Assignment 2 Report

Multi-layer Neural Network Implementation and experiments

# Objective

The objective of this assignment is to implement and evaluate a configurable multi-layer neural network for both classification and regression tasks. The experiments focus on:

1. Evaluating the effect of different activation functions (ReLU, Sigmoid, and Linear) on the convergence and weight updates in both 3-layer and 4-layer models.
2. Comparing the performance of a 3-layer network (with one hidden layer of 10 neurons) versus a 4-layer network (with two hidden layers of 5 neurons each).
3. Investigating how different batch sizes affect training and validation dynamics.

# Experimental Setup

## Data Generation

* Classification:
  + Method: Synthetic dataset generated using scikit-learn's make\_classification.
  + Features: 10 features per instance.
  + Samples: 1,000 samples, split into 80% training and 20% testing.
  + Preprocessing: Data is standardized using StandardScaler and converted to PyTorch tensors.
* Regression:
  + Method: Synthetic dataset generated using scikit-learn's make\_regression with added Gaussian noise.
  + Features: 10 features per instance.
  + Samples: 1,000 samples, split into 80% training and 20% testing.
  + Preprocessing: Standardization and conversion to PyTorch tensors.

## Neural Network Architecture

* Design:
  + A configurable NeuralNetwork class is implemented using PyTorch's nn.Module as the base class.
  + Parameters: Input size, list of hidden layer sizes, output size, and activation functions for both hidden and output layers.
  + Forward Pass: Implemented using PyTorch's nn.Sequential for efficient layer composition.
  + Backward Pass: Utilizes PyTorch's autograd functionality for automatic differentiation and gradient computation.
  + Weight Initialization:
    - He initialization for ReLU activation functions
    - Xavier initialization for Sigmoid and Linear activation functions
  + Gradient Handling: Gradient clipping is implemented for regression tasks to prevent exploding gradients.

## Training and Hyperparameters

* Optimization: Mini-batch Stochastic Gradient Descent (SGD)
* Learning Rate:
  + 0.01 for classification tasks
  + 0.001 for regression tasks (reduced to prevent numerical instability)
* Epochs: 50 for all experiments
* Batch Sizes: Varied (16, 32, 64) to evaluate training dynamics
* Loss Functions:
  + Cross-Entropy Loss for classification tasks
  + Mean Squared Error (MSE) for regression tasks
* Validation: 20% of data used for validation to monitor generalization

# Experiments and Results

## 1st Experiment: Effect of Activation Functions on Weight Updates (3-layer model)

### Setup:

* Network: 3-layer model with 10 neurons in the hidden layer
* Activation Functions: ReLU, Sigmoid, and Linear activations in the hidden layer
* Output Activation: Sigmoid for classification, Linear for regression
* Hyperparameters: Learning rate = 0.01 (classification) / 0.001 (regression), batch size = 32, epochs = 50

### Results:

* Classification Task:
  + **ReLU activation** showed the fastest convergence, reaching a final loss of ~0.53 (training) and ~0.55 (validation)
  + **Sigmoid activation** converged more slowly, with a final loss of ~0.66 (training) and ~0.66 (validation)
  + **A graph of a function

    AI-generated content may be incorrect.Linear activation** performed surprisingly well for classification, with a final loss of ~0.48 (training) and ~0.50 (validation)
* Regression Task:
  + All activation functions showed gradual improvement in the regression task
  + **Linear activation** performed best with a final loss of ~0.17093 (training) and ~0.16700 (validation)
  + **ReLU activation** achieved a final loss of ~0.17201 (training) and ~0.16875 (validation)
  + **A graph of a function

    AI-generated content may be incorrect.Sigmoid activation** showed the slowest convergence with a final loss of ~0.17479 (training) and ~0.17046 (validation)

### Weight Update Analysis:

* ReLU activation showed larger initial weight updates that gradually decreased over time
* Sigmoid activation had smaller, more consistent weight updates throughout training
* Linear activation showed moderate weight updates with less decay over time
* The magnitude of weight updates correlated with the rate of loss reduction

A graph of different colored lines

AI-generated content may be incorrect.A graph with a green line

AI-generated content may be incorrect.

## 2nd Experiment: Effect of Activation Functions on Weight Updates (4-layer model)

### Setup:

* Network: 4-layer model with two hidden layers (8 and 4 neurons)
* Activation Functions: ReLU, Sigmoid, and Linear activations in the hidden layers
* Output Activation: Sigmoid for classification, Linear for regression
* Hyperparameters: Learning rate = 0.01 (classification) / 0.001 (regression), batch size = 32, epochs = 50

### Results:

* Classification Task:
  + **Linear activation** performed best in the 4-layer model, with a final loss of ~0.46 (training) and ~0.49 (validation)
  + **ReLU activation** achieved a final loss of ~0.65 (training) and ~0.65 (validation)
  + A graph of a function

    AI-generated content may be incorrect.**Sigmoid activation** showed minimal improvement, with a final loss of ~0.70 (training) and ~0.70 (validation)
* Regression Task:
  + **Linear activation** again performed best with a final loss of ~0.17059 (training) and ~0.16669 (validation)
  + **ReLU activation** achieved a final loss of ~0.17446 (training) and ~0.17014 (validation)
  + **A graph of a function

    AI-generated content may be incorrect.Sigmoid activation** showed minimal improvement with a final loss of ~0.17570 (training) and ~0.17174 (validation)

### Weight Update Analysis:

* In the 4-layer model, weight updates were generally smaller compared to the 3-layer model
* The deeper architecture showed more stable weight updates, especially for sigmoid activation
* Linear activation maintained larger weight updates even in later epochs
* The vanishing gradient problem was more evident in the 4-layer model with sigmoid activation

A graph of different colored lines

AI-generated content may be incorrect.A graph with green line and orange line

AI-generated content may be incorrect.

