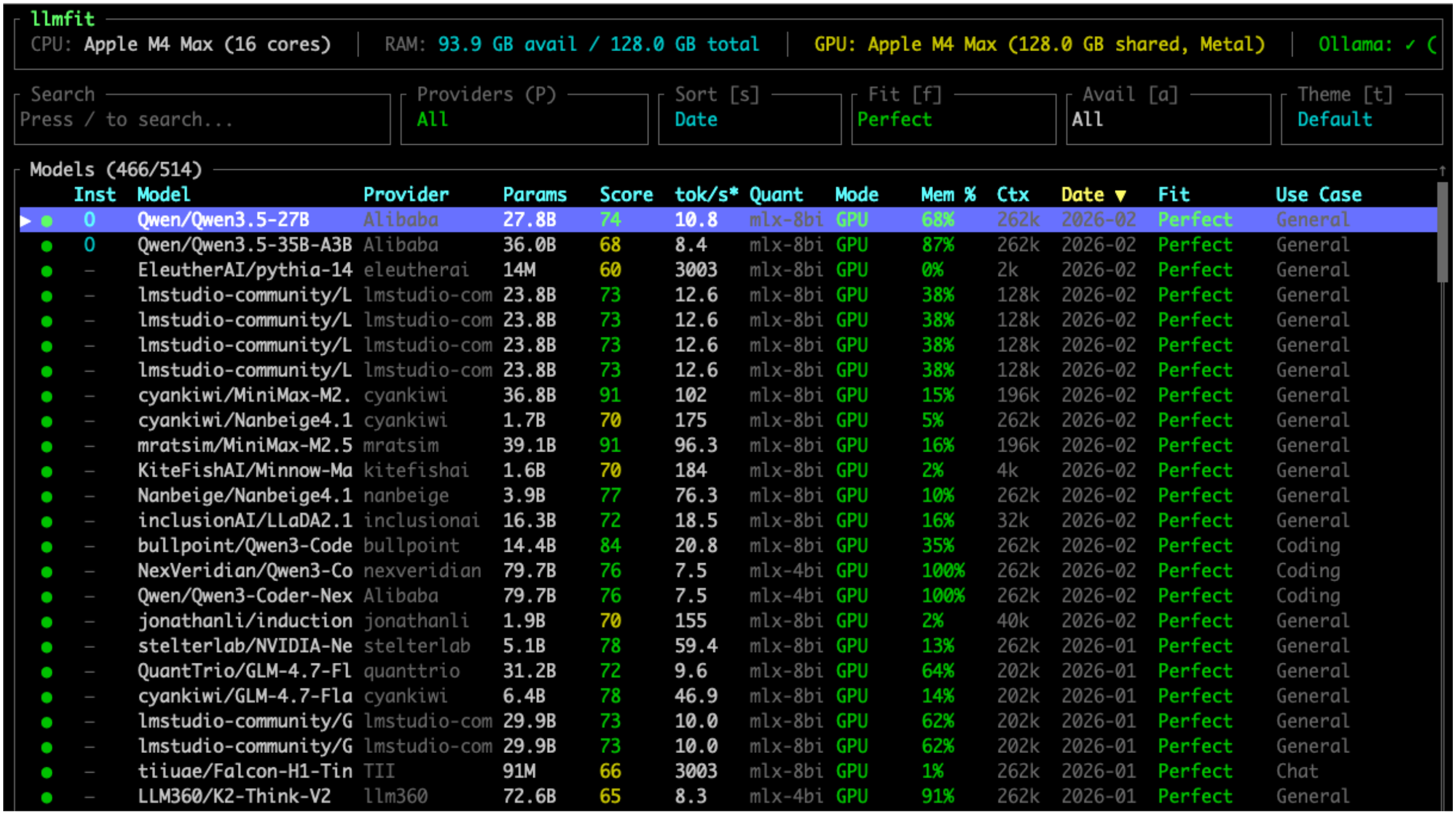
CS 668 Applied Large Language Models
Spring Semester, 2026

Doc 15 Pre-Training
Mar 3, 2026

LImfit

https://github.com/AlexsJones/llmfit

Tells you which LLMs will actually run well on your machine



```
llmfit
CPU: Apple M4 Max (16 cores) | RAM: 93.9 GB avail / 128.0 GB total | GPU: Apple M4 Max (128.0 GB shared, Metal) | Ollama: ✓ (

Search | Providers (P): All | Sort [s]: Date | Fit [f]: Perfect | Avail [a]: All | Theme [t]: Default

Plan: Qwen/Qwen3.5-27B

Model: Qwen/Qwen3.5-27B
Note: Estimate-based using current llmfit fit/speed heuristics.

Inputs (editable)
Context:    8192 tokens
Quant:      Q4_K_M
Target TPS: <none> tok/s

Minimum Hardware
VRAM: 18.4 GB   RAM: 8.0 GB   CPU: 4 cores

Recommended Hardware
VRAM: 25.9 GB   RAM: 12.0 GB   CPU: 8 cores

Run Paths
- GPU: yes  tps=21.6  fit=Marginal
- CPU offload: no  tps=-  fit=-
- CPU-only: yes  tps=3.7  fit=Marginal

Upgrade Deltas
- +0.0 GB VRAM -> Good
- +0.0 GB VRAM -> Perfect
```

Viewing the Model Structure

```
from transformers import GPT2Tokenizer, GPT2Model
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
model = GPT2Model.from_pretrained(‘gpt2')
print(model)
```

```
GPT2Model(
  (wte): Embedding(50257, 768)
  (wpe): Embedding(1024, 768)
  (drop): Dropout(p=0.1, inplace=False)
  (h): ModuleList(
    (0-11): 12 x GPT2Block(
      (ln_1): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (attn): GPT2Attention(
        (c_attn): Conv1D(nf=2304, nx=768)
        (c_proj): Conv1D(nf=768, nx=768)
        (attn_dropout): Dropout(p=0.1, inplace=False)
        (resid_dropout): Dropout(p=0.1, inplace=False)
      )
      (ln_2): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
      (mlp): GPT2MLP(
        (c_fc): Conv1D(nf=3072, nx=768)
        (c_proj): Conv1D(nf=768, nx=3072)
        (act): NewGELUActivation()
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
  )
  (ln_f): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
)
```

model.config

GPT2Config {
 "activation_function": "gelu_new",
 "add_cross_attention": false,
 "architectures": [
  "GPT2LMHeadModel"
 ],
 "attn_pdrop": 0.1,
 "bos_token_id": 50256,
 "dtype": "float32",
 "embd_pdrop": 0.1,
 "eos_token_id": 50256,
 "initializer_range": 0.02,
 "layer_norm_epsilon": 1e-05,
 "model_type": "gpt2",
 "n_ctx": 1024,
 "n_embd": 768,
 "n_head": 12,
 "n_inner": null,
 "n_layer": 12,
 "n_positions": 1024,
 "pad_token_id": null,
 "reorder_and_upcast_attn": false,
 "resid_pdrop": 0.1,
 "scale_attn_by_inverse_layer_idx": false,
 "scale_attn_weights": true,
 "summary_activation": null,
 "summary_first_dropout": 0.1,
 "summary_proj_to_labels": true,
 "summary_type": "cls_index",
 "summary_use_proj": true,
 "task_specific_params": {
  "text-generation": {
   "do_sample": true,
   "max_length": 50
  }
 },
 "tie_word_embeddings": true,
 "transformers_version": "5.2.0",
 "use_cache": true,
 "vocab_size": 50257
}

