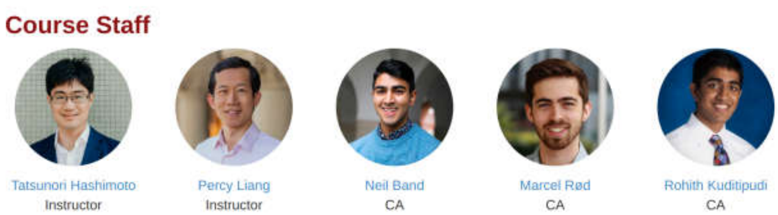
lecture_01.py

```
import regex
from abc import ABC
from dataclasses import dataclass
from collections import defaultdict
import random

from execute_util import link, image, text
from lecture_util import article_link, x_link, youtube_link
from references import gpt_3, gpt4, shannon1950, bengio2003, susketver2014, \
    bahdanau2015_attention, transformer_2017, gpt2, t5, kaplan_scaling_laws_2020, \
    the_pile, gpt_j, opt_175b, bloom, palm, chinchilla, llama, mistral_7b, \
    instruct_gpt, dpo, adamw2017, lima, deepseek_v3, adam2014, grpo, ppo2017, muon, \
    large_batch_training_2018, wsd_2024, cosine_learning_rate_2017, olmo_7b, moe_2017, \
    megatron_lm_2019, shazeer_2020, elmo, bert, qwen_2_5, deepseek_r1, moe_2017, \
    rms_norm_2019, rope_2021, soap, gqa, mla, deepseek_67b, deepseek_v2, brants2007, \
    layernorm_2016, pre_post_norm_2020, llama2, llama3, olmo2, \
    megabyte, byt5, blt, tfree, sennrich_2016, zero_2019, gpipe_2018
from data import get_common_crawl_urls, read_common_crawl, write_documents, markdownify_documents
from model_util import query_gpt4o

import tiktoken

def main():
    welcome()
    why_this_course_exists()
    current_landscape()

    what_is_this_program()

    course_logistics()
    course_components()

    tokenization()
```

Next time: PyTorch building blocks, resource accounting

```
def welcome():
```

CS336: Language Models From Scratch (Spring 2025)



This is the second offering of CS336.
Stanford edition has grown by 50%.
Lectures will be posted on YouTube and be made available to the whole world.

```
def why_this_course_exists():
```

Why did we make this course?

Let's ask GPT-4 [OpenAI+ 2023]

```
    response = query_gpt4o(prompt="Why teach a course on building language models from scratch? Answer in one sentence.")  # @inspect response
```

Problem: researchers are becoming **disconnected** from the underlying technology.
8 years ago, researchers would implement and train their own models.
6 years ago, researchers would download a model (e.g., BERT) and fine-tune it.
Today, researchers just prompt a proprietary model (e.g., GPT-4/Claude/Gemini).

Moving up levels of abstractions boosts productivity, but

- These abstractions are leaky (in contrast to programming languages or operating systems).
- There is still fundamental research to be done that require tearing up the stack.

Full understanding of this technology is necessary for **fundamental research**.

This course: **understanding via building**
But there's one small problem...

The industrialization of language models

GPT-4 supposedly has 1.8T parameters. [article]
GPT-4 supposedly cost $100M to train. [article]
xAI builds cluster with 200,000 H100s to train Grok. [article]
Stargate (OpenAI, NVIDIA, Oracle) invests $500B over 4 years. [article]

Also, there are no public details on how frontier models are built.
From the GPT-4 technical report [OpenAI+ 2023]:

2 Scope and Limitations of this Technical Report

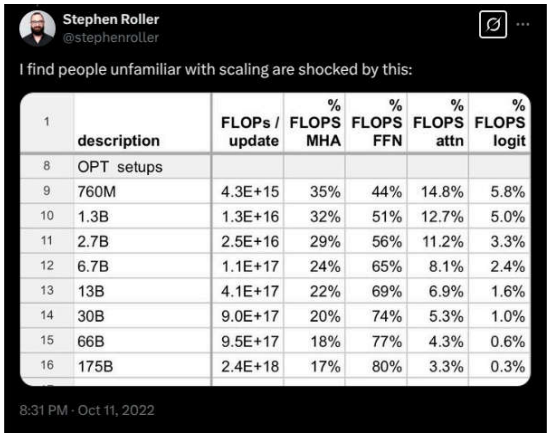
This report focuses on the capabilities, limitations, and safety properties of GPT-4. GPT-4 is a Transformer-style model [39] pre-trained to predict the next token in a document, using both publicly available data (such as internet data) and data licensed from third-party providers. The model was then 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.

More is different

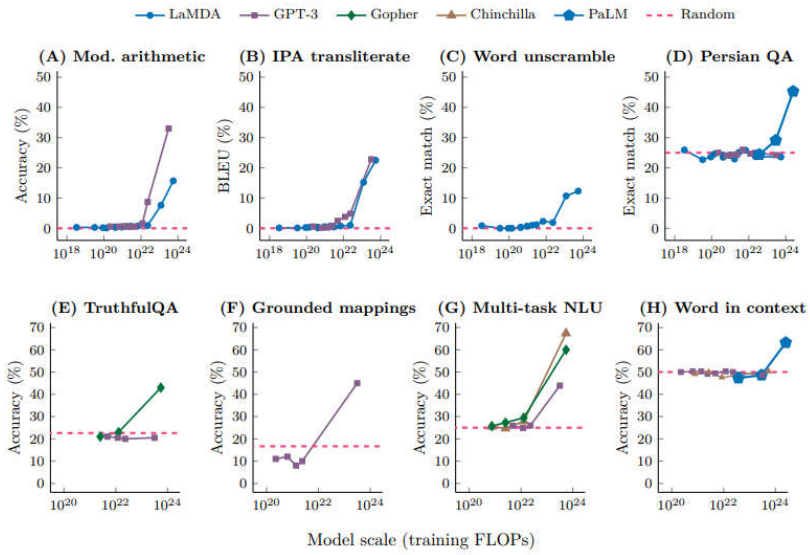
Frontier models are out of reach for us.
But building small language models (<1B parameters in this class) might not be representative of large language models.

Example 1: fraction of FLOPs spent in attention versus MLP changes with scale. [X]



	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%

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



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

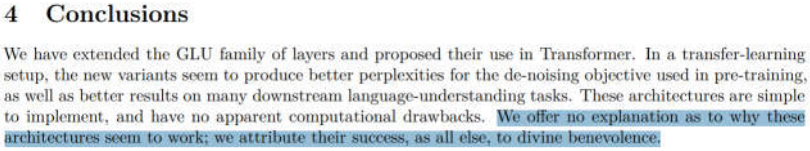
There are three types of knowledge:

- **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

We can teach mechanics and mindset (these do transfer).
We can only partially teach intuitions (do not necessarily transfer across scales).

Intuitions? 🤷

Some design decisions are simply not (yet) justifiable and just come from experimentation.
Example: Noam Shazeer paper that introduced SwiGLU [Shazeer 2020]



4 Conclusions

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.

The bitter lesson

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

accuracy = efficiency x resources

In fact, efficiency is way more important at larger scale (can't afford to be wasteful).
[Hernandez+ 2020] showed 44x algorithmic efficiency on ImageNet between 2012 and 2019

Framing: what is the best model one can build given a certain compute and data budget?
In other words, **maximize efficiency**!

```
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/claude-3-7-sonnet
- xAI's Grok 3 https://x.ai/news/grok-3
- Google's Gemini 2.5 https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/
- Meta's Llama 3.3 https://ai.meta.com/blog/meta-llama-3/
- DeepSeek's r1 [DeepSeek-AI+ 2025]
- Alibaba's Qwen 2.5 Max https://qwenlm.github.io/blog/qwen2.5-max/
- Tencent's Hunyuan-T1 https://tencent.github.io/llm.hunyuan.T1/README_EN.html

