



CSE473s (UG2018) - Computational Intelligence
Build Your Own Neural Network Library & Advanced Applications
Final Submission

Team members

Name	ID
Mariam Ashraf Mahmoud	2100849
Omnia Said Mohamed	2100742
Habiba Khaled Ahmed	2100656
Shahd Essam Ahmed	2100659
Shahd Amir Yehia	2100367

Table of Contents

Library design and architecture choices.....	3
Layers library:	3
Activation Layer Library;.....	5
Losses library:	6
Optimizer library:	7
Network library:	8
Results from the XOR test.	8
Analysis of the autoencoder's reconstruction quality	10
Encoder design choice.....	10
Decoder design choice.....	10
A detailed analysis of your SVM classification results. Discuss the quality of the latent space features your encoder learned	11
A summary of your TensorFlow comparison.	13

Library design and architecture choices.

Layers library:

```
import numpy as np
#Layer Abstraction section
class Layer:
    def forward(self, x):
        pass

    def backward(self, grad):
        pass

    def update(self, lr):
        pass
```

This is the parent class for all neural network layers.

Every layer in your library (Dense, Activation, etc.) must implement:

forward(x)

- Used during forward propagation
- Computes the output of this layer

backward(grad)

- Used during backpropagation
- Receives gradient from the next layer
- Returns gradient for the previous layer

update(lr)

- Used by optimizers to update parameters
- Only layers with weights (like Dense) will use it as input and output layers don't have parameters

```
#dense layer has inputs from neurons and outputs to other neurons, only layer with
parameters (w)
class Dense(Layer):
    #in_features is the size of a single sample
    #out_features is the no. of neurons
    def __init__(self, in_features, out_features, init_scale):
        #random values for weights, randn gives normal distribution (mean 0, std 1)
        #the weights are reversed so that we don't need to perform transpose on the
weights matrix
        self.W = np.random.randn(in_features, out_features) * init_scale
        #biases are initialized to zero
        self.b = np.zeros((1, out_features))
        self.x = None
        self.dW = None
```

```

self.db = None

def forward(self, x):
    #Save this input x inside the layer object so backward() can use it later
    self.x = x
    #it multiplies x by weights then adds bias, y=xw+b
    return np.dot(x, self.W) + self.b

def backward(self, grad_out):
    # Gradients
    #dL/dW = x T * dL/dy
    self.dW = self.x.T @ grad_out
    #sum of output gradients
    self.db = np.sum(grad_out, axis=0, keepdims=True)
    #dL/dX = dL/dy * W T
    grad_input = grad_out @ self.W.T
    return grad_input

def update(self, lr):
    #lr is the learning rate
    self.W -= lr * self.dW
    self.b -= lr * self.db

```

Weights (W) : They are random initialized to break symmetry, creates a matrix of shape **[in_features, out_features, init_scale]**

randn gives normal distribution (good for small networks like XOR) and it is multiplied by a variable called init_scale to be flexible (not hardcoded) as in XOR it worked better with leaving init_scale=1 while in autoencoder it worked better with init_scale being 0.001 as autoencoder were very sensitive to large weights and it's far more complicated than XOR network.

Bias (b) : Each neuron has a bias with shape **[1, out_features]**

The forward pass computes the output: $Y = XW + b$

The backward pass this is the most important part.

Input to backward: grad_out $\frac{dL}{dy}$

We calculate gradient with respect to weights $\frac{dL}{dW} = x^T \frac{dL}{dy}$

Then we calculate gradient with respect to bias (Bias affects output directly, so we sum over samples) $\frac{dL}{db} = \sum \frac{dL}{dy}$

Then gradient with respect to input x (sent to previous layer) $\frac{dL}{dx} = \frac{dL}{dy} W^T$

Finally return gradient for next layer by :return grad_input

SGD (stochastic gradient descent) update rule to update weights of dense layer:

$$W_{n+1} = W_n - \eta \frac{dl}{dW}$$

$$b_{n+1} = b_n - \eta \frac{dl}{db}$$

lr = learning rate

dW, db are gradients computed in backward pass

Activation Layer Library;

➤ **Sigmoid**

Non-linear → helps learning

Range (0, 1)

➤ **Tanh**

Even better → symmetric around 0

Range (-1, 1)

This helps hidden layer neurons capture XOR's shape.

