Deep Learning Lab Experiment -

1- Task: Given a sequence of alphabets (with some missing values), use an RNN and a Bidirectional RNN model to predict the missing values in the sequence.

Steps:

1. Create the dataset consisting of a sequence of alphabets.
2. Preprocess the data by encoding the alphabet characters and handling missing values.
3. Build and train an RNN model for sequence prediction.
4. Build and train a Bidirectional RNN model for comparison.
5. Predict the missing values using both models.

E.g. : M A C H I N __ predict E And using Bidirectional RNN - A C H I N E.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
import string

# 1. Create and prepare the dataset
def create_dataset():
    # Using alphabet sequence as base
    alphabet = list(string.ascii_uppercase)

    # Example sequence with missing values
    sequence = "M A C H I N _".split()  # "_" represents missing value
    complete_sequence = "M A C H I N E".split()  # Ground truth

    return sequence, complete_sequence, alphabet

# 2. Preprocess the data
def preprocess_data(sequence, alphabet):
    # Create character to index mapping
    char_to_idx = {char: idx for idx, char in enumerate(alphabet +
['_'])}
    idx_to_char = {idx: char for char, idx in char_to_idx.items()}

    # Convert sequence to numerical form
    X = np.array([char_to_idx[char] for char in sequence])
```

```
    # Create input-output pairs (shifted by 1)
    X_data = []
    y_data = []
    sequence_length = 3  # Look at 3 characters to predict next

    for i in range(len(X) - sequence_length):
        X_data.append(X[i:i + sequence_length])
        y_data.append(X[i + sequence_length])

    X_data = np.array(X_data)
    y_data = np.array(y_data)

    # Reshape for RNN [samples, timesteps, features]
    X_data = X_data.reshape((X_data.shape[0], X_data.shape[1], 1))

    return X_data, y_data, char_to_idx, idx_to_char

# 3. Build and train RNN model
def build_rnn_model(vocab_size, sequence_length):
    model = keras.Sequential([
        layers.SimpleRNN(64, input_shape=(sequence_length, 1),
return_sequences=False),
        layers.Dense(vocab_size, activation='softmax')
    ])
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# 4. Build and train Bidirectional RNN model
def build_birnn_model(vocab_size, sequence_length):
    model = keras.Sequential([
        layers.Bidirectional(layers.SimpleRNN(64,
return_sequences=False),
                             input_shape=(sequence_length, 1)),
        layers.Dense(vocab_size, activation='softmax')
    ])
    model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

# 5. Predict missing values
def predict_missing(model, sequence, char_to_idx, idx_to_char,
sequence_length):
    # Prepare input for prediction
    input_seq = sequence[-sequence_length:]
    X_pred = np.array([char_to_idx[char] for char in input_seq])
    X_pred = X_pred.reshape(1, sequence_length, 1)

    # Make prediction
```

```
    prediction = model.predict(X_pred)
    predicted_idx = np.argmax(prediction)
    return idx_to_char[predicted_idx]

def main():
    # Create dataset
    sequence, complete_sequence, alphabet = create_dataset()

    # Preprocess data
    X_data, y_data, char_to_idx, idx_to_char =
preprocess_data(sequence, alphabet)
    vocab_size = len(char_to_idx)

    # Build and train RNN model
    sequence_length = 3
    rnn_model = build_rnn_model(vocab_size, sequence_length)
    rnn_model.fit(X_data, y_data, epochs=100, verbose=0)

    # Build and train Bidirectional RNN model
    birnn_model = build_birnn_model(vocab_size, sequence_length)
    birnn_model.fit(X_data, y_data, epochs=100, verbose=0)

    # Predict using both models
    rnn_prediction = predict_missing(rnn_model, sequence, char_to_idx,
idx_to_char, sequence_length)
    birnn_prediction = predict_missing(birnn_model, sequence,
char_to_idx, idx_to_char, sequence_length)

    # Print results
    print(f"Original sequence: {' '.join(sequence)}")
    print(f"Complete sequence: {' '.join(complete_sequence)}")
    print(f"RNN prediction: {rnn_prediction}")
    print(f"Bidirectional RNN prediction: {birnn_prediction}")

