Multi-Layer Perceptron Programming Assignment

Connectionist Computing

Devon Knox

5th December, 2024

1 Introduction

This project implements and tests a Multi-Layer Perceptron (MLP) in Java, which learns by backpropagation and can accommodate variable quantities of inputs, hidden units, and outputs. The MLP was trained and tested on three problems: the XOR function, a sine function approximation, and letter recognition (taken from the UCI Letter Recognition dataset). Each experiment involved tuning parameters such as learning rate, number of hidden units, and activation functions in order to achieve high accuracy and low error rates. The results documented in the output files (text files generated by java files for each of the three experiments) demonstrate the MLP's ability to generalize effectively and achieve high precision on the unseen test data.

2 Design Choices for MLP

The Multi-Layer Perceptron (MLP) class was programmed in Java thanks to its flexibility and adaptability, supporting a variable number of inputs, hidden units, and outputs. The architecture consists of a fully connected network with an input layer, hidden layer, and output layer. The weights for connections between layers are represented by W1 and W2 for the lower and upper layers, respectively, and are initialized to small random values using the randomise() method to prevent symmetry issues arising during training.

Forward propagation is performed via the forward() method, which calculates the activations of the hidden layer and then the output layer. The activation functions used are configurable, allowing the model to choose between sigmoid, linear, and softmax depending on the task. The flexibility of activation functions was achieved using an enum (ActivationFunctionType.java), allowing the MLP to adapt to different problems, such as classification (XOR problem) and regression (sine function approximation).

The backward() method involves backpropagation to calculate the errors and update weights. With backpropagation, deltas are computed for both output and hidden layers, and the weight changes (dW1 and dW2) are accumulated in batches. These accumulated weight changes are applied during the updateWeights() method, which is needed to stabilize the training process and ensure effective learning. Furthermore, the learning rate is used to control the magnitude of weight updates, and which must be fine tuned in each problem to balance learning speed with stability.

For the hidden and output layers, different activation functions were used based on the requirements of the problem. Sigmoid activation was used primarily for non-linear transformations, while linear and softmax activations were intended to be used for output layers in regression and classification problems respectively.

The TrainingExample.java file defines a structure to store input-output pairs for training, enabling consistent handling of training data across experiments. This versatility is what made Java's object-oriented design appealing for this project, making it straightforward to experiment with various network setups and activation layers by simply changing parameter values.

3 Problem 1: XOR Experiment

3.1 Description and Coding Choices

The XOR experiment involved training an MLP with 2 inputs, 4 hidden units, and 1 output to predict the XOR function. A sigmoid activation function was initially used with a learning rate of 0.8. However, this configuration converged slowly. By increasing the learning rate to 1.0, the root mean square error (RSME) improved from 5.86% to 4.28% after 2000 epochs. The most significant improvement came from changing the activation function to linear, achieving a final error of 1.25% after 2000 epochs. As demonstrated by the output file XORExperimentResults.txt, the error was consistently low after just 100 epochs, however there was little to no change in the low error after 900 epochs (below 10^{-4}.

The linear activation function proved to be optimal because after processing the hidden layer, the XOR problem becomes linearly separable. By removing the non-linearity in the output layer (from the sigmoid activation), the model was able to approximate the expected outputs more effectively.

