obtained by setting the initialized weight coefficients to random numbers in the range of

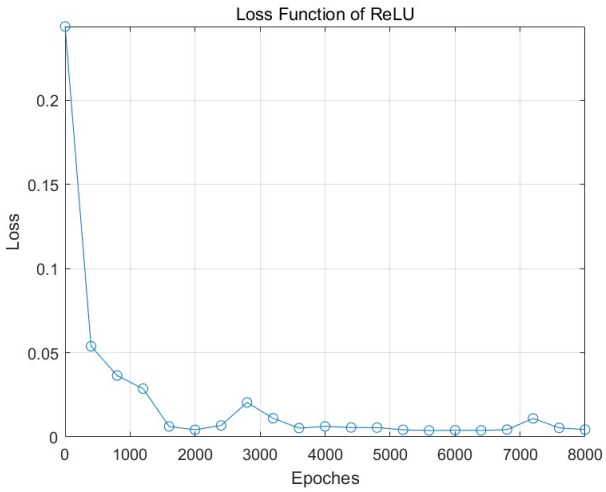


Fig. 11: Loss function variation using ReLU function.

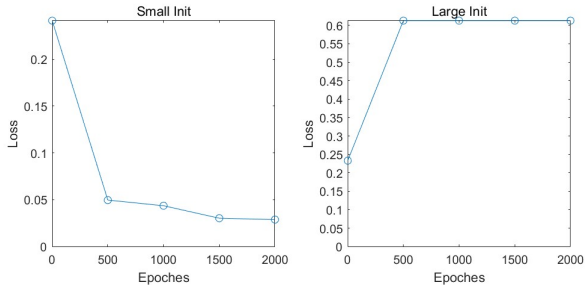


Fig. 12: Loss change with different inital values.

0 1(Small Init) and 100 200(Large Init), respectively, are quite different. Fig. 12 shows that the former achieves a reasonably good learning effect, while the latter even shows a gradient explosion.

```
Small Init:          Large Init:
Recall = 0.955       Recall =  1.00
Precision = 0.928    Precision = 0.410
F1 Score = 0.941     F1 Score = 0.582
Accuracy = 0.958     Accuracy = 0.410
```

Recalling 23, if the initial weights ω are set to large values, as the forward recursion proceeding, the $\frac{\partial E}{\partial \boldsymbol{h_k}}$ in the perceding layer will be increased hugely, potentially resulting in gradient explosion. Hence, the selection of an appropriate initial value for learning is of vital importance.

V. Advanced Discussion

A. Noise Training Data

The researcher put 3 noise data in the training set, to examine the learning result of the logistic regression, shown in Fig.13.

Consider 2 activation functions, sigmoid function and ReLU function, accessing 2 test results using the same test data:

```
Regression Model with Sigmoid:
```

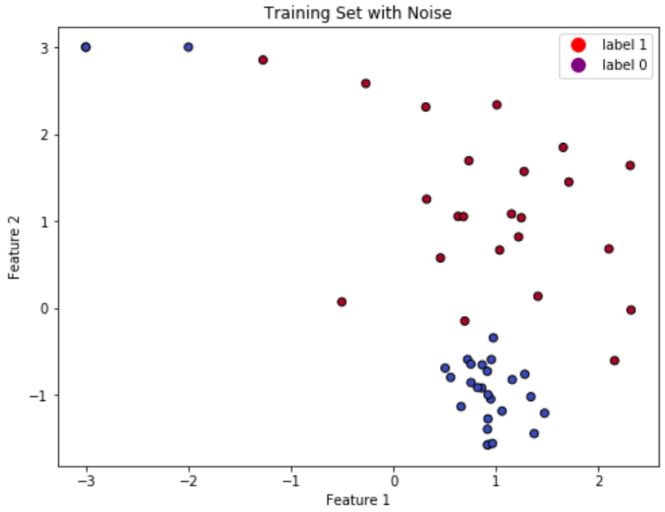
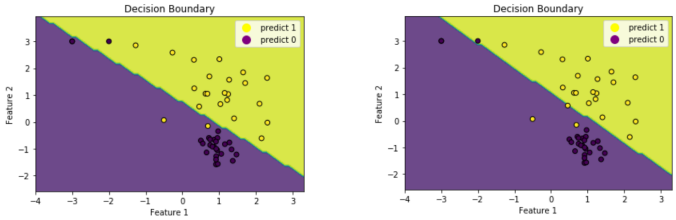


Fig. 13: Dataset with noise.

```
Recall = 0.428
Precision = 1.00
F1 Score = 0.600
Accuracy = 0.667
Final loss = 0.294

Regression Model with ReLU:
Recall = 0.714
Precision = 1.00
F1 Score = 0.833
Accuracy = 0.833
Final loss = 0.271
```

The decision boundaries are shown in Fig. 14a and Fig. 14b together with the training set.



