Name – Swarnim Shekhar
Class – LY AIEC 1
Roll - 2213489

Deep Learning Experiments

Experiment No: 1

Title:

Using only NumPy, design a simple neural network to classify the Iris flowers.

Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# One-hot encode the target labels
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y.reshape(-1, 1))

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize weights and biases
input_size = X_train.shape[1]
hidden_size = 10
output_size = y_train.shape[1]
np.random.seed(42)
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros(hidden_size)
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros(output_size)

# Activation function and derivative
```

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

def sigmoid_derivative(x):
    return x * (1 - x)

# Loss function
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

# Training loop
epochs = 5000
learning_rate = 0.01

for epoch in range(epochs):
    # Forward pass
    z1 = np.dot(X_train, W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    # Compute loss
    loss = mse_loss(y_train, a2)

    # Backward pass
    error = y_train - a2
    dW2 = np.dot(a1.T, error * sigmoid_derivative(a2))
    db2 = np.sum(error * sigmoid_derivative(a2), axis=0)
    dW1 = np.dot(X_train.T, (np.dot(error * sigmoid_derivative(a2), W2.T) *
sigmoid_derivative(a1)))
    db1 = np.sum((np.dot(error * sigmoid_derivative(a2), W2.T) * sigmoid_derivative(a1)),
axis=0)

    # Update weights and biases
    W2 += learning_rate * dW2
    b2 += learning_rate * db2
    W1 += learning_rate * dW1
    b1 += learning_rate * db1

    if epoch % 500 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Test the model
```

```
z1 = np.dot(X_test, W1) + b1
a1 = sigmoid(z1)
z2 = np.dot(a1, W2) + b2
a2 = sigmoid(z2)
test_loss = mse_loss(y_test, a2)
print(f"Test Loss: {test_loss:.4f}")
```

Experiment No: 2

Title:

Develop a CNN to classify images from the CIFAR-10 dataset.

Code:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()

# Normalize data
x_train, x_test = x_train / 255.0, x_test / 255.0

# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))

# Evaluate the model
```

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f"Test accuracy: {test_acc:.2f}")
```

Experiment No: 3

Title:

Train a neural network model with various learning rates, batch sizes, and optimizers.

Code:

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models

from tensorflow.keras.optimizers import Adam, SGD


# Load MNIST dataset

(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0


# Expand dimensions for compatibility with CNN

x_train = x_train[..., tf.newaxis]

x_test = x_test[..., tf.newaxis]


# Define model

def create_model():

  model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(128, activation='relu'),

    layers.Dense(10, activation='softmax')
```

```
    ])
    return model

# Experiment with hyperparameters
learning_rates = [0.01, 0.001]
batch_sizes = [32, 64]
optimizers = [Adam, SGD]

results = []

for lr in learning_rates:
    for batch_size in batch_sizes:
        for opt in optimizers:
            model = create_model()
            optimizer = opt(learning_rate=lr)
            model.compile(optimizer=optimizer,
                          loss='sparse_categorical_crossentropy',
                          metrics=['accuracy'])
            history = model.fit(x_train, y_train, batch_size=batch_size, epochs=3,
validation_split=0.1, verbose=0)
            test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
            results.append((lr, batch_size, opt.__name__, test_acc))

# Display results
for result in results:
    print(f"Learning Rate: {result[0]}, Batch Size: {result[1]}, Optimizer: {result[2]}, Test
Accuracy: {result[3]:.4f}")
```

Experiment No: 4

Title:

Evaluate and compare CNN, RNN, and MLP architectures.

Code:

```
import tensorflow as tf

from tensorflow.keras import datasets, layers, models

from tensorflow.keras.layers import SimpleRNN, Flatten, Dense


# Load MNIST dataset

(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0


# Reshape for MLP

x_train_mlp = x_train.reshape(x_train.shape[0], -1)

x_test_mlp = x_test.reshape(x_test.shape[0], -1)


# CNN Model

def cnn_model():

  model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),

    layers.MaxPooling2D((2, 2)),

    layers.Flatten(),

    layers.Dense(128, activation='relu'),

    layers.Dense(10, activation='softmax')

  ])

  return model
```

```
# RNN Model

def rnn_model():

    model = models.Sequential([

        SimpleRNN(64, activation='relu', input_shape=(28, 28)),

        Dense(10, activation='softmax')

    ])

    return model


# MLP Model

def mlp_model():

    model = models.Sequential([

        Dense(128, activation='relu', input_shape=(784,)),

        Dense(10, activation='softmax')