Model Parameter Names + Shapes - GPT2

```
for name, param in model.named_parameters():
    print(name, param.shape)
```

wte.weight torch.Size([50257, 768])
wpe.weight torch.Size([1024, 768])
h.0.ln_1.weight torch.Size([768])
h.0.ln_1.bias torch.Size([768])
h.0.attn.c_attn.weight torch.Size([768, 2304])
h.0.attn.c_attn.bias torch.Size([2304])
h.0.attn.c_proj.weight torch.Size([768, 768])
h.0.attn.c_proj.bias torch.Size([768])
h.0.ln_2.weight torch.Size([768])
h.0.ln_2.bias torch.Size([768])
h.0.mlp.c_fc.weight torch.Size([768, 3072])
h.0.mlp.c_fc.bias torch.Size([3072])
h.0.mlp.c_proj.weight torch.Size([3072, 768])
h.0.mlp.c_proj.bias torch.Size([768])
h.1.ln_1.weight torch.Size([768])
h.1.ln_1.bias torch.Size([768])
h.1.attn.c_attn.weight torch.Size([768, 2304])
h.1.attn.c_attn.bias torch.Size([2304])
h.1.attn.c_proj.weight torch.Size([768, 768])
h.1.attn.c_proj.bias torch.Size([768])

Viewing the Weights GPT2

```
state_dict = model.state_dict()
state_dict["h.0.ln_1.weight"]
```

```
tensor([0.2232, 0.1820, 0.1534, 0.1917, 0.2036, 0.1948, 0.1467, 0.1865, 0.2143,
        0.1956, 0.2118, 0.2153, 0.1882, 0.2074, 0.1871, 0.2040, 0.2044, 0.1900,
        0.1952, 0.0475, 0.1909, 0.2115, 0.1971, 0.2202, 0.1998, 0.2108, 0.2303,
        0.1879, 0.1939, 0.2018, 0.1891, 0.1861, 0.1958, 0.1832, 0.1978, 0.2243,
        0.0706, 0.1958, 0.1943, 0.1939, 0.1978, 0.1951, 0.1995, 0.1912, 0.2083,
        0.2037, 0.1849, 0.1945, 0.2189, 0.0419, 0.1977, 0.1979, 0.0608, 0.1824,
        0.2055, 0.0476, 0.1892, 0.2079, 0.2047, 0.2233, 0.2097, 0.2075, 0.2076,
        0.1793, 0.1312, 0.1841, 0.1939, 0.1561, 0.0577, 0.1948, 0.2048, 0.1717,
        0.1942, 0.1708, 0.1989, 0.1993, 0.2082, 0.1071, 0.1968, 0.1770, 0.2164,
        0.1864, 0.1938, 0.2184, 0.1343, 0.1707, 0.0683, 0.1401, 0.1823, 0.2045,
        0.2007, 0.1853, 0.1783, 0.1889, 0.1870, 0.1975, 0.2114, 0.2108, 0.2083,
        0.2409, 0.1938, 0.2022, 0.0857, 0.1823, 0.1879, 0.1979, 0.1850, 0.1029,
        0.1762, 0.1953, 0.2231, 0.2006, 0.2022, 0.2134, 0.1970, 0.1820, 0.0568,
        0.2269, 0.1882, 0.1770, 0.1880, 0.1910, 0.1872, 0.1613, 0.1946, 0.1930,
        0.1981, 0.2030, 0.1848, 0.2341, 0.1832, 0.1893, 0.2368, 0.2085, 0.1833,
        0.2083, 0.2009, 0.2212, 0.1342, 0.0614, 0.1913, 0.1812, 0.1041, 0.1957,
        0.1902, 0.1355, 0.2145, 0.1974, 0.1904, 0.1997, 0.1849, 0.1776, 0.2038,
        0.1773, 0.1878, 0.1793, 0.1960, 0.1935, 0.1786, 0.1532, 0.1185, 0.2015,
        0.1907, 0.2112, 0.1967, 0.2037, 0.1994, 0.0528, 0.1832, 0.1633, 0.1812,
        0.1988, 0.1742, 0.2177, 0.1901, 0.1778, 0.0706, 0.1987, 0.2417, 0.1658,
        0.1840, 0.1763, 0.1950, 0.2085, 0.1906, 0.2025, 0.1713, 0.2475, 0.1939,
```

Changing a Weight

```
state_dict["h.0.ln_1.weight"][0] = 0.1
state_dict[“h.0.ln_1.weight"]
```

```
tensor([0.1000, 0.1820, 0.1534, 0.1917, 0.2036, 0.1948, 0.1467, 0.1865, 0.2143,
        0.1956, 0.2118, 0.2153, 0.1882, 0.2074, 0.1871, 0.2040, 0.2044, 0.1900,
        0.1952, 0.0475, 0.1909, 0.2115, 0.1971, 0.2202, 0.1998, 0.2108, 0.2303,
        0.1879, 0.1939, 0.2018, 0.1891, 0.1861, 0.1958, 0.1832, 0.1978, 0.2243,
```

model.config

GPT2Config {
 "activation_function": "gelu_new",
 "add_cross_attention": false,
 "architectures": [
  "GPT2LMHeadModel"
 ],
 "attn_pdrop": 0.1,
 "bos_token_id": 50256,
 "dtype": "**float32**",
 "embd_pdrop": 0.1,
 "eos_token_id": 50256,
 "initializer_range": 0.02,
 "layer_norm_epsilon": 1e-05,
 "model_type": "gpt2",
 "n_ctx": 1024,
 "n_embd": 768,
 "n_head": **12**,
 "n_inner": null,
 "n_layer": **12**,
 "n_positions": **1024**,
 "pad_token_id": null,
 "reorder_and_upcast_attn": false,
 "resid_pdrop": 0.1,
 "scale_attn_by_inverse_layer_idx": false,
 "scale_attn_weights": true,
 "summary_activation": null,
 "summary_first_dropout": 0.1,
 "summary_proj_to_labels": true,
 "summary_type": "cls_index",
 "summary_use_proj": true,
 "task_specific_params": {
  "text-generation": {
   "do_sample": true,
   "max_length": 50
  }
 },
 "tie_word_embeddings": true,
 "transformers_version": "5.2.0",
 "use_cache": true,
 "vocab_size": 50257
}

Llama2


```
LlamaConfig {
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 1,
  "dtype": "float16",
  "eos_token_id": 2,
  "head_dim": 128,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 11008,
  "max_position_embeddings": 4096,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 32,
  "pad_token_id": null,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_parameters": {
    "rope_theta": 10000.0,
    "rope_type": "default"
  },
  "tie_word_embeddings": false,
  "transformers_version": "5.2.0",
  "use_cache": true,
  "vocab_size": 32000
}
```

