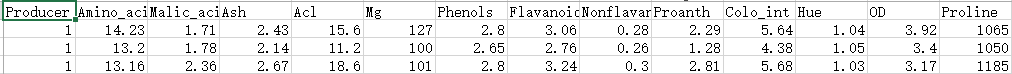
**JC3509 – Machine Learning**

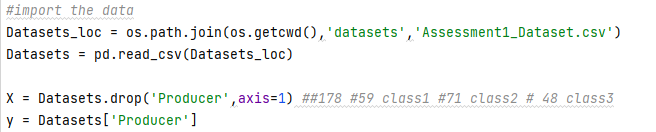
**Liu Kai 50079690**

**Task 1– Data Preparation**

**1.Import the dataset:**



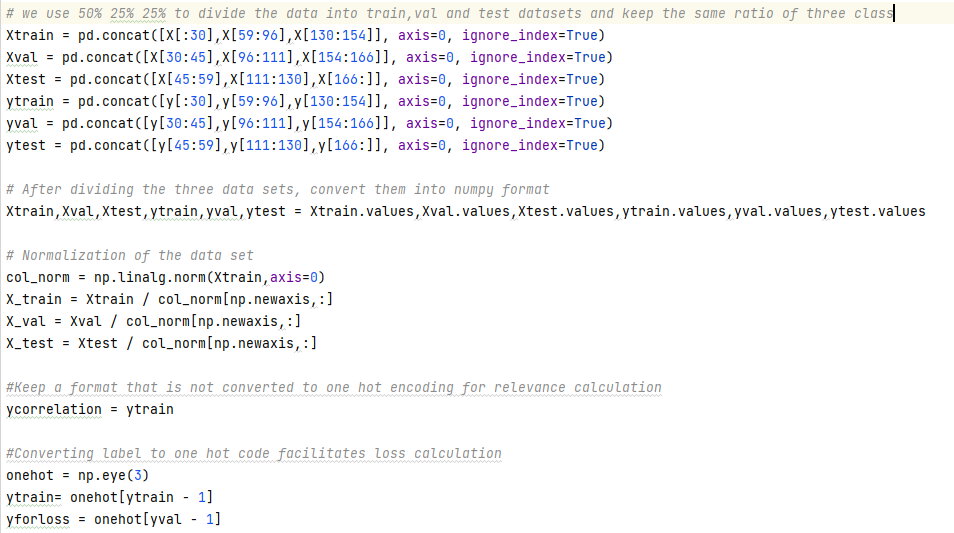
A total of 178 datapoints are provided in the dataset, and the producer is the Target(Response) for this training, which is of 3 types, so it is a triple classification problem. The data also consists of 13 Features(predictor), each of which will affect the prediction weights to varying degrees.



The code reads the dataset after setting up the dataset filter with os and then divides the data into x as well as y based on label and features.

**2.Preprocess the data:**

The data processing is divided into three parts: 1. Splitting the training set, validation set and test set 2. Normalizing the data 3. One-hot coding the label y.



2.1 Segmenting the training set, validation set, and test set: first I used a ratio of 50%, 25%, and 25% to segment the training set, validation set, and test set. Dividing the dataset in this way ensures that there is enough validation and test data for more reliable and stable model performance estimation. However, since only half of the data is used for training, it may limit the ability of the model to learn, and excessive pursuit of accuracy may lead to overfitting. But all things considered, this is a good choice of division ratio.

Since the total number of producers in the three different categories does not remain consistent, blindly randomizing the data may cause the model to have a preference for a particular category, leading to poorer model training. So after comparing the proportions of the three categories in the original data, the proportions were also maintained when splitting the dataset to ensure that the model would not be influenced by too much bias towards one category. However, I just spliced the data proportionally and did not randomize the data in a fixed proportion of each category, which may affect the model's ability to generalize somewhat.

2.2 Normalize the data: first col\_norm only uses Xtrain as a factor is to not expose the val and test datasets to the model during normalization thus leading to overfitting or spurious accuracy. Normalizing the data: 1. puts the features on a similar order of magnitude, which helps the model converge faster. 2. reduces the difference in feature weights so that the model doesn't rely more on larger scale features and ignore smaller scale features. 3. better robustness to noise, allowing the data to be more centralized and less susceptible to noise.

2.3 One-hot coding of label y. Since the original categories are in the form of integers 1, 2, 3, we need to map him to one-hot coding so that it is easier to follow up the calculation of the loss, because the classification problem is outputted as a probability matrix after the softmax function.

