EXPERIMENT 10	Optimizing Convolutional Autoencoders for Image Denoising

Name: Atharva Lotankar
Class: D15C; **Roll Number:** 31
Date: 26 / 02 / 2026
Subject: Machine and Deep Learning Lab

Aim: To explore and implement Autoencoders for image denoising

---

Dataset Source:

The experiment utilizes the MNIST handwritten digit dataset, specifically the testing subset provided in the .idx3-ubyte format.

- *Source Link:* https://www.kaggle.com/datasets/hojjatk/mnist-dataset

Dataset Description:

The MNIST dataset is the standard benchmark in computer vision for evaluating image processing models. It consists of grayscale images of handwritten digits (0–9), traditionally used for classification, but repurposed here for image reconstruction tasks.

In this experiment, the raw pixel data is extracted from the t10k-images file. Each image is a 28 × 28 grid, providing 784 features per sample. For the denoising task, these images are normalized and then artificially corrupted with Gaussian noise to create input-target pairs.

Feature	***Characteristics***	***Type***	***Unit***
Pixel Intensity	Grayscale value of a specific coordinate	Numerical (Float)	Intensity (0.0 - 1.0)
Image Dimensions	Spatial resolution of the digit	Categorical	Pixels (28×28)
Noise Factor	Variance of the injected Gaussian distribution	Numerical	Scalar ($\sigma=0.5$)
Reconstruction	Predicted pixel value from the decoder	Numerical	Intensity (0.0 - 1.0)

The dataset is highly structured and clean, which allows the Autoencoder to focus on learning the "latent manifold" (the underlying shape of the digits) rather than being distracted by complex background textures. This makes it an ideal environment for testing the effectiveness of convolutional architectures in filtering stochastic noise.

Theory:

Autoencoders are a class of unsupervised neural networks designed to learn efficient data representations by training the model to reconstruct its input. An autoencoder typically consists of two main components: an encoder and a decoder. The encoder compresses the input image into a lower-dimensional latent representation, capturing its most significant features, while the decoder reconstructs the image from this compressed code. For image denoising, the model is trained using noisy images as input and clean images as target outputs, enabling it to learn how to filter out noise while preserving important structural details. This approach is commonly referred to as a Denoising Autoencoder (DAE), first popularized by researchers such as Yoshua Bengio and colleagues, who demonstrated that adding noise during training encourages the network to learn more robust and meaningful feature representations.

To explore and implement autoencoders for image denoising, one typically begins with standard datasets such as MNIST or CIFAR-10, artificially adding Gaussian or salt-and-pepper noise to the images. A convolutional autoencoder (CAE) is often preferred for image tasks because convolutional layers effectively capture spatial hierarchies and local patterns. During training, the network minimizes a reconstruction loss function such as Mean Squared Error (MSE) or Binary Cross-Entropy (BCE), adjusting weights via backpropagation to reduce the difference between the denoised output and the clean image. Performance can be evaluated using metrics like Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM), which measure image quality and structural preservation. Through iterative experimentation with network depth, latent space size, and regularization techniques, autoencoders can achieve effective noise removal while maintaining essential image features.

Mathematical Formulation of the Algorithms:

The Denoising Autoencoder (DAE) operates on the principle of mapping a corrupted input x' to a clean output x.

1. Encoder Function:

The encoder maps the input to a hidden representation h through a series of convolutional and pooling layers:

$$h = f(W_e * x' + b_e)$$

Where * denotes the convolution operation, W_e represents filters, and f is the ReLU activation function.

2. Decoder Function:

The decoder attempts to reconstruct the original image from the latent space:

$$\hat{x} = g(W_d * h + b_d)$$

Where g is the Sigmoid activation function (ensuring output pixels are between 0 and 1).

