CS6910
DEEP LEARNING (SPRING 2024)
PROGRAMMING ASSIGNMENT 1

Optimization Methods, Normalization Methods and Autoencoders

Instructor: Prof. Chandra Shekhar Chellu
Group: Team #4

Students:
ME20B059 Dharani Govindasamy
ME20B087 Janmenjaya Panda
ME20B122 Nishant Sahoo
ME20B143 Raghav Jangid

Release Date: 26/02/2024
Submission Deadline: 25/03/2024

Department of Computer Science & Engineering
Indian Institute of Technology Madras

Contents

List of Figures

1 Problem 1

Task 1: Comparison of optimization methods for classification on Image dataset 1

- Model: MLFFNN with 2 hidden layers and `tanh` activation function
- Lossfunction: Cross-entropy
- Mode of learning: Pattern mode
- Stopping criterion: Change in average error below a threshold
- Weight update rules:
  1. Delta rule
  2. Generalized delta rule
  3. AdaGrad
  4. RMSProp
  5. AdaM
- Use the same value of the learning rate parameter
- Use the same initial random values of weights
- For each rule of weight update, the report should include the following:
  1. Plot of the average error on training data vs Epoch
  2. Confusion matrices for training data and test data
  3. Compare the number of epochs taken for convergence for different update rules.

In this task, the objective is to conduct a comparative analysis of various optimization methods concerning the classification on the predefined image dataset. This entails the implementation of the stated model, given loss function, prescribed mode of learning with stopping criterion and several weight update rules. Prior to delving into the plots, observations, experimental comparison of optimization methods and inferences, it is imperative to go through the theoretical aspects of the aforementioned components.

1.1 The Model: MLFFNN

A Multi-Layer Feedforward Neural Network, aka MLFFNN, is a common type of artificial neural network composed of multiple layers of interconnected nodes or neurons forming a directed acyclic graph, in which each layer serves a specific purpose in processing input data. The flow of information through the network is strictly forward, from the input layer to the output layer through the hidden layers, hence the term “feedforward". The hidden layers, situated between the input and output layers, play a crucial role in capturing and representing complex relationships within the data. They act as feature extractors, transforming the input data into a higher-dimensional space where patterns and relationships can be more effectively discerned.

The significance of Multilayer Feedforward Neural Networks (MLFFNN) resides in the universal approximation theorem. The Universal Approximation Theorem, as originally formulated by Cybenko in 1989 and later extended by Hornik in 1991, can be stated as follows:

Theorem 1: The Universal Approximation Theorem

Let $\Phi(x)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Then, for any continuous function f defined on a compact subset of $\mathbb{R}^n$, and any $\epsilon > 0$, there exist an integer m, real constants v_i, b_i, and real-valued vectors w_i such that the function

$$F(x) = \sum_{i=1}^{m} v_i \cdot \Phi(w_i^T x + b_i)$$

satisfies $|F(x) - f(x)| < \epsilon$ for all x in the compact subset.

The above theorem formally establishes the capability of MLFFNNs to approximate any continuous function within a compact input space with arbitrary precision, provided a suitable number of neurons and activation function. To achieve a threshold precision for the approximation, a single hidden layer with a larger number of neurons can be substituted by multiple hidden layers with a smaller number of neurons by layer.

The concerned problem asks to implement a multilayer feedforward neural network with two hidden layers with `tanh` activation function for the classification task. The following diagram captures the architecture of the same:

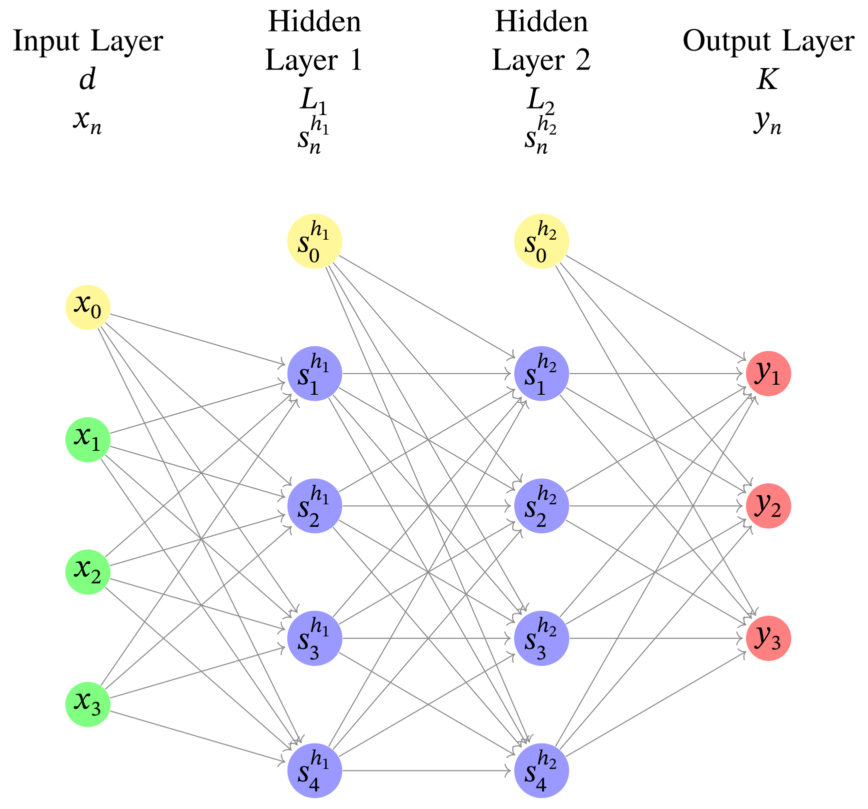


Figure 1: An illustration of an MLFFNN with two Hidden Layers

Here's a brief description of the key components and operations of a multilayer feedforward neural network:

- **Input Layer:** This is a visible layer, that consists of input neurons that receive the initial data or features. Each neuron represents a feature of the input data. The number of neurons in the input layer is determined by the dimensionality of the input data (d). The input layer is essentially denoted as a set of vectors:
$$\mathbf{X} := [x_1, x_2, x_3, \dots, x_N]$$
where n is the number of input vectors, and for each n in $[N]$, the vector $x_n \in \mathbb{R}^d$, where d is the dimension of the input vector.

- **Hidden Layers:** Between the input and output layers, there may be one or more hidden layers. Each hidden layer consists of neurons that perform computations on the input data. The term "hidden" refers to the fact that these layers do not directly interface with the input or output of the network.

- **Output Layer:** The output layer produces the final output of the network based on the computations performed by the hidden layers. The number of neurons in the output layer depends on the type of task the network is designed to solve (e.g., classification, regression).

- **Weights and Biases:** Each connection between neurons in adjacent layers is associated with a weight ($\mathbf{W}$), which determines the strength of the connection. Additionally, each neuron typically has an associated bias ($\mathbf{B}$), that is essentially a linear shift added in that layer, which allows the network to learn more complex patterns. For the weight joining the ith neuron in one layer and jth neuron in the next layer, the weight is represented by:

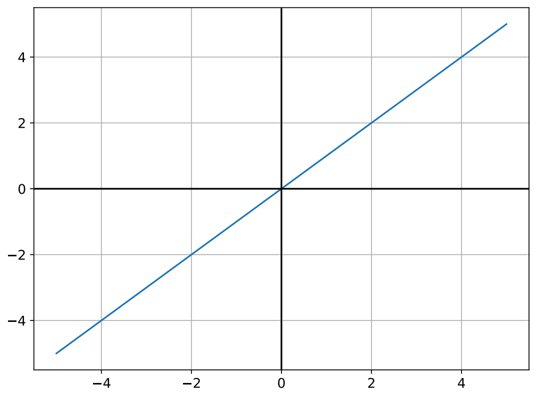
  1. w_{ij}^{o}, if the next layer is the output layer, and by
  2. $w_{ij}^{h_l}$, if the next layer is the hidden layer h_l.

1.2 Activation Function

Each neuron applies an activation function (Φ) to the weighted sum of its inputs, along with the bias term. The activation function introduces non-linearity to the network, enabling it to learn and approximate complex functions. Different layers may use different activation functions. There are several kinds of activation functions with multiple specifications and applications, some of which has been discussed below.

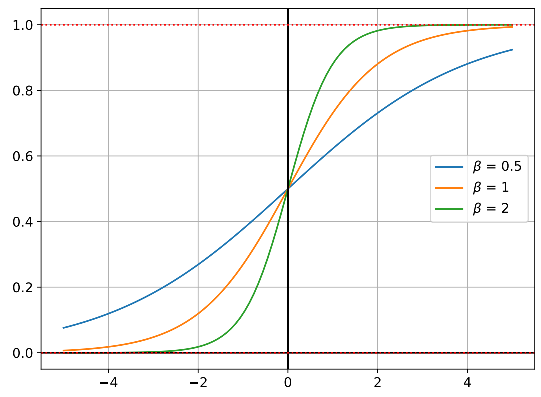
Linear Activation Function

- $\Phi(x) = x$
- $\dfrac{d\Phi(x)}{dx} = 1$
- It is utilized in the final layer of neural networks for tasks such as linear regression, where the model predicts continuous values.



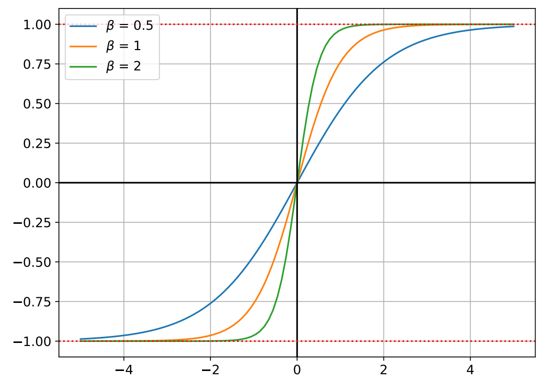
Logistic Sigmoid Activation Function

- $\Phi(x) = \dfrac{1}{1 + e^{-\beta x}} = \sigma(\beta x)$
- $\dfrac{d\Phi(x)}{dx} = \beta \cdot \Phi(x) \cdot (1 - \Phi(x))$
- Widely used in binary classification tasks, where the model outputs probabilities ranging from 0 to 1.



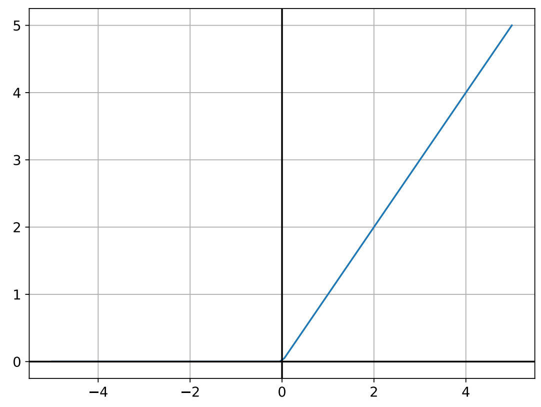
Hyperbolic Tangent Activation Function

- $\Phi(x) = \tanh(\beta x)$
- $\dfrac{d\Phi(x)}{dx} = \beta \cdot (1 - \Phi(x)^2)$
- Often used in neural networks for tasks such as classification and regression, where it helps in capturing non-linear relationships in the data.



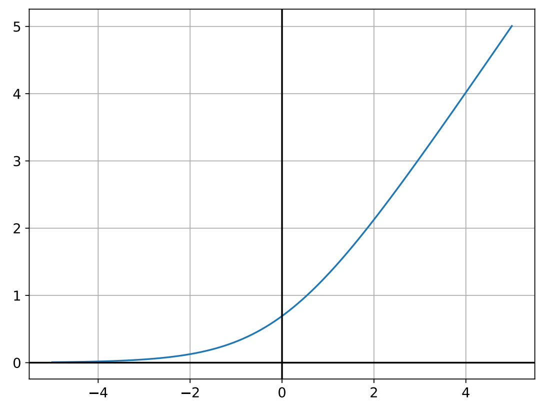
Rectified Linear Unit Activation Function

- $\Phi(x) = \max(0, x)$
- $\dfrac{d\Phi(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$
- Widely used in deep learning models for its simplicity and effectiveness in combating the vanishing gradient problem.



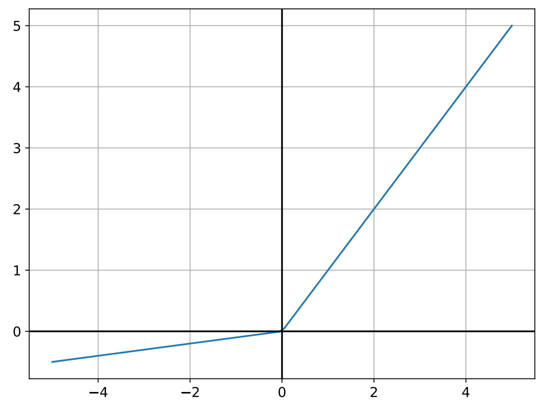
Smooth ReLU Activation Function

- $\Phi(x) = \log(1 + e^x)$
- $\dfrac{d\Phi(x)}{dx} = \dfrac{e^x}{1 + e^x} = \dfrac{1}{1 + e^{-x}} = \sigma(x)$
- Smooth ReLU is a smoothed version of ReLU, offering continuous and differentiable behavior.



