Autoencoders: Teaching Machines to Reconstruct What Matters

Student Number: 23111128
GitHub Link: https://github.com/Vinod3VK/Autoencoders.git

Introduction

When I first came across autoencoders, I was fascinated by how a neural network could learn to compress and reconstruct data—almost like giving it memory skills. In this tutorial, I explore how autoencoders work, how to build one from scratch using Keras, and what makes them so useful.

Autoencoders are a type of artificial neural network used for unsupervised learning. They take in input data, compress it into a smaller representation, and then try to reconstruct the original input from that compressed form. This concept forms the foundation for several cool applications like noise removal, image compression, and even anomaly detection.

Theoretical Background

Autoencoders are trained to minimize the difference between the input and the output. This difference is typically measured using a loss function such as mean squared error (MSE) or binary crossentropy. The network learns through backpropagation, updating the weights in both encoder and decoder to reduce reconstruction error.

Let x be the input, z the encoded representation, and x' the reconstruction. The encoder maps x to z = f(x), and the decoder maps z back to x' = g(z). The goal is to minimize L(x, x') where L is a loss function like MSE or binary crossentropy depending on the data distribution.

Mathematical Foundations of Autoencoders

To understand how autoencoders learn, let's formalize their structure mathematically. Let the input data be represented by a vector $x \in \mathbb{R}^n$. The encoder is a function f_θ that maps x to a latent vector $z \in \mathbb{R}^m$, where typically $m < n$. This encoding function can be expressed as:

$$z = f_\theta(x) = \sigma(Wx + b)$$

where W is the weight matrix, b is the bias, and σ is a non-linear activation function like ReLU or sigmoid.

The decoder is a function g_ϕ that attempts to reconstruct the input from z:

$$\hat{x} = g_\phi(z) = \sigma'(W'z + b')$$

Here, W' and b' are the weights and biases of the decoder, and σ' is the output activation function—commonly sigmoid for normalized image data.

The learning objective is to minimize the reconstruction loss $L(x, \hat{x})$. For binary or normalized input (like MNIST), binary crossentropy is a good choice:

$$L(x, \hat{x}) = -\sum [x \log(\hat{x}) + (1 - x) \log(1 - \hat{x})]$$

Alternatively, for continuous-valued input, mean squared error (MSE) can be used:

$$L(x, \hat{x}) = ||x - \hat{x}||^2 = \sum (x_i - \hat{x}_i)^2$$

During training, gradients of the loss with respect to all model parameters are computed using backpropagation, and optimization algorithms such as Adam or SGD are used to iteratively update the weights to minimize the loss.

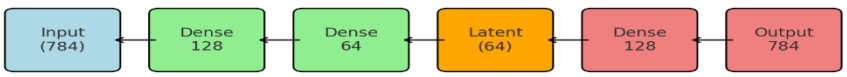
How Autoencoders Work

Autoencoders are made up of two main parts:

- **Encoder**: This part compresses the input into a smaller dimension, which is called the latent space.
- **Decoder**: This part reconstructs the input from the compressed version.

Together, these two parts create a network that learns to reproduce its input. But what's really interesting is that, because of the bottleneck (the smallest layer in the network), the model is forced to learn only the most important features of the data.

Example Architecture:



We use **ReLU** in the hidden layers and **Sigmoid** at the output so that pixel values stay between 0 and 1.

Variants I Explored

There are many interesting extensions of the basic autoencoder. Here are a few that stood out to me:

- **Denoising Autoencoder**: Add noise to the input images, then train the model to reconstruct the clean versions.
- **Sparse Autoencoder**: Introduce constraints so only a few neurons are active, encouraging feature selectivity.
- **Variational Autoencoder (VAE)**: Instead of learning exact values in the latent space, the model learns distributions. These are widely used in generative models.

In this project, I focused on the basic and denoising versions, but reading about the others gave me ideas for future exploration.

Dataset Used

I worked with the **MNIST dataset**, which contains 70,000 grayscale images of handwritten digits (0 to 9). Each image is 28×28 pixels. I used:

- 60,000 images for training
- 10,000 images for testing

Before feeding them into the model, I normalized the pixel values to a 0–1 range and flattened each image into a 784-dimensional vector.

