ASSIGNMENT 1: DEVELOPMENT MINIMALIST OF THE LLAMA2 MODEL

(core components of Llama2)

TO DO:

- Implement some important components of the Llama2 model to better understanding its architecture.
- Perform sentence classification on sst dataset and cfimdb dataset with this model.

DETAIL:

The code to implement can be found in llama.py, classifier.py and optimizer.py. You are reponsible for writing *core components* of Llama2 (one of the leading open source language models). In doing so, you will gain a strong understanding of neural language modeling. We will load pretrained weights for your language model from stories42M.pt; an 8-layer, 42M parameter language model pretrained on the TinyStories dataset (a dataset of machine-generated children's stories). Download tại link: https://www.cs.cmu.edu/~vijayv/stories42M.pt
This model is small enough that it can be trained (slowly) without a GPU. You are encouraged to use Colab or a personal GPU machine (e.g. a Macbook) to be able to iterate more quickly.

Once you have implemented these components, you will test our your model in 3 settings:

1. Generate a text completion (starting with the sentence "I have wanted to see this thriller for a while, and it didn't disappoint. Keanu Reeves, playing the hero John Wick, is"). You should see coherent, grammatical English being generated (though the content and topicality of the completion may be absurd, since this LM was pretrained exclusively on children's stories).
2. Perform zero-shot, prompt-based sentiment analysis on two datasets (SST-5 and CFIMDB). This will give bad results (roughly equal to choosing a random target class).
3. Perform task-specific finetuning of your Llama2 model, after implementing a classification head in classifier.py. This will give much stronger classification results.

Detail in Structure of Llama Section (Page 3)

Important Notes

- Follow setup.sh to properly setup the environment and install dependencies.
- There is a detailed description of the code structure in Page 5, including a description of which parts you will need to implement.
- You are only allowed to use libraries that are installed by setup.sh, no other external libraries are allowed (e.g., transformers).

- The data/cfimdb-test.txt file provided to you does not contain gold-labels, and contains a placeholder negative (-1) label. Evaluating your code against this set will show lower accuracies so do not worry if the numbers don't make sense.
- We will run your code with commands below (under "Reference outputs/accuracies"), so make sure that whatever your best results are reproducible using these commands.
    - Do not change any of the existing command options (including defaults) or add any new required parameters

Reference outputs/accuracies:

Text Continuation (python run_llama.py --option generate) You should see continuations of the sentence I have wanted to see this thriller for a while, and it didn't disappoint. Keanu Reeves, playing the hero John Wick, is.... We will generate two continuations - one with temperature 0.0 (which should have a reasonably coherent, if unusual, completion) and one with temperature 1.0 (which is likely to be logically inconsistent and may contain some coherence or grammar errors).

Zero Shot Prompting Zero-Shot Prompting for SST:

python run_llama.py --option prompt --batch_size 10 --train data/sst-train.txt --dev data/sst-dev.txt --test data/sst-test.txt --label-names data/sst-label-mapping.json --dev_out sst-dev-prompting-output.txt --test_out sst-test-prompting-output.txt [--use_gpu]

Prompting for SST: Dev Accuracy: 0.213 (0.000) Test Accuracy: 0.224 (0.000)

Zero-Shot Prompting for CFIMDB:

python run_llama.py --option prompt --batch_size 10 --train data/cfimdb-train.txt --dev data/cfimdb-dev.txt --test data/cfimdb-test.txt --label-names data/cfimdb-label-mapping.json --dev_out cfimdb-dev-prompting-output.txt --test_out cfimdb-test-prompting-output.txt [--use_gpu]

Prompting for CFIMDB: Dev Accuracy: 0.498 (0.000) Test Accuracy: -

Classification Finetuning

python run_llama.py --option finetune --epochs 5 --lr 2e-5 --batch_size 80 --train data/sst-train.txt --dev data/sst-dev.txt --test data/sst-test.txt --label-names data/sst-label-mapping.json --dev_out sst-dev-finetuning-output.txt --test_out sst-test-finetuning-output.txt [--use_gpu]

Finetuning for SST: Dev Accuracy: 0.414 (0.014) Test Accuracy: 0.418 (0.017)

python run_llama.py --option finetune --epochs 5 --lr 2e-5 --batch_size 10 --train data/cfimdb-train.txt --dev data/cfimdb-dev.txt --test data/cfimdb-test.txt --label-names data/cfimdb-label-mapping.json --dev_out cfimdb-dev-finetuning-output.txt --test_out cfimdb-test-finetuning-output.txt [--use_gpu]

Finetuning for CFIMDB: Dev Accuracy: 0.800 (0.115) Test Accuracy: -Mean reference accuracies over 10 random seeds with their standard deviation shown in brackets.

Submission

Report: your zip file can include a pdf file, named Group_ID-report.pdf. Report can present implement the requirements, accuracy, best results are with some hyperparameters other than the default, how running your code... no more than 3 pages

Canvas Submission

For submission via Canvas, the submission file should be a zip file with the following structure:

```
GROUP_ID/
├── run_llama.py
├── base_llama.py
├── llama.py
├── rope.py
├── classifier.py
├── config.py
├── optimizer.py
├── sanity_check.py
├── tokenizer.py
├── utils.py
├── README.md
├── structure.md
├── sanity_check.data
├── generated-sentence-temp-0.txt
├── generated-sentence-temp-1.txt
├── [OPTIONAL] sst-dev-advanced-output.txt
├── [OPTIONAL] sst-test-advanced-output.txt
├── sst-dev-prompting-output.txt
├── sst-test-prompting-output.txt
├── sst-dev-finetuning-output.txt
├── sst-test-finetuning-output.txt
├── [OPTIONAL] cfimdb-dev-advanced-output.txt
├── [OPTIONAL] cfimdb-test-advanced-output.txt
├── cfimdb-dev-prompting-output.txt
├── cfimdb-test-prompting-output.txt
├── cfimdb-dev-finetuning-output.txt
├── cfimdb-test-finetuning-output.txt
├── Group_ID-report.pdf
└── setup.sh
```

Structure of Llama

llama.py

This file contains the Llama2 model whose backbone is the transformer. We recommend walking through Section 3 of the paper to understand each component of the transformer.

