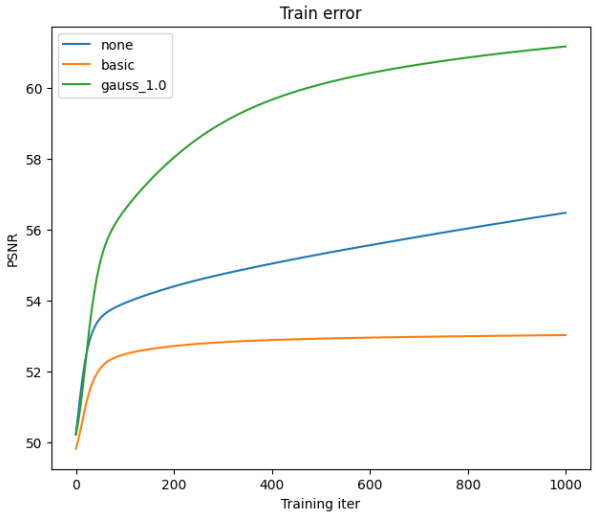
Assignment 2

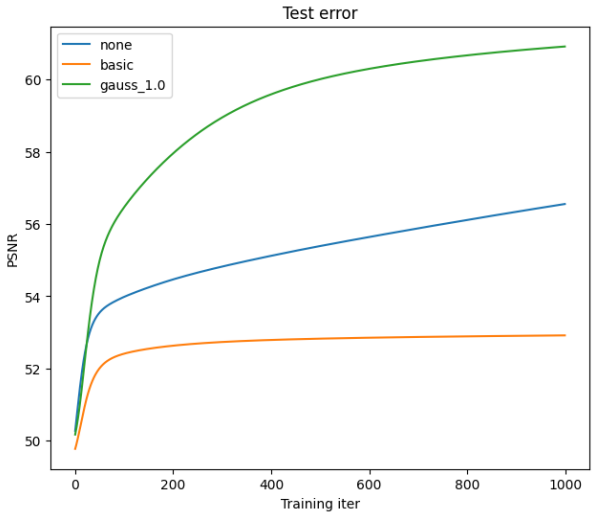
Name(s): Qi Long, Yihong Yang
NetID(s): qilong2, yihongy3

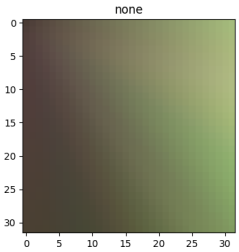
In each the following parts, you should insert the following:

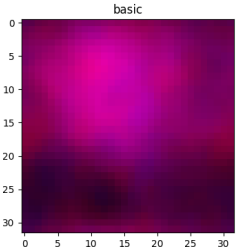
- Train/test loss plots
- Qualitative outputs for GT, No encoding, Basic Positional Encoding, and Fourier Feature Encoding

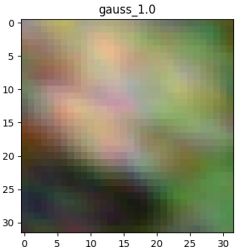
Part 1: Low resolution example – SGD

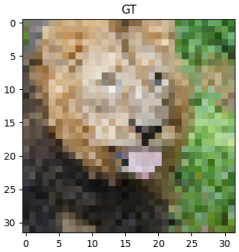




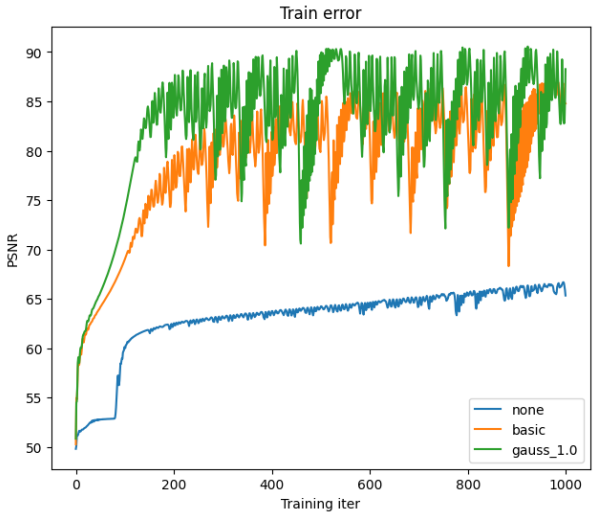


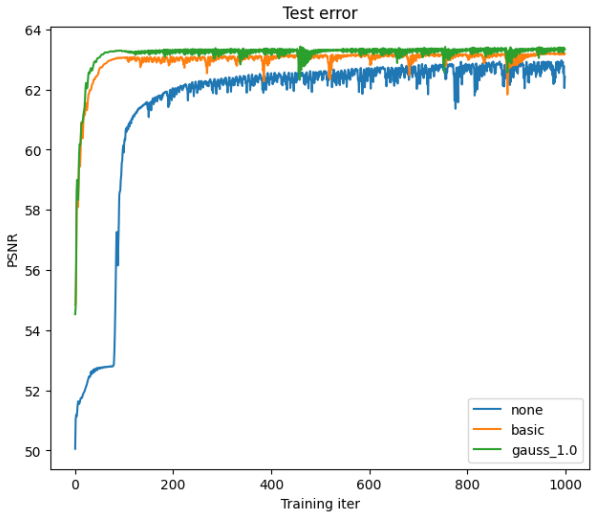


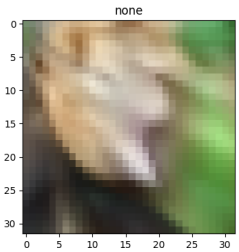


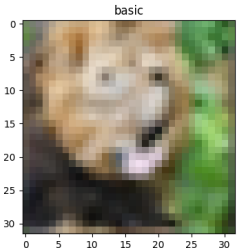


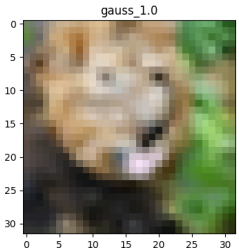
Part 2: Low resolution example – Adam

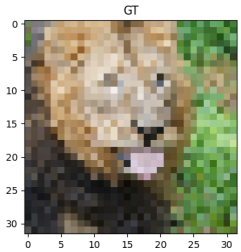






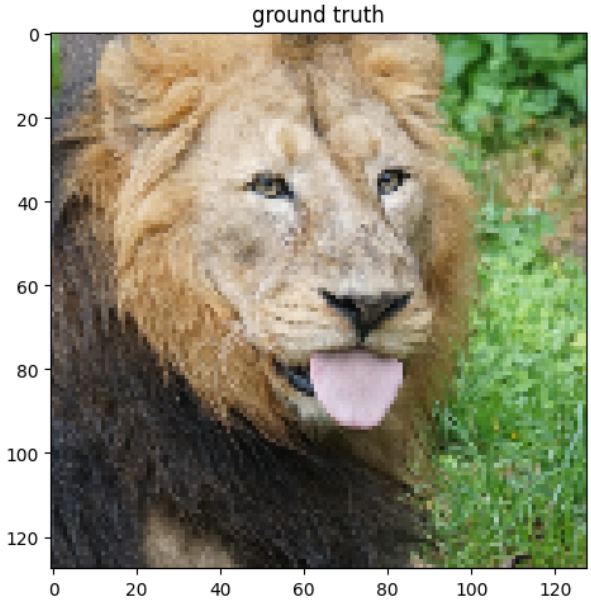




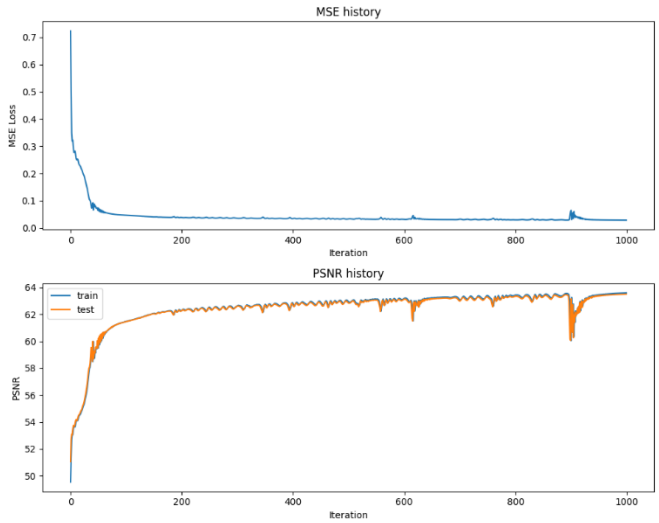


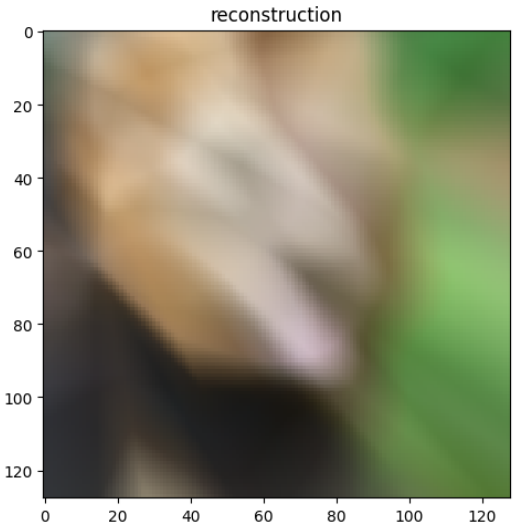
Part 3: High resolution example

1) Ground Truth:

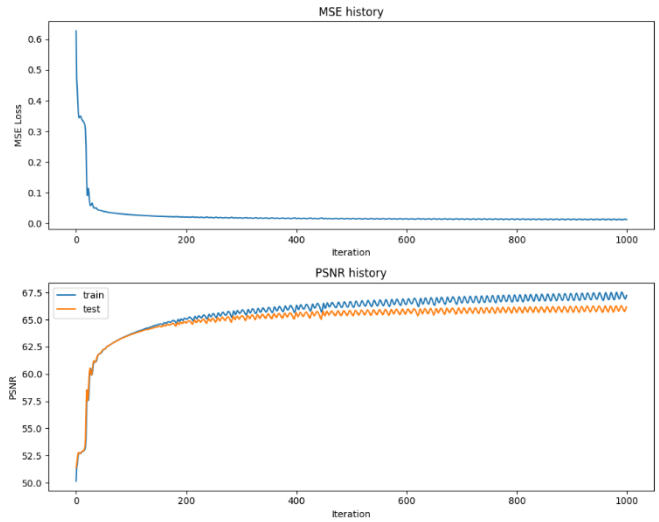


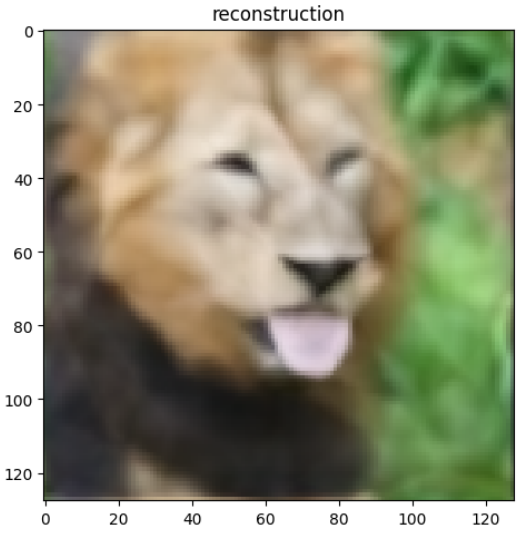
2) No mapping:



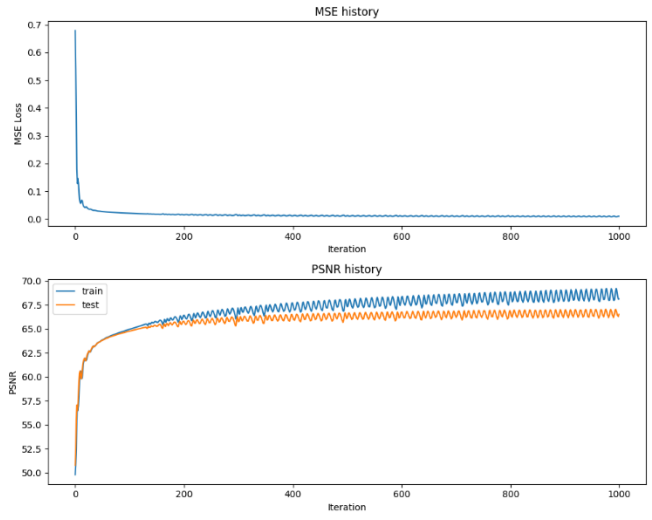


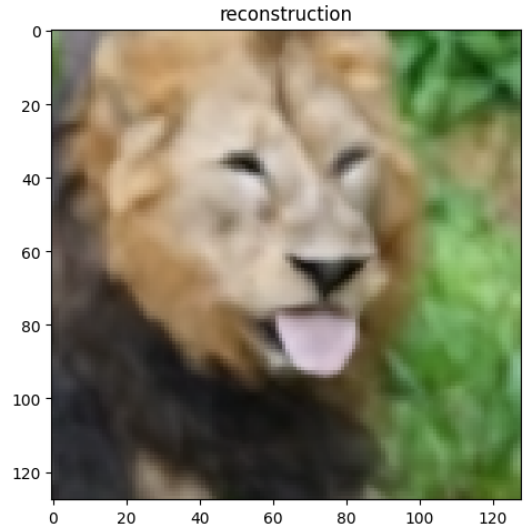
3) Basic mapping:





4) Fourier feature mapping:

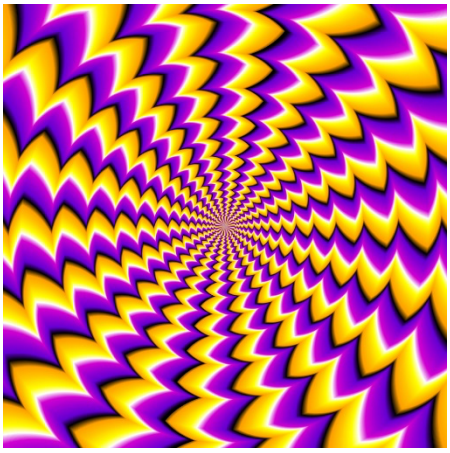


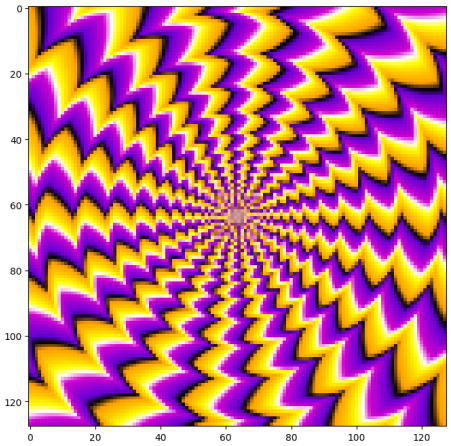


Part 4: High resolution (image of your choice)

(For this part, you can select an image of your choosing and show the performance of your model with the best hyperparameter settings and mapping functions from Part 3. You do not need to show results for all of the mapping functions.)

Selected Image:



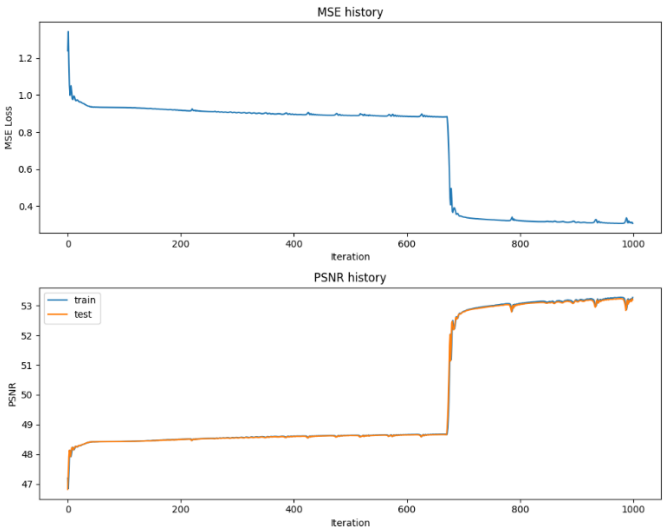


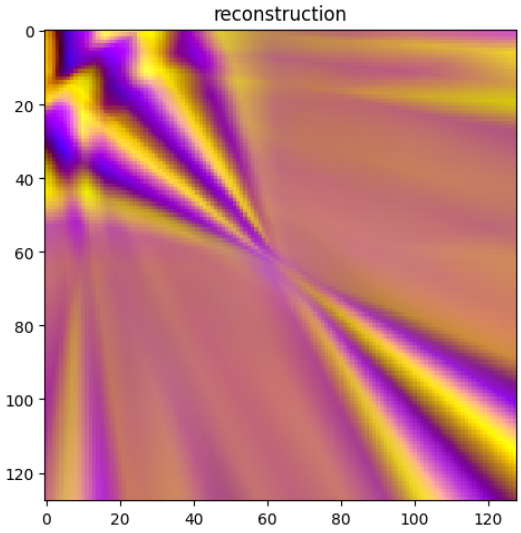
Reasons for the image:

1) This is an image of illusion, in a shape of spiral. An extra easy-to-check criterion for performance then is whether it maintains the illusion property after reconstruction.
2) It has frequent color changes in local regions. In other words, it changes between yellow and purple from place to place frequently that few pixels have close features with their neighbors. This can give better insight about how feature mapping works.
3) It has complex boundaries between two colors, which is hard to fit without feature mapping. This can give better insight about how feature mapping works.

