Lab6: Encoder, Decoder with RNNs

In this lab you will implement an encoder and a decoder with RNNs (recurrent neural networks). For now, we concentrate only on implementing these components correctly. In A3, you will train the models.

For both the encoder and the decoder, implement forward_step() (one step in the sequence) and forward() (the complete forward pass).

RNNs used for encoding are typically bidirectional, which allows them to use context not only from previous time steps, but also subsequent time steps.

Decoder training can be improved by teacher forcing, where the decoder is given the correct output of the previous time step, rather than its own (possibly wrong) output of the previous step. Note that teacher forcing is only used for some predefined percentage of the training samples, and that too much teacher forcing results in poor performance of the model on unseen data.

Before you begin

1. Create a new python project and a virtual environment for it in the usual way.
2. Add *Encoder.py*, *Decoder.py* and the test files to your project.

Encoder-Decoder for translation: Basic Model Architecture

Translation is a sequence-to-sequence task - we translate one sequence of inputs (words) into a sequence of outputs (words in a different language).

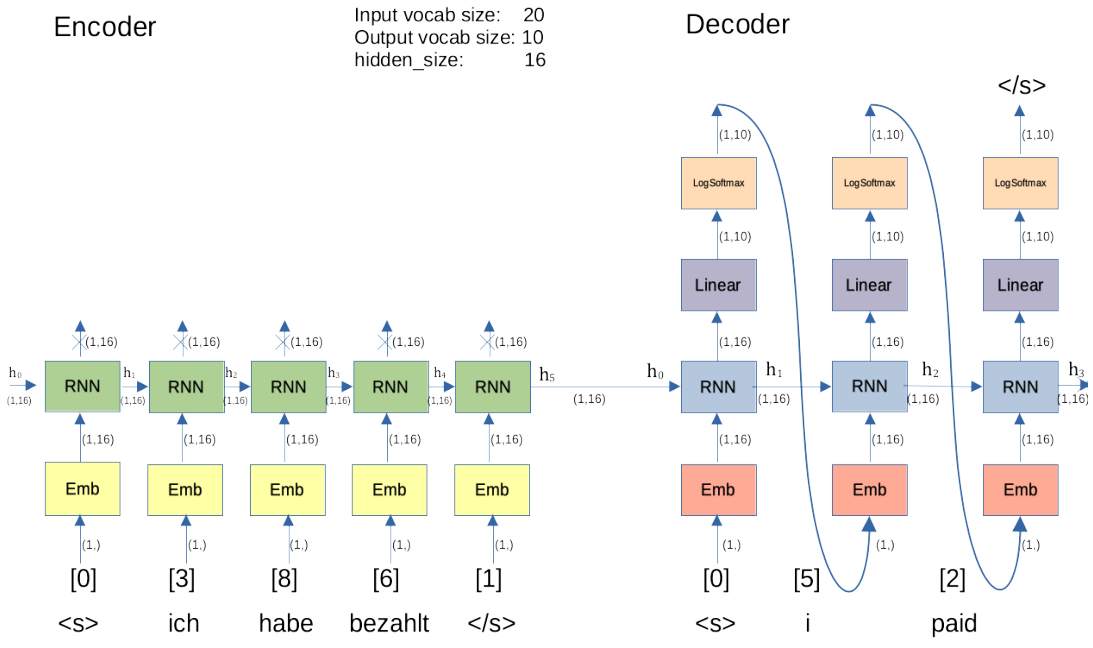
We use the model architecture visualized in the image below, which shows the inputs and outputs of each component at each time step, including shapes. Keep in mind that there are only **two** RNNs, one in the Encoder, and one in the Decoder. The inputs and hidden tensors of the encoder/decoder are fed into the same RNN at successive time steps.

At each timestep of the encoder, the next word and the previous hidden state are inputs to the encoder.

At each timestep of the decoder, the predicted word of the previous timestep and the hidden vector of the previous timestep are inputs to the decoder.

The final hidden vector of the encoder is used as the initial hidden vector of the decoder.

Note that we are not using pretrained embeddings in the embedding layers, although they could be used.



Task1: Encoder - one direction

Implement the forward_step() and forward() methods in the Encoder class. forward_step() is one time step in the forward pass and is a helper function for forward(). forward() is the full forward pass of one input sentence.

Ignore the bidirectional flag for the time being. You will implement it later.

Tests in test_encoder_1dir should pass when you are done with this task.

Task2: Decoder - without teacher forcing

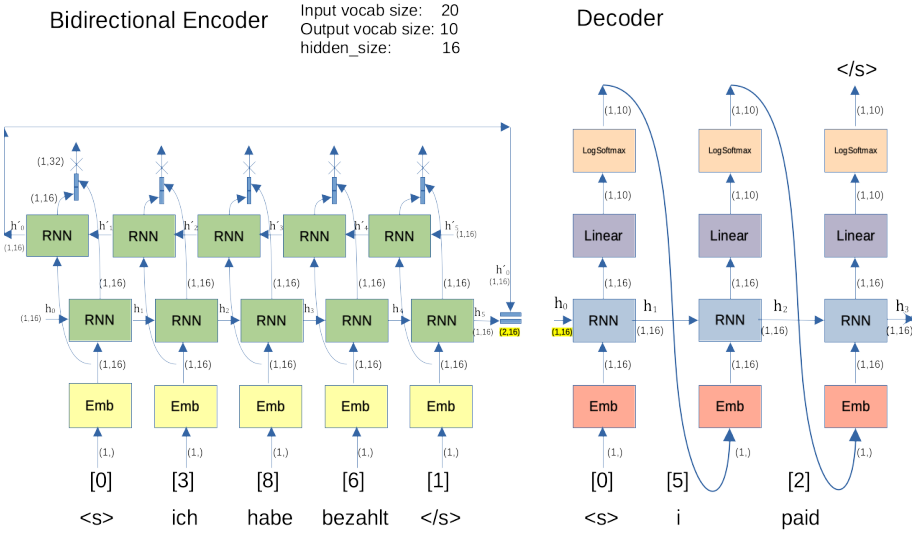
Implement the forward_step() and forward() methods in the Decoder class. forward_step() is one time step in the forward pass and is a helper function for forward(). forward() is the full forward pass.

