# Application of LR and MLP Models in Binary Classification Analysis

SDM274 2023 Fall Midterm Project

1st JingJun Xu
*Department of Electronic and Electrical Engineering*
*Southern University of Science and Technology*
Shenzhen, China
12210357@mail.sustech.edu.cn

***Abstract*—This project presents an implementation and verification of logistic regression and multi-layer perceptron (MLP) models, constructed using numpy. The core objective is to enhance the practical understanding of these models through simulation and application on varied binary datasets in a two-dimensional plane. The iterative process of model training, problem-solving, and summarization fosters a deeper familiarity with machine learning techniques and processes. Key strategies employed in this project include Xavier initialization for improved model initialization, the Adam optimization algorithm for parameter updates, batch normalization, and alternative activation functions to mitigate gradient vanishing. Furthermore, to address the challenge of overfitting, the project integrates L2 regularization. These methodologies collectively contribute to a more robust and efficient learning framework.**

***Index Terms*—Logistic regression, MLP, binary classification problem**

## I. Introduction

In the rapidly evolving field of machine learning, logistic regression and multi-layer perceptrons (MLPs) stand out as fundamental models for binary classification. This project aims to demystify these models by implementing them from scratch using Python and Numpy. By doing so, it provides a hands-on approach to understanding the mechanics and nuances of these algorithms, which are often obscured in higher-level machine-learning libraries.

The core objective of this project is to develop a logistic regression model and an MLP with varying configurations. This involves creating LogisticRegression and MLP classes, generating mock datasets for binary classification, and experimenting with different model architectures, including variations in the number of hidden layers and neurons. The class named LogisticRegression is in logistics_regression.py and the class named MLP is in mlp.py. The mock data is generated by a class named MockData stored in mock_data.py.

Through this work, we aim to not only build these models but also to compare their performance under different conditions. This comparison will be based on key metrics like recall, precision, and F1 score. The experiments are in the file named logistic_regression_test.ipynb, wide_mlp_test.ipynb and deep_mlp_test.ipynb. Additionally, the project will delve into the impact of hyperparameters, such as learning rate and tolerance levels, on the training process and model performance. The related experiments of hyperparameters are in the file hyperparameters_test.ipynb. By the way, some utilities functions are written in the file named utilities.py.

The report is structured to provide a comprehensive view of the project, starting with problem formulation, followed by detailed descriptions of the methods and algorithms used. Subsequent sections will present experimental results, including visual plots and tables, and conclude with an analysis of these findings. The final section will discuss potential future enhancements and applications of this work.

By the end of this project, we expect to have a deeper understanding of logistic regression and MLPs, along with insights into their practical application and performance in binary classification tasks.

## II. Problem formulation

This project aims to implement and evaluate two machine learning models: Logistic Regression and Multi-Layer Perceptron (MLP) for binary classification problems. The models will be applied to a simulated data set generated on a two-dimensional plane containing two separable classes.

Writing the model: Use Python and Numpy to write classes named LogisticRegression and MLP. The former implements the logistic regression model, and the latter implements a universal multi-layer perceptron that can control the number of hidden layers and the neurons in each hidden layer.

Test the performance of the model: train the model on the created simulation data set, display its classification results and analyze it. Especially for the multi-layer perceptron model, the width neural network and the deep neural network are trained separately, the number of neurons and the number of layers are changed, and the classification results are displayed and analyzed. By the way, use Recall, Precision and F1 score to evaluate the performance of different models for binary classification results.

Other analysis and summary: Show the changes in the loss function when changing the hyperparameters, etc.

## III. Method and algorithms

### A. *Logistic Regression Analysis*

Logistic regression is a statistical method for analyzing datasets in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes). It is often used for binary classification and predictive analytics.

The logistic regression model is formulated using the logistic function to model a binary dependent variable C. The logistic function, also known as the sigmoid function, is defined as:

$$\sigma(z) = \frac{1}{1+\exp{(-z)}} \tag{1}$$

where

$$z = \boldsymbol{w}^T\boldsymbol{x} + w_0 \tag{2}$$

This allows multiple independent variables input ($\boldsymbol{x}$ is a vector storing all the independent variables) and the parameter values ($\boldsymbol{w}$ and $w_0$) act to adjust the weight of each independent variable.

To be more specific, the probability of the default class, in this report labelled as '0', is modelled as a function of one or more independent variables. $P(C=0|\boldsymbol{x}) = \sigma(\boldsymbol{w}^T\boldsymbol{x}+w_0) = 1/(1+\exp{(-\boldsymbol{w}^T\boldsymbol{x}-w_0)})$, and $P(C=1|\boldsymbol{x}) = 1-\sigma(\boldsymbol{w}^T\boldsymbol{x}+w_0) = 1-1/(1+\exp{(-\boldsymbol{w}^T\boldsymbol{x}-w_0)})$, which indicates the decision boundary $P(C=1|\boldsymbol{x},\boldsymbol{w}) = P(C=0|\boldsymbol{x},\boldsymbol{w}) = 0.5$ (a linear decision boundary).

