lecture\_02.py☀️⚪️⬅️➡️↖️↗️⤴️

1from execute\_util import text, link, image

2from facts import a100\_flop\_per\_sec, h100\_flop\_per\_sec

3import torch.nn.functional as F

4import timeit

5import torch

6from typing import Iterable

7from torch import nn

8import numpy as np

9from lecture\_util import article\_link

10from jaxtyping import Float

11from einops import rearrange, einsum, reduce

12from references import zero\_2019

13

14

15def main():

16

Last lecture: overview, tokenization

17

18

Overview of this lecture:

19

* We will discuss all the **primitives** needed to train a model.

20

* We will go bottom-up from tensors to models to optimizers to the training loop.

21

* We will pay close attention to efficiency (use of **resources**).

22

23

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

24

* Memory (GB)

25

* Compute (FLOPs)

26

27 motivating\_questions()

28

29

We will not go over the Transformer.

30

There are excellent expositions:

31

[Assignment 1 handout](https://github.com/stanford-cs336/assignment1-basics/blob/main/cs336_spring2025_assignment1_basics.pdf)

32

[Mathematical description](https://johnthickstun.com/docs/transformers.pdf)

33

[Illustrated Transformer](http://jalammar.github.io/illustrated-transformer/)

34

[Illustrated GPT-2](https://jalammar.github.io/illustrated-gpt2/)

35

Instead, we'll work with simpler models.

36

37

What knowledge to take away:

38

* Mechanics: straightforward (just PyTorch)

39

* Mindset: resource accounting (remember to do it)

40

* Intuitions: broad strokes (no large models)

41

42

**Memory accounting**

43 tensors\_basics()

44 tensors\_memory()

45

46

**Compute accounting**

47 tensors\_on\_gpus()

48 tensor\_operations()

49 tensor\_einops()

50 tensor\_operations\_flops()

51 gradients\_basics()

52 gradients\_flops()

53

54

**Models**

55 module\_parameters()

56 custom\_model()

57

58

Training loop and best practices

59 note\_about\_randomness()

60 data\_loading()

61

62 optimizer()

63 train\_loop()

64 checkpointing()

65 mixed\_precision\_training()

66

67

68def motivating\_questions():

69

Let's do some napkin math.

70

71

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

72 total\_flops = 6 \* 70e9 \* 15e12 # @inspect total\_flops

73 assert h100\_flop\_per\_sec == 1979e12 / 2

74 mfu = 0.5

75 flops\_per\_day = h100\_flop\_per\_sec \* mfu \* 1024 \* 60 \* 60 \* 24 # @inspect flops\_per\_day

76 days = total\_flops / flops\_per\_day # @inspect days

77

78

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

79 h100\_bytes = 80e9 # @inspect h100\_bytes

80 bytes\_per\_parameter = 4 + 4 + (4 + 4) # parameters, gradients, optimizer state @inspect bytes\_per\_parameter

81 num\_parameters = (h100\_bytes \* 8) / bytes\_per\_parameter # @inspect num\_parameters

82

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]](https://arxiv.org/abs/1910.02054)

83

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

84

85

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

86

87

88def tensors\_basics():

89

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

90

[[PyTorch docs on tensors]](https://pytorch.org/docs/stable/tensors.html)

91

92

You can create tensors in multiple ways:

93 x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x

94 x = torch.zeros(4, 8) # 4x8 matrix of all zeros @inspect x

95 x = torch.ones(4, 8) # 4x8 matrix of all ones @inspect x

96 x = torch.randn(4, 8) # 4x8 matrix of iid Normal(0, 1) samples @inspect x

97

98

Allocate but don't initialize the values:

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

100

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

101 nn.init.trunc\_normal\_(x, mean=0, std=1, a=-2, b=2) # @inspect x

102

103

104def tensors\_memory():

105

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

106

107

**float32**

108

[[Wikipedia]](https://en.wikipedia.org/wiki/Single-precision_floating-point_format)

109

A number in a row

AI-generated content may be incorrect.

110

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

111

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

112

In deep learning, you can be a lot sloppier.

113

114

Let's examine memory usage of these tensors.

115

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

116 x = torch.zeros(4, 8) # @inspect x

117 assert x.dtype == torch.float32 # Default type

118 assert x.numel() == 4 \* 8

119 assert x.element\_size() == 4 # Float is 4 bytes

120 assert get\_memory\_usage(x) == 4 \* 8 \* 4 # 128 bytes

121

122

One matrix in the feedforward layer of GPT-3:

123 assert get\_memory\_usage(torch.empty(12288 \* 4, 12288)) == 2304 \* 1024 \* 1024 # 2.3 GB

124

...which is a lot!

