

lecture_01.py



```
1 import regex
2 from abc import ABC
3 from dataclasses import dataclass
4 from collections import defaultdict
5 import random
6
7 from execute_util import link, image, text
8 from lecture_util import article_link, x_link, youtube_link
9 from references import gpt_3, gpt4, shannon1950, bengio2003, susketver2014, \
10     bahdanau2015_attention, transformer_2017, gpt2, t5, kaplan_scaling_laws_2020, \
11     the_pile, gpt_j, opt_175b, bloom, palm, chinchilla, llama, mistral_7b, \
12     instruct_gpt, dpo, adamw2017, lima, deepseek_v3, adam2014, grpo, ppo2017, muon, \
13     large_batch_training_2018, wsd_2024, cosine_learning_rate_2017, olmo_7b, moe_2017, \
14     megatron_lm_2019, shazeer_2020, elmo, bert, qwen_2_5, deepseek_r1, moe_2017, \
15     rms_norm_2019, rope_2021, soap, gqa, mla, deepseek_67b, deepseek_v2, brants2007, \
16     layernorm_2016, pre_post_norm_2020, llama2, llama3, olmo2, \
17     megabyte, byt5, blt, tfree, sennrich_2016, zero_2019, gpipe_2018
18 from data import get_common_crawl_urls, read_common_crawl, write_documents, markdownify_documents
19 from model_util import query_gpt4o
20
21 import tiktoken
22
23 def main():
24     welcome()
25     why_this_course_exists()
26     current_landscape()
27
28     what_is_this_program()
29
30     course_logistics()
31     course_components()
32
33     tokenization()
34
35     Next time: PyTorch building blocks, resource accounting
36
37
38 def welcome():
39     CS336: Language Models From Scratch (Spring 2025)
40
41     Course Staff
```



Tatsunori Hashimoto
Instructor



Percy Liang
Instructor



Neil Band
CA



Marcel Rød
CA



Rohith Kuditipudi
CA

```
42
43     This is the second offering of CS336.
44     Stanford edition has grown by 50%.
45     Lectures will be posted on YouTube and be made available to the whole world.
46
47
48 def why_this_course_exists():
```

49 Why did we make this course?

50
51 Let's ask GPT-4 [OpenAI+ 2023]
52 response = query_gpt4o(prompt="Why teach a course on building language models from scratch? Answer in one sentence.")
@inspect response
53
54 Problem: researchers are becoming **disconnected** from the underlying technology.
55 8 years ago, researchers would implement and train their own models.
56 6 years ago, researchers would download a model (e.g., BERT) and fine-tune it.
57 Today, researchers just prompt a proprietary model (e.g., GPT-4/Claude/Gemini).
58
59 Moving up levels of abstractions boosts productivity, but
60 • These abstractions are leaky (in contrast to programming languages or operating systems).
61 • There is still fundamental research to be done that require tearing up the stack.
62
63 **Full understanding** of this technology is necessary for **fundamental research**.
64
65 This course: **understanding via building**
66 But there's one small problem...
67

68 The industrialization of language models



69
70 GPT-4 supposedly has 1.8T parameters. [article]
71 GPT-4 supposedly cost \$100M to train. [article]
72 xAI builds cluster with 200,000 H100s to train Grok. [article]
73 Stargate (OpenAI, NVIDIA, Oracle) invests \$500B over 4 years. [article]
74
75 Also, there are no public details on how frontier models are built.
76 From the GPT-4 technical report [OpenAI+ 2023]:
77

78 2 Scope and Limitations of this Technical Report

This report focuses on the capabilities, limitations, and safety properties of GPT-4. GPT-4 is a Transformer-style model [39] pre-trained to predict the next token in a document, using both publicly available data (such as internet data) and data licensed from third-party providers. The model was then fine-tuned using Reinforcement Learning from Human Feedback (RLHF) [40]. Given both the competitive landscape and the safety implications of large-scale models like GPT-4, this report contains no further details about the architecture (including model size), hardware, training compute, dataset construction, training method, or similar.

We are committed to independent auditing of our technologies, and shared some initial steps and ideas in this area in the system card accompanying this release.² We plan to make further technical details available to additional third parties who can advise us on how to weigh the competitive and safety considerations above against the scientific value of further transparency.

79

80 More is different

81 Frontier models are out of reach for us.
82 But building small language models (<1B parameters in this class) might not be representative of large language models.
83
84 Example 1: fraction of FLOPs spent in attention versus MLP changes with scale. [X]

85

Stephen Roller
@stephenroller

I find people unfamiliar with scaling are shocked by this:

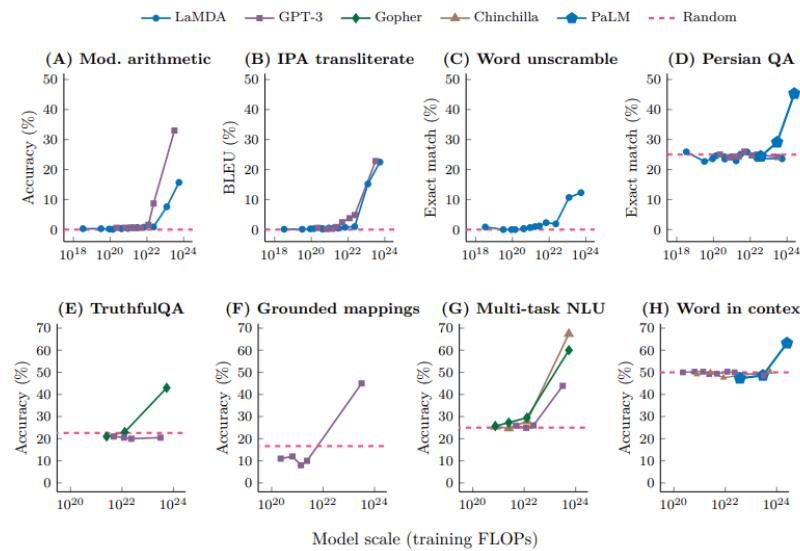
1	description	FLOPs / update	% FLOPS MHA	% FLOPS FFN	% FLOPS attn	% FLOPS logit
8	OPT setups					
9	760M	4.3E+15	35%	44%	14.8%	5.8%
10	1.3B	1.3E+16	32%	51%	12.7%	5.0%
11	2.7B	2.5E+16	29%	56%	11.2%	3.3%
12	6.7B	1.1E+17	24%	65%	8.1%	2.4%
13	13B	4.1E+17	22%	69%	6.9%	1.6%
14	30B	9.0E+17	20%	74%	5.3%	1.0%
15	66B	9.5E+17	18%	77%	4.3%	0.6%
16	175B	2.4E+18	17%	80%	3.3%	0.3%