```
def what_is_this_program():
```

This is an *executable lecture*, a program whose execution delivers the content of a lecture.
Executable lectures make it possible to:

- view and run code (since everything is code!),

```
total = 0  # @inspect total
for x in [1, 2, 3]:  # @inspect x
    total += x  # @inspect total
```

- see the hierarchical structure of the lecture, and
- jump to definitions and concepts: supervised_finetuning

```
def course_logistics():
```

All information online: https://stanford-cs336.github.io/spring2025/

This is a 5-unit class.
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.*

Why you should take this course

- You have an obsessive need to understand how things work.
- You want to build up your research engineering muscles.

Why you should not take this course

- You actually want to get research done this quarter.

(Talk to your advisor.)

- You are interested in learning about the hottest new techniques in AI (e.g., multimodality, RAG, etc.).

(You should take a seminar class for that.)

- You want to get good results on your own application domain.

(You should just prompt or fine-tune an existing model.)

How you can follow along at home

- All lecture materials and assignments will be posted online, so feel free to follow on your own.
- Lectures are recorded via CGOE, formally SCPD and be made available on YouTube (with some lag).
- We plan to offer this class again next year.

Assignments

- 5 assignments (basics, systems, scaling laws, data, alignment).
- No scaffolding code, but we provide unit tests and adapter interfaces to help you check correctness.
- Implement locally to test for correctness, then run on cluster for benchmarking (accuracy and speed).
- Leaderboard for some assignments (minimize perplexity given training budget).
- AI tools (e.g., CoPilot, Cursor) can take away from learning, so use at your own risk.

Cluster

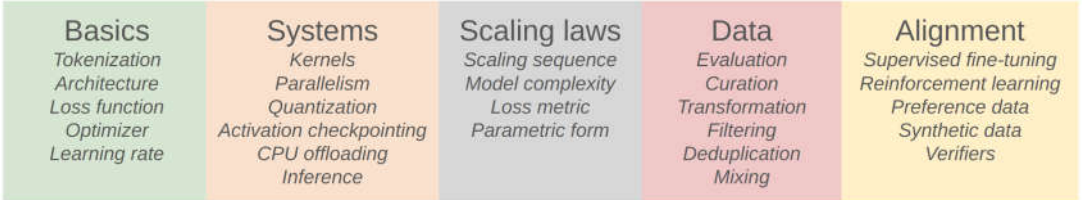
- Thanks to Together AI for providing a compute cluster. 🙏
- Please read the guide on how to use the cluster.
- Start your assignments early, since the cluster will fill up close to the deadline!

```
def course_components():
```

It's all about efficiency

Resources: data + hardware (compute, memory, communication bandwidth)
How do you train the best model given a fixed set of resources?
Example: given a Common Crawl dump and 32 H100s for 2 weeks, what should you do?

Design decisions:



Overview of the course

```
    basics()
    systems()
    scaling_laws()
    data()
    alignment()
```

Efficiency drives design decisions

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

- 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

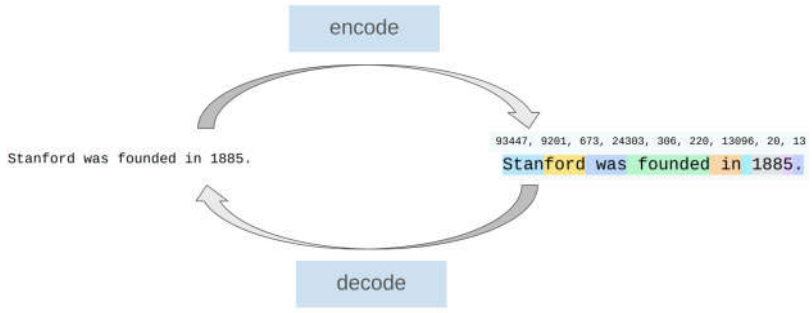
Tomorrow, we will become data-constrained...

