Ex No.1	Implement a simple feed-forward neural network

AIM:

To implement a simple feed-forward neural network.

a) Create a basic network
b) Analyze performance by varying the batch size, number of hidden layers, learning rate.
c) Create a confusion matrix to validate the performance of your model.
d) Visualize a neural network. 2.Solve XOR problem using Multi Layer Perceptron.

ALGORITHM:

1 Load and Preprocess the Data:

- Load the Iris dataset using load_iris from sklearn.
- Convert the target labels into one-hot encoded format using to_categorical.
- Split the data into training and testing sets using train_test_split.
- Standardize the features using StandardScaler to ensure mean 0 and variance 1.

2 Build the Feed-Forward Neural Network:

- Create a Sequential model with:
    - Two hidden layers (each with 10 neurons and ReLU activation).
    - An output layer with 3 neurons (for 3 classes) and softmax activation.
- Compile the model using the Adam optimizer, categorical cross-entropy loss, and accuracy as a performance metric.

3 Train the Model:

- Train the neural network for 50 epochs with a batch size of 32 on the training data. Monitor performance on the training data during training.

4 Experiment with Different Batch Sizes:

- Experiment with batch sizes of 16, 32, and 64. For each batch size, retrain the model for 50 epochs and compare the performance.

5 Experiment with Different Hidden Layers and Neurons:

- Test models with 1, 2, and 3 hidden layers (each with 10 neurons).

- For each configuration, rebuild, compile, and train the model for 50 epochs.

6 Experiment with Different Learning Rates and Evaluate the Model:

- Test different learning rates: 0.001, 0.01, and 0.1 by recompiling and retraining the model for 50 epochs.
- After training, predict the test set and compute the confusion matrix.
- Visualize the confusion matrix using seaborn and matplotlib.
- Visualize the model architecture using plot_model.
- Print the number of neurons in each layer of the final model.

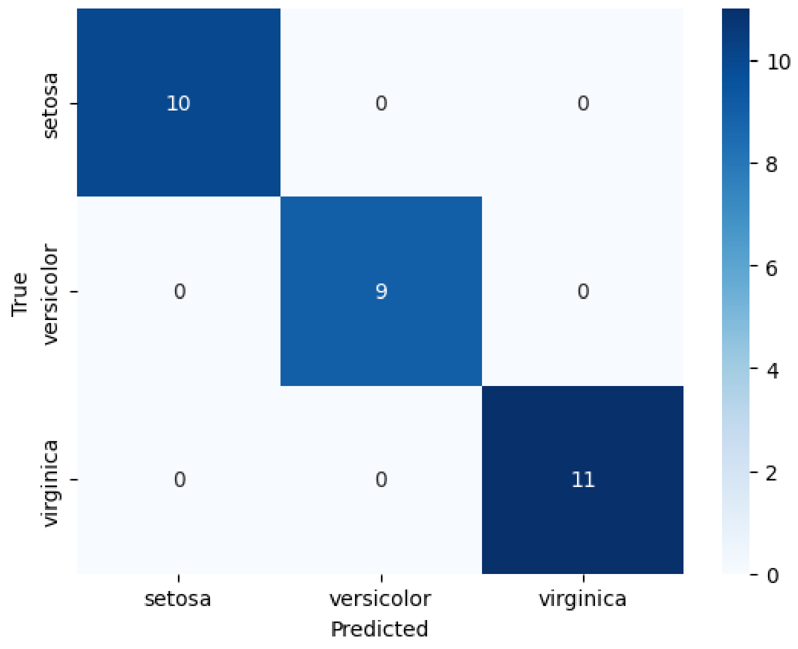
PROGRAM:

```
#exp 1
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = to_categorical(iris.target)  # Convert to one-hot encoded labels

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

# Standardize the data
```

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

# Define the basic feed-forward neural network
model = Sequential([
    Dense(10, input_shape=(4,), activation='relu'),  # Hidden layer with 10 neurons
    Dense(10, activation='relu'),                    # Another hidden layer
    Dense(3, activation='softmax')                   # Output layer for 3 classes
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.01), loss='categorical_crossentropy',
metrics=['accuracy'])

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

# Experiment with different batch sizes
for batch_size in [16, 32, 64]:
    print(f'Training with batch size: {batch_size}')
    model.fit(X_train, y_train, epochs=50, batch_size=batch_size, verbose=1)

