To what extent is the feasibility of adapting Autoencoder neural networks for image compression?

Word count: 3,992

CS EE World
https://cseeworld.wixsite.com/home
27/34 (A)
May 2025

Submitter info:
This essay was quite rushed, as I changed my RQ one month before the EE deadline lmao. Really important to make sure that your RQ will be something you can handle. Contact me at grieferpig [at] 163 [dot] com, and spoiler alert, pony inside.

Table of Contents

Introduction

As Internet and smartphones become widespread, the demand for data compression is arising since a representation of digital images requires large amounts of data (Guojun, 1992). JPEG is a common traditional algorithm for compressing photographs and other images online (Murooj et al., 2024). However, trade-offs between file size and visual quality are often made for efficient data transmission, especially with increasing resolution of modern displays and aesthetic standards (Shayan et al., 2024). In response to these challenges, Autoencoder provides a promising alternative for compressing images into latent representations with high performance (Umberto, 2022).

Autoencoders, which consist of encoder and decoder, compress and reconstruct input data into a low-dimensional latent space. (Dor et al., 2020). Their capability to compress data into essential features make them as possible candidates for image compression, which aims to preserve visual quality while minimizing data size. Traditional dimensionality reduction methods, such as PCA (Principal Component Analysis), while could achieve comparable accuracy to autoencoders at sufficiently large dimensionality (Quentin and Daniel, 2021), are based on linear transformations and cannot effectively capture non-linear relationships in the data (Abhishek, 2023), which leads to suboptimal performance on images (Adam, 2000).

Modern computational devices focus on improving power and processing efficiency for data-drive parallel computing. Most modern processors contain an NPU (Neural Processing unit) that can offload AI-related tasks from CPU, e.g. speech recognition and background blurring, with low power and comparable performance. (Techradar, 2024). Implementing a compression method that is inherently accelerable by NPU would allow for parallel decompression of images with lower energy consumption, increasing the throughput during batch decompression scenarios.

This paper proposes using Autoencoder models to encode and decode images for compression. To investigate the influential factors of efficiency, various Autoencoder models were programmed with different number of encoder/decoder layers, size of latent dimensions, and size of image set.

Background information

Convolutional Neural Network (CNN)

CNNs are a type of deep learning neural networks designed to process data in a grid fashion like images (LeCun et al., 2015). Their ability to learn spatial structures makes it widely used in computer vision. CNNs are composed of convolutional layers that learns a low-dimensional filter and apply them to the input in order to extract feature map that represents a pattern within an area, such as sharp edges and texture (Krizhevsky, 2012). To apply such filter, as seen in figure 1, the kernel “slides”

around the input grid starting from the left and multiply the kernel with the area underneath (image patch). The element-wise product of the two matrixes determines the value of the output image. By combining multiple kernels, a CNN is able to extract intricate features to be used for identification or embedding (transform an image to a latent vector that describes the features of such image) tasks.

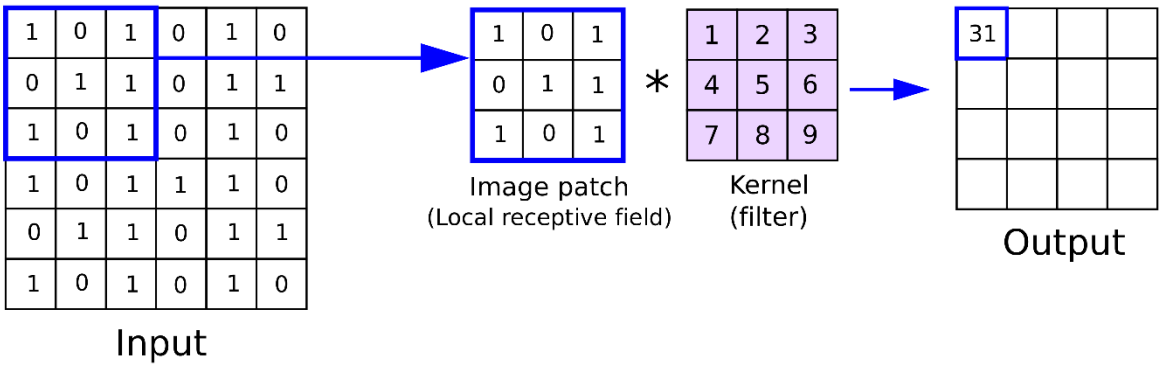


Fig. 1. A visualization of CNN kernel sliding mechanism (Elyasi, 2020)

Autoencoder

Autoencoders use unsupervised learning to learn lower-dimensional representation of unlabeled data (Wikipedia, 2024). They consist of two functions: encoder $E(x)$ and decoder $D(x)$. The encoder takes an image x as the input, passes through several layers of dense layers, with the goal of learning a compact latent representation c (known as the bottleneck). The decoder, which shares a similar but reversed structure with the encoder, tries to recreate the image x' from the latent representation. To compress the input data, each layer is designed to progressively reduce the dimensions of the data, to capture the most prominent features while minimizing the significances of noises (Hinton and Salakhutdinov, 2006). Several

variations of autoencoders have been introduced, e.g., Denoising Autoencoder (Vincent et al., 2008), Variational Autoencoder (Kingma and Welling, 2013), and Convolutional Autoencoder (Masci et al., 2011). The Convolutional Autoencoder variant is used for its superior ability to maintain the spatial arrangement of the image data and drastically fewer trainable parameters that make them less prone to overfitting and more scalable for large datasets (Masci et al., 2011).

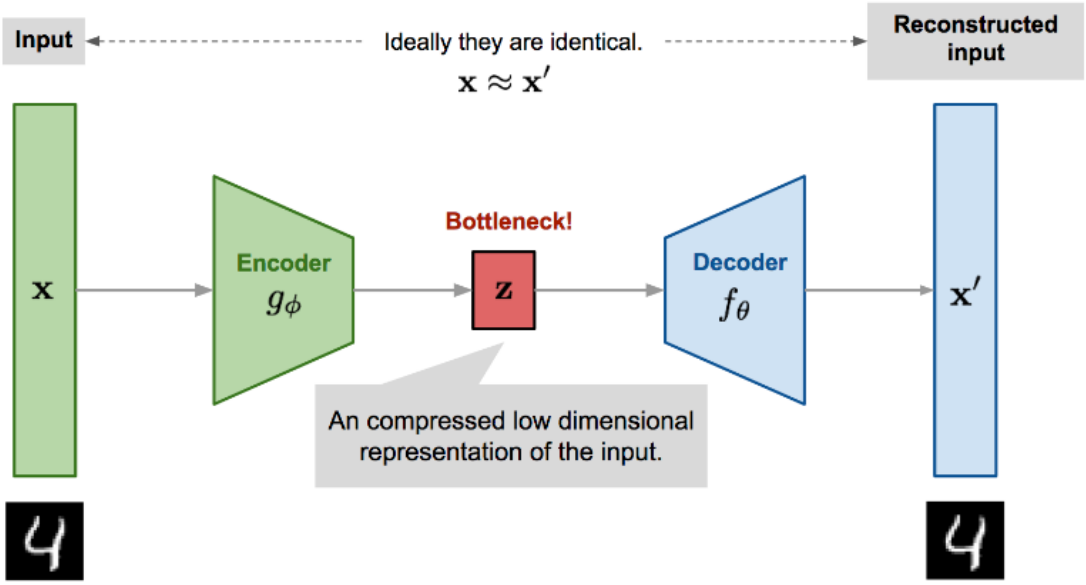


Fig. 2. A visual demonstration of the structure of an Autoencoder network (Lucas, 2020)

Methodology

The primary experimental data is the main source of data in this paper. The scarcity of secondary data sources on the topic of this paper led me to an experimental methodology, where independent variables are flexible and easily controlled. To streamline the experiment process, the Autoencoder models are defined with four parameters: input image dimension (`img_size`), encoder/decoder layer amount

(`num_layers`), latent vector dimension (**`latent_dim`**), and image set size (**`img_set_size`**) (appendix A, adapted from (Hasan, 2024)). To test the model's scalability, four model sizes are defined by the following specification. These models are chosen due to their balance on compression ratio and reconstruction quality (see **Results and Analysis**). For ease of identification, the model's hyperparameters will be written in (**`num_layers, img_set_size, latent_dim`**).

Name	`img_size`	`num_layers`	`img_set_size`	`latent_dim`
small	256	2	64	16
base	256	2	256	32
large	256	2	1024	64
xlarge	256	2	4096	128

Table 1: Hyperparameters of the defined models

The loss value, average MSE (mean-squared error), PSNR and SSIM value are recorded after each run, with 4 example images selected randomly from the dataset to evaluate its performance at reconstructing image through qualitative analysis. To further enhance the reconstruction quality and compression ratio, optimizations will be applied, including adding residual connections, dynamic quantization and appending a denoiser model. Due to the lossy nature of the autoencoder model introduced by the difficulty to fully converge to complex input, the reconstruction quality will be compared with JPEG, a commonly used lossy image compression format that is efficient at compressing photographic images used in the dataset (Wikipedia, 2024).

A limitation of this methodology is the dataset's high quality images, meaning that for more commercial applications, such as image backup services, the datasets are not representative of these use cases, as blurry and poorly taken images are abundant.

The demanding requirement of training hardware also restricts training larger models, hindering the potential performance of this network structure at larger scale.

Proposed model structure

The proposed model structure is dynamically determined based on three hyperparameters: image_size (dimension of input images, defaults to 256x256), **`num_layers`** (number of convolutional layers), **`latent_dim`** (size of the bottleneck vector). Later variations add residual connections (skip connections between layers) and a denoiser model, to help improve the reconstruction quality. These changes are marked as red and blue on the following figure. Below is an architecture demonstration of the small model.

