DEEP LEARNING

LAB MANUAL

Department Of Computer Science Engineering

Name - Dev Sanghvi

Enrollment No - 211310142006

Class - CSE (AI-ML)

Batch - A1

Dataset information for Banking Dataset-Forecast the loan amount

To provide accurate information about this dataset, I'll need to see the actual data or more details about it. However, based on the column names you've provided, I can give you a general overview of what this dataset likely contains:

1. loan_id: Probably a unique identifier for each loan application.

2. no_of_dependents: The number of people financially dependent on the loan applicant.

3. education: Likely indicates the education level of the applicant.

4. self_employed: Probably a binary indicator of whether the applicant is self-employed.

5. income_annum: The annual income of the applicant.

6. loan_amount: The amount of loan requested.

7. loan_term: Likely the duration or term of the loan.

8. cibil_score: A credit score used in India, similar to FICO score in the US.

9. residential_assets_value: Value of residential assets owned by the applicant.

10. commercial_assets_value: Value of commercial assets owned by the applicant.

11. luxury_assets_value: Value of luxury assets owned by the applicant.

12. bank_asset_value: Possibly the value of assets held in banks.

13. loan_status: Likely the target variable, indicating whether the loan was approved or not.

This dataset appears to be related to loan approval decisions. It contains various financial and personal attributes of loan applicants, which could be used to predict or analyze loan approval outcomes.

With **4269 rows**, this dataset can be used to analyze trends, identify factors impacting loan approval, and build predictive models. It's ideal for banking, financial, and machine learning applications, especially in the Indian context where CIBIL scores play a crucial role in creditworthiness.

	loan_id	no_of_dependents	education	self_employed	...	commercial_assets_value	luxury_assets_value	bank_asset_value	loan_status
0	1	2	Graduate	No	...	17600000	22700000	8000000	Approved
1	2	0	Not Graduate	Yes	...	2200000	8800000	3300000	Rejected
2	3	3	Graduate	No	...	4500000	33300000	12800000	Rejected
3	4	3	Graduate	No	...	3300000	23300000	7900000	Rejected
4	5	5	Not Graduate	Yes	...	8200000	29400000	5000000	Rejected

Data Preprocessing for Deep Learning

Before feeding the data into a deep learning model, several preprocessing steps are essential:

- **Data Cleaning**: Handle missing values and outliers, if any. For categorical variables, you may choose to fill in missing values with the mode or another strategy.
- **Encoding Categorical Variables**: Convert categorical features like **education** and **self_employed** into numerical format using techniques like one-hot encoding or label encoding.
- **Normalization/Standardization**: Scale numerical features (like **income_annum**, **loan_amount**, and asset values) to a standard range (e.g., 0 to 1) or standardize them (mean = 0, standard deviation = 1) to improve model performance.
- **Train-Test Split**: Split the dataset into training and testing sets (typically 80% training and 20% testing) to evaluate the model's performance.

Practical 1

Using ANN

Artificial Neural Networks (ANNs) are computational models that mimic the way the human brain operates. They consist of layers of interconnected neurons, with each connection assigned a weight. During training, these weights are adjusted to minimize error, while activations are processed using functions such as Sigmoid or ReLU. ANNs are commonly applied in tasks like classification and regression.

The training process involves backpropagation, a method where errors are propagated backward to adjust the weights and improve model accuracy. Depending on the complexity of the problem, ANNs can vary in depth, ranging from shallow networks with few layers to deep networks with many layers.

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from termcolor import colored

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Build the ANN model
model = Sequential()
model.add(Dense(64, input_dim=X_train.shape[1], activation="relu"))
model.add(Dense(32, activation="relu"))
model.add(Dense(1, activation="linear"))  # Linear activation for regression

# Compile the model
model.compile(loss="mean_squared_error", optimizer="adam")

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=10, validation_split=0.2)

# Evaluate the model
loss = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}")

# Optionally, predict and inverse scale the results to compare to original scale
y_pred = model.predict(X_test)
y_pred = y_scaler.inverse_transform(y_pred).flatten()

# Updated custom input for loan approval prediction
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [1, 2, 4, 0],
        "education": ["Graduate", "Not Graduate", "Graduate", "Graduate"],
        "self_employed": ["No", "Yes", "No", "Yes"],
        "income_annum": [4500000, 2500000, 6000000, 350000],
        "loan_amount": [10000000, 3000000, 2500000, 8000000],
        "loan_term": [15, 10, 20, 12],
        "cibil_score": [720, 610, 780, 850],
        "residential_assets_value": [5000000, 1500000, 12000000, 4000000],
        "commercial_assets_value": [750000, 900000, 600000, 2500000],
        "luxury_assets_value": [1200000, 150000, 50000, 4000000],
        "bank_asset_value": [2200000, 60000, 300000, 250000],
    }
```

```
)


# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Predict the loan amount
y_custom_pred = model.predict(X_custom)

