Session Summary

Introduction to Machine Translation

In today's global society, more and more people from different regions in the world need to interact with each other on a regular basis. These people who speak languages particular to their locations would not be able to communicate with each other if not for effective language translation.

But when the text or speech is long, it comes down to Machine Translation (MT) to translate the content quickly without a lot of human involvement to remove the language barriers created. One such example of the usage of Machine Translation from our daily lives is how Google, Facebook, and other digital giants integrate MT in their systems to translate a post, a comment or text on the search result page to enhance user experience. As you must have observed, these translations happen in a matter of seconds, thanks to Machine Translation.

Let's begin with the key take-aways from the first segment that helped you learn about the evolution of MT.

Evolution of Machine Translation

Machine Translation has developed a lot to become the sophisticated set of tools it is today which maps input sentences present in one language to sentences in another language.

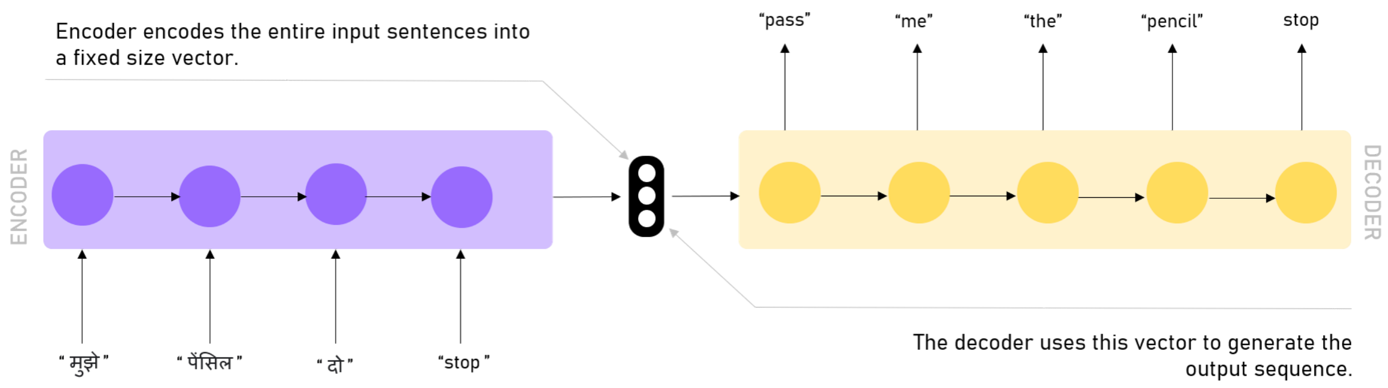
Here are the highlights of how MT evolved over the years:

- Before 1949, translation was a labour-intensive process that depended exclusively on human translators.
- In 1949, Warren Weaver ideated language translation using machines. This led to the development of **rule-based machine translation (RBMT)** which was based on the use of dictionaries and grammars that helped analyse the linguistic relationship between the source and the target language.
- Across 1990s, **statistical machine translation (SMT)** approach replaced RBMT approach, as it performed better by integrating statistical models to find the statistically most likely translation for use.
- Yoshua Bengio developed the **neural-based language models** which enabled better understanding of language. This model laid the foundation for using neural networks in machine translation.
- In 2014, the **sequence to sequence (seq2seq)** model was developed by Sutskever et al. and Cho et al. for the task of neural machine translation.
- The model seq2seq uses long short-term memory (LSTM) for both encoder and decoder to frame end-to-end learning.

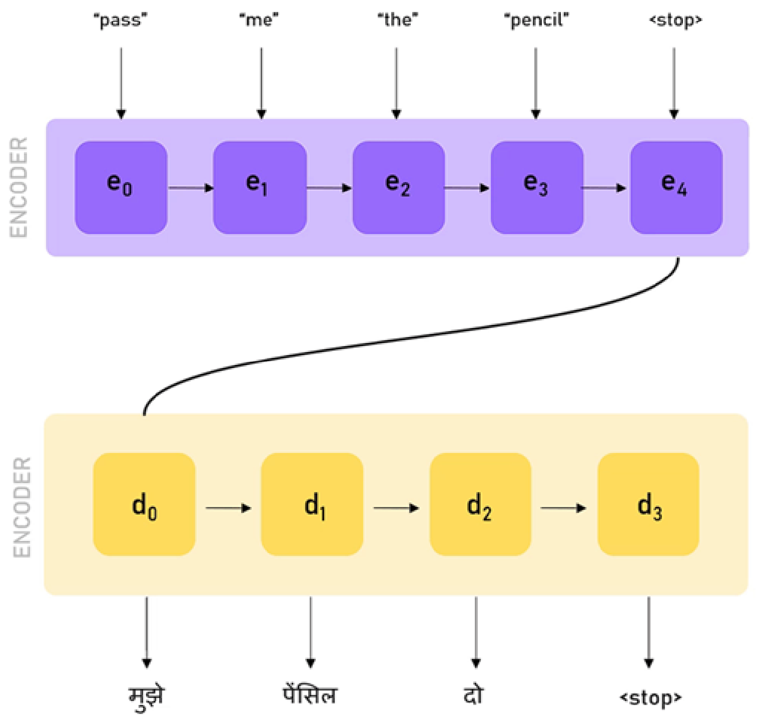
Let’s move to the next segment where you learnt about the encoder-decoder architecture used in the sequence to sequence model.

