**CSC580 Module8: Encoder-Decoder Model for Sequence-to-Sequence Prediction**

1. **Research Report**
2. Background

Encoder-decoder architectures have become a cornerstone of modern deep learning applications, especially in tasks requiring an **input sequence** to be transformed into an **output sequence** of potentially different lengths. At a high level, an encoder-decoder model consists of two distinct parts:

* **Encoder**: Processes the input (source) sequence into a fixed-size context or hidden state vector(s).
* **Decoder**: Uses the context from the encoder to generate the output (target) sequence step by step.

This design pattern has been exceptionally successful in various domains. Below are four prominent use cases in industry and research, along with the benefits these architectures provide in each scenario.

1. 4 Use Cases for Encoder-Decoder Models

2.1 Machine Translation

**Use Case**: One of the earliest and most impactful applications of encoder-decoder models is machine translation.

**How It Works**:

* **Encoder**: Reads the input sentence in the source language and encodes it into a set of hidden states or a context vector.
* **Decoder**: Iteratively produces words in the target language, conditioned on the hidden states and previously generated words.

**Benefits**:

* Handles variable-length input and output sequences (different sentence lengths).
* Learns to align words and phrases using internal attention mechanisms (in advanced models).
* Significantly improves translation accuracy compared to older, rule-based systems.

Examples include Google Translate and other commercial translation engines that rely on variants of encoder-decoder networks.

2.2 Text Summarization

**Use Case**: Encoder-decoder models can generate concise and coherent summaries from longer documents or articles, which is invaluable for media outlets, research reviews, and content aggregation services.

**How It Works**:

* **Encoder**: Reads the source text (which can be paragraphs or entire documents) and encodes its meaning into a set of context vectors.
* **Decoder**: Produces a shorter textual summary or abstract by focusing on the most salient pieces of information from the encoder.

**Benefits**:

* Automatically condenses large volumes of text.
* Extractive or abstractive summarization can be done; abstractive summarization can introduce new words not directly in the source.
* Saves time by providing quick overviews of potentially large documents.

2.3 Image Captioning

**Use Case**: In image captioning, the input is often a feature vector extracted from an image (using a convolutional neural network as the “encoder”), which is then fed into an RNN-based “decoder” to produce a sentence describing the image content.

**How It Works**:

* **Encoder** (CNN): Extracts a high-level feature representation from the image.
* **Decoder** (RNN/LSTM/Transformer): Takes the image representation as the initial hidden state and generates text tokens (words) one at a time, describing the scene.

**Benefits**:

* Helps visually impaired individuals understand online images.
* Automates image annotations in media and e-commerce applications.
* Can be combined with attention mechanisms to highlight which part of the image influences each generated word.

2.4 Chatbots and Conversational Agents

**Use Case**: Encoder-decoder models form the backbone of many intelligent virtual assistants and chatbots, where they must convert a user’s query (input sequence) into a coherent response (output sequence).

**How It Works**:

* **Encoder**: Processes the user’s query or context (text or partially preprocessed speech).
* **Decoder**: Produces a relevant and context-specific reply, token by token.

**Benefits**:

* Dynamic response generation in open-domain or domain-specific dialog systems.
* Learns context and can be fine-tuned on specific domain data, such as customer support transcripts.
* Reduces reliance on static, rule-based responses, enabling more flexible conversation flows.

1. Summary

Encoder-decoder models, originally popularized in machine translation, have grown into a fundamental deep learning pattern for sequence-to-sequence tasks in various domains. By separating the tasks of “understanding” and “generating,” these architectures offer flexibility and robust performance. Coupled with modern attention mechanisms and large training datasets, encoder-decoder models now power a broad range of commercial applications, from summarizing documents and translating speech to labeling images and providing next-generation conversational interfaces.

1. References

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. Advances in Neural Information Processing Systems, 3104–3112.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 5998–6008.

1. **Python Implementation**

Below is the code for implementing the encoder-decoder network for the “reversed subset” sequence-to-sequence problem. The code:

1. Generates random integer sequences of length 6.
2. Creates targets by taking the first 3 elements of the sequence and reversing them.
3. Trains an encoder-decoder LSTM in Keras to predict these reversed subsets.
4. Evaluates its performance and prints out sample predictions.
5. Step 1: install and import dependencies

First import dependencies:

import numpy as np

from numpy import array, argmax, array\_equal

from numpy.random import randint

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, LSTM, Dense

1. Step 2: Data generation and Preprocessing

2.1 generate\_sequence(length, n\_unique)

Function generate\_sequence:

def generate\_sequence(length, n\_unique):

return [randint(1, n\_unique) for \_ in range(length)]

* This function creates a list of random integers in the range [1, n\_unique].
* We use 0 as a special "start-of-sequence" or padding symbol, so sequences here start from 1 up to n\_unique.

