Neural Process for Image Completion:

A model-based approach for filling in missing regions in images using deep learning.

Aryan Sahoo

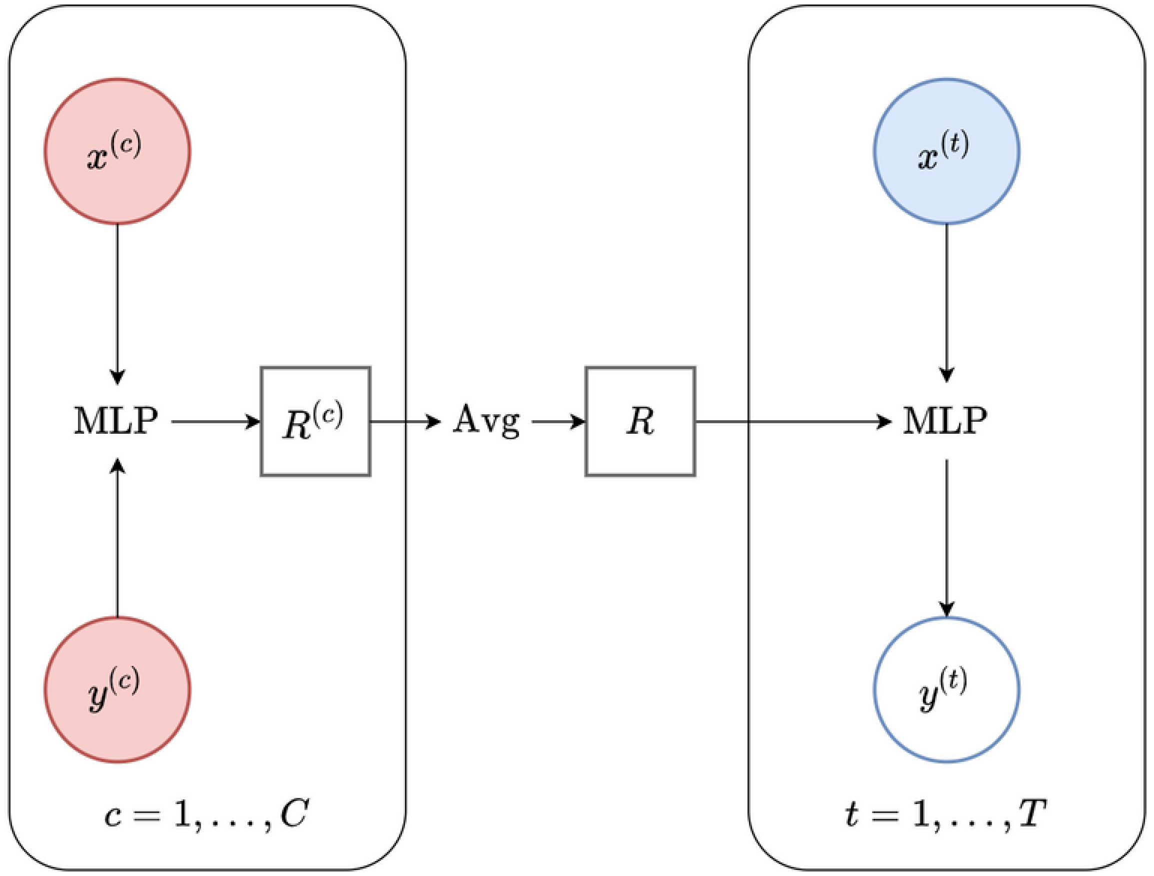
Configurations

- Image size: [3, 32, 32]
- Batch size: 16
- Representation dimension (r_dim): 512
- Latent space dimension (z_dim): 512
- Number of context points range: [3, 200]
- Number of extra target points range: [0, 200]
- Learning rate (lr): 4e-5
- Number of epochs: 100
- Dataset: CelebA
- Training Time: Approx. 1 hour
- Images : 3750

Model

Input :
X as location with shape (1,2,)
Y as intensity (RGB) with shape(1,3)

Output :
R with dimension 512



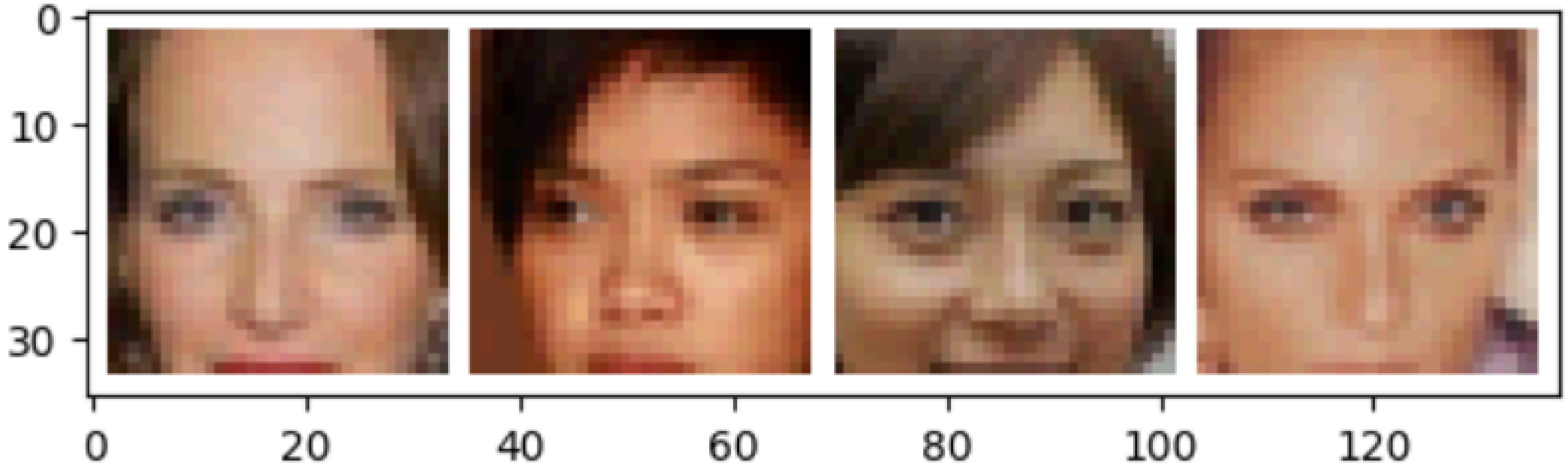
Input:
x: Input data tensor of size (batch_size, num_points, x_dim). It represents the input features, such as images.
z: Latent space tensor of size (batch_size, z_dim). It represents the sampled latent variables from the prior distribution.
Output:
y: Reconstructed output tensor of size (batch_size, num_points, y_dim). It represents the reconstructed output based on the given input x and sampled latent variables z.

The MuSigmaEncoder takes r as input and maps it to the mean (mu) and standard deviation (sigma) of the latent space.

A random sample z is drawn from the latent space using the reparameterization trick, where z = mu + sigma * epsilon, and epsilon ~ N(0, 1).

Experiments

In the experimental phase, I trained the model on the Celeb dataset, and a subset of four images was selected for testing purposes.