8:31 PM · Oct 11, 2022

86

Example 2: emergence of behavior with scale [Wei+ 2022]

87



88

What can we learn in this class that transfers to frontier models?

89

There are three types of knowledge:

90

- **Mechanics:** how things work (what a Transformer is, how model parallelism leverages GPUs)
- **Mindset:** squeezing the most out of the hardware, taking scale seriously (scaling laws)
- **Intuitions:** which data and modeling decisions yield good accuracy

91

We can teach mechanics and mindset (these do transfer).

92

We can only partially teach intuitions (do not necessarily transfer across scales).

93

94

Intuitions? 🤔

95

Some design decisions are simply not (yet) justifiable and just come from experimentation.

96

Example: Noam Shazeer paper that introduced SwiGLU [Shazeer 2020]

97

4 Conclusions

98

We have extended the GLU family of layers and proposed their use in Transformer. In a transfer-learning setup, the new variants seem to produce better perplexities for the de-noising objective used in pre-training, as well as better results on many downstream language-understanding tasks. These architectures are simple to implement, and have no apparent computational drawbacks. We offer no explanation as to why these architectures seem to work; we attribute their success, as all else, to divine benevolence.

99

100

The bitter lesson

101

Wrong interpretation: scale is all that matters, algorithms don't matter.

102

Right interpretation: algorithms that scale is what matters.

103

accuracy = efficiency x resources

107 In fact, efficiency is way more important at larger scale (can't afford to be wasteful).
108 [Hernandez+ 2020] showed 44x algorithmic efficiency on ImageNet between 2012 and 2019
109
110 Framing: what is the best model one can build given a certain compute and data budget?
111 In other words, **maximize efficiency!**
112
113
114 def current_landscape():

Pre-neural (before 2010s)

- Language model to measure the entropy of English [Shannon 1950]
- Lots of work on n-gram language models (for machine translation, speech recognition) [Brants+ 2007]

Neural ingredients (2010s)

- First neural language model [Bengio+ 2003]
- Sequence-to-sequence modeling (for machine translation) [Sutskever+ 2014]
- Adam optimizer [Kingma+ 2014]
- Attention mechanism (for machine translation) [Bahdanau+ 2014]
- Transformer architecture (for machine translation) [Vaswani+ 2017]
- Mixture of experts [Shazeer+ 2017]
- Model parallelism [Huang+ 2018][Rajbhandari+ 2019][Shoeybi+ 2019]

Early foundation models (late 2010s)

- ELMo: pretraining with LSTMs, fine-tuning helps tasks [Peters+ 2018]
- BERT: pretraining with Transformer, fine-tuning helps tasks [Devlin+ 2018]
- Google's T5 (11B): cast everything as text-to-text [Raffel+ 2019]

Embracing scaling, more closed

- OpenAI's GPT-2 (1.5B): fluent text, first signs of zero-shot, staged release [Radford+ 2019]
- Scaling laws: provide hope / predictability for scaling [Kaplan+ 2020]
- OpenAI's GPT-3 (175B): in-context learning, closed [Brown+ 2020]
- Google's PaLM (540B): massive scale, undertrained [Chowdhery+ 2022]
- DeepMind's Chinchilla (70B): compute-optimal scaling laws [Hoffmann+ 2022]

Open models

- EleutherAI's open datasets (The Pile) and models (GPT-J) [Gao+ 2020][Wang+ 2021]
- Meta's OPT (175B): GPT-3 replication, lots of hardware issues [Zhang+ 2022]
- Hugging Face / BigScience's BLOOM: focused on data sourcing [Workshop+ 2022]
- Meta's Llama models [Touvron+ 2023][Touvron+ 2023][Grattafiori+ 2024]
- Alibaba's Qwen models [Qwen+ 2024]
- DeepSeek's models [DeepSeek-AI+ 2024][DeepSeek-AI+ 2024][DeepSeek-AI+ 2024]
- AI2's OLMo 2 [Groeneveld+ 2024][OLMo+ 2024]

Levels of openness

- Closed models (e.g., GPT-4o): API access only [OpenAI+ 2023]
- Open-weight models (e.g., DeepSeek): weights available, paper with architecture details, some training details, no data details [DeepSeek-AI+ 2024]
- Open-source models (e.g., OLMo): weights and data available, paper with most details (but not necessarily the rationale, failed experiments) [Groeneveld+ 2024]