Understanding the Encoder-Decoder Architecture for NMT

Sequence-to-sequence (seq2seq) model follows an encoder-decoder architecture, where it is made up of two RNN’s. Both the encoder & decoder consists of a series of RNN cells where each layer’s output is the input to the next layer.


Encoder encodes the entire input sentences into a fixed size vector.
ENCODER
“मुझे”
“पेंसिल”
“दो”
“stop”
“pass”
“me”
“the”
“pencil”
stop
DECODER
The decoder uses this vector to generate the output sequence.

A quick look at the encoder-decoder architecture used for MT:



The seq2seq model is made up of two recurrent neural networks (RNN). The encoder and the decoder consist of a series of RNN cells where every cell’s output acts as the input for the next cell.

Here’s the process followed in this architecture:

- The encoder RNN takes the input sequence and encodes it into a fixed size context vector by reading the input tokens one at a time.
- Once all the input tokens are read by the encoder, a special token is passed to the encoder –<stop>/<end>. This token (which can be named anything as its just an indicator for the model) indicates the encoder to stop encoding and pass the last cell's hidden state to the decoder.
- The context vector generated by the final cell's hidden state is then fed to the decoder as the input.
- Along with the context vector, the decoder receives a special token –<start>/<sos> (start of sentence) that indicates the model to start decoding.
- The decoder RNN uses the context vector to generate the output sequence. The first cell of the decoder is initialised with the hidden state that it received from the encoder.
- The decoder acts as a language model and predicts the next word based on the previous prediction and the hidden step passed from the cell at the previous time step.
- Once the decoder has provided the relevant translation, a special token is generated – <end>/<eos> indicating the end of the target sentence.

This entire seq2seq/NMT model is called a **conditional language model** because:

- Decoder predicts based on the context input (condition) received from the encoder, making it **conditional**
- Decoder predicts the next word based on the previous prediction, bringing in the feature of **language**

Now let's move to the next segment to understand the special conditions that must be satisfied for the NMT model to produce effective translation.

Requirements of NMT Architecture

Here are the required characteristics for an NMT architecture to produce good translation:

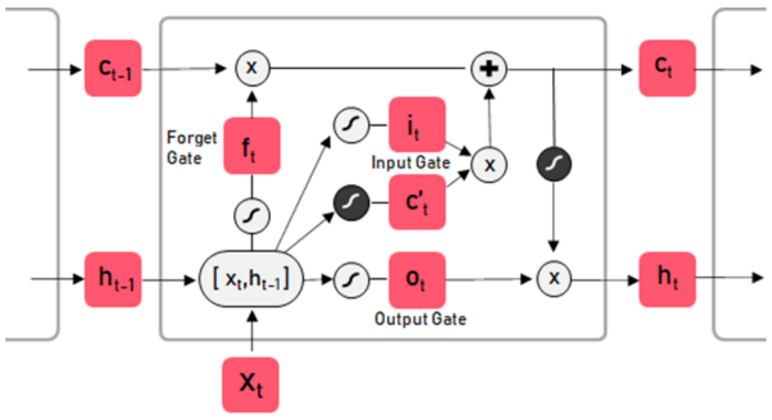
- **Handle sequences of variable length:** It should be able to intake any length input and generate any length output.
- **Maintain the correct information:** While providing translation, the decoder should maintain the syntactic as well as the semantic information of the input sequence without changing the meaning.
- **Share parameters across the sequences:** All parameters should be shared such that during the backpropagation, both the encoder and the decoder could be trained simultaneously.
- **Track long-term dependencies:** It should be able to remember information over long periods of time.

NMT Model vs RNN Vanilla Model:

Due to the problem of vanishing and exploding gradients in RNNs, the RNN Vanilla model is unable to generate long sequences. But this is solved by the long, short-term memory network (LSTM).

Different gating mechanisms and explicit memory allows for the LSTM memory to be updated or deleted in a manner that allows only the relevant part of the information to be kept at all times. This mechanism controls the problem of exploding/vanishing gradients and helps in retaining information about long-term dependencies.

Here's the architecture of an LSTM cell:



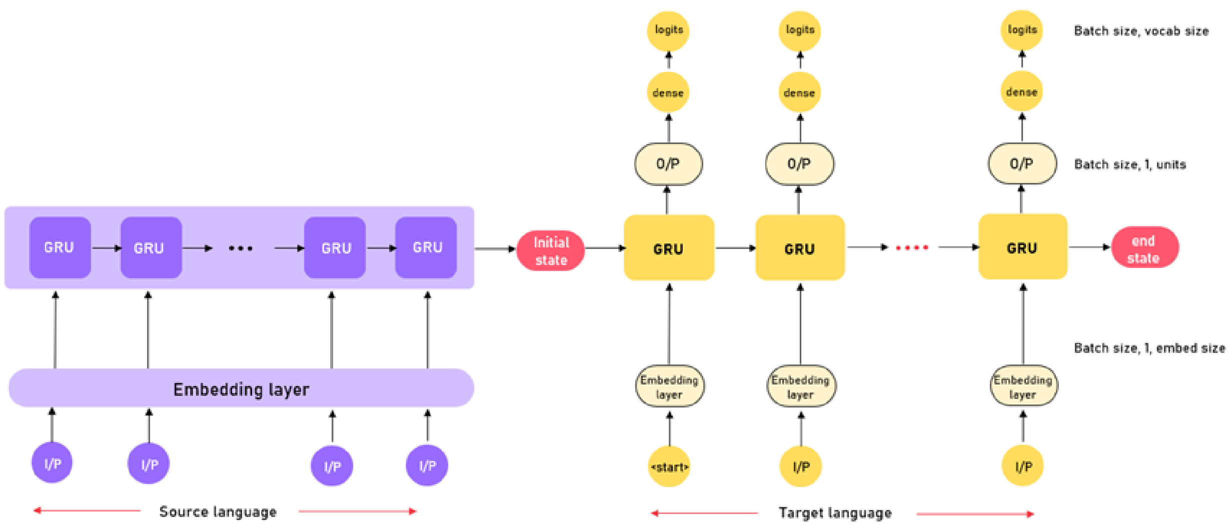
GRU Model:

A variation of LSTM, Gated recurrent unit (GRU) produces equally good results. By design and training the GRU is simpler than LSTM and has only two gates compared to its counterpart. This makes it computationally faster, making it the preferred choice to build bigger models.

Now, let's move to the next segment that covered the process of training NMT architecture.

Model Training and Prediction

The encoder-decoder training architecture for NMT:

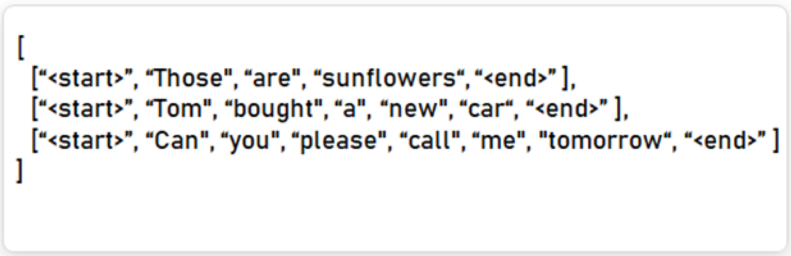


Pre-requirements to training a module:

- The input sequence to the NMT model needs to be of fixed length and, generally, it is limited to the maximum length of a sentence present in the sample data.
- The shape of the input data should be of (batch_size, max_length). Therefore, for sentences shorter than the maximum length, paddings (empty tokens) are added to make all sequences in a batch fit a given standard length.

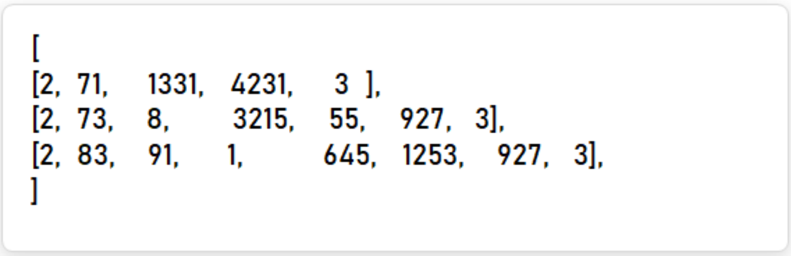
Inserting Paddings:

An example of inserting paddings that consists tokenized words:



```
[
 [“<start>”, “Those”, “are”, “sunflowers“, “<end>” ],
 [“<start>”, “Tom”, “bought”, “a”, “new”, “car“, “<end>” ],
 [“<start>”, “Can”, “you”, “please”, “call”, “me”, "tomorrow“, “<end>” ]
]
```

Converting these sentences to their respective tokens, the data will look like follows:

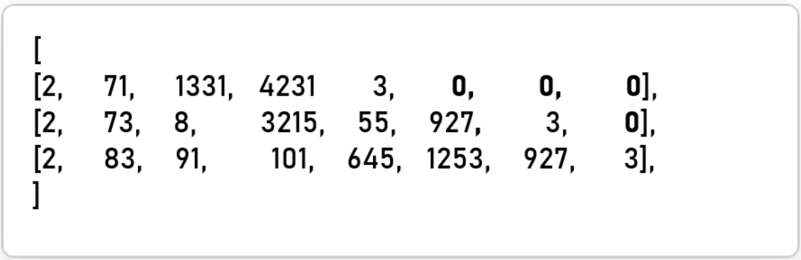


```
[
[2,  71,    1331,   4231,    3  ],
[2,  73,    8,       3215,   55,    927,   3],
[2,  83,    91,     1,         645,   1253,    927,   3],
]
```

The maximum length for the samples is 8. So, samples shorter than the 8 need to be padded with empty(0) tokens.

You can pad the samples to max length by using **tf.keras.preprocessing.sequence.pad_sequences**

```
padded_inputs = tf.keras.preprocessing.sequence.pad_sequences(tokens, padding="post")
print(padded_inputs)
```



```
[
[2,    71,    1331,   4231      3,      0,       0,       0],
[2,    73,    8,       3215,   55,    927,       3,       0],
[2,    83,    91,       101,   645,   1253,    927,       3],
]
```

Note: Since the padding is kept as ‘post’, all the empty tokens are added after each sequence. It is recommended to keep post padding when working with RNN cells.

Training Process:

Since the decoder is trained to predict the next characters of the target sequence given the previous prediction, if the model produces a wrong prediction at a certain timestep, all the following predictions can come our incorrect.

