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.

Homework Setup

Clone the repository for the homework:
https://github.com/llmsystem/llmsys_s25_hw2/

PSC Guide

For guidance on using PSC resources, refer to the following document:
PSC Guide

Setting up Your Code

Step 1: Install Requirements

Ensure you have Python 3.8+ installed. Install the required dependencies with the following commands:

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

Step 2: Install miniTorch

Install miniTorch in editable mode:

```
pip install -e .
```

Issue: Aborted (core dumped) due to nvcc and NVIDIA Driver Incompatibility with PyTorch

If you encounter an **"Aborted (core dumped)"** error while running PyTorch, it is likely due to an **incompatibility between `nvcc` and the NVIDIA driver version** used by PyTorch. This happens when: - `nvcc` (the CUDA compiler) is **newer than** the supported CUDA version in the NVIDIA driver. - PyTorch is built for a different CUDA version than the one installed.

To **fix this issue, downgrade CUDA** to match the **highest supported version by your NVIDIA driver** and install the corresponding PyTorch version.

🚀 Solution: Downgrade CUDA and Avoid Core Dump Errors

One of the solutions is to install **CUDA 12.1**, which has compatible PyTorch builds.

1️⃣ Uninstall Any Existing PyTorch Versions

```
pip uninstall torch torchvision torchaudio -y
```

2️⃣ Load CUDA 12.1 Module

Since CUDA 12.1 is available on your system, load it by running:

```
module purge
module load cuda-12.1
```

Verify the CUDA version:

```
nvcc --version
nvidia-smi
```

3️⃣ Install PyTorch for CUDA 12.1

Try **pip** first:

```
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121
```

or with **Conda**:

```
conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c
nvidia
```

4️⃣ Verify Installation

Run the following Python script:

```
import torch
print("PyTorch Version:", torch.__version__)
print("CUDA Available:", torch.cuda.is_available())
print("CUDA Version:", torch.version.cuda)
```

If `torch.cuda.is_available()` returns `False`, **recheck the CUDA installation**.

Step 3: Copy Files from Assignment 1

Copy the following files from Assignment 1 to the specified locations:

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

Note: The suffix for the sentiment file is slightly different: `"_linear"`.

Copy the CUDA kernel file:

- `combine.cu` → `src/combine.cu`

However, do not copy the entire `combine.cu` file. Instead, extract and transfer `only` the implementations of the following functions:

- `MatrixMultiplyKernel`
- `mapKernel`
- `zipKernel`
- `reduceKernel`

Step 4: Changes in Assignment 2

We have made some changes in `combine.cu` and `cuda_kernel_ops.py` for assignment 2 compared with assignment 1 :

- GPU memory allocation, deallocation, and memory copying operations have been relocated from `cuda_kernel_ops.py` to `combine.cu`, covering both host-to-device and device-to-host transfers.
- The datatype for `Tensor._tensor._storage` has been changed from `numpy.float64` to `numpy.float32`.

Step 5: Compile CUDA Kernels

Compile your CUDA kernels by running:

```
bash compile_cuda.sh
```

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 1: Implementing Tensor Functions (20 pts)

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

- **`logsumexp`**
- **`softmax_loss`**

The formula for the softmax loss (softmax + cross-entropy) is:

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

Refer to slide 5 here for more details.

Softmax Loss Function

The input to the softmax loss(softmax + cross entropy) function consists of:

- `logits`: A (minibatch, C) tensor, where each row represents a sample containing raw logits before applying softmax.
- `target`: A (minibatch,) tensor, where each row corresponds to the class of a sample.

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

Note:
The function should return results without setting `reduction=None`. The resulting output shape should be (minibatch,).

Check implementation

After correctly implementing the functions, you should be able to pass tests marked as logsumexp and softmax_loss by running:

```
python -m pytest -l -v -k "test_logsumexp_student"
python -m pytest -l -v -k "test_softmax_loss_student"
```

Problem 2: Implementing Basic Modules (20 pts)

Here are the modules you need to implement in `minitorch/modules_basic.py`:

1. `Linear`: You can use your implementation from Assignment 1 but adapt it slightly to account for the new `backend` argument.
2. `Dropout`: Applies dropout.
   Note: If the flag `self.training` is false, do not zero out any values in the input tensor. To match the autograder seed, please use `np.random.binomial` to generate a mask.
3. `LayerNorm1d`: Applies layer normalization to a 2D tensor.
4. `Embedding`: Maps one-hot word vectors from a dictionary of fixed size to embeddings.

