Lab 2: The One Where You Learn to Speak

Introduction

In the previous laboratory, you have used self-attention to implement an encoder-only model, such as BERT. However, in decoder-only architectures such as GPT-2, the model learns to generate text autoregressively, where each token is predicted based on the previously generated tokens.

This laboratory practice is designed to give you hands-on experience with the Decoder architecture. You will build the model from scratch, implement masked self-attention, and explore how autoregressive generation works by preventing the model from attending to future tokens.

Throughout this practice, you will:

- Implement masked self-attention, ensuring the model only attends to previous tokens when generating sequences.
- Build the decoder model by stacking multiple decoder layers, each using masked self-attention and feed-forward networks.
- Train a decoder model to generate a specific type of text: names.

By the end of this lab, you will understand how masked self-attention and decoder-based architectures function for sequence generation tasks.

Learning Outcomes

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

- Implement masked self-attention and understand how the look-ahead mask restricts future token visibility.
- Build and stack Decoder layers by combining masked self-attention and feed-forward networks.
- Understand the autoregressive nature of sequence generation, where each token is predicted based on previous tokens.

Deliverables

1. `decoder.py` - This file must implement all the modules related to the Decoder architecture.
2. `data_processing.py` - This file must implement the data loading and processing functions.
3. `train.py` - This file must implement the `train_model` function to train a `Decoder` model.
4. `generate_names.ipynb` - This file must train a decoder model and generate new names. Play with it!

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

Exercises

Exercise 1: Reload Your Weapons!

In this exercise, you will implement the `FeedForward` and `Embeddings` classes in the `decoder.py` file. You have already implemented them in the previous laboratory practice, so just copy and paste the code. You might want to keep track of the rest of the code of the previous practice for the following exercises, so do not close that file.

Exercise 2: Out of Sight, Out of Mind

In this exercise, you will implement the `MaskedAttentionHead` class. Masked self-attention (also known as Causal self-attention) must be implemented for tasks such as autoregressive text generation, where each word can only attend to itself and the preceding words. The key difference from regular self-attention is the use of a mask to zero out attention scores for future word positions. Do not worry right now on how to generate the mask, that is a job for the future you.

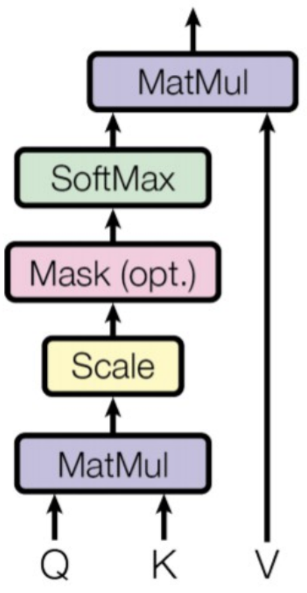

Figure 1: Scaled dot product attention diagram, with masking operation

In order to complete this class, check the code from the `AttentionHead` class from the previous laboratory practice. Also, you might want to check the `masked_fill_()` method of Pytorch tensors in order to set the elements selected by a mask to a suitable value.

Hint #1:

If you set an element of the input tensor to `float('-inf')` before applying the softmax, the output of the softmax in that position would be 0.

Exercise 3: Four Eyes See More Than Two

In this exercise, you will implement the `MultiHeadMaskedAttention` class. This class extends the functionality of the single masked attention head by using multiple heads. The mechanism is the same as the one used for the encoder but with masking applied to ensure that the decoder only attends to past and present tokens. Therefore, check the code from the `MultiHeadAttention` class from the previous laboratory practice.

Exercise 4: The Classical Sandwich

In this exercise, you will implement the `TransformerDecoderLayer` class. This class is also known as `Block` , `DecoderBlock` or `TransformerDecoderBlock` in other implementations. This class will contain the masked multi-head attention followed by a feed-forward network. Additionally, residual connections and layer normalization will be used, similar to the `TransformerEncoderLayer` from the previous practice.

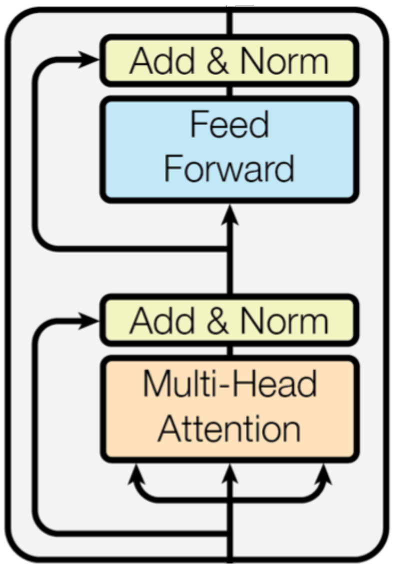

Figure 2: TransformerDecoderLayer diagram

Exercise 5: A Wild Multi-layered Sandwich appears!

