University of Tripoli

Faculty of Engineering

Computer Engineering Department

EE 569: Deep Learning

12-12-2024

Assignment 1 part a

2200208256....................................... بدر محمد شحيم

Instructed by:

د. نوري بن بركة

Abstract

This report explains how we improved a logistic regression model by creating a custom Linear computation node. This node performs a simple mathematical operation in the model and helps calculate the gradients needed for training. We replaced parts of the existing code with this new node to make the model work more efficiently. We also introduced batching, which allows the model to process multiple data points at once, speeding up training. Finally, we tested how different batch sizes affect the model's performance by measuring the training loss for various batch sizes and plotting the results. These experiments help us understand the best ways to optimize the model for faster and more accurate training.

Introduction

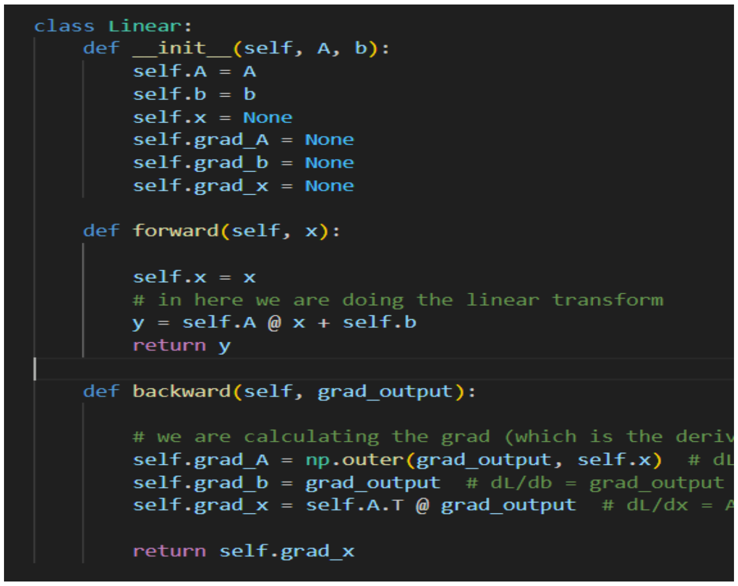
This report focuses on improving a logistic regression model by creating custom computation nodes, like the Linear node, to perform key calculations. It also introduces batching, which allows the model to process multiple data points at once for better performance. The project explores how changing the batch size affects the model's training and accuracy. All changes are made using Python libraries like NumPy, Matplotlib, and SciPy to ensure efficiency and simplicity. The goal is to better understand the logistic regression process and how batching impacts training.

Procedure and Results

A) Task 1 Implementing a Linear Computation Node

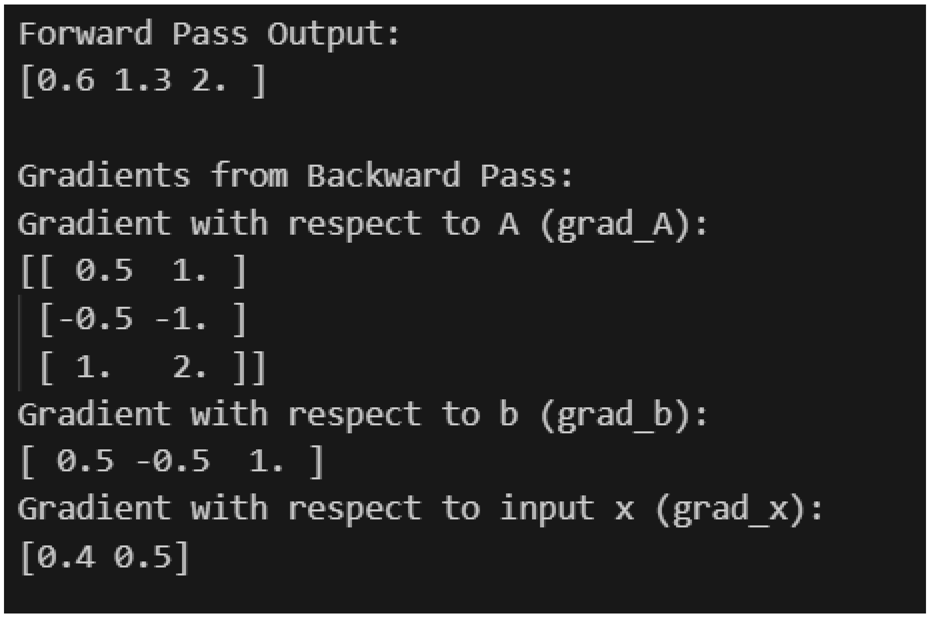
1. To implement the linear node first we need to follow the following formula for Linear Computation

$$Y = A.x + b$$



```python
class Linear:
    def __init__(self, A, b):
        self.A = A
        self.b = b
        self.x = None
        self.grad_A = None
        self.grad_b = None
        self.grad_x = None

    def forward(self, x):

        self.x = x
        # in here we are doing the linear transform
        y = self.A @ x + self.b
        return y

    def backward(self, grad_output):

        # we are calculating the grad (which is the deriv
        self.grad_A = np.outer(grad_output, self.x)  # dL
        self.grad_b = grad_output  # dL/db = grad_output
        self.grad_x = self.A.T @ grad_output  # dL/dx = A

        return self.grad_x
```