```
class Tokenizer(ABC):
    """Abstract interface for a tokenizer."""
    def encode(self, string: str) -> list[int]:
        raise NotImplementedError

    def decode(self, indices: list[int]) -> str:
        raise NotImplementedError


def basics():
```

Goal: get a basic version of the full pipeline working
Components: tokenization, model architecture, training

Tokenization

Tokenizers convert between strings and sequences of integers (tokens)



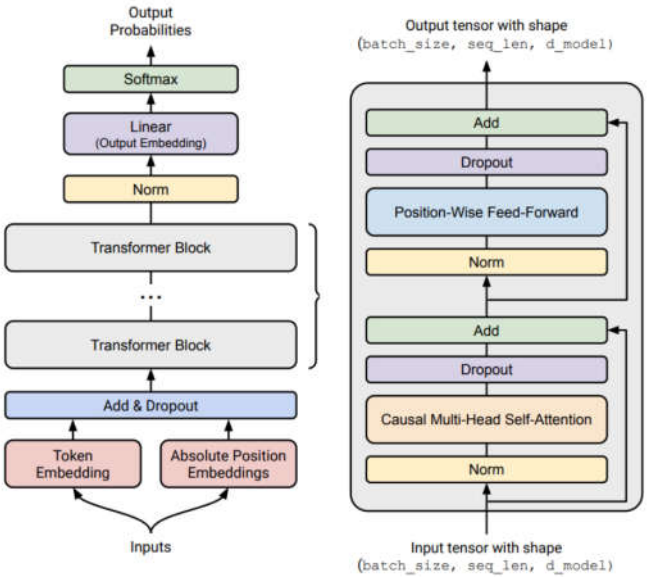
Intuition: break up string into popular segments

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

Tokenizer-free approaches: [Xue+ 2021][Yu+ 2023][Pagnoni+ 2024][Deiseroth+ 2024]
Use bytes directly, promising, but have not yet been scaled up to the frontier.

Architecture

Starting point: original Transformer [Vaswani+ 2017]



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]

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

Assignment 1

[GitHub][PDF]

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

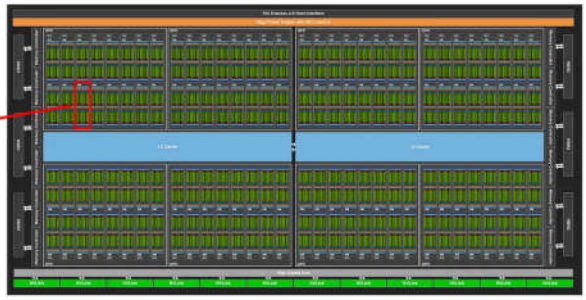
```
def systems():
```

Goal: squeeze the most out of the hardware

Components: kernels, parallelism, inference

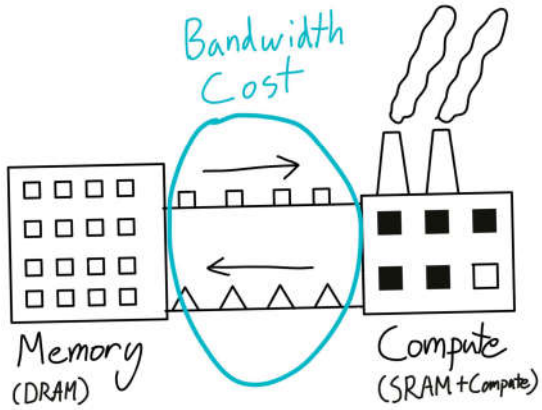
Kernels

What a GPU (A100) looks like:



GA100 Full GPU with 128 SMs

Analogy: warehouse : DRAM :: factory : SRAM

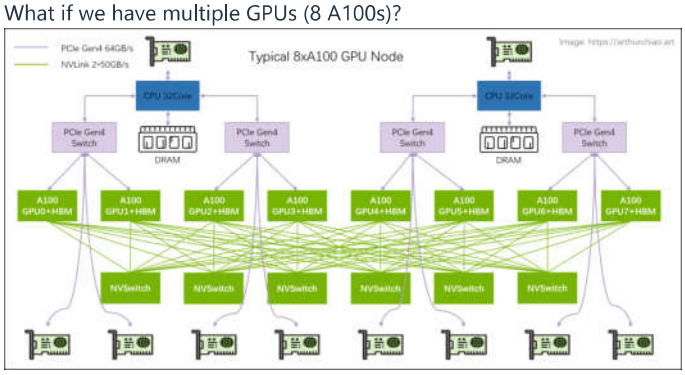


Trick: organize computation to maximize utilization of GPUs by minimizing data movement

Write kernels in CUDA/**Triton**/CUTLASS/ThunderKittens

Parallelism

What if we have multiple GPUs (8 A100s)?



Data movement between GPUs is even slower, but same 'minimize data movement' principle holds

Use collective operations (e.g., gather, reduce, all-reduce)

Shard (parameters, activations, gradients, optimizer states) across GPUs

How to split computation: {data,tensor,pipeline,sequence} parallelism

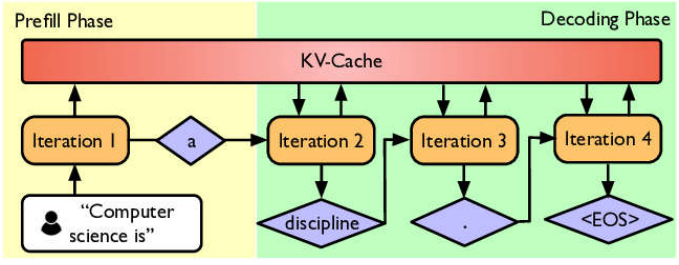
Inference

Goal: generate tokens given a prompt (needed to actually use models!)

Inference is also needed for reinforcement learning, test-time compute, evaluation

Globally, inference compute (every use) exceeds training compute (one-time cost)

Two phases: prefill and decode



Prefill (similar to training): tokens are given, can process all at once (compute-bound)
Decode: need to generate one token at a time (memory-bound)
Methods to speed up decoding:

- Use cheaper model (via model pruning, quantization, distillation)
- Speculative decoding: use a cheaper "draft" model to generate multiple tokens, then use the full model to score in parallel (exact decoding!)
- Systems optimizations: KV caching, batching

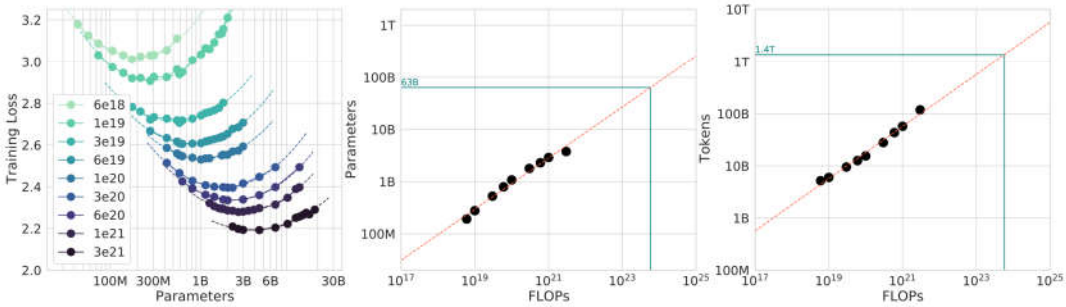
Assignment 2

[GitHub from 2024][PDF from 2024]

- Implement a fused RMSNorm kernel in Triton
- Implement distributed data parallel training
- Implement optimizer state sharding
- Benchmark and profile the implementations

