

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

1B 日本語 GPT-NeoX を学習した実践記録

開発・著者 ao-oo

Abstract

本稿では、リソースが限定された個人開発環境において、1B パラメータ規模の LLM を実用的な日本語能力へ引き上げる手法を解説する。特に、LLM やトークナイザの設計や学習、使い方など、基本的な開発から GPU・TPU の使い方、コーパスデータの作り方、使い方などの高度な内容を解説する。

1. Introduction: なぜ今、LLM を作るのか

LLM 開発は巨大企業だけの特権ではない。本稿では、Kaggle や Google Colaboratory を使用し個人が技術力で大規模モデルを制御する意義を述べる。

2. Model Architecture: 1B の設計

- **Base Model:** GPT-NeoX 1B を採用。
- **Tokenizer:** Sentence Piece (spm) を活用し、AutoTokenizer でのロードを可能にした設計。
- **Context Length:** 学習では 512 に設定しており、1024 でも出力可能。

Architectures	GPTNeoXForCausalLM	GPT-NeoX ベースの因果言語モデル（次の単語を予測するモデル）
Hidden Size	1536	各層のベクトルの次元数。モデルの「表現力」に直結します。
Hidden Layers	24	モデルの「深さ」。層が多いほど複雑な概念を理解できる。
Attention Heads	16	注意機構の数。一度に複数の単語間の関係を捉える力。
Intermediate Size	6114	フィードフォワード層の内部次元。通常、Hidden Size の 4 倍に設定される。
Max Position Embeddings	1024	モデルが一度に扱える最大トークン数（コンテキスト長）。
Activation Function	gelu	層の間で使われる活性化関数。GPT 系で標準的な設定。

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

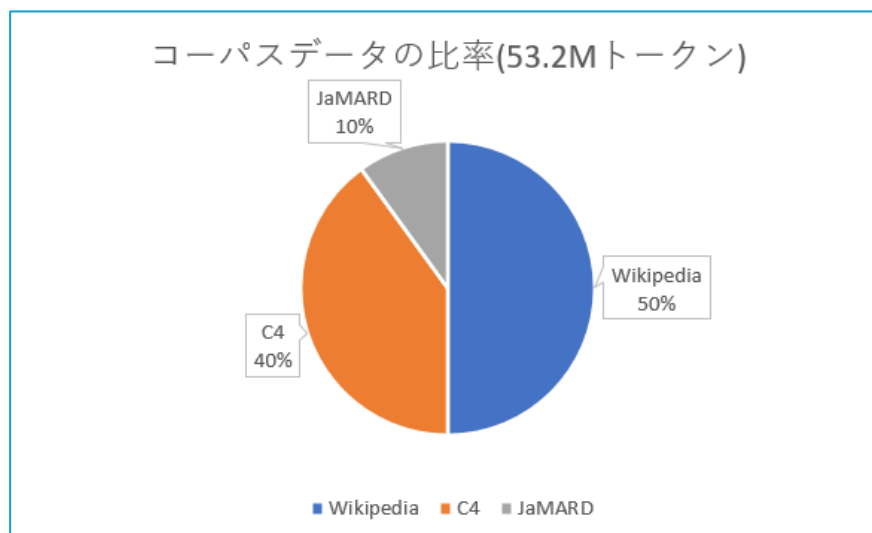
3. Data Engineering: コーパスの重要性

LLM の知能はデータの質に依存する。

- **Dataset Mixing:** Wikipedia、Wikibooks、C4、JaMARD の 4 種をブレンド。
- **計算データの重要性:** 単なるテキスト量ではなく、論理的思考を養うための web データ (C4) や計算データ (JaMARD) の比率が鍵となる。計算データが多いと数値や記号が出力に出やすくなるため Fine Tuning の前に Wikipedia など日本語データを少し入れるなどの工夫も必要になる。
- **Scaling Laws:** パラメータ数と必要トークン数の関係 (Chinchilla Scaling Laws) に基づいた学習計画。1 パラメータ×20 トークンを目標とする。

1 回あたりの学習で使用するデータのおおよその比率。

Datasets ライブラリの `interleave_datasets` を使用方法や、データの比率を決めて分割してから結合するという方法もある。



4. Hardware Optimization: GPU と TPU の使い分け

開発フェーズに合わせた計算資源の最適化を解説する。

- **T4 GPU (Google Colab/Kaggle):** 軽量のデバッグや少量の推論テスト。
- **TPU v5e-8 (Kaggle):** 1 回あたり 100M トークン規模の高速学習。
ParallelLoader を用いたスループットの最大化。
(バッチサイズによっては 1.92it/秒で速度が出ることもある)

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

5. Training Methodology

- **Continuous Pre-training (CPT)**: 知識と日本語を定着させる手法。
- **Instruction Tuning**: 指示に従う力を養う手法。
- **Evaluation Metrics**:
 - 予測精度の指標として ForCasualLLMLoss と Evaluate_Loss を採用。
 - 検証用テキストセットに対する **Average Perplexity (PPL)** による定量的評価を採用。

6. AI development tips

1. LLM とトークナイザの設計

• LLM の設計では Kaggle や Google Colab の環境で GPU メモリに収まるパラメータで設計をする必要がある。特に無料枠で利用できる T4 はメモリが 16GB と少なく、1B が限界である。Kaggle の TPU の場合 330GB とかなり余裕があるもののコーディングやバージョンの不整合など環境構築の難易度は高い。また1度学習を開始するとパラメータは変えられないため使用できる環境を見て、実行可能なサイズを見極める必要がある。