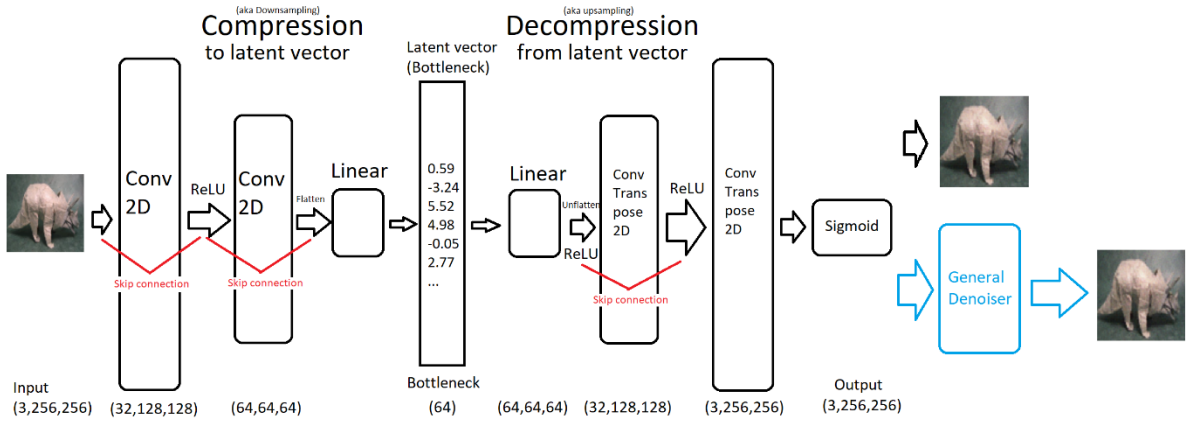


Fig. 3: Overview structure of the model structure, using the small model as example

Datasets

Each of the models was trained on a randomly sampled subset *mini_imagenet* (Google, 2024), a minimal version of the ImageNet dataset developed by Google DeepMind. The dataset contains 60000 images of 100 categories, 600 images each.

Images are resized to 256x256 to balance between computational load and visual clarity. Each image is converted to RGB format and normalized to a float32 value between 0 and 1.

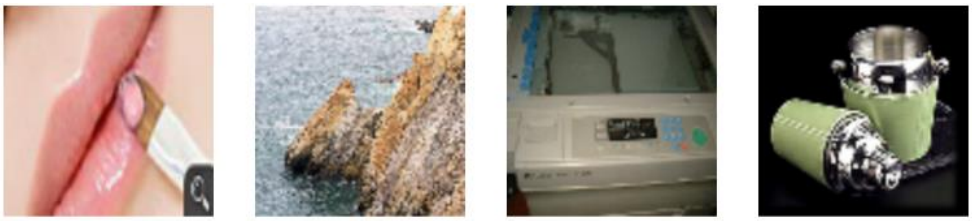


Fig. 4: Sample images from the dataset

Evaluation Metrics

Mean-squared error (MSE)

Mean-squared error is a measure of relative deviation between two data. It measures the average distance between two data samples using the following formula. (Wikipedia, 2024) A high MSE value indicates errors, while an MSE value of 0 indicates perfect reconstruction.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - Y_i')^2$$

Where n is the number of samples, Y is the observed value set and Y' is the expected value set. As it is a relative indicator about the expected value, the value represents the variance and bias of the observed (i.e. reconstructed images) compared to the ground truth (original images). Thus, this metric is indicative of the output image's noise level (variance) and color degradation (bias). However, as humans are not sensitive to mean-squared error in images (Wikipedia, 2024), this

metric is used to measure unnoticeable detail loss between images.

Structural Similarity Index (SSIM)

SSIM is a perceptual metric that measures degradation between two images. It shows a better correlation with human perception of quality than pixel-based metrics like MSE (Wang et al., 2004). It uses the original image as reference and considers luminance, contrast and structure of the image to determine perceived errors (Wikipedia, 2024). The formula of SSIM is given as:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

Peak Signal-to-Noise Ratio (PSNR)

PSNR measures the ratio between the maximum boundary of a signal, in this case, the flattened image pixels, and the level of noise. A higher value indicates a better quality of the reconstructed image. The formula is given as:

$$PSNR = 10 \cdot log_{10}\left(\frac{I_{max}^2}{MSE}\right)$$

Where I_{max} is the maximum value of a pixel (1.0 normalized). This metric is used to measure the noise level of output image that is evident at small latent dimensions in this experiment.

Compression ratio

Compression ratio measures the reduction in size of the compressed data compared

to the uncompressed data. Traditionally, compression ratio is defined as

$$Compression\ Ratio = \frac{1}{\mathrm{n}}\sum_{i=1}^{n}\frac{Compressed\ Size}{Uncompressed\ Size}$$

Where n is the number of images. A smaller ratio indicates better compression efficiency. However, since decompression of the proposed model requires the decoder weights (containing all images) and a latent representation, the compressed size will be calculated as an average size per image as

$$Compression\ Ratio = \frac{\frac{Model\ checkpoint\ size}{n} + Latent\ size\ per\ image}{Uncompressed\ size}$$

In this experiment, since the model is trained on 256×256 images using rgb888 format, where each pixel takes 3 bytes, the uncompressed size is

$$Uncompressed\ size = 256 \times 256 \times 3 = 196608\ bytes$$

Experiment Procedure

The images are shuffled and resized to 256x256 and normalized to float32 of range $[0,1]$. As the goal is to overfit the models, the dataset is not divided into train, validation, or test subsets. Mixed-precision training (using 16-bits float instead of 32-bits for certain weights) is applied to accelerate training and reduce memory usage while keeping comparable results to normal training (Carilli, 2024). When training, the whole encoder-decoder model is trained with input of the images and expected output of the images. The embeddings of respective images and the decoder network is then separated from the model to test decompression efficiency based on MSE, PSNR, SSIM, and compression ratio.

To determine whether the model achieves convergence to stop training, in every 100 epochs, a linear regression will be performed on the past 100 loss values, and the SSIM value on last checkpoint will be considered. If the slope of the linear regression function is below 1×10^6, and the difference between current and last SSIM value is smaller than 1×10^3, then the model is considered converge and the training process stops.

When training the denoiser network, however, a general model that applies to model of different parameters is expected. To condition the model specifically on reconstruction noise, pairs of clean and reconstructed images from base and large model are inferenced and saved locally. The saved dataset is then separated into train, validation and test subsets with percentage $80\%, 10\%, 10\%$ respectively. The model trains on train subset and validate on validation subset every epoch. After training, test subset is used to test the performance of the denoise model on unseen images. The same metrics except compression ratio are used to evaluate the denoise model.

Results and Analysis

Mass testing

To determine an optimized state of the model, tests are conducted from any combination of

$$\texttt{num\_layers} = [1,2,3,4]$$

`latent_dim` = [16,32,64]

`img_set_size` = [32,64,128,256]

Increasing the model depth help the model to learn more complex features, as the added filters in convolutional layers can learn more intricate features, resulting in faster convergence, as shown in the training loss curve (Fig. 5a). However, due to vanishing gradient problem, where previous layers failed to adjust their weight effectively through backpropagation, the reconstruction quality degrades for deeper models. This results in suboptimal reconstruction quality, as shown Fig. 5b and 5c.

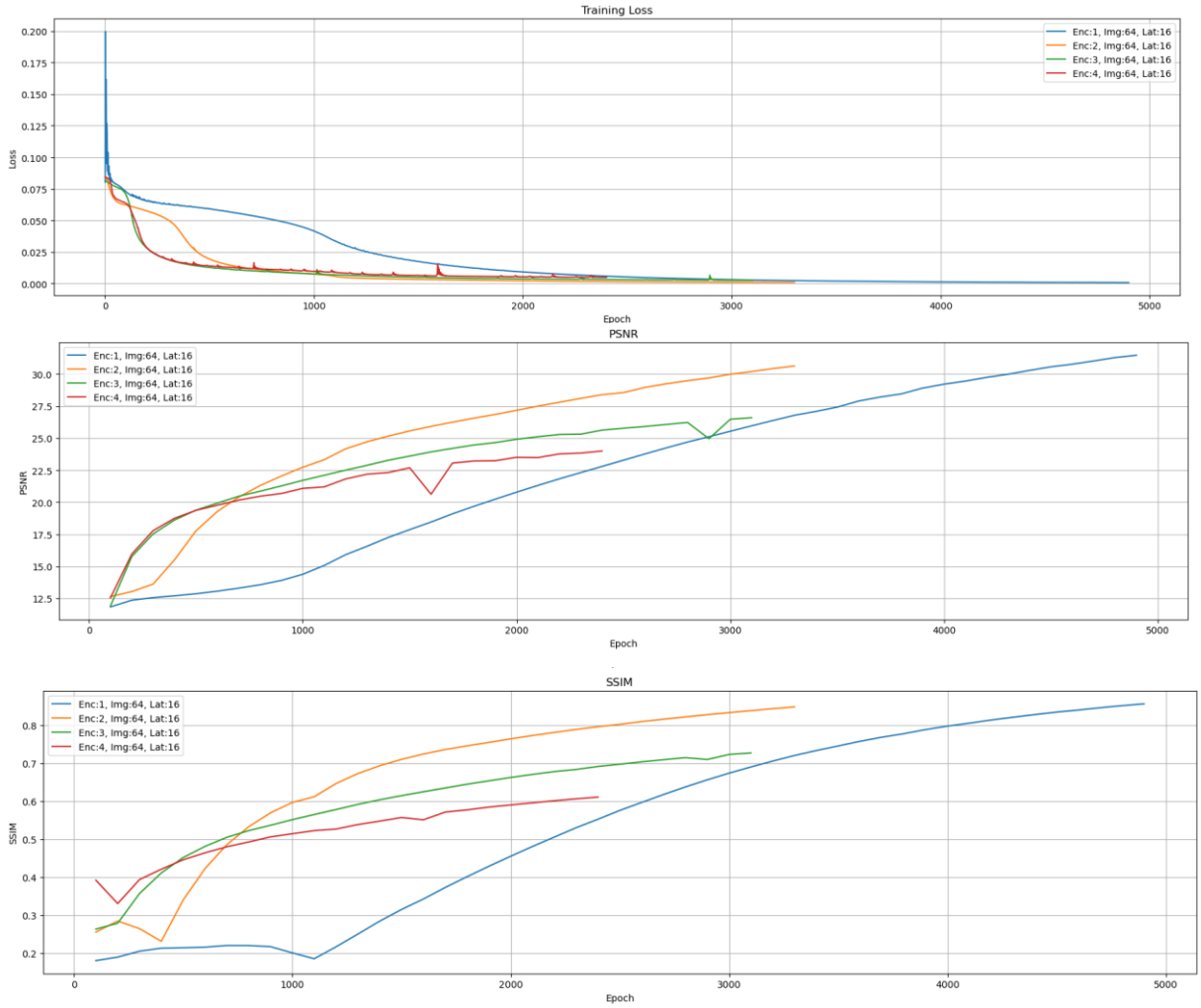
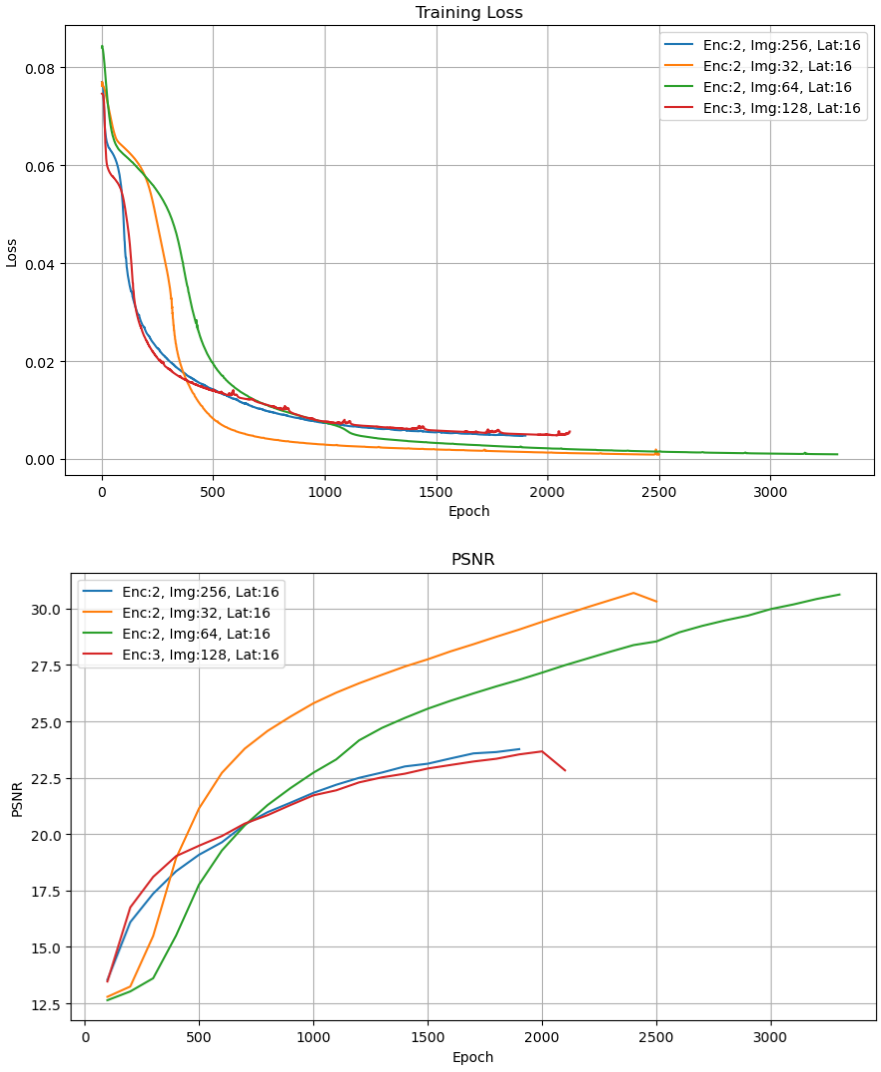