The goal of the model is to find the parameter values ($\boldsymbol{w}$ and $w_0$) that make the outputs of the model get as close as possible to the authentic data distribution of the dataset, which means to maximize the likelihood of observing the sample data (The Maximum Likelihood Estimation (MLE) method).

$$\underset{x_3}{max}\, L(\boldsymbol{w}) = \underset{x_3}{max} \prod_{i=1}^{N} P(t^{(i)|\boldsymbol{x}^{(i)},\boldsymbol{w}}) \tag{3}$$

Converting the maximization problem into minimization and converting the multiplication of probabilities into addition through the log function, the loss function is defined as:

$$\begin{aligned} l_{log}(\boldsymbol{w}) &= -\sum_{i=1}^{N} log(P(t^{(i)|\boldsymbol{x}^{(i)},\boldsymbol{w}})) \\ &= -\sum_{i=1}^{N} t^{(i)} log(1-P(C=0|\boldsymbol{x}^{(i)},\boldsymbol{w})) \\ &- \sum_{i=1}^{N}(1-t^{(i)}) log(P(C=0|\boldsymbol{x}^{(i),\boldsymbol{w}})) \\ &= \sum_{i=1}^{N} log(1+exp(-z^{(i)})) + \sum_{i=1}^{N} t^{(i)} z^{(i)} \end{aligned} \tag{4}$$

Take the derivatives of the loss function, the gradient is calculated as:

$$\frac{\partial l}{\partial w_j} = \sum_{i=1}^{N} x_j^{(i)}(t^{(i)} - P(C=1|\boldsymbol{x}^{(i)},\boldsymbol{w}) \tag{5}$$

Putting all together, the gradient descent for logistic regression is as below:

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \sum_{i=1}^{N} x_j^{(i)}(t^{(i)} - P(C=1|\boldsymbol{x}^{(i)},\boldsymbol{w}) \tag{6}$$

where $\lambda$ is the learning rate.

Based on the gradient descent formula, the model can be trained successfully. Using the loss function, the quality of the model can be estimated.

### B. *Multiple Layer Perceptron Analysis*

A Multilayer Perceptron (MLP) is a class of feedforward artificial neural network (ANN) that consists of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. It often served as a non-linear discriminative classifier.

The structure of MLP is shown in the graph below:

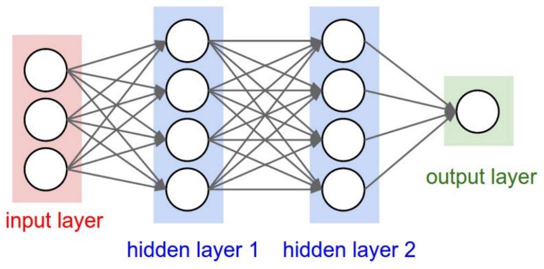

Fig. 1. Structure of MLP

Layers and layers are connected through lines. Each layer contains multiple circles, which represent neurons. The details of the neuron are shown by the graph below:

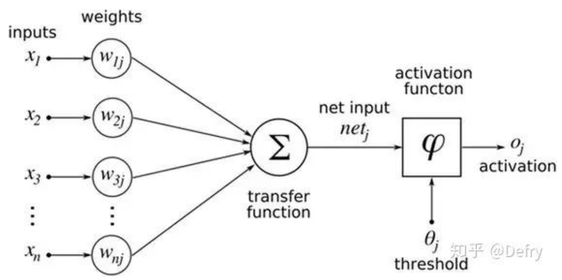

Fig. 2. details of neuron in MLP

Each neuron contains a transfer function, which implements the linear combination between the input and the weights (containing the bias). And an activation function is also a part of the neuron.

There are two algorithms of MLP, the Forward pass and the Backward pass. The Forward pass performs inference and the Backward pass performs learning.

The forward process of a Multilayer Perceptron (MLP), also known as forward propagation, involves the sequential computation of outputs from the input layer, through the hidden layers, to the output layer. This process utilizes the network's current set of weights and biases to make predictions based on the input data.

The input layer receives the feature vector $\boldsymbol{x}$. This layer essentially serves as a pass-through layer that feeds the input features into the subsequent layers. The input vector can be denoted as $\boldsymbol{x} = [x_1, x_2, ..., x_n]$,where $n$ is the number of the input features.

An MLP can have one or more hidden layers. Each neuron in a hidden layer computes a weighted sum of its inputs, adds a bias, and then applies an activation function. The computation in the $j$-th neuron of the $i$-th hidden layer can be expressed as:

$$a_j^{(i)} = f(\sum_{k=1}^{N_{i-1}} w_{jk}^{(i)} a_k^(i-1) + b_j^(i)) \tag{6}$$

Here, $a_j^{(i)}$ is the activation of the $j$-th neuron in the $i$-th hidden layer, $f$ is the activation function, $w_{jk}^{(i)}$ represents the weight from the $k$-th neuron in the $(i-1)$-th layer to the $j$-th neuron in the $i$-th layer, $a_k^{(i-1)}$ is the activation of the $k$-th neuron in the $(i-1)$-th layer, and $N_{i-1}$ is the number of neurons in the $(i-1)$-th layer.

The activation function introduces non-linearity into the model, enabling the network to model complex relationships. The most commonly used activation functions are as below:

$$Sigmoid : \sigma(z) = \frac{1}{1 + \exp(-z)} \tag{6}$$

$$Tanh : tanh(z) = \frac{exp(z) - exp(-z)}{exp(z) + exp(-z)} \tag{6}$$

$$ReLU : ReLu(z) = max(0, z) \tag{6}$$

The final layer of an MLP is the output layer. The computation in the output layer is similar to that in the hidden layers, but the function of the output layer is to produce the network's final output. For a single output neuron, the computation can be expressed as:

$$y = f(\sum_{k=1}^{N_{last}} w_k^{output} a_k^{last} + b^{output}) \tag{6}$$

where $y$ is the predicted output, $w_k^{output}$ and $b^{output}$ are are the weights and bias of the output neuron, respectively, and $a_k^{last}$ are the activations from the last hidden layer.

Moreover, the output $y$ of MLP can be a single value (for binary classification or regression tasks) or a vector (for multi-class classification tasks), depending on the design of the output layer.

Backpropagation in a Multilayer Perceptron (MLP) is a key algorithm for training neural networks, which involves the iterative adjustment of the network's weights and biases to minimize the error between the predicted output and the actual output. This process employs the chain rule of calculus to compute the gradient of a loss function with respect to each weight and bias in the network.

