Module 3.1 - Efficiency

Motivation

- NLP tools

Today’s Class

Context

- We now have a pytorch
- All wrappers around `ops`
- Need to make `ops` fast

Goal

Optimize:

- map
- zip
- reduce

Code

Example map ::

```
for i in range(len(out)):
       count(i, out_shape, out_index)
       broadcast_index(out_index, out_shape, in_shape, in_index)
       o = index_to_position(out_index, out_strides)
       j = index_to_position(in_index, in_strides)
       out[o] = fn(in_storage[j])
```

Why are Python (and friends) ”slow”?

- Function calls
- Types
- Loops

Function Calls

- Function calls are not free
- Checks for args, special keywords andm lists
- Methods check for overrides and class inheritance

Types

Critical code

```
out[o] = in_storage[j] + 3
```

- Doesn't know type of `in_storage[j]`
- May need to coerce 3 to float or raise error
- May even call `__add__` or `__ladd__`!

Loops

- Loops are always run as is.
- Can't combine similar loops or pull out constant computation.
- Very hard to run anything in parallel.

Other

Many other slow things...

- Lists
- Classes
- Magic of all kind

Fast Math

Numba

- Python library for speeding up numerical python
- API: Higher-order functions to produce fast mathmatical code
- Numba

How does it work?

Work

```
def my_code(x, y):
     for i in range(100):
         x[i] = y + 20
  ...
  my_code(x, y)
  fast_my_code = numba.njit()(my_code)
  fast_my_code(x, y)
  fast_my_code(x, y)
```

Notebook

[Colab Notebook]

Terminology : JIT Compiler

- Just-in-time
- Waits until you call a function to compile it
- Specializes code based on the argument types given.

Terminology : LLVM

- Underlying compiler framework to generate code
- Used by many different languages (C++, Swift, Rust, ...)
- Generates efficient machine code for the system

What do we lose?

- `njit` will fail for many python operations
- No lists, classes, python functions allowed
- Any different types will cause recompilation

Strategy

- Use Python for general operations
- Use Numba for the core tensor ops
- Allow users to add new Numba functions

Code Transformation

Transform

```
def my_code(x, y):
    for i in prange(100):
        x[i] = y + 20
 ...
 my_code(x, y)
 fast_my_code = numba.njit(parallel=True)(my_code)
 fast_my_code(x, y)
 fast_my_code(x, y)
```

Notebook

Colab Notebook

Parallel

Parallel

- Run code on multiple threads
- Particularly suited for map / zip
- Baby steps towards GPU

Parallel Range

- Replace `for` loops with parallel version
- Tells compiler it can run in any order
- Be careful! Ideally these loops don't change anything

Code Transformation

Transform ::

```
def my_code(x, y):
     for i in prange(100):
         x[i] = y + 20
  ...
  my_code(x, y)
  fast_my_code = numba.njit(parallel=True)(my_code)
  fast_my_code(x, y)
  fast_my_code(x, y)
```

Nondeterminism

- No guarantee on ordering
- Need to be careful with reductions
- Speedups will depend on system

Parallel Bugs

- Warning! Nasty bugs
- Tests failing randomly
- Crashes due to out-of-bounds

Parallel Diagnostics

- Diagnostics give parallel compilation
- Useful to see if you are getting benefits