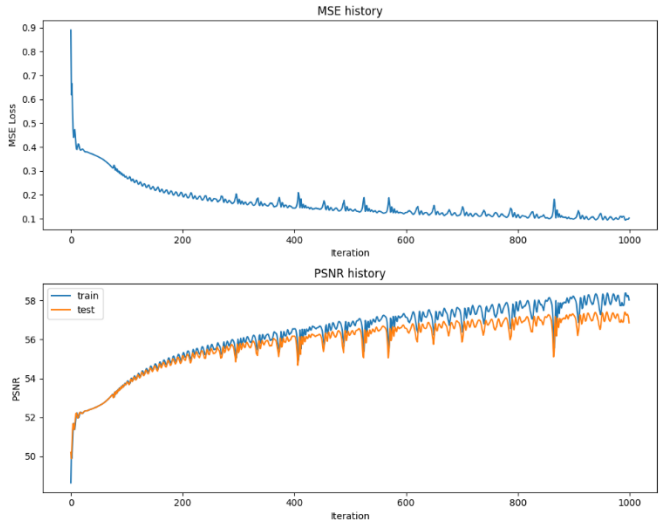
Results:

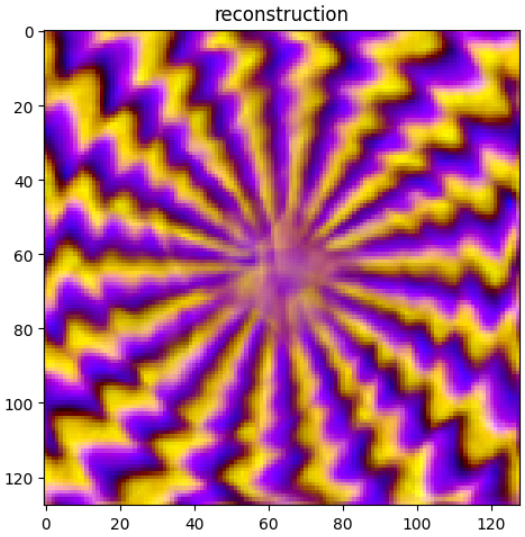
1) No mapping:



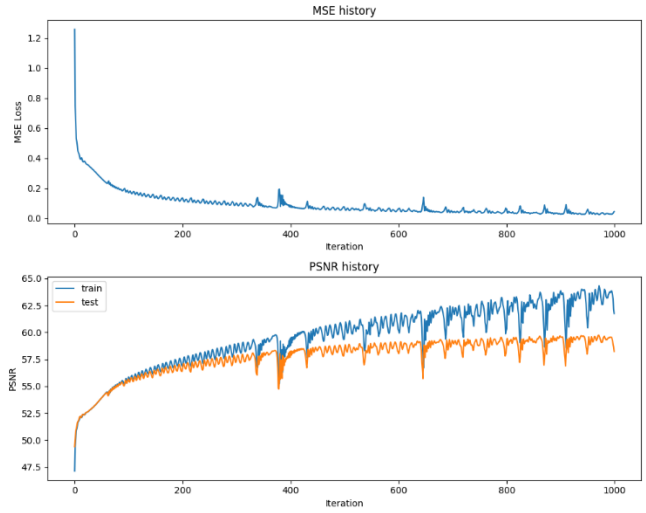


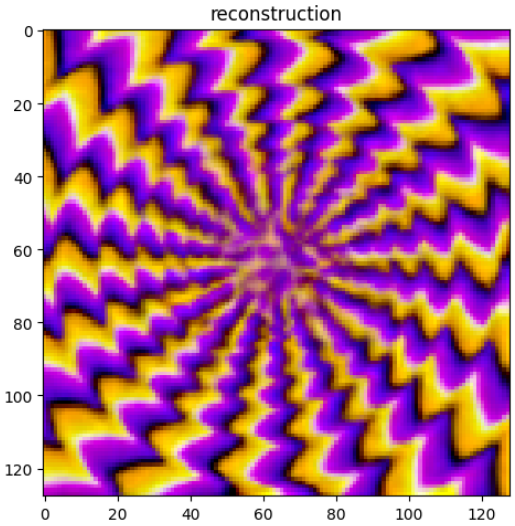
2) Basic mapping:





3) Fourier feature mapping:





Part 5: Discussion

Briefly describe the hyperparameter settings you tried and any interesting implementation choices you made.

How did the performance of SGD and Adam compare?

Adam has a much better performance than SGD.
I think three main reasons contribute to this difference:

1) The two momentums make it continue to search for smaller loss even when trapped at local optimum.

2) The momentums record past gradients through out the process, partly keep the update of parameters in the global optimization direction.
3) There is difference on each update of momentums with respect to epoch, making the step size getting smaller as process goes on, so that it will get closer and closer to optimum instead of bouncing back and forth near the optimum.

How did the different choices for coordinate mappings functions compare?

Fourier Feature mapping is better than basic mapping, which is better than no mapping.
I think it is mainly because mapping results in more feature representations of each location so that the learning process can make use of them and optimize parameters in a more elaborate manner. Fourier Feature mapping has the largest input_size for deep NN, therefore deep NN can have more parameters to learn in a elaborate manner, reaching the best performance.

Do you make any interesting observations from the train and test plots?

During the learning process, train error can reach a very low point (PSNR = 90) and become stable in a short time, but test error cannot reach such an optimal point, stopping at a sub-optimal level (PSNR = 63).
This is because the training process has no access to test sample so that the parameters may not be the best combinations for test performance. However, it have more training samples to directly fit the parameters so that it can have very high performance on them.

What insights did you gain from your own image example (Part 4)?

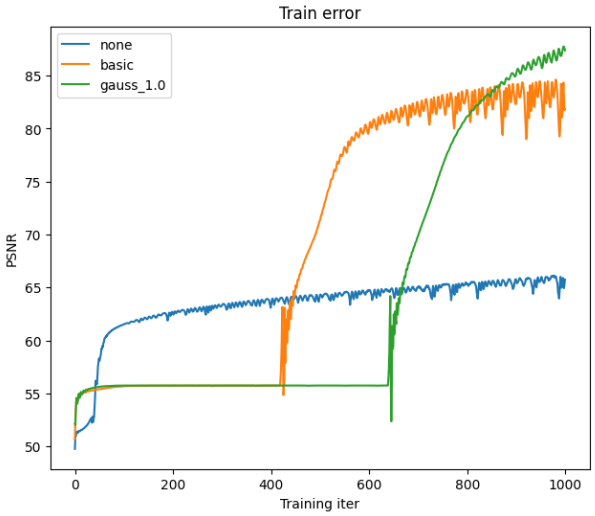
According to the experiment result shown above, three main insights are gained:
1) Without mapping, the frequent local color changes can not be learned, resulting in only two relatively optimal parts. The boundaries of different colors are mostly straight lines, which also shows its limitations.
2) The advantage of feature mapping, especially Fourier feature mapping is shown greatly. Compared with no mapping, feature mapping enables NN to learn elaborate edges and color difference, so that different locations have more freedom to be decided the RGB value independently.
3) The comparison between basic mapping and Fourier feature mapping shows that more elaborate mapping leads to closer and detailed reconstruction. In the pictures, Fourier feature mapping can learn the small black edges between yellow and purple space but basic mapping cannot, choosing the red color mistakingly.

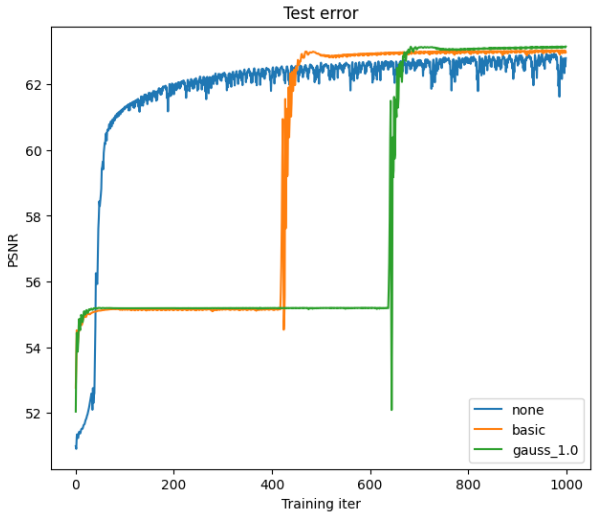
Part 6: Extra Credit

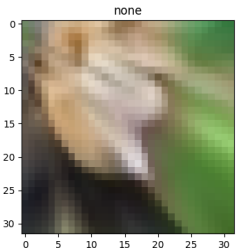
Explain what you experimented with in detail and provide output images.

