A Report on

CS671: Assignment 3

FCNN using different Optimizers

Team Members(Group 04):

Ananya Angra(S22025)
Shilpa Chandra(S22004)
Pushkar Kumar(S22038)

Course Instructor

Dr. Dileep A. D.

IIT Mandi Academic Year 2022-2023

Contents

List of Figures

Chapter 1

Optimizers

Optimizers are an essential component of the backpropagation algorithm, which is used to train artificial neural networks by updating the weights and biases to minimize the error between the predicted outputs and the actual outputs, which is a measure of how well the network is performing on the task given. There are many other optimizers available, and the choice of optimizer depends on the specific problem and network architecture. We will be trying several different optimizers and comparing their performance on a validation set to choose the best one for the task at hand.

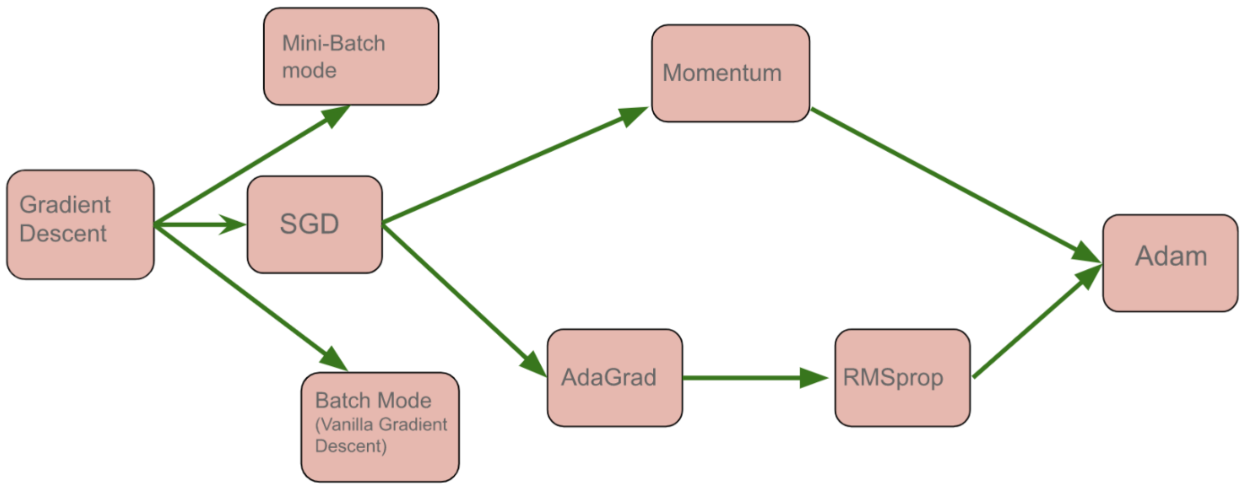


Figure 1.1: Different Optimizers

1.1 Stochastic gradient descent (SGD)

In this method the algorithm goes over the training data one by one and keeps updating the parameters after every iteration. The total gradient is dependent on a single data point, hence it provides an approximate gradient value. Every step does not guarantee a decrease in loss, therefore there will be oscillations before convergence as each step makes a greedy decision by pushing the parameters in the direction which is locally favorable for that point.
Weight update equation :

$$w(m+1) = w(m) + \Delta w(m)$$

Momentum-based Gradient Descent (Generalized Delta Rule)

- The learning rate η in backpropagation algorithm has to kept small in order to maintain a smooth trajectory in weight space
- Large learning rate can lead to oscillations during learning.
- Gradient descent algorithm takes a lot of time to navigate regions having a gentle slope. This is because the gradient in these regions is very small.
- Hence **momentum** term is introduced, to increasing the rate of learning while maintaining stability.