After the forward pass (forward propagation), where the input data is passed through the network to compute the output, the backpropagation process begins. Initially, the error between the predicted output and the actual output is calculated using a loss function. The loss function quantifies the difference between the predicted output and the actual target values. For a given training example, the loss function can be represented as $L(y, \hat{y})$, where $y$ is the actual output and $\hat{y}$ is the predicted output. The most commonly used loss functions are defined below:

$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y_i})^2 \tag{6}$$

$$BinaryCross{-}Entropy = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y_i}) + (1-y_i) \log(1-\hat{y_i})] \tag{6}$$

The core of backpropagation is to compute the gradient of the loss function with respect to each weight and bias in the network. This is done using the chain rule. For a weight $w_{jk}^{(i)}$ in the network, the partial derivative of the loss function with respect to this weight is given by:

$$\frac{\partial L}{\partial w_{jk}^{(i)}} = \frac{\partial L}{\partial a_j^{(i)}} \cdot \frac{\partial a_j^{(i)}}{\partial z_j^{(i)}} \cdot \frac{\partial z_j^{(i)}}{\partial w_{jk}^{(i)}} \tag{6}$$

where $a_j^{(i)}$ is the activation of the $j$-th neuron in the $i-th$ layer, and $z_j^{(i)}$ is the weighted sum of inputs to that neuron.

The error is propagated backwards through the network, from the output layer to the input layer. This involves computing the gradient of the loss with respect to the activations of each layer and then with respect to the weights and biases. The gradient with respect to the activation of the $j$-th neuron in the $i$-th layer is calculated as :

$$\delta_j^{(i)} = \frac{\partial L}{\partial a_j^{(i)}} \tag{6}$$

Once the gradients are computed, the weights and biases are updated in the direction that minimally decreases the loss. This is done using the gradient descent (or a variant of it) algorithm. The update rule for a weight is typically of the form:

$$w_{jk}^{(i)} = w_{jk}^{i} - \eta \cdot \frac{\partial L}{\partial w_{jk}^{(i)}} \tag{6}$$

The forward pass and backpropagation steps are repeated iteratively for multiple epochs or until a certain convergence criterion is met (such as a minimum change in loss or a maximum number of iterations).

## C. Algorithms for Xavier initialization

Random initialization parameters are highly uncertain. A weight that is too small may cause the network to collapse, and a weight that is too large may cause the network to be saturated. According to Glorot's research, a good initialization method is to use Xavier initialization. This method ensures that all neurons in the network start with approximately the same output distribution. Practical experience has proven that doing so can improve the speed of convergence.

Assume that the inner product of the weight $\boldsymbol{w}$ of the neuron and the input formula $\boldsymbol{b}$ is $z = \sum_i^n w_i x_i$. The variance of s at this time is calculated as:

$$\begin{aligned} Var(s) &= Var(\sum_i^n w_i x_i) \\ &= \sum_i^n Var(w_i x_i) \\ &= \sum_i^n ([E(w_i)]^2 Var(x_i) \\ &+ [E(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i)) \\ &= \sum_i^n Var(x_i)Var(w_i) \\ &= nVar(w)Var(x) \end{aligned} \tag{6}$$

The first three steps use the properties of variance (additivity, multiplication of independent variables). In the third step, it is assumed that the mean values of input and weight are both 0, which means $E(x_i) = E(w_i) = 0$ (except for the ReLu function, which is supposed to be positive). In the last step, it is assumed that all the $w_i$ and $x_i$ have the same date distribution. From this derivation process, we can see that if we want $s$ to have the same variance as the input $x$, then during initialization we must ensure that the variance of each weight $w$ is $\frac{1}{n}$. And because for a random variable $X$ and scalar $a$, $Var(aX) = a^2 Var(X)$. This means that $w$ can be sampled based on the standard Gaussian distribution (variance 1), and then multiplied by $a = \sqrt{1/n}$, which means $Var(\sqrt{1/n} \cdot w) = \frac{1}{n} \cdot Var(w) = \frac{1}{n}$. In this way, $Var(s) = Var(x)$ is guaranteed.

The corresponding code in Python is:

```
W = np.random.randn(n_in, n_out)
    / np.sqrt(n_in + n_out)
```

Moreover, when confronting the ReLu function, the corresponding code in Python should be modified as:(according to the Research of Kaiming He in 2015)

```
W = np.random.randn(n_in, n_out) / np.sqrt(n_in/2)
```

## D. Algorithms for Batch_normalization

Batch normalization is proposed by Loffe and Szegedy. This method solves to a certain extent the thorny problem of how to properly initialize the neural network. The method is to let the activation data pass through a network before training starts, and the network processes the data to make it obey the standard Gaussian distribution. Normalization is a simple and derivable operation, so it is feasible. At the implementation level, applying this technique usually means adding a BatchNorm layer between the fully connected layer and the activation function.

In the forward propagation, for each mini-batch, the mean and variance are computed for each feature. Let $B$ be a mini-batch of size $m$ with $d$-dimensional input data. The mean $\mu_B$ and variance $\sigma_B^2$ for each feature are calculated as:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i \tag{6}$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{6}$$

where $x_i$ is the $i$-th input in the mini-batch.

The inputs are then standardized using the calculated mean and variance:

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_b^2 + \epsilon}} \tag{6}$$

Here, $\epsilon$ is a small constant added for numerical stability (to avoid division by zero).

The standardized inputs are then linearly transformed using learnable parameters, $\gamma$ and $\beta$, which allow the network to undo the normalization if it is beneficial for learning. This transformation is given by:

$$y_i = \gamma \hat{x_i} + \beta \tag{6}$$

where $y_i$ is the output of the batch normalization layer, and $\gamma$ and $\beta$ are parameters learned during training.

The backward propagation involves the calculation and update of the gradients with respect to the batch normalization parameters and the modification of the gradient with respect to the original input.