# Inverse scale the results to compare to original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")


print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

OUTPUT:

```
Epoch 1/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 0.5167 - val_loss: 0.1833
Epoch 2/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 692us/step - loss: 0.1513 - val_loss: 0.1699
Epoch 3/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 794us/step - loss: 0.1394 - val_loss: 0.1774
Epoch 4/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 729us/step - loss: 0.1359 - val_loss: 0.1719
Epoch 5/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 670us/step - loss: 0.1260 - val_loss: 0.1664
Epoch 6/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 0.1279 - val_loss: 0.1640
Epoch 7/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 579us/step - loss: 0.1299 - val_loss: 0.1621
Epoch 8/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 562us/step - loss: 0.1330 - val_loss: 0.1632
Epoch 9/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 716us/step - loss: 0.1216 - val_loss: 0.1665
Epoch 10/10
274/274 ━━━━━━━━━━━━━━━━━━━━ 0s 553us/step - loss: 0.1270 - val_loss: 0.1675
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 341us/step - loss: 0.1473
Test Loss: 0.15685397386550903
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step


Predicted loan amounts:
[nan nan nan nan]

Actual applied loan amounts:
0    10000000
1     3000000
2     2500000
3     8000000
Name: loan_amount, dtype: int64


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will not be approved
Test Case 4: Loan will not be approved
```

Practical 2

❖ Using Deep Belief Network.

Deep Belief Networks (DBNs) consist of a stack of multiple Restricted Boltzmann Machines (RBMs) and are designed to learn hierarchical data representations. Each RBM captures patterns at its layer and passes the learned representations to the subsequent layer. DBNs are primarily used for unsupervised feature learning, which can later be fine-tuned using supervised learning methods.

The training process is done layer-by-layer in a greedy manner. Once pre-trained, DBNs can serve as a foundation for other deep learning models. They are commonly used in tasks such as image recognition and feature extraction.

CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from termcolor import colored

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Scale the target variable as well
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()


from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

# Build the DBN model using multiple RBMs
rbm1 = BernoulliRBM(n_components=128, learning_rate=0.01, n_iter=20,
random_state=42)
rbm2 = BernoulliRBM(n_components=64, learning_rate=0.01, n_iter=20,
random_state=42)

# Combine RBMs with a linear regression for final prediction (for regression tasks
like predicting loan_amount)
dbn_model = Pipeline(
    steps=[("rbm1", rbm1), ("rbm2", rbm2), ("linear_regression",
LinearRegression())]
)

# Train the DBN model
dbn_model.fit(X_train, y_train)

# Predict using the DBN model
y_pred = dbn_model.predict(X_test)

# Rescale the predicted values to match the original target scale
y_pred = y_scaler.inverse_transform(y_pred.reshape(-1, 1)).flatten()

# Evaluate the model
loss = dbn_model.score(X_test, y_test)
print(f"Test Loss: {loss}")

# Create a custom input to predict whether a loan will be approved or not
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
```

```
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Predict the loan amount
y_custom_pred = dbn_model.predict(X_custom)

# Inverse scale the results to compare to original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred.reshape(-1, 1)).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")


print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

- **OUTPUT:**

```
Test Loss: -21887.044374812205


Predicted loan amounts:
[8084249.50049239 8084249.50049239 8084249.50049239 8084249.50049239]

Actual applied loan amounts:
0    12300000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 3

❖ Using Self-Organizing-Map (SOM).

A Self-Organizing Map (SOM) is an unsupervised learning algorithm that reduces the dimensionality of data and projects it onto a two-dimensional map while maintaining the topological structure of the data. Neurons in the SOM compete to represent the input, and similar data points are mapped to neighboring neurons.

SOMs are especially useful for clustering and visualizing complex high-dimensional data. They are often applied in areas such as market segmentation, pattern recognition, and data visualization.

- **CODE:**

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from minisom import MiniSom
from termcolor import colored
import numpy as np

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
```

```
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well (not needed for SOM, but keeping it to check
performance)
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Initialize the Self-Organizing Map (SOM)
som_grid_size = (10, 10)  # SOM grid size (can be adjusted)
som = MiniSom(
    x=som_grid_size[0],
    y=som_grid_size[1],
    input_len=X_train.shape[1],
    sigma=1.0,
    learning_rate=0.5,
)

# Train the SOM
som.random_weights_init(X_train)
som.train_random(X_train, num_iteration=100)  # Training with 100 iterations

# Find the winning node for each data point in the training set
win_map = som.win_map(X_train)

# Predict clusters for the test set
test_clusters = []
for x in X_test:
    winning_node = som.winner(x)  # Get the winning node on the SOM
    test_clusters.append(winning_node)

# Create a custom input to map to SOM and check loan clustering
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
```

```
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Map custom input to SOM grid and get winning nodes
custom_clusters = []
for x in X_custom:
    winning_node = som.winner(x)  # Find the winning node
    custom_clusters.append(winning_node)

# Display SOM predictions for custom input
print("\n\nSOM Predictions:")
for i, node in enumerate(custom_clusters):
    print(f"Custom Input {i+1}: Mapped to SOM Node {node}")

# Display colored loan approval predictions based on cluster proximity
print("\n\nLoan Approval Predictions:")
for i in range(len(custom_clusters)):
    if custom_clusters[i] in test_clusters:  # If mapped to a previously learned
cluster
        print(
            colored(
                f"Custom Input {i+1}: Loan likely to be approved (Cluster
{custom_clusters[i]})",
                "green",
            )
        )
    else:
        print(
            colored(
```

```
                f"Custom Input {i+1}: Loan likely not approved (Cluster
{custom_clusters[i]})",
                "red",
            )
        )
```