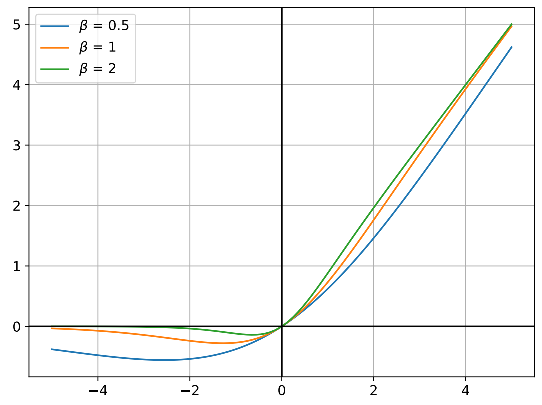
Leaky ReLU Activation Function

- $\Phi(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}$
- $\dfrac{d\Phi(x)}{dx} = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{otherwise} \end{cases}$
- Leaky ReLU is similar to ReLU but allows a small, non-zero gradient when the input is negative.



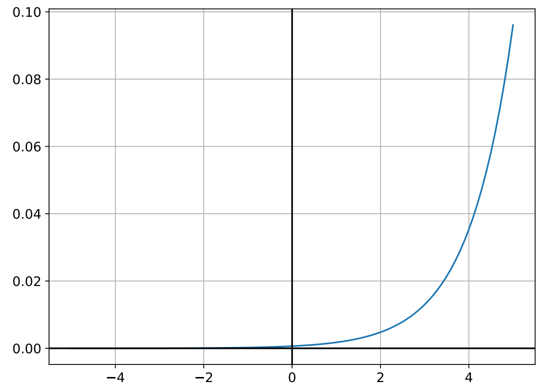
Swish Activation Function

- $\Phi(x) = x \cdot \sigma(\beta x)$
- $\dfrac{d\Phi(x)}{dx} = \sigma(\beta x) + \beta \cdot \Phi(x) \cdot (1 - \Phi(x))$
- Swish is a novel activation function that tends to perform better than ReLU and its variants in certain scenarios.



Softmax Activation Function

- $\Phi(x_l) = \dfrac{e^{x_l}}{\sum_{j=1}^{n} e^{x_j}}$ for $l = 1, 2, \dots, K$
- $\dfrac{d\Phi(x_l)}{dx_l} = \Phi(x_l) \cdot (1 - \Phi(x_l))$
- Softmax function is commonly used in the output layer of neural networks for multi-class classification tasks.



Here, we shall be using the the `tanh` activation function.

1.3 Loss Function

The output of the network is compared to the ground truth (target) using a loss function, which measures the difference between the predicted output and the actual output. The goal of training is to minimize this loss function by adjusting the network's weights and biases. Depending upon the application, there are several kinds of loss functions. Two popular loss functions, the mean square error and the cross entropy loss function, have been discussed below:

1. **Mean Squared Error (MSE):** MSE is a common loss function used primarily for regression tasks. It measures the average squared difference between the predicted values and the actual values in a dataset. For a dataset with N samples let t_n be the true target value and let y_n be the predicted target value for the nth sample for some $n \in [N]$. The mean squared error concerning this data point (MSE) goes as follows:

$$MSE(\tilde{\mathcal{E}_n}) := \frac{1}{2}\sum_{i=1}^{K}(t_{ni} - y_{ni})^2 \tag{1}$$

where K is the dimension of the tagert value, that is $y_n \in \mathbb{R}^K$.

2. **Cross Entropy Loss Function:** The Cross-Entropy Loss function, also known as Log Loss, is a popular choice for training classification models. It quantifies the difference between two probability distributions: the predicted probabilities output by the model and the true distribution of the labels. It penalizes confident and wrong predictions more than less confident ones, which is desirable in a loss function. Let t_n and y_n be the true and the predicted output for a sample datapoint X_n. Then, the cross entropy loss function (CE) for this data point goes as follows:

$$CE(\tilde{\mathcal{E}_n}) := -\sum_{i=1}^{K} t_{ni}\log y_{ni} \tag{2}$$

Here, we shall be using the cross entropy loss function.

1.4 The Mode of Learning

Let's discuss the different modes of learning commonly used in training machine learning models: pattern mode, batch mode, and mini-batch mode.

1. **Pattern Mode Learning:** In pattern mode learning, aka online learning or stochastic learning, the model updates its parameters after processing each individual training sample. It involves sequentially presenting training samples to the model, computing gradients, and updating parameters based on the gradients.

   - *Error Calculation:* In pattern mode of weight update, at step m, we pick a sample x_n and the error $\mathcal{E}$ is considered to be precisely $\tilde{\mathcal{E}_n}$. Hence, one epoch corresponds to N updates where N is the number of samples.
   - *Usage :* Pattern mode learning is suitable when dealing with large datasets or streaming data, where it may not be feasible to load the entire dataset into memory at once. It is commonly used in scenarios where the data is continuously arriving, such as real-time prediction tasks or online learning environments.
   - *Benefits:* Pattern mode learning requires minimal memory as it processes one sample at a time. Also, in pattern mode the model can adapt quickly to changes in the data distribution or concept drift.
   - *Challenges:* Updates based on individual samples can be noisy and less stable compared to batch updates. Convergence may be slower due to frequent updates and noisy gradients.

2. **Batch Mode Learning:** In batch mode learning, the model updates its parameters after processing the entire training dataset in one go. It involves computing the gradients for all training samples, averaging them, and then updating the parameters.

    - *Error Calculation:* In batch mode of weight update, at step m, we consider each of the sample x_n and the error is given by $\mathcal{E} \leftarrow \mathcal{E}_{avg}$, where $\mathcal{E}_{avg} = \frac{1}{N}\sum_{n=1}^{N} \tilde{\mathcal{E}_n}$. Hence, one epoch corresponds to 1 update. Here N stands for the number of data points.
    - *Usage:* Batch mode learning is suitable for smaller datasets that can fit into memory and where parallel processing can be utilized efficiently. It is commonly used in scenarios where stability in updates and convergence speed are important, such as offline training or batch processing pipelines.
    - *Benefits:* Updates based on the entire dataset tend to be more stable and less noisy compared to pattern mode learning. Also, Batch updates can lead to smooth and faster convergence as they utilize more information from the entire dataset.
    - *Challenges:* Batch mode learning may require significant memory to store the entire dataset and compute gradients, limiting its applicability to large datasets. Computing gradients for the entire dataset can be computationally expensive, especially for large models or high-dimensional data.

3. **Mini-batch Mode learning:** Mini-batch mode learning strikes a balance between pattern mode and batch mode learning. It involves dividing the training dataset into smaller subsets called mini-batches and updating the parameters based on the gradients computed on each mini-batch.

    - *Error Calculation:* Let $\mathcal{D}$ denote the entire dataset with $|\mathcal{D}| = N$ and let M denote the size of a minibatch. The bth minibatch is denoted as $\mathcal{D}_b$ for some $b \in [B]$, where $B := \left\lceil \frac{N}{M} \right\rceil$. The error $\mathcal{E}$ is given as by the average error concerning the bth minibatch $\mathcal{D}_b$ for some b. In other word, $\mathcal{E} \leftarrow \mathcal{E}^b_{avg}$, where $\mathcal{E}^b_{avg} = \frac{1}{M}\sum_{x_n \in \mathcal{D}_b} \tilde{\mathcal{E}_n}$.
    - *Usage:* Mini-batch mode learning is widely used in deep learning and neural network training due to its efficiency and scalability. It allows for parallel processing of mini-batches, making it suitable for training on GPUs and distributed systems.
    - *Benefits:* Mini-batch mode learning requires less memory compared to batch mode learning, as it processes smaller subsets of the data at a time. Mini-batches strike a balance between stability in updates (similar to batch mode) and computational efficiency (similar to pattern mode).
    - *Challenges:* The choice of mini-batch size can impact convergence speed and model performance, requiring experimentation and tuning. Also, mini-batch mode learning introduces additional hyperparameters (e.g., learning rate schedules) that may need to be tuned for optimal performance.

In summary, the choice of learning mode depends on factors such as the size of the dataset, computational resources available, and the desired trade-off between stability, efficiency, and convergence speed. Pattern mode learning is suitable for streaming data or real-time applications, batch mode learning is ideal for offline training on smaller datasets, and mini-batch mode learning is commonly used in deep learning for its efficiency and scalability. **For the given task, we shall consider the Pattern mode of learning.**

1.5 Forward Propagation

During the forward pass, input data is fed into the network, and computations are performed layer by layer until the output layer produces the final prediction or output. Each neuron in a layer receives inputs from the previous layer, applies the activation function, and passes the result to the next layer. Consider the following path taken from the above illustration:

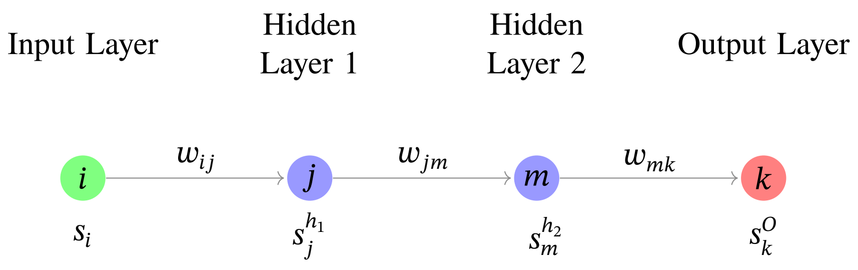


Figure 10: A sample path from an input feature to the one output in an MLFFNN

Here, the forward propagation at different layers go as follows:

1. *Input Layer:* At input layer, we consider the ith feature of the nth input data point for some $i \in [d]$ and some $n \in [N]$

$$s_i = x_{ni}$$

   Note that, here there are no activation functions; in other words, the activation function is linear.

2. *Hidden Layer:* Analogously, at the hidden layers, we compute the linear sum of the outputs of the previous layer weighted by the corresponding weights and bias parameter followed by the activation through the activation function specified at that layer. For the jth neuron in the Hidden Layer 1, which enjoys the input from the input layer, the equation goes as follows:

$$a_j^{h_1} = \sum_{i=1}^{d} w_{ij}^{h_1} s_i + w_{0j}^{h_1}$$
$$s_j^{h_1} = \Phi^{h_1}(a_j^{h_1})$$

   where d is the dimension/ total no of features in the input vector. $w_{0j}^{h_1}$ is the weight corresponding to the connection joining bias term in Input Layer and the jth neuron in the output layer and Φ^{h_1} is the activation function used in the Hidden Layer 1. Analogously, for the mth neuron in the Hidden Layer 2:

$$a_m^{h_2} = \sum_{j=1}^{L_1} w_{jm}^{h_2} s_j^{h_1} + w_{0m}^{h_2}$$
$$s_m^{h_2} = \Phi^{h_2}(a_m^{h_2})$$

   Here L_1 is the total number of neurons (except the bias term) in Hidden Layer 1. $w_{0j}^{h_2}$ is the weight corresponding to the connection joining bias term in the Hidden Layer 1 and the mth neuron in Hidden Layer 2 and Φ^{h_2} is the activation function used in the Hidden Layer 2.

3. *Output Layer:* Similarly, the forward propagation corresponding to the kth neuron in the output layer is as follows:

$$a_k^O = \sum_{m=1}^{L_2} w_{mk}^O s_m^{h_2} + w_{0k}^O$$
$$s_k^O = \Phi^O(a_k^O)$$

   Here L_2 is the total number of neurons (except the bias term) in Hidden Layer 2. w_{0k}^O is the weight corresponding to the connection joining bias term in the Hidden Layer 2 and the kth neuron in Output Layer, and Φ^O is the activation function used in the Output.

1.6 Back Propagation

Once the loss is calculated, the network updates its weights and biases using backpropagation, which involves computing the gradient of the loss function with respect to the network parameters. This gradient is then used to update the parameters through several optimization algorithms. These have been discussed below.

1.7 The Weight Update Rules

Weight update rules are essential for training neural networks, as they dictate how the network's weights are adjusted during the optimization process to minimize the loss function. Here are five fundamental weight update rules:

1.7.1 Delta Rule

Delta rule is a **first-order optimization algorithm** that iteratively updates the weights of the neural network in the direction of the negative gradient of the loss function. It aims to **minimize the loss function by taking steps proportional to the negative of the gradient.** Delta rule guarantees convergence to a local minimum (or saddle point) of the loss function under certain conditions, such as convexity or convexity assumptions. It is conceptually the simplest weight update rule. The weight update rule is given by:

$$w(m+1) \leftarrow w(m) - \eta \cdot g_w(m) \tag{3}$$

where:

- $w(m)$ represents the weights at iteration m,
- $w(m+1)$ represents the weights at iteration $m+1$,
- η is the learning rate,
- $g_w(m)$ is the gradient of the loss function $\tilde{\mathcal{E}}(w)$ with respect to the weights evaluated at iteration m, that is $g_w(m) := \left.\frac{\partial \tilde{\mathcal{E}}(w)}{\partial w}\right|_{w=w(m)}$.