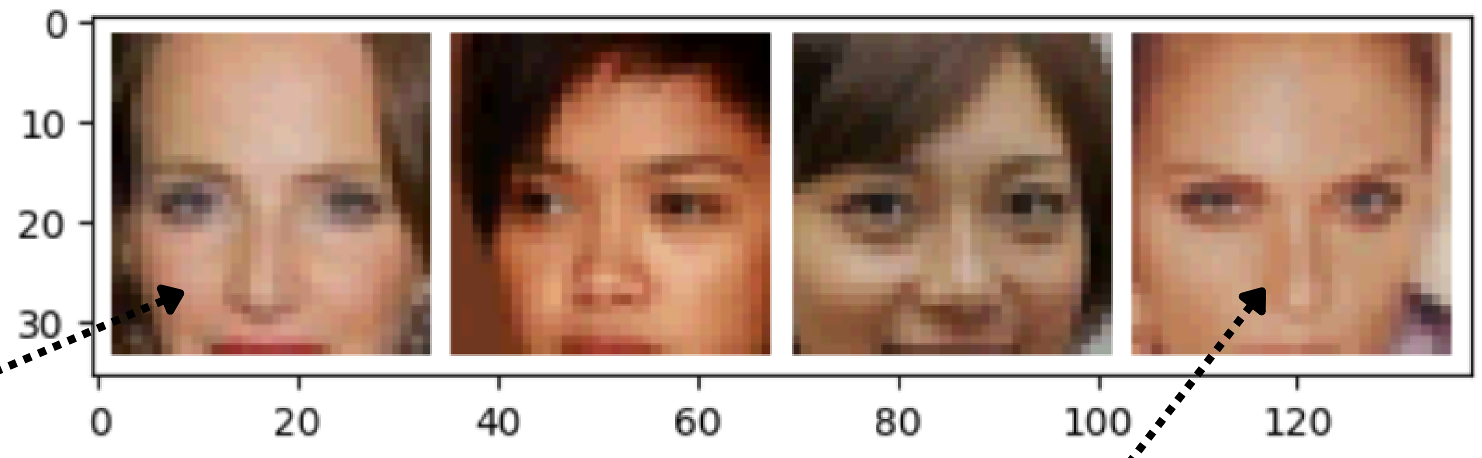


1.) It takes the img as input with trained model.
2.) Then, using the target batch, we predict the intensities there (RGB)
3.) Then we will pass to reconstruct the image using x_target & predicted intensities.
4.) Then we will take the context points and pass them to fill the full image

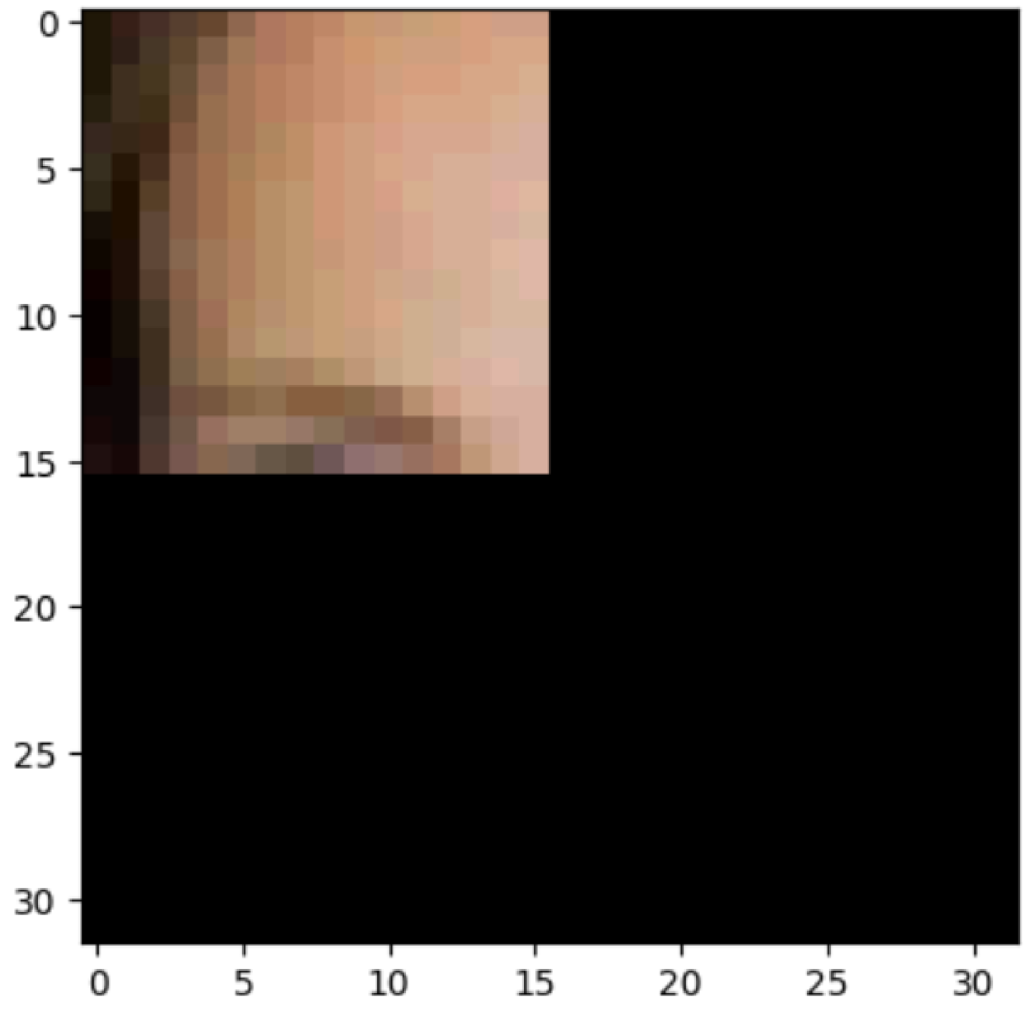
```python
def inpaint(model, img, context_mask, device):
    """
    Given an image and a set of context points, the model samples pixel
    intensities for the remaining pixels in the image.

    """
    is_training = model.neural_process.training
    model.neural_process.training = False
    target_mask = 1 - context_mask  # All pixels which are not in context
    img_batch = img.unsqueeze(0).to(device)
    context_batch = context_mask.unsqueeze(0).to(device)
    target_batch = target_mask.unsqueeze(0).to(device)
    p_y_pred = model(img_batch, context_batch, target_batch)
    x_target, _ = img_mask_to_np_input(img_batch, target_batch)
    # Using the mean parameter of normal distribution as predictions for y_target
    img_rec = xy_to_img(x_target.cpu(), p_y_pred.loc.detach().cpu(), img.size())
    img_rec = img_rec[0]  # Remove batch dimension
    # Add context points back to image
    context_mask_img = context_mask.unsqueeze(0).repeat(3, 1, 1)
    img_rec[context_mask_img] = img[context_mask_img]
    # Reset model to mode it was in before inpainting
    model.neural_process.training = is_training
    return img_rec
```

