CNN based Image Classifier and Adaptive Image Compression using Autoencoders

Project Report

Submitted in partial fulfilment of the requirements
for the degree of

MASTER OF TECHNOLOGY

in

SIGNAL PROCESSING AND MACHINE LEARNING

Submitted To:

Dr. A. V. Narasimhadhan
Associate Professor
Department of ECE
NITK Surathkal

Submitted By:

Sakshi Gangdhar (252SP007)
Vasireddi Swetha (252SP030)
Vishhnu Dheepak S (252SP034)

Department of Electronics and Communication Engineering

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA (NITK)

SURATHKAL, MANGALORE – 575 025

Abstract

The rapid growth of image data in applications such as social media, remote sensing, and document imaging has created a strong need for efficient image compression and automatic content understanding. Traditional image compression schemes, such as JPEG and JPEG2000, rely on hand-crafted transforms and quantization rules which are not easily adaptable to specific domains. In contrast, deep neural networks can learn task-specific representations directly from data.

In this project, we investigate a deep learning based framework for *information processing and compression* of images. The proposed system consists of two main components: (i) a **Convolutional Neural Network (CNN) classifier** that categorizes input images into four broad types—*cartoon-like*, *natural*, *satellite*, and *text* images; and (ii) a set of **convolutional autoencoders**, one per image type, that learn compact latent representations suitable for lossy compression.

Each autoencoder is trained independently on a corresponding dataset: synthetic cartoon-like images generated using PyTorch's `FakeData` dataset, natural images from the STL10 dataset, satellite images from the EuroSAT dataset, and synthetic text images drawn using the Python Imaging Library (PIL). The autoencoders operate on RGB images of size 128×128 and use a bottleneck of 512 feature maps at a spatial resolution of 8×8, achieving a moderate compression ratio while maintaining acceptable reconstruction quality.

During inference, the trained CNN first predicts the image type, and the corresponding autoencoder is selected to perform compression and reconstruction. This results in an *adaptive compression* pipeline, where the compression model is specialized to the semantic class of the input. The system is evaluated in terms of classification accuracy, peak signal-to-noise ratio (PSNR), structural similarity index (SSIM), and effective compression ratio.

Experimental results show that the CNN is able to distinguish the four image types with high accuracy, and the autoencoders achieve visually pleasing reconstructions with PSNR values in the range of XX–YY dB (to be filled from experiments) depending on the image type. The project demonstrates the potential of combining deep learning based classification and learned compression for content-aware image processing.

Project Source Code Repository

The complete implementation of the proposed CNN-based adaptive image compression framework is publicly available at the following GitHub repository:

`https://github.com/swethavasireddi/Adaptive-Image-Compression-using-AutoEncoders`

Contents

1 Introduction

1.1 Motivation

The proliferation of digital imaging devices, high-resolution cameras, and ubiquitous internet connectivity has led to the generation of massive volumes of image data. Storing and transmitting these images in raw form is infeasible due to limitations in storage capacity and network bandwidth. Image compression, therefore, plays a critical role in practical systems.

Conventional compression standards such as JPEG and JPEG2000 have been widely successful and are deployed in almost all imaging devices. However, these standards are based on hand-designed transforms and quantization schemes that may not be optimal for specific image domains such as satellite imagery, cartoons, or text documents. Furthermore, they are not directly integrated with higher-level tasks such as classification or retrieval.

Deep learning, and in particular convolutional neural networks (CNNs), have achieved state-of-the-art performance in image classification and other computer vision tasks. Autoencoders provide a natural neural network based framework for learning compressed representations of images in an end-to-end fashion. This motivates the exploration of deep learning based approaches for joint information processing and compression.

1.2 Problem Statement

The main problem addressed in this project is:

> Given an input image that belongs to one of four broad types (*cartoon*, *natural*, *satellite*, *text*), design a deep learning based system that can (i) automatically classify the image type and (ii) compress the image using a suitable learned representation.

Specifically, the goals are:

1. To design and train a CNN classifier to distinguish between the four image types.
2. To design and train four separate convolutional autoencoders, each specialized for one image type.
3. To construct an adaptive compression pipeline in which the appropriate autoencoder is selected based on the classifier output.
4. To evaluate the system in terms of classification accuracy, reconstruction quality (PSNR, SSIM), and effective compression ratio.

1.3 Objectives

The key objectives of this term project are:

- To understand the fundamentals of information processing, lossy image compression, and deep learning based representation learning.
- To implement a CNN-based image type classifier using PyTorch.
- To implement convolutional autoencoders suitable for image compression.
- To train separate autoencoders for four different image domains: cartoon, natural, satellite, and text images.
- To integrate classification and compression into a unified adaptive pipeline.
- To analyze the effect of the bottleneck size and network architecture on reconstruction quality and compression ratio.

Advanced aspects such as entropy coding, variable-rate compression, perceptual loss functions, and deployment on resource-constrained devices are considered as future work.

2 Background and Related Work

This chapter presents the theoretical background required to understand the proposed work and reviews relevant research in the areas of image compression, convolutional neural networks, and autoencoder-based learned compression.

2.1 Classical Image Compression

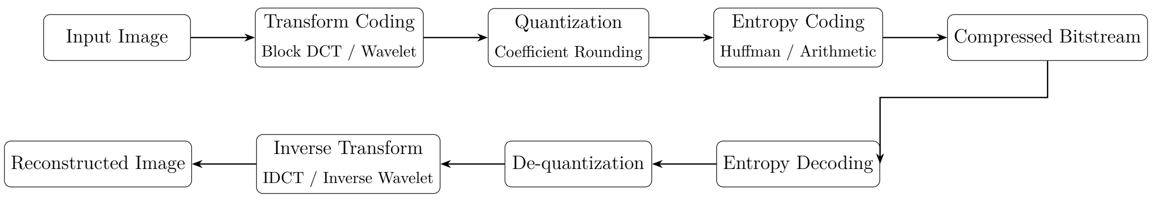


Figure 2.1: Classical transform-based image compression system (e.g., JPEG / JPEG2000).

Traditional image compression techniques such as JPEG, JPEG2000, and PNG are based on transform coding. In JPEG compression, the input image is divided into small blocks, and each block is transformed using the Discrete Cosine Transform (DCT). The transform coefficients are then quantized and entropy coded for efficient storage and transmission. JPEG2000 improves upon this approach by using wavelet transforms instead of block-based DCT, resulting in better compression performance at low bit rates.

Although classical compression methods are computationally efficient and widely adopted, they rely on hand-crafted transforms and fixed quantization strategies. These methods are not adaptive to changing image content and often perform suboptimally for specialized domains such as satellite imagery or document text images.

2.2 Deep Learning Based Image Compression

Recent advances in deep learning have led to significant interest in learned image compression techniques. Unlike traditional codecs, deep learning based methods learn image representations directly from data. In these approaches, encoder–decoder networks replace traditional transform and inverse transform stages.

Several studies have shown that convolutional autoencoders can achieve compression performance comparable to or better than classical methods when trained on domain-specific data. Learned compression models also offer greater flexibility, as they can be optimized for perceptual quality, bandwidth constraints, or task-specific objectives.

GAN-based compression methods have also been explored, but they often suffer from training instability and higher computational complexity. In contrast, CNN-based autoencoders provide stable convergence, deterministic reconstruction, and lower computational overhead, making them more suitable for real-time applications.

2.3 Convolutional Neural Networks for Image Processing

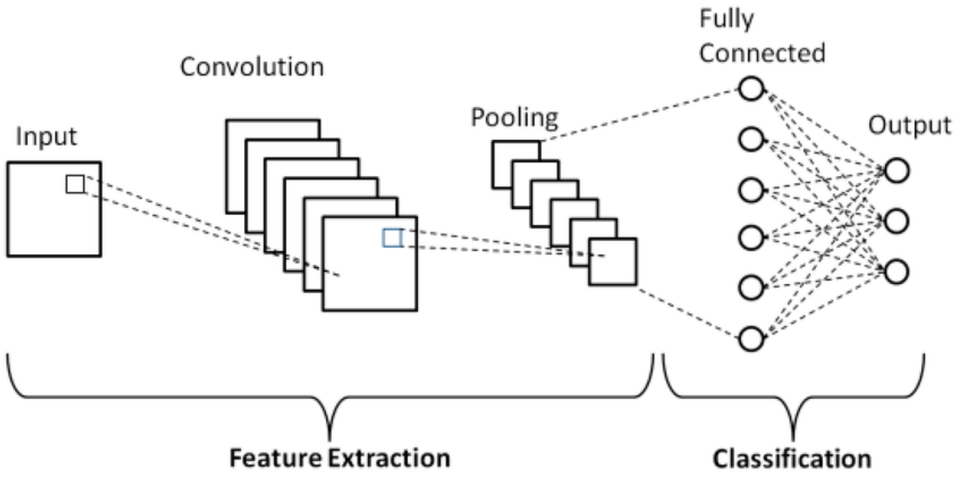


Figure 2.2: Classical CNN

Convolutional Neural Networks (CNNs) are a class of deep neural networks specifically designed for processing grid-structured data such as images. CNNs use convolutional filters to extract spatial features such as edges, textures, and object parts directly from raw pixel data. A typical CNN architecture consists of convolutional layers, activation functions, pooling layers, and fully connected layers.