(a) Decision boundary using sigmoid.

(b) Decision boundary using ReLU.

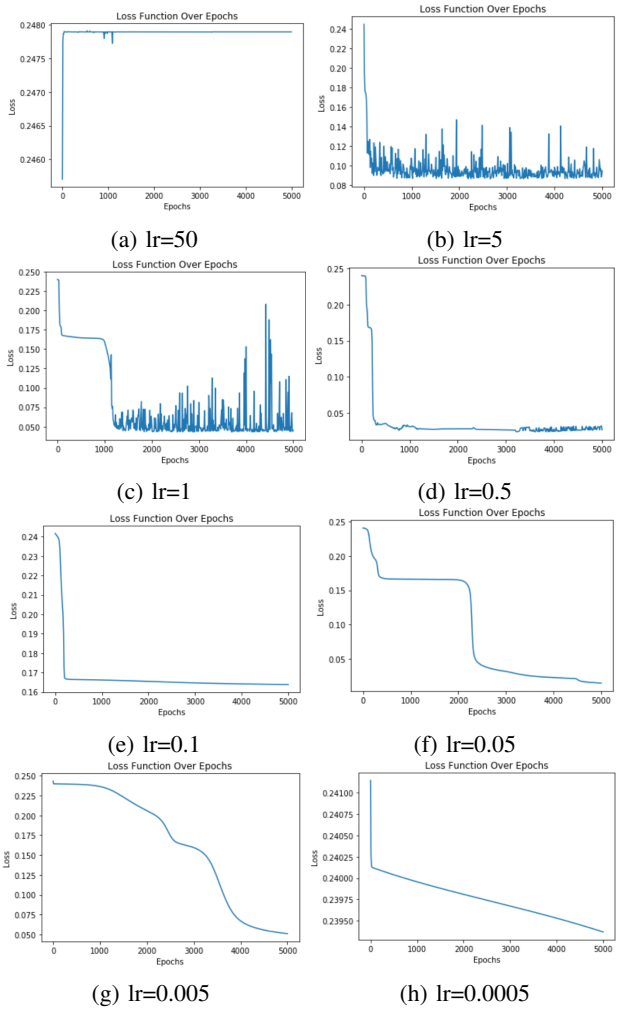
Fig. 14: Decision results using different activation functions.

It is noteworthy that the model employing the sigmoid function exhibits superior predictive performance compared to the model utilizing the Rectified Linear Unit (ReLU). This discrepancy can be attributed to **the distinct impact of their respective activation function derivatives** when the same training data is subjected to reverse computation. Specifically, the derivative function of the sigmoid function (as expressed in **Eq.25**) indicates a gradual decrease as the training data deviates further from the center. In contrast, the derivative function of ReLU maintains a constant value of 1. Consequently, the sigmoid function demonstrates a more robust learning effect in the presence of noisy data due to its adaptive response to deviations in training data.

B. Learning Rate in MLP

The learning rate stands out as a crucial hyperparameter in machine learning, exerting a significant influence on the efficacy of Multilayer Perceptron (MLP) training. An excessively large learning rate is prone to inducing oscillations in the model's proximity to the optimal solution(Fig. 15b, Fig. 15c), and in extreme scenarios, it may instigate model divergence, marked by gradient explosion(Fig. 15a). Conversely, an excessively small learning rate can result in sluggish convergence, potentially trapping the optimization process in a local optimum solution, thus impeding the learning effectiveness and contributing to the issue of vanishing gradients(Fig. 15h).

The researcher selected a three-layer neural network with hidden layers comprising 3 neurons and 7 neurons, and 2 inputs. Through 5000 iterations, the assessment of the loss function was conducted across different learning rates, and the outcomes are illustrated in Fig.15



(a) lr=50

(b) lr=5

(c) lr=1

(d) lr=0.5

(e) lr=0.1

(f) lr=0.05

(g) lr=0.005

(h) lr=0.0005

Fig. 15: Loss values with different learning rates(lr).