Check implementation

After correctly implementing the functions, you should be able to pass tests marked as linear, dropout, layernorm, and embedding by running:

```
python -m pytest -l -v -k "test_linear_student"
python -m pytest -l -v -k "test_dropout_student"
python -m pytest -l -v -k "test_layernorm_student"
python -m pytest -l -v -k "test_embedding_student"
```

Problem 3: Implementing a Decoder-only Transformer Language Model (20 pts)

Finally, you'll implement the GPT-2 architecture in `minitorch/modules_transformer.py`, utilizing four modules and your earlier work:

- **`MultiHeadAttention`**: Implements masked multi-head attention.
- **`FeedForward`**: Implements the feed-forward operation. [We have implemented this for you.]
- **`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 each K, Q, V tensor formed from X is partitioned into h heads. The self-attention operation is performed for each batch and head, and the output is reshaped to the correct shape. The final output is passed through an output 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 $X \in R^{B \times S \times D}$ 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 $D = h \times D_h$:

   - $X \in R^{B \times S \times D}$ gets projected to $Q, K, V \in R^{B \times S \times D}$ *(Note: We could actually do this with a single layer and split the output into 3.)*
   - $Q \in R^{B \times S \times (h \times D_h)}$ gets unraveled to $Q \in R^{B \times S \times h \times D_h}$
   - $Q \in R^{B \times S \times h \times D_h}$ gets permuted to $Q \in R^{B \times h \times S \times D_h}$

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

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

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

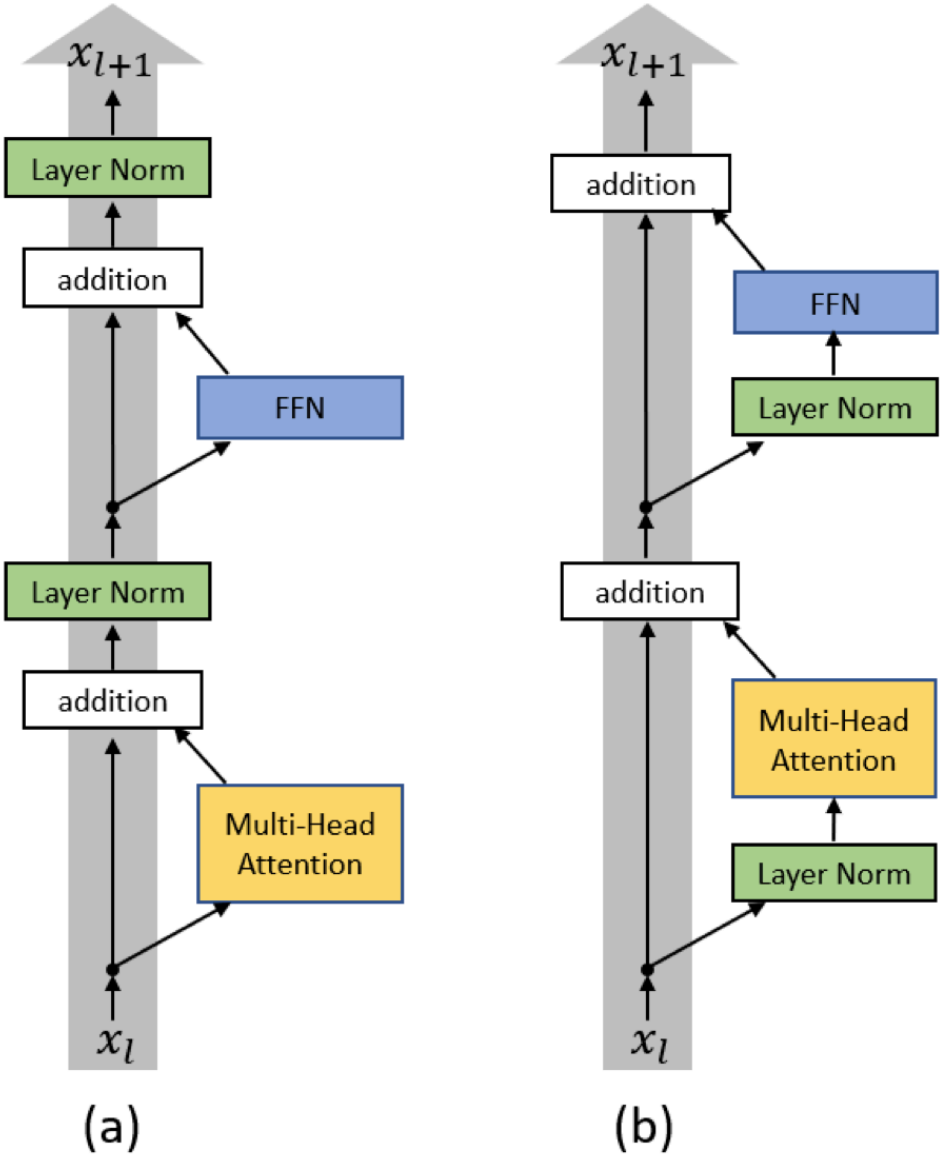
---

FeedForward

We have implemented the feed-forward module for you. The feed-forward module consists of two linear layers with an activation in between. You can go through the implementation for reference.