1.7.2 Generalized Delta Rule

Generalized Delta rule, aka the momentum method, is an extension of delta rule that **aims to accelerate convergence and dampen oscillations by introducing a momentum term**. This term accumulates the gradient updates over time and adjusts the weights in the direction of the accumulated gradients. It helps accelerate convergence by smoothing out oscillations and facilitating faster updates in the direction of the gradient; also it dampens the oscillations in the weight updates, leading to more stable optimization trajectories. However, the momentum coefficient (α) needs to be tuned, and improper tuning can lead to suboptimal performance. Also, it requires storing and updating additional momentum vectors, increasing memory requirements. The weight update rule is given by:

$$\begin{aligned} w(m+1) &\leftarrow w(m) - \eta \cdot g_w(m) + \alpha \cdot \Delta w(m) \\ \Delta w(m) &:= w(m) - w(m-1) \end{aligned} \tag{4}$$

where:

- the terms $w(m)$, $w(m+1)$, η and $g_w(m)$ are precisely the same as explained in the delta rule update, and
- α is the momentum factor. Typically, $\alpha \approx 0.9$.

1.7.3 Adaptive Gradient (AdaGrad)

Adagrad is an adaptive learning rate optimization algorithm designed **to adjust the learning rates of individual parameters based on their historical gradients.** It adapts the learning rates by scaling them inversely proportional to the square root of the sum of squared gradients for each parameter. It adapts the learning rates for each parameter based on the historical gradients, allowing for faster convergence and better scaling for different parameters. Also, it does not require manual tuning of learning rates or hyperparameters, making it easier to use. However, as it accumulates squared gradients over time, the learning rates may become too small, causing slow convergence or premature convergence. Adagrad does not incorporate momentum, which can hinder its ability to navigate saddle points or noisy landscapes compared to momentum-based methods like AdaM. The accumulation of historical gradients can lead to memory inefficiency, particularly for large models or long training sequences. The weight update rule for AdaGrad is given by:

$$\begin{aligned} w(m+1) &\leftarrow w(m) - \frac{\eta \cdot g_w(m)}{\epsilon + \sqrt{r_w(m)}} \\ r_w(m) &:= \sum_{i=0}^{m-1} g_w^2(i) \end{aligned} \tag{5}$$

where:

- the terms $w(m)$, $w(m+1)$, η and $g_w(m)$ are precisely the same as explained in the delta rule update,
- $r_w(m)$ is the accumulated squared gradient,
- ϵ is a small constant added for numerical stability; as per `torch.optim` $\epsilon \approx 10^{-8}$.

Adagrad is particularly useful in scenarios where the gradients for different parameters vary widely or when there is little prior knowledge about appropriate learning rates. However, it may not be as effective in scenarios with highly non-convex loss surfaces or when training deep neural networks with complex architectures.

1.7.4 Root-Mean-Square Propagation (RMS Prop)

RMSprop is an adaptive learning rate optimization algorithm designed **to address the issues of vanishing or exploding gradients** in deep neural networks. It modifies AdaGrad by normalizing the gradient updates by dividing them by a running average of their magnitudes. It adjusts the learning rate for each parameter based on the magnitude of the gradients, improving convergence in different directions. and helps stabilize the optimization process by preventing large gradient updates. Once again, the hyperparameters (β) requires proper tuning. The weight update rule for RMSProp is given by:

$$\begin{aligned} w(m+1) &\leftarrow w(m) - \frac{\eta \cdot g_w(m)}{\epsilon + \sqrt{r_w(m)}} \\ r_w(m) &:= \beta \cdot r_w(m-1) + (1-\beta) \cdot g_w^2(m) \end{aligned} \tag{6}$$

where:

- the terms $w(m)$, $w(m+1)$, η and $g_w(m)$ are precisely the same as explained in the delta rule update,
- $r_w(m)$ is the exponentially decaying average of squared past gradients (as per `torch.optim`, $\beta \approx 0.99$),
- ϵ is a small constant added for numerical stability. $\epsilon \approx 10^{-8}$.

1.7.5 AdaM (Adaptive Moment Estimation)

AdaM **combines the benefits of momentum and RMSprop** by integrating momentum into the computation of the adaptive learning rates. It maintains both a running average of past gradients and their squared gradients, which are then used to update the weights. The weight update rule for AdaM is given by:

$$\begin{aligned} w(m+1) &\leftarrow w(m) - \frac{\eta \cdot \hat{q}_w(m)}{\epsilon + \sqrt{\hat{r}_w(m)}} \\ \hat{q}_w(m) &:= \frac{q_w(m)}{1-\beta_1^m} \\ \hat{r}_w(m) &:= \frac{r_w(m)}{1-\beta_2^m} \\ q_w(m) &:= \beta_1 q_w(m-1) + (1-\beta_1) g_w(m) \\ r_w(m) &:= \beta_2 r_w(m-1) + (1-\beta_2) g_w^2(m) \end{aligned} \tag{7}$$

where:

- the terms $w(m)$, $w(m+1)$, η and $g_w(m)$ are precisely the same as explained in the delta rule update, $r_w(m)$ is analogous to the same term defined in RMSProp. As per `torch.optim`, $\beta_{\approx}0.9$ and $\beta_2 \approx 0.999$.

- ϵ is a small constant added for numerical stability; as per `torch.optim` $\epsilon \approx 10^{-8}$.

Note that $\mathrm{g}_w(m)$ and $\mathrm{g}_w^2(m)$ are the first and the second order moment of the gradient respectively. Also, the reader may recall that $\mathbb{E}[\mathrm{g}_w] \approx \hat{q}_w(m)$ and $\mathbb{E}[\mathrm{g}_w^2] \approx \hat{r}_w(m)$.

AdaM is widely used in practice due to its effectiveness in handling non-stationary and noisy gradients, making it suitable for several neural network architectures and optimization tasks.

1.8 Plots

1.8.1 Delta Rule

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	70
# Neurons (HL2)	35
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-5}

Result	Value
Converged at epoch #	479
Train Loss	0.0148
Train Accuracy	1.0
Test Loss	3.8889
Test Accuracy	0.518
Validation Loss	4.10205
Validation Accuracy	0.516

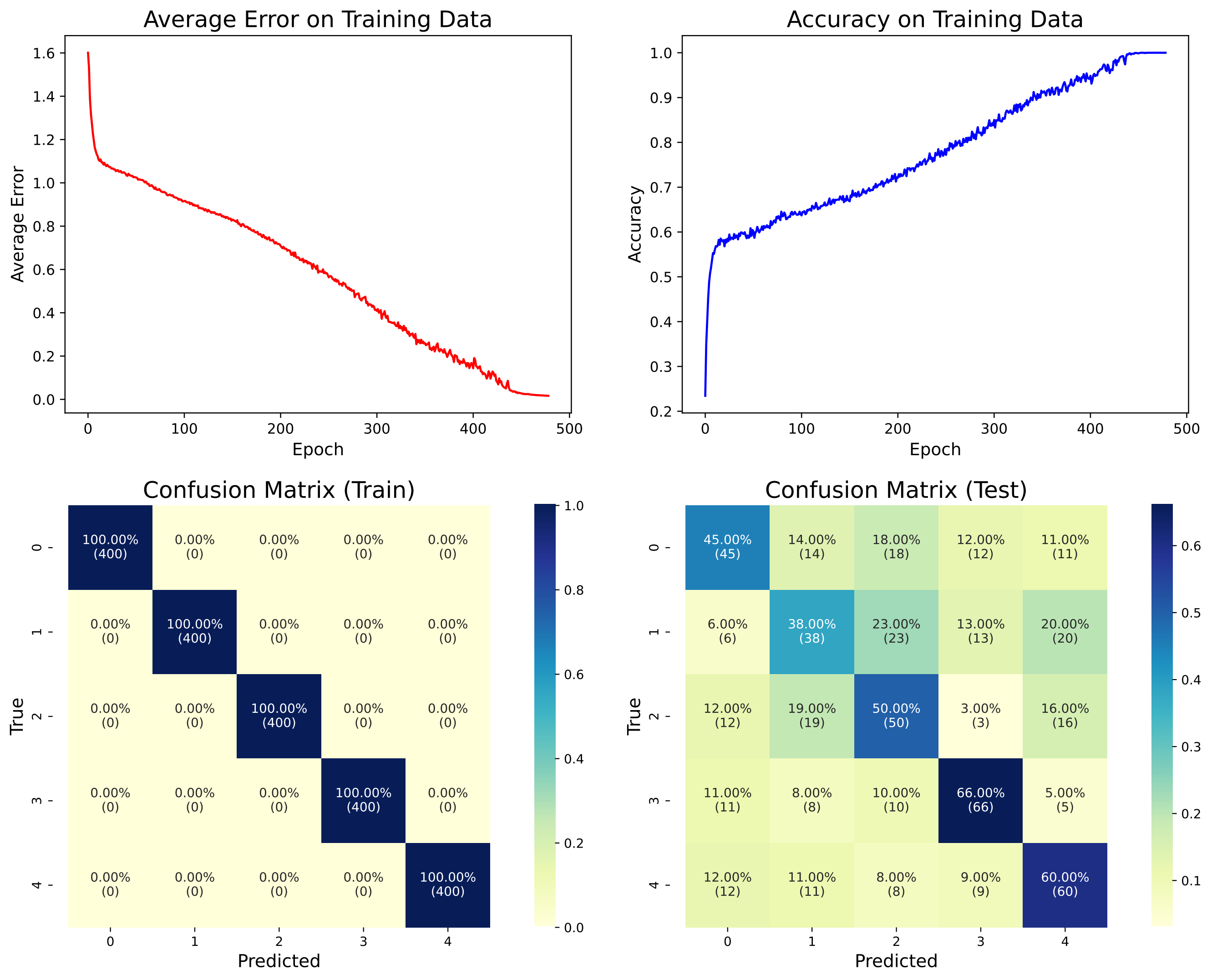


Figure 11: Plots for Delta Rule

1.8.2 Generalized Delta Rule

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	70
# Neurons (HL2)	35
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-5}

Result	Value
Converged at epoch #	511
Train Loss	0
Train Accuracy	1.0
Test Loss	3.1347
Test Accuracy	0.543
Validation Loss	3.97451
Validation Accuracy	0.531

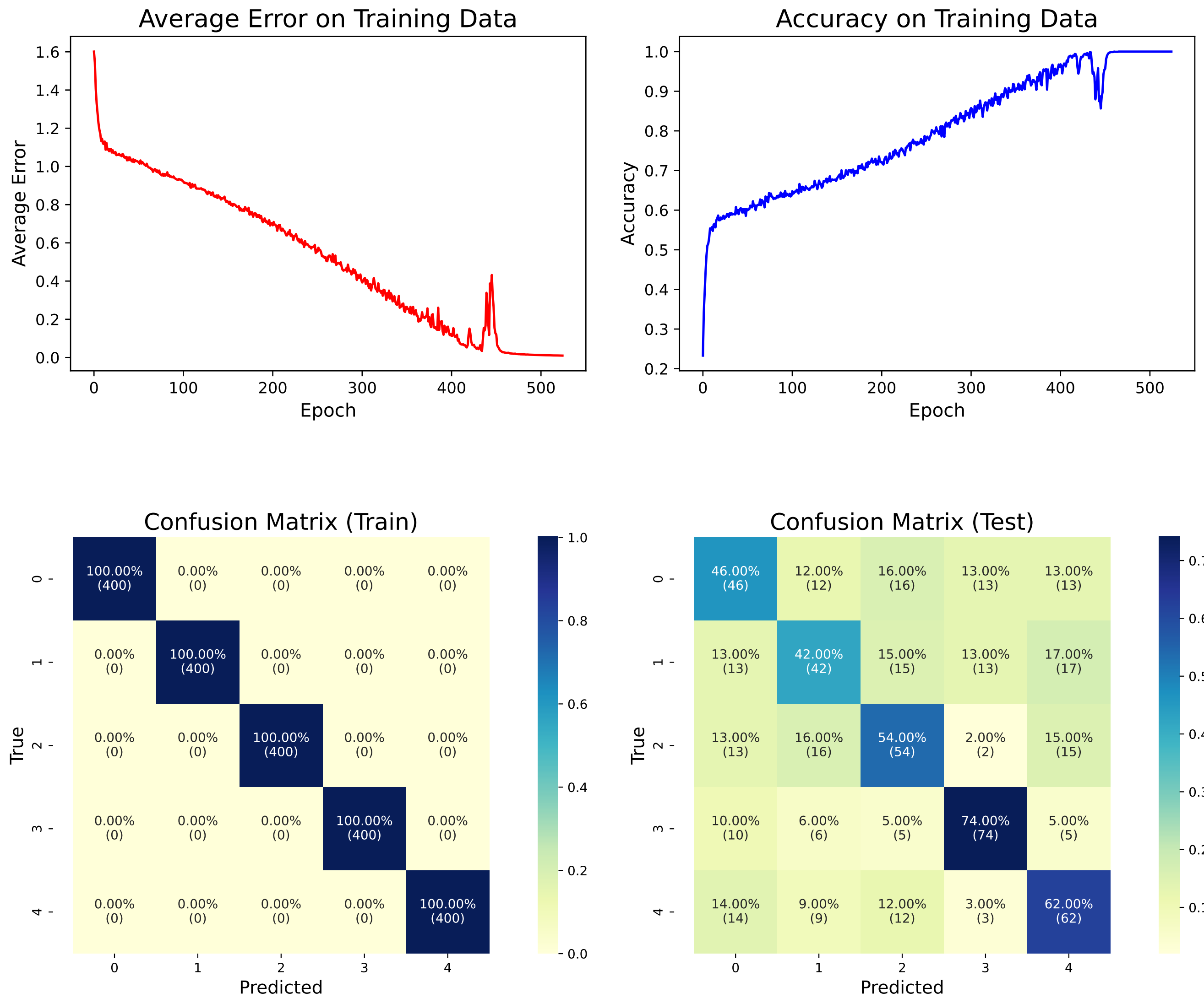


Figure 12: Plots for Generalized Delta Rule

1.8.3 Adaptive Gradient (AdaGrad)

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	70
# Neurons (HL2)	35
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-5}

