Experiment 1: Solving XOR Problem using Multilayer Perceptron

Aim:

To demonstrate the ability of a multilayer perceptron to solve the XOR problem, highlighting the power of deep architectures in handling non-linear relationships.

Algorithm:

STEP 1: Prepare a dataset with input features (0 or 1) and corresponding XOR outputs.

STEP 2: Design a multilayer perceptron with input, hidden, and output layers.

STEP 3: Randomly initialize the network's weights and biases.

STEP 4: Choose a loss function (e.g., mean squared error) and an optimization algorithm (e.g., stochastic gradient descent).

STEP 5: Train the network using the dataset, adjusting weights and biases iteratively.

STEP 6: Evaluate the trained network's performance on XOR inputs and compare with expected outputs.

Program:

```
import numpy as np

import tensorflow as tf

# Define the XOR dataset

x_train = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y_train = np.array([[0], [1], [1], [0]])

# Build the Multilayer Perceptron model

model = tf.keras.Sequential([

    tf.keras.layers.Dense(2, input_dim=2, activation='sigmoid'),  # Hidden layer

    tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer

])

# Compile the model

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

# Train the model

model.fit(x_train, y_train, epochs=10000, verbose=0)
```

```
# Evaluate the model

loss, accuracy = model.evaluate(x_train, y_train)

# Predict using the trained model

predictions = model.predict(x_train)

print("Model Loss:", loss)

print("Model Accuracy:", accuracy)

print("Predictions:")

for i in range(len(x_train)):

    print(f"Input: {x_train[i]} | Predicted Output: {predictions[i]} | Actual Output: {y_train[i]}")
```

Sample Output:

```
4/4 [==============================] - 0s 5ms/sample - loss: 0.0029 - accuracy: 1.0000

Model Loss: 0.0029488941383951902

Model Accuracy: 1.0

Predictions:

Input: [0 0] | Predicted Output: [0.00200376] | Actual Output: [0]

Input: [0 1] | Predicted Output: [0.99755025] | Actual Output: [1]

Input: [1 0] | Predicted Output: [0.9976636] | Actual Output: [1]

Input: [1 1] | Predicted Output: [0.00268477] | Actual Output: [0]
```

Experiment 2: Implement Character and Digit Recognition using ANN

Aim:

To develop an artificial neural network (ANN) capable of accurately recognizing characters and digits in images, showcasing the effectiveness of neural networks in pattern recognition tasks.

Algorithm:

STEP 1: Prepare a dataset of character and digit images along with their labels.

STEP 2: Design an artificial neural network (ANN) with input, hidden, and output layers.

STEP 3: Initialize weights and biases randomly.

STEP 4: Choose an appropriate activation function and loss function.

STEP 5: Use backpropagation and gradient descent to optimize the network's parameters.

STEP 6: Evaluate the trained ANN's accuracy on a test dataset of character and digit images.

Program :

```
import tensorflow as tf

from tensorflow import keras

from sklearn.model_selection import train_test_split

import numpy as np

# Load the dataset (you may need to replace this with your own dataset)

mnist = keras.datasets.mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Preprocess the data

train_images = train_images / 255.0

test_images = test_images / 255.0

# Build the ANN model

model = keras.Sequential([

    keras.layers.Flatten(input_shape=(28, 28)),  # Flatten the 28x28 input images

    keras.layers.Dense(128, activation='relu'),  # Hidden layer with 128 neurons

    keras.layers.Dense(10, activation='softmax')  # Output layer with 10 classes (digits 0-9)
```

```
])

# Compile the model

model.compile(optimizer='adam',

              loss='sparse_categorical_crossentropy',

              metrics=['accuracy'])

# Split the data for training and validation

train_images, val_images, train_labels, val_labels = train_test_split(train_images,
train_labels, test_size=0.1)

# Train the model

model.fit(train_images, train_labels, epochs=5, validation_data=(val_images, val_labels))

# Evaluate the model on test data

test_loss, test_acc = model.evaluate(test_images, test_labels)

print("\nTest accuracy:", test_acc)

# Make predictions on a sample test image

sample_image = test_images[0]

sample_label = test_labels[0]

predictions = model.predict(np.expand_dims(sample_image, axis=0))

predicted_label = np.argmax(predictions)

print("\nSample Test Image Label:", sample_label)

print("Predicted Label:", predicted_label)
```

