DLAssignment2A

April 20, 2025

0.1 ROLL NO 160122737199 V JITESH KUMAR

1 Step 1: Install and Import Libraries

[3]:
```
# Install TensorFlow (already available in Colab but ensuring latest version)
!pip install -q tensorflow

# Import Necessary Libraries
import tensorflow as tf
import numpy as np
import os
import re
import io
import string
import zipfile
import requests
from sklearn.model_selection import train_test_split

# Check TensorFlow version
print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.18.0
```

2 Step 2: Download and Load Dakshina Dataset

[4]:
```
# Step 2: Corrected Paths for Your Drive

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Set correct file paths
train_path = '/content/drive/MyDrive/dl assignment dataset/hi/lexicons/hi.
↪translit.sampled.train.tsv'
valid_path = '/content/drive/MyDrive/dl assignment dataset/hi/lexicons/hi.
↪translit.sampled.dev.tsv'
```

```
test_path  = '/content/drive/MyDrive/dl assignment dataset/hi/lexicons/hi.
 ↪translit.sampled.test.tsv'

# Define function to load data
def load_data(filepath):
    inputs = []
    targets = []
    with open(filepath, 'r', encoding='utf-8') as f:
        for line in f:
            parts = line.strip().split('\t')
            if len(parts) >= 2:
                inputs.append(parts[0])                         # Latin script
                targets.append('\t' + parts[1] + '\n')       # Devanagari script
    return inputs, targets

# Load Train, Validation, Test sets
input_texts, target_texts = load_data(train_path)
input_texts_val, target_texts_val = load_data(valid_path)
input_texts_test, target_texts_test = load_data(test_path)

# Check a sample
print("Sample Input (Latin):", input_texts[0])
print("Sample Target (Devanagari):", target_texts[0])
```

```
Mounted at /content/drive
Sample Input (Latin):
Sample Target (Devanagari):     an
```

3 Step 3: Preprocessing - Tokenization and Vectorization

[5]:
```
# Step 3: Preprocessing - Prepare Vocabulary, Tokenization

# Build input (Latin) and target (Devanagari) character vocabularies
input_characters = sorted(list(set(''.join(input_texts))))
target_characters = sorted(list(set(''.join(target_texts))))

# Number of unique tokens
num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)

# Mapping from characters to integers
input_token_index = dict([(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict([(char, i) for i, char in␣
 ↪enumerate(target_characters)])
```

```python
# Reverse mapping from integers to characters
reverse_input_char_index = dict((i, char) for char, i in input_token_index.
↪items())
reverse_target_char_index = dict((i, char) for char, i in target_token_index.
↪items())

# Max sequence lengths
max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

print("Number of unique input (Latin) tokens:", num_encoder_tokens)
print("Number of unique output (Devanagari) tokens:", num_decoder_tokens)
print("Max sequence length for inputs:", max_encoder_seq_length)
print("Max sequence length for outputs:", max_decoder_seq_length)

# Vectorize the data (prepare numpy arrays)
import numpy as np

# Create 3D zero matrices
encoder_input_data = np.zeros((len(input_texts), max_encoder_seq_length),␣
↪dtype="int32")
decoder_input_data = np.zeros((len(target_texts), max_decoder_seq_length),␣
↪dtype="int32")
decoder_target_data = np.zeros((len(target_texts), max_decoder_seq_length,␣
↪num_decoder_tokens), dtype="float32")

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t] = input_token_index[char]
    for t, char in enumerate(target_text):
        decoder_input_data[i, t] = target_token_index[char]
        # decoder_target_data is ahead by one timestep
        if t > 0:
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.0

print("Preprocessing completed successfully!")
```

```
Number of unique input (Latin) tokens: 63
Number of unique output (Devanagari) tokens: 28
Max sequence length for inputs: 19
Max sequence length for outputs: 22
Preprocessing completed successfully!
```

