CS 4/591: Neural Network

Midterm Project: Training Feedforward Neural Networks

Jyrus Cadman Robert McCourt Bethany Peña Gabriel Urbaitis

25 October 2024

Contents

1 Introduction

This project focuses on the implementation of a feed-forward neural network (FNN), the first of the multi-layer networks we will explore in this course. Th objective of our project is to develop an FNN class in Python that is capable of performing supervised learning tasks, such as regression and classification.

Our implementation includes developing a modular object-oriented design scheme to represent the network and its layers, and to maintain the data required during training. Our network implements forward propagation to compute the values of weights and outputs of our model, and the backward propagation algorithm to calculate the gradient of our losses so we can update our weights using the gradient descent and stochastic gradient descent algorithms.

Our network is tested on three distinct cases so we can fully evaluate and observe its behavior. We first test the ability of our network perform a simple regression. Then, we model the one-step reachability of the Van Der Pol system. Lastly, we explore image classification using the MNIST dataset of handwritten digits. Through testing we began to explore different strategies for initializing our model weights, and gradient clipping to handle exploding gradients.

This report provides details of the design and implementation of the feed forward neural network in Python, in addition to details about our testing and analysis of the results obtained. Through this project we were able to develop our understanding of multi-layer neural networks, the algorithms that perform the learning, and how to troubleshoot certain problems in training and testing a neural network.

2 The Feedforward Neural Network

Our implementation adopts a modular, object-oriented approach to neural networks through two primary classes: `FNN` and `Layer`. This design philosophy emphasizes flexibility and extensibility, allowing for the construction of neural networks with varying architectures and depths. The `FNN` class serves as the container for the network, managing a collection of `Layer` objects that represent each layer of the neural network.

Each layer in the network is encapsulated as a `Layer` object, maintaining its own state and operations. The layer's state includes a weight matrix that inherently incorporates bias terms by extending the input dimensionality (shape: $n_{\text{input}} + 1 \times n_{\text{output}}$). This implementation detail elegantly handles bias terms by augmenting input data with ones using NumPy's horizontal stack operation:

```
np.hstack([X, np.ones((X.shape[0], 1))]).
```

Each layer also manages its activation function and the corresponding derivative, essential components for both forward- and back-propagation.

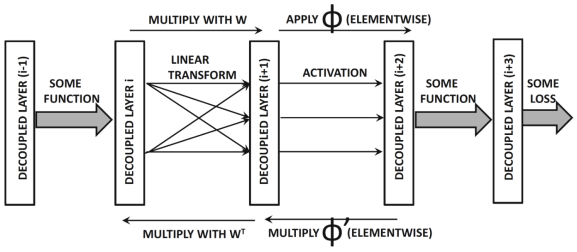
Additionally, the implementation supports a rich set of activation functions, providing the flexibility needed for various types of neural network applications. These functions include the commonly used ReLU (Rectified Linear Unit), sigmoid, and tanh, as well as more specialized functions like the identity function, sign function, hard tanh, and log softmax. This versatility is demonstrated effectively across our test cases. For instance, in our regression task (`case1.py`), we employ ReLU activation in the hidden layers with an identity function in the output layer, while our classification task (`case3.py`) utilizes ReLU in the hidden layers with log softmax in the output layer.

The modular nature of our implementation becomes particularly valuable when constructing networks of varying complexity. Through simple layer stacking, we can create networks of arbitrary depth, each layer potentially using different activation functions and having different dimensions. This flexibility allows the

network to be adapted for a wide range of applications, from simple regression tasks to complex classification problems, as evidenced by our diverse test cases.

2.1 Backpropagation

Backpropagation is a process used in training neural networks that calculates the gradient of the loss function with respect to the weights of the network. It is based on the chain rule of calculus, and starts with the derivative of the loss function with respect to the output layer's activation.



Function	Type	Forward	Backward
Linear	Many-Many	$\overline{z}_{i+1} = W\overline{z}_i$	$\overline{g}_i = W^T\overline{g}_{i+1}$
Sigmoid	One-One	$\overline{z}_{i+1}$ =sigmoid($\overline{z}_i$)	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (\overline{1} - \overline{z}_{i+1})$
Tanh	One-One	$\overline{z}_{i+1}$ =tanh($\overline{z}_i$)	$\overline{g}_i = \overline{g}_{i+1} \odot (\overline{1} - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	One-One	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	One-One	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Many-One	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	Anything	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T\overline{g}_{i+1}$ J is Jacobian (Equation 2.21)

Figure 1: Vector Centric View of Backpropagation

Moving backward, layer by layer, the derivative vector from the following layer is multiplied by the derivative of the activation function element-wise (see Figure 1), dL/dactivation, then the dot product is taken of it and the transpose of the weights (also Figure 1), which is passed to the previous layer where the process begins again. However, before moving back to the next layer, the dot product of the transpose of the inputs of the layer and dL/dactivation is taken to calculate the derivative of the loss with respect to the layer's weights. This vector is pre-pended to the gradient matrix, which by the end of the process, after the first layer is finished, is the same size as the weight matrix. As part of Regular Gradient Descent, the learning rate is multiplied by the gradient matrix and the result is subtracted from the previous weight matrix as part of the weight update step.