Today's frontier models

- OpenAI's o3 <https://openai.com/index/openai-o3-mini/>
- Anthropic's Claude Sonnet 3.7 <https://www.anthropic.com/news/clause-3-7-sonnet>
- xAI's Grok 3 <https://x.ai/news/grok-3>

```
158     • Google's Gemini 2.5 https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/
159     • Meta's Llama 3.3 https://ai.meta.com/blog/meta-llama-3/
160     • DeepSeek's r1 [DeepSeek-AI+ 2025]
161     • Alibaba's Qwen 2.5 Max https://qwenlm.github.io/blog/qwen2.5-max/
162     • Tencent's Hunyuan-T1 https://tencent.github.io/llm.hunyuan.T1/README\_EN.html
163
164
165 def what_is_this_program():
166     This is an executable lecture, a program whose execution delivers the content of a lecture.
167     Executable lectures make it possible to:
168     • view and run code (since everything is code!),
169     total = 0 # @inspect total
170     for x in [1, 2, 3]: # @inspect x
171         total += x # @inspect total
172     • see the hierarchical structure of the lecture, and
173     • jump to definitions and concepts: supervised_finetuning
174
175
176 def course_logistics():
177     All information online: https://stanford-cs336.github.io/spring2025/
178
179     This is a 5-unit class.
180     Comment from Spring 2024 course evaluation: The entire assignment was approximately the same amount of work as all 5 assignments from CS 224n plus the final project. And that's just the first homework assignment.
181
182 Why you should take this course
183     • You have an obsessive need to understand how things work.
184     • You want to build up your research engineering muscles.
185
186 Why you should not take this course
187     • You actually want to get research done this quarter.  

188     (Talk to your advisor.)
189     • You are interested in learning about the hottest new techniques in AI (e.g., multimodality, RAG, etc.).  

190     (You should take a seminar class for that.)
191     • You want to get good results on your own application domain.  

192     (You should just prompt or fine-tune an existing model.)
193
194 How you can follow along at home
195     • All lecture materials and assignments will be posted online, so feel free to follow on your own.
196     • Lectures are recorded via CGOE, formally SCPD and be made available on YouTube (with some lag).
197     • We plan to offer this class again next year.
198
199 Assignments
200     • 5 assignments (basics, systems, scaling laws, data, alignment).
201     • No scaffolding code, but we provide unit tests and adapter interfaces to help you check correctness.
202     • Implement locally to test for correctness, then run on cluster for benchmarking (accuracy and speed).
203     • Leaderboard for some assignments (minimize perplexity given training budget).
204     • AI tools (e.g., CoPilot, Cursor) can take away from learning, so use at your own risk.
205
206 Cluster
207     • Thanks to Together AI for providing a compute cluster. 🎉
208     • Please read the guide on how to use the cluster.
209     • Start your assignments early, since the cluster will fill up close to the deadline!
```

```
209 def course_components():
```

It's all about efficiency

211 Resources: data + hardware (compute, memory, communication bandwidth)

212 How do you train the best model given a fixed set of resources?

213 Example: given a Common Crawl dump and 32 H100s for 2 weeks, what should you do?

214

215 Design decisions:

216

Basics	Systems	Scaling laws	Data	Alignment
Tokenization Architecture Loss function Optimizer Learning rate	Kernels Parallelism Quantization Activation checkpointing CPU offloading Inference	Scaling sequence Model complexity Loss metric Parametric form	Evaluation Curation Transformation Filtering Deduplication Mixing	Supervised fine-tuning Reinforcement learning Preference data Synthetic data Verifiers

217

Overview of the course

```
219 basics()
```

```
220 systems()
```

```
221 scaling_laws()
```

```
222 data()
```

```
223 alignment()
```

224

Efficiency drives design decisions

226

227 Today, we are compute-constrained, so design decisions will reflect squeezing the most out of given hardware.

228

- Data processing: avoid wasting precious compute updating on bad / irrelevant data
- Tokenization: working with raw bytes is elegant, but compute-inefficient with today's model architectures.
- Model architecture: many changes motivated by reducing memory or FLOPs (e.g., sharing KV caches, sliding window attention)
- Training: we can get away with a single epoch!
- Scaling laws: use less compute on smaller models to do hyperparameter tuning
- Alignment: if tune model more to desired use cases, require smaller base models

231

232 Tomorrow, we will become data-constrained...

233

234

```
238 class Tokenizer(ABC):
```

```
239     """Abstract interface for a tokenizer."""
```

```
240     def encode(self, string: str) -> list[int]:
```

```
241         raise NotImplementedError
```

242

```
243     def decode(self, indices: list[int]) -> str:
```

```
244         raise NotImplementedError
```

245

246

```
247     def basics():
```

248 Goal: get a basic version of the full pipeline working

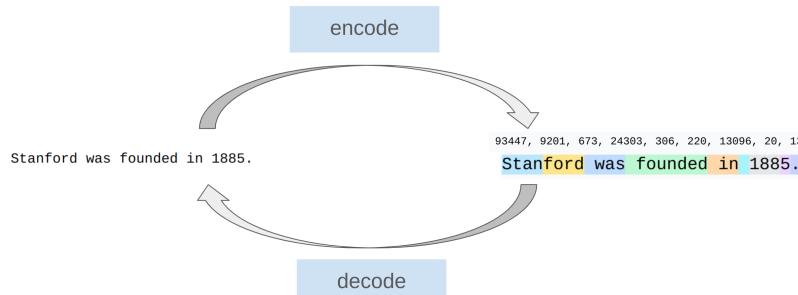
249 Components: tokenization, model architecture, training

