

LLM の自作方法

Python で最小 LLM を作り、学習・推論・GPU 化・Semantic 拡張まで進める実践ガイド

1
最小 LLM

2
学習

3
推論

4
GPU 化

5
Semantic

対象読者 Transformer/LLM を「利用する」のではなく、構成要素を自分で実装し、学習・評価・改良の流れを理解したい人。

推奨方針 最初は小さく作り、正しく動くことを確認してから GPU 化・大規模化する。最初から巨大モデルを目指さない。

文書構成

- 1. 目標と全体像
- 2. 最小 LLM の構成
- 3. 実装順序
- 4. Tokenizer とデータ
- 5. Transformer の中核
- 6. 学習
- 7. 推論・会話
- 8. GPU 化
- 9. 評価・デバッグ
- 10. Semantic 拡張
- 11. 実用化への拡張
- 12. 推奨ロードマップ

作成日: 2026 年 9 月 30 日

1. 目標と全体像

LLM 自作の目的は、既存 API を呼ぶことではなく、テキストが「トークン → ベクトル → Transformer → 次トークン確率」へ変換され、誤差逆伝播によって重みが更新される一連の仕組みを自分で制御できる状態にすることです。

1.1 最小構成の処理フロー

段階	入力	処理	出力
1	テキスト	Tokenizer	token IDs
2	token IDs	Embedding + Position	ベクトル列
3	ベクトル列	Transformer blocks	文脈表現
4	文脈表現	LM Head	語彙ごとの logits
5	logits	Softmax / Sampling	次トークン
6	予測と正解	Cross Entropy + Backprop	勾配・更新重み

1.2 最初に狙うモデル規模

学習やデバッグを容易にするため、最初は次のような小型モデルが扱いやすいです。

項目	開始値の例	意味
語彙数	2,000～10,000	Tokenizer が扱う記号数
d_model	64～256	各トークンのベクトル次元
Transformer 層	2～6	文脈処理の深さ
Attention head	2～8	複数の関係を並列に見る
FFN hidden	d_model の 4 倍程度	各位置での非線形変換
Context length	64～512	一度に扱うトークン数

重要 小型モデルでも「LLM の構造」は同じです。パラメータ数を増やすことより、学習・推論・評価が一貫して動くことが最初の成功条件です。

2. 最小 LLM の構成

2.1 推奨ファイル構成

```

llm_project/
data/
  train.txt
tokenizer.py
tensor.py      # NumPy 自作路線なら独自 Tensor
autograd.py   # 自動微分を自作する場合
embedding.py
attention.py
ffn.py
layers.py
transformer.py
llm.py
loss.py
optimizer.py
train.py
demo.py
chat.py       # 会話 UI を追加する場合
model/        # checkpoint
results/      # 評価結果

```

NumPy だけで仕組みを理解する場合は `tensor.py` / `autograd.py` / `optimizer.py` を自作します。GPU 利用を優先する場合は PyTorch の Tensor と autograd を使い、Transformer 本体の設計に集中する方が効率的です。

2.2 モジュール依存関係

下位層	主な役割	上位層
Tokenizer	文字列 $\leftrightarrow$ ID	Embedding
Tensor / autograd	数値演算・勾配	全 NN 層
Embedding	ID $\rightarrow$ ベクトル	Transformer
Attention	トークン間の関係	Transformer block
FFN	位置ごとの非線形変換	Transformer block
Transformer	Attention+FFN を積層	LLM
LM Head	隠れ表現 $\rightarrow$ 語彙 logits	Loss / Generation
Optimizer	重み更新	Training loop

3. 実装順序

下位層から上位層へ作ると、問題の切り分けが容易です。

1. Tokenizer: encode/decode の往復を確認
2. Tensor / autograd: 加算・乗算・行列積・勾配を確認
3. Embedding: token ID からベクトルを取得

4. Linear / LayerNorm: 基本ニューラルネット層を実装
5. Causal Self-Attention: 未来トークンを見ないマスクを実装
6. FFN: Linear→活性化→Linear
7. Transformer Block: Residual + Norm + Attention + FFN
8. LLM: Embedding→Blocks→Norm→LM Head
9. Loss: 次トークン予測の Cross Entropy
10. Optimizer: SGD または Adam
11. train.py: データ取得→forward→loss→backward→update
12. demo/chat: 自己回帰生成

