DLE Home Assignment

In this assignment, you will be presented with an architecture for a small NLP Transformer model and asked to implement it. We will also supply weights for this model which you'll need to load into the model, run a forward pass on some inputs, and make sure the results are similar to our pre-computed results. Then, you'll be asked to measure the latency of the forward pass, make some small changes in the architecture, and see how they affect results and latency. As a bonus, you'll be asked to parallelize a component of the Transformer.

The Architecture

For this assignment, we'll use a standard GPT2-like decoder-only (causal) transformer architecture. We'll start by defining the architecture.

Notations

- S - the sequence length of the input
- $S_{\max}$- the maximum sequence length the model can handle
- V - the vocabulary size of the model
- d - the hidden dimension of the model
- d_u- the dimension of the higher projection in the MLP block
- N_l - the number of Transformer layers in the model
- N_h- the number of heads in the Attention mechanism
- h - the hidden dimension of each Attention head ($h = d/N_h$)

Model

The model gets as input a tensor x of integers with shape $[1, S]$ representing the token IDs, and produces an output tensor y of floats with shape $[S, V]$ representing the logits of the tokens. Here is the definition of the forwad pass of the model. Next to each line we show the shape of the resulting tensor.

$$
\begin{array}{ll}
\text{input:} \quad x & [1, S] \\
y = \text{Embedding}(x) & [S, d] \\
\text{for } i \text{ in } N_l: & \\
\quad y = \text{TransformerLayer}_i(y) & [S, d] \\
y = \text{LayerNorm}(y) & [S, d] \\
y = \text{LMHead}(y) & [S, V] \\
\text{output:} \quad y & [S, V]
\end{array}
$$

Embedding

This is the embedding layer that transforms the input from a sequence of integers to a sequence of dense floating-point embeddings. It's composed of 2 simple lookup tables, and its forward pass is:

$$
\begin{array}{ll}
\text{input:} \quad x & [1, S] \\
s \text{ is the index along the sequence dim} & \\
y_s = E_w^{(x_s)} + E_p^{(s)} & [1, d] \\
\text{output:} \quad y & [S, d]
\end{array}
$$

E_w is the "word embeddings" matrix and has a shape of $[V, d]$. $E_w^{(i)}$ is the i-th row of the matrix.

E_p is the "positional embeddings" matrix and has a shape of $[S_{\max}, d]$. $E_p^{(i)}$ is the i-th row of the matrix.

Note: Each of the embedding matrices is exactly what you get from torch.nn.Embedding.

TransformerLayer

Our model has N_l transformer layers. Each one of them implements the following forward pass:

$$
\begin{array}{ll}
\text{input:} \quad x & [S, d] \\
t = x + \text{Attention}\left(\text{LayerNorm}(x)\right) & [S, d] \\
y = t + \text{MLP}\left(\text{LayerNorm}(\text{t})\right) & [S, d] \\
\text{output:} \quad y & [S, d]
\end{array}
$$

LayerNorm

This is the standard LayerNorm, which normalizes the input along the hidden dimension. It's defined by:

$$
\begin{array}{ll}
\text{input:} \quad x & [S, d] \\
s \text{ is the index along the sequence dim} & \\
y_s = \frac{x_s - \bar{x_s}}{\sqrt{\text{Var}(x_s) + \epsilon}} \odot \gamma + \beta & [1, d] \\
\text{output:} \quad y & [S, d]
\end{array}
$$

$\bar{x_s}$ is the average of the s-th row of the input matrix (along the hidden dimension d)

$\text{Var}(x_s)$ is the unbiased variance of the s-th row of the input matrix (along the hidden dimension d)

ϵ is a scalar constant.

γ and β are the weights and biases of this layer, both have shapes of $[1, d]$.

Note that $\odot$ is the element-wise product between 2 tensors of the same shape.

Note: This LayerNorm is exactly what you get from torch.nn.LayerNorm.