Result	Value
Converged at epoch #	171
Train Loss	1.0113
Train Accuracy	0.6125
Test Loss	1.0567
Test Accuracy	0.564
Validation Loss	1.1272
Validation Accuracy	0.564

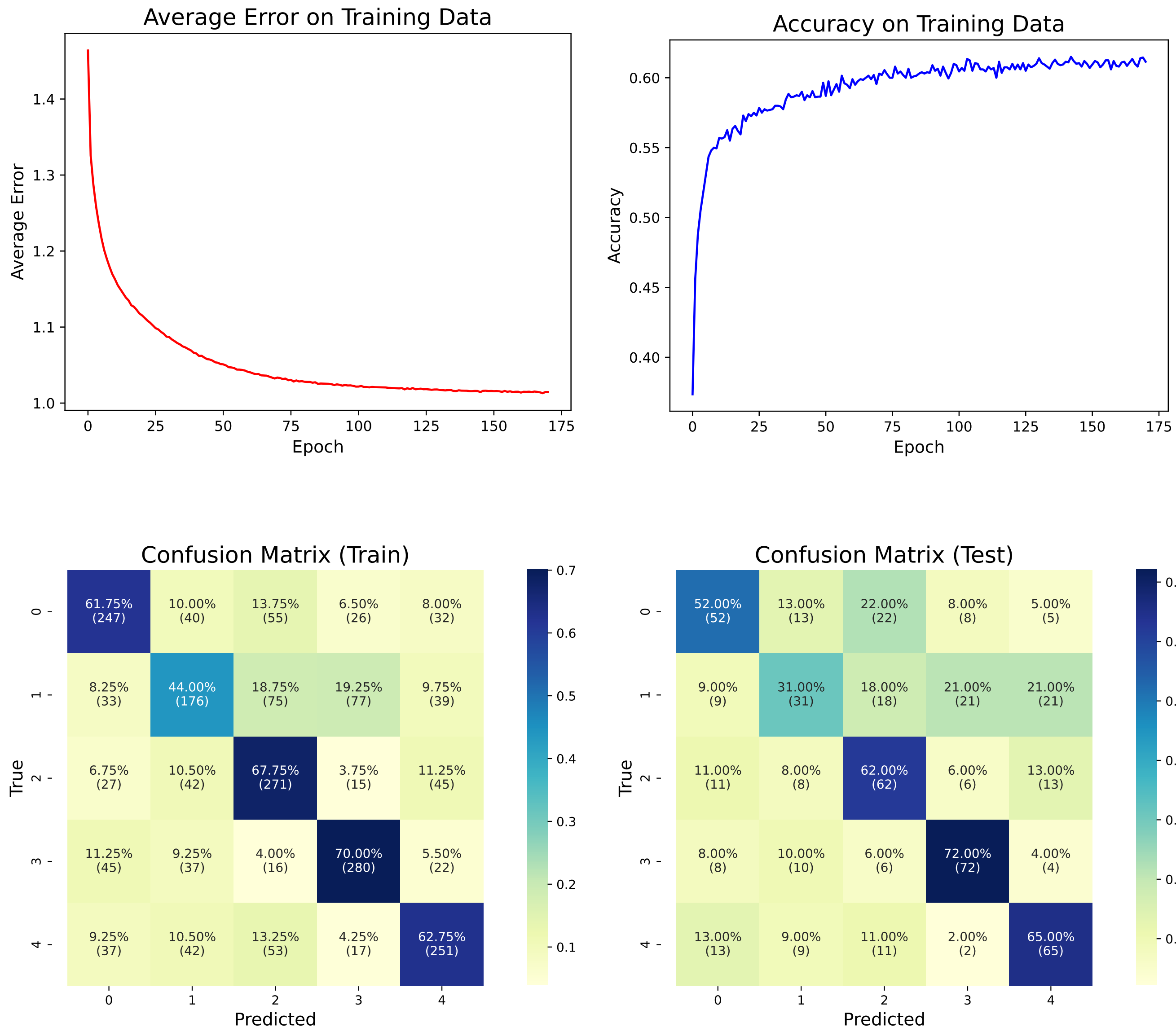


Figure 13: Plots for AdaGrad

1.8.4 Root-Mean-Square Propagation (RMSProp)

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	70
# Neurons (HL2)	35
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-5}

Result	Value
Converged at epoch #	2473
Train Loss	0.47391
Train Accuracy	0.8565
Test Loss	2.1975
Test Accuracy	0.528
Validation Loss	2.5291
Validation Accuracy	0.500

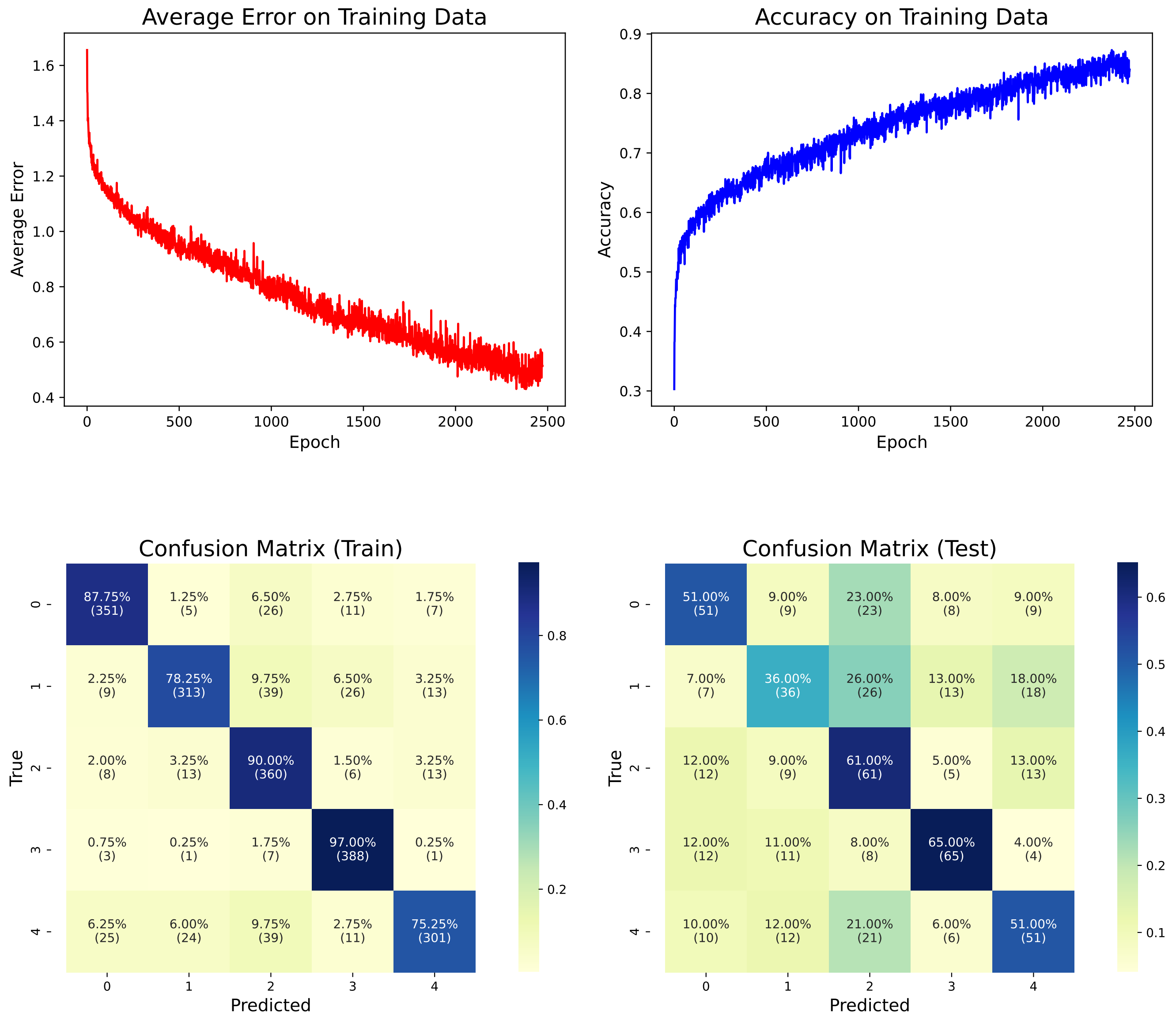


Figure 14: Plots for RMSProp

1.8.5 Adaptive Moment Estimation (AdaM)

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	70
# Neurons (HL2)	35
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-5}

Result	Value
Converged at epoch #	1535
Train Loss	1.0311
Train Accuracy	0.591
Test Loss	1.32098
Test Accuracy	0.54
Validation Loss	1.4279
Validation Accuracy	0.49

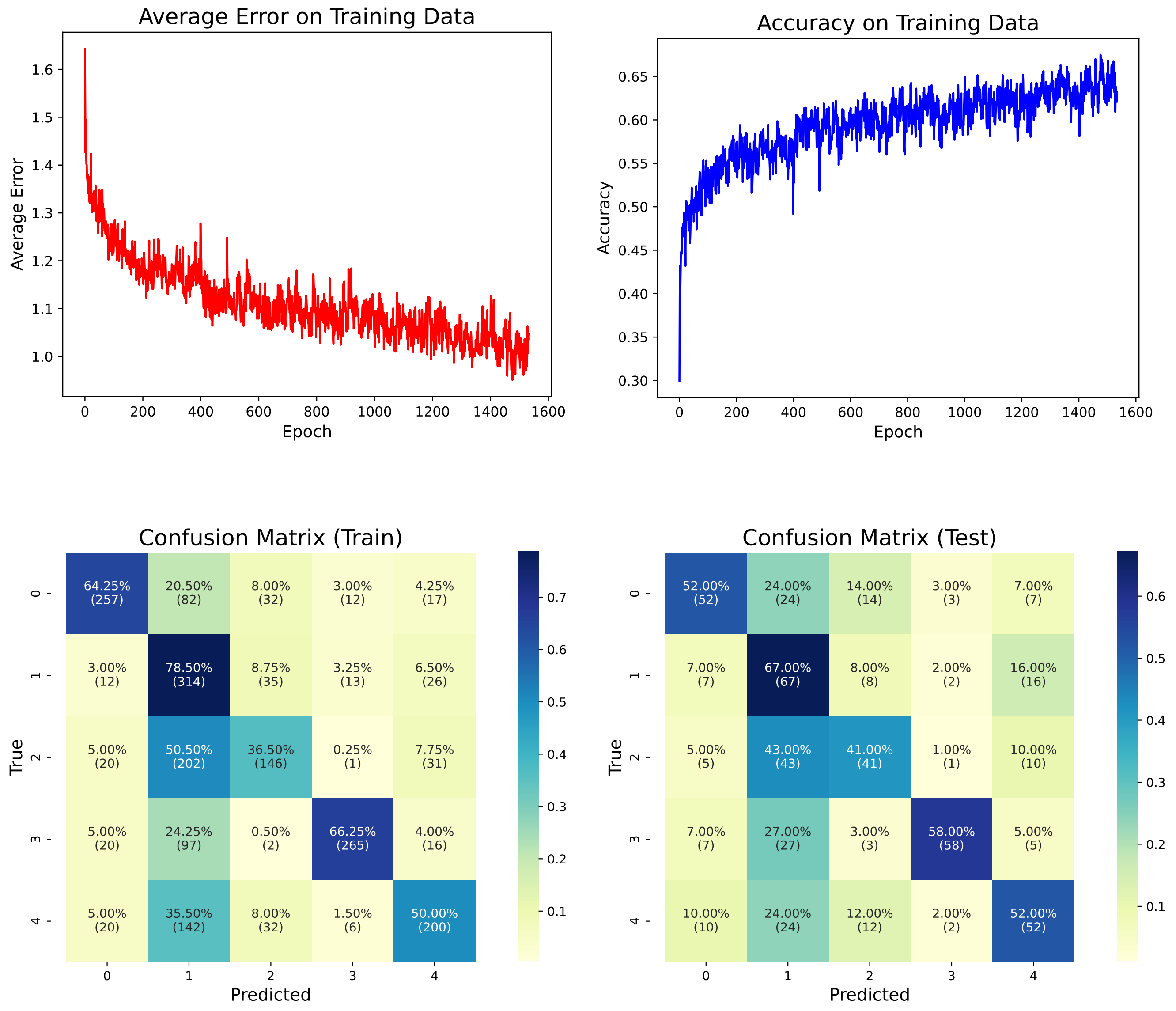


Figure 15: Plots for AdaM

1.9 Tabulation:

Optimization methods	Converged at epoch #	Accuracy Training	Accuracy Validation	Accuracy Testing
Delta Rule	479	100%	51.60%	51.80%
Generalized Delta Rule	511	100%	53.10%	54.30%
AdaGrad	171	61.25%	56.40%	56.40%
RMSProp	2473	85.65%	50.00%	52.80%
AdaM	1535	59.110%	49.00%	54.00%

