llmsystem / llmsys_s25_hw2

Code Pull requests Actions Projects Security Insights

Your recovery codes have not been saved in the past year. Make sure you still have them stored somewhere safe by viewing and downloading them again.

View recovery codes

llmsys_s25_hw2 Public

forked from llmsystem/llmsys_s24_hw2

1 Branch 0 Tags Go to file t Go to file + Add file Code

This branch is 2 commits ahead of `llmsystem/llmsys_s24_hw2:main` .

rishi1001 Update requirements.txt 42de644 · 4 days ago

minitorch	added feedforward code	2 weeks ago
project	updated machine_transl...	2 months ago
src	updated pow, tanh	2 weeks ago
tests	update tensor_general	last year
README.md	Update README.md wit...	last year
compile_cuda.sh	hw2 release.	last year
pytest.ini	hw2 release.	last year

About

No description, website, or topics provided.

Readme

Activity

Custom properties

0 stars

0 watching

3 forks

Report repository

Releases

No releases published

Packages

requirements.extr...	Update requirements.ex...	4 days ago
requirements.txt	Update requirements.txt	4 days ago
setup.cfg	hw2 release.	last year
setup.py	hw2 release.	last year

No packages published

Languages

- Python 94.6%
- Cuda 5.4%

README

LLM Systems Assignment 2

Preparation

Install requirements

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

Install minitorch

```
pip install -e .
```

Copy files from Assignment 1

Copy the files below from your Assignment 1 implementation:

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

Note the slight different suffix `_linear`.

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

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

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: Adding Pow and Tanh

We're still missing a few important arithmetic operations for Transformers, namely element-wise (e-wise) power and element-wise tanh.

1. Implement the forward and backward functions for the Tanh and PowerScalar tensor function in `minitorch/tensor_functions.py`

Recall from lecture the structure of minitorch. Calling `.tanh()` on a tensor for example will call a Tensor Function defined in `tensor_functions.py`. These functions are implemented on the CudaKernelBackend, which execute the actual operations on the tensors.

You should utilize `tanh_map` and `pow_scalar_zip`, which have already been added to the `TensorBackend` class, which your CudaKernelOps should then implement.

Don't forget to save the necessary values in the context in the forward pass for your backward pass when calculating the derivatives.

Since we're taking e-wise tanh and power, your gradient calculation should be very simple.

2. Implement the power and tanh function in combine.cu.

Edit the following snippet in your `__device__ float fn` function in `minitorch/combine.cu`

```
case POW: {
    return;
}
case TANH: {
    return;
}
```

Complete the Cuda code to support element-wise power and tanh.

You can look up the relevant mathematical functions here: CUDA Math API

3. Recompile your code with the bash command above.

4. Run the tests below.

The accompanying tests are in `tests/test_tensor_general_student.py`

Run the following to test an individual function eg.

```
python -m pytest -l -v -k "test_pow_1_student"
```

Run the following to test all parts to problem 1.

```
python -m pytest -l -v -m a2_1
```

Adam Optimizer

We provide Adam optimizer for HW2 at optim.py. You should be able to verify Adam's performance by the performance of run_sentiment_linear.py after you've implemented Pow.

```
python project/run_sentiment_linear.py
```

Its validation performance should get above `60%` in 5 epochs.

Problem 2: Implementing Tensor Functions

You will be implementing all the necessary functions and modules to implement a decoder-only transformer model. **PLEASE READ THE IMPLEMENTATION DETAILS SECTION BEFORE STARTING** regarding advice for working with miniTorch.

Implement the GELU activation, logsumexp, one_hot, and softmax_loss functions in `minitorch/nn.py` The accompanying tests are in `tests/test_nn_student.py`

Hints:

- **one_hot**: Since MiniTorch doesn't support slicing/indexing with tensors, you'll want to utilize Numpy's eye function. You can use the .to_numpy() function for MiniTorch Tensors here. (Try to avoid using this in other functions because it's expensive.)
- **softmax_loss**: You'll want to make use of your previously implemented one_hot function.