# Experiment with different numbers of hidden layers and neurons
for layers in [1, 2, 3]:
    print(f'Training with {layers} hidden layers')
    model = Sequential()
    model.add(Dense(10, input_shape=(4,), activation='relu'))
    for _ in range(layers - 1):
        model.add(Dense(10, activation='relu'))
    model.add(Dense(3, activation='softmax'))
    model.compile(optimizer=Adam(learning_rate=0.01), loss='categorical_crossentropy', metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)

# Experiment with different learning rates
for lr in [0.001, 0.01, 0.1]:
    print(f'Training with learning rate: {lr}')
    model.compile(optimizer=Adam(learning_rate=lr), loss='categorical_crossentropy', metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1)

from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Predict on the test set
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

# Compute the confusion matrix
cm = confusion_matrix(y_true, y_pred_classes)
print("Confusion Matrix:")
print(cm)

# Plot the confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.xlabel('Predicted')
plt.ylabel('True')

plt.show()

from tensorflow.keras.utils import plot_model

# Visualize the model architecture

plot_model(model, show_shapes=True, show_layer_names=True, to_file='model.png')

from tensorflow.keras.utils import plot_model

# Visualize the model architecture

plot_model(model, show_shapes=True, show_layer_names=True, to_file='model.png')

]

# Print the number of neurons in each layer

print("Neurons in each layer of the model:")

for layer in model.layers:

    print(f"Layer {layer.name}: {layer.units} neurons")
```

OUTPUT:

```
Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
```



```
Neurons in each layer of the model:
Layer dense_8: 10 neurons
Layer dense_9: 10 neurons
Layer dense_10: 10 neurons
Layer dense_11: 3 neurons
```

RESULT:

Thus Python program to implement a simple feed-forward neural network with Creating a basic network, Analysing performance by varying the batch size, number of hidden layers, learning rate, confusion matrix to validate the performance of your model, and to visualize the neural network has been successfully executed and verified.

Ex No.2	XOR problem using Multi Layer Perceptron

AIM:

To solve XOR problem using Multi Layer Perceptron.

ALGORITHM:

1. **Data Preparation**:
    - Define the XOR input data x as a 2D array of pairs representing the binary inputs.
    - Define the target output y as the expected results of the XOR operation for each input pair.
2. **Model Definition**:
    - Use Sequential to create a feed-forward neural network.
    - Add two hidden layers:
        - First hidden layer: 8 neurons with ReLU activation.
        - Second hidden layer: 8 neurons with sigmoid activation.
    - Add an output layer with 1 neuron using sigmoid activation (since it's a binary classification problem).
3. **Model Compilation**:
    - Compile the model using the Adam optimizer (with a learning rate of 0.01), binary cross-entropy as the loss function, and accuracy as the evaluation metric.
4. **Model Training**:
    - Train the model on the XOR data (x, y) for 1000 epochs with a verbose level of 0 (no output during training).
5. **Model Evaluation**:
    - Evaluate the trained model on the same XOR dataset (x, y) to calculate the accuracy of the model.
6. **Predictions**:
    - Use the trained model to make predictions on the XOR input data (x).
    - Round the predictions to get the binary outputs.
7. **Display Results**:

- Print the model's accuracy as a percentage.
- Print the predicted XOR outputs for each input pair.

PROGRAM:

```
!pip install tensorflow
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
#xor data
x=np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([[0],[1],[1],[0]])
#Define the MLP model
model=Sequential(
   [
      Dense(8,input_dim=2,activation='relu'),   #Hidden layer with 8
      Dense(8,activation='sigmoid'),        #another hidden layer
      Dense(1,activation='sigmoid')   #output
   ]
)
# compile the model
model.compile(optimizer=Adam(learning_rate=0.01),loss='binary_crossentropy',metrics=['a
ccuracy'])
#train model
model.fit(x,y,epochs=1000,verbose=0) #train for 1000 epochs
#evaluate  the model
_, accuracy=model.evaluate(x,y)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

```
#test model

predictions=model.predict(x)

print("Predictions:")

print(np.round(predictions))
```

OUTPUT:

```
1/1 [==============================] - 0s 29ms/step - loss: 2.4559e-06 - accuracy:
1.0000

Accuracy: 100.00%

1/1 [==============================] - 0s 32ms/step

Predictions:

[[0.]

 [1.]

 [1.]

 [0.]]
```

RESULT:

Thus Python program to solve XOR problem using Multi Layer Perceptron has been successfully executed and verified.