This is to say, choosing an appropriate learning rate is, therefore, pivotal in achieving a balance between convergence speed and stability during the training of MLP.

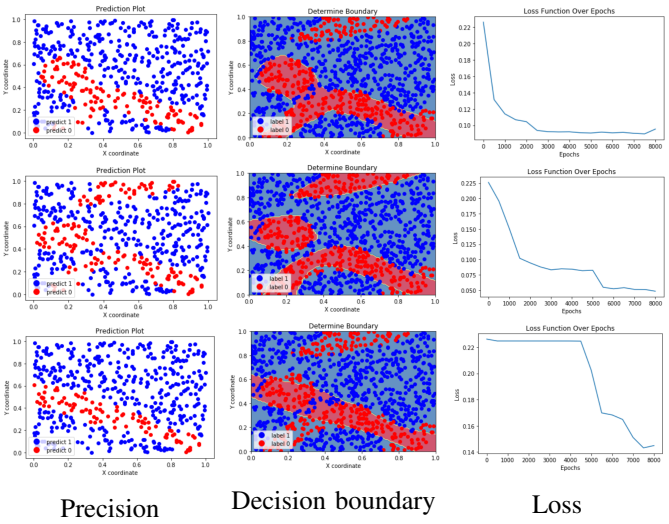
C. Complexity of Neural Layers

The architecture of the multilayer perceptron (MLP) has influences on the training outcomes.The researcher investigated the impact of MLP training across various configurations of hidden layers. Results shown in Fig. 16, Fig. 17.

The parameters are as follows in TABLE.VI.

TABLE VI: Parameters of Different Hidden Layers

Hidden Layer Size	Parameters
2	[4, 3]
3	[4, 3, 4]
6	[4, 3, 4, 7, 2, 3]



Precision Decision boundary Loss

Fig. 16: Different layers

The number of hidden layers in the MLP is 2, 3, 6 in each row from top to bottom.

```
2 hidden layers:
Recall: 0.935
Precision: 0.839
F1 Score: 0.884
ACC: 0.834

3 hidden layers:
Recall: 0.965
Precision: 0.932
F1 Score: 0.948
ACC: 0.928

6 hidden layers:
Recall: 0.953
Precision: 0.794
F1 Score: 0.866
ACC: 0.800
```

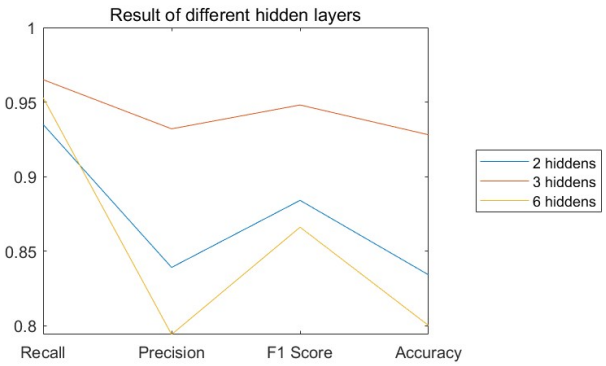


Fig. 17: Comparison of training effects of different number of hidden layers

D. Discussion on Gradient Problems

Originally, the experimenter chose the sigmoid function as the activation function in the neural network structure, with excellent training results achieving in simple neural network models. However, with the increase of the complexity of neural networks, the effect of training has changed somewhat.

The main cause of this phenomenon is that, the update of the gradients is decreasing when calculating backward, according to the inverse calculating formula. This is called the **gradient vanishing**,

In fact, there exist an improved function of ReLU - Leaky Rectified Linear Unit (LReLU) [1] function(Eq.28), which performs better in deeper networks of MLP. Usually, the α is considered much less than 1, like 0.01.