3.1 最小成功条件

- 1 バッチで forward が通る
- loss が有限値になる
- backward で各 Parameter に gradient が入る
- 数十～数百 step で loss が下がる
- checkpoint 保存と復元で同じ出力が得られる
- 短い prompt から自己回帰生成できる

4. Tokenizer と学習データ

4.1 Tokenizer

最初は文字単位 Tokenizer でも構いません。次に単語・subword へ拡張します。Tokenizer の最重要条件は、学習時と推論時で完全に同じ語彙表・ID 対応を使うことです。

```
class Tokenizer:
    def encode(self, text: str) -> list[int]:
        ...

    def decode(self, ids: list[int]) -> str:
        ...

# 必須テスト
text = "こんにちは LLM"
assert tokenizer.decode(tokenizer.encode(text)) == text
```

4.2 学習データの作り方

Language Model 学習では、文章を token 列にし、1 トークンずらした入力 X と正解 Y を作ります。

```
tokens = [10, 25, 8, 91, 7]
X = [10, 25, 8, 91] # 入力
Y = [25, 8, 91, 7] # 次トークン正解
```

- 文字コードを UTF-8 で統一する

- 重複・壊れた行・極端に短い断片を除去する
- 学習用・検証用・テスト用を分離する
- 会話能力を学習させたい場合は **User/Assistant** 形式を明示する
- データ量だけでなく、目的に合う分布を重視する

4.3 データ量の考え方

自作実験では、「巨大なデータ」より「モデル規模に見合う量」と「再現可能な評価セット」が重要です。まず数十万～数百万 token で挙動を確認し、その後に数千万 token へ増やします。

注意 小規模モデルに大量データを投入しても能力が無限に伸びるわけではありません。モデル容量・データ量・学習 step のバランスが必要です。

5. Transformer の中核

5.1 Embedding と位置情報

token ID を d_{model} 次元のベクトルへ変換し、位置情報を加えます。位置情報には固定 Sin/Cos、learned positional embedding、RoPE などがあります。最小実装では learned position embedding が簡単です。

```
x = token_embedding[token_ids] # [B, T, D]
pos = position_embedding[:,T]  # [T, D]
x = x + pos
```

5.2 Causal Self-Attention

各トークンから Query(Q)、Key(K)、Value(V)を作り、 QK^T の類似度に基づいて V を混合します。生成モデルでは未来の token を参照しない Causal Mask が必須です。

```
Q = x @ Wq
K = x @ Wk
V = x @ Wv
scores = (Q @ K.transpose(-2, -1)) / sqrt(head_dim)
scores = causal_mask(scores)
attn = softmax(scores, dim=-1)
out = attn @ V
```

5.3 Transformer Block

```
x = x + Attention(LayerNorm(x))
x = x + FFN(LayerNorm(x))
```

Residual 接続は深いモデルの学習を安定させます。LayerNorm の位置は Pre-Norm 構成が小規模実験でも扱いやすいです。

5.4 FFN

```
h = Linear(D, 4*D)(x)
h = GELU(h)
out = Linear(4*D, D)(h)
```

6. 学習の実装

6.1 Training Loop

```
for step, (x, y) in enumerate(loader):
    logits = model(x)
    loss = cross_entropy(logits.view(-1, vocab), y.view(-1))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if step % 100 == 0:
        print(step, float(loss))
```

6.2 Adam の役割

Adam は各 Parameter の勾配について 1 次・2 次モーメントを保持し、スケールを調整しながら更新します。自作する場合は、Parameter ごとに **m**、**v**、**step** を保持します。

```
m = beta1*m + (1-beta1)*grad
v = beta2*v + (1-beta2)*(grad*grad)
m_hat = m / (1-beta1**t)
v_hat = v / (1-beta2**t)
param -= lr * m_hat / (sqrt(v_hat) + eps)
```

6.3 保存すべき Checkpoint

- model state / 全 Parameter
- optimizer state
- step / epoch
- loss
- Tokenizer 語彙・設定
- model config (vocab, d_model, layers, heads, context)
- 乱数 seed (再現性が必要なら)