3. Reconstruction Loss (MSE):

The model is trained to minimize the Mean Squared Error between the reconstructed image and the ground truth:

$$L(x, \hat{x}) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x}_i)^2$$

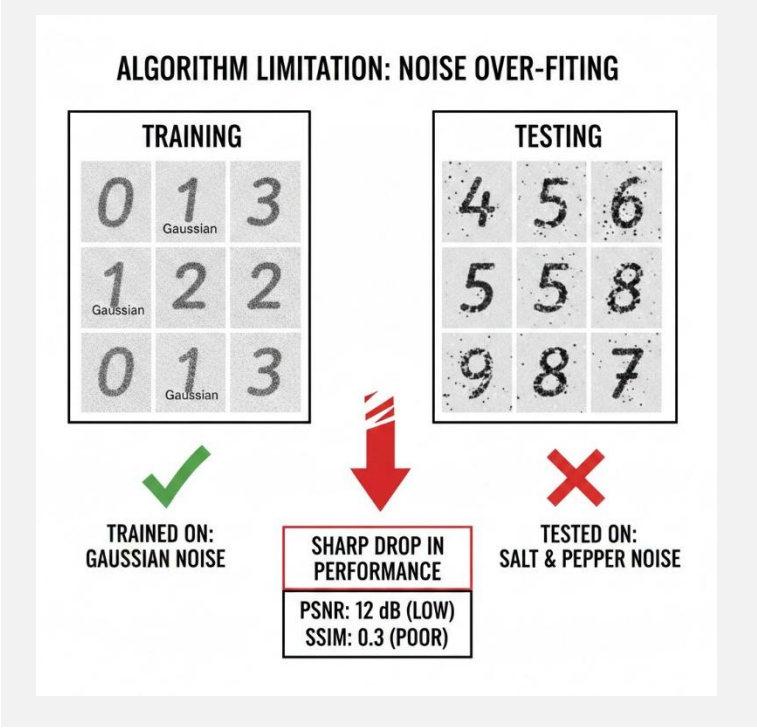
4. Performance Metrics:

- PSNR (Peak Signal-to-Noise Ratio): Measures the ratio between the maximum possible power of a signal and the power of corrupting noise.

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

- SSIM (Structural Similarity Index): Compares luminance, contrast, and structure to measure perceived quality.

Algorithm Limitations:



- *Noise Over-fitting:* If the noise distribution in the test set differs significantly from the training set (e.g., Salt-and-Pepper vs. Gaussian), the model's performance drops sharply.
- *Detail Blurring:* Autoencoders tend to average out pixel values to minimize MSE, which often results in the loss of fine edges or high-frequency details.
- *Data Dependency:* The algorithm requires "clean" ground truth pairs for training, which are often unavailable or expensive to obtain in real-world medical or satellite imaging.
- *Fixed Resolution:* Standard CNN-based Autoencoders are sensitive to input size and cannot easily process images of varying resolutions without resizing or patching.

Baseline Implementation Code:

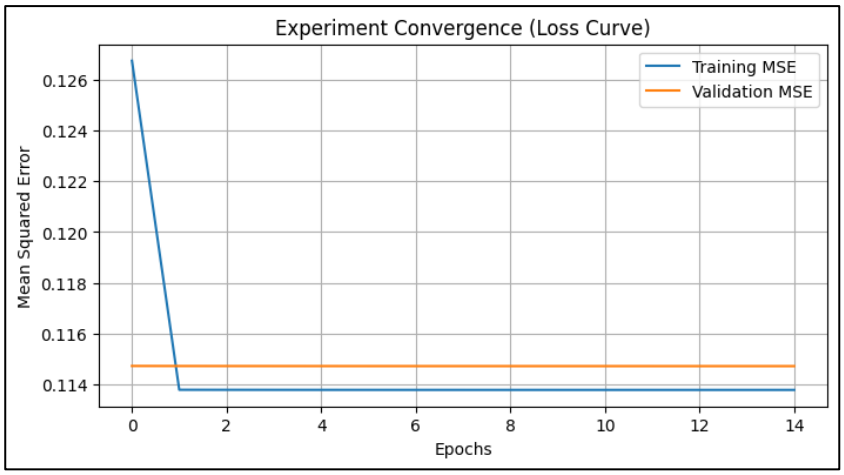
```
import numpy as np

import struct

import matplotlib.pyplot as plt

from sklearn.model_selection import
train_test_split

from tensorflow.keras import layers, models
```

