**PRACTICAL NO.: - 6**

**AIM** : Backpropagation in Neural Networks using Numpy.

**INPUT**

|  |
| --- |
| import numpy as np  # Define the sigmoid activation function and its derivative  def sigmoid(x):      return 1 / (1 + np.exp(-x))  def sigmoid\_derivative(x):      return x \* (1 - x)  # Define the neural network architecture  input\_size = 2  hidden\_size = 3  output\_size = 1  learning\_rate = 0.1  # Initialize weights and biases  np.random.seed(0)  input\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  output\_data = np.array([[0], [1], [1], [0]])  # Weights and biases initialization  weights\_input\_hidden = np.random.uniform(size=(input\_size, hidden\_size))  biases\_hidden = np.zeros((1, hidden\_size))  weights\_hidden\_output = np.random.uniform(size=(hidden\_size, output\_size))  biases\_output = np.zeros((1, output\_size))  # Training loop  epochs = 10000  for epoch in range(epochs):      # Forward pass      hidden\_layer\_input = np.dot(input\_data, weights\_input\_hidden) + biases\_hidden      hidden\_layer\_output = sigmoid(hidden\_layer\_input)      output\_layer\_input = np.dot(hidden\_layer\_output, weights\_hidden\_output) + biases\_output      predicted\_output = sigmoid(output\_layer\_input)      # Compute the loss      loss = 0.5 \* np.mean((predicted\_output - output\_data) \*\* 2)      # Backpropagation      output\_error = output\_data - predicted\_output      output\_delta = output\_error \* sigmoid\_derivative(predicted\_output)      hidden\_layer\_error = output\_delta.dot(weights\_hidden\_output.T)      hidden\_layer\_delta = hidden\_layer\_error \* sigmoid\_derivative(hidden\_layer\_output)      # Update weights and biases      weights\_hidden\_output += hidden\_layer\_output.T.dot(output\_delta) \* learning\_rate      biases\_output += np.sum(output\_delta, axis=0, keepdims=True) \* learning\_rate      weights\_input\_hidden += input\_data.T.dot(hidden\_layer\_delta) \* learning\_rate      biases\_hidden += np.sum(hidden\_layer\_delta, axis=0, keepdims=True) \* learning\_rate      if epoch % 1000 == 0:          print(f"Epoch {epoch}, Loss: {loss}")  print("Training completed.") |

**Output**

|  |
| --- |
| Epoch 0, Loss: 0.1714629896398691  Epoch 1000, Loss: 0.12310016740452781  Epoch 2000, Loss: 0.11202152064381576  Epoch 3000, Loss: 0.08233179820965757  Epoch 4000, Loss: 0.02650020295068723  Epoch 5000, Loss: 0.008485970497104004  Epoch 6000, Loss: 0.004449333904478412  Epoch 7000, Loss: 0.002906299373724993  Epoch 8000, Loss: 0.002124174486298295  Epoch 9000, Loss: 0.0016598326873063268  Training completed. |

