lecture_02.py

```python
from execute_util import text, link, image
from facts import a100_flop_per_sec, h100_flop_per_sec
import torch.nn.functional as F
import timeit
import torch
from typing import Iterable
from torch import nn
import numpy as np
from lecture_util import article_link
from jaxtyping import Float
from einops import rearrange, einsum, reduce
from references import zero_2019


def main():
```

Last lecture: overview, tokenization

Overview of this lecture:

- We will discuss all the **primitives** needed to train a model.
- We will go bottom-up from tensors to models to optimizers to the training loop.
- We will pay close attention to efficiency (use of **resources**).

In particular, we will account for two types of resources:

- Memory (GB)
- Compute (FLOPs)

```python
motivating_questions()
```

We will not go over the Transformer.
There are excellent expositions:
Assignment 1 handout
Mathematical description
Illustrated Transformer
Illustrated GPT-2
Instead, we'll work with simpler models.

What knowledge to take away:

- Mechanics: straightforward (just PyTorch)
- Mindset: resource accounting (remember to do it)
- Intuitions: broad strokes (no large models)

Memory accounting

```python
tensors_basics()
tensors_memory()
```

Compute accounting

```python
tensors_on_gpus()
tensor_operations()
tensor_einops()
tensor_operations_flops()
gradients_basics()
gradients_flops()
```

Models

```python
module_parameters()
custom_model()
```

Training loop and best practices

```
    note_about_randomness()
    data_loading()

    optimizer()
    train_loop()
    checkpointing()
    mixed_precision_training()


def motivating_questions():
```

Let's do some napkin math.

Question: How long would it take to train a 70B parameter model on 15T tokens on 1024 H100s?

```
    total_flops = 6 * 70e9 * 15e12  # @inspect total_flops
    assert h100_flop_per_sec == 1979e12 / 2
    mfu = 0.5
    flops_per_day = h100_flop_per_sec * mfu * 1024 * 60 * 60 * 24  # @inspect flops_per_day
    days = total_flops / flops_per_day  # @inspect days
```

Question: What's the largest model that can you can train on 8 H100s using AdamW (naively)?

```
    h100_bytes = 80e9  # @inspect h100_bytes
    bytes_per_parameter = 4 + 4 + (4 + 4)  # parameters, gradients, optimizer state  @inspect bytes_per_parameter
    num_parameters = (h100_bytes * 8) / bytes_per_parameter  # @inspect num_parameters
```

Caveat 1: we are naively using float32 for parameters and gradients. We could also use bf16 for parameters and gradients (2 + 2) and keep an extra float32 copy of the parameters (4). This doesn't save memory, but is faster. [Rajbhandari+ 2019]

Caveat 2: activations are not accounted for (depends on batch size and sequence length).

This is a rough back-of-the-envelope calculation.

```
def tensors_basics():
```

Tensors are the basic building block for storing everything: parameters, gradients, optimizer state, data, activations.

[PyTorch docs on tensors]

You can create tensors in multiple ways:

```
    x = torch.tensor([[1., 2, 3], [4, 5, 6]])  # @inspect x
    x = torch.zeros(4, 8)  # 4x8 matrix of all zeros @inspect x
    x = torch.ones(4, 8)  # 4x8 matrix of all ones @inspect x
    x = torch.randn(4, 8)  # 4x8 matrix of iid Normal(0, 1) samples @inspect x
```

Allocate but don't initialize the values:

```
    x = torch.empty(4, 8)  # 4x8 matrix of uninitialized values @inspect x
```

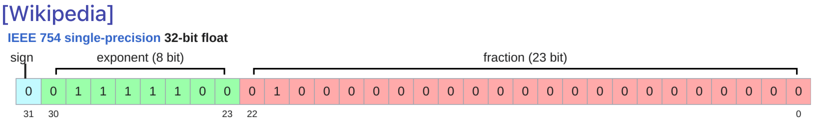
...because you want to use some custom logic to set the values later

```
    nn.init.trunc_normal_(x, mean=0, std=1, a=-2, b=2)  # @inspect x


def tensors_memory():
```

Almost everything (parameters, gradients, activations, optimizer states) are stored as floating point numbers.

float32

[Wikipedia]



The float32 data type (also known as fp32 or single precision) is the default.

Traditionally, in scientific computing, float32 is the baseline; you could use double precision (float64) in some cases.

In deep learning, you can be a lot sloppier.

Let's examine memory usage of these tensors.