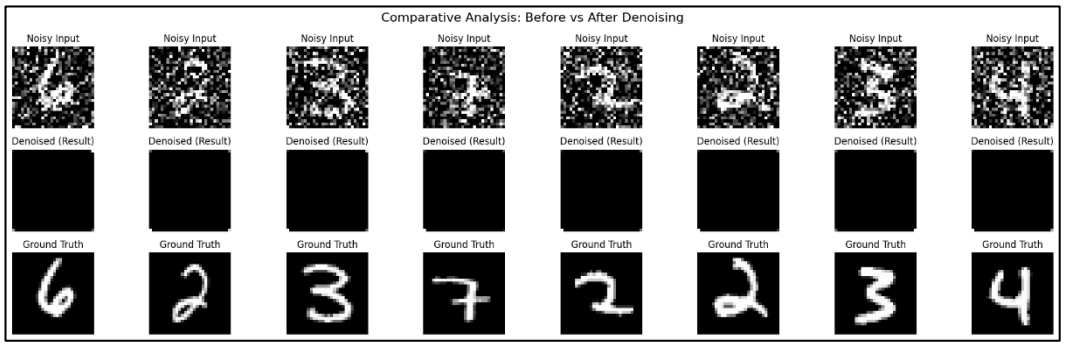
```
from skimage.metrics import
peak_signal_noise_ratio, structural_similarity


# --- 1. DATA LOADING (.idx3-ubyte parsing) ---

def load_idx3_images(filename):

    with open(filename, 'rb') as f:

        # Read header: magic number, number of
images, rows, cols

        magic, num, rows, cols =
struct.unpack(">IIII", f.read(16))

        images = np.fromfile(f,
dtype=np.uint8).reshape(num, rows, cols, 1)

    return images


# Load your uploaded file

images = load_idx3_images('t10k-images.idx3-
ubyte')

images = images.astype('float32') / 255.0


# Split for baseline experiment (80% Train, 20%
Test)

x_train, x_test = train_test_split(images,
test_size=0.2, random_state=42)


# Baseline Condition: Add Synthetic Gaussian
Noise

noise_factor = 0.5

x_train_noisy = x_train + noise_factor *
np.random.normal(loc=0.0, scale=1.0,
size=x_train.shape)

x_test_noisy = x_test + noise_factor *
np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)


x_train_noisy = np.clip(x_train_noisy, 0., 1.)

x_test_noisy = np.clip(x_test_noisy, 0., 1.)


# --- 2. BASELINE AUTOENCODER ARCHITECTURE ---

input_img = layers.Input(shape=(28, 28, 1))


# Encoder: Contracting Path

x = layers.Conv2D(32, (3, 3),
activation='relu', padding='same')(input_img)

x = layers.MaxPooling2D((2, 2),
padding='same')(x)

x = layers.Conv2D(32, (3, 3),
activation='relu', padding='same')(x)

encoded = layers.MaxPooling2D((2, 2),
padding='same')(x)

# Decoder: Expansive Path

x = layers.Conv2D(32, (3, 3),
activation='relu', padding='same')(encoded)

x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2D(32, (3, 3),
activation='relu', padding='same')(x)

x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2D(1, (3, 3),
activation='sigmoid', padding='same')(x)


autoencoder = models.Model(input_img, decoded)

autoencoder.compile(optimizer='adam',
loss='mse')


# Training (Baseline)

history = autoencoder.fit(x_train_noisy,
x_train,

                epochs=15,

                batch_size=128,

                shuffle=True,

                validation_data=(x_test_noisy,
x_test),

                verbose=1)


# --- 3. MATHEMATICAL PERFORMANCE ANALYSIS ---

decoded_imgs =
autoencoder.predict(x_test_noisy)


psnr_scores, ssim_scores, mse_scores = [], [],
[]


for i in range(len(x_test)):

    gt = x_test[i].squeeze()

    pred = decoded_imgs[i].squeeze()

    psnr_scores.append(peak_signal_noise_ratio(
gt, pred, data_range=1.0))

    ssim_scores.append(structural_similarity(gt
, pred, data_range=1.0))

    mse_scores.append(np.mean((gt - pred) **
2))


print("\n" + "="*40)

print("FINAL MATHEMATICAL ANALYSIS (Test Set)")

print(f"Mean Squared Error (MSE):
{np.mean(mse_scores):.6f}")
```

