```python
#EXP 1: Analyze the impact of various activation functions on neural
networkperformance in a regression task using a house price prediction
dataset.
# The objective is to assess how each activation function affects
training and validation loss (MSE) to determine the most suitable
function for modeling non-linear data.

# STEP 1: Imports
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.datasets import fetch_california_housing  # Import the
dataset
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import warnings

# Suppress TensorFlow warnings
warnings.filterwarnings('ignore', category=FutureWarning)
pd.options.mode.chained_assignment = None

print("--- Imports Complete ---")

# STEP 2: Load Dataset
# This dataset is built into scikit-learn and will be downloaded
automatically.
housing = fetch_california_housing()
print(f"--- Dataset Loaded: {housing.DESCR.splitlines()[0]} ---")

# STEP 3: Feature + Target Setup
# Create DataFrame for features
X = pd.DataFrame(housing.data, columns=housing.feature_names)
# Get the target variable (Median House Value)
y = housing.target

# Normalize the target variable (y) to be between 0 and 1
y_scaler = MinMaxScaler()
y = y_scaler.fit_transform(y.reshape(-1, 1)).ravel()

# STEP 4: Preprocessing
# The California Housing dataset is all-numeric, so we only need
StandardScaler.
scaler = StandardScaler()
X_processed = scaler.fit_transform(X)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X_processed, y,
```

```python
                                                    test_size=0.2,
                                                    random_state=42)
print(f"--- Data Processed: Training shape {X_train.shape} ---")


# STEP 5: Train Model Function (Optimized for Speed)
def train_model(activation):
    """
    Builds, compiles, and trains a deep neural network for regression.
    """
    model = Sequential()
    model.add(Dense(128, input_dim=X_train.shape[1],
                    activation=activation))
    model.add(Dense(64, activation=activation))
    model.add(Dense(32, activation=activation))

    # Output layer for regression (linear)
    model.add(Dense(1))

    model.compile(optimizer='adam', loss='mean_squared_error')

    start = time.time()

    # --- OPTIMIZATIONS ---
    # Reduced epochs from 100 to 50
    # Increased batch_size from 16 to 32
    # This combination trains much faster.
    model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=0)
    # ---------------------

    duration = time.time() - start

    # Evaluate on the test set
    loss = model.evaluate(X_test, y_test, verbose=0)

    return duration, loss

# STEP 6: Run 3 trials per activation and average results
activations = ['relu', 'tanh', 'sigmoid']
results = {}

print("\n--- Starting Model Training (3 trials per activation) ---")

for act in activations:
    print(f"Testing {act.upper()}...")
    total_time, total_loss = 0, 0
    for i in range(3): # Run multiple trials
        t, l = train_model(act)
        total_time += t
        total_loss += l
```

```
        print(f"  Trial {i+1}/3: MSE={l:.4f}, Time={t:.2f}s")

    results[act] = {
        'avg_time': total_time / 3,
        'avg_mse': total_loss / 3
    }

# STEP 7: Print Results
print("\n--- Activation Function Performance (Averaged) ---")
print("=" * 50)
for act in results:
    print(f"{act.upper():<8} | Avg. Time: {results[act]
['avg_time']:>6.2f}s | Avg. MSE: {results[act]['avg_mse']:.4f}")
print("=" * 50)


# STEP 8: Plot Bar Chart
labels = list(results.keys())
mse_vals = [results[k]['avg_mse'] for k in labels]

plt.figure(figsize=(8,6))
bars = plt.bar(labels, mse_vals, color=['skyblue', 'skyblue',
'skyblue'])

# Highlight the best-performing (lowest MSE) activation
min_mse = min(mse_vals)
best_act_index = mse_vals.index(min_mse)
bars[best_act_index].set_color('orange')

plt.title("Average MSE for Different Activation Functions (50
Epochs)")
plt.ylabel("Mean Squared Error (MSE) - Lower is Better")
plt.xlabel("Activation Function")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

#EXP 2: To train and evaluate a single-layer feedforward neural
network on a real-world binary classification dataset using Stochastic
Gradient Descent (SGD) and Momentum-based Gradient Descent (Momentum
GD) as optimization techniques.
# The objective is to compare and analyze the performance of both
optimizers in terms of:
#· Convergence Rate: How quickly the training loss decreases over
epochs.
#· Training Speed: The computational efficiency and time taken during
training.
#· Classification Accuracy: The predictive performance on unseen test
data.
#This study aims to highlight the impact of optimization strategy on
neural network training effectiveness, particularly in low-complexity
```

```
models such as single-layer networks.