250

Tokenization

251 Tokenizers convert between strings and sequences of integers (tokens)

253



254 Intuition: break up string into popular segments

255

256 This course: Byte-Pair Encoding (BPE) tokenizer [Sennrich+ 2015]

257

258 Tokenizer-free approaches: [Xue+ 2021][Yu+ 2023][Pagnoni+ 2024][Deisereth+ 2024]

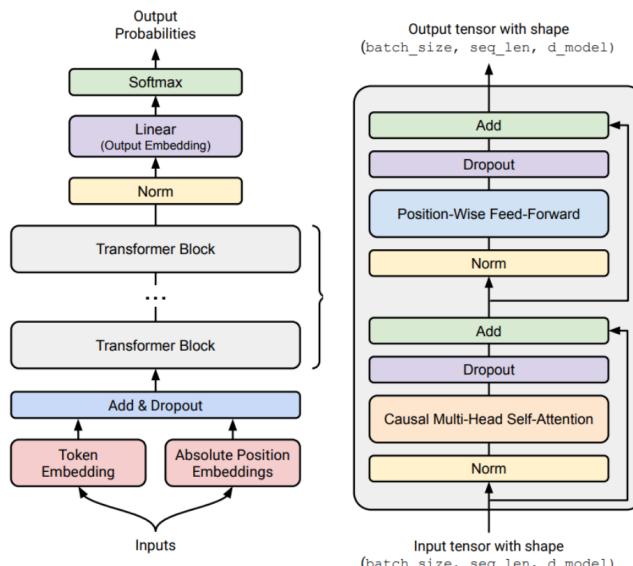
259 Use bytes directly, promising, but have not yet been scaled up to the frontier.

260

261 Architecture

262 Starting point: original Transformer [Vaswani+ 2017]

263



264

265 Variants:

- Activation functions: ReLU, SwiGLU [Shazeer 2020]
- Positional encodings: sinusoidal, RoPE [Su+ 2021]
- Normalization: LayerNorm, RMSNorm [Ba+ 2016][Zhang+ 2019]
- Placement of normalization: pre-norm versus post-norm [Xiong+ 2020]
- MLP: dense, mixture of experts [Shazeer+ 2017]
- Attention: full, sliding window, linear [Jiang+ 2023][Katharopoulos+ 2020]
- Lower-dimensional attention: group-query attention (GQA), multi-head latent attention (MLA) [Ainslie+ 2023][DeepSeek-AI+ 2024]
- State-space models: Hyena [Poli+ 2023]

274

275 Training

- Optimizer (e.g., AdamW, Muon, SOAP) [Kingma+ 2014][Loshchilov+ 2017][Keller 2024][Vyas+ 2024]
- Learning rate schedule (e.g., cosine, WSD) [Loshchilov+ 2016][Hu+ 2024]
- Batch size (e.g., critical batch size) [McCandlish+ 2018]
- Regularization (e.g., dropout, weight decay)
- Hyperparameters (number of heads, hidden dimension): grid search

281

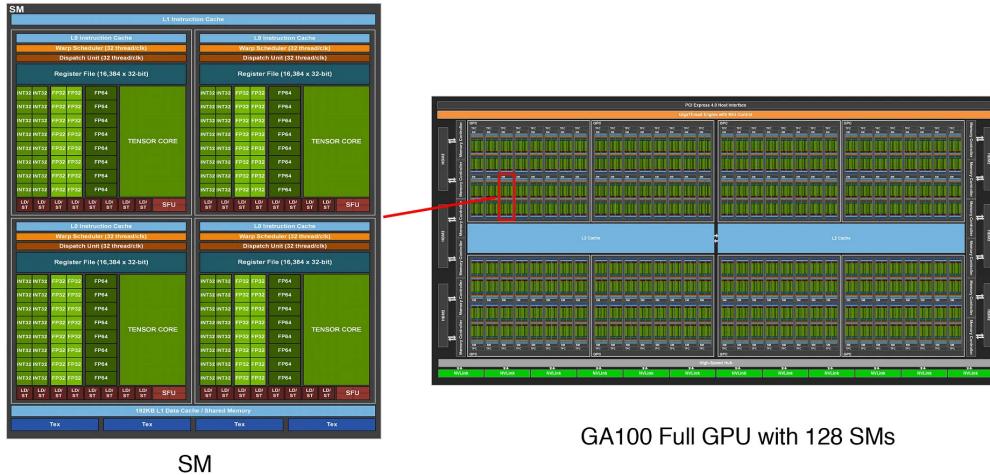
282 Assignment 1

- 283 [GitHub][PDF]
 284 • Implement BPE tokenizer
 285 • Implement Transformer, cross-entropy loss, AdamW optimizer, training loop
 286 • Train on TinyStories and OpenWebText
 287 • Leaderboard: minimize OpenWebText perplexity given 90 minutes on a H100 [last year's leaderboard]

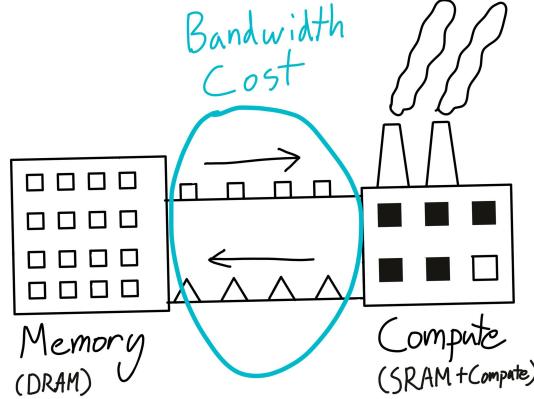
289
 290 `def systems():`
 291 Goal: squeeze the most out of the hardware
 292 Components: kernels, parallelism, inference