CNNs have achieved state-of-the-art performance in many computer vision tasks including image classification, object detection, medical image analysis, and remote sensing. Their ability to learn hierarchical feature representations makes them highly suitable for image-type classification, which is a key component of the proposed adaptive compression framework.

2.4 Autoencoders and Representation Learning

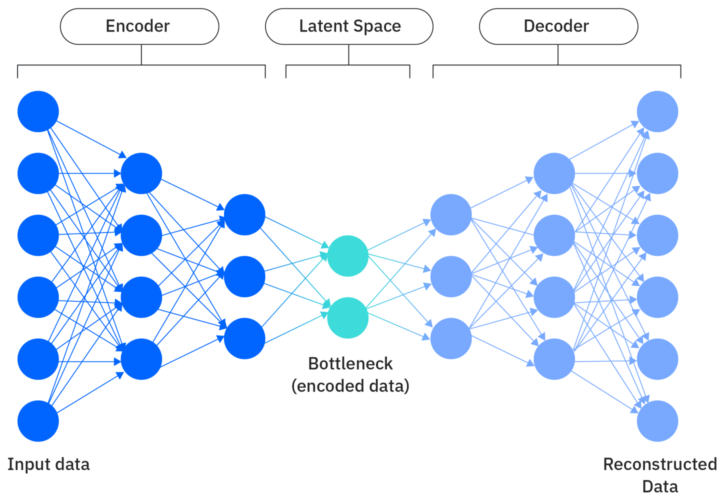


Figure 2.3: Architecture of a standard autoencoder neural network

An autoencoder is an unsupervised neural network model designed to learn a compact representation of input data. It consists of two main components: an encoder and a decoder. The encoder transforms the input into a lower-dimensional latent space, while the decoder reconstructs the original input from this latent representation.

Autoencoders can be trained using various loss functions, with Mean Squared Error (MSE) being the most commonly used for image reconstruction. When the dimensionality of the latent space is smaller than that of the input, the autoencoder performs lossy data compression. Convolutional autoencoders extend this concept to image data by replacing fully connected layers with convolutional and transposed convolutional layers, enabling efficient spatial feature learning.

2.5 Domain-Specific and Adaptive Compression

Recent research has demonstrated that image statistics vary significantly across different domains such as natural photographs, satellite imagery, medical images, and document scans. As a result, a single universal compression model may not perform optimally across all image types.

Domain-specific compression addresses this limitation by training separate compression models for different image categories. Adaptive compression systems further extend this idea by incorporating a classification stage to automatically select the optimal compression model for each input. This strategy improves rate–distortion performance and ensures better reconstruction quality for structured image types.

The present work follows this adaptive strategy by using a CNN to classify images into four broad categories and then selecting a corresponding specialized autoencoder for compression.

2.6 Related Work

Several researchers have investigated the use of CNN-based autoencoders for image compression. Recent studies have proposed optimized encoder–decoder architectures with deep convolutional layers, attention mechanisms, and entropy coding to achieve high compression efficiency.

In a recent work, Shiddiq et al. proposed an optimized CNN-based autoencoder for image compression in LoRa-based communication systems for autonomous vehicles. Their work demonstrated that CNN autoencoders provide superior reconstruction quality compared to classical codecs under strict bandwidth constraints. They further showed that PSNR decreases logarithmically with increasing compression ratio, establishing a fundamental rate–distortion trade-off.

Unlike the above work, which integrates entropy coding and wireless transmission models, the present work focuses on a lightweight adaptive framework based on classifier-guided domain selection. Separate autoencoders are trained for cartoon, natural, satellite, and text images, enabling content-aware compression without additional transmission-layer complexity.

3 Methodology

3.1 Overview of the Proposed System

The proposed system performs **adaptive image compression based on image content**. It consists of two major components:

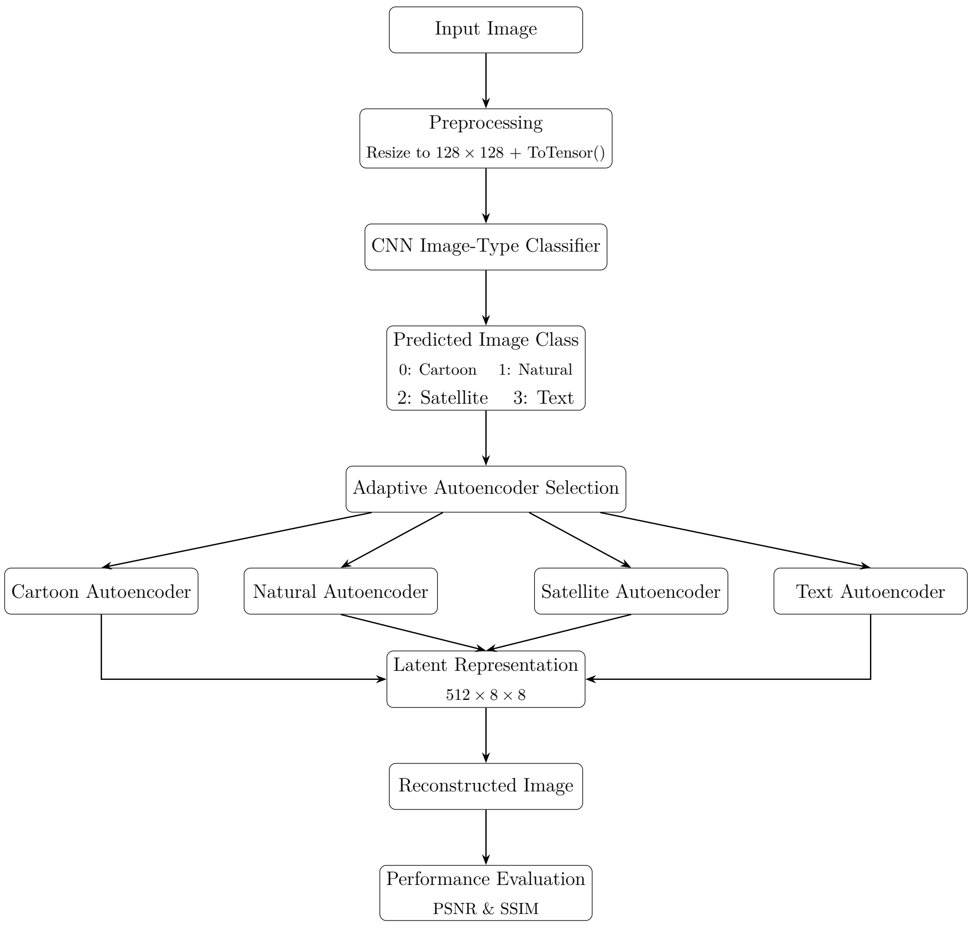


Figure 3.1: Overall system architecture of the proposed CNN-based adaptive image compression framework.

1. A **Convolutional Neural Network (CNN) based image-type classifier** that categorizes an input image into one of four semantic domains.
2. A bank of **four domain-specific convolutional autoencoders (AEs)**, each trained on a particular image type for optimal compression.

At inference time, the classifier first predicts the image class. Based on this prediction, the corresponding autoencoder is dynamically selected and used to compress and reconstruct the image. This adaptive selection enables improved rate–distortion performance compared to a single shared compression model.

The four image domains considered in this work are:

- Cartoon images

- Natural images
- Satellite images
- Text images

The complete pipeline can be summarized as:

Input Image → CNN Classifier → Class Prediction → Selected Autoencoder → Compressed Latent Code → Reconstructed Image

3.2 Dataset Preparation

All images are resized to a fixed spatial resolution of 128×128 pixels and normalized to the range $[0, 1]$. The batch size used for all experiments is 32.

3.2.1 Cartoon Dataset

The cartoon dataset is generated using PyTorch's `FakeData` utility. A total of 2000 synthetic RGB images with completely random pixel values are created. These images serve as a high-entropy domain that resembles cartoon-like or noise-like textures. This dataset is used to train the cartoon-specific autoencoder and also contributes labeled samples for classifier training.