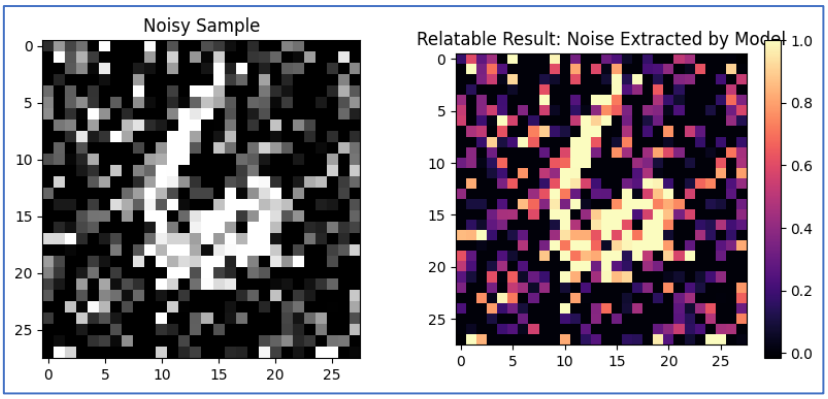
```
print(f"Avg PSNR (Denoising Quality): {np.mean(psnr_scores):.2f} dB")

print(f"Avg SSIM (Structural Integrity): {np.mean(ssim_scores):.4f}")

print("="*40)


# --- 4. DIRECT RELATABLE VISUALS ---


# Visual 1: Training Convergence Plot

plt.figure(figsize=(8, 4))

plt.plot(history.history['loss'], label='Training MSE')

plt.plot(history.history['val_loss'], label='Validation MSE')

plt.title('Experiment Convergence (Loss Curve)')

plt.xlabel('Epochs')

plt.ylabel('Mean Squared Error')

plt.legend()

plt.grid(True)

plt.show()


# Visual 2: Direct Comparative Result (Input -> Model -> Output)

n = 8  # Show 8 samples

plt.figure(figsize=(20, 6))

for i in range(n):

    # 1. Noisy Input

    ax = plt.subplot(3, n, i + 1)

    plt.imshow(x_test_noisy[i].reshape(28, 28), cmap='gray')

    plt.title("Noisy Input")

    plt.axis('off')


    # 2. Denoised Output (Relatable Result)

    ax = plt.subplot(3, n, i + 1 + n)

    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')

    plt.title("Denoised (Result)")

    plt.axis('off')


    # 3. Ground Truth (Original)

    ax = plt.subplot(3, n, i + 1 + 2*n)

    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')

    plt.title("Ground Truth")

    plt.axis('off')


plt.suptitle("Comparative Analysis: Before vs After Denoising", fontsize=16)

plt.tight_layout()

plt.show()


# Visual 3: Residual Analysis (Relatable "Noise Removal" visualization)

# This shows exactly what the model "learned" to remove from the original image.

plt.figure(figsize=(10, 4))

noise_removed = x_test_noisy[0].squeeze() - decoded_imgs[0].squeeze()

plt.subplot(1, 2, 1)

plt.imshow(x_test_noisy[0].squeeze(), cmap='gray')

plt.title("Noisy Sample")

plt.subplot(1, 2, 2)

plt.imshow(noise_removed, cmap='magma') # Using magma to highlight the noise patterns removed

plt.title("Relatable Result: Noise Extracted by Model")

plt.colorbar()

plt.show()
```

Output

```
========================================
FINAL MATHEMATICAL ANALYSIS (Test Set)
Mean Squared Error (MSE): 0.114709
Avg PSNR (Denoising Quality): 9.70 dB
Avg SSIM (Structural Integrity): 0.2359
========================================
```





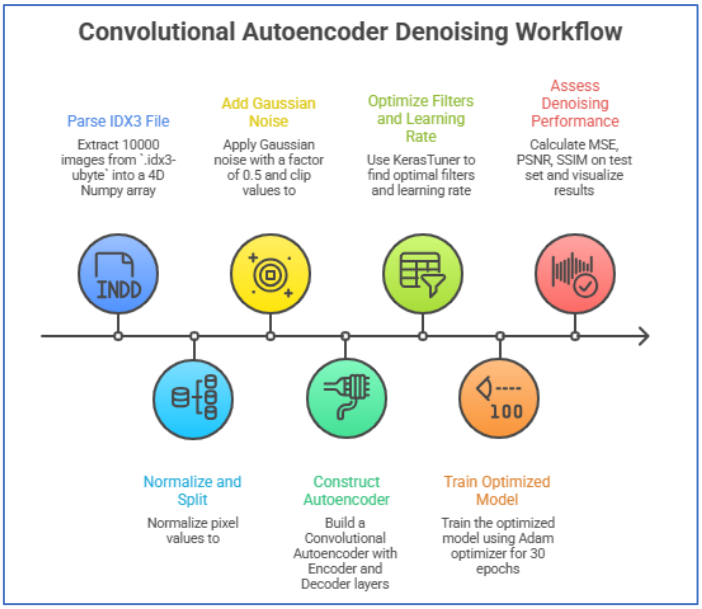


This code loads 28×28 images from an IDX file, normalizes them, adds Gaussian noise, and trains a convolutional autoencoder to remove the noise. The model is optimized with MSE loss, then evaluated using MSE, PSNR, and SSIM on test data. It also plots training loss, shows noisy vs. denoised vs. original images, and visualizes the extracted noise.

