COMP30230 Connectionist Computing

Multi-Layer Perceptron MLP Project Report

Student Name: Zairui Zhang

Student ID: 19209905

Coding choice:

Python is a popular choice for students studying Computer Science with Data Science because of its familiarity with the curriculum and its widespread use in business applications. Neural networks, including MLP, were covered in a number of modules, including data mining, machine learning, and deep learning (which will be covered next semester), which helped students grasp their importance in contemporary data analysis.

Although there are several advanced neural network libraries available, the choice to create MLP from scratch in Python was deliberate. Using this method allowed us to go beyond the abstractions offered by pre-built libraries and gain a deeper grasp of the fundamental processes and complexities of MLP. This practical application promotes a thorough comprehension of MLP's fundamental principles, providing a more sophisticated understanding of its inner workings and facilitating a deeper investigation of neural network architecture and optimization strategies.

This method seeks to build a strong foundation in neural networks, guaranteeing a firm grasp of the theoretical and practical elements, by eschewing the convenience of well-established libraries. A deeper understanding that is essential for future academic endeavors and practical applications in the fields of data science and machine learning is fostered by this in-depth knowledge, which also helps with the ability to adjust and improve network structures.

The decision to implement MLP in Python from scratch signifies a commitment to mastering the fundamentals, allowing for a richer understanding of neural networks that extends beyond the functionalities offered by existing libraries. This approach equips with a profound comprehension of MLP and neural networks, laying a strong groundwork for future exploration and innovation in this domain.

1. mlp.py

In this class we create basic attributes(NI, NH, No etc….) and initialize them. Then we create several functions, a randomize function (random the 2 layers weight into a small value and also set the changes in two weight to zero), a forward function(Performs the forward pass through the network), a backward function(Performs the backward pass to compute errors and weight updates, In this function for the weight update DW1 and DW2 which use two ways for like sin test we should do reshape at here) and a updateweight function(Updates the weights using the computed gradients and a specified learning rate).

2. xor.py

This class is to train and test XOR data.

```
NI = 2
NH = 3
NO = 1
learning_rate = 0.4
epochs = 10000
```

When I use these parameters I got :

Print result every 1000 times, and a full result in the output file called ‘ errors_xor.txt ’

```
Epoch 0: Loss - 0.3183
Epoch 1000: Loss - 0.1275
Epoch 2000: Loss - 0.1146
Epoch 3000: Loss - 0.0548
Epoch 4000: Loss - 0.0299
Epoch 5000: Loss - 0.0197
Epoch 6000: Loss - 0.0145
Epoch 7000: Loss - 0.0113
Epoch 8000: Loss - 0.0093
Epoch 9000: Loss - 0.0078
Final predictions:
Input: [0 0] -> Predicted Output: [0.10375618]
Input: [0 1] -> Predicted Output: [0.92742047]
Input: [1 0] -> Predicted Output: [0.92301035]
Input: [1 1] -> Predicted Output: [0.07049986]
```

It seems that the model has learned the XOR logic gate quite well. For the XOR gate, the expected outputs are [0, 1, 1, 0] respectively for the inputs [0 0], [0 1], [1 0], and [1 1]. The model's predictions align well with these expected outputs. The decreasing loss values over epochs indicate that the model is learning and minimizing the error between predicted and actual outputs.

3. sin.py

This class is to train and test SIN data. With these steps:

Generates 500 random vectors with 4 components each, with values between -1 and 1.Calculates the sine of a specific combination of components (x1 - x2 + x3 - x4) to obtain the output values. Splits the dataset into a training set of 400 vectors and a test set of 100 vectors for input and output.