weight update equation : $\Delta w(m) = -\eta \nabla w(m) + \alpha \Delta w(m-1)$

$$w(m+1) = w(m) + \Delta w(m)$$

Nesterov Accelerated Gradient Descent (NAG) The Nesterov accelerated gradient (NAG) optimizer is a popular optimization algorithm used in deep learning. It is an extension of the standard gradient descent algorithm that uses a momentum term to accelerate convergence. The NAG optimizer works by calculating the gradient of the loss function with respect to the model's parameters, and then updating the parameters by subtracting a fraction of the gradient multiplied by the learning rate. The momentum term is then used to adjust the step size based on the previous gradient descent steps. weight update equation :

$$w_{lookahead} = w(m) + \alpha \Delta w(m-1)$$

$$\Delta w(m) = \alpha \Delta w(m-1) - \eta \nabla w_{lookahead}$$

$$w(m+1) = w(m) + [\alpha \Delta w(m-1) - \eta \nabla w_{lookahead}]$$

where $\Delta w(m)$ is the change in weight at the mth iteration, α is the learning rate, η is the momentum coefficient, and $\nabla w_{\text{lookahead}}$ is the gradient at the lookahead point.
The Nesterov Accelerated Gradient (NAG) method uses a lookahead point to help correct the course of weight updates quicker than momentum-based gradient descent, leading to smaller oscillations.

1.2 Batch gradient descent algorithm (vanilla gradient descent)

- In batch mode, algorithm goes over the entire trainingdata once before updating the parameters
- Update rule is based on true gradient of the loss
  - True gradient: sum of the gradients of the losses(errors) corresponding to each data point
- Since there is no approximation, each step guaranteesthat the loss will decrease
- Major **disadvantage** with batch is that algorithm makes huge calculations to make one update which leads to very slow convergence

1.3 Adagrad Optimizer

This optimizer is based on the idea that having different learning rates for different parameters will help speed up the process of parameter update. In this , we adapt the learning rates explicitly for each parameter. These approaches are designed for batch and mini-batch gradient descent rather than stochastic gradient descent (SGD). This is because errors in SGD can get magnified. Decay the learning rate for parameters in proportion to their update history. For parameter associated with dense feature, learning rate will decrease aggressively as compared to the sparse features as the update history for the dense features is large and for the sparse feature is less.

Adagrad acts like accelerated SGD as it helps in moving faster towards the solution.

Weight parameter update rule at iteration m:

$$v_l(m) = v_l(m-1) + (\nabla w_l(m))^2$$
$$w_l(m+1) = w_l(m) - \frac{\eta}{\sqrt{v_l(m) + \epsilon}} * \nabla w_l(m)$$

where $v_l(m)$ is the update history variable for an l^{th} parameter(running estimate of square of the gradient of current weight)
L is total number of parameters
η is small positive value such as 10^{-8}
Limitations of Adagrad: As Adagrad decays the learning rate very aggressively so it sometimes prematurely become very slow It has been observed that over the iterations, large history for dense features are accumulated and dividing with large history makes the learning rate to decay at such an extent that there will not be any updates and the update prematurely becomes slow.

1.4 RMSprop Optimizer

To overcome the limitation of adagrad i.e. the premature slowing of the updates due to the division by a large update history RMSprop decays the learning rate for parameters and at the same time avoid rapid growth of update history of gradient. Weight parameter update rule at iteration m:

$$v_l(m) = \beta v_l(m-1) + (1-\beta)(\nabla w_l(m))^2$$
$$w_l(m+1) = w_l(m) - \frac{\eta}{\sqrt{v_l(m) + \epsilon}} * \nabla w_l(m)$$

where $v_l(m)$ is the exponential moving average value, it is the smooth running estimate
L is total number of parameters
η is small positive value such as 10^{-8}

Limitations of RMSprop: Since the running estimate of squared gradient is initialised to 0 some undesirable bias is added to second order moment in the early iterations, which disappear over longer iterations.

1.5 Adam Optimizer

The goal of ADAM is to do everything that RMSProp does to solve the decay problem in AdaGrad.
Update the parameter by combining momentum and rate (adaptive learning rate) Weight parameter update rule at iteration m:

$$u_{\mathrm{l}}(m) = \beta_1 u_{\mathrm{l}}(m-1) + (1-\beta_1)(\nabla w_{\mathrm{l}}(m))$$

$$v_{\mathrm{l}}(m) = \beta_2 v_{\mathrm{l}}(m-1) + (1-\beta_2)(\nabla w_{\mathrm{l}}(m))^2$$