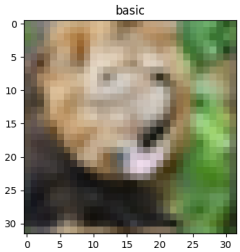
1. Deeper Network

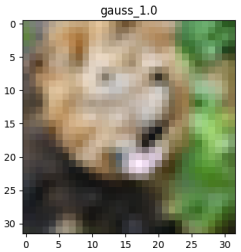
1) Num_layers = 5:

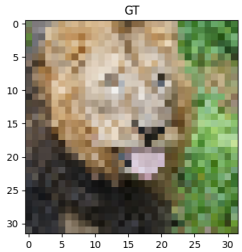




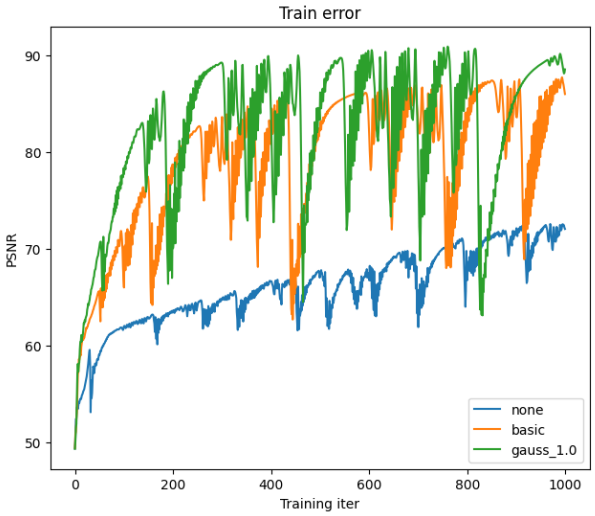


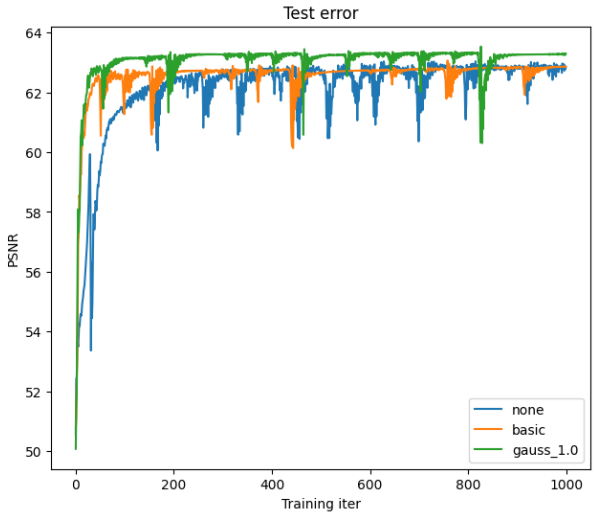


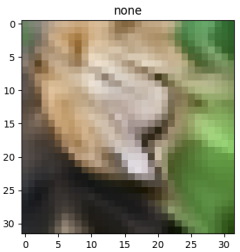


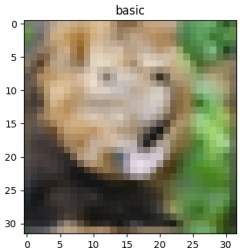


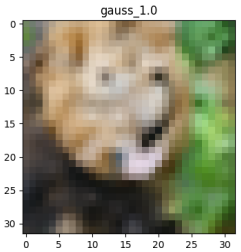
2) Num_layers = 10:

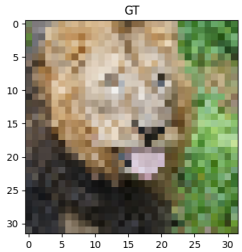




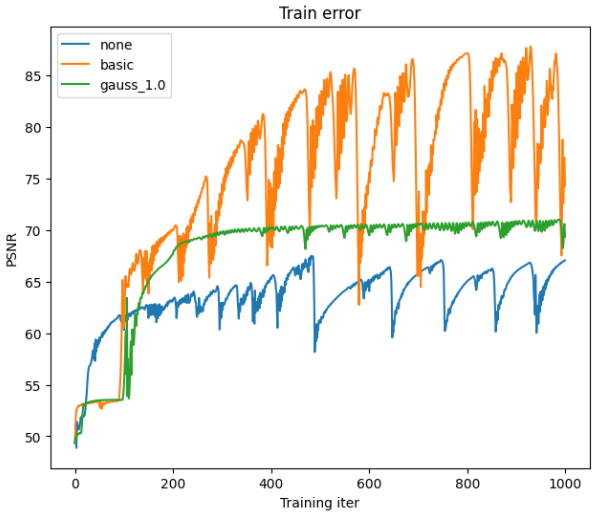


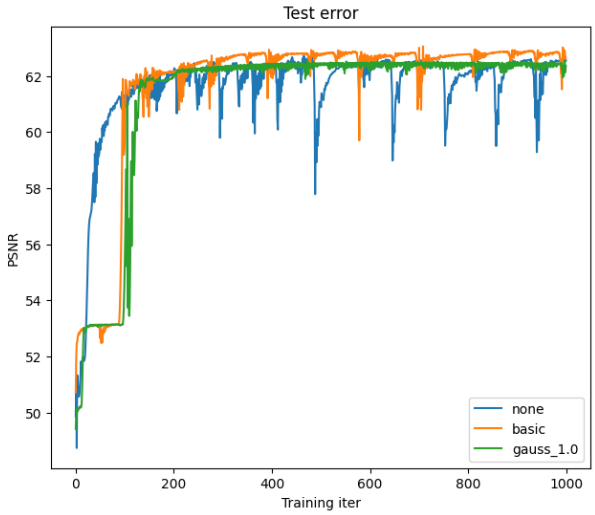


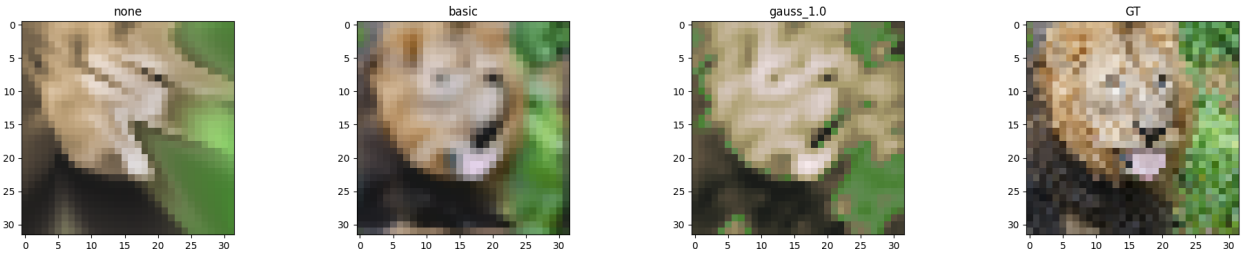




3) Num_layers = 15:





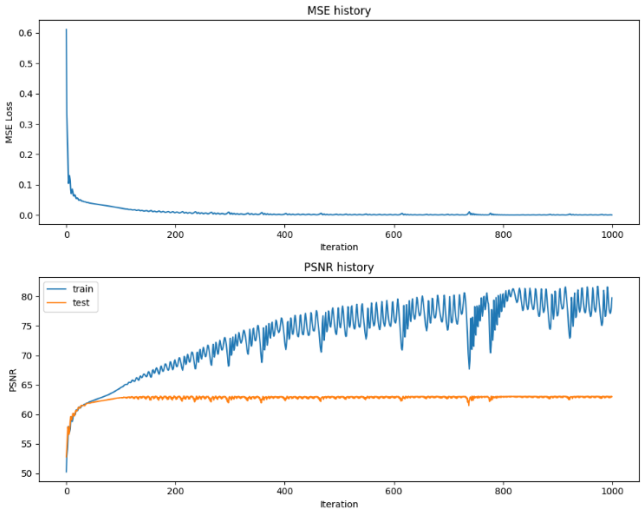


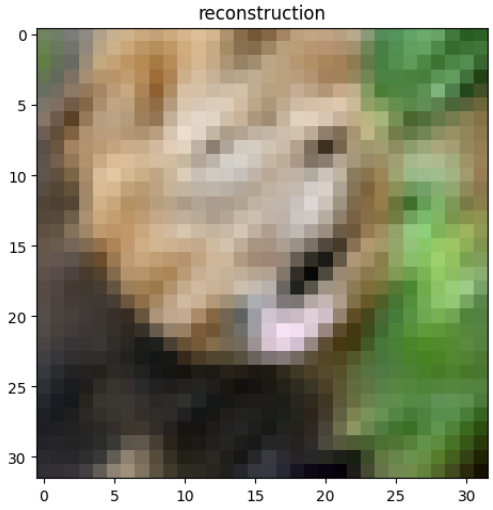
4) Discussion:

- Parameters: deeper network means more parameters to learn, theoretically adding more layers will improve the possible capability of the model. As indicated in the experiment, num_layers = 10 is better than num_layers = 5, showing better data fitting.
- Complexity: deeper networks take more time to train, since more parameters need to update. The time cost for num_layers = 15 is much longer than 10 or 5.
- Divergence: deeper network is hard to converge, since a large number of parameters update each epoch. As shown in experiements, num_layers = 15 has worse performance since it is too hard to fit and did not converge, resulting in gauss_1.0's training error became stable at sub-optimal stage.
- Optimal: overall, num_layers = 10 has the highest performance.

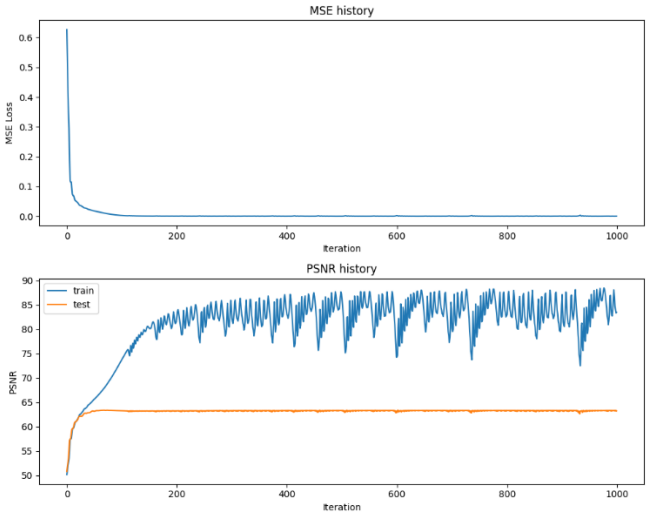
2. Gaussian Fourier Feature mapping hyperparameter – mapping size.