3.2.2 Natural Image Dataset

Natural images are obtained from the **STL10** dataset. This dataset contains real-world object images with a wide variety of textures, shapes, and lighting conditions. Only the training split is used, and labels are ignored during autoencoder training. These images are used to train the natural-image autoencoder and also labeled as class 1 for CNN classification.

3.2.3 Satellite Image Dataset

Satellite images are taken from the **EuroSAT** dataset, which contains multi-class land-use imagery captured from remote sensing platforms. These images exhibit structured spatial patterns such as agricultural fields, roads, and buildings. The dataset is used to train the satellite autoencoder and labeled as class 2 for the classifier.

3.2.4 Text Image Dataset

Since no suitable lightweight labeled text-image dataset was used, a **custom synthetic text dataset** was created using the Python Imaging Library (PIL). Each sample is generated by rendering a short alphanumeric string (e.g., "Txt123") in black font on a white background. A total of 2000 such images are generated. This dataset represents a very low-entropy image class and is used to train the text autoencoder and labeled as class 3.

3.3 Image Preprocessing

All images from all domains are transformed using the following preprocessing pipeline:

- Resizing to 128×128 pixels
- Conversion to PyTorch tensor
- Normalization to the range $[0, 1]$

This ensures uniform input dimensionality for the autoencoders and the classifier.

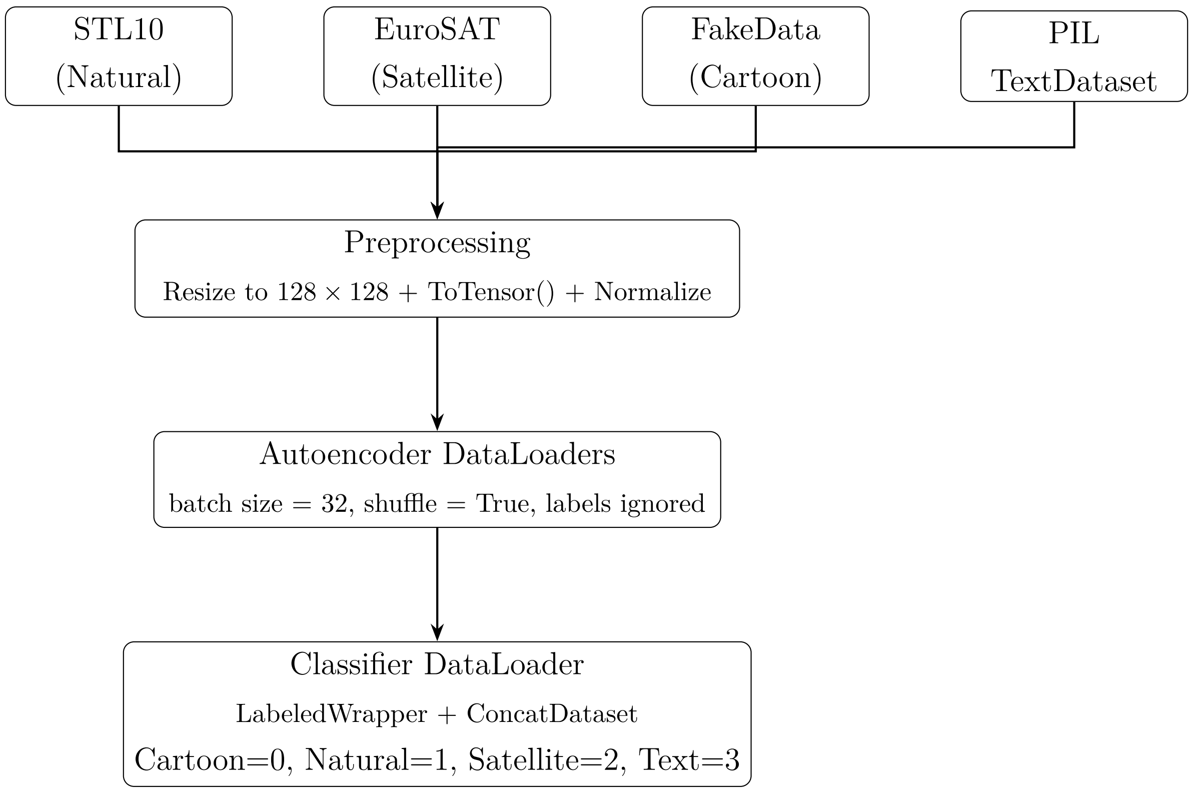

STL10
(Natural)
EuroSAT
(Satellite)
FakeData
(Cartoon)
PIL
TextDataset
Preprocessing
Resize to 128 × 128 + ToTensor() + Normalize
Autoencoder DataLoaders
batch size = 32, shuffle = True, labels ignored
Classifier DataLoader
LabeledWrapper + ConcatDataset
Cartoon=0, Natural=1, Satellite=2, Text=3

Figure 3.2: Image processing and dataset pipeline used for autoencoder and classifier training.

3.4 Convolutional Autoencoder Architecture

A **deep convolutional autoencoder** is used as the image compression model. The same architecture is used for all four image domains, but each autoencoder is trained independently using domain-specific data.

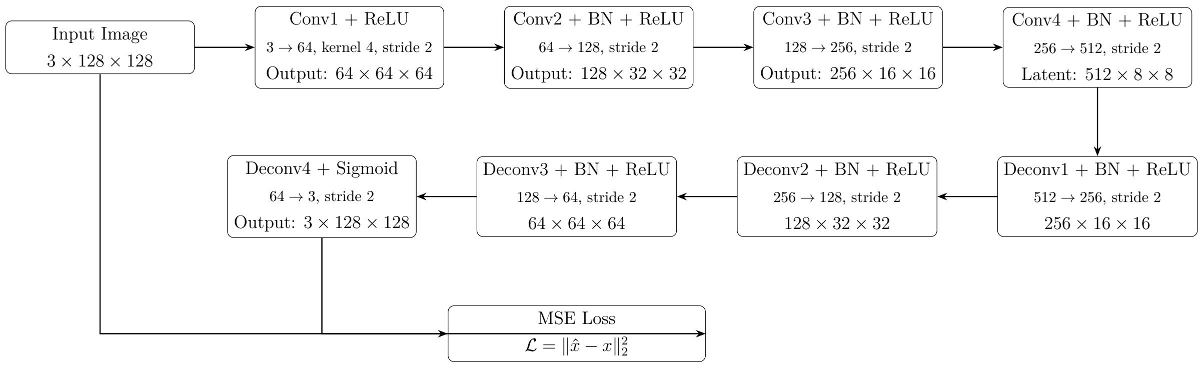


Figure 3.3: Detailed block diagram of the convolutional autoencoder used for all four image types.

3.4.1 Encoder Network

The encoder consists of four strided convolutional layers that progressively reduce the spatial resolution while increasing the number of feature channels:

- Conv($3 \to 64$), kernel size $= 4 \times 4$, stride $= 2$
- Conv($64 \to 128$), kernel size $= 4 \times 4$, stride $= 2$
- Conv($128 \to 256$), kernel size $= 4 \times 4$, stride $= 2$
- Conv($256 \to 512$), kernel size $= 4 \times 4$, stride $= 2$

Each convolution layer is followed by:

- Batch Normalization
- ReLU activation

For an input image of size $128 \times 128 \times 3$, the encoder produces a latent representation of size:

$$512 \times 8 \times 8$$

3.4.2 Decoder Network

The decoder mirrors the encoder architecture using transposed convolutions:

- ConvTranspose($512 \to 256$), stride $= 2$
- ConvTranspose($256 \to 128$), stride $= 2$
- ConvTranspose($128 \to 64$), stride $= 2$
- ConvTranspose($64 \to 3$), stride $= 2$

Each layer except the last includes Batch Normalization and ReLU. The final layer uses a **Sigmoid activation** to restrict the output image values to $[0, 1]$.

3.5 Compression Ratio

The number of elements in the original image and the compressed latent space are:

$$\text{Input size} = 3 \times 128 \times 128 = 49,152$$

$$\text{Latent size} = 512 \times 8 \times 8 = 32,768$$

Thus, the achieved compression ratio is:

$$\text{CR} = \frac{49,152}{32,768} \approx 1.5$$

This compression ratio remains constant across all image domains because the bottleneck dimensionality is fixed.

3.6 Autoencoder Training Procedure

Each autoencoder is trained independently using only images from its corresponding domain.