Sample Output:

```
Train on 54000 samples, validate on 6000 samples

Epoch 1/5

54000/54000 [==============================] - 4s 77us/sample - loss: 0.2936 -
accuracy: 0.9170 - val_loss: 0.1373 - val_accuracy: 0.9595

Epoch 2/5

54000/54000 [==============================] - 4s 66us/sample - loss: 0.1249 -
accuracy: 0.9640 - val_loss: 0.0970 - val_accuracy: 0.9712
```

Epoch 3/5

54000/54000 [==============================] - 3s 61us/sample - loss: 0.0845 - accuracy: 0.9752 - val_loss: 0.0812 - val_accuracy: 0.9758

Epoch 4/5

54000/54000 [==============================] - 4s 70us/sample - loss: 0.0619 - accuracy: 0.9810 - val_loss: 0.0740 - val_accuracy: 0.9782

Epoch 5/5

54000/54000 [==============================] - 4s 69us/sample - loss: 0.0483 - accuracy: 0.9850 - val_loss: 0.0745 - val_accuracy: 0.9782

10000/10000 [==============================] - 0s 30us/sample - loss: 0.0789 - accuracy: 0.9755

Test accuracy: 0.9755

Sample Test Image Label: 7

Predicted Label: 7

Experiment 3: Implement Analysis of X-ray Image using Autoencoders

Aim:

To apply autoencoders to X-ray images for feature extraction and anomaly detection, illustrating the potential of unsupervised learning in medical image analysis.

Algorithm:

STEP 1: Gather a dataset of X-ray images for analysis.

STEP 2: Design an autoencoder architecture with an encoder and a decoder.

STEP 3: Define a suitable loss function, often using mean squared error.

STEP 4: Train the autoencoder using X-ray images to learn compact representations.

STEP 5: Evaluate the reconstruction quality and encoded representations.

STEP 6: Use encoded representations for tasks like anomaly detection or image denoising.

Program:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.models import Model

import matplotlib.pyplot as plt

# Load a sample X-ray image (you may need to replace this with your own dataset)

x_ray_image = np.random.rand(256, 256)  # Example grayscale image

# Add noise to the image

noisy_x_ray = x_ray_image + np.random.normal(0, 0.1, x_ray_image.shape)

# Normalize the images

x_ray_image = x_ray_image / 255.0

noisy_x_ray = noisy_x_ray / 255.0

# Build the autoencoder architecture

input_layer = Input(shape=(256, 256))

encoded = Dense(128, activation='relu')(input_layer)
```

```
decoded = Dense(256, activation='sigmoid')(encoded)
autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
# Train the autoencoder
autoencoder.fit(noisy_x_ray, x_ray_image, epochs=100)
# Denoise a noisy image using the trained autoencoder
denoised_x_ray = autoencoder.predict(np.expand_dims(noisy_x_ray, axis=0))
# Plot the original, noisy, and denoised images
plt.figure(figsize=(10, 5))
plt.subplot(1, 3, 1)
plt.imshow(x_ray_image, cmap='gray')
plt.title("Original X-ray")
plt.subplot(1, 3, 2)
plt.imshow(noisy_x_ray, cmap='gray')
plt.title("Noisy X-ray")
plt.subplot(1, 3, 3)
plt.imshow(denoised_x_ray[0], cmap='gray')
plt.title("Denoised X-ray")
plt.tight_layout()
plt.show()
```

Sample Output:

The sample output will display three images in a single figure:

1. The original X-ray image.
2. The noisy X-ray image (original image with added noise).
3. The denoised X-ray image, reconstructed using the autoencoder.

Experiment 4: Implement Speech Recognition using NLP

Aim:

To create a speech recognition system using natural language processing (NLP) techniques, demonstrating the fusion of audio data and language understanding in enabling voice-controlled applications.

Algorithm:

STEP 1: Collect a dataset of spoken audio samples and their corresponding transcripts.

STEP 2: Preprocess the audio data by converting it into spectrograms or other suitable representations.

STEP 3: Design a deep learning model, such as a recurrent neural network (RNN) or a transformer, for speech recognition.

STEP 4: Implement a suitable loss function, like connectionist temporal classification (CTC) loss.

STEP 5: Train the model on the audio-transcript pairs.

STEP 6: Evaluate the model's performance by measuring word error rate or other relevant metrics.

Program:

```
import tensorflow as tf

from tensorflow.keras.layers import Input, Embedding, LSTM, Dense

from tensorflow.keras.models import Model

import numpy as np

# Generate sample audio features (you would use actual audio features in practice)

sample_audio_features = np.random.rand(100, 20)  # 100 time steps, 20 features each

# Define the speech recognition model

input_layer = Input(shape=(100, 20))  # Input shape: (time steps, features)

embedding = Embedding(input_dim=10000, output_dim=128)(input_layer)

lstm = LSTM(128)(embedding)

output_layer = Dense(10, activation='softmax')(lstm)  # 10 possible words
```

```
speech_recognition_model = Model(inputs=input_layer, outputs=output_layer)

# Compile the model

speech_recognition_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Generate sample labels (you would use actual labels in practice)

sample_labels = np.random.randint(10, size=(100, 10))  # 10 possible words

# Train the model

speech_recognition_model.fit(sample_audio_features, sample_labels, epochs=10, batch_size=32)

# Sample audio input for prediction (you would use actual audio input in practice)

sample_input_audio = np.random.rand(1, 100, 20)  # Single input, 100 time steps, 20 features each

# Predict using the trained model

predicted_probs = speech_recognition_model.predict(sample_input_audio)

# Get the predicted word index

predicted_word_index = np.argmax(predicted_probs)

print("Predicted Word Index:", predicted_word_index)
```

Sample Output:

The output will display the predicted word index based on the sample audio input.

Experiment 5: Develop Object Detection and Classification for Traffic Analysis using CNN

Aim:

To construct a convolutional neural network (CNN) that can simultaneously detect and classify objects in traffic images, highlighting the role of CNNs in complex tasks like traffic analysis and surveillance.

Algorithm:

STEP 1: Assemble a dataset of traffic images with object annotations and labels.

STEP 2: Design a convolutional neural network (CNN) architecture for object detection and classification.

STEP 3: Implement non-maximum suppression for eliminating duplicate object detections.

STEP 4: Choose appropriate loss functions for object detection and classification, such as region proposal network (RPN) loss and categorical cross-entropy.

STEP 5: Train the CNN on the dataset and fine-tune the model.

STEP 6: Evaluate the model's performance in terms of object detection accuracy and classification accuracy.

Program:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

from tensorflow.keras.models import Sequential

# Generate sample traffic images and labels (you would use actual data in practice)

sample_images = np.random.rand(100, 64, 64, 3)  # 100 images of size 64x64 with 3 channels (RGB)

sample_labels = np.random.randint(2, size=100)  # Binary labels (0 or 1)

# Build the CNN model for object detection and classification

model = Sequential([

    Conv2D(32, (3, 3), activation='relu', input_shape=(64, 64, 3)),

    MaxPooling2D((2, 2)),

    Conv2D(64, (3, 3), activation='relu'),
```

```
    MaxPooling2D((2, 2)),

    Conv2D(128, (3, 3), activation='relu'),

    MaxPooling2D((2, 2)),

    Flatten(),

    Dense(128, activation='relu'),

    Dense(1, activation='sigmoid')  # Binary classification output

])

# Compile the model

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

# Train the model

model.fit(sample_images, sample_labels, epochs=10, batch_size=32)

# Sample traffic image for prediction (you would use actual images in practice)

sample_traffic_image = np.random.rand(1, 64, 64, 3)  # Single image of size 64x64 with 3 channels (RGB)

# Predict using the trained model

prediction = model.predict(sample_traffic_image)

print("Predicted Probability:", prediction)
```

Sample Output:

The output will display the predicted probability of the sample traffic image belonging to the positive class (1) based on the trained model. This is a simplified example, and in practice, you would use real traffic images and labels for training and testing.