The gradients with respect to the batch normalization parameters, are calculated as follows:

$$\frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \hat{x_i} \tag{6}$$

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i} \tag{6}$$

where $\frac{\partial L}{\partial y_i}$ is The gradient of the loss function $L$ with respect to the output of the batch normalization layer$y_i$is calculated during back propagation, $\hat{x_i}$ is the normalized input, and $m$ is the size of the mini-batch.

The gradients with respect to the original inputs $x_i$ before normalization are more complex. These gradients take into account the effect of the mean and variance used in the normalization step. The gradient is given by:

$$\begin{aligned}\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \hat{x}_i} \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{2(x_i - \mu_B)}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i} \frac{(x_i - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} \\ + \frac{1}{m} \sum_{i=1}^{m} \frac{\partial L}{\partial \hat{x}_i}\end{aligned} \tag{6}$$

Here, $\mu_B$ and $\sigma_B^2$ are the mean and variance of the inputs in the batch, and $\epsilon$ is a small constant for numerical stability.

The parameters $\gamma$ and $\beta$ are updated using gradient descent.

## *E. Algorithms for Adam*

Adam, short for Adaptive Moment Estimation, is an optimization algorithm that has been designed to adapt the learning rate for each parameter individually, based on estimates of the first and second moments of the gradients. It combines ideas from the Momentum and RMSprop algorithms and is particularly effective for problems with large datasets or many parameters.

Initially, initialize the first ($m$) and second momentum ($v$) vector to 0.

At each iteration $t$, compute the gradient $g_t$ with respect to the current parameters $\theta_t$ of the model.

$$g_t = \nabla_\theta L(\theta_t) \tag{6}$$

Here, $L(\theta_t)$ is the loss function evaluated with the current parameters $\theta_t$.

Update the biased first-moment estimate, which is a moving average of the gradients:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{6}$$

where $\beta_1$ is the exponential decay rate for the first moment estimates. Normally, $\beta_1 = 0.9$

Update the biased second raw moment estimate, which is a moving average of the squared gradients:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{6}$$

where $\beta_2$ is the exponential decay rate for the second-moment estimates. Normally, $\beta_1 = 0.99$

Compute the bias-corrected second raw moment estimate:

$$\hat{m_t} = \frac{m_t}{1 - \beta_1^t} \tag{6}$$

This correction helps counteract the initialization bias of $m$ towards 0.

Compute the bias-corrected second raw moment estimate:

$$\hat{v_t} = \frac{v_t}{1 - \beta_2^t} \tag{6}$$

This correction helps counteract the initialization bias of $v$ towards 0.

Update the parameters in the direction that reduces the loss:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}} + \epsilon} \cdot \hat{m_t} \tag{6}$$

where $\eta$ is the learning rate and $\epsilon$ is a small scalar (to prevent division by zero, typically on the order of $10^{-8}$).

Repeat the steps mentioned above for each iteration until the parameters converge, or another stopping criterion is met.

## *F. Algorithms for Regularization*

Regularization is a technique used to prevent overfitting in machine learning models by penalizing complexity, typically in the form of large weights. Regularization methods such as L1 (Lasso), L2 (Ridge), and Max-norm constraints are employed to constrain or reduce the weights during model training. In this project, L2 (Ridge) is used.

L2 regularization, also known as Ridge regularization or Tikhonov regularization, penalizes the square of the weights in the loss function. It adds a regularization term to the loss function, which is proportional to the sum of the squares of the weights:

$$L_{new}(\theta) = L(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2 \tag{6}$$

where, $L(\theta)$ is the original loss function, $\theta$ is the vector of model parameters, $\lambda$ is the regularization parameter that controls the strength of the penalty, and $n$ is the number of parameters.

The gradient of the loss function with L2 regularization term with respect to the weights becomes:

$$\nabla_\theta L_{new}(\theta) = \nabla_\theta L(\theta) + 2\lambda\theta \tag{6}$$

## G. Algorithm for evaluation of the model

In the binary classification problem of machine learning, the predicted results are generally divided into four categories. True positive(TP), true negative(TN), false negative(FN) and false positive(FP). TP and TN are correctly predicted data, FP and FN are incorrectly predicted data.

*1) Accuracy:* Accuracy represents the proportion of correctly classified samples to the total number of samples.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6}$$

*2) Recall:* Recall indicates that the prediction result is the proportion of the actual number of positive samples in the positive samples to the total number of positive samples in the total sample.

$$Recall = \frac{TP}{TP + FN} \tag{6}$$

*3) Precision:* Precision represents the proportion of samples that are actually positive among the samples that are predicted to be positive.

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

*4) F1 score:* The F1 score is a weighted average of precision and recall.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{6}$$

## IV. Experiment Results and Analysis

### A. Binary Classification using Logistic Regression model

In this experiment, the logistic regression model was trained on three data sets, namely linear data set, quadratic data set, and random data set. The results are as follows.

For the linear data:

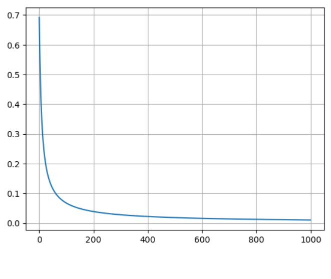

Fig. 3. loss

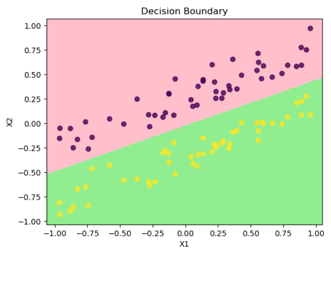

Fig. 4. decision boundary

For the quadratic date:

TABLE I
EVALUATION FOR LR IN LINEAR DATA

| | |
|---|---|
| Accuracy | 1.0 |
| Precision | 1.0 |
| Recall | 1.0 |
| F1 score | 1.0 |

