CMU 11868 LLM Systems
Spring 2024

Assignment 2

We will continue adding modules to miniTorch framework. In this assignment, students will implement a decoder-only transformer architecture (GPT-2), train it on machine translation task (IWSLT14 German-English), and benchmark their implementation.

Clone the repository for the homework `https://github.com/llmsystem/llmsys_s24_hw2/`

Setting up your code

- Install requirements

  ```
  pip install -r requirements.extra.txt
  pip install -r requirements.txt
  ```

- Install minitorch

  ```
  pip install -e .
  ```

- Copy `autodiff.py` and `run_sentiment.py` from Assignment 1

  ```
  autodiff.py -> minitorch/autodiff.py
  run_sentiment.py -> project/run_sentiment_linear.py
  ```

  Note the slight different suffix `"_linear"`.

- Copy `combine.cu` from Assignment 1

  ```
  combine.cu -> src/combine.cu
  ```

  Please ONLY copy your solution of assignment 1 in MatrixMultiplyKernel, mapKernel, zipKernel, reduceKernel to the combine.cu file for assignment 2.

  We have made some changes in `combine.cu` and `cuda_kernel_ops.py` for assignment 2 compared with assignment 1. We have relocated the GPU memory allocation, deallocation, and memory copying operations from `cuda_kernel_ops.py` to `combine.cu`, both for host-to-device and device-to-host transfers. We also change the datatype of `Tensor._tensor._storage` from `numpy.float64` to `numpy.float32`.

- Compile your cuda kernels

  ```
  bash compile_cuda.sh
  ```

Problem 1: Implementing Scalar Power and Tanh (20 pts)

MiniTorch is still missing a few important arithmetic operations that we need to implement a Transformer model: the **element-wise power** function and **element-wise tanh** function.

1. For each function, you'll have to fill out the forward and backward function in `minitorch/tensor_functions.py` as described in the minitorch demo.
2. Complete the `POW` and `TANH` function in `src/combine.cu`

Check out this link for relevant math functions.

Adam Optimizer

We provide Adam optimizer for HW2 at minitorch/optim.py. To verify the Adam optimizer (which now uses your Pow function), the validation accuracy of project/run_sentiment_linear.py should get above 60% in around 5 epochs.

Reference Performance

```
Epoch 1, loss 0.6930629134178161, train accuracy: 48.22%
Validation accuracy: 57.00%
Best Valid accuracy: 57.00%
Epoch 2, loss 0.6879702541563246, train accuracy: 55.78%
Validation accuracy: 55.00%
Best Valid accuracy: 57.00%
Epoch 3, loss 0.674045901828342, train accuracy: 60.44%
Validation accuracy: 62.00%
Best Valid accuracy: 62.00%
Epoch 4, loss 0.6554572939872741, train accuracy: 64.44%
Validation accuracy: 60.00%
Best Valid accuracy: 62.00%
Epoch 5, loss 0.6466168310907152, train accuracy: 64.67%
Validation accuracy: 59.00%
Best Valid accuracy: 62.00%
```

Implementing a Decoder-only Transformer Model

You will be implementing a Decoder-only Transformer model in `modules_transformer.py`. This will require you to first implement additional modules in `modules_basic.py`, similar to the Linear module from assignment 1.

We will recreate the GPT-2 architecture as described in Language Models are Unsupervised Multitask Learners.

Please read the implementation details section of the README file before starting.

Problem 2: Implementing Tensor Functions (20 pts)

You will need to implement the following functions in `minitorch/nn.py`. Additional details are provided in the `README.md` and each function's docstring.

- `GELU`
- `logsumexp`
- `one_hot`
- `softmax_loss`[1]

$$\ell(z, y) = \log \sum_{i=1}^{k} \exp z_i - z_y. \tag{1}$$

  The input to our softmax loss (softmax + cross entropy) function is `logits` and `target`. `logits` is a (minibatch, C) tensor where each row is a sample and contains the raw logits before softmax. `target` is a (minibatch,) tensor where each row correspond to the class of a sample.

  You'll want to utilize a combination of `logsumexp`, `one_hot`, and other tensor functions to compute this efficiently. (Our solution is only 3 lines long).

  Note: we want to return without reduction = None, so the resulting shape is (minibatch,)

[1] See slide 5 here for formula `https://llmsystem.github.io/llmsystem2024spring/assets/files/llmsys-03-autodiff-d3f8a17139dbf41fe16150b3d86ccdce.pdf`

Problem 3: Implementing Basic Modules (20 pts)

Here are the following modules you will have to implement:

1. `Linear`: You can use your implementation from Assignment 1, but will need to adapt it slightly given the new `backend` argument.
2. Dropout: Applies dropout. **Note**: if the flag `self.training` is false, then do not zero out any values in the input tensor.
3. LayerNorm1d: Applies layer normalization to a 2D tensor.
4. Embedding: Maps one-hot word vectors from a dictionary of fixed size to embeddings.