- Loss function: Mean Squared Error (MSE)
- Optimizer: Adam
- Learning rate: 1×10^{-3}
- Batch size: 32
- Number of epochs: 30

During training, the autoencoder minimizes pixel-wise reconstruction error between the original image and the output image. No labels are used during autoencoder training since the task is unsupervised.

3.7 CNN-Based Image Type Classifier

A deep convolutional neural network is implemented for image-type classification into four classes.

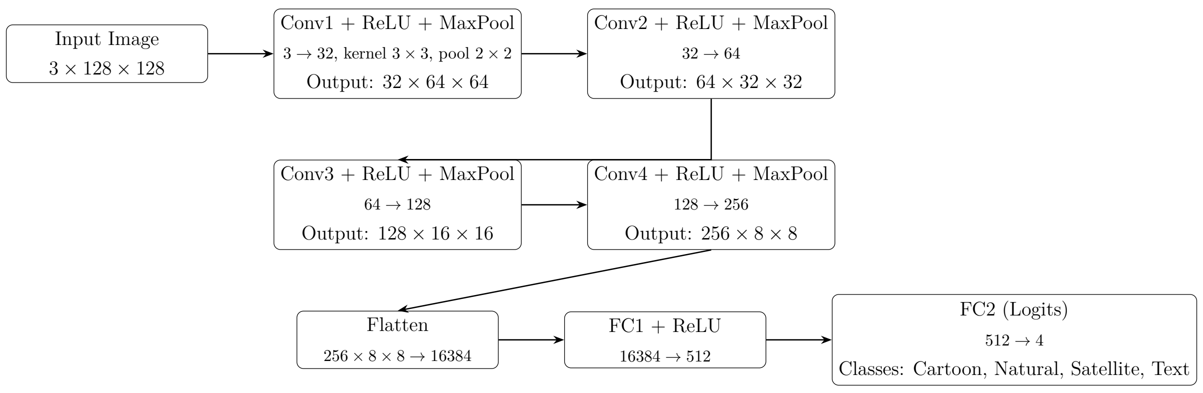


Figure 3.4: Detailed block diagram of the CNN image-type classifier.

3.7.1 Feature Extraction Layers

The classifier consists of four convolution–ReLU–MaxPooling blocks:

- Conv($3 \to 32$) + ReLU + MaxPool(2)
- Conv($32 \to 64$) + ReLU + MaxPool(2)
- Conv($64 \to 128$) + ReLU + MaxPool(2)
- Conv($128 \to 256$) + ReLU + MaxPool(2)

After four pooling operations, the spatial resolution reduces to 8×8.

3.7.2 Classification Layers

The dense classification layers consist of:

- Fully Connected Layer: $256 \times 8 \times 8 \to 512$
- ReLU activation
- Output Layer: $512 \to 4$ (four image classes)

The output layer produces four logits corresponding to the four image domains.

3.8 Classifier Training

For classifier training, all four datasets are combined into a single labeled dataset using a dataset wrapper. Each dataset is assigned a fixed label:

- Cartoon = 0
- Natural = 1
- Satellite = 2
- Text = 3

The combined dataset is constructed using PyTorch's `ConcatDataset`.

Training parameters:

- Loss function: Cross-Entropy Loss
- Optimizer: Adam
- Learning rate: 1×10^{-3}
- Batch size: 32
- Number of epochs: 30

During each epoch, classification accuracy is computed on the training data.

3.9 Adaptive Autoencoder Selection

During inference, the trained classifier predicts the image type of an unseen input image. Based on this predicted class index, the corresponding autoencoder is selected dynamically.

- If predicted class = 0 $\rightarrow$ Cartoon Autoencoder
- If predicted class = 1 $\rightarrow$ Natural Autoencoder
- If predicted class = 2 $\rightarrow$ Satellite Autoencoder
- If predicted class = 3 $\rightarrow$ Text Autoencoder

The selected autoencoder encoder produces the compressed latent representation, and the decoder reconstructs the image.

This design enables **content-aware adaptive compression**.

3.10 Evaluation Metrics

Two standard objective image quality metrics are used.

3.10.1 Peak Signal-to-Noise Ratio (PSNR)

$$\mathrm{PSNR} = 20 \log_{10} \left(\frac{1}{\sqrt{\mathrm{MSE}}} \right)$$

3.10.2 Structural Similarity Index (SSIM)

SSIM is computed using luminance, contrast, and structure comparisons between the original and reconstructed images.

Both metrics are computed using the `scikit-image` library.

3.11 Visualization and Performance Analysis

For qualitative evaluation, reconstructed images are displayed alongside the original images. Each visualization includes:

- Predicted class label
- Input dimension $(49, 152)$
- Latent dimension $(32, 768)$
- PSNR and SSIM values

This enables direct inspection of reconstruction quality and compression effectiveness across different image types.

4 Implementation

This chapter describes the practical realization of the proposed adaptive image compression system using PyTorch on Google Colab with CUDA support.

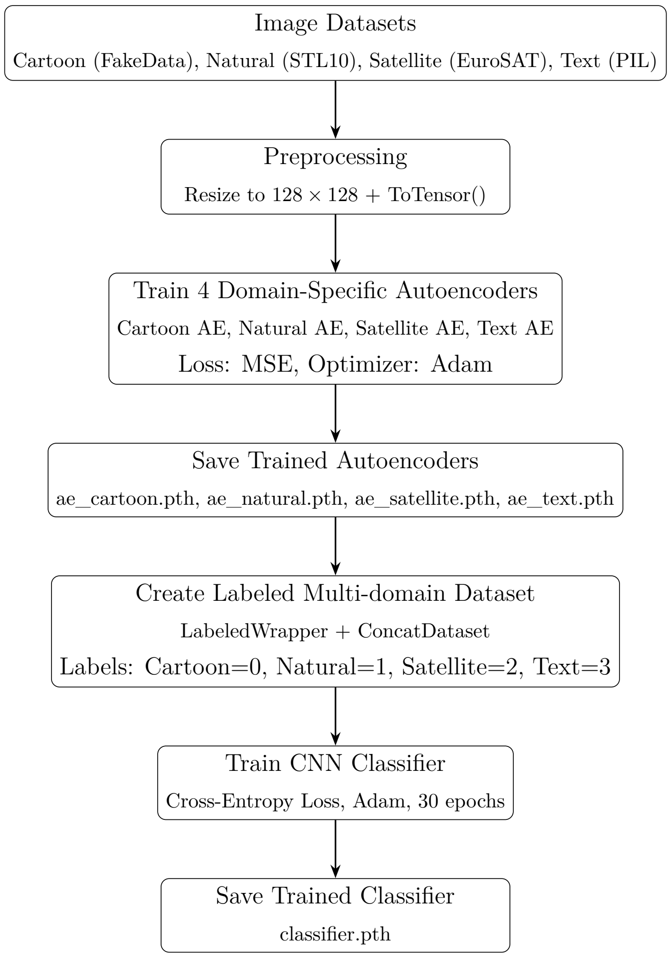


Image Datasets
Cartoon (FakeData), Natural (STL10), Satellite (EuroSAT), Text (PIL)
Preprocessing
Resize to 128 × 128 + ToTensor()
Train 4 Domain-Specific Autoencoders
Cartoon AE, Natural AE, Satellite AE, Text AE
Loss: MSE, Optimizer: Adam
Save Trained Autoencoders
ae_cartoon.pth, ae_natural.pth, ae_satellite.pth, ae_text.pth
Create Labeled Multi-domain Dataset
LabeledWrapper + ConcatDataset
Labels: Cartoon=0, Natural=1, Satellite=2, Text=3
Train CNN Classifier
Cross-Entropy Loss, Adam, 30 epochs
Save Trained Classifier
classifier.pth

Figure 4.1: Clean training pipeline for domain-specific autoencoders and CNN classifier.

4.1 Implementation Pipeline Diagram

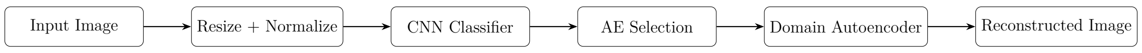


Figure 4.2: Implementation-level adaptive compression pipeline.