2.1.1 Backpropagation implementation details (methods, calculations etc.)

Backpropagation occurs globally in the FNN's `backward` method:

```
def backward(self, y, y_pred, loss_func='mse'):
    if loss_func == 'mse':
        dL_dout = 2 * (y_pred - y) / y.shape[0]
    elif loss_func == 'nll':
        dL_dout = y_pred - y
    gradients_W = []
    # Proceeding backward through the layers, add each new calculation to the front
```

```
        # to create the gradients array
        for layer in reversed(self.layers):
            grad_W, dL_dout = layer.backward(dL_dout)
            gradients_W.insert(0, grad_W)
        return gradients_W
```

and locally in the layer's `backward` method:

```
    def backward(self, dL_dout):
        dL_dout = np.nan_to_num(dL_dout)
        activation_deriv = self.activation_deriv(self.z)
        dL_dout *= activation_deriv
        # partial derivative of the loss w.r.t. the weights
        grad_W = np.dot(self.input_data.T, dL_dout)
        # accumulation of partial derivative of the loss for each layer
        dL_din = np.dot(dL_dout, self.weights.T)

        # Remove the bias
        dL_din = dL_din[:, :-1]

        grad_W = np.clip(grad_W, -3, 3)

        return grad_W, dL_din
```

In the layer's `backward` method, the activation derivative (`activation_deriv`) is calculated based on whatever activation function was chosen for the layer. As described above, this is multiplied by the derivative passed in from the following layer element-wise (dL_dout *= activation_deriv), and then the result is used in the calculation for the derivative of the loss with respect to the weights (`grad_W`) used to concatenate the gradient matrix layer by layer. That same result is also used in the calculation of the partial derivative of the loss for each layer (`dL/din`) which is accumulated from all the following layers and is passed to the previous layer.

Additionally, because the bias was handled as an extra column vector added to each layer in the forward pass, these are removed in the backward calculations as the bias is not pertinent. Gradient clipping and NumPy's `nan_to_num` method are used to restrain exploding gradients when they occur.

In the FNN's `backward` method, the initial output gradient is calculated based on the specified loss function, the empty gradient matrix is initialized, and then to begin calculation of the gradient matrix, the weight matrix is reversed to begin at the last layer. Looping through the layers in reverse, the accumulated partial derivative of the loss for each layer is updated through a single pass through the layer's backward method for each iteration, and the partial derivative of the loss with respect to the weights, which is also returned from the layer's backward method, is pre-pended to the gradient matrix to build a gradient matrix that lines up with the weight matrix. We pre-pend because we are building the gradient matrix starting with the last layer and ending with the first.

2.2 Mini-Batch Stochastic Gradient Descent (SGD)

The FNN implementation includes both standard gradient descent and mini-batch SGD, with the latter being more sophisticated and practical for large datasets like those in Cases 2 and 3.

2.2.1 Mini-Batch SGD Implementation details (methods, calculations etc.)

Observe our core SGD method:

```
def sgd(self, X, y, batch_size, learning_rate, loss_func='mse'):
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices)

    for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
        batch_indices = indices[start_idx:start_idx + batch_size]
        X_batch = X[batch_indices]
        y_batch = y[batch_indices]
```

Its key features include:

- Random shuffling of data indices for each epoch.
- Support for different loss functions (MSE and NLL)
- Batch processing for memory efficiency
- Integrated weight updates within the batch loop

The implementation supports two primary loss functions: Mean Squared Error (MSE) and Negative Log-Likelihood.

Mean Squared Error (MSE)

Recall that for predicted values $\hat{y}$ and true values y, MSE is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

In our implementation, the gradient of MSE with respect to the network output $\hat{y}$ is:

$$\frac{\partial \,\text{MSE}}{\partial \hat{y}} = \frac{2}{n}(\hat{y} - y)$$

This is exactly what we see in the implementation's `backward` method. The factor of 2 comes from the derivative of the square term, and the division by `y.shape[0]` normalizes by the batch size.

Negative Log-Likelihood (NLL)

Recall that for a classification task with C classes, given predicted probabilities $\hat{y}$ and true labels y (one-hot encoded):

$$\text{NLL} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

Additionally, the implementation cleverly combines NLL with log-softmax activation in the output layer.

This combination implements the common softmax + NLL loss pattern used in classification tasks. The log-softmax activation:

$$\log(\text{softmax}(z_i)) = z_i - \log\left(\sum_j \exp(z_j)\right)$$

has better numerical stability than separate softmax and log operations.

