MLP

Wavelix

1 Requirment

1. Develop a MLP model using numpy, which can handling any number of layers and units per layer. forward and backward propagation processes are needed.
2. Both MBGD and SGD can be implemented during the training.
3. Use k-fold volidation to assess the model's performance under different hyperparameters.
4. Select a nonlinear function to generate a dataset, and use the model above to train and Analyze.
5. Create a dataset which is suitable for binary classification, and also use the model above to classify the dataset.

2 Idea

The weights and bias are generated according to the parameter **layers**, which indicates the numbers of layers and the neurons in each layer. Three different activition functions are realized:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \mathbf{ReLU}(x) = \mathbf{max}(0, x) \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Here, I choose the **ReLU** as the activition function for the output layer, and that of the hidden layers can be selected from the 3 functions above. The backward propagation updades the weights and biases by computing gradient based on the error between predictions and actual values, which are generated from cross entropy.

The realization for k-fold validation and gradient decent methods are integrated in the method **train()**. For every k epochs, dataset is randomly divided into k folders, and in each epoch we choose 1 folder as validation set in sequence, while the rest are regarded as training set. **train()** can accept a parameter named **batch_size**. If **batch_size=1**, SGD will be applied, otherwise MBGD will be implied, and the mini-batch-size of which is equal to **batch_size**. Cross entropy loss will be calculated for every epochs.

The nonlinear function is calculated by

$$y = \sin(\pi \cdot x) + \text{noise} \qquad x \in [-2, 2]$$

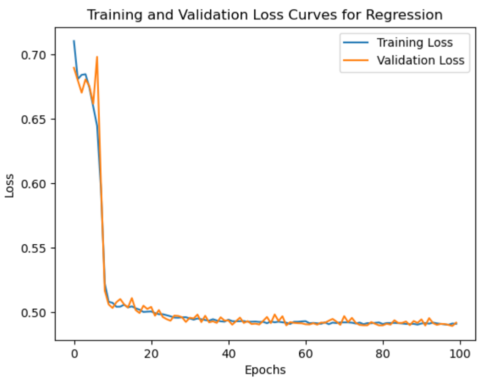
To obtain the dataset for binary classification, we generate 2 sets of points, one of which are generated from a a standard normal distribution centered around $(1, 1)$, and the other are generated from a a standard normal distribution centered around $(-1, -1)$. Then, assign label 0 to the first cluster and 1 to the second.

The training task is up to your input. Input 'r' for linear regression model and 'c' for classification model. Here, Since I was struggled with the image plot of classification result, I asked AI to help me plot the result of classification, including the boundary.

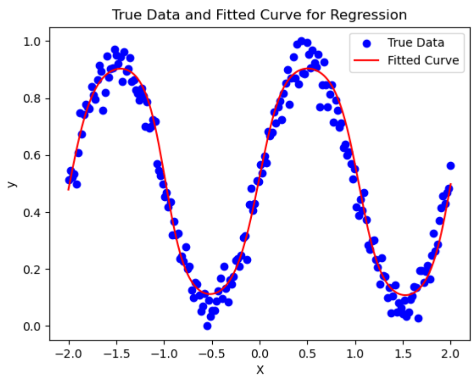
3 Result

3.1 Linear Regression

Firstly, we set the neuron networks as $[1, 10, 1]$, and apply sigmoid funcion in all hidden layers. When using SGD, a good realization is as follows:



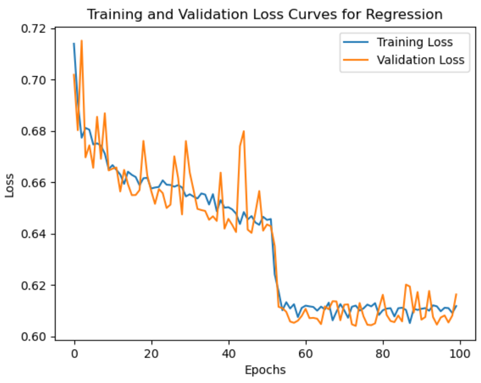
(a) loss curve



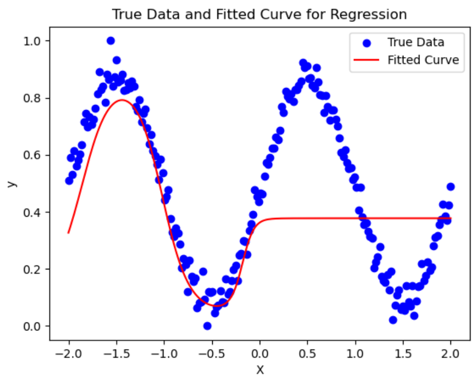
(b) prediction

图 1: SGD, sigmoid, layers=$[1, 10, 1]$, epochs=100, lr=0.5

When increase the number of layers, such as $[1, 10, 10, 1]$, the performance become bad under the same epochs and learning rate:



(a) loss curve

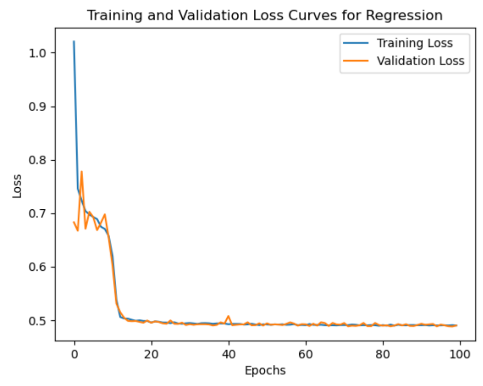


(b) prediction

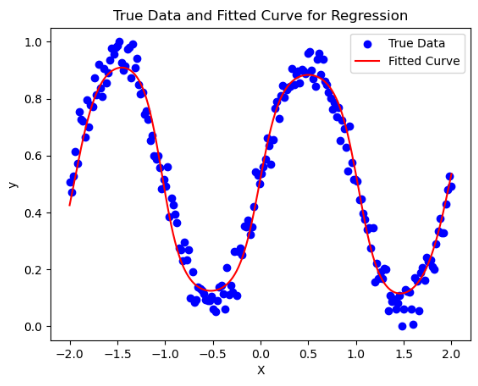
图 2: SGD, sigmoid, layers=$[1, 10, 10, 1]$, epochs=100, lr=0.5

To reachieve the previous performance, methods like increase the epochs and decrease the learning rate can be applied. However, problems like sharp loss curve changes will occur. Better methods can be found from more dicoveries below.

Next, we try to modify the number of neurons in the hidden layer, and it seems there is no obvious change in performance:



(a) loss curve



(b) prediction

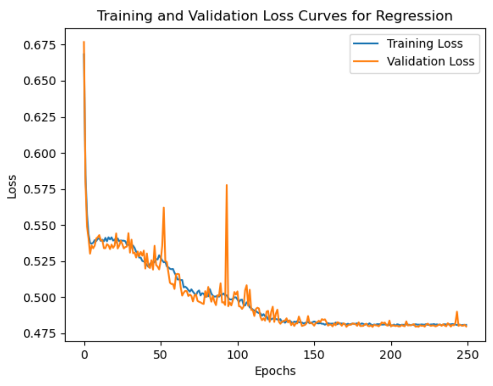
图 3: SGD, sigmoid, layers=$[1, 100, 1]$, epochs=100, lr=0.5

After modifying the activation function in the hidden layers, the training becomes tough. Finally we found the initialization of weights is too small:

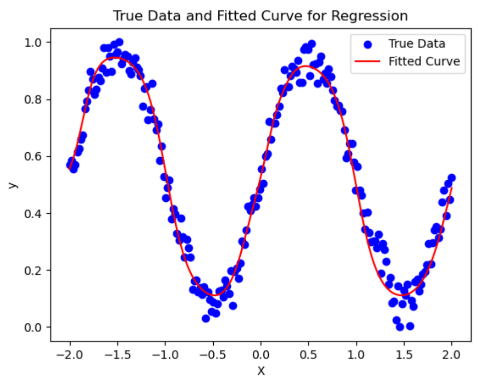
```
self.W.append(np.random.randn(layers[i], layers[i+1])*0.05)
```

After removing ***0.05**, training becomes easy, even for larger number of layers.

