Experiment No 4

Name: Bhuvi Ghosh

SAPID:60009210191

Aim: Implement Language translator using Encoder Decoder model.

Theory:

Introduction

The encoder-decoder model is a way of using recurrent neural networks for sequence-to-sequence prediction problems.

It was initially developed for machine translation problems, although it has proven successful at related sequence-to-sequence prediction problems such as text summarization and question answering.

The approach involves two recurrent neural networks, one to encode the input sequence, called the encoder, and a second to decode the encoded input sequence into the target sequence called the decoder.

Following are some of the application of sequence to sequence models-

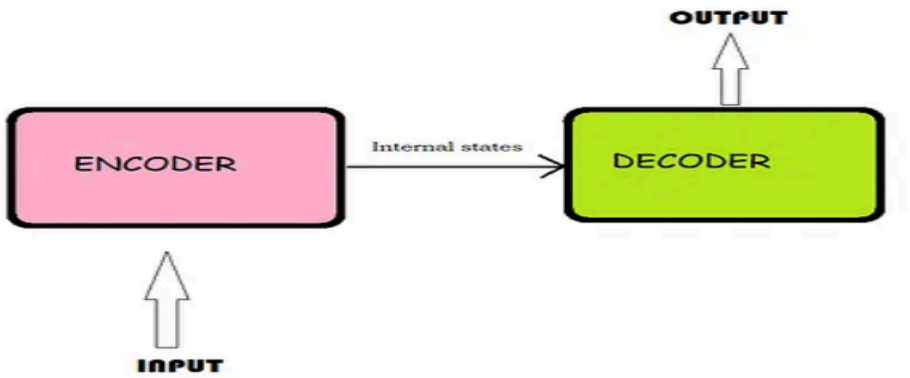
Chatbots

Machine Translation

Text summary

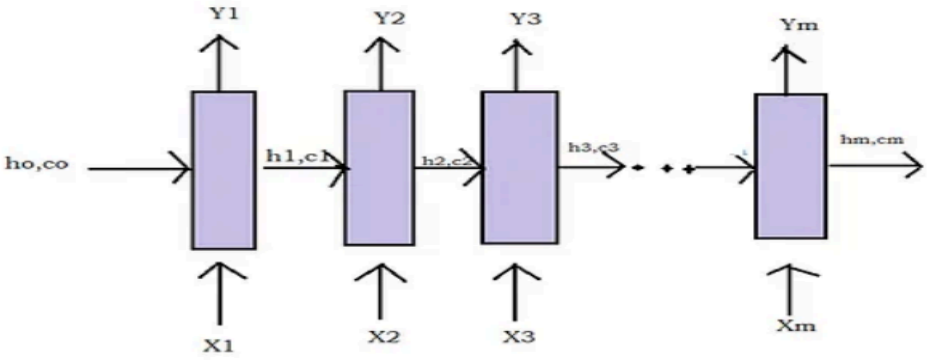
Image captioning

The architecture of Encoder-Decoder : The overall structure of sequence to sequence model(encoder-decoder) which is commonly used is as shown below



It consists of 3 parts: **encoder, intermediate vector and decoder.**

- **Encoder-**It accepts a single element of the input sequence at each time step, process it, collects information for that element and propagates it forward.
- **Intermediate vector**- This is the final internal state produced from the encoder part of the model. It contains information about the entire input sequence to help the decoder make accurate predictions.
- **Decoder-** given the entire sentence, it predicts an output at each time step.
- **Understanding the Encoder part of the model**
- The encoder is basically an LSTM/GRU cell.
- An encoder takes the input sequence and encapsulates the information as the internal state vectors.
- Outputs of the encoder are rejected and only internal states are used.
- Let's understand how encoder part of the model works



LSTM for encoder

LSTM takes only one element at a time, so if the input sequence is of length m, then LSTM takes m time steps to read the entire sequence.

- Xt is the input at time step t.
- ht and ct are internal states at time step t of the LSTM and for GRU there is only one internal state ht.
- Yt is the output at time step t.

