NNDL-lab

week=1

```
import numpy as np
class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size):

        self.weights_input_hidden = np.random.randn(input_size, hidden_size)

        self.bias_hidden = np.zeros((1, hidden_size))

        self.weights_hidden_output = np.random.randn(hidden_size, output_size)

        self.bias_output = np.zeros((1, output_size))

        print("Initialized weights and biases:")
        print("Weights (Input -> Hidden):\n", self.weights_input_hidden)
        print("Biases (Hidden Layer):\n", self.bias_hidden)
        print("Weights (Hidden -> Output):\n", self.weights_hidden_output)
        print("Biases (Output Layer):\n", self.bias_output)

    def forward(self, X):
         self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
         self.hidden_output = self.sigmoid(self.hidden_input)
         self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
         self.output = self.sigmoid(self.output_input)
         return self.output
```

```
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

input_size = 3
hidden_size = 5
output_size = 2

nn = NeuralNetwork(input_size, hidden_size, output_size)
X_input = np.array([[0.1, 0.5, 0.9], [0.2, 0.8, 0.6]])
output = nn.forward(X_input)

print("\nOutput of the neural network for the input data:")

\

print(output)
## week=2
import numpy as np

def purelin(z):
    return z
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
def relu(z):
    return np.maximum(0, z)
X = np.array([0.5, -1.2, 3.3])      # Example inputs: x1, x2, x3
W = np.array([0.8, -0.5, 1.0])      # Weights: w1, w2, w3
b = 0.2
Z = np.dot(W, X) + b
```

```
output_purelin = purelin(Z)
output_sigmoid = sigmoid(Z)
output_relu = relu(Z)
print("Weighted sum (Z):", Z)
print("Output with Purelin (Linear):", output_purelin)
print("Output with Sigmoid:", output_sigmoid)
print("Output with ReLU:", output_relu)

## week=3
import numpy as np

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size,
activation_function='sigmoid', loss_function='mse'):
        # Initialize the weights and biases
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))

        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

        # Set the activation function
        self.activation_function = activation_function
        self.loss_function = loss_function

    def forward(self, X):
        # Forward pass through the network
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
```

```
        self.hidden_output = self.apply_activation(self.hidden_input)

        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
        self.output = self.apply_activation(self.output_input)

        return self.output

    def apply_activation(self, x):
        if self.activation_function == 'sigmoid':
            return self.sigmoid(x)
        elif self.activation_function == 'relu':
            return self.relu(x)
        elif self.activation_function == 'tanh':
            return self.tanh(x)
        elif self.activation_function == 'leaky_relu':
            return self.leaky_relu(x)
        elif self.activation_function == 'softmax':
            return self.softmax(x)
        elif self.activation_function == 'softplus':
            return self.softplus(x)
        else:
            raise ValueError(f"Activation function {self.activation_function} not
supported.")

    # Activation functions
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def relu(self, x):
```

```
    return np.maximum(0, x)

def tanh(self, x):
    return np.tanh(x)

def leaky_relu(self, x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def softmax(self, x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))  # Numerical stability
    return e_x / np.sum(e_x, axis=-1, keepdims=True)

def softplus(self, x):
    return np.log(1 + np.exp(x))

# Loss function
def compute_loss(self, y_true, y_pred):
    if self.loss_function == 'mse':
        return self.mean_squared_error(y_true, y_pred)
    elif self.loss_function == 'cross_entropy':
        return self.cross_entropy_loss(y_true, y_pred)
    else:
        raise ValueError(f"Loss function {self.loss_function} not supported.")

