Lab 3: The One Where You Learn to Think

Introduction

In the previous laboratory, you have used self-attention to implement an encoder-only model and masked-attention to implement a decoder-only model. In this laboratory, you will imlpement cross-attention to implement a full encoder-decoder transformer architecture.

This laboratory practice is designed to give you hands-on experience with the Transformer architecture. You will build the model using the code from previous laboratories, implement cross-attention, and explore how different decoding strategies affect how the model generate text.

Throughout this practice, you will:

- Implement cross-attention, ensuring that the decoder part of the model attends to the output of the encoder part.
- Implement different decoding strategies in the transformer model.
- Explore how the different decoding strategies affect how the model generates text, evaluating their efficiency in two different datasets.

By the end of this lab, you will understand how the encoder and decoder parts of a transformer model interacts, and how the different strategies can be used to improve the model efficiency in NLP tasks.

Learning Outcomes

Upon completion of this laboratory exercise, you will be able to:

- Implement cross-attention and understand how the encoder output informs the decoder layers.
- Build and stack Encoder and Decoder layers to create the Transformer model.
- Understand how the different decoder strategies affect the model text generation process.

Deliverables

1. `encoder.py` - This file must implement all the modules related to the Encoder architecture, with some changes respect to the first laboratory to mask padding tokens.
2. `decoder.py` - This file must implement all the modules related to the Decoder architecture, with some changes respect to the previous laboratory to mask padding tokens.
3. `utils.py` - This file must implement the common functions for the encoder and the decoder, heritage from previous laboratories.
4. `transformer.py` - This file must implement the `transformer` model including the different decoding strategies, using the modules from `decoder.py` and `encoder.py`.

5. `decoding_strategies.ipynb` - This file must use a huggingface encoder-decoder model to generate text in two different use cases. Play with it!

These files should be uploaded to Github maintaining the format handled in the laboratory practice.

Exercises

Exercise 1: Reload Your Weapons (II): Back to the Code!

In this exercise, you will retrieve the modules you have made in the previous two laboratories. In order to guide you, we have divided this exercise in three sub-exercises depending on the target file.

Exercise 1.1: `utils.py`

In this file you have to include all the common parts of the `decoder` and `encoder` models:

- `Feedforward` module
- `Embeddings` module
- `AttentionHead` and `MultiHeadAttention` modules.

The code for these modules can be copy-pasted from previous laboratories. However, some changes in the `AttentionHead` and `MultiHeadAttention` modules must be implemented in order for them to be able to receive the queries and the keys and values from two different modules. Check the `forward()` method definitions and docstrings if you have any doubts.

Exercise 1.2: `encoder.py`

In this file you have to include all the modules of the `encoder` part of the transformer:

- `TransformerEncoderLayer`
- `TransformerEncoder`

Again, you will have to implement some changes. In this case, one thing that we did not do in the first laboratory is to mask the padding input tokens. Now, a `mask` argument has been included in the `forward()` method to indicate which of the input tokens are padding and should be masked.

Hint #1:

Be careful when implementing the mask, ensuring that the code does not fail if `mask` is `None`.

Exercise 1.3: `decoder.py`

In this file you have to include all the modules of the `decoder` part of the transformer:

- `TransformerDecoderLayer`
- `TransformerDecoder`

These modules are the ones where you have to make the most changes. In this case, a new argument has appeared in the forward method: `enc_output` . This is the output of the encoder part of the transformer, and shall be connected to all the decoder blocks as depicted in figure 1.