Ignore the teacher forcing for the time being. You will implement it later.

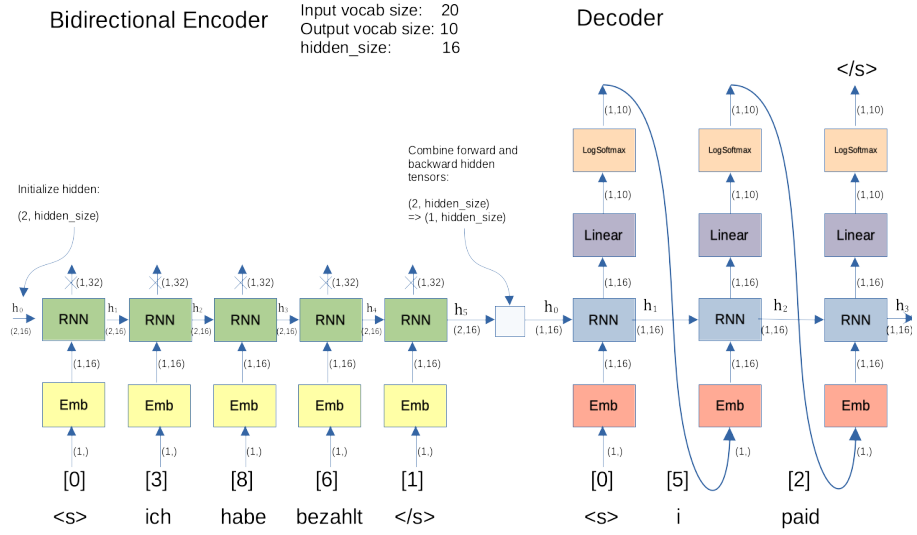
Tests in test_decoder_no_tf should pass when you are done with this task.

Task3: BiDirectional Encoder

A bidirectional encoder processes the input (sentence) both forwards and backwards:



In practice, most of the details are implemented internally in the RNN/GRU. We just need to initialize the hidden tensor correctly, and combine the forward and backward hidden states at the end.

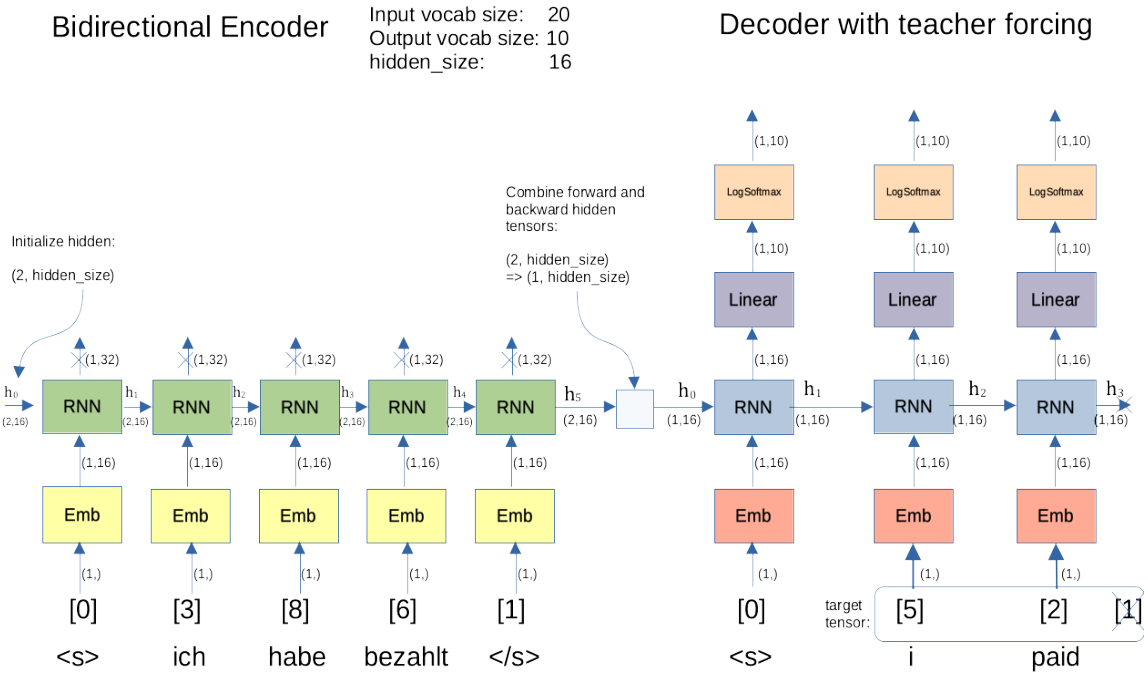


Make the necessary changes in the encoder to allow use of a bidirectional RNN. Tests in test_encoder_2dir should pass when you are done with this task.

Task4: Decoder with Teacher Forcing

When teacher forcing is used, the decoder is fed the correct output of the previous step, rather than its own prediction at the previous step.

Teacher forcing is used for some portion of sentences in the training data, and the other sentences are trained in the usual way. The ratio is set by the *teacher_forcing_ratio* passed to the Decoder constructor.



Implement teacher forcing in the decoder as instructed in the code. Tests in test_decoder_with_tf should pass when you've finished this task.