import pandas as pd
from sklearn.preproce ssing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_breast_cancer  # <-- IMPORT BUILT-IN
DATASET
import time
import numpy as np
import matplotlib.pyplot as plt

# --- STEP 1, 2, 3: Load and Prepare Data ---
# Load the built-in breast cancer dataset
print("--- Loading Breast Cancer Dataset ---")
data = load_breast_cancer()
X = data.data
y = data.target  # 0 = malignant, 1 = benign

print(f"Features shape: {X.shape}, Target shape: {y.shape}")

# --- STEP 4: Normalize the features ---
scaler = StandardScaler()
X = scaler.fit_transform(X)

# --- STEP 5: Split the data ---
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
print(f"Training samples: {X_train.shape[0]}, Test samples:
{X_test.shape[0]}")

# --- Helper Functions ---

# Sigmoid activation function
def sigmoid(z):
    # Clip z to prevent overflow in np.exp
    z = np.clip(z, -500, 500)
    return 1 / (1 + np.exp(-z))

# Binary cross-entropy loss function
def binary_cross_entropy(y_true, y_pred):
    epsilon = 1e-15  # Use a smaller epsilon for more stability
    # Clip predictions to prevent log(0)
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1
- y_pred))

# Initialize weights and bias
def initialize_weights(n_features):
```

```python
    W = np.random.randn(n_features, 1) * 0.01
    b = 0.0
    return W, b

# --- OPTIMIZED Training function (Batch Gradient Descent) ---
def train(X, y, optimizer='sgd', lr=0.01, epochs=100, beta=0.9):
    n_samples, n_features = X.shape
    W, b = initialize_weights(n_features)

    # Reshape y to (n_samples, 1) for matrix operations
    y = y.reshape(-1, 1)

    loss_history = []
    velocity_W = np.zeros_like(W)
    velocity_b = 0.0

    start_time = time.time()

    for epoch in range(epochs):
        # --- BATCH FORWARD PASS ---
        # Calculate predictions for all samples at once
        z = np.dot(X, W) + b
        a = sigmoid(z)  # 'a' is (n_samples, 1)

        # --- BATCH LOSS ---
        loss = binary_cross_entropy(y, a)
        loss_history.append(loss)

        # --- BATCH BACKWARD PASS (GRADIENTS) ---
        # Calculate gradients for all samples at once
        dz = a - y  # (n_samples, 1)
        dW = (1 / n_samples) * np.dot(X.T, dz)  # (n_features, 1)
        db = (1 / n_samples) * np.sum(dz)      # scalar

        # --- PARAMETER UPDATE ---
        if optimizer == 'sgd':
            W -= lr * dW
            b -= lr * db

        elif optimizer == 'momentum':
            # Classic Polyak Momentum
            velocity_W = (beta * velocity_W) + dW
            velocity_b = (beta * velocity_b) + db

            W -= lr * velocity_W
            b -= lr * velocity_b

    training_time = time.time() - start_time
    print(f"Trained {optimizer.upper()} for {epochs} epochs in
{training_time:.4f}s")
```

```python
    return W, b, loss_history, training_time

# Prediction function
def predict(X, W, b):
    z = np.dot(X, W) + b
    a = sigmoid(z)
    return (a > 0.5).astype(int)

# --- Run Experiment ---
LR = 0.01
EPOCHS = 200  # Increased epochs as BGD is fast

# Train using SGD
W_sgd, b_sgd, loss_sgd, time_sgd = train(X_train, y_train,
optimizer='sgd', lr=LR, epochs=EPOCHS)

# Train using Momentum GD
W_mom, b_mom, loss_mom, time_mom = train(X_train, y_train,
optimizer='momentum', lr=LR, epochs=EPOCHS, beta=0.9)

# --- Evaluate ---
# Predictions
y_pred_sgd = predict(X_test, W_sgd, b_sgd)
y_pred_mom = predict(X_test, W_mom, b_mom)

# Accuracy
acc_sgd = accuracy_score(y_test, y_pred_sgd)
acc_mom = accuracy_score(y_test, y_pred_mom)

# --- Output Results ---
print("\n--- Final Results ---")
print("Sigmoid Activation Function is being used")
print(f"SGD:      Accuracy: {acc_sgd*100:.2f}% | Training Time:
{time_sgd:.4f} sec | Final Loss: {loss_sgd[-1]:.4f}")
print(f"Momentum: Accuracy: {acc_mom*100:.2f}% | Training Time:
{time_mom:.4f} sec | Final Loss: {loss_mom[-1]:.4f}")

# Plot loss
plt.figure(figsize=(10, 6))
plt.plot(loss_sgd, label='SGD Loss')
plt.plot(loss_mom, label='Momentum Loss (beta=0.9)', linestyle='--')
plt.xlabel('Epochs')
plt.ylabel('Loss (Binary Cross-Entropy)')
plt.title('Loss Convergence Comparison')
plt.legend()
plt.grid(True)
plt.show()

# --- Nuanced Conclusion ---
print("\n--- Analysis ---")
```

```python
print(f"Convergence: Momentum's final loss ({loss_mom[-1]:.4f}) is likely lower than SGD's ({loss_sgd[-1]:.4f}).")
print(f"Speed:       Training times are similar ({time_mom:.4f}s vs {time_sgd:.4f}s) because the extra step is just simple addition.")
print(f"Accuracy:    Momentum achieved {acc_mom*100:.2f}%, SGD achieved {acc_sgd*100:.2f}%.")

if acc_mom > acc_sgd and loss_mom[-1] < loss_sgd[-1]:
    print("\nConclusion: Momentum GD was the clear winner, achieving a lower loss and higher accuracy.")
else:
    print("\nConclusion: The results are mixed, but Momentum generally helps converge to a better minimum.")

#EXP 3: Implement and compare three advanced gradient descent optimization algorithms: Nesterov Gradient Descent, Adagrad, RMSprop and Adam—in training neural networks.
# Design and implement a neural network to classify handwritten digits from the MNIST dataset using three advanced gradient descent optimization algorithms:
#1. Nesterov Accelerated Gradient (NAG)
#2. Adagrad
#3. RMSprop
#4. Adam

import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import SGD, Adagrad, Adam, RMSprop
from tensorflow.keras.utils import to_categorical
import warnings

# Suppress warnings
warnings.filterwarnings('ignore', category=FutureWarning)
print("--- Imports Complete ---")

# --- STEP 1: Load MNIST Dataset ---
# Keras provides MNIST as a built-in dataset
# We use the standard 'test' set as our validation data
(X_train, y_train), (X_val, y_val) = mnist.load_data()
print(f"Loaded MNIST data: {X_train.shape[0]} train samples, {X_val.shape[0]} validation samples")

# --- STEP 2: Preprocess the Data ---

# Normalize pixel values from [0, 255] to [0.0, 1.0]
X_train = X_train.astype('float32') / 255.0
X_val = X_val.astype('float32') / 255.0
```

```python
# One-hot encode the labels (e.g., 7 -> [0,0,0,0,0,0,0,1,0,0])
y_train_cat = to_categorical(y_train, 10)
y_val_cat = to_categorical(y_val, 10)
print(f"Data processed. X_train shape: {X_train.shape}, y_train_cat
shape: {y_train_cat.shape}")

# --- STEP 3: Define Model Creation Function ---
def create_model():
    model = Sequential()
    # Add a Flatten layer to convert 28x28 images to 784-element
vectors
    model.add(Flatten(input_shape=(28, 28)))

    # Hidden layers
    model.add(Dense(64, activation='relu'))
    model.add(Dense(32, activation='relu'))

    # Output layer: 10 neurons (one for each digit) with softmax
    model.add(Dense(10, activation='softmax'))
    return model

# --- STEP 4: Define Optimizers ---
results = {}
optimizers = {
    # Nesterov Accelerated Gradient (NAG)
    "NAG": SGD(learning_rate=0.01, momentum=0.9, nesterov=True),

    # Adagrad (Adaptive Gradient)
    "Adagrad": Adagrad(learning_rate=0.01),

    # RMSprop
    "RMSprop": RMSprop(learning_rate=0.01),

    # Adam (Adaptive Moment Estimation)
    "Adam": Adam(learning_rate=0.01)
}

# --- STEP 5: Run Training Loop ---
EPOCHS = 10  # Reduced for speed. 10 is enough to see the difference.