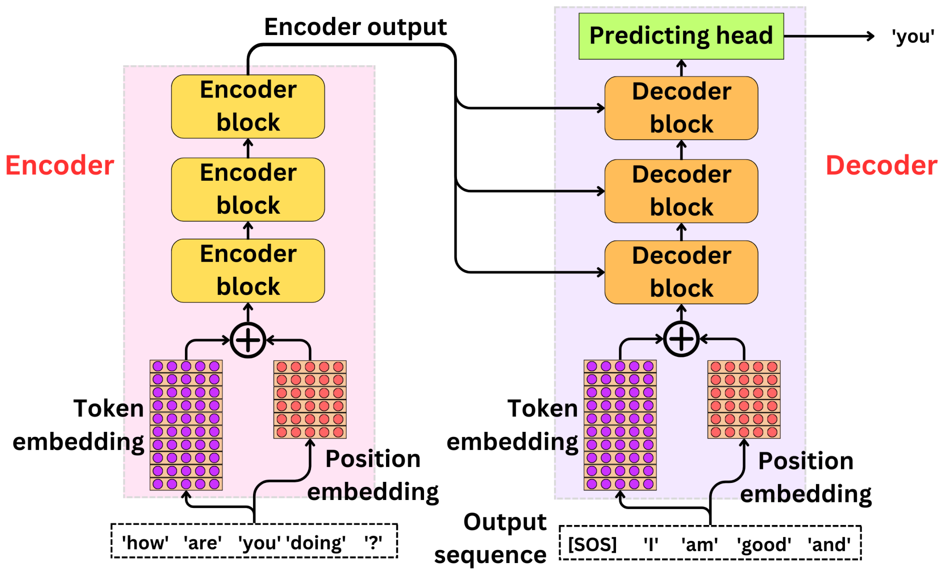

Figure 1: Cross-attention connection between the encoder and decoder part of the transformer.

The encoder output shall be entered into each decoder block tweeking the `MultiHeadAttention` block, where the `queries` comes from the previous decoder layers (or decoder embeddings) and the `keys` and `values` comes from the encoder output, as depicted in figure 2.

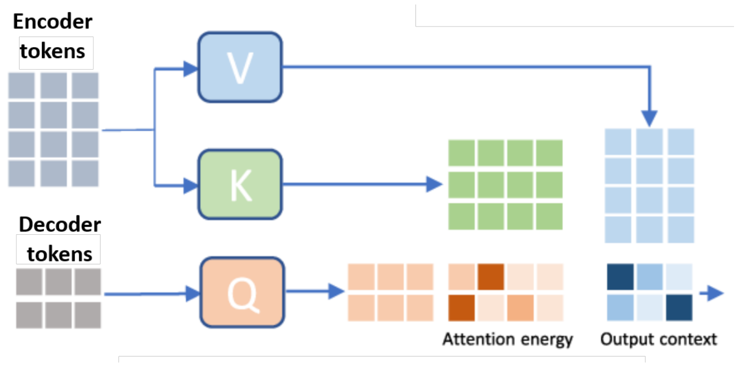

Figure 2: Cross-attention diagram

Exercise 2: Transformers, No Autobots Needed

In this exercise, you will implement a full encoder-decoder `transformer` architecture using the modules from the previous sections. To do so, implement an encoder, decoder, and a lineal layer that will be in charge of calculating that will act as a language modelling head .

After you implement the `__init__()` and `forward()` method of the `transformer` class, you will have to implement the different decoding strategies used in the `generate()` method.

Exercise 2.1: Decode Fast, Stay Greedy

The first decoding strategy to implement is the greedy decoding in the `__greedy_decode()` method. Note that the method is preceded by two '_' signs, because it is a private method that should not be accessed by any object outside the class. Greedy decoding is simple and efficient but might not always generate the most optimal or diverse sequences because it does not explore alternative token possibilities.

At each time step t, the model predicts a probability distribution over the target vocabulary, $P(y_t|y_{1:t-1}, \mathbf{x})$, where $\mathbf{x}$ is the source input, and $y_{1:t-1}$ are the previously generated tokens. The greedy decoding approach selects the token with the highest probability from this distribution:

$$y_t = \arg\max_{y \in V} P(y|y_{1:t-1}, \mathbf{x})$$

Where V is the vocabulary, and $P(y|y_{1:t-1}, \mathbf{x})$ is the probability distribution over the vocabulary produced by the decoder at step t. This process repeats for each time step until either the maximum sequence length is reached or the end-of-sequence token (EOS) is generated.

A basic implementation of this decoding strategy is:

1. Initialize the target sequence with the start-of-sequence token (SOS).
2. For each time step t:
   (a) Pass the source input $\mathbf{x}$ and the current target sequence $y_{1:t-1}$ through the decoder.
   (b) Compute the probability distribution over the vocabulary for the next token.
   (c) Select the token with the highest probability.
   (d) Append this token to the target sequence.
3. Repeat until the end-of-sequence token (EOS) is generated or the maximum length is reached.

Exercise 2.2: Beam Me Up Scotty!

In this exercise, you will implement the beam search strategy in the `__beam_search_decode()` method. Beam search is a more advanced decoding strategy than greedy decoding. Instead of selecting the single most probable token at each time step, beam search keeps track of the top B most likely sequences, where B is called the *beam size*. At each time step, all possible extensions of the current sequences are considered, and only the top B sequences (with the highest cumulative probabilities) are retained.