3 Testing

3.1 Weight Initialization

Due to the strong influence weight initialization has on the model's training and eventual accuracy, we experimented with some different ways to initialize the weights for each case. As a general strategy, we initialized all the weights to values uniformly selected at random from values between -1 and 1. For our test cases using the data-driven model (see Section 4.2) and the handwritten number image identification (see Section 4.3) we experimented with scaling these initial values and observed improvements in performance.

For both cases we tested scaling the weight initialization by 0.5 and improved results in both test cases. We hypothesize that the improved performance must have been a result of initializing the weights in a way that we are closer to the global minimum of our loss function. We did not perform an extensive exploration or further analysis, which could be interesting for future work.

3.2 Handling Exploding Gradients

During out initial training of the handwritten number image identification, we observed some unexpected behavior, which we later identified as being a result of exploding gradients. We observed that our model was not updating its performance, and upon investigating, realized that after a few epochs, the gradients overflowed and were converted to NaNs. To alleviate this problem, we decided to explore using the gradient clipping technique.

Although there exist more advanced methods for determining a gradient clipping threshold, we took a naive approach, due to time constraints, and set a threshold of -3, and 3. This greatly improved the performance of our model. We did not perform an extensive exploration or further analysis, however, this could be interesting for future work.

3.3 Simple Regression

Our task was to train our feedforward neural network to approximate the *sine* function over the rane of $[-3, 3]$. The objective of this testing task was to produce a model capable of accurately predicting the *sine* values for a given input x

3.3.1 Neural Architecture

The neural network consists of three layers. The input layer contains 1 node for the input x dimension. The first hidden layer has 25 nodes and utilizes the ReLu activation function and its derivative to capture the non-linear characteristics of the sine function. The output layer consists of 1 node that employs the identity activation function, production continuous output values representing the predicted *sine* value.

3.3.2 Training

We generated random samples of x uniformly distributed in the range of $[-3, 3]$ and computed their corresponding *sine* values to create our training dataset.

To train the mdoel, we used the Mean Squared Error (MSE) as our loss function, which measures the average squared distance between the predicted values and the actual *sine* values.

3.3.3 Forward Propagation

During forward propagation, we computed the output of each layer in sequence, applying the activation functions to obtain the final predicted values. This invloved calculating the linear combination of inputs and weights for each neruron, as well as applying the activation function (ReLU for hidden layers, and identity for the output layer) to obtain the neuron's output.

3.3.4 Backward Propagation

We implemented the backward propagation algorithm to compute the gradients of the loss function concerning the weights associated with the edges of the model. This process included calculating the gradient of the loss function with respect to the output, and propagating these gradients backward through the network, adjusting the weights based on the learning rate and calculated gradient.

3.3.5 Optimization

We trained the model for 1000 epochs with a learning rate of 0.01. The training process iteratively updates the weights of the network's edges based on the calculated gradients of the loss function, converging toward an optimal solution.

3.3.6 Evaluation

After training, we evaluated the model's performance by comparing the predicted sine values against the true values. We visualized the results using Matplotlib, plotting both the true data points and the FNN's predictions. This visual assessment allowed us to gauge the accuracy of our approximation and confirm that the model effectively learned the underlying pattern of the sine function.

3.4 Data-Driven Model

Our task was to train our feed forward neural network to approximate the one-step (0.5 s) reachability relation of the Van der Pol system whose ODE is

$$\dot{x}_1 = x_2, \quad \dot{x}_2 = -x_1 + (1 - x_2^2)x_2$$

The goal of this testing task is to produce a relation in our network as similar to the Van der Pol system.

3.4.1 Neural Architecture

The neural network consists of four layers. The first, input layer contains 2 nodes, one for x_1 and one for x_2. We have two hidden layers, each with 64 input and output nodes. Our input and hidden layers use the

RELU activation function. Our final, output layer, has 2 nodes and uses an identity function. Our output contains predicted next state values for x_1 and x_2 of the Van der Pol oscillator.

3.4.2 Training

We compiled the batched data from the Pytorch data loader into numpy arrays for training. Because our output layer consisted of 10 nodes, we one-hot encoded the labels in our data.

To train, we used a negative log likelihood function and mini-batch stochastic gradient descent as our optimizer. We trained over 100 epochs using a learning rate of 0.001 and a mini-batch size of 25 samples.

After generating X and Y as PyTorch tensors, we convert them to Numpy arrays. (X_np and Y_np) To train, we used a mean squared error loss function and mini-batch stochastic gradient descent as our optimizer. We trained over 100 epochs using a learning rate of 0.01 and a mini-batch size of 25 samples.