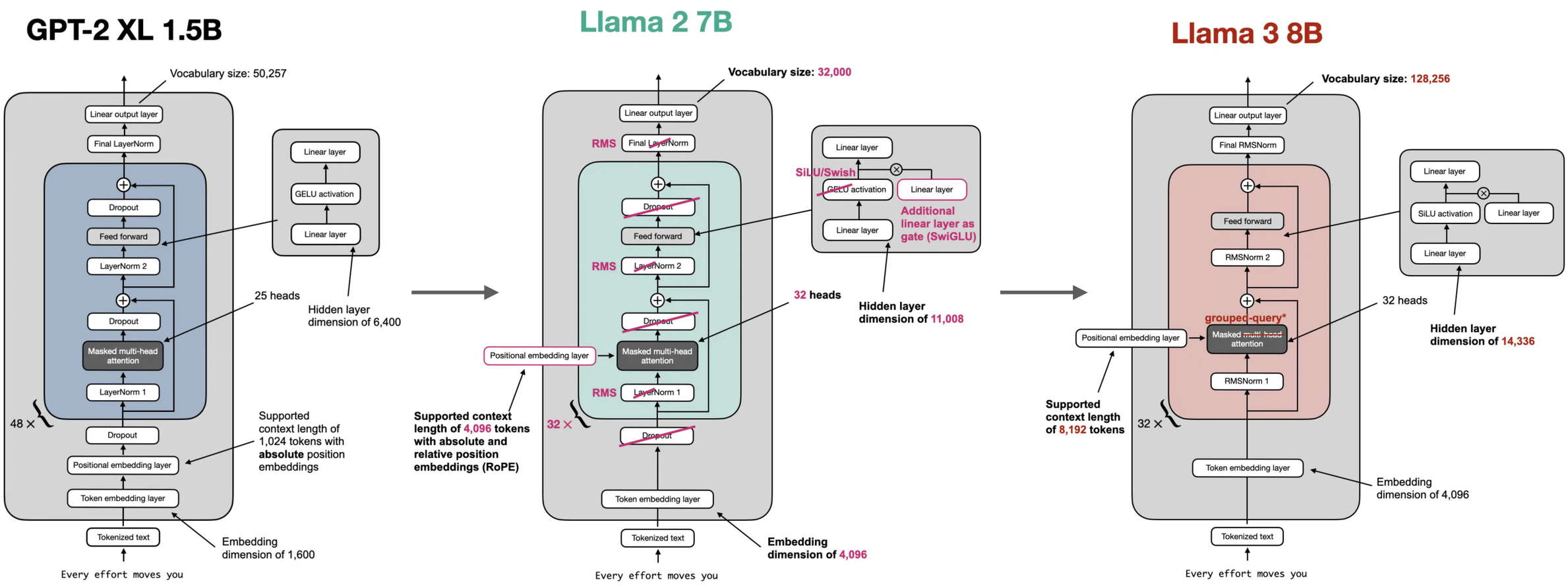
Llama 3

LlamaConfig {
  "architectures": [
    "LlamaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 128000,
  "dtype": "**bfloat16**",
  "eos_token_id": 128001,
  "head_dim": 128,
  "hidden_act": "silu",
  "hidden_size": 4096,
  "initializer_range": 0.02,
  "intermediate_size": 14336,
  "max_position_embeddings": **131072**,
  "mlp_bias": false,
  "model_type": "llama",
  "num_attention_heads": 32,
  "num_hidden_layers": 32,
  "num_key_value_heads": 8,
  "pad_token_id": null,
  "pretraining_tp": 1,
  "rms_norm_eps": 1e-05,
  "rope_parameters": {
    "factor": 8.0,
    "high_freq_factor": 4.0,
    "low_freq_factor": 1.0,
    "original_max_position_embeddings": 8192,
    "rope_theta": **500000.0**,
    "rope_type": "llama3"
  },
  "tie_word_embeddings": false,
  "transformers_version": "5.2.0",
  "use_cache": true,
  "vocab_size": **128256**
}

Gemma2


```
GemmaConfig {
  "architectures": [
    "GemmaForCausalLM"
  ],
  "attention_bias": false,
  "attention_dropout": 0.0,
  "bos_token_id": 2,
  "dtype": "bfloat16",
  "eos_token_id": 1,
  "head_dim": 256,
  "hidden_act": "gelu",
  "hidden_size": 2048,
  "initializer_range": 0.02,
  "intermediate_size": 16384,
  "max_position_embeddings": 8192,
  "model_type": "gemma",
  "num_attention_heads": 8,
  "num_hidden_layers": 18,
  "num_key_value_heads": 1,
  "pad_token_id": 0,
  "rms_norm_eps": 1e-06,
  "rope_parameters": {
    "rope_theta": 10000.0,
    "rope_type": "default"
  },
  "tie_word_embeddings": true,
  "transformers_version": "5.2.0",
  "use_bidirectional_attention": null,
  "use_cache": true,
  "vocab_size": 256000
}
```

GPT -> Llama2


GPT-2 XL 1.5B
Vocabulary size: 50,257
Linear output layer
Final LayerNorm
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
48 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you
Linear layer
GELU activation
Linear layer
25 heads
Hidden layer dimension of 6,400
Supported context length of 1,024 tokens with absolute position embeddings
Embedding dimension of 1,600
Llama 2 7B
Vocabulary size: 32,000
Linear output layer
RMS
Final LayerNorm
Dropout
Feed forward
RMS
LayerNorm 2
Dropout
Masked multi-head attention
RMS
LayerNorm 1
32 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you
Linear layer
SiLU/Swish
GELU activation
Linear layer
Linear layer
Additional linear layer as gate (SwiGLU)
32 heads
Hidden layer dimension of 11,008
Supported context length of 4,096 tokens with absolute and relative position embeddings (RoPE)
Embedding dimension of 4,096
Llama 3 8B
Vocabulary size: 128,256
Linear output layer
Final RMSNorm
Feed forward
RMSNorm 2
grouped-query*
Masked multi-head attention
RMSNorm 1
32 ×
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you
Linear layer
SiLU activation
Linear layer
Linear layer
32 heads
Hidden layer dimension of 14,336
Supported context length of 8,192 tokens
Embedding dimension of 4,096
* The larger Llama 2 34B and 70B also used grouped-query attention

GPT -> Llama2

Replace LayerNorm with RMSNorm layer

$$y_i = \frac{x_i}{\text{RMS}(x)}\gamma_i, \quad \text{where} \quad \text{RMS}(x) = \sqrt{\epsilon + \frac{1}{n}\sum x_i^2}$$

where x is the input γ is a trainable parameter

Replace GELU with SiLU activation

$\text{silu}(x) = x \cdot \sigma(x)$, where $\sigma(x)$ is the logistic sigmoid.

Update the FeedForward module

Llama uses a "Gates Linear Unit" (GLU) variant of SiLU called SwiGLU

Implement RoPE

Add RoPE to MultiHeadAttention module

GPT applies the positional embeddings to the inputs

Llama applies rotations to the query and key vectors

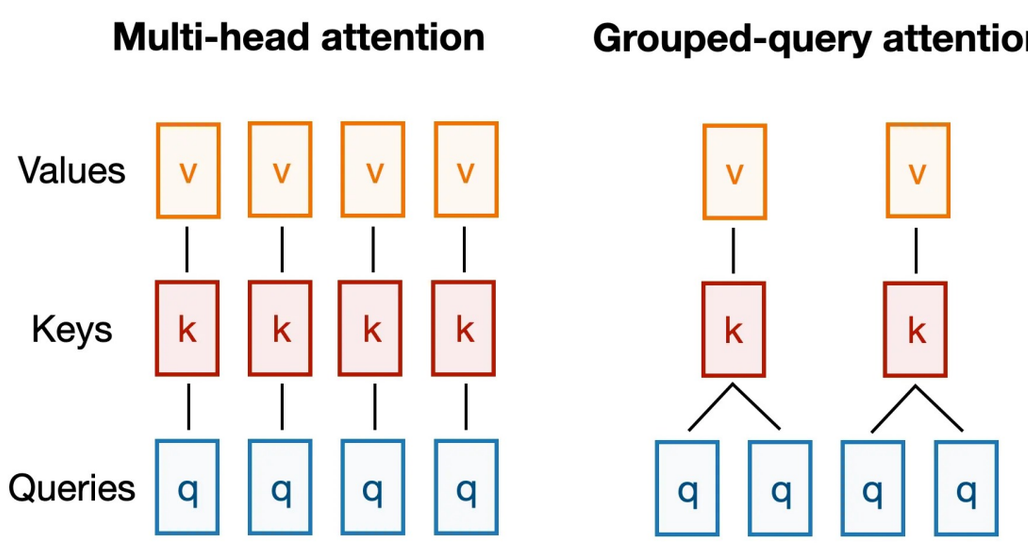
Lama2 -> Llama3

Modified RoPE

Llama 3 supports 8,192 tokens, twice as many as Llama 2 (4,096)

Base value for RoPE θ was increased from 10,000 to 500,000

Grouped-query attention



Pretraining Issues

Pretraining is 98% of compute budget

The Scaling Laws: Compute, Data, and Parameters

- Chinchilla Optimality - 20 tokens per parameter
- Inference-Optimal (Over-training)