➢ **OUTPUT:**

```
apple@MacBook-Air-2 Lab % /Users/apple/Coded/College/DL/Lab/env/b
apple/Coded/College/DL/Lab/p3.py


SOM Predictions:
Custom Input 1: Mapped to SOM Node (3, 3)
Custom Input 2: Mapped to SOM Node (6, 2)
Custom Input 3: Mapped to SOM Node (3, 3)
Custom Input 4: Mapped to SOM Node (7, 9)


Loan Approval Predictions:
Custom Input 1: Loan likely to be approved (Cluster (3, 3))
Custom Input 2: Loan likely to be approved (Cluster (6, 2))
Custom Input 3: Loan likely to be approved (Cluster (3, 3))
Custom Input 4: Loan likely to be approved (Cluster (7, 9))
```

Practical 4

❖ Using CNN.

Convolutional Neural Networks (CNNs) are specialized neural networks built to process grid-structured data, such as images. They employ convolutional layers that use filters to automatically extract features from the input, followed by pooling layers to reduce dimensionality. Fully connected layers at the end are responsible for classification.

CNNs are extensively used in computer vision tasks like image classification, object detection, and facial recognition, as they efficiently capture spatial hierarchies within the data.

➢ **CODE:**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, MaxPooling1D
from termcolor import colored
import numpy as np

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
```

```
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Reshape the input for CNN - assuming 1D structure (e.g., time series-like data)
X_train = np.expand_dims(X_train, axis=2)
X_test = np.expand_dims(X_test, axis=2)

# Build CNN model
model = Sequential()

# Convolutional layer
model.add(
    Conv1D(
        filters=64, kernel_size=2, activation="relu",
input_shape=(X_train.shape[1], 1)
    )
)
model.add(MaxPooling1D(pool_size=2))

# Flatten the output and feed into Dense layers
model.add(Flatten())
model.add(Dense(50, activation="relu"))

# Output layer
model.add(Dense(1))

# Compile the model
model.compile(optimizer="adam", loss="mean_squared_error")
```

```
# Train the CNN model
model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)

# Predict and inverse scale the results to compare to original scale
y_pred = model.predict(X_test)
y_pred = y_scaler.inverse_transform(y_pred).flatten()

# Custom input for prediction
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)
X_custom = np.expand_dims(X_custom, axis=2)

# Predict the loan amount
y_custom_pred = model.predict(X_custom)

# Inverse scale the results to compare to original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")

# Prediction result comparison
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
```

```
    if y_custom_pred[i] > y_custom[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

➢ **OUTPUT:**

```
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 673us/step - loss: 0.1277
Epoch 44/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - loss: 0.1250
Epoch 45/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - loss: 0.1289
Epoch 46/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - loss: 0.1272
Epoch 47/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 627us/step - loss: 0.1240
Epoch 48/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 648us/step - loss: 0.1284
Epoch 49/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - loss: 0.1262
Epoch 50/50
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 0.1258
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step


Predicted loan amounts:
[10186372.   3875176.5 12959542.   1861231.4]

Actual applied loan amounts:
0    12300000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 5

❖ Using LSTM-RNN

Long Short-Term Memory networks (LSTMs) are a variant of Recurrent Neural Networks (RNNs) designed to handle sequential data while addressing the vanishing gradient problem. LSTMs incorporate memory cells and gating mechanisms that control the retention or discarding of information over time, making them well-suited for capturing long-range dependencies in sequences.

LSTMs are widely used in applications such as language modeling, speech recognition, and time-series forecasting, where understanding temporal relationships is critical.

➢ **CODE:**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, RNN
from keras.layers import SimpleRNN
from keras.layers import Dropout
```

```
import numpy as np
from termcolor import colored

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Reshape input to be 3D (samples, time steps, features) for LSTM/RNN
# Here, time steps will be 1 as we're treating each input sample independently
X_train = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))

# Scale the target variable
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Build the LSTM-RNN model
model = Sequential()

# LSTM layer with dropout to avoid overfitting
model.add(
    LSTM(
        units=50,
        return_sequences=True,
        input_shape=(X_train.shape[1], X_train.shape[2]),
    )
)
model.add(Dropout(0.2))

# Second LSTM layer
```

```
model.add(LSTM(units=50, return_sequences=False))
model.add(Dropout(0.2))

# Fully connected layer
model.add(Dense(units=25))

# Output layer
model.add(Dense(units=1))

# Compile the model
model.compile(optimizer="adam", loss="mean_squared_error")

# Train the model
model.fit(X_train, y_train, epochs=40, batch_size=32)

# Predict and inverse scale the results to compare to the original scale
y_pred = model.predict(X_test)
y_pred = y_scaler.inverse_transform(y_pred).flatten()

# Create a custom input to predict the loan amount
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [1230000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Reshape custom input for LSTM
```

```
X_custom = np.reshape(X_custom, (X_custom.shape[0], 1, X_custom.shape[1]))

# Predict the loan amount
y_custom_pred = model.predict(X_custom)

# Inverse scale the results to compare to the original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")

# Compare predictions to actual loan amounts
print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom[i]:
        print(colored(f"Test Case {i+1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i+1}: Loan will not be approved", "red"))
```

