Feedforward Neural Networks: Midterm Project Report

Alexander Bernal, Barath Kurapati, Nicholas Livingstone, Danny Metcalfe, Key Vollers

October 2024

1 Introduction

This report presents the development, testing, and analysis of a Feedforward Neural Network (FNN). The report covers the implementation of the neural network, including forward propagation, back-propagation, and gradient descent algorithms. We then test our implementation on three tasks and analyze the results: approximation of the sin function, a Van der Pol ODE system, and the MNIST handwritten digits dataset. The code for this report and its usage can be found at this LoboGit repository.

2 Implementation of the Feedforward Neural Network

2.1 Activation Functions (activation.py)

Purpose: The activation functions introduce non-linearity into the network, allowing it to learn complex patterns. Each function has a corresponding derivative for backpropagation.

Key Functions:

- `relu` and `relu_derivative`: Keeps positive values and zeroes out negatives, often used in hidden layers for efficiency.
- `tanh` and `tanh_derivative`: Maps values to the range of -1 to 1, suitable for inputs with varying scales.
- `sigmoid` and `sigmoid_derivative`: Outputs values between 0 and 1, commonly used for binary classification.
- `linear` and `linear_derivative`: A linear activation, typically used where no transformation is needed.

2.2 Feedforward Neural Network (feed_forward_neural_network.py)

Class: `FeedforwardNeuralNetwork`

Purpose: Manages the structure of the network, including forward propagation (data through the network) and backpropagation (error calculation and weight adjustment).

Key Methods:

- `__init__`: Initializes the network layers.
- `forward`: Sends input through each layer sequentially, resulting in the network's prediction.
- `backward`: Calculates errors for each layer and propagates them backward.
- `gd`: Updates weights based on computed gradients.
- `zero_grad`: Resets all gradients to zero for a new training pass.
- `train`: Repeats forward and backward passes over epochs to train the network.

2.3 Layer (layer.py)

Class: `Layer`

Purpose: Represents an individual layer within the network, handling input transformation and weight adjustments.

Key Methods:

- `__init__`: Initializes weights and activation function.
- `forward`: Computes the weighted sum of inputs and applies activation.
- `backward`: Propagates error backward and adjusts weights.

2.4 Loss Functions (loss.py)

Purpose: Provides loss derivatives for backpropagation, measuring prediction accuracy.

Key Functions:

- `mse_derivative`: Computes Mean Squared Error gradient.
- `cross_entropy_derivative`: Computes Cross-Entropy gradient.

2.5 Utility Functions (utils.py)

Function: `get_batches`

Purpose: Divides data into batches, improving learning efficiency through mini-batch training.

2.6 How it All Works Together

2.6.1 Initialization

The network is initialized by creating a `FeedforwardNeuralNetwork` object with multiple `Layer` objects. Each layer is configured with specific sizes and activation functions, establishing the neural network architecture.

2.6.2 Training (Forward and Backward Pass)

- **Forward Pass**: Input data is passed sequentially through each layer using `FeedforwardNeuralNetwork.forward(X)`. The final layer outputs the network's prediction.
- **Backpropagation**: After obtaining predictions, the `FeedforwardNeuralNetwork.backward` function calculates gradients by comparing the predictions to actual target values. This error propagates back through each layer, allowing weight adjustment.
- **Gradient Descent**: The `FeedforwardNeuralNetwork.gd` function updates each layer's weights based on gradients.

2.6.3 Mini-Batch Training

The `utils.get_batches` function organizes data into mini-batches, allowing the network to incrementally adjust weights. This helps improve gradient stability.