➤ **ReLU**

Only outputs positive values → *not useful for XOR hidden layer*, but still a fundamental activation for bigger networks.

Sigmoid activation:

```
class Sigmoid:
    def forward(self, x):
        self.out = 1 / (1 + np.exp(-x))
        return self.out
#i need to save sigmoid value to use it while calculating its gradient
#dL/dx = dL/dy * (gradient of sigmoid)
    def backward(self, grad):
        return grad * (self.out * (1 - self.out))
```

Sigmoid formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

We return self.out in the forward pass to save the value of $\sigma(x)$ as we'll need while calculating the gradient of sigmoid

$$\sigma' = \frac{e^{-x}}{(1+e^{-x})^2} = \sigma(1 - \sigma)$$

Then in the backward pass $\frac{dL}{dx} = \frac{dL}{dy} * \dot{\sigma} = \frac{dL}{dy} * \sigma(1 - \sigma)$

Tanh activation:

```
class Tanh:
    def forward(self, x):
        self.out = np.tanh(x)
        return self.out
#derivative of tanh(x)= 1- (tanh(x)^2)
    def backward(self, grad):
        return grad * (1 - self.out**2)
```

Tanh formula:

$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh = 1 - (\tanh)^2$$

In the backward pass, we calculate $\frac{dL}{dx}$

$$\frac{dL}{dx} = \frac{dL}{dy} * 1 - (\tanh)^2$$

Losses library:

```
import numpy as np
#mean squared error to calculate loss
class MSE:
    #Loss= (1/(2*n)) sum(y - y^)^2
    def forward(self, y_true, y_pred):
        return np.mean((y_true - y_pred)**2)

    def backward(self, y_true, y_pred):
        return 2 * (y_pred - y_true) / y_true.size
```

We calculated the loss by the Mean Squared Error (MSE):

$$\xi = \frac{1}{N} \sum (y - \hat{y})^2$$

In backward pass: $\dot{\xi} = \frac{2}{N} \sum (y - \hat{y})$

```
class BCE:
    def forward(self, y_pred, y_true):
        eps = 1e-8
        y_pred = np.clip(y_pred, eps, 1 - eps)
        self.y_pred = y_pred
        self.y_true = y_true
        return -np.mean(
```

```

        y_true * np.log(y_pred) +
        (1 - y_true) * np.log(1 - y_pred)
    )

def backward(self, y_pred, y_true):
    eps = 1e-8
    y_pred = np.clip(y_pred, eps, 1 - eps)
    return (y_pred - y_true) / (y_pred * (1 - y_pred) * y_true.shape[0])

```

We also have BCE (binary cross entropy loss)

$$\xi = \frac{-1}{N} \sum (y \log(\hat{y}) + (1 - y) \log(1 - \hat{y}))$$

It's mostly used with multi class classification, we used i

Optimizer library:

The optimizer is the component responsible for updating the weights and biases of your network after backpropagation computes gradients.

```

class SGD:
    #stochastic gradient descent
    #sets learning rate for gradient descent
    def __init__(self, lr=0.5):
        self.lr = lr

#only updates parameters in dense layers , checks if layer has attribute of update
def step(self, layers):
    for layer in layers:
        if hasattr(layer, "update"):
            layer.update(self.lr)

```

init sets the learning rate (lr) here we assumed it be 0.5

We update weights and biases using these formulas:

$$W_{n+1} = W_n - \eta \frac{dl}{dW}$$

$$b_{n+1} = b_n - \eta \frac{dl}{db}$$

Lastly, the step function :

- Loops through all layers in the network
- Only updates layers that *have parameters* (i.e., Dense layers)
- Calls their update(lr) method to change W and b in each iteration.

We added Adam optimizer which was very helpful in training the autoencoder as SGD was ineffective and the autoencoder failed, while Adam made the losses very small and the input images match the reconstructed images as its lr (learning rate) is adaptive.

Network library:

```
class Network:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def backward(self, grad):
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self, optimizer):
        optimizer.step(self.layers)
```

This is the most important library in the project, because it is the brain that connects:

- Dense layers
- Activation layers
- Loss
- Optimizer

The init function: Passes a list of layers (Dense + activation layers), The network stores them in order

The forward pass: sends data through each layer one-by-one.