Table 1: A comparison on several optimization methods

1.10 Observations and Inferences:

Here, we are using the pattern mode of learning, so as a result, many of the methods showcase a degree of stochasticity and fluctuation that is not visible during the batch mode of learning. From the graphs, we observe many different results for each of the modes of weight updation and these are elaborated below:

- The delta rule is the simplest of the updation rules, without any momentum or adaptive gradient terms. Hence, it depends heavily on the initialization parameters and this lead good results. Also, the learning rate does not change with time. Moreover, it performed better because of its simplicity, robustness to hyperparameters, and potential to avoid overfitting
- The generalized delta rule adds in a momentum term to accelerate convergence and reduce oscillations slightly. Note that it performed fairly good enough on test data mostly because of its simplicity and on test data, it achieved slightly more accuracy than the delta rule update.
- The adaptive gradient method utilizes an adaptive learning rate method in order to speed up convergence and prevent oscillations as compared to the delta rule methods. This can be seen in the graphs as the accuracy and error quickly reach a stable value and then slightly oscillate around a point. Also, due to a better learning method, the testing accuracy is higher than the delta rules which allows for more generalization to new data.
- RMSProp adds in a term to combat the issues of vanishing and exploding gradients by adding a denominator term to normalize the values. However, in pattern mode of learning, this leads to a more jagged graph and higher stochasticity since there is more variability in the updates which goes against the idea of using normalization in RMSprop.
- AdaM is theoretically the best method, using both momention and RMSprop to create the fastest converging and most accurate learning algorithms. However, due to the pattern mode of learning that is used, it is observed that the graph is very jagged in nature and it exhibits a high degree of stochasticity.
- We observe that for RMSprop and AdaM we use adaptive learning rates, compared to a very aggressive decay in the AdaGrad method. While this is more beneficial to accelerate convergence, this also increases the variability in the optimization process and thus leads to stochasticity.

- RMSprop and AdaM are much more sensitive to the initial choice of parameters compared to generalized delta rule and AdaGrad. Since we have chosen the same set for all the methods, we observe variablility in updates for the momentum methods and thus there are oscillations.

2 Problem 2

Task2: Comparison of normalization methods for classification on Image dataset 2

- Model: MLFFNN with 2 hidden layers and tanh activation function
- Lossfunction: Cross-entropy
- Mode of learning: Mini-batch mode
- Stopping criterion: Change in the average error below a threshold
- Weight update rule: AdaM
- Normalization method:
  1. Batch Normalization
  2. Batch normalization with post-activation normalization
- Use the same value of the learning rate parameter
- Use the same initial random values of weights
- For each normalization method, the report should include the following:
  1. Plot of the average error on training data vs Epoch
  2. Confusion matrices for training data and test data
- Compare the number of epochs taken for convergence for normalization methods

---

In this section, we shall be observing the effects of normalization on the training of an MLFFNN and compare it to no batch normalization. In particular, we shall be using an MLFFNN with two hidden layers and tanh as the activation function, cross-entropy as the loss function in the mini-batch mode of learning with AdaM weight update rule and batch normalization with both pre- and post-activation normalization to conduct our study.

2.1 The Internal Covariate Shift:

Internal Covariate Shift refers to the change in the distribution of network activations that occurs within deep neural networks during training. As the parameters of the network are updated during training, the distribution of inputs to each layer (i.e., activations) changes, which can slow down the training process.

This phenomenon arises due to the fact that the parameters of earlier layers affect the distribution of inputs to subsequent layers. As the network learns, the parameters of earlier layers change, causing the distribution of inputs to shift. This shift can make it challenging for the network to converge efficiently because the optimal learning rate may vary throughout training. This means that convergence will be slower and there is a possibility of instability due to vanishing or exploding gradients.

Batch normalization is one technique used to mitigate internal covariate shifts. It normalizes the inputs to each layer by subtracting the batch mean and dividing by the batch standard deviation, effectively stabilizing the distribution of inputs and reducing the internal covariate shift. This normalization helps in improving the convergence of deep neural networks by ensuring that the inputs to each layer remain within a reasonable range throughout the training process. Note that without the scaling and shifting being done, there can be some loss in the non-linearity of the model and hence we add these (γ_j and ρ_j) as learnable parameters.

2.2 Normalization Method:

This section talks about different kinds of normalization methods. Normalization methods are techniques used to standardize or normalize the input data or intermediate representations within neural networks. These methods aim to improve the convergence and stability of the training process by reducing the internal covariate shift and accelerating the optimization process. Two common normalization techniques used in neural networks are Batch Normalization and Layer Normalization. Note that, usually Batch Normalization is used in Convolutional Neural Networks, aka CNNs, whereas Layer Normalization is used in Recurrent Neural Networks, aka RNNs and transformer models.

2.2.1 No Normalization:

In this case, there is no normalization that takes place anywhere in the model, whether at the input data or the data concerning the hidden layer outputs.

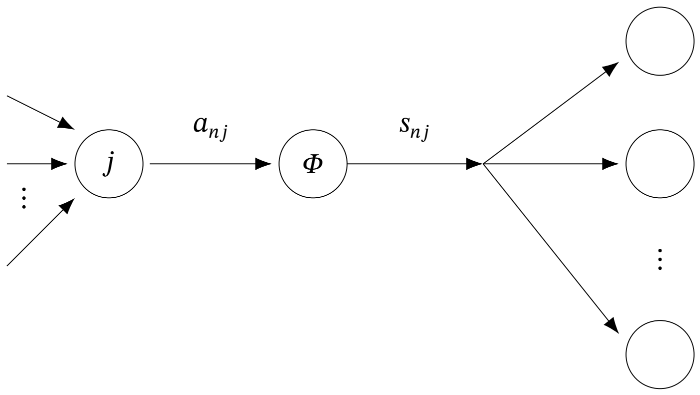


Figure 16: An illustration of no batch normalization

2.2.2 Batch Normalization with Post-Activation Normalization:

In post-batch normalization, the normalization operation is applied to the outputs of each layer. This means that the activations of each layer are normalized before being passed as inputs to the subsequent layer.

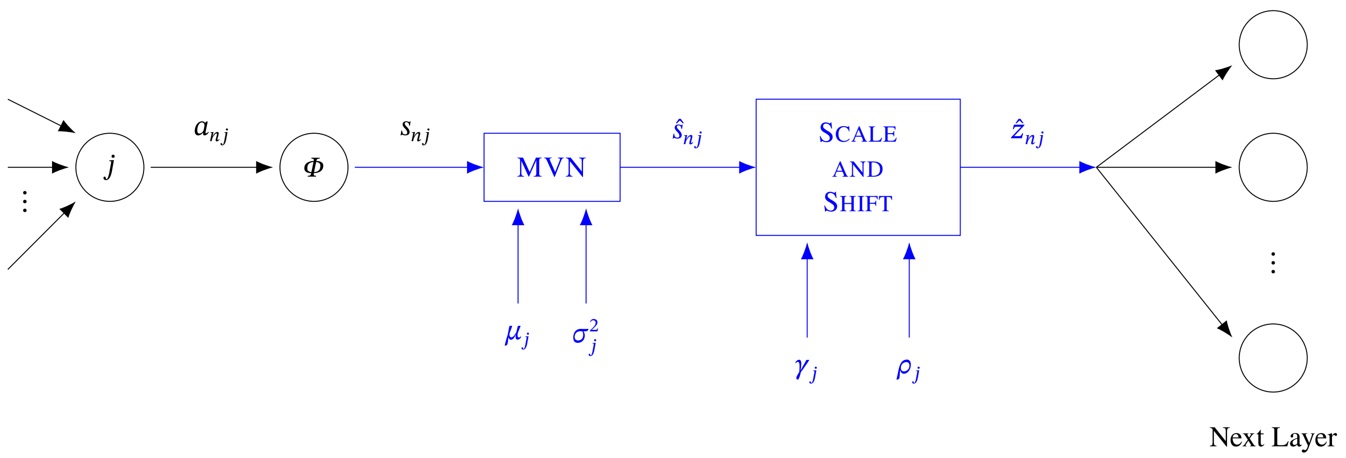


Figure 17: An illustration of post-activation batch normalization

Consider some node j in some non-output layer. The expression a_{nj} and s_{nj} have been discussed in MLFFNN in Problem 1.

After obtaining s_{nj}, the Mean-Variance Normalization, aka MVN, is performed on the activated value. For every example in $\mathcal{D}_b$, the term $\hat{s}_{nj}$ is obtained.

$$\mu_j = \frac{1}{M} \sum_{x_n \in \mathcal{D}_b} s_{nj}^h$$

$$\sigma_j^2 = \frac{1}{M} \sum_{x_n \in \mathcal{D}_b} (s_{nj}^h - \mu_j)^2$$

Thus,

$$\hat{s}_{nj} = \frac{s_{nj}^h - \mu_j}{\sigma_j} \tag{8}$$

where, $\hat{s}_{nj}$ is a normalized term, with mean 0 and variance 1. Hence, almost surely, $-3.0 \leqslant \hat{s}_{nj} \leqslant +3.0$.

Furthermore, scaling and shifting of the normalized data leads us to the following expression of $\hat{z}_{nj}$.

$$\hat{z}_{nj} = \gamma_j \hat{s}_{nj} + \rho_j \tag{9}$$

Here, γ_j and ρ_j are parameters that are learned by the model during training and weight updation.

Due to the addition of these new parameters, the backpropagation equation changes and the expression goes as follows:

$$\frac{d\tilde{\mathcal{E}}_n}{d\gamma_j} = -\sum_{m=1}^{K} w_{jm}^o \underbrace{(t_{nm} - s_{nm}^o)\frac{d\phi^o(a_m^o)}{da_m^o}}_{\delta_m^o} s_{nj}^h$$

$$\frac{d\tilde{\mathcal{E}}_n}{d\rho_j} = -\sum_{m=1}^{K} w_{jm}^o \underbrace{(t_{nm} - s_{nm}^o)\frac{d\phi^o(a_m^o)}{da_m^o}}_{\delta_m^o}$$

1. *Advantages:* It normalizes the activations of each layer, making the optimization landscape more consistent and potentially accelerating convergence. It helps in addressing the vanishing/exploding gradients problem, especially in deeper networks, by ensuring that the activations are within a reasonable range. Post-batch normalization allows for greater flexibility in network architecture, as normalization is applied after each layer.

2. *Disadvantages:* Post-batch normalization may introduce some computational overhead, as it requires additional calculations at each layer during both training and inference. It can sometimes lead to over-smoothing of the learned representations, reducing the capacity of the network to capture fine-grained features.

2.2.3 Batch Normalization with Pre-Activation Normalization:

In pre-batch normalization, the normalization operation is applied to the activation values of each layer before feeding them into the activation function. This means that the input features of each layer are normalized separately, typically by subtracting the mean and dividing by the standard deviation computed across the entire training dataset or mini-batch. This can be followed by scaling and shifting.

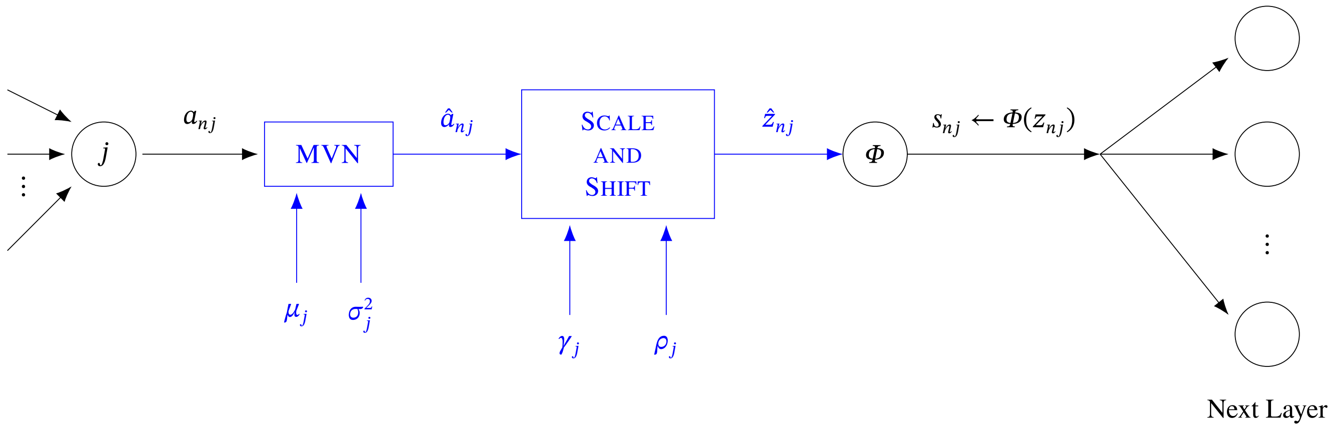


Figure 18: An illustration of pre-activation batch normalization

Consider some node j in some non-output layer. The expression a_{nj} has been discussed in MLFFNN in Problem 1.