    ])

    return model


# Compile and train all models

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

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

    model.fit(x_train, y_train, epochs=5, verbose=2)

    _, test_acc = model.evaluate(x_test, y_test, verbose=0)

    return test_acc


# Prepare datasets for CNN and RNN

x_train_cnn = x_train[..., tf.newaxis]
```

```
x_test_cnn = x_test[..., tf.newaxis]


cnn_acc = train_and_evaluate(cnn_model(), x_train_cnn, y_train, x_test_cnn, y_test)
rnn_acc = train_and_evaluate(rnn_model(), x_train, y_train, x_test, y_test)
mlp_acc = train_and_evaluate(mlp_model(), x_train_mlp, y_train, x_test_mlp, y_test)


print(f"CNN Accuracy: {cnn_acc:.4f}")
print(f"RNN Accuracy: {rnn_acc:.4f}")
print(f"MLP Accuracy: {mlp_acc:.4f}")
```

Experiment No: 5

Title:

Design a neural network with advanced techniques like dropout and batch normalization.

Code:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models


# Load CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0


# Build model with dropout and batch normalization
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
```

```
    layers.Dropout(0.3),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation='softmax')
])

# Compile and train the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
```

Experiment No: 6

Title:

Apply model quantization and pruning techniques.

Code:

```
import tensorflow as tf
from tensorflow_model_optimization.sparsity import keras as sparsity

# Load a pretrained model
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

```python
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

# Define a simple CNN
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])

# Train the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(x_train, y_train, epochs=3)

# Apply pruning
pruning_params = {
    'pruning_schedule': sparsity.PolynomialDecay(initial_sparsity=0.2,
                                                 final_sparsity=0.8,
                                                 begin_step=0,
                                                 end_step=100)
}
pruned_model = sparsity.prune_low_magnitude(model, **pruning_params)
```

```
# Fine-tune pruned model

pruned_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

pruned_model.fit(x_train, y_train, epochs=3)


# Save and evaluate

pruned_model.save('pruned_model.h5')

print(f"Pruned model accuracy: {pruned_model.evaluate(x_test, y_test)[1]:.4f}")
```

Experiment No: 7

Title:

Implement a transfer learning model using a pretrained ResNet.

Code:

```
import tensorflow as tf

from tensorflow.keras.applications import ResNet50

from tensorflow.keras import layers, models


# Load CIFAR-10 dataset

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

x_train, x_test = x_train / 255.0, x_test / 255.0


# Load pretrained ResNet50 model (exclude top layer)

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))


# Freeze base model layers

base_model.trainable = False
```

```
# Build model
model = models.Sequential([
    base_model,
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Compile and train
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

# Evaluate
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Transfer Learning Test Accuracy: {test_acc:.4f}")
```

Experiment No: 8

Title:

Build an autoencoder to compress and reconstruct images.

Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Load MNIST dataset
(x_train, _), (x_test, _) = tf.keras.datasets.mnist.load_data()
```

```
x_train, x_test = x_train / 255.0, x_test / 255.0

# Flatten images for input into autoencoder
x_train = x_train.reshape((x_train.shape[0], -1))
x_test = x_test.reshape((x_test.shape[0], -1))

# Define autoencoder
input_dim = x_train.shape[1]
encoding_dim = 64

input_img = layers.Input(shape=(input_dim,))
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
decoded = layers.Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = models.Model(input_img, decoded)
encoder = models.Model(input_img, encoded)

# Compile and train autoencoder
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
autoencoder.fit(x_train, x_train, epochs=20, batch_size=256, validation_data=(x_test, x_test))

# Evaluate reconstruction
reconstructed = autoencoder.predict(x_test)
```

Experiment No: 9

Title:

Train a GAN to generate MNIST-like images.