    # For reverse example "_ A C H I N E"
    reverse_sequence = "_ A C H I N E".split()
    X_data_rev, y_data_rev, _, _ = preprocess_data(reverse_sequence,
alphabet)

    # Retrain models for reverse sequence
    rnn_model.fit(X_data_rev, y_data_rev, epochs=100, verbose=0)
    birnn_model.fit(X_data_rev, y_data_rev, epochs=100, verbose=0)

    reverse_rnn_pred = predict_missing(rnn_model, reverse_sequence,
char_to_idx, idx_to_char, sequence_length)
    reverse_birnn_pred = predict_missing(birnn_model,
reverse_sequence, char_to_idx, idx_to_char, sequence_length)

    print(f"\nReverse sequence: {' '.join(reverse_sequence)}")
```

```
    print(f"RNN prediction: {reverse_rnn_pred}")
    print(f"Bidirectional RNN prediction: {reverse_birnn_pred}")

if __name__ == "__main__":
    main()
```

```
C:\Users\shubh\anaconda3\Lib\site-packages\keras\src\layers\rnn\
rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
C:\Users\shubh\anaconda3\Lib\site-packages\keras\src\layers\rnn\
bidirectional.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(**kwargs)

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 276ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 733ms/step
Original sequence: M A C H I N _
Complete sequence: M A C H I N E
RNN prediction: _
Bidirectional RNN prediction: _
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 77ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step

Reverse sequence: _ A C H I N E
RNN prediction: N
Bidirectional RNN prediction: N
```

2- Predict the next word in a sentence using an RNN. Consider the following sentence

Dataset: The cat sat on the mat. The dog sat on the rug. The bird flew in the sky. The cat jumped over the fence. And predict “The cat sat on __-“

```
# Step 1: Text Preprocessing
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

# Define the dataset
sentences = [
```

```python
    "The cat sat on the mat",
    "The dog sat on the rug",
    "The bird flew in the sky",
    "The cat jumped over the fence"
]

# Tokenize the text
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sentences)
total_words = len(tokenizer.word_index) + 1  # +1 for padding/indexing

# Generate input sequences
input_sequences = []
for sentence in sentences:
    token_list = tokenizer.texts_to_sequences([sentence])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

# Pad sequences
max_seq_len = max(len(x) for x in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_seq_len,
padding='pre')
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)

# Step 2: Model Building
model = Sequential()
model.add(Embedding(input_dim=total_words, output_dim=10))  # Removed
input_length
model.add(SimpleRNN(64))
model.add(Dense(total_words, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.build(input_shape=(None, max_seq_len - 1))  # Explicitly build
the model
model.summary()
```

```
Model: "sequential_3"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                           ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
━━━━━━━━━━━━━━━━━┩
│ embedding_3 (Embedding)                │ (None, 5, 10)               │
150 │
├────────────────────────────────────────┼─────────────────────────────┤
```

```
──────────────────┤
│ simple_rnn_3 (SimpleRNN)              │ (None, 64)                 │
4,800 │
├───────────────────────────────────────┼────────────────────────────┼
──────────────────┤
│ dense_3 (Dense)                       │ (None, 15)                 │
975 │
└───────────────────────────────────────┴────────────────────────────┴
──────────────────┘

 Total params: 5,925 (23.14 KB)

 Trainable params: 5,925 (23.14 KB)

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

```
# Step 3: Training the Model
model.fit(X, y, epochs=500, verbose=0)

<keras.src.callbacks.history.History at 0x22fc307dfa0>

# Step 4: Prediction
def predict_next_word(seed_text, max_sequence_len):
    token_list = tokenizer.texts_to_sequences([seed_text])[0]
    token_list = pad_sequences([token_list], maxlen=max_sequence_len -
1, padding='pre')
    predicted = model.predict(token_list, verbose=0)
    predicted_index = np.argmax(predicted)
    for word, index in tokenizer.word_index.items():
        if index == predicted_index:
            return word

seed_text = "The cat sat on"
predicted_word = predict_next_word(seed_text, max_seq_len)
print(f"{seed_text} {predicted_word}")

The cat sat on the
```

3- Develop a sequence generator for Indian Classical Music Raga using an RNN to predict the next note in a series. The notes involved are Sa, Re, Ga, Ma, Pa, Dha, Ni, and Sha.