This problem can be solved by training the model using **teacher forcing**, where the model is fed with the correct target token at each timestep, regardless of the model's predictions at previous timestep making the model more robust.

Therefore, the decoder learns to generate target[: i+1] given target[: i].
The process followed for training the model is given below:

1. Once the decoder receives the context vector, it is fed with an input token <start>.
2. Based on this input, the trained decoder produces a list of predictions (probability scores) for the first word.
3. By taking the argmax of the predictions, the next word is computed and appended to an empty list 'Result'.
4. This previous prediction, stored in 'Result', is then sent as an input to the model in the next timestep.
5. The entire process is repeated until the model produces an <end> token, which indicates the end of the prediction process.

BLEU Score:

The BLEU score is a well-acknowledged evaluation metric employed for model prediction which determines the 'difference' between the predicted sentence from the human-created sentence.

It measures the similarity of one hypothesis sentence to multiple reference sentences by returning a value between 0 and 1. The metric close to 1 means that the two are very similar.

The next segment covers the implementation of the NMT model in TensorFlow.

Implementing NMT using TensorFlow

The goal is to convert a sentence from the source language to the target language.

Data preparation process:

The pre-processing steps can be summarised as :

- Create the tokenized vectors by tokenizing the sentences and stripping the input and output of extra unnecessary characters. This gives us a vocabulary of all of the unique words in the data. Keep the total number of sentences to 70000 for saving memory.
- Add <sos> (start-of-sentence) and <eos> (end-of-sentence) to each sentences.
- Create word-to-index and index-to-word mappings.
- Pad all sequences to be the same length as the longest one.

The next step is to split the input & target data into training and validation sets using an 80-20 split.

After creating the train and validation set, the next task is to create the tf.dataset.

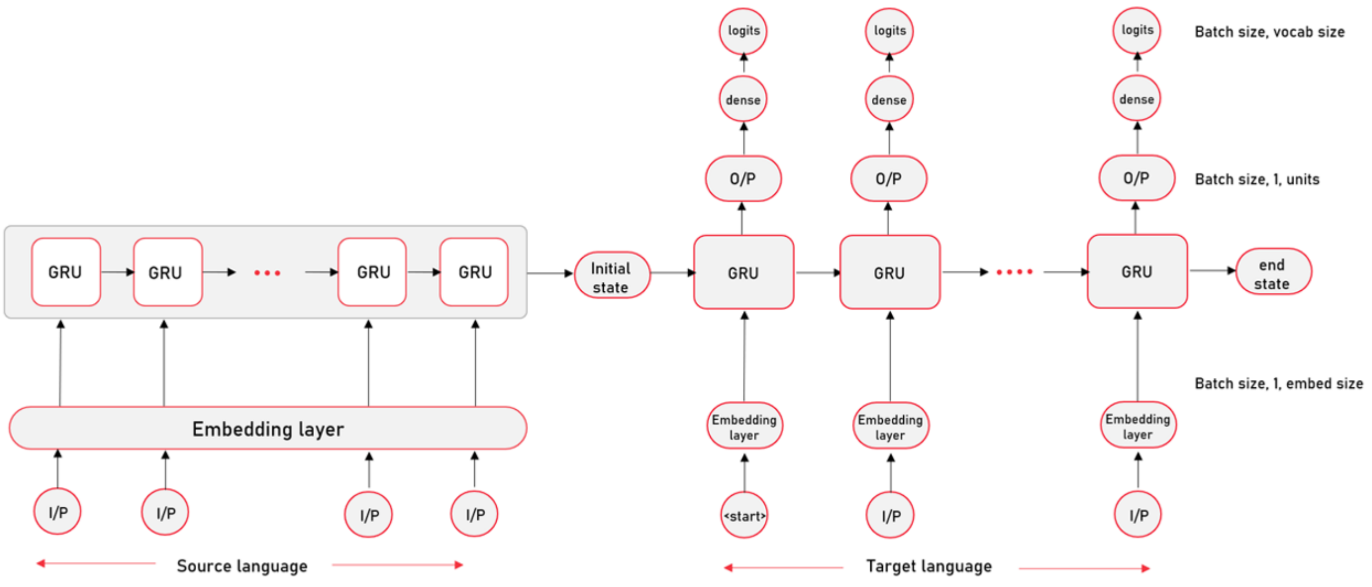
Once you have created the training dataset, the shape of the input_sequence and target_sequence in the dataset should be of an order [batch_size,max_length].

```
example_input_batch, example_target_batch = next(iter(dataset))
example_input_batch.shape, example_target_batch.shape
```

(TensorShape([64, 72]), TensorShape([64, 69]))

where 64 is the batch_size, 72 is the max_length of the input_sequence and 69 is the max_length of the output_sequence.