125

126

**float16**

127

[[Wikipedia]](https://en.wikipedia.org/wiki/Half-precision_floating-point_format)

128

A graph of a fraction

AI-generated content may be incorrect.

129

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

130 x = torch.zeros(4, 8, dtype=torch.float16) # @inspect x

131 assert x.element\_size() == 2

132

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

133 x = torch.tensor([1e-8], dtype=torch.float16) # @inspect x

134 assert x == 0 # Underflow!

135

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

136

137

**bfloat16**

138

[[Wikipedia]](https://en.wikipedia.org/wiki/Bfloat16_floating-point_format)

139

A number bar with numbers and a black text

AI-generated content may be incorrect.

140

Google Brain developed bfloat (brain floating point) in 2018 to address this issue.

141

bfloat16 uses the same memory as float16 but has the same dynamic range as float32!

142

The only catch is that the resolution is worse, but this matters less for deep learning.

143 x = torch.tensor([1e-8], dtype=torch.bfloat16) # @inspect x

144 assert x != 0 # No underflow!

145

146

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

147 float32\_info = torch.finfo(torch.float32) # @inspect float32\_info

148 float16\_info = torch.finfo(torch.float16) # @inspect float16\_info

149 bfloat16\_info = torch.finfo(torch.bfloat16) # @inspect bfloat16\_info

150

151

**fp8**

152

In 2022, FP8 was standardized, motivated by machine learning workloads.

153

<https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html>

154

A row of numbers with a number in the center

AI-generated content may be incorrect.

155

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

156

Reference:

[[Micikevicius+ 2022]](https://arxiv.org/pdf/2209.05433.pdf)

157

158

Implications on training:

159

* Training with float32 works, but requires lots of memory.

160

* Training with fp8, float16 and even bfloat16 is risky, and you can get instability.

161

* Solution (later): use mixed precision training, see

[mixed\_precision\_training](https://stanford-cs336.github.io/spring2025-lectures/?trace=var/traces/lecture_02.json)

162

163

164def tensors\_on\_gpus():

165

By default, tensors are stored in CPU memory.

166 x = torch.zeros(32, 32)

167 assert x.device == torch.device("cpu")

168

169

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

170

A diagram of a bus

AI-generated content may be incorrect.

171

172

Let's first see if we have any GPUs.

173 if not torch.cuda.is\_available():

174 return

175

176 num\_gpus = torch.cuda.device\_count() # @inspect num\_gpus

177 for i in range(num\_gpus):

178 properties = torch.cuda.get\_device\_properties(i) # @inspect properties

179

180 memory\_allocated = torch.cuda.memory\_allocated() # @inspect memory\_allocated

181

182 text("Move the tensor to GPU memory (device 0).")

183 y = x.to("cuda:0")

184 assert y.device == torch.device("cuda", 0)

185

186 text("Or create a tensor directly on the GPU:")

187 z = torch.zeros(32, 32, device="cuda:0")

188

189 new\_memory\_allocated = torch.cuda.memory\_allocated() # @inspect new\_memory\_allocated

190 memory\_used = new\_memory\_allocated - memory\_allocated # @inspect memory\_used

191 assert memory\_used == 2 \* (32 \* 32 \* 4) # 2 32x32 matrices of 4-byte floats

192

193

194

195def tensor\_operations():

196

Most tensors are created from performing operations on other tensors.

197

Each operation has some memory and compute consequence.

198

199 tensor\_storage()

200 tensor\_slicing()

201 tensor\_elementwise()

202 tensor\_matmul()

203

204

205def tensor\_storage():

206

What are tensors in PyTorch?

207

PyTorch tensors are pointers into allocated memory

208

...with metadata describing how to get to any element of the tensor.

209

A screenshot of a computer

AI-generated content may be incorrect.

210

[[PyTorch docs]](https://pytorch.org/docs/stable/generated/torch.Tensor.stride.html)

211 x = torch.tensor([

212 [0., 1, 2, 3],

213 [4, 5, 6, 7],

214 [8, 9, 10, 11],

215 [12, 13, 14, 15],

216 ])

217

218

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

219 assert x.stride(0) == 4

220

221

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

222 assert x.stride(1) == 1

223

224

To find an element:

225 r, c = 1, 2

226 index = r \* x.stride(0) + c \* x.stride(1) # @inspect index

227 assert index == 6

228

229

230def tensor\_slicing():

231 x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x

232

233

Many operations simply provide a different **view** of the tensor.