Fig. 5a, 5b, 5c: Training Loss, PSNR and SSIM curve at different **`num_layers`**

As shown in Fig. 6a, models tend to converge faster on larger datasets than small

datasets, due to their lower diversity in data. This allows the model to reuse common filters in the convolutional layers for common objects, resulting in more efficient feature extraction. However, as the amount of information the encoder passed to the decoder is constrained by the size of the bottleneck, the model struggles to describe the difference between images in the larger dataset, resulting in worse PSNR and SSIM values (Fig. 6b and 6c).



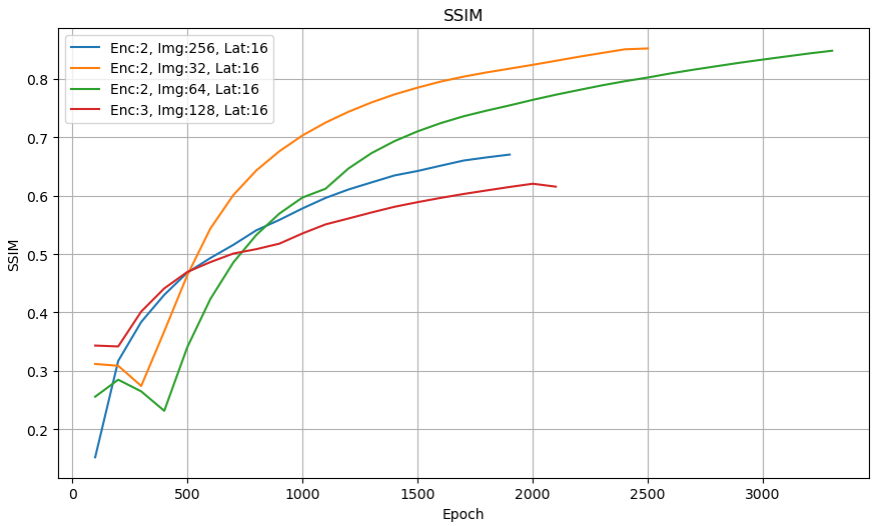
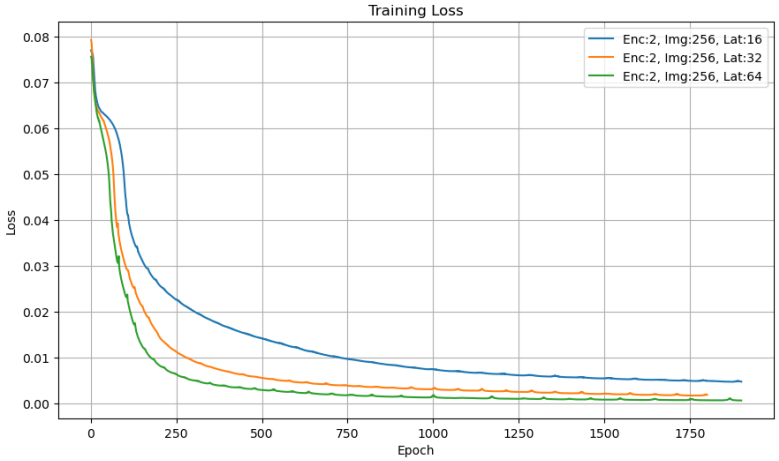


Fig. 6a, 6b, 6c: Training Loss, PSNR and SSIM curve at different ***`img_set_size`***

Larger `latent_dim` solves the problem of insufficient features above, resulting in faster model convergence and superior reconstruction quality (Fig. 7a, 7b, 7c). However, increasing `latent_dim` means increasing the output size of the linear layer, resulting in larger checkpoint file size.



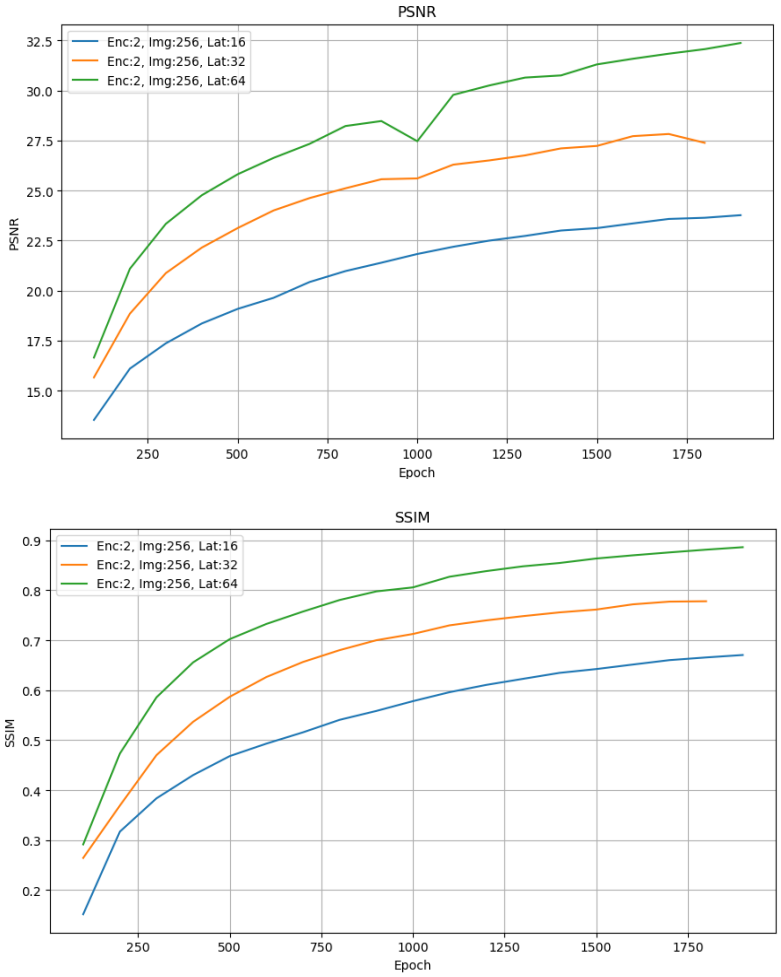


Fig. 7a, 7b, 7c: Training Loss, PSNR and SSIM curve at different ***latent_dim***

Extreme conditions

Two extreme conditions, **img_set_size** > 4096 and epoch $= 99999$ are tested to evaluate the model's theoretical scalability and reconstruction quality. Fig. 8b and 8c shows that for extremely large **img_set_size**, although many common features exist in dataset, the bottleneck is too small to pass through any significant difference, resulting in a noisy reconstruction image. The larger the dataset is, the more the model's reconstruction quality fluctuates at early stages of training. However, the loss stabilizes at around 1000 epochs, indicates the model has reached its capacity limitation. Thus, it is necessary to increase **latent_dim** as **img_set_size**

increases.