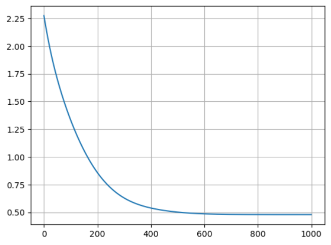

Fig. 5. loss

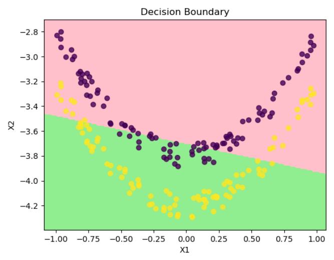

Fig. 6. decision boundary

TABLE II
EVALUATION FOR LR IN QUADRATIC DATA

| | |
|---|---|
| Accuracy | 0.7 |
| Precision | 0.727273 |
| Recall | 0.727273 |
| F1 score | 0.727273 |

For the stochastic data:

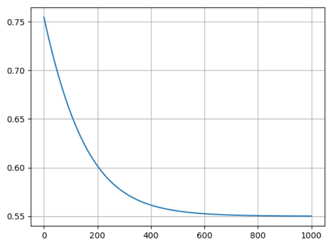

Fig. 7. loss

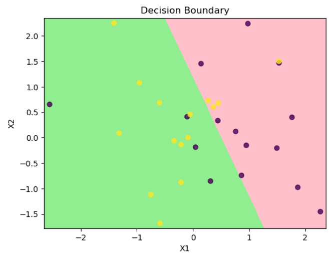

Fig. 8. decision boundary

TABLE III
EVALUATION FOR LR IN STOCHASTIC DATA

| | |
|---|---|
| Accuracy | 0.666667 |
| Precision | 0.0 |
| Recall | 0.0 |
| F1 score | 0.0 |

From the above results, it is obvious that the Logistics regression model can handle the linear classification boundary problem very well, which is in line with the derivation of the decision boundary of the model in the previous section. This model has the ability to solve quadratic curve classification problems with nonlinear decision boundaries and random point classification problems. The values of Precision, Recall and F1 score from the quadratic curve classification are much better than the results from the random point classification. This is because the linear part of the former's data classification is more than the latter.

## B. *Binary Classification using Wide MLP*

The mlp model used in this experiment only uses a single hidden layer. It aims to solve the binary classification problem. Using different numbers of neurons, the results are as follows.

For the linear data:

The optimised method is batch update. Do not use Adam, batch_normalization or regularization. And the epoch_num is 1000. The activation function in the output layer is Sigmoid and the activation function in the hidden layers is ReLu.

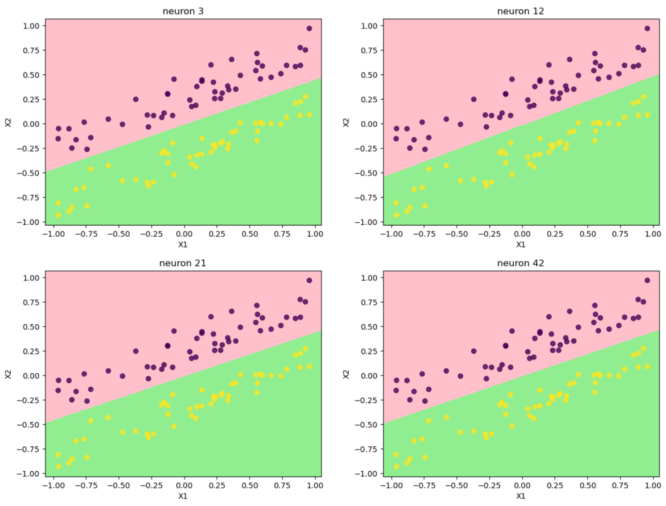

Fig. 9. Decision boundary

The picture in the upper left corner is the result of mlp with 3 neurons, the picture in the upper right corner is the result of mlp with 12 neurons, the picture in the lower left corner is the result of mlp with 21 neurons, and the picture in the lower right corner is the result of mlp with 42 neurons.

TABLE IV
WIDE MLP LINEAR DATA EVALUATION

| Neuron_num | 3 | 12 | 21 | 42 |
|---|---|---|---|---|
| Accuracy | 1.00 | 1.00 | 1.00 | 1.00 |
| Precision | 1.00 | 1.00 | 1.00 | 1.00 |
| Recall | 1.00 | 1.00 | 1.00 | 1.00 |
| F1 score | 1.00 | 1.00 | 1.00 | 1.00 |

The correctness of the mlp model can be seen from the decision boundary and various evaluation indicators, and it can easily handle problems with linear decision boundaries.

For stochastic data:

The optimised method is batch update. Do not use Adam, batch_normalization or regularization. The epoch_num of the experiment for neuron 3 and neuron 12 is 50000, the epoch_num of the experiment for neuron 21 is 10000, and the epoch_num of the experiment for neuron 42 is and The activation function in the output layer is Sigmoid and the activation function in the hidden layers is ReLu.

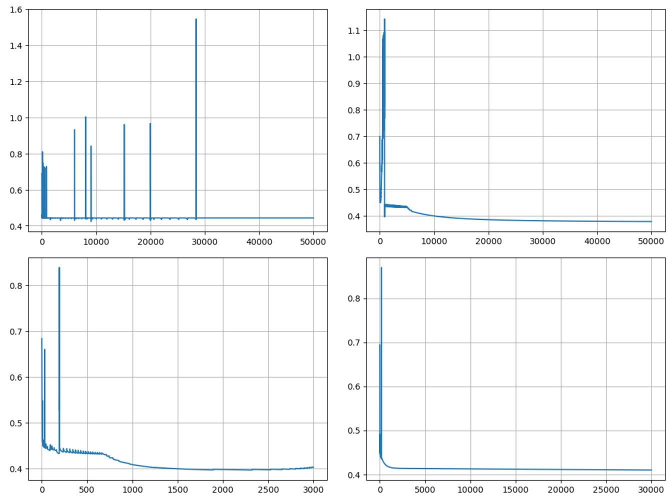

Fig. 10. Loss

It can be seen from the changing image of loss that although the overall trend of loss is declining, its rate of decline is prolonged, that is, the phenomenon of gradient disappearance occurs. This project attempts to solve this problem through batch_normalization, Adam optimization method and the method of replacing the activation function, because the gradient is prone to explosion after using Adam, that is, the value of loss will become very large, and trying to adjust the regularization intensity still has no good results. result. Using batch_normalization will cause the phenomenon of gradient disappearance, and the phenomenon of gradient disappearance is more serious than not adding it. Therefore, we finally used the replacement activation function, setting the activation function of the output layer to sigmoid and the rest to ReLu, which can reduce the impact of gradient disappearance to a certain extent.