Let's consider an example of English to Hindi translation

Inputs of Encoder Xt-

Consider the English sentence- India is beautiful country. This sequence can be thought of as a sentence containing 4 words (India, is, beautiful, country). So here

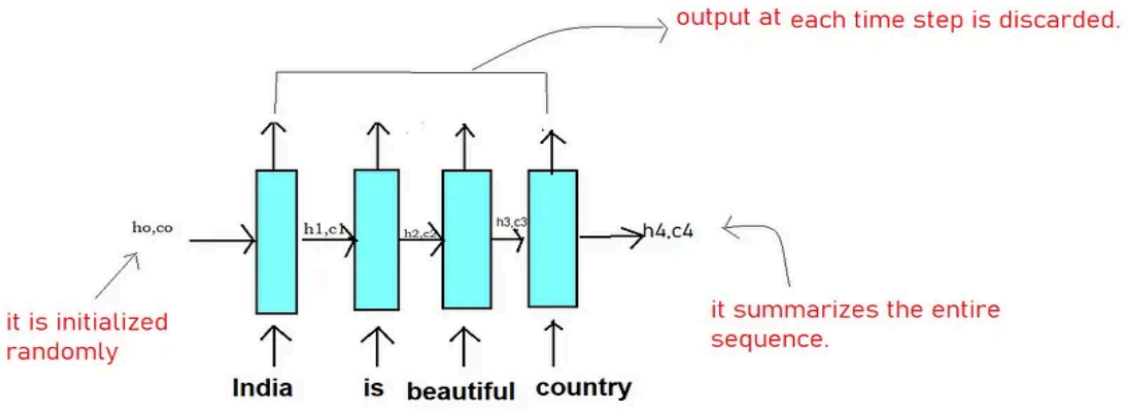
X1 =’India’

X2=’is’

X3= ‘beautiful’

X4=’country’.

Therefore LSTM will read this sequence word by word in 4-time step as follows



Here each Xt (each word) is represented as a vector using the word embedding, which converts each word into a vector of fixed length.

Now coming to internal states (ht,ct) -

- It learns what the LSTM has read until time step t. For e.g when t=2, it remembers that LSTM has read ‘*India is ‘*.
- The initial states ho, co(both are vectors) is initialized randomly or with zeroes.
- Remember the dimension of ho, co is same as the number of units in LSTM cell.
- The final state h4,c4 contains the crux of the entire input sequence I*ndia is beautiful country.*

The output of encoder Yt-

Yt at each time steps is the predictions of the LSTM at each time step. In machine translation problems, we generate the outputs when we have read the entire input sequence. So Yt at each time step in the encoder is of no use so we discard it.

The encoder will read the English sentence word by word and store the final internal states (known as an intermediate vector) of the LSTM generated after the last time step and since the output will be generated once the entire sequence is read, therefore outputs (Yt) of the Encoder at each time step are discarded.

Understanding the Decoder part of the model in the Training Phase-

The working of the decoder is different during the training and testing phase unlike the encoder part of the model which works in the same fashion in training and test phase.

Let's understand the working of the decoder part during training phase-

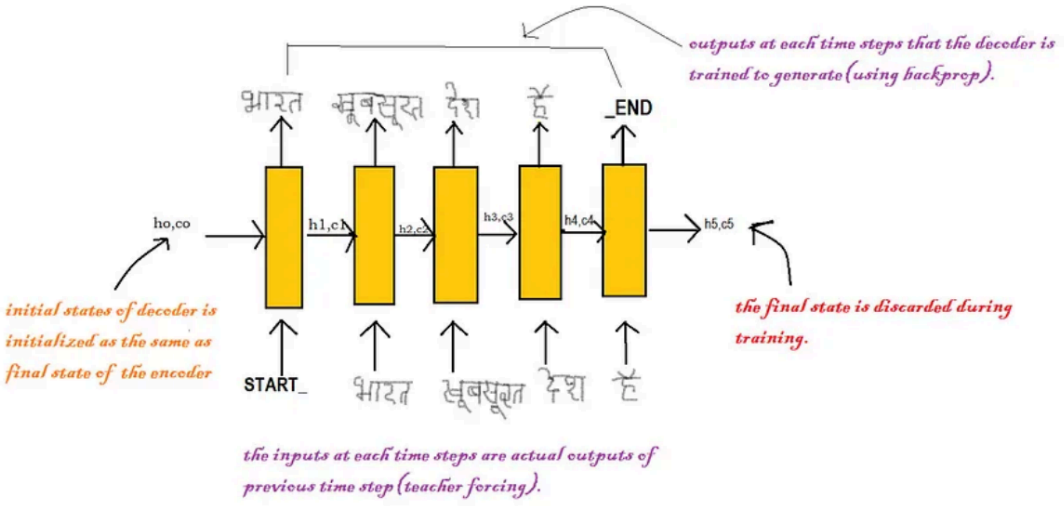
Taking the running example of translating ***India is beautiful country*** to its Hindi counterpart, just like encoder, the decoder also generates the output sentence word by word.