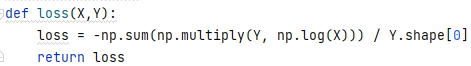
**Task 2 – Model Construction**

**1.Loss function:**

Loss function in this paper I have chosen the cross entropy function because it is widely used in binary and multi classification problems. The cross-entropy function gives more attention to samples with large false predictions, allowing the model to learn from its mistakes and thus speeding up convergence. When the difference between the prediction and the result is particularly large, it becomes particularly small and thus large, contributing a larger value to the loss, and vice versa. And can be very compatible with the output probability of softmax, a direct measure of the difference between the predicted probability distribution and the actual distribution.

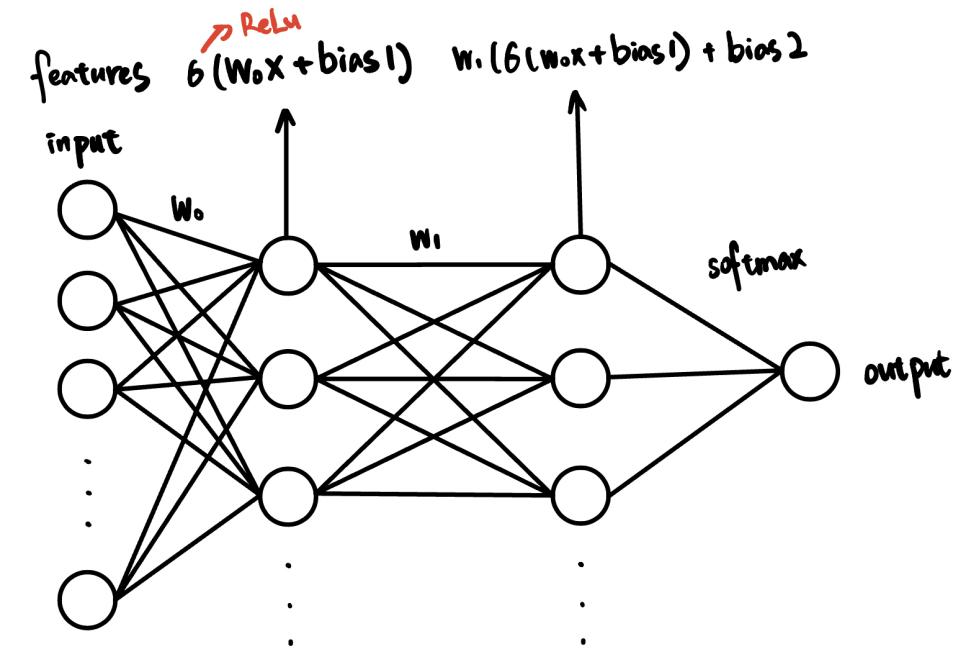


However, if the loss function is calculated using the mean square error, it is not very compatible with the probability preference of the classification problem, because MSE enables the calculation of the square difference between the predicted value and the true value, and is sensitive to noise. However, if MAE and MSE are adopted, they have the same problem, that is, the lack of only probabilistic explanation, and the gradient change is relatively smooth, so the efficiency is too low.



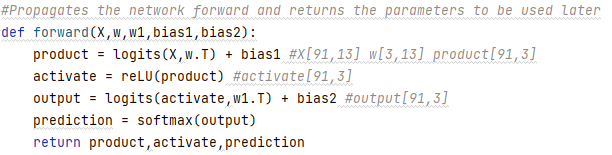
As for the code implementation of the cross-entropy loss function, it is also simple to multiply the probability matrix after the softmax output with the one-hot encoded true label Y element-by-element, and then sum it up, and invert the negative sign to find the average cross-entropy.

**2.Network Design:**

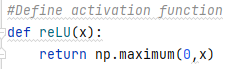


Here is my neural network architecture, with the leftmost input receiving raw data i.e. features. Then in the hidden layer, each neuron receives the data of the input layer, performs weighted W0 sum and bias, and then performs nonlinear transformation through ReLU activation (the effect is to map nonlinear, so that the network can learn and simulate complex functions). Then we go to the output layer, where each neuron again takes the row-weighted average of the data from the hidden layer. Finally, the final prediction result is transformed into a probability distribution by softmax function. This is the architecture of my neural network design, next we will detail the code implementation and the reason for the choice of functions.

2.1 The Forward Pass



In fact, forward propagation is easy to implement, just take the original data through a series of weighted sums of rows plus bias and function activation through rows. The Logits function is just a simple np.dot multiplication function, but it's worth noting that you need to pay attention to the shape of the matrix.



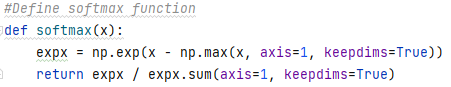
The code implementation of the ReLu function is also very simple, but it is worth noting why the ReLU function is chosen as the activation function for this classification, because the choice of activation function is a key factor affecting the performance of the model!