Code:

```
import tensorflow as tf

from tensorflow.keras import layers

import numpy as np


# Load MNIST dataset

(x_train, _), _ = tf.keras.datasets.mnist.load_data()

x_train = x_train / 255.0

x_train = np.expand_dims(x_train, axis=-1)


# Define generator

def build_generator():

    model = tf.keras.Sequential([

        layers.Dense(128, activation='relu', input_dim=100),

        layers.BatchNormalization(),

        layers.LeakyReLU(),

        layers.Dense(28 * 28 * 1, activation='sigmoid'),

        layers.Reshape((28, 28, 1))

    ])

    return model


# Define discriminator

def build_discriminator():
```

```
    model = tf.keras.Sequential([
        layers.Flatten(input_shape=(28, 28, 1)),
        layers.Dense(128, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])
    return model

# Compile GAN
generator = build_generator()
discriminator = build_discriminator()
discriminator.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

gan = tf.keras.Sequential([generator, discriminator])
discriminator.trainable = False
gan.compile(optimizer='adam', loss='binary_crossentropy')

# Train GAN
def train_gan(epochs=10000, batch_size=128):
    for epoch in range(epochs):
        # Generate fake images
        noise = np.random.normal(0, 1, (batch_size, 100))
        fake_images = generator.predict(noise)

        # Combine real and fake images
        real_images = x_train[np.random.randint(0, x_train.shape[0], batch_size)]
        labels_real = np.ones((batch_size, 1))
```

```
        labels_fake = np.zeros((batch_size, 1))

        # Train discriminator
        discriminator.trainable = True
        d_loss_real = discriminator.train_on_batch(real_images, labels_real)
        d_loss_fake = discriminator.train_on_batch(fake_images, labels_fake)

        # Train generator
        noise = np.random.normal(0, 1, (batch_size, 100))
        labels = np.ones((batch_size, 1))
        discriminator.trainable = False
        g_loss = gan.train_on_batch(noise, labels)

        if epoch % 1000 == 0:
            print(f"Epoch {epoch}, D Loss Real: {d_loss_real[0]:.4f}, D Loss Fake: {d_loss_fake[0]:.4f}, G Loss: {g_loss:.4f}")

train_gan()
```

Experiment No: 10

Title:

Build and train an LSTM for sequence prediction.

Code:

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

```
# Generate dummy sequence data
data = np.sin(np.linspace(0, 100, 5000))
X = []
y = []
seq_length = 50

for i in range(len(data) - seq_length):
    X.append(data[i:i+seq_length])
    y.append(data[i+seq_length])

X = np.array(X).reshape(-1, seq_length, 1)
y = np.array(y)

# Split dataset
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Define LSTM model
model = Sequential([
    LSTM(50, activation='relu', input_shape=(seq_length, 1)),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))
```

Experiment No: 11

Title:

Implement a model to classify textual data using an embedding layer and an RNN.

Code:

```
import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences


# Sample text data

texts = [

    "I love machine learning",

    "Deep learning is amazing",

    "I enjoy solving data science problems",

    "AI is the future",

    "TensorFlow makes things easier",

]

labels = [1, 1, 1, 0, 0]  # Binary labels for classification


# Tokenize and pad sequences

tokenizer = Tokenizer(num_words=100)

tokenizer.fit_on_texts(texts)

sequences = tokenizer.texts_to_sequences(texts)

padded_sequences = pad_sequences(sequences, maxlen=10)


# Build RNN model
```

```
model = models.Sequential([
    layers.Embedding(input_dim=100, output_dim=16, input_length=10),
    layers.SimpleRNN(32, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

# Compile and train the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(padded_sequences, labels, epochs=10, verbose=2)
```

Experiment No: 12

Title:

Use reinforcement learning to solve a grid-world problem.

Code:

```
import numpy as np

# Define the environment
grid_size = 4
state_space = grid_size * grid_size
action_space = 4  # Up, Down, Left, Right
q_table = np.zeros((state_space, action_space))
gamma = 0.9  # Discount factor
alpha = 0.1  # Learning rate
epsilon = 0.1  # Exploration factor

# Define rewards and transitions
reward = -1 * np.ones((grid_size, grid_size))
```

```
reward[3, 3] = 100  # Goal state

def get_state(row, col):
    return row * grid_size + col

def get_action(state):
    if np.random.uniform(0, 1) < epsilon:
        return np.random.choice(action_space)  # Explore
    return np.argmax(q_table[state])  # Exploit

def get_next_state(state, action):
    row, col = divmod(state, grid_size)
    if action == 0 and row > 0:  # Up
        row -= 1
    elif action == 1 and row < grid_size - 1:  # Down
        row += 1
    elif action == 2 and col > 0:  # Left
        col -= 1
    elif action == 3 and col < grid_size - 1:  # Right
        col += 1
    return get_state(row, col), reward[row, col]

# Train the Q-learning agent
episodes = 500
for _ in range(episodes):
    state = get_state(0, 0)  # Start at top-left corner
```

```
    while state != get_state(3, 3):  # Until goal state
        action = get_action(state)
        next_state, reward_value = get_next_state(state, action)
        q_table[state, action] += alpha * (reward_value + gamma * np.max(q_table[next_state]) - q_table[state, action])
        state = next_state

# Print learned Q-table
print("Learned Q-table:")
print(q_table)
```