**NANDHA ENGINEERING COLLEGE (AUTONOMOUS), ERODE**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**DEEP LEARNING**

**ASSIGNMENT I**

**ACADEMIC YEAR : 2024-2025 CLASS : III - CSE –‘B’**

**MARKS : 20 marks SEM : V**

**TEAM 6:(22CS080, 22CS082,22CS081,22CS079)**

|  |  |  |
| --- | --- | --- |
| **S.No** | **QUESTION** | **Marks** |
| 1 | Develop a neural network to recommend products or content  to users based on their past behavior and preferences,  enhancing user engagement and satisfaction. | 10 |
| 2 | Develop an autoencoder for generating text sequences.  the goal is to train the model to capture semantic and syntactic patterns in text data and generate coherent sequences. | 10 |

**Faculty signature Student signature**

1. **Develop a neural network to recommend products or content to users based on their past behavior and preferences,enhancing user engagement and satisfaction**.

import numpy as np

import pandas as pd

from sklearn.model\_ selection import train\_test\_split

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Embedding, Flatten, Concatenate, Dense, Dropout

from tensorflow.keras.optimizers import Adam

num\_users = 1000 # Number of unique users

num\_items = 500 # Number of unique items

interaction\_size = 10000 # Number of interactions (user-item pairs)

# Randomly generate user, item pairs and a rating/preference score

user\_ids = np.random.randint(0, num\_users, interaction\_size)

item\_ids = np.random.randint(0, num\_items, interaction\_size)

ratings = np.random.randint(1, 6, interaction\_size) # Rating scale from 1 to 5

# Create a DataFrame to store the dataset

df = pd.DataFrame({'user\_id': user\_ids, 'item\_id': item\_ids, 'rating': ratings})

# Split the dataset into training and testing

train, test = train\_test\_split(df, test\_size=0.2, random\_state=42)

# Set hyperparameters

embedding\_size = 50 # Size of the embedding for users and items

dropout\_rate = 0.3

# Inputs

user\_input = Input(shape=(1,), name='user\_input')

item\_input = Input(shape=(1,), name='item\_input')

# User and item embedding layers

user\_embedding = Embedding(input\_dim=num\_users, output\_dim=embedding\_size, name='user\_embedding')(user\_input)

item\_embedding = Embedding(input\_dim=num\_items, output\_dim=embedding\_size, name='item\_embedding')(item\_input)

# Flatten the embeddings

user\_flatten = Flatten()(user\_embedding)

item\_flatten = Flatten()(item\_embedding)

# Concatenate user and item embeddings

concat = Concatenate()([user\_flatten, item\_flatten])

# Fully connected layers

dense1 = Dense(128, activation='relu')(concat)

dropout1 = Dropout(dropout\_rate)(dense1)

dense2 = Dense(64, activation='relu')(dropout1)

dropout2 = Dropout(dropout\_rate)(dense2)

# Output layer (predicting the rating)

output = Dense(1, activation='linear')(dropout2)

# Create the model

model = Model(inputs=[user\_input, item\_input], outputs=output)

model.compile(optimizer=Adam(learning\_rate=0.001), loss='mean\_squared\_error')

# Summary of the model

model.summary()

output:

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃ **Connected to** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩

│ user\_input (InputLayer) │ (None, 1) │ 0 │ - │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ item\_input (InputLayer) │ (None, 1) │ 0 │ - │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ user\_embedding (Embedding) │ (None, 1, 50) │ 50,000 │ user\_input[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ item\_embedding (Embedding) │ (None, 1, 50) │ 25,000 │ item\_input[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ flatten\_2 (Flatten) │ (None, 50) │ 0 │ user\_embedding[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ flatten\_3 (Flatten) │ (None, 50) │ 0 │ item\_embedding[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ concatenate\_1 (Concatenate) │ (None, 100) │ 0 │ flatten\_2[0][0], │

│ │ │ │ flatten\_3[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ dense\_4 (Dense) │ (None, 128) │ 12,928 │ concatenate\_1[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ dropout (Dropout) │ (None, 128) │ 0 │ dense\_4[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ dense\_5 (Dense) │ (None, 64) │ 8,256 │ dropout[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ dropout\_1 (Dropout) │ (None, 64) │ 0 │ dense\_5[0][0] │

├───────────────────────────────┼───────────────────────────┼─────────────────┼────────────────────────────┤

│ dense\_6 (Dense) │ (None, 1) │ 65 │ dropout\_1[0][0] │

└───────────────────────────────┴───────────────────────────┴─────────────────┴────────────────────────────┘

**Total params:** 96,249 (375.97 KB)

**Trainable params:** 96,249 (375.97 KB)

**Non-trainable params:** 0 (0.00 B)

# Prepare training and testing data

train\_user\_input = train['user\_id'].values

train\_item\_input = train['item\_id'].values

train\_ratings = train['rating'].values

test\_user\_input = test['user\_id'].values

test\_item\_input = test['item\_id'].values

test\_ratings = test['rating'].values

# Train the model

history = model.fit(

[train\_user\_input, train\_item\_input], train\_ratings,

validation\_data=([test\_user\_input, test\_item\_input], test\_ratings),

epochs=10, batch\_size=64

)

output:

Epoch 1/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 5s 10ms/step - loss: 6.5486 - val\_loss: 2.0183

Epoch 2/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - loss: 2.0976 - val\_loss: 2.0585

Epoch 3/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 1.9531 - val\_loss: 2.1333

Epoch 4/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - loss: 1.9037 - val\_loss: 2.1459

Epoch 5/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - loss: 1.8551 - val\_loss: 2.1718

Epoch 6/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.7908 - val\_loss: 2.2687

Epoch 7/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 8ms/step - loss: 1.7643 - val\_loss: 2.3100

Epoch 8/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.6645 - val\_loss: 2.2296

Epoch 9/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.5605 - val\_loss: 2.2742

Epoch 10/10

125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 1.4045 - val\_loss: 2.4556

# Evaluate on the test set

test\_loss = model.evaluate([test\_user\_input, test\_item\_input], test\_ratings)

print(f"Test Loss (MSE): {test\_loss}")

63/63 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - loss: 2.5114

Test Loss (MSE): 2.4556198120117188

# Predict ratings for a user-item pair

user\_id = 10 # Example user

item\_id = 50 # Example item

predicted\_rating = model.predict([np.array([user\_id]), np.array([item\_id])])

print(f"Predicted rating for user {user\_id} on item {item\_id}: {predicted\_rating[0][0]}")

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 373ms/step

Predicted rating for user 10 on item 50: 1.884240746498108

**2. Develop an autoencoder for generating text sequences. The goal is to train the model to capture semantic and syntactic patterns in text data and generate coherent sequences.**

**AIM:**

The aim of this project is to develop an autoencoder model capable of generating coherent text sequences. The autoencoder is trained to capture both semantic and syntactic patterns in text data.

**STEP-BY-STEP ALGORITHM FOR DENOISING AUTOENCODER:**

**Step 1: Data Preparation**

1. **Collect Text Data**:
   * Gather a dataset consisting of sentences or text sequences. This could be any corpus relevant to the language model you want to train (e.g., news articles, books, etc.).
2. **Text Preprocessing**:
   * **Tokenization**: Split the text into individual tokens (words or characters).
   * **Vocabulary Creation**: Create a dictionary mapping each unique token to an integer index.
   * **Padding**: Ensure that all sequences have the same length by padding shorter sequences with a special token (e.g., <PAD>).
   * **Encoding**: Convert tokens in each sequence to their corresponding integer indices using the vocabulary.

**Step 2: Define the Model Architecture**

1. **Input Layer**:
   * Define an input layer that takes sequences of fixed length (e.g., 10 tokens).
2. **Embedding Layer**:
   * Add an embedding layer to transform input tokens into dense vectors of fixed size. This helps in capturing semantic relationships between words.
3. **Encoder**:
   * Use a Recurrent Neural Network (RNN) or a variant like LSTM (Long Short-Term Memory) to process the input sequence. The encoder compresses the sequence into a fixed-size vector (latent space representation).
4. **Latent Space Representation**:
   * The output from the encoder is the latent space, which contains a compressed representation of the input sequence. This vector will be used to reconstruct the sequence.
5. **Decoder**:
   * Implement a decoder, which is another RNN or LSTM, that takes the latent space vector and tries to reconstruct the original sequence. The decoder generates one token at a time until the entire sequence is reconstructed.
6. **Output Layer**:
   * Add a TimeDistributed Dense layer with a softmax activation function to produce a probability distribution over the vocabulary for each token position in the sequence.

**Step 3: Compile the Model**

1. **Loss Function**:
   * Use categorical\_crossentropy as the loss function, which is suitable for sequence prediction tasks where the output is a probability distribution over the vocabulary.
2. **Optimizer**:
   * Choose an optimizer like adam, which adapts the learning rate and converges quickly.
3. **Metrics**:
   * Include accuracy as a metric to monitor how well the model is performing during training.

**Step 4: Train the Model**

1. **Prepare Training Data**:
   * Split the data into training and validation sets.
   * Convert the target sequences to one-hot encoded format if using categorical\_crossentropy.
2. **Fit the Model**:
   * Train the model using the training data. Set the number of epochs (e.g., 10) and batch size (e.g., 32).
   * Monitor the training loss and accuracy to ensure the model is learning effectively.
3. **Validation**:
   * Evaluate the model on the validation set to check for overfitting and to ensure generalization.

**Step 5: Evaluate and Fine-Tune the Model**

1. **Test the Model**:
   * Generate new sequences by feeding the encoder with input sequences and using the decoder to generate output sequences.
   * Compare the generated sequences with the original to assess quality.
2. **Fine-Tuning**:
   * Adjust hyperparameters (e.g., learning rate, batch size) and retrain if necessary to improve performance.
   * Experiment with different model architectures, such as using bidirectional LSTM, or increasing the latent space dimension.

