AUTOENCODERS

1.Introduction

Autoencoders are a type of neural network that learn to shrink data into a smaller form and then rebuild it to look almost like the original. They have two main parts: An encoder that picks out key features by making the data simpler. A decoder that puts the data back together from this simplified version. The model is trained by reducing the difference between the original and the rebuilt data, using methods like Mean Squared Error or Binary Cross-Entropy. They're used for things like cleaning up noisy data, finding errors, and pulling out useful information where having a smart way to represent data matters. This guide goes into detail about autoencoders, shows how to use them for improving image quality with a convolutional autoencoder, and includes both visual and numerical results.

---

2. Autoencoders

2.1 What is an Autoencoder?

An autoencoder is a type of neural network that learns to represent data in a more efficient way. It takes in data, compresses it into a simpler form, and then tries to rebuild it back to something close to the original. If it does this well, it means it has picked up on the most important parts of the data.

Autoencoder Structure:

The structure of an autoencoder has three main parts that work together to compress and then rebuild the data:

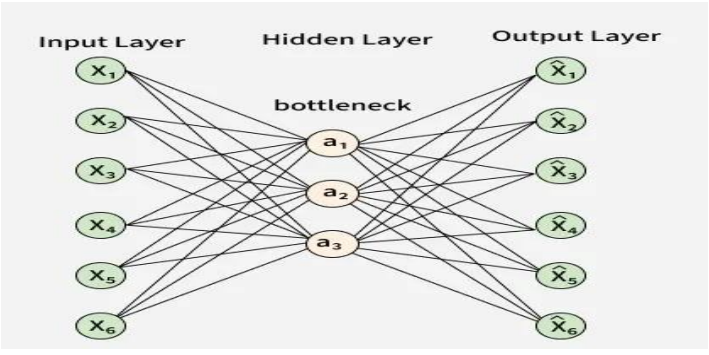

Figure 1: shows the structure of a simple autoencoder, with an input layer, a compressed middle layer, and a reconstructed output layer. The middle layer makes the model learn efficient, simpler features.

Autoencoders work in two main steps:

1. Turn the input into a compact, simplified version.
2. Use that version to create a copy of the original input.

This helps the network focus on learning useful patterns instead of just memorizing details.

2.2 Key Parts of an Autoencoder

Encoder

- Changes the input into a smaller version.
- Catches the most important details.
- Example: A 128×128 image turned into a smaller feature map.

Latent Space / Bottleneck

- The "compressed code".
- Has fewer numbers than the original input.
- Keeps only the important features.
- Shows what the autoencoder learned.

Decoder

- Takes the compressed code and builds the original input back.
- Goes from a small representation to full size.

Reconstruction

- The image the decoder makes.
- It’s trained to look like the original input (or a high-resolution image).

2.3 How Autoencoders Work

Encoding Phase

The input goes through several layers that shrink the size of the data (using pooling or strided convolutions), picking out important patterns.

Latent Representation

The middle layer, called the bottleneck, acts like a compressed memory. Since it's small, the model has to keep only the most important information.

Decoding Phase

The compressed data is expanded and cleaned up using convolutional layers to recreate the original image.

Loss Function

Reconstruction error is usually calculated with:

- MSE (Mean Squared Error): works well for pixel values that are continuous
- Binary Cross-Entropy (BCE): for normalized or binary inputs the model updates its weights to make the reconstructed image as close as possible to the original.

2.4 Why Use Autoencoders?

Autoencoders have a few key benefits:

- They can learn from data without labels.
- They help reduce the amount of data needed.
- They find useful features in the data.
- They handle complex data like images very well.

Helps with tasks like removing noise, making images sharper, finding unusual patterns, and learning useful features.

2.5 Advantages

- Works without needing labeled data.
- Can be set up in different ways: fully connected, convolutional, or recurrent.
- Extracts useful features that are easy to use for other tasks.
- Does better than PCA at handling complex, high-level data.

2.6 Disadvantages

- May overfit if the model is too big.
- The hidden features it learns aren't easy for people to understand.
- Needs careful planning when designing the structure.
- Usually produces blurry results because of how it measures errors.

2.7 Challenges

- Needs a lot of data to work well.
- Results can look blurry if the loss function or design is too simple.
- May not work well unless properly controlled.
- Training deep models takes a lot of computing power.