There are several benefits to choosing ReLU as the activation function first: The function of ReLU is very simple and computationally inexpensive, thus making the training of the model very efficient. For positive inputs, its derivative is 1, which helps with the vanishing gradient problem of the deep network (after many iterations there is almost no gradient leading to ineffective training or loss of information). ReLU makes the output of some neurons of the network 0, making the network more sparse and reducing the consumption of computing resources.

However, if the data is always negative, the ReLU activation function may lead to the problem that the neurons have not been able to activate, which affects the learning of the model.

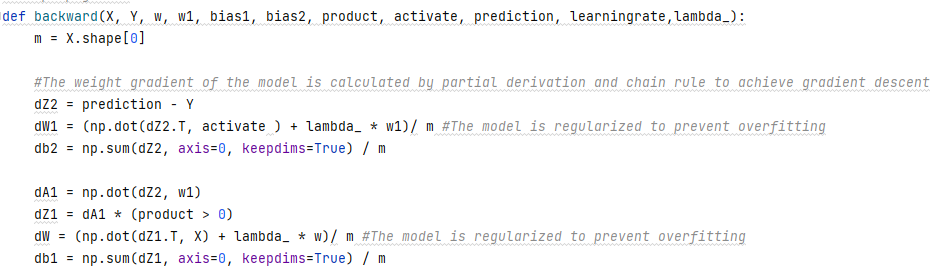
Compared with other activation functions such as Sigmoid function, it is more suitable for binary classification problems, but the gradient will close to 0 when the input value is too large or too small, and the gradient will disappear. The Tanh function suffers from the same problem even though it is centered around 0. Although Swish performs better than ReLU, it has high computational complexity and is not as intuitive as ReLU.

In summary, we chose ReLU function as our activation function in this task.



The Softmax function converts the output into a probability distribution. The first step in the implementation is to subtract the highest value in the data from the raw score, because exponenting a high value might cause a numeric overflow, and subtracting this value does not change the output of the function; it makes the value more stable. And then we take the exponent. In the second step, the output is row-normalized to ensure that the softmax outputs for each sample sum up to 1, which can be interpreted as a probability.

2.2 The Backward Pass

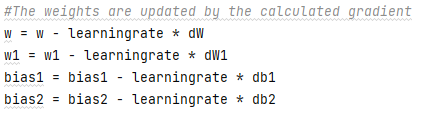


First, according to the output obtained by forward propagation, the difference is calculated by comparing with the true label through the loss function. According to the chain rule, starting from the output layer, the contribution of each weight to the loss is calculated backward layer by layer, which is the gradient.

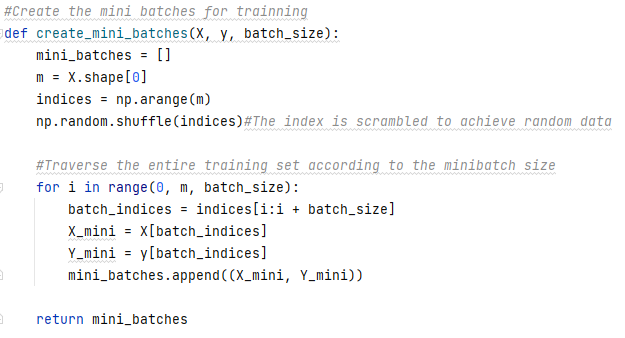
All derivations are done manually and then written in code. Firstly, dZ2 is to calculate the error of the output layer, and take the partial derivative of the cross-entropy loss function. According to the characteristics of the softmax function and the cross-entropy loss function, in the later simplification and summation, except for the terms of the correct category k = j, other terms can cancel each other out. So we can get that for each output node j, the gradient of the linear activation for that node reduces to the predicted probability minus the actual label, expressed in vector form as: prediction-Y. Later dW1, db2 also calculates the gradient of the second layer weights and the gradient of the second layer bias respectively. The dA1 error is backpropagated to the activation function, dZ1 calculates the gradient of the ReLU activation function, and the latter two calculate the gradient of the first layer weights and the gradient of the bias, respectively.

**3 Gradient Descent：**

Gradient descent is the easiest part to implement once you've done the forward and backpropagation, which is updating the weights, and it's pretty simple to do in code:



Based on the gradients obtained through backpropagation, the new weights are obtained by subtracting the learning rate and multiplying the gradient from the old weights. The direction of the gradient is the direction of the fastest increase in loss, so we subtract the gradient in order to reduce the loss. Over many iterations, gradient descent brings the parameters of the model closer to the values that minimize the loss function.