**PRACTICAL NO.: - 7**

**AIM** : Neural Recommender Systems with Implicit Feedback and the Triplet Loss

**INPUT**

|  |
| --- |
| import numpy as np  import tensorflow as tf  from tensorflow import keras  from tensorflow.keras import layers  from sklearn.model\_selection import train\_test\_split  # Load your dataset or create a toy dataset  # For this example, we'll use a random toy dataset  num\_users = 100  num\_items = 50  embedding\_dim = 50  # Generate toy implicit feedback data  np.random.seed(0)  user\_ids = np.random.randint(0, num\_users, 1000)  positive\_items = np.random.randint(0, num\_items, 1000)  # Create triplets (user, positive\_item, negative\_item)  def create\_triplets(user\_ids, positive\_items, num\_items):  triplets = []  for user, positive\_item in zip(user\_ids, positive\_items):  negative\_item = np.random.randint(0, num\_items)  while negative\_item == positive\_item:  negative\_item = np.random.randint(0, num\_items)  triplets.append([user, positive\_item, negative\_item])  return np.array(triplets)  triplets = create\_triplets(user\_ids, positive\_items, num\_items)  # Split the data into training and validation sets  train\_triplets, val\_triplets = train\_test\_split(triplets, test\_size=0.1)  # Define the neural network model  user\_input = keras.Input(shape=(1,))  positive\_item\_input = keras.Input(shape=(1,))  negative\_item\_input = keras.Input(shape=(1,))  embedding\_layer = layers.Embedding(num\_users, embedding\_dim, input\_length=1)  user\_embedding = embedding\_layer(user\_input)  positive\_item\_embedding = embedding\_layer(positive\_item\_input)  negative\_item\_embedding = embedding\_layer(negative\_item\_input)  # Define the triplet loss layer as a custom layer  class TripletLossLayer(layers.Layer):  def \_\_init\_\_(self, margin=0.2, \*\*kwargs):  super(TripletLossLayer, self).\_\_init\_\_(\*\*kwargs)  self.margin = margin  def call(self, inputs):  user\_embedding, positive\_item\_embedding, negative\_item\_embedding = inputs  positive\_distance = tf.reduce\_sum(tf.square(user\_embedding - positive\_item\_embedding), axis=1)  negative\_distance = tf.reduce\_sum(tf.square(user\_embedding - negative\_item\_embedding), axis=1)  loss = tf.maximum(0.0, positive\_distance - negative\_distance + self.margin)  return loss  triplet\_loss\_layer = TripletLossLayer()([user\_embedding, positive\_item\_embedding, negative\_item\_embedding])  model = keras.Model(inputs=[user\_input, positive\_item\_input, negative\_item\_input], outputs=triplet\_loss\_layer)  # Compile the model  model.compile(optimizer="adam", loss="mean\_absolute\_error")  # Training  batch\_size = 64  num\_epochs = 10  model.fit(  [train\_triplets[:, 0], train\_triplets[:, 1], train\_triplets[:, 2]],  np.zeros(len(train\_triplets)),  batch\_size=batch\_size,  epochs=num\_epochs,  validation\_data=(  [val\_triplets[:, 0], val\_triplets[:, 1], val\_triplets[:, 2]],  np.zeros(len(val\_triplets)),  ),  ) |

**Output**

|  |
| --- |
| Epoch 1/10  15/15 [==============================] - 1s 24ms/step - loss: 0.2000 - val\_loss: 0.2000  Epoch 2/10  15/15 [==============================] - 0s 9ms/step - loss: 0.1998 - val\_loss: 0.2000  Epoch 3/10  15/15 [==============================] - 0s 8ms/step - loss: 0.1997 - val\_loss: 0.2000  Epoch 4/10  15/15 [==============================] - 0s 5ms/step - loss: 0.1995 - val\_loss: 0.2000  Epoch 5/10  15/15 [==============================] - 0s 4ms/step - loss: 0.1993 - val\_loss: 0.2000  Epoch 6/10  15/15 [==============================] - 0s 5ms/step - loss: 0.1991 - val\_loss: 0.2000  Epoch 7/10  15/15 [==============================] - 0s 5ms/step - loss: 0.1988 - val\_loss: 0.2000  Epoch 8/10  15/15 [==============================] - 0s 5ms/step - loss: 0.1985 - val\_loss: 0.2000  Epoch 9/10  15/15 [==============================] - 0s 4ms/step - loss: 0.1981 - val\_loss: 0.2000  Epoch 10/10  15/15 [==============================] - 0s 5ms/step - loss: 0.1977 - val\_loss: 0.2000  <keras.src.callbacks.History at 0x7eb78a2e9540> |

**PRACTICAL NO.: - 8**

**AIM :** Fully Convolutional Neural Networks.

**INPUT**

|  |
| --- |
| import tensorflow as tf  from tensorflow import keras  from tensorflow.keras import layers  # Define a basic Fully Convolutional Neural Network  def create\_fully\_convolutional\_network(input\_shape, num\_classes):      model = keras.Sequential()      # Encoder      model.add(layers.Input(shape=input\_shape))      model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))      model.add(layers.MaxPooling2D((2, 2), strides=(2, 2)))      # Middle      model.add(layers.Conv2D(128, (3, 3), activation='relu', padding='same'))      # Decoder      model.add(layers.UpSampling2D((2, 2)))      model.add(layers.Conv2D(num\_classes, (1, 1), activation='softmax', padding='valid'))      return model  # Define input shape and number of classes  input\_shape = (256, 256, 3)  # Input image dimensions (e.g., 256x256 RGB image)  num\_classes = 21  # Number of segmentation classes  # Create the FCN model  fcn\_model = create\_fully\_convolutional\_network(input\_shape, num\_classes)  # Compile the model with an appropriate loss and optimizer  fcn\_model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])  # Summary of the model architecture  fcn\_model.summary() |