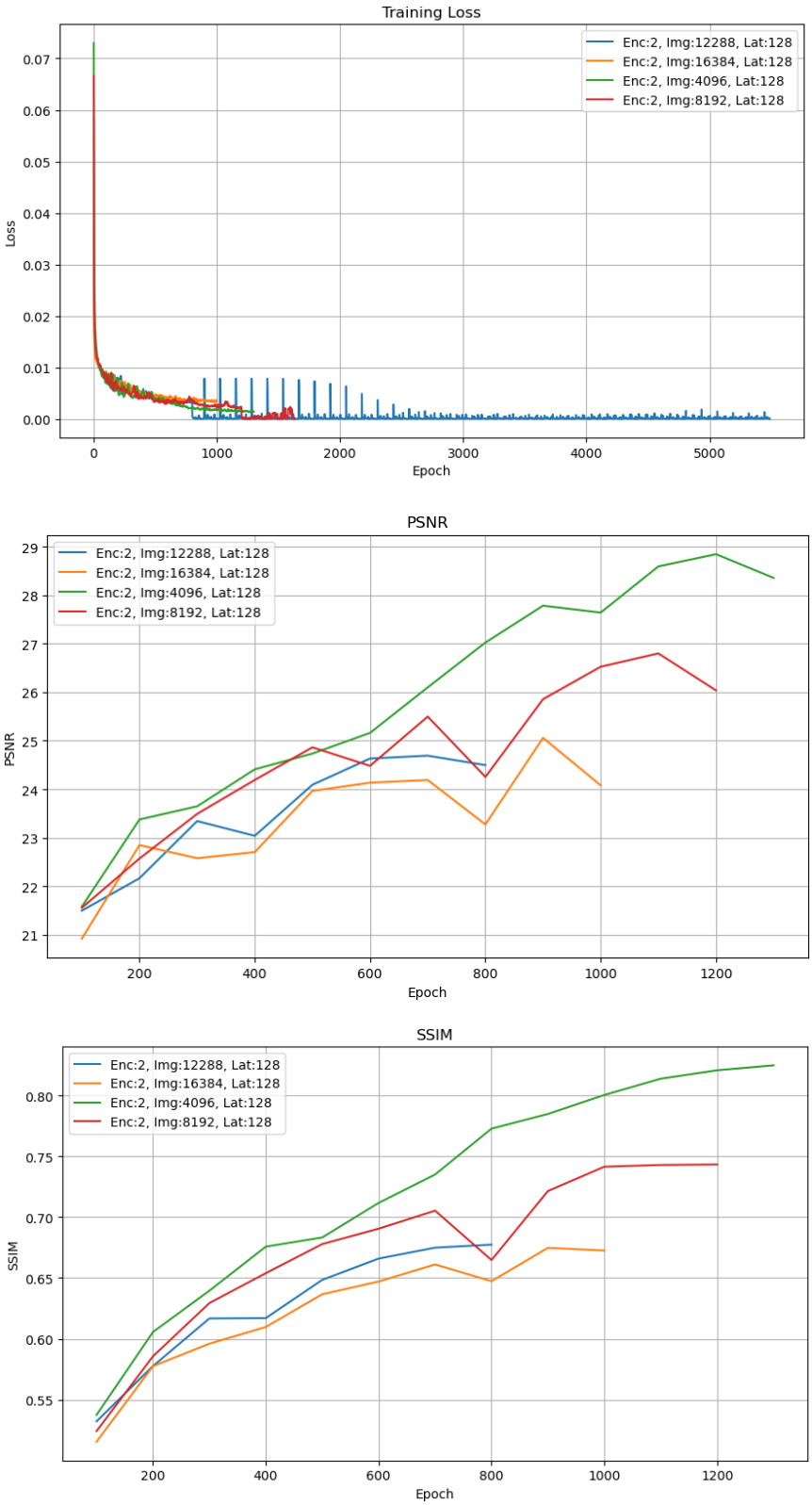


Fig. 8a, 8b, 8c: Training Loss, PSNR and SSIM curve at extreme

Due to insufficient computational power, only the small model is tested with

epoch=99999. This model demonstrates the best reconstruction quality in this experiment (Fig. 9), with PSNR=42.99 and SSIM=0.9880, approximately equal to JPEG algorithm at 90% quality and visually indistinguishable from its uncompressed counterpart. The cost is a 5733% increase in training time (4.33hr vs 4.5min), which makes it impractical for actual use. For this reason, the convergence detection algorithm in “Experiment Procedure” is established to balance training time and reconstruction quality.

Fig. 9: Reconstructions from the epoch=99999 model

Specialized Dataset

Training on a specialized dataset is also tested. Specialized datasets refer to datasets that are homogeneous (containing data that is similar to each other). In this case, the small and base model is trained on 64 cartoon drawings of a character (Appendix. B), with the following results:

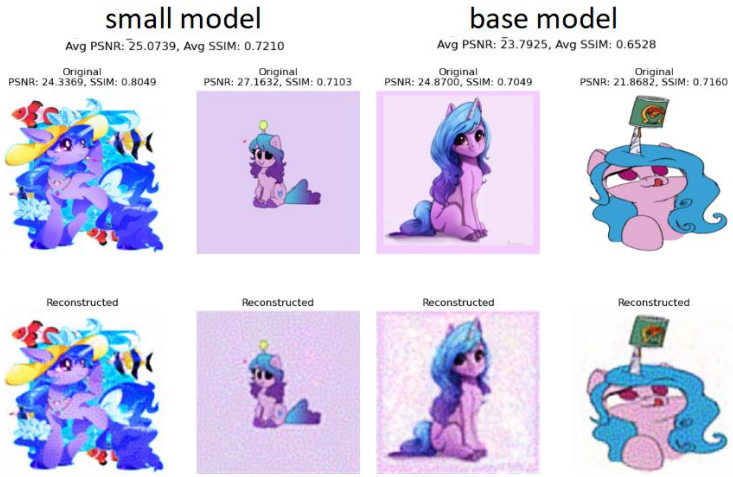


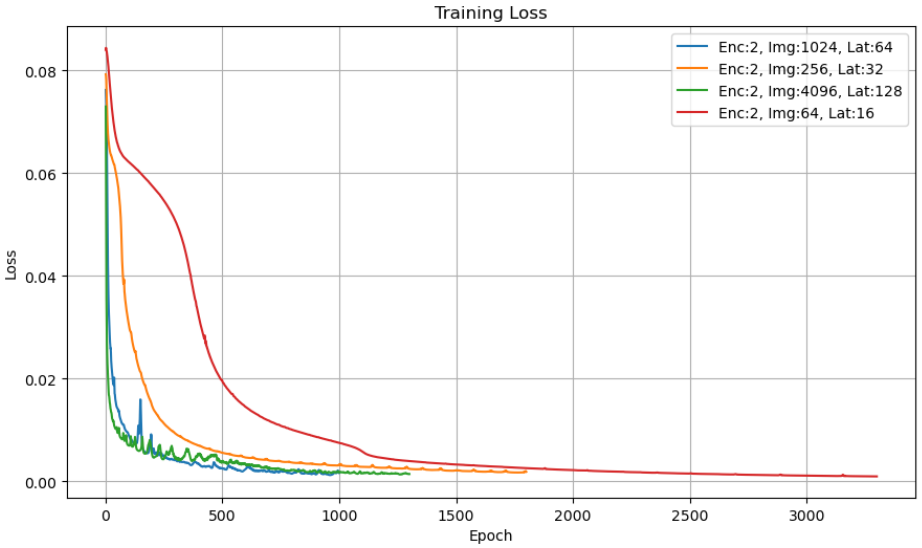
Fig. 10: Sample outputs from small and base model that is trained on a homogenous dataset

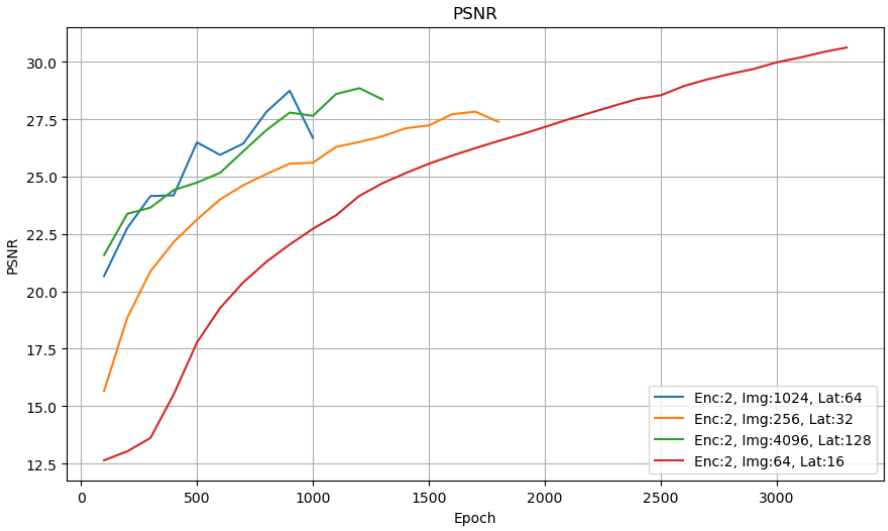
While in theory, a homogeneous dataset will result in easier training as the variance is small, training on such dataset results in a significant reconstruction quality degradation, as shown in a 17% decrease in SSIM value and significant visible noise, compared to non-specialized counterparts. The variance for the selected dataset may be too small for the model to overfit to certain small features in training data, resulting in reconstruction noise. Hence, using a heterogeneous dataset will yield best results for training this model.

Selected Models

Considering factors above, the four model categories are chosen for their balance on convergence speed, reconstruction quality, and representation of scalability. As shown in Fig. 11c, all the models achieved a similar SSIM score of around 0.8, indicating the model's scalability. However, when inspecting visually (Fig. 12), the reconstructed image suffers from reconstruction noise, color degradation, and loss of detail. This is due to underfitting caused by vanishing gradient, that the model fails to

capture the structural and chromatic differences between images, leading to an average-out reconstruction.