Ex No.3	**Implement Stochastic Gradient Descent Algorithm**

AIM:

To implement Stochastic Gradient Descent Algorithm.

ALGORITHM:

1 **Define XOR Inputs and Outputs:**

- Create the input (X) and output (y) pairs for the XOR logic gate. The inputs are four combinations of binary values, and the outputs are the corresponding XOR results.

2 **Create the Neural Network:**

- Use MLPClassifier from sklearn to create a neural network with one hidden layer containing 5 neurons.
- Use the tanh activation function for the neurons.
- Set the optimizer to stochastic gradient descent (SGD) with momentum (0.9) to speed up convergence.
- Use an adaptive learning rate starting at 0.01, and set the maximum number of iterations to 10,000 for sufficient training.

3 **Train the Model:**

- Train the neural network on the XOR dataset by fitting the model using the input data X and the corresponding labels y.

4 **Predict Using the Model:**

- After training, use the trained model to make predictions on the same input data X to evaluate how well the model has learned the XOR function.

5 **Output the Predictions:**

- Print the predicted values for each input combination. The predictions will be compared with the expected XOR results.

6 **Evaluate Accuracy:**

- Use accuracy_score from sklearn to calculate and print the accuracy of the model's predictions compared to the actual XOR outputs (y). This gives an indication of how well the model has learned the XOR logic.

PROGRAM:

```
# SGD Optimizer

from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Define XOR inputs and outputs
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 1, 1, 0]

# Create a neural network with 1 hidden layer of 5 neurons, using SGD optimizer with momentum
model = MLPClassifier(hidden_layer_sizes=(5,), activation='tanh', solver='sgd',
                      learning_rate_init=0.01, momentum=0.9, learning_rate='adaptive',
                      max_iter=10000, random_state=1)

# Train the model
model.fit(X, y)
# Predict on the same inputs
predictions = model.predict(X)

# Output the results
print("Predictions:", predictions)
print("Accuracy:", accuracy_score(y, predictions))
```

OUTPUT:

```
Predictions: [0 1 1 0]

Accuracy: 1.0
```

RESULT:

Thus python program to implement Stochastic Gradient Descent Algorithm has been successfully executed and verified.

Ex No.4	**Implement Gradient Descent with AdaGrad**

AIM:

To implement Gradient Descent with AdaGrad.

ALGORITHM:

1 **Define XOR Inputs and Outputs:**

- Create the input data X representing the XOR logic gate (4 examples, each with 2 binary features).
- Define the corresponding labels y representing the output of the XOR gate (0, 1, 1, 0).

2 **Create the MLP Classifier:**

- Use MLPClassifier from sklearn to create a neural network with 2 hidden layers, each containing 10 neurons.
- Set the activation function to 'tanh' and use the Adam optimizer (solver='adam').
- Set the learning rate to 0.01 and the maximum number of iterations to 10,000 to ensure convergence.

3 **Train the Model:**

- Train the MLP model on the XOR data (X and y) using the fit method.

4 **Make Predictions:**

- Use the trained model to predict the outputs for the same XOR inputs X.

5 **Evaluate the Predictions:**

- Compare the predicted outputs with the actual labels (y) using the accuracy_score function from sklearn to calculate the accuracy.

6 **Display Results:**

- Print the predicted outputs and the accuracy of the model on the XOR problem.

PROGRAM:

```
from sklearn.neural_network import MLPClassifier

from sklearn.metrics import accuracy_score


# Define XOR inputs and outputs
```

```
X = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 1, 1, 0]

# Create a neural network with 2 hidden layers of 10 neurons each, using 'tanh' activation
model = MLPClassifier(hidden_layer_sizes=(10, 10), activation='tanh', solver='adam',
                      learning_rate_init=0.01, max_iter=10000, random_state=1)

# Train the model
model.fit(X, y)

# Predict on the same inputs
predictions = model.predict(X)

# Output the results
print("Predictions:", predictions)
print("Accuracy:", accuracy_score(y, predictions))
```

OUTPUT:

```
Predictions: [0 1 1 0]
Accuracy: 1.0
```

RESULT:

Thus python program to implement Gradient Descent with AdaGrad has been successfully executed and verified

Ex No.5	Implement Recurrent Neural Networks (RNN)

AIM:

To implement a Recurrent Neural Networks (RNN) and process any sequential data such as characters, words or video frames.