**Output**

|  |
| --- |
| Model: "sequential"  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type) Output Shape Param #  =================================================================  conv2d (Conv2D) (None, 256, 256, 64) 1792    max\_pooling2d (MaxPooling2 (None, 128, 128, 64) 0  D)    conv2d\_1 (Conv2D) (None, 128, 128, 128) 73856    up\_sampling2d (UpSampling2 (None, 256, 256, 128) 0  D)    conv2d\_2 (Conv2D) (None, 256, 256, 21) 2709    =================================================================  Total params: 78357 (306.08 KB)  Trainable params: 78357 (306.08 KB)  Non-trainable params: 0 (0.00 Byte) |

**PRACTICAL NO.: - 9**

**AIM :** ConvNets for Classification and Localization.

**INPUT**

|  |
| --- |
| import tensorflow as tf  from tensorflow import keras  from tensorflow.keras import layers  # Define a Localization CNN model for classification and localization  def create\_localization\_cnn(input\_shape, num\_classes, num\_coords):  input\_tensor = layers.Input(shape=input\_shape)  # Convolutional layers for feature extraction  x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(input\_tensor)  x = layers.MaxPooling2D((2, 2))(x)  x = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)  x = layers.MaxPooling2D((2, 2))(x)  x = layers.Conv2D(256, (3, 3), activation='relu', padding='same')(x)  x = layers.MaxPooling2D((2, 2))(x)  # Flatten the feature map for classification  flat = layers.Flatten()(x)  # Classification head  classification = layers.Dense(num\_classes, activation='softmax', name='classification')(flat)  # Localization head  localization = layers.Dense(num\_coords, activation='linear', name='localization')(flat)  return keras.Model(inputs=input\_tensor, outputs=[classification, localization])  # Define input shape, number of classes, and number of coordinates (e.g., x, y)  input\_shape = (224, 224, 3) # Input image dimensions (e.g., 224x224 RGB image)  num\_classes = 10 # Number of classes for classification  num\_coords = 4 # Number of coordinates (e.g., x, y, width, height) for localization  # Create the Localization CNN model  localization\_cnn = create\_localization\_cnn(input\_shape, num\_classes, num\_coords)  # Compile the model with appropriate loss functions and optimizers  localization\_cnn.compile(  optimizer='adam',  loss={'classification': 'categorical\_crossentropy', 'localization': 'mean\_squared\_error'},  loss\_weights={'classification': 1.0, 'localization': 1.0} )  # Summary of the model architecture  localization\_cnn.summary() |

**Output**

|  |
| --- |
| Model: "model\_2"  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  Layer (type) Output Shape Param # Connected to  ==================================================================================================  input\_35 (InputLayer) [(None, 224, 224, 3)] 0 []    conv2d\_3 (Conv2D) (None, 224, 224, 64) 1792 ['input\_35[0][0]']    max\_pooling2d\_1 (MaxPoolin (None, 112, 112, 64) 0 ['conv2d\_3[0][0]']  g2D)    conv2d\_4 (Conv2D) (None, 112, 112, 128) 73856 ['max\_pooling2d\_1[0][0]']    max\_pooling2d\_2 (MaxPoolin (None, 56, 56, 128) 0 ['conv2d\_4[0][0]']  g2D)    conv2d\_5 (Conv2D) (None, 56, 56, 256) 295168 ['max\_pooling2d\_2[0][0]']    max\_pooling2d\_3 (MaxPoolin (None, 28, 28, 256) 0 ['conv2d\_5[0][0]']  g2D)    flatten (Flatten) (None, 200704) 0 ['max\_pooling2d\_3[0][0]']    classification (Dense) (None, 10) 2007050 ['flatten[0][0]']    localization (Dense) (None, 4) 802820 ['flatten[0][0]']    ==================================================================================================  Total params: 3180686 (12.13 MB)  Trainable params: 3180686 (12.13 MB)  Non-trainable params: 0 (0.00 Byte) |