234

This does not make a copy, and therefore mutations in one tensor affects the other.

235

236

Get row 0:

237 y = x[0] # @inspect y

238 assert torch.equal(y, torch.tensor([1., 2, 3]))

239 assert same\_storage(x, y)

240

241

Get column 1:

242 y = x[:, 1] # @inspect y

243 assert torch.equal(y, torch.tensor([2, 5]))

244 assert same\_storage(x, y)

245

246

View 2x3 matrix as 3x2 matrix:

247 y = x.view(3, 2) # @inspect y

248 assert torch.equal(y, torch.tensor([[1, 2], [3, 4], [5, 6]]))

249 assert same\_storage(x, y)

250

251

Transpose the matrix:

252 y = x.transpose(1, 0) # @inspect y

253 assert torch.equal(y, torch.tensor([[1, 4], [2, 5], [3, 6]]))

254 assert same\_storage(x, y)

255

256

Check that mutating x also mutates y.

257 x[0][0] = 100 # @inspect x, @inspect y

258 assert y[0][0] == 100

259

260

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

261 x = torch.tensor([[1., 2, 3], [4, 5, 6]]) # @inspect x

262 y = x.transpose(1, 0) # @inspect y

263 assert not y.is\_contiguous()

264 try:

265 y.view(2, 3)

266 assert False

267 except RuntimeError as e:

268 assert "view size is not compatible with input tensor's size and stride" in str(e)

269

270

One can enforce a tensor to be contiguous first:

271 y = x.transpose(1, 0).contiguous().view(2, 3) # @inspect y

272 assert not same\_storage(x, y)

273

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

274

275

276def tensor\_elementwise():

277

These operations apply some operation to each element of the tensor

278

...and return a (new) tensor of the same shape.

279

280 x = torch.tensor([1, 4, 9])

281 assert torch.equal(x.pow(2), torch.tensor([1, 16, 81]))

282 assert torch.equal(x.sqrt(), torch.tensor([1, 2, 3]))

283 assert torch.equal(x.rsqrt(), torch.tensor([1, 1 / 2, 1 / 3])) # i -> 1/sqrt(x\_i)

284

285 assert torch.equal(x + x, torch.tensor([2, 8, 18]))

286 assert torch.equal(x \* 2, torch.tensor([2, 8, 18]))

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

288

289

triu takes the upper triangular part of a matrix.

290 x = torch.ones(3, 3).triu() # @inspect x

291 assert torch.equal(x, torch.tensor([

292 [1, 1, 1],

293 [0, 1, 1],

294 [0, 0, 1]],

295 ))

296

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

297

298

299def tensor\_matmul():

300

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

301 x = torch.ones(16, 32)

302 w = torch.ones(32, 2)

303 y = x @ w

304 assert y.size() == torch.Size([16, 2])

305

306

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

307

A green rectangular object with text

AI-generated content may be incorrect.

308 x = torch.ones(4, 8, 16, 32)

309 w = torch.ones(32, 2)

310 y = x @ w

311 assert y.size() == torch.Size([4, 8, 16, 2])

312

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

313

314

315def tensor\_einops():

316 einops\_motivation()

317

318

Einops is a library for manipulating tensors where dimensions are named.

319

It is inspired by Einstein summation notation (Einstein, 1916).

320

[[Einops tutorial]](https://einops.rocks/1-einops-basics/)

321

322 jaxtyping\_basics()

323 einops\_einsum()

324 einops\_reduce()

325 einops\_rearrange()

326

327

328def einops\_motivation():

329

Traditional PyTorch code:

330 x = torch.ones(2, 2, 3) # batch, sequence, hidden @inspect x

331 y = torch.ones(2, 2, 3) # batch, sequence, hidden @inspect y

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

333

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

334

335

336def jaxtyping\_basics():

337

How do you keep track of tensor dimensions?

338

339

Old way:

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

341

342

New (jaxtyping) way:

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

344

Note: this is just documentation (no enforcement).

345

346

347def einops\_einsum():

348

Einsum is generalized matrix multiplication with good bookkeeping.