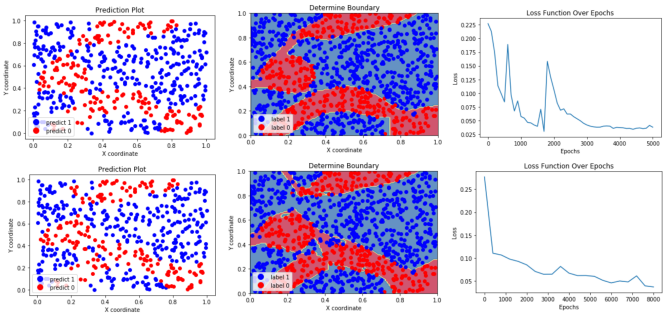
$$LReLU(z) = max(z : \alpha z) \tag{28}$$

Consider LReLU($\alpha = 0.01$) and ReLU functions, training a neural network with 8000 iterations and 11 hidden layers (parameters: [6, 19, 12, 2, 8, 7, 3, 10, 7, 10, 7]) using a learning rate of 0.0001 yields the results depicted in the Fig. 18 and Fig. 19.

```
Using LReLU:
Recall: 0.944
Precision: 0.982
F1 Score: 0.962
ACC: 0.950
```

```
Using ReLU:
Recall: 0.932
Precision: 0.966
F1 Score: 0.949
ACC: 0.932
```

It's noteworthy that the Multilayer Perceptron (MLP) with Leaky Rectified Linear Unit (LReLU) as the activation function exhibits a marginally superior learning performance compared to the neural network employing Rectified Linear Unit (ReLU) as the activation function. This distinction is likely attributed to the fact that the derivative of ReLU is defined as 0 when the input is less than 0. In contrast, LReLU introduces a small bias, which can be advantageous in the backward calculation process.



Prediction Decision boundary Loss

Fig. 18: Different layers

The first row stands for the result of using LReLU, and the second row stands for the result using ReLU.

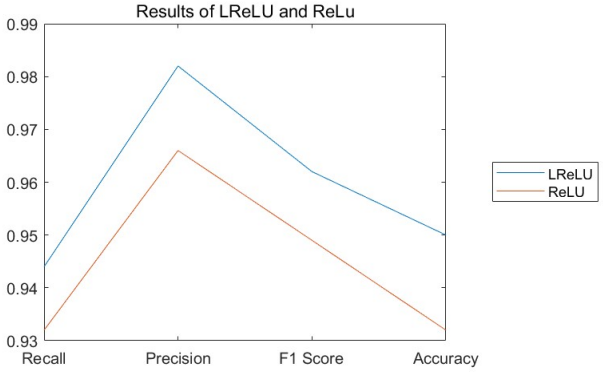


Fig. 19: Training effects of using LReLU and ReLU

E. A Trial of Using Batch Normalization

Recall Section.IV-A, the issue of vanishing gradients manifests in the MLP when utilizing the sigmoid function. This phenomenon arises due to the cumulative multiplication of the derivative of the sigmoid function during backward calculation. As this process unfolds through the layers of the MLP, the outputs become exceedingly small, leading to the vanishing of gradients.

If we can normalize each layer before calculating, the vanishing of gradients might be solve.

Batch normalization [3] is employed as a remedy for the vanishing gradient issue in the MLP. This technique helps mitigate the problem by normalizing the inputs within each mini-batch during training, thereby reducing internal covariate shift. By normalizing the inputs, batch normalization aims to maintain a stable distribution of activations throughout the network, preventing the vanishing gradient problem and enhancing the overall training stability and efficiency of the MLP.

The basic process for batch normalization is, assuming input variable X, size of the input is N:

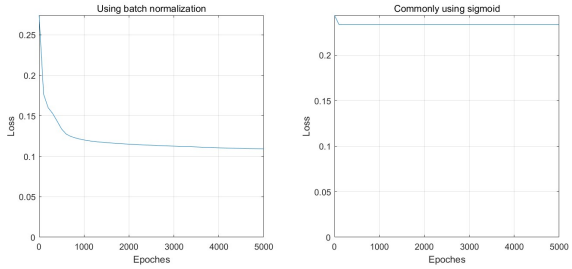


Fig. 20: Loss changes of using batch normalization or not.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} X_i$$
$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(X_i - \mu)^2 \tag{29}$$
$$\bar{X}_i = \frac{X_i - \mu}{\sigma}$$

For testing, the researcher use an MLP with 4 hidden layers, and test it. One test for using traditional sigmoid function without batch normalization, and another one use batch normalization to test. And inplementation function is shown in Appendix.7.