Attention

This is the heart of the Transformer model (and the most complicated part). This is the part that's in charge of computing and storing the dependencies between different tokens in the sequence:

$$
\begin{array}{ll}
\text{input:} \quad x & [S, d] \\
\text{for } i \text{ in } N_h: & \\
\quad Q^{(i)} = xW_Q^{(i)} + b_Q^{(i)} & [S, h] \\
\quad K^{(i)} = xW_K^{(i)} + b_K^{(i)} & [S, h] \\
\quad V^{(i)} = xW_V^{(i)} + b_V^{(i)} & [S, h] \\
\quad y^{(i)} = \frac{Q^{(i)} K^{(i)T}}{\sqrt{h}} & [S, S] \\
\quad y_{mn}^{(i)} = \begin{cases} y_{mn}^{(i)} & \text{if } m <= n \\ -\infty & \text{if } m > n \end{cases} & [S, S] \\
\quad y^{(i)} = \text{softmax}\left(y^{(i)}\right) V^{(i)} & [S, h] \\
y = \text{concat}\left[y^{(i)}\right] & [S, d] \\
y = yW_O + b_O & [S, d] \\
\text{output:} \quad y & [S, d]
\end{array}
$$

We have N_h attention heads. Each attention head (i) has three weight matrices $W_Q^{(i)}$, $W_K^{(i)}$, $W_V^{(i)}$ with shape $[d, h]$, and three bias vectors $W_Q^{(i)}$, $W_K^{(i)}$, $b_V^{(i)}$ with shape $[1, h]$.

The concat operation is a simple concatenation along the last dimension of the tensor.

W_O and b_O are the weights and biases for the output projection. They have shapes of $[d, d]$ and $[1, d]$ accordingly.

The second to last line inside the "for" clause is the one responsible for the causality of the model. It makes sure every token can attend only the previous tokens in the sequence (and to itself). Without it, we'll have an "all-to-all" attention mechanism where each token can attend to all other tokens in the sequence. That's what is usually done in encoder transformer models.

Note: This is the standard Attention mechanism. For more information and in-depth analysis of what happens here you can refer to The Illustrated Transformer blogpost, as well as various other resources.

MLP

The MLP block of the Transformer layer projects its input to a higher dimension, applies some non-linear activation function, and then projects back down to the hidden dimension of the model:

$$
\begin{array}{ll}
\text{input:} \quad x & [S, d] \\
t = \left[\text{act}\left(xW_u + b_u\right)\right] W_d + b_d & [S, d] \\
\text{output:} \quad y & [S, d]
\end{array}
$$

act is the non-linear activation function.

W_u and b_u are the weights and biases for the projection to the higher dimension. They have shapes of $[d, d_u]$ and $[1, d_u]$ respectively.

W_d and b_d are the weights and biases for the projection back down to the model dimension. They have shapes of $[d_u, d]$ and $[1, d]$ respectively.

LMHead

This layer's job is to project from the hidden dimension of the model back to the vocab dimension. It's a simple matrix multiplication:

$$\begin{array}{ll} \text{input:} \quad x & [S, d] \\ y = xE_w^T & [S, V] \\ \text{output:} \quad y & [S, V] \end{array}$$

E_w is the same word embeddings matrix from the Embedding layer.

Task

Implement

You now need to implement this architecture, with the following params:

- $S_{\max} = 1024$
- $V = 512$
- $d = 256$
- $d_u = 1280$ (5 times the hidden dimension of the model)
- $N_l = 3$
- $N_h = 2$
- $h = 128$ ($= d/N_h$)
- $\epsilon = 1 \cdot e^{-5}$ (for all the different LayerNorms)
- MLP activation function: the non-approximated GELU (as defined in torch.nn.functional.gelu with `approximate='none'`)

This configuration results in a small model with only about 3.1 million parameters. You should be able to run it easily on your PC/mac. No GPU needed.

You can implement this architecture however is most convenient for you. You can implement it directly using PyTorch/JAX/TensorFlow/any other ML framework. You can also use dedicated NLP frameworks like Huggingface.

1. How much RAM is the model consuming? What's the connection between RAM usage and model parameter count?

Reminder: This model is a standard GPT2-like model.

Measure Latency

Measure the latency of the forward pass of this model on the hardware you're using.

1. What input characteristics affect latency? What's the effect?
2. What model configuration params affect latency? What's the effect?

Supplied State Dict

You're supplied with a state dict with weights that match the model architecture, saved in a `.pt` format in `dummy_gpt2_model.pt`. The names of the weights in the state dict are pretty self explanatory and you should be able to easily match them with the weights and biases described in the Architecture. You are also supplied with `example_outputs.pt` which contains a list of examples of input IDs and their pre-computed output logits.

1. Load this state dict into the model you created in the previous step.
2. Run the model on the input IDs in `example_outputs.pt`.
3. Make sure you're getting the expected output logits.