# Mean Squared Error (MSE)
def mean_squared_error(self, y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Cross-Entropy Loss (for classification)
```

```
    def cross_entropy_loss(self, y_true, y_pred):
        # To avoid log(0), we add a small epsilon value
        epsilon = 1e-15
        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
        return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))

# Example usage:
input_size = 3    # Number of input neurons
hidden_size = 5   # Number of neurons in the hidden layer
output_size = 2   # Number of output neurons

# Create a neural network instance
nn = NeuralNetwork(input_size, hidden_size, output_size,
activation_function='sigmoid', loss_function='mse')

# Example input data (for a batch of 2 samples with 3 features each)
X_input = np.array([[0.1, 0.5, 0.9], [0.2, 0.8, 0.6]])

# Actual target values (e.g., for regression, use continuous values)
y_true = np.array([[0.5, 0.2], [0.7, 0.3]])

# Perform a forward pass with the input data
output = nn.forward(X_input)

# Calculate the loss using the true values and the predicted output
loss = nn.compute_loss(y_true, output)

print("\nPredicted output:")
print(output)
```

```
print("\nLoss (MSE):")
print(loss)

## week=4
import numpy as np

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.01):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))
        self.learning_rate = learning_rate
        def forward(self, X):
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_input)
        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
        self.output = self.sigmoid(self.output_input)
        return self.output

  def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
       def sigmoid_derivative(self, x):
        return x * (1 - x)

      def compute_loss(self, y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)
```

```
    def backward(self, X, y_true, y_pred):
    output_error = y_pred - y_true
    output_delta = output_error * self.sigmoid_derivative(y_pred)
    hidden_error = output_delta.dot(self.weights_hidden_output.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
    self.weights_hidden_output -= self.learning_rate *
self.hidden_output.T.dot(output_delta)
    self.bias_output -= self.learning_rate * np.sum(output_delta, axis=0,
keepdims=True)
    self.weights_input_hidden -= self.learning_rate * X.T.dot(hidden_delta)
    self.bias_hidden -= self.learning_rate * np.sum(hidden_delta, axis=0,
keepdims=True)


  def train(self, X, y_true, epochs=1000):
   for epoch in range(epochs):
     y_pred = self.forward(X)
     loss = self.compute_loss(y_true, y_pred)
     self.backward(X, y_true, y_pred)


    if epoch % 100 == 0:
          print(f"Epoch {epoch} | Loss: {loss}")
input_size = 3  # Number of input neurons
hidden_size = 5  # Number of neurons in the hidden layer
output_size = 1  # Number of output neurons(for regression, it can be 1)
nn = NeuralNetwork(input_size, hidden_size, output_size)
X_input = np.array([[0.1, 0.5, 0.9],
            [0.2, 0.6, 0.8],
            [0.9, 0.1, 0.4],
            [0.5, 0.3, 0.8]])
y_true = np.array([[0.5], [0.7], [0.2], [0.9]])
```

```
nn.train(X_input, y_true, epochs=1000)
y_pred = nn.forward(X_input)
print("\nPredicted output after training:")
print(y_pred)

## week=5
import numpy as np

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):

        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))

        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

        # Learning rate for gradient descent
        self.learning_rate = learning_rate

    def forward(self, X):
        # Forward pass: Calculate the activations for hidden and output layers
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_input)

        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
        self.output = self.sigmoid(self.output_input)
```

```
        return self.output

    def sigmoid(self, x):
        # Sigmoid activation function
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        # Derivative of the sigmoid function
        return x * (1 - x)

    def compute_loss(self, y_true, y_pred):
        # Mean Squared Error loss function
        return np.mean((y_true - y_pred) ** 2)

    def backward(self, X, y_true, y_pred):
        # Backward pass: Calculate the gradients and update weights using gradient
descent

        # Calculate output layer error and delta
        output_error = y_pred - y_true
        output_delta = output_error * self.sigmoid_derivative(y_pred)

        # Calculate hidden layer error and delta
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

        # Update weights and biases using the gradients
        self.weights_hidden_output -= self.learning_rate *
self.hidden_output.T.dot(output_delta)
```

```
        self.bias_output -= self.learning_rate * np.sum(output_delta, axis=0,
keepdims=True)


        self.weights_input_hidden -= self.learning_rate * X.T.dot(hidden_delta)
        self.bias_hidden -= self.learning_rate * np.sum(hidden_delta, axis=0,
keepdims=True)


    def train(self, X, y_true, epochs=10000):
        # Train the neural network using forward and backward propagation
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X)

            # Compute loss
            loss = self.compute_loss(y_true, y_pred)

            # Backward pass (backpropagation)
            self.backward(X, y_true, y_pred)

            # Print loss every 1000 epochs
            if epoch % 1000 == 0:
                print(f"Epoch {epoch} | Loss: {loss}")

    def test(self, X):
        # Test the neural network with the input data
        return self.forward(X)

# Example usage for the XOR problem:
# Input data for XOR (4 examples, 2 features each)
X_input = np.array([[0, 0],
```

```
                [0, 1],
                [1, 0],
                [1, 1]])

# Output data for XOR (4 examples, 1 output each)
y_true = np.array([[0], [1], [1], [0]])

# Create a neural network instance
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1, learning_rate=0.1)

# Train the neural network
print("Training the neural network...")