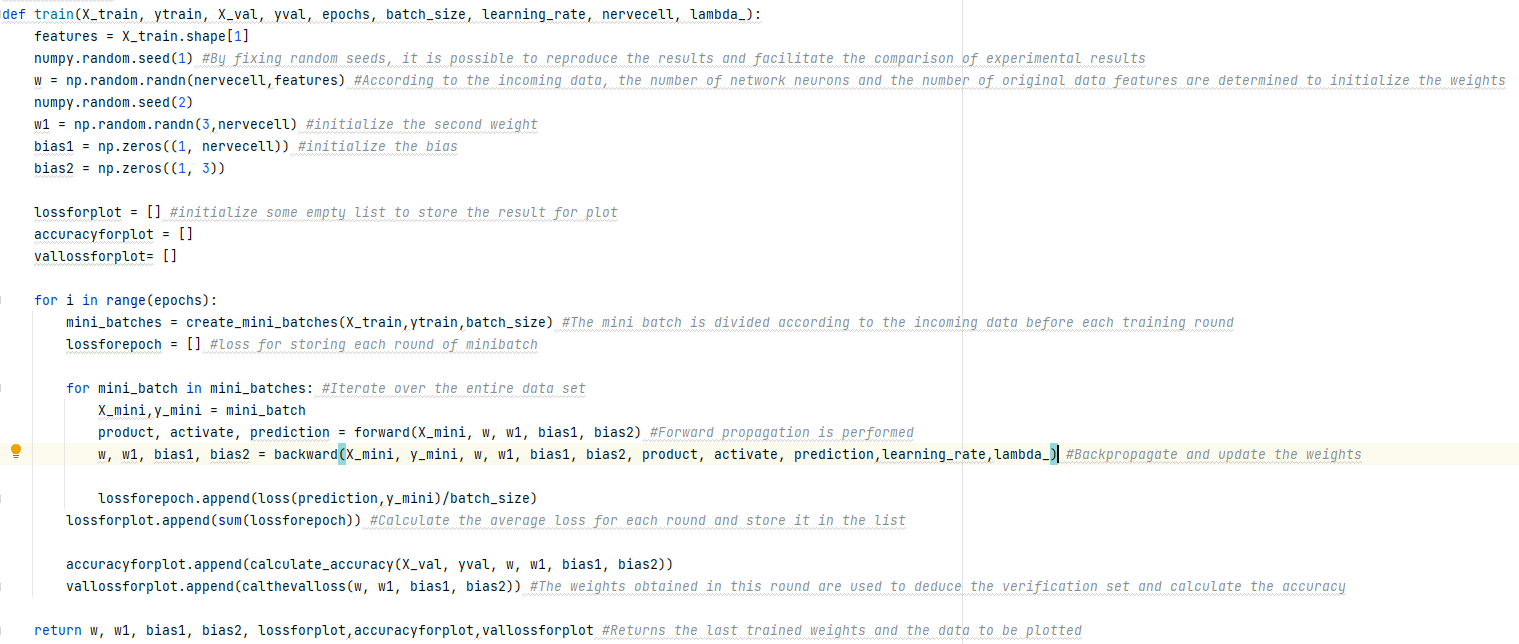


This article also uses mini-batch stochastic gradient descent. The mini batch implementation is shown here, and the train application code is shown later. The implementation of mini batch is also simple, the core idea is to use indexes to process data in batches of rows, so we initialize an array of indexes of the same length as the data, and then shuffle the indexes with the shuffle function to add randomness to the data. Then the index is used to separate the original data, traverse the entire data set, and send it to the training to realize the batch gradient descent. In training, the loss is calculated for each batch and then the weights are updated. To make the loss curve smoother, I choose to average the loss over the entire dataset (called an epoch). Compared with gradient descent using only one sample at a time, mini-batch uses the advantages of matrix, so that the model can operate efficiently. Compared with stochastic gradient descent, each update uses multiple samples, which reduces the variance of the parameter update and makes the training more stable. Of course, for large-scale data, it can reduce the memory required for word iteration and reduce the cost. Randomness is introduced to make the model have a chance to jump out of the local minimum. It also allows the model to converge faster to the optimal solution, because the model only needs to calculate the gradient over a smaller subset of the data, allowing the model to update its parameters more frequently.