2.8 Types of Autoencoders:

Types of Autoencoders Here are different types of autoencoders that are built for specific tasks and have special features:

1.Denoising Autoencoder

A denoising autoencoder is trained to work with noisy or damaged data. It learns how to clean up the input and create a clear version of it. This helps the network focus on important features instead of just copying the input.

2. Sparse Autoencoder

A sparse autoencoder has more hidden units than the number of input features, but only a small number of them are active at the same time. This is done by setting some units to zero, changing how they activate, or adding a penalty to the loss function to keep things simple.

3. Variational Autoencoder

A variational autoencoder (VAE) makes guesses about how the data is spread out and tries to find a better way to represent that. It uses a method called stochastic gradient descent to improve its understanding of hidden patterns in the data. It's often used to create new data, like realistic pictures or text. It works by assuming the data comes from a certain kind of model and tries to figure out a good approximation of how the hidden parts of the data relate to the input. The encoder and decoder each have their own set of parameters that help with this process.

4. Convolutional Autoencoder

A convolutional autoencoder uses convolutional neural networks, which are good at handling images. The encoder looks for important features in the image using convolutional layers, and the decoder builds the image back up using a process called deconvolution, also known as upsampling.

Applications:

- Compressing images.
- Finding unusual patterns (using how well the model can recreate the input).
- Removing noise.
- Making images sharper.
- Adding color to black-and-white photos.
- Getting important features from data.

5. PRACTICAL IMPLEMENTATION ON AUTOENCODERS FOR IMAGE SUPER-RESOLUTION

This part shows how to build a convolutional autoencoder for improving image resolution. The aim is to teach the model how to turn low-resolution images into high-resolution ones. It uses a simple encoder-decoder setup and is trained directly in TensorFlow/Keras.

5.1 Preparing the Dataset

The dataset has 3,762 pairs of images. Each pair includes:

- A low-resolution (LR) image.
- A high-resolution (HR) image of the same scene.

These are listed in a CSV file called image_data.csv, which has two columns: low_res and high_res. Each filename is automatically turned into a full path by adding the right folder:

- low res/ for low-res images.
- high res/ for high-res images.

A special class called PairedImageSequence is used with Keras to make sure:

- Images are loaded in batches quickly.
- LR and HR images stay matched.
- The order changes each time the data is used.
- The number of batches per round stays the same.

All images are resized to 256 × 384 × 3 and adjusted to be between 0 and 1.

Figure 2 – Dataset Preview (CSV Head)

	low_res	high_res
0	1_2.jpg	1.jpg
1	2_2.jpg	2.jpg
2	3_2.jpg	3.jpg
3	4_2.jpg	4.jpg
4	5_2.jpg	5.jpg

Figure 2: Sample entries from image_data.csv, showing the filenames for low-resolution images (low_res) and their corresponding high-resolution targets (high_res). Each row represents one LR–HR image pair used during training.
A table with two columns: low_res and high_res. Entries include filenames such as 1_2.jpg, 1.jpg, 2_2.jpg, 2.jpg, indicating paired low- and high-resolution images.

5.2 Autoencoder Architecture

The convolutional autoencoder has a straightforward and efficient design.

Encoder

- Conv2D(16) → MaxPooling
- Conv2D(32) → MaxPooling
- Conv2D(64) → MaxPooling

Decoder

- Conv2D(64) → UpSampling
- Conv2D(32) → UpSampling
- Conv2D(16) → UpSampling
- Conv2D(3, sigmoid)

This setup has 84,035 trainable parameters, which gives a good mix of performance and efficiency.

Figure 3 – Model Architecture Summary

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 256, 384, 3)	0
conv2d_21 (Conv2D)	(None, 256, 384, 16)	448
max_pooling2d_9 (MaxPooling2D)	(None, 128, 192, 16)	0
conv2d_22 (Conv2D)	(None, 128, 192, 32)	4,640
max_pooling2d_10 (MaxPooling2D)	(None, 64, 96, 32)	0
conv2d_23 (Conv2D)	(None, 64, 96, 64)	18,496
max_pooling2d_11 (MaxPooling2D)	(None, 32, 48, 64)	0
conv2d_24 (Conv2D)	(None, 32, 48, 64)	36,928
up_sampling2d_9 (UpSampling2D)	(None, 64, 96, 64)	0
conv2d_25 (Conv2D)	(None, 64, 96, 32)	18,464
up_sampling2d_10 (UpSampling2D)	(None, 128, 192, 32)	0
conv2d_26 (Conv2D)	(None, 128, 192, 16)	4,624
up_sampling2d_11 (UpSampling2D)	(None, 256, 384, 16)	0
conv2d_27 (Conv2D)	(None, 256, 384, 3)	435