294 Kernels

295 What a GPU (A100) looks like:



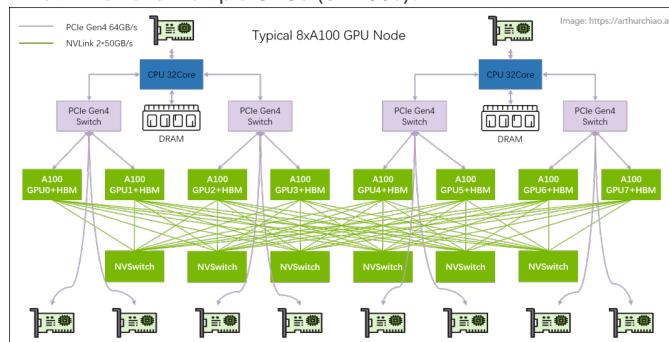
297 Analogy: warehouse : DRAM :: factory : SRAM
 298



299 Trick: organize computation to maximize utilization of GPUs by minimizing data movement
 300 Write kernels in CUDA/Triton/CUTLASS/ThunderKittens

302 Parallelism

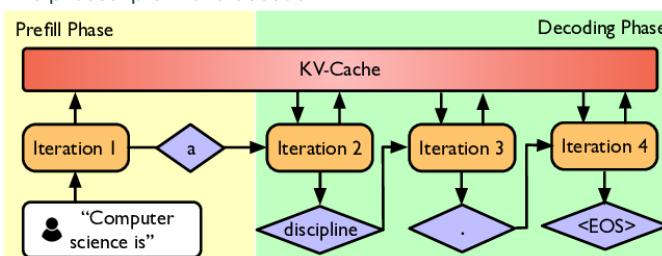
303 What if we have multiple GPUs (8 A100s)?



305 Data movement between GPUs is even slower, but same 'minimize data movement' principle holds
 306 Use collective operations (e.g., gather, reduce, all-reduce)
 307 Shard (parameters, activations, gradients, optimizer states) across GPUs
 308 How to split computation: {data,tensor,pipeline,sequence} parallelism
 309

310 Inference

311 Goal: generate tokens given a prompt (needed to actually use models!)
 312 Inference is also needed for reinforcement learning, test-time compute, evaluation
 313 Globally, inference compute (every use) exceeds training compute (one-time cost)
 314 Two phases: prefill and decode



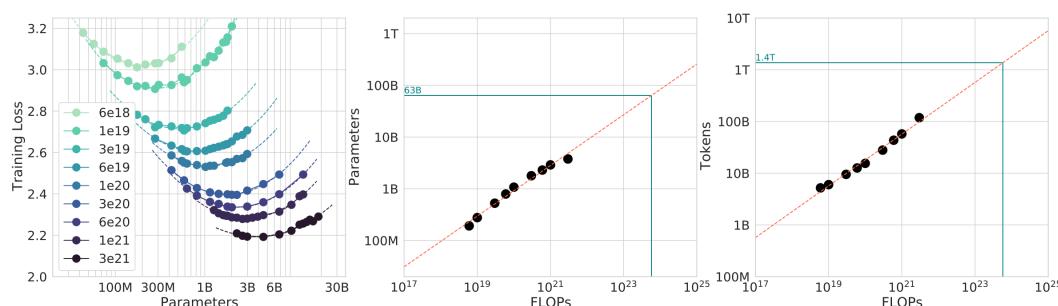
315
 316 Prefill (similar to training): tokens are given, can process all at once (compute-bound)
 317 Decode: need to generate one token at a time (memory-bound)
 318 Methods to speed up decoding:
 319 • Use cheaper model (via model pruning, quantization, distillation)
 320 • Speculative decoding: use a cheaper "draft" model to generate multiple tokens, then use the full model to score in parallel (exact decoding!)
 321 • Systems optimizations: KV caching, batching
 322

323 Assignment 2

324 [\[GitHub from 2024\]](#)[\[PDF from 2024\]](#)
 325 • Implement a fused RMSNorm kernel in Triton
 326 • Implement distributed data parallel training
 327 • Implement optimizer state sharding
 328 • Benchmark and profile the implementations
 329

330
 331 `def scaling_laws():`

332 Goal: do experiments at small scale, predict hyperparameters/loss at large scale
 333 Question: given a FLOPs budget (C), use a bigger model (N) or train on more tokens (D)?
 334 Compute-optimal scaling laws: [\[Kaplan+ 2020\]](#)[\[Hoffmann+ 2022\]](#)
 335



336 TL;DR: $D^* = 20N^*$ (e.g., 1.4B parameter model should be trained on 28B tokens)
 337 But this doesn't take into account inference costs!

338

339 Assignment 3

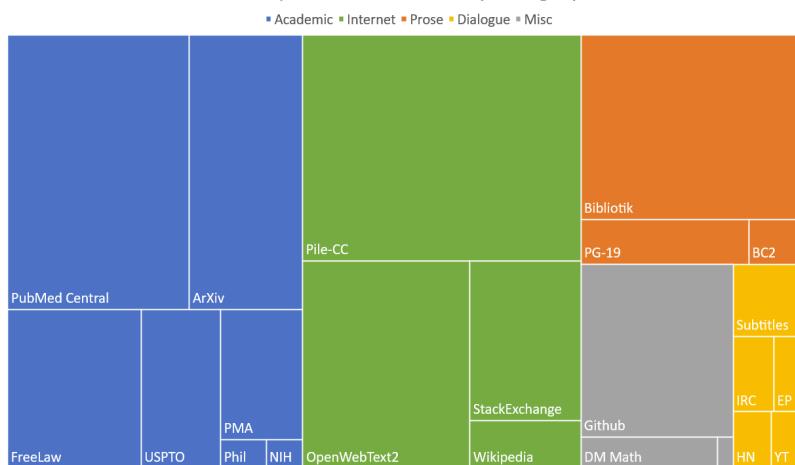
340 [\[GitHub from 2024\]](#)[\[PDF from 2024\]](#)
 341 • We define a training API (hyperparameters -> loss) based on previous runs
 342 • Submit "training jobs" (under a FLOPs budget) and gather data points
 343 • Fit a scaling law to the data points
 344 • Submit predictions for scaled up hyperparameters

```

345 • Leaderboard: minimize loss given FLOPs budget
346
347
348 def data():
349 Question: What capabilities do we want the model to have?
350 Multilingual? Code? Math?
351