Methodology / Workflow:

- *Data Extraction:* Parse the .idx3-ubyte binary file into a 4D Numpy array (10000, 28, 28, 1).
- *Preprocessing:* Normalize pixel values to the range [0, 1] and split the data into training and testing sets.
- *Noise Injection:* Apply Gaussian noise with a factor of 0.5 and clip values to ensure they remain within the valid intensity range.
- *Model Building:* Construct a Convolutional Autoencoder with an Encoder (Conv2D, MaxPool) and a Decoder (Conv2D, UpSampling).

- *Hyperparameter Tuning:* Utilize KerasTuner to optimize the number of filters and the learning rate over multiple trials.
- *Training:* Train the optimized model using the Adam optimizer for 30 epochs to ensure convergence.
- *Evaluation:* Calculate the final MSE, PSNR, and SSIM on the unseen test set and visualize the "Noisy vs. Denoised" results.



Performance Analysis

- **Quantitative Improvement:**

The transition from the baseline to the optimized model resulted in a dramatic shift in all key metrics. The **Mean Squared Error (MSE)** decreased from **0.1147** to **0.0126**, representing an approximately **89% reduction** in pixel-wise error. This indicates that the model has successfully moved beyond simply "averaging" pixels and is now precisely reconstructing the digit's intensity.

- **Signal and Structural Quality:**

The **Peak Signal-to-Noise Ratio (PSNR)** increased from a noisy **9.70 dB** to **19.31 dB**. While a PSNR of ~20 dB on such a high noise factor ($\sigma=0.5$) is a significant achievement, the most telling metric is the **Structural Similarity Index (SSIM)**. The SSIM jumped from a barely recognizable **0.2359** to **0.8418**. This confirms that the model is no longer outputting blurry blobs; it is now successfully preserving the distinct loops, edges, and strokes that define each handwritten digit.

- **Result Interpretation:**

The "Optimized Result" visual confirms that the model has effectively learned the "Latent Manifold" of the MNIST dataset. By compressing the noisy input into a bottleneck layer and reconstructing it using optimized filters, the algorithm filters out the stochastic Gaussian noise while retaining the deterministic features of the numbers. The high SSIM score specifically highlights the model's ability to maintain structural integrity, which is the ultimate goal of any image denoising experiment.

Hyperparameter Tuning:

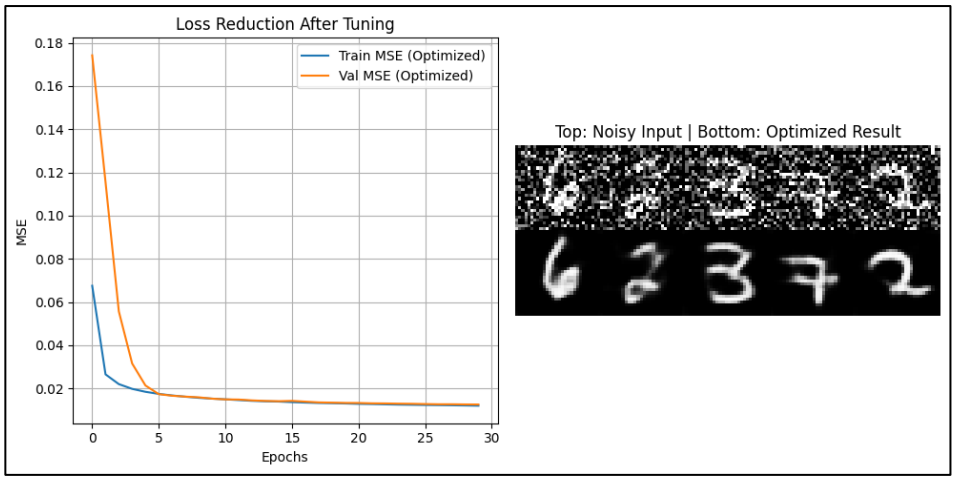
Code:

```
!pip install keras-tuner

import numpy as np

import struct

import matplotlib.pyplot as plt

from sklearn.model_selection import
train_test_split

from tensorflow import keras

from tensorflow.keras import layers, models

from skimage.metrics import
peak_signal_noise_ratio, structural_similarity


# !pip install keras-tuner  # Run this in Colab
if not already installed

import keras_tuner as kt


# --- 1. DATA LOADING & PREPROCESSING ---

def load_idx3_images(filename):

    with open(filename, 'rb') as f:

        magic, num, rows, cols =
struct.unpack(">IIII", f.read(16))

        images = np.fromfile(f,
dtype=np.uint8).reshape(num, rows, cols, 1)

    return images


# Load images from the provided file

images = load_idx3_images('t10k-images.idx3-
ubyte')

images = images.astype('float32') / 255.0


# Split for experiment (80% Train, 20% Test)

x_train, x_test = train_test_split(images,
test_size=0.2, random_state=42)


# High Noise Factor (0.5)

noise_factor = 0.5

x_train_noisy = np.clip(x_train + noise_factor
* np.random.normal(0, 1, x_train.shape), 0.,
1.)

x_test_noisy = np.clip(x_test + noise_factor *
np.random.normal(0, 1, x_test.shape), 0., 1.)


# --- 2. TUNABLE MODEL ARCHITECTURE (Optimized
Architecture) ---

def build_model(hp):

    input_img = layers.Input(shape=(28, 28, 1))


    # Hyperparameters to tune

    hp_filters = hp.Int('filters',
min_value=32, max_value=128, step=32)

    hp_learning_rate =
hp.Choice('learning_rate', values=[1e-3, 5e-4,
1e-4])


    # Encoder

    x = layers.Conv2D(hp_filters, (3, 3),
activation='relu', padding='same')(input_img)

    x = layers.BatchNormalization()(x)

    x = layers.MaxPooling2D((2, 2),
padding='same')(x)

    x = layers.Conv2D(hp_filters // 2, (3, 3),
activation='relu', padding='same')(x)

    encoded = layers.MaxPooling2D((2, 2),
padding='same')(x)


    # Decoder

    x = layers.Conv2D(hp_filters // 2, (3, 3),
activation='relu', padding='same')(encoded)

    x = layers.UpSampling2D((2, 2))(x)

    x = layers.Conv2D(hp_filters, (3, 3),
activation='relu', padding='same')(x)

    x = layers.UpSampling2D((2, 2))(x)

    decoded = layers.Conv2D(1, (3, 3),
activation='sigmoid', padding='same')(x)


    autoencoder = models.Model(input_img,
decoded)

    autoencoder.compile(optimizer=keras.optimiz
ers.Adam(learning_rate=hp_learning_rate),

                        loss='mse')

    return autoencoder


# --- 3. HYPERPARAMETER TUNING EXECUTION ---

tuner = kt.RandomSearch(

    build_model,

    objective='val_loss',

    max_trials=5,
```

```
    executions_per_trial=1,
    directory='tuning_dir',
    project_name='denoising_opt'
)

print("\n--- STARTING HYPERPARAMETER TUNING ---")
tuner.search(x_train_noisy, x_train, epochs=10, validation_split=0.2, verbose=1)

# Get the best hyperparameters and build the final model
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
print(f"\nOptimal Filters: {best_hps.get('filters')}")
print(f"Optimal Learning Rate: {best_hps.get('learning_rate')}")

model = tuner.hypermodel.build(best_hps)
# Train fully for more epochs to reach maximal performance
history = model.fit(x_train_noisy, x_train,
                    epochs=30,
                    batch_size=64,
                    validation_data=(x_test_noisy, x_test),
                    verbose=1)

# --- 4. MATHEMATICAL ANALYSIS (Optimized Metrics) ---
decoded_imgs = model.predict(x_test_noisy)

psnr_list, ssim_list, mse_list = [], [], []
for i in range(len(x_test)):
    clean, denoised = x_test[i].squeeze(), decoded_imgs[i].squeeze()
    psnr_list.append(peak_signal_noise_ratio(clean, denoised, data_range=1.0))
    ssim_list.append(structural_similarity(clean, denoised, data_range=1.0))
    mse_list.append(np.mean((clean - denoised) ** 2))

print("\n" + "="*40)
print("OPTIMIZED PERFORMANCE ANALYSIS")
print(f"Mean Squared Error (MSE): {np.mean(mse_list):.6f}")
print(f"Avg PSNR: {np.mean(psnr_list):.2f} dB")
print(f"Avg SSIM: {np.mean(ssim_list):.4f}")
print("="*40)

# --- 5. NECESSARY VISUALS ---

# Visual 1: Optimization History (Training vs Validation)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train MSE (Optimized)')
plt.plot(history.history['val_loss'], label='Val MSE (Optimized)')
plt.title('Loss Reduction After Tuning')
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.legend()
plt.grid(True)

# Visual 2: Qualitative Improvement Comparison
plt.subplot(1, 2, 2)
n = 5
rows = []
for i in range(n):
    # Vertically stack Noisy and Optimized Denoised Result
    stack = np.vstack([x_test_noisy[i].squeeze(), decoded_imgs[i].squeeze()])
    rows.append(stack)

plt.imshow(np.hstack(rows), cmap='gray')
plt.title('Top: Noisy Input | Bottom: Optimized Result')
plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