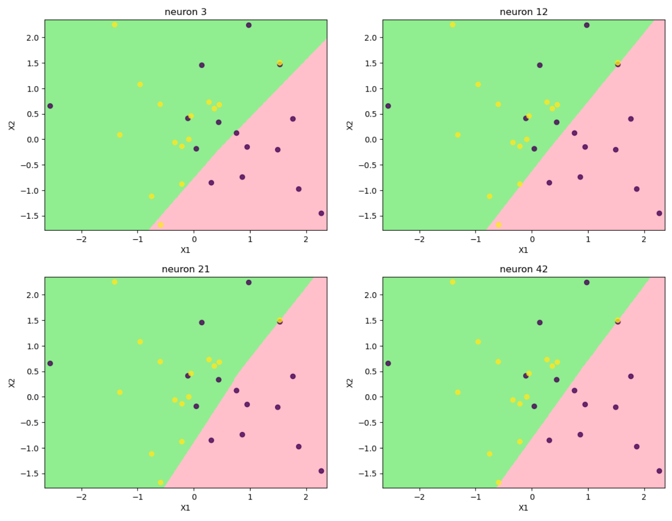

Fig. 11. Decision boundary

The picture in the upper left corner is the result of mlp with 3 neurons, the picture in the upper right corner is the result of mlp with 12 neurons, the picture in the lower left corner is the result of mlp with 21 neurons, and the picture in the lower right corner is the result of mlp with 42 neurons.

TABLE V
WIDE MLP STOCHASTIC DATA EVALUATION

| Neuron_num | 3 | 12 | 21 | 42 |
|---|---|---|---|---|
| Accuracy | 0.5000 | 0.5000 | 0.6667 | 0.6667 |
| Precision | 0.3333 | 0.3333 | 0.5000 | 0.5000 |
| Recall | 0.4000 | 0.4000 | 1.0000 | 0.5000 |
| F1 score | 0.5000 | 0.5000 | 0.6667 | 0.6667 |

From the decision boundaries and the indicators evaluated by each model, it can be seen that the training effect of the width neural network is not good. To a certain extent, this is due to the fact that the hyperparameters have not been adjusted well, and the model itself has differentiated limitations, which cannot capture the accurate position of the nonlinear decision boundary.

## C. Binary Classification using Deep MLP

The mlp model used in this experiment uses multiple hidden layers with different numbers of neurons. It aims to solve the binary classification problem. The results are as follows.

For the linear data:

The optimised method is batch update. Do not use Adam, batch_normalization or regularization. And the epoch_num is 1000. The activation function in the output layer is Sigmoid and the activation function in the hidden layers is ReLu.

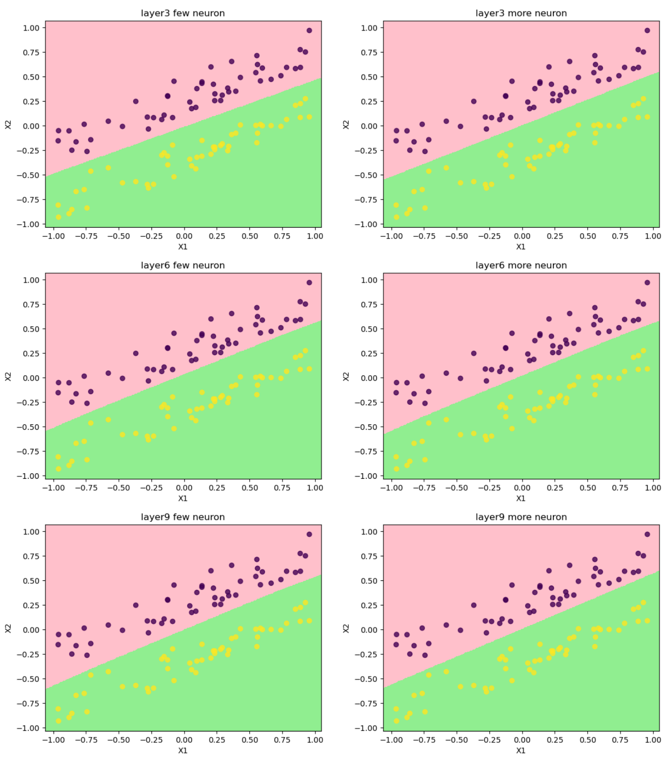

Fig. 12. Decision boundary

The first picture is the result of a 3-layer mlp with few neurons, the second picture is the result of a 3-layer mlp with more neurons, the third picture is the result of a 6-layer mlp with few neurons, the fourth picture is the result of a 6-layer mlp with more neurons, the fifth picture is the result of a 9-layer mlp with few neurons, and the sixth picture is the result of a 9-layer mlp with more neurons.

The correctness of the mlp model can be seen from the decision boundary and various evaluation indicators, and it can easily handle problems with linear decision boundaries.

For stochastic data:

The optimised method is batch update. Do not use Adam, batch_normalization or regularization. The epoch_num of the experiment for neuron 3 and neuron 12 is 50000, the epoch_num of the experiment for neuron 21 is 10000, and the epoch_num of the experiment for neuron 42 is and The activation function in the output layer is Sigmoid and the activation function in the hidden layers is ReLu.