```

Composition of the Pile by Category



```

352
353 Evaluation
354 • Perplexity: textbook evaluation for language models
355 • Standardized testing (e.g., MMLU, HellaSwag, GSM8K)
356 • Instruction following (e.g., AlpacaEval, IFEval, WildBench)
357 • Scaling test-time compute: chain-of-thought, ensembling
358 • LM-as-a-judge: evaluate generative tasks
359 • Full system: RAG, agents
360

```

361 Data curation

```

362 • Data does not just fall from the sky.
363 look_at_web_data()
364 • Sources: webpages crawled from the Internet, books, arXiv papers, GitHub code, etc.
365 • Appeal to fair use to train on copyright data? [Henderson+ 2023]
366 • Might have to license data (e.g., Google with Reddit data) [article]
367 • Formats: HTML, PDF, directories (not text!)
368

```

369 Data processing

```

370 • Transformation: convert HTML/PDF to text (preserve content, some structure, rewriting)
371 • Filtering: keep high quality data, remove harmful content (via classifiers)
372 • Deduplication: save compute, avoid memorization; use Bloom filters or MinHash
373

```

374 Assignment 4

```

375 [GitHub from 2024][PDF from 2024]
376 • Convert Common Crawl HTML to text
377 • Train classifiers to filter for quality and harmful content
378 • Deduplication using MinHash
379 • Leaderboard: minimize perplexity given token budget
380
381
382 def look_at_web_data():
383     urls = get_common_crawl_urls()[:3] # @inspect urls
384     documents = list(read_common_crawl(urls[1], limit=300))
385     random.seed(40)
386     random.shuffle(documents)

```

```
387     documents = markdownify_documents(documents[:10])
388     write_documents(documents, "var/sample-documents.txt")
389     [sample documents]
390     It's a wasteland out there! Need to really process the data.
391
392
393 def alignment():
394     So far, a base model is raw potential, very good at completing the next token.
395     Alignment makes the model actually useful.
396
397     Goals of alignment:
398     • Get the language model to follow instructions
399     • Tune the style (format, length, tone, etc.)
400     • Incorporate safety (e.g., refusals to answer harmful questions)
401
402     Two phases:
403     supervised_finetuning()
404     learning_from_feedback()
405
```

Assignment 5

```
406 [GitHub from 2024][PDF from 2024]
407     • Implement supervised fine-tuning
408     • Implement Direct Preference Optimization (DPO)
409     • Implement Group Relative Preference Optimization (GRPO)
410
411
412
413 @dataclass(frozen=True)
414 class Turn:
415     role: str
416     content: str
417
418
419 @dataclass(frozen=True)
420 class ChatExample:
421     turns: list[Turn]
422
423
424 @dataclass(frozen=True)
425 class PreferenceExample:
426     history: list[Turn]
427     response_a: str
428     response_b: str
429     chosen: str
430
431
432 def supervised_finetuning():
```

Supervised finetuning (SFT)

```
433 Instruction data: (prompt, response) pairs
434 sft_data: list[ChatExample] = [
435     ChatExample(
436         turns=[
437             Turn(role="system", content="You are a helpful assistant."),
438             Turn(role="user", content="What is 1 + 1?"),
439             Turn(role="assistant", content="The answer is 2."),
440         ],
441     ),
442 ]
443 ]
```

445 Data often involves human annotation.
446 Intuition: base model already has the skills, just need few examples to surface them. [Zhou+ 2023]
447 Supervised learning: fine-tune model to maximize $p(\text{response} | \text{prompt})$.
448
449
450 `def learning_from_feedback():`
451 Now we have a preliminary instruction following model.
452 Let's make it better without expensive annotation.
453
454 **Preference data**

455 Data: generate multiple responses using model (e.g., [A, B]) to a given prompt.
456 User provides preferences (e.g., A < B or A > B).
457 `preference_data: list[PreferenceExample] = [`
458 `PreferenceExample(`
459 `history=[`
460 `Turn(role="system", content="You are a helpful assistant."),`
461 `Turn(role="user", content="What is the best way to train a language model?"),`
462 `],`
463 `response_a="You should use a large dataset and train for a long time.",`
464 `response_b="You should use a small dataset and train for a short time.",`
465 `chosen="a",`
466 `)`
467 `]`
468
469

Verifiers

- 470 • Formal verifiers (e.g., for code, math)
471 • Learned verifiers: train against an LM-as-a-judge

Algorithms

- 474 • Proximal Policy Optimization (PPO) from reinforcement learning [Schulman+ 2017][Ouyang+ 2022]
475 • Direct Policy Optimization (DPO): for preference data, simpler [Rafailov+ 2023]
476 • Group Relative Preference Optimization (GRPO): remove value function [Shao+ 2024]

477
478
479 #####
480 # Tokenization
481
482 # https://github.com/openai/tiktoken/blob/main/tiktoken_ext/openai_public.py#L23
483 GPT2_TOKENIZER_REGEX = \
484 r'''^(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^s\p{L}\p{N}]+|\s+(?!S)|\s+'''
485
486
487 `def tokenization():`

488 This unit was inspired by Andrej Karpathy's video on tokenization; check it out! [video]

489
490 `intro_to_tokenization()`
491 `tokenization_examples()`
492 `character_tokenizer()`
493 `byte_tokenizer()`
494 `word_tokenizer()`
495 `bpe_tokenizer()`
496