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
```

```python
import random

# Step 1: Dataset Preparation
def create_raga_sequences():
    # Basic notes (Swaras)
    notes = ['Sa', 'Re', 'Ga', 'Ma', 'Pa', 'Dha', 'Ni', 'Sha']

    # Define some raga scales (simplified versions)
    raga_scales = {
        'Bhairav': ['Sa', 'Re', 'Ga', 'Ma', 'Pa', 'Dha', 'Ni', 'Sha'],
# Full scale for simplicity
        'Bhopali': ['Sa', 'Re', 'Ga', 'Pa', 'Dha', 'Sa'],  # Arohana
scale
        'Bageshree': ['Sa', 'Ga', 'Ma', 'Dha', 'Ni', 'Sa'],  #
Simplified Arohana
    }

    # Note to integer mapping
    note_to_int = {note: i for i, note in enumerate(notes)}
    int_to_note = {i: note for i, note in enumerate(notes)}

    # Generate sequences
    sequences = []
    for raga, scale in raga_scales.items():
        for _ in range(20):  # Generate 20 sequences per raga
            seq_length = random.randint(5, 10)
            sequence = [random.choice(scale) for _ in
range(seq_length)]
            sequences.append(sequence)

    return sequences, note_to_int, int_to_note, notes, raga_scales

# Step 2: Preprocess Data
def preprocess_data(sequences, note_to_int, seq_length=10):
    X, y = [], []
    for seq in sequences:
        if len(seq) < 2:
            continue
        # Convert notes to integers
        num_seq = [note_to_int[note] for note in seq]
        # Create input-output pairs
        for i in range(len(num_seq) - 1):
            X.append(num_seq[:i + 1])
            y.append(num_seq[i + 1])

    # Pad sequences
    X = keras.utils.pad_sequences(X, maxlen=seq_length, padding='pre')
    y = keras.utils.to_categorical(y, num_classes=len(note_to_int))

    return np.array(X), np.array(y)
```

```
# Step 3: Build RNN Model
def build_rnn_model(vocab_size=8):
    model = keras.Sequential([
        layers.Embedding(vocab_size, 64),
        layers.LSTM(128, return_sequences=False),  # Using LSTM 
instead of SimpleRNN for better memory
        layers.Dense(vocab_size, activation='softmax')
    ])
    model.compile(loss='categorical_crossentropy', optimizer='adam', 
metrics=['accuracy'])
    return model

# Step 4 & 5: Train and Generate Sequences
def generate_sequence(model, seed_sequence, note_to_int, int_to_note, 
length=10, temperature=1.0):
    generated = seed_sequence.copy()
    num_seq = [note_to_int[note] for note in seed_sequence]

    for _ in range(length):
        padded_seq = keras.utils.pad_sequences([num_seq], maxlen=10, 
padding='pre')
        prediction = model.predict(padded_seq, verbose=0)[0]

        # Apply temperature to predictions
        prediction = np.log(prediction + 1e-7) / temperature
        exp_preds = np.exp(prediction)
        prediction = exp_preds / np.sum(exp_preds)

        # Sample next note
        next_note_idx = np.random.choice(len(prediction), 
p=prediction)
        next_note = int_to_note[next_note_idx]
        generated.append(next_note)
        num_seq.append(next_note_idx)
        num_seq = num_seq[1:]  # Slide window

    return generated

# Main execution
def main():
    # Prepare data
    sequences, note_to_int, int_to_note, notes, raga_scales = 
create_raga_sequences()
    X, y = preprocess_data(sequences, note_to_int)

    # Split data
    split = int(0.8 * len(X))
    X_train, X_test = X[:split], X[split:]
    y_train, y_test = y[:split], y[split:]
```

```
    # Build and train model
    model = build_rnn_model()
    print("Training RNN Model...")
    model.fit(X_train, y_train, epochs=50, batch_size=32,
validation_data=(X_test, y_test), verbose=1)