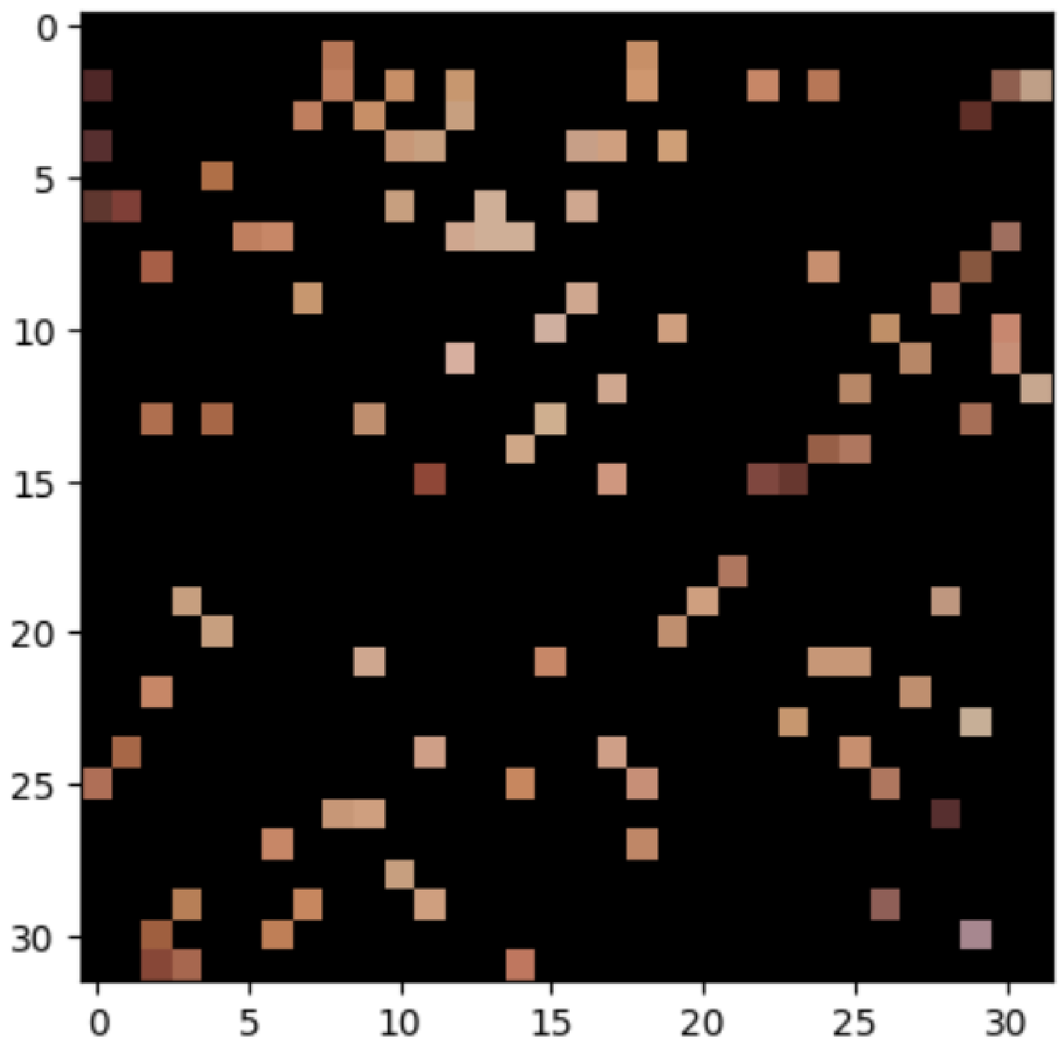
Inpainting



1.



2.



In this experiment, I compared the performance of three activation functions to determine which one is working better than the others.

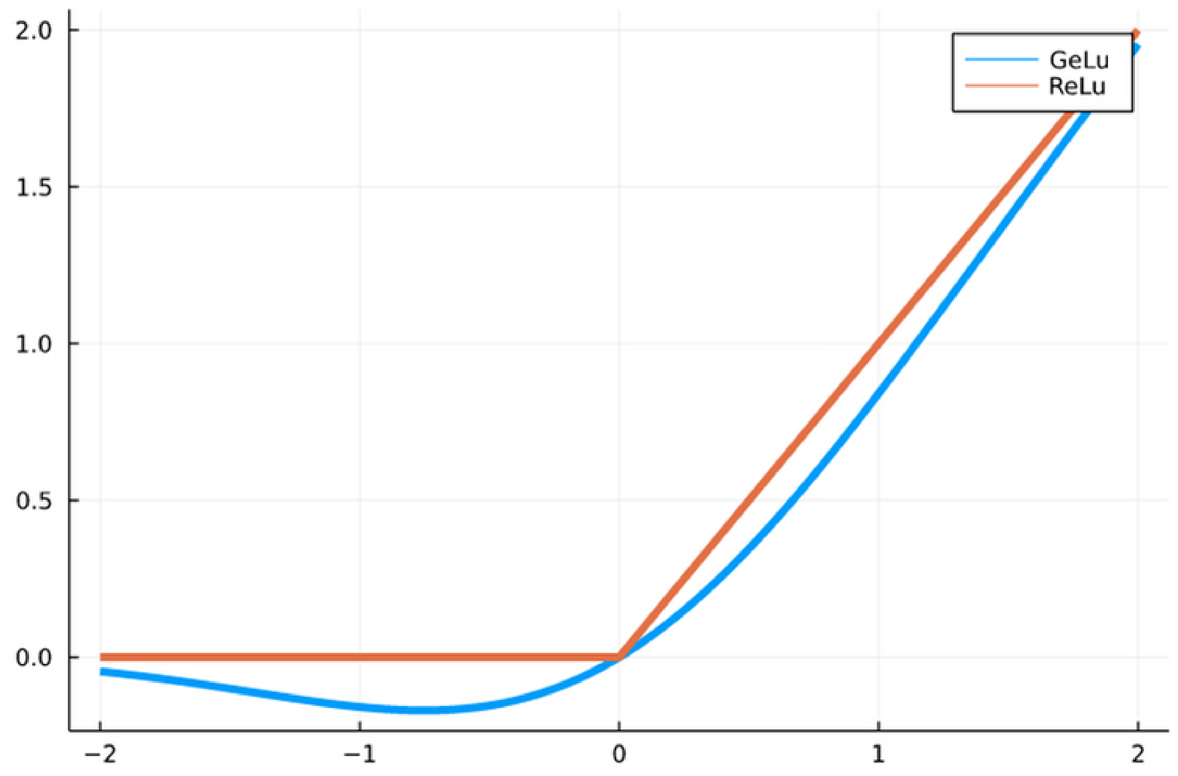
Activation functions

RELU

GELU

SINE

$$\mathrm{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \mathrm{erf}(x/\sqrt{2})\right]$$