**PRACTICAL NO.: - 10**

**AIM :** Text Classification and Word Vectors.

**INPUT**

|  |
| --- |
| import numpy as np  from gensim.models import Word2Vec  from sklearn.model\_selection import train\_test\_split  from sklearn.feature\_extraction.text import TfidfVectorizer  from sklearn.naive\_bayes import MultinomialNB  from sklearn.metrics import accuracy\_score  # Step 1: Sample data (replace with your dataset)  positive\_reviews = ["This movie is fantastic!", "I loved it.", "Great film."]  negative\_reviews = ["Terrible movie.", "Hated it.", "Awful film."]  # Label the data: 1 for positive and 0 for negative  labels = [1] \* len(positive\_reviews) + [0] \* len(negative\_reviews)  reviews = positive\_reviews + negative\_reviews  # Step 2: Load pre-trained Word2Vec embeddings  # Download pre-trained Word2Vec embeddings from a source like GloVe or Word2Vec  # In this example, we'll use dummy Word2Vec embeddings for illustration purposes.  word\_vectors = {      "this": np.array([0.1, 0.2, 0.3]),      "movie": np.array([0.2, 0.3, 0.4]),      "is": np.array([0.3, 0.4, 0.5]),      "fantastic!": np.array([0.4, 0.5, 0.6]),      "terrible": np.array([-0.3, -0.4, -0.5]),      "hated": np.array([-0.4, -0.5, -0.6]),  }  # Step 3: Convert text data to numerical representations using word vectors  def text\_to\_vector(text):      words = text.split()      vectors = [word\_vectors[word] for word in words if word in word\_vectors]      if not vectors:          return np.zeros(3)  # Return a zero vector if no known words are present      return np.mean(vectors, axis=0)  X = np.array([text\_to\_vector(review) for review in reviews])  # Step 4: Split the data and train a classification model (e.g., Naive Bayes)  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, labels, test\_size=0.2, random\_state=42)  # Train a simple Naive Bayes classifier  classifier = MultinomialNB()  classifier.fit(X\_train, y\_train)  # Make predictions  y\_pred = classifier.predict(X\_test)  # Calculate accuracy  accuracy = accuracy\_score(y\_test, y\_pred)  print(f"Accuracy: {accuracy}") |

