19CSE456 Neural Network and Deep Learning Laboratory

List of Experiments

Week #	Experiment Title
1	Introduction to the lab and Implementation of a simple Perceptron (Hardcoding)
2	Implementation of Perceptron for Logic Gates (Hardcoding, Sklearn, TF)
3	Implementation of Multilayer Perceptron for XOR Gate and other classification problems with ML toy datasets (Hardcoding & TF)
4	Implementation of MLP for Image Classification with MNIST dataset (Hardcoding & TF)
5	Activation Functions, Loss Functions, Optimizers (Hardcoding & TF)
6	Lab Evaluation 1 (based on topics covered from w1 to w5)
7	Convolution Neural Networks for Toy Datasets (MNIST & CIFAR)
8	Convolution Neural Networks for Image Classification (Oxford Pets, Tiny ImageNet, etc.)
9	Recurrent Neural Networks for Sentiment Analysis with IMDB Movie Reviews
10	Long Short Term Memory for Stock Prices (Yahoo Finance API)

List of Experiments contd.

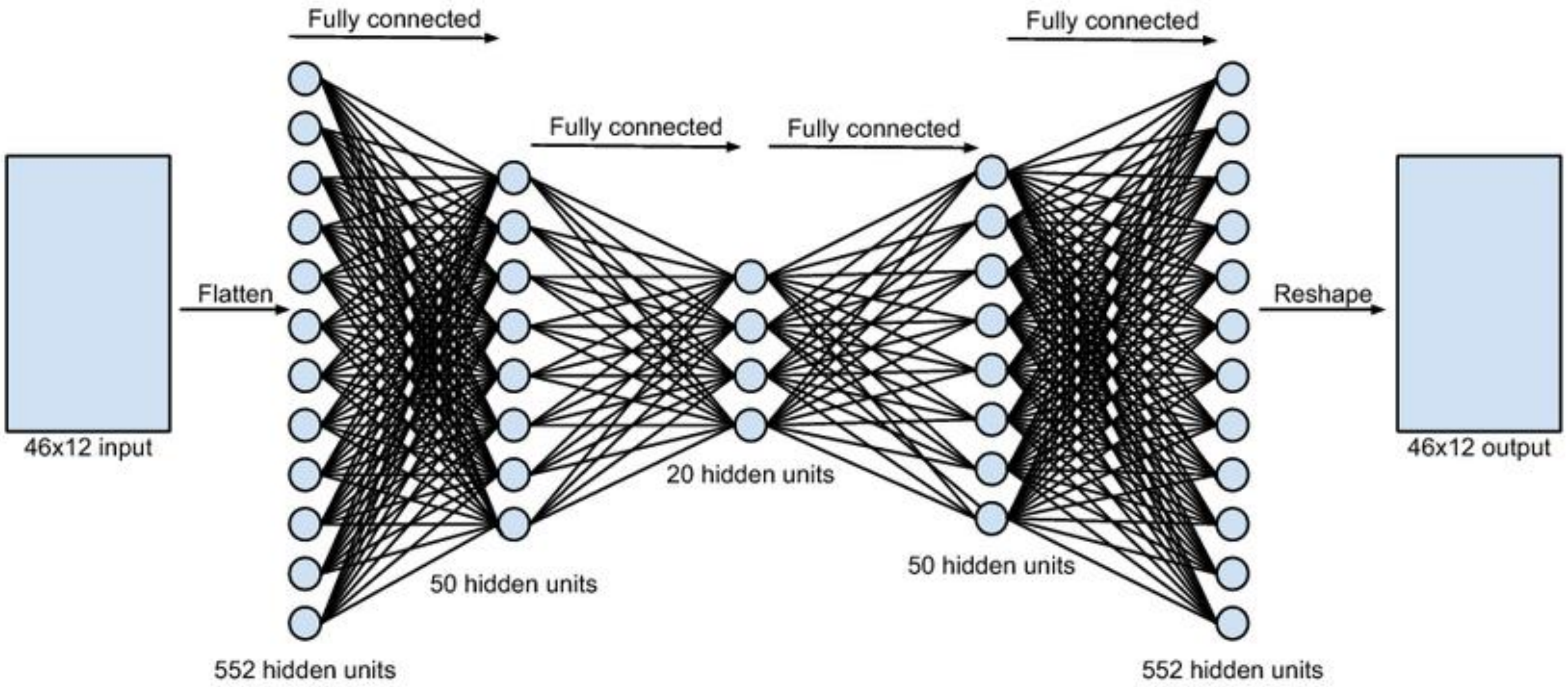
Week #	Experiment Title
11	Implementation of Autoencoders and Denoising Autoencoders (MNIST/CIFAR)
12	Boltzmann Machines (MNIST/CIFAR)
13	Restricted Boltzmann Machines (MNIST/CIFAR)
14	Hopfield Neural Networks (MNIST/CIFAR)
15	Lab Evaluation 2 (based on CNN, RNN, LSTM, and AEs)
16	Case Study Review (Phase 1)
17	Case Study Review (Phase 1)

