Alexandria University
Faculty of Engineering
Computer and Systems Department

CSE: NLP
Assigned: Sa, Nov 29, 2025
Due: We, Dec 10, 2025

Lab # 3

Sequence-to-Sequence Models

Objective

In this assignment, you will implement two sequence-to-sequence (seq2seq) models based on recurrent networks and the Transformer architecture to build a neural machine translation (NMT) systems translating from French to English.

Dataset

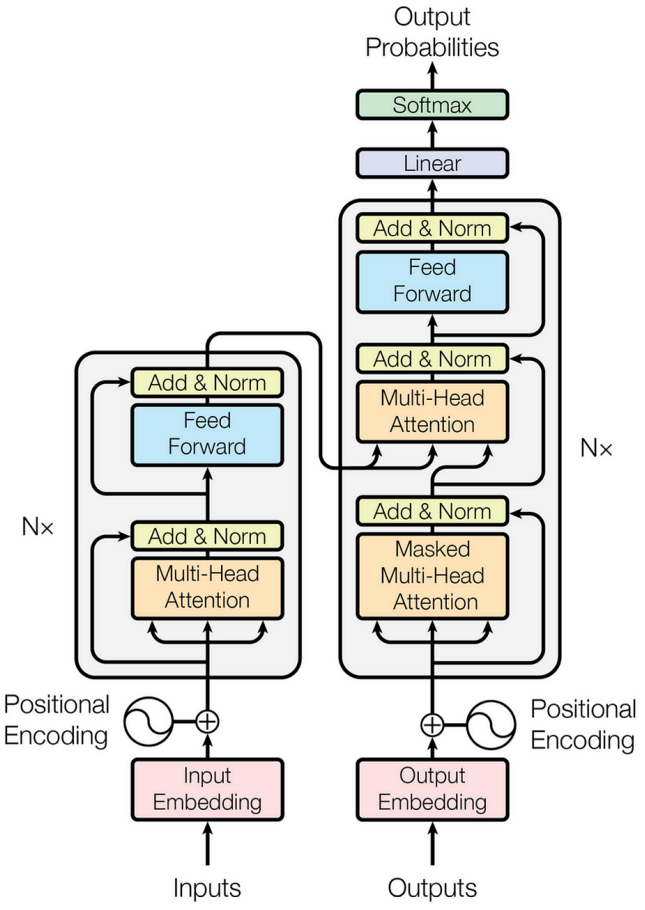
You will use a dataset consisting of parallel French and English sentences and partitioned into training, validation, and test splits. You will need to tokenize the data using French and English tokenizers, which are provided with the assignment resources. You will use BPE (byte pair encoding) tokenization, which can split a less common word into multiple subword tokens. The dataset and tokenizers are available at this link.

Part 1. Transformer-Based Architecture

The model you will be implementing is an encoder-decoder Transformer model. For simplicity, we made several modifications to the architecture described in this paper:

- Use learned positional embeddings instead of sinusoidal positional embeddings.
- No dropout in the embedding layer and the attention weights.
- Use weight tying between the decoder's input embeddings and the output projection matrix.

We will describe each block of the model. Let $\mathbf{d}$ be the embedding dimension of input embeddings and hidden states, $\mathbf{N}$ the maximum sequence length, $\mathbf{V^{(e)}}$ and $\mathbf{V^{(d)}}$ the vocab sizes for encoder and decoder vocabulary, respectively. Let $\mathbf{n}$ be the length of a particular input sequence. In practice, the model will process $\mathbf{B}$ sequences in a batch in parallel, and the sequence might contain **pad tokens**, which should be excluded from the attention mechanism and the loss function.



- **Embedding:** Let $\mathbf{E}^{(e)} \in \mathbb{R}^{V^{(e)} \times d}$ be the token embedding tables and $\mathbf{P}^{(e)} \in \mathbb{R}^{N \times d}$ be the positional embedding tables for the encoder. To embed a token with token id t at position i, we look up the token embedding at index t and position embedding at index i and add the two embeddings. The same procedure is done in the decoder embedding layer with separate matrices $\mathbf{E}^{(d)} \in \mathbb{R}^{V^{(d)} \times d}$ and $\mathbf{P}^{(d)} \in \mathbb{R}^{N \times d}$. The output of each embedding layer is a sequence of vectors $\mathbf{h}_1^{(0)}, \mathbf{h}_2^{(0)}, \ldots, \mathbf{h}_n^{(0)} \in \mathbb{R}^d$.