**Output**

|  |
| --- |
| Accuracy: 0.0 |

**PRACTICAL NO.: - 11**

**AIM:** Character Level Language Model (GPU required).

**INPUT**

|  |
| --- |
| import torch  import torch.nn as nn  import torch.optim as optim  # Define the data  text = "Your training data goes here..."  # Create a character-level vocabulary  vocab = set(text)  vocab\_size = len(vocab)  char\_to\_idx = {char: idx for idx, char in enumerate(vocab)}  idx\_to\_char = {idx: char for char, idx in char\_to\_idx.items()}  # Hyperparameters  hidden\_size = 100  num\_layers = 2  learning\_rate = 0.01  num\_epochs = 100  # Define the model  class CharRNN(nn.Module):      def \_\_init\_\_(self, input\_size, hidden\_size, num\_layers):          super(CharRNN, self).\_\_init\_\_()  # Corrected super call          self.hidden\_size = hidden\_size          self.num\_layers = num\_layers          self.embedding = nn.Embedding(input\_size, hidden\_size)          self.rnn = nn.LSTM(hidden\_size, hidden\_size, num\_layers, batch\_first=True)          self.fc = nn.Linear(hidden\_size, input\_size)      def forward(self, x, hidden):          out = self.embedding(x)          out, hidden = self.rnn(out, hidden)          out = self.fc(out)          return out, hidden  model = CharRNN(vocab\_size, hidden\_size, num\_layers)  criterion = nn.CrossEntropyLoss()  optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  # Training loop  for epoch in range(num\_epochs):      hidden = (torch.zeros(num\_layers, 1, hidden\_size), torch.zeros(num\_layers, 1, hidden\_size))      total\_loss = 0        for i in range(0, len(text) - 100, 100):          input\_seq = text[i:i+100]          target\_seq = text[i+1:i+101]            input\_tensor = torch.tensor([char\_to\_idx[c] for c in input\_seq], dtype=torch.long)          target\_tensor = torch.tensor([char\_to\_idx[c] for c in target\_seq], dtype=torch.long)            output, hidden = model(input\_tensor.view(1, -1), hidden)          loss = criterion(output.view(1, -1, vocab\_size), target\_tensor.view(1, -1))            optimizer.zero\_grad()          loss.backward()          optimizer.step()            total\_loss += loss.item()      if (epoch + 1) % 10 == 0:          print(f'Epoch [{epoch+1}/{num\_epochs}], Loss: {total\_loss:.4f}')  # Generating text  with torch.no\_grad():      seed\_text = "Your seed text..."      generated\_text = seed\_text      hidden = torch.zeros(num\_layers, 1, hidden\_size), torch.zeros(num\_layers, 1, hidden\_size)        for char in seed\_text:          if char in char\_to\_idx:              input\_char = torch.tensor(char\_to\_idx[char], dtype=torch.long)              output, hidden = model(input\_char.view(1, -1), hidden)          else:              # Handle characters not in the vocabulary, such as whitespace or special characters.              # You can choose to skip or replace them as needed.              continue        for \_ in range(1000):          output\_softmax = torch.softmax(output.view(-1), dim=0)          predicted\_idx = torch.multinomial(output\_softmax, 1)          predicted\_char = idx\_to\_char[predicted\_idx.item()]          generated\_text += predicted\_char          input\_char = predicted\_idx          output, hidden = model(input\_char.view(1, -1), hidden)        print(generated\_text) |

**Output**

|  |
| --- |
| Epoch [10/100], Loss: 0.0000  Epoch [20/100], Loss: 0.0000  Epoch [30/100], Loss: 0.0000  Epoch [40/100], Loss: 0.0000  Epoch [50/100], Loss: 0.0000  Epoch [60/100], Loss: 0.0000  Epoch [70/100], Loss: 0.0000  Epoch [80/100], Loss: 0.0000  Epoch [90/100], Loss: 0.0000  Epoch [100/100], Loss: 0.0000  Your seed text... Yed iaeYegea s.tss.un uson.hhgitneti.gou tssa.u gidoi aaed nadutahYY.ingtsui.ditYase.sentggnnag .groeYu.YheeiYrasgnitengau. au.egsrite tetnnYo.ehr.asrYYnYhYe.ohrisgheae.ihYiidsnYsti r drhaians.uouYdssgtrY.dr.isggeodsauYud.tnhteea edYeuduYin.Y uggYdhoar ndsttitie.odtstaYnsrnarsudunohhh hYnYi Ygiirnru.ni gthaon.o.rne g tgnnr r eiairugenhtrn ttadsrtYtgdesoguseushhoe ha.sgnutndYgdu r asYaoaoss g ..Yinn idi ha .d.tnus .idsei.. sa oaYe.dr i.eds.nu. gauhse aenniohooeursururreaoghiueoih.uguaiginuedatgtdd.ennnid sei.rdnd seY rr.ugns.suda.tishd doru rianhah.dggs.do.iirsneonaYg oshsrtYsndunuusggdd.gtorh rYnedttti.aYY.eh.eoiYsgngeYgahensuhhotgidt.uetdtsstneusotdnuaeanYasao tgneehihtYd. .ieshnYdthtin.nshnrdth .nhdoinghhttuu.dsuhtggoguiud.ogoYgagd...egtsioig.Yngutre hds gdrnYs dgoano dhuge..n...otnYtghegeeooda.uhr. rdnogahrouho.a...tt.shg. iisassireurrsagadiou. snaeot dt n diegg.naasa usihhton .oh ds adrrushn.nderndiYggYe oudghnYrgrhdgositnYusYtYsunddosnhdsgguun urho..hasagsrearootnegounaon nsniet |