Data Curation: Quality vs. Quantity

- Deduplication & Filtering
  - Remove low-quality and near-duplicate text
- Data Mixture
  - Code and math data boost a model's reasoning capabilities
- Synthetic Data

Pretraining Issues

Architecture Choices

- Dense Models
- Mixture of Experts (MoE)

Tokenization & Vocabulary

- Vocabulary Size
- Byte-Pair Encoding (BPE)

Training Stability & Hyperparameters

- Precision (FP16/BF16/FP8) BF16
- Learning Rate Schedulers
  - "Warm up" and "decay" the learning rate

Knowledge Distillation

Smaller student model learns from larger teacher model

Type	What's Distilled
Response-based	Final output logist
Feature-based	Intermediate hidden states
Relation-based	Relationships between layers/samples
Black-box	Only output text

Knowledge Distillation

Sequence-level KD

Generate full output sequences from the teacher and use them as training data for the student

Word-level KD

Align token-by-token distributions at every decoding step

Chain-of-Thought distillation

Distill the teacher's reasoning traces, not just answers

Layer-mapping

Map teacher layers to corresponding student layers for feature distillation

Student	Teacher	Method
DistilBERT (66M)	BERT-base (110M)	Response + feature
TinyLLaMA (1.1B)	LLaMA 2 (7B+)	Sequence-level
GPT-4	Phi-1/2/3	Black-box + curated data

Black-Box

Large, pre-trained teacher model generates output for lare number of prompts

Student model is trained using (prompt, Teacher response)

Using Teachers Probability Distribution

Both the teacher & student are given the same prompt

Use the differences in probability distribution for the next token to train the student

Common loss function

$$L = (1 - \alpha)L_{CE}(\text{Student, Ground Truth}) + \alpha L_{KD}(\text{Student, Teacher})$$

L_{KD} Kullback-Leibler (KL) Divergence between the teacher's and student's softened distributions

DeepSeek-R1 Distillation

The Teacher: DeepSeek-R1, Chain-of-thought

The Dataset:

800,000 high-quality reasoning samples (questions + step-by-step logic + final answers)

The Student Training:

Smaller open-source backbones (like Qwen-2.5 and Llama-3.1) were fine-tuned using Supervised Fine-Tuning (SFT) on these 800k "thinking" traces.

Benchmark	Teacher: R1 (671B)	Student: Llama-70B	Student: Qwen-32B	OpenAI o1-mini
AIME 2024	79.8%	70.0%	72.6%	63.6%
MATH-500	97.3%	94.5%	94.3%	90.0%
GPQA Diamond	71.5%	65.2%	62.1%	60.0%

DistilBERT

Student DistilBERT 66M parameters vs. Teacher BERT’s 110M.

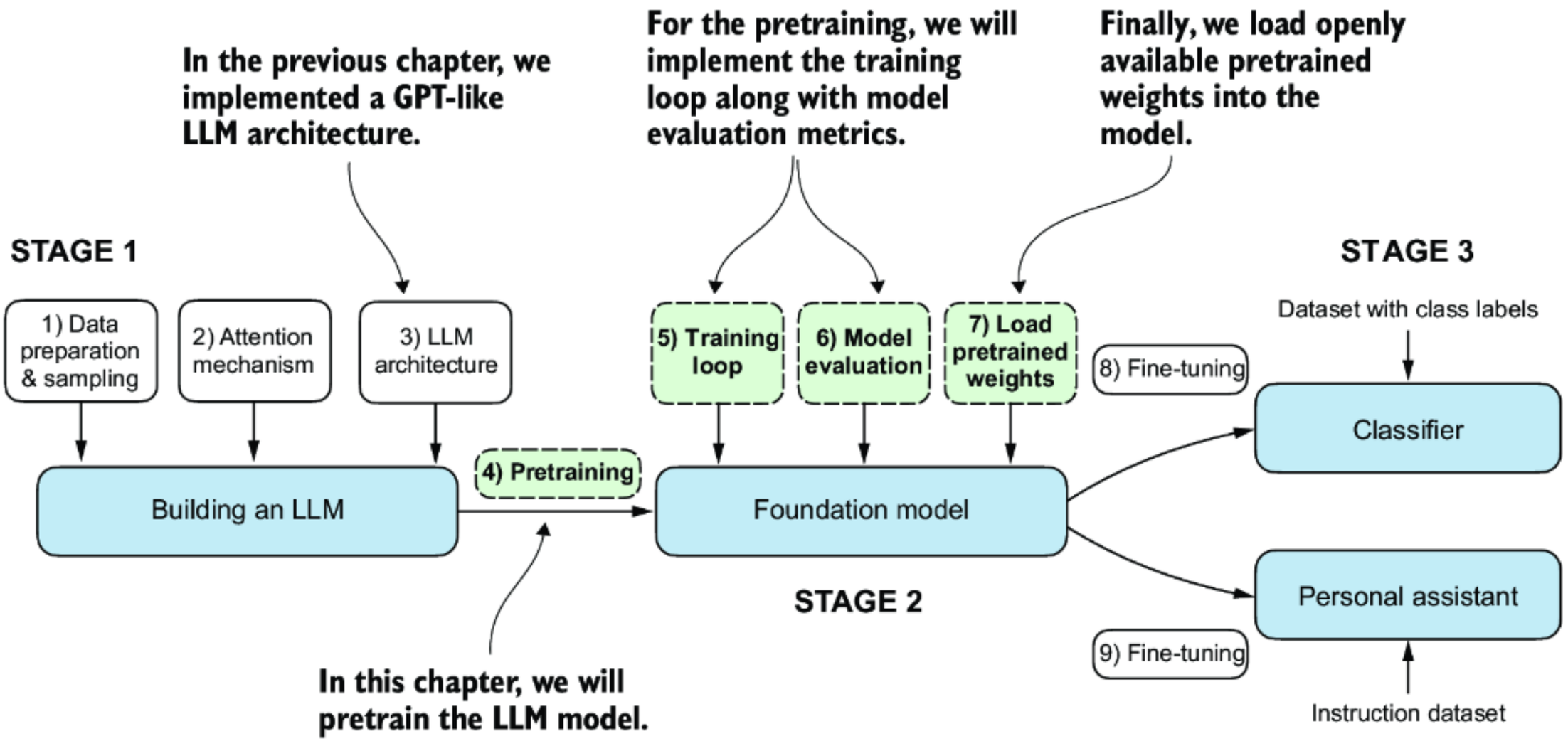
Result:

- 97% of BERT’s performance
- 40% smaller
- 60% faster

Technique:

- Student starts with 1 out of every 2 layers from the teacher

Chapter 5 - Pretraining



Model to Train

import torch GPTModel From

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,   # Vocabulary size
    "context_length": 256, # Shortened context length (orig: 1024)
    "emb_dim": 768,        # Embedding dimension
    "n_heads": 12,         # Number of attention heads
    "n_layers": 12,        # Number of layers
    "drop_rate": 0.1,      # Dropout rate
    "qkv_bias": False      # Query-key-value bias
}
```

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval();  # Disable dropout during inference
```