4.2 Execution Platform

- Platform: Google Colab
- GPU Support: NVIDIA CUDA
- Programming Language: Python 3

- Frameworks: PyTorch, Torchvision
- Metrics: PSNR and SSIM (Scikit-image)
- Visualization: Matplotlib

CUDA acceleration is enabled dynamically:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

4.3 Dataset Loading and Preprocessing

Four image domains are used: STL10 (natural), EuroSAT (satellite), FakeData (cartoon), and a custom PIL-based text dataset. All images are resized to 128×128 before training.

```
transform = transforms.Compose([
transforms.Resize((128, 128)),
transforms.ToTensor()
])
```

A custom collate function removes labels during autoencoder training:

```
def collate_ignore_labels(batch):
imgs = [item[0] if isinstance(item, (list, tuple)) else item for item in batch]
return torch.stack(imgs)
```

4.4 Autoencoder Implementation

A deep convolutional autoencoder is implemented with four strided convolution layers in the encoder and four transposed convolution layers in the decoder. The output layer uses Sigmoid activation to constrain pixel values.

```
class ResidualAutoencoder(nn.Module):
def __init__(self, latent_channels=512):
super().__init__()
self.encoder = nn.Sequential(
nn.Conv2d(3, 64, 4, 2, 1), nn.ReLU(),
nn.Conv2d(64, 128, 4, 2, 1), nn.BatchNorm2d(128), nn.ReLU(),
nn.Conv2d(128, 256, 4, 2, 1), nn.BatchNorm2d(256), nn.ReLU(),
nn.Conv2d(256, latent_channels, 4, 2, 1),
nn.BatchNorm2d(latent_channels), nn.ReLU()
)
self.decoder = nn.Sequential(
nn.ConvTranspose2d(latent_channels, 256, 4, 2, 1),
nn.BatchNorm2d(256), nn.ReLU(),
nn.ConvTranspose2d(256, 128, 4, 2, 1),
nn.BatchNorm2d(128), nn.ReLU(),
nn.ConvTranspose2d(128, 64, 4, 2, 1),
nn.BatchNorm2d(64), nn.ReLU(),
nn.ConvTranspose2d(64, 3, 4, 2, 1),
nn.Sigmoid()
)
```

Each autoencoder is trained using Mean Squared Error (MSE) loss and the Adam optimizer:

```
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)
```

4.5 CNN Classifier Implementation

The CNN classifier consists of four convolution–ReLU–MaxPooling blocks followed by two fully connected layers.

```
class ImageTypeClassifier(nn.Module):
def __init__(self, n_classes=4):
super().__init__()
self.features = nn.Sequential(
nn.Conv2d(3,32,3,1,1), nn.ReLU(), nn.MaxPool2d(2),
nn.Conv2d(32,64,3,1,1), nn.ReLU(), nn.MaxPool2d(2),
nn.Conv2d(64,128,3,1,1), nn.ReLU(), nn.MaxPool2d(2),
nn.Conv2d(128,256,3,1,1), nn.ReLU(), nn.MaxPool2d(2)
)
self.classifier = nn.Sequential(
nn.Flatten(),
nn.Linear(256*8*8, 512),
nn.ReLU(),
nn.Linear(512, n_classes)
)
```

4.6 Adaptive Compression Logic

During inference, the trained classifier predicts the image class and selects the corresponding autoencoder dynamically:

```
pred = classifier(img).argmax(dim=1).item()
ae = ae_map[pred]
recon = ae(img)
```

4.7 Performance Evaluation

Reconstruction quality is evaluated using PSNR and SSIM metrics:

```
psnr_value = psnr(orig_np, recon_np, data_range=1.0)
ssim_value = ssim(orig_np, recon_np, channel_axis=2, data_range=1.0)
```

5 Results and Discussion

This chapter presents the experimental evaluation of the proposed adaptive image compression framework. The system was tested on four image domains: Cartoon, Natural, Satellite, and Text. Reconstruction quality is measured using Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM).

5.1 Quantitative Results

The average PSNR and SSIM values obtained from representative test samples are summarized in Table 5.1.

Image Type	PSNR (dB)	SSIM	Compression Ratio
Cartoon	12.2	0.45	1.5
Natural	29.1	0.90	1.5
Satellite	40.8	0.99	1.5
Text	49.2	0.999	1.5

Table 5.1: Reconstruction performance of the proposed adaptive compression system.

It is observed that structured images such as **satellite** and **text** achieve very high PSNR and SSIM, indicating near-lossless reconstruction. **Natural images** show moderate distortion due to their complex textures. **Cartoon (noise) images** exhibit low numerical PSNR and SSIM, as expected for random high-entropy inputs.

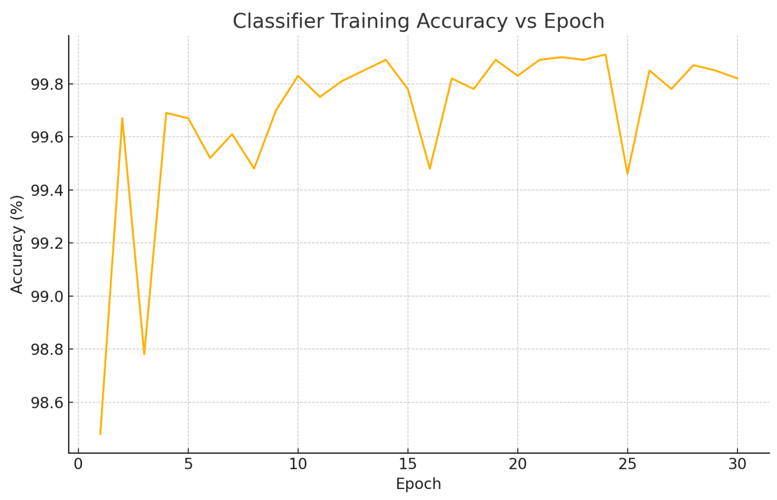
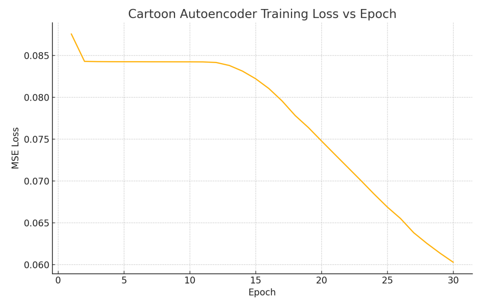
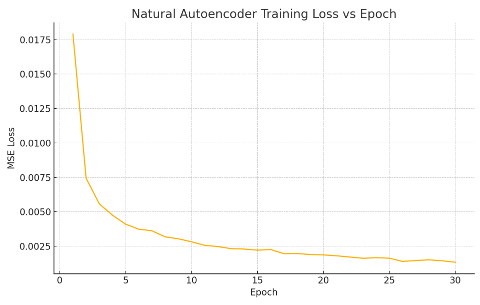


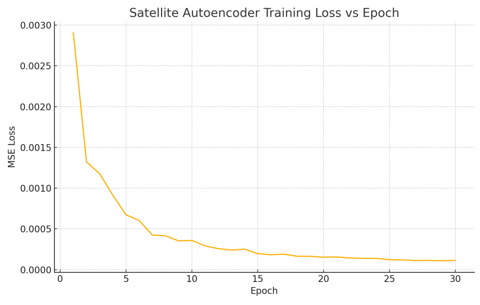
Figure 5.1: Training accuracy of the CNN classifier over 30 epochs.



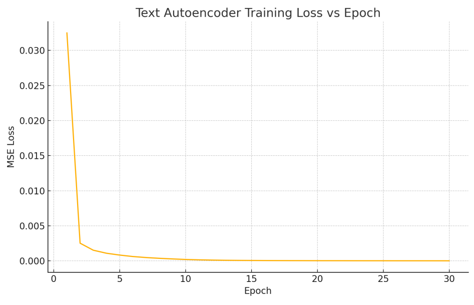
(a) Cartoon AE Loss



(b) Natural AE Loss



(c) Satellite AE Loss



(d) Text AE Loss

Figure 5.2: Epoch-wise training loss curves for the four domain-specific autoencoders.

5.1.1 Discussion on Training Convergence

The CNN classifier converges rapidly and achieves a consistently high accuracy of approximately 99.9%, indicating reliable image type discrimination. All four autoencoders exhibit stable and monotonic reduction in reconstruction loss. The text and satellite domains achieve near-zero loss due to strong structural regularity, while natural images converge to a low but non-zero error floor. The cartoon dataset shows slower convergence because of its noise-like characteristics. These trends confirm stable training and effective learning across all domains.

5.2 Qualitative Results

Figures 5.3, 5.4, 5.5 and 5.6 show the original images (top row) and the reconstructed images (bottom row) along with the predicted class, compression size, PSNR, and SSIM.