Memory is determined by the (i) number of values and (ii) data type of each value.

```python
x = torch.zeros(4, 8)  # @inspect x
assert x.dtype == torch.float32  # Default type
assert x.numel() == 4 * 8
assert x.element_size() == 4  # Float is 4 bytes
assert get_memory_usage(x) == 4 * 8 * 4  # 128 bytes
```

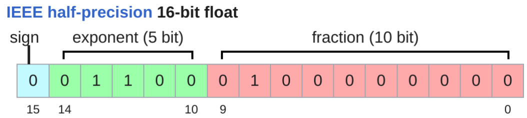
One matrix in the feedforward layer of GPT-3:

```python
assert get_memory_usage(torch.empty(12288 * 4, 12288)) == 2304 * 1024 * 1024  # 2.3 GB
```

...which is a lot!

float16

[Wikipedia]



The float16 data type (also known as fp16 or half precision) cuts down the memory.

```python
x = torch.zeros(4, 8, dtype=torch.float16)  # @inspect x
assert x.element_size() == 2
```

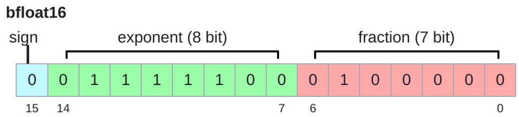
However, the dynamic range (especially for small numbers) isn't great.

```python
x = torch.tensor([1e-8], dtype=torch.float16)  # @inspect x
assert x == 0  # Underflow!
```

If this happens when you train, you can get instability.

bfloat16

[Wikipedia]



Google Brain developed bfloat (brain floating point) in 2018 to address this issue.
bfloat16 uses the same memory as float16 but has the same dynamic range as float32!
The only catch is that the resolution is worse, but this matters less for deep learning.

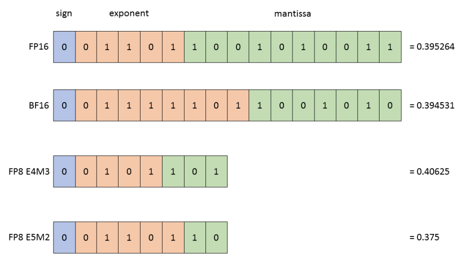
```python
x = torch.tensor([1e-8], dtype=torch.bfloat16)  # @inspect x
assert x != 0  # No underflow!
```

Let's compare the dynamic ranges and memory usage of the different data types:

```python
float32_info = torch.finfo(torch.float32)  # @inspect float32_info
float16_info = torch.finfo(torch.float16)  # @inspect float16_info
bfloat16_info = torch.finfo(torch.bfloat16)  # @inspect bfloat16_info
```

fp8

In 2022, FP8 was standardized, motivated by machine learning workloads.
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html



H100s support two variants of FP8: E4M3 (range [-448, 448]) and E5M2 ([-57344, 57344]).
Reference: [Micikevicius+ 2022]

Implications on training:

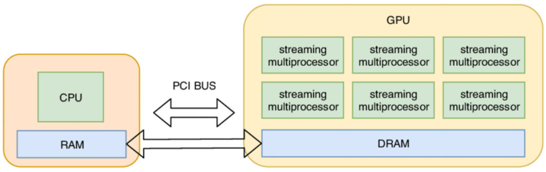
- Training with float32 works, but requires lots of memory.
- Training with fp8, float16 and even bfloat16 is risky, and you can get instability.
- Solution (later): use mixed precision training, see mixed_precision_training

```
def tensors_on_gpus():
```

By default, tensors are stored in CPU memory.

```
    x = torch.zeros(32, 32)
    assert x.device == torch.device("cpu")
```

However, in order to take advantage of the massive parallelism of GPUs, we need to move them to GPU memory.



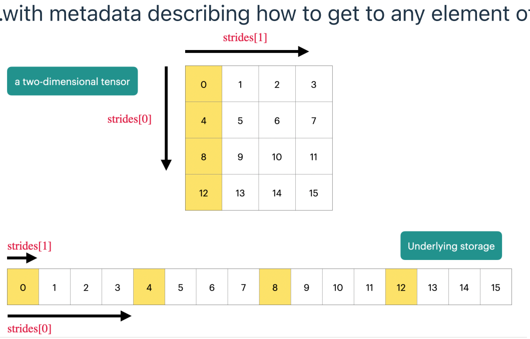
Let's first see if we have any GPUs.

```
    if not torch.cuda.is_available():
        return

    num_gpus = torch.cuda.device_count()  # @inspect num_gpus
    for i in range(num_gpus):
        properties = torch.cuda.get_device_properties(i)  # @inspect properties

    memory_allocated = torch.cuda.memory_allocated()  # @inspect memory_allocated

    text("Move the tensor to GPU memory (device 0).")
    y = x.to("cuda:0")
    assert y.device == torch.device("cuda", 0)

    text("Or create a tensor directly on the GPU:")
    z = torch.zeros(32, 32, device="cuda:0")

    new_memory_allocated = torch.cuda.memory_allocated()  # @inspect new_memory_allocated
    memory_used = new_memory_allocated - memory_allocated  # @inspect memory_used
    assert memory_used == 2 * (32 * 32 * 4)  # 2 32x32 matrices of 4-byte floats



def tensor_operations():
```

Most tensors are created from performing operations on other tensors.
Each operation has some memory and compute consequence.

```
    tensor_storage()
    tensor_slicing()
    tensor_elementwise()
    tensor_matmul()


def tensor_storage():
```

What are tensors in PyTorch?
PyTorch tensors are pointers into allocated memory
...with metadata describing how to get to any element of the tensor.