Training

Convert text <-> tokens <-> max logits

Need inputs and targets

Determine how “off” model(inputs) are from targets

Use loss function to adjust the weights

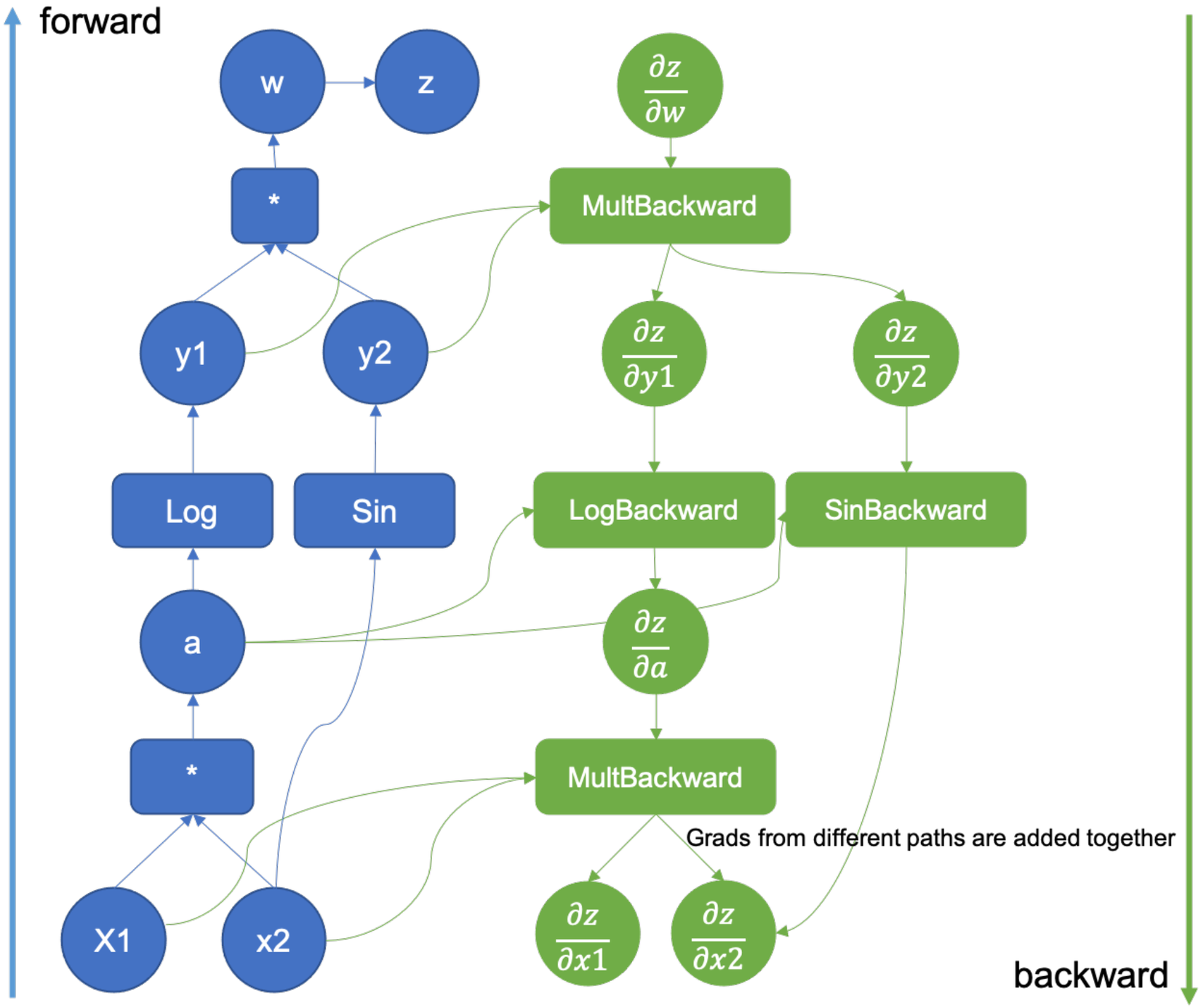
text <-> tokens

```
def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add batch dimension
    return encoded_tensor


def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) # remove batch dimension
    return tokenizer.decode(flat.tolist())
```

```
import torch
x = torch.tensor([1, 2, 3, 4])
print(x.shape)                  # torch.Size([4])
x = x.unsqueeze(0)
print(x.shape)                  # torch.Size([1, 4])
```

```
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (B, T) array of indices in the current context
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]

        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]

        idx_next = torch.argmax(logits, dim=-1, keepdim=True)  # (batch, 1)

        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

return idx
```

log(x1*x2) * sin(x2)



PyTorch Autograd Saved Tensors

For some computations, Torch will store

- Input tensors
- Intermediate tensors

Done to make backpropagation more efficient

```
import torch
x = torch.randn(5, requires_grad=True)
y = x.exp()
print(y.equal(y.grad_fn._saved_result))  # True
print(y is y.grad_fn._saved_result)
```

```
True
False
```

requires_grad flag

If true tensors will have gradients accumulated in their .grad field

Defaults to false unless wrapped in an nn.Parameter

requires_grad=False

- Means they will not be part of the backward graph
- So will not be updated backward calculation

```
with torch.no_grad():
    logits = model(inputs)
```

softmax

Converts collection of values to probabilities

Exponential of each value

Normalize result

Logits	Exponential	Normalized
2.5	12.18	0.7856
1.0	2.72	0.1753
-0.5	0.61	0.0391

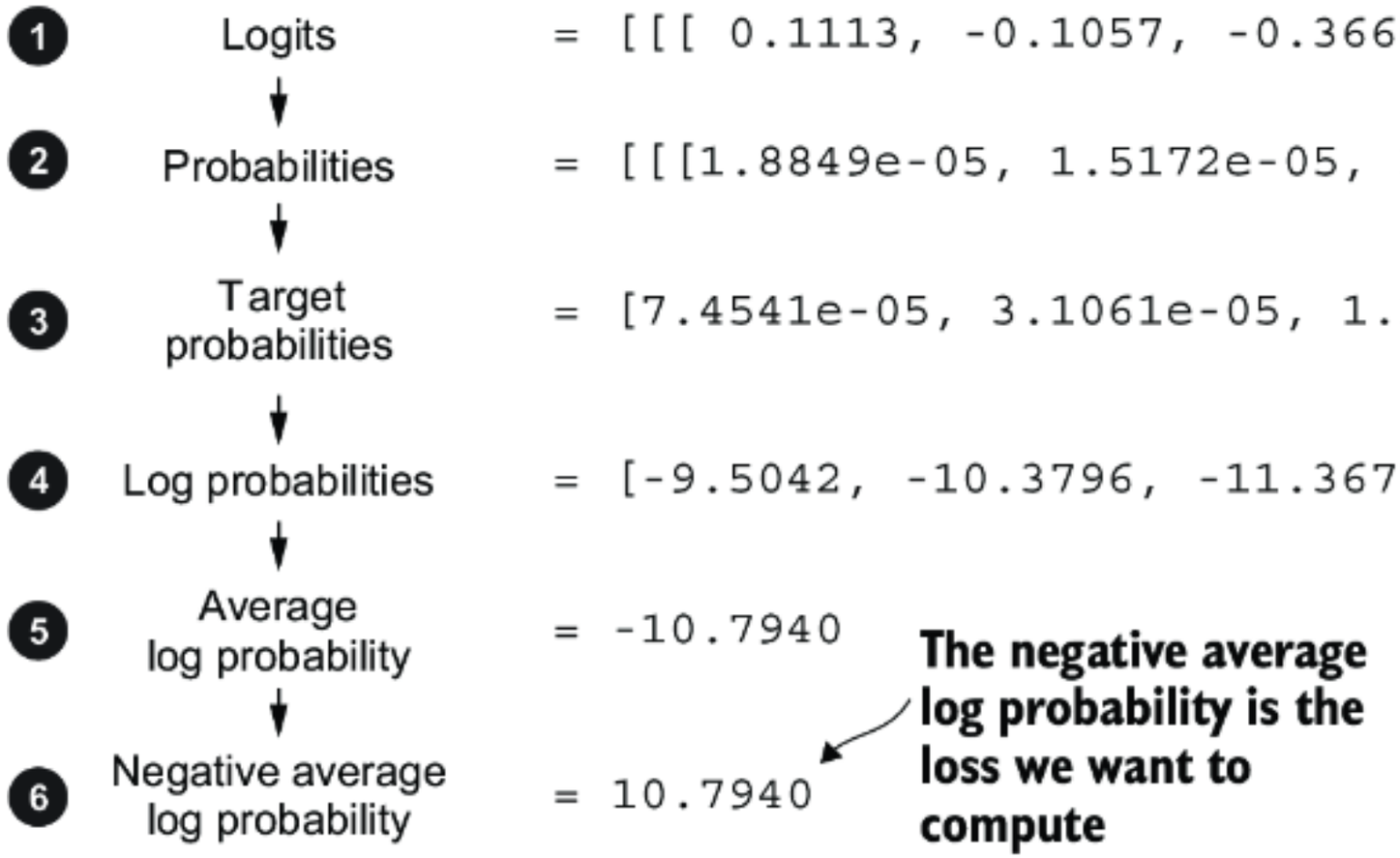
```
import torch

x = torch.tensor([2.5, 1.0, -0.5])
y = torch.softmax(x, dim=0)

print(y)

  tensor([0.7856, 0.1753, 0.0391])
```