Approximately,

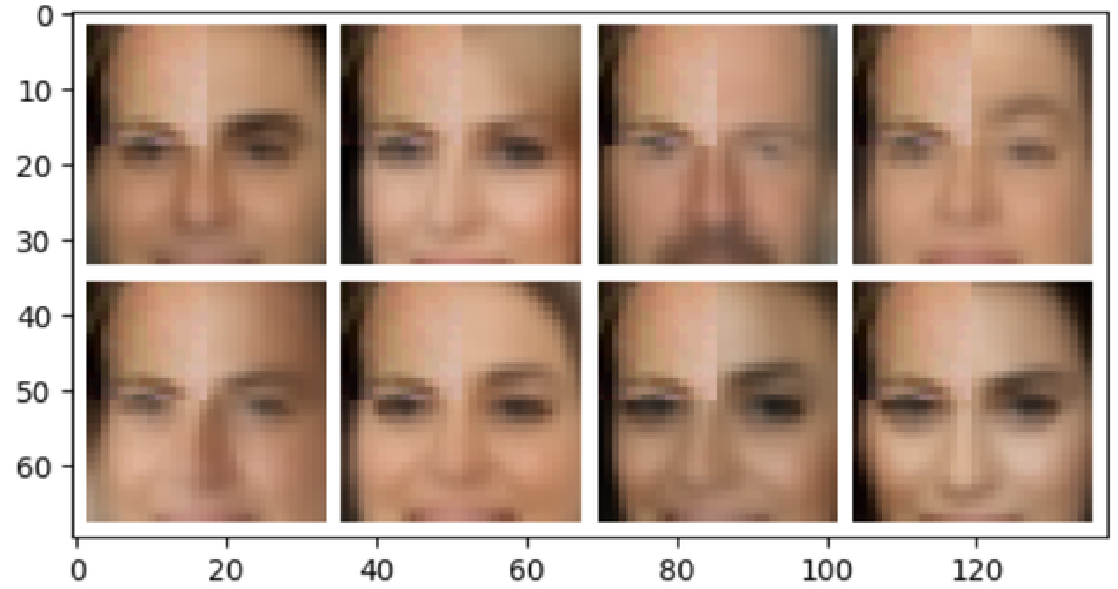
- $GELU(x) = 0.5x(tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$
- $GELU(x) = x\sigma(1.702x)$

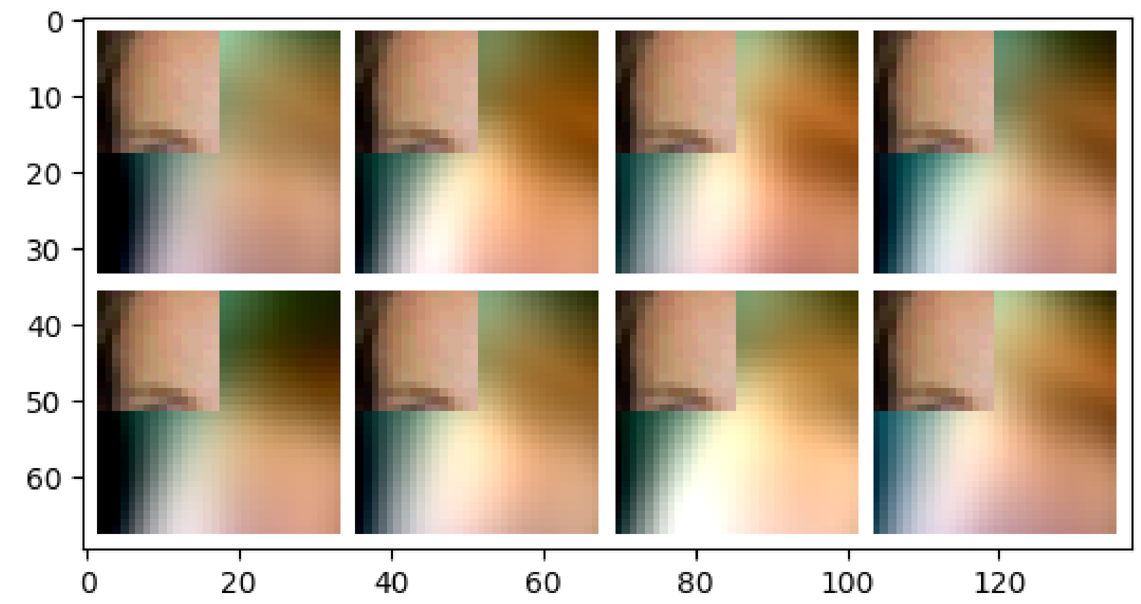
Results

ReLU performs better than other activation functions, but weight initialization methods can impact performance. Alternative techniques may outperform ReLU.