The backward pass: reverses gradients flow backwards

Loss → Output layer → Hidden layer → Input layer

That's what `reversed(self.layers)` does.

Results from the XOR test.

```
import numpy as np

# XOR dataset
X = np.array([[0,0],
              [0,1],
              [1,0],
              [1,1]])
```

```

y = np.array([[0],
              [1],
              [1],
              [0]])

# Build network: 2 → 4 → 1
# 2 input neurons (x1,x2), 4 hidden neurons, 1 output neuron
model = Network([
    Dense(2, 4),
    #tanh provide non linearity to the 4 hidden neurons
    Tanh(),
    #takes 4 inputs from the hidden layer and outputs 0 or 1
    Dense(4, 1),
    #sigmoid outputs values between 0 and 1
    Sigmoid()
])

loss_fn = MSE()
optimizer = SGD(lr=0.1)

# Training loop, 5000 iterations
for epoch in range(5000):
    pred = model.forward(X) #y^
    loss = loss_fn.forward(y, pred)
    grad = loss_fn.backward(y, pred) #dL/dy^
    model.backward(grad) #backpropagate through the network
    model.update(optimizer)
#print loss each 50 iterations
    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss = {loss:.5f}")

# Final predictions
print("\nFinal Predictions:")
print(model.forward(X))

```

The results:

We can see the loss is decreasing by each iteration

Epoch 0, Loss = 0.40811
 Epoch 500, Loss = 0.13989
 Epoch 1000, Loss = 0.02369
 Epoch 1500, Loss = 0.00983
 Epoch 2000, Loss = 0.00588
 Epoch 2500, Loss = 0.00411
 Epoch 3000, Loss = 0.00313
 Epoch 3500, Loss = 0.00252
 Epoch 4000, Loss = 0.00210
 Epoch 4500, Loss = 0.00179

Final Predictions:

[[0.02249629]

[0.9540998]

[0.96065617]

[0.04571082]]

As we can see the loss is decreasing with every epoch and the predictions are correct which shows that we succeeded in solving XOR using our libraries from scratch.

Analysis of the autoencoder's reconstruction quality

Encoder design choice

We built our encoder using the neural networks layers we created, the input to the encoder is 60,000 images from MNIST datatest for training and 10,000 images for testing.

The input dimiension for the encoder is 784 as the images are originally 28x28 pixels which will be flattened to a 784 vector and normalized.

The network of the encoder consists of 4 dense layers, 3 ReLU activations functions and the last dense function with no activation layer as we want linear latent representation.

It does the following to images :

$784 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 64$

Decoder design choice

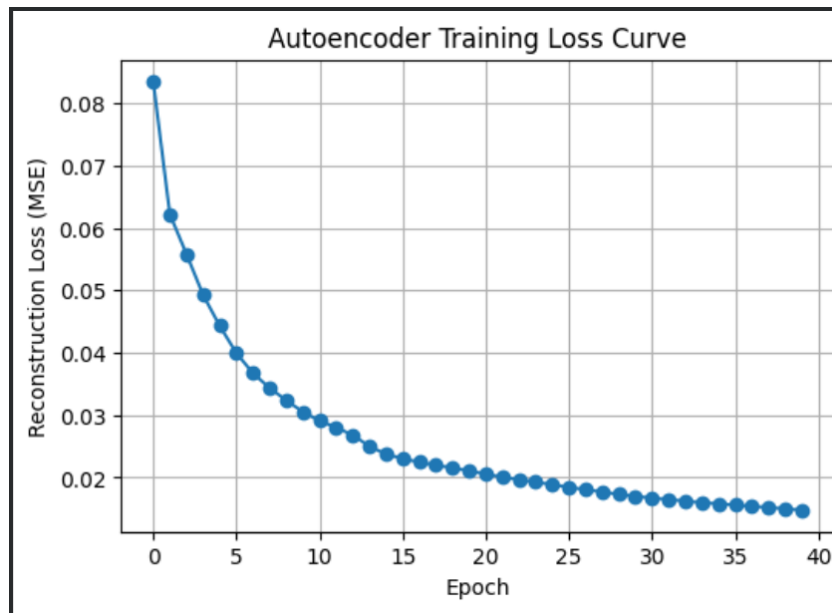
The decoder is also a neural network layer made of 3 Dense layers, 3 ReLU activation functions and a Sigmoid activation function in the last dense layer as it works best with the loss we used which BCE.