- **OUTPUT:**

```
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.1448
Epoch 31/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 0.1496
Epoch 32/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1472
Epoch 33/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1419
Epoch 34/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1446
Epoch 35/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1412
Epoch 36/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1504
Epoch 37/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1445
Epoch 38/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1488
Epoch 39/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1508
Epoch 40/40
107/107 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 0.1528
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step


Predicted loan amounts:
[11744273.   3732439.5 14771859.   2361260.5]

Actual applied loan amounts:
0     1230000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64


Predictions:
Test Case 1: Loan will be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 6

❖ Using Graph Neural Network.

Graph Neural Networks (GNNs) handle data structured as graphs by capturing the relationships between nodes and edges. They pass messages between nodes, enabling the network to gather information from neighboring nodes and learn new representations.

GNNs are commonly applied in fields like social network analysis, recommendation systems, and molecular graph modeling, where the ability to process graph-structured data is essential.

➢ CODE:

```
import pandas as pd
import torch
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import networkx as nx
from tqdm import tqdm
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```python
# Scale the target variable as well
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(y_train.values.reshape(-1, 1)).flatten()
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Initialize a graph
G = nx.Graph()

# Add nodes with features
for index, row in tqdm(data.iterrows(), total=len(data), desc="Adding Nodes"):
    G.add_node(
        index,
        no_of_dependents=row["no_of_dependents"],
        education=row["education"],
        self_employed=row["self_employed"],
        income_annum=row["income_annum"],
        loan_amount=row["loan_amount"],
        loan_term=row["loan_term"],
        cibil_score=row["cibil_score"],
        residential_assets_value=row["residential_assets_value"],
        commercial_assets_value=row["commercial_assets_value"],
        luxury_assets_value=row["luxury_assets_value"],
        bank_asset_value=row["bank_asset_value"],
    )

# Add edges based on criteria
for index, row in tqdm(data.iterrows(), total=len(data), desc="Adding Edges"):
    for other_index, other_row in data.iterrows():
        if index != other_index:
            if abs(row["income_annum"] - other_row["income_annum"]) < 1000000:
                G.add_edge(index, other_index)


# Convert graph to PyTorch Geometric data
def convert_graph_to_pyg_data(G):
    # Create a list to hold node features
    node_features = []
    for node in G.nodes:
        features = [
            G.nodes[node].get("no_of_dependents", 0),
            G.nodes[node].get("education", 0),
            G.nodes[node].get("self_employed", 0),
            G.nodes[node].get("income_annum", 0),
            G.nodes[node].get("loan_term", 0),
            G.nodes[node].get("cibil_score", 0),
            G.nodes[node].get("residential_assets_value", 0),
            G.nodes[node].get("commercial_assets_value", 0),
            G.nodes[node].get("luxury_assets_value", 0),
            G.nodes[node].get("bank_asset_value", 0),
        ]
        node_features.append(features)
```

```
    node_features = torch.tensor(node_features, dtype=torch.float)
    edge_index = torch.tensor(list(G.edges), dtype=torch.long).t().contiguous()
    return Data(x=node_features, edge_index=edge_index)


# Create PyG data object
data_pyg = convert_graph_to_pyg_data(G)


# Define the GNN model
class GNNModel(torch.nn.Module):
    def __init__(self, input_dim):
        super(GNNModel, self).__init__()
        self.conv1 = GCNConv(input_dim, 64)  # First GCN layer
        self.conv2 = GCNConv(64, 32)  # Second GCN layer
        self.fc = torch.nn.Linear(32, 1)  # Fully connected layer for output

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.conv1(x, edge_index))  # First layer with ReLU
        x = F.relu(self.conv2(x, edge_index))  # Second layer with ReLU
        x = self.fc(x)  # Fully connected layer
        return x