For the first exp,

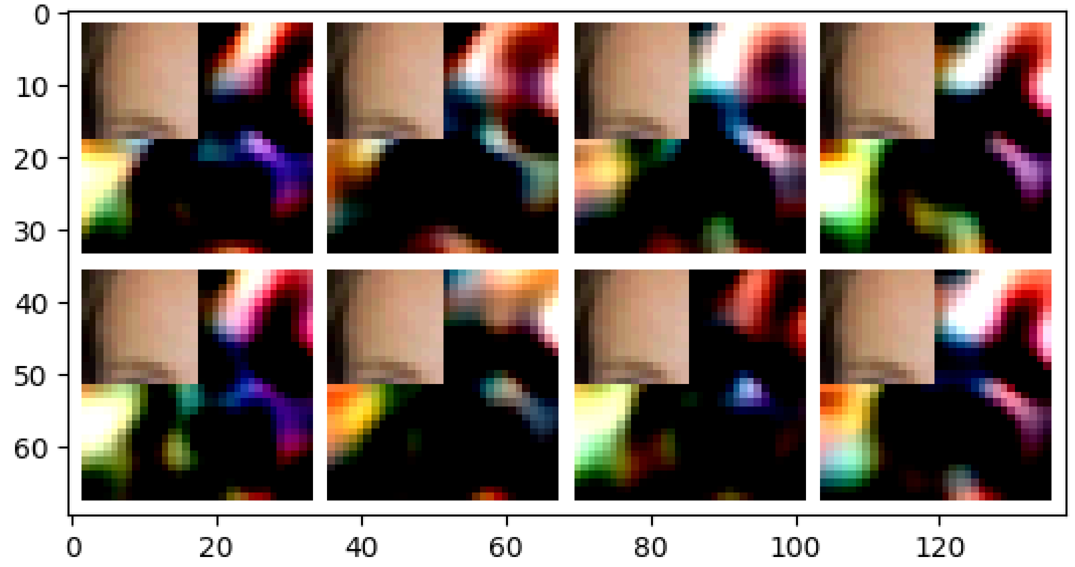
RELU





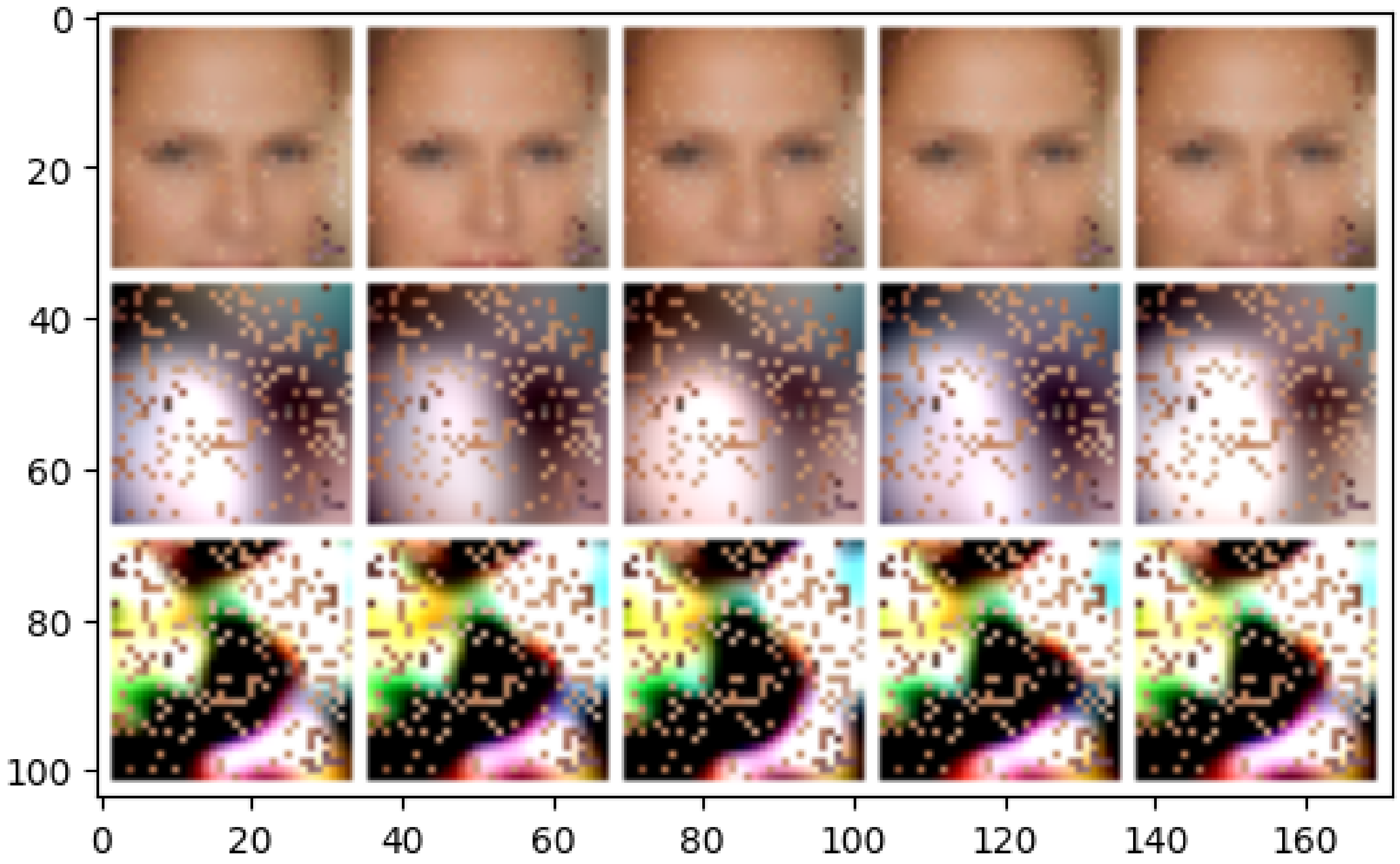
GELU

SINE



For the 2nd exp,

The Neural Process model successfully generated realistic images despite significant pixel masking, showcasing its ability to leverage context points and their spatial arrangement for accurate image reconstruction.



I have utilized **three error metrics** to evaluate the performance of three different activation functions.