TABLE VII: Parameters of the Normalizatino Experiment

Learning Rate	Epochs	Hidden Layers
0.001	8000	[3,4,5,6]

Results:

```
Using normalization      No normalization
Recall = 0.772           Recall = 0.000
Precision = 0.788        Precision = 0.000
F1 Score = 0.779         F1 Score = 0.000
Accuracy = 0.828         Accuracy = 0.606
```

And the change of the loss function is shown in Fig. 20. This implies, batch normalization can be used to solve the problem of vanishing gradients to some extent. Regrettably, constrained by time limitations and the researcher's capabilities, it is unfortunate that advanced batch normalization techniques have not been subjected to testing in this study, shown in Eq.30.

$$\bar{X}_i = \gamma \cdot \frac{X_i - \mu}{\sigma} + \beta \tag{30}$$

Other methods provided for solving problems in gradients.

- Residual Networks (ResNets): ResNets introduce the concept of residual learning. Instead of learning the desired output directly, the network learns the residual (the difference between the current state and the target). This residual is then added to the input, allowing the network to learn to adjust the identity mapping. [4]

- Gradient Clipping [5]: Another prevalent issue is gradient explosion, often stemming from excessively large initialization weights during backward calculation, inappropriate activation functions, and misjudged learning rates. In addition to adjusting the parameters mentioned above, gradient clipping emerges as a valuable solution to address this problem. With gradient clipping, during gradient computation, if the calculated gradient surpasses a predetermined threshold, it is forcibly truncated. This proactive measure proves effective in mitigating the challenges associated with exploding gradients.

VI. Conclusion and Prospect

In this experiment, the experimenter successfully constructed and tested logistic regression and Multilayer Perceptron (MLP), meticulously adjusting hyperparameters and garnering a series of experimental results. The experiments underscored the impact of different learning rates and iterations on learning outcomes, elucidated the influence of distinct activation functions on gradient calculations, and explored the relationship between neural network complexity and MLP learning results. As a final endeavor, batch standardization was introduced to mitigate the issue of gradient disappearance in sigmoid function training within complex neural networks.

However, the limitations of the experiment include the insufficient number of repeated trials and the consequent challenge in ensuring the reliability of experimental conclusions. The authors did not extensively explore the effects of initial values on gradient calculations, and the treatment of noisy data received limited attention.

For future investigations, a more in-depth exploration could be conducted in the optimization selection of activation functions, adaptive adjustment of learning rates, and detailed examination of noisy data processing and batch processing. Such endeavors would contribute to a richer understanding of the nuances in training neural networks and further refine the techniques employed in machine learning experiments.

Details of this experiment (including dependencies and usage of `cpp` files), see `Readme.md` in the same file folder, or visit MLP Realization to check the update of the author.

Acknowledgment

Completing a course project on multilayer perceptrons poses a significant challenge for authors with limited theoretical foundations in artificial intelligence. Therefore, the author expresses sincere gratitude to Professor Lin for his comprehensive teaching in the Fundamentals of Artificial Intelligence and Machine Learning course. Special thanks are extended to the teaching assistant for providing timely and effective responses (with a hopeful suggestion for a salary increase for them), facilitating improved debugging of the project.

Furthermore, acknowledgment is extended to Fan Site (github.com/GuTaoZi) from the Department of Computer Science and Engineering for proposing the idea of BP calculation through Jacobian matrix. Appreciation is also expressed to Li Xiaoyang and Zhang Zhiyi from the School of System Design and Intelligent Manufacturing, Wu Yanchen from the Department of Mechanical and Energy Engineering for suggestion of generating training data and testing methods.

The author would like to thank Lei Zeyu from the Department of Computer Science, Sichuan University, for guiding the structure of the author's course report. Gratitude is extended to all departments and individuals who provided assistance throughout the project completion process. The project's success is attributed to the collaborative efforts of these entities, and without their support, the project might have faced considerable challenges.

References

[1] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in Proceedings of the 27th international conference on machine learning (ICML-10), 2010, pp. 807-814.

[2] Y. Bengio, P. Simard and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," in IEEE Transactions on Neural Networks, vol. 5, no. 2, pp. 157-166, March 1994, doi: 10.1109/72.279181.

[3] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in International conference on machine learning, 2015: pmlr, pp. 448-456.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770-778.

[5] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in International conference on machine learning, 2013: Pmlr, pp. 1310-1318.

[6] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in Neural Networks: Tricks of the Trade: Second Edition: Springer, 2012, pp. 437-478.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in Proceedings of the thirteenth international conference on artificial intelligence and statistics, 2010: JMLR Workshop and Conference Proceedings, pp. 249-256.

Appendix

Here is some additional information in the appendix.

A. Sample Codes in Python

Below are the primary functions and classes designed for the implementation of the Logistic Regression and Multilayer Perceptron (MLP) using the python programming language.

