Session Summary

Attention-Based NMT Model

Context is key to every communication, which is why it is also the most important aspect of language translations. As you saw, in traditional encore-decoder models, the model is unable to retain the information and thus cannot understand the context of the input system.

This is where the Attention-Based NMT model comes into the picture!

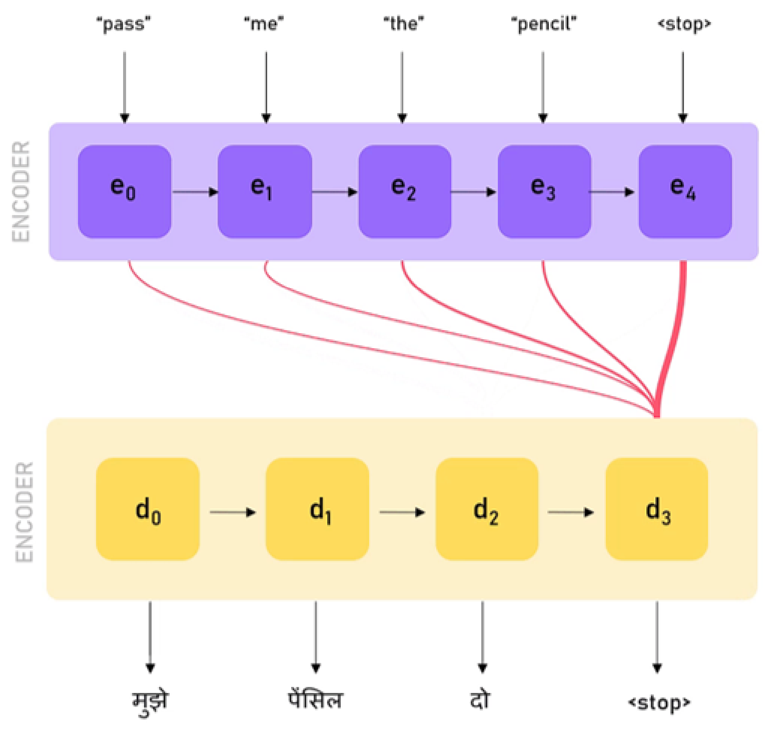
Let's go to the first segment that covered how the NMT model can be expanded to solve for intuition.

Intuition to Attention Mechanism

In the traditional model it is assumed that the context vector can be summarised using the last timestep of the encoder. However, in the Attention-Based NMT model, all the encoder states are taken and for each decoder state the weights of encoder' states are varied according to their perceived importance.

The weighted average of this information provides the new context vector at each timestep of the decoder. This technique is called an **attention mechanism** which becomes the interpreter between the encoder and the decoder.

Here's the encoder-decoder architecture for the Attention-Based NMT:



- To assign the degree of importance, the model needs all the encoder hidden states, providing a better context vector.
- As the hidden state of the decoder changes at each timestamp, the context vector changes as well.
- For a particular word in the decoder, information from more than one encoder's state in the context vector helps in prediction.
- Hence the weighted sum of all the areas is taken and passed onto the decoder.
- So, the decoder gets two inputs from the RNNs – one is the output of the previous layer and the second is the output coming from the attention model, i.e., the context vector.
- This process is repeated, and a new context vector is provided at each timestep till receival of the <end> token.

Thus, the core principle of attention mechanism is spatial understanding that helps focus on the essential and ignore other areas.

Now that you know what is attention-mechanism, let's move to the next segment to better understand the workings of this mechanism.

Mathematics Behind the Attention Model

Mathematical Calculation in the Attending Model:

In the traditional NMT model, which uses the normal encoder-decoder, there is only one context/feature vector which would be passed on in the hidden state while decoding during translation. However, with an attention model, you get a varying vector based on the previously generated word at each time step. This helps you to look at different parts of the input sequence for better translation.