RMSE

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

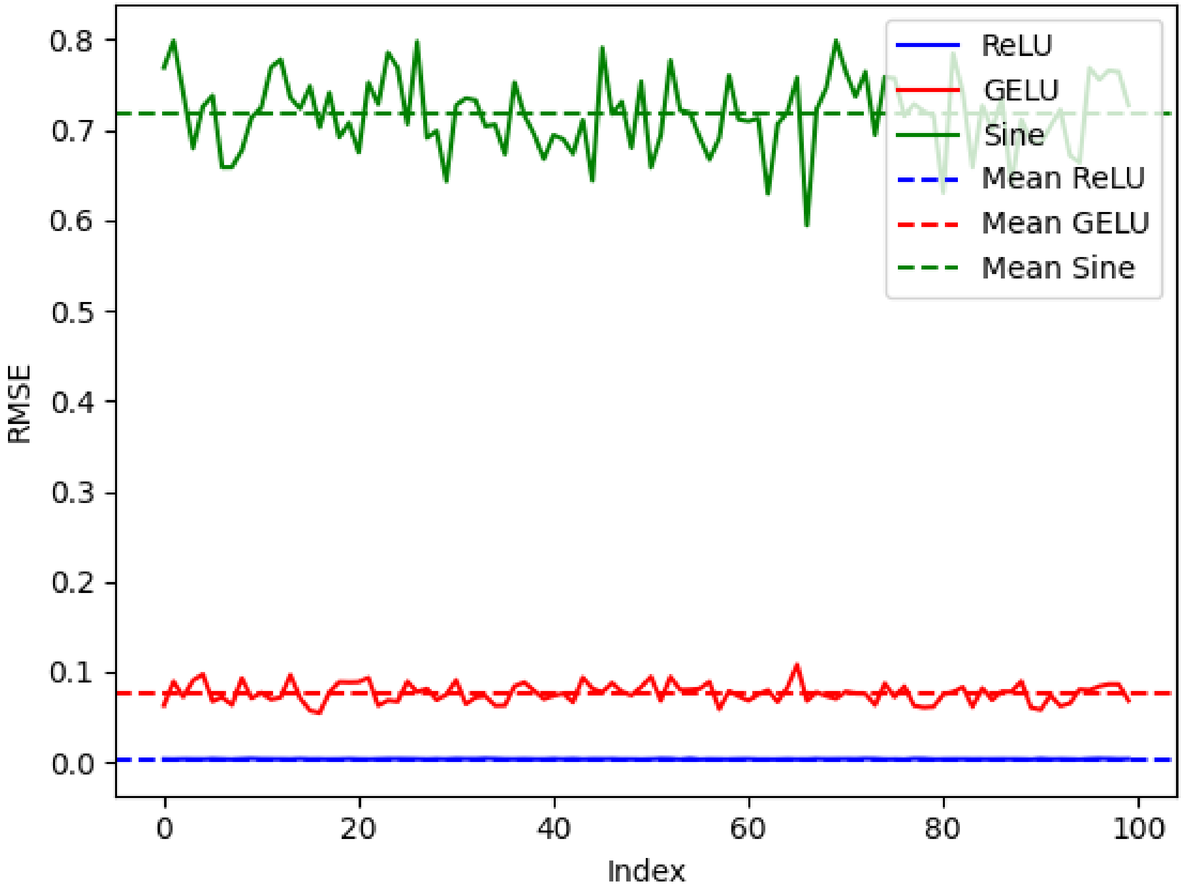
PSNR

$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE). \end{aligned}$$

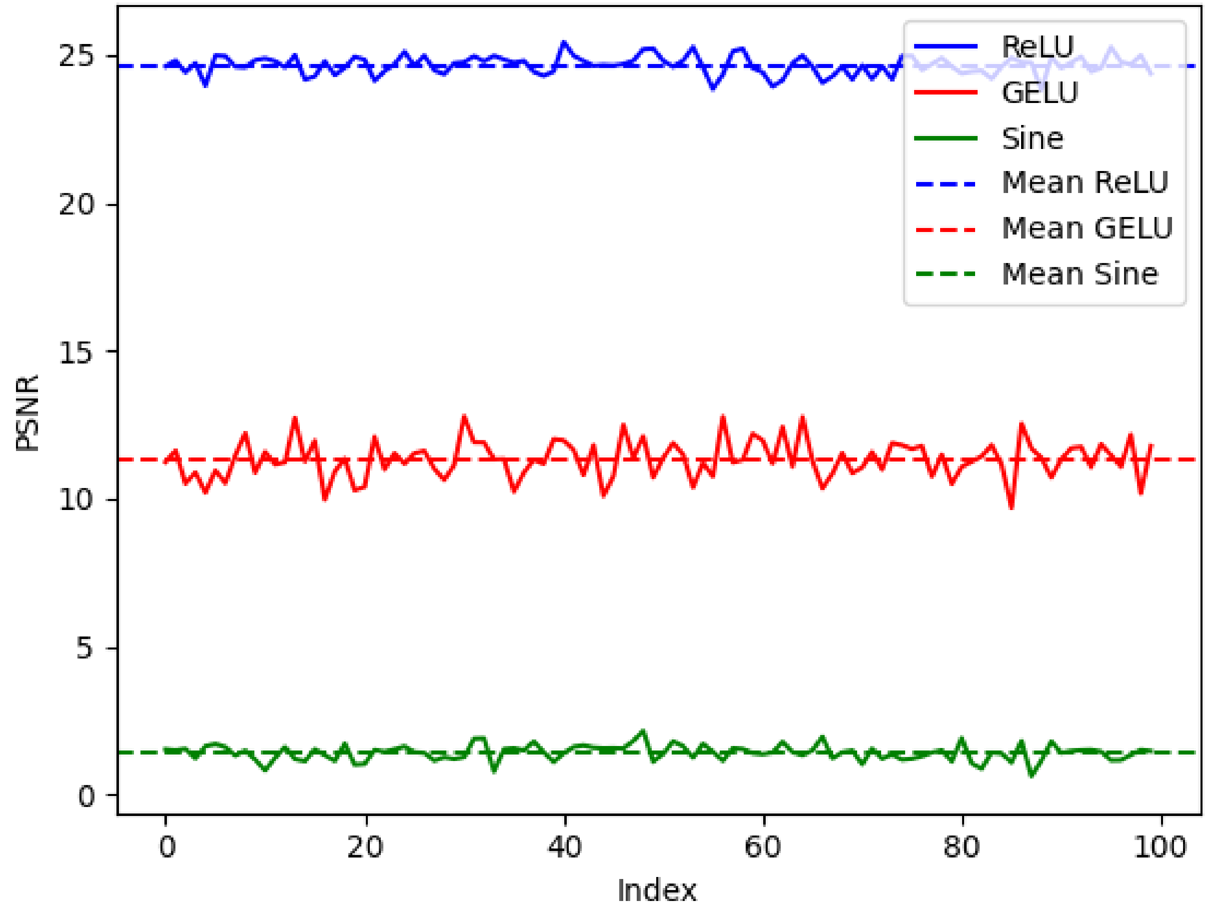
SSIM

The Structural Similarity Index (SSIM) ranges between **-1 and 1,** where a value of 1 indicates a perfect match between two images, 0 indicates no similarity, and values below 0 indicate significant dissimilarity.

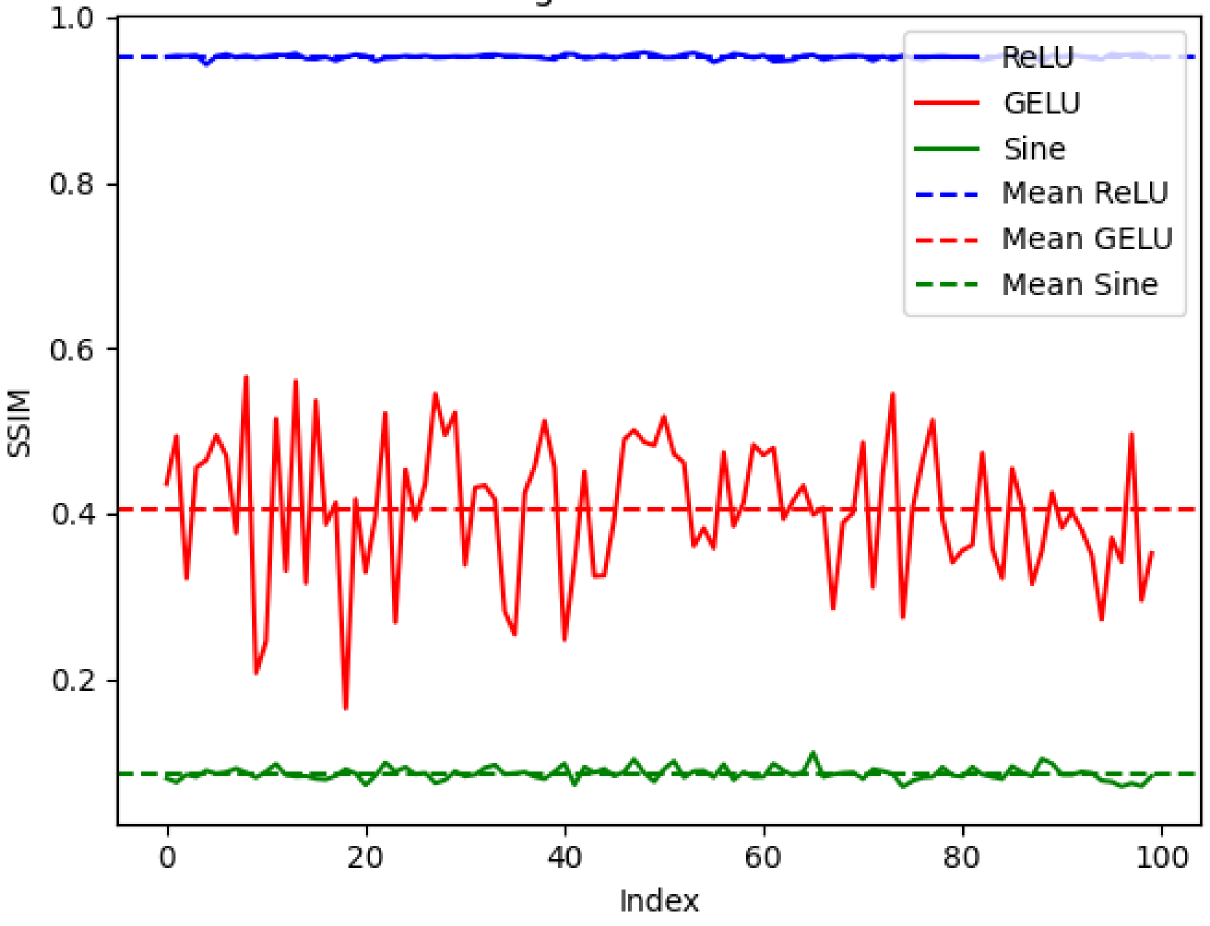
Comparison of RMSE values for ReLU,GELU & Sine
Lower is desirable



Comparison of PSNR values for ReLU,GELU & Sine
Higher is desirable



Comparison of SSIM values for ReLU,GELU & Sine
Higher is desirable



Comparision Between 3 activation functions

	MEAN RMSE	MEAN PSNR	MEAN SSIM
RELU	0.003	24.65	0.9521
GELU	0.076	11.33	0.4059
SINE	0.717	1.42	0.0853

Now using **SIREN** weight-initialization scheme

```python
class SirenLinear(nn.Module):
    # See paper sec. 3.2, final paragraph, and supplement Sec. 1.5 for discussion of omega_0.

    # If is_first=True, omega_0 is a frequency factor which simply multiplies the activations before the
    # nonlinearity. Different signals may require different omega_0 in the first layer - this is a
    # hyperparameter.

    # If is_first=False, then the weights will be divided by omega_0 so as to keep the magnitude of
    # activations constant, but boost gradients to the weight matrix (see supplement Sec. 1.5)

    def __init__(self, in_features, out_features, bias=True,
                 is_first=False, omega_0=30,is_linear = False):
        super().__init__()
        self.omega_0 = omega_0
        self.is_first = is_first
        self.is_linear = is_linear
        self.in_features = in_features
        self.linear = nn.Linear(in_features, out_features, bias=bias)

        self.init_weights()

    def init_weights(self):
        with torch.no_grad():
            if self.is_first:
                self.linear.weight.uniform_(-1 / self.in_features,
                                             1 / self.in_features)
            else:
                self.linear.weight.uniform_(-np.sqrt(6 / self.in_features) / self.omega_0,
                                             np.sqrt(6 / self.in_features) / self.omega_0)

    def forward(self, input):
        if self.is_linear:
          return (self.omega_0 * self.linear(input))
        return torch.sin(self.omega_0 * self.linear(input))
```