nn.train(X_input, y_true, epochs=10000)

# After training, test the neural network on the XOR input
print("\nTesting the neural network on the XOR input:")
y_pred = nn.test(X_input)

print("\nPredicted output after training:")
print(y_pred)

## week=6
import numpy as np

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        # Initialize weights and biases
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
```

```
        self.bias_hidden = np.zeros((1, hidden_size))

        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))
        # Learning rate for gradient descent
        self.learning_rate = learning_rate

    def forward(self, X):
        # Forward pass: Calculate the activations for hidden and output layers
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_input)

        self.output_input = np.dot(self.hidden_output, self.weights_hidden_output) +
self.bias_output
        self.output = self.sigmoid(self.output_input)

        return self.output

    def sigmoid(self, x):
        # Sigmoid activation function
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        # Derivative of the sigmoid function
        return x * (1 - x)

    def compute_loss(self, y_true, y_pred):
        # Mean Squared Error loss function
        return np.mean((y_true - y_pred) ** 2)
```

```
    def backward(self, X, y_true, y_pred):
        # Backward pass: Calculate the gradients and update weights using gradient descent

        # Calculate output layer error and delta
        output_error = y_pred - y_true
        output_delta = output_error * self.sigmoid_derivative(y_pred)

        # Calculate hidden layer error and delta
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

        # Update weights and biases using the gradients
        self.weights_hidden_output -= self.learning_rate * self.hidden_output.T.dot(output_delta)
        self.bias_output -= self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)

        self.weights_input_hidden -= self.learning_rate * X.T.dot(hidden_delta)
        self.bias_hidden -= self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

    def train(self, X, y_true, epochs=10000):
        # Train the neural network using forward and backward propagation
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X)

            # Compute loss
```

```
            loss = self.compute_loss(y_true, y_pred)

            # Backward pass (backpropagation)
            self.backward(X, y_true, y_pred)

            # Print loss every 1000 epochs
            if epoch % 1000 == 0:
                print(f"Epoch {epoch} | Loss: {loss}")

    def test(self, X):
        # Test the neural network with the input data
        return self.forward(X)

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        # Initialize weights and bias
        self.weights = np.random.randn(input_size)
        self.bias = np.random.randn()
        self.learning_rate = learning_rate

    def forward(self, X):
        # non Linear combination + step function
        non_linear_output = np.dot(X, self.weights) + self.bias
        return self.step_function(non_linear_output)

    def step_function(self, x):
        # Binary step activation function
        return np.where(x >= 0, 1, 0)
```

```
def train(self, X, y, epochs=100):
    print("Training perceptron for non-linearly separable problem...")

    for epoch in range(epochs):
        total_error = 0

        for i in range(len(X)):
            # Forward pass
            prediction = self.forward(X[i])

            # Calculate error
            error = y[i] - prediction
            total_error += abs(error)

            # Update weights and bias (Perceptron Learning Rule)
            self.weights += self.learning_rate * error * X[i]
            self.bias += self.learning_rate * error

        # Stop early if converged
        if total_error == 0:
            print(f"Converged at epoch {epoch}")
            break

        if epoch % 10 == 0:
            print(f"Epoch {epoch} | Total Error: {total_error}")

def predict(self, X):
    return self.forward(X)
```

```
    # Example usage for the XOR problem:

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


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


nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1, learning_rate=0.1)

print("Training the neural network...")