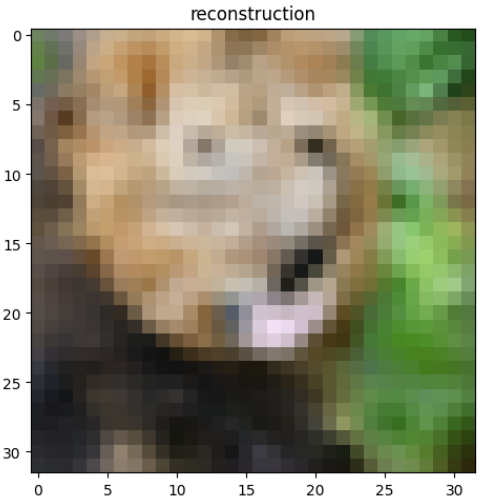
1) Mapping_size = 4



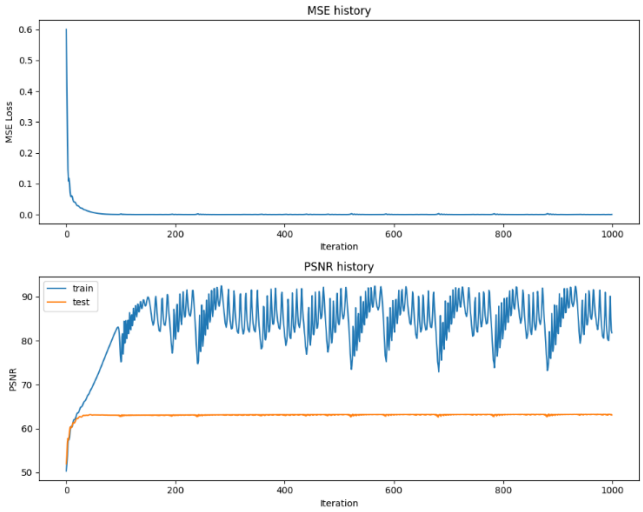


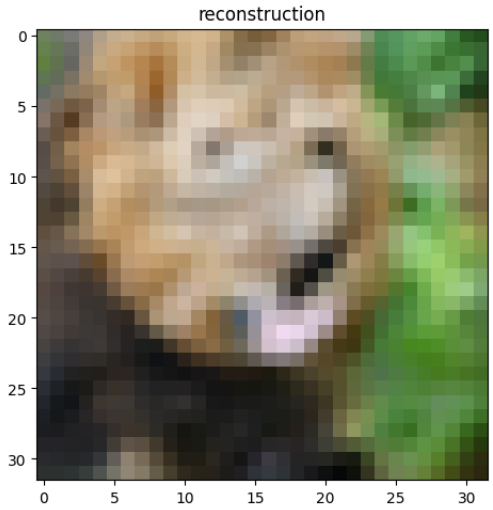
2) Mapping_size = 64



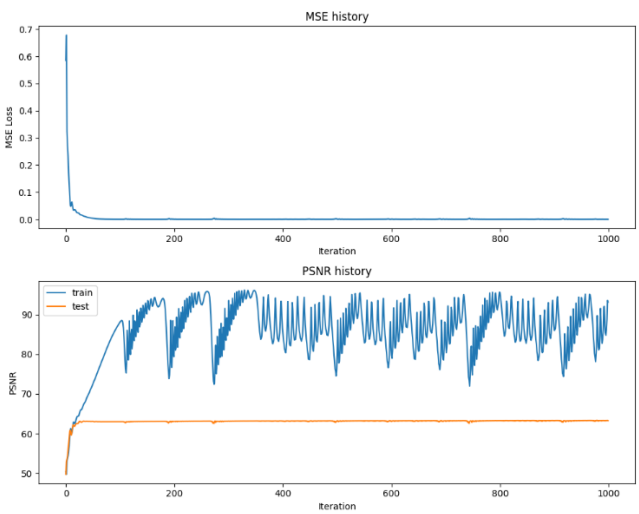


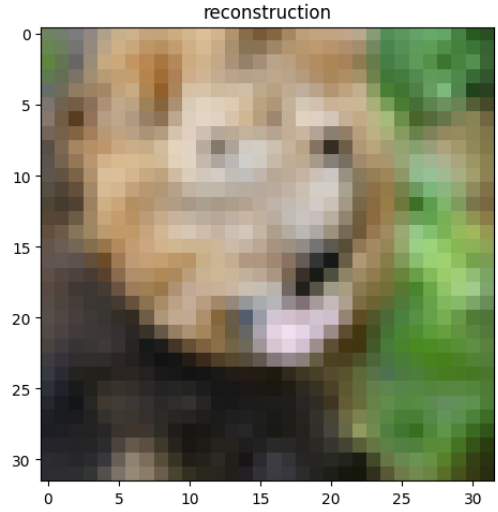
3) Mapping_size = 256





4) Mapping_size = 1024



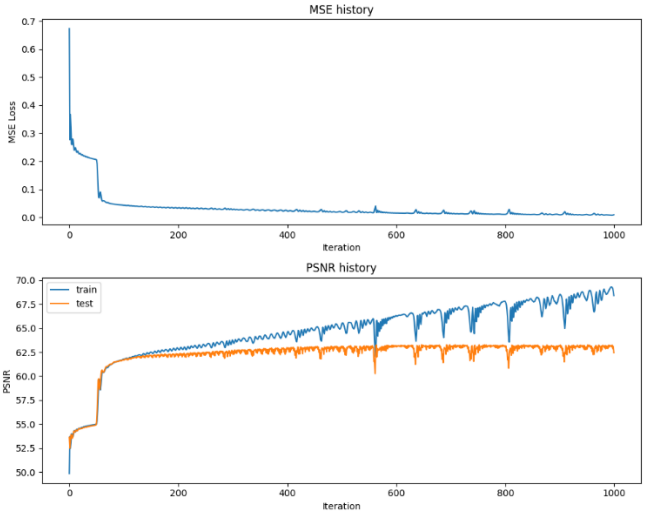


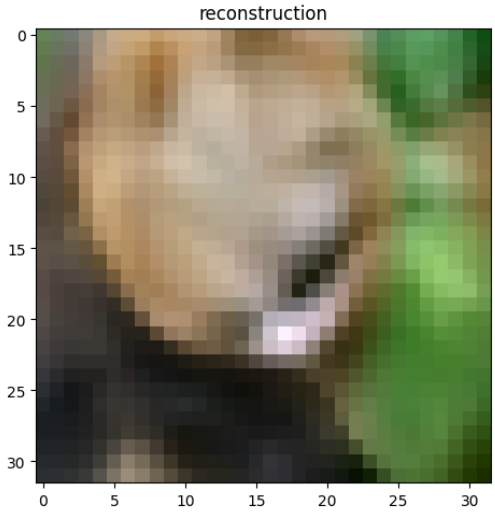
5) Discussion:

- Convergence: according to the experiment, smaller mapping_size makes change in training loss during training process smoother (mapping_size = 4 & 64). This can be explained that smaller mapping_size results in smaller number of inputs into NN, so the NN can learn and fit data faster.
- Potential: larger mapping_size leads to stronger model potential. Since the input has higher dimension, relationship of different locations of the image cab be further discovered during training process. This is verified by the evidence that mapping_size = 256 & 1024 are better than mapping_size = 4.
- Capacity limitation: I also observe that there is little difference between mapping_size = 256 & 1024's output. This can be explained that since the network is fixed with 5 layers and width of 256 each layer, it may fail to make full use of all 1024 input dimensions, resulting in bouncing back and forth during later updates (mapping_size = 256 plot) but no further improvement.
- Optimal: overall, mapping_size = 64 or 256 have optimal performance.