The Training Process for the seq2seq model:



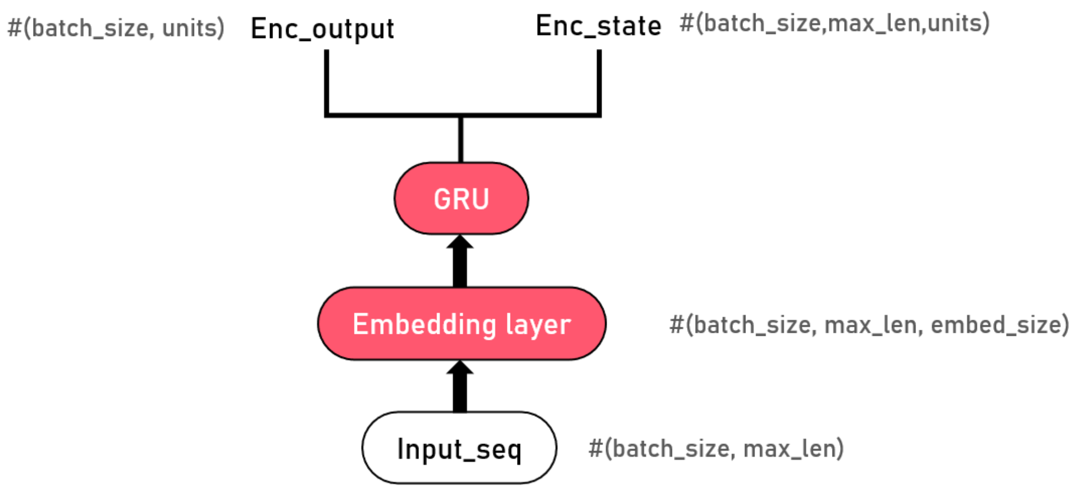
Encoder-Decoder training architecture for NMT

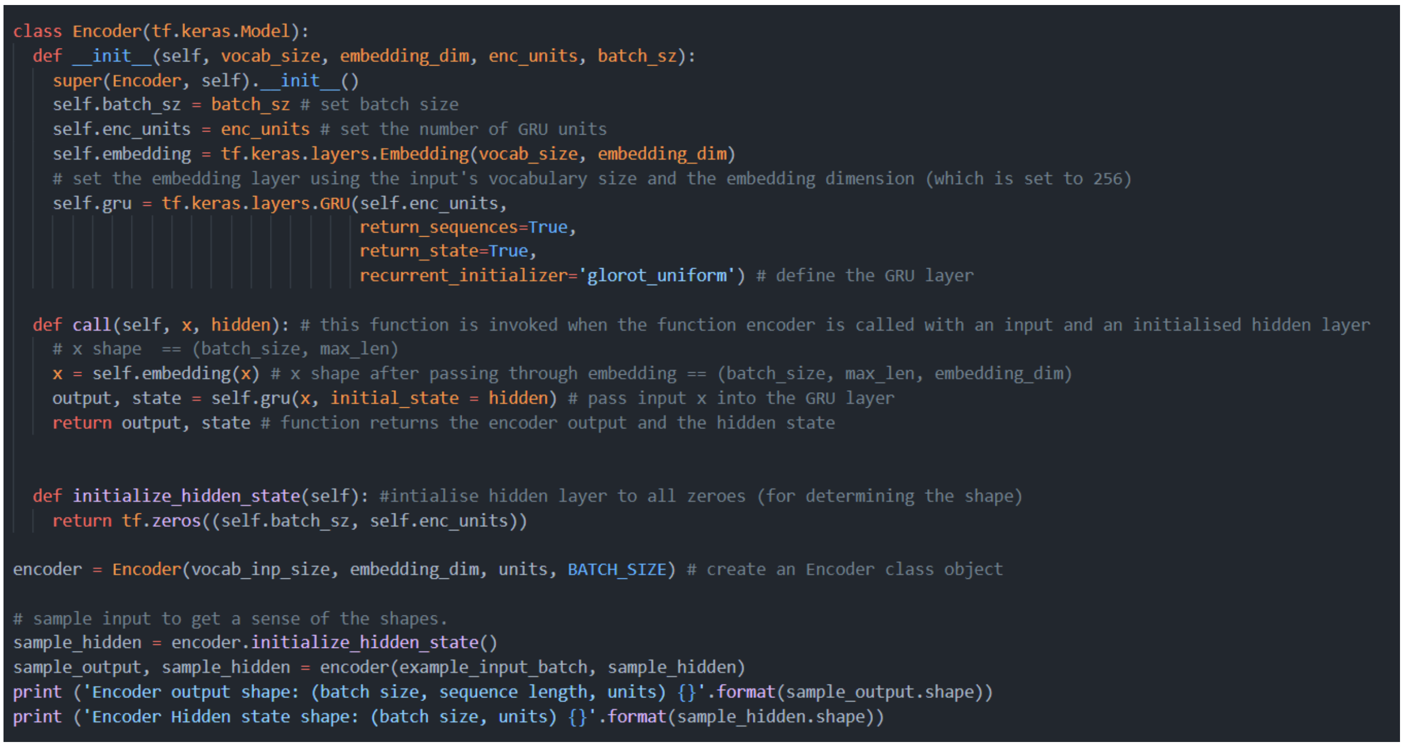
Let's break down the encoder and decoder separately and see the components.

Encoder model:

- It consists of an Embedding layer(layers.Embedding) which creates an embedding vector for each token ID.
- Once the embeddings are generated the GRU(layers.GRU)units processes them to a new sequence .