3.5 Handwriting Numbers

For Case 3, our task was to train the feed-forward neural network to recognize handwritten numbers from the MNIST dataset. The MNIST dataset is a collection of handwritten digit images, 0-9. We obtained the data using PyTorch. Our primary objective for this case was to train our neural network with the optimal set of hyperparameters, such as learning rate, epochs, and batch size, to correctly identify the handwritten digit in a given image.

3.5.1 Neural Architecture

The neural network consists of four layers. The first, input layer contains 784 (28×28) nodes for each of the pixels in the image. We have two hidden layers, each with 64 input and output nodes. Our input and hidden layers use the RELU activation function. Our final, output layer, has 10 nodes and uses a log-softmax function. Our output contains the log likelihood that our input belongs to each class (digits 0-9).

3.5.2 Training

We compiled the batched data from the Pytorch data loader into numpy arrays for training. Because our output layer consisted of 10 nodes, we one-hot encoded the labels in our data.

To train, we used a negative log-likelihood loss function and mini-batch stochastic gradient descent for our optimizer. We obtained our best results training the network over 100 epochs, with a learning rate of 0.001 and a mini-batch size of 25 samples.

4 Results

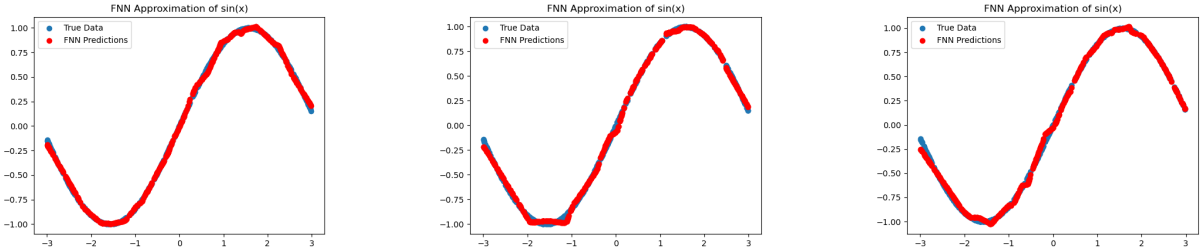
4.1 Simple Regression

Due to the influence of weight initialization on the final training and testing results, we compare the results of three separate runs to understand what the "average" behavior of our model is. Table 1 contains the initial and final mean squared errors for three separate runs. We plot the corresponding predictions against the true value in Figure 2.

As can be observed in the table, the initial mean squared error varies widely by the initial weights. We are able to achieve a very small final mean squared error, less than 0.001, for all three runs.

Iteration	Initial MSE	Final MSE
1	24.26	0.00037
2	67.13	0.00058
3	4.73	0.00071

Table 1: MSE for three different iterations.

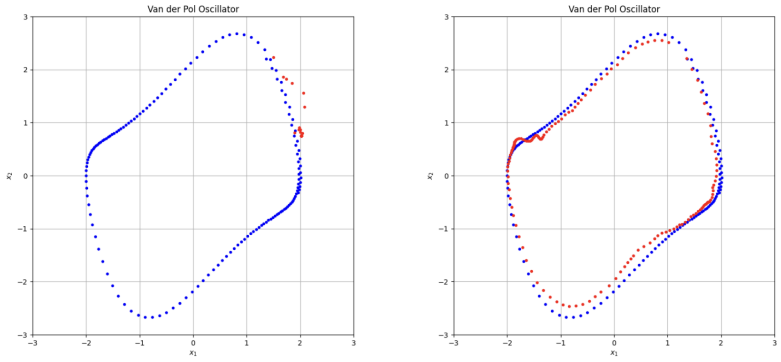


((a)) Run 1 ((b)) Run 2 ((c)) Run 3

Figure 2: Comparing results of different runs to observe impact of initialization on training.

4.2 Data-Driven Model

In the data-driven model, we were asked to use a mini-batch stochastic gradient descent for optimization. To observe the impact the optimization method has on our training, we compared the results of training with the standard gradient descent algorithm, and with stochastic gradient descent. Figure 3 compares the results of training using the two different algorithms. As can be observed, the stochastic gradient descent model results in markedly better predictions.



((a)) Regular Gradient Descent ((b)) Stochastic Gradient Descent

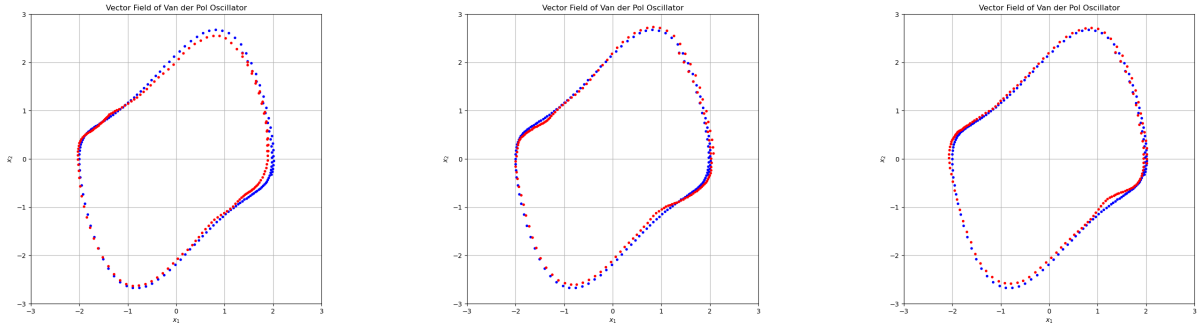
Figure 3: Comparing results of different runs to observe impact of optimization method on training.