4 Step 4: Build Flexible Encoder-Decoder Model (LSTM by default)

[6]:
```
# Step 4: Build Encoder-Decoder Seq2Seq Model

# Hyperparameters
embedding_dim = 256      # Embedding dimension
hidden_units = 512       # Hidden state size
cell_type = 'LSTM'       # Options: 'LSTM', 'GRU', 'SimpleRNN'

# Encoder
encoder_inputs = tf.keras.Input(shape=(None,), name='encoder_inputs')
enc_emb = tf.keras.layers.Embedding(input_dim=num_encoder_tokens,␣
↪output_dim=embedding_dim, name='encoder_embedding')(encoder_inputs)

# Choose Encoder RNN Cell
if cell_type == 'LSTM':
    encoder_rnn = tf.keras.layers.LSTM(hidden_units, return_state=True,␣
↪name='encoder_lstm')
elif cell_type == 'GRU':
    encoder_rnn = tf.keras.layers.GRU(hidden_units, return_state=True,␣
↪name='encoder_gru')
elif cell_type == 'SimpleRNN':
    encoder_rnn = tf.keras.layers.SimpleRNN(hidden_units, return_state=True,␣
↪name='encoder_rnn')

# Encoder Outputs
if cell_type == 'LSTM':
    _, state_h, state_c = encoder_rnn(enc_emb)
    encoder_states = [state_h, state_c]
else:
    _, state_h = encoder_rnn(enc_emb)
    encoder_states = [state_h]

# Decoder
decoder_inputs = tf.keras.Input(shape=(None,), name='decoder_inputs')
dec_emb_layer = tf.keras.layers.Embedding(input_dim=num_decoder_tokens,␣
↪output_dim=embedding_dim, name='decoder_embedding')
dec_emb = dec_emb_layer(decoder_inputs)

# Choose Decoder RNN Cell
if cell_type == 'LSTM':
    decoder_rnn = tf.keras.layers.LSTM(hidden_units, return_sequences=True,␣
↪return_state=True, name='decoder_lstm')
elif cell_type == 'GRU':
```

```python
    decoder_rnn = tf.keras.layers.GRU(hidden_units, return_sequences=True, return_state=True, name='decoder_gru')
elif cell_type == 'SimpleRNN':
    decoder_rnn = tf.keras.layers.SimpleRNN(hidden_units, return_sequences=True, return_state=True, name='decoder_rnn')

# Decoder Outputs
if cell_type == 'LSTM':
    decoder_outputs, _, _ = decoder_rnn(dec_emb, initial_state=encoder_states)
else:
    decoder_outputs, _ = decoder_rnn(dec_emb, initial_state=encoder_states)

# Dense layer to generate probabilities
decoder_dense = tf.keras.layers.Dense(num_decoder_tokens, activation='softmax', name='decoder_output')
decoder_outputs = decoder_dense(decoder_outputs)

# Define the full model
model = tf.keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)

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

# Model Summary
model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
encoder_inputs (InputLayer)	(None, None)	0	-
decoder_inputs (InputLayer)	(None, None)	0	-
encoder_embedding (Embedding)	(None, None, 256)	16,128	encoder_inputs[0…
decoder_embedding (Embedding)	(None, None, 256)	7,168	decoder_inputs[0…
encoder_lstm (LSTM)	[(None, 512), (None, 512), (None, 512)]	1,574,912	encoder_embeddin…

```
 decoder_lstm (LSTM)    [(None, None,         1,574,912   decoder_embeddin…
                        512), (None,                      encoder_lstm[0][…
                        512), (None,                      encoder_lstm[0][…
                        512)]

 decoder_output         (None, None, 28)         14,364   decoder_lstm[0][…
 (Dense)
```

```
 Total params: 3,187,484 (12.16 MB)

 Trainable params: 3,187,484 (12.16 MB)