[PyTorch docs]

```
    x = torch.tensor([
        [0., 1, 2, 3],
```

```
        [4, 5, 6, 7],
        [8, 9, 10, 11],
        [12, 13, 14, 15],
    ])
```

To go to the next row (dim 0), skip 4 elements in storage.

```
    assert x.stride(0) == 4
```

To go to the next column (dim 1), skip 1 element in storage.

```
    assert x.stride(1) == 1
```

To find an element:

```
    r, c = 1, 2
    index = r * x.stride(0) + c * x.stride(1)  # @inspect index
    assert index == 6


def tensor_slicing():
    x = torch.tensor([[1., 2, 3], [4, 5, 6]])  # @inspect x
```

Many operations simply provide a different **view** of the tensor.
This does not make a copy, and therefore mutations in one tensor affects the other.

Get row 0:

```
    y = x[0]  # @inspect y
    assert torch.equal(y, torch.tensor([1., 2, 3]))
    assert same_storage(x, y)
```

Get column 1:

```
    y = x[:, 1]  # @inspect y
    assert torch.equal(y, torch.tensor([2, 5]))
    assert same_storage(x, y)
```

View 2x3 matrix as 3x2 matrix:

```
    y = x.view(3, 2)  # @inspect y
    assert torch.equal(y, torch.tensor([[1, 2], [3, 4], [5, 6]]))
    assert same_storage(x, y)
```

Transpose the matrix:

```
    y = x.transpose(1, 0)  # @inspect y
    assert torch.equal(y, torch.tensor([[1, 4], [2, 5], [3, 6]]))
    assert same_storage(x, y)
```

Check that mutating x also mutates y.

```
    x[0][0] = 100  # @inspect x, @inspect y
    assert y[0][0] == 100
```

Note that some views are non-contiguous entries, which means that further views aren't possible.

```
    x = torch.tensor([[1., 2, 3], [4, 5, 6]])  # @inspect x
    y = x.transpose(1, 0)  # @inspect y
    assert not y.is_contiguous()
    try:
        y.view(2, 3)
        assert False
    except RuntimeError as e:
        assert "view size is not compatible with input tensor's size and stride" in str(e)
```

One can enforce a tensor to be contiguous first:

```
    y = x.transpose(1, 0).contiguous().view(2, 3)  # @inspect y
    assert not same_storage(x, y)
```

Views are free, copying take both (additional) memory and compute.

```
def tensor_elementwise():
```

These operations apply some operation to each element of the tensor
...and return a (new) tensor of the same shape.

```
    x = torch.tensor([1, 4, 9])
    assert torch.equal(x.pow(2), torch.tensor([1, 16, 81]))
    assert torch.equal(x.sqrt(), torch.tensor([1, 2, 3]))
    assert torch.equal(x.rsqrt(), torch.tensor([1, 1 / 2, 1 / 3]))  # i -> 1/sqrt(x_i)

    assert torch.equal(x + x, torch.tensor([2, 8, 18]))
    assert torch.equal(x * 2, torch.tensor([2, 8, 18]))
    assert torch.equal(x / 0.5, torch.tensor([2, 8, 18]))
```

`triu` takes the upper triangular part of a matrix.

```
    x = torch.ones(3, 3).triu()  # @inspect x
    assert torch.equal(x, torch.tensor([
        [1, 1, 1],
        [0, 1, 1],
        [0, 0, 1]],
    ))
```

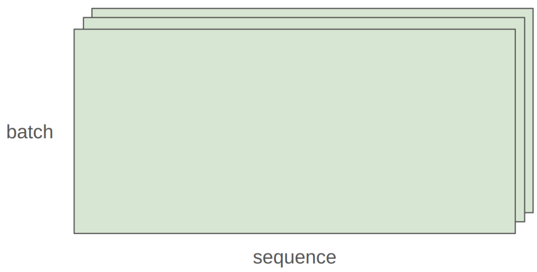
This is useful for computing an causal attention mask, where M[i, j] is the contribution of i to j.

```
def tensor_matmul():
```

Finally, the bread and butter of deep learning: matrix multiplication.

```
    x = torch.ones(16, 32)
    w = torch.ones(32, 2)
    y = x @ w
    assert y.size() == torch.Size([16, 2])
```

In general, we perform operations for every example in a batch and token in a sequence.




```
    x = torch.ones(4, 8, 16, 32)
    w = torch.ones(32, 2)
    y = x @ w
    assert y.size() == torch.Size([4, 8, 16, 2])
```