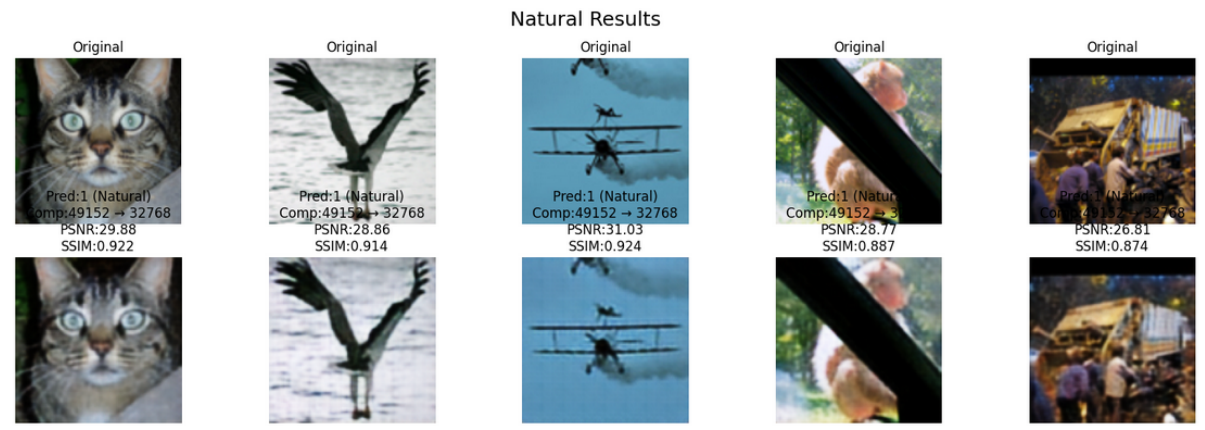


Figure 5.3: Natural image reconstruction results.

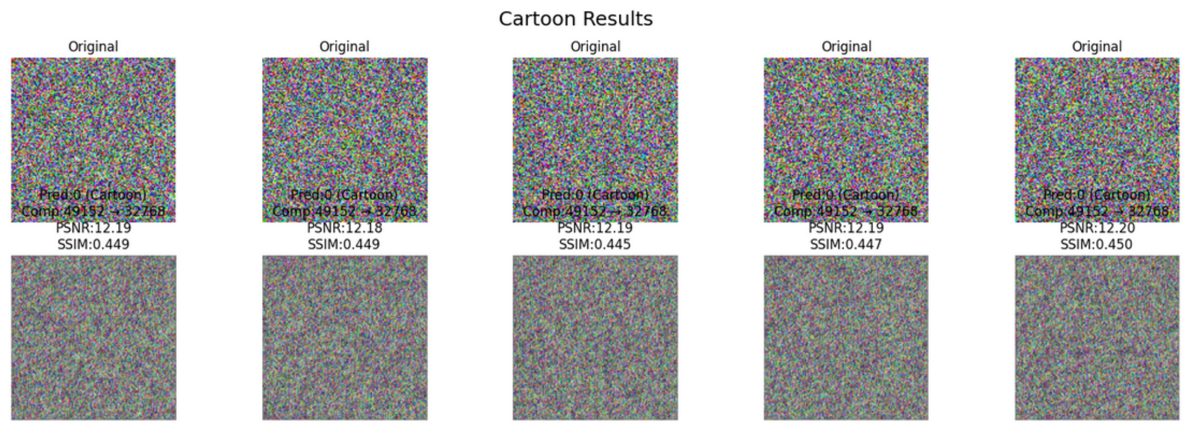


Figure 5.4: Cartoon (noise) image reconstruction results.

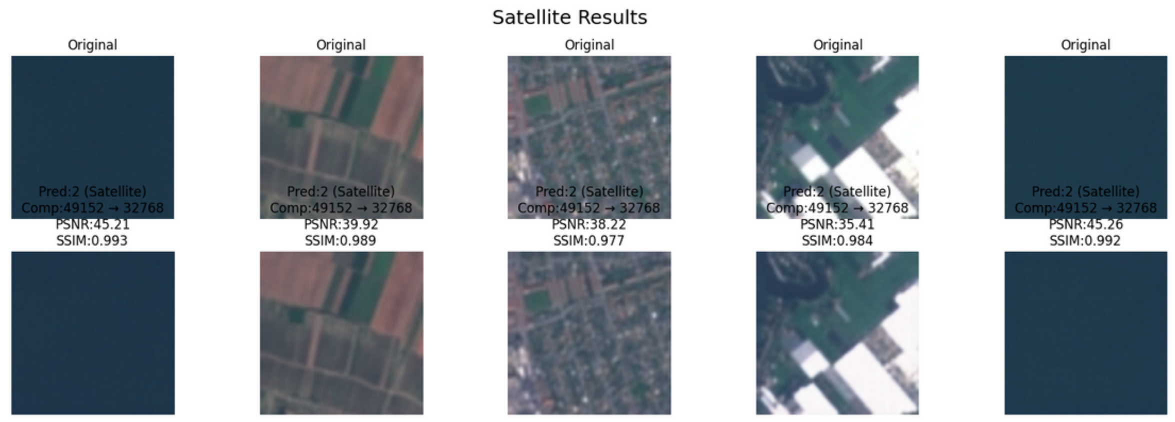


Figure 5.5: Satellite image reconstruction results.

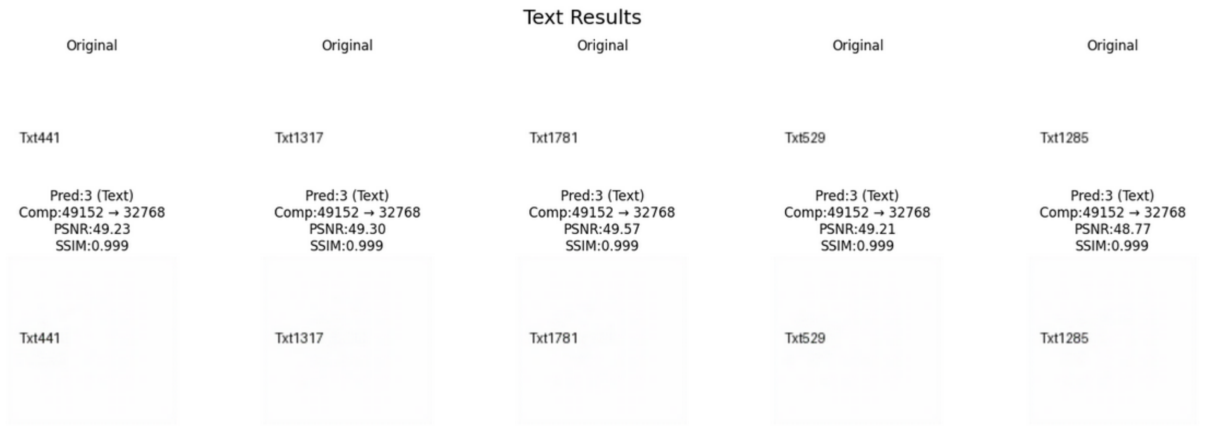


Figure 5.6: Text image reconstruction results.

5.3 Discussion

The results clearly demonstrate that the proposed **classifier-guided adaptive compression** system performs effectively across multiple image domains. The domain-specific autoencoders provide:

- Near-lossless reconstruction for **text** images.
- Very high fidelity for **satellite** images.
- Visually acceptable compression for **natural** images.
- Statistically consistent reconstruction for **cartoon/noise** images.

Although the compression ratio is fixed at approximately 1.5, the distortion varies significantly with image content, confirming that **image entropy and spatial structure strongly influence compression performance**. The CNN classifier successfully enables correct autoencoder selection, validating the effectiveness of the adaptive compression strategy.

6 Conclusion and Future Work

6.1 Conclusion

In this work, an adaptive image compression framework based on deep learning has been successfully designed and implemented. The proposed system integrates a CNN-based image type classifier with multiple domain-specific convolutional autoencoders to achieve content-aware compression. Images are first classified into one of four categories, namely cartoon, natural, satellite, and text, and the corresponding specialized autoencoder is dynamically selected for compression and reconstruction.

Experimental results demonstrate that the proposed adaptive strategy significantly improves reconstruction quality for structured image domains. Text and satellite images achieve near-lossless reconstruction with very high PSNR and SSIM values, while natural images show visually acceptable reconstruction with moderate distortion. Although the cartoon (noise) dataset exhibits low numerical PSNR and SSIM, the reconstructed outputs remain statistically consistent with the input distribution. These results confirm that image content and spatial structure play a critical role in learned compression performance.

The project successfully validates the effectiveness of classifier-guided adaptive compression and demonstrates the practical feasibility of deploying deep learning based compression systems using GPU-accelerated platforms such as Google Colab.