Autoencoders

- Autoencoders are a type of artificial neural network designed to learn an efficient representation, or encoding, of input data
- Their goal is to compress the input into a smaller, latent representation (the bottleneck) and then reconstruct the original data as closely as possible.
- Essentially, they consist of two parts:
  1. Encoder:
     - This part compresses the input data into a lower-dimensional representation
     - It captures the most important features or patterns while reducing noise or irrelevant information
  2. Decoder:
     - This part reconstructs the data from the compressed representation
     - The output of the decoder is compared to the original input, and the network is trained to minimize the reconstruction error

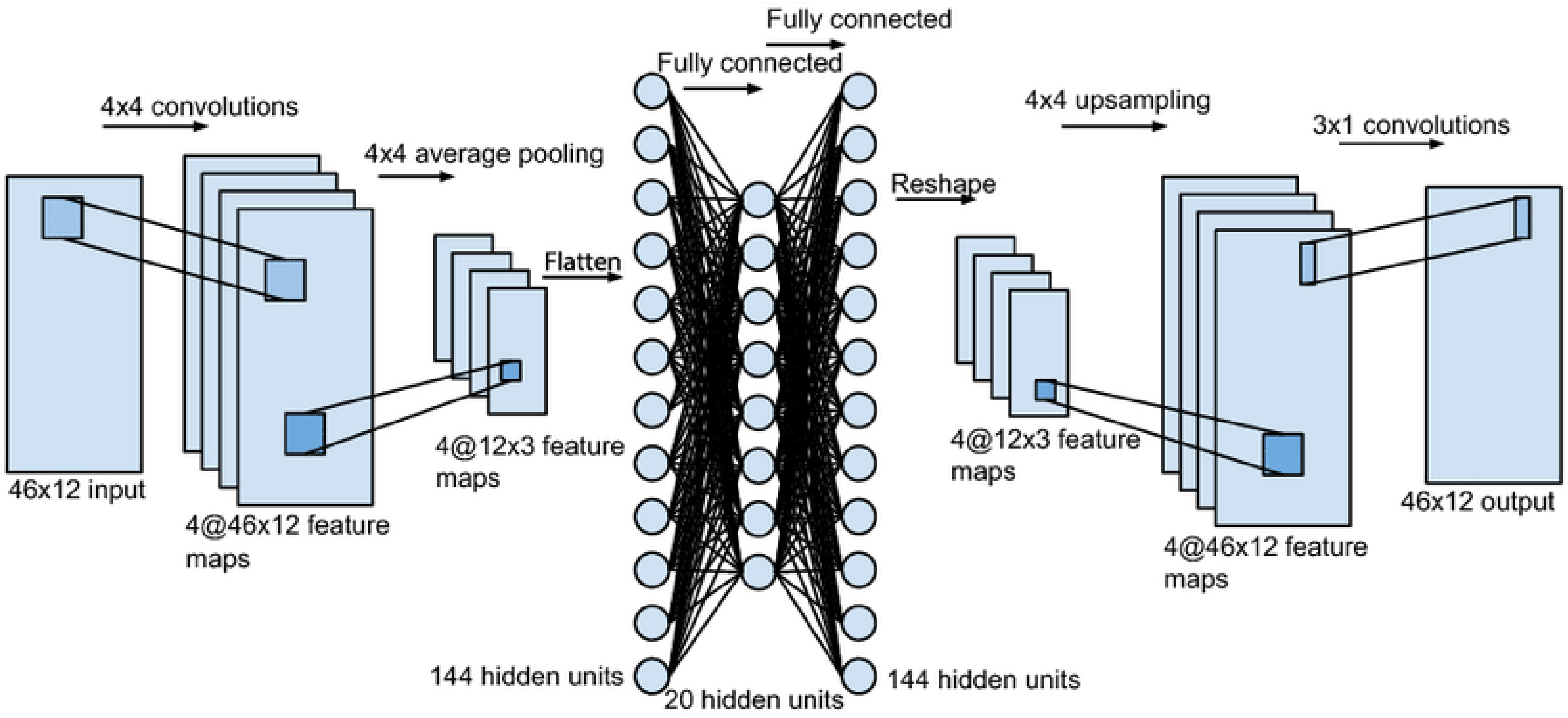
Autoencoder Implementation Types: MLP

- MLP is simpler and works best for lower-dimensional or structured data
- However, it struggles with high-dimensional data like images, as it doesn't preserve spatial context



Autoencoder Implementation Types: Convolution

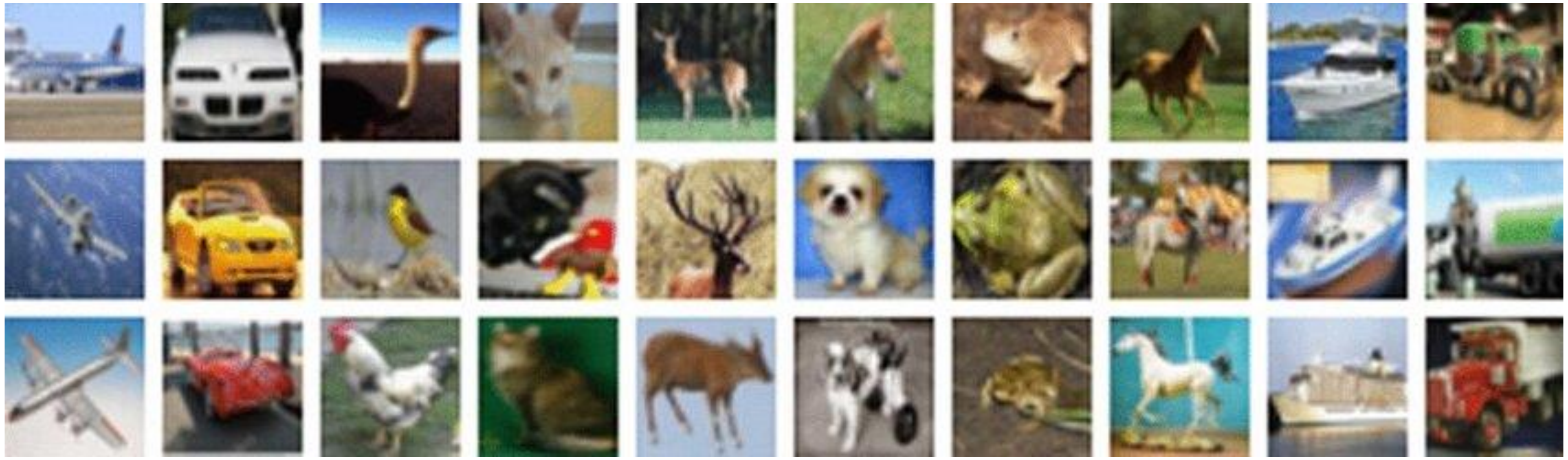
- Convolutional Layers excel in tasks where spatial relationships are essential, like image reconstruction or generation



CIFAR-10 Dataset

The CIFAR-10 dataset is a widely used benchmark dataset in machine learning and computer vision, particularly for image classification tasks.

