CS 261: Deep Generative Models
Homework 1 - Large Language Models

Available: 01/17/2024; **Due Date:** 23:59 PM PST, 02/04/2024

---

1 Motivation

In this homework, you will implement an autoregressive model with Transformers, an architecture that powers all modern Large Language Models (LLMs) like GPT and Llama. For those unfamiliar with this architecture, we will do a review session during the discussion on 01/19/2024. You will implement the core components of a Transformers model and several variants of Transformers such as Encoder-only, Decoder-only, and Encoder-Decoder models. You will then use these as the basis to train a (small) Language Model from scratch on Wikihow articles. The code skeleton is provided at `https://colab.research.google.com/drive/1BUtvB3WdKM_TsMpc8P7CFjHXcm6RygTQ`.

2 Logistics

Colab tutorial Students who are not familiar with Google Colab can checkout the tutorial at `https://colab.research.google.com/?utm_source=scs-index`.

Code structure We provided a code skeleton for the homework. Students are required to complete the code within the green comment blocks. See the figure below for an example. We also provided several simple test functions you can use as a sanity check for your implementation.

```
class Embedding(DummyEmbedding):
    def forward(self, idx):
        """
        :param idx: intTensor of shape (B,T)
        :returns embeddings: floatTensor of shape (B,T,n_embd)
        """
        B, T = idx.size()
        embeddings = None
        ##############################################################################
        # TODO:
        # Implement the embedding lookup.                                            #
        #                                                                            #
        # This will take a few lines.                                                #
        ##############################################################################
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################
        return embeddings

#Do not change, it will break the AutoGrader
embedding_def = In[-1]
```

Submission After completing the Colab, run the last cell and upload the `my_llm_implementation.py` file to GradeScope.

Grading scheme There are 9 unit tests for the implementation of each code block and an additional unit test for the whole pipeline of training and testing the Encoder-Decoder model. The first 9 unit tests are given 1 point, while the last test is given 2 points.

3 Useful tips

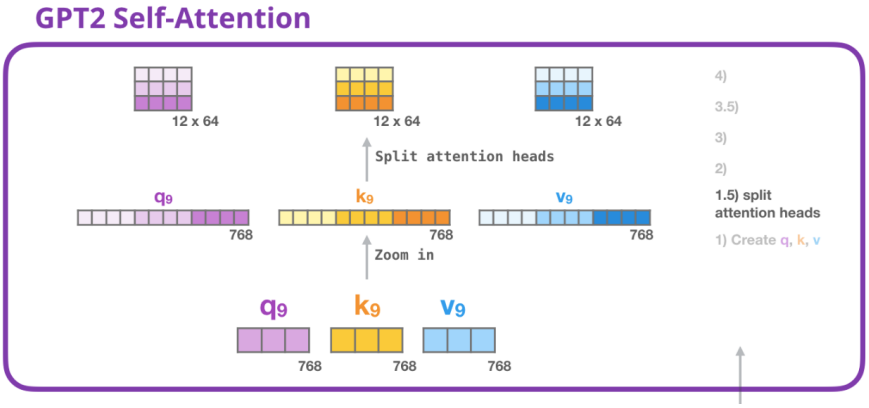
1) `reshape` and `transpose` are not equivalent functions. You'll need to use both of these functions in a correct implementation of MHA. See the example below to illustrate how these methods differ.

```
a = torch.arange(0,3).repeat(4).reshape(1,3,4)
print("Swapping Last Dimension Using Transpose")
print(a.transpose(1, 2))
print()
print("Reshaping Tensor (doesn't actually Transpose)")
print(a.reshape(1, 4, 3))
```

2) The tests require assumptions about where dropout is applied, according to what GPT-2 does in practice. Since this is a somewhat arbitrary decision in the original GPT-2 implementation, we specify this in the instructions.

```
# Implement multi-headed self-attention in GPT-2 Style                    #
# Use the provided layers initialized in the DummySelfAttention constructor #
# Apply dropout to the attention values after softmax and the final output  #
#                                                                          #
```

3) The implementation is Multi-Headed attention. If your implementation doesn't have code to handle the step illustrated below from the reference linked in the code, you are likely computing single-headed attention instead.



See the hint from the Colab as well

```
# Note that we need this to be true in GPT-style MHA
# Knowing this might come in handy :)
assert config.n_embd % config.n_head == 0
```

4) Your implementation should not use a for loop over `n_heads`. There's a for-loop version that is theoretically equivalent, but it's an order of magnitude slower so not used in practice. The instructions specify that you should compute all heads in parallel and the tests are designed to capture the Floating Point drift introduced by the for-loop version.

> Note that while this is "Multi-head", all heads can be computed in parallel with a single matrix multiplication. You can find an in-depth description of this in the reference linked in the code.

Acknowledgements: We acknowledge Will Held and Danfei Xu for the materials used in creating this homework assignment.