Notes:

- The state dict contains dummy random weights (it wasn't trained).
- For the Attention weights W_Q, W_K, and W_V (and their respective biases), the state dict does **not** have different weights for each Attention head, but rather has a concatenated single matrix that holds all those weights. This matrix is called `attn.qkv_proj.weight`, it has a shape of $[d, 3d]$, and schematically holds the weights like this:

$$\text{attn.qkv\_proj.weight} = \left[\text{concat}\left(W_Q^{(i)}\right), \text{concat}\left(W_K^{(i)}\right), \text{concat}\left(W_V^{(i)}\right)\right]$$

  - Similarly, for the Attention biases (with shape $[1, 3d]$):

$$\text{attn.qkv\_proj.bias} = \left[\text{concat}\left(b_Q^{(i)}\right), \text{concat}\left(b_K^{(i)}\right), \text{concat}\left(b_V^{(i)}\right)\right]$$

- If there's anything you don't understand about the state dict format, don't hesitate to contact us.

Architecture Changes

You will now make some small changes in the architecture. For each change answer the following questions:

1. What's the effect of the change on the output logits for some given input IDs? How can it be qunatified?
2. Does this change have an effect on latency? Even if it doesn't in the implementation you're using, does it have the potential to effect latency?

These are the changes you should make (each one is independent of the others):

- Change the LayerNorm ϵ to $1 \cdot e^{-4}$
- In the Attention mechanism, we scale the product of the Q and K activations by $\sqrt{h}$. The change you should make is to not scale the activations at all.
- Change the MLP activation function to the approximated version of GELU (so torch.nn.functional.gelu with `approximate='tanh'`). This is also known as the OpenAI GELU.
- BONUS: Change the connectivity in the Transformer layer (each of these changes is independent of the others):

  Reminder: the original implementation looks like this (x is the input, y is the output, and both have shape $[S, d]$:

$$
\begin{aligned}
t &= x + \text{Attention}\left(\text{LayerNorm}(x)\right) && [S, d] \\
y &= t + \text{MLP}\left(\text{LayerNorm}(\text{t})\right) && [S, d]
\end{aligned}
$$

- Create a residual connection between the input and the MLP output, instead of the one betwen the Attention output and MLP output:

$$
\begin{aligned}
t &= x + \text{Attention}\left(\text{LayerNorm}(x)\right) && [S, d] \\
y &= x + \text{MLP}\left(\text{LayerNorm}(\text{t})\right) && [S, d]
\end{aligned}
$$

- Drop the intermediate residual connection entirely:

$$
\begin{aligned}
t &= x + \text{Attention}\left(\text{LayerNorm}(x)\right) && [S, d] \\
y &= \text{MLP}\left(\text{LayerNorm}(\text{t})\right) && [S, d]
\end{aligned}
$$

- PaLM parallel layers - the MLP works on the input instead of on the attention output:

$$
y = x + \text{Attention}\left(\text{LayerNorm}(x)\right) + \text{MLP}\left(\text{LayerNorm}(\text{x})\right) \qquad [S, d]
$$

Note: All the changes proposed here do not change the model weights so the provided state dict is still relevant for the resulting model.

BONUS - Parallelize MatMul

As you've seen by now, the most prominent operation in the Transformer architecture (and frankly in almost all DL models) is matrix multiplication. The model presented here is small and there's no problem with performing these operations. Yet, modern LLMs are very large and some of them (including our in-house models) have to be parallelized across different hardware chips because a single chip doesn't have enough memory to hold the model. In this section, you'll parallelize this basic matrix multiplication operation.

In the real world, LLMs are parallelized across different GPUs. For the sake of this assignment, you don't need any GPUs. You can parallelize the matrix multiplication on different CPU processes, using python's native `mutliprocessing` package.

You get 2 matrices as input: A with shape $[N, K]$ and B with shape $[K, M]$. You need to compute the output matrix $C = AB$ with shape $[N, M]$.

In the parallelized setup, each process holds only part of the input matrices. Let's assume we're parallelizing the MatMul over 2 processes and along the inner dimension with size K. In this case:

- process 1 holds A_1 with shape $[N, K_1]$ and B_1 with shape $[K_1, M]$
- process 2 holds A_2 with shape $[N, K_2]$ and B_2 with shape $[K_2, M]$
- $K_1 + K_2 = K$

You are required to compute the matrix multiplication in each process on its part of the input matrices, and then combine the results to get the full output matrix C. Since we're not really dealing with huge matrices that have to be parallelized, you should compare the result you're getting from the parallelized operation to the one from a non-parallelized MatMul, and make sure they're identical.

Hint: Recall the mathematic definition of matrix multiplication:

$$
C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}
$$