```
def scaling_laws():
```

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



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

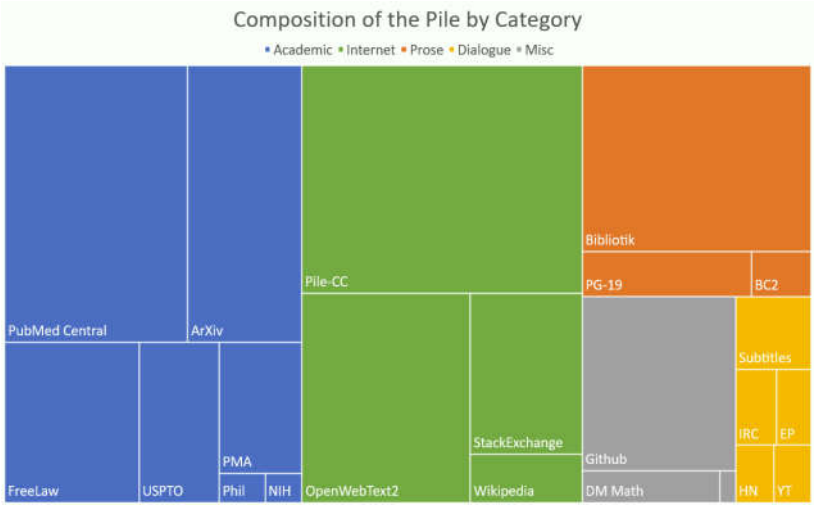
Assignment 3

[GitHub from 2024][PDF from 2024]

- We define a training API (hyperparameters -> loss) based on previous runs
- Submit "training jobs" (under a FLOPs budget) and gather data points
- Fit a scaling law to the data points
- Submit predictions for scaled up hyperparameters
- Leaderboard: minimize loss given FLOPs budget