Logits	Softmax
5.5	0.9866
1.0	0.0110
-0.5	0.0024

Logits	Softmax
10.5	0.9991
1.0	7.4845E-05
-0.5	1.6700E-05

Extreme values push softmax results to 1 & 0

Backpropagation - cross_entropy



loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)

Perplexity

How well probability distribution given by the model
matches the actual distribution of the words in the dataset

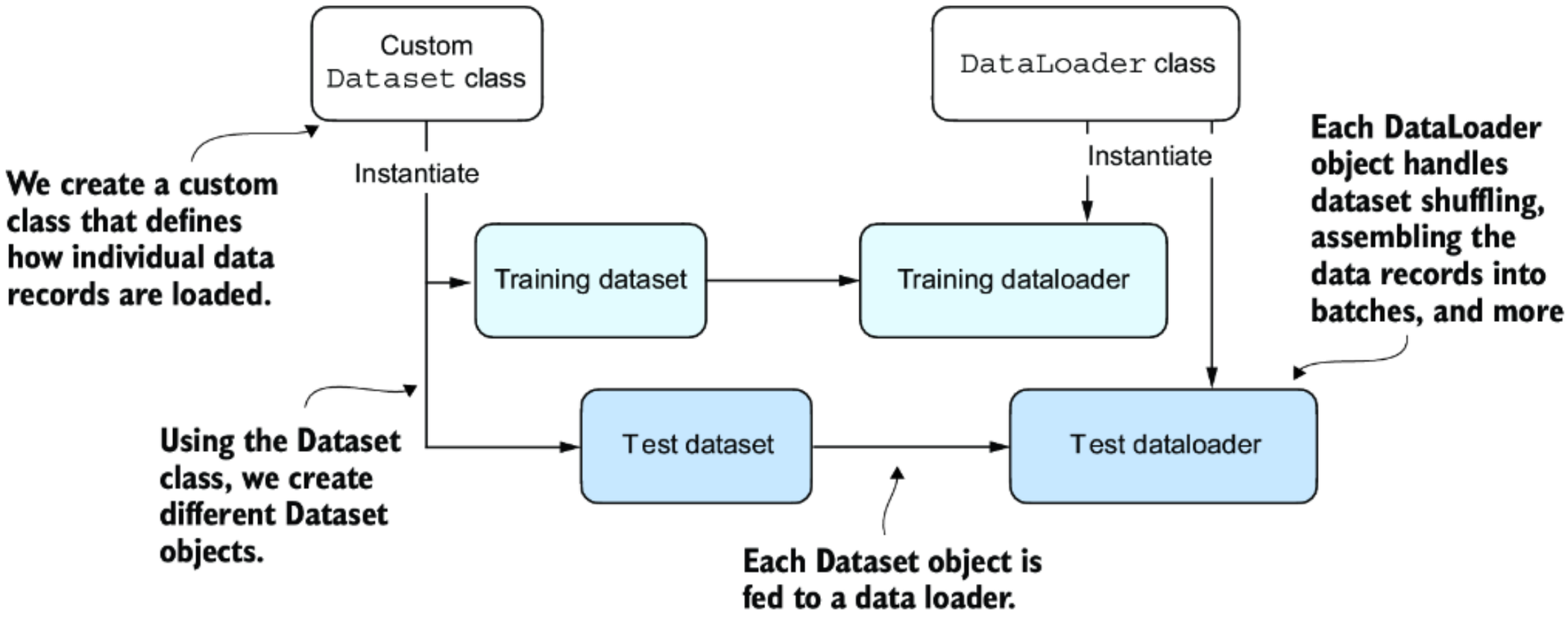
How uncertain a model is about the next word in a sequence.

Effective vocabulary size that the model is uncertain about at each step

torch.exp(loss)

tensor(48725.8203)

Reading Data



Dataset Types

Map-style datasets

Subclass of torch.utils.data.Datase

__getitem__()

__len__()

Map from (possibly non-integral) indices/keys to data samples

Iterable-style datasets

subclass of torch.utils.data.IterableDataset

__iter__()

Useful when data comes from a stream

Simple Dataset

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):
        one_x = self.features[index]
        one_y = self.labels[index]
        return one_x, one_y

    def __len__(self):
        return self.labels.shape[0]

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

From Chapter 2

```
import torch
from torch.utils.data import Dataset, DataLoader
class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt)

    for i in range(0, len(token_ids) - max_length, stride):
        input_chunk = token_ids[i:i + max_length]
        target_chunk = token_ids[i + 1: i + max_length + 1]
        self.input_ids.append(torch.tensor(input_chunk))
        self.target_ids.append(torch.tensor(target_chunk))

def __len__(self):
    return len(self.input_ids)

def __getitem__(self, idx):
    return self.input_ids[idx], self.target_ids[idx]
```

DataLoader

Batching, shuffling, and parallelizing data loading

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

DataLoader Arguments

dataset:

The Dataset object from which the data is loaded

batch_size:

The number of samples in each batch.

shuffle:

A boolean indicating whether to shuffle the data.

num_workers:

Number of workers to process data in parallel

collate_fn:

The default collate function works for most common use cases.

pin_memory:

Copy Tensors into CUDA pinned memory before returning them.

This can improve data transfer speeds to GPU devices.

drop_last:

Drop the last incomplete batch, if the dataset size is not evenly divisible by the batch size.

Creating the Loader

```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True, num_workers=0):
    tokenizer = tiktoken.get_encoding("gpt2")
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
    dataloader = DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle, drop_last=drop_last,
        num_workers=num_workers)

    return dataloader
```

Reading Data

```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()


train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)                                  #To make things consistent

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True,
    num_workers=0
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False,
    num_workers=0
)
```