In this case, we iterate over values of the first 2 dimensions of `x` and multiply by `w`.

```
def tensor_einops():
    einops_motivation()
```

Einops is a library for manipulating tensors where dimensions are named.
It is inspired by Einstein summation notation (Einstein, 1916).
[Einops tutorial]

```
    jaxtyping_basics()
    einops_einsum()
    einops_reduce()
    einops_rearrange()


def einops_motivation():
```

Traditional PyTorch code:

```
    x = torch.ones(2, 2, 3)  # batch, sequence, hidden  @inspect x
    y = torch.ones(2, 2, 3)  # batch, sequence, hidden  @inspect y
    z = x @ y.transpose(-2, -1)  # batch, sequence, sequence  @inspect z
```

Easy to mess up the dimensions (what is -2, -1?)...

```
def jaxtyping_basics():
```

How do you keep track of tensor dimensions?

Old way:

```
x = torch.ones(2, 2, 1, 3)  # batch seq heads hidden  @inspect x
```

New (jaxtyping) way:

```
x: Float[torch.Tensor, "batch seq heads hidden"] = torch.ones(2, 2, 1, 3)  # @inspect x
```

Note: this is just documentation (no enforcement).

```
def einops_einsum():
```

Einsum is generalized matrix multiplication with good bookkeeping.

Define two tensors:

```
x: Float[torch.Tensor, "batch seq1 hidden"] = torch.ones(2, 3, 4)  # @inspect x
y: Float[torch.Tensor, "batch seq2 hidden"] = torch.ones(2, 3, 4)  # @inspect y
```

Old way:

```
z = x @ y.transpose(-2, -1)  # batch, sequence, sequence  @inspect z
```

New (einops) way:

```
z = einsum(x, y, "batch seq1 hidden, batch seq2 hidden -> batch seq1 seq2")  # @inspect z
```

Dimensions that are not named in the output are summed over.

Or can use `...` to represent broadcasting over any number of dimensions:

```
z = einsum(x, y, "... seq1 hidden, ... seq2 hidden -> ... seq1 seq2")  # @inspect z
```

```
def einops_reduce():
```

You can reduce a single tensor via some operation (e.g., sum, mean, max, min).

```
x: Float[torch.Tensor, "batch seq hidden"] = torch.ones(2, 3, 4)  # @inspect x
```

Old way:

```
y = x.mean(dim=-1)  # @inspect y
```

New (einops) way:

```
y = reduce(x, "... hidden -> ...", "sum")  # @inspect y
```

```
def einops_rearrange():
```

Sometimes, a dimension represents two dimensions

...and you want to operate on one of them.

```
x: Float[torch.Tensor, "batch seq total_hidden"] = torch.ones(2, 3, 8)  # @inspect x
```

...where `total_hidden` is a flattened representation of `heads * hidden1`

```
w: Float[torch.Tensor, "hidden1 hidden2"] = torch.ones(4, 4)
```

Break up `total_hidden` into two dimensions (`heads` and `hidden1`):

```
x = rearrange(x, "... (heads hidden1) -> ... heads hidden1", heads=2)  # @inspect x
```

Perform the transformation by `w`:

```
x = einsum(x, w, "... hidden1, hidden1 hidden2 -> ... hidden2")  # @inspect x
```

Combine `heads` and `hidden2` back together:

```
x = rearrange(x, "... heads hidden2 -> ... (heads hidden2)")  # @inspect x
```

```
def tensor_operations_flops():
```

Having gone through all the operations, let us examine their computational cost.

A floating-point operation (FLOP) is a basic operation like addition (x + y) or multiplication (x y).

Two terribly confusing acronyms (pronounced the same!):

- FLOPs: floating-point operations (measure of computation done)
- FLOP/s: floating-point operations per second (also written as FLOPS), which is used to measure the speed of hardware.

Intuitions

Training GPT-3 (2020) took 3.14e23 FLOPs. [article]
Training GPT-4 (2023) is speculated to take 2e25 FLOPs [article]
US executive order: any foundation model trained with >= 1e26 FLOPs must be reported to the government (revoked in 2025)

A100 has a peak performance of 312 teraFLOP/s [spec]

```
assert a100_flop_per_sec == 312e12
```

H100 has a peak performance of 1979 teraFLOP/s with sparsity, 50% without [spec]

```
assert h100_flop_per_sec == 1979e12 / 2
```

8 H100s for 2 weeks:

```
total_flops = 8 * (60 * 60 * 24 * 7) * h100_flop_per_sec  # @inspect total_flops
```

Linear model

As motivation, suppose you have a linear model.