After processing the entire sequence the model returns the encoder output and the hidden state.





```python
class Encoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
    super(Encoder, self).__init__()
    self.batch_sz = batch_sz # set batch size
    self.enc_units = enc_units # set the number of GRU units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    # set the embedding layer using the input's vocabulary size and the embedding dimension (which is set to 256)
    self.gru = tf.keras.layers.GRU(self.enc_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform') # define the GRU layer

  def call(self, x, hidden): # this function is invoked when the function encoder is called with an input and an initialised hidden layer
    # x shape  == (batch_size, max_len)
    x = self.embedding(x) # x shape after passing through embedding == (batch_size, max_len, embedding_dim)
    output, state = self.gru(x, initial_state = hidden) # pass input x into the GRU layer
    return output, state # function returns the encoder output and the hidden state


  def initialize_hidden_state(self): #intialise hidden layer to all zeroes (for determining the shape)
    return tf.zeros((self.batch_sz, self.enc_units))

encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE) # create an Encoder class object

# sample input to get a sense of the shapes.
sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample_output.shape))
print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden.shape))
```

Once you have built your encoder successfully, you can observe:

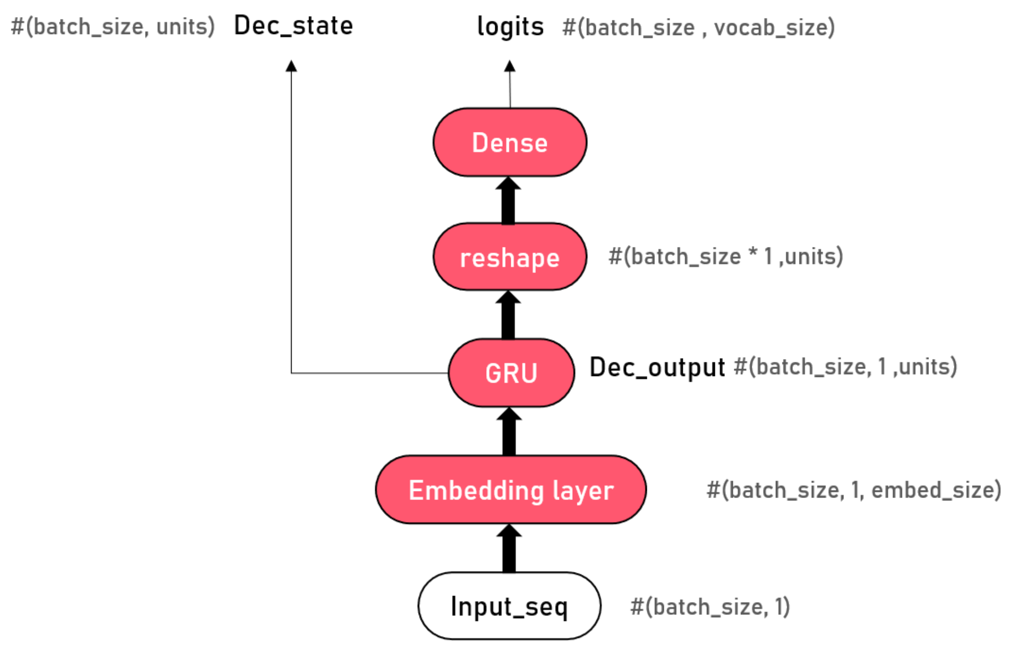
Encoder output shape: **(batch size, sequence length, units)** = (64, 72, 1024)

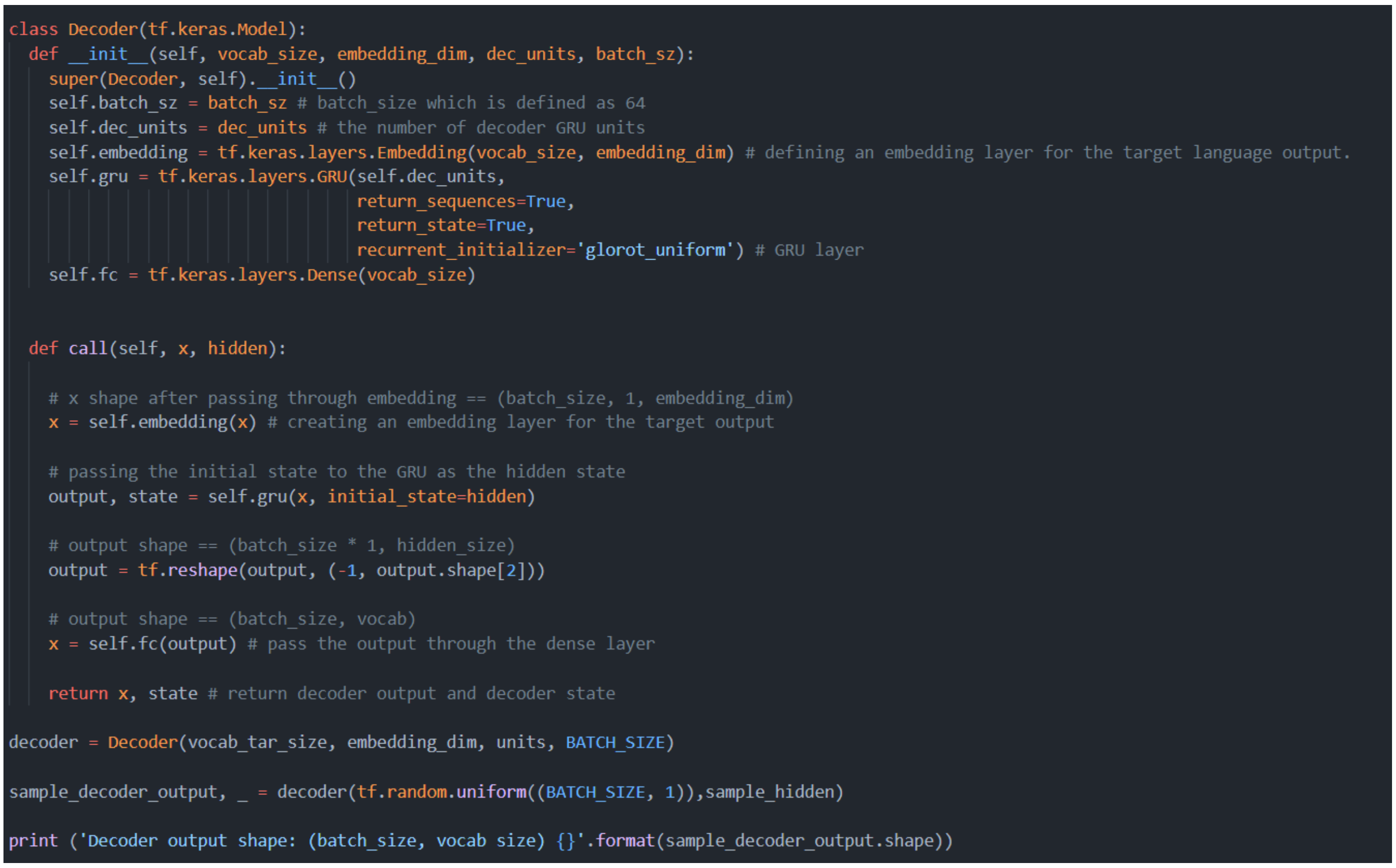
Encoder Hidden state shape: **(batch size, units)** = (64, 1024)