Now that we have the individual decoder block, let´s stack blocks. As a curious fact, GPT-2 have 12 decoder blocks stacked over each other. In this exercise, you will implement the `TransformerDecoder` class. This class will combine an `Embeddings` layer (that has been prepared for you) and a stack of `TransformerDecoderLayer` instances.

Now you are your future self, so you will have to generate the mask that would be sent to the `MaskedAttentionHead` blocks. In order to implement correctly the mask, you will have to obtain the dimensions of the input data in order to know the length of the sequence. Once the length of the sequence is known, create a lower-triangular matrix to indicate which are the elements that do not need to be masked to prevent information leaks from future words.

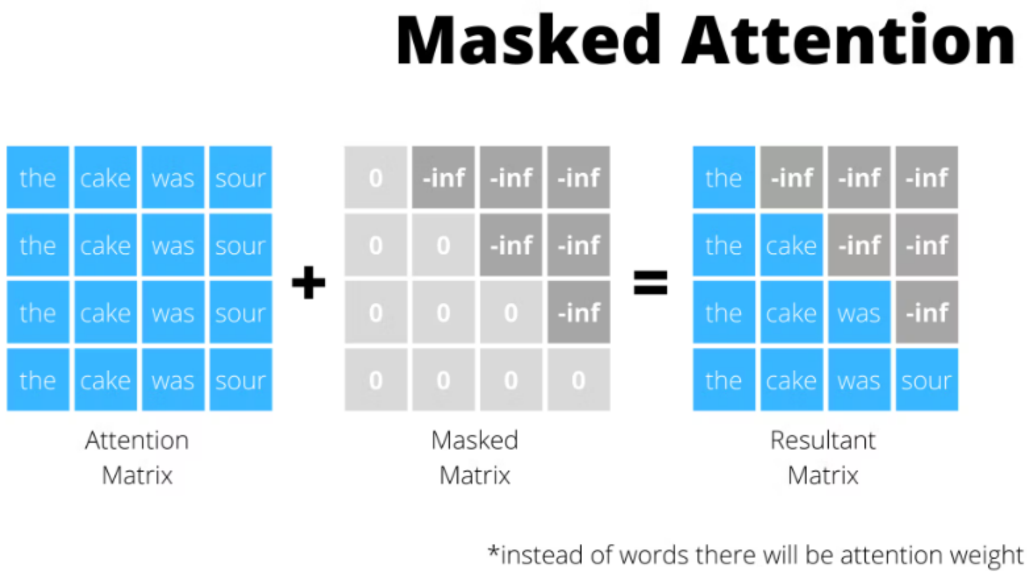

Figure 3: Masked Attention example

Hint #2:

Remember to propagate the mask through all the samples in the batch and to send it to the device where the inputs are stored to allow GPU training.

Exercise 6: Musk wishes he had had this

Open the `generate_names.ipynb` notebook file. In this notebook, you will train a decoder-only model to generate new names using the `names_raw.txt` dataset you already know from last year. In order to ease this exercise, the function to load, preprocess the data, create the data loaders and train `TransformerDecoder` model has already been done for you. However, you will have to answer some questions based on these functions, so read carefully the notebook cells.

Your task is to implement the `generate_name()` function to generate new names using your trained Decoder model, with an optional prefix as a beginning of the new name. This function should:

- Tokenize the `start_token`, `end_token` and `prefix` (if any) using the `CharTokenizer` used during training.
- Character by character, used the `TransformerDecoder` model to predict the following character. In order to do this, you will have to:
  1. Obtain the logits predicted by the model based on the current tokens of your name (in the first iteration, it should be the `start_token` and the `prefix_tokens` (if any)).
  2. Divide the obtained logits by the `temperature` parameter. This temperature can be understood as the confidence the model has on its prediction (lower temperature resembles greater confidence). You have an in-depth explanation of the temperature concept in the notebook.

3. Obtained the probabilities of each token in the alphabet based on the logits (after temperature correction). Sample the next token using an appropriate probability distribution (its name begins with multi, ends with nomial).
4. This process shall end if the maximum sequence length is reached or the `end_token` is sampled.

Hint #3:

Remember to decode the name!

After you have implemented this function, you are ready to play with it. We have implemented for you some code cells that modify the temperature and the prefix parameters, feel free to modify them!

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, except for the `generate_name` function.

Useful links

1. **torch.tril**: Creates a lower triangular matrix by setting all elements above the diagonal to zero. This is useful for generating look-ahead masks to prevent attending to future tokens in the decoder.
Documentation: `https://pytorch.org/docs/stable/generated/torch.tril.html`

2. **torch.Tensor.expand**: Expands the shape of a tensor, allowing it to be broadcasted across different dimensions without making copies of the data. This is particularly useful when applying masks across batches.
Documentation: `https://pytorch.org/docs/stable/generated/torch.Tensor.expand.html`

3. **torch.multinomial**: Draws samples from a multinomial distribution based on the input probabilities. This is useful for sampling the next token during sequence generation according to the predicted probability distribution over the vocabulary.
Documentation: `https://pytorch.org/docs/stable/generated/torch.multinomial.html`