Summary

- 498 • Tokenizer: strings <-> tokens (indices)
499 • Character-based, byte-based, word-based tokenization highly suboptimal
500 • BPE is an effective heuristic that looks at corpus statistics
501 • Tokenization is a necessary evil, maybe one day we'll just do it from bytes...

```
502
503 @dataclass(frozen=True)
504 class BPETokenizerParams:
505     """All you need to specify a BPETokenizer."""
506     vocab: dict[int, bytes]      # index -> bytes
507     merges: dict[tuple[int, int], int] # index1,index2 -> new_index
508
509
510
511 class CharacterTokenizer(Tokenizer):
512     """Represent a string as a sequence of Unicode code points."""
513     def encode(self, string: str) -> list[int]:
514         return list(map(ord, string))
515
516     def decode(self, indices: list[int]) -> str:
517         return "".join(map(chr, indices))
518
519
520 class ByteTokenizer(Tokenizer):
521     """Represent a string as a sequence of bytes."""
522     def encode(self, string: str) -> list[int]:
523         string_bytes = string.encode("utf-8") # @inspect string_bytes
524         indices = list(map(int, string_bytes)) # @inspect indices
525         return indices
526
527     def decode(self, indices: list[int]) -> str:
528         string_bytes = bytes(indices) # @inspect string_bytes
529         string = string_bytes.decode("utf-8") # @inspect string
530         return string
531
532
533 def merge(indices: list[int], pair: tuple[int, int], new_index: int) -> list[int]: # @inspect indices, @inspect pair,
@inspect new_index
534     """Return `indices`, but with all instances of `pair` replaced with `new_index`."""
535     new_indices = [] # @inspect new_indices
536     i = 0 # @inspect i
537     while i < len(indices):
538         if i + 1 < len(indices) and indices[i] == pair[0] and indices[i + 1] == pair[1]:
539             new_indices.append(new_index)
540             i += 2
541         else:
542             new_indices.append(indices[i])
543             i += 1
544     return new_indices
545
546
547 class BPETokenizer(Tokenizer):
548     """BPE tokenizer given a set of merges and a vocabulary."""
549     def __init__(self, params: BPETokenizerParams):
550         self.params = params
551
552     def encode(self, string: str) -> list[int]:
553         indices = list(map(int, string.encode("utf-8"))) # @inspect indices
554         # Note: this is a very slow implementation
555         for pair, new_index in self.params.merges.items(): # @inspect pair, @inspect new_index
556             indices = merge(indices, pair, new_index)
557         return indices
558
559     def decode(self, indices: list[int]) -> str:
```

```
560     bytes_list = list(map(self.params.vocab.get, indices)) # @inspect bytes_list
561     string = b"".join(bytes_list).decode("utf-8") # @inspect string
562     return string
563
564
565 def get_compression_ratio(string: str, indices: list[int]) -> float:
566     """Given `string` that has been tokenized into `indices`, ."""
567     num_bytes = len(bytes(string, encoding="utf-8")) # @inspect num_bytes
568     num_tokens = len(indices) # @inspect num_tokens
569     return num_bytes / num_tokens
570
571
572 def get_gpt2_tokenizer():
573     # Code: https://github.com/openai/tiktoken
574     # You can use cl100k_base for the gpt3.5-turbo or gpt4 tokenizer
575     return tiktoken.get_encoding("gpt2")
576
577
578 def intro_to_tokenization():
579     Raw text is generally represented as Unicode strings.
580     string = "Hello, 🌎! 你好!"
581
582     A language model places a probability distribution over sequences of tokens (usually represented by integer indices).
583     indices = [15496, 11, 995, 0]
584
585     So we need a procedure that encodes strings into tokens.
586     We also need a procedure that decodes tokens back into strings.
587     A Tokenizer is a class that implements the encode and decode methods.
588     The vocabulary size is number of possible tokens (integers).
589
590
591 def tokenization_examples():
592     To get a feel for how tokenizers work, play with this interactive site
```

Observations

- A word and its preceding space are part of the same token (e.g., " world").
- A word at the beginning and in the middle are represented differently (e.g., "hello hello").
- Numbers are tokenized into every few digits.

```
599 Here's the GPT-2 tokenizer from OpenAI (tiktoken) in action.
600 tokenizer = get_gpt2_tokenizer()
601 string = "Hello, 🌎! 你好!" # @inspect string
602
603 Check that encode() and decode() roundtrip:
604 indices = tokenizer.encode(string) # @inspect indices
605 reconstructed_string = tokenizer.decode(indices) # @inspect reconstructed_string
606 assert string == reconstructed_string
607 compression_ratio = get_compression_ratio(string, indices) # @inspect compression_ratio
608
609
610 def character_tokenizer():
```

Character-based tokenization

```
611
612
613 A Unicode string is a sequence of Unicode characters.
614 Each character can be converted into a code point (integer) via ord.
615 assert ord("a") == 97
616 assert ord("🌐") == 127757
```

```
617 It can be converted back via chr.  
618 assert chr(97) == "a"  
619 assert chr(127757) == "🌐"  
620  
621 Now let's build a Tokenizer and make sure it round-trips:  
622 tokenizer = CharacterTokenizer()  
623 string = "Hello, 🌐! 你好!" # @inspect string  
624 indices = tokenizer.encode(string) # @inspect indices  
625 reconstructed_string = tokenizer.decode(indices) # @inspect reconstructed_string  
626 assert string == reconstructed_string  
627  
628 There are approximately 150K Unicode characters. \[Wikipedia\]  
629 vocabulary_size = max(indices) + 1 # This is a lower bound @inspect vocabulary_size  
630 Problem 1: this is a very large vocabulary.  
631 Problem 2: many characters are quite rare (e.g., 🌐, which is inefficient use of the vocabulary.  
632 compression_ratio = get_compression_ratio(string, indices) # @inspect compression_ratio  
633  
634  
635 def byte_tokenizer():
```