So we want to generate the output — ***भारत खूबसूरत देश है***

For the decoder to recognize the starting and end of the sequence, we will add START_ at the beginning of the output sequence and _END at the end of the output sequence.

So our Output sentence will be ***START_ भारत खूबसूरत देश है _END***

Let's understand the working visually

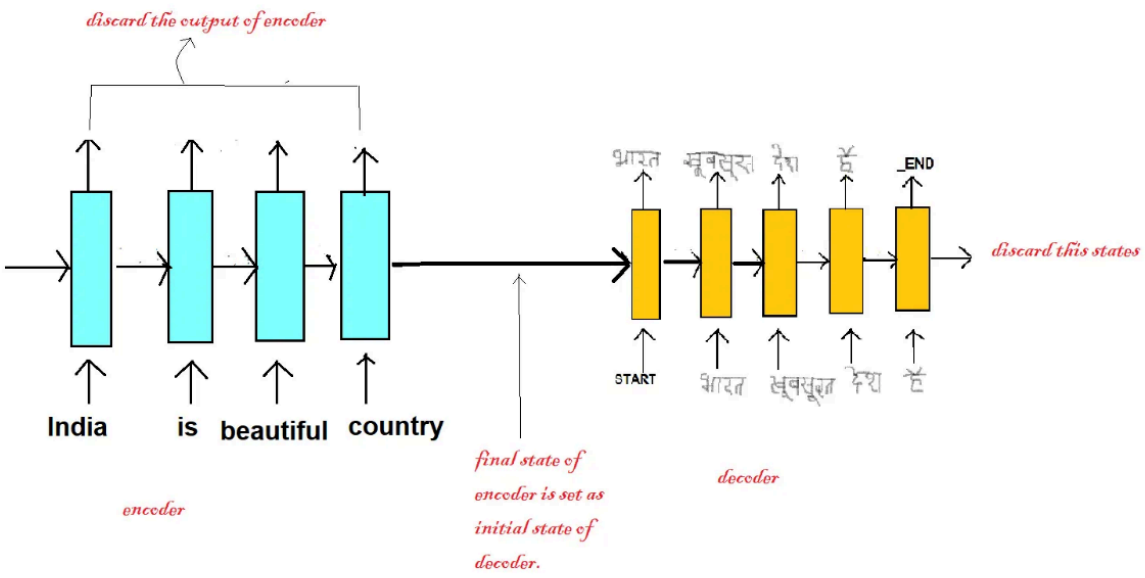


Decoder LSTM at training

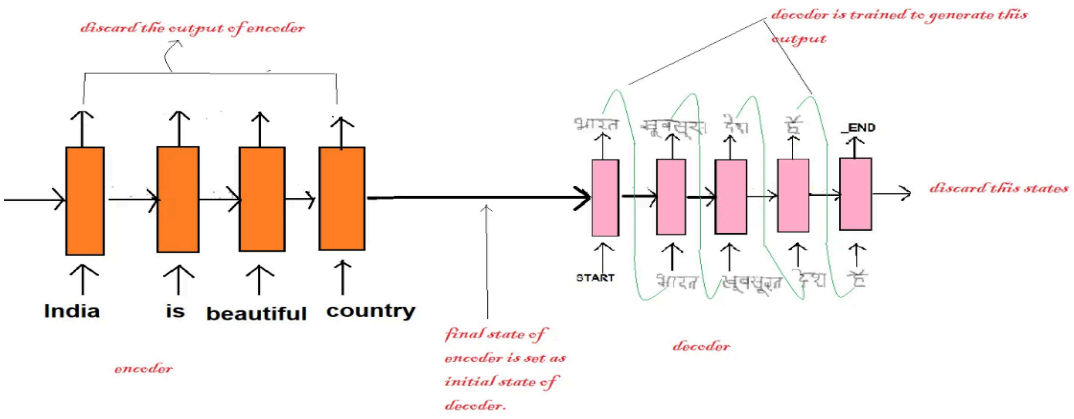
- The initial states (ho, co) of the decoder is set to the final states of the encoder. It can be thought of as that the decoder is trained to generate the output based on the information gathered by the encoder.