349

350

Define two tensors:

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

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

353

354

Old way:

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

356

357

New (einops) way:

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

359

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

360

361

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

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

363

364

365def einops\_reduce():

366

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

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

368

369

Old way:

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

371

372

New (einops) way:

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

374

375

376def einops\_rearrange():

377

Sometimes, a dimension represents two dimensions

378

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

379

380 x: Float[torch.Tensor, "batch seq total\_hidden"] = torch.ones(2, 3, 8) # @inspect x

381

...where total\_hidden is a flattened representation of heads \* hidden1

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

383

384

Break up total\_hidden into two dimensions (heads and hidden1):

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

386

387

Perform the transformation by w:

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

389

390

Combine heads and hidden2 back together:

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

392

393

394def tensor\_operations\_flops():

395

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

396

397

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

398

399

Two terribly confusing acronyms (pronounced the same!):

400

* FLOPs: floating-point operations (measure of computation done)

401

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

402

403

**Intuitions**

404

Training GPT-3 (2020) took 3.14e23 FLOPs.

[[article]](https://lambdalabs.com/blog/demystifying-gpt-3)

405

Training GPT-4 (2023) is speculated to take 2e25 FLOPs

[[article]](https://patmcguinness.substack.com/p/gpt-4-details-revealed)

406

US executive order: any foundation model trained with >= 1e26 FLOPs must be reported to the government (revoked in 2025)

407

408

A100 has a peak performance of 312 teraFLOP/s

[[spec]](https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf)

409 assert a100\_flop\_per\_sec == 312e12

410

411

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

[[spec]](https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet)

412 assert h100\_flop\_per\_sec == 1979e12 / 2

413

414

8 H100s for 2 weeks:

415 total\_flops = 8 \* (60 \* 60 \* 24 \* 7) \* h100\_flop\_per\_sec # @inspect total\_flops

416

417

**Linear model**

418

As motivation, suppose you have a linear model.

419

* We have n points

420

* Each point is d-dimsional

421

* The linear model maps each d-dimensional vector to a k outputs

422

423 if torch.cuda.is\_available():

424 B = 16384 # Number of points

425 D = 32768 # Dimension

426 K = 8192 # Number of outputs

427 else:

428 B = 1024

429 D = 256

430 K = 64

431

432 device = get\_device()

433 x = torch.ones(B, D, device=device)

434 w = torch.randn(D, K, device=device)

435 y = x @ w

436

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

437 actual\_num\_flops = 2 \* B \* D \* K # @inspect actual\_num\_flops

438

439

**FLOPs of other operations**

440

* Elementwise operation on a m x n matrix requires O(m n) FLOPs.

441

* Addition of two m x n matrices requires m n FLOPs.

442

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

443

444

Interpretation:

445

* B is the number of data points

446

* (D K) is the number of parameters

447

* FLOPs for forward pass is 2 (# tokens) (# parameters)

448

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

449

450

How do our FLOPs calculations translate to wall-clock time (seconds)?

451

Let us time it!

452 actual\_time = time\_matmul(x, w) # @inspect actual\_time

453 actual\_flop\_per\_sec = actual\_num\_flops / actual\_time # @inspect actual\_flop\_per\_sec

454

455

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

456

* A100

[[spec]](https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf)

457

* H100

[[spec]](https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet)

458

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

459 promised\_flop\_per\_sec = get\_promised\_flop\_per\_sec(device, x.dtype) # @inspect promised\_flop\_per\_sec

460

461

**Model FLOPs utilization (MFU)**

462

463

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

464 mfu = actual\_flop\_per\_sec / promised\_flop\_per\_sec # @inspect mfu

465

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

466

467

Let's do it with bfloat16:

468 x = x.to(torch.bfloat16)

469 w = w.to(torch.bfloat16)

470 bf16\_actual\_time = time\_matmul(x, w) # @inspect bf16\_actual\_time

471 bf16\_actual\_flop\_per\_sec = actual\_num\_flops / bf16\_actual\_time # @inspect bf16\_actual\_flop\_per\_sec

472 bf16\_promised\_flop\_per\_sec = get\_promised\_flop\_per\_sec(device, x.dtype) # @inspect bf16\_promised\_flop\_per\_sec

473 bf16\_mfu = bf16\_actual\_flop\_per\_sec / bf16\_promised\_flop\_per\_sec # @inspect bf16\_mfu

474

Note: comparing bfloat16 to float32, the actual FLOP/s is higher.

475

The MFU here is rather low, probably because the promised FLOPs is a bit optimistic.

476

477

**Summary**

478

* Matrix multiplications dominate: (2 m n p) FLOPs

479

* FLOP/s depends on hardware (H100 >> A100) and data type (bfloat16 >> float32)

480

* Model FLOPs utilization (MFU): (actual FLOP/s) / (promised FLOP/s)

481

482

483def gradients\_basics():

484

So far, we've constructed tensors (which correspond to either parameters or data) and passed them through operations (forward).

485

Now, we're going to compute the gradient (backward).