BATCH_SIZE = 128

for name, opt in optimizers.items():
    print(f"\n--- Training with {name} optimizer ---")
    model = create_model()

    # Compile the model with categorical_crossentropy
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
```

```
    # Train the model
    history = model.fit(
        X_train, y_train_cat,
        validation_data=(X_val, y_val_cat),
        epochs=EPOCHS,
        batch_size=BATCH_SIZE,
        verbose=1  # Set to 1 to see epoch progress
    )
    results[name] = history

# --- STEP 6: Plot Results ---
def plot_results(metric):
    plt.figure(figsize=(12, 8))
    for name, history in results.items():
        # Plot training metric
        plt.plot(history.history[metric], label=f'{name} Train', lw=2)
        # Plot validation metric
        plt.plot(history.history['val_' + metric], linestyle='--',
label=f'{name} Val')

    plt.title(f'Optimizer Comparison: {metric.capitalize()}',
fontsize=16)
    plt.xlabel('Epochs', fontsize=12)
    plt.ylabel(metric.capitalize(), fontsize=12)
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot both loss and accuracy
print("\n--- Generating Plots ---")
plot_results('loss')
plot_results('accuracy')

#EXP 4: Design and implement a Convolutional Neural Network (CNN) to
classify handwritten digits from the MNIST dataset.
#Evaluate the model performance and demonstrate how convolutional
layers improve image classification accuracy compared to traditional
dense networks.
#Objective:
#• To understand and apply CNN architecture for image classification.
#• To evaluate the impact of convolutional layers, pooling, and
activation functions.
#• To compare CNN with a basic fully connected neural network.

import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Conv2D,
MaxPooling2D
```

```
import matplotlib.pyplot as plt
import numpy as np
import warnings

# Suppress warnings
warnings.filterwarnings('ignore', category=FutureWarning)
print("--- Imports Complete ---")

# --- STEP 1: Load and Preprocess MNIST Data ---
(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(f"Data loaded: {x_train.shape[0]} train samples,
{x_test.shape[0]} test samples")

# Normalize pixel values from [0, 255] to [0.0, 1.0]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Reshape data for the CNN (add a channel dimension: 1 for grayscale)
# Shape goes from (60000, 28, 28) to (60000, 28, 28, 1)
x_train_cnn = np.expand_dims(x_train, -1)
x_test_cnn = np.expand_dims(x_test, -1)

# Note: We will use 'sparse_categorical_crossentropy' as the loss
function.
# This means we can keep our labels (y_train, y_test) as simple
integers (0-9)
# and do NOT need to one-hot encode them.
print(f"Shape for Dense model: {x_train.shape} (uses Flatten layer)")
print(f"Shape for CNN model:   {x_train_cnn.shape} (needs channel
dimension)")


# --- STEP 2: Define and Train Baseline Dense Model (Objective 3) ---

def create_dense_model():
    model = Sequential([
        # Flatten the 28x28 image into a 784-element vector
        Flatten(input_shape=(28, 28)),

        # Fully connected layers
        Dense(128, activation='relu'),
        Dense(64, activation='relu'),

        # Output layer (10 classes for 10 digits)
        Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
```

```
    return model

print("\n--- Training Basic Dense Network ---")
dense_model = create_dense_model()
# The Dense model trains on the original (28, 28) data
history_dense = dense_model.fit(x_train, y_train, epochs=5,
                                validation_data=(x_test, y_test),
                                verbose=1)
dense_loss, dense_acc = dense_model.evaluate(x_test, y_test,
verbose=0)


# --- STEP 3: Define, Compile, and Train CNN Model (Objectives 1 & 2)
---

def create_cnn_model():
    model = Sequential([
        # First Convolutional Layer
        Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
1)),
        MaxPooling2D((2, 2)),

        # Second Convolutional Layer
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),

        # Third Convolutional Layer (as in your snippet)
        Conv2D(64, (3, 3), activation='relu'),

        # Flatten the results to feed into a Dense layer
        Flatten(),

        # Fully connected layer
        Dense(64, activation='relu'),

        # Output layer (10 classes)
        Dense(10, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model

print("\n--- Training Convolutional Neural Network (CNN) ---")
cnn_model = create_cnn_model()
# The CNN trains on the reshaped (28, 28, 1) data
history_cnn = cnn_model.fit(x_train_cnn, y_train, epochs=5,
                            validation_data=(x_test_cnn, y_test),
                            verbose=1)
```

```
cnn_loss, cnn_acc = cnn_model.evaluate(x_test_cnn, y_test, verbose=0)


# --- STEP 4: Compare Performance (Objective 3) ---

print("\n--- Model Performance Comparison ---")
print(f"Basic Dense Network Test Accuracy: {dense_acc * 100:.2f}%")
print(f"CNN Test Accuracy:                 {cnn_acc * 100:.2f}%")

print("\n--- Analysis ---")
print("The CNN is significantly more accurate. This is because the
Conv2D layers learn")
print("spatial features (like edges, curves, and patterns) directly
from the image.")
print("The Dense network, after flattening, loses all spatial
information and only sees")
print("a long list of pixels, making it much harder to learn image-
specific patterns.")