After obtaining a_{nj}, the Mean-Variance Normalization, aka MVN, is performed on the activation value. For every example in $\mathcal{D}_b$, the term $\hat{a}_{nj}$ is obtained.

$$\mu_j = \frac{1}{M} \sum_{x_n \in \mathcal{D}_b} a_{nj}^h$$

$$\sigma_j^2 = \frac{1}{M} \sum_{x_n \in \mathcal{D}_b} (a_{nj}^h - \mu_j)^2$$

Thus,

$$\hat{a}_{nj} = \frac{a_{nj}^h - \mu_j}{\sigma_j} \tag{10}$$

where, $\hat{a}_{nj}$ is a normalized term, with mean 0 and variance 1. Hence, almost surely, $-0.3 \leqslant \hat{a}_{nj} \leqslant +3.0$.

Furthermore, scaling and shifting of the normalized data leads us to the following expression of $\hat{z}_{nj}$.

$$\hat{z}_{nj} = \gamma_j \hat{a}_{nj} + \rho_j \tag{11}$$

Here, γ_j and ρ_j are parameters that are learned by the model during training and weight updation.

The activation function acts on $\hat{z}_{nj}$ to generate $\hat{s}_{nj}$ that is fed to the next layer.

$$\hat{s}_{nj} = \Phi(\hat{z}_{nj})\rho_j \tag{12}$$

Due to the addition of these new parameters, the backpropagation equation changes and the expression goes as follows.

- *Advantages:* It helps in mitigating the internal covariate shift problem by ensuring that the input to each layer remains normalized throughout the training process. By stabilizing the inputs to each layer, it can accelerate the convergence of the training process. It provides some degree of regularization, reducing the reliance on techniques like dropout or weight decay.

- *Disadvantages:* Pre-batch normalization may not be as effective if the input distributions vary significantly across different mini-batches or during inference.

2.3 Plots

2.3.1 No Normalization

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	60
# Neurons (HL2)	30
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-6}

Result	Value
Converged at epoch #	1435
Train Loss	1.1036
Train Accuracy	0.783
Test Loss	0.9768
Test Accuracy	0.603
Validation Loss	1.0373
Validation Accuracy	0.631

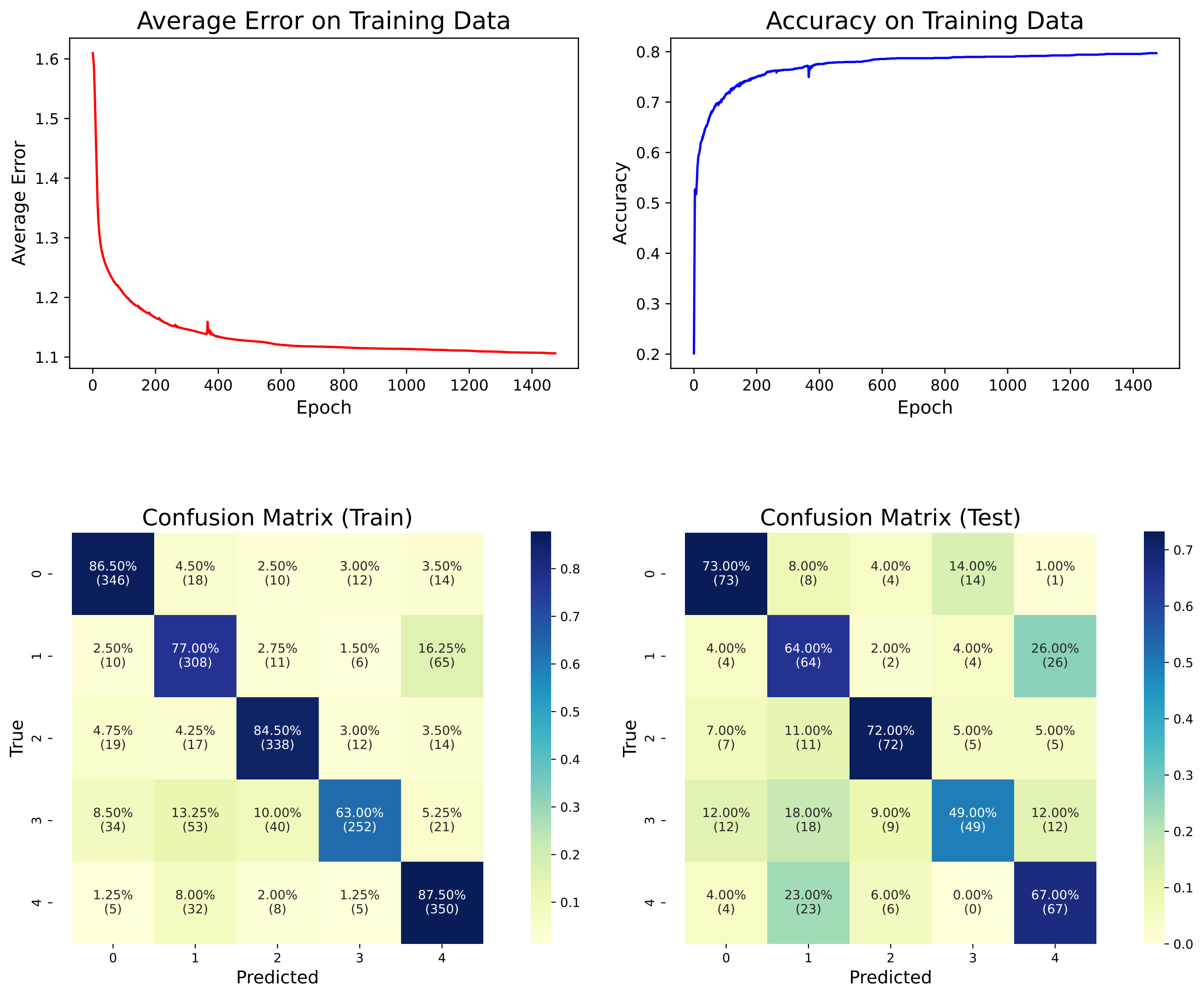


Figure 19: Plots for No Normalization

2.3.2 Post-Activation Normalization

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	60
# Neurons (HL2)	30
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-6}

Result	Value
Converged at epoch #	893
Train Loss	0.9816
Train Accuracy	0.923
Test Loss	0.9638
Test Accuracy	0.897
Validation Loss	0.9413
Validation Accuracy	0.837

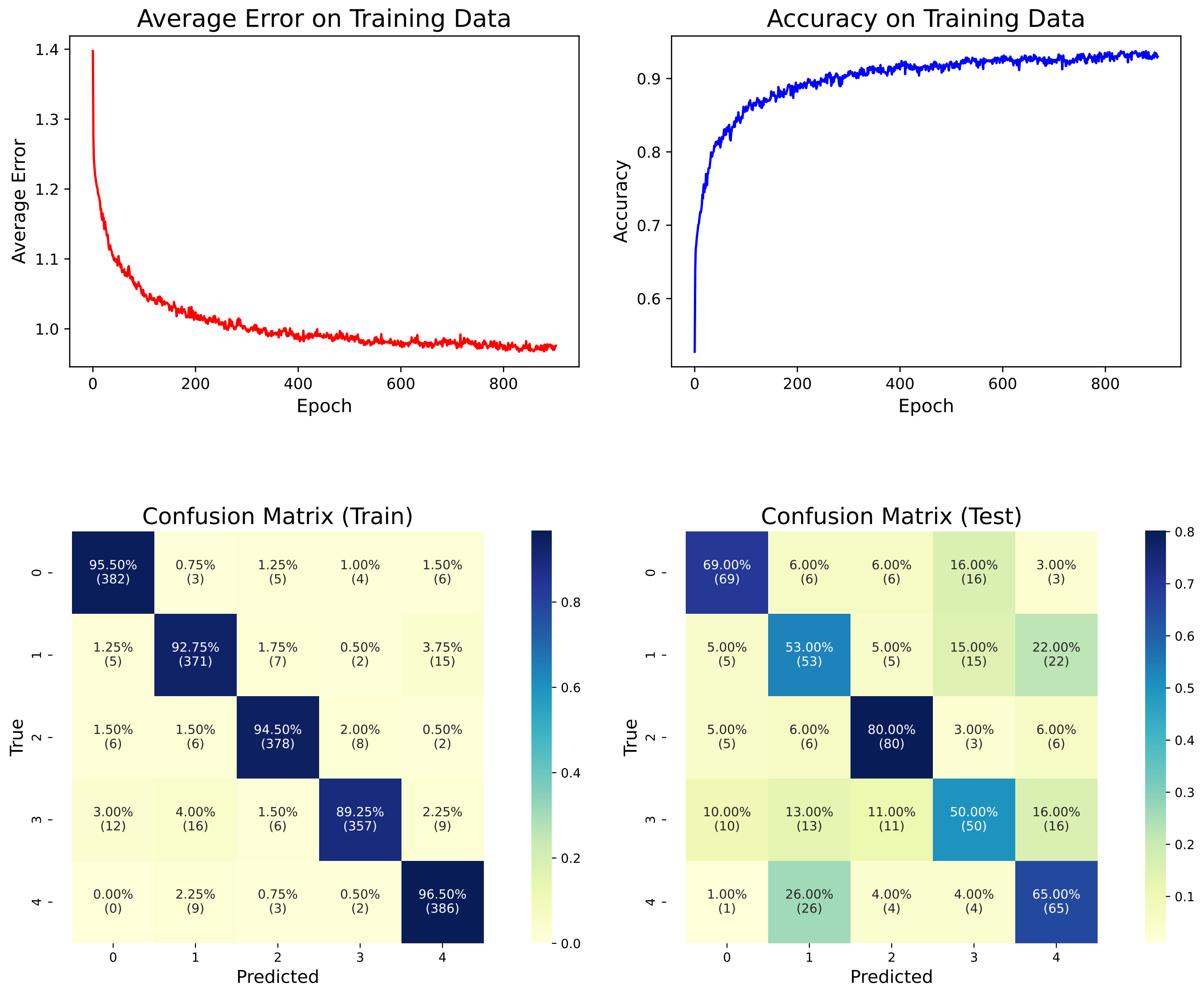


Figure 20: Plots for Post-Activation Normalization

2.3.3 Pre-Activation Normalization

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	60
# Neurons (HL2)	30
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-6}

Result	Value
Converged at epoch #	785
Train Loss	0.9618
Train Accuracy	0.934
Test Loss	0.9741
Test Accuracy	0.887
Validation Loss	0.9672
Validation Accuracy	0.821