## 3rd Experiment: Effect of Number of Layers on Loss (3-layer vs 4-layer models)

### Setup:

* Comparison: 3-layer network (one hidden layer with 10 neurons) vs. 4-layer network (two hidden layers with 5 neurons each)
* Activation: ReLU used for hidden layers
* Hyperparameters: Consistent across both models (learning rate = 0.01 for classification / 0.001 for regression, batch size = 32, epochs = 50)

### Results:

* Classification Task:
  + **The 3-layer model** achieved better performance with a final loss of ~0.48 (training) and ~0.51 (validation)
  + **The 4-layer model** reached a final loss of ~0.51 (training) and ~0.53 (validation)
  + A graph of a graph

    AI-generated content may be incorrect.The 3-layer model converged faster in the early epochs

A graph of a graph

AI-generated content may be incorrect.A graph of different colored lines

AI-generated content may be incorrect.

* Regression Task:
  + **The 3-layer model** performed better with a final loss of ~0.17173 (training) and ~0.16701 (validation)
  + **The 4-layer model** achieved a final loss of ~0.17595 (training) and ~0.17148 (validation)
  + A graph of a graph

    AI-generated content may be incorrect.Both models showed steady improvement throughout training

A graph with a blue line

AI-generated content may be incorrect. A graph of different colored lines

AI-generated content may be incorrect.

### Observations:

* The 3-layer model generally outperformed the 4-layer model in both tasks
* This suggests that the additional complexity of the 4-layer model may not be beneficial for these particular datasets
* The 4-layer model showed more stable but slower convergence
* The difference in performance was more pronounced in the regression task

## 4th Experiment: Effect of Batch Size on Training and Validation Error

### Setup:

* Batch Sizes: Experiments conducted with batch sizes of 16, 32, and 64
* Network: A consistent 3-layer model with ReLU activation
* Hyperparameters: Learning rate = 0.01 for classification / 0.001 for regression, epochs = 50

### Results:

* Classification Task:
  + **Batch size 16** achieved the lowest final loss of ~0.48 (training) and ~0.51 (validation)
  + **Batch size 32** reached a final loss of ~0.54 (training) and ~0.57 (validation)
  + **Batch size 64** showed the slowest convergence with a final loss of ~0.65 (training) and ~0.65 (validation)

A graph of different colored lines

AI-generated content may be incorrect.A graph of different colored lines

AI-generated content may be incorrect. A graph of different colored lines

AI-generated content may be incorrect.

* Regression Task:
  + **Batch size 16** performed best with a final loss of ~0.17050 (training) and ~0.16508 (validation)
  + **Batch size 32** achieved a final loss of ~0.17521 (training) and ~0.17111 (validation)
  + **Batch size 64** showed the highest final loss of ~0.17878 (training) and ~0.17240 (validation)

A graph with green and blue lines

AI-generated content may be incorrect.A graph with different colored lines

AI-generated content may be incorrect.A graph of different colored lines

AI-generated content may be incorrect.

### Observations:

* Smaller batch sizes (16) consistently achieved better final performance in both tasks
* Larger batch sizes (64) resulted in smoother loss curves but slower convergence
* The training-validation gap was smallest with batch size 64, suggesting better generalization
* Smaller batch sizes showed more variability in the loss curves due to noisier gradient estimates
* The effect of batch size was more pronounced in the classification task

# Graphs

The loss versus iteration plots for each experiment are saved as PNG files in the results directory. Key takeaways include:

* Activation functions significantly influence convergence speed and final performance
* Linear activation performed surprisingly well, especially in the 4-layer model
* Network depth impacts loss reduction, but deeper is not always better
* Batch size selection affects both stability and speed of convergence, with smaller batch sizes generally achieving better final performance

# Conclusion

The experiments demonstrate that:

1. The choice of activation function is critical for training dynamics:
   * ReLU generally provides faster initial convergence
   * Linear activation can perform well in certain scenarios, challenging the conventional wisdom that non-linear activations are always superior
   * Sigmoid activation tends to converge more slowly, especially in deeper networks
2. Network depth considerations:
   * For the datasets used in these experiments, the 3-layer architecture outperformed the 4-layer architecture
   * Deeper networks may require more careful tuning of hyperparameters and initialization
   * The vanishing gradient problem was observed with sigmoid activation in deeper networks
3. Batch size optimization:
   * Smaller batch sizes (16) achieved better final performance but with more noisy training curves
   * Larger batch sizes (64) provided more stable training but slower convergence
   * The optimal batch size depends on the specific task and dataset characteristics
4. Regression vs. Classification:
   * Regression tasks required more careful handling, including lower learning rates and gradient clipping
   * The effects of architectural choices were more pronounced in regression tasks
   * Weight initialization played a crucial role in preventing numerical instability in regression

# References

These findings emphasize the importance of hyperparameter tuning and network configuration in neural network design. The choice of activation function, network depth, and batch size should be tailored to the specific task and dataset characteristics.

Resources

* Course Lecture Slides and Notes on Neural Networks and Deep Learning
* PyTorch Documentation: https://pytorch.org/docs/stable/index.html
* Scikit-learn Documentation: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make\_classification.html