# --- STEP 5: Plot Comparison Curves ---

plt.figure(figsize=(12, 6))

# Plot training & validation accuracy
plt.subplot(1, 2, 1)
plt.plot(history_dense.history['val_accuracy'], label='Dense Val
Accuracy', linestyle='--')
plt.plot(history_cnn.history['val_accuracy'], label='CNN Val
Accuracy', lw=2)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
plt.title('Model Accuracy Comparison')

# Plot training & validation loss
plt.subplot(1, 2, 2)
plt.plot(history_dense.history['val_loss'], label='Dense Val Loss',
linestyle='--')
plt.plot(history_cnn.history['val_loss'], label='CNN Val Loss', lw=2)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.title('Model Loss Comparison')

plt.tight_layout()
plt.show()
```

```
#EXP 5: A. Image Classification Using Le Netl Neural Network (LE Net)
#B. Image Classification Using AlexNet Neural Network
#C. ResNet, DenseNet, and EfficientNet are all advanced convolutional
neural network (CNN) architectures
#D. Compare Le-net.Alex-net, ResNet, DenseNet, and EfficientNet



# A . Le Netl
# Step 1: Import Libraries
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Step 2: Load and Preprocess Dataset
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
[cite: 3]

x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train[..., tf.newaxis] [cite: 3]
x_test = x_test[..., tf.newaxis] [cite: 3]

# Resize MNIST to 32x32 for LeNet-5
x_train = tf.image.resize(x_train, [32, 32]) [cite: 3]
x_test = tf.image.resize(x_test, [32, 32]) [cite: 3]

# Step 3: Build LeNet-5 Model with ReLU
lenet5_relu = models.Sequential([ [cite: 3]
    layers.Conv2D(6, kernel_size=(5,5), activation='relu',
input_shape=(32,32,1), padding='same'), [cite: 3]
    layers.AveragePooling2D(pool_size=(2,2), strides=2), [cite: 4]
    layers.Conv2D(16, kernel_size=(5,5), activation='relu'), [cite: 4]
    layers.AveragePooling2D(pool_size=(2,2), strides=2), [cite: 4]
    layers.Flatten(), [cite: 4]
    layers.Dense(120, activation='relu'), [cite: 4]
    layers.Dense(84, activation='relu'), [cite: 4]
    layers.Dense(10, activation='softmax') [cite: 4]
])

# Step 4: Compile the Model
lenet5_relu.compile(optimizer='adam', [cite: 4]
              loss='sparse_categorical_crossentropy', [cite: 4]
              metrics=['accuracy']) [cite: 4]

# Step 5: Train the Model
history = lenet5_relu.fit(x_train, y_train, epochs=10, [cite: 4]
                    validation_data=(x_test, y_test)) [cite: 4]

# Step 6: Evaluate the Model
test_loss, test_acc = lenet5_relu.evaluate(x_test, y_test) [cite: 4]
print(f"Test Accuracy: {test_acc:.4f}") [cite: 4]
```

```
# Step 7: Plot Accuracy
plt.plot(history.history['accuracy'], label='Train Acc') [cite: 4]
plt.plot(history.history['val_accuracy'], label='Val Acc') [cite: 4]
plt.xlabel('Epoch') [cite: 4]
plt.ylabel('Accuracy') [cite: 4]
plt.legend() [cite: 4]
plt.show() [cite: 4]


# 2.Alexnet
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Step 1: Load Dataset
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
[cite: 4]

# Normalize images to [0,1]
x_train, x_test = x_train / 255.0, x_test / 255.0 [cite: 4]

# Add channel dimension
x_train = x_train[..., tf.newaxis] [cite: 5]
x_test = x_test[..., tf.newaxis] [cite: 5]

# Reduce dataset size (use only first 20,000 train, 5,000 test
samples)
x_train, y_train = x_train[:20000], y_train[:20000] [cite: 5]
x_test, y_test = x_test[:5000], y_test[:5000] [cite: 5]

# Resize MNIST to 128x128 instead of 227x227
x_train = tf.image.resize(x_train, [128, 128]) [cite: 5]
x_test = tf.image.resize(x_test, [128, 128]) [cite: 5]

# Step 2: Build Smaller AlexNet
alexnet = models.Sequential([ [cite: 5]
    layers.Conv2D(96, kernel_size=(11,11), strides=4,
activation='relu', input_shape=(128,128,1)), [cite: 5]
    layers.MaxPooling2D(pool_size=(3,3), strides=2), [cite: 5]
    layers.Conv2D(256, kernel_size=(5,5), padding='same',
activation='relu'), [cite: 5]
    layers.MaxPooling2D(pool_size=(3,3), strides=2), [cite: 5]
    layers.Conv2D(384, kernel_size=(3,3), padding='same',
activation='relu'), [cite: 5]
    layers.Conv2D(384, kernel_size=(3,3), padding='same',
activation='relu'), [cite: 5]
    layers.Conv2D(256, kernel_size=(3,3), padding='same',
activation='relu'), [cite: 5]
    layers.MaxPooling2D(pool_size=(3,3), strides=2), [cite: 5]
```

```
    layers.Flatten(), [cite: 5]
    layers.Dense(1024, activation='relu'), # reduced from 4096
    layers.Dropout(0.5), [cite: 5]
    layers.Dense(512, activation='relu'),  # reduced from 4096
    layers.Dropout(0.5), [cite: 5]
    layers.Dense(10, activation='softmax') # MNIST = 10 classes
])

# Step 3: Compile Model
alexnet.compile(optimizer='adam', [cite: 5]
              loss='sparse_categorical_crossentropy', [cite: 5]
              metrics=['accuracy']) [cite: 5]

# Step 4: Train (only 2 epochs)
history = alexnet.fit(x_train, y_train, epochs=2,
validation_data=(x_test, y_test)) [cite: 5]

# Step 5: Evaluate
test_loss, test_acc = alexnet.evaluate(x_test, y_test) [cite: 6]
print(f"Test Accuracy: {test_acc:.4f}") [cite: 6]

# Step 6: Plot
plt.plot(history.history['accuracy'], label='Train Acc') [cite: 6]
plt.plot(history.history['val_accuracy'], label='Val Acc') [cite: 6]
plt.xlabel('Epoch') [cite: 6]
plt.ylabel('Accuracy') [cite: 6]
plt.legend() [cite: 6]
plt.show() [cite: 6]


# 3.Resnet
# Step 1: Import Libraries
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Step 2: Load and Preprocess Dataset
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
[cite: 6]

x_train, x_test = x_train / 255.0, x_test / 255.0 [cite: 6]
x_train = x_train[..., tf.newaxis] [cite: 6]
x_test = x_test[..., tf.newaxis] [cite: 6]