    # Generate sequences for different ragas
    print("\nGenerated Raga Sequences:")
    for raga, scale in raga_scales.items():
        seed = scale[:3]  # Use first 3 notes as seed
        generated = generate_sequence(model, seed, note_to_int,
int_to_note, length=10)
        print(f"\nRaga {raga}:")
        print("Seed:", ' '.join(seed))
        print("Generated:", ' '.join(generated))

        # Generate another variation with different temperature
        generated_temp = generate_sequence(model, seed, note_to_int,
int_to_note, length=10, temperature=0.8)
        print("Generated (temp=0.8):", ' '.join(generated_temp))

if __name__ == "__main__":
    main()
```

```
Training RNN Model...
Epoch 1/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 5s 101ms/step - accuracy: 0.1705 - loss:
2.0603 - val_accuracy: 0.0800 - val_loss: 2.0787
Epoch 2/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.2046 - loss:
1.9896 - val_accuracy: 0.3333 - val_loss: 2.1022
Epoch 3/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.2265 - loss:
1.9529 - val_accuracy: 0.4133 - val_loss: 2.0124
Epoch 4/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.1848 - loss:
1.9708 - val_accuracy: 0.4133 - val_loss: 2.0010
Epoch 5/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.2420 - loss:
1.9272 - val_accuracy: 0.4133 - val_loss: 2.0230
Epoch 6/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.2013 - loss:
1.9379 - val_accuracy: 0.1600 - val_loss: 2.0358
Epoch 7/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - accuracy: 0.2292 - loss:
1.9524 - val_accuracy: 0.3733 - val_loss: 1.9542
Epoch 8/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step - accuracy: 0.2685 - loss:
1.9021 - val_accuracy: 0.3600 - val_loss: 2.0146
Epoch 9/50
```

```
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - accuracy: 0.2494 - loss:
1.8917 - val_accuracy: 0.3600 - val_loss: 1.9553
Epoch 10/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 42ms/step - accuracy: 0.2111 - loss:
1.8880 - val_accuracy: 0.3733 - val_loss: 1.9171
Epoch 11/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.2900 - loss:
1.8396 - val_accuracy: 0.4133 - val_loss: 1.9114
Epoch 12/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - accuracy: 0.2490 - loss:
1.8514 - val_accuracy: 0.3333 - val_loss: 1.8934
Epoch 13/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.2741 - loss:
1.8282 - val_accuracy: 0.3467 - val_loss: 1.8509
Epoch 14/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - accuracy: 0.2644 - loss:
1.8298 - val_accuracy: 0.3333 - val_loss: 1.9017
Epoch 15/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 47ms/step - accuracy: 0.2819 - loss:
1.8458 - val_accuracy: 0.4000 - val_loss: 1.8997
Epoch 16/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.2656 - loss:
1.8421 - val_accuracy: 0.3333 - val_loss: 1.9308
Epoch 17/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 40ms/step - accuracy: 0.2595 - loss:
1.8013 - val_accuracy: 0.3467 - val_loss: 1.8910
Epoch 18/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.2548 - loss:
1.8292 - val_accuracy: 0.4000 - val_loss: 1.9042
Epoch 19/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.2736 - loss:
1.8175 - val_accuracy: 0.3467 - val_loss: 1.8752
Epoch 20/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.2836 - loss:
1.7888 - val_accuracy: 0.3200 - val_loss: 1.9253
Epoch 21/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 58ms/step - accuracy: 0.2728 - loss:
1.8211 - val_accuracy: 0.4000 - val_loss: 1.8811
Epoch 22/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 36ms/step - accuracy: 0.3361 - loss:
1.7318 - val_accuracy: 0.4000 - val_loss: 1.8524
Epoch 23/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 41ms/step - accuracy: 0.3296 - loss:
1.7369 - val_accuracy: 0.2800 - val_loss: 2.0067
Epoch 24/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.2832 - loss:
1.7706 - val_accuracy: 0.2800 - val_loss: 1.9185
Epoch 25/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.3259 - loss:
```