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

5 Step 5: Train the Seq2Seq Model

[7]:

```python
# Step 5: Train the Seq2Seq Model

# Set training hyperparameters
batch_size = 64
epochs = 30  # You can increase if time permits in Colab Pro

# Train the model
history = model.fit(
    [encoder_input_data, decoder_input_data],
    decoder_target_data,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.2
)
```

```
Epoch 1/30
553/553              14s 17ms/step -
accuracy: 0.0917 - loss: 0.9948 - val_accuracy: 0.1172 - val_loss: 1.0758
Epoch 2/30
553/553              9s 17ms/step -
accuracy: 0.1649 - loss: 0.7377 - val_accuracy: 0.1548 - val_loss: 0.9091
Epoch 3/30
553/553              10s 17ms/step -
accuracy: 0.2262 - loss: 0.5212 - val_accuracy: 0.2147 - val_loss: 0.5897
Epoch 4/30
553/553              10s 16ms/step -
```

```
accuracy: 0.2714 - loss: 0.3557 - val_accuracy: 0.2483 - val_loss: 0.4607
Epoch 5/30
553/553              9s 17ms/step -
accuracy: 0.2962 - loss: 0.2670 - val_accuracy: 0.2606 - val_loss: 0.4203
Epoch 6/30
553/553              9s 17ms/step -
accuracy: 0.3081 - loss: 0.2234 - val_accuracy: 0.2740 - val_loss: 0.3712
Epoch 7/30
553/553              9s 17ms/step -
accuracy: 0.3154 - loss: 0.1962 - val_accuracy: 0.2748 - val_loss: 0.3669
Epoch 8/30
553/553              9s 17ms/step -
accuracy: 0.3200 - loss: 0.1754 - val_accuracy: 0.2785 - val_loss: 0.3556
Epoch 9/30
553/553              10s 17ms/step -
accuracy: 0.3243 - loss: 0.1626 - val_accuracy: 0.2807 - val_loss: 0.3449
Epoch 10/30
553/553              10s 17ms/step -
accuracy: 0.3268 - loss: 0.1512 - val_accuracy: 0.2804 - val_loss: 0.3490
Epoch 11/30
553/553              10s 18ms/step -
accuracy: 0.3289 - loss: 0.1427 - val_accuracy: 0.2843 - val_loss: 0.3286
Epoch 12/30
553/553              9s 16ms/step -
accuracy: 0.3304 - loss: 0.1336 - val_accuracy: 0.2836 - val_loss: 0.3364
Epoch 13/30
553/553              11s 17ms/step -
accuracy: 0.3317 - loss: 0.1293 - val_accuracy: 0.2851 - val_loss: 0.3255
Epoch 14/30
553/553              10s 17ms/step -
accuracy: 0.3327 - loss: 0.1233 - val_accuracy: 0.2850 - val_loss: 0.3255
Epoch 15/30
553/553              10s 17ms/step -
accuracy: 0.3335 - loss: 0.1200 - val_accuracy: 0.2839 - val_loss: 0.3317
Epoch 16/30
553/553              9s 17ms/step -
accuracy: 0.3352 - loss: 0.1145 - val_accuracy: 0.2822 - val_loss: 0.3303
Epoch 17/30
553/553              10s 16ms/step -
accuracy: 0.3359 - loss: 0.1119 - val_accuracy: 0.2821 - val_loss: 0.3332
Epoch 18/30
553/553              10s 17ms/step -
accuracy: 0.3357 - loss: 0.1079 - val_accuracy: 0.2833 - val_loss: 0.3293
Epoch 19/30
553/553              10s 18ms/step -
accuracy: 0.3372 - loss: 0.1043 - val_accuracy: 0.2840 - val_loss: 0.3288
Epoch 20/30
553/553              10s 17ms/step -
```

```
accuracy: 0.3366 - loss: 0.1025 - val_accuracy: 0.2831 - val_loss: 0.3333
Epoch 21/30
553/553              10s 17ms/step -
accuracy: 0.3381 - loss: 0.1008 - val_accuracy: 0.2807 - val_loss: 0.3358
Epoch 22/30
553/553              10s 18ms/step -
accuracy: 0.3376 - loss: 0.0975 - val_accuracy: 0.2850 - val_loss: 0.3232
Epoch 23/30
553/553              10s 17ms/step -
accuracy: 0.3394 - loss: 0.0954 - val_accuracy: 0.2838 - val_loss: 0.3305
Epoch 24/30
553/553              11s 18ms/step -
accuracy: 0.3387 - loss: 0.0951 - val_accuracy: 0.2796 - val_loss: 0.3444
Epoch 25/30
553/553              10s 17ms/step -
accuracy: 0.3393 - loss: 0.0930 - val_accuracy: 0.2828 - val_loss: 0.3336
Epoch 26/30
553/553              10s 17ms/step -
accuracy: 0.3397 - loss: 0.0917 - val_accuracy: 0.2799 - val_loss: 0.3416
Epoch 27/30
553/553              10s 18ms/step -
accuracy: 0.3402 - loss: 0.0893 - val_accuracy: 0.2838 - val_loss: 0.3259
Epoch 28/30
553/553              10s 17ms/step -
accuracy: 0.3409 - loss: 0.0874 - val_accuracy: 0.2788 - val_loss: 0.3489
Epoch 29/30
553/553              10s 18ms/step -
accuracy: 0.3404 - loss: 0.0870 - val_accuracy: 0.2795 - val_loss: 0.3416
Epoch 30/30
553/553              10s 18ms/step -
accuracy: 0.3408 - loss: 0.0847 - val_accuracy: 0.2801 - val_loss: 0.3431
```