We compare the results of three separate runs to understand what the "average" behavior of our model is. Table 2 contains the initial and final mean squared errors for three separate runs. We plot the corresponding predictions against the true value in Figure 4.

As can be observed in the table, we were able to achieve a significant decrease in the mean squared error for all three runs. We also achieved similar magnitudes of error across all three iterations.

Iteration	Initial MSE	Final MSE
1	0.0057	7.17e-05
2	0.0050	5.46e-05
3	0.0058	5.70e-05

Table 2: MSE for three different iterations.



((a)) Run 1 ((b)) Run 2 ((c)) Run 3

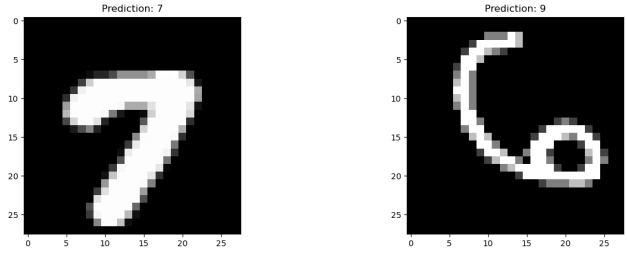
Figure 4: Comparing results of different runs to observe impact of initialization on training.

4.3 Handwriting Numbers

Table 3 compares the initial accuracy and final accuracy of our model across three separate runs. Although we started with a low, approximately 10% accuracy, we were able to achieve a fairly high accuracy at approximately 90% of correct classifications.

We also compared the rate of accuracy improvement across the 100 epochs that we trained our model, seen in figure 6. All three runs had similar rates of growth, and appeared to converge to similar values.

Figure 5 gives an example of a correct and incorrect classification result.



((a)) Correctly Classified ((b)) Incorrectly Classified

Figure 5: Examples of classification results.

Iteration	Initial Accuracy	Final Testing Accuracy
1	11.97%	90.71%
2	9.61%	90.60%
3	5.06%	89.90%

Table 3: Accuracy for three different iterations.

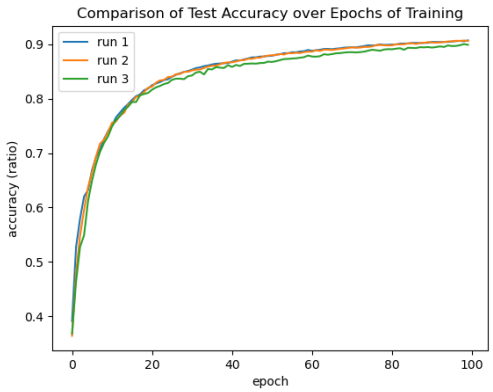


Figure 6: Comparing the test accuracy over epochs of training for three different runs.

5 Discussion

Overall, in our experiments we achieved relatively good training success and began to explore different techniques for troubleshooting and improving our models.

In the simple regression test case we observed the significant impact of the weight initialization in the initial MSE. However, we were able to achieve consistent final MSEs across different runs.

Testing the data-driven model highlighted the effectiveness of the stochastic gradient descent method compared to the standard gradient descent model. We hypothesize that the optimization step is performed on a random variety of samples, which may make it more robust compared to the standard gradient descent, and less likely to overfit.

We also experimented with initializing our weights to a smaller range, -0.5,0.5, which improved our model. A consistent theme we observed is the importance of weight initialization.

The task of identifying handwritten numbers from the MNIST data was the most complex, and led us to explore methods for handling exploding gradients. Prior to implementing gradient clipping, we noticed our model failed to learn. After we added the gradient clip, our model was able to achieve higher than expected accuracy in training. We implemented a rather naive approach to gradient clipping, but believe it would be worth further exploration so we can understand how best to incorporate this method in our model.

6 Contributions

Initially, in the first half of the project, our team contributions were divided evenly across the FNN and Layer classes. This was a straightforward process once we decided how we wanted to structure the neural network. Once we got to the testing stage, our responsibilities became mixed and our contributions crossed over with each other.