- **Multi-head attention:** The input to this layer is a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \ldots, \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. We project each of these vectors to query, key, and value vectors:

$$\mathbf{q}_i^{(l)} = \mathbf{W}^{(q)} \mathbf{h}_i^{(l-1)}, \quad \mathbf{k}_i^{(l)} = \mathbf{W}^{(k)} \mathbf{h}_i^{(l-1)}, \quad \mathbf{v}_i^{(l)} = \mathbf{W}^{(v)} \mathbf{h}_i^{(l-1)},$$

  where $\mathbf{q}_i^{(l)}, \mathbf{k}_i^{(l)}, \mathbf{v}_i^{(l)} \in \mathbb{R}^d$. For the sake of clarity, we are omitting the layer index (l) from the weight matrices and intermediate variables. In cross-attention, the input key and value matrices would be the output states of the encoder, whereas the queries would be computed from the hidden states of the decoder.

  We then split these vectors into H separate vectors, where H is the number of attention heads:

$$\mathbf{q}_i = \begin{bmatrix} \mathbf{q}_{i,1} \\ \mathbf{q}_{i,2} \\ \vdots \\ \mathbf{q}_{i,H} \end{bmatrix}, \quad \mathbf{k}_i = \begin{bmatrix} \mathbf{k}_{i,1} \\ \mathbf{k}_{i,2} \\ \vdots \\ \mathbf{k}_{i,H} \end{bmatrix}, \quad \mathbf{v}_i = \begin{bmatrix} \mathbf{v}_{i,1} \\ \mathbf{v}_{i,2} \\ \vdots \\ \mathbf{v}_{i,H} \end{bmatrix},$$

  where $\mathbf{q}_{i,h}, \mathbf{k}_{i,h}, \mathbf{v}_{i,h} \in \mathbb{R}^{d_H}$ and $d_H = \frac{d}{H}$ is the head dimension.

  For each head h, we compute a scaled dot product attention:

$$\mathbf{y}_{i,h} = \sum_{j=1}^{n} \left(\frac{\exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j,h} / \sqrt{d_H})}{\sum_{j'=1}^{n} \exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j',h} / \sqrt{d_H})} \right) \mathbf{v}_{j,h},$$

  where each $\mathbf{y}_{i,h} \in \mathbb{R}^{d_H}$.

  We need to make sure to avoid attending to pad tokens, which should not affect the model's output. In practice, this is achieved by setting the attention score $(\mathbf{q}_i^{(h)} \cdot \mathbf{k}_j^{(h)})/\sqrt{d_H}$ to a very large negative value if there is a pad token at position j. When computing the self-attention in the decoder, we use causal masking to avoid attending to future values:

$$\mathbf{y}_{i,h} = \sum_{j=1}^{i} \left(\frac{\exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j,h} / \sqrt{d_H})}{\sum_{j'=1}^{i} \exp(\mathbf{q}_{i,h} \cdot \mathbf{k}_{j',h} / \sqrt{d_H})} \right) \mathbf{v}_{j,h}.$$

  Finally, we stack the attention outputs and project each token to obtain a sequence of output vectors $\mathbf{h}_1^{(l)}, \mathbf{h}_2^{(l)}, \ldots, \mathbf{h}_n^{(l)} \in \mathbb{R}^d$:

$$\mathbf{h}_i^{(l)} = \mathbf{W}^{(o)} \begin{bmatrix} \mathbf{y}_{i,1} \\ \mathbf{y}_{i,2} \\ \vdots \\ \mathbf{y}_{i,H} \end{bmatrix}.$$

- **Feedforward layers:** The input to this layer is a sequence of hidden state vectors from the previous layer $\mathbf{h}_1^{(l-1)}, \mathbf{h}_2^{(l-1)}, \ldots, \mathbf{h}_n^{(l-1)} \in \mathbb{R}^d$. Each feedforward layer learns two feedforward matrices: $\mathbf{W}^{(1)} \in \mathbb{R}^{d_I \times d}$ and $\mathbf{W}^{(2)} \in \mathbb{R}^{d \times d_I}$ (we are omitting the layer index again). These matrices are used to project each input vector to a larger intermediate dimension d_I, apply a ReLU activation, and then project back to the original embedding space. Finally, dropout is applied.

$$\mathbf{h}_i^{(l)} = \text{Dropout}\Big(\mathbf{W}^{(2)}\,\text{ReLU}\Big(\mathbf{W}^{(1)}\mathbf{h}_i^{(l-1)}\Big)\Big)$$

- **Add & Norm:** After each attention and feedforward block, we add a residual network connection and perform layer normalization. This helps with optimization issues of training deep Transformer networks.

- **Output layer:** We will re-use the token input embeddings and perform next token prediction at each position i in the decoder:

$$logits_i = \mathbf{E}^{(d)}\mathbf{h}_i^{(l-1)} \in \mathbb{R}^{\mathbb{V}()}$$

- **Model Parameters:** Here are the model's parameters that are required to be implemented. You need to test different ones after that, and you will be required to justify your choices:

Parameter	Value
hidden_size	32
intermediate_size	32×4
num_attention_heads	4
num_encoder_layers	3
num_decoder_layers	3
max_sequence_length	32
hidden_dropout_prob	0.1
batch_size	32
learning_rate	1×10^{-3}
max_epochs	10

Part 2. Recurrent Network-Based Architecture

In the second part of this assignment, you will implement a Sequence-to-Sequence network using Recurrent Neural Networks (RNNs). Specifically, you will use a **Bidirectional LSTM** for the encoder and a **Unidirectional LSTM** for the decoder, coupled with an Additive Attention mechanism.