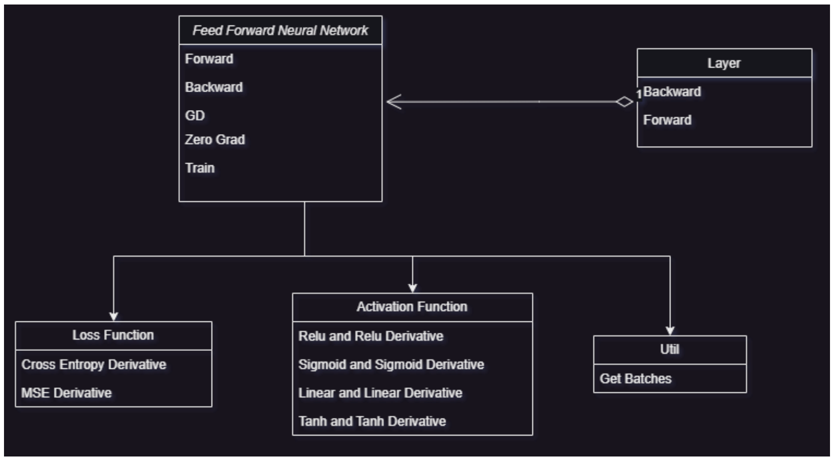

Figure 1: Class Flow Diagram of Feed Forward Neural Network

3 Case Studies

3.1 Case 1: Simple Regression

This experiment trains the FNN to approximate the function $y = \sin(x)$. Random samples of $x \in [-3, 3]$ are generated, and the network is trained using the Mean Squared Error (MSE) loss function. Figure 2 shows the network's predictions compared to the true sine function.

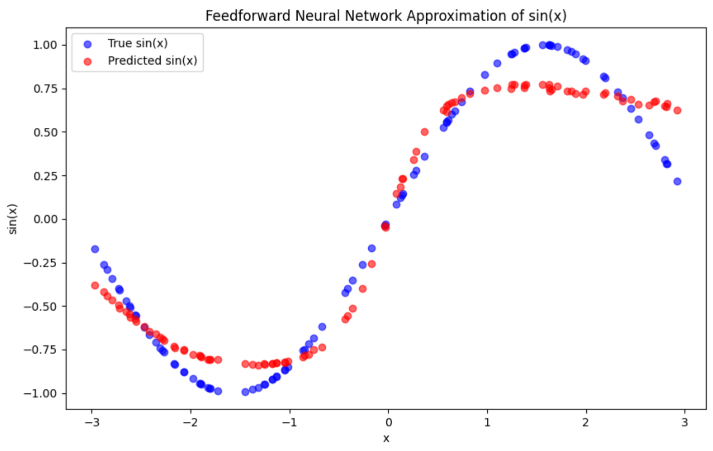

Figure 2: Sine function approximation using the FNN

3.2 Case 2: Data-Driven Model for Van der Pol System

In this task, the FNN is trained to approximate the one-step reachability relation for a Van der Pol system, using mini-batch stochastic gradient descent (SGD). Samples are generated from the system's state space ($x_1, x_2 \in [-3, 3]$). In this test case, we initially encountered issues with the networks numerical stability. Early attempts to train the network resulted in both underflow and overflow or very poor results as seen in Figure 3. To handle this, we implemented clipping of weights during gradient descent, normalized weight initialization, and derivative threshold to prevent values from becoming too large and too small.

We then trained the model with various batch sizes to show impacts to convergence and accuracy. We split the generated data into a 80-20 train and test set and computed the mean loss of the batch and the loss of the test after running all batches for 1000 epochs. The results are shown in Figure 4. In

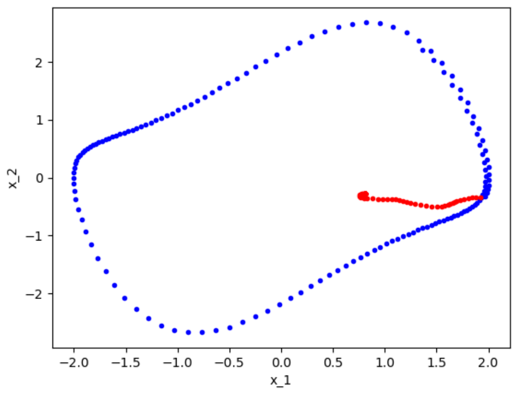

Figure 3: Early Results of Van der Pol System Prediction

general, the losses did not change dramatically. However we can see that the variance of the train loss was much smaller when using a large batch size, however with a batch size of 128, the model begins to appear to over fit slightly. A batch size did result in the smallest overall loss, but exhibited a larger variance in training, it additionally took longer to train than the other batch sizes.