- Content: 60,000 color images, each of size 32x32 pixels, divided into 10 distinct classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks
- Challenges: The images are low-resolution and include various poses, lighting conditions, and occlusions, making it a valuable dataset for testing the robustness of algorithms



Autoencoders

Load and Preprocess the Data → Model Building → Model Training → Model Evaluation → Visualization

```
def load_and_preprocess_data(add_noise=True, noise_factor=0.2):
    (x_train, _), (x_test, _) = cifar10.load_data()
    x_train = x_train.astype('float32') / 255.0
    x_test = x_test.astype('float32') / 255.0
    print(f"Training data shape: {x_train.shape}")
    print(f"Test data shape: {x_test.shape}")
    x_train_noisy = x_train.copy()
    x_test_noisy = x_test.copy()
    # Add random Gaussian noise if requested
    if add_noise:
        x_train_noisy = x_train_noisy + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_train.shape)
        x_test_noisy = x_test_noisy + noise_factor * np.random.normal(loc=0.0, scale=1.0, size=x_test.shape)
        # Clip the values to be between 0 and 1
        x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
        x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)
    return x_train_noisy, x_train, x_test_noisy, x_test
```

A scaling factor controlling the intensity of the noise added

Converts the pixel values to floats and normalizes them to the range [0, 1]

Generates random Gaussian noise with mean (loc=0.0) and standard deviation (scale=1.0), shaped to match the data dimensions

Noise addition can cause some pixel values to go beyond the valid range [0, 1] - np.clip ensures all values remain within this range, preventing invalid data

Autoencoders - MLP

Load and Preprocess the Data → **Model Building** → Model Training → Model Evaluation → Visualization

```
def build_autoencoder(input_shape=(32, 32, 3), encoding_dim=256):
    input_dim = np.prod(input_shape)
    input_img = Input(shape=input_shape)
    flat_img = Flatten()(input_img)
    encoded = Dense(512, activation='relu')(flat_img)
    encoded = Dense(encoding_dim, activation='relu')(encoded)
    decoded = Dense(512, activation='relu')(encoded)
    decoded = Dense(input_dim, activation='sigmoid')(decoded)
    output_img = Reshape(input_shape)(decoded)
    autoencoder = Model(input_img, output_img)
    encoder = Model(input_img, encoded)
    encoded_input = Input(shape=(encoding_dim,))
    decoder_layers = autoencoder.layers[-3](encoded_input)
    decoder_layers = autoencoder.layers[-2](decoder_layers)
    decoder_output = Reshape(input_shape)(decoder_layers)
    decoder = Model(encoded_input, decoder_output)
    return autoencoder, encoder, decoder
```

The size of the compressed latent representation

Calculates the total number of elements (pixels) in a single image

Creates an input layer for the autoencoder

Converts the 3D image input into a 1D array

Adds a dense layer with 512 neurons and ReLU activation

Further reduces the data's dimensionality to create a compressed latent representation

Expands the latent representation back to a higher-dimensional feature space

Reconstructs the flattened input image using sigmoid activation

Reshapes the 1D output back into its original 3D format

Construction of an Autoencoder and Separation of Encoder and Decoder models

Autoencoders - Convolutional

Load and Preprocess the Data → **Model Building** → Model Training → Model Evaluation → Visualization

```
def build_convolutional_autoencoder(input_shape=(32, 32, 3)):
    # Input layer
    input_img = Input(shape=input_shape)
    # Encoder
    x = Conv2D(32, (3, 3), padding='same')(input_img)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)  # 16x16x32
    x = Conv2D(64, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    x = MaxPooling2D((2, 2), padding='same')(x)  # 8x8x64
    x = Conv2D(128, (3, 3), padding='same')(x)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)
    encoded = MaxPooling2D((2, 2), padding='same')(x)  # 4x4x128
```

There is no explicit encoding_dim as this convolutional autoencoder encodes images into a smaller spatial resolution with a deeper feature space

Extract spatial features using 3x3 filters

Normalizes layer outputs to stabilize and speed up training

Applies the ReLU activation function for non-linearity

Downsample the feature maps to reduce spatial dimensions (e.g., 32x32 → 16x16 → 8x8 → 4x4)

The encoder ends with a bottleneck representation of size 4x4x128, capturing the compressed feature space.