Experiment 6: Implement Online Fraud Detection of Share Market Data using Data Analytics Tools

Aim:

To implement a data analytics solution for real-time fraud detection in share market transactions, showcasing the application of data analytics in financial security.

Algorithm:

STEP 1: Acquire a dataset of share market transaction data.

STEP 2: Preprocess the data by cleaning, transforming, and aggregating relevant features.

STEP 3: Choose a suitable data analytics tool, such as Pandas, R, or SQL.

STEP 4: Implement data exploration and visualization to identify potential patterns of fraudulent transactions.

STEP 5: Apply machine learning algorithms, like isolation forests or clustering, to detect anomalies.

STEP 6: Evaluate the effectiveness of the fraud detection method using metrics like precision, recall, and F1-score.

Program:

```
import pandas as pd

from sklearn.ensemble import IsolationForest

from sklearn.model_selection import train_test_split

# Load sample share market data (you would use actual data in practice)

data = pd.read_csv('sample_market_data.csv')

# Select relevant features (you would choose appropriate features in practice)

features = ['price', 'volume']

# Split data into train and test sets

train_data, test_data = train_test_split(data[features], test_size=0.2, random_state=42)

# Train an Isolation Forest model for fraud detection

model = IsolationForest(contamination=0.05)  # Assuming 5% of data is fraudulent

model.fit(train_data)

# Predict anomalies on the test set
```

```
predictions = model.predict(test_data)

# Count the number of anomalies detected

num_anomalies = len(predictions[predictions == -1])

print("Number of Anomalies Detected:", num_anomalies)
```

Sample Output:

The output will display the number of anomalies detected by the Isolation Forest model.

Experiment 7: Implement Image Augmentation using Deep RBM

Aim:

To enhance the dataset by generating augmented images using deep restricted Boltzmann machines (RBMs), emphasizing the importance of data augmentation in improving model robustness.

Algorithm:

STEP 1: Collect a dataset of images for augmentation.

STEP 2: Design a deep restricted Boltzmann machine (RBM) architecture.

STEP 3: Train the RBM using the input images to learn patterns and features.

STEP 4: Implement image augmentation techniques using the learned RBM representations, such as noise injection or distortion.

STEP 5: Apply the augmented images for training a separate deep learning model (e.g., CNN).

STEP 6: Compare the performance of the model trained with and without image augmentation.

Program:

```
import numpy as np

from sklearn.neural_network import BernoulliRBM

import matplotlib.pyplot as plt

# Generate a sample dataset of images (you would use actual images in practice)

num_samples = 100

image_size = 28 * 28

original_images = np.random.randint(0, 2, size=(num_samples, image_size))

# Define a deep RBM with two hidden layers

rbm = BernoulliRBM(n_components=128, n_iter=10, batch_size=10)

rbm2 = BernoulliRBM(n_components=64, n_iter=10, batch_size=10)

# Fit the RBM layers

rbm.fit(original_images)

hidden_features = rbm.transform(original_images)
```

```
rbm2.fit(hidden_features)

augmented_features = rbm2.transform(hidden_features)

# Reconstruct augmented images

reconstructed_hidden_features = rbm2.inverse_transform(augmented_features)

reconstructed_images = rbm.inverse_transform(reconstructed_hidden_features)

# Plot original and augmented images

plt.figure(figsize=(10, 4))

for i in range(5):

    plt.subplot(2, 5, i + 1)

    plt.imshow(original_images[i].reshape(28, 28), cmap='gray')

    plt.title("Original")

    plt.subplot(2, 5, i + 6)

    plt.imshow(reconstructed_images[i].reshape(28, 28), cmap='gray')

    plt.title("Augmented")

plt.tight_layout()

plt.show()
```

Sample Output:

The output will display a figure with rows of images:

1. The top row contains the original images.
2. The bottom row contains the augmented images generated using the deep RBM.

Experiment 8: Implement Sentiment Analysis using LSTM

Aim:

To develop a sentiment analysis model using long short-term memory (LSTM) networks, demonstrating the use of recurrent neural networks in understanding and classifying emotions in text data.