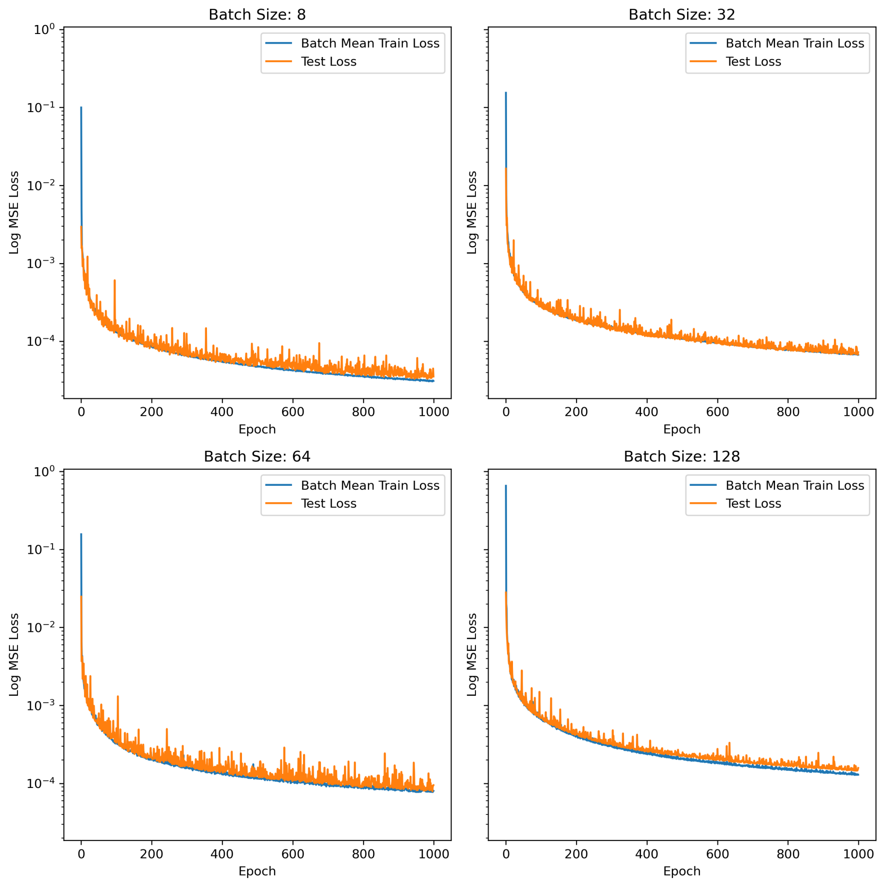

Figure 4: Van der Pol Batch Size Comparison

Training with a batch size of 32 and a learning rate of 0.1, the network was able to achieving close alignment with the original ODE trajectories (Figure 5.