Attention

The multi-head attention layer of the transformer. This layer maps a query and a set of key-value pairs to an output. The output is calculated as the weighted sum of the values, where the weight of each value is computed by a function that takes the query and the corresponding key. To implement this layer, you can:

1. linearly project the queries, keys, and values with their corresponding linear layers
2. split the vectors for multi-head attention
3. follow the equation to compute the attended output of each head
4. concatenate multi-head attention outputs to recover the original shape

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Llama2 uses a modified version of this procedure called Grouped-Query Attention where, instead of each attention head having its own "query", "key", and "vector" head, some groups of "query" heads share the same "key" and "vector" heads. To simplify your implementation, we've taken care of steps #1, 2, and 4 here; you only need to follow the equation to compute the attended output of each head.

LlamaLayer

This corresponds to one transformer layer which has

1. layer normalization of the input (via Root Mean Square layer normalization)
2. self-attention on the layer-normalized input
3. a residual connection (i.e., add the input to the output of the self-attention)
4. layer normalization on the output of the self-attention
5. a feed-forward network on the layer-normalized output of the self-attention

6. a residual connection from the unnormalized self-attention output added to the output of the feed-forward network

Llama

This is the Llama model that takes in input ids and returns next-token predictions and contextualized representation for each word. The structure of Llama is:

1. an embedding layer that consists of token embeddings tok_embeddings.
2. llama encoder layer which is a stack of config.num_hidden_layers LlamaLayer
3. a projection layer for each hidden state which predicts token IDs (for next-word prediction)
4. a "generate" function which uses temperature sampling to generate long continuation strings. Note that, unlike most practical implementations of temperature sampling, you should not perform nucleus/top-k sampling in your sampling procedure.

The desired outputs are

1. logits: logits (output scores) over the vocabulary, predicting the next possible token at each point
2. hidden_state: the final hidden state at each token in the given document

To be implemented

Components that require your implementations are comment with #todo. The detailed instructions can be found in their corresponding code blocks

- llama.Attention.forward
- llama.RMSNorm.norm
- llama.Llama.forward
- llama.Llama.generate
- rope.apply_rotary_emb (this one may be tricky! you can use rope_test.py to test your implementation)
- optimizer.AdamW.step
- classifier.LlamaEmbeddingClassifier.forward

ATTENTION: you are free to re-organize the functions inside each class, but please don't change the variable names that correspond to Llama2 parameters. The change to these variable names will fail to load the pre-trained weights.

Sanity check (Llama forward pass integration test)

We provide a sanity check function at sanity_check.py to test your Llama implementation. It will reload two embeddings we computed with our reference implementation and check whether your implementation outputs match ours.

classifier.py

This file contains the pipeline to

- load a pretrained model
- generate an example sentence (to verify that your implemention works)
- call the Llama2 model to encode the sentences for their contextualized representations
- feed in the encoded representations for the sentence classification task
- fine-tune the Llama2 model on the downstream tasks (e.g. sentence classification)

LlamaSentClassifier (to be implemented)

This class is used to

- encode the sentences using Llama2 to obtain the hidden representation from the final word of the sentence.
- classify the sentence by applying dropout to the pooled-output and project it using a linear layer.

optimizer.py (to be implemented)

This is where AdamW is defined. You will need to update the step() function based on Decoupled Weight Decay Regularization and Adam: A Method for Stochastic Optimization. There are a few slight variations on AdamW, please note the following:

- The reference uses the "efficient" method of computing the bias correction mentioned at the end of section 2 "Algorithm" in Kigma & Ba (2014) in place of the intermediate m hat and v hat method.
- The learning rate is incorporated into the weight decay update (unlike Loshchiloc & Hutter (2017)).
- There is no learning rate schedule.

You can check your optimizer implementation using optimizer_test.py.

rope.py (to be implemented)

Here, you will implement rotary positional embeddings. This may be tricky; you can refer to slide 22 in https://phontron.com/class/anlp2024/assets/slides/anlp-05-transformers.pdf and Section 3 in https://arxiv.org/abs/2104.09864 for reference. To enable you to test this component modularly, we've provided a unit test at RoPE_test.py

base_llama.py

This is the base class for the Llama model. You won't need to modify this file in this assignment.

tokenizer.py

This is the tokenizer we will use. You won't need to modify this file in this assignment.

config.py

This is where the configuration class is defined. You won't need to modify this file in this assignment.

utils.py

This file contains utility functions for various purpose. You won't need to modify this file in this assignment.

Reference

[Vaswani el at. + 2017] Attention is all you need https://arxiv.org/abs/1706.03762

[Touvron el at. + 2023] Llama 2: Open Foundation and Fine-Tuned Chat Models
https://arxiv.org/abs/2307.09288