Byte-based tokenization

```
636 Unicode strings can be represented as a sequence of bytes, which can be represented by integers between 0  
637 and 255.  
638 The most common Unicode encoding is UTF-8
```

```
639 Some Unicode characters are represented by one byte:
```

```
640 assert bytes("a", encoding="utf-8") == b"a"  
641 Others take multiple bytes:  
642 assert bytes("🌐", encoding="utf-8") == b"\xf0\x9f\x8c\x8d"
```

```
643 Now let's build a Tokenizer and make sure it round-trips:
```

```
644 tokenizer = ByteTokenizer()  
645 string = "Hello, 🌐! 你好!" # @inspect string  
646 indices = tokenizer.encode(string) # @inspect indices  
647 reconstructed_string = tokenizer.decode(indices) # @inspect reconstructed_string  
648 assert string == reconstructed_string
```

```
649 The vocabulary is nice and small: a byte can represent 256 values.
```

```
650 vocabulary_size = 256 # @inspect vocabulary_size  
651 What about the compression rate?  
652 compression_ratio = get_compression_ratio(string, indices) # @inspect compression_ratio  
653 assert compression_ratio == 1
```

```
654 The compression ratio is terrible, which means the sequences will be too long.
```

```
655 Given that the context length of a Transformer is limited (since attention is quadratic), this is not looking  
656 great...
```

```
657  
658
```

```
659 def word_tokenizer():
```

Word-based tokenization

```
660 Another approach (closer to what was done classically in NLP) is to split strings into words.  
661 string = "I'll say supercalifragilisticexpialidocious!"
```

```
662 segments = regex.findall(r"\w+|.", string) # @inspect segments  
663 This regular expression keeps all alphanumeric characters together (words).
```

```
664 Here is a fancier version:  
665 pattern = GPT2_TOKENIZER_REGEX # @inspect pattern
```

```
673 segments = regex.findall(pattern, string) # @inspect segments
674
675 To turn this into a Tokenizer, we need to map these segments into integers.
676 Then, we can build a mapping from each segment into an integer.
677
678 But there are problems:
679 • The number of words is huge (like for Unicode characters).
680 • Many words are rare and the model won't learn much about them.
681 • This doesn't obviously provide a fixed vocabulary size.
682
683 New words we haven't seen during training get a special UNK token, which is ugly and can mess up perplexity
calculations.
```

```
684
685 vocabulary_size = "Number of distinct segments in the training data"
686 compression_ratio = get_compression_ratio(string, segments) # @inspect compression_ratio
687
688
689 def bpe_tokenizer():
```

Byte Pair Encoding (BPE)

[\[Wikipedia\]](#)

The BPE algorithm was introduced by Philip Gage in 1994 for data compression. [\[article\]](#)

It was adapted to NLP for neural machine translation. [\[Sennrich+ 2015\]](#)

(Previously, papers had been using word-based tokenization.)

BPE was then used by GPT-2. [\[Radford+ 2019\]](#)

Basic idea: *train* the tokenizer on raw text to automatically determine the vocabulary.

Intuition: common sequences of characters are represented by a single token, rare sequences are represented by many tokens.

The GPT-2 paper used word-based tokenization to break up the text into initial segments and run the original BPE algorithm on each segment.

Sketch: start with each byte as a token, and successively merge the most common pair of adjacent tokens.

Training the tokenizer

```
704 string = "the cat in the hat" # @inspect string
705 params = train_bpe(string, num_merges=3)
706
```

Using the tokenizer

Now, given a new text, we can encode it.

```
709 tokenizer = BPETokenizer(params)
710 string = "the quick brown fox" # @inspect string
711 indices = tokenizer.encode(string) # @inspect indices
712 reconstructed_string = tokenizer.decode(indices) # @inspect reconstructed_string
713 assert string == reconstructed_string
714
```

In Assignment 1, you will go beyond this in the following ways:

- encode() currently loops over all merges. Only loop over merges that matter.
- Detect and preserve special tokens (e.g., <|endoftext|>).
- Use pre-tokenization (e.g., the GPT-2 tokenizer regex).
- Try to make the implementation as fast as possible.

```
720
721
722 def train_bpe(string: str, num_merges: int) -> BPETokenizerParams: # @inspect string, @inspect num_merges
723     Start with the list of bytes of string.
724     indices = list(map(int, string.encode("utf-8"))) # @inspect indices
725     merges: dict[tuple[int, int], int] = {} # index1, index2 => merged index
726     vocab: dict[int, bytes] = {x: bytes([x]) for x in range(256)} # index -> bytes
```

```
727
728     for i in range(num_merges):
729         Count the number of occurrences of each pair of tokens
730         counts = defaultdict(int)
731         for index1, index2 in zip(indices, indices[1:]): # For each adjacent pair
732             counts[(index1, index2)] += 1 # @inspect counts
733
734     Find the most common pair.
735     pair = max(counts, key=counts.get) # @inspect pair
736     index1, index2 = pair
737
738     Merge that pair.
739     new_index = 256 + i # @inspect new_index
740     merges[pair] = new_index # @inspect merges
741     vocab[new_index] = vocab[index1] + vocab[index2] # @inspect vocab
742     indices = merge(indices, pair, new_index) # @inspect indices
743
744     return BPETokenizerParams(vocab=vocab, merges=merges)
745
746
747 if __name__ == "__main__":
748     main()
```