Total params: 84,035 (328.26 KB)
Trainable params: 84,035 (328.26 KB)
Non-trainable params: 0 (0.00 B)

Figure 3: Shows the full autoencoder setup, including the encoder and decoder layers, what the output shapes are, and how many parameters each part has. There's also a table that lists the Conv2D, MaxPooling2D, and UpSampling2D layers along with their shapes and parameter counts.

5.3 Training Setup The training uses:

- **Loss:** Mean Squared Error (MSE)
- **Optimizer:** Adam
- **Batch size:** 8

• **Number of epochs:** 10

• **Steps per epoch:** 50

• **Validation steps:** 10 Callbacks include:

• **ModelCheckpoint** (saves the best model)

• **EarlyStopping** (stops training if it starts to overfit)

• **ReduceLROnPlateau** (lowers the learning rate if progress stops) These help keep training steady and make sure the model learns well.

5.4 Training Results

The training and validation loss both drop steadily and smoothly, which shows:

• The model is learning well.

• It works well on new data.

• It's not overfitting.

The lowest validation loss was about 0.00586 around epoch 7 or 8.

Figure 4 – Training vs Validation Loss Curve

Figure 4: Training and validation MSE loss curves showing steady convergence. A two-line graph shows the training and validation loss for each epoch.

5.5 Reconstruction Results:

A key part of evaluating super-resolution models is looking at the results visually. The comparison grid shows:

- Strong color recovery.
- Good structural accuracy.
- Typical texture smoothing from using MSE.

The three-row layout includes:

1. Low-resolution input.
2. High-resolution target.
3. Output from the autoencoder.

Figure 5 – Reconstruction Grid (Low-Res → High-Res → Predicted) Caption:

Figure 5: Low-resolution inputs on top, high-resolution targets in the middle, and predicted results at the bottom. Alt-text: Three-row grid comparing low-res inputs, high-res targets, and autoencoder predictions.

---

6. Evaluation and Discussion:

The model has some good parts and some areas that could be better.

Strengths

- **Good at turning low-res images into high-res:** It does a good job of keeping the overall shape and colors right.
- **Stable training:** The model learns smoothly, and what it learns during training matches what it does on test data.
- **Small model size:** Only 84k parameters means it runs quickly and doesn't need much power.
- **Looks better visually:** The results are much sharper than the original low-res images. Limitations.
- **Loses fine details:** Using MSE makes some textures look softer, especially in detailed areas.
- **Loses some information:** Without skip connections, some small details get lost and can't be fixed later.
- **Not as realistic as others:** Other models like SRGAN produce sharper and more realistic textures.

7. Conclusion

The convolutional autoencoder learned to turn low-resolution images into higher-quality ones, with better structure and color. The loss kept going down, and the reconstructed images looked better, showing the model picked up the key patterns for improving resolution. Some small details weren't as sharp as they could be, which makes sense because the model didn't have skip connections and used MSE loss, which tends to produce smoother results.

Even with these limits, the method gives a solid way to understand how neural networks can improve image quality. It's a good starting point for learning about super-resolution before moving on to more complex models like U-Net, residual networks, or GANs, which are better at keeping small details and making images look sharper. Overall, the results show that autoencoders work well for basic image improvements and also suggest where things can be improved in the future.

Github link:

Reference:

1. Li, Pengzhi, Yan Pei, and Jianqiang Li. "A comprehensive survey on design and application of autoencoder in deep learning." *Applied Soft Computing* 138 (2023): 110176.

2. J. Zheng and L. Peng, "An Autoencoder-Based Image Reconstruction for Electrical Capacitance Tomography," in *IEEE Sensors Journal*, vol. 18, no. 13, pp. 5464-5474, 1 July1, 2018, doi: 10.1109/JSEN.2018.2836337.

3. Y. Yang, Q. M. J. Wu and Y. Wang, "Autoencoder With Invertible Functions for Dimension Reduction and Image Reconstruction," in *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 48, no. 7, pp. 1065-1079, July 2018, doi: 10.1109/TSMC.2016.2637279.