- Let **$(h1,h2, \dots, hn)$** represent the encoder's hidden state which captures the entire input sequence.
- These encoder states are then passed to the attention model which provides us with the context vector C_i **that summarises the context of the entire input sequence at every ith timestep.**

To calculate the importance of any encoder hidden state hj, you **calculate a score between h_j and the decoder's hidden state generated at previous timestep h_{i-1}**.

- This score, represented by e_{ij} defines the importance of the encoder's state for the given decoder's state. This is represented by the equation $e_{ij}=a(h_{i-1},h_j)$.
- To an attention model, which is a kind of neural network, you pass in h_{i-1}(decoder's previous hidden state) and h_j (encoder's hidden state). Therefore, with this information, you can get **how important is the j_{th}state of the encoder at the i_{th}time stamp.**
- The scoring function can be imagined as a dot product which is a similarity function. If the decoder state h_{i-1} has a high similarity with h_j(encoder state), then the score e_{ij} will be higher and vice versa.

This score is just a representative value between h_{i-1} and h_j, but ideally, we want a distribution of score between h_{i-1} and all the encoder states (h1,h2, . . . , hn). Also, this distribution of score should add up to 1, i.e., it should have a probability distribution. This can be done by normalising the result using a softmax function.

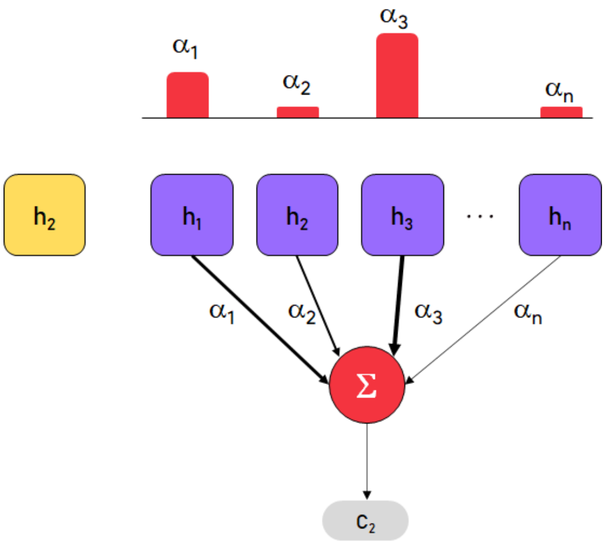
α_{ij}=softmax(e_{ij})

The α_{ij} depicts our attention weights (the probability distribution) of all the encoder states. Once you have this value, you just need to take a weighted sum of the attention weights for each encoder state. This will represent the context vector at ith timestep.

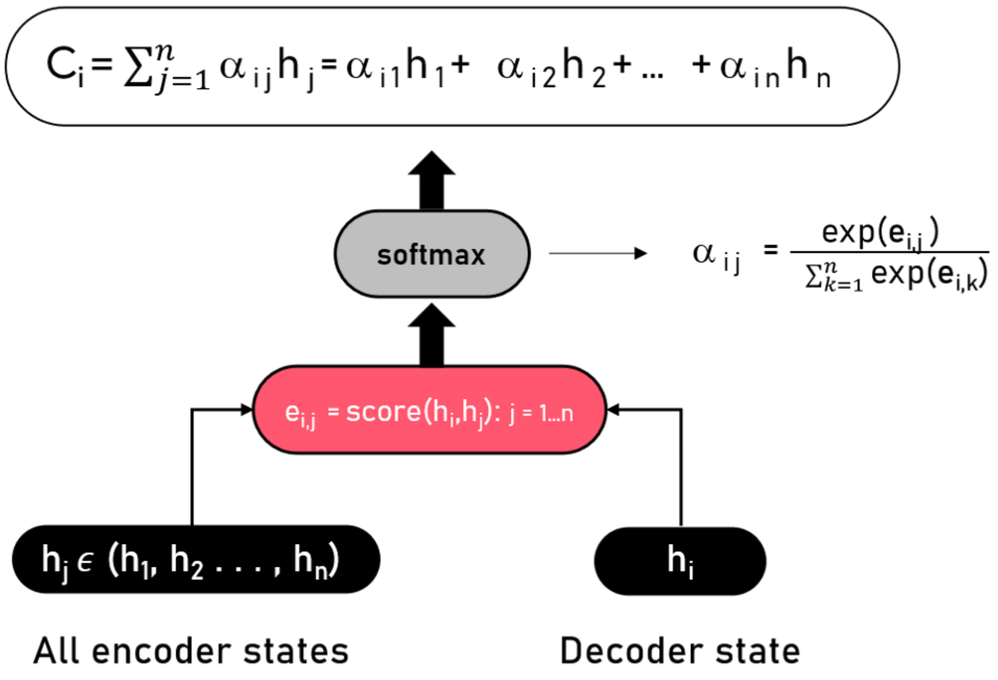
This weighted sum produces the context vector $Ci=\sum_{nj=1}\alpha_{ij}h_j$