```
1.7560 - val_accuracy: 0.4133 - val_loss: 1.9218
Epoch 26/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step - accuracy: 0.3395 - loss:
1.7206 - val_accuracy: 0.2933 - val_loss: 1.9556
Epoch 27/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.2899 - loss:
1.7420 - val_accuracy: 0.3867 - val_loss: 1.8829
Epoch 28/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - accuracy: 0.3258 - loss:
1.7103 - val_accuracy: 0.4000 - val_loss: 1.9530
Epoch 29/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3017 - loss:
1.7224 - val_accuracy: 0.2933 - val_loss: 1.9903
Epoch 30/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.3051 - loss:
1.7265 - val_accuracy: 0.4000 - val_loss: 1.9832
Epoch 31/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.3679 - loss:
1.7053 - val_accuracy: 0.2800 - val_loss: 2.0911
Epoch 32/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.3017 - loss:
1.6929 - val_accuracy: 0.3733 - val_loss: 1.9513
Epoch 33/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - accuracy: 0.3417 - loss:
1.6755 - val_accuracy: 0.3867 - val_loss: 1.9808
Epoch 34/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3664 - loss:
1.6581 - val_accuracy: 0.3200 - val_loss: 2.1333
Epoch 35/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.3290 - loss:
1.6875 - val_accuracy: 0.4000 - val_loss: 2.0111
Epoch 36/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - accuracy: 0.3632 - loss:
1.6619 - val_accuracy: 0.3200 - val_loss: 1.9889
Epoch 37/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 43ms/step - accuracy: 0.3333 - loss:
1.6737 - val_accuracy: 0.3333 - val_loss: 2.1231
Epoch 38/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.3648 - loss:
1.5985 - val_accuracy: 0.2933 - val_loss: 2.0882
Epoch 39/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3952 - loss:
1.6041 - val_accuracy: 0.2933 - val_loss: 1.9930
Epoch 40/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.3858 - loss:
1.5775 - val_accuracy: 0.3200 - val_loss: 2.2147
Epoch 41/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - accuracy: 0.3626 - loss:
1.6061 - val_accuracy: 0.4000 - val_loss: 2.1003
```

```
Epoch 42/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.4329 - loss:
1.5712 - val_accuracy: 0.4000 - val_loss: 2.0466
Epoch 43/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 50ms/step - accuracy: 0.3773 - loss:
1.6196 - val_accuracy: 0.2933 - val_loss: 2.2291
Epoch 44/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 33ms/step - accuracy: 0.3896 - loss:
1.5339 - val_accuracy: 0.2667 - val_loss: 2.2193
Epoch 45/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 49ms/step - accuracy: 0.3775 - loss:
1.5778 - val_accuracy: 0.3867 - val_loss: 2.1745
Epoch 46/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.3883 - loss:
1.5489 - val_accuracy: 0.3200 - val_loss: 2.2539
Epoch 47/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.4265 - loss:
1.5213 - val_accuracy: 0.3733 - val_loss: 2.2818
Epoch 48/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 35ms/step - accuracy: 0.4468 - loss:
1.5023 - val_accuracy: 0.3867 - val_loss: 2.2747
Epoch 49/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 38ms/step - accuracy: 0.3999 - loss:
1.4933 - val_accuracy: 0.3067 - val_loss: 2.3557
Epoch 50/50
10/10 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step - accuracy: 0.4346 - loss:
1.4778 - val_accuracy: 0.4000 - val_loss: 2.3063

Generated Raga Sequences:

Raga Bhairav:
Seed: Sa Re Ga
Generated: Sa Re Ga Re Pa Re Re Re Sha Pa Pa Pa Re
Generated (temp=0.8): Sa Re Ga Pa Sa Dha Dha Sa Sa Ma Dha Dha Ga

Raga Bhopali:
Seed: Sa Re Ga
Generated: Sa Re Ga Pa Re Pa Pa Pa Dha Ga Re Dha Sa
Generated (temp=0.8): Sa Re Ga Sa Re Sa Sa Dha Pa Ga Sa Pa Re

Raga Bageshree:
Seed: Sa Ga Ma
Generated: Sa Ga Ma Ma Sa Ga Dha Sa Dha Dha Dha Ma Ni
Generated (temp=0.8): Sa Ga Ma Dha Ga Sa Sa Sa Sa Ga Sa Ga Sa
```