$$\hat{u}_l = \frac{u_{\mathrm{l}}(m)}{1-\beta_1^m}$$

$$\hat{v}_l = \frac{v_{\mathrm{l}}(m)}{1-\beta_2^m}$$

$$w_l(m+1) = w_l(m) - \frac{\eta}{\sqrt{\hat{v}_l(m)+\epsilon}} * \hat{u}_l$$

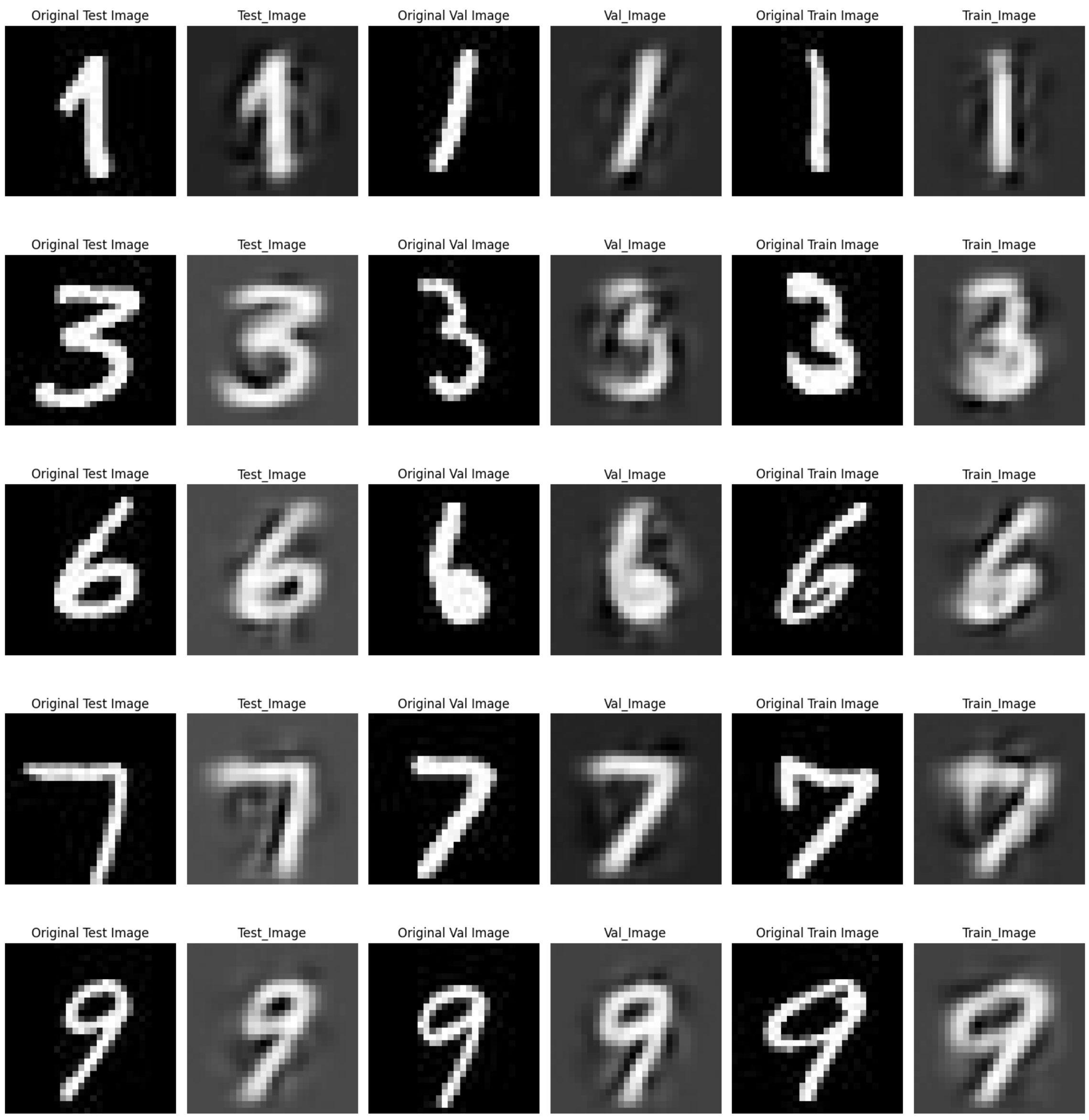
where $v_{\mathrm{l}}(m)$ is the exponential moving average value, it is the smooth running estimate
L is total number of parameters
η is small positive value such as 10^{-8}
Update rule is a combination of momentum and decaying learning rate, hence the speed up in converging to solution