Listing 1: Class LogisticRegression

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations
        =1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
        self.final_loss =0.0

    # activition function - sigmoid
    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    # deprecated
    def d_sigmoid(self, z):
        sig = 1 / (1 + np.exp(-z))
        return sig * (1 - sig)

    def train(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
```

```
        for i in range(self.n_iterations):
            model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(model)
            derivative = self.d_sigmoid(model)
            # derivative = 1

            # dw = (1 / n_samples) * np.dot(X.T, (
                y_predicted - y)*derivative)
            # db = (1 / n_samples) * np.sum((y_predicted -
                y)*derivative)

            dw = (1 / n_samples) * np.dot(X.T, (y_predicted
                 - y))
            db = (1 / n_samples) * np.sum((y_predicted - y)
                )

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db
        # get loss
        self.final_loss = self.compute_loss(y, y_predicted)


    def predict(self, X):
        model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(model)
        y_predicted_cls = [1 if i > 0.5 else 0 for i in
             y_predicted]
        return np.array(y_predicted_cls)

    # binary classification, we use cross entrofy
    def compute_loss(self, y_true, y_predicted):
        # y_predicted = np.clip(y_predicted, 1e-10, 1-1e
            -10)  # avoid log(0)
        m = y_true.shape[0]
        return -1/m * np.sum(y_true * np.log(y_predicted) +
             (1 - y_true) * np.log(1 - y_predicted))
```

Listing 2: Class LogisticRegression with ReLU

```
class LogisticRegressionWithReLU(LogisticRegression):
    def __init__(self, learning_rate=0.01, n_iterations
        =1000):
        super().__init__(learning_rate, n_iterations)

    def sigmoid(self, z):
        # ReLU function
        return np.maximum(0, z)

    def d_sigmoid(self, z):
        return (z > 0).astype(float)

    def compute_loss(self, y_true, y_predicted):
        y_predicted = np.clip(y_predicted, 1e-10, 1-1e-10)
             # avoid log(0)
        m = y_true.shape[0]
        return -1/m * np.sum(y_true * np.log(y_predicted) +
             (1 - y_true) * np.log(1 - y_predicted))
```

Listing 3: Generate Train Data from skylearn

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import recall_score, precision_score,
    f1_score, accuracy_score
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# use sklearn generate a training set.
X, y = make_classification(n_samples=60, n_features=2,
    n_redundant=0, n_clusters_per_class=1, random_state=42)

new_data_points_X = np.array([[-3, 3], [-3, 3],[-2,3]])
new_data_points_y = np.array([0, 0,0])

X_train = np.concatenate((X_train, new_data_points_X), axis
    =0)
y_train = np.concatenate((y_train, new_data_points_y), axis
    =0)
```

Listing 4: Logic for Generating test data

```
def getOrCondition(x, y):
    # n1 = np.logical_or(y>x+0.6, y-0.8<0.5*((x-0.3)**2))
    n2 = np.logical_and((x-0.2)**2+(y-0.5)**2>0.15**2, np.
        logical_and(y-0.4>-2*((x-0.5)**2), np.logical_and(y
        -0.8 < 0.5*(x-0.3)**2,y>np.sin(x-0.7))))
    n3 = np.logical_and(y-0.2<-2*((x-0.5)**2), y>np.sin(x
        -0.7))
    return np.logical_or(y>x+0.6, np.logical_or(n2,n3))
```