Still, a smaller learning rate is needed, so it seems that **Tanh** and **ReLU** trains faster than **sigmoid**. Below is the result using **Tanh**:



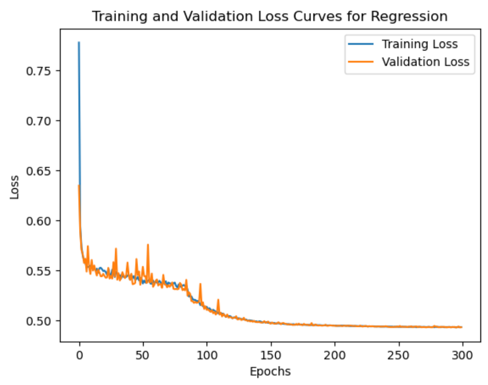
(a) loss curve



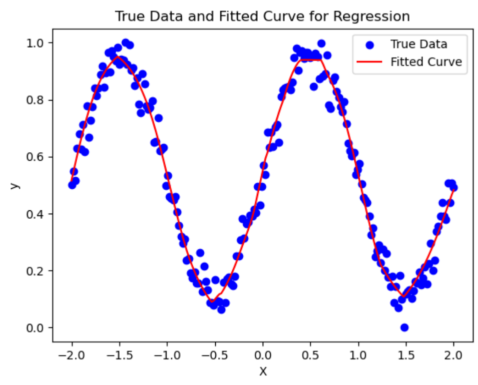
(b) prediction

图 4: SGD, Tanh, layers=$[1, 10, 1]$, epochs=100, lr=0.1

ReLU function needs a learning rate that is even smaller, and its performance is not so good in shallow neuron networks. If we increase the number of layers in the MLP, performance becomes better:



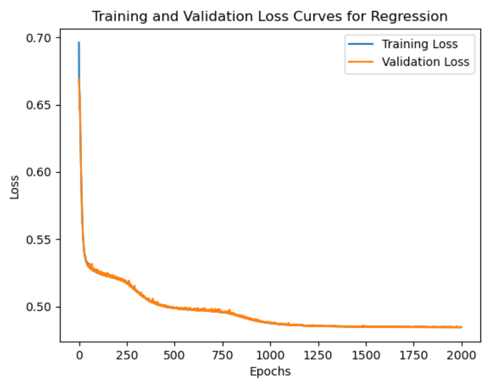
(a) loss curve



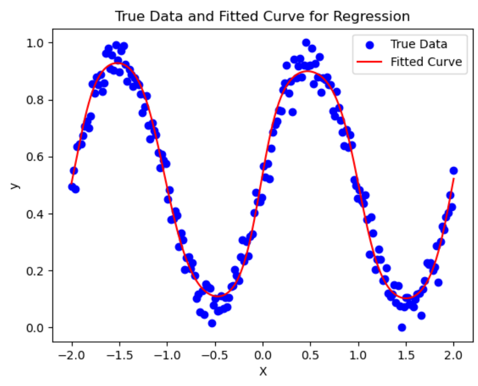
(b) prediction

图 5: SGD, ReLU, layers=$[1, 10, 10, 10, 1]$, epochs=300, lr=0.01

Next, we try MBGD. Still, it trains slower but smoother than SGD. Different number of layers and units has the same effects on MBGD. Below is a example using MBGD with **sigmoid** function for hidden layers:



(a) loss curve

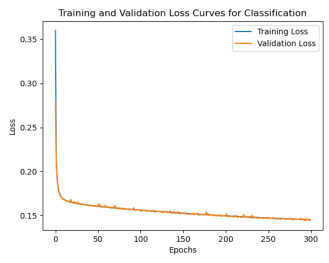


(b) prediction

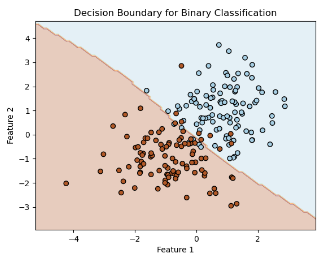
图 6: MBGD, sigmoid, layers=$[1, 10, 1]$, epochs=2000, lr=0.5, batch size=20