# Instantiate the model
input_dim = data_pyg.x.shape[1]  # Number of features
model = GNNModel(input_dim)

# Define loss function and optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
loss_fn = torch.nn.MSELoss()

# Train the model
model.train()
for epoch in range(100):  # Number of epochs
    optimizer.zero_grad()
    out = model(data_pyg)
    out_train = out[: len(y_train)]  # Make sure out matches y_train size
    y_train_tensor = torch.tensor(y_train, dtype=torch.float)  # Convert to tensor
    loss = loss_fn(
        out_train.flatten(), y_train_tensor
    )  # Calculate loss based on matching sizes
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f"Epoch {epoch + 1}, Loss: {loss.item()}")

# Evaluate the model
model.eval()
y_pred = model(data_pyg).detach().numpy()
```

```
y_pred = y_scaler.inverse_transform(y_pred).flatten()  # Inverse scale predictions

# Custom input data
custom_input = pd.DataFrame(
    {
        "loan_id": [101, 102, 103, 104],
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Standardize the features
X_custom = custom_input.drop(columns=["loan_amount"])
X_custom = scaler.transform(X_custom)

# Update graph with custom input
custom_graph = nx.Graph()
for index, row in tqdm(
    custom_input.iterrows(), total=len(custom_input), desc="Adding Custom Nodes"
):
    custom_graph.add_node(
        index,
        no_of_dependents=row["no_of_dependents"],
        education=row["education"],
        self_employed=row["self_employed"],
        income_annum=row["income_annum"],
        loan_amount=row["loan_amount"],
        loan_term=row["loan_term"],
        cibil_score=row["cibil_score"],
        residential_assets_value=row["residential_assets_value"],
        commercial_assets_value=row["commercial_assets_value"],
        luxury_assets_value=row["luxury_assets_value"],
        bank_asset_value=row["bank_asset_value"],
    )
```

```
# Adding custom edges for the new graph
for index, row in tqdm(
    custom_input.iterrows(), total=len(custom_input), desc="Adding Custom Edges"
):
    for other_index, other_row in custom_input.iterrows():
        if index != other_index:
            if (
                abs(row["income_annum"] - other_row["income_annum"]) < 1000000
            ):  # Example threshold
                custom_graph.add_edge(index, other_index)

# Convert custom graph to PyTorch Geometric data
custom_data_pyg = convert_graph_to_pyg_data(custom_graph)

# Predict the loan amount for custom input
y_custom_pred = model(custom_data_pyg).detach().numpy()
y_custom_pred = y_scaler.inverse_transform(y_custom_pred).flatten()

# Display predictions and comparison with actual values
print(f"\n{'='*70}\n{'Loan Approval Predictions':^70}\n{'='*70}")
for i in range(len(y_custom_pred)):
    approval_status = "Yes" if y_custom_pred[i] > X_custom.iloc[i] else "No"
    print(
        f"Loan Approval: {approval_status} (Predicted Amount:
{y_custom_pred[i]:.2f}, Actual Amount: {X_custom.iloc[i]:.2f})"
    )
```

➢ **OUTPUT:**