3. Gaussian Fourier Feature mapping hyperparameter – sigma.

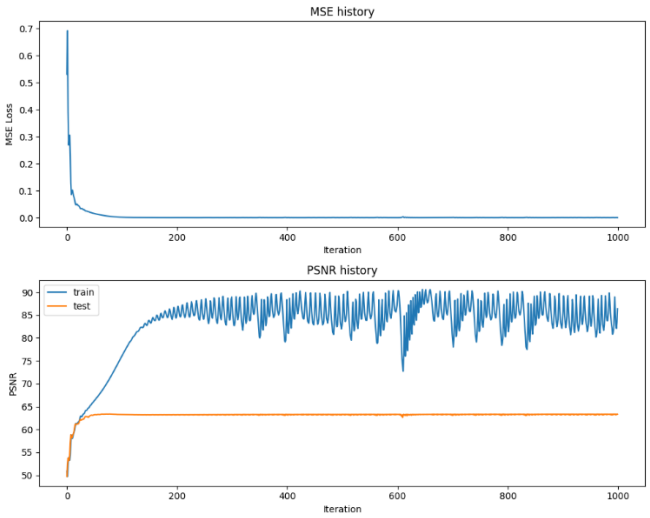
1) Sigma = 0.1

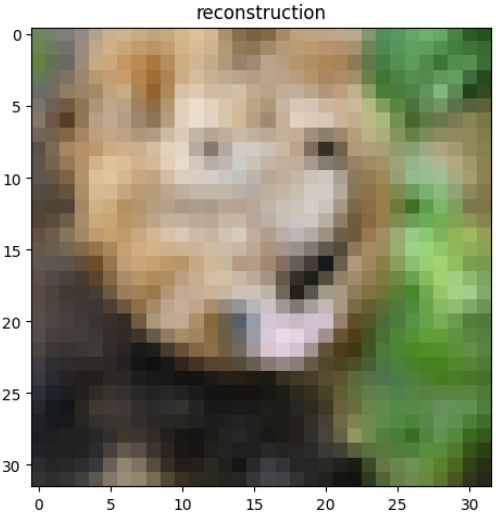




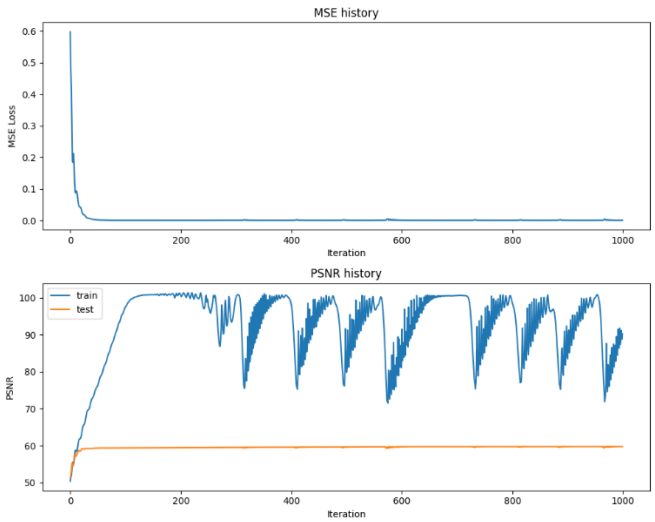
2) Sigma = 1: this is default setting, see previous parts for detail results.

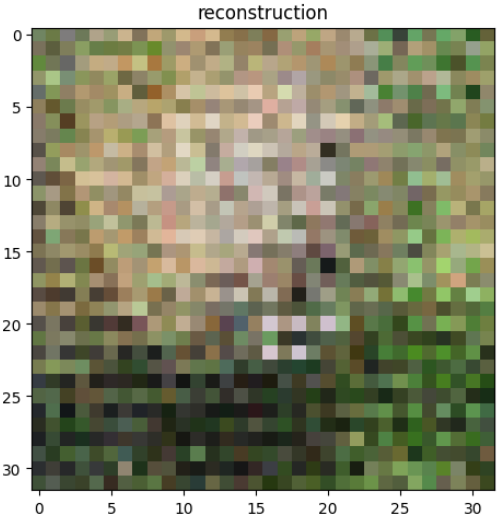
3) Sigma = 10





4) Sigma = 100





5) Discussion:
   - Difference caused by sigma can be interpreted as degree of randomness of input processing.
   - Input diversity: smaller sigma results in difference between features after mapping close to their original difference, so that the parameters learned based on this kind of mapping lead to smoother reconstruction along location. Sigma = 0.1's blurred pattern proved this finding.
   - Neighborhood information loss: larger sigma results in too much noise added to input features while mapping that the input into NN is largely based on Gaussian randomness instead of its relationship to neighbor locations. Sigma = 100's wrong colors at randomly wrong locations proved this finding.
   - Training difficulty: observed from sigma = 100, the bouncing pattern of training loss shows its struggling through update process. Large sigma adds too much noise to input that makes it not learnable.
   - Optimal: overall, sigma = 1 & 10 have optimal results.

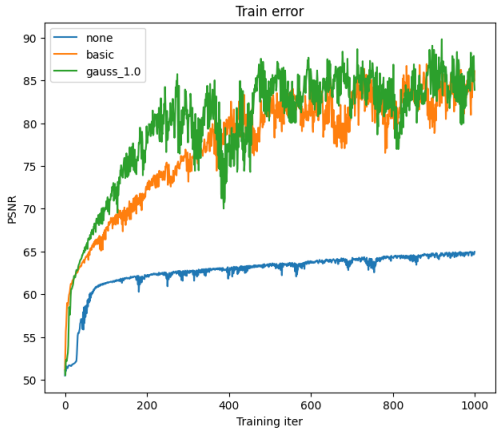
4. L1 Loss (neural_net_ec.py)
   1) Implementation:
      - L1 loss is computed based on formula:

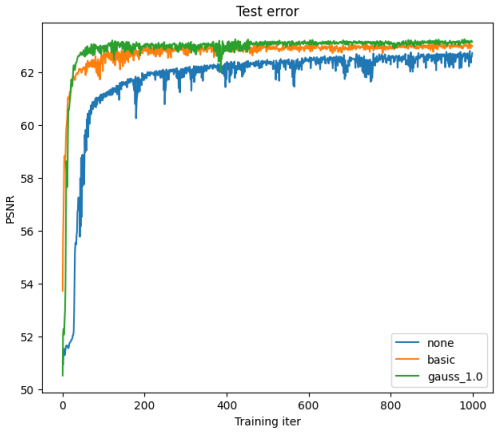
$$loss(x, y) = \frac{1}{n} \sum_{i=1}^{n} |y_i - f(x_i)|$$

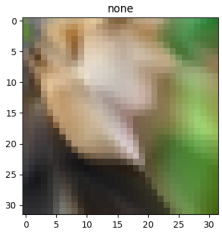
      - Gradient is computed using code:

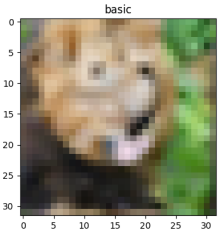
```
def l1loss_grad(self, y: np.ndarray, p: np.ndarray) -> np.ndarray:
    return (-1 * ((y - p) >= 0) + 1 * ((y - p) < 0)) / np.shape(y)[0]
```

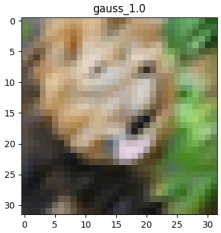
   2) Experiment result:

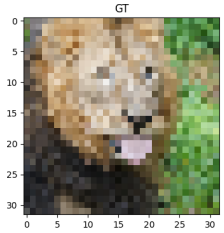










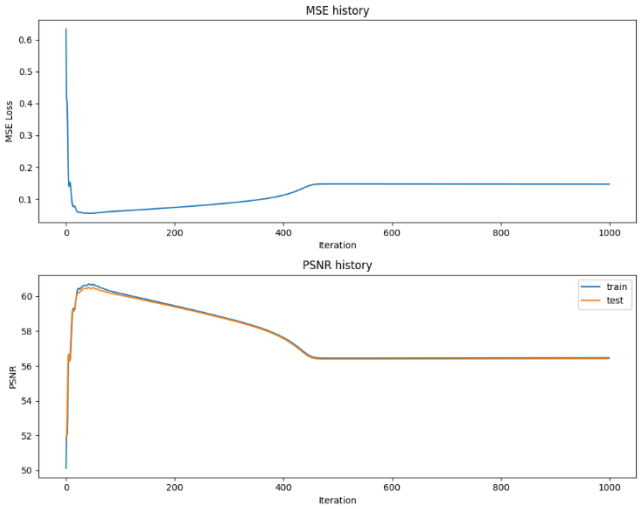


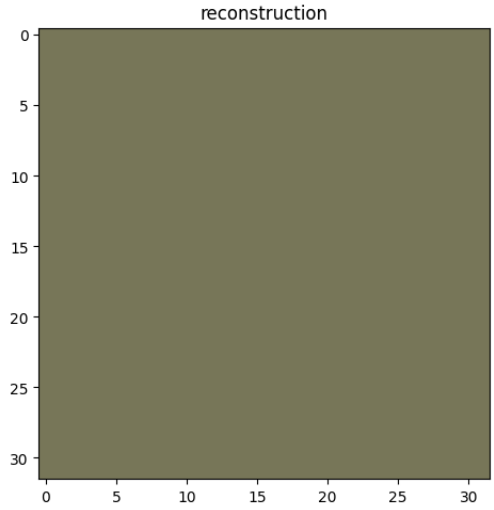
3) Discussion:
   - Sparsity: L1 loss has the property that the gradients of all samples have magnitude 1, which is independent of error (y – f(x)). This encourages sparsity among the dataset, that loss will pay attention to majority samples instead of the high-error samples.
   - Converging speed: compared with L2 (MSE) loss, L1 loss results in smoother change in training error. This is because all samples have evenly contribution towards the loss and update of parameters, so that the mean update is smoother. Besides the gradient is either 1 or -1, but L2 loss can have greater magnitude.
   - Performance: overall both losses give similar good final result.

5. Regularization (neural_net_ec.py)
   1) Implementation:
      By setting the weight_decay hyperparameter of Adam optimizer.

