

LLMS WITH PYTORCH

01 GENERAL

A GPT-style LLM is a **decoder-only Transformer** trained on **next-token prediction**: given tokens $t_1, \dots, t_n$, predict t_{n+1} . The loss is **cross-entropy** between the logits and the inputs **shifted by one**. Finetuning reuses the same loss on curated data.

B	batch size	V	vocabulary size
T	sequence / context length	H	attention heads
d	model (embedding) dim	L	Transformer blocks

Params $\approx 12 \cdot L \cdot d^2 + V \cdot d$. Shapes flow as $[B, T] \rightarrow [B, T, d] \rightarrow [B, T, V]$.

02 BUILDING BLOCKS

	nn.Embedding(V, d) : token id $\rightarrow$ vector (+ position)		MLP / SwiGLU : $d \rightarrow 4d \rightarrow d$, applied to each token
	F.scaled_dot_product_attention(q, k, v, is_causal=True) : fused causal attention		RoPE : rotary position encoding on q & k (Llama, Qwen)
	nn.RMSNorm(d) : cheaper LayerNorm, used pre-attention & pre-MLP		Residual $x + f(x)$ around every sub-layer; weight tying head $\leftrightarrow$ embedding

03 TOKENIZE

Text $\rightarrow$ sub-word ids with **BPE**. Always use the tokenizer and **chat template** that match the model.

```
1 from transformers import AutoTokenizer
2 tok = AutoTokenizer.from_pretrained("Qwen/Qwen3-0.6B")
3 ids = tok("Hello world", return_tensors="pt").input_ids # [1,T]
4 msgs = [{"role": "user", "content": "Explain LoRA"}]
5 prompt = tok.apply_chat_template(msgs, tokenize=False,
6                                   add_generation_prompt=True)
7 tok.pad_token = tok.pad_token or tok.eos_token # for batching
```

SMALL OPEN MODELS (HUGGING FACE IDS)

HuggingFaceTB/SmolLM2-135M	135M / 360M / 1.7B, great for laptops
Qwen/Qwen3-0.6B	0.6B / 1.7B / 4B, strong multilingual
google/gemma-3-270m	270M / 1B, built for task finetunes
meta-llama/llama-3.2-1B	1B / 3B, gated (accept license)

04 GENERATE

```
1 inputs = tok(prompt, return_tensors="pt").to(dev)
2 out = model.generate(**inputs, max_new_tokens=200,
3                      do_sample=True, temperature=0.7, top_p=0.9)
4 print(tok.decode(out[0], skip_special_tokens=True))
```

greedy / temperature	argmax, or logits / τ (<1 sharper, >1 more random)
top_k / top_p	sample only from the k best / p-mass tokens
KV cache	reuse past keys/values so each step is O(T)

05 BUILD A TINY GPT

```
1 import torch, torch.nn as nn, torch.nn.functional as F
2
3 class Block(nn.Module):
4     def __init__(self, d, h):
5         super().__init__()
6         self.h, self.n1, self.n2 = h, nn.RMSNorm(d), nn.RMSNorm(d)
7         self.qkv = nn.Linear(d, 3 * d, bias=False)
8         self.proj = nn.Linear(d, d, bias=False)
9         self.mlp = nn.Sequential(nn.Linear(d, 4 * d), nn.GELU(),
10                                  nn.Linear(4 * d, d))
11
12     def forward(self, x):
13         B, T, C = x.shape
14         q, k, v = self.qkv(self.n1(x)).split(C, dim=-1)
15         q, k, v = (t.view(B, T, self.h, -1).transpose(1, 2)
16                   for t in (q, k, v)) # [B, h, T, d/h]
17         a = F.scaled_dot_product_attention(q, k, v, is_causal=True)
18         x = x + self.proj(a.transpose(1, 2).reshape(B, T, C))
19         return x + self.mlp(self.n2(x))
20
21 class TinyGPT(nn.Module):
22     def __init__(self, V, d=384, h=6, L=6, T=256):
23         super().__init__()
24         self.tok, self.pos = nn.Embedding(V, d), nn.Embedding(T, d)
25         self.blocks = nn.Sequential(*(Block(d, h) for _ in range(L)))
26         self.norm = nn.RMSNorm(d)
27         self.head = nn.Linear(d, V, bias=False)
28         self.head.weight = self.tok.weight # weight tying
29
30     def forward(self, idx, targets=None):
31         pos = torch.arange(idx.size(1), device=idx.device)
32         x = self.blocks(self.tok(idx) + self.pos(pos))
33         logits = self.head(self.norm(x)) # [B, T, V]
34         if targets is None: return logits, None
35         loss = F.cross_entropy(logits.flatten(0, 1),
36                               targets.flatten(), ignore_index=-100)
37         return logits, loss
```