```
def data():
```

Question: What capabilities do we want the model to have?
Multilingual? Code? Math?



Evaluation

- Perplexity: textbook evaluation for language models
- Standardized testing (e.g., MMLU, HellaSwag, GSM8K)
- Instruction following (e.g., AlpacaEval, IFEval, WildBench)
- Scaling test-time compute: chain-of-thought, ensembling
- LM-as-a-judge: evaluate generative tasks
- Full system: RAG, agents

Data curation

- Data does not just fall from the sky.

```
look_at_web_data()
```

- Sources: webpages crawled from the Internet, books, arXiv papers, GitHub code, etc.
- Appeal to fair use to train on copyright data? [Henderson+ 2023]
- Might have to license data (e.g., Google with Reddit data) [article]
- Formats: HTML, PDF, directories (not text!)

Data processing

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

Assignment 4

[GitHub from 2024][PDF from 2024]

- Convert Common Crawl HTML to text
- Train classifiers to filter for quality and harmful content
- Deduplication using MinHash
- Leaderboard: minimize perplexity given token budget

```
def look_at_web_data():
    urls = get_common_crawl_urls()[:3]  # @inspect urls
    documents = list(read_common_crawl(urls[1], limit=300))
    random.seed(40)
    random.shuffle(documents)
    documents = markdownify_documents(documents[:10])
    write_documents(documents, "var/sample-documents.txt")
```

[sample documents]

It's a wasteland out there! Need to really process the data.

```
def alignment():
```

So far, a **base model** is raw potential, very good at completing the next token.
Alignment makes the model actually useful.

Goals of alignment:

- Get the language model to follow instructions
- Tune the style (format, length, tone, etc.)
- Incorporate safety (e.g., refusals to answer harmful questions)

Two phases:

```
supervised_finetuning()
learning_from_feedback()
```

Assignment 5

[GitHub from 2024][PDF from 2024]

- Implement supervised fine-tuning
- Implement Direct Preference Optimization (DPO)
- Implement Group Relative Preference Optimization (GRPO)

```
@dataclass(frozen=True)
class Turn:
    role: str
```

```
    content: str


@dataclass(frozen=True)
class ChatExample:
    turns: list[Turn]


@dataclass(frozen=True)
class PreferenceExample:
    history: list[Turn]
    response_a: str
    response_b: str
    chosen: str


def supervised_finetuning():
```

Supervised finetuning (SFT)

Instruction data: (prompt, response) pairs

```
    sft_data: list[ChatExample] = [
        ChatExample(
            turns=[
                Turn(role="system", content="You are a helpful assistant."),
                Turn(role="user", content="What is 1 + 1?"),
                Turn(role="assistant", content="The answer is 2."),
            ],
        ),
    ]
```

Data often involves human annotation.

Intuition: base model already has the skills, just need few examples to surface them. [Zhou+ 2023]

Supervised learning: fine-tune model to maximize p(response | prompt).