Jyrus: Jyrus was initially responsible for setting up and defining the structure of the project, including setting up the Git repo and the report. He then transitioned to implementing both the standard and stochastic gradient descent algorithms in the `FNN` class. Additionally, Jyrus took inspiration from Bethany's contributions, as well as `handwritingnumbers.py`, and assisted testing for the most optimal hyperparameters in Case 3.

Robert: Robert was responsible for developing the initial object-oriented design diagram, as well as updating it to reflect the current state of the model with the help of Jyrus, Bethany and Gabe. This diagram provides high-level insight into the instances (objects) and their relationships, illustrating how data will interact with the system. Robert also helped implement the first case, simple regression, using the structure (methods, parameters etc.) defined by Gabe and Bethany. He consulted Professor Chen on the best approach for bias absorption into the weight matrix, translating this guidance into effective code. Robert also implemented activation functions and their derivatives, except for the log-softmax derivative, which was completed by Jyrus. Additionally, he addressed challenges in the layer class related to forward and backward propagation, resolving issues with bias handling and dimensionality.

Bethany: Bethany created the initial scripts to setup and run the three test cases and generate plots to show the training results. She also assisted the rest of the team with designing the neural network so that the bias was absorbed. When we faced challenges with mismatching sizes in some of the test cases, Bethany helped identify and fix the problems. When faced with unexpected behavior during the handwritten numbers testing, Bethany identified the exploding gradient issue and proposed using gradient clipping as a solution. Gabe initially experimented with scaling the weight initialization and Bethany helped identify how the weight scaling impacted performance.

Gabriel: Gabriel helped define the project structure by specifying the input parameters for each function. He implemented both backward functions for Backpropagation, debugging them with the help of Robert. He, like Jyrus, identified the need for one-hot encoding in the Handwriting Numbers test, and plotted the Van der Pol set with Bethany's data preparation. He helped handle exploding gradients with the rest of the team and worked to find optimal hyperparameters for Case 2 (Van der Pol).

7 Conclusion

In this project, we successfully designed and implemented a feedforward neural network, the backpropagation algorithm, gradient descent, and mini-batch stochastic gradient descent. We also demonstrated its ability to perform supervised learning tasks such as regression and classification using three different datasets. For all test cases, our model was able to achieve decent performance, which we was partly a result of a naive exploration of different strategies to initialize our weights, and address exploding gradients. Overall, this project allowed us to further develop our understanding of neural network fundamental concepts and algorithms.

Appendix

```python
import numpy as np

from layer import Layer

class FNN:
    """
    A Feed-Forward Neural Network.
    """

    # Initialize the network with a list of layers
    def __init__(self, layers):
        self.layers = layers

    # Perform forward propagation through all layers
    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X

    """
    Calculate gradients for all layers.
    X: Input data
    y: True labels
    y_pred: Predicted output from the forward pass
    loss_func: Loss function ('mse' or 'nll')
    """
    def backward(self, y, y_pred, loss_func='mse'):
        if loss_func == 'mse':
            dL_dout = 2 * (y_pred - y) / y.shape[0]
        elif loss_func == 'nll':
            dL_dout = y_pred - y
        gradients_W = []
        # Proceeding backward through the layers, add each new calculation to the front
        # to create the gradients array
        for layer in reversed(self.layers):
            grad_W, dL_dout = layer.backward(dL_dout)
            gradients_W.insert(0, grad_W)
        return gradients_W

    # Update weights and biases using gradient descent
    def gd(self, gradients_W, learning_rate):
        for layer, grad_W in zip(self.layers, gradients_W):
            layer.weights -= learning_rate * grad_W


    def sgd(self, X, y, batch_size, learning_rate, loss_func='mse'):
        indices = np.arange(X.shape[0])
        np.random.shuffle(indices)

        for start_idx in range(0, X.shape[0] - batch_size + 1, batch_size):
            batch_indices = indices[start_idx:start_idx + batch_size]
            X_batch = X[batch_indices]
            y_batch = y[batch_indices]

            # Forward pass
            y_pred = self.forward(X_batch)

            # Backward pass
            gradients = self.backward(y_batch, y_pred, loss_func)

            # Update weights
            for layer, gradient in zip(self.layers, gradients):
```

```
                layer.weights -= learning_rate * gradient

    # Train the network using forward and backward propagation
    def train(self, X, y, learning_rate, epochs):
        for _ in range(epochs):
            y_pred = self.forward(X)
            gradients_W = self.backward(y,y_pred)
            self.gd(gradients_W, learning_rate)
# Train the network using stochastic gradient descent
    def trainsgd(self, X, y, learning_rate, epochs, batch_size, loss_func='mse'):
        for epoch in range(epochs):
            self.sgd(X, y, batch_size, learning_rate, loss_func)

            # Calculate and print loss for monitoring
            y_pred = self.forward(X)
            loss = self._calculate_loss(y, y_pred, loss_func)
            print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss}")

    def _calculate_loss(self, y, y_pred, loss_func):
        if loss_func == 'mse':
            return np.mean((y_pred - y) ** 2)
        elif loss_func == 'nll':
            return -np.mean(y * np.log(y_pred + 1e-8))
        else:
            raise ValueError("Unsupported loss function")
```