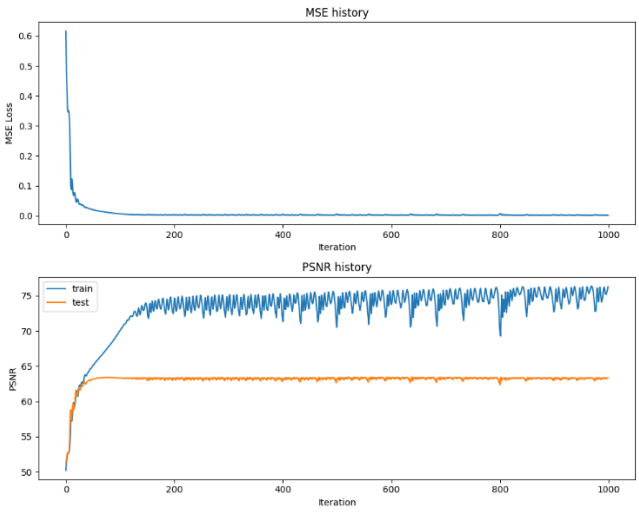
```
## regularization
if self.lamb_adam != 0:
    self.gradients["W" + str(layer_idx)] += self.lamb_adam * self.params["W" + str(layer_idx)]
    self.gradients["b" + str(layer_idx)] += self.lamb_adam * self.params["b" + str(layer_idx)]
```

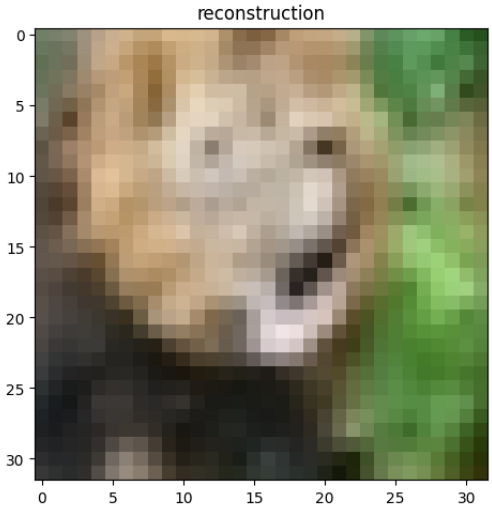
   2) Experiment:
      Take gauss_0.1 feature mapping as an example here.
      For no mapping and basic mapping, see jupyter notebook for further details.
   3) Weight_decay = 0.1



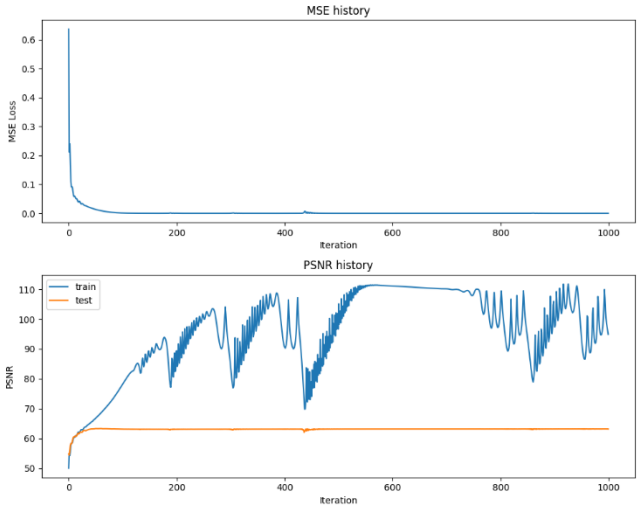


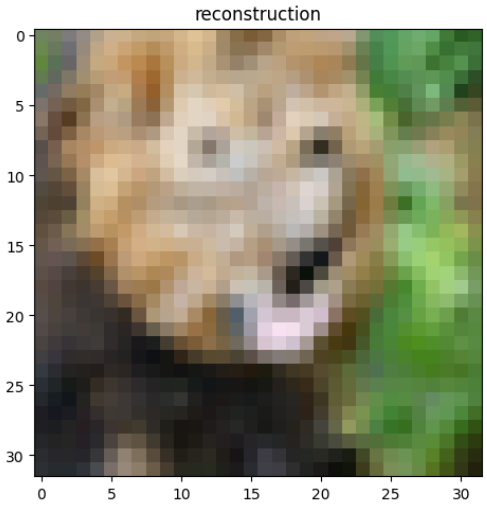
   4) Weight_decay = 0.001





5) Weight_decay = 0.00001





6) Discussion:

- Balance: regularization strikes a balance between misclassified data points and the splitting effectiveness of boundary (global margin).
- Overlook: large weight decay overlooks misclassified data points and pay more attention to global margin of decision boundary, resulting in a global mean. Weight_decay = 0.1 proved this finding, a mean single color is selected.
- Convergence difficulty: small weight decay pays too much attention to misclassified points that it may be easily misled by outliers and cannot converge. Weight_decay = 0.00001 proved this finding, training loss kept bouncing and hard to converge.
- Optimal: overall, weight_decay around 0.001 gives the optimal result.

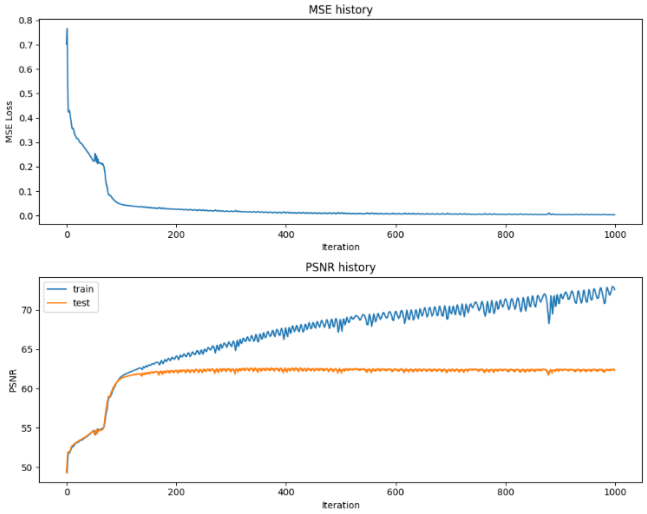
6. Normalization

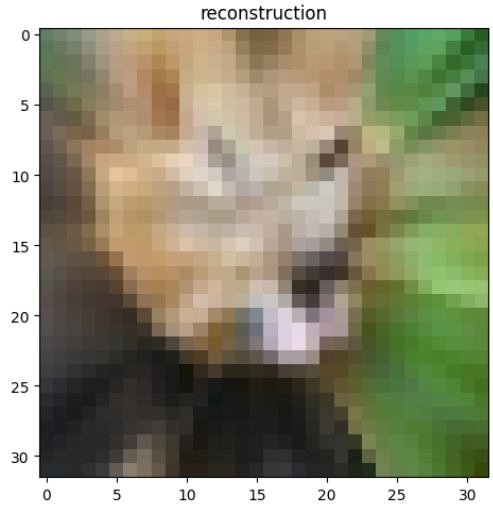
1) Implementation: at data-preprocessing stage.

- Zero-centering: subtract mean.

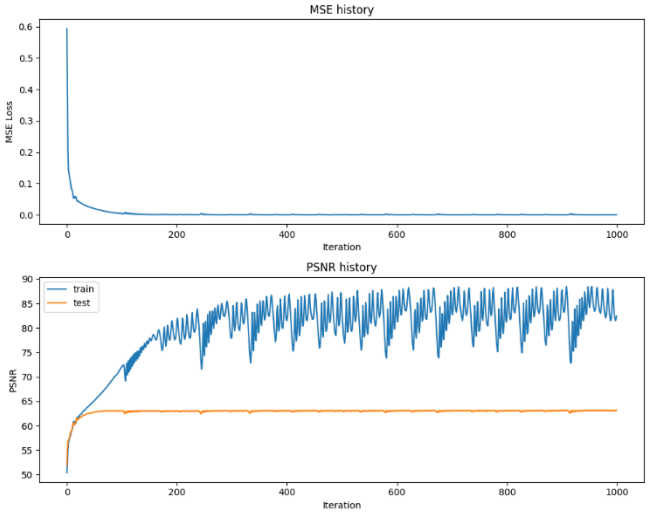
- Re-scaling: divided by standard deviation.