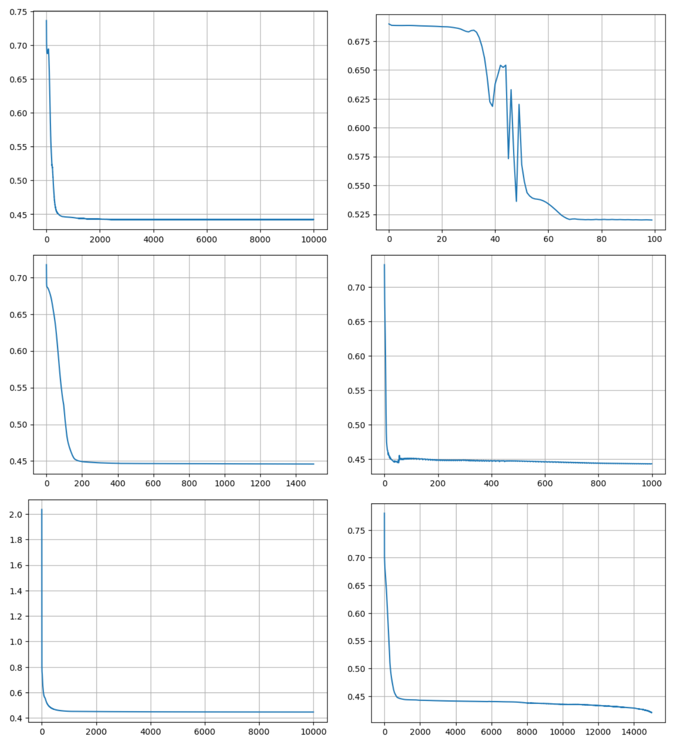

Fig. 13. Loss

The first picture is the result of a 3-layer mlp with few neurons, the second picture is the result of a 3-layer mlp with more neurons, the third picture is the result of a 6-layer mlp with few neurons, the fourth picture is the result of a 6-layer mlp with more neurons, the fifth picture is the result of a 9-layer mlp with few neurons, and the sixth picture is the result of a 9-layer mlp with more neurons.

It can be seen from the change curve of loss that although the overall trend of loss is downward, there is an obvious gradient disappearance problem, and as in the case of width neural networks, the use of popular methods cannot achieve the ideal effect. Therefore, only the method of changing the activation function is adopted, that is, the output layer is a sigmoid function, and all hidden layers are relu functions.

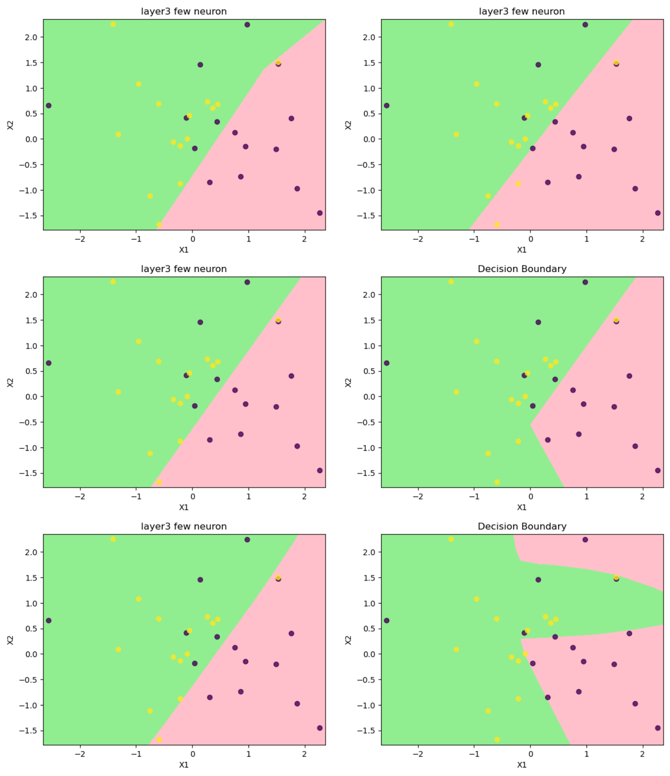

Fig. 14. Decision boundary

The first picture is the result of a 3-layer mlp with few neurons, the second picture is the result of a 3-layer mlp with more neurons, the third picture is the result of a 6-layer mlp with few neurons, the fourth picture is the result of a 6-layer mlp with more neurons, the fifth picture is the result of a 9-layer mlp with few neurons, and the sixth picture is the result of a 9-layer mlp with more neurons.

TABLE VI
DEEP MLP STOCHASTIC DATA EVALUATION

| Neuron_layer | 3 few units | 3 more units | 6 few units |
|---|---|---|---|
| Accuracy | 0.5000 | 0.6667 | 0.5000 |
| Precision | 0.3333 | 0.5000 | 0.3333 |
| Recall | 0.5000 | 0.5000 | 0.5000 |
| F1 score | 0.4000 | 0.5000 | 0.4000 |
| Neuron_layer | 6 more units | 9 few units | 9 more units |
| Accuracy | 0.6667 | 0.5000 | 0.6667 |
| Precision | 0.5000 | 0.3333 | 0.5000 |
| Recall | 1.0000 | 0.5000 | 1.0000 |
| F1 score | 0.6667 | 0.4000 | 0.6667 |

Judging from the decision boundaries and various indicators of the evaluation model, the overall learning effect is average and the model is not well differentiated. The main reason for this is improper adjustment of hyperparameters, and the distribution of the data set also has certain influencing factors. However, we can conclude that as the number of neurons and neural network layers continues to increase, the model has stronger differentiation capabilities and is more capable of learning more complex features.