Chapter 2

Task 2 Results

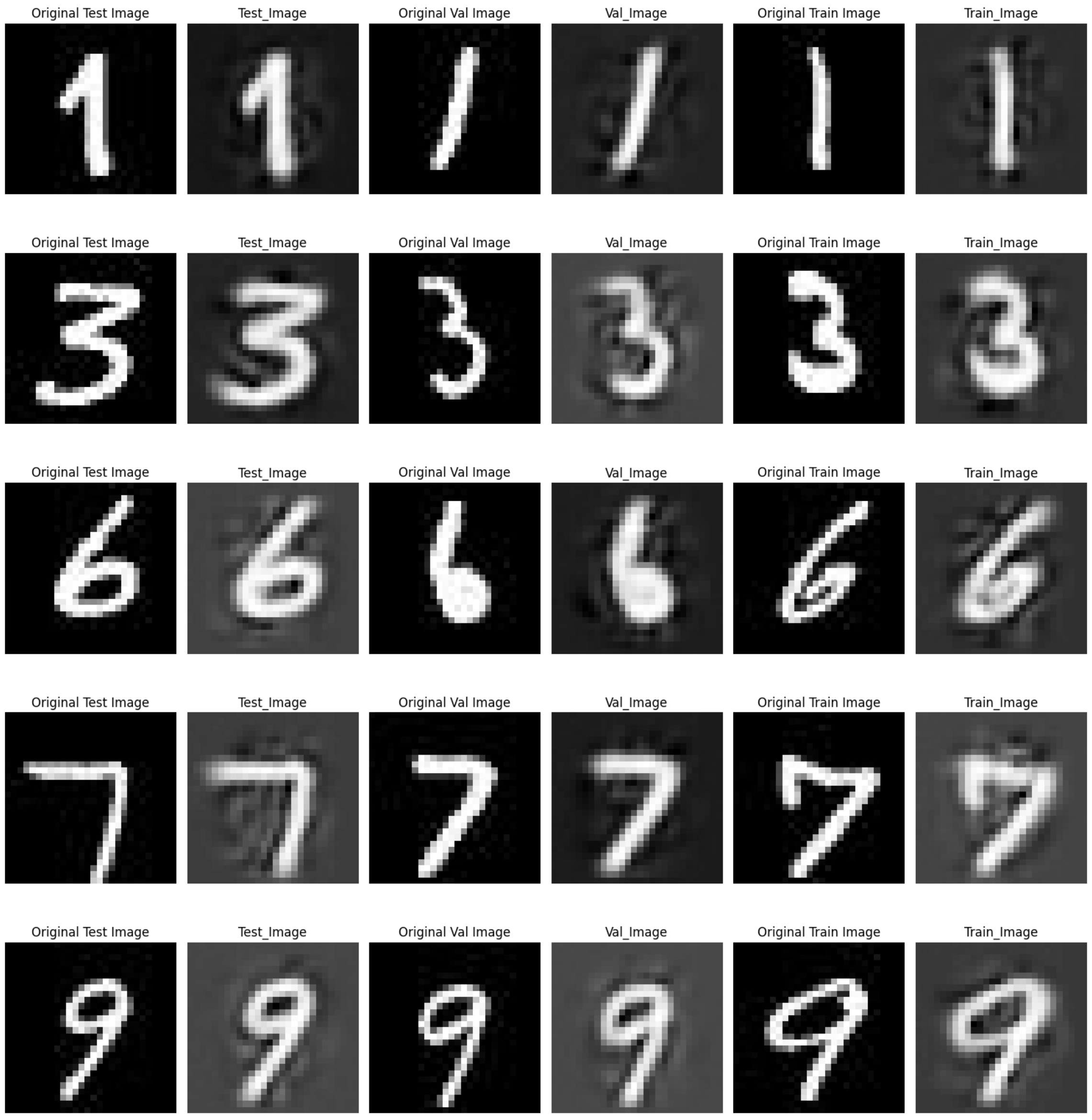
Autoencoders with 1 hidden layer:

- **With 32 neurons in the bottleneck layer**

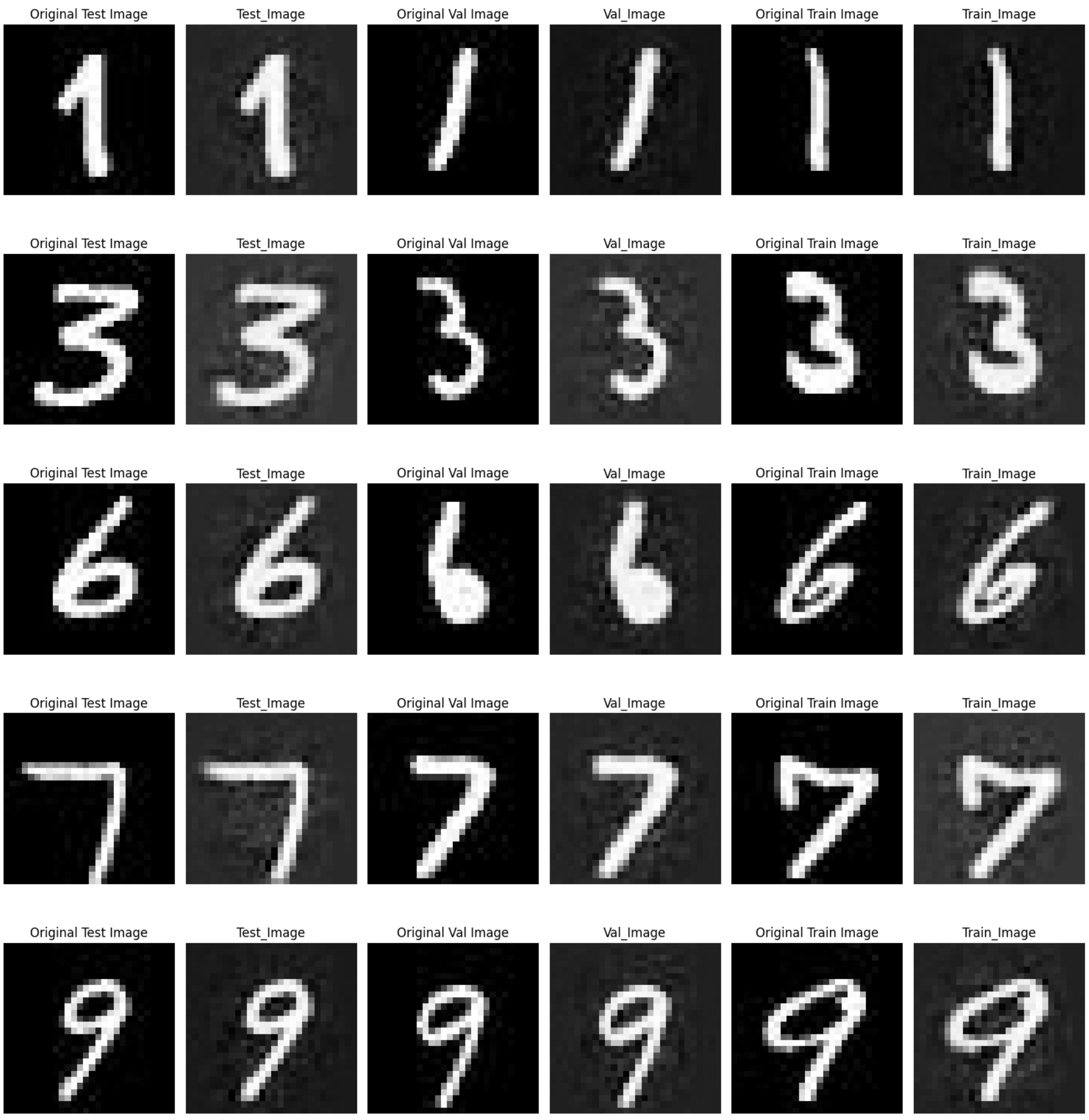


Reconstruction errors for training validation test data:

- **With 64 neurons in the bottleneck layer**



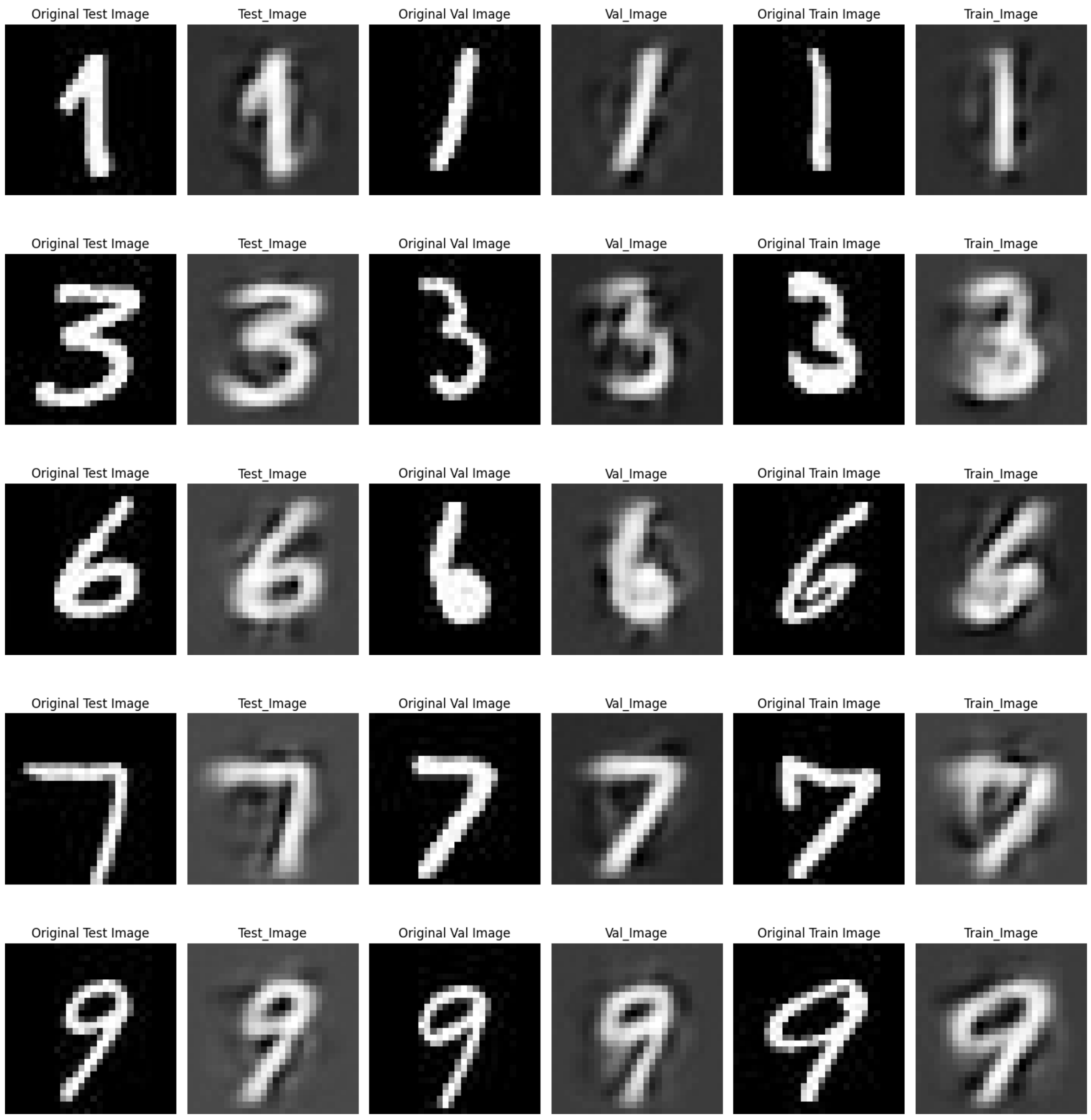
- **With 128 neurons in the bottleneck layer**
- **With 256 neurons in the bottleneck layer**



Reconstruction errors for training validation test data:

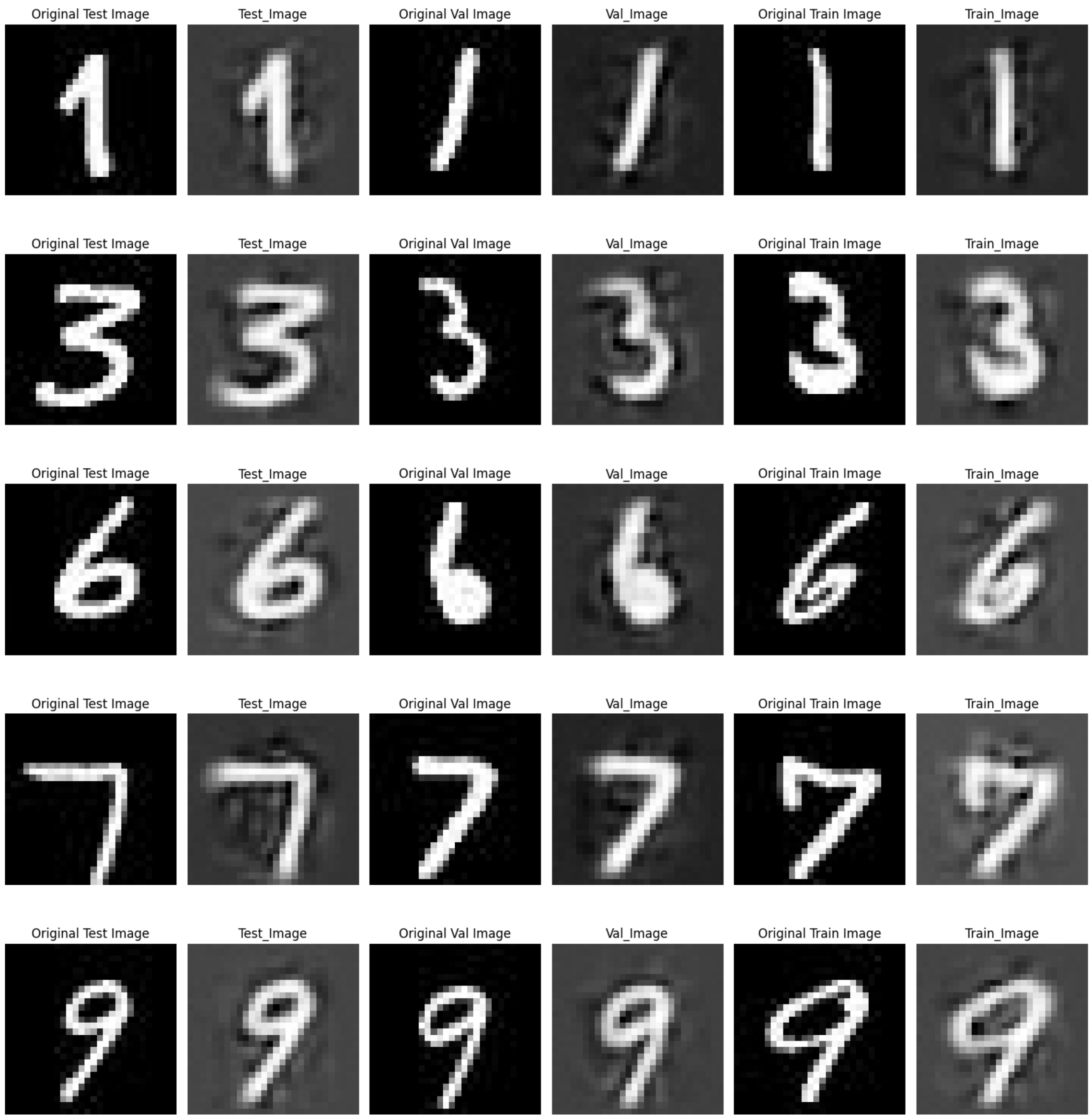
2.1 Autoencoders with 3 hidden layer:

- **With 32 neurons in the bottleneck layer**

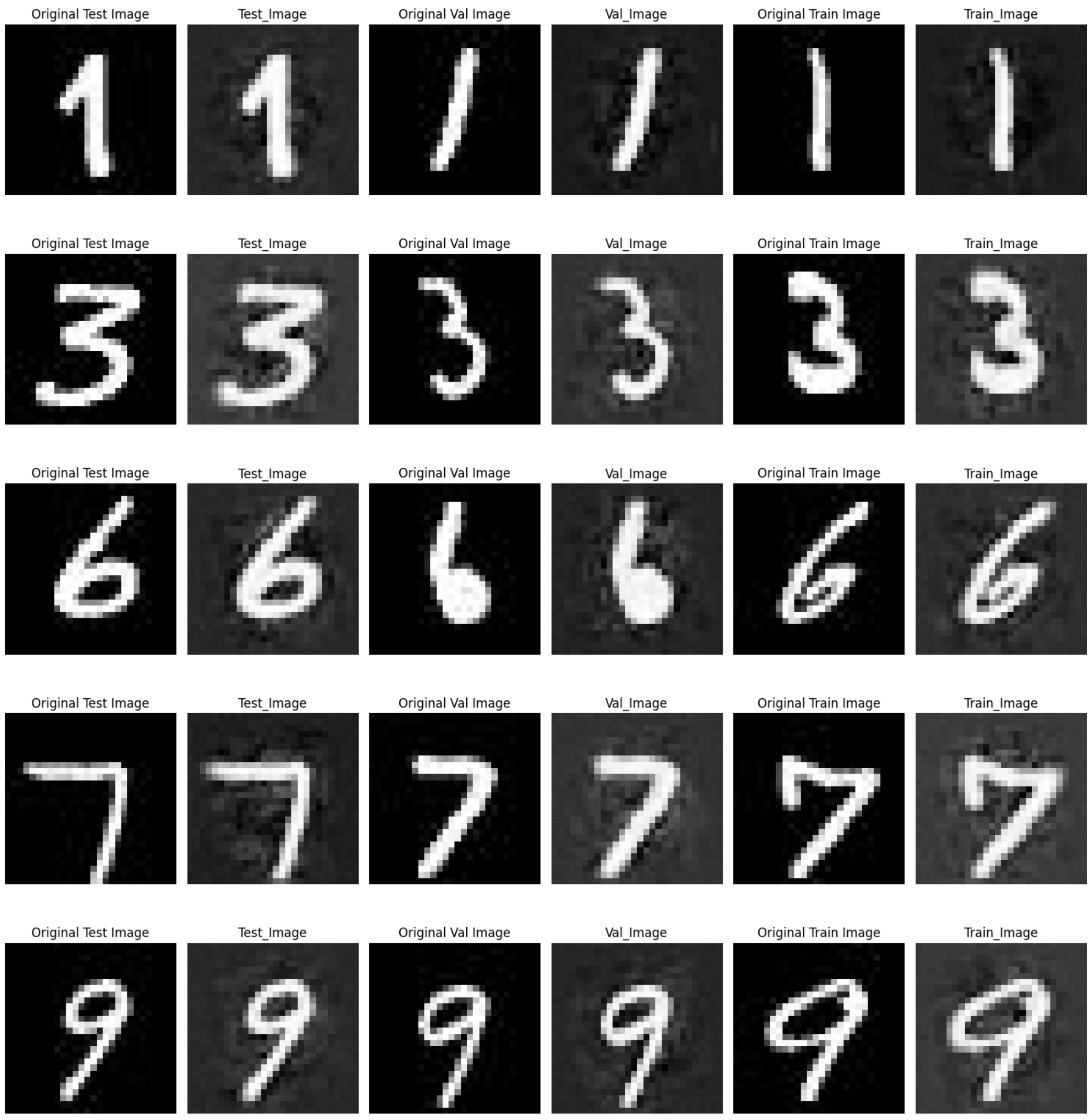


Reconstruction errors for training validation test data:

- **With 64 neurons in the bottleneck layer**



- **With 128 neurons in the bottleneck layer**
- **With 256 neurons in the bottleneck layer**


Original Test Image
Test_Image
Original Val Image
Val_Image
Original Train Image
Train_Image

Chapter 3

References

[1] http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

[2] Lecture notes of CS671 Feb 2023-24 by Dr.Dileep A D