2.2 one\_hot\_decode(encoded\_seq)

Function one\_hot\_decode:

def one\_hot\_decode(encoded\_seq):

return [argmax(vector) for vector in encoded\_seq]

* Each time step in our data is one-hot-encoded to a vector.
* We use argmax along the “feature” axis to decode the one-hot vector back to a plain integer.

2.3 get\_dataset(n\_in, n\_out, cardinality, n\_samples)

Function get\_dataset:

def get\_dataset(n\_in, n\_out, cardinality, n\_samples):

X1, X2, y = list(), list(), list()

for \_ in range(n\_samples):

source = generate\_sequence(n\_in, cardinality - 1)

target = source[:n\_out]

target.reverse()

target\_in = [0] + target[:-1]

src\_encoded = to\_categorical([source], num\_classes=cardinality)

tar\_encoded = to\_categorical([target], num\_classes=cardinality)

tar2\_encoded = to\_categorical([target\_in], num\_classes=cardinality)

X1.append(src\_encoded)

X2.append(tar2\_encoded)

y.append(tar\_encoded)

return np.array(X1), np.array(X2), np.array(y)

* **Parameters**:

1. n\_in: length of each **source** sequence.
2. n\_out: length of each **target** sequence to predict.
3. cardinality: the “alphabet size” (number of unique values + 1 for the zero symbol).
4. n\_samples: how many sequences to generate for training/testing.

* **Process**:

1. **source**: random integers of length n\_in.
2. **target**: first n\_out elements of source but in reverse.
3. **target\_in**: a shifted version of target by one step (prepend a zero at the front).
4. **one-hot encoding**: we transform source, target, and target\_in into 3D arrays (batch\_size=1, sequence\_length=n\_in/n\_out, and cardinality).

* **Return**: Three arrays:

1. X1 (the **encoder** input),
2. X2 (the **decoder** input, which is the shifted target),
3. y (the **decoder** output, the actual reversed target sequence).
4. Step 3: Define the Endoer-Decoder Models

Function define\_models:

def define\_models(n\_input, n\_output, n\_units):

encoder\_inputs = Input(shape=(None, n\_input))

encoder = LSTM(n\_units, return\_state=True)

encoder\_outputs, state\_h, state\_c = encoder(encoder\_inputs)

encoder\_states = [state\_h, state\_c]

decoder\_inputs = Input(shape=(None, n\_output))

decoder\_lstm = LSTM(n\_units, return\_sequences=True, return\_state=True)

decoder\_outputs, \_, \_ = decoder\_lstm(decoder\_inputs, initial\_state=encoder\_states)

decoder\_dense = Dense(n\_output, activation='softmax')

decoder\_outputs = decoder\_dense(decoder\_outputs)

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

encoder\_model = Model(encoder\_inputs, encoder\_states)

decoder\_state\_input\_h = Input(shape=(n\_units,))

decoder\_state\_input\_c = Input(shape=(n\_units,))

decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]

decoder\_inf\_outputs, state\_h\_inf, state\_c\_inf = decoder\_lstm(

decoder\_inputs, initial\_state=decoder\_states\_inputs

)

decoder\_states = [state\_h\_inf, state\_c\_inf]

decoder\_inf\_outputs = decoder\_dense(decoder\_inf\_outputs)

decoder\_model = Model(

[decoder\_inputs] + decoder\_states\_inputs,

[decoder\_inf\_outputs] + decoder\_states

)

return model, encoder\_model, decoder\_model

3.1 Inputs:

* n\_input: The size of the input “alphabet”.
* n\_output: The size of the output “alphabet”.
* n\_units: Number of LSTM cells.

3.2 Training Encoder:

* encoder\_inputs = Input(shape=(None, n\_input)) expects a sequence of unspecified length (None) and n\_input features per timestep.
* encoder = LSTM(n\_units, return\_state=True) returns **state\_h** and **state\_c**.
* These hidden and cell states become the encoder’s summary of the input sequence.