Autoencoders - Convolutional

Load and Preprocess the Data → Model Building → Model Training → Model Evaluation → Visualization

```
# Decoder
x = Conv2D(128, (3, 3), padding='same')(encoded)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = UpSampling2D((2, 2))(x)  # 8x8x128
x = Conv2D(64, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = UpSampling2D((2, 2))(x)  # 16x16x64
x = Conv2D(32, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = UpSampling2D((2, 2))(x)  # 32x32x32

decoded = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)  # 32x32x3
```

Reconstruct the image feature maps from the encoded representation

Gradually increase spatial dimensions (e.g., 4x4 → 8x8 → 16x16 → 32x32)

The final layer outputs an image with 3 channels (RGB) and pixel values constrained between 0 and 1 (using the sigmoid activation)

Autoencoders - Convolutional

Load and Preprocess the Data → Model Building → Model Training → Model Evaluation → Visualization

```
# Create the autoencoder model
autoencoder = Model(input_img, decoded)
# Create the encoder model
encoder = Model(input_img, encoded)
# Create the decoder model
encoded_input = Input(shape=(4, 4, 128))
# Get the decoder layers from the autoencoder
decoder_layers = autoencoder.layers[-13:]
# Build the decoder model
x = encoded_input
for layer in decoder_layers:
    x = layer(x)
decoder = Model(encoded_input, x)

return autoencoder, encoder, decoder
```

Combines the encoder and decoder into a complete autoencoder model that maps input images to reconstructed images

Extracts only the encoder portion of the model, allowing you to encode input images into the bottleneck feature space

Reuses the last 13 layers of the autoencoder

Iteratively reconstructs the image using the decoder layers, starting from

Autoencoders

Load and Preprocess the Data → Model Building → Model Training → Model Evaluation → Visualization

```
def train_autoencoder(autoencoder, x_train_noisy, x_train, x_test_noisy, x_test, batch_size=128, epochs=50):
    # Compile the model
    autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
    # Use early stopping to prevent overfitting
    early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
    # Train the model
    history = autoencoder.fit(
        x_train_noisy, x_train,
        epochs=epochs,
        batch_size=batch_size,
        shuffle=True,
        validation_data=(x_test_noisy, x_test),
        callbacks=[early_stopping]
    )
    return history
```

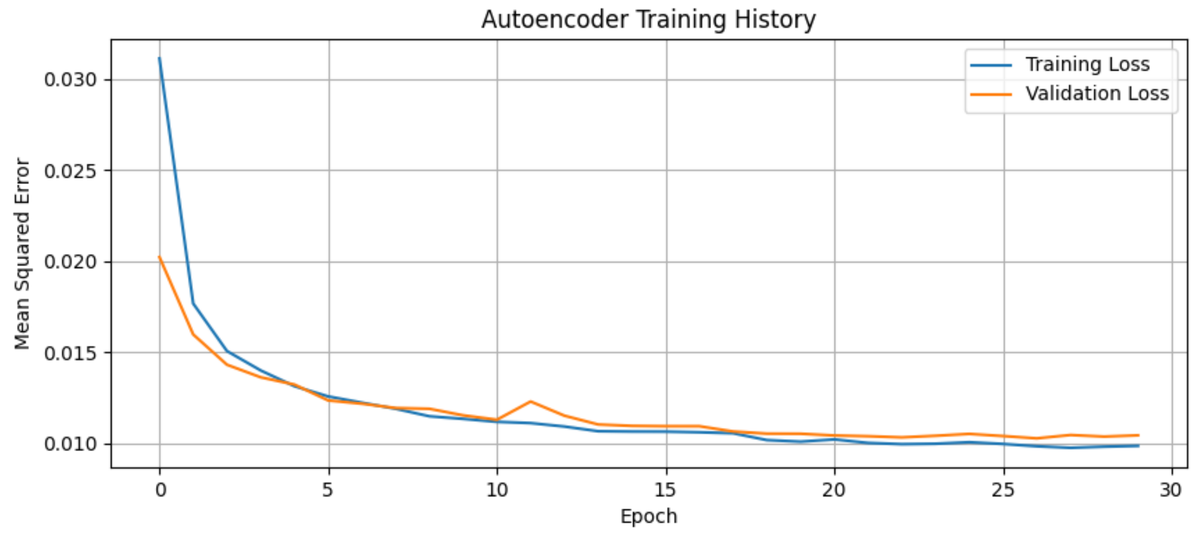
This function, train_autoencoder, is designed to train an autoencoder using a noisy dataset (x_train_noisy, x_test_noisy) as input and the corresponding clean dataset (x_train, x_test) as output. Here's a detailed explanation