nn.train(X_input, y_true, epochs=10000)


print("\nTesting the neural network on the XOR input:")
y_pred = nn.test(X_input)

print("\nPredicted output after training:")
print(y_pred)
## week=7

import tensorflow as tf
from tensorflow.keras import layers, models, datasets
import matplotlib.pyplot as plt

def load_data():
```

```
  (x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
  x_train, x_test = x_train / 255.0, x_test / 255.0 # Normalize to [0, 1]
  y_train = tf.keras.utils.to_categorical(y_train, 10) # One-hot encoding
  y_test = tf.keras.utils.to_categorical(y_test, 10)
  return (x_train, y_train), (x_test, y_test)

def create_model():

  model = models.Sequential()
  model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.MaxPooling2D((2, 2)))
  model.add(layers.Conv2D(64, (3, 3), activation='relu'))
  model.add(layers.Flatten())
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax')) # Output layer for 10 classes
  return model

def compile_model(model):

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

def train_model(model, x_train, y_train, x_test, y_test):

  history = model.fit(x_train, y_train, epochs=10,
```

```
    validation_data=(x_test, y_test), batch_size=64)
  return history

def plot_history(history):
  plt.plot(history.history['accuracy'], label='accuracy')
  plt.plot(history.history['val_accuracy'], label='val_accuracy')
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
  plt.ylim([0, 1])
  plt.legend(loc='lower right')
  plt.show()
if __name__ == "__main__":
  (x_train, y_train), (x_test, y_test) = load_data()
  model = create_model()
  compile_model(model)
  history = train_model(model, x_train, y_train, x_test, y_test)
  test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
  print(f"Test accuracy: {test_acc:.2f}")
  plot_history(history)
## week=9
!pip install keras
!pip install tensorflow
uploaded ="/content/group.jpeg"
import cv2
import matplotlib.pyplot as plt

# Load pre-trained Haar Cascade classifier for face detection
haarcascade_path = cv2.data.haarcascades + 'haarcascade_frontalface_default.xml'
face_cascade = cv2.CascadeClassifier(haarcascade_path)
```

```
def detect_faces(uploaded):
    """Detect faces in an image using Haar Cascade."""
    image = cv2.imread(uploaded)
    if image is None:
        print(f"Error: Cannot read image at {uploaded}")
        return
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)  # Convert to grayscale
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=5,
minSize=(30, 30))
    for (x, y, w, h) in faces:
        cv2.rectangle(image, (x, y), (x + w, y + h), (255, 0, 0), 2)
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    plt.imshow(image_rgb)
    plt.axis("off")
    plt.show()


    print(f"Detected {len(faces)} face(s).")
detect_faces(uploaded)
## week=11
!pip install keras
!pip install tensorflow
!pip install scikit-learn
from google.colab import files
uploaded = files.upload()
uploaded ="/content/Dr_ A. P. J.jpeg"
!pip install face_recognition opencv-python numpy
import numpy as np
import tensorflow as tf
```

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.preprocessing import image
import matplotlib.pyplot as plt
image_path = '/content/Dr_ A. P. J.jpeg'

def preprocess_image(path, target_size=(128, 128)):
    # Load the image and resize it
    img = image.load_img(path, target_size=target_size)
    # Convert image to numpy array
    img = image.img_to_array(img)
    # Reshape the array to a batch format
    img = np.expand_dims(img, axis=0)
    # Normalize pixel values to the range [0, 1]
    img = img.astype('float32') / 255.0
    return img

input_image = preprocess_image(image_path)
print(f"Input image shape: {input_image.shape}")
# Step 3: Build the autoencoder model
# Encoder
input_img = Input(shape=(128, 128, 3))

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
```

```
x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')
autoencoder.fit(input_image, input_image, epochs=50, shuffle=True, batch_size=1)
reconstructed_image = autoencoder.predict(input_image)
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(np.squeeze(input_image))
plt.axis('off')
plt.subplot(1, 2, 2)
plt.title("Reconstructed Image")
plt.imshow(np.squeeze(reconstructed_image))
plt.axis('off')
plt.show()
```