At each time step t, the model predicts a probability distribution over the target vocabulary, $P(y_t|y_{1:t-1}, \mathbf{x})$, where $\mathbf{x}$ is the source input, and $y_{1:t-1}$ are the previously generated tokens. Unlike greedy decoding, which selects only one sequence, beam search keeps multiple hypotheses. The goal is to keep the top B sequences with the highest cumulative log probability at each time step. This process continues until the beam only contains sequences that have generated the end-of-sequence token (EOS) or until a predefined maximum sequence length is reached.

A basic implementation of this decoding strategy is:

1. Initialize the beam with the start-of-sequence token (SOS), and set the initial score to 0.
2. For each time step t:

   (a) For each sequence in the beam, pass the source input $\mathbf{x}$ and the current sequence $y_{1:t-1}$ through the decoder.
   (b) Compute the probability distribution over the vocabulary for the next token for each sequence.
   (c) For each sequence, create new sequences by adding each token of the vocabulary.
   (d) Score each candidate sequence by summing the log probabilities of the tokens in the sequence.
   (e) Keep only the top B candidate sequences with the highest cumulative log probabilities.

3. Repeat until all sequences in the beam have generated the end-of-sequence token (EOS) or the maximum length is reached.

4. Return the sequence with the highest score.

Exercise 2.3: Is It Hot in Here or Is It Me?

In this exercise, you will implement the multinomial sampling with temperature decoding strategy in the `__sampling_decode()` method. In multinomial sampling with temperature, we introduce a degree of randomness by sampling from the model's probability distribution over the vocabulary. The temperature parameter is used to control how much randomness is added to the sampling process. Lower temperature values result in more deterministic outputs, while higher values increase diversity by making the sampling more random. You have already done this in the previous laboratory, so check your code!

At each time step t, the model predicts a probability distribution $P(y_t|y_{1:t-1}, \mathbf{x})$, where $\mathbf{x}$ is the source input, and $y_{1:t-1}$ are the previously generated tokens. The temperature parameter τ is applied to modify the logits of the probability distribution, making the distribution sharper or flatter, depending on the value of τ. The modified probability distribution is then used to sample the next token:

$$P_\tau(y_t|y_{1:t-1}, \mathbf{x}) = \frac{\exp\left(\frac{\log P(y_t|y_{1:t-1}, \mathbf{x})}{\tau}\right)}{\sum_{y \in V} \exp\left(\frac{\log P(y|y_{1:t-1}, \mathbf{x})}{\tau}\right)}$$

Where $P(y_t|y_{1:t-1}, \mathbf{x})$ is the original probability distribution, τ (temperature) controls the sharpness of the distribution and V is the vocabulary. Based on this formula, we can conclude that:

- $\tau = 1$ means no change to the original distribution.
- $\tau < 1$ makes the distribution sharper, increasing the likelihood of selecting high-probability tokens.
- $\tau > 1$ makes the distribution flatter, increasing the diversity of sampled tokens.

After applying temperature, the next token is sampled from the adjusted probability distribution using multinomial sampling.

A basic implementation of this decoding strategy is:

1. Initialize the target sequence with the start-of-sequence token (SOS).

2. For each time step t:

   (a) Pass the source input $\mathbf{x}$ and the current target sequence $y_{1:t-1}$ through the decoder.
   (b) Compute the logits over the vocabulary for the next token.

   (c) Apply temperature τ to the logits, scaling them by $1/\tau$.
   (d) Normalize the modified logits to produce the adjusted probability distribution using the softmax function.
   (e) Sample the next token from this adjusted probability distribution using multinomial sampling.
   (f) Append the sampled token to the target sequence.

3. Repeat until the end-of-sequence token (EOS) is generated or the maximum sequence length is reached.

Exercise 2.4: Keep the Best, Leave the Rest

In this exercise, you will implement the top-k sampling decoding strategy in the `__top_k_sampling_decode()` method. Top-k sampling is a decoding strategy where, instead of considering the entire probability distribution over the vocabulary, we only focus on the top k tokens with the highest probabilities. This strategy allows us to maintain a balance between diversity and quality, as it filters out low-probability tokens that are unlikely to produce meaningful outputs while introducing some randomness through sampling.