- Firstly, we input the START_ so that the decoder starts generating the next word. And after the last word in the Hindi sentence, we make the decoder learn to predict the _END.
- Here we use the teacher forcing technique where the input at each time step is actual output and not the predicted output from the last time step.
- At last, the loss is calculated on the predicted outputs from each time step and the errors are backpropagated through time to update the parameters of the model.
- The final states of the decoder are discarded as we got the output hence it is of no use.

Summarizing the encoder-decoder visually



Understanding the Decoder part of the model in Test Phase



\

Process of the Decoder in the test period-

- The initial states of the decoder are set to the final states of the encoder.
- LSTM in the decoder process single word at every time step.
- Input to the decoder always starts with the START_.
- The internal states generated after every time step is fed as the initial states of the next time step. for e.g At t=1, the internal states produced after inputting START_ is fed as the initial states at t=2.
- The output produced at each time step is fed as input in the next time step.
- We get to know about the end of the sequence when the decoder predicts the END_.

Lab Experiment to be performed in the session :

Exercise-1: Create a base encoder-decoder machine translation model with LSTM

Step 1:

Use a dataset of pairs of English sentences and their Marathi translation, which you can download from manythings.org/anki. There are several other translations available, you can choose whichever you like. The file to download is called mar-eng.zip.

Step 2:

Implement a *character-level* sequence-to-sequence model, processing the input character-by-character and generating the output character-by-character.

OR

Implement a word-level model, which tends to be more common for machine translation.

Setup

+ Code + Text

```
[ ] import numpy as np
    import keras
    import os
    from pathlib import Path
```

Configuration

```
[ ] batch_size = 64  # Batch size for training.
    epochs = 100  # Number of epochs to train for.
    latent_dim = 1024  # Latent dimensionality of the encoding space.
    num_samples = 10000  # Number of samples to train on.
    # Path to the data txt file on disk.
    data_path = '/content/hin.txt'
```

Prepare the data

```
[ ] # Vectorize the data.
    input_texts = []
    target_texts = []
    input_characters = set()
    target_characters = set()
    with open(data_path, "r", encoding="utf-8") as f:
        lines = f.read().split("\n")
    for line in lines[: min(num_samples, len(lines) - 1)]:
        input_text, target_text, _ = line.split("\t")
        # We use "tab" as the "start sequence" character
        # for the targets, and "\n" as "end sequence" character.
        target_text = "\t" + target_text + "\n"
        input_texts.append(input_text)
        target_texts.append(target_text)
        for char in input_text:
            if char not in input_characters:
                input_characters.add(char)
        for char in target_text:
            if char not in target_characters:
                target_characters.add(char)

    input_characters = sorted(list(input_characters))
    target_characters = sorted(list(target_characters))
    num_encoder_tokens = len(input_characters)
    num_decoder_tokens = len(target_characters)
    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 samples:", len(input_texts))
    print("Number of unique input tokens:", num_encoder_tokens)
    print("Number of unique output tokens:", num_decoder_tokens)
    print("Max sequence length for inputs:", max_encoder_seq_length)
    print("Max sequence length for outputs:", max_decoder_seq_length)

    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)])

    encoder_input_data = np.zeros(
        (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
        dtype="float32",
    )
    decoder_input_data = np.zeros(
        (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
        dtype="float32",
    )
    decoder_target_data = np.zeros(
        (len(input_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]] = 1.0
        encoder_input_data[i, t + 1 :, input_token_index[" "]] = 1.0
        for t, char in enumerate(target_text):
```

```
[ ]     encoder_input_data[i, t + 1 :, input_token_index[" "]] = 1.0
        for t, char in enumerate(target_text):
            # decoder_target_data is ahead of decoder_input_data by one timestep
            decoder_input_data[i, t, target_token_index[char]] = 1.0
            if t > 0:
                # decoder_target_data will be ahead by one timestep
                # and will not include the start character.
                decoder_target_data[i, t - 1, target_token_index[char]] = 1.0
        decoder_input_data[i, t + 1 :, target_token_index[" "]] = 1.0
        decoder_target_data[i, t:, target_token_index[" "]] = 1.0
```

```
Number of samples: 2779
Number of unique input tokens: 70
Number of unique output tokens: 92
Max sequence length for inputs: 107
Max sequence length for outputs: 123
```