- We have n points
- Each point is d-dimsional
- The linear model maps each d-dimensional vector to a k outputs

```
if torch.cuda.is_available():
    B = 16384  # Number of points
    D = 32768  # Dimension
    K = 8192   # Number of outputs
else:
    B = 1024
    D = 256
    K = 64

device = get_device()
x = torch.ones(B, D, device=device)
w = torch.randn(D, K, device=device)
y = x @ w
```

We have one multiplication (x[i][j] * w[j][k]) and one addition per (i, j, k) triple.

```
actual_num_flops = 2 * B * D * K  # @inspect actual_num_flops
```

FLOPs of other operations

- Elementwise operation on a m x n matrix requires O(m n) FLOPs.
- Addition of two m x n matrices requires m n FLOPs.

In general, no other operation that you'd encounter in deep learning is as expensive as matrix multiplication for large enough matrices.

Interpretation:

- B is the number of data points
- (D K) is the number of parameters
- FLOPs for forward pass is 2 (# tokens) (# parameters)

It turns out this generalizes to Transformers (to a first-order approximation).

How do our FLOPs calculations translate to wall-clock time (seconds)?
Let us time it!

```
actual_time = time_matmul(x, w)  # @inspect actual_time
actual_flop_per_sec = actual_num_flops / actual_time  # @inspect actual_flop_per_sec
```

Each GPU has a specification sheet that reports the peak performance.

- A100 [spec]

- H100 [spec]

Note that the FLOP/s depends heavily on the data type!

```
promised_flop_per_sec = get_promised_flop_per_sec(device, x.dtype)  # @inspect promised_flop_per_sec
```

Model FLOPs utilization (MFU)

Definition: (actual FLOP/s) / (promised FLOP/s) [ignore communication/overhead]

```
mfu = actual_flop_per_sec / promised_flop_per_sec  # @inspect mfu
```

Usually, MFU of >= 0.5 is quite good (and will be higher if matmuls dominate)

Let's do it with bfloat16:

```
x = x.to(torch.bfloat16)
w = w.to(torch.bfloat16)
bf16_actual_time = time_matmul(x, w)  # @inspect bf16_actual_time
bf16_actual_flop_per_sec = actual_num_flops / bf16_actual_time  # @inspect bf16_actual_flop_per_sec
bf16_promised_flop_per_sec = get_promised_flop_per_sec(device, x.dtype)  # @inspect bf16_promised_flop_per_sec
bf16_mfu = bf16_actual_flop_per_sec / bf16_promised_flop_per_sec  # @inspect bf16_mfu
```

Note: comparing bfloat16 to float32, the actual FLOP/s is higher.
The MFU here is rather low, probably because the promised FLOPs is a bit optimistic.

Summary

- Matrix multiplications dominate: (2 m n p) FLOPs
- FLOP/s depends on hardware (H100 >> A100) and data type (bfloat16 >> float32)
- Model FLOPs utilization (MFU): (actual FLOP/s) / (promised FLOP/s)

```
def gradients_basics():
```

So far, we've constructed tensors (which correspond to either parameters or data) and passed them through operations (forward).
Now, we're going to compute the gradient (backward).

As a simple example, let's consider the simple linear model:
y = 0.5 (x * w - 5)^2

Forward pass: compute loss

```
x = torch.tensor([1., 2, 3])
w = torch.tensor([1., 1, 1], requires_grad=True)  # Want gradient
pred_y = x @ w
loss = 0.5 * (pred_y - 5).pow(2)
```

Backward pass: compute gradients

```
loss.backward()
assert loss.grad is None
assert pred_y.grad is None
assert x.grad is None
assert torch.equal(w.grad, torch.tensor([1, 2, 3]))
```

```
def gradients_flops():
```

Let us do count the FLOPs for computing gradients.

Revisit our linear model

```
if torch.cuda.is_available():
    B = 16384  # Number of points
    D = 32768  # Dimension
    K = 8192   # Number of outputs
else:
    B = 1024
    D = 256
    K = 64

device = get_device()
x = torch.ones(B, D, device=device)
```

```
w1 = torch.randn(D, D, device=device, requires_grad=True)
w2 = torch.randn(D, K, device=device, requires_grad=True)
```

Model: x --w1--> h1 --w2--> h2 -> loss

```
h1 = x @ w1
h2 = h1 @ w2
loss = h2.pow(2).mean()
```

Recall the number of forward FLOPs: tensor_operations_flops

- Multiply x[i][j] * w1[j][k]
- Add to h1[i][k]
- Multiply h1[i][j] * w2[j][k]
- Add to h2[i][k]

```
num_forward_flops = (2 * B * D * D) + (2 * B * D * K)  # @inspect num_forward_flops
```

How many FLOPs is running the backward pass?

```
h1.retain_grad()  # For debugging
h2.retain_grad()  # For debugging
loss.backward()
```

Recall model: x --w1--> h1 --w2--> h2 -> loss

- h1.grad = d loss / d h1
- h2.grad = d loss / d h2
- w1.grad = d loss / d w1
- w2.grad = d loss / d w2