$64 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 784$

The reconstruction images look very similar to the original image which shows that the autoencoder succededed, the top row reoresents the original images while the bottom row is the reconstructed images



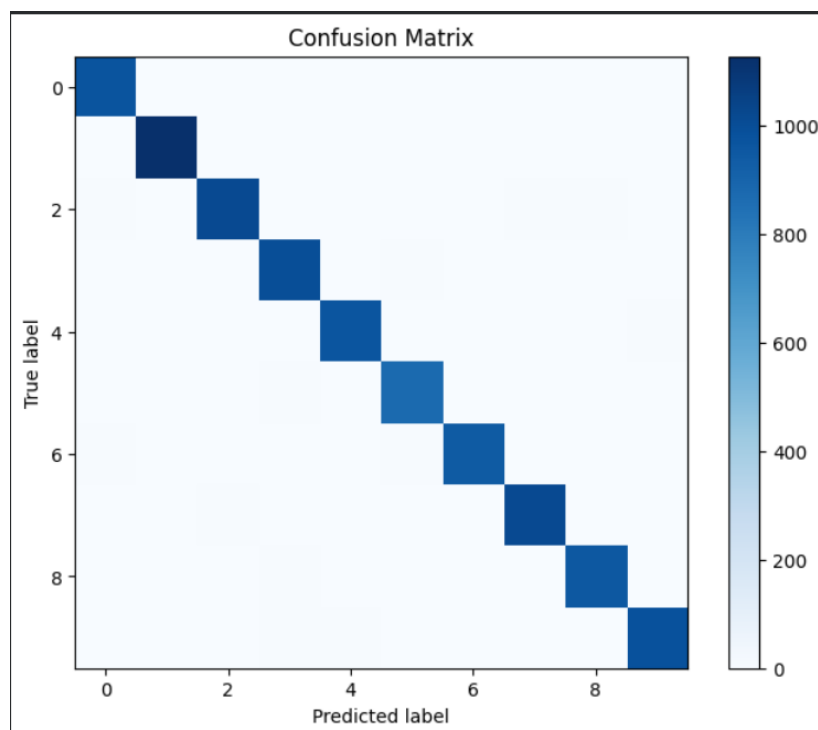
Loss curve:



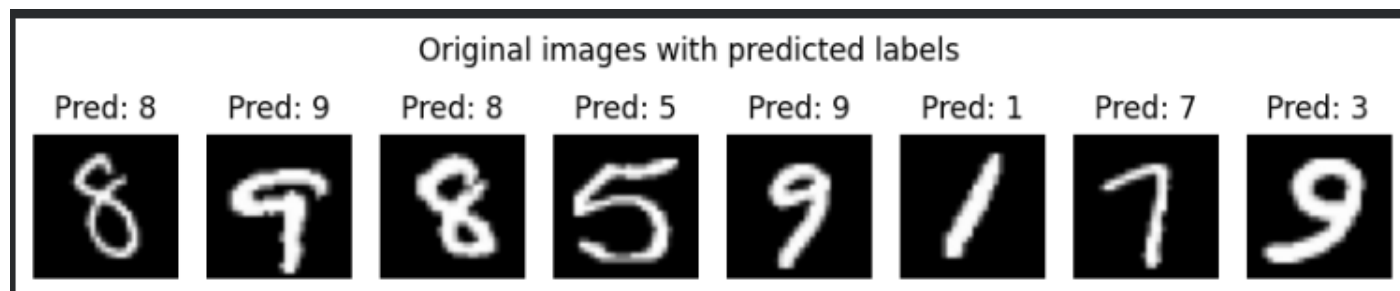
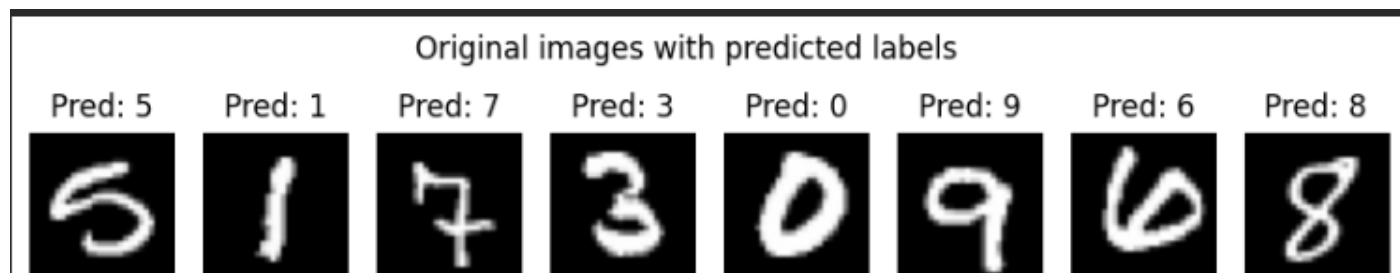
A detailed analysis of your SVM classification results. Discuss the quality of the latent space features your encoder learned