At each time step t, the model predicts a probability distribution $P(y_t|y_{1:t-1}, \mathbf{x})$, where $\mathbf{x}$ is the source input and $y_{1:t-1}$ are the previously generated tokens. In top-k sampling, we first sort the vocabulary by the predicted probabilities and keep only the top k tokens. The rest of the tokens are set to zero probability, and the distribution is normalized. After applying top-k filtering, the next token is sampled from the remaining tokens using multinomial sampling.

A basic implementation of this decoding strategy is:

1. Initialize the target sequence with the start-of-sequence token (SOS).
2. For each time step t:
   (a) Pass the source input $\mathbf{x}$ and the current target sequence $y_{1:t-1}$ through the decoder.
   (b) Compute the logits over the vocabulary for the next token.
   (c) Apply softmax to the logits to get the probability distribution.
   (d) Sort the probabilities and keep only the top k tokens, setting the probabilities of all other tokens to zero.
   (e) Normalize the remaining probabilities.
   (f) Sample the next token from the top-k tokens using multinomial sampling.
   (g) Append the sampled token to the target sequence.
3. Repeat until the end-of-sequence token (EOS) is generated or the maximum sequence length is reached.

Exercise 2.5: Top-p or Not Top-p?

In this exercise, you will implement the top-p sampling decoding strategy in the `__top_p_sampling_decode()` method. Top-p (nucleus) sampling is a decoding strategy where we dynamically select the smallest set of top tokens whose cumulative probability exceeds a certain threshold p. Unlike top-k, where we always consider a fixed number of tokens, top-p adjusts the number of tokens considered based on their probabilities, allowing for more flexibility and dynamic sampling.

At each time step t, the model predicts a probability distribution $P(y_t|y_{1:t-1}, \mathbf{x})$, where $\mathbf{x}$ is the source input, and $y_{1:t-1}$ are the previously generated tokens. In top-p sampling, we first sort the tokens by their probabilities and select the smallest set of tokens whose cumulative probability exceeds a predefined threshold p. All other tokens are set to zero probability. After applying top-p filtering, the next token is sampled from the remaining tokens using multinomial sampling.

A basic implementation of this decoding strategy is:

1. Initialize the target sequence with the start-of-sequence token (SOS).
2. For each time step t:
   (a) Pass the source input $\mathbf{x}$ and the current target sequence $y_{1:t-1}$ through the decoder.
   (b) Compute the logits over the vocabulary for the next token.
   (c) Apply softmax to the logits to get the probability distribution.
   (d) Sort the probabilities in descending order.
   (e) Select the smallest set of top tokens such that their cumulative probability exceeds threshold p.
   (f) Set the probabilities of all other tokens to zero and normalize the remaining probabilities.
   (g) Sample the next token from the top-p tokens using multinomial sampling.
   (h) Append the sampled token to the target sequence.
3. Repeat until the end-of-sequence token (EOS) is generated or the maximum sequence length is reached.

Exercise 2.6: Founding Coherence in Diversity

In this exercise, you will implement the Contrastive Search decoding strategy in the `__contrastive_decode()` method. Contrastive Search decoding is a decoding strategy that aims to balance coherence (high-probability tokens) with diversity (avoiding repetitive or overly similar tokens). This is achieved by scoring candidate tokens based on both their probability and their distinctiveness from previously generated tokens, encouraging more diverse and contextually appropriate generations.

Given the preceding decoder tokens $\mathbf{y}_{<t} = y_1, \ldots, y_{t-1}$, contrastive search selects as next token

$$x_t = \underset{v \in V_k}{\operatorname{argmax}} \left\{ (1-\alpha) \times \underbrace{P_{\mathrm{LM}}(v|\mathbf{y}_{<t}, \mathbf{x})}_{\text{model confidence}} - \alpha \times \underbrace{(\max\{s(\mathbf{h}_v, \mathbf{h}_j) \mid 1 \leq j \leq t-1\})}_{\text{degeneration penalty}} \right\}$$

Where V_k represents the set of the top-k predictions from the language model's probability distribution $P_{\mathrm{LM}}(v \mid x_{<t})$. The first component, *model confidence*, is the probability assigned by the model to the candidate token v. The second component, the *degeneration penalty*, evaluates how distinct v is in relation to the prior decoder context $\mathbf{y}_{<t}$, with the function $s(\cdot, \cdot)$ computing the *cosine similarity* between token representations. Specifically, the degeneration penalty is the maximum cosine similarity between the representation of v, denoted $\mathbf{h}_v$, and the representations of tokens in the context $\mathbf{y}_{<t}$. The candidate's representation $\mathbf{h}_v$ is generated by the model by passing the concatenation of $\mathbf{x}_{<t}$ and v through the decoder. $\mathbf{h}_v$ and $\mathbf{h}_j$ denote here the **output** of the last layer of the decoder for the tokens x and j (i.e., the vectors produced by the decoder right below the language modeling head).