**Task 3 – Model Training**

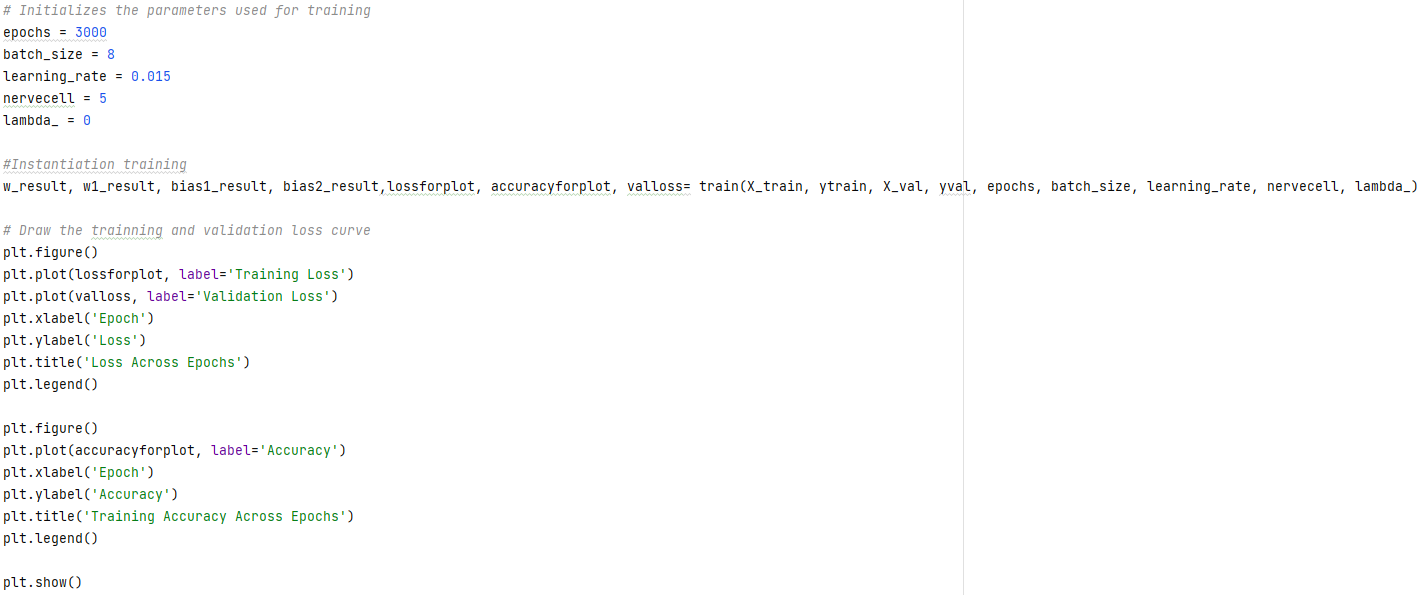
**1.Model Training:**

With an already built model we can add a few features to train it:

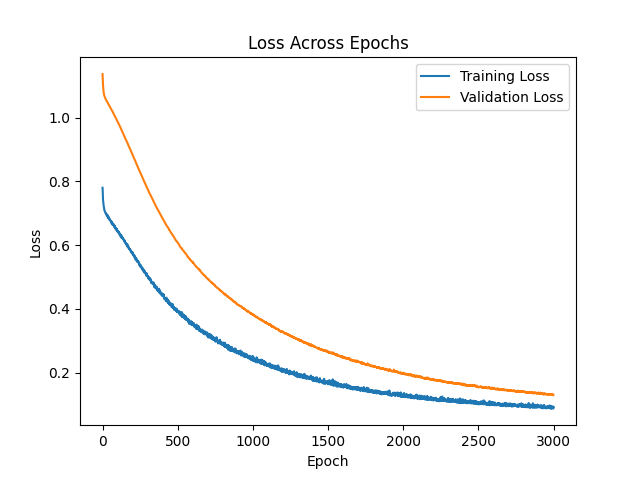
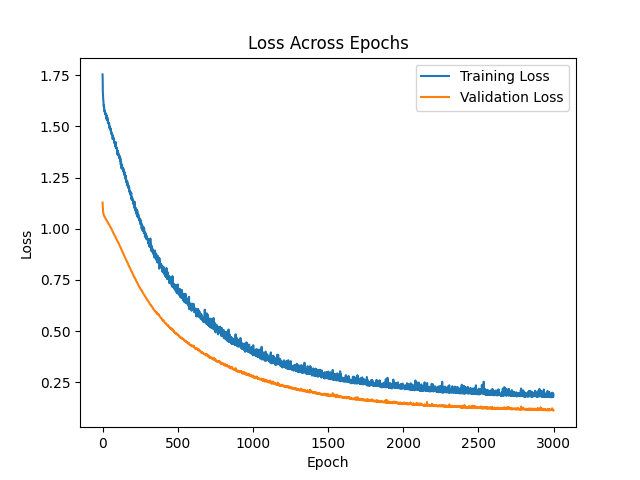


You can see that our train function accepts 9 parameters, the training set for training, and the validation set for real-time feedback on the performance of the model, as well as other custom parameters such as the training epoch, batch size, learning rate, number of neurons, and regularization parameters. We first initialize two weights and two biases from the data we pass in for training, and then initialize some empty lists to hold the plot data.

Then, for each round, we create a mini-batch and initialize the empty list to compute the average loss for each batch. Loop through each batch, propagate forward, propagate back, update the weights and record the data.