Figure 1.1: Linear Class



```
Forward Pass Output:
[0.6 1.3 2. ]

Gradients from Backward Pass:
Gradient with respect to A (grad_A):
[[ 0.5  1. ]
 [-0.5 -1. ]
 [ 1.   2. ]]
Gradient with respect to b (grad_b):
[ 0.5 -0.5  1. ]
Gradient with respect to input x (grad_x):
[0.4 0.5]
```

Figure 1.2: Results of task a with Mock Data

B) Task 2: Integrating the Linear Node into the Logistic Regression Code

We used the Sigmoid activation function and Binary Cross-Entropy (BCE) loss for this logistic regression model. The learning rate was set to 0.01, and the model was trained for 100 epochs, meaning the dataset was processed 100 times. After training, we visualized the results by plotting the data and the decision boundary. This setup helps us understand how well the model learns to classify the data.

$$\sum_{i=1}^{N} \underbrace{-y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)}_{\text{Binary Cross Entropy Loss } \mathcal{L}(\hat{y}_i, y_i)}$$

Figure 1.3: BCE logic function

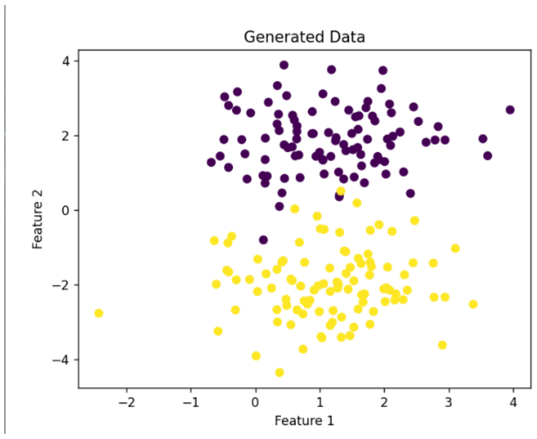


Figure 1.4: Task 2 Data Plot

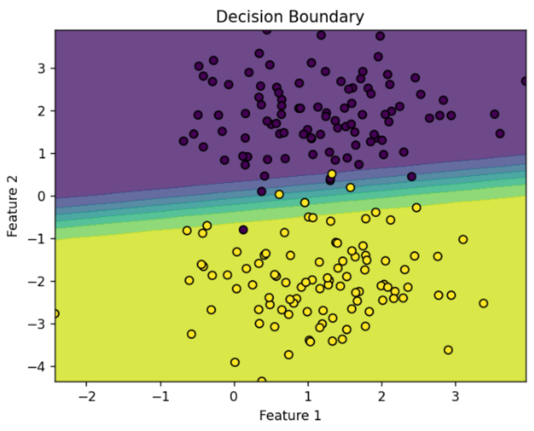
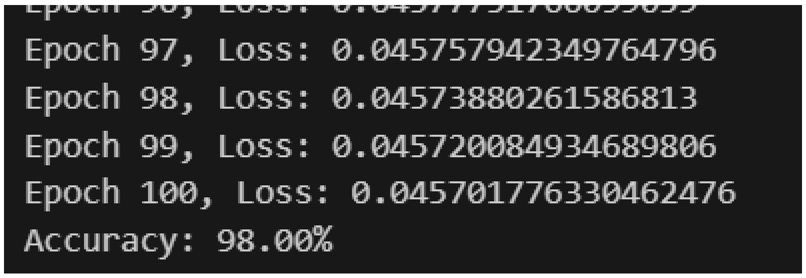


Figure 1.4: Task 2 Decision Boundary for the Data



```
Epoch 97, Loss: 0.045757942349764796
Epoch 98, Loss: 0.04573880261586813
Epoch 99, Loss: 0.045720084934689806
Epoch 100, Loss: 0.045701776330462476
Accuracy: 98.00%
```

Figure 1.4: Task 2 Results of the Data

C) Task 3: Introducing Batching

Training a neural network by processing the entire dataset at once can be inefficient for several reasons, including computational limitations and hardware constraints. To address this, we divide the dataset into smaller, manageable subsets called **batches**.

we modified the Linear class, Sigmoid class and BCE class to take in Batches as well.