Decoder model:

- It is initialised with the hidden state from the encoder.
- The embedding layer present in it creates an embedding vector for the target output. The initial input to the layer will be <sos> tag and in the shape of **(batch_size,1) as you are passing one token at each timestep.**

- These vectors are then passed on to GRU units which creates output and hidden state.
- The hidden state is fed to the next GRU cell and the output is passed the output through a dense layer.


#(batch_size, units) Dec_state
logits #(batch_size , vocab_size)
Dense
reshape
#(batch_size * 1 ,units)
GRU
Dec_output #(batch_size, 1 ,units)
Embedding layer
#(batch_size, 1, embed_size)
Input_seq
#(batch_size, 1)



```python
class Decoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
    super(Decoder, self).__init__()
    self.batch_sz = batch_sz # batch_size which is defined as 64
    self.dec_units = dec_units # the number of decoder GRU units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim) # defining an embedding layer for the target language output.
    self.gru = tf.keras.layers.GRU(self.dec_units,
                                   return_sequences=True,
                                   return_state=True,
                                   recurrent_initializer='glorot_uniform') # GRU layer
    self.fc = tf.keras.layers.Dense(vocab_size)


  def call(self, x, hidden):

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x) # creating an embedding layer for the target output

    # passing the initial state to the GRU as the hidden state
    output, state = self.gru(x, initial_state=hidden)

    # output shape == (batch_size * 1, hidden_size)
    output = tf.reshape(output, (-1, output.shape[2]))

    # output shape == (batch_size, vocab)
    x = self.fc(output) # pass the output through the dense layer

    return x, state # return decoder output and decoder state

decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)

sample_decoder_output, _ = decoder(tf.random.uniform((BATCH_SIZE, 1)),sample_hidden)

print ('Decoder output shape: (batch_size, vocab size) {}'.format(sample_decoder_output.shape))
```

After creating the decoder model successfully, you can observe:

Decoder output shape: **(batch_size, vocab size)** = (64, 22224)