TransformerLayer

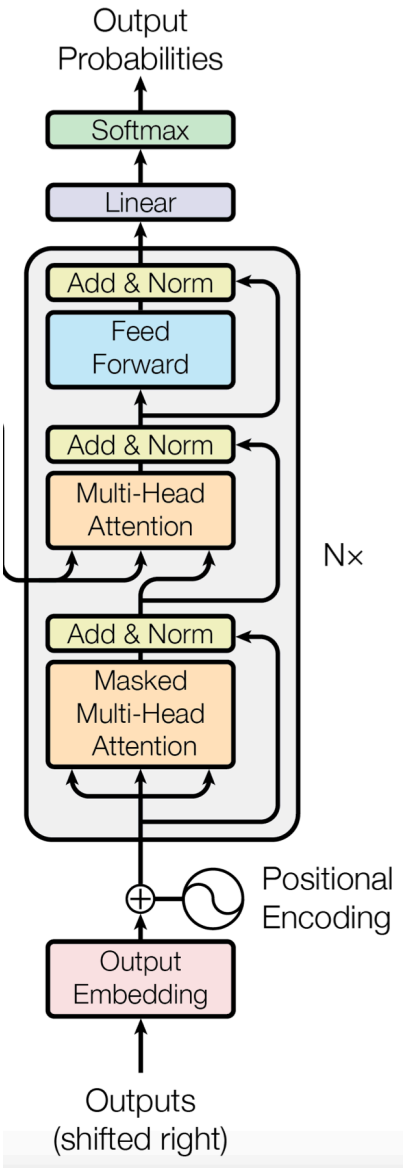
Combine the MultiHeadAttention and FeedForward modules to form one transformer layer. GPT-2 employs the **pre-LN architecture** (pre-layer normalization). Follow the **pre-LN variant** shown below:



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

For more details, refer to On Layer Normalization in the Transformer Architecture.

DecoderLM



(Image from Transformer Decoder)

Combine all components to create the final model. Given an input tensor X with shape $(\text{batch size}, \text{sequence length})$:

1. Retrieve token and positional embeddings for X.
2. Add the embeddings together (Jurafsky and Martin, Chapter 10.1.3) and pass through a dropout layer.
3. Pass the resulting input shape $(\text{batch size}, \text{sequence length}, \text{embedding dimension})$ through all transformer layers.
4. Apply a final LayerNorm.

5. Use a final linear layer to project the hidden dimension to the vocabulary size for inference or loss computation.

Check implementation

After correctly implementing the functions, you should be able to pass tests marked as multihead, transformerlayer, and decoderlm by running:

```
python -m pytest -l -v -k "test_multihead_attention_student"
python -m pytest -l -v -k "test_transformer_layer_1_student"
python -m pytest -l -v -k "test_transformer_layer_2_student"
python -m pytest -l -v -k "test_decoder_lm_student"
```

Problem 4: Machine Translation Pipeline (20 pts)

Implement a training pipeline of machine translation on IWSLT (De-En). You will need to implement the following functions in `project/run_machine_translation.py`:

1. `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.

Note

We recommend you going through the `collate_batch` and `loss_fn` functions in the file to understand the data processing and loss computation steps in the training pipeline.

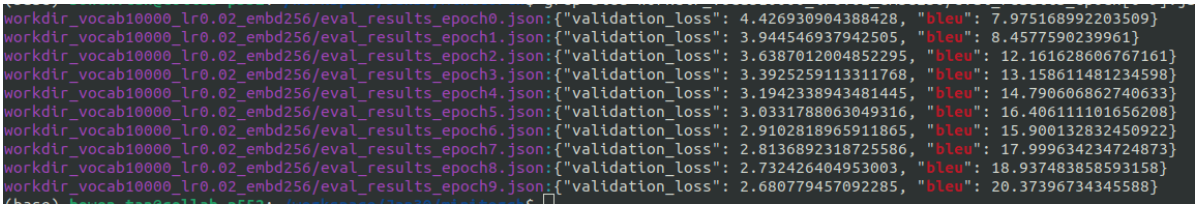
Test Performance

Once all blanks are filled, run:

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

The outputs and BLEU scores will be saved in `./workdir_vocab10000_lr0.02_embd256`. You should get a 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



Submission

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