06 PRETRAINING LOOP

```
1 data = torch.tensor(tok.encode(text)) # 1-D token stream
2 def get_batch(B=32, T=256):
3     i = torch.randint(len(data) - T - 1, (B,))
4     x = torch.stack([data[j:j + T] for j in i])
5     y = torch.stack([data[j + 1:j + T + 1] for j in i])
6     return x.to(dev), y.to(dev)
7 model = torch.compile(TinyGPT(V=len(tok)).to(dev))
8 opt = torch.optim.AdamW(model.parameters(), lr=3e-4,
9                          betas=(0.9, 0.95), weight_decay=0.1)
10 for step in range(5000):
11     x, y = get_batch()
12     with torch.autocast("cuda", dtype=torch.bfloat16):
13         _, loss = model(x, y)
14     loss.backward()
15     nn.utils.clip_grad_norm_(model.parameters(), 1.0)
16     opt.step(); sched.step(); opt.zero_grad(set_to_none=True)
```

y is x shifted by one token. Use **warmup + cosine decay** for sched. Track **perplexity** = $\exp(\text{val loss})$; random-init loss $\approx \ln V$.

EFFICIENCY & MEMORY TRICKS

dtype=torch.bfloat16	Half the memory, no loss scaling (unlike fp16)
model.gradient_checkpointing_enable()	Less activation memory, ~30% slower
attn_implementation="sdpa"	Fused attention (or "flash_attention_2")
gradient_accumulation_steps=8	Large effective batch on a small GPU

07 FINETUNING

Full finetune	Update every weight: best quality, most memory
LoRA	Freeze W, learn low-rank $\Delta W = B \cdot A$ (~0.1-1% params)
QLoRA	LoRA on a frozen 4-bit NF4 base model
DoRA	Magnitude + direction LoRA (use_dora=True)
SFT $\rightarrow$ DPO	Instruction-tune, then align on preference pairs

GPU MEMORY PER PARAMETER (TRAINING)

Full FT, AdamW	~16 B: weights, grads, fp32 copy, Adam m & v
LoRA, bf16 base	~2 B + adapters + activations
QLoRA, 4-bit base	~0.5-0.7 B: a 7B model fits in ~6 GB

08 LORA

$h = W \cdot x + (\alpha/r) \cdot B \cdot A \cdot x$, $A: r \times k$, $B: d \times r$, $r \ll d$. **B starts at 0**, so step 0 equals the base model.

```
1 class LoRALinear(nn.Module):
2     def __init__(self, base, r=8, alpha=16):
3         super().__init__()
4         self.base = base.requires_grad_(False) # freeze W
5         self.A = nn.Linear(base.in_features, r, bias=False)
6         self.B = nn.Linear(r, base.out_features, bias=False)
7         nn.init.zeros_(self.B.weight) # dW = 0 at start
8         self.s = alpha / r
9
10    def forward(self, x):
11        return self.base(x) + self.s * self.B(self.A(x))
12
13 from peft import LoraConfig, get_peft_model # with PEFT
14 cfg = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05,
15                 target_modules="all-linear", task_type="CAUSAL_LM")
16 model = get_peft_model(model, cfg)
17 model.print_trainable_parameters()
```

Typical: $r \in [8, 64]$, $\alpha = r$ or $2r$, $\ln 1e-4 \sim 2e-4$ ($\approx 10 \times$ full-FT lr).

09 QLoRA

```
1 from transformers import AutoModelForCausalLM
2 from transformers import BitsAndBytesConfig as BnB
3 from peft import prepare_model_for_kbit_training
4 bnb = BnB(load_in_4bit=True,
5           bnb_4bit_quant_type="nf4", # NormalFloat4
6           bnb_4bit_use_double_quant=True, # quantize scales too
7           bnb_4bit_compute_dtype=torch.bfloat16)
8 model = AutoModelForCausalLM.from_pretrained("Qwen/Qwen3-0.6B",
9        quantization_config=bnb, device_map="auto")
10 model = prepare_model_for_kbit_training(model)
11 model = get_peft_model(model, cfg) # same LoRA config
```

SFT, SAVE & MERGE

labels = -100	Mask prompt tokens: loss on answers only
trl.SFTTrainer	Chat data + peft_config , packing, bf16
model.save_pretrained()	Saves the adapter only (a few MB)
model.merge_and_unload()	Fold BA into W (merge on a bf16 base)