PART 3 / MICROGPT 実装

Mojo入門

MicroGPT 実装

Karpathy の microgpt を Mojo・MAX・PyTorch・MLX で書き直す

Hiromi / Mojo 0.26.2 / MAX / PyTorch / MLX

github.com/h3adeu/mojo-from-scratch

AGENDA

Part 3 の流れ — 6 セクション

01

microgpt とは

Karpathy 製・依存ゼロ最小 GPT

02

microgpt.py 解読

Value・gpt・学習・推論

03

Mojo で書き直す

B1〜B7 のモジュール構成

04

MAX で高速化

Graph / Module / Weight

05

PyTorch / MLX 比較

MPS GPU / Unified Memory

06

全体総括と性能比較

99.5 秒 → 3.0 秒の旅路

01

CHAPTER 01

microgpt とは

Karpathy 氏の依存ゼロ最小 GPT 実装。

WHAT IS MICROGPT?

依存ゼロ・単一ファイルの最小 GPT

> ***The most atomic way to train and run inference for a GPT in pure, dependency-free Python.***
> *This file is the complete algorithm. Everything else is just efficiency.*

AUTHOR

Andrej Karpathy

深層学習・LLM 教育で著名。@karpathy。

SOURCE

GitHub Gist

リポジトリではなく単一の Gist で公開。

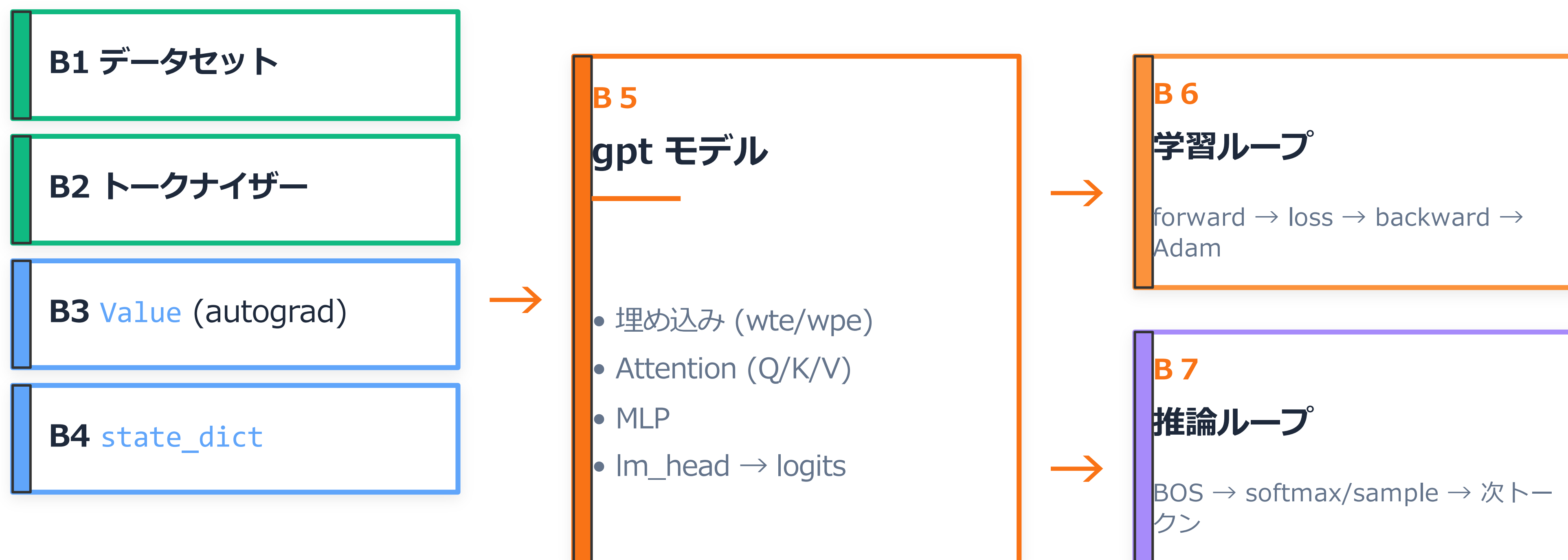
STDLIB

標準ライブラリのみ

PyTorch などの依存なし、純粋 Python。

STRUCTURE

microgpt.py の 7 ブロック構成



02

CHAPTER 02

microgpt.py を読み解く

autograd・gpt・学習ループの内側。

AUTOGRAD CORE

Value — 履歴つきスカラー

Value 1 個 = データ + 勾配 + 履歴

data	順伝播の値（スカラー）
grad	逆伝播で計算する勾配
_op	演算の種類（add / mul など）

forward で値を作る → *backward* で勾配を流す。

→ 微分の連鎖をトポロジー順に辿るだけで autograd が成立する。

autograd.py

```
a = Value(2.0)
b = Value(3.0)
c = a * b   # c.data == 6.0

c.backward()

# 逆伝播で grad が決まる
a.grad == 3.0  # ∂c/∂a = b.data
b.grad == 2.0  # ∂c/∂b = a.data
```

BUILDING BLOCKS

ユーティリティ演算 — 全部 Value 上で実装

linear

```
out = W · x
```

重み行列との行列ベクトル積。出力1 個ごとに W の行と x の各要素を足し合わせる。

rmsnorm

```
out[i] = x[i] / √(mean(x²) + ε)
```

二乗平均で正規化。LayerNorm より軽量で、Llama 系でも採用されている。

softmax

```
p[i] = exp(x[i] − max) / Σ exp(...)
```

数値安定化のため最大値を引く。logits → 確率分布に変換する。

***Value* のスカラー演算を組み合わせるだけで、行列演算も*backward* もすべて表現できる。**

MODEL

gpt — forward の流れ

1. Embedding