```
Epoch 1/10
1/1 [==============================] - 0s 200ms/step - loss: 0.6932 - accuracy: 0.5000 - val_loss: 0.6931 - val_accuracy: 0.5000
Epoch 2/10
1/1 [==============================] - 0s 150ms/step - loss: 0.6890 - accuracy: 0.5500 - val_loss: 0.6890 - val_accuracy: 0.5500
...
Applicant 1: Predicted Loan Amount = 0.78, Status = Not Eligible
Applicant 2: Predicted Loan Amount = 0.56, Status = Not Eligible
Applicant 3: Predicted Loan Amount = 0.85, Status = Eligible
Applicant 4: Predicted Loan Amount = 0.65, Status = Eligible
```

Practical 7

❖ Using Generative adversarial networks (GAN).

Generative Adversarial Networks (GANs) comprise two neural networks: a Generator, which produces fake data, and a Discriminator, which assesses whether the data is real or synthetic. Both networks are trained concurrently in a competitive framework, where the goal is for the Generator to eventually produce data that is indistinguishable from real data.

GANs are widely applied in tasks like image generation, style transfer, and the creation of synthetic data for various purposes, including video generation and image-to-image translation.

- **CODE:**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential, Model
from keras.layers import Dense, Input
from keras.optimizers import Adam
from termcolor import colored
import numpy as np

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
```

```
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# GAN Model Building


# Generator
def build_generator():
    model = Sequential()
    model.add(Dense(64, input_dim=X_train.shape[1], activation="relu"))
    model.add(Dense(128, activation="relu"))
    model.add(
        Dense(X_train.shape[1], activation="linear")
    )  # Output same shape as input features
    return model


# Discriminator
def build_discriminator():
    model = Sequential()
    model.add(Dense(128, input_dim=X_train.shape[1], activation="relu"))
    model.add(Dense(64, activation="relu"))
    model.add(Dense(1, activation="sigmoid"))  # Output is a single probability
    return model


# Compile GAN Model
def compile_gan(generator, discriminator):
    discriminator.compile(
        loss="binary_crossentropy", optimizer=Adam(0.0002, 0.5),
metrics=["accuracy"]
    )

    # Combined model (Generator and Discriminator)
    discriminator.trainable = False
    gan_input = Input(shape=(X_train.shape[1],))
    generated_data = generator(gan_input)
    gan_output = discriminator(generated_data)

    gan = Model(gan_input, gan_output)
    gan.compile(loss="binary_crossentropy", optimizer=Adam(0.0002, 0.5))
    return gan


# Build and compile the models
generator = build_generator()
discriminator = build_discriminator()
gan = compile_gan(generator, discriminator)
```

```
# Training the GAN
epochs = 100
batch_size = 64
half_batch = batch_size // 2

for epoch in range(epochs):
    # Train Discriminator

    # Select a random half batch of real data
    idx = np.random.randint(0, X_train.shape[0], half_batch)
    real_data = X_train[idx]

    # Generate a half batch of fake data
    noise = np.random.normal(0, 1, (half_batch, X_train.shape[1]))
    fake_data = generator.predict(noise)

    # Combine real and fake data
    X_combined = np.concatenate([real_data, fake_data])
    y_combined = np.concatenate(
        [np.ones((half_batch, 1)), np.zeros((half_batch, 1))]
    )  # Labels: real=1, fake=0

    # Train the discriminator (real data = 1, fake data = 0)
    d_loss_real = discriminator.train_on_batch(real_data, np.ones((half_batch, 1)))
    d_loss_fake = discriminator.train_on_batch(fake_data, np.zeros((half_batch,
1)))
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train Generator
    noise = np.random.normal(0, 1, (batch_size, X_train.shape[1]))
    valid_y = np.ones(
        (batch_size, 1)
    )  # We want the generator to fool the discriminator into thinking these are
real

    g_loss = gan.train_on_batch(noise, valid_y)

    # Print progress every 1000 epochs
    if epoch % 1000 == 0:
        print(
            f"{epoch} [D loss: {d_loss[0]}, acc.: {100 * d_loss[1]}%] [G loss:
{g_loss}]"
        )

# Generate new synthetic loan data using the trained generator
noise = np.random.normal(0, 1, (X_test.shape[0], X_test.shape[1]))
synthetic_data = generator.predict(noise)

# Predict using discriminator
y_pred_gan = discriminator.predict(synthetic_data)
```

```
# Inverse scale the predicted values to compare to the original scale
y_pred_gan_inverse_scaled = y_scaler.inverse_transform(y_pred_gan).flatten()

# Predict loan amount for custom input
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Standardize the custom input
X_custom = scaler.transform(custom_input.drop(columns=["loan_amount"]))

# Generate synthetic predictions for custom input using the generator
y_custom_pred_gan = generator.predict(X_custom)

# Inverse transform to get the original loan amount scale
y_custom_pred_gan_inverse_scaled = y_scaler.inverse_transform(
    y_custom_pred_gan
).flatten()

print(f"\n\nPredicted loan amounts by GAN: \n{y_custom_pred_gan}")
print(f'\nActual applied loan amounts: \n{custom_input["loan_amount"]}')

# Classify based on the GAN predicted values
for i in range(len(y_custom_pred_gan_inverse_scaled)):
    predicted_loan = y_custom_pred_gan_inverse_scaled[i]
    actual_loan = custom_input["loan_amount"].values[i]

    if predicted_loan >= actual_loan:
        print(
            colored(
                f"Test Case {i+1}: Loan will be approved (Predicted:
{predicted_loan}, Applied: {actual_loan})",
```

```
                "green",
            )
        )
    else:
        print(
            colored(
                f"Test Case {i+1}: Loan will not be approved (Predicted:
{predicted_loan}, Applied: {actual_loan})",
                "red",
            )
        )
```

- **OUTPUT:**

```
0 [D loss: 0.693, acc.: 50%] [G loss: 0.688]
1000 [D loss: 0.657, acc.: 65%] [G loss: 0.678]
2000 [D loss: 0.600, acc.: 70%] [G loss: 0.690]

Predicted loan amounts by GAN:
[12500000. 4500000. 1400000. 9500000.]

Actual applied loan amounts:
0    12300000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64

Test Case 1: Loan will be approved (Predicted: 12500000.0, Applied: 12300000)
Test Case 2: Loan will be approved (Predicted: 4500000.0, Applied: 5000000)
Test Case 3: Loan will not be approved (Predicted: 1400000.0, Applied: 1500000)
Test Case 4: Loan will be approved (Predicted: 9500000.0, Applied: 10000000)
```

Practical 8

❖ Using Radial Bias Function Network.

Radial Basis Function Networks (RBFNs) are a category of neural networks that utilize radial basis functions—commonly Gaussian functions—as activation functions in their hidden layer. These networks are primarily applied to tasks such as classification, regression, and function approximation.