Intuitively, a higher degeneration penalty for v indicates greater similarity to the context in the representation space, increasing the risk of model degeneration. The hyperparameter α controls the balance between these two factors. When $\alpha = 0$, contrastive search reduces to standard greedy search.

1. Initialize the target sequence with the start-of-sequence token (SOS).
2. For each time step t:
   (a) Pass the current target sequence $\mathbf{y}_{<t}$ through the decoder.
   (b) Compute the logits for the next token over the vocabulary.
   (c) Apply the softmax function to the logits to obtain the probability distribution $P(y_t|\mathbf{y}_{1:t-1}, \mathbf{x})$.
   (d) Identify the set V_k of the top k tokens from $P(y_t|\mathbf{y}_{1:t-1}, \mathbf{x})$.
   (e) For each top k token $v \in V_k$, concatenate it with $\mathbf{y}_{1:t-1}$ and feed this concatenation into the decoder. Perform a forward pass to retrieve the output vectors from the last decoder block, denoted as $\mathbf{h}_1, \ldots, \mathbf{h}_{t-1}, \mathbf{h}_v$.
   (f) For each $v \in V_k$, compute the maximum cosine similarity score $s(\mathbf{h}_v, \mathbf{h}_j)$ between $\mathbf{h}_v$ and any $\mathbf{h}_j$, leading to the degeneration penalty term:

   $$\max\{s(\mathbf{h}_v, \mathbf{h}_j) \mid 1 \leq j \leq t-1\}$$

   (g) Once all degeneration penalty terms are computed, perform a full argmax to select the next token.

   $$x_t = \underset{v \in V_k}{\operatorname{argmax}} \left\{ (1-\alpha) \times \underbrace{P_{\text{LM}}(v|\mathbf{y}_{<t}, \mathbf{x})}_{\text{model confidence}} - \alpha \times \underbrace{(\max\{s(\mathbf{h}_v, \mathbf{h}_j) \mid 1 \leq j \leq t-1\})}_{\text{degeneration penalty}} \right\}$$

   (h) Append the selected token to the target sequence.
3. Repeat until the end-of-sequence token (EOS) is generated or the maximum sequence length is reached.

Exercise 3: Choose Your Own Decoding Adventure

To apply what you've learned, navigate to the `decoding_strategies.ipynb` notebook. In this notebook, you'll download Google's T5 model from HuggingFace and experiment with various decoding strategies on two datasets: the WMT16 German-to-English dataset for Neural Machine Translation, and the Writing Prompts dataset for short story generation.

You will evaluate how the different decoding strategies affect the quality and diversity of the generated outputs. Pay special attention to how each strategy works in each use case. This hands-on exercise will help you understand the trade-offs between different decoding methods in real-world tasks.

Test your work!

Specific test functions have been developed to test your work. These tests are designed to automatically check the correctness of your implementations.

Useful links

1. **torch.gather**: Gathers values along an axis specified by ‘dim‘. For each value in the output tensor, its index in the input tensor is specified by an index tensor.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.gather.html`
2. **torch.full**: Returns a tensor filled with the specified value, with the shape defined by the input shape.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.full.html`
3. **torch.argmax**: Returns the indices of the maximum values along a specified dimension.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.argmax.html`
4. **torch.topk**: Returns the ‘k‘ largest elements of the given input tensor along a specified dimension.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.topk.html`
5. **torch.Tensor.contiguous**: Returns a contiguous tensor containing the same data as the input tensor. If the tensor is already contiguous, it is returned as is.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.Tensor.contiguous.html`
6. **torch.nn.functional.normalize**: Applies Lp normalization over a specified dimension of the input tensor.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.functional.normalize.html`
7. **torch.nn.functional.log_softmax**: Applies the log softmax function to an input tensor.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.functional.log_softmax.html`
8. **torch.nn.functional.softmax**: Applies the softmax function to an input tensor.
   Documentation: `https://pytorch.org/docs/stable/generated/torch.nn.functional.softmax.html`