`wte + wpe (token + 位置)`

2. Attention

`Q · Kᵀ / √d → softmax → V`

3. MLP

`rmsnorm → fc1 → ReLU → fc2 (残差)`

4. lm_head

`logits ← n_embd → vocab_size`

STATE_DICT のパラメータ

学習で更新される重み行列の束

wte	[vocab, n_embd]	トークン埋め込み
wpe	[block, n_embd]	位置埋め込み
wq/wk/wv	[n_embd, n_embd]	Attention 重み
wo	[n_embd, n_embd]	Attention 出力
fc1/fc2	[4n, n] / [n, 4n]	MLP
lm_head	[n_embd, vocab]	最終線形

TRAIN & INFER

学習ループと推論ループ

学習ループ	推論ループ
1. 文書取得 → トークン化	1. BOS から開始
2. forward (gpt) で logits	2. forward で logits 取得
3. cross-entropy で loss	3. logits / temperature
4. loss.backward() で勾配	4. softmax → 確率分布
5. Adam で state_dict 更新	5. multinomial でサンプル
6. 学習率を線形減衰	6. BOS が出るまで繰り返す
ハイパー: lr=0.01, β1=0.85, β2=0.99	*temperature: 低 → 無難 / 高 → 多様*

03

CHAPTER 03

Mojo で書き直す

型安全と所有権で再構築する。

MICROGPT_MOJO

B1～B7 のモジュール構成

microgpt_mojo/

```
├── b01_dataset.mojo
├── b02_tokenizer.mojo
├── b03_value.mojo
│       # Tape (SoA autograd)
├── b04_state_dict.mojo
├── b05_ops.mojo
│       # linear / rmsnorm / softmax
├── b05_gpt.mojo
├── b06_train.mojo
├── b07_infer.mojo
├── main.mojo
└── input.txt
```

microgpt.py との対応

class Value	→ **B3 Tape (SoA)**
linear / rms	→ **B5 ops**
gpt() 関数	→ **B5 gpt**
学習ループ	→ **B6 train**
推論ループ	→ **B7 infer**
オブジェクト参照	→ **List[T] + ID 参照**
Python の dict	→ **Mojo struct**

DESIGN SHIFT

オブジェクト参照 → SoA テープ

Python: オブジェクト参照

```
class Value:
    self.data, self.grad
    self._prev = (a, b)  # 親への参照
```

Mojo: SoA テープ + ID

```
struct Tape:
    var data: List[Float64]
    var grad: List[Float64]
    var prev_a, prev_b: List[Int]  # ID
```

なぜ SoA に変えるのか

- List[T] に格納するため、カスタム copy/move を避けて Mojo の所有権規則に乗せやすい
- ID 参照ならノードがリストの中で動いても問題なし — 参照ではなく整数で繋ぐ
- メモリ局所性が上がる — data だけを連続走査できれば SIMD 化の余地が広がる

04

CHAPTER 04

MAX で高速化

Tape の限界の先へ ― グラフ実行という選択。

WHY MAX?