Focus on the parameter w2.
Invoke the chain rule.

```
num_backward_flops = 0  # @inspect num_backward_flops
```

w2.grad[j,k] = sum_i h1[i,j] * h2.grad[i,k]

```
assert w2.grad.size() == torch.Size([D, K])
assert h1.size() == torch.Size([B, D])
assert h2.grad.size() == torch.Size([B, K])
```

For each (i, j, k), multiply and add.

```
num_backward_flops += 2 * B * D * K  # @inspect num_backward_flops
```

h1.grad[i,j] = sum_k w2[i,j] * h2.grad[i,k]

```
assert h1.grad.size() == torch.Size([B, D])
assert w2.size() == torch.Size([D, K])
assert h2.grad.size() == torch.Size([B, K])
```

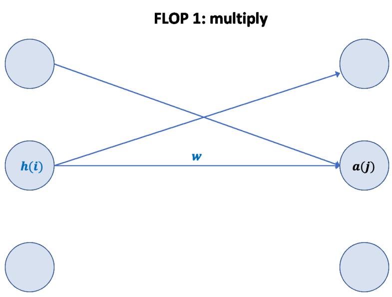
For each (i, j, k), multiply and add.

```
num_backward_flops += 2 * B * D * K  # @inspect num_backward_flops
```

This was for just w2 (D*K parameters).
Can do it for w1 (D*D parameters) as well (though don't need x.grad).

```
num_backward_flops += (2 + 2) * B * D * D  # @inspect num_backward_flops
```

A nice graphical visualization: [article]



Putting it togther:

- Forward pass: 2 (# data points) (# parameters) FLOPs
- Backward pass: 4 (# data points) (# parameters) FLOPs
- Total: 6 (# data points) (# parameters) FLOPs

```
def module_parameters():
    input_dim = 16384
    output_dim = 32
```

Model parameters are stored in PyTorch as `nn.Parameter` objects.

```
    w = nn.Parameter(torch.randn(input_dim, output_dim))
    assert isinstance(w, torch.Tensor)  # Behaves like a tensor
    assert type(w.data) == torch.Tensor  # Access the underlying tensor
```

Parameter initialization

Let's see what happens.

```
    x = nn.Parameter(torch.randn(input_dim))
    output = x @ w  # @inspect output
    assert output.size() == torch.Size([output_dim])
```

Note that each element of `output` scales as sqrt(input_dim): -84.09819030761719.
Large values can cause gradients to blow up and cause training to be unstable.

We want an initialization that is invariant to `input_dim`.
To do that, we simply rescale by 1/sqrt(input_dim)

```
    w = nn.Parameter(torch.randn(input_dim, output_dim) / np.sqrt(input_dim))
    output = x @ w  # @inspect output
```

Now each element of `output` is constant: -0.12920215725898743.

Up to a constant, this is Xavier initialization. [paper][stackexchange]

To be extra safe, we truncate the normal distribution to [-3, 3] to avoid any chance of outliers.

```
    w = nn.Parameter(nn.init.trunc_normal_(torch.empty(input_dim, output_dim), std=1 / np.sqrt(input_dim), a=-3, b=3))


def custom_model():
```

Let's build up a simple deep linear model using `nn.Parameter`.

```
    D = 64  # Dimension
    num_layers = 2
    model = Cruncher(dim=D, num_layers=num_layers)

    param_sizes = [
        (name, param.numel())
        for name, param in model.state_dict().items()
    ]
    assert param_sizes == [
        ("layers.0.weight", D * D),
        ("layers.1.weight", D * D),
        ("final.weight", D),
    ]
    num_parameters = get_num_parameters(model)
    assert num_parameters == (D * D) + (D * D) + D
```

Remember to move the model to the GPU.

```
    device = get_device()
    model = model.to(device)
```

Run the model on some data.

```
    B = 8  # Batch size
    x = torch.randn(B, D, device=device)
    y = model(x)
    assert y.size() == torch.Size([B])
```

```
class Linear(nn.Module):
    """Simple linear layer."""
    def __init__(self, input_dim: int, output_dim: int):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(input_dim, output_dim) / np.sqrt(input_dim))

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return x @ self.weight


class Cruncher(nn.Module):
    def __init__(self, dim: int, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList([
            Linear(dim, dim)
            for i in range(num_layers)
        ])
        self.final = Linear(dim, 1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        # Apply linear layers
        B, D = x.size()
        for layer in self.layers:
            x = layer(x)

        # Apply final head
        x = self.final(x)
        assert x.size() == torch.Size([B, 1])

        # Remove the last dimension
        x = x.squeeze(-1)
        assert x.size() == torch.Size([B])

        return x


def get_batch(data: np.array, batch_size: int, sequence_length: int, device: str) -> torch.Tensor:
```

Sample `batch_size` random positions into `data`.

```
    start_indices = torch.randint(len(data) - sequence_length, (batch_size,))
    assert start_indices.size() == torch.Size([batch_size])
```

Index into the data.

```
    x = torch.tensor([data[start:start + sequence_length] for start in start_indices])
    assert x.size() == torch.Size([batch_size, sequence_length])
```

Pinned memory

By default, CPU tensors are in paged memory. We can explicitly pin.

```
    if torch.cuda.is_available():
        x = x.pin_memory()
```

This allows us to copy `x` from CPU into GPU asynchronously.

```
    x = x.to(device, non_blocking=True)
```

This allows us to do two things in parallel (not done here):

- Fetch the next batch of data into CPU
- Process `x` on the GPU.