The SVM classifier was able to classify the images into the correct digits with very minimal error rate and an accuracy of 0.9843 with respect to the labels of the images

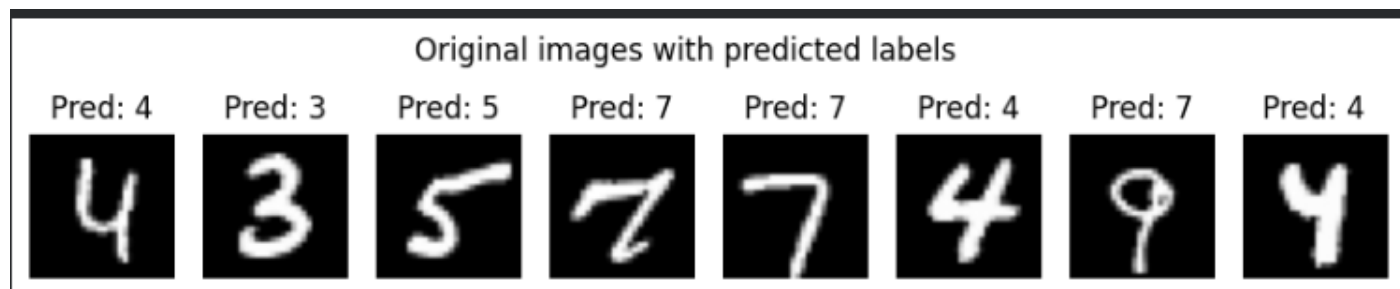
The confusion matrix



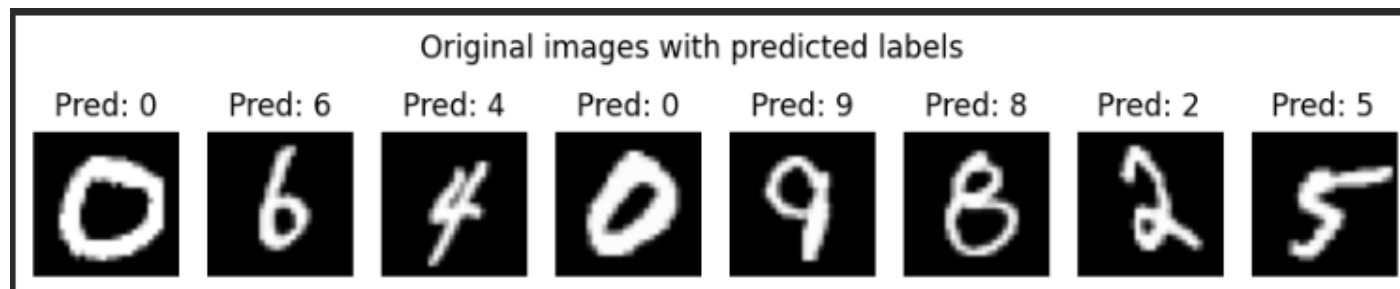
Predictions:



Here it didn't predict 9



It didn't predict 9



A summary of your TensorFlow comparison.