Let N be the maximum sequence length, d be the embedding dimension, and h be the hidden state dimension.

- **Encoder (Bi-LSTM):** The encoder takes the input sequence of token embeddings $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where $\mathbf{x}_i \in \mathbb{R}^d$. We utilize a bidirectional LSTM, which consists of two LSTMs processing the sequence in opposite directions.

  The forward LSTM computes hidden states $\overrightarrow{\mathbf{h}}_1, \ldots, \overrightarrow{\mathbf{h}}_n$ and the backward LSTM computes $\overleftarrow{\mathbf{h}}_1, \ldots, \overleftarrow{\mathbf{h}}_n$. We concatenate the states at each time step to form the final encoder hidden states:

$$\mathbf{h}_i = \Big[\overrightarrow{\mathbf{h}}_i; \overleftarrow{\mathbf{h}}_i\Big] \in \mathbb{R}^{2h}$$

- **Decoder Initialization:** Unlike the Transformer, the LSTM decoder requires an initial hidden state $\mathbf{s}_0$. To bridge the encoder and decoder, we project the final hidden and cell states of the encoder (from both directions) using a linear layer $\mathbf{W}^{(init)}$ to match the decoder's hidden dimension h:

$$\mathbf{s}_0 = \tanh(\mathbf{W}^{(init)}[\overrightarrow{\mathbf{h}}_n; \overleftarrow{\mathbf{h}}_1]), \quad \mathbf{s}_0 \in \mathbb{R}^h$$

- **Additive Attention:** At decoding time step t, we compute context vectors based on the decoder's previous hidden state $\mathbf{s}_{t-1}$ and all encoder hidden states $\mathbf{h}_1, \ldots, \mathbf{h}_n$.

  First, we calculate the attention alignment scores $e_{t,i}$. Note that unlike the Dot-Product attention in Part 1, Additive attention uses a feed-forward network:

$$e_{t,i} = \mathbf{v}^\top \tanh(\mathbf{W}^{(a)}\mathbf{s}_{t-1} + \mathbf{U}^{(a)}\mathbf{h}_i)$$

  where $\mathbf{W}^{(a)} \in \mathbb{R}^{h \times h}$, $\mathbf{U}^{(a)} \in \mathbb{R}^{h \times 2h}$, and $\mathbf{v} \in \mathbb{R}^h$ are learned parameters.

  We then normalize these scores using the softmax function to obtain attention weights $\alpha_{t,i}$:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{n} \exp(e_{t,j})}$$

  Finally, we compute the context vector $\mathbf{c}_t$ as the weighted sum of encoder hidden states:

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i}\mathbf{h}_i \in \mathbb{R}^{2h}$$

- **Decoder Step:** The input to the decoder LSTM at step t is the concatenation of the embedding of the previous token y_{t-1} (embedded as $\mathbf{y}_{t-1} \in \mathbb{R}^d$) and the context vector $\mathbf{c}_t$.

$$\mathbf{s}_t = \text{LSTM}([\mathbf{y}_{t-1}; \mathbf{c}_t], \mathbf{s}_{t-1})$$

- **Output Layer:** To predict the next token, we concatenate the new hidden state $\mathbf{s}_t$, the context vector $\mathbf{c}_t$, and the input embedding $\mathbf{y}_{t-1}$, and pass them through a linear projection to the vocabulary size:

$$logits_t = \mathbf{W}^{(out)}[\mathbf{s}_t; \mathbf{c}_t; \mathbf{y}_{t-1}]$$

- **Model Parameters:** For the LSTM implementation, use the following hyperparameters.

Parameter	Value
embed_size (d)	256
hidden_size (h)	512
encoder_bidirectional	True
num_layers	1 (or 2)
lstm_dropout	0.3
batch_size	32
learning_rate	1×10^{-3}
max_epochs	10

Notes

1. You can use the `transformers` package only for the tokenization process.
2. Implement the vectorized multi-head attention module. You are not allowed to use `nn.MultiheadAttention` or `nn.functional.scaled_dot_product_attention`.
3. Implement the embedding layer, which computes and sums the token embeddings and the positional embeddings.
4. For feedforward and normalization layers, you can use them from `torch.nn` directly.
5. For the transformers part, make your code modular, then combine different sub-modules into a single class.
6. You need to implement the **beam search** decoding algorithm and compute results of both models using **the BLEU score**.
7. You are required to visualize some of the attention weights of both models.
8. You need to implement saving and loading checkpoints.
9. Be prepared to test the model live during the discussion.