6 Step 6: Evaluate Model and Generate Sample Predictions

[8]:
```
# Step 6: Evaluate Model and Generate Predictions

# Function to decode sequences (from predictions back to Devanagari text)
def decode_sequence(input_seq):
    # Encode the input as state vectors
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1 with only the start character␣
 ↪'\t'
    target_seq = np.zeros((1, 1))
    target_seq[0, 0] = target_token_index['\t']

    # Sampling loop for a batch of sequences
```

```
    stop_condition = False
    decoded_sentence = ''

    while not stop_condition:
        if cell_type == 'LSTM':
            output_tokens, h, c = decoder_model.predict([target_seq] +␣
↪states_value)
            states_value = [h, c]
        else:
            output_tokens, h = decoder_model.predict([target_seq] +␣
↪states_value)
            states_value = [h]

        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        if (sampled_char == '\n' or len(decoded_sentence) >␣
↪max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1)
        target_seq = np.zeros((1, 1))
        target_seq[0, 0] = sampled_token_index

    return decoded_sentence

# Build encoder model for inference
encoder_model = tf.keras.Model(encoder_inputs, encoder_states)

# Build decoder model for inference
decoder_state_inputs = []
decoder_states = []

decoder_inputs_single = tf.keras.Input(shape=(1,),␣
↪name='decoder_input_inference')
decoder_emb2 = dec_emb_layer(decoder_inputs_single)

if cell_type == 'LSTM':
    decoder_state_input_h = tf.keras.Input(shape=(hidden_units,))
    decoder_state_input_c = tf.keras.Input(shape=(hidden_units,))
    decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

    decoder_outputs2, state_h2, state_c2 = decoder_rnn(
        decoder_emb2, initial_state=decoder_states_inputs)
    decoder_states = [state_h2, state_c2]
else:
```

```python
    decoder_state_input_h = tf.keras.Input(shape=(hidden_units,))
    decoder_states_inputs = [decoder_state_input_h]

    decoder_outputs2, state_h2 = decoder_rnn(
        decoder_emb2, initial_state=decoder_states_inputs)
    decoder_states = [state_h2]

decoder_outputs2 = decoder_dense(decoder_outputs2)
decoder_model = tf.keras.Model(
    [decoder_inputs_single] + decoder_states_inputs,
    [decoder_outputs2] + decoder_states
)

# Test and Show 10 Sample Predictions
for seq_index in range(10):
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)

    print('Input word:', input_texts[seq_index])
    print('Actual Devanagari:', target_texts[seq_index][1:-1])  # Remove \t and
 ↪\n
    print('Predicted Devanagari:', decoded_sentence.strip())
    print('----------------------------------------------')
```

```
1/1              0s 134ms/step
1/1              0s 143ms/step
1/1              0s 32ms/step
1/1              0s 32ms/step
Input word:
Actual Devanagari: an
Predicted Devanagari: an
---------------------------------------------
1/1              0s 28ms/step
1/1              0s 32ms/step
1/1              0s 31ms/step
1/1              0s 30ms/step
1/1              0s 33ms/step
1/1              0s 33ms/step
1/1              0s 30ms/step
1/1              0s 31ms/step
1/1              0s 30ms/step
1/1              0s 34ms/step
Input word:
Actual Devanagari: ankganit
Predicted Devanagari: ankganit
---------------------------------------------
1/1              0s 41ms/step
```

```
1/1            0s 32ms/step
1/1            0s 32ms/step
1/1            0s 32ms/step
1/1            0s 30ms/step
1/1            0s 30ms/step
1/1            0s 32ms/step
Input word:
Actual Devanagari: uncle
Predicted Devanagari: uncle
---------------------------------------------
1/1            0s 29ms/step
1/1            0s 33ms/step
1/1            0s 30ms/step
1/1            0s 32ms/step
1/1            0s 34ms/step
1/1            0s 31ms/step
1/1            0s 35ms/step
Input word:
Actual Devanagari: ankur
Predicted Devanagari: ankur
---------------------------------------------
1/1            0s 26ms/step
1/1            0s 30ms/step
1/1            0s 30ms/step
1/1            0s 29ms/step
1/1            0s 34ms/step
1/1            0s 31ms/step
1/1            0s 31ms/step
1/1            0s 31ms/step
1/1            0s 30ms/step
Input word:
Actual Devanagari: ankuran
Predicted Devanagari: ankuran
---------------------------------------------
1/1            0s 29ms/step
1/1            0s 33ms/step
1/1            0s 34ms/step
1/1            0s 42ms/step
1/1            0s 33ms/step
1/1            0s 34ms/step
1/1            0s 32ms/step
1/1            0s 32ms/step
1/1            0s 32ms/step
Input word:
Actual Devanagari: ankurit
Predicted Devanagari: ankurit
---------------------------------------------
1/1            0s 30ms/step
```

```
1/1            0s 32ms/step
1/1            0s 31ms/step
1/1            0s 31ms/step
1/1            0s 48ms/step
1/1            0s 47ms/step
1/1            0s 67ms/step
1/1            0s 51ms/step
Input word:
Actual Devanagari: aankush
Predicted Devanagari: ankush
---------------------------------------------
1/1            0s 42ms/step
1/1            0s 52ms/step
1/1            0s 142ms/step
1/1            0s 135ms/step
1/1            0s 49ms/step
1/1            0s 158ms/step
1/1            0s 128ms/step
1/1            0s 89ms/step
Input word:
Actual Devanagari: ankush
Predicted Devanagari: ankush
---------------------------------------------
1/1            0s 67ms/step
1/1            0s 89ms/step
1/1            0s 83ms/step
1/1            0s 52ms/step
1/1            0s 49ms/step
Input word:
Actual Devanagari: ang
Predicted Devanagari: ang
---------------------------------------------
1/1            0s 97ms/step
1/1            0s 52ms/step
1/1            0s 121ms/step
1/1            0s 70ms/step
1/1            0s 90ms/step
Input word:
Actual Devanagari: anga
Predicted Devanagari: ang
---------------------------------------------
```