ALGORITHM:

1 Generate Data for Binary Classification:

- Use numpy to generate 1000 sequences of random data. Each sequence contains 10 time steps, with 1 feature per time step.
- Generate binary labels (0 or 1) for each sequence, which will be used for classification**.**

2 Split the Data:

- Split the data into training and testing sets using train_test_split from sklearn.
- Set aside 20% of the data for testing, and use 80% for training.

3 Build the RNN Model:

- Use a Sequential model from tensorflow.
- Add a SimpleRNN layer with 50 units that processes sequences of shape (10, 1) (10 time steps, 1 feature).
- Add a Dense layer with a single unit and a sigmoid activation function to output a probability for binary classification.

4Compile the Model:

- Compile the model using the Adam optimizer and the binary cross-entropy loss function, which is appropriate for binary classification.
- Include the accuracy metric to monitor performance during training.

5 Train the Model:

- Train the model on the training data (X_train and y_train) for 20 epochs with a batch size of 32.

6 Make Predictions and Evaluate Accuracy:

- Use the trained model to make predictions on the test data (X_test). Convert the output probabilities to binary predictions (0 or 1) based on a threshold of 0.5.

PROGRAM:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import SimpleRNN, Dense

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Generate random binary classification data

data = np.random.rand(1000, 10, 1)  # 1000 sequences, 10 time steps, 1 feature

labels = np.random.randint(2, size=(1000, 1))  # Binary labels (0 or 1)


# Split into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2,
random_state=42)


# Build RNN model

model = Sequential()

model.add(SimpleRNN(50, input_shape=(10, 1)))

model.add(Dense(1, activation='sigmoid'))  # Sigmoid for binary classification


# Compile model

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


# Train the model

model.fit(X_train, y_train, epochs=20, batch_size=32)


# Predict on test set
```

```
predictions = model.predict(X_test)

predictions = (predictions > 0.5).astype(int)  # Convert probabilities to binary (0 or 1)


# Calculate accuracy

accuracy = accuracy_score(y_test, predictions)


print("Test Accuracy:", accuracy)
```

OUTPUT:

```
Test Accuracy: 0.515
```

RESULT:

Thus python program to implement a Recurrent Neural Networks (RNN) and process any sequential data such as characters, words or video frames has been successfully executed and verified.

Ex No.6	Implement RNN with Long Short Term Networks (LSTM)

AIM:

To implement RNN with Long Short Term Networks (LSTM).

ALGORITHM:

1 **Generate Sequential Data:**

- Use numpy to generate a random dataset of sequential data. The data has 1000 sequences, each containing 10 time steps, and 1 feature per time step. The labels are also randomly generated with 1000 samples and 1 feature each.

2 **Build the LSTM Model:**

- Initialize a sequential model using tensorflow.
- Add an LSTM layer with 50 units that takes input sequences of shape (10, 1) (10 time steps with 1 feature).
- Add a Dense layer with 1 unit to produce a single regression output for each sequence.

3 **Compile the Model:**

- Compile the model using the Adam optimizer and the mean squared error (MSE) loss function, suitable for regression tasks.

4 **Train the Model:**

- Train the model on the generated data for 20 epochs with a batch size of 32.
- The input data is the sequential data, and the labels are the corresponding output values for each sequence.

5 **Make Predictions:**

- Use the trained model to make predictions on the input data. The output will be a set of predicted values corresponding to each input sequence.

6 **Display Predictions:**

- Print the shape of the predicted output, which should be (1000, 1), matching the number of sequences and the regression output shape.

PROGRAM:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Generate simple sequential data
# Example: 1000 sequences, each with 10 time steps and 1 feature
data = np.random.rand(1000, 10, 1)
labels = np.random.rand(1000, 1)

# Build the LSTM model
model = Sequential()
# Add an LSTM layer with 50 units
model.add(LSTM(50, input_shape=(10, 1)))
# Add a Dense layer with 1 unit for regression output
model.add(Dense(1))
# Compile the model
model.compile(optimizer='adam', loss='mse')
# Train the model
model.fit(data, labels, epochs=20, batch_size=32)
# Predict on some data
predictions = model.predict(data)
print("Predictions Shape:", predictions.shape)
```

OUTPUT:

```
Predictions Shape: (1000, 1)
```

RESULT:

Thus python program to implement RNN with Long Short Term Networks (LTSM) has been successfully executed and verified.