	Using the libraries	Tensorflow
XOR	Final Predictions: [[0.02391272] [0.95249545] [0.95406858] [0.05189731]]	XOR predictions: [[0.4277513] [0.5894922] [0.49505532] [0.5218849]]
Autoencoder	Epoch 1/40, Loss: 0.0842 Epoch 2/40, Loss: 0.0597 Epoch 3/40, Loss: 0.0537 Epoch 4/40, Loss: 0.0491 Epoch 5/40, Loss: 0.0444 Epoch 6/40, Loss: 0.0418 Epoch 7/40, Loss: 0.0396 Epoch 8/40, Loss: 0.0367 Epoch 9/40, Loss: 0.0342 Epoch 10/40, Loss: 0.0323 Epoch 11/40, Loss: 0.0308 Epoch 12/40, Loss: 0.0296 Epoch 13/40, Loss: 0.0286 Epoch 14/40, Loss: 0.0273 Epoch 15/40, Loss: 0.0260 Epoch 16/40, Loss: 0.0252 Epoch 17/40, Loss: 0.0246 Epoch 18/40, Loss: 0.0240 Epoch 19/40, Loss: 0.0233 Epoch 20/40, Loss: 0.0227	Epoch 1/20 235/235 _____ _____ 10s 36ms/step - loss: 0.0756 - val_loss: 0.0223 Epoch 2/20 235/235 _____ _____ 10s 41ms/step - loss: 0.0196 - val_loss: 0.0142 Epoch 3/20 235/235 _____ _____ 11s 42ms/step - loss: 0.0135 - val_loss: 0.0112 Epoch 4/20 235/235 _____ _____ 10s 43ms/step - loss: 0.0107 - val_loss: 0.0091 Epoch 5/20 235/235 _____ _____ 9s 37ms/step - loss: 0.0091 - val_loss: 0.0079 Epoch 6/20 235/235 _____ _____

10s 41ms/step - loss: 0.0079 -
val_loss: 0.0071
Epoch 7/20
235/235 —————

10s 41ms/step - loss: 0.0071 -
val_loss: 0.0066
Epoch 8/20
235/235 —————

11s 42ms/step - loss: 0.0064 -
val_loss: 0.0060
Epoch 9/20
235/235 —————

8s 35ms/step - loss: 0.0060 -
val_loss: 0.0056
Epoch 10/20
235/235 —————

11s 39ms/step - loss: 0.0056 -
val_loss: 0.0052
Epoch 11/20
235/235 —————

11s 44ms/step - loss: 0.0053 -
val_loss: 0.0050
Epoch 12/20
235/235 —————

19s 40ms/step - loss: 0.0050 -
val_loss: 0.0048
Epoch 13/20
235/235 —————

8s 36ms/step - loss: 0.0048 -
val_loss: 0.0047
Epoch 14/20
235/235 —————

10s 41ms/step - loss: 0.0046 -
val_loss: 0.0044
Epoch 15/20

	235/235 ————— <hr/> 10s 43ms/step - loss: 0.0044 - val_loss: 0.0044 Epoch 16/20 235/235 ————— <hr/> 9s 36ms/step - loss: 0.0043 - val_loss: 0.0042 Epoch 17/20 235/235 ————— <hr/> 10s 37ms/step - loss: 0.0041 - val_loss: 0.0040 Epoch 18/20 235/235 ————— <hr/> 10s 41ms/step - loss: 0.0040 - val_loss: 0.0039 Epoch 19/20 235/235 ————— <hr/> 10s 42ms/step - loss: 0.0038 - val_loss: 0.0038 Epoch 20/20 235/235 ————— <hr/> 8s 35ms/step - loss: 0.0037 - val_loss: 0.0040 Final reconstruction loss (Keras): 0.0037123048678040504
--	---

Of course Tensorflow is reasier in implementation and ready to use and much faster.
However, building your own libraries from scratch makes you grasp machine learning and neural networks.

Challenges faced and lessons learned.

In XOR, it was important that the weight be randomly generated and large enough to give correct predictions

In autoencoders, the reconstructed images were very bad and looked nothing like the original images when the optimizer was SGD and also when the weights were large it also gave non sense reconstructed images and the autoencoder failed and generated the same blurry image for all input images but likely when we change to another optimizer Adam which provides adaptive learning rate, the reconstructed images became very similar to the input ones.