Configuration

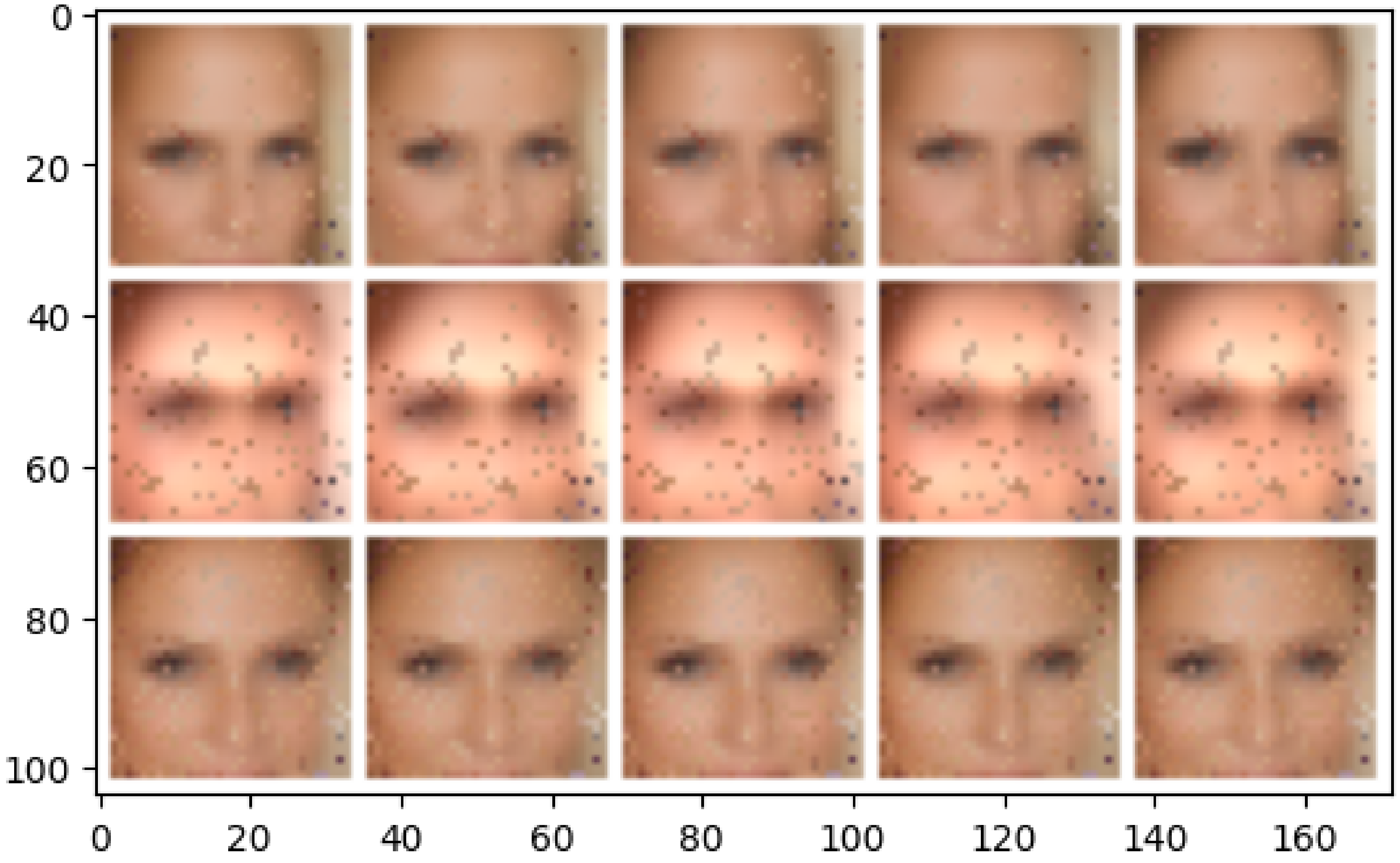
- For the first layer:
  W ~ U(-1/fan_in, 1/fan_in).

- For subsequent layers:
  W ~ U(−√6/fan_in/Ω, √6/fan_in/Ω).

Weights scaled for consistent activations.

Results

Relu in Encoder
Gelu in Decoder(smoothness)

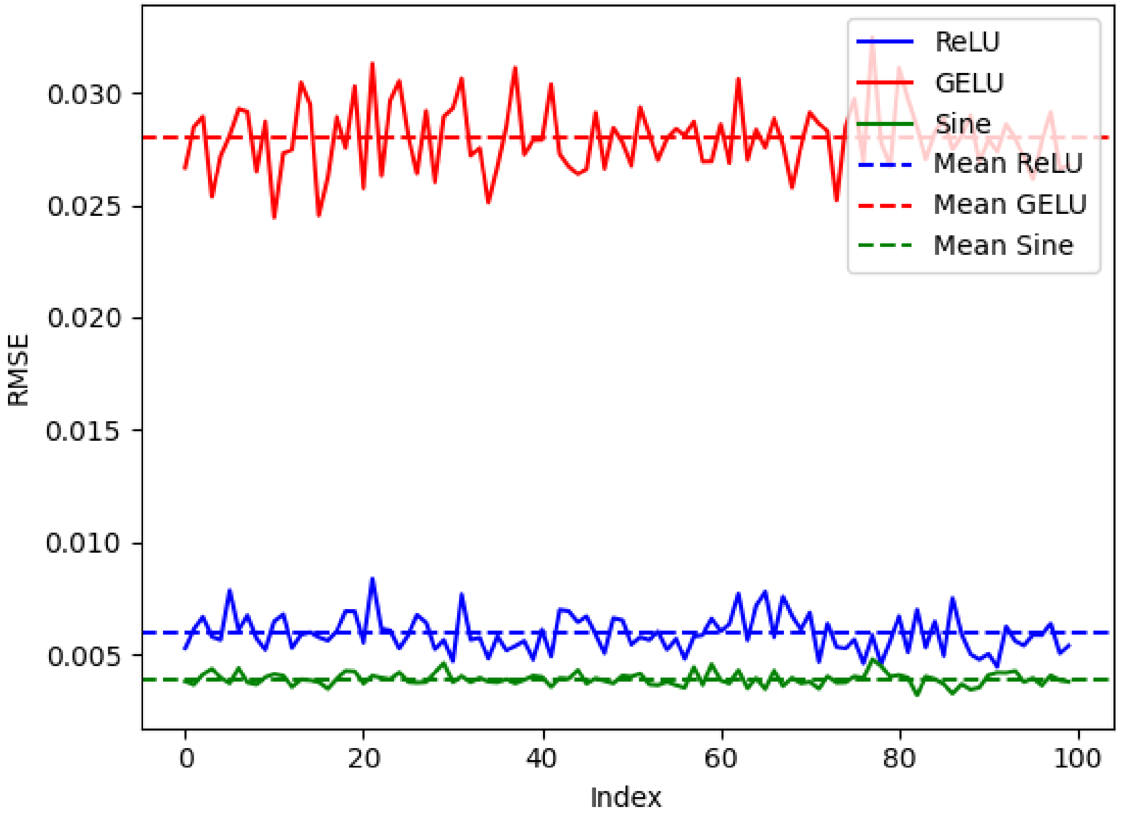


SINE
(With SIREN Initial. Scheme)

I have utilized **three error metrics** to evaluate the performance of three different activation functions.