Build the model

```
# Define an input sequence and process it.
encoder_inputs = keras.Input(shape=(None, num_encoder_tokens))
encoder = keras.layers.LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)

# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = keras.Input(shape=(None, num_decoder_tokens))

# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = keras.layers.LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = keras.layers.Dense(num_decoder_tokens, activation="softmax")
decoder_outputs = decoder_dense(decoder_outputs)
```

```
[ ] # Define the model that will turn
    # `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
    model = keras.Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

Train the model

```
model.compile(
    optimizer="rmsprop", loss="categorical_crossentropy", metrics=["accuracy"]
)
model.fit(
    [encoder_input_data, decoder_input_data],
    decoder_target_data,
    batch_size=batch_size,
    epochs=epochs,
    validation_split=0.2,
)
# Save model
model.save("s2s_model.keras")
```

```
Epoch 1/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 8s 155ms/step - accuracy: 0.6586 - loss: 2.7116 - val_accuracy: 0.6850 - val_loss: 2.6179
Epoch 2/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 9s 142ms/step - accuracy: 0.8047 - loss: 1.4517 - val_accuracy: 0.6862 - val_loss: 1.7608
Epoch 3/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 142ms/step - accuracy: 0.8029 - loss: 1.1638 - val_accuracy: 0.6862 - val_loss: 1.8610
Epoch 4/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 143ms/step - accuracy: 0.8067 - loss: 1.1105 - val_accuracy: 0.6886 - val_loss: 1.6431
Epoch 5/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 143ms/step - accuracy: 0.8076 - loss: 0.9468 - val_accuracy: 0.6886 - val_loss: 1.4469
Epoch 6/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 144ms/step - accuracy: 0.8066 - loss: 0.8965 - val_accuracy: 0.6890 - val_loss: 1.5109
Epoch 7/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 152ms/step - accuracy: 0.8128 - loss: 0.8604 - val_accuracy: 0.6896 - val_loss: 1.8171
Epoch 8/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 145ms/step - accuracy: 0.8102 - loss: 0.8610 - val_accuracy: 0.6890 - val_loss: 1.4466
Epoch 9/100
35/35 ━━━━━━━━━━━━━━━━━━━━ 5s 146ms/step - accuracy: 0.8114 - loss: 0.8544 - val_accuracy: 0.6896 - val_loss: 1.5188
Epoch 10/100
```

Run inference (sampling)

1. encode input and retrieve initial decoder state
2. run one step of decoder with this initial state and a "start of sequence" token as target. Output will be the next target token.
3. Repeat with the current target token and current states

```
# Define sampling models
# Restore the model and construct the encoder and decoder.
model = keras.models.load_model("s2s_model.keras")

encoder_inputs = model.input[0]  # input_1
encoder_outputs, state_h_enc, state_c_enc = model.layers[2].output  # lstm_1
encoder_states = [state_h_enc, state_c_enc]
encoder_model = keras.Model(encoder_inputs, encoder_states)

decoder_inputs = model.input[1]  # input_2
decoder_state_input_h = keras.Input(shape=(latent_dim,))
decoder_state_input_c = keras.Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_lstm = model.layers[3]
decoder_outputs, state_h_dec, state_c_dec = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs
)
decoder_states = [state_h_dec, state_c_dec]
decoder_dense = model.layers[4]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = keras.Model(
    [decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states
)

# Reverse-lookup token index to decode sequences back to
# something readable.
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())
```

```
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq, verbose=0)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index["\t"]] = 1.0

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ""
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value, verbose=0
        )

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

        # Exit condition: either hit max length
        # or find stop character.
        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, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.0

        # Update states
        states_value = [h, c]
    return decoded_sentence
```

You can now generate decoded sentences as such:

```
for seq_index in range(3):
    # Take one sequence (part of the training set)
    # for trying out decoding.
    input_seq = encoder_input_data[seq_index : seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print("-")
    print("Input sentence:", input_texts[seq_index])
    print("Decoded sentence:", decoded_sentence)
```

```
-
Input sentence: Wow!
Decoded sentence: मैं तुम्हारे पास करता हूँ।
```