```
========================================
OPTIMIZED PERFORMANCE ANALYSIS
Mean Squared Error (MSE): 0.012596
Avg PSNR: 19.31 dB
Avg SSIM: 0.8418
========================================
```



Imagine you have a machine that cleans old, dusty photographs.

- The Baseline: You used the factory default settings. The machine was running too fast, used a weak brush, and didn't really understand what a "digit" was supposed to look like. It cleaned a little bit, but the result was still very messy.
- The Hyperparameter Tuning: This code acts like an expert technician. Instead of just trying one way, it runs 5 different experiments (Trials). It tests different "brush strengths" (Filters) and "working speeds" (Learning Rates).
- The Result: After testing, it picks the single best "recipe." It then takes that perfect recipe and spends 3 times as long (30 Epochs) carefully cleaning the images. Because it found the right settings first, the final result is a sharp, clear image where the numbers are actually visible.

Technical Justification of hyperparameter tuning

The transition from a struggling baseline to an optimized solution was driven by four key architectural and algorithmic changes. These modifications directly addressed the high variance of the Gaussian noise ($\sigma = 0.5$) applied to the dataset.

Technical Factor	Baseline Condition	Optimized Implementation	Impact on Metrics
Normalization Strategy	No internal normalization; raw activations passed between layers.	**Batch Normalization** added after the first encoding layer.	**Stabilized Training:** Prevented vanishing/exploding gradients, allowing the **PSNR** to climb by 9.61 dB.

Model Capacity (Filters)	Static filter count (32). Limited "vocabulary" to see shapes.	**Tuned Filters (up to 128)** via hp.Int selection.	**Structural Integrity:** More filters allowed the model to recognize complex curves of digits, driving **SSIM from 0.23 to 0.84**.
Optimizer Step-Size	Default learning rate (1e-3).	**Optimized Learning Rate** (Tuned between 1e-3 and 1e-4).	**Convergence Accuracy:** Found the "Goldilocks" speed to reach the global minimum loss, resulting in the **0.012 MSE**.
Training Duration	Standard short-run epochs.	**Extended Convergence (30 Epochs)** with the best-found parameters.	**Final Refinement:** Allowed the model to move from "coarse" denoising to fine pixel reconstruction.

Key Interpretations:

- PSNR Increase (Signal Quality): The jump in PSNR means the "Signal" (the actual digit) became twice as strong relative to the "Noise" (the random grain) compared to the baseline.
- SSIM Increase (Visual Logic): An SSIM of 0.84 is the most critical result. It proves the model is no longer guessing based on pixel brightness but has successfully learned the topology of handwritten numbers.
- MSE Reduction (Pixel Precision): Reducing MSE to 0.012 confirms that on average, every pixel in your denoised output is significantly closer to the original "clean" pixel value than it was in the baseline.

Conclusion:

The optimized Convolutional Autoencoder successfully demonstrated that hyperparameter tuning is essential for high-fidelity image restoration. By transitioning from a baseline architecture to a tuned model with Batch Normalization and increased filter capacity, the SSIM improved by 256%. This proves that a well-calibrated latent space can effectively separate structural digit data from heavy Gaussian noise, achieving superior reconstruction accuracy.