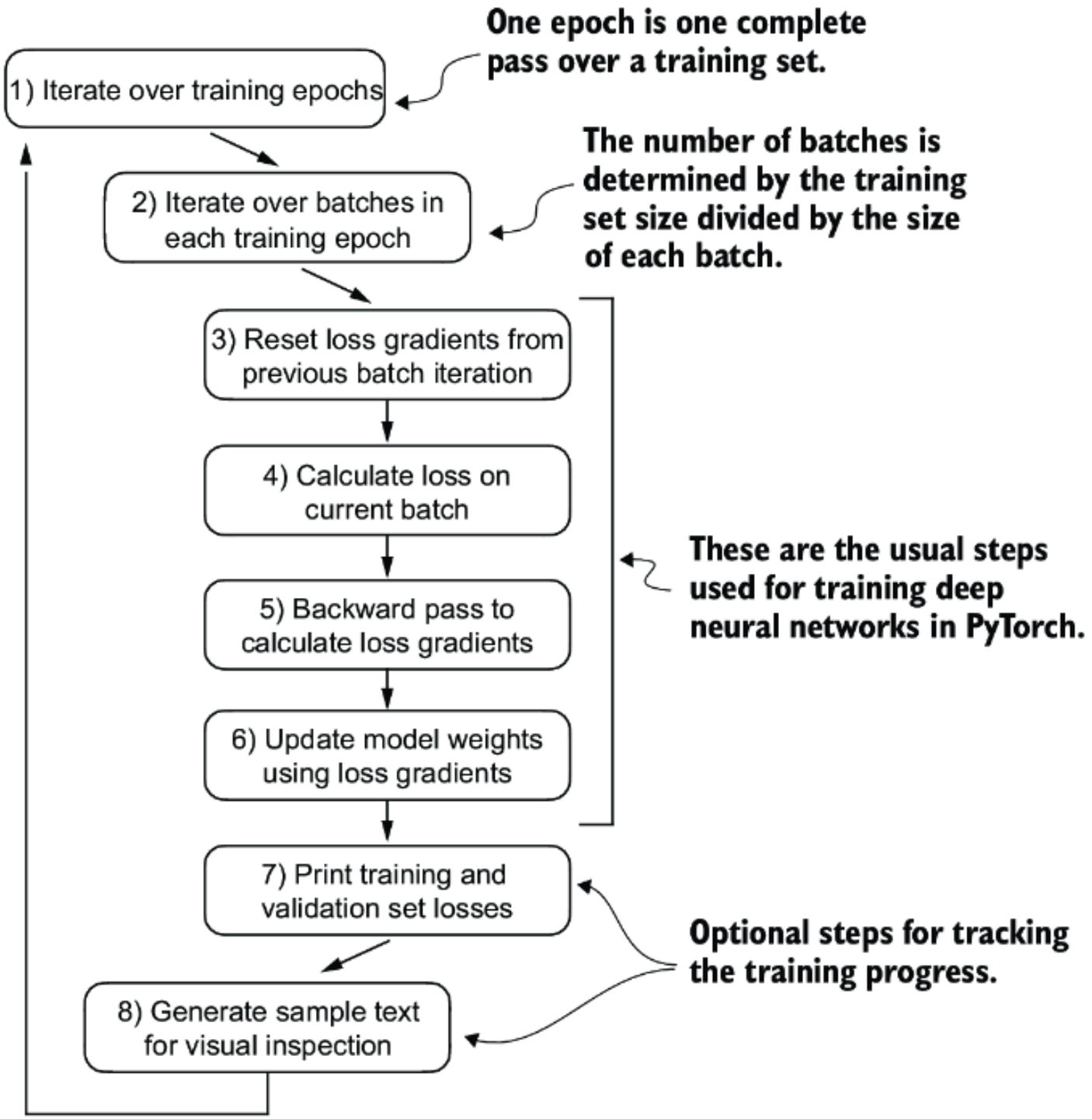
Loss for one Batch

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch = input_batch.to(device)        #1
    target_batch = target_batch.to(device)
    logits = model(input_batch)
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss
```

Loss for All Batched

```
def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
for i, (input_batch, target_batch) in enumerate(data_loader):
    if i < num_batches:
        loss = calc_loss_batch(
            input_batch, target_batch, model, device
        )
        total_loss += loss.item()
    else:
        break
return total_loss / num_batches
```

Training an LLM



```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        generate_and_print_sample(
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

evaluate_model

```
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(
            train_loader, model, device, num_batches=eval_iter
        )
        val_loss = calc_loss_loader(
            val_loader, model, device, num_batches=eval_iter
        )
    model.train()
    return train_loss, val_loss
```

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))      #1
    model.train()
```

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

torch.optim.Adam

Adaptive Moment Estimation

Momentum

Accelerate gradient descent
by adding a fraction of the previous update to the current update.

RMSProp (Root Mean Square Propagation)

Adapts the learning rate for each parameter based on the magnitude of recent gradients

Adaptive Learning Rates

Bias Correction

torch.optim.Adam

Adaptive Moment Estimation

Computational Efficiency

- Computationally efficient
- Low memory requirements,
- Suitable for training large neural networks

Robust

- Performs well across a wide range of deep learning tasks and model architectures

Fast Convergence

- Often converges faster than traditional optimization algorithms

torch.optim.AdamW

Effective in training large and complex models

Decouples of weight decay from the gradient-based updates

Weight decay is a separate step
Applying it directly to the weights after the gradient update

Newer Optimizers

Optimizer	Memory Cost	Convergence Speed	Main Advantage
AdamW	High (2x params)	Baseline	"It just works."
AdEMAMix	High (3x params)	**Very Fast**	Best "Sample Efficiency" (fewer tokens).
Lion	**Low** (1x params)	Fast	Massive memory savings for large models.
Muon	Medium	**Superior**	Optimized for the "geometry" of matrices.

gpt_train.py

Trains the model on the-verdict.txt

Ep 1 (Step 000000): Train loss 9.812, Val loss 9.846

Ep 1 (Step 000005): Train loss 7.588, Val loss 8.044

Every effort moves you,,,,,,,,,,,,.

Ep 2 (Step 000010): Train loss 6.582, Val loss 6.800

Ep 2 (Step 000015): Train loss 5.920, Val loss 6.592

Every effort moves you, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and, and

…

Ep 9 (Step 000075): Train loss 0.762, Val loss 6.329

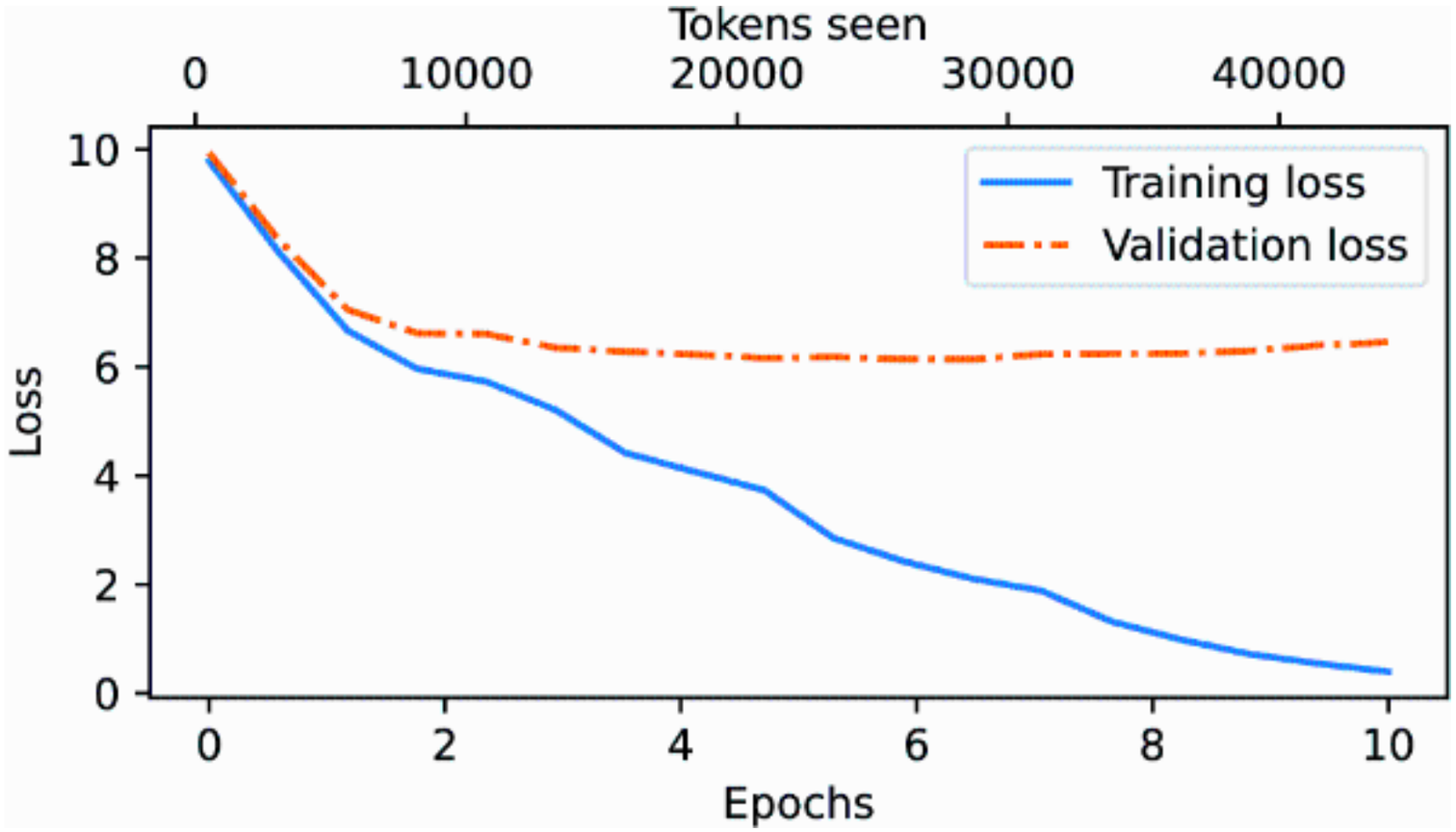
Ep 9 (Step 000080): Train loss 0.563, Val loss 6.462

Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me!
He laughed again, and threw back the window-curtains, I saw that, and down the room, my eyes

Ep 10 (Step 000085): Train loss 0.358, Val loss 6.542

Every effort moves you?" "Yes--quite insensible to the irony. She wanted him vindicated--and by me!
He laughed again, and threw back his head to look up at the sketch of the donkey. "There were days
when I

Overfitting Past Epoch 2



Always selecting Highest