D) Task 4: Investigating the Effect of Batch Size

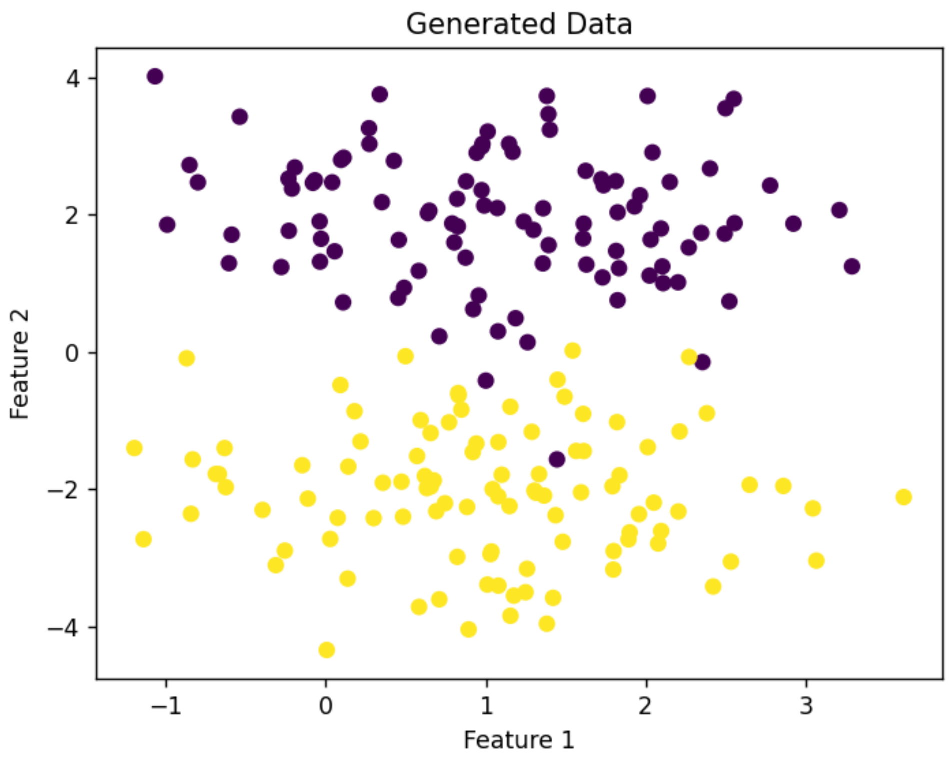


Figure 1.5: Generated Data

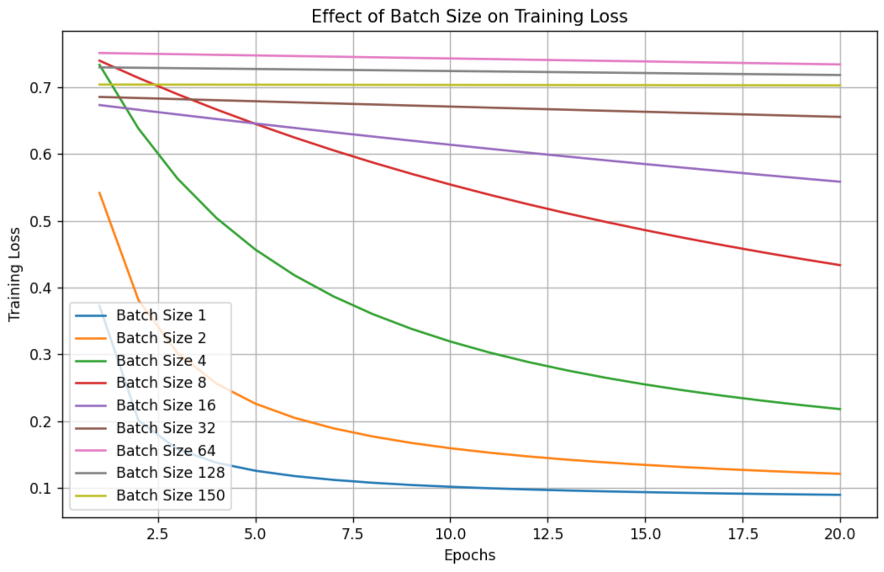
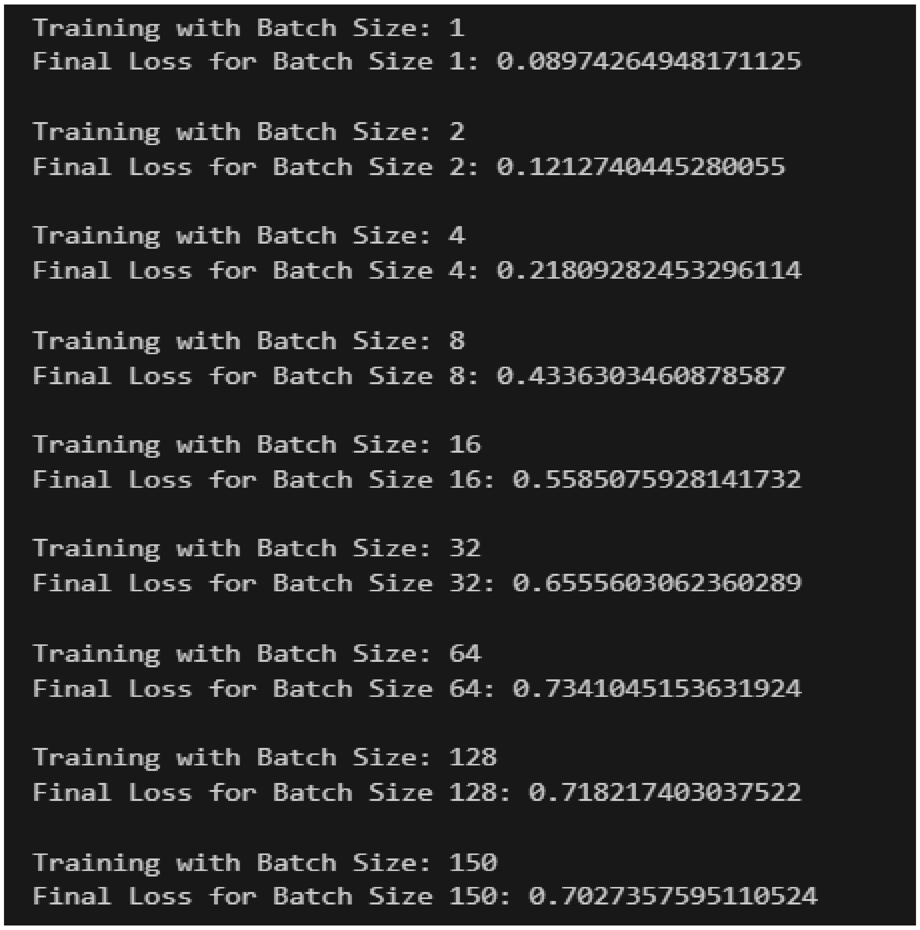


Figure 1.6: Graph Results of Task 4 and for each batch.



```
Training with Batch Size: 1
Final Loss for Batch Size 1: 0.08974264948171125

Training with Batch Size: 2
Final Loss for Batch Size 2: 0.1212740445280055

Training with Batch Size: 4
Final Loss for Batch Size 4: 0.21809282453296114

Training with Batch Size: 8
Final Loss for Batch Size 8: 0.4336303460878587

Training with Batch Size: 16
Final Loss for Batch Size 16: 0.5585075928141732

Training with Batch Size: 32
Final Loss for Batch Size 32: 0.6555603062360289

Training with Batch Size: 64
Final Loss for Batch Size 64: 0.7341045153631924

Training with Batch Size: 128
Final Loss for Batch Size 128: 0.718217403037522

Training with Batch Size: 150
Final Loss for Batch Size 150: 0.7027357595110524
```

Figure 1.6: Final Loss for each batch

Discussion

Task 1: Linear Node Implementation
We implemented a Linear node to perform y=A·x+by with forward and backward methods to compute outputs and gradients efficiently.

Task 2: Integrating the Linear Node
The Linear node replaced separate multiplication and addition nodes in the logistic regression code, improving modularity and ensuring correct functionality.

Task 3: Introducing Batching
Batching was added to process multiple inputs simultaneously. The Linear node was updated to handle matrix operations, optimizing performance and reducing computational overhead.

Task 4: Effect of Batch Size
We tested batch sizes 1,2,4,8,…1, 2, 4, 8… and observed that smaller batches provided faster updates but noisier results, while larger batches produced smoother convergence but required more memory.

Conclusion

The project demonstrated the benefits of modular design and batching in machine learning:

- The **Linear node** simplified computation.
- **Batching** improved efficiency and scalability.
- Batch size significantly impacts training speed and convergence.

This approach resulted in cleaner, faster, and more optimized logistic regression training.

Overall, the experiment provided valuable insights into the practical implementation of op-amp circuits and highlighted the importance of understanding real-world imperfections when designing and analyzing electronic systems.

References

1. Numpy Library:
   https://numpy.org/

2. Matplotlib
   https://matplotlib.org/

3. Scipy
   https://docs.scipy.org/doc/scipy/