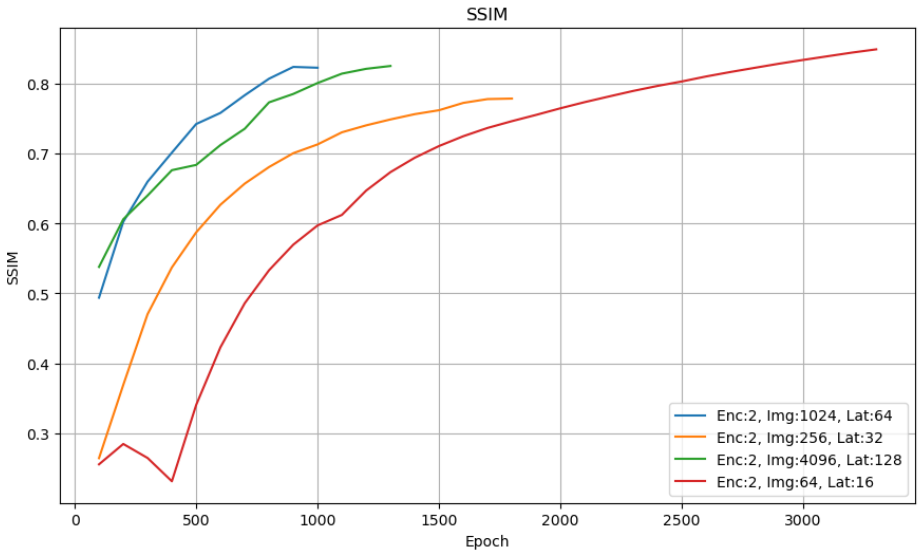


Fig. 11a, 11b, 11c: Training Loss, PSNR and SSIM curve at different model size

Fig. 12: visible noise, color degradation and loss of detail

Optimizations

Residual connections

Residual connections, introduced originally in ResNet (He et al., 2015), address challenges like vanishing gradients in deep networks. They work by allowing the input of a layer to bypass in-between layers and be added directly to the output,

creating shortcuts for backpropagation that ensure substantial gradients in early layers. Suppose x is the input of a neural network layer and $H(x)$ is the intended behavior, the residual network restructures the network to learn a residual function $F(x)$ so that

$$F(x) = H(x) - x$$

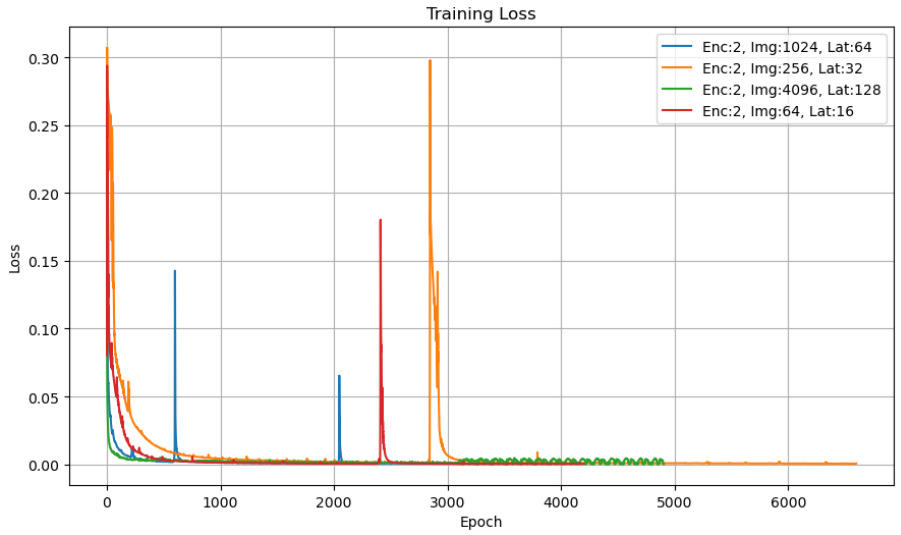
Hence, the original function becomes

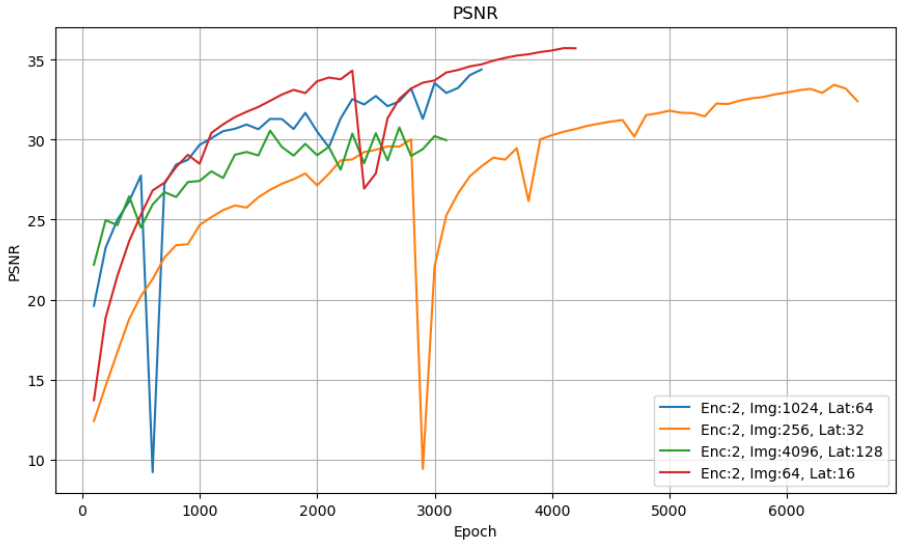
$$H(x) = F(x) + x$$

This allows the network to focus on learning the difference between inputs and outputs rather than completely transforming the input. This helps latter layers to gain understanding of outputs from previous layers, which provides structural/detail information that helps to recreate the image. This also solves the vanishing gradient problem, allowing gradients to flow directly from the output back to earlier layers, enhancing the stability of learning.

As shown in Fig. 13b and 13c, the new models show improved PSNR and SSIM, e.g. large model's SSIM value increases from 0.8021 to 0.9224 and a speedup in convergence, as shown by the loss curves. Interestingly, unlike older models that achieve different PSNR and SSIM values, the newer models all achieved a SSIM of 0.91 ± 0.01, indicating better scalability, except for the xlarge model that achieved a SSIM of 0.86, presumably is still underfit based on the trend of PSNR and SSIM curve. Visually, the new models show no color degradation or blurry details, albeit with slightly reconstruction noise (Fig. 14). Since the dimension of input and output stays same after residual connection, the models' size and therefore compression

ratio does not change. Hence, incorporating residual connections significantly benefits the model in terms of reconstruction quality.





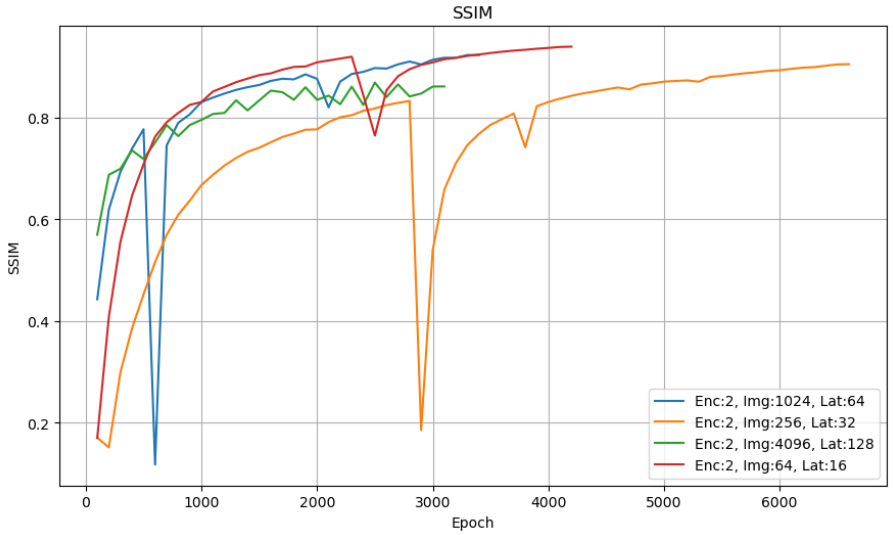


Fig. 13a, 13b, 13c: *Training Loss, PSNR and SSIM curve after implementing residual connections*

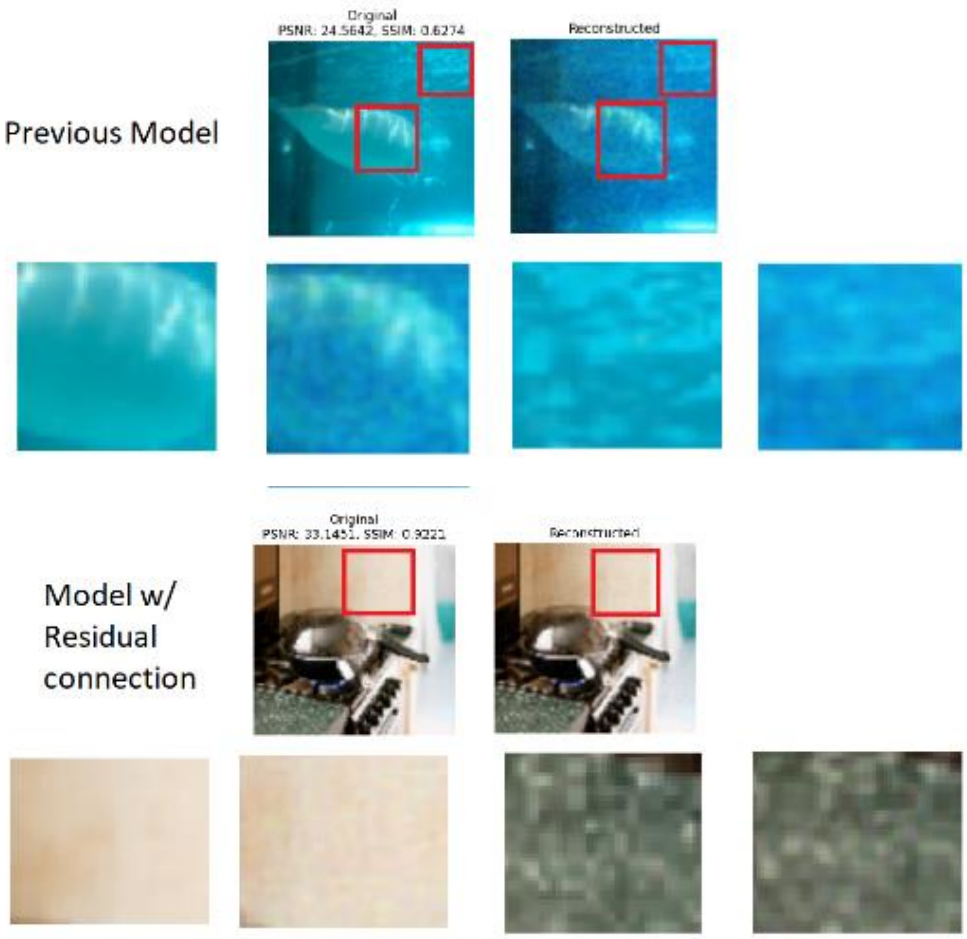


Fig. 14: Comparison of model with and without residual connections

Denoising

A more efficient way to reduce noise in reconstructed images is to apply a denoising network after reconstruction. Denoiser are a type of algorithm that removes noise from input images to increase color coherence and sharpness (Kim). Considering the client-side application of this model, the denoiser model should be as lightweight and efficient as possible. Due to this, the model proposed in *"An efficient lightweight network for image denoising using progressive residual and convolutional attention*

feature fusion" (Wang et al., 2024) is selected for its small model size (< 1 M parameters) and state-of-the-art performance on denoising images. As shown below, the noisy image is first passed through a convolution layer to extract basic features, then uses three dense and convolution layers to progressively extract deeper features. The residual connection ensures that both shallow and deep features are passed onto the next layers. The CAFFM module employs channel attention mechanism, shown in Fig. 16. Since not all channels are equally important, the channel attention mechanism generates an attention weights map that indicates the importance of certain features, allowing the network to selectively focus on important features.