Listing 5: Encapsulation of MLP in Python

```
class MyMPL:
    def __init__(self,epoches = 5000,lr = 0.03,inputSize =
        2,hidden_layer_sizes=[3]):
        self.mn = cppyy.gbl.PY_MPL()
        hlv=cppyy.gbl.std.vector[int](hidden_layer_sizes)
        self.mn.initMPL(epoches,lr,inputSize,hlv)

    def reset(self, epoches = 5000,lr = 0.03,inputSize = 2,
         hidden_layer_sizes=[3]):
        hlv=cppyy.gbl.std.vector[int](hidden_layer_sizes)
        self.mn.initMPL(epoches,lr,inputSize,hlv)

    def train(self, data, labels):
        # judge thether labels are NumPy array
        if isinstance(data, np.ndarray):
            data_list = data.tolist()
        else:
            data_list = data

        if isinstance(labels, np.ndarray):
            labels_list = labels.tolist()
        else:
            labels_list = labels

        # declare std::vector in python
        vector_vector_double = cppyy.gbl.std.vector[cppyy.
            gbl.std.vector['double']]
        vector_double = cppyy.gbl.std.vector['double']

        # transverse inputs
        inputs = vector_vector_double()
        for sample in data_list:
            sample_vector = vector_double()
            for feature in sample:
                sample_vector.push_back(feature)
            inputs.push_back(sample_vector)

        # transverse labels
        labels_vector = vector_double()
        for label in labels_list:
            labels_vector.push_back(label)

        self.mn.train(inputs, labels_vector)

    def predict(self, data):
        if isinstance(data, np.ndarray):
            test_data_list = data.tolist()
        else:
            test_data_list = data
        vector_double = cppyy.gbl.std.vector['double']
        test_input = vector_double()
        for sample in test_data_list:
            test_input.push_back(sample)
        return self.mn.predict(test_input)

    def setLR_VOKE(self, LR_VOKE):
        self.mn.setLR_VOKE(LR_VOKE)

    def getLosses(self):
        losses = cppyy.gbl.std.vector['double']()
        losses = self.mn.getLosses()
        losses_py = [losses[i] for i in range(len(losses))]
        return losses_py
```

B. Sample Codes in Cpp

Below are the primary functions designed for the implementation of the Multilayer Perceptron (MLP) using the C++ programming language.

Listing 6: Implementation of backward calculation

```
/*
typedef std::vector<double> my_vector
*/
void MyNeuron::backward(my_vector&data, double y_label){
    my_vector&output_o = forward(data);
    my_vector&output_h = h[h.size()-1];
    double y_pred = output_o[0];

    double error = y_pred - y_label;

    my_vector outputLayerGradient(1, error*d_sigmoid(
         output_h[0]));

    std::vector<my_vector> layerGradients;
        layerGradients.push_back(outputLayer);
    //inverse promoting
    for(int layerIndex = h.size()-2; layerIndex>=0; --
         layerIndex){
        my_vector layerGradient;
        //calculate the power of current neuron
        for(int neuronIndex = 0; neuronIndex<h[layerIndex].
             size; ++neuronIndex){
            double gradientSum = 0.0;
            for(int nextNeuronIndex = 0; nextNeuronIndex<h[
                 layerIndex+1]; ++nextNeuronIndex){
                gradientSum += w[layerIndex][neuronIndex][
                     nextNeuronIndex] * layerGradients[0][
                     nextNeuronIndex];//todo: After updating
                      to sequential operations, check the
                     training results.
            }
            layerGradient.push_back(gradientSum*d_sigmoid(h
                 [layerIndex][neuronIndex]));
        }
        layerGradients.insert(0, layerGradient);//Insert to
              the first position
    }

    for (int layerIndex = 0; layerIndex < w.size(); ++
         layerIndex) {
        for (int neuronIndex = 0; neuronIndex < w[
             layerIndex].size(); ++neuronIndex) {
            for (int nextNeuronIndex = 0; nextNeuronIndex <
                  w[layerIndex][neuronIndex].size(); ++
                 nextNeuronIndex) {
                w[layerIndex][neuronIndex][nextNeuronIndex]
                      -= learning * o[layerIndex][
                     neuronIndex] * layerGradients[
                     layerIndex][nextNeuronIndex];//check:
                     After updating to sequential operations
                     , check the training results for any
                     problems.
            }
        }
        for (int biasIndex = 0; biasIndex < b[layerIndex].
             size(); ++biasIndex) {
            b[layerIndex][biasIndex] -= learning *
                 layerGradients[layerIndex][biasIndex];
        }
    }
}
```

Listing 7: Implementation of Batch Normalization

```
void MyNeuron::orth(my_vector & self_h)
{
    static const double epsilon = 0.0001;
    double mu = 0.0, delta_2 = 0.0;
    int length = self_h.size();
    for (size_t i = 0; i < length; i++)
    {
        mu += self_h[i];
    }

    mu /= (double)length;
    for (double& p : self_h) {
        double val = p - mu;
        delta_2 += val*val;
    }
    delta_2 /= (double)length;

    double fenmu = (delta_2 > epsilon) ? sqrt(delta_2) :
         epsilon;
    for (size_t i = 0; i < length; ++i)
    {
        self_h[i] = (self_h[i] - mu) / fenmu;
    }
}
```