Listing 1: `fnn.py`

```
import random

import numpy as np

class Layer:
    """
    A layer in the Feedforward Neural Network (FNN).
    """

    # Randomly initialize weights and biases
    def __init__(self, n_input, n_output, activation='relu'):
        random.seed(2400)
        self.weights = np.random.uniform(-1, 1, (n_input+1, n_output))
        self.activation_function = activation
        self.n_input = n_input

    def forward(self, X):
        X = np.hstack([X, np.ones((X.shape[0], 1))])

        self.z = np.dot(X, self.weights)
        self.a = self.activate(self.z)
        self.input_data = X

        return self.a

    # Activation functions
    def activate(self, z):
        activations = {
            'relu': lambda z: np.maximum(0, z),
            'sigmoid': lambda z: 1 / (1 + np.exp(-z)),
            'id': lambda z: z,
            'sign': lambda z: np.sign(z),
            'tanh': lambda z: np.tanh(z),
            'hard tanh': lambda z: np.clip(z, -1, 1),
            'logsoftmax': lambda z: z - np.log(np.sum(np.exp(z - np.max(z, axis=1, keepdims=True)),
                axis=1, keepdims=True) + 1e-8)
```

```
        }

        return activations[self.activation_function](z)

    # Derivatives of activation functions
    """
    If an error arises using the 'sign' activation function, it is because the derivative is
        undefined at z = 0. (Will return NaN)
    """
    def activation_deriv(self, z):
        derivs = {
            'relu': lambda z: np.where(z > 0, 1, 0),
            'sigmoid': lambda z: (sig := 1 / (1 + np.exp(-z))) * (1 - sig),
            'id': lambda _: np.ones_like(z),
            'sign': lambda z: np.zeros_like(z),  # Derivative undefined at z = 0
            'tanh': lambda z: 1 - np.tanh(z) ** 2,
            'hard tanh': lambda z: np.where(np.abs(z) <= 1, 1, 0),
            # logsoftmax derivative here
            'logsoftmax': lambda z: np.exp(z - np.max(z, axis=1, keepdims=True)) / (
                        np.sum(np.exp(z - np.max(z, axis=1, keepdims=True)), axis=1, keepdims=True) +
                            1e-8)
        }

        return derivs[self.activation_function](z)

    def backward(self, dL_dout):
        dL_dout = np.nan_to_num(dL_dout)
        activation_deriv = self.activation_deriv(self.z)
        dL_dout *= activation_deriv
        # partial derivative of the loss w.r.t. the weights
        grad_W = np.dot(self.input_data.T, dL_dout)
        # accumulation of partial derivative of the loss for each layer
        dL_din = np.dot(dL_dout, self.weights.T)

        # Remove the bias
        dL_din = dL_din[:, :-1]

        grad_W = np.clip(grad_W, -3, 3)

        return grad_W, dL_din
```

Listing 2: `layer.py`

```
import matplotlib.pyplot as plt
import numpy as np

from layer import Layer
from fnn import FNN

# Generate Data
X = np.random.uniform(-3, 3, 500).reshape(-1, 1)  # Reshape X to be (n_samples, 1)
y = np.sin(X)

# Create Layers
layer1 = Layer(n_input=1, n_output=30, activation='relu')  # Input layer with 1 feature (x)
layer2 = Layer(n_input=30, n_output=30, activation='relu') # Hidden layer with 10 neurons
layer3 = Layer(n_input=30, n_output=1, activation='id')    # Output layer with 1 neuron (regression
    output)

# Create FNN
fnn = FNN(layers=[layer1, layer2, layer3])

# Train
learning_rate = 0.01
```

```
epochs = 1000
fnn.train(X, y, learning_rate, epochs)

y_pred = fnn.forward(X)

plt.figure()
plt.scatter(X, y, label='True Data')
plt.scatter(X, y_pred, color='red', label='FNN Predictions')
plt.title("FNN Approximation of sin(x)")
plt.legend()
plt.show()
```

Listing 3: `case1.py`

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import odeint
import torch

from layer import Layer
from fnn import FNN

# Generate Data - this code comes from the vanderpol.py file
# from lecture
def ode_model(x, t):
    return [x[1], -x[0] + (1 - x[0]**2)*x[1]]

def Phi(x):
    t = np.linspace(0, 0.05, 101)
    sol = odeint(ode_model, x, t)
    return sol[-1]

# compute the samples
# X is a set of samples in a 2D plane
# Y consists of the corresponding outputs of the samples in X