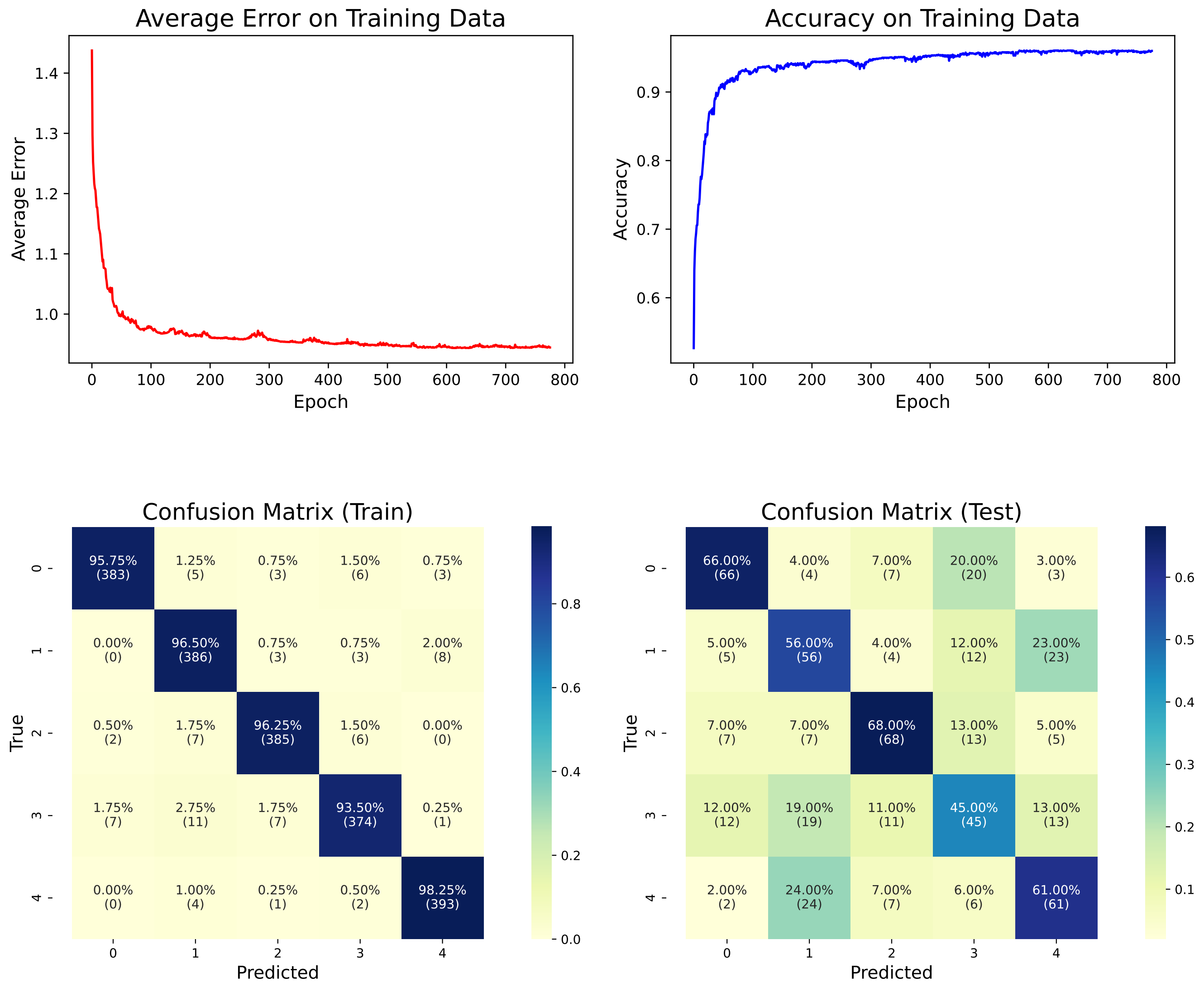


Figure 21: Plots for Pre-Activation Normalization

2.4 Tabulation

Normalization methods	Converged at epoch #	Accuracy Training	Validation	Testing
No Normalization	1435	78.34%	63.14%	60.38%
Post-activation Batch Normalizaiton	893	92.35%	83.78%	89.74%
Pre-activation Batch Normalization	785	93.74%	82.19%	88.71%

Table 2: A comparison on several batch Normalization methods

2.5 Observation and Inferences

We observe that the graphs for average error and accuracy in the case without any normalization do not show much oscillation and hence they appear to be smooth. However, the graphs involving either pre- or post- activation normalization are much more jagged in nature, indicating oscillations. We also notice that there is a higher amount of epochs to reach convergence in the case without normalization, as compared to those with normalization. Therefore, we can infer the following:

- Since the number of epochs is more in the case of non-normalized data, the training process is longer. We can thus conclude that normalizing the data leads to faster convergence and lower training time, making it useful.
- Normalizing the data allows for a standardized distribution of the data that is supplied to every neuron. This allows the outputs to stay within a certain range of values which thus prevents any blowing up or vanishing of gradients.
- However, this also leads to greater sensitivity of the learning rate. This means that there is an oscillation whenever learning rate is changed, which happens in this case since we are using AdaM method of weight updation.
- Due to the higher speed of training during normalization, there is a faster adaptation to new data. Hence, this speed leads to oscillations in the accuracy and error which manifest in the jagged nature of the graphs.
- Also, during normalization, we invariably introduce some noise to the data due to the nature of the process itself. This noise is also part of the reason for the jaggedness of the graph.
- We observe a higher training accuracy in the cases involving normalization as compared to the case where normalization is not performed. This is due to normalizing mitigating the effects of internal covariate shift and also preventing the effect of vanishing/exploding gradients.
- There is also a generally higher test accuracy when we use normalization. This is because batch normalization introduces a regularization effect by means of adding noise to the data, reducing the over-reliance on particular activations or variables. Due to this regularization effect, overfitting is prevented and this allows for better generalization on new, test data.

3 Problem 3

Task 3: Stacked autoencoder (with 3 autoencoders) based pre-training of a DFNN-based classifier for Image dataset 3

- Model of AANN: 5-layer structure
- Mode of learning for AANNs: Mini-batch mode
- Stopping criterion: Change in the average error below a threshold
- Weight update rule: AdaM
- Report should include the confusion matrices for training data and test data, for
  1. DFNN trained using only the labelled data
  2. DFNN using a stacked autoencoder pre-trained using unlabeled data and fine-tuned using the labelled data

  (DFNN configuration should be the same in both (1) and (2))

n this task, the objective is to implement auto associative neural network both using labelled data and using a stacked autoencoder pre-trained using unlabelled data and fine-tuned using labelled data. Before diving into the plots, observations, experimental comparison of the above two models and inferences, it is imperative to go through the theoretical aspects of autoencoders.

3.1 AutoEncoder

An autoencoder is a type of neural network architecture used for unsupervised learning and dimensionality reduction. It aims to learn a compressed representation (encoding) of the input data and reconstruct the input data from this representation (decoding).

3.1.1 Architecture:

Autoencoders consist of two main components: an encoder and a decoder. The encoder compresses the input data into a lower-dimensional latent space representation. The decoder reconstructs the input data from the latent space representation. The encoder and decoder are typically symmetric, with the encoder reducing the dimensionality of the input and the decoder expanding it back to the original dimension. Figure 22 shows the sample architecture of an Autoencoder. Note that here we have not included the bias paramter corresponding to each of the layer.

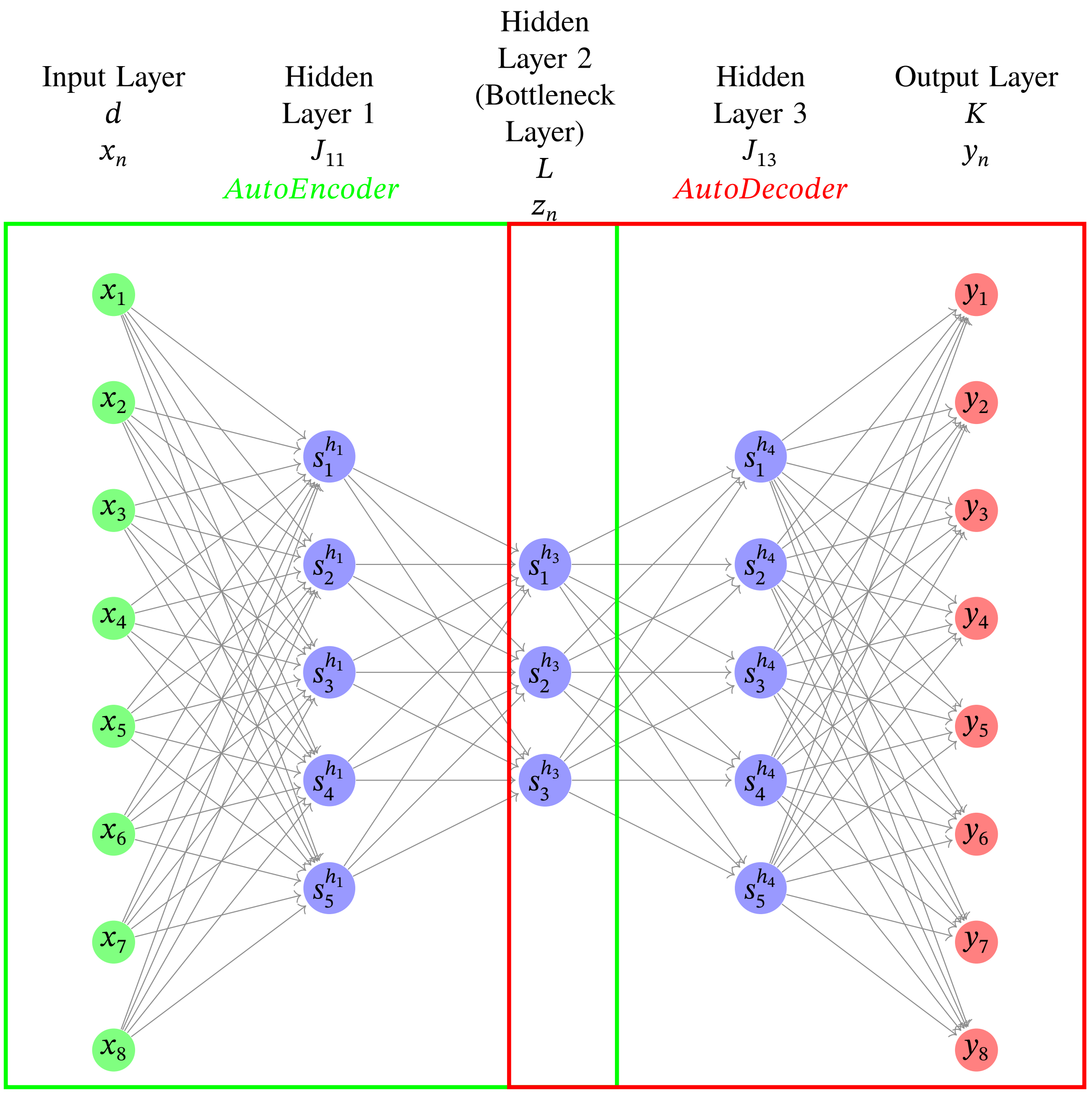

Input Layer
d
x_n
Hidden Layer 1
J_{11}
AutoEncoder
Hidden Layer 2 (Bottleneck Layer)
L
z_n
Hidden Layer 3
J_{13}
AutoDecoder
Output Layer
K
y_n
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
$s_1^{h_1}$ $s_2^{h_1}$ $s_3^{h_1}$ $s_4^{h_1}$ $s_5^{h_1}$
$s_1^{h_3}$ $s_2^{h_3}$ $s_3^{h_3}$
$s_1^{h_4}$ $s_2^{h_4}$ $s_3^{h_4}$ $s_4^{h_4}$ $s_5^{h_4}$
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8

Figure 22: An illustration of an AutoEncoder

3.1.2 Objective:

The primary objective of an autoencoder is to minimize the reconstruction error between the input data and the reconstructed data. It learns to reconstruct the input data as accurately as possible, encouraging the model to capture the most salient features of the data in the latent space.

3.1.3 Loss Function:

The loss function used in training autoencoders is typically a measure of the discrepancy between the input data and the reconstructed data. Mean Squared Error (MSE) or Cross-Entropy Loss are commonly used loss functions for reconstruction tasks. Note that here we have used MSE for pre-training and used CE for fine-tuning.

3.1.4 Applications:

Autoencoders has several applications.

- *Dimensionality Reduction:* Autoencoders can be used for dimensionality reduction by learning a lower-dimensional representation of high-dimensional data.
- *Data Denoising:* Autoencoders can learn to denoise data by reconstructing clean data from noisy input samples.
- *Feature Learning:* Autoencoders can learn useful representations or features from unlabeled data, which can be transferred to downstream supervised learning tasks.
- *Anomaly Detection:* Autoencoders can detect anomalies or outliers in data by measuring the reconstruction error, where large reconstruction errors indicate anomalies.

Autoencoders are typically trained using gradient descent-based optimization algorithms, such as stochastic gradient descent (SGD) or AdaM. They can be trained on unlabeled data using an unsupervised learning approach, where the model learns to reconstruct the input data without explicit labels. Autoencoders have become popular in various domains due to their ability to learn useful representations from data in an unsupervised manner. They serve as a foundational building block for many applications in machine learning and deep learning, including data compression, generative modeling, and representation learning.

3.2 Stacked AutoEncoders:

Stacked autoencoders, also known as deep autoencoders or multilayer autoencoders, are a type of neural network architecture composed of multiple layers of encoders and decoders stacked on top of each other. They are used for learning hierarchical representations of data by progressively reducing the dimensionality of the input through successive encoding and decoding layers. Stacked autoencoders extend the concept of traditional autoencoders by introducing multiple hidden layers, enabling the model to learn more complex and abstract representations of the input data.

3.2.1 Architecture:

Stacked autoencoders consist of an input layer, multiple hidden layers (encoder layers), and a symmetrically arranged set of decoder layers. Each encoder layer reduces the dimensionality of the input data, capturing higher-level features and patterns. Conversely, each decoder layer reconstructs the data from the encoded representation, progressively expanding the data back to its original dimension. Figure 23 shows the architecture of a 3-stacked autoencoder.

3.2.2 Hierarchical Representation Learning:

Stacked autoencoders are capable of learning hierarchical representations of the input data, where lower layers capture simple features (e.g., edges, textures), and higher layers capture more abstract and complex features (e.g., shapes, objects). By stacking multiple layers, the model can learn increasingly abstract representations of the input data, potentially capturing semantically meaningful features.