### D. *Influences of different hyperparameters*

*1) learning rate:* In this experiment, three different learning rates are used to train the linear distribution data. The results of how the learning rate will influence the loss function are as follows.

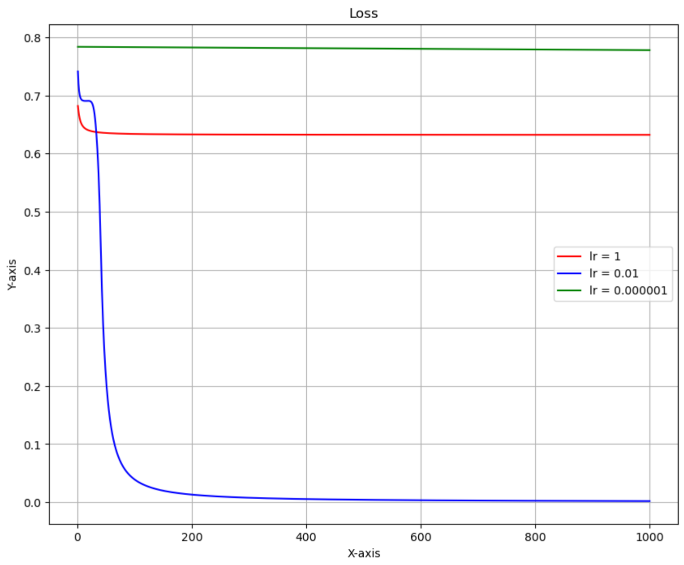

Fig. 15. learning rate influences loss

From the graph, it is clear that when the learning rate is too big or too small, it will lead to the gradient disappear problem. Specifically, the large learning rate can also increase the loss during the training. Thus, when doing experiments, choosing an appropriate learning rate is essential for the success of the model training. When the loss is increased, consider decreasing the learning rate and when the loss is decreasing slowly, consider increasing the learning rate.

*2) number of epochs:* In this experiment, three different epochs are used to train the linear distribution date. The results of how the learning rate will influence the loss function are as follows.

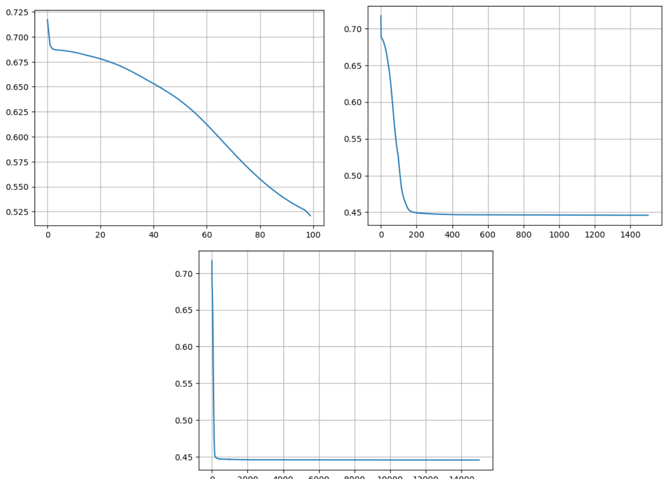
Fig. 16. epoch_num influence loss

The epoch_num of the first picture is 100, the epoch_num of the second picture is 1500 and the epoch_num of the third picture is 15000.

According to the graphs, a small number of epochs will stop the training before the coverage of the model, which leads to a bad performance of the model. Meanwhile, a large number of epochs will waste much time after the coverage of the model, which leads to a waste of time and a waste of computation resources when training the mode. Thus, choosing an appropriate number of epochs will not only better the performance of the model, but it can also use the time and computational resources more efficiently.

*3) number of neurons and layers:* According to the former experiments in the Binary Classification using Wide MLP and Binary Classification using Deep MLP, the more neurons and layers have been used, the capability of the model will increase. However, it will make the training more difficult. Thus, wisely considering the number of neurons and layers will in one hand better the performance of the model. On the other hand, lowers the difficulties of training.

## V. Conclusion and future problems

This project has been a substantial learning experience, offering me an enriched understanding of key concepts discussed in class. Through the implementation of logistic regression and multi-layer perceptron (MLP) models in Python, based on numpy, my coding skills have notably improved. Moreover, this hands-on approach deepened my grasp of the underlying principles and algorithms of each model.

In applying logistic regression to binary data classification, I gained an intuitive understanding of its linear decision boundary and recognized its inherent limitations. The exploration of wide and deep neural networks illustrated that while increasing neurons and network layers enhance the model's differentiation capability, it simultaneously complicates training. My experiments suggested that deep neural networks slightly outperform wide networks in terms of fitting and training efficiency.

The investigation into the effects of various hyperparameters, particularly learning rate and iteration number, on the loss function and overall training outcome, provided invaluable insights into model training dynamics.

However, the project was not without challenges. While using the MLP model for binary classification, issues such as rising loss and gradient disappearance were encountered. Although I sought solutions online, such as Xavier initialization, Adam optimization, batch normalization, and replacing the sigmoid activation with ReLU to address gradient vanishing, and L2 regularization to mitigate overfitting, the results were not always as expected. In most experiments, I resorted to using Xavier for parameter initialization, basic batch updates for data optimization, and switching to ReLU for the hidden layer activation to counter gradient descent issues.

Looking forward, there is a clear pathway for further exploration and improvement. A deeper dive into the theoretical aspects of the methods used, such as Xavier initialization, Adam optimizer, and batch normalization, could enhance the application and effectiveness of these techniques. Additionally, experimenting with different architectures and hyperparameters in neural network models could offer more insights into optimizing training processes and improving model performance. Investigating alternative regularization techniques and more advanced optimization algorithms may also prove beneficial. By the way, solving the problem of gradient descent through residual networks is worthy of continued research in the future. This continued research and experimentation will undoubtedly contribute to a more profound and nuanced understanding of machine learning models and their practical applications.