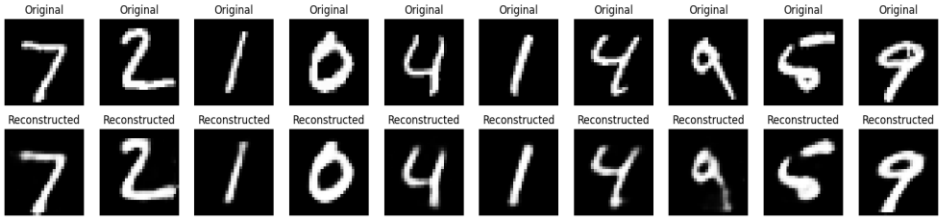
Implementation and Results

I implemented the autoencoder in **Keras** with the TensorFlow backend. Here's a quick summary of the model setup:

- **Encoder**: Dense(128) → Dense(64)
- **Decoder**: Dense(128) → Dense(784)
- **Activation functions**: ReLU in hidden layers, Sigmoid at the output
- **Loss function**: Binary crossentropy

- **Optimizer**: Adam
- **Epochs**: 20
- **Batch size**: 256

Figure 1: Original vs Reconstructed Digits

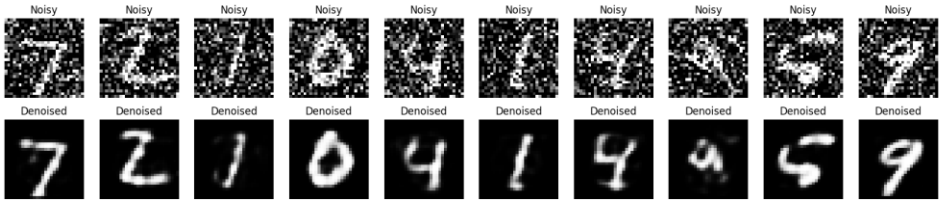


The reconstructions looked quite accurate. Even with such a compact latent space, the model was able to reproduce the digits well.

Denoising Autoencoder

Then I added Gaussian noise to the test images and trained the same model to clean them up.

Figure 2: Noisy Input vs Denoised Output



It was really cool to see the model clean up the noisy digits. Even though I didn't use complex architectures, the model handled noise pretty well!

Why Autoencoders Work

Autoencoders work because data often has structure and patterns. For example, most handwritten 8s look similar. So even if we compress them, we can still reconstruct the shape. The model learns to extract the **most essential features**, which is kind of like how we humans focus on outlines or edges rather than remembering every pixel.

Where Autoencoders Are Used

Here are some real-world use cases that I learned about:

- **Anomaly Detection**: If an autoencoder is trained on "normal" data, it will do badly on "weird" data—like fraud. That spike in reconstruction error becomes a red flag.
- **Noise Reduction**: In images or audio, they can remove unwanted noise.
- **Compression**: Learn efficient representations for storage or transmission.
- **Feature Learning**: Use the encoder as a feature extractor for downstream tasks.

Things You Can Try

Here are some things I played with and some I hope to try next:

1. **Change the latent space size**: Try 32, 16, or even 2 neurons. The reconstructions get blurrier, but it's fun to test limits.
2. **Visualize the 2D latent space**: With 2 neurons, you can plot the encodings and see clustering of digits.
3. **Try different datasets**: Fashion-MNIST gives you images of clothes—it's a good test for generalization.
4. **Build deeper models**: Add more layers or use convolutional layers for better image handling.
5. **Anomaly detection**: Train on digits 0–8, test on digit 9. Watch how the reconstruction fails.

References

1. Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*, MIT Press (2016) https://www.deeplearningbook.org/
2. Keras Documentation – Autoencoder Example (MNIST) https://keras.io/examples/mnist_autoencoder/
3. François Chollet, *Deep Learning with Python*, Manning Publications https://www.manning.com/books/deep-learning-with-python
4. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow by Aurélien Géron https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/

5. Towards Data Science – Understanding Autoencoders
   https://towardsdatascience.com/understanding-autoencoders-2f5d2dbadee1
6. TensorFlow Tutorials – Autoencoders
   https://www.tensorflow.org/tutorials/generative/autoencoder
7. Scikit-learn Documentation
   https://scikit-learn.org/stable/modules/neural_networks_unsupervised.html