3.2 Results

In the XOR experiment, the MLP was trained to reduce the root mean square error for each XOR input combination. Every 50 epochs, the error was evaluated as part of the training process. A final root mean square error was computed to evaluate the model's overall performance after 2000 epochs of training; the text file below also displays the expected and predicted outputs after the model has been trained.

```
XOR Experiment Results
==========

Configuration:
Number of Inputs: 2
Number of Hidden Units: 4
Number of Outputs: 1
Learning Rate: 1.0
Max Epochs: 2000
Activation Function: LINEAR

Training Error Over Epochs (Logged Every 50 Epochs):
Epoch    Error
0        0.6082357727815336
50       0.1072360332279278
100      0.05644482221123161
150      0.03473182920448676
200      0.021646105253817983
250      0.013130979278656448
300      0.008248733757571357
350      0.005607057332665432
400      0.0041012244363623565
450      0.0031729289635033424
500      0.002559002968437437
550      0.0021292498188253905
600      0.001814543565166245
650      0.0015756359091102044
700      0.0013889082959010074
750      0.0012394307027781238
800      0.0011173649531533835
850      0.0010159954899329872
900      9.305979647499673E-4
950      8.577613268766056E-4
1000     7.949664881099457E-4
1050     7.403159156890133E-4
1100     6.923551123783865E-4
1150     6.499517697585561E-4
1200     6.122121160765477E-4
1250     5.784218412964521E-4
1300     5.480036206909859E-4
1350     5.204860718250671E-4
1400     4.95480729302065E-4
1450     4.7266473445189923E-4
1500     4.517676597259172E-4
1550     4.325613657153957E-4
1600     4.148521106544485E-4
1650     3.984743525295313E-4
1700     3.832858368552019E-4
1750     3.691636708304775E-4
1800     3.560011613400035E-4
1850     3.4370524963282963E-4
1900     3.3219441589968666E-4
1950     3.213969567365609E-4

Results:
Input: [0.0, 0.0], Expected Output: 0.0, Predicted Output: 0.0001
Input: [0.0, 1.0], Expected Output: 1.0, Predicted Output: 0.9996
Input: [1.0, 0.0], Expected Output: 1.0, Predicted Output: 0.9995
Input: [1.0, 1.0], Expected Output: 0.0, Predicted Output: 0.0249

Final Root Mean Squared Error: 1.25%
```

Figure 1: Results of the XOR Experiment

The output text file above shows that the MLP accurately and successfully predicts each of the four

XOR function examples. The MLP successfully learned the XOR relationship, as evidenced by the low error numbers.

4 Problem 2: Sine Function Approximation

4.1 Description

The sine function approximation involved training the MLP with 4 inputs, 5 hidden units, and 1 output to approximate the function $\sin(x_1 - x_2 + x_3 - x_4)$. 500 vectors, each with 4 components (random numbers between -1 and 1) were generated such that the model was trained on 400 of these and then tested on the final 100 samples. A high learning rate of 0.1 initially caused instability, but reducing it to 0.01 allowed the model to learn and steadily converge on final weights. Lowering the batch size from 20 to 5 further improved accuracy due to more frequent weight updates. The final RMSE was 0.291 on the training set and 0.105 on the test set.

4.2 Results

During the experiments, the root mean square error was calculated every 500 epochs to monitor the model's progress in minimizing the error. The error was computed by comparing the model's predicted output against the expected target values for each training sample, accumulating the squared differences, and averaging them to calculate the RMSE. Each training sample was determined using one of the 400 randomly generated input vectors, consisting of four components with values between -1 and 1. After the training phase, the final training error was compared to the test error to evaluate how well the model generalized to unseen data. Additionally, 5 samples from the test set were displayed, and the predicted outputs were shown alongside the expected outputs to analyze the model's predictions.

```
Sine Experiment Results
=======================
Configuration:
Number of Inputs: 4
Number of Hidden Units: 5
Number of Outputs: 1
Learning Rate: 0.01
Max Epochs: 5000
Activation Function: LINEAR

Training Error Over Selected Epochs:
Epoch   Error
0       79.70531854643652
500     1.3988408161750026
1000    0.9902310618872627
1500    0.7567755358468317
2000    0.590811902795019
2500    0.49437699308915384
3000    0.42997521143697137
3500    0.3830126413939165
4000    0.3451407922155777
4500    0.31022547887955243
5000    0.2914805740056281

Final Training Error: 0.2914805740056281
Final Test Error: 0.10535250203705555

Sample Calculations from Test Set:
Input: [-0.9024342063882056, 0.07443566140779811, 0.026346954392452826, 0.20014645010412258], Expected: -0.9130, Predicted: -0.9049, Squared Error: 0.0001
Input: [0.21874875040415542, -0.40620825250015913, 0.5429918935609492, -0.5467471043795213], Expected: 0.9897, Predicted: 0.9139, Squared Error: 0.0057
Input: [0.8878743781769676, 0.4132807705156254, 0.10540423475698968, 0.048616545789011 09], Expected: 0.5067, Predicted: 0.5506, Squared Error: 0.0019
Input: [0.43303296272910585, -0.3001633699476 5336, -0.33679504280603934, 0.08487750176196318], Expected: 0.3065, Predicted: 0.3264, Squared Error: 0.0004
Input: [0.8378349137849483, 0.030237161789541833, 0.567841368539379, -0.8352932577157548], Expected: 0.8021, Predicted: 0.7744, Squared Error: 0.0008
```

Figure 2: Results of the Sine Function Approximation Experiment

According to the sine experiment results, the final training error was 0.291, while the testing error was 0.105. This indicates that the model successfully learned the underlying sine function without overfitting. The small discrepancy between the training and testing errors suggests strong generalization, as the model

performed well on both the training and unseen test data, demonstrating that it has learned satisfactorily. Furthermore, by simply assessing the five test examples at the bottom of the text file, it is demonstrated that the model can sufficiently approximate the sine function.