# Resize MNIST to 32x32 (ResNet expects bigger input)
x_train = tf.image.resize(x_train, [32, 32]) [cite: 6]
x_test = tf.image.resize(x_test, [32, 32]) [cite: 6]

# Step 3: Define Residual Block
def residual_block(x, filters, downsample=False): [cite: 6]
    shortcut = x [cite: 6]
```

```
    strides = 2 if downsample else 1 [cite: 6]

    # First conv
    x = layers.Conv2D(filters, (3,3), strides=strides, padding="same",
activation='relu')(x) [cite: 6]
    # Second conv
    x = layers.Conv2D(filters, (3,3), strides=1, padding="same")(x)
[cite: 6]

    # Adjust shortcut if shape mismatch
    if downsample or shortcut.shape[-1] != filters: [cite: 6]
        shortcut = layers.Conv2D(filters, (1,1), strides=strides,
padding="same")(shortcut) [cite: 7]

    # Add skip connection
    x = layers.Add()([x, shortcut]) [cite: 7]
    x = layers.Activation('relu')(x) [cite: 7]
    return x [cite: 7]

# Step 4: Build ResNet Model
inputs = layers.Input(shape=(32,32,1)) [cite: 7]

# Initial conv
x = layers.Conv2D(16, (3,3), strides=1, padding="same",
activation='relu')(inputs) [cite: 7]

# Residual blocks
x = residual_block(x, 16) [cite: 7]
x = residual_block(x, 16) [cite: 7]
x = residual_block(x, 32, downsample=True) [cite: 7]
x = residual_block(x, 32) [cite: 7]
x = residual_block(x, 64, downsample=True) [cite: 7]
x = residual_block(x, 64) [cite: 7]

# Global average pooling and output
x = layers.GlobalAveragePooling2D()(x) [cite: 7]
outputs = layers.Dense(10, activation='softmax')(x) [cite: 7]
resnet_model = models.Model(inputs, outputs) [cite: 7]

# Step 5: Compile the Model
resnet_model.compile(optimizer='adam', [cite: 7]
                   loss='sparse_categorical_crossentropy', [cite: 7]
                   metrics=['accuracy']) [cite: 7]

# Step 6: Train the Model
history = resnet_model.fit(x_train, y_train, epochs=10, [cite: 7]
                     validation_data=(x_test, y_test)) [cite: 7]

# Step 7: Evaluate the Model
test_loss, test_acc = resnet_model.evaluate(x_test, y_test) [cite: 7]
```

```
print(f"Test Accuracy: {test_acc:.4f}") [cite: 7]

# Step 8: Plot Accuracy
plt.plot(history.history['accuracy'], label='Train Acc') [cite: 8]
plt.plot(history.history['val_accuracy'], label='Val Acc') [cite: 8]
plt.xlabel('Epochs') [cite: 8]
plt.ylabel('Accuracy') [cite: 8]
plt.legend() [cite: 8]
plt.show() [cite: 8]



# 4.Dense net
# densenet_mnist.py
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt

# Dense Block
def dense_block(x, num_convs, growth_rate): [cite: 8]
    for _ in range(num_convs): [cite: 8]
        out = layers.BatchNormalization()(x) [cite: 8]
        out = layers.ReLU()(out) [cite: 8]
        out = layers.Conv2D(growth_rate, (3,3), padding="same")(out)
[cite: 8]
        x = layers.Concatenate()([x, out]) [cite: 8]
    return x [cite: 8]

# Transition Layer
def transition_layer(x, reduction): [cite: 8]
    out = layers.BatchNormalization()(x) [cite: 8]
    out = layers.ReLU()(out) [cite: 8]
    out = layers.Conv2D(int(x.shape[-1]*reduction), (1,1))(out) [cite:
8]
    out = layers.AveragePooling2D(pool_size=(2,2), strides=2)(out)
[cite: 8]
    return out [cite: 8]

def build_densenet(input_shape=(32,32,1), num_classes=10): [cite: 8]
    inputs = layers.Input(shape=input_shape) [cite: 8]

    # Initial conv
    x = layers.Conv2D(64, (3,3), padding="same", activation="relu")
(inputs) [cite: 8]

    # Dense Block 1
    x = dense_block(x, num_convs=2, growth_rate=12) [cite: 8]
    x = transition_layer(x, reduction=0.5) [cite: 8]

    # Dense Block 2
```

```
    x = dense_block(x, num_convs=2, growth_rate=12) [cite: 8]
    x = transition_layer(x, reduction=0.5) [cite: 9]

    # Dense Block 3
    x = dense_block(x, num_convs=2, growth_rate=12) [cite: 9]

    # Classification Layer
    x = layers.BatchNormalization()(x) [cite: 9]
    x = layers.ReLU()(x) [cite: 9]
    x = layers.GlobalAveragePooling2D()(x) [cite: 9]
    outputs = layers.Dense(num_classes, activation="softmax")(x)
[cite: 9]

    model = models.Model(inputs, outputs) [cite: 9]
    return model [cite: 9]

def main(): [cite: 9]
    # Load Dataset
    (x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
[cite: 9]
    x_train, x_test = x_train / 255.0, x_test / 255.0 [cite: 9]
    x_train = x_train[..., tf.newaxis] [cite: 9]
    x_test = x_test[..., tf.newaxis] [cite: 9]

    # Resize MNIST to 32x32 for DenseNet
    x_train = tf.image.resize(x_train, [32, 32]) [cite: 9]
    x_test = tf.image.resize(x_test, [32, 32]) [cite: 9]

    # Build DenseNet
    densenet = build_densenet() [cite: 9]

    # Compile
    densenet.compile(optimizer='adam', [cite: 9]
                     loss='sparse_categorical_crossentropy', [cite: 9]
                     metrics=['accuracy']) [cite: 9]

    # Train
    history = densenet.fit(x_train, y_train, epochs=5,
validation_data=(x_test, y_test)) [cite: 9]

    # Evaluate
    test_loss, test_acc = densenet.evaluate(x_test, y_test) [cite: 9]
    print(f"\n □  DenseNet Test Accuracy: {test_acc:.4f}") [cite: 9]

    # Plot
    plt.plot(history.history['accuracy'], label='Train Acc') [cite: 9]
    plt.plot(history.history['val_accuracy'], label='Val Acc') [cite:
9]
    plt.title("DenseNet Training vs Validation Accuracy (MNIST)")
[cite: 9]
    plt.xlabel('Epochs') [cite: 10]
```

```
    plt.ylabel('Accuracy') [cite: 10]
    plt.legend() [cite: 10]
    plt.savefig("densenet_accuracy.png") [cite: 10]
    plt.show() [cite: 10]

if __name__ == "__main__": [cite: 10]
    main() [cite: 10]

#EXP 6:  Design and implement an Image classification model to
classify a dataset of images using Deep Feed Forward NN.