Ex No.7	Implement Different types of autoencoders

AIM:

To implement different types of autoencoders

ALGORITHM:

Undercomplete Autoencoder Algorithm:

1. **Generate Random Data:**
    - Generate a dataset with 1000 samples, each having 5 features using numpy to simulate the data to be compressed and reconstructed.
2. **Build the Autoencoder:**
    - Create an autoencoder using tensorflow.
    - The encoder reduces the dimensionality from 5 to 2 using a dense layer with ReLU activation.
    - The decoder reconstructs the original 5 features from the 2-dimensional compressed representation using a dense layer with sigmoid activation.
3. **Compile and Train the Autoencoder:**
    - Compile the autoencoder with the Adam optimizer and mean squared error (MSE) loss function.
    - Train the autoencoder using the generated data for 50 epochs with a batch size of 32.
4. **Build and Use the Encoder:**
    - Extract the encoder part of the model to compress the original data into a lower-dimensional representation.
    - Predict the compressed data (2-dimensional representation) using the encoder.
5. **Build and Use the Decoder:**
    - Extract the decoder part of the model to reconstruct the original data from the compressed 2D data.
    - Predict the reconstructed data using the decoder.
6. **Display Results:**
    - Print the shapes of the original data, compressed data, and reconstructed data to verify the dimensionality and results.

Contractive Autoencoder Algorithm:

1. **Generate Random Data:**
   - Similar to the undercomplete autoencoder, generate a dataset with 1000 samples, each with 5 features.
2. **Build the Autoencoder:**
   - Create an autoencoder using tensorflow, where the encoder reduces the dimensionality to 2 (with a ReLU activation), and the decoder reconstructs the 5 features using a sigmoid activation.
3. **Define and Use the Contractive Loss:**
   - Create a custom loss function that combines MSE loss with a contractive term that penalizes the magnitude of the weights to encourage robustness.
   - Calculate the contractive term by summing the squares of the encoder's weights and add it to the MSE loss with a small weight (1e-4).
4. **Compile and Train the Autoencoder:**
   - Compile the autoencoder using the Adam optimizer and the custom contractive loss.
   - Train the autoencoder on the generated data for 50 epochs with a batch size of 32.
5. **Build and Use the Encoder:**
   - Extract the encoder model from the autoencoder to compress the original data into 2-dimensional data.
6. **Build and Use the Decoder:**
   - Extract the decoder model from the autoencoder to reconstruct the original data from the compressed data.

PROGRAM:

```
# undercomplete

import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.models import Model
```

```
# Generate random data with 5 features
data = np.random.rand(1000, 5)

# Build the autoencoder
input_data = Input(shape=(5,))
encoded = Dense(2, activation='relu', name='encoder')(input_data)
decoded = Dense(5, activation='sigmoid', name='decoder')(encoded)
autoencoder = Model(input_data, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='mse')
# Train the autoencoder
autoencoder.fit(data, data, epochs=50, batch_size=32, shuffle=True)

# Use encoder to compress data
encoder = Model(input_data, encoded)
compressed_data = encoder.predict(data)

# Build decoder to reconstruct data from compressed representation
encoded_input = Input(shape=(2,))
decoder_layer = autoencoder.get_layer('decoder')(encoded_input)
decoder = Model(encoded_input, decoder_layer)

# Use decoder to reconstruct the original data from compressed data
reconstructed_data = decoder.predict(compressed_data)

print("Original Data Shape:", data.shape)
print("Compressed Data Shape:", compressed_data.shape)
print("Reconstructed Data Shape:", reconstructed_data.shape)
```

```
# contractive
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import tensorflow.keras.backend as K

# Generate random data with 5 features
data = np.random.rand(1000, 5)

# Build the autoencoder
input_data = Input(shape=(5,))
encoded = Dense(2, activation='relu', name='encoded')(input_data)
decoded = Dense(5, activation='sigmoid', name='decoded')(encoded)
autoencoder = Model(input_data, decoded)

# Contractive loss function
def contractive_loss(y_true, y_pred):
    mse_loss = tf.reduce_mean(tf.square(y_true - y_pred))
    W = autoencoder.get_layer('encoded').kernel
    contractive = K.sum(K.square(W))
    return mse_loss + 1e-4 * contractive

# Compile the autoencoder
autoencoder.compile(optimizer='adam', loss=contractive_loss)
autoencoder.fit(data, data, epochs=50, batch_size=32, shuffle=True)

# Build the encoder model
```

```
encoder = Model(input_data, encoded)

compressed_data = encoder.predict(data)


# Build the decoder model

encoded_input = Input(shape=(2,))

decoder_layer = autoencoder.get_layer('decoded')(encoded_input)

decoder = Model(encoded_input, decoder_layer)


# Use the decoder to reconstruct the data from the compressed data

reconstructed_data = decoder.predict(compressed_data)


print("Original Data Shape:", data.shape)

print("Compressed Data Shape:", compressed_data.shape)

print("Reconstructed Data Shape:", reconstructed_data.shape)
```