5 Problem 3: Letter Recognition

5.1 Description

The letter recognition experiment used the UCI Letter Recognition Dataset. The MLP was set to 16 inputs, 40 hidden units, and 26 outputs. Training for 2000 epochs with a learning rate of 0.1 resulted in a final test accuracy of 91.15%. Increasing hidden units from 10 to 40 significantly reduced the error, as this allows the model to account for more complexity due to the increased number of parameters compared to previous problems.

The error logged over epochs shows steady learning, and the test accuracy indicates strong generalization. Sample predictions further demonstrate the model's ability to classify letters accurately.

5.2 Results

For the letter recognition experiment, the error was recorded every 200 epochs throughout the training process. After the model was trained using 80% of the dataset, it was then evaluated on the remaining 20% of the unseen data to assess its generalization ability. The final test accuracy was logged after 2000 epochs, allowing for analysis of the model's classification performance on unseen examples.

```
Letter Recognition Experiment Results
=====================================

Configuration:
Number of Inputs: 16
Number of Hidden Units: 40
Number of Outputs: 26
Learning Rate: 0.1
Max Epochs: 2000

Training Error Over Selected Epochs:
Epoch    Error
200      0.18518083348507483
400      0.17040640355092498
600      0.15518985054491885
800      0.1463564895277696
1000     0.14258375593544836
1200     0.14485163698798498
1400     0.13603406936966356
1600     0.13762858071856798
1800     0.13471268870338582
2000     0.13446064228935012

Final Test Set Accuracy: 91.43%
```

Figure 3: Results of the Letter Recognition Experiment

With an accuracy of 91.43% after 2000 epochs, the MLP demonstrates a fairly high level of precision in classifying the testing data, indicating that the model has successfully learned the features from the

training set. This proves that the model can successfully generalize from training to testing, achieving the goal of recognizing letters with a substantial level of precision after being trained for 2000 epochs.

6 Conclusion

Because the MLP was able to successfully predict outputs across the three problems after being trained on enough data, the MLP proved to be both versatile and effective. By adjusting the learning rate, number of hidden units, and activation functions, the MLP consistently showed strong performance in both classification and regression problems. The results demonstrated the model's ability to generalize well to unseen data, thanks to the robustness of the design.

This project has taught me that an effective MLP must be both dynamic and flexible, as different problems require different approaches. For example, the XOR problem required the linear activation function, while the letter recognition problem required softmax. Another thing that I learned is that there had to be a lot of fine tuning in order to produce a successful MLP model. For example, I had to fine tune the learning rates and hidden layers for each problem several times before it was able to successfully predict the outputs. For example, for the sine function approximation problem, I tried adding more hidden units despite the model already working, and while this made the MLP predict the sine function outputs with slightly more precision, it led to overfitting, as evidenced by the poor performance when the MLP was tested against data it had not seen prior. This project also taught me just how crucial and delicate the learning rate it for each problem. While increasing it substantially in the XOR problem helped the MLP successfully classify the inputs after fewer epochs, doing this on the other two problems causes the model to "blow up" and output values that did not make any sense. Lastly, after performing the letter recognition experiment, I became interested in the performance of the MLP's runtime. After increasing the number of inputs and hidden units for this problem (since the past two problems had far fewer of these), I noticed how much longer it took to run the program, especially after I doubled or even tripled the number of hidden layers to improve the model. Specifically, each additional input or hidden unit exponentially increased the number of weight parameters and, consequently, the number of calculations required during both forward and backward propagation. With 16 inputs, 40 hidden units, and 26 outputs, the MLP needed to compute thousands of weight updates, resulting in a substantial increase in the number of matrix multiplications and gradient calculations. This highlights the trade-off between model complexity and computational efficiency, as larger and more complex neural networks and machine learning models must be able to produce fast yet reliable results.