Autoencoders

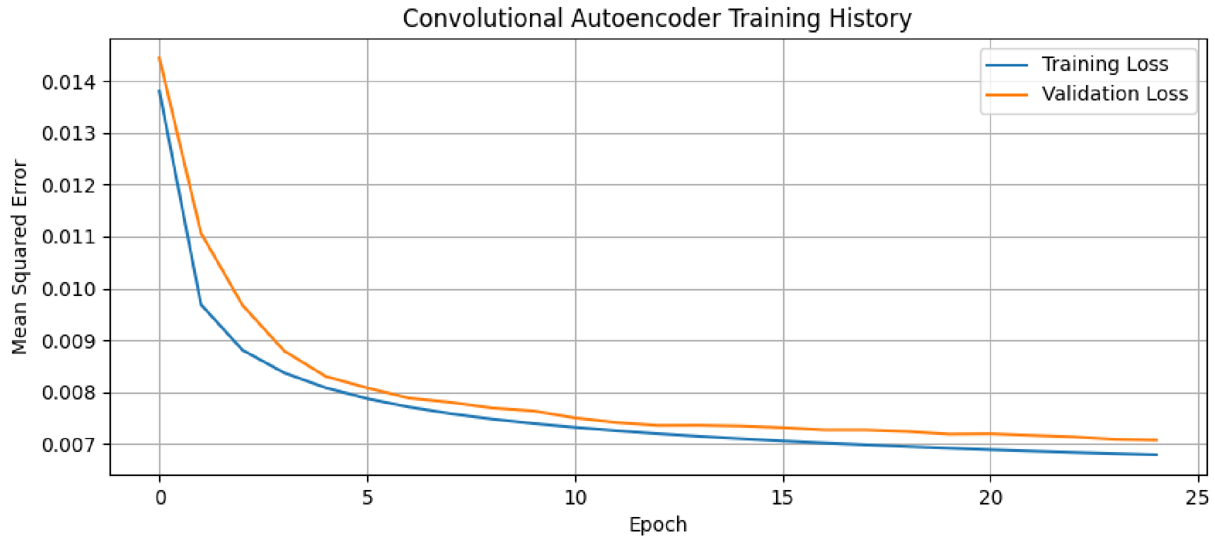
Load and Preprocess the Data → Model Building → Model Training → **Model Evaluation** → Visualization

```
def evaluate_autoencoder(autoencoder, x_test_noisy, x_test):
    test_loss = autoencoder.evaluate(x_test_noisy, x_test)
    print(f"Test loss (MSE): {test_loss}")
    return test_loss
```

This evaluate_autoencoder function is designed to evaluate the performance of an autoencoder model using noisy test data (x_test_noisy) as input and clean test data (x_test) as the target. Here's a detailed explanation