```
def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

torch.argmax

Returns the indices of the maximum value of all elements in the input tensor

```
a = torch.randn(4, 4)
a
```

```
tensor([[ 8.9682e-01, -2.1756e+00, -7.1390e-01, -4.4147e-01],
        [ 6.5534e-01,  4.5381e-01,  1.5842e+00, -3.0665e+00],
        [-4.3658e-01,  3.0377e-04,  1.9257e+00,  2.9086e-01],
        [ 3.1287e-01,  4.9243e-02, -5.2300e-01, -1.2458e+00]])
```

```
torch.argmax(a)
```

```
tensor(10)
```

Better Selection of Next Token

Probabilistic Sampling

Temperature Scaling

Top-k Sampling

Probabilistic Sampling

torch.multinomial(input, num_samples, replacement=False, *, generator=None, out=None)

Returns tensor with index selected with probability of the value of that position

```
import numpy as np
import torch

weights = torch.tensor([2.0, 10.0, 8.0, 5.0])

sample = [torch.multinomial(weights, 1).item() for i in range(100)]

sampled_ids = np.bincount(sample)
for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {i}")
```

```
8 x 0
44 x 1
35 x 2
13 x 3
```

Temperature Scaling

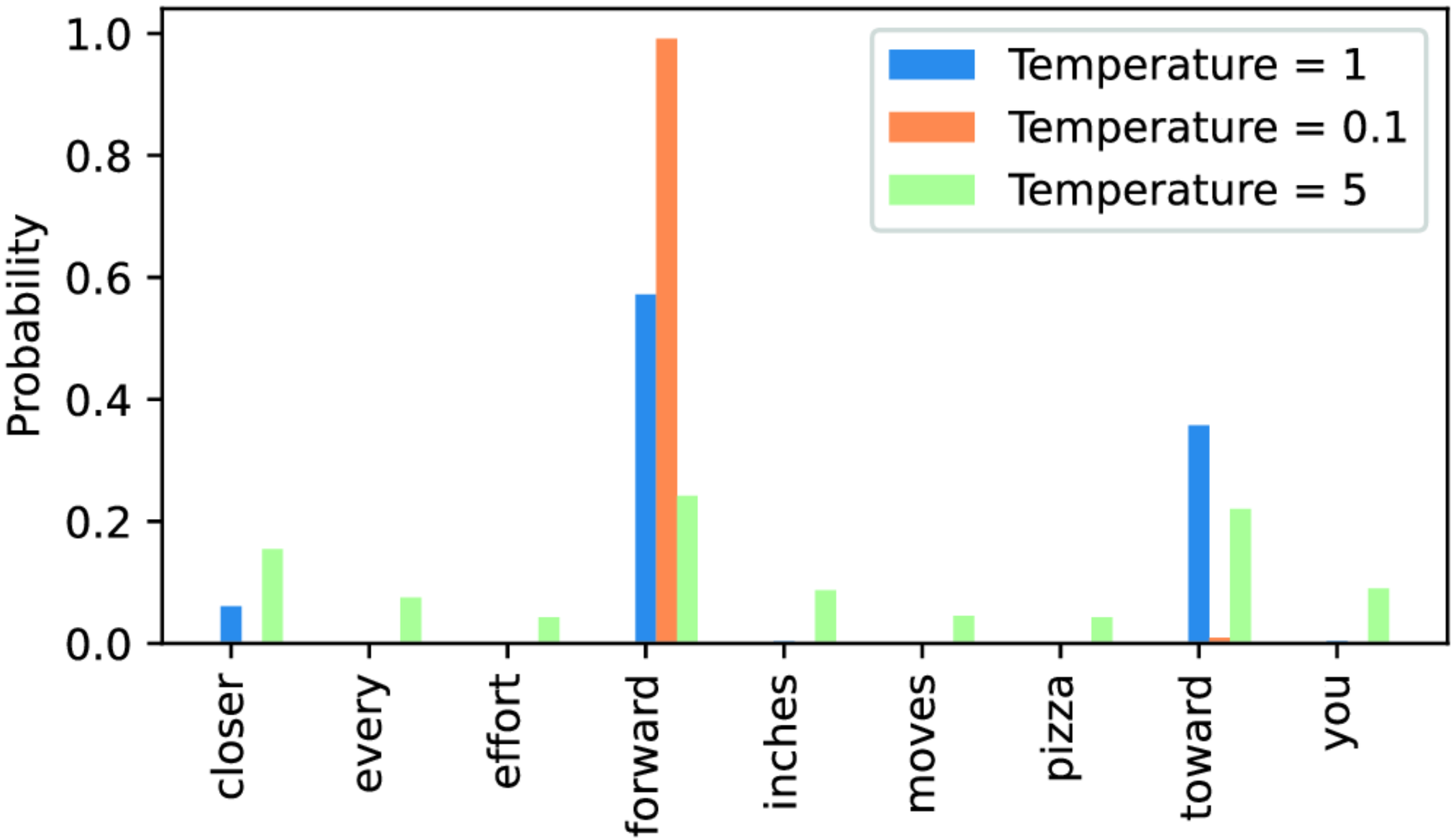
Divide logits by a value

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

Large values push softmax values to extremes

Temperature

- Less than 1, makes the model more certain
- More than 1 reduces the certainty


Temperature = 1
Temperature = 0.1
Temperature = 5
Probability
1.0
0.8
0.6
0.4
0.2
0.0
closer
every
effort
forward
inches
moves
pizza
toward
you

Top-k Sampling

Only select from the top k items

Benefits

Enhanced Coherence

Efficiency

torch.argmax vs Dynamic Token Selection

Every effort moves you know," was one of the axioms he laid down across the Sevres and silver of an exquisitely appointed lun

Every effort moves you stand to work on surprise, a one of us had gone with random-

Saving & Reloading The Model

```
torch.save(model.state_dict(), "model.pth")
```

```
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(torch.load("model.pth", map_location=device))
model.eval()
```

Saving & Reloading The Model

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

```
checkpoint = torch.load("model_and_optimizer.pth", map_location=device)
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

Some Performance Improvements

	Avg tok/sec	Reserved memory GB
Base	12,526	26.2422
Use tensor cores	27,648	26.2422
Fused AdamW optimizer	28,399	26.2422
Pinned memory in the data loader	28,402	26.2422
Using bfloat16 precision	45,486	13.7871
Replacing from-scratch code by PyTorch classes	55,256	11.5645
Using FlashAttention	91,901	5.9004
Using pytorch.compile	112,046	6.1875