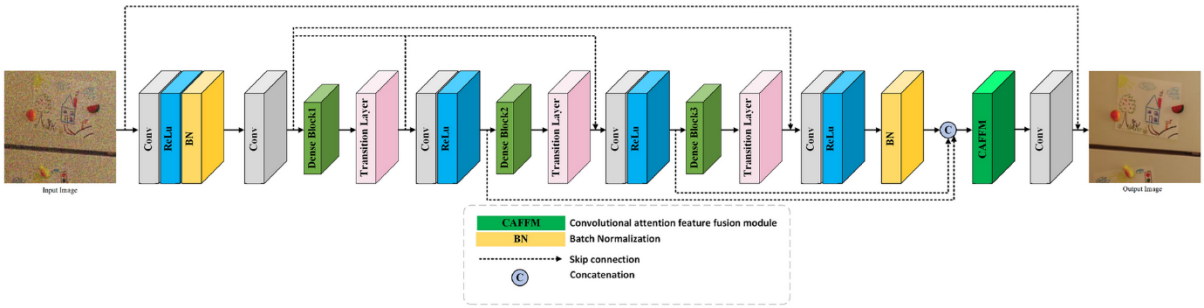


Fig. 15: Model structure of the denoiser model

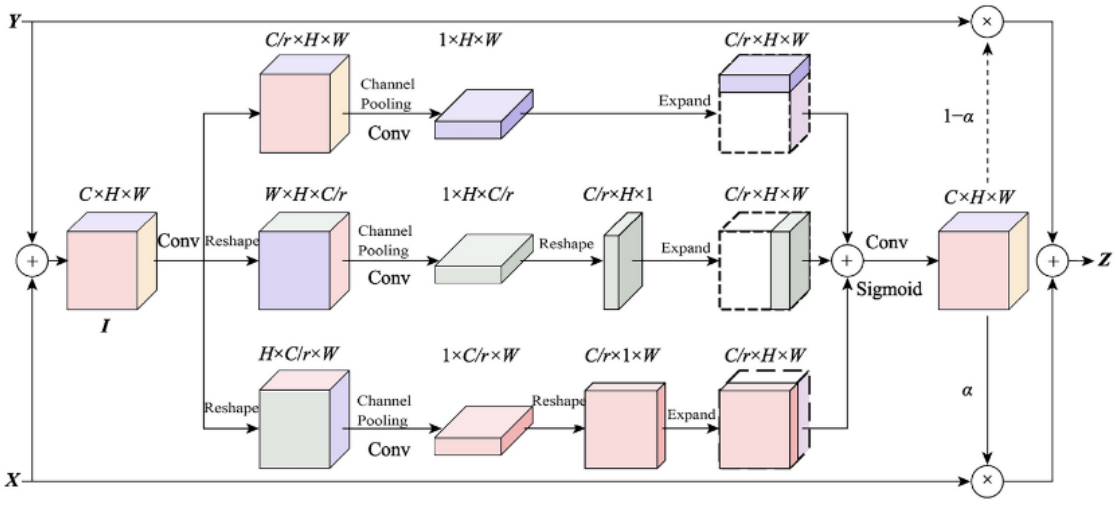


Fig. 16: Structure of CAFFM module

Training this denoiser model involves preparing pairs of clean and noisy (reconstructed) images. A total of 1280 pairs of images from the base and large models are gathered and trained on the denoiser model for 100 epochs, achieving a 0.00252 loss on train dataset and 0.00246 loss on validation dataset. Four results randomly selected from the test dataset are shown in Fig. 17. The denoiser model effectively reduces noise where areas of solid color exist (left), but when encountering complex textures, the model failed to reconstruct details due to insufficient information caused by the noise (right). Nonetheless, appending a denoiser network results in a slight increase in PSNR and SSIM, e.g. the small model increases its SSIM value from 0.9343 to 0.9562, while the xlarge model increases from 0.8875 to 0.9249

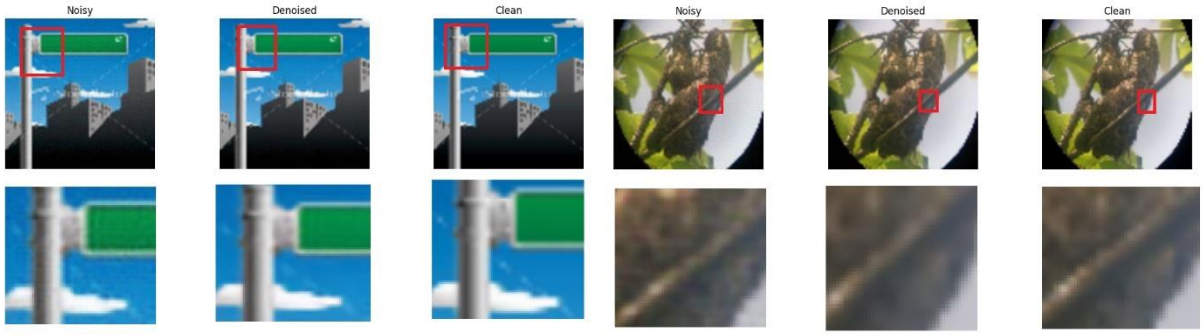


Fig. 17: Comparison of reconstructed (noisy), denoised and original (clean) image

Quantization

Quantization is a memory optimization technique that significantly reduces memory requirements, at a cost of slightly decreased accuracy. In theory, quantizing the model to int8 (8-bits integers) should reduce the model size by 4x, because the weights can be represent using $\frac{1}{4}$ bits compared to the original format. However,

since ConvTranspose2D layers do not support quantization in PyTorch, the actual size reduction rate for my model ranges from 3.33 (small) to 3.89 (xlarge), since the larger `latent_dim`, the larger proportion the quantizable linear layer is in the model file.

Evaluation

Quantitative Evaluation

The model is benchmarked against JPEG to test its performance against traditional compression methods. The performance of JPEG is measured by its reconstruction quality (indicated by PSNR and SSIM) and compression ratio. To factor both PSNR and SSIM into account, a reconstruction quality score is defined as follows:

$$Q = \alpha \times \frac{1}{n}\sum_{i=1}^{n}\left(\frac{PSNR_i - \min(PSNR_{all})}{max(PSNR_{all}) - \min\left(PSNR_{all}\right)}\right) + (1-\alpha) \times \frac{1}{n}\sum_{i=1}^{n} SSIM_i$$

Where α is a weight factor controlling the importance of PSNR and SSIM scores. To consider both PSNR and SSIM equally, $\alpha = 0.5$ is used for balance both PSNR and SSIM with equal importance.

Since JPEG will output different file sizes depending on the complexity of the image, a set of 25 samples are randomly selected from the dataset and resized to 256×256. Each sample is tested against 10 compression qualities, ranging $[10\%, 100\%]$. The average of reconstruction quality and compression ratio is calculated for each tested compression quality and plot as a curve. Each model size is evaluated with and without quantization and denoising on all training datasets. The comparison

between JPEG and the models are shown in the graph below.

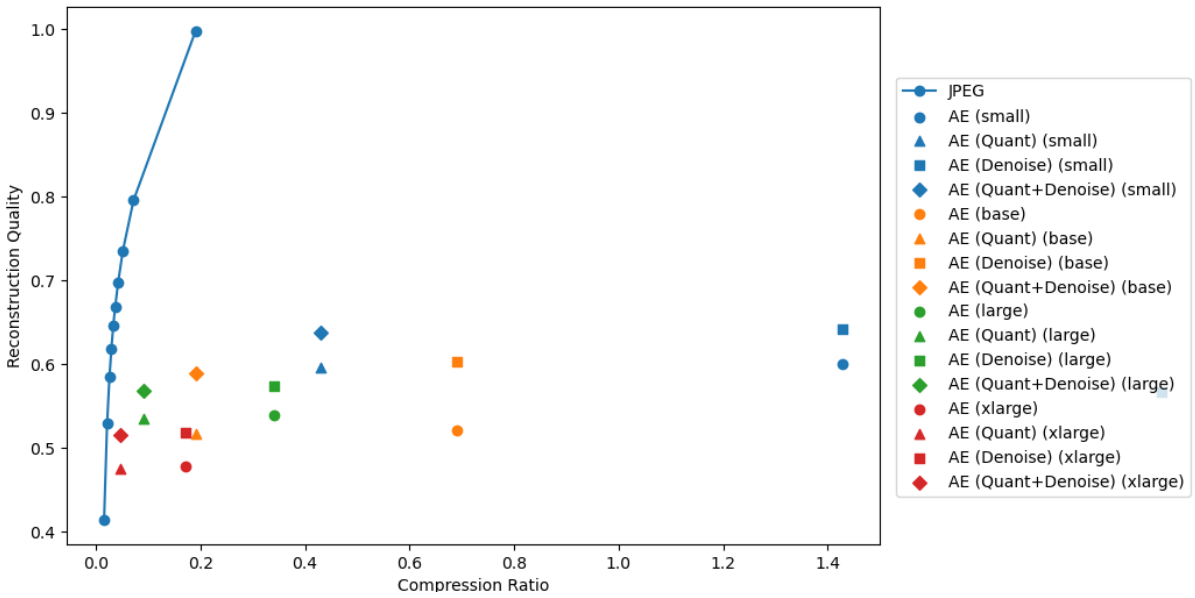


Fig. 18: Comparison between the compression ratio and reconstruction quality between proposed model and JPEG algorithms

It is evident that JPEG outperforms every autoencoder model in terms of reconstruction quality, achieving almost lossless compression ($Q > 0.99$) with a compression ratio of 0.19. For models that are not quantized and with small datasets, e.g. the small model has a compression ratio is 1.43, indicating a bigger per-image file size than storing the raw image directly. However, by applying quantization to the model, the compression ratio for the small model decreases to 0.43, with only 30% the size compared to the original model file. In terms of reconstruction quality, the SSIM decreases from 0.934 to 0.932, a 0.2% decrease that is negligible in human perception. Hence, quantization can significantly reduce the model's size with minimal impact on performance.