```
def learning_from_feedback():
```

Now we have a preliminary instruction following model.

Let's make it better without expensive annotation.

Preference data

Data: generate multiple responses using model (e.g., [A, B]) to a given prompt.

User provides preferences (e.g., A < B or A > B).

```
    preference_data: list[PreferenceExample] = [
        PreferenceExample(
            history=[
                Turn(role="system", content="You are a helpful assistant."),
                Turn(role="user", content="What is the best way to train a language model?"),
            ],
            response_a="You should use a large dataset and train for a long time.",
            response_b="You should use a small dataset and train for a short time.",
            chosen="a",
        )
    ]
```

Verifiers

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

Algorithms

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

```
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
497     ## 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
```

```
        else:
            new_indices.append(indices[i])
            i += 1
    return new_indices


class BPETokenizer(Tokenizer):
    """BPE tokenizer given a set of merges and a vocabulary."""
    def __init__(self, params: BPETokenizerParams):
        self.params = params

    def encode(self, string: str) -> list[int]:
        indices = list(map(int, string.encode("utf-8")))  # @inspect indices
        # Note: this is a very slow implementation
        for pair, new_index in self.params.merges.items():  # @inspect pair, @inspect new_index
            indices = merge(indices, pair, new_index)
        return indices

    def decode(self, indices: list[int]) -> str:
        bytes_list = list(map(self.params.vocab.get, indices))  # @inspect bytes_list
        string = b"".join(bytes_list).decode("utf-8")  # @inspect string
        return string


def get_compression_ratio(string: str, indices: list[int]) -> float:
    """Given `string` that has been tokenized into `indices`, ."""
    num_bytes = len(bytes(string, encoding="utf-8"))  # @inspect num_bytes
    num_tokens = len(indices)                       # @inspect num_tokens
    return num_bytes / num_tokens


def get_gpt2_tokenizer():
    # Code: https://github.com/openai/tiktoken
    # You can use cl100k_base for the gpt3.5-turbo or gpt4 tokenizer
    return tiktoken.get_encoding("gpt2")


def intro_to_tokenization():
```

Raw text is generally represented as Unicode strings.

```
    string = "Hello, 🌎! 你好!"
```

A language model places a probability distribution over sequences of tokens (usually represented by integer indices).

```
    indices = [15496, 11, 995, 0]
```

So we need a procedure that *encodes* strings into tokens.
We also need a procedure that *decodes* tokens back into strings.
A Tokenizer is a class that implements the encode and decode methods.
The **vocabulary size** is number of possible tokens (integers).

```


def tokenization_examples():
```

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.

Here's the GPT-2 tokenizer from OpenAI (tiktoken) in action.

```
    tokenizer = get_gpt2_tokenizer()
    string = "Hello, 🌎! 你好!"  # @inspect string
```

Check that encode() and decode() roundtrip:

```
    indices = tokenizer.encode(string)  # @inspect indices
    reconstructed_string = tokenizer.decode(indices)  # @inspect reconstructed_string
    assert string == reconstructed_string
    compression_ratio = get_compression_ratio(string, indices)  # @inspect compression_ratio


def character_tokenizer():
```

Character-based tokenization

A Unicode string is a sequence of Unicode characters.
Each character can be converted into a code point (integer) via `ord`.

```
    assert ord("a") == 97
    assert ord("🌎") == 127757
```

It can be converted back via `chr`.