486

487

As a simple example, let's consider the simple linear model:

488

y = 0.5 (x \* w - 5)^2

489

490

Forward pass: compute loss

491 x = torch.tensor([1., 2, 3])

492 w = torch.tensor([1., 1, 1], requires\_grad=True) # Want gradient

493 pred\_y = x @ w

494 loss = 0.5 \* (pred\_y - 5).pow(2)

495

496

Backward pass: compute gradients

497 loss.backward()

498 assert loss.grad is None

499 assert pred\_y.grad is None

500 assert x.grad is None

501 assert torch.equal(w.grad, torch.tensor([1, 2, 3]))

502

503

504def gradients\_flops():

505

Let us do count the FLOPs for computing gradients.

506

507

Revisit our linear model

508 if torch.cuda.is\_available():

509 B = 16384 # Number of points

510 D = 32768 # Dimension

511 K = 8192 # Number of outputs

512 else:

513 B = 1024

514 D = 256

515 K = 64

516

517 device = get\_device()

518 x = torch.ones(B, D, device=device)

519 w1 = torch.randn(D, D, device=device, requires\_grad=True)

520 w2 = torch.randn(D, K, device=device, requires\_grad=True)

521

522

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

523 h1 = x @ w1

524 h2 = h1 @ w2

525 loss = h2.pow(2).mean()

526

527

Recall the number of forward FLOPs:  [tensor\_operations\_flops](https://stanford-cs336.github.io/spring2025-lectures/?trace=var/traces/lecture_02.json)

528

* Multiply x[i][j] \* w1[j][k]

529

* Add to h1[i][k]

530

* Multiply h1[i][j] \* w2[j][k]

531

* Add to h2[i][k]

532 num\_forward\_flops = (2 \* B \* D \* D) + (2 \* B \* D \* K) # @inspect num\_forward\_flops

533

534

How many FLOPs is running the backward pass?

535 h1.retain\_grad() # For debugging

536 h2.retain\_grad() # For debugging

537 loss.backward()

538

539

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

540

541

* h1.grad = d loss / d h1

542

* h2.grad = d loss / d h2

543

* w1.grad = d loss / d w1

544

* w2.grad = d loss / d w2

545

546

Focus on the parameter w2.

547

Invoke the chain rule.

548

549 num\_backward\_flops = 0 # @inspect num\_backward\_flops

550

551

w2.grad[j,k] = sum\_i h1[i,j] \* h2.grad[i,k]

552 assert w2.grad.size() == torch.Size([D, K])

553 assert h1.size() == torch.Size([B, D])

554 assert h2.grad.size() == torch.Size([B, K])

555

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

556 num\_backward\_flops += 2 \* B \* D \* K # @inspect num\_backward\_flops

557

558

h1.grad[i,j] = sum\_k w2[i,j] \* h2.grad[i,k]

559 assert h1.grad.size() == torch.Size([B, D])

560 assert w2.size() == torch.Size([D, K])

561 assert h2.grad.size() == torch.Size([B, K])

562

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

563 num\_backward\_flops += 2 \* B \* D \* K # @inspect num\_backward\_flops

564

565

This was for just w2 (D\*K parameters).

566

Can do it for w1 (D\*D parameters) as well (though don't need x.grad).

567 num\_backward\_flops += (2 + 2) \* B \* D \* D # @inspect num\_backward\_flops

568

569

A nice graphical visualization:

[[article]](https://medium.com/@dzmitrybahdanau/the-flops-calculus-of-language-model-training-3b19c1f025e4)

570

A diagram of a diagram of a diagram

AI-generated content may be incorrect.

571

572

Putting it togther:

573

* Forward pass: 2 (# data points) (# parameters) FLOPs

574

* Backward pass: 4 (# data points) (# parameters) FLOPs

575

* Total: 6 (# data points) (# parameters) FLOPs

576

577

578def module\_parameters():

579 input\_dim = 16384

580 output\_dim = 32

581

582

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

583 w = nn.Parameter(torch.randn(input\_dim, output\_dim))

584 assert isinstance(w, torch.Tensor) # Behaves like a tensor

585 assert type(w.data) == torch.Tensor # Access the underlying tensor

586

587

**Parameter initialization**

588

589

Let's see what happens.

590 x = nn.Parameter(torch.randn(input\_dim))

591 output = x @ w # @inspect output

592 assert output.size() == torch.Size([output\_dim])

593

Note that each element of output scales as sqrt(input\_dim): -84.09819030761719.

594

Large values can cause gradients to blow up and cause training to be unstable.

595

596

We want an initialization that is invariant to input\_dim.