N = 101 # number of samples in each dimension
samples_x1 = torch.linspace(-3, 3, N)
samples_x2 = torch.linspace(-3, 3, N)

X = torch.empty((0,2))

for x1 in samples_x1:
    for x2 in samples_x2:
        sample_x = torch.Tensor([[x1,x2]])
        X = torch.cat((X, sample_x))

Y = torch.empty((0,2))
for x in X:
    y = Phi(x)
    sample_y = torch.Tensor([[y[0],y[1]]])
    Y = torch.cat((Y, sample_y))

# Plot Data
X_np = X.numpy()
Y_np = Y.numpy()

# Pre-Process Data
# X = torch.cat((X, torch.ones(X.shape[0], 1)), dim=1)
print(X.shape, Y.shape)

# Create Layers
layer1 = Layer(n_input=2, n_output=64, activation='relu')
layer2 = Layer(n_input=64, n_output=64, activation='relu')
```

```
layer3 = Layer(n_input=64, n_output=64, activation='relu')
layer4 = Layer(n_input=64, n_output=2, activation='id')

# Create FNN
fnn = FNN(layers=[layer1, layer2, layer3, layer4])

# Train
learning_rate = 0.01
epochs = 1000
#fnn.train(X_np, Y_np, learning_rate, epochs)
fnn.trainsgd(X_np, Y_np, learning_rate, epochs, 25, "mse")

x0 = np.array([1.25, 2.35])
#Predict
Y_pred = fnn.forward(X_np)

plt.figure(figsize=(8, 8))
for i in range(150):
    y = Phi(x0)
    plt.plot(y[0], y[1], 'b.')
    x0 = y

x0 = np.array([1.25, 2.35])
for i in range(150):
    y_pred = fnn.forward(x0.reshape(1, -1))
    plt.plot(y_pred[0, 0], y_pred[0, 1], 'r.')
    x0 = y_pred.flatten()

plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.title('Van der Pol Oscillator')
plt.grid(True)
plt.show()
```

Listing 4: `case2.py`

```
import numpy as np
import torch
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import MNIST
import matplotlib.pyplot as plt

from layer import Layer
from fnn import FNN


def get_data_loader(is_train):
    to_tensor = transforms.Compose([transforms.ToTensor()])
    data_set = MNIST("", is_train, transform=to_tensor, download=True)
    return DataLoader(data_set, batch_size=1000, shuffle=True)


def evaluate(test_data, net):
    n_correct = 0
    n_total = 0
    for batch_X, batch_y in test_data:
        batch_X = batch_X.view(batch_X.shape[0], -1).numpy()
        batch_y = batch_y.numpy()

        outputs = net.forward(batch_X)
        predicted = np.argmax(outputs, axis=1)
```

```
            n_correct += (predicted == batch_y).sum()
            n_total += batch_y.shape[0]

        return n_correct / n_total


# Create network
input_size = 28 * 28
hidden_size = 64
output_size = 10

layer1 = Layer(input_size, hidden_size, activation='relu')
layer2 = Layer(hidden_size, hidden_size, activation='relu')
layer3 = Layer(hidden_size, hidden_size, activation='relu')
layer4 = Layer(hidden_size, output_size, activation='logsoftmax')

net = FNN(layers=[layer1, layer2, layer3, layer4])

# Get data
train_data = get_data_loader(is_train=True)
test_data = get_data_loader(is_train=False)

# Initial accuracy
print("Initial accuracy:", evaluate(test_data, net))

# Training
learning_rate = 0.001
epochs = 100
batch_size = 25

# Prepare the entire training dataset
X_train = []
y_train = []
for batch_X, batch_y in train_data:
    X_train.append(batch_X.view(batch_X.shape[0], -1).numpy())
    y_train.append(batch_y.numpy())

X_train = np.concatenate(X_train)
y_train = np.concatenate(y_train)

# Convert labels to one-hot encoding
y_train_one_hot = np.zeros((y_train.size, output_size))
y_train_one_hot[np.arange(y_train.size), y_train] = 1

# Train using SGD
loss_func = 'nll'
for epoch in range(epochs):
    net.sgd(X_train, y_train_one_hot, batch_size, learning_rate, loss_func)

    # Evaluate after each epoch
    accuracy = evaluate(test_data, net)
    print(f"Epoch {epoch + 1}/{epochs}, Accuracy: {accuracy:.4f}")

# Visualize some predictions
for n, (x, _) in enumerate(test_data):
    if n > 5:
        break

    x_flat = x[0].view(-1).numpy()
    pred = np.argmax(net.forward(x_flat.reshape(1, -1)))

    plt.figure(n)
    plt.imshow(x[0].view(28, 28), cmap='gray')
    plt.title(f"Prediction: {pred}")

```

```
92  plt.show()
```

Listing 5: `case3.py`

Design Diagram