Tape の限界とグラフ実行への移行

Tape 方式 (Mojo / Python)	MAX 方式
逐次・スカラー実行	**グラフ → コンパイル → 実行**
• 演算ごとにノードを 1 個追記 • 推論 1 ステップで数千〜数万ノード • SIMD/GPU の活用が難しい • 自動微分には向くが推論は遅い	• 先に計算グラフを構築 • 演算融合 (kernel fusion) • SIMD/GPU への自動マッピング • メモリレイアウト最適化
`t.add(a, b) → 追記 → 即実行`	`Graph → load → 一括実行`

MAX CONCEPTS

Graph / Module / Weight — 3 つの中心概念

GRAPH

計算の設計図

入力 → 演算 → 出力を実行前に定義する。InferenceSession に渡してコンパイル。

```
with Graph(...) as g:
    out = ops.matmul(w, x)
```

MODULE

層と重みのまとまり

PyTorch の nn.Module に近い概念。複数の Weight と演算を 1 単位で管理。

```
class LinearLayer(Module):
    self.w = Weight(...)
```

WEIGHT

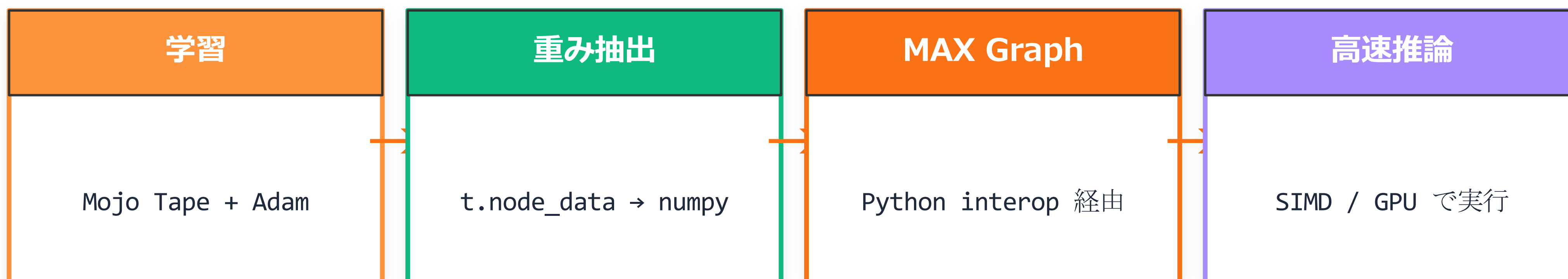
学習可能パラメータ

入力テンソル（毎回変わる）と重み（固定）をグラフ上で明確に区別する。

```
Weight(TensorType(...),
  default_value=w_init)
```

MAX × MOJO

学習は Mojo / 推論は MAX へ



役割分担

● モデルの計算グラフを組む	`MAX (Python API)`	● 標準にない演算を高速実装	`Mojo (custom ops)`
● 学習 (autograd / Adam)	`Mojo Tape`	● 推論の高速実行	`MAX (InferenceSession)`

学習は *Tape* のまま活かし、推論だけ *MAX* に切り替える — 段階的高速化の第一歩。

05

CHAPTER 05

PyTorch / MLX 比較

学習も含めて速度を取りに行く。

PYTORCH

学習も含めて高速化 — PyTorch + MPS

PyTorch を使う動機

1. **学習も推論も autograd**
   MAX は推論専用。学習込みなら PyTorch。
2. **Apple Silicon MPS 対応**
   GPU バックエンドで CPU の数倍〜十数倍。
3. **nn.Module のエコシステム**
   Adam / Linear / Embedding が即使える。
4. **Mojo から interop で操作**
   main.mojo から Python 層を駆動できる。

microgpt_torch.py

```
device = "mps" if torch.backends.mps
         .is_available() else "cpu"

model = MicroGPT(...).to(device)
opt = Adam(model.parameters(),
          lr=lr, betas=(b1,b2))

logits = model(tokens[:, :-1])
loss = F.cross_entropy(...)
loss.backward();  opt.step()
```