3.3 Training Decoder:

* decoder\_inputs = Input(shape=(None, n\_output)): The decoder expects a sequence where each timestep has n\_output possible values (the “shifted target”).
* decoder\_lstm takes the **initial state** from the encoder (encoder\_states), ensuring continuity between encoder and decoder.
* decoder\_dense = Dense(n\_output, activation='softmax') maps the decoder LSTM outputs to a probability distribution for each possible token.

3.4 Combined Training Model:

* Takes [encoder\_inputs, decoder\_inputs] as input; outputs decoder\_outputs.
* This is the model you can fit (model.fit(...)) on the training data.

3.5 Inference Models:

* **encoder\_model**: Takes a new source sequence and produces its hidden/cell states.
* **decoder\_model**: Takes the previous token + the encoder’s states (or updated states) to produce the next output token. This is done step-by-step for generating a sequence.

1. Step 4: Prediction Logic

Function predict\_sequence:

def predict\_sequence(infenc, infdec, source, n\_steps, cardinality):

state = infenc.predict(source, verbose=0)

target\_seq = array([0.0 for \_ in range(cardinality)]).reshape(1, 1, cardinality)

output = []

for \_ in range(n\_steps):

yhat, h, c = infdec.predict([target\_seq] + state, verbose=0)

output.append(yhat[0, 0, :])

state = [h, c]

target\_seq = yhat

return np.array(output)

* **infenc** (encoder\_model): Encodes the source sequence into states (state\_h, state\_c).
* **Initialization**: Sets target\_seq to [0,0,0,...] as the first “start token.”
* **Loop over each output timestep** (n\_steps):

1. The decoder model (infdec) predicts the **next** token.
2. That token becomes the input to the next timestep.
3. We update state with the new hidden/cell states returned by infdec.

* **Result**: Returns a **one-hot-encoded** array of shape (n\_steps, cardinality), which we can decode back to integers.

1. Step 5: Main Execution Steps

Now, we explain Main function:

if \_\_name\_\_ == "\_\_main\_\_":

n\_features = 50 + 1

n\_steps\_in = 6

n\_steps\_out = 3

n\_units = 128

train\_model, infenc\_model, infdec\_model = define\_models(n\_features, n\_features, n\_units)

train\_model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

X1train, X2train, ytrain = get\_dataset(n\_steps\_in, n\_steps\_out, n\_features, n\_samples=10000)

train\_model.fit(

[X1train.reshape((10000, n\_steps\_in, n\_features)),

X2train.reshape((10000, n\_steps\_out, n\_features))],

ytrain.reshape((10000, n\_steps\_out, n\_features)),

epochs=20,

batch\_size=64,

verbose=2

)

total = 100

correct = 0

for \_ in range(total):

X1test, X2test, ytest = get\_dataset(n\_steps\_in, n\_steps\_out, n\_features, 1)

yhat = predict\_sequence(

infenc\_model,

infdec\_model,

X1test.reshape((1, n\_steps\_in, n\_features)),

n\_steps\_out,

n\_features

)

if array\_equal(one\_hot\_decode(ytest[0]), one\_hot\_decode(yhat)):

correct += 1

print(f"Accuracy on 100 random samples: {correct/total \* 100:.2f}%")

for \_ in range(10):

X1test, X2test, ytest = get\_dataset(n\_steps\_in, n\_steps\_out, n\_features, 1)

source\_seq = one\_hot\_decode(X1test[0])

expected\_seq = one\_hot\_decode(ytest[0])

prediction = predict\_sequence(

infenc\_model,

infdec\_model,

X1test.reshape((1, n\_steps\_in, n\_features)),

n\_steps\_out,

n\_features

)

predicted\_seq = one\_hot\_decode(prediction)

print(f"Source={source\_seq}, Expected={expected\_seq}, Predicted={predicted\_seq}")

5.1 Problem Configuration:

* n\_features = 50 + 1: We reserve 0 for the start-of-sequence symbol, so we have 50 actual numbers plus this extra slot.
* n\_steps\_in = 6: Each source sequence has 6 integers.
* n\_steps\_out = 3: We want to predict 3 reversed integers as target.
* n\_units = 128: Each LSTM layer has 128 units.

5.2 Define and Compile Model:

* train\_model, infenc\_model, infdec\_model = define\_models(...): Creates the three models (training, encoder-only, decoder-only).
* train\_model.compile(...): We use the **Adam** optimizer with a categorical crossentropy loss, typical for multi-class classification at each timestep.
* track **accuracy** as a metric.