3.2 Binary classification

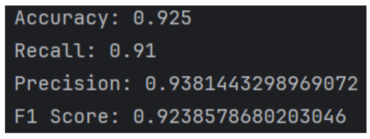
Firsty, we set the layer size as $[1, 10, 1]$, and apply sigmoid function in all hidden layers. a good realization with SGD is as follows:



(a) loss curve



(b) prediction

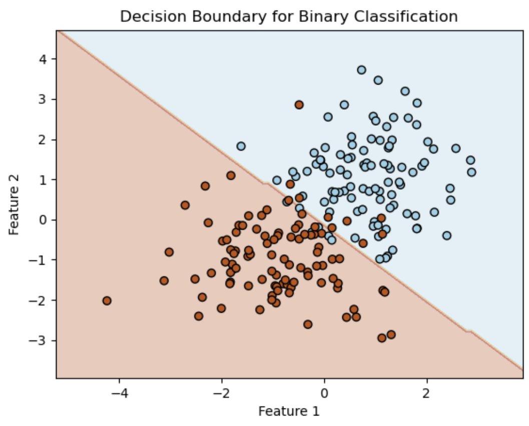


```
Accuracy: 0.925
Recall: 0.91
Precision: 0.9381443298969072
F1 Score: 0.9238578680203046
```

(c) metrics

图 7: SGD, sigmoid, layers=$[2, 10, 1]$, epochs=300, lr=0.01

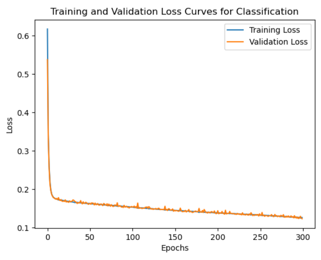
Then, decrease the number of neurons in the hidden layer, $[1, 2, 1]$ for example, and the decision boundary is more like a straight line (no change for metrics in this case):



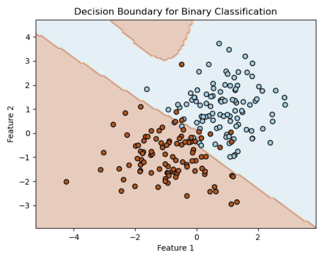
(a) prediction

图 8: SGD, sigmoid, layers=$[2, 2, 1]$, epochs=300, lr=0.01

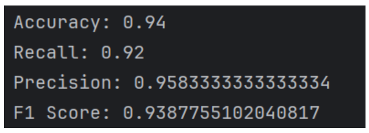
Next, increase the number of layers, $[2, 10, 10, 10, 1]$ for example. Overfiting occurs.



(a) loss curve



(b) prediction



```
Accuracy: 0.94
Recall: 0.92
Precision: 0.9583333333333334
F1 Score: 0.9387755102040817
```

(c) metrics

图 9: SGD, sigmoid, layers=$[1, 10, 10, 10, 1]$, epochs=300, lr=0.01

The influence of different activation function and gradient decent method on binary classification is the similar to that on linear regression.

4 Analysis and Conclusion

MLP with less number of layers or neorons is suitable for simple tasks for its fast training speed, but it may meet difficulties when faced with complex dataset. For more complex cases, deep MLP or more neorons is needed, which can learn more during the training. However, the risk of overfiting also increases, and increasing the layers of MLP is more likely to cause overfiting than increasing number of neorons in each layer. Apart from that, large epochs also tends to lead overfiting.

For activation function in hidden layers, **ReLU** is the fastest to converge, while **sigmoid** is the slowest. **Tanh** seems perform better than **ReLU** since it makes the data closer to 0. **ReLU** may meet difficulties when the MLP is shallow. However, when MLP is deep, ReLU can perform better.

The effects of SGD and MBGD on the training is similar to that on the previous tasks.