[article]
[article]

```
    return x
```

```python
def note_about_randomness():
```

Randomness shows up in many places: parameter initialization, dropout, data ordering, etc.
For reproducibility, we recommend you always pass in a different random seed for each use of randomness.
Determinism is particularly useful when debugging, so you can hunt down the bug.

There are three places to set the random seed which you should do all at once just to be safe.

```python
    # Torch
    seed = 0
    torch.manual_seed(seed)

    # NumPy
    import numpy as np
    np.random.seed(seed)

    # Python
    import random
    random.seed(seed)


def data_loading():
```

In language modeling, data is a sequence of integers (output by the tokenizer).

It is convenient to serialize them as numpy arrays (done by the tokenizer).

```python
    orig_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=np.int32)
    orig_data.tofile("data.npy")
```

You can load them back as numpy arrays.
Don't want to load the entire data into memory at once (LLaMA data is 2.8TB).
Use memmap to lazily load only the accessed parts into memory.

```python
    data = np.memmap("data.npy", dtype=np.int32)
    assert np.array_equal(data, orig_data)
```

A *data loader* generates a batch of sequences for training.

```python
    B = 2  # Batch size
    L = 4  # Length of sequence
    x = get_batch(data, batch_size=B, sequence_length=L, device=get_device())
    assert x.size() == torch.Size([B, L])


class SGD(torch.optim.Optimizer):
    def __init__(self, params: Iterable[nn.Parameter], lr: float = 0.01):
        super(SGD, self).__init__(params, dict(lr=lr))

    def step(self):
        for group in self.param_groups:
            lr = group["lr"]
            for p in group["params"]:
                grad = p.grad.data
                p.data -= lr * grad


class AdaGrad(torch.optim.Optimizer):
    def __init__(self, params: Iterable[nn.Parameter], lr: float = 0.01):
        super(AdaGrad, self).__init__(params, dict(lr=lr))

    def step(self):
        for group in self.param_groups:
            lr = group["lr"]
            for p in group["params"]:
                # Optimizer state
                state = self.state[p]
```

```
            grad = p.grad.data

            # Get squared gradients g2 = sum_{i<t} g_i^2
            g2 = state.get("g2", torch.zeros_like(grad))

            # Update optimizer state
            g2 += torch.square(grad)
            state["g2"] = g2

            # Update parameters
            p.data -= lr * grad / torch.sqrt(g2 + 1e-5)


def optimizer():
```

Recall our deep linear model.

```
    B = 2
    D = 4
    num_layers = 2
    model = Cruncher(dim=D, num_layers=num_layers).to(get_device())
```

Let's define the AdaGrad optimizer

- momentum = SGD + exponential averaging of grad
- AdaGrad = SGD + averaging by grad^2
- RMSProp = AdaGrad + exponentially averaging of grad^2
- Adam = RMSProp + momentum

AdaGrad: https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

```
    optimizer = AdaGrad(model.parameters(), lr=0.01)
    state = model.state_dict()  # @inspect state
```

Compute gradients

```
    x = torch.randn(B, D, device=get_device())
    y = torch.tensor([4., 5.], device=get_device())
    pred_y = model(x)
    loss = F.mse_loss(input=pred_y, target=y)
    loss.backward()
```

Take a step

```
    optimizer.step()
    state = model.state_dict()  # @inspect state
```

Free up the memory (optional)

```
    optimizer.zero_grad(set_to_none=True)
```

Memory

```
    # Parameters
    num_parameters = (D * D * num_layers) + D  # @inspect num_parameters
    assert num_parameters == get_num_parameters(model)

    # Activations
    num_activations = B * D * num_layers  # @inspect num_activations

    # Gradients
    num_gradients = num_parameters  # @inspect num_gradients

    # Optimizer states
    num_optimizer_states = num_parameters  # @inspect num_optimizer_states

    # Putting it all together, assuming float32
    total_memory = 4 * (num_parameters + num_activations + num_gradients + num_optimizer_states)  # @inspect total_memory
```

Compute (for one step)

```
    flops = 6 * B * num_parameters  # @inspect flops
```

Transformers

The accounting for a Transformer is more complicated, but the same idea.
Assignment 1 will ask you to do that.