Algorithm:

STEP 1: Gather a dataset of text samples labeled with sentiment labels.

STEP 2: Preprocess the text data by tokenizing, padding, and converting to word embeddings.

STEP 3: Design a long short-term memory (LSTM) neural network architecture for sentiment analysis.

STEP 4: Choose a suitable loss function, like binary cross-entropy, for sentiment prediction.

STEP 5: Train the LSTM on the preprocessed text data.

STEP 6: Evaluate the model's performance in terms of sentiment classification accuracy.

Program:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences

# Sample text data for sentiment analysis (you would use actual text data in practice)

texts = [

    "I love this product!",

    "This is terrible.",

    "The movie was amazing.",

    "I'm not sure how I feel about it."

]

# Corresponding sentiment labels (0 for negative, 1 for positive)

labels = np.array([1, 0, 1, 0])
```

```
# Tokenize the text data

tokenizer = Tokenizer(num_words=1000, oov_token="<OOV>")

tokenizer.fit_on_texts(texts)

word_index = tokenizer.word_index

# Convert texts to sequences of word indices

sequences = tokenizer.texts_to_sequences(texts)

# Pad sequences to a uniform length

padded_sequences = pad_sequences(sequences, maxlen=10, padding='post',
truncating='post')

# Build the LSTM model

model = tf.keras.Sequential([

   tf.keras.layers.Embedding(input_dim=len(word_index) + 1, output_dim=16,
input_length=10),

   tf.keras.layers.LSTM(16),

   tf.keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

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

# Train the model

model.fit(padded_sequences, labels, epochs=10, verbose=1)

# Sample test text for prediction (you would use actual text in practice)

sample_test_text = ["This is a great product!"]

# Convert the test text to a sequence and pad it

sample_test_sequence = tokenizer.texts_to_sequences(sample_test_text)

sample_padded_sequence = pad_sequences(sample_test_sequence, maxlen=10,
padding='post', truncating='post')

# Predict sentiment using the trained model

prediction = model.predict(sample_padded_sequence)
```

```
print("Predicted Sentiment:", "Positive" if prediction > 0.5 else "Negative")
```

Sample Output:

The output will display the predicted sentiment (positive or negative) of the sample test text based on the trained LSTM model.

Experiment 9: Number Plate Recognition of Traffic Video Analysis (Mini Project)

Aim:

To create a system that can automatically recognize and extract number plate information from traffic videos, showcasing the practical application of computer vision techniques in traffic management and law enforcement.

Algorithm:

STEP 1: Collect a dataset of traffic videos containing scenes with number plates.

STEP 2: Extract frames from the videos and preprocess them.

STEP 3: Design a pipeline that combines object detection for locating number plates and optical character recognition (OCR) for reading the characters.

STEP 4: Implement a suitable OCR algorithm, such as Tesseract, to extract characters from number plates.

STEP 5: Train and fine-tune the OCR model using labeled character data.

STEP 6: Apply the pipeline to the video frames, recognize number plates, and output the results with accuracy scores.

Progrsm:

```
import cv2

import numpy as np

import pytesseract

# Load a sample traffic video (you would use an actual video in practice)

video_path = 'sample_traffic_video.mp4'

cap = cv2.VideoCapture(video_path)

# Initialize the pytesseract OCR engine

pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'

# Loop through video frames

while cap.isOpened():

    ret, frame = cap.read()

    if not ret:

        break
```

```
    # Convert the frame to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Apply image processing techniques (you would use appropriate techniques in practice)
    processed_frame = cv2.GaussianBlur(gray, (5, 5), 0)
    edges = cv2.Canny(processed_frame, 50, 150)
    # Perform text recognition using pytesseract
    number_plate_text = pytesseract.image_to_string(edges, config='--psm 6')
    # Display the frame and recognized text
    cv2.imshow('Number Plate Recognition', frame)
    print("Recognized Number Plate:", number_plate_text.strip())
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Sample Output:

As the code runs, a window will open displaying the traffic video frames. In the terminal, the recognized number plate text will be printed for each frame.