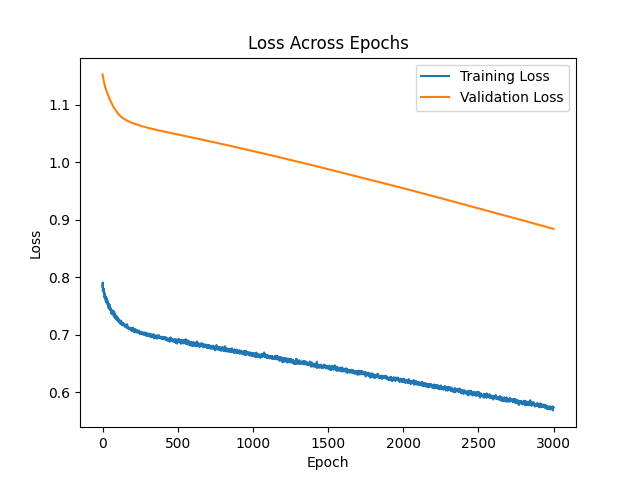
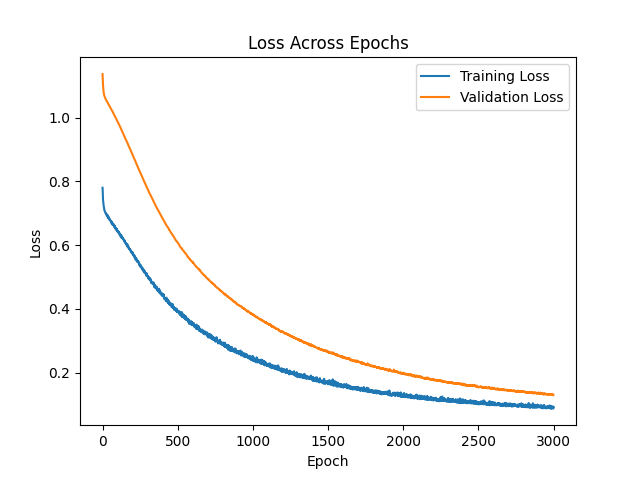
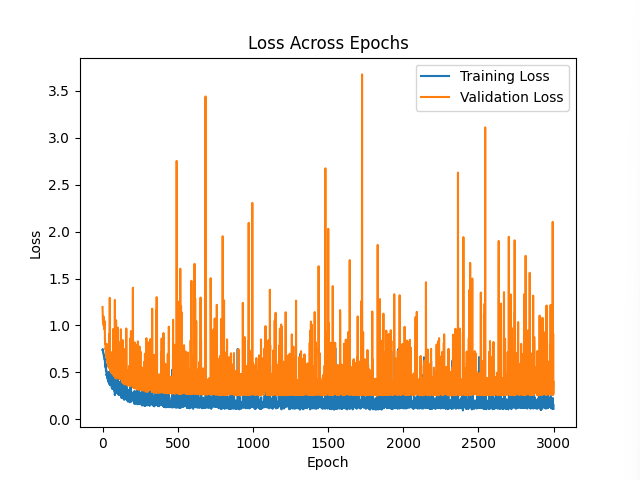


This is where we initialize the parameters and start the training and visualize the loss, accuracy. For the selection of parameters, I did manual hyperparameter, 1. The first adjustment is batch-size: Usually in general machine learning, batch-size is usually selected according to the memory size of 32-256. However, we take into account that since our training set only has 80 or 90 datapoints, a large batch-size may not have much difference with the whole training set, and lack of randomness. In the process of parameter adjustment, it is found that if the batch size is too small, it may lead to too much noise introduced in each update, resulting in unstable training, which may lead to high loss of the training set, poor fitting effect, and longer time consumption. For example, compare the following two figures:



The left figure shows the training with batch-size 8 and the right figure shows the training with batch-size 12. We can easily see from the blue curve of the training loss function that when the batch size is smaller, the model training effect is not good, and the convergence speed loss is high. And it is susceptible to noise so as to compare the fluctuation.

Then we adjusted the learning rate. After experiments, we selected the best learning rate from (10, 1, 0.1, 0.01, 0.001), and then spread half of the learning rate around this point to determine the optimal learning rate. Finally, it is concluded that if the learning rate is too high, the loss function will oscillate or even diverge, and if the learning rate is too small, the model may converge too slowly or even fall into a local minimum. (from largest to smallest) 0.015 is optimal.



The number of neurons is chosen as 5, if too many may lead to model overfitting, poor robustness, and large computational load. Being too small can lead to an underfit model that fails to capture the complexity of the data.

The choice of rounds is as simple as observing that the Loss curve converges and levels off. Blindly adding too many training rounds can lead to overfitting, or too few can lead to underfitting as the model is not learning enough from the data.

The choice of regularization lambda is also relatively simple, as too large will lead to an underfitting model that is too simple to capture the characteristics of the data. On the other hand, being too small can lead to overfitting the model, rather than preventing overfitting. We can see from the results of cross validation that the model is better after manual hyperparameter.