597

To do that, we simply rescale by 1/sqrt(input\_dim)

598 w = nn.Parameter(torch.randn(input\_dim, output\_dim) / np.sqrt(input\_dim))

599 output = x @ w # @inspect output

600

Now each element of output is constant: -0.12920215725898743.

601

602

Up to a constant, this is Xavier initialization.

[[paper]](https://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)

[[stackexchange]](https://ai.stackexchange.com/questions/30491/is-there-a-proper-initialization-technique-for-the-weight-matrices-in-multi-head)

603

604

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

605 w = nn.Parameter(nn.init.trunc\_normal\_(torch.empty(input\_dim, output\_dim), std=1 / np.sqrt(input\_dim), a=-3, b=3))

606

607

608def custom\_model():

609

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

610

611 D = 64 # Dimension

612 num\_layers = 2

613 model = Cruncher(dim=D, num\_layers=num\_layers)

614

615 param\_sizes = [

616 (name, param.numel())

617 for name, param in model.state\_dict().items()

618 ]

619 assert param\_sizes == [

620 ("layers.0.weight", D \* D),

621 ("layers.1.weight", D \* D),

622 ("final.weight", D),

623 ]

624 num\_parameters = get\_num\_parameters(model)

625 assert num\_parameters == (D \* D) + (D \* D) + D

626

627

Remember to move the model to the GPU.

628 device = get\_device()

629 model = model.to(device)

630

631

Run the model on some data.

632 B = 8 # Batch size

633 x = torch.randn(B, D, device=device)

634 y = model(x)

635 assert y.size() == torch.Size([B])

636

637

638class Linear(nn.Module):

639 """Simple linear layer."""

640 def \_\_init\_\_(self, input\_dim: int, output\_dim: int):

641 super().\_\_init\_\_()

642 self.weight = nn.Parameter(torch.randn(input\_dim, output\_dim) / np.sqrt(input\_dim))

643

644 def forward(self, x: torch.Tensor) -> torch.Tensor:

645 return x @ self.weight

646

647

648class Cruncher(nn.Module):

649 def \_\_init\_\_(self, dim: int, num\_layers: int):

650 super().\_\_init\_\_()

651 self.layers = nn.ModuleList([

652 Linear(dim, dim)

653 for i in range(num\_layers)

654 ])

655 self.final = Linear(dim, 1)

656

657 def forward(self, x: torch.Tensor) -> torch.Tensor:

658 # Apply linear layers

659 B, D = x.size()

660 for layer in self.layers:

661 x = layer(x)

662

663 # Apply final head

664 x = self.final(x)

665 assert x.size() == torch.Size([B, 1])

666

667 # Remove the last dimension

668 x = x.squeeze(-1)

669 assert x.size() == torch.Size([B])

670

671 return x

672

673

674def get\_batch(data: np.array, batch\_size: int, sequence\_length: int, device: str) -> torch.Tensor:

675

Sample batch\_size random positions into data.

676 start\_indices = torch.randint(len(data) - sequence\_length, (batch\_size,))

677 assert start\_indices.size() == torch.Size([batch\_size])

678

679

Index into the data.

680 x = torch.tensor([data[start:start + sequence\_length] for start in start\_indices])

681 assert x.size() == torch.Size([batch\_size, sequence\_length])

682

683

**Pinned memory**

684

685

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

686 if torch.cuda.is\_available():

687 x = x.pin\_memory()

688

689

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

690 x = x.to(device, non\_blocking=True)

691

692

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

693

* Fetch the next batch of data into CPU

694

* Process x on the GPU.

695

696

[[article]](https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/)

697

[[article]](https://gist.github.com/ZijiaLewisLu/eabdca955110833c0ce984d34eb7ff39?permalink_comment_id=3417135)

698

699 return x

700

701

702def note\_about\_randomness():

703

Randomness shows up in many places: parameter initialization, dropout, data ordering, etc.

704

For reproducibility, we recommend you always pass in a different random seed for each use of randomness.

705

Determinism is particularly useful when debugging, so you can hunt down the bug.

706

707

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

708

709 # Torch

710 seed = 0

711 torch.manual\_seed(seed)

712

713 # NumPy

714 import numpy as np

715 np.random.seed(seed)

716

717 # Python

718 import random

719 random.seed(seed)

720

721

722def data\_loading():

723

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

724

725

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

726 orig\_data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], dtype=np.int32)

727 orig\_data.tofile("data.npy")

728

729

You can load them back as numpy arrays.

730

Don't want to load the entire data into memory at once (LLaMA data is 2.8TB).

731

Use memmap to lazily load only the accessed parts into memory.