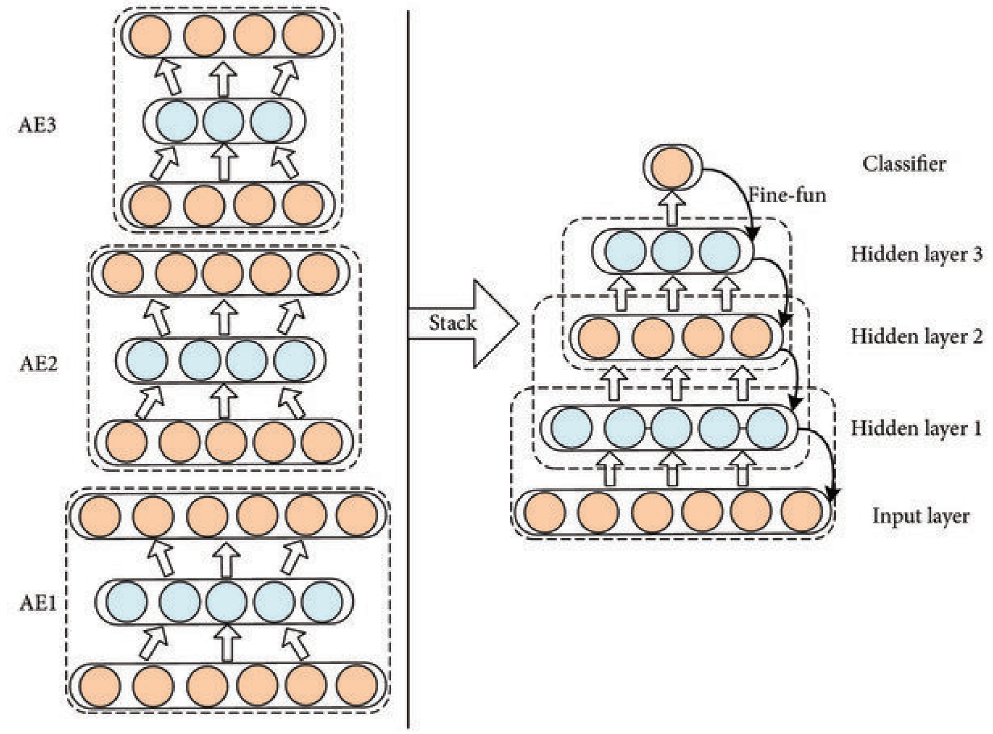


Figure 23: A 3− stacked AutoEncoder

3.2.3 Training:

Stacked autoencoders are typically trained using unsupervised learning techniques, where the objective is to minimize the reconstruction error between the input data and the reconstructed data. They are often pre-trained layer by layer using greedy layer-wise pretraining, where each layer is trained independently as a traditional autoencoder before fine-tuning the entire network. After pretraining, the entire stacked autoencoder can be fine-tuned using backpropagation and gradient descent-based optimization algorithms.

3.2.4 Applications:

Stacked Autoencoders have several applications

- *Feature Learning:* Stacked autoencoders are used for learning useful representations or features from high-dimensional data, which can be transferred to downstream supervised learning tasks.
- *Image Denoising:* They can be trained to denoise images by reconstructing clean images from noisy input samples, capturing underlying structures and patterns.
- *Dimensionality Reduction:* Stacked autoencoders can compress high-dimensional data into a lower-dimensional representation, enabling efficient storage and processing.

Stacked autoencoders have demonstrated success in various domains, including computer vision, natural language processing, and anomaly detection, among others. They are a powerful tool for learning hierarchical representations of data and extracting meaningful features in an unsupervised manner.

However, training stacked autoencoders can be computationally intensive and may require careful hyperparameter tuning to achieve optimal performance.

3.3 Plots

3.3.1 DFNN

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	30
# Neurons (HL2)	20
# Neurons (HL3)	10
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-6}

Result	Value
Converged at epoch #	733
Train Loss	0.8175
Train Accuracy	0.702
Test Loss	1.5201
Test Accuracy	0.476
Validation Loss	1.2672
Validation Accuracy	0.621

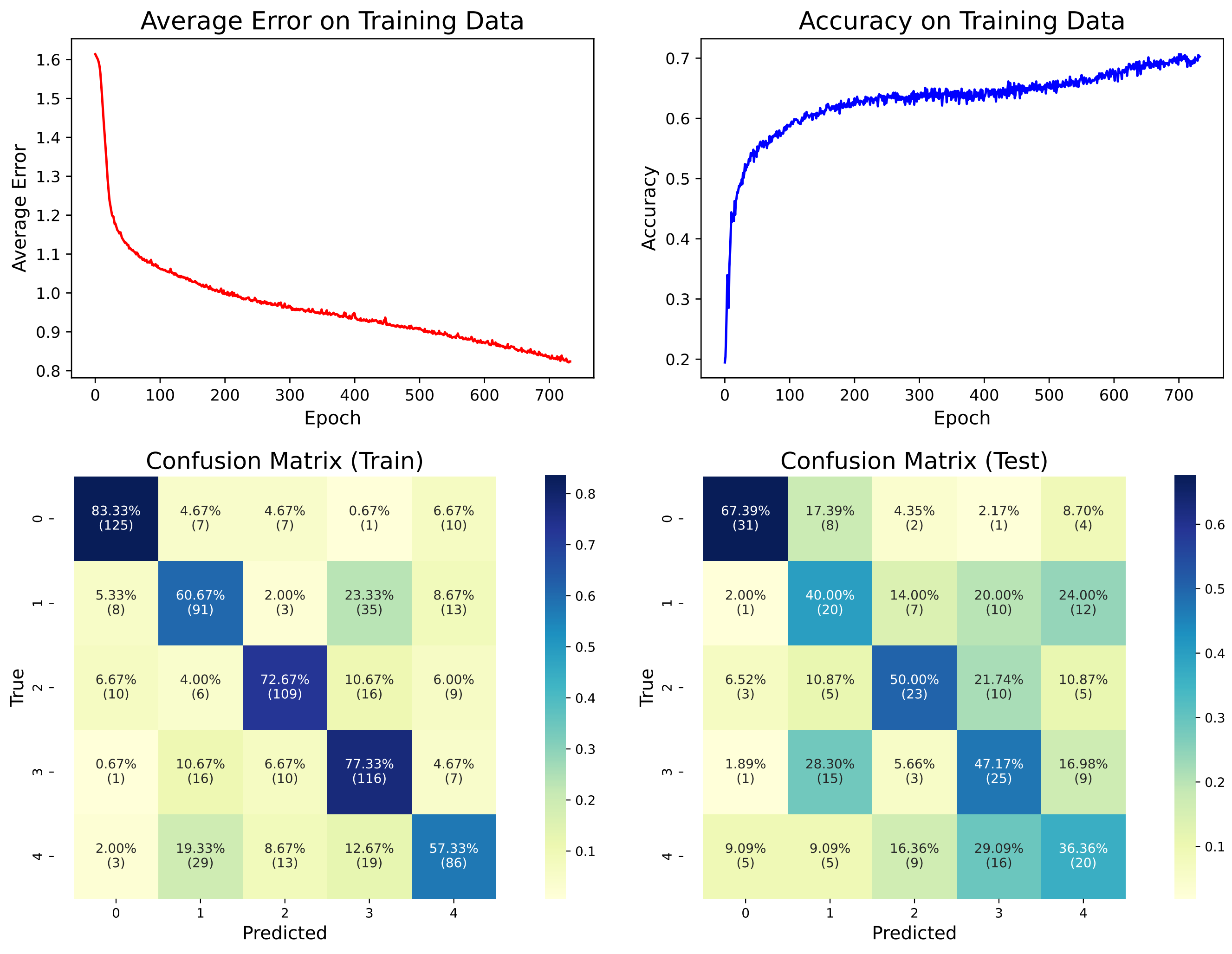


Figure 24: Plots for DFNN

3.3.2 DFNN SAE

Hyperparameters	Value
Learning Rate	0.01
# Neurons (Input)	36
# Neurons (HL1)	30
# Neurons (HL2)	20
# Neurons (HL3)	10
# Neurons (Output)	5
Total Epochs	10^4
Threshold	10^{-6}

Result	Value
Converged at epoch #	823
Train Loss	0.7122
Train Accuracy	0.725
Test Loss	1.4912
Test Accuracy	0.513
Validation Loss	1.2345
Validation Accuracy	0.661

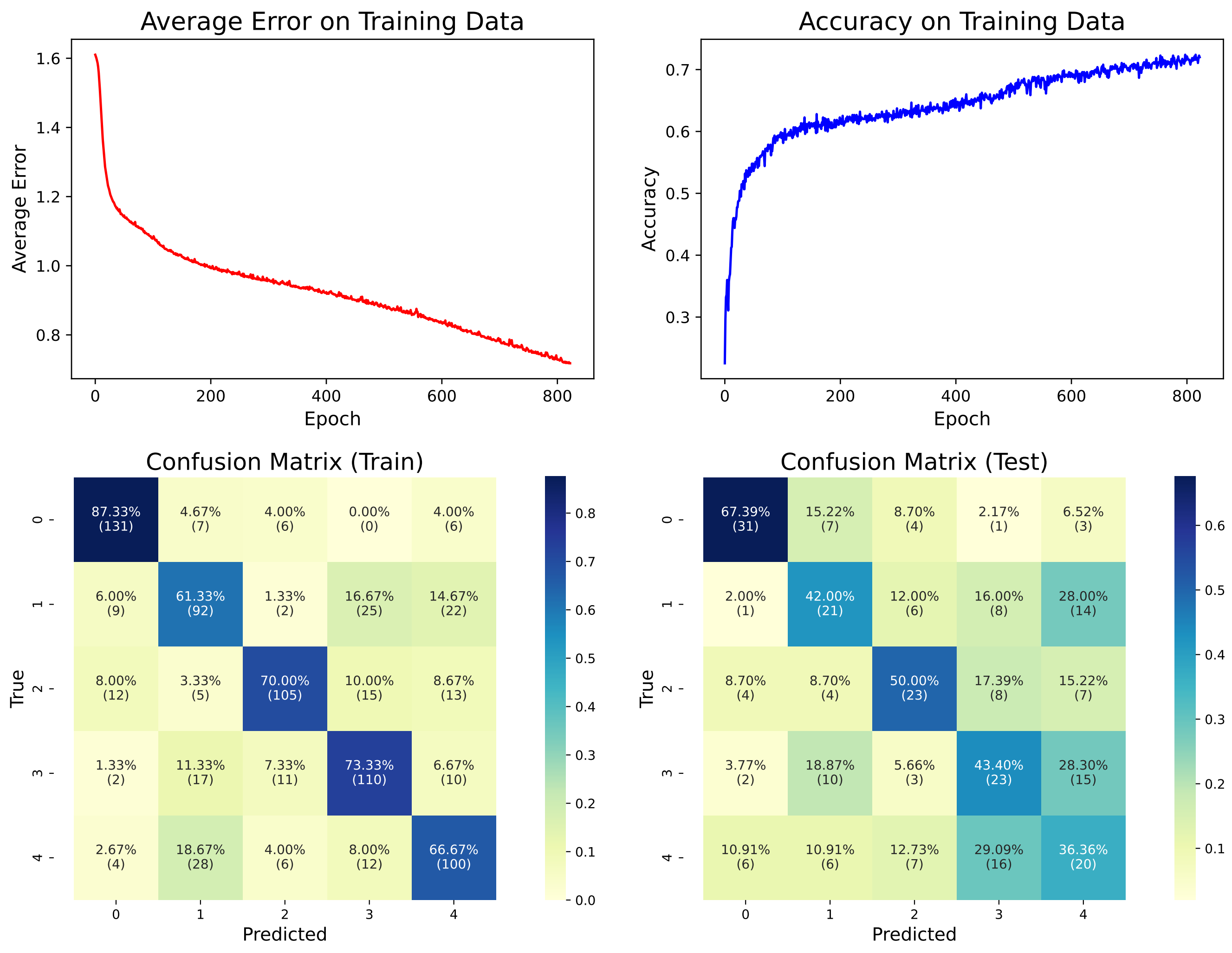


Figure 25: Plots for DFNN SAE

3.4 Tabulation

AANN	Converged at epoch #	Accuracy Training	Accuracy Validation	Accuracy Testing
DFNN	733	0.702	0.621	0.476
SAE	823	0.725	0.661	0.513

Table 3: A comparison between DFNN and SAE

3.5 Observations and Inferences

We observe that the DFNN using a stacked autoencoder for pretraining and then fine-tuning it with labelled data has overall better accuracy and lower training error as compared to the DFNN trained normally. Also, the graphs for the DFNN without autoencoder are far more jagged and stochastic in nature. We can thus infer the following:

- For the DFNN without autoencoder, the model is trained solely based on the existing data points, which leads to oscillatory behavior as the model tries to adapt to each point that is given. When the autoencoder is used, the model learns a general representation of the overall data and has an idea of the trends that are followed, hence there is not as much oscillation.
- When we use the DFNN without autoencoder, the initialization of weights and biases is done randomly and updates are done during the learning process. When an autoencoder is used, weights are pre-trained and allows for a better starting point when training, leading to lower variability and hence a smoother graph.
- Also, when an autoencoder is used, pre-training acts as a kind of regularization for the data and adds extra constraints by means of the patterns observed in the original data. This does not happen whtn only a DFNN is used and so there is a greater chance of overfitting the data, leading to fluctuations in performance and thus creating stochasticity in the graphs.