Even on untrained data, the denoiser model effectively increases SSIM and PSNR values. For example, on the xlarge model, the SSIM value is increased from 0.8875

to 0.9249, a 4% improvement that makes it visually on par with non-denoised large models. Hence, the general denoising model enhances the quality of reconstructed images without increasing compression rate.

The model approaches JPEG performance at large scale. It is hypothesized that with a sufficiently large model scale, this model can outperform JPEG algorithm at the lowest quality. However, even with a low quality parameter (20%) the JPEG algorithm produces an image with SSIM of 0.899, indicating that it is not significantly noticeable in human perception.

Qualitative Evaluation

The models that are the most optimal (quantized and denoised) are compared with JPEG algorithms with a quality parameter of 30%. Sixty-four samples were randomly chosen from each model, with a total of 256 images. A website (Fig. 19) is made to allow users to select the image with the most resemblance to the original image, and four users who have never seen the reconstructed images of the models are chosen to conduct this evaluation. The wins for each model and the JPEG algorithm are recorded, as shown below (Fig. 20)

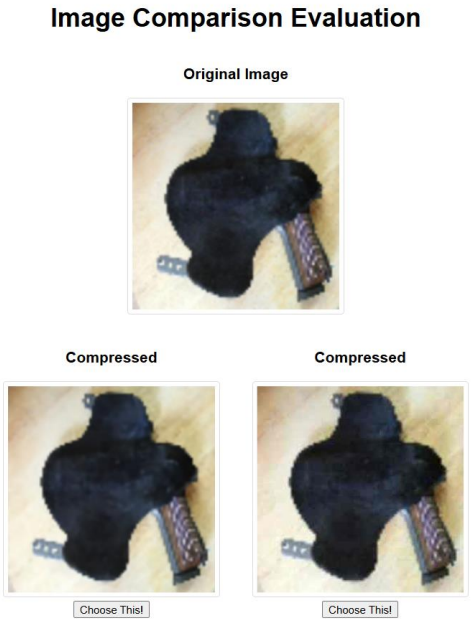


Fig. 19: Website used for algorithm voting

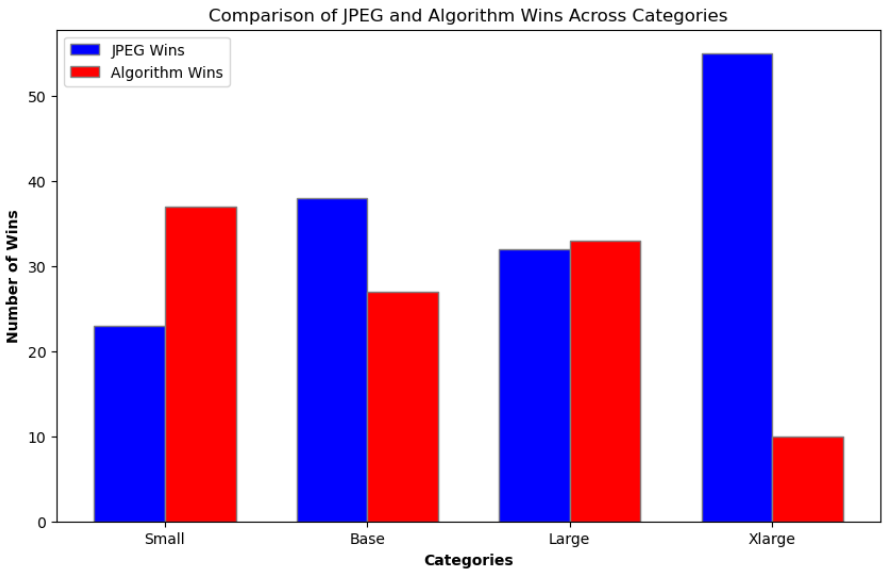


Fig. 20: Comparison of JPEG and Algorithm wins across different models

This result is similar to the qualitative analysis, where the small model outperforms JPEG at 30% quality on reconstruction quality, while xlarge model performs worse. This, again, shows the model's current incompetence to reach both comparable reconstruction quality and compression rate as the JPEG algorithm does.

Decompression latency

We define the decompression latency as the average time to decompress an image. To test this, the quantized and unquantized model are exported into ONNX (Open Neural Network Exchange) format and a Rust program is written to test the average per-image latency of decompressing using JPEG and the models into the memory. The test program is ran on a variety of client devices with different specifications. (Appendix. C)

Fig. 21a, 21b: Per-image Loading and Decompressing time for JPEG and proposed models

As shown on Fig. 21b, unlike in JPEG algorithm where minimal setup is required, decompression latency is high as all the weights are loaded before use. Quantized model loads faster due to their smaller file size and thus less demanding I/O. In terms of inference latency, smaller models have an advantage against larger models due to fewer outputs in linear layer to calculate, e.g. the small model's decompressing time is 20% faster than the base model. Quantization makes the inference slightly faster (e.g. 6016us vs 5433us for the large model) in decompressing time due to the weights in linear layer is quantized. Overall, in terms of decompressing latency, JPEG significantly outperforms the proposed model due to its simplicity to set up a working environment, and its ability to decompress single images.

Conclusion

This research experimented with the feasibility of adapting Autoencoder neural networks through a series of model testing and improvements, and is evaluated against a commonly used algorithm JPEG through quantitative and qualitative evaluation, in terms of compression rate, reconstruction quality, and decompression latency. Although the outcome model failed to achieve a compression latency that makes it applicable to client applications, through a series of improvements, the model shows promising scalability that may outperform JPEG on large scale datasets. Furthermore, as dedicated hardware like NPU become common on client devices, the latency problem may become feasible as device-side inference become

more powerful.

Although image quality metrics demonstrate JPEG having superior performance, human evaluation proved that the proposed model could perform better than JPEG due to the smoother color changes generated by the denoiser. Furthermore, candidates all report that sometimes it is indistinguishable between the compressed and uncompressed results, regardless of JPEG or proposed model. Hence, at such high reconstruction quality, PSNR and SSIM values are not fully representative of human perception. Therefore, the proposed model can be applicable for image archive purposes, where datasets are usually large and heterogeneous, and archived data are rarely read or written.

For future reference, the proposed model would benefit from structural changes, such as incorporating text embedded labelling to help the model associate objects with text vectors, and concatenating features information (e.g. embeddings) with the output image to assist the denoiser with more information to reconstruct the image with. Sub-int8 quantization, such as qint1 (1 bit per weight) can be explored, where bit operation can be used to inferring the model, significantly reducing memory cost and improving the model's compatibility with low-end devices. With the rising popularity of dedicated neural network hardware, this paper will hopefully raise attention on an alternative compression pathway for researchers, leading to more efficient data transmission that enables more people to connect with the world.

Bibliography

1. **Ibraheem, Murooj et al.** "A Comprehensive Literature Review on Image and Video Compression: Trends, Algorithms, and Techniques." *International Journal of Imaging and Robotics*, vol. 29, no. 3, 2024, pp. 863-876. doi:10.18280/isi.290307. Accessed 5 Jan 2025.

2. **Hassan, Shayan Ali, et al.** "Rethinking Image Compression on the Web with Generative AI." *arXiv preprint arXiv:2407.04542v1*, 2024. Accessed 5 Jan 2025.

3. **Micucci, Umberto.** "An Introduction to Autoencoders." *arXiv preprint arXiv:2201.03898*, 2022. Accessed 5 Jan 2025.

4. **Fournier, Quentin, and Daniel Aloise.** "Empirical Comparison Between Autoencoders and Traditional Dimensionality Reduction Methods." *arXiv preprint arXiv:2103.04874*, 2021. Accessed 5 Jan 2025.

5. **Mahajan, Abhishek.** "Linear vs. Non-Linear Dimensionality Reduction: PCA and Kernel-PCA." *Medium*, 4 Nov. 2023, https://medium.com/@abhishek8694/linear-vs-non-linear-dimensionality-reduction-pca-and-kernel-pca-10490f345ba9. Accessed 5 Jan 2025.

6. **Weißenberger, André, and Bertil Schmidt.** "Accelerating JPEG Decompression on GPUs." *arXiv preprint arXiv:2111.09219*, 17 Nov. 2021, https://arxiv.org/abs/2111.09219. Accessed 5 Jan 2025.

7. **Loeffler, John.** "What Is an NPU: The New AI Chips Explained." *TechRadar*, 15 Jan. 2024, https://www.techradar.com/computing/cpu/what-is-an-npu. Accessed 5 Jan 2025.

8. **Tiantian, Wang et al.** "An Efficient Lightweight Network for Image Denoising Using Progressive Residual and Attention Mechanism Fusion." *Scientific Reports*, vol. 14, no. 1, 2024, https://www.nature.com/articles/s41598-024-60139-x. Accessed 5 Jan 2025.

9. **Weigend, Andreas S., et al.** "Nonlinear Principal Component Analysis by Neural Networks: Theory and Application to the Lorenz System." *Journal of Climate*, vol. 13, no. 4, 2000, pp. 821-835, https://journals.ametsoc.org/view/journals/clim/13/4/15200442_2000_013_0821_npcabn_2.0.co_2.xml. Accessed 5 Jan 2025.

10. **Holdsworth, Jim, and Mark Scapicchio.** "What Is Deep Learning?" *IBM*, 17 June 2024, https://www.ibm.com/topics/deep-learning. Accessed 5 Jan 2025.