Lambda = 0



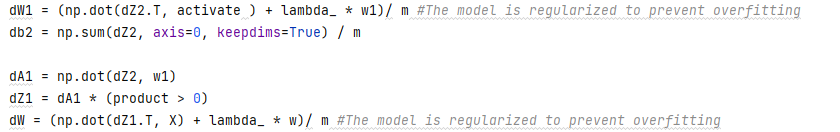
Lambda = 0.015



**2.Model Regularisation:**

Regularization is a technique used to prevent overfitting by penalizing the complexity of the model by adding an additional term to the loss function. If the model learns too well on the training set, accurately capturing noise and error, rather than understanding the underlying data distribution, it will lead to a model with high complexity but low generalization ability, which will not perform well on new data.

In my code I used L2 regularization, also known as weight decay, which adds the trivial sum of all weights times a regularization factor to the loss function.



**3.Model inference:**

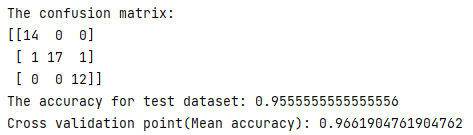
Feedback from the validation set is needed to evaluate the model in real time during training or to adjust the model parameters to achieve better training results. In my code, the train function evaluates the model against the validation set after each weight update: by calculating the validation loss and the prediction accuracy.



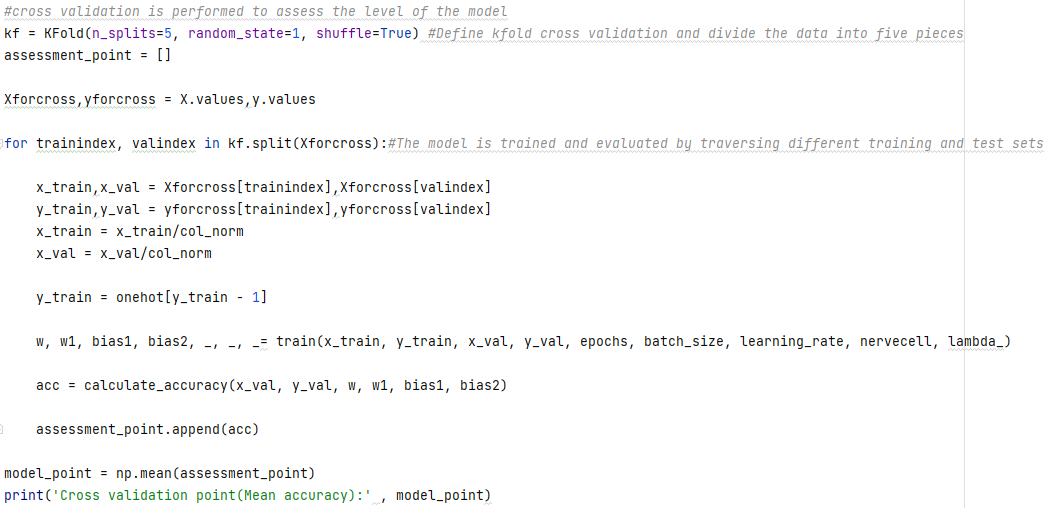
**Task 4 – Evaluation**

**1.Present Results:**

This paper uses a variety of methods to evaluate the quality of the model: 1. Calculate the confusion matrix on the test set 2. The accuracy of the test set is calculated 3. The model is tested by cross validation.