7 Step 7: Theoretical Answers

(a) Total Computations Assumptions given:

Embedding size = m

Hidden units in LSTM = k

Input and output sequence length = T

Vocabulary size (input and output) = V

Computation inside Encoder LSTM per timestep:

Matrix multiplications: (input embedding + hidden state) → hidden units

At each timestep:

$4 \times (m + k) \times k$ computations (because LSTM has 4 gates: input, forget, cell, output)

Total encoder computations:

× 4 × (+) × $T\times4\times(m+k)\times k$

Computation inside Decoder LSTM per timestep:

Similar:

4 × (embedding size + hidden units) × hidden units

Plus Dense softmax layer output:

$k \times V$ computations

Total decoder computations:

× (4 × (+) × + ×) $T\times(4\times(m+k)\times k+k\times V)$

Final Total Computations = × [4 (+) + 4 (+) +] $T\times[4(m+k)k+4(m+k)k+kV]$
Simplified:

× (8 (+) +) $T\times(8(m+k)k+kV)$ (b) Total Number of Parameters Input Embedding Layer (encoder):

× $V\times m$

Input Embedding Layer (decoder):

× $V\times m$

Encoder LSTM:

4 × [(+) × +] $4\times[(m+k)\times k+k]$

Decoder LSTM:

4 × [(+) × +] $4\times[(m+k)\times k+k]$

Dense Layer:

× + $k\times V+V$

Final Total Parameters = 2 + 8 (+) + 2 × 4 + + $2Vm+8(m+k)k+2\times4k+kV+V$
where

m = embedding size

k = hidden units

V = vocabulary size