- The αjt depicts attention weights (the probability distribution) of all the encoder states.
- Once this value is known, a weighted sum of the attention weights for each encoder state is taken which represents the context vector at ith timestep.
- This weighted sum produces the context vector Ci=∑αijhj.

In the end, the context vector (Ci) and the decoder's hidden state hi-1 is concatenated. This concatenated information is then passed onto the RNN along with the previous prediction.



This vector captures the relevant contextual information from the input sentence for the i_{th} step of the decoder. It uses a weighted sum of all the hidden states instead of just the last one, thus providing better interpretability.


$C_i = \sum_{j=1}^{n} \alpha_{ij} h_j = \alpha_{i1} h_1 + \alpha_{i2} h_2 + \ldots + \alpha_{in} h_n$
softmax
$\alpha_{ij} = \frac{\exp(e_{i,j})}{\sum_{k=1}^{n} \exp(e_{i,k})}$
$e_{i,j} = \text{score}(h_i, h_j)$: $j = 1 \ldots n$
$h_j \in (h_1, h_2 \ldots, h_n)$
h_i
All encoder states
Decoder state

Thus, the attention is a technique used to compute which value (encoder state) to focus on and how much for any given query (decoder state).

In the end, we concatenate the context vector (C_i) and the decoder's hidden state **h_{i-1}**. This concatenated information is then passed onto the RNN along with the previous prediction.

To summarise, given a set of vector values and a vector query, the attention function **computes a weighted sum of the values dependent on the query**. This weighted sum (context vector) is a **selective summary of the information represented by values, which the query attends to.**

Now let's look at the attention functions you can use to calculate the score eij between the decoder hidden state and the encoder states.

Scoring function:

- hiT hj: Dot product attention
- hiT W hj: Multiplicative attention, where $\boldsymbol{W} \in \mathbb{R}^{d_2 \times d_1}$ is a weight matrix
- vTtanh(W1hi + W2hj): Additive attention, where $\boldsymbol{W}_1 \in \mathbb{R}^{d_3 \times d_1}, \boldsymbol{W}_2 \in \mathbb{R}^{d_3 \times d_2}$ are weight matrices and $\boldsymbol{v} \in \mathbb{R}^{d_3}$ is a weight vector