11. **Delua, Julianna.** "Supervised vs. Unsupervised Learning: What's the Difference?" *IBM*, 12 Mar. 2021, https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning. Accessed 5 Jan

2025.

12. **"What Is Gradient Descent?"** *IBM*, https://www.ibm.com/topics/gradient-descent. Accessed 5 Jan 2025.

13. **LeCun, Yann et al. "Deep learning."** *Nature*, vol. 536, no. 7617, 2016, pp. 355-359, https://www.nature.com/articles/nature14539. Accessed 5 Jan 2025.

14. **Elyasi, N., and M. Hosseini Moghadam.** "Single CNN with Filter Size of 3." *ResearchGate*, https://www.researchgate.net/figure/Single-CNN-with-filter-size-of-3_fig4_343987422. Accessed 5 Jan 2025.

15. **"Autoencoder."** *Wikipedia*, https://en.wikipedia.org/wiki/Autoencoder. Accessed 5 Jan 2025.

16. **Hinton, Geoffrey E., and Ruslan R. Salakhutdinov.** "Reducing the Dimensionality of Data with Neural Networks." *Science*, vol. 313, no. 5786, 2006, pp. 504–507. https://doi.org/10.1126/science.1127647. Accessed 5 Jan 2025.

17. **Bengio, Yoshua, et al.** "Learning Deep Architectures for AI." *Foundations and Trends in Machine Learning*, vol. 2, no. 1, 2009, pp. 1–127. https://doi.org/10.1561/2200000006. Accessed 5 Jan 2025.

18. **Vincent, Pascal, et al.** "Extracting and Composing Robust Features with Denoising Autoencoders." *Proceedings of the 25th International Conference on Machine Learning*, 2008, pp. 1096–1103. https://dl.acm.org/doi/10.1145/1390156.1390294. Accessed 5 Jan 2025.

19. **Ranzato, Marc'Aurelio, et al.** "Deep Learning for Nonlinear Time Series Forecasting." *Proceedings of the 26th International Conference on Machine Learning*, 2009, pp. 1–8. https://dl.acm.org/doi/10.1145/2689746.2689747. Accessed 5 Jan 2025.

20. **Ranzato, Marc'Aurelio, et al**. "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction." *Artificial Neural Networks and Machine Learning – ICANN 2011*, edited by Danesh Tarapore et al., Springer, 2011, pp. 652–659. https://doi.org/10.1007/978-3-642-21735-7_7. Accessed 5 Jan 2025.

21. **Robinet, Lucas.** "Autoencoders and the Denoising Feature: From Theory to Practice." *Towards Data Science*, 26 Nov. 2020, https://towardsdatascience.com/autoencoders-and-the-denoising-feature-from-theory-to-practice-db7f7ad8fc78. Accessed 5 Jan 2025.

22. **Hasan, Syed.** "AutoEncoders: Theory + PyTorch Implementation." *Medium*, 24 Feb. 2024, https://medium.com/@syed_hasan/autoencoders-theory-pytorch-implementation-a2e72f6f7cb7. Accessed 5 Jan 2025.

23. **"Structural Similarity Index Measure."** *Wikipedia*,

https://en.wikipedia.org/wiki/Structural_similarity_index_measure. Accessed 5 Jan 2025.

24. **Wang, Zhou, et al.** "Image Quality Assessment: From Error Visibility to Structural Similarity." *IEEE Transactions on Image Processing*, vol. 13, no. 4, Apr. 2004, pp. 600–612, https://doi.org/10.1109/TIP.2003.819861. Accessed 5 Jan 2025.

25. **"JPEG."** *Wikipedia*, https://zh.wikipedia.org/wiki/JPEG. Accessed 5 Jan 2025.

26. **"Rate–Distortion Theory."** *Wikipedia*, https://en.wikipedia.org/wiki/Rate–distortion_theory. Accessed 5 Jan 2025.

27. **"ImageNet."** *ImageNet*, https://image-net.org. Accessed 5 Jan 2025.

28. **He, Kaiming, et al.** "Deep Residual Learning for Image Recognition." *arXiv*, 10 Dec. 2015, https://arxiv.org/abs/1512.03385. Accessed 5 Jan 2025.

29. **"Automatic Mixed Precision."** *PyTorch Tutorials*, https://pytorch.org/tutorials/recipes/recipes/amp_recipe.html. Accessed 5 Jan 2025.

30. **Kim, JJ.** "What Is Denoising?" *NVIDIA Blog*, 9 Nov. 2022, https://blogs.nvidia.com/blog/what-is-denoising/. Accessed 5 Jan 2025.

Appendix

A) Autoencoder Model Implementation With Residual Connection

```
import torch
import torch.nn as nn

class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride, transpose=False):
        super(ResidualBlock, self).__init__()
        self.transpose = transpose
        self.stride = stride

        # Main convolutional layer
        self.conv = (
            nn.ConvTranspose2d(
                in_channels,
                out_channels,
                kernel_size=4,
                stride=stride,
                padding=1,
                output_padding=0,
            )
            if transpose
            else nn.Conv2d(
                in_channels, out_channels, kernel_size=4, stride=stride,
padding=1
            )
        )
        self.activation = nn.ReLU(inplace=True)

        # Adjust residual connection to match spatial dimensions and channels
        if transpose:
            if in_channels != out_channels or stride != 1:
                # Calculate appropriate output_padding
                # Typically, output_padding=1 when stride=2 to align
dimensions
                output_padding = stride - 1 if stride > 1 else 0
                self.residual = nn.ConvTranspose2d(
                    in_channels,
                    out_channels,
                    kernel_size=1,
```

```
                    stride=stride,
                    padding=0,
                    output_padding=output_padding,
                )
            else:
                self.residual = nn.Identity()
        else:
            if in_channels != out_channels or stride != 1:
                self.residual = nn.Conv2d(
                    in_channels, out_channels, kernel_size=1, stride=stride,
padding=0
                )
            else:
                self.residual = nn.Identity()

        self.skip_add = nn.quantized.FloatFunctional()

    def forward(self, x):
        identity = self.residual(x)  # Adjust residual if needed
        out = self.conv(x)
        out = self.activation(out)

        return self.skip_add.add(out, identity)


class Autoencoder(nn.Module):
    def __init__(
        self,
        input_channels=3,
        output_channels=3,
        image_size=128,
        num_layers=3,
        latent_dim=256,
    ):
        super(Autoencoder, self).__init__()
        self.latent_dim = latent_dim
        self.num_layers = num_layers

        # Calculate the size after downsampling
        assert (
            image_size % (2**num_layers) == 0
        ), "non integer downsampling"
        final_size = image_size // (2**num_layers)
```

```
        # Encoder with residual connections
        encoder_layers = []
        channels = 32  # Starting number of channels
        current_channels = input_channels
        for _ in range(num_layers):
            encoder_layers.append(
                ResidualBlock(
                    in_channels=current_channels,
                    out_channels=channels,
                    stride=2,
                )
            )
            current_channels = channels
            channels = channels * 2  # Double the number of channels each
layer

        encoder_layers.append(nn.Flatten())
        encoder_layers.append(
            nn.Linear(current_channels * final_size * final_size, latent_dim)
        )
        self.encoder = nn.Sequential(*encoder_layers)

        # Decoder with residual connections
        decoder_layers = []
        decoder_layers.append(
            nn.Linear(latent_dim, current_channels * final_size * final_size)
        )
        decoder_layers.append(nn.ReLU(inplace=True))
        decoder_layers.append(
            nn.Unflatten(
                dim=1, unflattened_size=(current_channels, final_size,
final_size)
            )
        )

        for _ in range(num_layers - 1):
            channels = current_channels // 2  # Halve the number of channels
each layer
            decoder_layers.append(
                ResidualBlock(
                    in_channels=current_channels,
                    out_channels=channels,
                    stride=2,
                    transpose=True,
```

```
                )
            )
            current_channels = channels

        # Final layer to get back to output channels
        decoder_layers.append(
            nn.ConvTranspose2d(
                in_channels=current_channels,
                out_channels=output_channels,
                kernel_size=4,
                stride=2,
                padding=1,
            )
        )
        decoder_layers.append(nn.Sigmoid())  # Ensures output is between 0 and
1

        self.decoder = nn.Sequential(*decoder_layers)

    def encode(self, x):

        latent = self.encoder(x)
        return latent

    def decode(self, latent):

        reconstructed = self.decoder(latent)
        return reconstructed

    def forward(self, x):
        latent = self.encode(x)
        reconstructed = self.decode(latent)
        return reconstructed, latent
```

B) Homogeneous Dataset collected

The homogeneous dataset was scrapped from derpibooru.org on 1/30/2023 with the following search criteria:

“izzy moonbow, safe, solo, score.gte:200, -webm, -animate || *izzy moonbow, suggestive, solo, score.gte:200, -webm, -animate”*

Due to the creative nature of this dataset, I am unable to provide the dataset used in this experiment. However, the scrapper script (written in Node.js) is available here.

C) Test results of the models on different client devices

For testing code, see “benchmark” folder in Appendix 4. All data are measured in microseconds.

specs	Intel Core i5-1135G7, 4c/8t, 8GB DDR4-3200	AMD Ryzen 7 5800U, 8c/16t, 16GB LPDDR4-4266	AMD Ryzen 7 5800, 8c/16t, 32GB DDR4-2133
jpeg_load	549.27	88.12	140.23
jpeg_decomp	292.12	214.36	265.12
small_load	687021	76889	33540
small_decomp	7034.11	5014.06	2762.25
base_load	946748	108430	57191
base_decomp	8171.78	6016.66	5019.21
large_load	1811075	83423	9325
large_decomp	11125.22	6911.18	2988.19
small_quant_load	155518	24447	13434
small_quant_decomp	11513.38	4944.5	3170.38
base_quant_load	44112	50047	9325
base_quant_decomp	9372.38	5433.15	2988.19
large_quant_load	58643	42135	25430
large_quant_decomp	7900.95	6331.81	3387.5

D) Complete experiment code repository

https://github.com/GrieferPig/autoencoder_compressor

Folder description:

- Benchmark: benchmark Rust program source code
- Quantized: Final quantized and unquantized model
- Stats: Statistics for PSNR and SSIM for PNG and proposed models
- Voting_server: Python server host that hosts the voting website