**Step 6: Save the Model**

1. **Model Saving**:
   * Save the trained model and its weights to a file for future use.
2. **Model Loading**:
   * Implement code to load the model later for generating new sequences or continuing training.

**Step 7: Generate New Text Sequences**

1. **Encoding Input Text**:
   * Use the encoder to convert input text sequences to their latent space representations.
2. **Decoding**:
   * Use the decoder to generate new sequences from the latent representations.
   * Optionally, implement sampling techniques to introduce variability in the generated sequences.
3. **Post-Processing**:
   * Convert the generated sequences back from integer indices to text format.
   * Remove padding or special tokens to produce readable text.

**CODE:**

**import tensorflow as tf**

**from tensorflow.keras.models import Model**

**from tensorflow.keras.layers import Input, LSTM, Dense, RepeatVector, TimeDistributed**

**# Hyperparameters**

**input\_dim = 100 # Vocabulary size (e.g., 100 unique tokens)**

**embedding\_dim = 64 # Embedding dimension**

**latent\_dim = 256 # Latent space dimension**

**timesteps = 10 # Number of tokens in each sequence**

**# Encoder**

**encoder\_inputs = Input(shape=(timesteps,))**

**x = tf.keras.layers.Embedding(input\_dim, embedding\_dim)(encoder\_inputs)**

**x, state\_h, state\_c = LSTM(latent\_dim, return\_state=True)(x)**

**encoder\_states = [state\_h, state\_c]**

**# Decoder**

**decoder\_inputs = RepeatVector(timesteps)(state\_h) # Use the last state of the encoder as the initial input**

**x = LSTM(latent\_dim, return\_sequences=True)(decoder\_inputs, initial\_state=encoder\_states)**

**decoder\_outputs = TimeDistributed(Dense(input\_dim, activation='softmax'))(x)**

**# Autoencoder**

**autoencoder = Model(encoder\_inputs, decoder\_outputs)**

**# Compile model**

**autoencoder.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])**

**# Summarize model**

**autoencoder.summary()**

**# Dummy data for training (Replace this with real text sequences)**

**import numpy as np**

**X\_train = np.random.randint(0, input\_dim, size=(1000, timesteps)) # Example training data**

**y\_train = tf.keras.utils.to\_categorical(X\_train, num\_classes=input\_dim) # One-hot encoding of sequences**

**# Train the model**

**autoencoder.fit(X\_train, y\_train, epochs=10, batch\_size=32)**

**# To save the model**

**autoencoder.save('autoencoder\_model.h5')**

**# To load the model later**

**# autoencoder = tf.keras.models.load\_model('autoencoder\_model.h5')**

**OUTPUT:**

**Model: "functional"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃ **Connected to** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━┩

│ input\_layer (InputLayer) │ (None, 10) │ 0 │ - │

├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤

│ embedding (Embedding) │ (None, 10, 64) │ 6,400 │ input\_layer[0][0] │

├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤

│ lstm (LSTM) │ [(None, 256), (None, │ 328,704 │ embedding[0][0] │

│ │ 256), (None, 256)] │ │ │

├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤

│ repeat\_vector │ (None, 10, 256) │ 0 │ lstm[0][1] │

│ (RepeatVector) │ │ │ │

├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤

│ lstm\_1 (LSTM) │ (None, 10, 256) │ 525,312 │ repeat\_vector[0][0], │

│ │ │ │ lstm[0][1], lstm[0][2] │

├───────────────────────────┼────────────────────────┼────────────────┼────────────────────────┤

│ time\_distributed │ (None, 10, 100) │ 25,700 │ lstm\_1[0][0] │

│ (TimeDistributed) │ │ │ │

└───────────────────────────┴────────────────────────┴────────────────┴────────────────────────┘

**Total params:** 886,116 (3.38 MB)

**Trainable params:** 886,116 (3.38 MB)

**Non-trainable params:** 0 (0.00 B)

Epoch 1/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **9s** 90ms/step - accuracy: 0.0221 - loss: 4.6007

Epoch 2/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **6s** 129ms/step - accuracy: 0.0403 - loss: 4.4696

Epoch 3/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **4s** 89ms/step - accuracy: 0.0706 - loss: 4.1437

Epoch 4/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **4s** 109ms/step - accuracy: 0.0980 - loss: 3.8822

Epoch 5/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **4s** 127ms/step - accuracy: 0.1095 - loss: 3.6880

Epoch 6/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **4s** 115ms/step - accuracy: 0.1305 - loss: 3.5163

Epoch 7/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **5s** 106ms/step - accuracy: 0.1476 - loss: 3.3688

Epoch 8/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **5s** 108ms/step - accuracy: 0.1559 - loss: 3.2863

Epoch 9/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **5s** 140ms/step - accuracy: 0.1763 - loss: 3.1222

Epoch 10/10

**32/32** ━━━━━━━━━━━━━━━━━━━━ **4s** 110ms/step - accuracy: 0.1884 - loss: 3.0010

**RESULT:**

Hence ,the goal is to train the model to capture semantic and syntactic patterns in text data and generate coherent sequences.