OUTPUT:

Undercomplete :

```
Original Data Shape: (1000, 5)

Compressed Data Shape: (1000, 2)

Reconstructed Data Shape: (1000, 5)
```

Contractive :

```
Original Data Shape: (1000, 5)

Compressed Data Shape: (1000, 2)

Reconstructed Data Shape: (1000, 5)
```

RESULT:

Thus python program to implement different types of autoencoders has been successfully executed and verified.

Ex No.8	Solve a real world problem using CBM

AIM:

To solve a real world problem using CBM.

ALGORITHM:

1 **Define Data and Weights:**

- Define the smartphone options and their scores for each factor (e.g., price, performance, battery life, etc.).
- Define the weights for each factor, indicating their relative importance.

2 **Create a DataFrame:**

- Use the pandas library to create a DataFrame from the defined smartphone data. Each factor (price, performance, etc.) will be represented as columns in the DataFrame.

3 **Invert the 'Price' Scores:**

- Since a lower price is better, invert the 'price' scores by subtracting them from 10. This makes higher numbers represent better prices.

4 **Calculate Weighted Scores:**

- Multiply the score for each factor by its respective weight. This gives more importance to factors with higher weights.

5 **Compute Total Scores:**

- Sum the weighted scores across all factors for each smartphone. This gives a final score for each phone, considering the importance of each factor.

6 **Rank the Smartphones:**

- Sort the smartphones by their total weighted scores in descending order to rank them from best to worst. Display the ranked list.

PROGRAM:

```
import pandas as pd


# Define the smartphone options and scores for each factor (price, performance, etc.)
```

```
data = {
    "smartphone": ["Phone A", "Phone B", "Phone C", "Phone D", "Phone E"],
    "price": [8, 7, 5, 6, 9],  # Lower price is better
    "performance": [7, 9, 6, 8, 5],
    "battery_life": [9, 8, 7, 5, 8],
    "camera_quality": [6, 9, 7, 8, 7],
    "user_reviews": [8, 7, 9, 6, 7]
}

# Define the weights for each factor
weights = {
    "price": 5,
    "performance": 8,
    "battery_life": 7,
    "camera_quality": 6,
    "user_reviews": 4
}

# Create a DataFrame from the data
df = pd.DataFrame(data)
# Invert the 'price' scores since lower price is better
df['price'] = 10 - df['price']
# Calculate the weighted total score for each smartphone
for factor in weights:
    df[factor] = df[factor] * weights[factor]
# Sum up the weighted scores for each smartphone
df['total_score'] = df[list(weights.keys())].sum(axis=1)
# Sort by total score to rank the smartphones
df_sorted = df[['smartphone', 'total_score']].sort_values(by='total_score', ascending=False)
```

```
# Display the ranked smartphones
print(df_sorted)
```

OUTPUT:

```
  smartphone  total_score
1    Phone B         225
2    Phone C         200
0    Phone A         197
3    Phone D         191
4    Phone E         171
```

RESULT:

Thus python program to solve a real world problem using CBM has been successfully executed and verified.