#Record the accuracy corresponding to the number of epochs. Use the
MNIST datasets.

# Deep Feed Forward Neural Network (DFFNN) for MNIST
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Flatten
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
import warnings

# Suppress warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# 1. Load MNIST dataset
(X_train, y_train), (X_test, y_test) =
tf.keras.datasets.mnist.load_data()
print("--- Data Loading ---")
print(f"Shape of X_train: {X_train.shape}")
print(f"Shape of y_train: {y_train.shape}")
print(f"Shape of X_test: {X_test.shape}")
print(f"Shape of y_test: {y_test.shape}")

# 2. Display first 10 images with labels
print("\n--- Displaying Sample Data ---")
fig, axs = plt.subplots(2, 5, figsize=(12, 6))
fig.suptitle('First 10 MNIST Images', fontsize=16)
n = 0
for i in range(2):
    for j in range(5):
        axs[i, j].imshow(X_train[n], cmap='gray')
        axs[i, j].set_title(f"Label: {y_train[n]}")
        axs[i, j].axis('off')
        n += 1
plt.show()

# 3. Reshape (Flatten) and Normalize
print("\n--- Data Preprocessing ---")
# DFFNN cannot read 2D images, so we flatten 28x28 images into 784-
```

```
element vectors
X_train = X_train.reshape(60000, 784).astype("float32") / 255
X_test = X_test.reshape(10000, 784).astype("float32") / 255
print(f"New shape of X_train (flattened): {X_train.shape}")
print(f"New shape of X_test (flattened): {X_test.shape}")

# 4. Define Deep Feed Forward Neural Network
model = Sequential(name="DFF-Model")
model.add(Input(shape=(784,), name='Input-Layer'))
model.add(Dense(128, activation='relu',
kernel_initializer='he_normal',
                name='Hidden-Layer-1'))
model.add(Dense(64, activation='relu', kernel_initializer='he_normal',
                name='Hidden-Layer-2'))
model.add(Dense(32, activation='relu', kernel_initializer='he_normal',
                name='Hidden-Layer-3'))
model.add(Dense(10, activation='softmax', name='Output-Layer')) # 10
classes (0-9)

# 5. Compile model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', # Use sparse
because y_train is integers (0,1,2...)
              metrics=['accuracy'])

# 6. Train model
print("\n--- Starting Model Training ---")
history = model.fit(X_train, y_train,
                    batch_size=64,
                    epochs=10, # We will get 10 accuracy points
                    validation_split=0.2, # Use 20% of training data
for validation
                    verbose=1)

# 7. Plot accuracy vs epochs (Objective)
print("\n--- Plotting Accuracy vs Epochs ---")
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Training Accuracy',
marker='o')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy',
marker='o')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epochs")
plt.legend()
plt.grid()
plt.show()

# 8. Predictions
# Use np.argmax to get the class index with the highest probability
```

```python
pred_labels_tr = np.argmax(model.predict(X_train), axis=1)
pred_labels_te = np.argmax(model.predict(X_test), axis=1)

# 9. Model Summary
print("\n--- Model Summary ---")
model.summary()

# 10. Classification Report
print("\n---------- Evaluation on Training Data -----------")
print(classification_report(y_train, pred_labels_tr))

print("\n---------- Evaluation on Test Data (Final) -----------")
print(classification_report(y_test, pred_labels_te))

#EXP 7: Implement RNN for sentiment analysis on movie reviews

# RNN Sentiment Analysis on IMDB Movie Reviews
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN, Embedding
import matplotlib.pyplot as plt
import warnings

# Suppress warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# 1. Load IMDB dataset (top 10,000 most frequent words)
num_words = 10000
(X_train, y_train), (X_test, y_test) =
imdb.load_data(num_words=num_words)
print("--- Data Loading ---")
print(f"Training samples: {len(X_train)}")
print(f"Test samples: {len(X_test)}")

# 2. Pad sequences to have equal length (e.g., 50 words per review)
# This ensures all input vectors have the same dimension.
maxlen = 50
X_train = pad_sequences(X_train, maxlen=maxlen, padding='post')
X_test = pad_sequences(X_test, maxlen=maxlen, padding='post')
print(f"Shape of X_train (padded): {X_train.shape}")
print(f"Shape of X_test (padded): {X_test.shape}")

# 3. Build RNN model
print("\n--- Building Model ---")
model = Sequential()
# Embedding layer: Turns word indices (e.g., 10) into dense vectors
(e.g., [0.1, 0.5, ...])