MLP

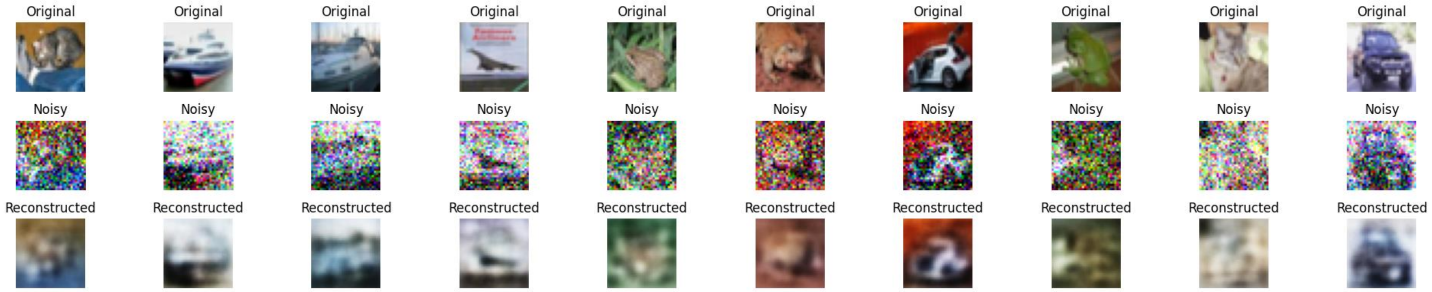


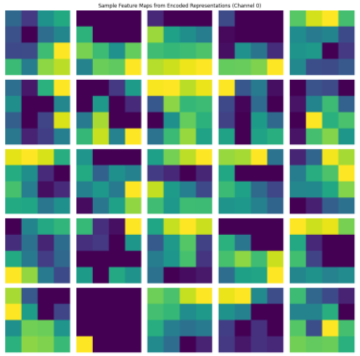
Convolution

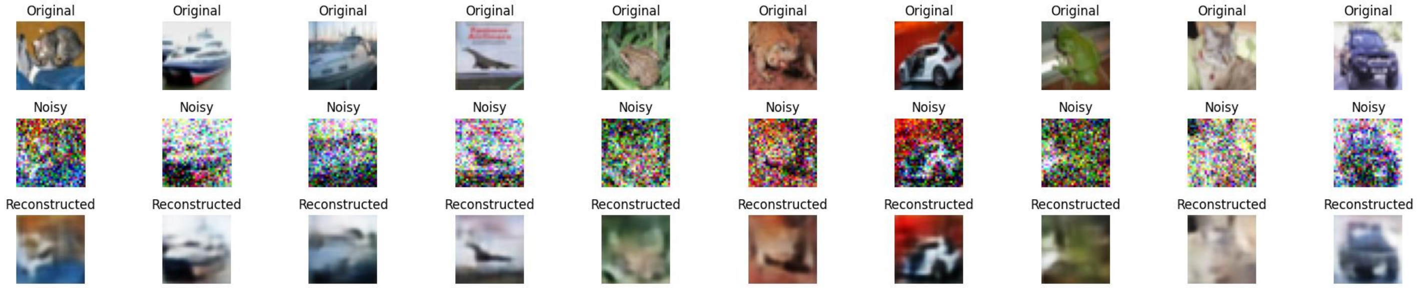
Autoencoders

Load and Preprocess the Data → Model Building → Model Training → **Model Evaluation** → Visualization

```
def plot_reconstructed_images(autoencoder, x_test_noisy, x_test, n=10):
    reconstructed_imgs = autoencoder.predict(x_test_noisy[:n])
    plt.figure(figsize=(20, 4))
    for i in range(n):
        # Original image
        ax = plt.subplot(3, n, i + 1)
        plt.imshow(x_test[i])
        plt.title("Original")
        plt.axis("off")
        # Noisy image
        ax = plt.subplot(3, n, i + n + 1)
        plt.imshow(x_test_noisy[i])
        plt.title("Noisy")
        plt.axis("off")
        # Reconstructed image
        ax = plt.subplot(3, n, i + 2*n + 1)
        plt.imshow(reconstructed_imgs[i])
        plt.title("Reconstructed")
        plt.axis("off")
    plt.tight_layout()
    plt.show()
```







Week 10 Exercise

1. Autoencoder-based Image Reconstruction using STL-10 Image Recognition Dataset

Objective: Explore the use of dense and convolutional autoencoders for image denoising, compression, and reconstruction tasks using the STL-10 dataset.

Dataset: STL-10 is an image recognition dataset inspired by CIFAR-10 dataset with some improvements. (https://www.kaggle.com/datasets/jessicali9530/stl10)

Tasks:

1. Data Loading, Preprocessing, and Noise Addition
2. Building Dense and Convolutional AEs
3. Train the Models separately
4. Evaluate the Models
5. Reconstruct Images and Visualize with Noise and Clean Inputs

Experiment with different hyperparameters and model architectures to cut down the reconstruction loss.