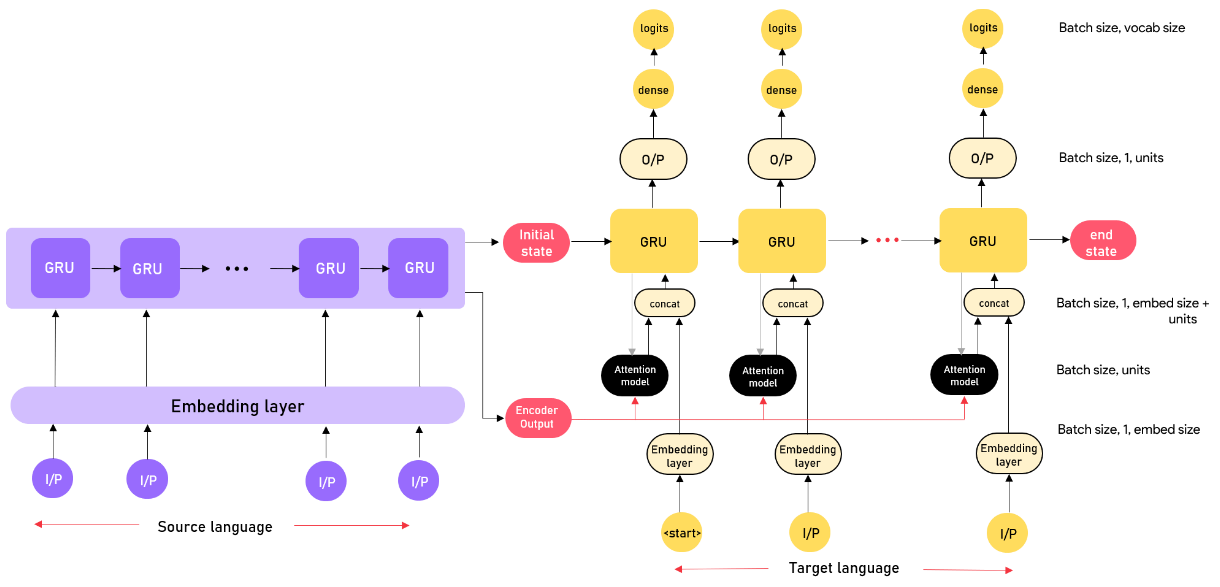
The most popular among these functions is the **additive attention (Bahdanau et al 2015**) represented by VT∗tanh(W1∗hi+W2∗hj). Here the weight matrices W1 and W2 are the dense layers which are operated on decoder and encoder states, respectively.

Once the dense layers are structured, a non-linear mapping is added on top of it using the tanh activation function. The ejt resulting from the non-linear mapping is introduced with VTatt to convert ejt into a scalar value.

Now let's see how to include this attention mechanism in the encoder-decoder architecture to create a better model.

Model Training and Prediction

Here's a pictorial representation of the attention-based encoder-decoder training architecture used for NMT:

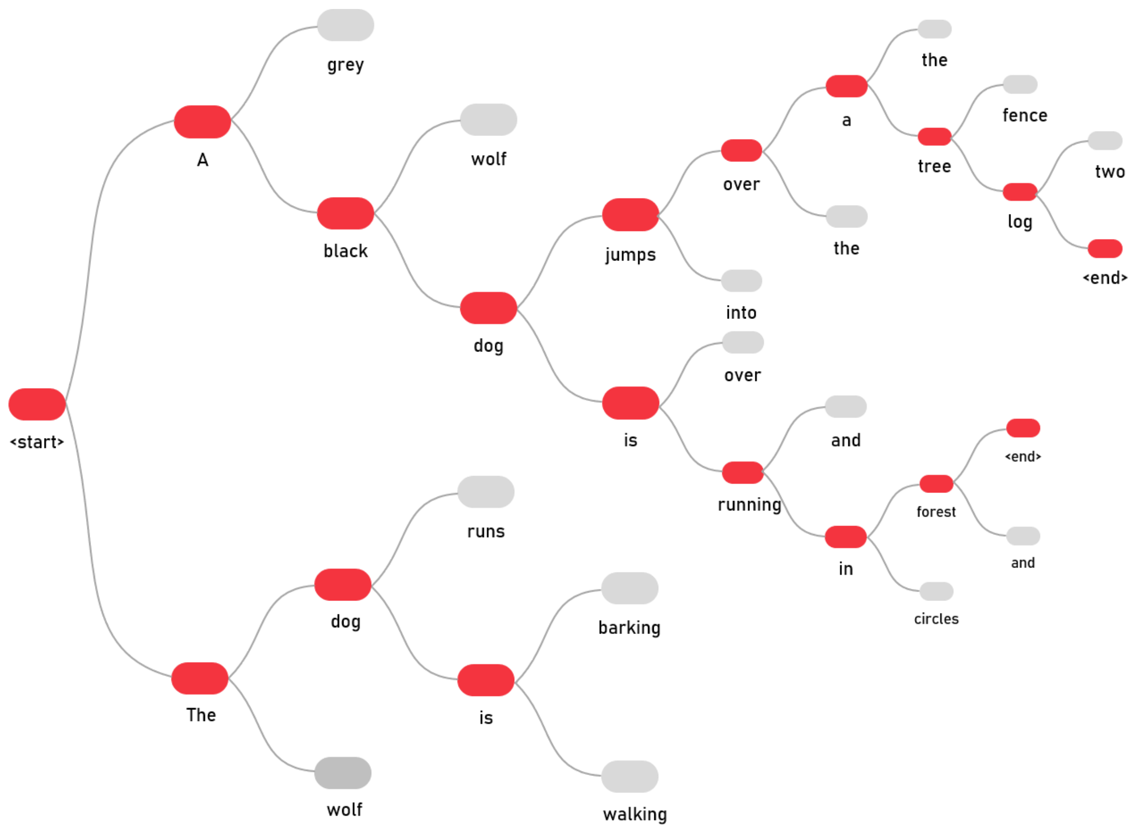


The following models are used in the model to predict words by finding out the probabilities of different words in the vocabulary:

- **Greedy search:** This method calculates the probability of the words according to their occurrence in the vocabulary. It takes the sample of the words, finds the probability of each of the words and then outputs the word with the highest probability. This makes the computational speed of the model fast, but the accuracy might not be up to the mark.