732 data = np.memmap("data.npy", dtype=np.int32)

733 assert np.array\_equal(data, orig\_data)

734

735

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

736 B = 2 # Batch size

737 L = 4 # Length of sequence

738 x = get\_batch(data, batch\_size=B, sequence\_length=L, device=get\_device())

739 assert x.size() == torch.Size([B, L])

740

741

742class SGD(torch.optim.Optimizer):

743 def \_\_init\_\_(self, params: Iterable[nn.Parameter], lr: float = 0.01):

744 super(SGD, self).\_\_init\_\_(params, dict(lr=lr))

745

746 def step(self):

747 for group in self.param\_groups:

748 lr = group["lr"]

749 for p in group["params"]:

750 grad = p.grad.data

751 p.data -= lr \* grad

752

753

754class AdaGrad(torch.optim.Optimizer):

755 def \_\_init\_\_(self, params: Iterable[nn.Parameter], lr: float = 0.01):

756 super(AdaGrad, self).\_\_init\_\_(params, dict(lr=lr))

757

758 def step(self):

759 for group in self.param\_groups:

760 lr = group["lr"]

761 for p in group["params"]:

762 # Optimizer state

763 state = self.state[p]

764 grad = p.grad.data

765

766 # Get squared gradients g2 = sum\_{i<t} g\_i^2

767 g2 = state.get("g2", torch.zeros\_like(grad))

768

769 # Update optimizer state

770 g2 += torch.square(grad)

771 state["g2"] = g2

772

773 # Update parameters

774 p.data -= lr \* grad / torch.sqrt(g2 + 1e-5)

775

776

777def optimizer():

778

Recall our deep linear model.

779 B = 2

780 D = 4

781 num\_layers = 2

782 model = Cruncher(dim=D, num\_layers=num\_layers).to(get\_device())

783

784

Let's define the AdaGrad optimizer

785

* momentum = SGD + exponential averaging of grad

786

* AdaGrad = SGD + averaging by grad^2

787

* RMSProp = AdaGrad + exponentially averaging of grad^2

788

* Adam = RMSProp + momentum

789

790

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

791 optimizer = AdaGrad(model.parameters(), lr=0.01)

792 state = model.state\_dict() # @inspect state

793

794

Compute gradients

795 x = torch.randn(B, D, device=get\_device())

796 y = torch.tensor([4., 5.], device=get\_device())

797 pred\_y = model(x)

798 loss = F.mse\_loss(input=pred\_y, target=y)

799 loss.backward()

800

801

Take a step

802 optimizer.step()

803 state = model.state\_dict() # @inspect state

804

805

Free up the memory (optional)

806 optimizer.zero\_grad(set\_to\_none=True)

807

808

**Memory**

809

810 # Parameters

811 num\_parameters = (D \* D \* num\_layers) + D # @inspect num\_parameters

812 assert num\_parameters == get\_num\_parameters(model)

813

814 # Activations

815 num\_activations = B \* D \* num\_layers # @inspect num\_activations

816

817 # Gradients

818 num\_gradients = num\_parameters # @inspect num\_gradients

819

820 # Optimizer states

821 num\_optimizer\_states = num\_parameters # @inspect num\_optimizer\_states

822

823 # Putting it all together, assuming float32

824 total\_memory = 4 \* (num\_parameters + num\_activations + num\_gradients + num\_optimizer\_states) # @inspect total\_memory

825

826

**Compute (for one step)**

827 flops = 6 \* B \* num\_parameters # @inspect flops

828

829

**Transformers**

830

831

The accounting for a Transformer is more complicated, but the same idea.

832

Assignment 1 will ask you to do that.

833

834

Blog post describing memory usage for Transformer training

[[article]](https://erees.dev/transformer-memory/)

835

Blog post descibing FLOPs for a Transformer:

[[article]](https://www.adamcasson.com/posts/transformer-flops)

836

837

838def train\_loop():

839

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

840 D = 16

841 true\_w = torch.arange(D, dtype=torch.float32, device=get\_device())

842 def get\_batch(B: int) -> tuple[torch.Tensor, torch.Tensor]:

843 x = torch.randn(B, D).to(get\_device())

844 true\_y = x @ true\_w

845 return (x, true\_y)

846

847

Let's do a basic run

848 train("simple", get\_batch, D=D, num\_layers=0, B=4, num\_train\_steps=10, lr=0.01)

849

850

Do some hyperparameter tuning

851 train("simple", get\_batch, D=D, num\_layers=0, B=4, num\_train\_steps=10, lr=0.1)