Problem 4: Implementing a Decoder-only Transformer Language Model (20 pts)

Finally, you will be implementing GPT-2 architecture in `minitorch/modules_transformer.py` with four modules and your earlier work.

- **MultiHeadAttention**: implements masked multi-head attention.
- **FeedForward**: implements the feed-forward operation.
- **TransformerLayer**: implements a transformer layer with the pre-LN architecture.
- **DecoderLM**: implements the full model with input and positional embeddings.

MultiHeadAttention

GPT-2 implements multi-head attention, meaning that each K, Q, V tensors formed from X is partitioned into h heads. The self-attention operation is then carried out for each batch and head, and the output is then reshaped to obtain the correct shape. Finally, the output is passed through a final out projection layer.

1. Projecting X into Q, K^T, V in the `project_to_query_key_value` function
   In the `project_to_query_key_value` function, the K, Q, V matrices are formed by projecting the input $X \in \mathbb{R}^{B \times S \times D}$ where B is the batch size, S is the sequence length, and D the hidden dimension. Formally, let h be the number of heads, D be the dimension of the input, and D_h be the dimension of each head where $D = h \times D_h$:

   - $X \in \mathbb{R}^{B \times S \times D}$ gets projected[2] to $Q, K, V \in \mathbb{R}^{B \times S \times D}$
   - $Q \in \mathbb{R}^{B \times S \times (h \times D_h)}$ gets unraveled to $Q \in \mathbb{R}^{B \times S \times h \times D_h}$
   - $Q \in \mathbb{R}^{B \times S \times h \times D_h}$ gets permuted to $Q \in \mathbb{R}^{B \times h \times S \times D_h}$

   Note you'll do the same for the V matrix and take care to transpose K along the last two dimensions.

2. Computing Self-Attention
   Let Q_i, K_i, V_i be the Queries, Keys, and Values for head i. You'll need to compute

$$\text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{D_h}} + M\right) V_i$$

   with batched matrix multiplication (which we've implemented for you) across each batch and head. M is the causal mask added to prevent your transformer from attending to positions in the future, which is crucial in an auto-regressive language model.

   Before returning, let $A \in \mathbb{R}^{B \times h \times S \times D_h}$ denote the output of self-attention. You'll need to:

   - Permute A to $A \in \mathbb{R}^{B \times S \times h \times D_h}$

[2] We could actually do this with a single layer and split the output in 3.

- Reshape A to $A \in \mathbb{R}^{B \times S \times D}$

3. Finally pass self-attention output through the out projection layer

FeedForward

You'll pass the output of self-attention through:

1. The first linear layer to expand the hidden dimension to 256
2. The GELU activation function
3. The second linear layer to shrink the dimension back to D
4. A dropout layer

TransformerLayer

Let's combine the MultiHeadAttention and Feedforward modules to form one layer of our transformer. Note that GPT-2 architecture employs the pre-LN architecture as shown below. **You'll want to follow the pre-LN variant on the right.**

The differences can be described in On Layer Normalization in the Transformer Architecture.

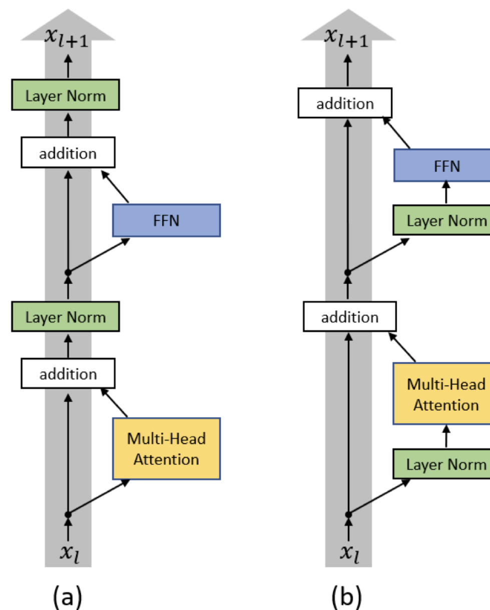

Figure 1: (a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

DecoderLM

Let's combine all our work to create our final model. Your input will be a tensor containing a minibatch of tokens X of shape $(\text{batch size}, \text{sequence length})$. You'll need to:

1. Get the token and positional embeddings for your input X
2. Add the embeddings together [3] before passing them through a dropout layer
3. Pass your input now of shape $(\text{batch size}, \text{sequence length}, \text{embedding dimension})$ through all your transformer layers
4. Pass your input through a final LayerNorm
5. Pass your input through a final linear layer to project your input's hidden dimension to the vocabulary size to perform inference or pass through your loss function.

