Efficiency

In addition to helping simplify code, tensors provide a basis for speeding up computation. In fact, they are really the only way to efficiently write deep learning code in a slow language like Python. However, nothing we have done so far really makes anything faster than `module0`. This module is focused on taking advantage of tensors to write fast code, first on standard CPUs and then using GPUs.

You need the files from previous assignments, so make sure to pull them over to your new repo.

Guides

- Parallelism
- Matrix Multiply
- CUDA

For this assignment, you will need access to a GPU. We recommend running commands in a Google Colab environment. Follow these instructions for Colab setup.

This assignment does not require you to change the main tensor object. Instead you will only change the core higher-order operations code.

- `fast_ops.py` : Low-level CPU operations
- `cuda_ops.py` : Low-level GPU operations

Task 3.1: Parallelization

Note

This task requires basic familiarity with Numba `prange`.

Be sure to very carefully read the section on parallelism, Numba.

The main backend for our codebase are the three functions `map`, `zip`, and `reduce`. If we can speed up these three, everything we built so far will get better. This exercise asks you to utilize Numba and the `njit` function to speed up these functions. In particular if you can utilize parallelization through `prange` you can get some big wins. Be careful though! Parallelization can lead to funny bugs.

In order to help debug this code, we have created a parallel analytics script for you

```
python project/parallel_check.py
```

Running this script will run NUMBA diagnostics on your functions.

Todo

Complete the following in `minitorch/fast_ops.py` and pass tests marked as `task3_1`.

- Include the diagnostics output from the above script in your README.
- Be sure that the code implements the optimizations specified in the docstrings. We will check for this explicitly.

`minitorch.fast_ops.tensor_map(fn: Callable[[float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]`

NUMBA low_level tensor_map function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel
- All indices use numpy buffers
- When `out` and `in` are stride-aligned, avoid indexing

Parameters:

- **fn** (`Callable[[float], float]`) – function mappings floats-to-floats to apply.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]` – Tensor map function.

`minitorch.fast_ops.tensor_zip(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]`

NUMBA higher-order tensor zip function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel

- All indices use numpy buffers
- When `out`, `a`, `b` are stride-aligned, avoid indexing

Parameters:

- **fn** (`Callable[[float, float], float]`) – function maps two floats to float to apply.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]` – Tensor zip function.

`minitorch.fast_ops.tensor_reduce(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]`

NUMBA higher-order tensor reduce function. See `tensor_ops.py` for description.

Optimizations:

- Main loop in parallel
- All indices use numpy buffers
- Inner-loop should not call any functions or write non-local variables

Parameters:

- **fn** (`Callable[[float, float], float]`) – reduction function mapping two floats to float.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]` – Tensor reduce function

Task 3.2: Matrix Multiplication

Matrix multiplication is key to all the models that we have trained so far. In the last module, we computed matrix multiplication using broadcasting. In this task, we ask you to implement it directly as a function. Do your best to make the function efficient, but for now all that matters is that you correctly produce a multiply function that passes our tests and has some parallelism.

In order to use this function, you will also need to add a new `MatMul` Function to `tensor_functions.py`. We have added a version in the starter code you can copy. You might also find it useful to add a slow broadcasted `matrix_multiply` to `tensor_ops.py` for debugging.

In order to help debug this code, you can use the parallel analytics script.

After you finish this task, you may want to skip to 3.5 and experiment with training on the real task under speed conditions.

Todo

Complete the following function in `minitorch/fast_ops.py`. Pass tests marked as `task3_2`.

- Include the diagnostics output from the above script in your README.
- Be sure that the code implements the optimizations specified in the docstrings. We will check for this explicitly.

`minitorch.fast_ops._tensor_matrix_multiply(out: Storage, out_shape: Shape, out_strides: Strides, a_storage: Storage, a_shape: Shape, a_strides: Strides, b_storage: Storage, b_shape: Shape, b_strides: Strides) -> None`

NUMBA tensor matrix multiply function.

Should work for any tensor shapes that broadcast as long as

```
assert a_shape[-1] == b_shape[-2]
```

Optimizations:

- Outer loop in parallel
- No index buffers or function calls
- Inner loop should have no global writes, 1 multiply.

Parameters:

- **out** (`Storage`) – storage for `out` tensor
- **out_shape** (`Shape`) – shape for `out` tensor
- **out_strides** (`Strides`) – strides for `out` tensor
- **a_storage** (`Storage`) – storage for `a` tensor
- **a_shape** (`Shape`) – shape for `a` tensor
- **a_strides** (`Strides`) – strides for `a` tensor
- **b_storage** (`Storage`) – storage for `b` tensor
- **b_shape** (`Shape`) – shape for `b` tensor
- **b_strides** (`Strides`) – strides for `b` tensor

Returns:

- **None**(`None`) – Fills in `out`

Task 3.3: CUDA Operations

We can do even better than parallelization if we have access to specialized hardware. This task asks you to build a GPU implementation of the backend operations. It will be hard to equal what PyTorch does, but if you are clever you can make these computations really fast (aim for 2x of task 3.1).