Model Initialization:

```
NI = 4
# try different hidden size from 5,6,7,8,9
NH = 5
NO = 1
learning_rate = 0.1
epochs = 1000
```

Iterates through epochs and the training dataset. For each epoch, computes forward pass (prediction) and backward pass (backpropagation) for each input vector. Updates the model's weights based on the calculated error. Evaluates the trained model on the test set. Computes the mean squared error between the predicted and actual output values for the test set.

```
Epoch 0: Training Loss - 196.8041
Epoch 100: Training Loss - 94.6989
Epoch 200: Training Loss - 94.4732
Epoch 300: Training Loss - 94.2935
Epoch 400: Training Loss - 94.1802
Epoch 500: Training Loss - 94.0866
Epoch 600: Training Loss - 93.9630
Epoch 700: Training Loss - 93.8169
Epoch 800: Training Loss - 93.6611
Epoch 900: Training Loss - 93.4846
Test Loss: 0.2762
```

The training loss values decreasing over epochs indicate that the model is learning from the training data. The rapid decrease in loss during the initial epochs suggests that the model experiences significant improvements in its predictions at the start of training. However, as training progresses, the loss keeps decreasing after a thousand times, We can conclude that as the number of training times increases, the accuracy will gradually improve, but it may not change much when it reaches a critical point. The reported test loss of **0.2762**, due to the random data we use, I tried multiple times and the result is around 0.2 which means MLP still works for sin.

4. Special-Test

This class for train an MLP on the letter recognition

with these steps:

Downloads the "letter-recognition" dataset. Preprocesses the data by converting character labels to numerical values and converting string values to integers. Splits the dataset into training and testing sets. Initializes (MLP) and trains it. Prints the training loss at intervals during training. Tests the trained MLP on the test set and computes the accuracy. Input 16 for 1-17 attributes, 26 outputs for 26 different letters in the alphabet, hidden units, learning rate and epochs(>1000) can be various.

```
# parameters
NI = 16
NO = 26
NH = 20
learning_rate = 0.01
epochs = 1000
```

Then we got the result: Test Accuracy **70.47%**

```
Epoch 0: Loss - 1277.6570
Epoch 100: Loss - 369.6422
Epoch 200: Loss - 344.0286
Epoch 300: Loss - 315.9571
Epoch 400: Loss - 272.9855
Epoch 500: Loss - 258.5901
Epoch 600: Loss - 245.3830
Epoch 700: Loss - 240.9110
Epoch 800: Loss - 238.7284
Epoch 900: Loss - 237.8960
```

The loss decreases considerably from the initial epoch to the 900th epoch, indicating that the model is learning from the training data and improving its performance. However, it's essential to note that the loss values are relatively high. The accuracy 70.47% still have room for improvement by hyperparameter tuning. And we can also test with more epochs to see the enhancement.

5. Conclusion:

Objective: To train an MLP on XOR.

The training process showed that the model struggled to converge and make accurate predictions using a sigmoid activation function. The predictions were not consistent and seemed erratic. Adjusting the learning rate could lead to better convergence, but the performance was still not optimal.

Objective: Training an MLP to predict a SIN function.

The model showed better convergence and prediction accuracy on the sinusoidal function when compared to the XOR dataset. Utilizing a sin activation function in the hidden layer facilitated better predictions for this type of function.

Special Test: Applying the MLP for letter recognition.

Training the MLP on a letter recognition dataset showed a decreasing trend in loss over epochs, implying the model learned progressively. However, the accuracy achieved was around 70.47%, indicating that further improvements might be necessary to enhance performance.

Overall:

This assignment let me do a deep research on Multi-Layer Perceptron (MLP) and implement it without any existing library in Python. I learned a lot of theoretical knowledge from this module and online and also have this practical assignment to apply that knowledge to a real project. Which gives me a solid foundation for next semester’s module deep learning. However, I aim to refine the code by exploring hyperparameter tuning for better accuracy across diverse datasets. Focusing on optimization techniques like grid search, I aim to enhance model performance. I'll also streamline the codebase for better readability and flexibility. Overall, while this assignment was enlightening, I look forward to optimizing parameters and code for more resilient neural networks in the future.