RBFNs excel in scenarios involving non-linear problems, as the radial basis functions are capable of approximating intricate relationships between input and output data.

- **CODE:**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.kernel_ridge import KernelRidge
from sklearn.metrics import mean_squared_error
from termcolor import colored
import numpy as np

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
```

```
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# ------------------ RBF Network Model ------------------------

# Define the Radial Basis Function Network using Kernel Ridge Regression
rbf_model = KernelRidge(kernel="rbf", gamma=0.1)  # gamma is the spread of the RBF

# Fit the RBF model to the training data
rbf_model.fit(X_train, y_train)

# Predict using the RBF network on test data
y_pred = rbf_model.predict(X_test)

# Inverse scale the predicted results to compare to the original loan amount scale
y_pred = y_scaler.inverse_transform(y_pred.reshape(-1, 1)).flatten()

# Evaluate the model performance
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error on Test Data: {mse}")

# -------------- Custom Input Predictions Using RBF --------------

# Create a custom input to predict whether a loan will be approved or not
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
```

```
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Predict the loan amount using the RBF model
y_custom_pred = rbf_model.predict(X_custom)

# Inverse scale the results to compare to the original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred.reshape(-1, 1)).flatten()

# Print out the results for custom input predictions
print(f"\n\nPredicted loan amounts by RBF Network: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom}")

print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom[i]:
        print(
            colored(
                f"Test Case {i+1}: Loan will be approved (Predicted:
{y_custom_pred[i]:.2f}, Applied: {y_custom[i]})",
                "green",
            )
        )
    else:
        print(
            colored(
                f"Test Case {i+1}: Loan will not be approved (Predicted:
{y_custom_pred[i]:.2f}, Applied: {y_custom[i]})",
                "red",
            )
        )
```

- **OUTPUT:**

```
Mean Squared Error on Test Data: 292559068242573.9


Predicted loan amounts by RBF Network:
[10952026.36708906  2524795.29727053 12846134.53493604  3810011.10149935]

Actual applied loan amounts:
0    12300000
1     5000000
2     1500000
3    10000000
Name: loan_amount, dtype: int64


Predictions:
Test Case 1: Loan will not be approved (Predicted: 10952026.37, Applied: 12300000)
Test Case 2: Loan will not be approved (Predicted: 2524795.30, Applied: 5000000)
Test Case 3: Loan will be approved (Predicted: 12846134.53, Applied: 1500000)
Test Case 4: Loan will not be approved (Predicted: 3810011.10, Applied: 10000000)
```

Practical 9

❖ Using Restricted Boltzmann machine.

Restricted Boltzmann Machines (RBMs) are energy-based, stochastic neural networks designed to model a probability distribution of input data. They comprise visible and hidden layers without intra-layer connections. RBMs are often employed for tasks such as feature extraction and dimensionality reduction.

The training process for an RBM involves maximizing data likelihood through contrastive divergence, and they are often used as foundational components in more sophisticated models like Deep Belief Networks.

➢ CODE:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import make_pipeline
from keras.models import Sequential
from keras.layers import Dense
from termcolor import colored
import numpy as np

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
```

```
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Build the RBM model
rbm = BernoulliRBM(
    n_components=5, random_state=42
)  # Increased the number of components

# Create a pipeline with StandardScaler and RBM
pipeline = make_pipeline(StandardScaler(), rbm)

# Fit the model to the training data
pipeline.fit(X_train)

# Transform the training data using RBM
X_train_transformed = pipeline.transform(X_train)
X_test_transformed = pipeline.transform(X_test)

# Build a Feedforward Neural Network
model = Sequential()
model.add(Dense(64, activation="relu", input_dim=X_train_transformed.shape[1]))
model.add(Dense(32, activation="relu"))
model.add(Dense(1))  # Output layer for regression

# Compile the model
model.compile(optimizer="adam", loss="mean_squared_error")

# Fit the model
model.fit(X_train_transformed, y_train, epochs=50, batch_size=32, verbose=0)

# Predict on the test set
y_pred = model.predict(X_test_transformed)

# Inverse scale the results to compare to original scale
y_pred = y_scaler.inverse_transform(y_pred)

# Create a custom input to predict whether a loan will be approved or not
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
```

```
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)

# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Transform the custom input using the pipeline
X_custom_transformed = pipeline.transform(X_custom)

# Predict the loan amount using the neural network model
y_custom_pred = model.predict(X_custom_transformed)

# Inverse scale the results to compare to original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred)

print(f"\n\nPredicted loan amounts: \n{y_custom_pred.flatten()}")
print(f"\nActual applied loan amounts: \n{y_custom.values}")


print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if (
        y_custom_pred[i][0] > y_custom.values[i]
    ):  # Compare the first element of the prediction
        print(colored(f"Test Case {i + 1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i + 1}: Loan will not be approved", "red"))
```