852

853

854def train(name: str, get\_batch,

855 D: int, num\_layers: int,

856 B: int, num\_train\_steps: int, lr: float):

857 model = Cruncher(dim=D, num\_layers=0).to(get\_device())

858 optimizer = SGD(model.parameters(), lr=0.01)

859

860 for t in range(num\_train\_steps):

861 # Get data

862 x, y = get\_batch(B=B)

863

864 # Forward (compute loss)

865 pred\_y = model(x)

866 loss = F.mse\_loss(pred\_y, y)

867

868 # Backward (compute gradients)

869 loss.backward()

870

871 # Update parameters

872 optimizer.step()

873 optimizer.zero\_grad(set\_to\_none=True)

874

875

876def checkpointing():

877

Training language models take a long time and certainly will certainly crash.

878

You don't want to lose all your progress.

879

880

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

881

882 model = Cruncher(dim=64, num\_layers=3).to(get\_device())

883 optimizer = AdaGrad(model.parameters(), lr=0.01)

884

885

Save the checkpoint:

886 checkpoint = {

887 "model": model.state\_dict(),

888 "optimizer": optimizer.state\_dict(),

889 }

890 torch.save(checkpoint, "model\_checkpoint.pt")

891

892

Load the checkpoint:

893 loaded\_checkpoint = torch.load("model\_checkpoint.pt")

894

895

896def mixed\_precision\_training():

897

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

898

* Higher precision: more accurate/stable, more memory, more compute

899

* Lower precision: less accurate/stable, less memory, less compute

900

901

How can we get the best of both worlds?

902

903

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

904

905

A concrete plan:

906

* Use {bfloat16, fp8} for the forward pass (activations).

907

* Use float32 for the rest (parameters, gradients).

908

909

* Mixed precision training

[[Micikevicius+ 2017]](https://arxiv.org/pdf/1710.03740.pdf)

910

911

Pytorch has an automatic mixed precision (AMP) library.

912

<https://pytorch.org/docs/stable/amp.html>

913

<https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/>

914

915

NVIDIA's Transformer Engine supports FP8 for linear layers

916

Use FP8 pervasively throughout training

[[Peng+ 2023]](https://arxiv.org/pdf/2310.18313.pdf)

917

918

919############################################################

920

921def get\_memory\_usage(x: torch.Tensor):

922 return x.numel() \* x.element\_size()

923

924

925def get\_promised\_flop\_per\_sec(device: str, dtype: torch.dtype) -> float:

926 """Return the peak FLOP/s for `device` operating on `dtype`."""

927 if not torch.cuda.is\_available():

928

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

929 return 1

930 properties = torch.cuda.get\_device\_properties(device)

931

932 if "A100" in properties.name:

933 # https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf")

934 if dtype == torch.float32:

935 return 19.5e12

936 if dtype in (torch.bfloat16, torch.float16):

937 return 312e12

938 raise ValueError(f"Unknown dtype: {dtype}")

939

940 if "H100" in properties.name:

941 # https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet")

942 if dtype == torch.float32:

943 return 67.5e12

944 if dtype in (torch.bfloat16, torch.float16):

945 return 1979e12 / 2 # 1979 is for sparse, dense is half of that

946 raise ValueError(f"Unknown dtype: {dtype}")

947

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

949

950

951def same\_storage(x: torch.Tensor, y: torch.Tensor):

952 return x.untyped\_storage().data\_ptr() == y.untyped\_storage().data\_ptr()

953

954

955def time\_matmul(a: torch.Tensor, b: torch.Tensor) -> float:

956 """Return the number of seconds required to perform `a @ b`."""

957

958 # Wait until previous CUDA threads are done

959 if torch.cuda.is\_available():

960 torch.cuda.synchronize()

961

962 def run():

963 # Perform the operation

964 a @ b

965

966 # Wait until CUDA threads are done

967 if torch.cuda.is\_available():

968 torch.cuda.synchronize()

969

970 # Time the operation `num\_trials` times

971 num\_trials = 5

972 total\_time = timeit.timeit(run, number=num\_trials)

973

974 return total\_time / num\_trials

975

976

977def get\_num\_parameters(model: nn.Module) -> int:

978 return sum(param.numel() for param in model.parameters())

979

980def get\_device(index: int = 0) -> torch.device:

981 """Try to use the GPU if possible, otherwise, use CPU."""

982 if torch.cuda.is\_available():

983 return torch.device(f"cuda:{index}")

984 else:

985 return torch.device("cpu")

986

987if \_\_name\_\_ == "\_\_main\_\_":

988 main()