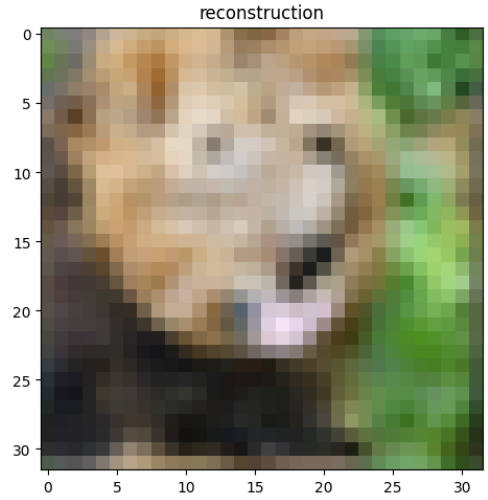
2) No mapping



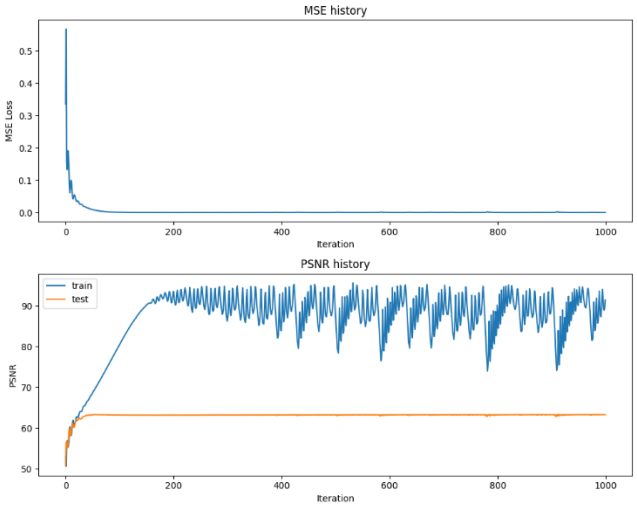


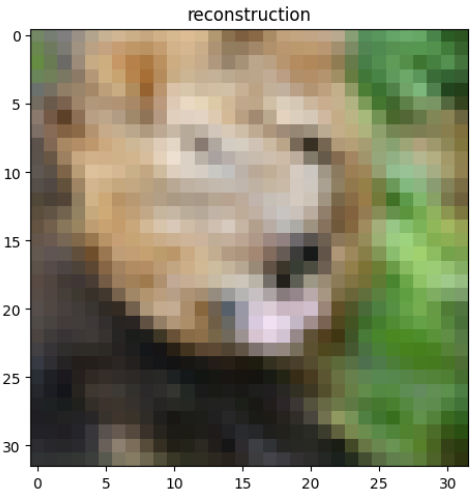
3) Basic mapping





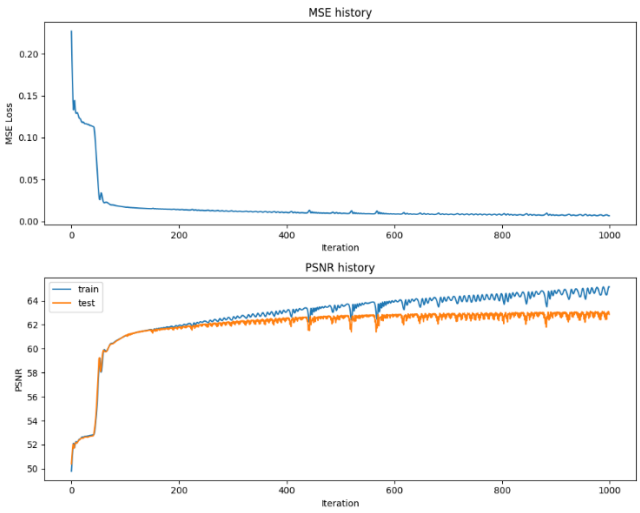
4) Gauss_1.0 mapping

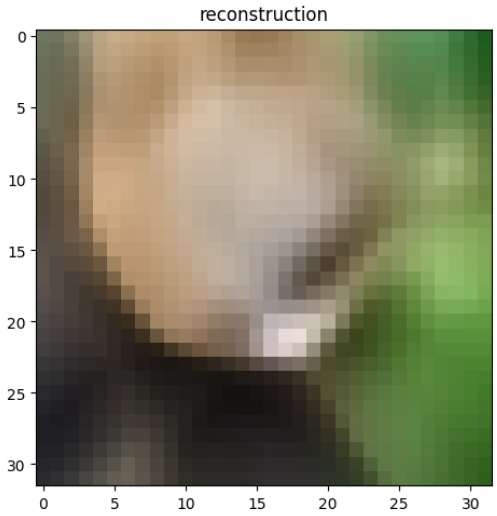




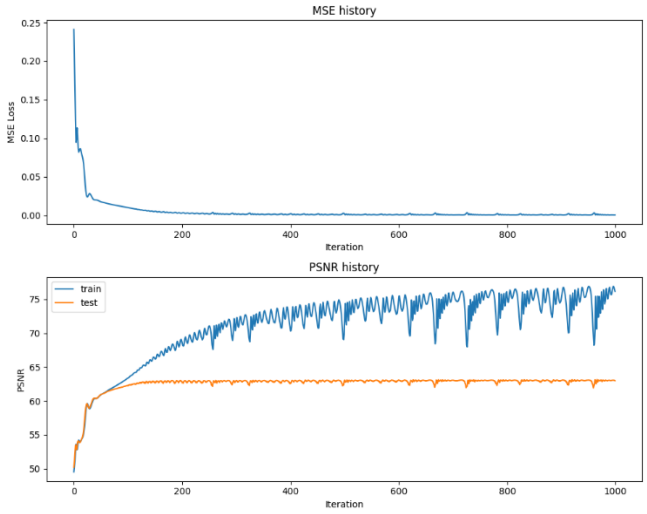
5) Discussion:
   - Better convergence: benefit of normalization is that it reduces the difference between input samples, making parameter updates more reflective to the output change and converge quickly.
   - Smooth input: normalization on huge disparity training input can smooth the difference between samples and update parameters more efficiently. This can be proved by comparing Gauss_1.0, basic and none mapping, the training loss of Gauss_1.0 converges quickly and reaches better place than basic and none, because the Gaussian randomness and high dimension mapping make input diverge. Normalization can scale down them while keeping the difference between them.
   - Impede learning: normalization on similar training input or small datasets can slow down the converging speed. This can be seen from no mapping experiment, the training loss converge slowly because the normalization reduces the difference between inputs, make features and relationships harder to recognize.
   - Optimal: overall, normalization applied on diverge or large dataset will be helpful.

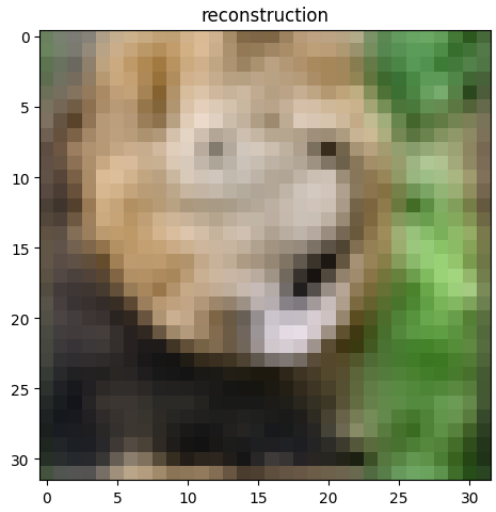
7. PyTorch (neural_net_pytorch.py)
   1) Implementation: see neural_net_pytorch.py for detailed implementation.
      - NN structure: nn.Linear, nn.ReLU.
      - Loss function: nn.MSELoss()
      - Optimizer: torch.optim.Adam()
   2) No mapping



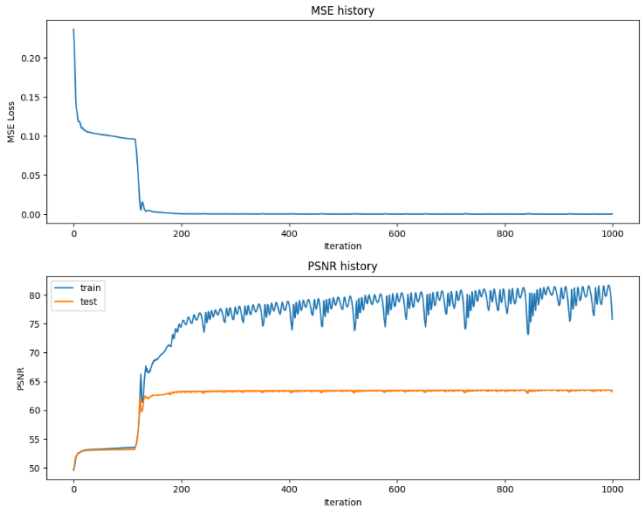


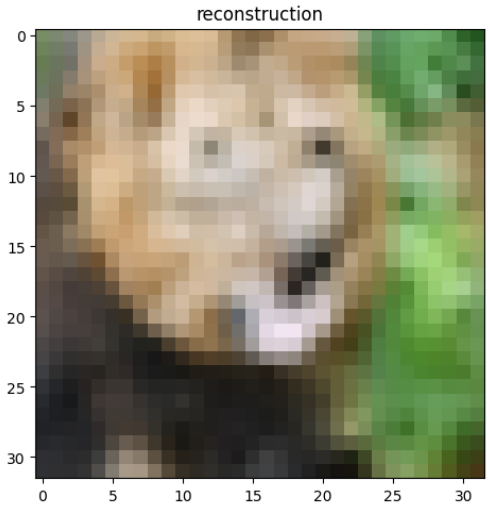
   3) Basic mapping





4) Gauss_1.0 mapping





5) Comparison:

- Training curves: Almost the same, except that there is a stable stage at around 50-150 epochs in Gauss_1.0 mapping training.
- Computational Requirements: PyTorch has a higher time and computational cost than my numpy implementation. This is mainly because tensors automatically store gradient for themselves, including features and labels, whose gradient is not needed for parameter update.
- Output quality: Almost the same, except that for no mapping case, PyTorch results in a more blurred version without obvious edges.