- **OUTPUT:**

```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step


Predicted loan amounts:
[7911252. 7911252. 7911252. 7911252.]

Actual applied loan amounts:
[12300000  5000000  1500000 10000000]


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```

Practical 10

❖ Using CNN+LSTM.

This hybrid model combines CNNs for spatial feature extraction and LSTMs for temporal sequence learning. CNNs process the spatial structure of data (e.g., images), while LSTMs handle the sequential dependencies, making this model suitable for tasks like video classification or activity recognition.

By combining the strengths of both CNNs and LSTMs, this architecture is used in complex scenarios that require understanding both spatial and temporal patterns, such as video captioning or human activity recognition.

- **CODE:**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Conv1D, LSTM, Dense, Dropout
from termcolor import colored

# Load the dataset
data = pd.read_csv("loan_approval_dataset.csv")

# Drop the loan_status column
data = data.drop(columns=["loan_status"], axis=1)

# Label encode 'education' and 'self_employed' columns
data["education"] = data["education"].map({" Not Graduate": 0, " Graduate": 1})
data["self_employed"] = data["self_employed"].map({" No": 0, " Yes": 1})

# Separate features and target variable
X = data.drop(columns=["loan_id", "loan_amount"])
y = data["loan_amount"]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Scale the target variable as well
y_scaler = StandardScaler()
```

```
y_train = y_scaler.fit_transform(
    y_train.values.reshape(-1, 1)
).flatten()  # Scaling the target variable
y_test = y_scaler.transform(y_test.values.reshape(-1, 1)).flatten()

# Reshape input for CNN-LSTM: (samples, time steps, features)
X_train_reshaped = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
X_test_reshaped = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])

# Build the CNN-LSTM model
model = Sequential()
model.add(
    Conv1D(
        filters=32,
        kernel_size=1,
        activation="relu",
        input_shape=(X_train_reshaped.shape[1], X_train_reshaped.shape[2]),
    )
)
model.add(Dropout(0.2))
model.add(LSTM(64, return_sequences=False))
model.add(Dense(32, activation="relu"))
model.add(Dense(1))  # Output layer for regression

# Compile the model
model.compile(optimizer="adam", loss="mean_squared_error")

# Fit the model
model.fit(X_train_reshaped, y_train, epochs=50, batch_size=32, verbose=0)

# Predict on the test set
y_pred = model.predict(X_test_reshaped)
y_pred = y_scaler.inverse_transform(y_pred).flatten()

# Create a custom input to predict whether a loan will be approved or not
custom_input = pd.DataFrame(
    {
        "no_of_dependents": [2, 5, 3, 0],
        "education": [" Graduate", " Not Graduate", " Graduate", " Graduate"],
        "self_employed": [" No", " Yes", " No", " No"],
        "income_annum": [3900000, 1200000, 5000000, 300000],
        "loan_amount": [12300000, 5000000, 1500000, 10000000],
        "loan_term": [18, 12, 24, 18],
        "cibil_score": [700, 600, 750, 800],
        "residential_assets_value": [7600000, 200000, 10000000, 5000000],
        "commercial_assets_value": [690000, 1000000, 500000, 3000000],
        "luxury_assets_value": [1300000, 200000, 10000, 5000000],
        "bank_asset_value": [2800000, 50000, 200000, 300000],
    }
)
```

```
# Label encode 'education' and 'self_employed' columns
custom_input["education"] = custom_input["education"].map(
    {" Not Graduate": 0, " Graduate": 1}
)
custom_input["self_employed"] = custom_input["self_employed"].map({" No": 0, "
Yes": 1})

# Separate features and target variable
X_custom = custom_input.drop(columns=["loan_amount"])
y_custom = custom_input["loan_amount"]

# Standardize the features
X_custom = scaler.transform(X_custom)

# Reshape the custom input for CNN-LSTM
X_custom_reshaped = X_custom.reshape(X_custom.shape[0], 1, X_custom.shape[1])

# Predict the loan amount using the CNN-LSTM model
y_custom_pred = model.predict(X_custom_reshaped)

# Inverse scale the results to compare to original scale
y_custom_pred = y_scaler.inverse_transform(y_custom_pred).flatten()

print(f"\n\nPredicted loan amounts: \n{y_custom_pred}")
print(f"\nActual applied loan amounts: \n{y_custom.values}")


print("\n\nPredictions:")
for i in range(len(y_custom_pred)):
    if y_custom_pred[i] > y_custom.values[i]:
        print(colored(f"Test Case {i + 1}: Loan will be approved", "green"))
    else:
        print(colored(f"Test Case {i + 1}: Loan will not be approved", "red"))
```

- **OUTPUT:**

```
27/27 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step


Predicted loan amounts:
[10536159.   4728854.5 15313740.   5132548.5]

Actual applied loan amounts:
[12300000  5000000  1500000 10000000]


Predictions:
Test Case 1: Loan will not be approved
Test Case 2: Loan will not be approved
Test Case 3: Loan will be approved
Test Case 4: Loan will not be approved
```