# 'num_words' = vocabulary size, 'output_dim' = vector size for each
```

```
word
model.add(Embedding(input_dim=num_words, output_dim=32,
                    input_length=maxlen))

# A SimpleRNN layer that processes the sequence of vectors
model.add(SimpleRNN(32, return_sequences=False)) # False = only return
the final output

# Output layer: Sigmoid for binary classification (positive/negative)
model.add(Dense(1, activation='sigmoid'))

model.summary()

# 4. Compile model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 5. Train the model
print("\n--- Starting Model Training ---")
history = model.fit(X_train, y_train,
                    epochs=5,
                    batch_size=128,
                    validation_data=(X_test, y_test),
                    verbose=1)

# 6. Evaluate model
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=0)
print("\n--- Evaluation Complete ---")
print(f"Test Loss: {test_loss:.4f}")
print(f"Test Accuracy: {test_acc * 100:.2f}%")

# 7. Plot accuracy vs epochs
print("\n--- Plotting Accuracy vs Epochs ---")
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label="Training Accuracy",
marker='o')
plt.plot(history.history['val_accuracy'], label="Validation Accuracy",
marker='o')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("RNN Accuracy on IMDB Sentiment Analysis")
plt.legend()
plt.grid()
plt.show()

#EXP 8: Create an RNN model that can classify the sentiment of tweets
in real time.

# -----------------------------
```

```python
# 1. Imports
# ------------------------------
!pip install -q tensorflow
import numpy as np
import matplotlib.pyplot as plt
import json, os
import tensorflow as tf
from tensorflow.keras.layers import Embedding, Bidirectional, LSTM,
Dense, Dropout, Input
from tensorflow.keras.models import Model, load_model
from tensorflow.keras.preprocessing.text import Tokenizer,
tokenizer_from_json
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import warnings

warnings.filterwarnings('ignore', category=FutureWarning)
print("--- Imports Complete ---")

# ------------------------------
# 2. Synthetic dataset (for demo)
# Using a tiny dataset for a quick (< 10 seconds) training example.
# ------------------------------
positive_examples = [
    "I love this!", "This is amazing", "What a fantastic day",
    "So happy with the results", "Great job", "Absolutely wonderful",
    "I enjoyed this a lot", "Highly recommend", "This made me smile"
]
negative_examples = [
    "I hate this", "This is terrible", "Worst experience ever",
    "So disappointed", "Very bad", "I will never use this again",
    "Horrible service", "This ruined my day", "Not recommended"
]

texts = positive_examples + negative_examples
labels = [1]*len(positive_examples) + [0]*len(negative_examples) #
1=Positive, 0=Negative

# Shuffle for training
rng = np.random.default_rng(seed=42)
idx = rng.permutation(len(texts))
texts = [texts[i] for i in idx]
labels = np.array([labels[i] for i in idx])

print(f"--- Dataset Created: {len(texts)} total samples ---")

# ------------------------------
# 3. Tokenization & padding
# ------------------------------
vocab_size = 5000      # Max words in our vocabulary
```

```
oov_token = "<OOV>"    # Token for words not in the vocabulary
max_len = 20           # Max length of a tweet/sentence

tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_token)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Pad all sequences to be the same length (max_len)
padded = pad_sequences(sequences, maxlen=max_len, padding="post",
truncating="post")

# Split into training and validation
split = int(0.8 * len(padded))
x_train, x_val = padded[:split], padded[split:]
y_train, y_val = labels[:split], labels[split:]

print(f"Training samples: {len(x_train)}, Validation samples:
{len(x_val)}")

# -----------------------------
# 4. Build Bidirectional LSTM model
# -----------------------------
embedding_dim = 64
lstm_units = 64
dropout_rate = 0.3

def build_model():
    inp = Input(shape=(max_len,), name='input_ids')
    # Embedding layer
    x = Embedding(vocab_size, embedding_dim, input_length=max_len)
(inp)
    # BiLSTM reads the sequence forwards and backwards
    x = Bidirectional(LSTM(lstm_units))(x)
    x = Dropout(dropout_rate)(x)
    # Output layer for binary classification
    out = Dense(1, activation="sigmoid")(x)

    model = Model(inputs=inp, outputs=out)
    model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])
    return model

model = build_model()
model.summary()

# -----------------------------
# 5. Train
# -----------------------------
print("\n--- Starting Model Training ---")
callbacks = [
```

```
    # Stop training if 'val_loss' doesn't improve for 3 epochs
    EarlyStopping(monitor="val_loss", patience=3,
restore_best_weights=True),
    # Save the best model found so far
    ModelCheckpoint("best_model.h5", save_best_only=True,
monitor="val_loss")
]

history = model.fit(
    x_train, y_train,
    validation_data=(x_val, y_val),
    epochs=12,
    batch_size=4,
    callbacks=callbacks,
    verbose=2
)

# ------------------------------
# 6. Plot training curves
# ------------------------------
print("\n--- Plotting Results ---")
plt.figure(figsize=(8,4))
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.title("Loss")
plt.xlabel("Epoch")
plt.legend()
plt.grid()
plt.show()

plt.figure(figsize=(8,4))
plt.plot(history.history['accuracy'], label='train_acc')
plt.plot(history.history['val_accuracy'], label='val_acc')
plt.title("Accuracy")
plt.xlabel("Epoch")
plt.legend()
plt.grid()
plt.show()

# ------------------------------
# 7. Save model & tokenizer for production
# ------------------------------
model.save("sentiment_rnn_model.keras")

# CRITICAL: Save the tokenizer so we can preprocess
# new tweets exactly the same way as the training data.
with open("tokenizer.json", "w") as f:
    f.write(tokenizer.to_json())

print(f"\nModel saved to 'sentiment_rnn_model.keras'")
```

```python
print(f"Tokenizer saved to 'tokenizer.json'")