Run the following to test an individual function eg.

```
python -m pytest -l -v -k "test_gelu_student"
```

Run the following to test all the parts to Problem 2

```
python -m pytest -l -v -m a2_2
```

Problem 3: Implementing Basic Modules

Implement the Embedding, Dropout, Linear, and LayerNorm1d modules in `minitorch/modules_basic_student.py` The accompanying tests are in `tests/test_modules_basic.py`

Updates:

- **Dropout** : Feel free to ignore the 3rd section of the dropout test that employs p=0.5. This is failing unexpectedly because of a random seed problem.
- **Linear** : For people who've cloned the repo already, there is a typo in the initialization of the Linear Layer. Please use the Uniform(-sqrt(1/in_features), sqrt(1/in_features)) to initialize your weights as per PyTorch.

Hints:

- **Embedding**: You'll want to use your one_hot function to easily get embeddings for all your tokens. This function will test both your one_hot function in combination with your Embedding module.
- **Dropout** : Please use numpy.random.binomial with the appropriate parameters and shape for your mask.

Run the following to test an individual function eg.

```
python -m pytest -l -v -k "test_embedding_student"
```

Run the following to test all the parts to Problem 3

```
python -m pytest -l -v -m a2_3
```

Problem 4: Implementing a Decoder-only Transformer

Implement the MultiHeadAttention, FeedForward, TransformerLayer, and DecoderLM module in `minitorch/modules_transfomer_student.py` . The accompanying tests are in `tests/test_modules_transformer.py`

Run the following to test an individual function eg.

```
python -m pytest -l -v -k "test_multihead_attention_student"
```

Run the following to test question 1.1

```
python -m pytest -l -v -m a2_4
```

Problem 5

Implement a machine translation pipeline in `project/run_machine_translation.py`

Once all blanks are filled, run

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

The outputs and bleu scores will be save in `./workdir` . you should get BLEU score around 7 in the first epoch, and around 20 in 10 epochs. Every epoch takes around an hour. You'll get all points if your performance goes beyond 10.

Implementation Details

- **Always add backend**

Always ensure your parameters are initialized with the correct backend (with your CudaKernelOps) to ensure they're run correctly.

- Initializing parameters

When initializing weights in a Module, **always** wrap them with `Parameter(.)`, otherwise miniTorch will not update it.

- Requiring Gradients

When you initialize parameters eg. in LayerNorm, **make sure you set the require_grad_ field** for parameters or tensors for which you'll need to update.

- Using `_from_numpy` functions

We've provided a new set of tensor initialization functions eg. `tensor_from_numpy`. Feel free to use them in functions like one_hot, since minitorch doesn't support slicing, or other times **when you need numpy functions and minitorch doesn't support them**. In this case, you can call `.to_numpy()` and compute your desired operation. However, use this sparingly as this impacts your performance.

- Initializing weights

You'll need to initialize weights from certain distributions. You may want to do so with Numpy's random functions and use tensor_from_numpy to create the corresponding tensor.

- Broadcasting - implicit broadcasting

Unlike numpy or torch, we don't have the `broadcast_to` function available. However, we do have *implicit broadcasting*. eg. given a tensors of shape (2, 2) and (1, 2), you can add the two tensors and the second tensor will be broadcasted to the first tensor using standard broadcasting rules. You will encounter this when building your modules, so keep this in mind if you ever feel like you need `broadcast_to` .

- Contiguous Arrays

Some operations like view require arrays to be contiguous. Sometimes adding a .contiguous() may fix your error.

- No sequential

There is easy way to add sequential modules. **Do not put transformer layers in a list/iterable** and iterate through it in your forward function, because miniTorch will not recognize it

- Batch Matrix Multiplication

We support batched matrix multiplication: Given tensors A and B of shape (a, b, m, n) and (a, b, n, p), A @ B will be of shape (a, b, m, p), whereby matrices are multiplied elementwise across dimensions 0 and 1.