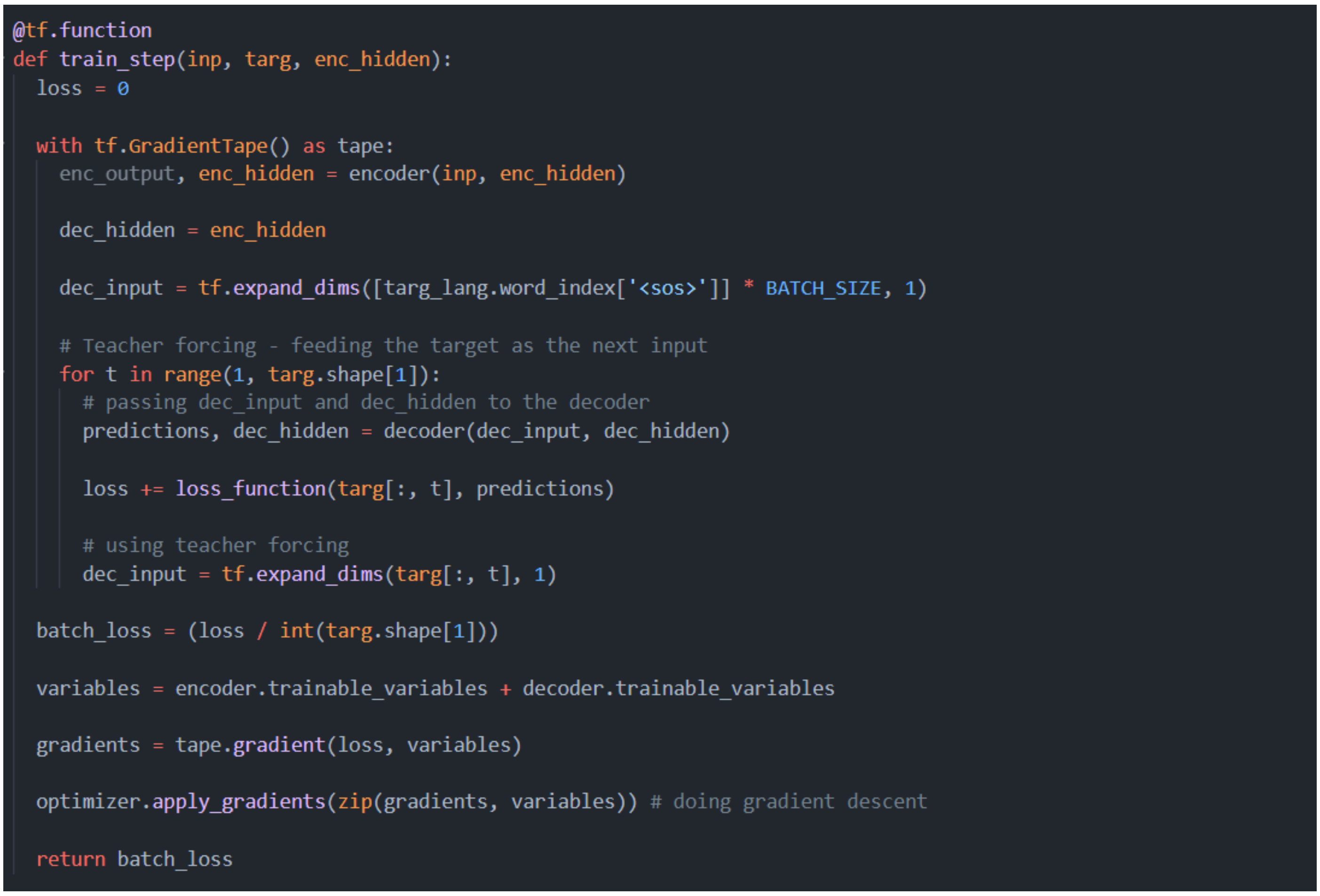
After building all the encoder and decoder models you need to set up the training process for the entire NMT architecture. For this, you have to do the following:

- Set the optimizer(Adam) & loss object(SparseCategoricalCrossentropy).
- Create your checkpoint path.
- Create your training step functions, which will define how the model will be updated for each input/target batch.

The entire sequence of the model training can be summarised as follows:

Overall the implementation for the Model.train_step method is as follows:

1. input_sequence, target_sequence is received as a batch from the `training dataset` and passed to train_step .
2. Theinput_sequence is fed to the encoder along with the hidden state to produce enc_output and enc_hidden.
3. The decoder is initialised with the enc_hiddenand is employed with teacher forcing where the target is fed as the next input. Therefore the prediction process is looped over the entire target_sequence :
   - The decoder is run one step at a time. Once the predictions are created the **loss is calculated for each step/word.**
   - **Teacher forcing** is applied to change the decoder input to **targ[: , t],** which denotes the target word at the t th timestep present in the batch.
4. The average loss is accumulated(by dividing the loss by sentence length) to calculate the batch_loss.
5. The gradient is calculated using thetape.gradient and all thetrainable_variables (Encoder and Decoder) are updated using the optimizer.
6. The model is saved at the checkpoint path after every 2 epochs.



```python
@tf.function
def train_step(inp, targ, enc_hidden):
  loss = 0

  with tf.GradientTape() as tape:
    enc_output, enc_hidden = encoder(inp, enc_hidden)

    dec_hidden = enc_hidden

    dec_input = tf.expand_dims([targ_lang.word_index['<sos>']] * BATCH_SIZE, 1)

    # Teacher forcing - feeding the target as the next input
    for t in range(1, targ.shape[1]):
      # passing dec_input and dec_hidden to the decoder
      predictions, dec_hidden = decoder(dec_input, dec_hidden)

      loss += loss_function(targ[:, t], predictions)

      # using teacher forcing
      dec_input = tf.expand_dims(targ[:, t], 1)

  batch_loss = (loss / int(targ.shape[1]))

  variables = encoder.trainable_variables + decoder.trainable_variables

  gradients = tape.gradient(loss, variables)

  optimizer.apply_gradients(zip(gradients, variables)) # doing gradient descent

  return batch_loss
```

Once the model is trained you need to create a evaluate function to run the model inference.

Overall the function is similar to train_step except that the input to the decoder at each time step is the last prediction from the decoder.

The major changes to the function are as follows:

- The input text is processed first by preprocess_sentence and padded using the keras.preprocessing.sequence.pad_sequences .
- Once the text is converted to their respective token IDs, the trained decoder will produce a list of predictions (probability scores) for the first word. By taking the argmax of the predictions, the next word is computed and appended to an empty list 'Result'.
- This previous prediction, stored in 'Result', is then sent as an input to the model in the next timestep.

The entire process is repeated until the model produces an <eos> token, which indicates the end of the prediction process.

This brings us to the end of NMT implementation.

With this you have reached the end of the session.