MLX

Apple Silicon 専用 — MLX で最速

UNIFIED MEMORY

CPU / GPU が同じメモリ

PyTorch MPS は CPU↔GPU コピーが発生。MLX は不要
→ これが最大の差。

LAZY EVALUATION

全演算がデフォルト遅延

mx.eval() で確定するまで全演算がキューに溜まり、その時点で最適化される。

PyTorch (MPS) vs MLX

観点	PyTorch	MLX
メモリ	別々 / コピー	**統合 / コピー不要**
API スタイル	オブジェクト指向	**NumPy 風 関数型**
遅延評価	一部のみ	**全演算デフォルト**
対応 HW	汎用 + MPS	**Apple Silicon 専用**

→ 結果として *MLX* が最速 *(3.0 秒)*

06

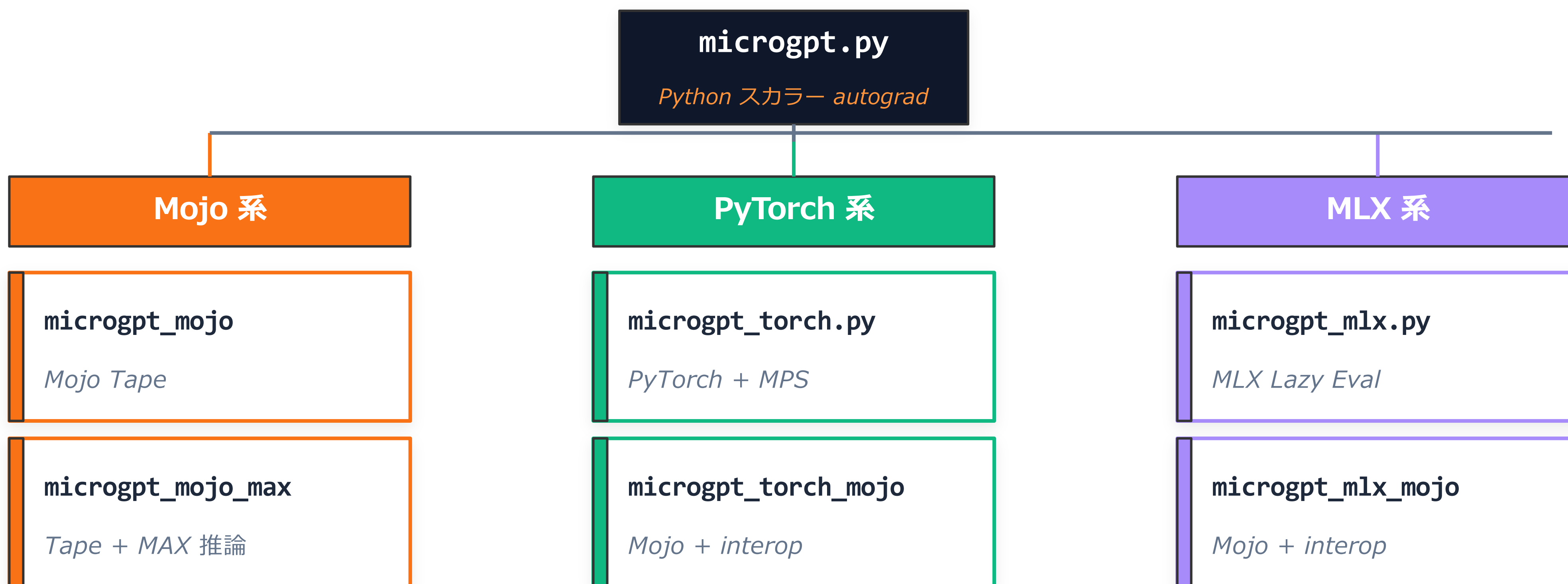
CHAPTER 06

全体総括と性能比較

99.5 秒から 3.0 秒へ — Mojo と ML スタック。

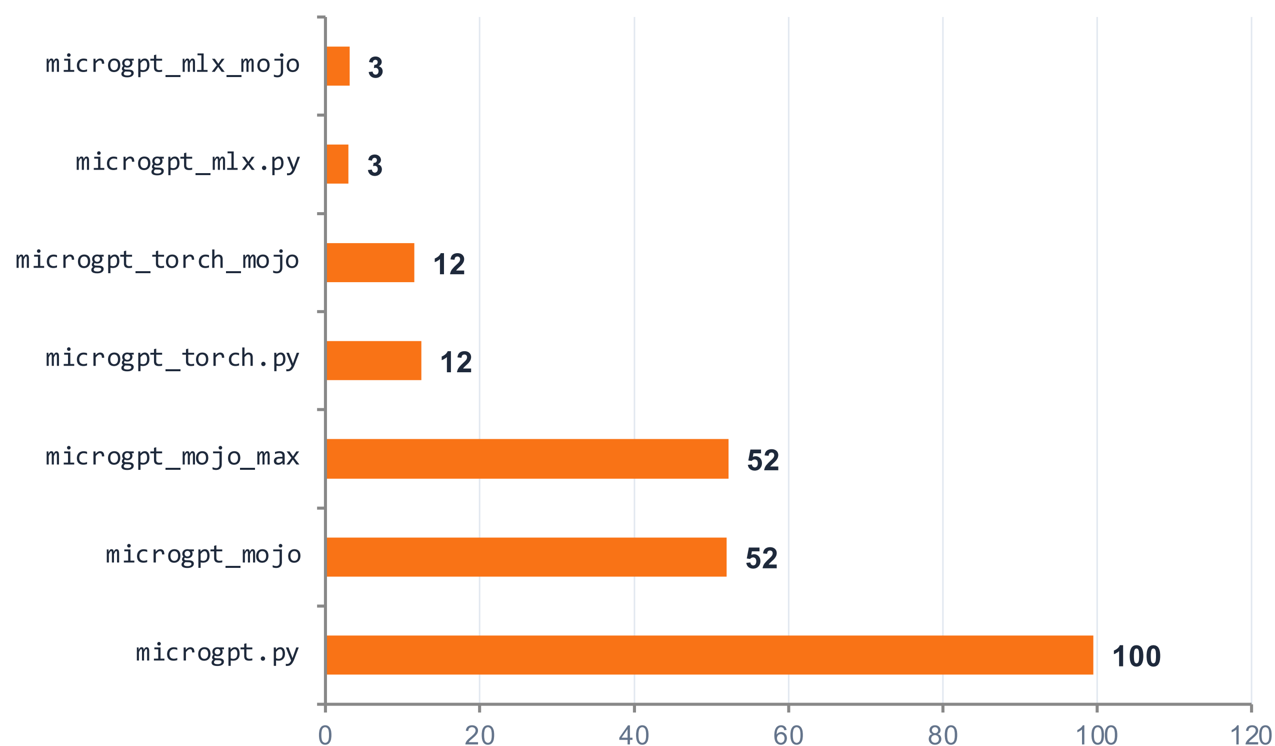
LINEAGE

7 つの実装の系譜



PERFORMANCE

学習 1000 ステップの壁時計時間



INSIGHTS

33×

MLX は Python 版の約33倍速い (Unified Memory)

8×

PyTorch は Python 版の約8倍 (行列演算)

1.9×

Mojo は Python 版の約2倍 (静的型・ネイティブ)

≈

Mojo + interop はPython 版とほぼ同等

CONCLUSION

スタックの役割分担と本書の結論

Mojo	ホットパスの型安全実装	静的型 / 所有権 / *interop*
MAX	推論の最適化 (Graph API)	コンパイル・*HW* 抽象化
PyTorch	学習フレームワーク (MPS 対応)	*autograd・nn.Module*
MLX	Apple Silicon 特化の学習	*Unified Memory・Lazy Eval*

MOJO の真価

Python の書きやすさ +

C のような速度

型システム・所有権で
安全性と性能を両立し、
Python エコシステムに
乗り続けられる。

本書で学んだ仕組みは、より大きなモデルを読み解く土台になる。