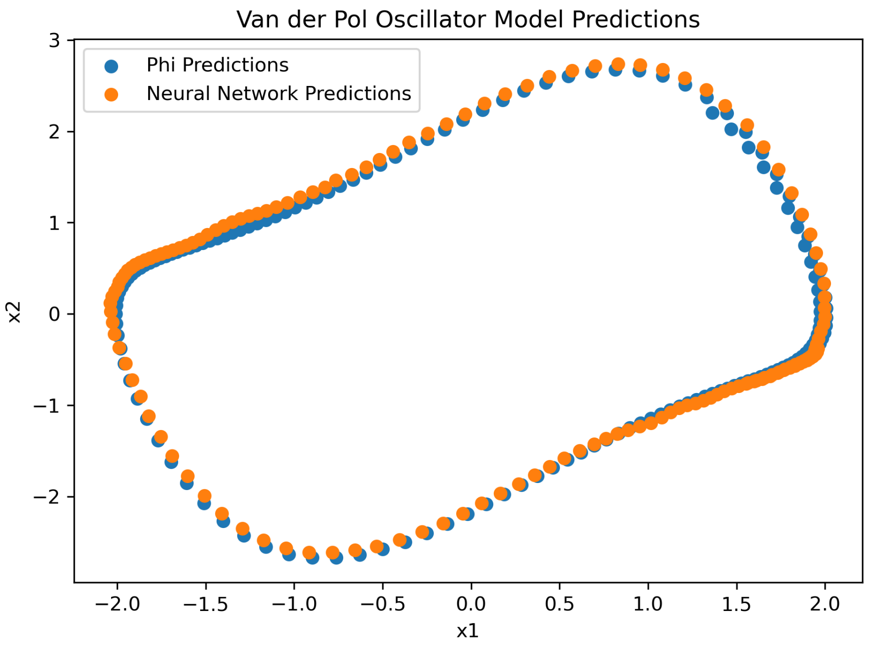

Figure 5: Prediction with numerical stability

3.3 Case 3: Handwriting Numbers (MNIST Dataset)

The FNN was trained to recognize handwritten digits from the MNIST dataset. The output layer uses the LogSoftmax activation, with NLLLoss employed as the loss function. Mini-batch SGD is utilized for training. As in case 2, we compared the impacts of batch size on training. The results are shown in Figure 6. We can see that these experiments gave much more distinctive results than the Van der Pol prediction. It's clear that the model here is much more likely to overfit. This is emphasized when training with a batch size of 8, as the test loss appears to converge to a single value. When training with a batch size of 32, we were able to achieve a near perfect accuracy on the test set as shown in Figure 7.

4 Conclusion

In this report, we implemented a Feed Forward Neural network in python and tested it on three varied test cases. Our results show that a basic neural network is effective at modeling non-linear functions, ODE systems, and classifying image data. This project provided hands-on experience in dealing with the subtle nuances of implementing the mathematically foundations of a Neural Network and effective code design. Future work could involve experimenting with deeper networks, alternative activation functions, or advanced optimization methods.