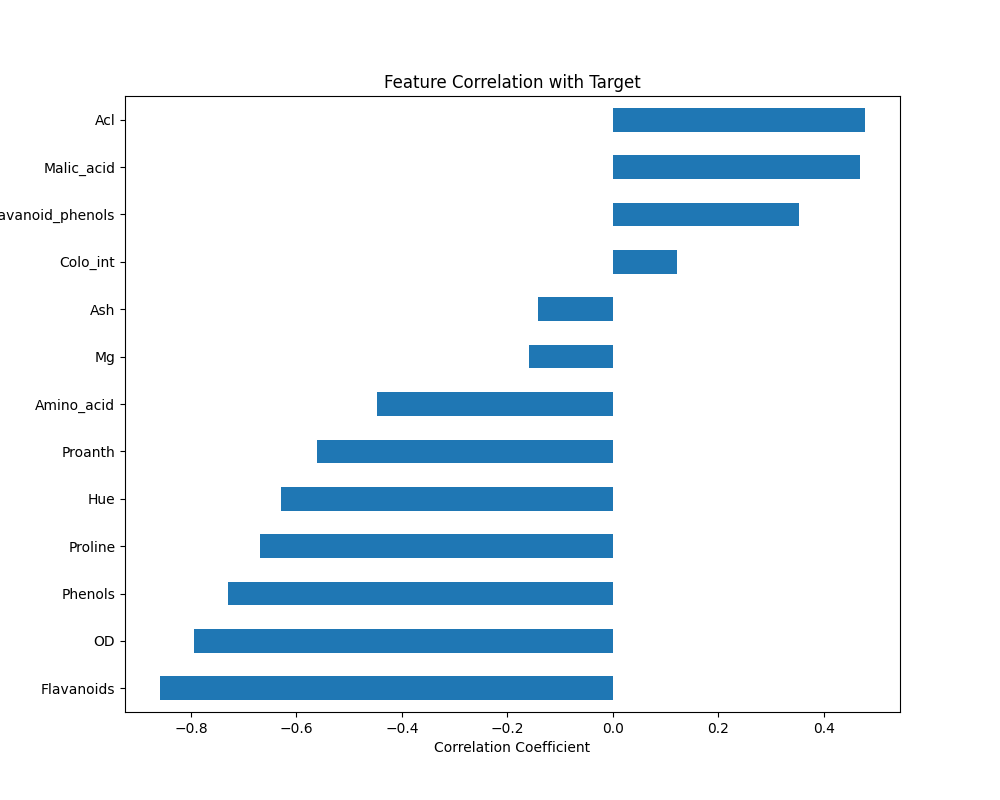
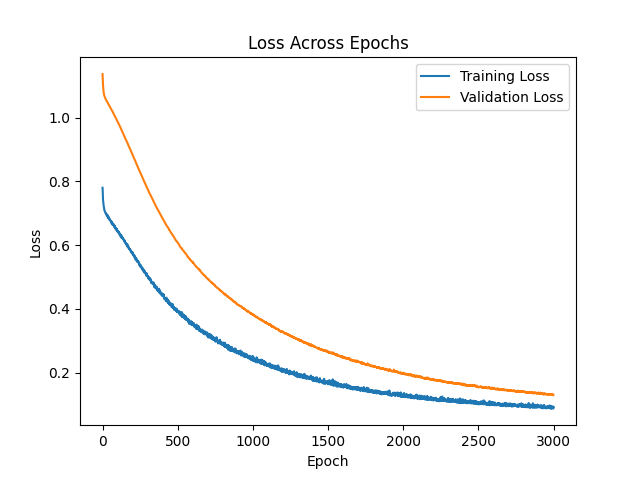


After training and tuning the model parameters, we can see that the prediction on the test set has only two errors, with an accuracy of 95.6%. cross validation tells us that the model has strong generalization ability, is suitable for different new data and maintains a high accuracy of 96.6%.



**2.Plots:**

The results presented in this plots are: loss on the training set and loss on the validation set. Accuracy on the validation set Impact of different features on model predictions.



From the loss function plot we can see that both loss functions decrease as the number of epochs increases, which means that the model is learning and improving its performance on the validation set. As the number of epochs increases, both the training loss function and the validation loss function tend to be stable and show a convergence trend. The relationship between the two curves also tells us that this is the best fit model because if we overfit, the training loss will continue to decrease while the validation loss will stop decreasing or even increase. There are also no cases of underfitting: both the training loss and the validation loss are relatively high.

We can also see in the validation set accuracy image that with the increase of rounds, the accuracy of the validation set gradually increases and tends to 1 at the end.

The smooth decrease in loss also shows that the model learning rate is set appropriately, without oscillations or too slow decrease.

It is hard to see from the image whether the regularization of the model has a big impact. However, as we mentioned above, after setting the appropriate lambda, the performance of the model in cross validation is improved, which proves that the regularization of the model effectively prevents the model from overfitting and brings a certain generalization ability, so as to better reason about new data.

**3.Explain results:**

After my efforts, I finally successfully built a neural network with a hidden layer in this task, and had a high accuracy rate for the prediction of the manufacturer. Through code writing and experiments, I learned that in the construction of a deep learning network, the architecture, function and each parameter will have a great impact on the training and learning of the model.

**5.Discussion**

Even though I think I have completed this job completely and excellently, completed every subtask and implemented the neural network without the help of external packages, there are still some defects. However, on the basis of completion, there is no attempt to implement some more advanced algorithms such as potential energy, dynamic adjustment of learning rate, and roc function evaluation. There is also a lack of standardization in the writing of the code. Due to the excessive nested call functions, the modification and maintenance become particularly troublesome, and many parameters need to be added to ensure the normal operation of the code. In the end, it was too expensive to modify the code to implement the validation loss function and chose to use the global variable directly (but only this one function!). . So there is still a lot to learn and a lot of room for improvement in code writing in the future. But on the whole, I am still satisfied with myself, because I thought I understood the principle very well when I took the class, but after practice, I found that there were still many things I did not understand. However, after exploration, I clarified the concepts and architectures that I did not understand, learned a lot of additional knowledge, and exercised my coding ability, benefiting a lot.