Note

Go to the end to download the full example code.

Low-Memory Dropout

In this tutorial, you will write a memory-efficient implementation of dropout whose state will be composed of a single int32 seed. This differs from more traditional implementations of dropout, whose state is generally composed of a bit mask tensor of the same shape as the input.

In doing so, you will learn about:

- The limitations of naive implementations of Dropout with PyTorch.
- Parallel pseudo-random number generation in Triton.

Baseline

The *dropout* operator was first introduced in [SRIVASTAVA2014] as a way to improve the performance of deep neural networks in low-data regime (i.e. regularization).

It takes a vector as input and produces a vector of the same shape as output. Each scalar in the output has a probability p of being changed to zero and otherwise it is copied from the input. This forces the network to perform well even when only $1 - p$ scalars from the input are available.

At evaluation time we want to use the full power of the network so we set $p = 0$. Naively this would increase the norm of the output (which can be a bad thing, e.g. it can lead to artificial decrease in the output softmax temperature). To prevent this we multiply the output by $\frac{1}{1-p}$, which keeps the norm consistent regardless of the dropout probability.

Let's first take a look at the baseline implementation.

```
import tabulate
import torch

import triton
import triton.language as tl

DEVICE = triton.runtime.driver.active.get_active_torch_device()


@triton.jit
def _dropout(
    x_ptr,  # pointer to the input
    x_keep_ptr,  # pointer to a mask of 0s and 1s
    output_ptr,  # pointer to the output
    n_elements,  # number of elements in the `x` tensor
    p,  # probability that an element of `x` is changed to zero
    BLOCK_SIZE: tl.constexpr,
):
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements
    # Load data
    x = tl.load(x_ptr + offsets, mask=mask)
    x_keep = tl.load(x_keep_ptr + offsets, mask=mask)
    # The line below is the crucial part, described in the paragraph above!
    output = tl.where(x_keep, x / (1 - p), 0.0)
    # Write-back output
    tl.store(output_ptr + offsets, output, mask=mask)


def dropout(x, x_keep, p):
    output = torch.empty_like(x)
    assert x.is_contiguous()
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    _dropout[grid](x, x_keep, output, n_elements, p, BLOCK_SIZE=1024)
    return output


# Input tensor
x = torch.randn(size=(10, ), device=DEVICE)
# Dropout mask
p = 0.5
x_keep = (torch.rand(size=(10, ), device=DEVICE) > p).to(torch.int32)
#
output = dropout(x, x_keep=x_keep, p=p)
print(tabulate.tabulate([
    ["input"] + x.tolist(),
    ["keep mask"] + x_keep.tolist(),
    ["output"] + output.tolist(),
]))
```

Out:

```
/home/runner/.work/triton/triton/python/triton/language/semantic.
  warnings.warn(
-----------  ----------  -------  --------  --------  --------  -------  ----
input        -0.940469   0.17792  0.529538  0.13197   0.135063  1.640
keep mask    0           0        0         0         0         1
output       0           0        0         0         0         3.281
-----------  ----------  -------  --------  --------  --------  -------  ----
```

Seeded dropout

The above implementation of dropout works fine, but it can be a bit awkward to deal with. Firstly, we need to store the dropout mask for backpropagation. Secondly, dropout state management can get very tricky when using recompute/checkpointing (e.g. see all the notes about *preserve_rng_state* in https://pytorch.org/docs/stable/checkpoint.html). In this tutorial we'll describe an alternative implementation that (1) has a smaller memory footprint; (2) requires less data movement; and (3) simplifies the management of persisting randomness across multiple invocations of the kernel.

Pseudo-random number generation in Triton is simple! In this tutorial we will use the `triton.language.rand` function which generates a block of uniformly distributed `float32` values in [0, 1), given a seed and a block of `int32` offsets. But if you need it, Triton also provides other random number generation strategies.

Note

Triton's implementation of PRNG is based on the Philox algorithm (described on [SALMON2011]).

Let's put it all together.

```
@triton.jit
def _seeded_dropout(
    x_ptr,
    output_ptr,
    n_elements,
    p,
    seed,
    BLOCK_SIZE: tl.constexpr,
):
    # compute memory offsets of elements handled by this instance
    pid = tl.program_id(axis=0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # load data from x
    mask = offsets < n_elements
    x = tl.load(x_ptr + offsets, mask=mask)
    # randomly prune it
    random = tl.rand(seed, offsets)
    x_keep = random > p
    # write-back
    output = tl.where(x_keep, x / (1 - p), 0.0)
    tl.store(output_ptr + offsets, output, mask=mask)


def seeded_dropout(x, p, seed):
    output = torch.empty_like(x)
    assert x.is_contiguous()
    n_elements = x.numel()
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']),)
    _seeded_dropout[grid](x, output, n_elements, p, seed, BLOCK_SIZE=16
    return output


x = torch.randn(size=(10, ), device=DEVICE)
# Compare this to the baseline - dropout mask is never instantiated!
output = seeded_dropout(x, p=0.5, seed=123)
output2 = seeded_dropout(x, p=0.5, seed=123)
output3 = seeded_dropout(x, p=0.5, seed=512)

print(
    tabulate.tabulate([
        ["input"] + x.tolist(),
        ["output (seed = 123)"] + output.tolist(),
        ["output (seed = 123)"] + output2.tolist(),
        ["output (seed = 512)"] + output3.tolist(),
    ]))
```

Out:

```
-----------------  -------  ---------  ---------  -------  ----
input              1.48333  -0.239537  -0.640795  1.62631  0.26
output (seed = 123)  0        -0.479074  0          0        0
output (seed = 123)  0        -0.479074  0          0        0
output (seed = 512)  0         0         -1.28159   3.25261  0
-----------------  -------  ---------  ---------  -------  ----
```

Et Voilà! We have a triton kernel that applies the same dropout mask provided the seed is the same! If you'd like explore further applications of pseudorandomness in GPU programming, we encourage you to explore the *python/triton/language/random.py*!

Exercises

1. Extend the kernel to operate over a matrix and use a vector of seeds - one per row.
2. Add support for striding.
3. (challenge) Implement a kernel for sparse Johnson-Lindenstrauss transform which generates the projection matrix on the fly each time using a seed.

References

[SALMON2011] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw, "Parallel Random Numbers: As Easy as 1, 2, 3", 2011

[SRIVASTAVA2014] Nitish Srivastava and Geoffrey Hinton and Alex Krizhevsky and Ilya Sutskever and Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", JMLR 2014

Total running time of the script: (0 minutes 0.740 seconds)

Download Jupyter notebook: 04-low-memory-dropout.ipynb

Download Python source code: 04-low-memory-dropout.py

Download zipped: 04-low-memory-dropout.zip