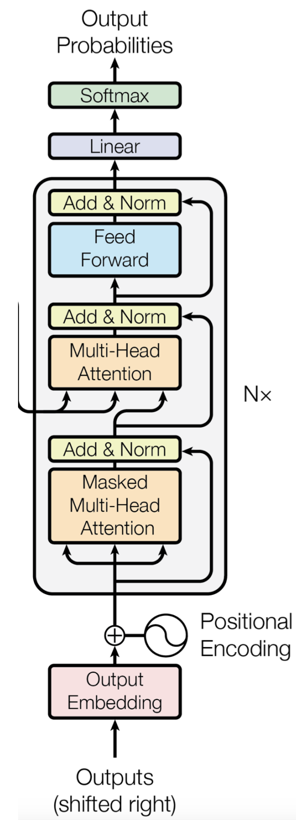

Figure 2: Transformer Decoder

[3] See Jurasky and Martin, Chapter 10.1.3 for more details

Problem 5: Machine Translation Pipeline (20 pts)

Implement a training pipeline of machine translation on IWSLT (De-En).

1. collate_batch: Prepares a batch of examples for model training or evaluation by tokenizing and padding them.

```python
def collate_batch(
        examples, src_key, tgt_key, tokenizer, model_max_length, backend):
    ...
```

Parameters

- examples: A list of examples to be processed.
- src_key: The key for accessing source texts in the examples.
- tgt_key: The key for accessing target texts in the examples.
- tokenizer: The tokenizer to be used for encoding the texts.
- model_max_length: The maximum sequence length the model can handle.
- backend: The backend of minitorch tensors.

Returns A dictionary containing keys: "input_ids", "labels", "label_token_weights", each indicates a minitorch tensor with shape (len(examples), model_max_length).

Notes ["input_ids"] for every example in the De-En translation, the "input_ids" will be: <de_token_ids> + <de_eos_id> + <en_token_ids> + <en_eos_id> + <pad_ids> where the pad_ids makes the length of input_ids to be model_max_length

["labels"]: the next tokens to be predicted, which will be used in the cross-entropy loss function, e.g., for a example tokenized as [a, b, c, d], input_ids and labels can be [a, b, c] and [b, c, d], respectively.

["label_token_weights"] The "label_token_weights" are used to differentiate between the source (weight = 0) and target (weight = 1) tokens for loss calculation purposes. (the MLE loss is computed on target tokens only.)

2. loss_fn: Compute MLE loss for a batch.

```python
def loss_fn(batch, model):
    ...
```

Parameters

- `batch`: The result of `collate_fn`, a dict with `"input_ids"`, `"labels"`, and `"label_token_weights"`.
- `model`: The minitorch model to be trained.

Returns A scalar loss value for this batch, averaged across all target tokens.

Hint Use the function minitorch.nn.softmax_loss

3. `generate`: Generates target sequences for the given source sequences using the model, based on argmax decoding. Note that it runs generation on examples one-by-one instead of in a batched manner.

```
def generate(model,
             examples,
             src_key,
             tgt_key,
             tokenizer,
             model_max_length,
             backend,
             desc):
    ...
```

Parameters

- `model`: The model used for generation.
- `examples`: The dataset examples containing source sequences.
- `src_key`: The key for accessing source texts in the examples.
- `tgt_key`: The key for accessing target texts in the examples.
- `tokenizer`: The tokenizer used for encoding texts.
- `model_max_length`: The maximum sequence length the model can handle.
- `backend`: The backend of minitorch tensors.
- `desc`: Description for the generation process (used in progress bars).

Returns A list of texts as generated target sequences.

Test Performance

Once all blanks are filled, run

```
python project/run_machine_translation.py
```

The outputs and bleu scores will be save in `"./workdir_vocab10000_lr0.02_embd256"`. you should get BLEU score around 7 in the first epoch, and around 20 in 10 epochs. *Every epoch takes around an hour, and every training step takes around 25 seconds on A10G.*

Reference Performance

```
workdir_vocab10000_lr0.02_embd256/eval_results_epoch0.json:{"validation_loss": 4.426930904388428, "bleu": 7.975168992203509}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch1.json:{"validation_loss": 3.944546937942505, "bleu": 8.4577590239961}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch2.json:{"validation_loss": 3.6387012004852295, "bleu": 12.161628606767161}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch3.json:{"validation_loss": 3.3925259113311768, "bleu": 13.158611481234598}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch4.json:{"validation_loss": 3.1942338943481445, "bleu": 14.790606862740633}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch5.json:{"validation_loss": 3.0331788063049316, "bleu": 16.406111101656208}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch6.json:{"validation_loss": 2.9102818965911865, "bleu": 15.900132832450922}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch7.json:{"validation_loss": 2.8136892318725586, "bleu": 17.999634234724873}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch8.json:{"validation_loss": 2.732426404953003, "bleu": 18.937483858593158}
workdir_vocab10000_lr0.02_embd256/eval_results_epoch9.json:{"validation_loss": 2.680779457092285, "bleu": 20.37396734345588}
```

Submission

Please submit the whole `llmsys_s24_hw2` as a zip on canvas. Your code will be automatically compiled and graded with private test cases.