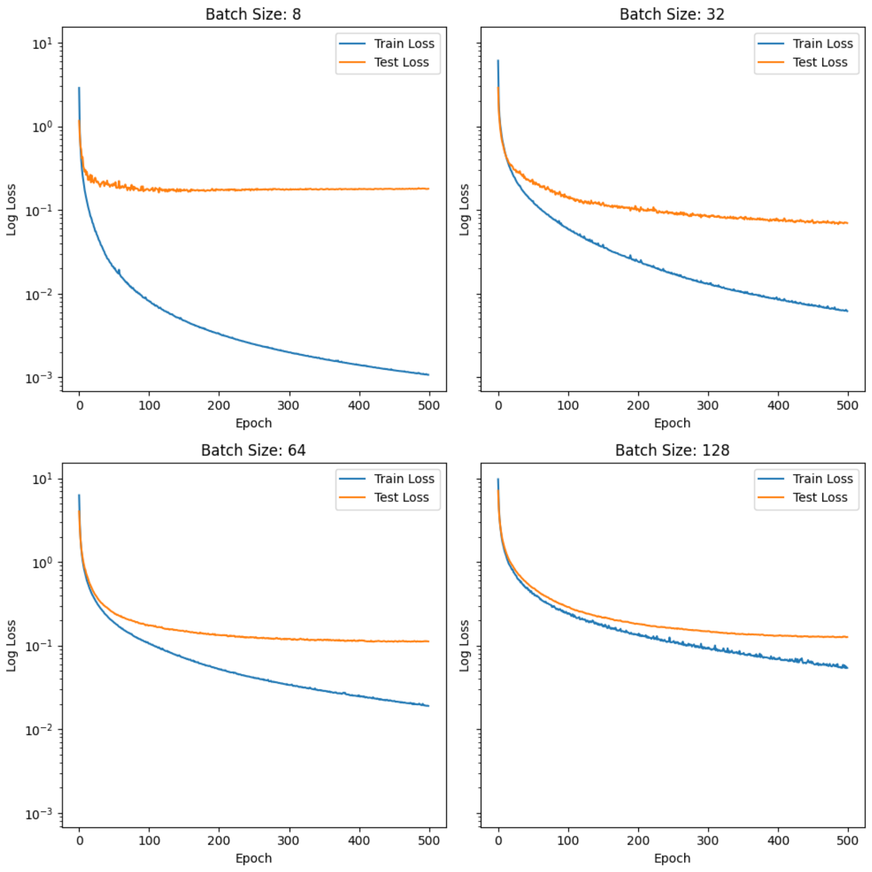

Figure 6: Batch Size Comparison for training on MNIST Digits

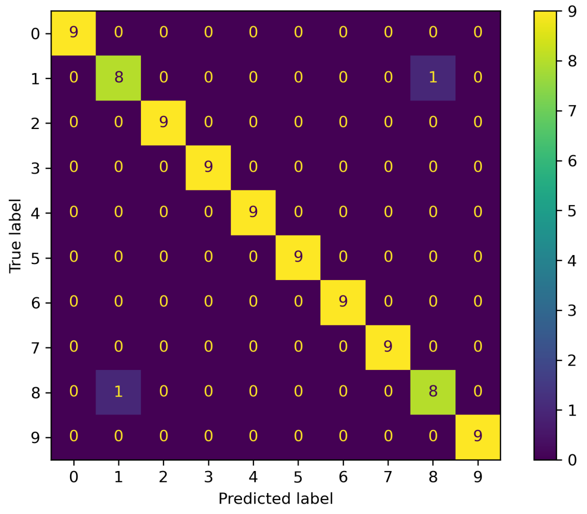

Figure 7: Confusion Matrix of MNIST Digits Test Set

5 Contributions

Key Vollers Built second draft of Feed Forward Network System. Finalized back propagation. Put together case 1 script. Wrote Section 2 of essay. Implemented batched experiment of case 3.

Danny Metcalfe Worked on changing layer structure and activation and loss function setup with the neural network. Also helped with fixing backpropogation and gradient decent and setup for case 3 with softmax.

Nicholas Livingstone Implemented mini-batching and capability for model to handle batching. Designed batch experiments for case 2 and 3 and wrote some of the analysis for each. Implemented NLLLoss and softmax activation. Added numerical stability code. Conducted code review and provided design suggestions throughout development.

Alexander Bernal Made a first draft, I developed the core structure of a feedforward neural network by implementing a modular design with two main components: a `Layer` class to represent individual neural layers and a `FeedforwardNeuralNetwork` class to manage the full sequence of layers. The network was designed to allow straightforward adjustments to layer sizes and includes methods for forward propagation, backpropagation, and gradient descent to support training. My design emphasizes flexibility and clarity, allowing for easy adaptation to different neural network configurations.

Barath Kuarpati Was responsible for implementing the backpropagation algorithm, ensuring the correct computation of gradients for weight updates across all layers. This involved designing the logic to incorporate activation function derivatives and loss derivatives, as well as efficiently organizing intermediate outputs for backward passes. Additionally, Barath developed customized testing methods to validate the network's functionality beyond the provided test cases, particularly focusing on debugging and verifying gradient descent convergence. Barath also took the lead in preparing the entire project report, documenting the implementation, detailing the backpropagation logic, and summarizing the observations and results from the experiments.