RMSE

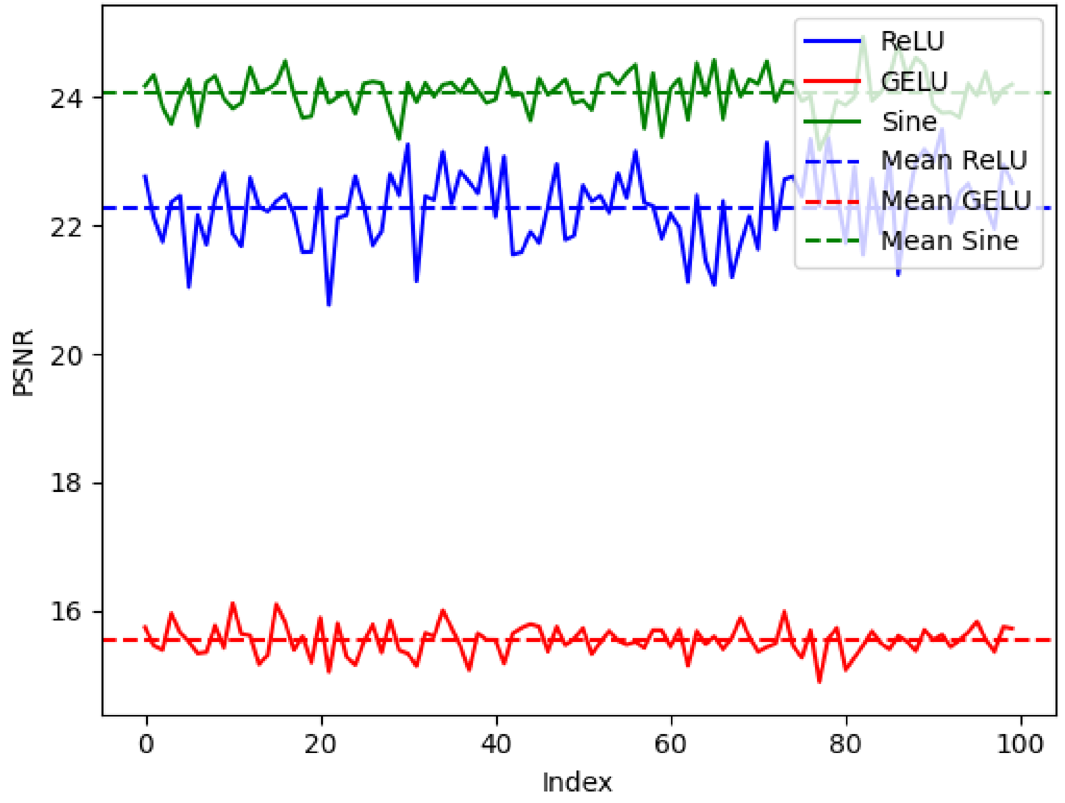
$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$



Comparison of RMSE values for ReLU,GELU & Sine
Lower is desirable

PSNR

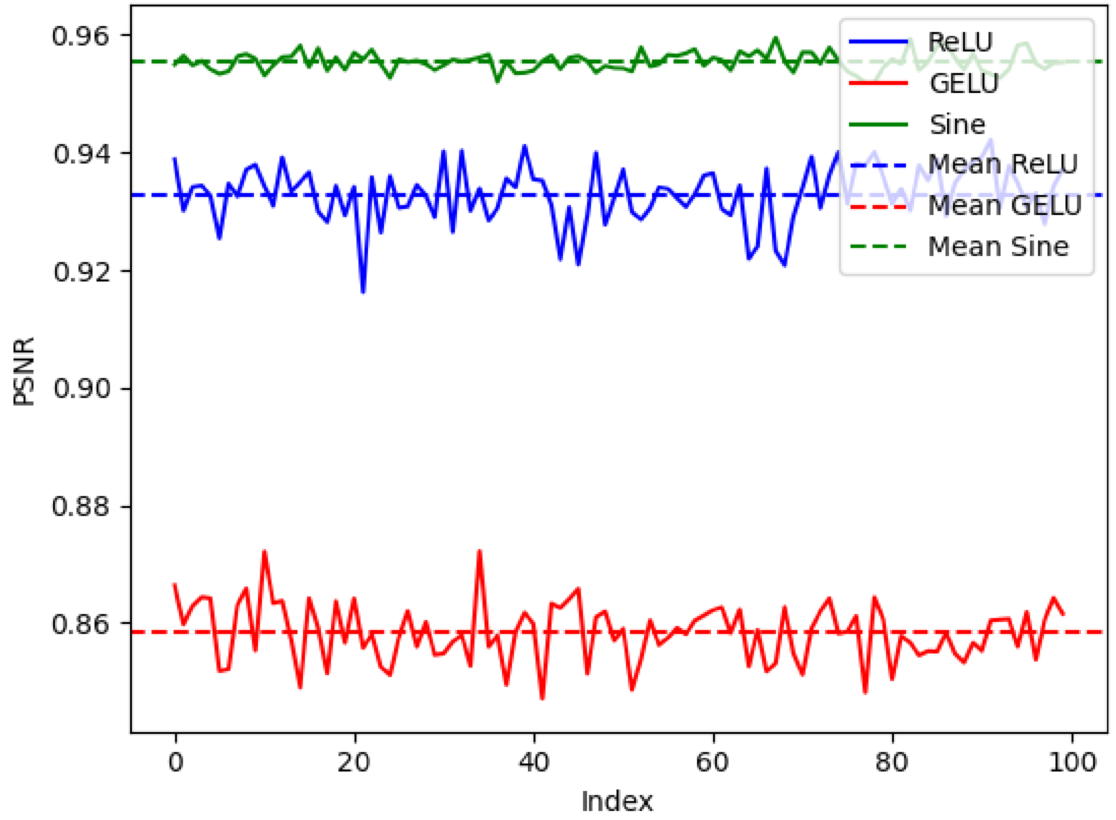
$$\begin{aligned} PSNR &= 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \\ &= 20 \cdot \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \\ &= 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE). \end{aligned}$$



Comparison of PSNR values for ReLU,GELU & Sine
Higher is desirable

SSIM

The Structural Similarity Index (SSIM) ranges between **-1 and 1,** where a value of 1 indicates a perfect match between two images, 0 indicates no similarity, and values below 0 indicate significant dissimilarity.



Comparison of SSIM values for ReLU and GELU
Higher is better

Comparision Between 4 activation functions

	MEAN RMSE	MEAN PSNR	MEAN SSIM
RELU	0.0059	22.28	0.932
GELU+RELU	0.0275	15.54	0.858
SIREN	0.0039	24.08	0.955

SIREN & RELU similar