# -----------------------------------------------------
# 8. "Real-time" prediction function
# This part simulates a separate application
# loading the saved files to make predictions.
# -----------------------------------------------------

print("\n--- Loading Model for 'Real-Time' Prediction ---")
# Load the trained model
loaded_model = load_model("sentiment_rnn_model.keras")

# Load and re-create the tokenizer
with open("tokenizer.json") as f:
    tok_json = f.read()
loaded_tokenizer = tokenizer_from_json(tok_json)

def predict_tweet_sentiment(text):
    # Preprocess the new text using the *loaded* tokenizer
    seq = loaded_tokenizer.texts_to_sequences([text])
    pad = pad_sequences(seq, maxlen=max_len, padding="post")

    # Predict
    prob = float(loaded_model.predict(pad, verbose=0)[0][0])

    # Determine label
    label = "positive" if prob >= 0.5 else "negative"
    return {"text": text, "probability_positive": prob, "label":
label}

# Test predictions
print("\n--- Testing 'Real-Time' Predictions ---")
sample_tweets = [
    "OMG this product is awesome, I'm so happy!",
    "Totally disappointed with the service today.",
    "Not sure how I feel about this.",
    "Best experience ever, thank you!",
    "This is the worst, will complain."
]

for t in sample_tweets:
    prediction = predict_tweet_sentiment(t)
    print(f"Prediction: {prediction['label']} (Prob:
{prediction['probability_positive']:.4f}) | Tweet:
{prediction['text']}")

#EXP 9: Implement Auto encoders for image denoising on MNIST, Fashion,
MNIST or any suitable dataset.
```

```
!pip install -q tensorflow matplotlib numpy
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models, losses, optimizers
from tensorflow.keras.datasets import mnist
import warnings

warnings.filterwarnings('ignore', category=FutureWarning)

# ---------------- Parameters ----------------
noise_factor = 0.5   # Amount of noise to add
batch_size = 128
epochs = 15          # 15 epochs is enough for a good result
num_display = 10     # How many images to display at the end
save_dir = "autoencoder_denoising_output"
os.makedirs(save_dir, exist_ok=True)
print("--- Parameters Set ---")

# ---------------- Load & Preprocess Data ----------------
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize pixel values to [0,1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Add channel dimension (N, 28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)

# Add Gaussian noise to inputs
print(f"--- Adding Noise (Factor: {noise_factor}) ---")
rng = np.random.RandomState(42)
x_train_noisy = x_train + noise_factor * rng.normal(loc=0.0,
scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * rng.normal(loc=0.0, scale=1.0,
size=x_test.shape)

# Clip to keep in [0,1] range
x_train_noisy = np.clip(x_train_noisy, 0.0, 1.0)
x_test_noisy = np.clip(x_test_noisy, 0.0, 1.0)

print("□ Data prepared")
print("Training input (noisy):", x_train_noisy.shape)
print("Training target (clean):", x_train.shape)

# ---------------- Build Model ----------------
print("\n--- Building Convolutional Autoencoder ---")
input_img = layers.Input(shape=(28, 28, 1))
```

```
# Encoder
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')
(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)  # Bottleneck
-> (7, 7, 64)

# Decoder
x = layers.Conv2D(64, (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)  # Output -> (28, 28, 1)

# Define model
autoencoder = models.Model(input_img, decoded)
autoencoder.compile(optimizer=optimizers.Adam(1e-3),
                    loss=losses.MeanSquaredError())
autoencoder.summary()

# ---------------- Train Model ----------------
print("\n--- Starting Model Training ---")
history = autoencoder.fit(
    x_train_noisy, x_train,  # <-- Target is the CLEAN image
    epochs=epochs,
    batch_size=batch_size,
    shuffle=True,
    validation_data=(x_test_noisy, x_test)
)

# ---------------- Plot Training Loss ----------------
print("\n--- Plotting Training History ---")
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label="Training Loss")
plt.plot(history.history['val_loss'], label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("MSE Loss")
plt.legend()
plt.title("Training History")
plt.grid(True)
plt.show()

# ---------------- Predict on Noisy Test Images ----------------
print("\n--- Generating Denoised Images ---")
decoded_imgs = autoencoder.predict(x_test_noisy[:num_display])
```

```python
# ---------------- Visualization ----------------
print("--- Displaying Results ---")
plt.figure(figsize=(20, 6))
for i in range(num_display):
    # Display Original
    ax = plt.subplot(3, num_display, i + 1)
    plt.imshow(x_test[i].squeeze(), cmap='gray')
    ax.set_title("Original")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display Noisy Input
    ax = plt.subplot(3, num_display, i + 1 + num_display)
    plt.imshow(x_test_noisy[i].squeeze(), cmap='gray')
    ax.set_title("Noisy")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display Denoised Output
    ax = plt.subplot(3, num_display, i + 1 + 2 * num_display)
    plt.imshow(decoded_imgs[i].squeeze(), cmap='gray')
    ax.set_title("Denoised")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.suptitle("Autoencoder Denoising Results", fontsize=16)
viz_path = os.path.join(save_dir, "mnist_denoising_result.png")
plt.savefig(viz_path, bbox_inches='tight')
plt.show()

# ---------------- Save Model ----------------
model_path = os.path.join(save_dir, "autoencoder_mnist.h5")
autoencoder.save(model_path)
print("\n□ Training Complete!")
print(f"□ Saved model: {model_path}")
print(f"□ Saved visualization: {viz_path}")

# ---------------- Extra: Evaluate with PSNR & SSIM ----------------
psnr = tf.image.psnr(x_test[:num_display], decoded_imgs, max_val=1.0)
ssim = tf.image.ssim(x_test[:num_display], decoded_imgs, max_val=1.0)

print("\n--- Image Quality Metrics (on displayed samples) ---")
print(f"□ Average PSNR (Higher is better):
{np.mean(psnr.numpy()):.2f}")
print(f"□ Average SSIM (Closer to 1 is better):
{np.mean(ssim.numpy()):.4f}")
```