```
    assert chr(97) == "a"
    assert chr(127757) == "🌎"
```

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

```
    tokenizer = CharacterTokenizer()
    string = "Hello, 🌎! 你好!"  # @inspect string
    indices = tokenizer.encode(string)  # @inspect indices
    reconstructed_string = tokenizer.decode(indices)  # @inspect reconstructed_string
    assert string == reconstructed_string
```

There are approximately 150K Unicode characters. [Wikipedia]

```
    vocabulary_size = max(indices) + 1  # This is a lower bound @inspect vocabulary_size
```

Problem 1: this is a very large vocabulary.
Problem 2: many characters are quite rare (e.g., 🌎), which is inefficient use of the vocabulary.

```
    compression_ratio = get_compression_ratio(string, indices)  # @inspect compression_ratio


def byte_tokenizer():
```

Byte-based tokenization

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

Some Unicode characters are represented by one byte:

```
    assert bytes("a", encoding="utf-8") == b"a"
```

Others take multiple bytes:

```
    assert bytes("🌎", encoding="utf-8") == b"\xf0\x9f\x8c\x8d"
```

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

```
    tokenizer = ByteTokenizer()
    string = "Hello, 🌎! 你好!"  # @inspect string
    indices = tokenizer.encode(string)  # @inspect indices
    reconstructed_string = tokenizer.decode(indices)  # @inspect reconstructed_string
    assert string == reconstructed_string
```

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

```
    vocabulary_size = 256  # @inspect vocabulary_size
```

What about the compression rate?

```
    compression_ratio = get_compression_ratio(string, indices)  # @inspect compression_ratio
    assert compression_ratio == 1
```

The compression ratio is terrible, which means the sequences will be too long.
Given that the context length of a Transformer is limited (since attention is quadratic), this is not looking great...

```
def word_tokenizer():
```

Word-based tokenization

Another approach (closer to what was done classically in NLP) is to split strings into words.

```
    string = "I'll say supercalifragilisticexpialidocious!"

    segments = regex.findall(r"\w+|.", string)  # @inspect segments
```

This regular expression keeps all alphanumeric characters together (words).

Here is a fancier version:

```
    pattern = GPT2_TOKENIZER_REGEX  # @inspect pattern
    segments = regex.findall(pattern, string)  # @inspect segments
```

To turn this into a `Tokenizer`, we need to map these segments into integers.
Then, we can build a mapping from each segment into an integer.

But there are problems:

- The number of words is huge (like for Unicode characters).
- Many words are rare and the model won't learn much about them.
- This doesn't obviously provide a fixed vocabulary size.

New words we haven't seen during training get a special UNK token, which is ugly and can mess up perplexity calculations.

```
    vocabulary_size = "Number of distinct segments in the training data"
    compression_ratio = get_compression_ratio(string, segments)  # @inspect compression_ratio


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 inital 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

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

Using the tokenizer

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

```
    tokenizer = BPETokenizer(params)
    string = "the quick brown fox"  # @inspect string
    indices = tokenizer.encode(string)  # @inspect indices
    reconstructed_string = tokenizer.decode(indices)  # @inspect reconstructed_string
    assert string == reconstructed_string
```

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.

```
def train_bpe(string: str, num_merges: int) -> BPETokenizerParams:  # @inspect string, @inspect num_merges
```

Start with the list of bytes of `string`.

```
    indices = list(map(int, string.encode("utf-8")))  # @inspect indices
    merges: dict[tuple[int, int], int] = {}  # index1, index2 => merged index
```

```
    vocab: dict[int, bytes] = {x: bytes([x]) for x in range(256)}  # index -> bytes

    for i in range(num_merges):
        Count the number of occurrences of each pair of tokens
        counts = defaultdict(int)
        for index1, index2 in zip(indices, indices[1:]):  # For each adjacent pair
            counts[(index1, index2)] += 1  # @inspect counts

        Find the most common pair.
        pair = max(counts, key=counts.get)  # @inspect pair
        index1, index2 = pair

        Merge that pair.
        new_index = 256 + i  # @inspect new_index
        merges[pair] = new_index  # @inspect merges
        vocab[new_index] = vocab[index1] + vocab[index2]  # @inspect vocab
        indices = merge(indices, pair, new_index)  # @inspect indices

    return BPETokenizerParams(vocab=vocab, merges=merges)


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