- **Beam search:** The alternative for greedy search is beam search. In beam search, the model finds the k most likely words instead of selecting a single word and these k words are passed onto the next timestep. It works on the breadth-first search algorithm.

Out of the two methods, the beam search algorithm provides us a better prediction during model inference as it searches for k parallel hypotheses based on conditional probability. At the end, the sentence which has the highest probability is selected.

Here's a representation of a Beam search for width 2:



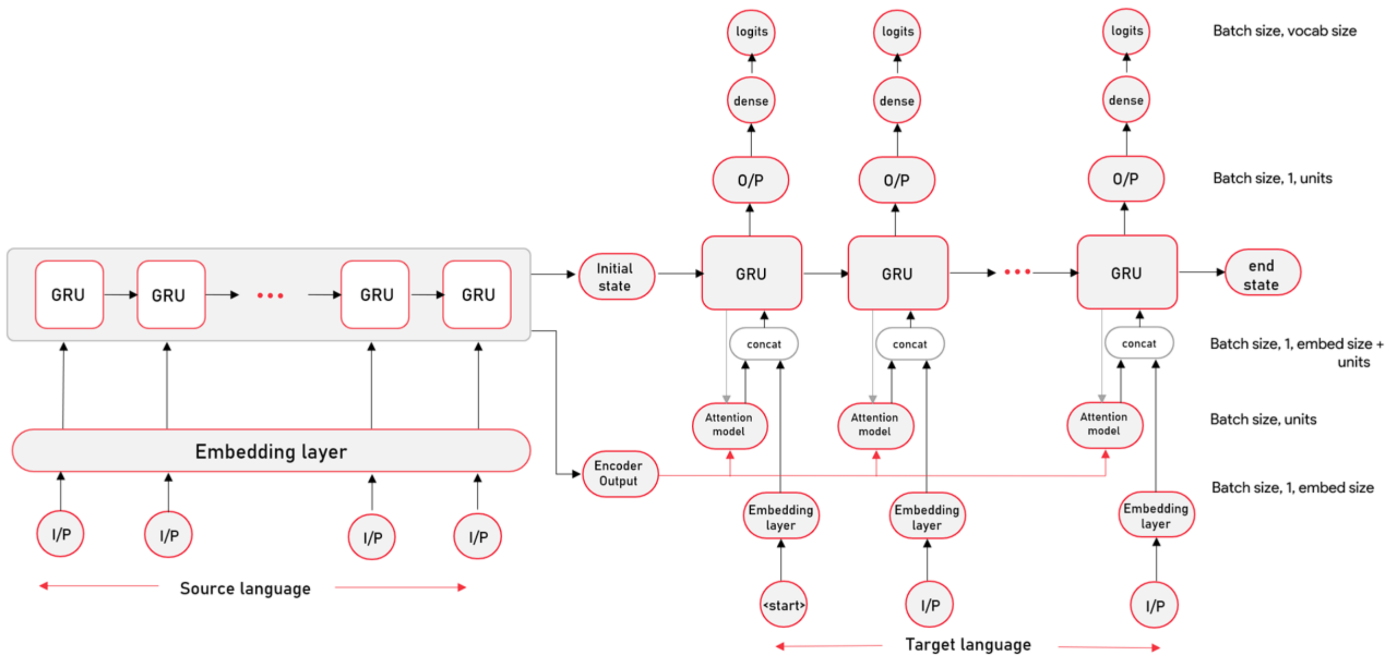
Training Process in Attention-Based NMT Model:

As seen in the traditional model, **teacher forcing** is employed in Attention-Based NMT model also to regulate the model training and converge the process faster.

Let's move to the last segment of the session to know how to implement the Attention-Based NMT mode in TensorFlow.

Implementing Attention-Based NMT Using TensorFlow

Here's how to implement attention-modelling to the model created in the earlier session to get better solutions:

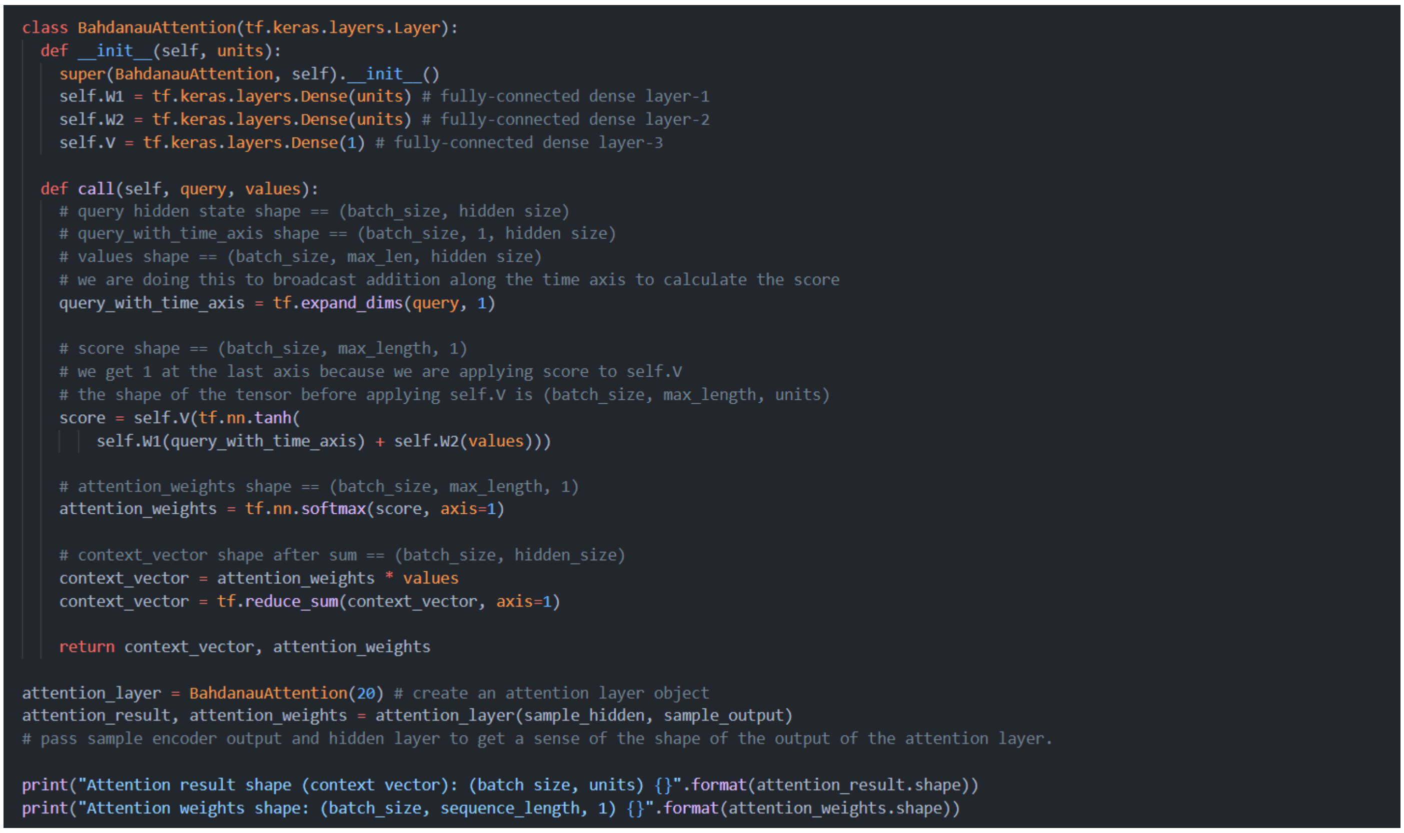


Attention based Encoder-Decoder training architecture for NMT

As you can see, the process for the most part remains the same as earlier since the work is happening on the same data set. Let's now see the entire implementation to understand the changes that are introduced to make the model prediction better and solve the information bottleneck problem.

To the decoder, the **encoder_output** is added as an added input to generate the context vector. The encoder_output and decoder's hidden state is passed as input to the **Bahdanau's attention model** (Additive attention).

Here is the code for the attention model:



```
class BahdanauAttention(tf.keras.layers.Layer):
  def __init__(self, units):
    super(BahdanauAttention, self).__init__()
    self.W1 = tf.keras.layers.Dense(units) # fully-connected dense layer-1
    self.W2 = tf.keras.layers.Dense(units) # fully-connected dense layer-2
    self.V = tf.keras.layers.Dense(1) # fully-connected dense layer-3

  def call(self, query, values):
    # query hidden state shape == (batch_size, hidden size)
    # query_with_time_axis shape == (batch_size, 1, hidden size)
    # values shape == (batch_size, max_len, hidden size)
    # we are doing this to broadcast addition along the time axis to calculate the score
    query_with_time_axis = tf.expand_dims(query, 1)

    # score shape == (batch_size, max_length, 1)
    # we get 1 at the last axis because we are applying score to self.V
    # the shape of the tensor before applying self.V is (batch_size, max_length, units)
    score = self.V(tf.nn.tanh(
        self.W1(query_with_time_axis) + self.W2(values)))

    # attention_weights shape == (batch_size, max_length, 1)
    attention_weights = tf.nn.softmax(score, axis=1)

    # context_vector shape after sum == (batch_size, hidden_size)
    context_vector = attention_weights * values
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights

attention_layer = BahdanauAttention(20) # create an attention layer object
attention_result, attention_weights = attention_layer(sample_hidden, sample_output)
# pass sample encoder output and hidden layer to get a sense of the shape of the output of the attention layer.

print("Attention result shape (context vector): (batch size, units) {}".format(attention_result.shape))
print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(attention_weights.shape))
```

Once you have built your attention model successfully you can observe:

Attention result shape (context vector): **(batch size, units) =** (64, 1024)

Attention weights shape: **(batch_size, sequence_length, 1) =** (64, 72, 1)

Once the context vector is generated, it is then concatenated with the output from the embedding layer. This concatenated result is fed to the GRU layer as input. The other components of the Decoder remain the same.

With the addition of beam search the model performance is improved when it is fed with longer sentences.

With this you have reached the end of the session.