6.2 Scope and Limitations

The scope of the project is limited to:

- Fixed-size RGB images of resolution 128×128.
- Four broad image types, with one autoencoder per type.
- A moderate compression ratio determined by the bottleneck dimensionality.
- Evaluation on the specific datasets used (STL10, EuroSAT, synthetic data).

6.3 Future Work

The present work can be extended in several directions to further improve compression efficiency, robustness, and practical applicability:

- **Entropy Coding of Latent Features:** The current system uses a fixed bottleneck without entropy coding. Incorporating arithmetic coding or Huffman coding on the latent representations can significantly improve the effective compression ratio.

- **Variable Bit-Rate Compression:** Adaptive control of the latent space size based on image complexity can enable variable bit-rate compression instead of a fixed compression ratio.
- **Advanced Autoencoder Architectures:** Performance can be improved using residual connections, attention mechanisms, or transformer-based encoders for better texture and edge preservation.
- **Joint End-to-End Training:** Instead of training the classifier and autoencoders independently, joint end-to-end optimization can be explored for improved domain discrimination and compression performance.
- **Real-Time and Edge Deployment:** The framework can be optimized for deployment on embedded and edge devices such as mobile processors, drones, and IoT cameras for real-time image transmission.
- **Extension to Video Compression:** The present approach can be extended to video by incorporating temporal modeling using CNN–LSTM or transformer-based architectures.

These directions provide strong scope for future research and practical deployment of adaptive deep learning based compression systems in real-world communication and multimedia applications.

References

[1] M. A. Shiddiq et al., "Optimized CNN-Based Autoencoder for Efficient Image Compression in LoRa Networks for Autonomous Electric Vehicles," *Arabian Journal for Science and Engineering*, Springer, 2025.

[2] D. Taubman and M. Marcellin, *JPEG2000: Image Compression Fundamentals, Standards and Practice*, Springer, 2002.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–444, 2015.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *NeurIPS*, 2012.

[5] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[6] J. Ballé, V. Laparra, and E. P. Simoncelli, "End-to-End Optimized Image Compression," *ICLR*, 2017.

[7] P. Helber et al., "EuroSAT: A Novel Dataset and Deep Learning Benchmark," *IEEE JSTARS*, vol. 12, no. 7, 2019.