6.4 学習が成功しているサイン

観測	良い状態	問題の可能性
loss	緩やかに低下	横ばい: LR/実装/データ
gradient	0 でも極端でもない	NaN/Inf: 数値不安定
生成	データ分布らしい文字列	同じ token 連打: collapse

train vs val	両方改善	train のみ改善: 過学習
速度	step 時間が安定	徐々に遅い: メモリリーク

7. 推論・会話機能

7.1 自己回帰生成

```
ids = tokenizer.encode(prompt)
for _ in range(max_new_tokens):
    x = ids[-context_length:]
    logits = model(x)
    next_logits = logits[-1]
    next_id = sample(next_logits, temperature=0.8, top_k=40)
    ids.append(next_id)
print(tokenizer.decode(ids))
```

7.2 Sampling

greedy は毎回最大 **logit** を選びます。**temperature** を上げると多様性が増え、**top-k/top-p** は低確率候補を切り捨てます。自作 LLM では、生成品質の比較実験がしやすいようパラメータを CLI で変更できるようにします。

7.3 会話化

chat.py では会話履歴を決まったフォーマットで **prompt** へ結合します。ただし、小規模な通常 LM は「会話形式を理解する」保証がないため、会話データを含めた追加学習が必要です。

```
<user> LLM とは何ですか？
<assistant> LLM は...
<user> Transformer との関係は？
<assistant>
```

7.4 入力から学習する場合

会話中の情報を即座にモデル重みへ書き込む方法は、忘却や誤学習を起こしやすいため、まずは「外部メモリ」と「後段の再学習データ追加」を分離するのが安全です。

方式	特徴	用途
Context memory	会話履歴に保持	短期記憶
Retrieval memory	外部 DB から検索	事実・知識の追加
Online fine-tune	重みを更新	小規模な適応実験
Batch retraining	検証後まとめて学習	再現性のある更新

8. GPU 化

8.1 GPU 化の基本

PyTorch を使う場合、モデルと入力 Tensor を同じ device へ置くだけで、多くの行列演算は CUDA へ移ります。

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
x = x.to(device)
y = y.to(device)
```

8.2 GPU 化で重要な点

- CPU と GPU を頻繁に往復させない
- 小さすぎる batch は GPU を使い切れない
- VRAM 不足時は batch/context/model 次元を下げる
- mixed precision(fp16/bf16)は高速化できるが、まず fp32 で正しさを確認する
- 学習速度は tokens/sec と step/sec の両方を記録する

8.3 CPU 版と GPU 版を分ける意味

教育目的の CPU 版では内部実装の理解を優先し、GPU 版では PyTorch/CUDA を利用して実験回数を増やす、という二段構えが有効です。構造・Tokenizer・評価方法を共通化すれば比較も容易です。

9. 評価とデバッグ

9.1 必ず測る指標

指標	意味	用途
Loss	平均的な予測誤差	学習進行
Perplexity	$\exp(\text{loss})$	LM 品質の目安
tokens/sec	生成・学習速度	性能比較
Validation loss	未知データ誤差	過学習検出
Parameter count	モデル規模	比較再現性
VRAM/RAM	必要資源	実用性

9.2 典型的な不具合

症状	確認箇所	対策
loss が NaN	LR、softmax、division	LR 低下・clamp・安定 softmax
loss が下がらない	ラベル shift、gradient	X/Y ずれ、requires_grad 確認
同じ文字を反復	sampling、データ偏り	temperature/top-k、データ見直し
保存後に結果が違う	config/tokenizer	checkpoint に設定も保存

GPU が遅い	batch、転送	batch 増加、device 統一
評価が高すぎる	train leakage	holdout を完全分離

9.3 回帰テスト

chat.py や Tokenizer を改修するたび、既知入力・壊れた入力・境界条件を自動テストします。LLM は生成が確率的でも、前処理・ルーティング・保存形式などは決定的にテストできます。

10. Semantic 拡張

通常 LM の次段階として、「生成する」だけでなく「意味を取り出し、分類・ルーティング・Unknown 判定をする」層を追加できます。これは小型 LLM をタスク分配や Semantic OS の部品として使う際に有効です。

10.1 基本構成

```

Text
↓
Base LLM / Encoder
↓ hidden representation
Semantic Projection
↓
Semantic Vector
├─ Centroid / kNN Router
├─ Class Radius
├─ Unknown Detection
└─ Task / Purpose Selection

```

10.2 Semantic Projection

ベースモデルを凍結し、小さな Projection Head だけを学習すると、生成能力を壊さずに意味空間を調整できます。