Reduce is a particularly challenging function. We provide guides and a simple practice function to help you get started.

Todo

Complete the following functions in `minitorch/cuda_ops.py`, and pass the tests marked as `task3_3`.

`minitorch.cuda_ops.tensor_map(fn: Callable[[float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]`

CUDA higher-order tensor map function. ::

fn_map = tensor_map(fn) fn_map(out, ...)

Parameters:

- **fn** (`Callable[[float], float]`) – function mappings floats-to-floats to apply.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides], None]` – Tensor map function.

`minitorch.cuda_ops.tensor_zip(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]`

CUDA higher-order tensor zipWith (or map2) function ::

fn_zip = tensor_zip(fn) fn_zip(out, ...)

Parameters:

- **fn** (`Callable[[float, float], float]`) – function mappings two floats to float to apply.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides, Storage, Shape, Strides], None]` – Tensor zip function.

`minitorch.cuda_ops._sum_practice(out: Storage, a: Storage, size: int) -> None`

This is a practice sum kernel to prepare for reduce.

Given an array of length n and out of size $n//extblockDIM$ it should sum up each blockDim values into an out cell.

$[a_1, a_2, \ldots, a_{100}]$

|

$[a_1 + \ldots + a_{31}, a_{32} + \ldots + a_{64}, \ldots,]$

Note: Each block must do the sum using shared memory!

Parameters:

- **out** (`Storage`) – storage for `out` tensor.
- **a** (`Storage`) – storage for `a` tensor.
- **size** (`int`) – length of a.

`minitorch.cuda_ops.tensor_reduce(fn: Callable[[float, float], float]) -> Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]`

CUDA higher-order tensor reduce function.

Parameters:

- **fn** (`Callable[[float, float], float]`) – reduction function maps two floats to float.

Returns:

- `Callable[[Storage, Shape, Strides, Storage, Shape, Strides, int], None]` – Tensor reduce function.

Task 3.4: CUDA Matrix Multiplication

Finally we can combine both these approaches and implement CUDA `matmul`. This operation is probably the most important in all of deep learning and is central to making models fast. Again, we first strive for accuracy, but, the faster you can make it, the better.

Implementing matrix multiplication and reduction efficiently is hugely important for many deep learning tasks. Follow the guides provided in class for implementing these functions.

You should document your code to show us that you understand each line. Prove to us that these lead to speed-ups on large matrix operations by making a graph comparing them to naive operations.

Todo

Implement `minitorch/cuda_ops.py` with CUDA, and pass tests marked as `task3_4`. Follow the requirements specified in the docs.

`minitorch.cuda_ops._mm_practice(out: Storage, a: Storage, b: Storage, size: int) -> None`

This is a practice square MM kernel to prepare for matmul.

Given a storage `out` and two storage `a` and `b`. Where we know both are shape [size, size] with strides [size, 1].

Size is always < 32.

Requirements:

- All data must be first moved to shared memory.
- Only read each cell in `a` and `b` once.
- Only write to global memory once per kernel.

Compute

```
for i:
    for j:
         for k:
             out[i, j] += a[i, k] * b[k, j]
```

Parameters:

- **out** (`Storage`) – storage for `out` tensor.
- **a** (`Storage`) – storage for `a` tensor.
- **b** (`Storage`) – storage for `b` tensor.
- **size** (`int`) – size of the square

`minitorch.cuda_ops._tensor_matrix_multiply(out: Storage, out_shape: Shape, out_strides: Strides, out_size: int, a_storage: Storage, a_shape: Shape, a_strides: Strides, b_storage: Storage, b_shape: Shape, b_strides: Strides) -> None`

CUDA tensor matrix multiply function.

Requirements:

- All data must be first moved to shared memory.
- Only read each cell in `a` and `b` once.
- Only write to global memory once per kernel.

Should work for any tensor shapes that broadcast as long as ::

```
assert a_shape[-1] == b_shape[-2]
```

Returns:

- **None**(`None`) – Fills in `out`

Task 3.5: Training

If your code works, you should now be able to move on to the tensor training script in `project/run_fast_tensor.py`. This code is the same basic training setup as `module2`, but now utilizes your fast tensor code. We have left the `matmul` layer blank for you to implement with your tensor code.

Here is the command ::

```
python run_fast_tensor.py --BACKEND gpu --HIDDEN 100 --DATASET split --RATE
0.05
python run_fast_tensor.py --BACKEND cpu --HIDDEN 100 --DATASET split --RATE
0.05
```

Todo

- Implement the missing functions in `project/run_fast_tensor.py`. These should do exactly the same thing as the corresponding functions in `project/run_tensor.py`, but now use the faster backend
- Train a tensor model and add your results for all dataset to the README.
- Run a bigger model and record the time per epoch reported by the trainer.

Train a tensor model and add your results for all three dataset to the README. Also record the time per epoch reported by the trainer. (As a reference, our parallel implementation gave a 10x speedup). On a standard Colab GPU setup, aim for you CPU to get below 2 seconds per epoch and GPU to be below 1 second per epoch. (With some cleverness you can do much better.)