Blog post describing memory usage for Transformer training [article]
Blog post descibing FLOPs for a Transformer: [article]

```
def train_loop():
```

Generate data from linear function with weights (0, 1, 2, ..., D-1).

```
    D = 16
    true_w = torch.arange(D, dtype=torch.float32, device=get_device())
    def get_batch(B: int) -> tuple[torch.Tensor, torch.Tensor]:
        x = torch.randn(B, D).to(get_device())
        true_y = x @ true_w
        return (x, true_y)
```

Let's do a basic run

```
    train("simple", get_batch, D=D, num_layers=0, B=4, num_train_steps=10, lr=0.01)
```

Do some hyperparameter tuning

```
    train("simple", get_batch, D=D, num_layers=0, B=4, num_train_steps=10, lr=0.1)


def train(name: str, get_batch,
          D: int, num_layers: int,
          B: int, num_train_steps: int, lr: float):
    model = Cruncher(dim=D, num_layers=0).to(get_device())
    optimizer = SGD(model.parameters(), lr=0.01)

    for t in range(num_train_steps):
        # Get data
        x, y = get_batch(B=B)

        # Forward (compute loss)
        pred_y = model(x)
        loss = F.mse_loss(pred_y, y)

        # Backward (compute gradients)
        loss.backward()

        # Update parameters
        optimizer.step()
        optimizer.zero_grad(set_to_none=True)


def checkpointing():
```

Training language models take a long time and certainly will certainly crash.
You don't want to lose all your progress.

During training, it is useful to periodically save your model and optimizer state to disk.

```
    model = Cruncher(dim=64, num_layers=3).to(get_device())
    optimizer = AdaGrad(model.parameters(), lr=0.01)
```

Save the checkpoint:

```
    checkpoint = {
        "model": model.state_dict(),
        "optimizer": optimizer.state_dict(),
    }
    torch.save(checkpoint, "model_checkpoint.pt")
```

Load the checkpoint:

```
loaded_checkpoint = torch.load("model_checkpoint.pt")


def mixed_precision_training():
```

Choice of data type (float32, bfloat16, fp8) have tradeoffs.

- Higher precision: more accurate/stable, more memory, more compute
- Lower precision: less accurate/stable, less memory, less compute

How can we get the best of both worlds?

Solution: use float32 by default, but use {bfloat16, fp8} when possible.

A concrete plan:

- Use {bfloat16, fp8} for the forward pass (activations).
- Use float32 for the rest (parameters, gradients).

- Mixed precision training [Micikevicius+ 2017]

Pytorch has an automatic mixed precision (AMP) library.
https://pytorch.org/docs/stable/amp.html
https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/

NVIDIA's Transformer Engine supports FP8 for linear layers
Use FP8 pervasively throughout training [Peng+ 2023]

```
############################################################

def get_memory_usage(x: torch.Tensor):
    return x.numel() * x.element_size()


def get_promised_flop_per_sec(device: str, dtype: torch.dtype) -> float:
    """Return the peak FLOP/s for `device` operating on `dtype`."""
    if not torch.cuda.is_available():
```

No CUDA device available, so can't get FLOP/s.

```
        return 1
    properties = torch.cuda.get_device_properties(device)

    if "A100" in properties.name:
        # https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf")
        if dtype == torch.float32:
            return 19.5e12
        if dtype in (torch.bfloat16, torch.float16):
            return 312e12
        raise ValueError(f"Unknown dtype: {dtype}")

    if "H100" in properties.name:
        # https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet")
        if dtype == torch.float32:
            return 67.5e12
        if dtype in (torch.bfloat16, torch.float16):
            return 1979e12 / 2  # 1979 is for sparse, dense is half of that
        raise ValueError(f"Unknown dtype: {dtype}")

    raise ValueError(f"Unknown device: {device}")


def same_storage(x: torch.Tensor, y: torch.Tensor):
    return x.untyped_storage().data_ptr() == y.untyped_storage().data_ptr()
```

```python

def time_matmul(a: torch.Tensor, b: torch.Tensor) -> float:
    """Return the number of seconds required to perform `a @ b`."""

    # Wait until previous CUDA threads are done
    if torch.cuda.is_available():
        torch.cuda.synchronize()

    def run():
        # Perform the operation
        a @ b

        # Wait until CUDA threads are done
        if torch.cuda.is_available():
            torch.cuda.synchronize()

    # Time the operation `num_trials` times
    num_trials = 5
    total_time = timeit.timeit(run, number=num_trials)

    return total_time / num_trials


def get_num_parameters(model: nn.Module) -> int:
    return sum(param.numel() for param in model.parameters())

def get_device(index: int = 0) -> torch.device:
    """Try to use the GPU if possible, otherwise, use CPU."""
    if torch.cuda.is_available():
        return torch.device(f"cuda:{index}")
    else:
        return torch.device("cpu")

if __name__ == "__main__":
    main()
```