```

base_vector = frozen_llm.encode(text)
semantic_vector = projection(base_vector)
loss = separation_loss + lambda_preserve * preservation_loss

```

10.3 ベクトルだけでは十分でない理由

- ラベル・目的・関係などの意味付けが必要
- ベクトル空間の距離尺度や閾値の定義が必要
- Unknown を判定するには既知分布の境界が必要
- 学習時の発話パターン・難易度・データ由来を保持した方が診断しやすい
- 運用時にはルーティング結果と実行先の意味を結び付ける必要がある

11. 実用化への拡張

11.1 改良の優先順

1. Tokenizer を subword 化
2. データ量・品質を増やす
3. モデル幅/層数を増やす
4. 長い context へ拡張
5. 学習率 schedule・weight decay 導入
6. AMP/mixed precision
7. checkpoint resume
8. 評価ベンチマークの固定
9. Instruction / Chat fine-tuning
10. Retrieval / Memory
11. Semantic Router / Agent 連携

11.2 実用 LLM とのギャップ

自作小型 LLM は構造理解と独自機構の研究には十分有用ですが、汎用商用 LLM と同等の知識量・推論力を得るには、桁違いのモデル規模、データ、計算資源、データ整備、安全評価が必要です。したがって、独自 LLM では「小型でも価値が出る機能」を狙うのが現実的です。

方向	小型自作 LLM と相性が良い例
ドメイン特化	特定技術・社内用語・機器操作
Semantic Router	タスク分類・実行先選択
Edge/Local	オフライン・低遅延・プライバシー
研究	表現空間・Unknown・継続学習
Agent 部品	Purpose 抽出・Intent 分類・ツール選択

11.3 Git 管理のすすめ

大きな変更はブランチを分け、実験結果とソースを対応付けます。release/tag には「モデル config」「checkpoint」「Tokenizer」「評価結果」「再現コマンド」を残すと、後から比較できます。

```
main
├─ v0.1 minimal CPU
├─ v0.2 full backward / Adam
├─ v0.4 GPU training
├─ v1.x chat / learning experiments
└─ semantic branch / separate repository
```

12. 推奨ロードマップ

Phase	目的	完成条件
A. Minimal	LM の全体を動かす	loss 低下 + 生成
B. Correctness	自動微分・保存を安定化	回帰テスト合格
C. GPU	実験速度を上げる	CUDA 学習・推論
D. Data	多様なデータで改善	validation 改善
E. Chat	対話形式へ適応	複数 turn 応答
F. Semantic	意味分類・Unknown	holdout 評価
G. Integration	Router/Agent/QHA へ接続	タスク実行まで一貫

12.1 最短で作るなら

- PyTorch を使う
- Tokenizer は最初は文字単位
- 2 層・d_model=64 程度から始める
- 学習データは小さくてもよいので clean にする
- train loss だけでなく holdout を用意する
- 生成を確認してからモデルを大きくする
- 独自性は Semantic/Memory/Router など上位機能で出す

12.2 「自作した」と言える範囲

自作には段階があります。PyTorch の Tensor/autograd を使って Transformer 構造・学習ループを自作するだけでも十分に「自作 LLM」です。さらに NumPy で Tensor や自動微分まで作れば、より低い層まで自作した研究・教育実装になります。重要なのは、どこまでを自作したかを明確にすることです。

付録 A. 最小モデルの擬似コード

```
class MiniLLM(nn.Module):
    def __init__(self, vocab, d_model, n_layers, context):
        super().__init__()
        self.tok = nn.Embedding(vocab, d_model)
        self.pos = nn.Embedding(context, d_model)
        self.blocks = nn.ModuleList([Block(d_model) for _ in range(n_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab, bias=False)

    def forward(self, ids):
        B, T = ids.shape
        x = self.tok(ids) + self.pos(torch.arange(T, device=ids.device))
        for block in self.blocks:
            x = block(x)
        return self.head(self.norm(x))
```

付録 B. 実験記録テンプレート

項目	記録例
Version / branch	v0.x.x
Date	YYYY-MM-DD
Dataset	ファイル名、token 数
Model	vocab / d_model / layers / heads / FFN
Optimizer	Adam, lr=...
Epoch / step	...
Train loss	...
Validation loss	...
Perplexity	...
Speed	tokens/sec
GPU	型番 / VRAM
Checkpoint	ファイル名
変更点	1〜3 行
結論	採用 / 保留 / 却下