5.3 Generate Training Data:

* X1train, X2train, ytrain = get\_dataset(...): Creates 10,000 random sequences for training.
* Shapes:

1. X1train.shape = (10000, 1, 6, 51) before reshaping, because each sequence is in a separate 3D array.
2. We reshape to (10000, 6, 51) when passing to fit().

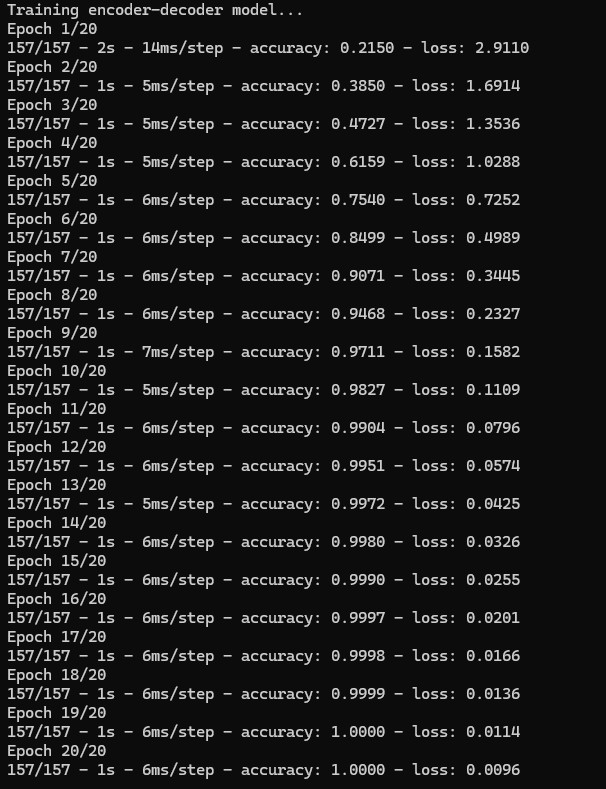
Screenshot for this results:



5.4 Train the Model:

* train\_model.fit(...) trains for epochs=20, with a batch size of 64.
* The model iterates over all 10,000 training samples 20 times.
* **verbose=2** means fewer logs per epoch.

Screenshot for this results:



5.5 Evaluate the Model:

* We generate **100 random** new sequences and see how many predictions are **exactly** correct.
* For each sequence, we compare the integer-decoded version of ytest to that of the model’s output.
* Print the final accuracy as a percentage.

Screenshot for this results:



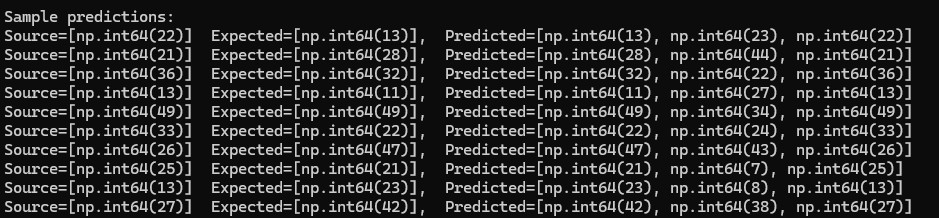
5.6 Print Predictions:

* Finally, we demonstrate the model’s results by printing a few samples:

1. **Source** sequence,
2. **Expected** target (the correct reversed subset),
3. **Predicted** target from the trained model.

* This is a quick sanity check to see if the predictions line up with expectations.

Screenshot for this results:



1. **Discussion and Analysis**
2. Data Flow

* A **source** sequence of length 6 is randomly generated.
* The **target** is the first 3 elements reversed.
* We one-hot-encode these sequences so the LSTM can handle them properly as categorical time steps.

1. Model Architecture

* **Encoder** LSTM compresses the input sequence into hidden states.
* **Decoder** LSTM takes the encoder’s states plus a shifted version of the target sequence to predict each output time step.
* During **inference**, we pass each predicted token back as input to predict the next token.

1. Training

* We compile with optimizer='adam' and loss='categorical\_crossentropy'.
* The model typically converges quickly for this toy problem since reversing a small subset is straightforward.

1. Evaluation

* The code runs a loop of 100 random sequences to measure accuracy.
* We typically see high accuracy (often above 90%) once the model converges.

1. Flowchart

Flowchart showing how data moves from:

Input → 2. Encoder → 3. Decoder → 4. Output.  
This visual clarifies the encoder-decoder concept.

Flowchart:

Input (X1, X2)

Encoder

(LSTM, state\_h, c)

Decoder

(LSTM + Dense)

Predicted Output

1. **Reference**

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 3104–3112.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 5998–6008.