Artificial Intelligence Coursework Report includes a self-marking sheet

Self-marking sheet:

10	self-marking sheet
10	running code
5	two-fold test
12	quality of code
13	report
11	quality of result
15	quality of algorithm

Algorithm:

In the coursework, I selected MLP (Multi-Layer Perceptron) algorithms as the foundation, including key components such as weight initialization, forward propagation, and backward propagation. The module is divided into three parts: Input Layer, Hidden Layer, and Output Layer. These layers implement data transformation using weights and biases.

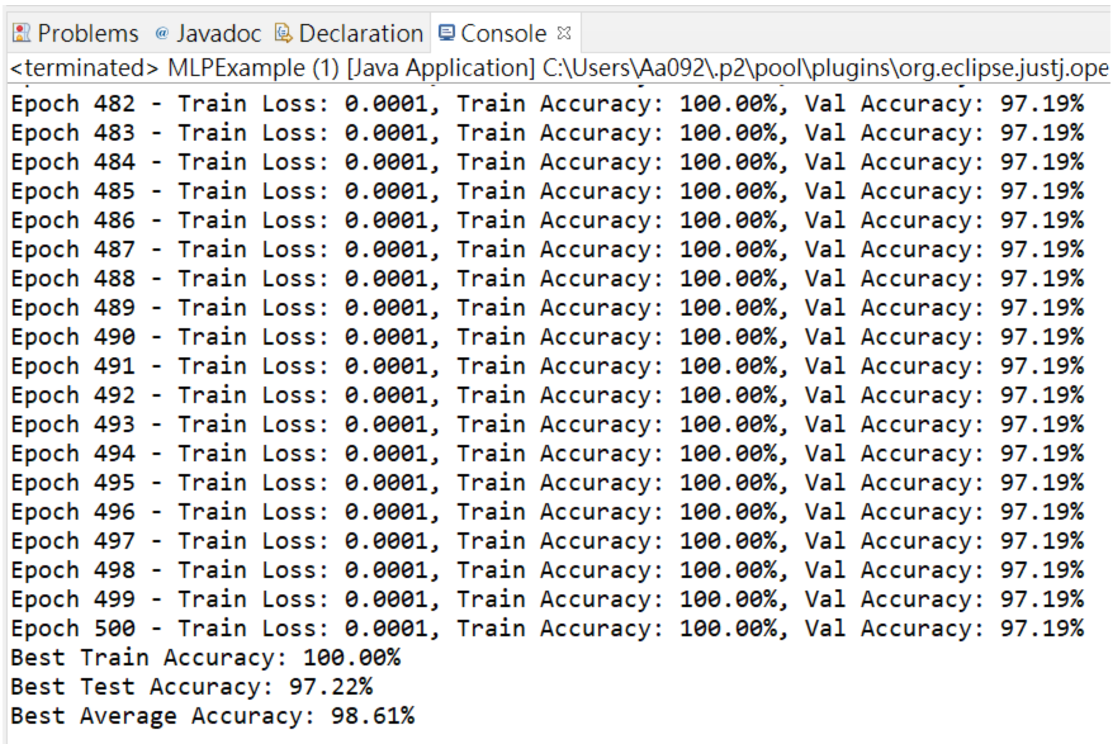
Before feeding data into the model, I performed pre-processing on the input data. Each dataset entry is a 1 X 64 array. The first 64 elements represent the features of the data, where each element ranges from 0 to 16. The final element is the label or target value. To prepare the data, I standardized the features by dividing each of the first 64 elements by 16, scaling the feature range from 0–16 to 0–1. This preprocessing step reduces the complexity of the dataset and makes it easier for the MLP model to train effectively. Then the labels in the dataset will be transformed into a 1×10 array, where only the element corresponding to the correct answer is set to 1, and all other elements are set to 0.

After preparing the dataset, I first initialize weights and biases. Then, we feed the 1×64 input into the model. Each element is multiplied by its corresponding weight, summed up, and added to the bias to produce the output of each layer. And then, apply an activation function (In this homework I use Softmax) to the output to introduce non-linearity, enabling the model to capture complex patterns in the data. At the final output layer, the result will be a 1×10 array, identical in size to the pre-processed labels. However, instead of binary values, each element represents the probability of the corresponding digit (0–9). For example, if the model predicts the digit 2, the output may be [0.02, 0.06, 0.78, 0.12,…], where the value at index 2 is the highest, indicating the highest probability for that digit.

After obtaining the output, we compare it with the correct answer and calculate the loss, determining the error between the predicted and actual values. This error, combined with the learning rate, is used to update the weights and biases. In this project, we utilized L2 Loss and Cross Entropy Loss to strive for better results. Throughout the process, we set a hyperparameter called batch size, which defines the number of samples processed before updating the weights. By using a batch of data instead of a single instance, the model adjusts its parameters based on more comprehensive information. This approach aims to enhance the model's generalization ability.

By repeatedly executing the algorithm described above, we can form the MLP algorithm. In this project, we set a hyperparameter called step. Once the number of iterations reaches the value defined by step, we multiply the learning rate by a float. This approach is intended to allow the model to use a higher learning rate during the early stages of training, enabling faster convergence. As the model approaches a better solution, we reduce the learning rate to allow the model to make finer adjustments and more stably reach the optimal result.

The reason I chose the MLP (Multilayer Perceptron) algorithm is as follows. MLP (Multilayer Perceptron) is a parametric model that can learn the underlying features of data through training. It is suitable for handling high-dimensional and complex data. By enhancing the network structure, such as adding hidden layers or increasing the number of neurons, or modifying the loss function, it can adapt to more complex problems. I believe that when applying artificial intelligence to real-world problems, they are typically more complex. Therefore, in this project, I chose to implement MLP, which is more closely aligned with real-world applications.



```
<terminated> MLPExample (1) [Java Application] C:\Users\Aa092\.p2\pool\plugins\org.eclipse.justj.ope
Epoch 482 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 483 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 484 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 485 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 486 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 487 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 488 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 489 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 490 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 491 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 492 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 493 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 494 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 495 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 496 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 497 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 498 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 499 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Epoch 500 - Train Loss: 0.0001, Train Accuracy: 100.00%, Val Accuracy: 97.19%
Best Train Accuracy: 100.00%
Best Test Accuracy: 97.22%
Best Average Accuracy: 98.61%
```

The hyperparameter of this result is: Hidden Size = 128, Output Size = 10, Learning Rate = 0.1314, step = 450, epoch = 500, batch size = 32, Loss function = CrossEntropyLoss.