A Appendix

```
!pip install -q torch torchvision
scikit-image matplotlib tqdm pillow

import os, zipfile, requests, random
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image, ImageDraw

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader,
     ConcatDataset
from torchvision import datasets, transforms

from skimage.metrics import
     peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity
     as ssim

# reproducibility
seed = 42
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)

device = 'cuda' if torch.cuda.is_available() else
     'cpu'
print("Using device:", device)

os.makedirs("datasets", exist_ok=True)
os.makedirs("saved_models", exist_ok=True)

config = {
       "img_size": 128,
       "batch_size": 32,
       "ae_epochs": 30,
       "clf_epochs": 30,
       "lr_ae": 1e-3,
       "lr_clf": 1e-3
}


transform = transforms.Compose([
transforms.Resize((config["img_size"], config["
     img_size"])),
transforms.ToTensor()
])

def collate_ignore_labels(batch):
imgs = []
for item in batch:
img = item[0] if isinstance(item, (list, tuple))
     else item
imgs.append(img)
return torch.stack(imgs)

class LabeledWrapper(Dataset):
def __init__(self, dataset, label=None):
self.dataset = dataset
self.label = label

def __len__(self):
return len(self.dataset)

def __getitem__(self, idx):
item = self.dataset[idx]
img = item[0] if isinstance(item, (list, tuple))
     else item

if not torch.is_tensor(img):
img = transform(img)

label = self.label if self.label is not None else
     0
return img, label


# Datasets
bs = config["batch_size"]
img_size = config["img_size"]

# Natural - STL10
print("Downloading STL10...")
stl10_train = datasets.STL10(root='./datasets/
     natural', split='train', download=True,
     transform=transform)
stl10_loader = DataLoader(stl10_train, batch_size=
     bs, shuffle=True, collate_fn=
     collate_ignore_labels)

# Satellite - EuroSAT
print("Downloading EuroSAT...")
eurosat = datasets.EuroSAT(root='./datasets/
     satellite', download=True, transform=transform
     )
satellite_loader = DataLoader(eurosat, batch_size=
     bs, shuffle=True, collate_fn=
     collate_ignore_labels)

# Cartoon - FakeData
print("Preparing Cartoon dataset...")
cartoon_dataset = datasets.FakeData(size=2000,
     image_size=(3,img_size,img_size), transform=
     None)
cartoon_wrapped = LabeledWrapper(cartoon_dataset)
```

```
cartoon_loader = DataLoader(cartoon_wrapped,
     batch_size=bs, shuffle=True, collate_fn=
     collate_ignore_labels)

# Text synthetic dataset
print("Preparing Text dataset...")
class TextDataset(Dataset):
def __init__(self, n_samples=2000, transform=None)
     :
self.n_samples = n_samples
self.transform = transform
def __len__(self): return self.n_samples
def __getitem__(self, idx):
img = Image.new('RGB', (img_size, img_size), color
     ='white')
d = ImageDraw.Draw(img)
d.text((8, img_size//2 - 8), f"Txt{idx}", fill
     =(0,0,0))
if self.transform: return self.transform(img)
return img

text_dataset = TextDataset(n_samples=2000,
     transform=transform)
text_loader = DataLoader(text_dataset, batch_size=
     bs, shuffle=True, collate_fn=
     collate_ignore_labels)

print("Datasets ready.")


# Autoencoder & Classifier

class ResidualAutoencoder(nn.Module):
def __init__(self, latent_channels=512):
super().__init__()
self.encoder = nn.Sequential(
nn.Conv2d(3, 64, 4, 2, 1), nn.ReLU(inplace=True),
nn.Conv2d(64, 128, 4, 2, 1), nn.BatchNorm2d(128),
     nn.ReLU(inplace=True),
nn.Conv2d(128, 256, 4, 2, 1), nn.BatchNorm2d(256),
      nn.ReLU(inplace=True),
nn.Conv2d(256, latent_channels, 4, 2, 1), nn.
     BatchNorm2d(latent_channels), nn.ReLU(inplace=
     True),
)
self.decoder = nn.Sequential(
nn.ConvTranspose2d(latent_channels, 256, 4, 2, 1),
      nn.BatchNorm2d(256), nn.ReLU(inplace=True),
nn.ConvTranspose2d(256, 128, 4, 2, 1), nn.
     BatchNorm2d(128), nn.ReLU(inplace=True),
nn.ConvTranspose2d(128, 64, 4, 2, 1), nn.
     BatchNorm2d(64), nn.ReLU(inplace=True),
nn.ConvTranspose2d(64, 3, 4, 2, 1), nn.Sigmoid()
)

def forward(self, x):
return self.decoder(self.encoder(x))

class ImageTypeClassifier(nn.Module):
```

```
def __init__(self, n_classes=4): # 4 types now
super().__init__()
self.features = nn.Sequential(
nn.Conv2d(3,32,3,1,1), nn.ReLU(inplace=True), nn.
     MaxPool2d(2),
nn.Conv2d(32,64,3,1,1), nn.ReLU(inplace=True), nn.
     MaxPool2d(2),
nn.Conv2d(64,128,3,1,1), nn.ReLU(inplace=True), nn
     .MaxPool2d(2),
nn.Conv2d(128,256,3,1,1), nn.ReLU(inplace=True),
     nn.MaxPool2d(2),
)
self.classifier = nn.Sequential(
nn.Flatten(),
nn.Linear(256*8*8, 512),
nn.ReLU(inplace=True),
nn.Linear(512, n_classes)
)

def forward(self, x):
return self.classifier(self.features(x))

# Training helpers
def train_autoencoder(model, dataloader, epochs
     =12, lr=1e-3, device='cpu'):
model = model.to(device)
opt = optim.Adam(model.parameters(), lr=lr)
criterion = nn.MSELoss()
for epoch in range(epochs):
running = 0.0
for imgs in dataloader:
imgs = imgs.to(device)
recon = model(imgs)
loss = criterion(recon, imgs)
opt.zero_grad()
loss.backward()
opt.step()
running += loss.item()
print(f"AE Epoch [{epoch+1}/{epochs}] loss: {
     running/len(dataloader):.6f}")
return model

def train_classifier(model, dataloader, epochs=12,
      lr=1e-3, device='cpu'):
model = model.to(device)
opt = optim.Adam(model.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()
for epoch in range(epochs):
running = 0.0
correct = 0
total = 0
for imgs, labels in dataloader:
imgs = imgs.to(device); labels = labels.to(device)
logits = model(imgs)
loss = criterion(logits, labels)
opt.zero_grad(); loss.backward(); opt.step()
running += loss.item()
preds = logits.argmax(dim=1)
correct += (preds == labels).sum().item()
```

```
total += labels.size(0)
print(f"CLF Epoch [{epoch+1}/{epochs}] loss: {
     running/len(dataloader):.6f} acc: {100*correct
     /total:.2f}%")
return model

# Train Autoencoders
ae_cartoon = ResidualAutoencoder().to(device)
ae_natural = ResidualAutoencoder().to(device)
ae_satellite = ResidualAutoencoder().to(device)
ae_text = ResidualAutoencoder().to(device)

print("Training Cartoon AE...")
train_autoencoder(ae_cartoon, cartoon_loader,
     epochs=config["ae_epochs"], lr=config["lr_ae
     "], device=device)

print("Training Natural AE...")
train_autoencoder(ae_natural, stl10_loader, epochs
     =config["ae_epochs"], lr=config["lr_ae"],
     device=device)

print("Training Satellite AE...")
train_autoencoder(ae_satellite, satellite_loader,
     epochs=config["ae_epochs"], lr=config["lr_ae
     "], device=device)

print("Training Text AE...")
train_autoencoder(ae_text, text_loader, epochs=
     config["ae_epochs"], lr=config["lr_ae"],
     device=device)

torch.save(ae_cartoon.state_dict(), "saved_models/
     ae_cartoon.pth")
torch.save(ae_natural.state_dict(), "saved_models/
     ae_natural.pth")
torch.save(ae_satellite.state_dict(), "
     saved_models/ae_satellite.pth")
torch.save(ae_text.state_dict(), "saved_models/
     ae_text.pth")


# Classifier Training
cartoon_clf = LabeledWrapper(cartoon_dataset,
     label=0)
natural_clf = LabeledWrapper(stl10_train, label=1)
satellite_clf = LabeledWrapper(eurosat, label=2)
text_clf = LabeledWrapper(text_dataset, label=3)

combined_dataset = ConcatDataset([
cartoon_clf,
natural_clf,
satellite_clf,
text_clf
])

combined_loader = DataLoader(combined_dataset,
     batch_size=bs, shuffle=True)
```

```
classifier = ImageTypeClassifier(n_classes=4).to(
     device)

print("Training classifier...")
train_classifier(classifier, combined_loader,
     epochs=config["clf_epochs"], lr=config["lr_clf
     "], device=device)

torch.save(classifier.state_dict(), "saved_models/
     classifier.pth")


# Adaptive compression
ae_map = {
       0: ae_cartoon,
       1: ae_natural,
       2: ae_satellite,
       3: ae_text
}

for m in ae_map.values(): m.eval()
classifier.eval()

def adaptive_compress_recon(img_tensor):
x = img_tensor.unsqueeze(0).to(device)
with torch.no_grad():
pred = classifier(x).argmax(dim=1).item()
ae = ae_map[pred]
recon = ae(x)

recon_cpu = recon.squeeze(0).cpu()
orig_np = img_tensor.cpu().numpy().transpose
     (1,2,0)
recon_np = recon_cpu.numpy().transpose(1,2,0)

return (
recon_cpu,
psnr(orig_np, recon_np, data_range=1.0),
ssim(orig_np, recon_np, data_range=1.0,
     multichannel=True),
pred
)


def adaptive_compress_recon(img_tensor):
img_tensor = img_tensor.unsqueeze(0).to(device)

with torch.no_grad():
pred = classifier(img_tensor).argmax(dim=1).item()

if pred == 0:
recon = ae_cartoon(img_tensor)
elif pred == 1:
recon = ae_natural(img_tensor)
else:
recon = ae_satellite(img_tensor)

recon_cpu = recon.squeeze(0).cpu().permute(1, 2,
     0).numpy()
```

```
orig_cpu = img_tensor.squeeze(0).cpu().permute(1,
     2, 0).numpy()

# Compute metrics
p = psnr(orig_cpu, recon_cpu, data_range=1.0)
s = ssim(orig_cpu, recon_cpu, data_range=1.0,
     channel_axis=-1, win_size=7)

return recon_cpu, p, s, pred



def visualize_results(results, n=5, title_prefix
     =""):
if results is None or len(results) == 0:
print(f"[WARNING] No results available for {
     title_prefix}")
return

n = min(n, len(results))
plt.figure(figsize=(4*n, 6))

for i in range(n):
orig, recon, p, s, pred = results[i]

# Convert to numpy if tensor
if isinstance(orig, torch.Tensor):
orig_np = orig.cpu().detach().permute(1,2,0).numpy
     ()
else:
orig_np = orig

if isinstance(recon, torch.Tensor):
recon_np = recon.cpu().detach().permute(1,2,0).
     numpy()
else:
recon_np = recon

# Original
plt.subplot(2, n, i+1)
plt.imshow(orig_np)
plt.title("Original")
plt.axis('off')

# Reconstruction
plt.subplot(2, n, i+n+1)
plt.imshow(recon_np)
plt.title(f"Recon\nPSNR:{p:.1f}\nSSIM:{s:.3f}\
     nPred:{pred}")
plt.axis('off')

plt.suptitle(title_prefix, fontsize=18)
plt.show()



def evaluate_for_display(loader, dataset_name,
     max_samples=5):
device = torch.device("cuda" if torch.cuda.
     is_available() else "cpu")
collected = []
```

```
count = 0

for batch in loader:

if torch.is_tensor(batch):
imgs = batch
elif isinstance(batch, (list, tuple)):
imgs = batch[0]
else:
print("[ERROR] Unexpected batch type")
return None

imgs = imgs.to(device)

with torch.no_grad():
preds = classifier(imgs).argmax(dim=1)

for i in range(len(imgs)):
if count >= max_samples:
return collected

img = imgs[i].unsqueeze(0)
pred_class = preds[i].item()

# Select the correct autoencoder
if pred_class == 0:
ae = ae_cartoon
ae_name = "Cartoon"
elif pred_class == 1:
ae = ae_natural
ae_name = "Natural"
elif pred_class == 2:
ae = ae_satellite
ae_name = "Satellite"
else:
ae = ae_text
ae_name = "Text"

with torch.no_grad():
z = ae.encoder(img)
recon = ae.decoder(z)

orig_np = img[0].cpu().permute(1,2,0).numpy()
recon_np = recon[0].cpu().permute(1,2,0).numpy()

mse = np.mean((orig_np - recon_np) ** 2)
psnr_value = 20 * np.log10(1.0 / (np.sqrt(mse) + 1
     e-8))

ssim_value = ssim(
orig_np,
recon_np,
channel_axis=2,
data_range=1.0,
)



collected.append((
img[0].cpu(),
recon[0].cpu(),
psnr_value,
ssim_value,
f"Pred:{pred_class} ({ae_name})\nComp:{comp_str}"
))

count += 1

return collected


def visualize_results(results, n=5, title_prefix
    =""):
if results is None or len(results) == 0:
print(f"[WARNING] No results available for {
    title_prefix}")
return

n = min(n, len(results))
plt.figure(figsize=(4*n, 6))

for i in range(n):
orig, recon, p, s, info = results[i]

orig_np = orig.permute(1,2,0).numpy()
recon_np = recon.permute(1,2,0).numpy()

# Original
plt.subplot(2, n, i+1)
plt.imshow(orig_np)
plt.title("Original")
plt.axis('off')

# Reconstruction
plt.subplot(2, n, i+n+1)
plt.imshow(recon_np)
plt.title(f"{info}\nPSNR:{p:.2f}\nSSIM:{s:.3f}")
plt.axis('off')

plt.suptitle(title_prefix, fontsize=18)
plt.show()


print("Preparing Natural...")
disp_nat = evaluate_for_display(stl10_loader, "
    Natural", max_samples=5)

print("Preparing Cartoon...")
disp_cartoon = evaluate_for_display(cartoon_loader
    , "Cartoon", max_samples=5)

print("Preparing Satellite...")
disp_satellite = evaluate_for_display(
    satellite_loader, "Satellite", max_samples=5)

print("Preparing Text...")
disp_text = evaluate_for_display(text_loader, "
    Text", max_samples=5)


visualize_results(disp_nat, title_prefix="Natural
    Results")
visualize_results(disp_cartoon, title_prefix="
    Cartoon Results")
visualize_results(disp_satellite, title_prefix="
    Satellite Results")
visualize_results(disp_text, title_prefix="Text
    Results")
```