• 今回のトークナイザの設計では日本語を使用するため Sentence Piece を採用。日本語は英語のようにスペースで区切られないため、形態素解析 (単語分け) を行わずにサブワード (単語より小さい単位) に分割できる Sentence Piece が非常に有効である。

(トークナイザの学習コード。今回の LLM で使用)

```
13  spm.SentencePieceTrainer.Train(  
14      input="all.txt",  ← 学習用コーパス。日本語だけでなく記号や計算データを入れると良い  
15      model_prefix="spm_ja_gpt2",  ← 出力モデル名  
16      vocab_size=32000,  ← 語彙数。辞書のようなもので、単語を保存できるサイズ。  
17      model_type="unigram",  ← 確率ベースの分割アルゴリズム  
18  
19      # 日本語向け重要設定  
20      character_coverage=0.9995,  ← データに含まれる文字の何パーセントを、トークナイザのトークン候補に含めるかを定義する  
21      normalization_rule_name="nfkc",  ← Unicode正規化 (全角半角の統一)  
22      remove_extra_whitespaces=True,  
23      split_by_whitespace=False,  
24      split_by_number=False,  
25      split_by_unicode_script=False,  
26  
27      # GPT系ではほぼ必須  
28      byte_fallback=True,  ← 未知語をバイト単位で処理し、エラーを防ぐ  
29  
30      # special tokens  
31      unk_id=0,  
32      pad_id=1,  
33      bos_id=2,  
34      eos_id=3,  
35  )
```

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

2. コーパスデータの作り方と使い方

- ・コーパスデータは単純に日本語 (Wikipedia、Wikibooks など) を入れるだけでできるものではなく、計算データ (JaMARD) など、推論力を伸ばすデータも必要になる。また、**3. Data Engineering: コーパスの重要性**にあるように順番に入れるのではなく比率を決めて混ぜて入れることで破壊的忘却※1を防止し、出力を安定させることができる。Instruction tuning でも同様である。

- ・コーパスデータの中には記号が混ざっているものもある。LLM に記号を正しく使わせるには日本語データの記号を一定の水準で削除し、正しい使い方は計算データで学習させるのが有効である。ただし句読点など文を構成する上で重要な記号まで消すと文法が壊れる危険性があるため文法が壊れない程度に削除すると良い。

3. 学習/評価のヒント

- ・TPU (Tensor Processing Unit) の性能を最大限に引き出すには、ParallelLoader を活用してデータ転送のボトルネックを解消することが不可欠である。また、バッチサイズやシーケンス長を固定し、計算グラフの再コンパイルを防ぐことで、**1.92it/s** といった高いスループットを維持できる。ただし TPU の使用は難易度が高いため、まずは GPU でコードを組み、学習をしながら TPU を使ってみることを勧める。(実用化に 8 時間かけました)

- ・学習で利用できる評価指標には ForCasualLM Loss と Evaluate Loss がある。前者は自動回帰型の LLM において、次のトークンをどれだけ正確に予測できたかを評価するための誤差関数であり、事前学習がどれだけ進行しているかの確認に使用できる。後者は学習中または学習後の LLM が、訓練に使用されていない未知のデータに対して、どれだけ予測を誤ったかを数値化した指標であり、事前学習後の Instruction tuning で使用できる。両者は過学習の検知にも使用できる。過学習が起きた場合、ForCasualLM Loss は 1.0 以下まで低下し、Evaluate Loss は上昇することが多い。

- ・PPL は、モデルが次の単語を予測する際の「迷い」を数値化したものである。数値が低いほど、モデルは自信を持って正確な日本語を生成できていることを示す。学習初期は数値が高いが学習していくうちに数値が右肩下がりに収束していれば、学習は成功していると判断して良い。また、単一の Loss だけでなく、複数の検証用文章で平均 PPL を取ることで、特定のデータへの過学習を防ぎ、日本語能力を客観的に測定することも可能になる。

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

なお、事前学習の途中で推論をすると、このようになる。

```
<S>
太陽は何色ですか？ `光るしかないのは` Handa Knowfromaです。 Photelは、宇宙で見たことで地球に
```

途中ということもあり出力がおかしい状態で、出力も途切れている。この問題は次の Instruction Tuning で解決することができる。

4. Instruction Tuning のヒント

・Instruction Tuning は、指示 (Prompt) とそれに対する期待される回答 (Response) のペアを用いて学習を行うプロセスである。(SFT などと混同されることもあるが、ここでは指示に追従するように学習する Instruction Tuning について解説する。) そのため使用するデータはコーパスとは違い Prompt と Response が分離したデータを使用する。

(↓実際に使用しているデータ。入力は prompt、出力は target のコラムにしている。)

```
19 {"prompt": "GitHub Actionsとは何ですか？", "target": "GitHub Actionsは、ビルド、テスト、デプロイのパイプラインを自動化"}
20 {"prompt": "「コストセンター」と「プロフィットセンター」の違いについて説明してください。", "target": "もちろん、その2つの"}
21 {"prompt": "振られたばかりの友人を元気づけるには、どんな方法があるのでしょうか？", "target": "お友達が最近振られたとのこ"}
22 {"prompt": "「侵略戦争」と「防衛戦争」という概念について詳しく教えてください。", "target": "「侵略戦争」と「防衛戦争」は"}
23 {"prompt": "日本語の敬語について学びたいのですが、適切な教材やリソースはありますか？", "target": "もちろん、日本語の敬語"}
24 {"prompt": "太陽は暑いですか？", "target": "しかし、太陽の表面は華氏10,000度、世界では摂氏5,500度であることが知られてい"}
25 {"prompt": "なぜ人はグレイフル・デッドが好きなのか？", "target": "多くの音楽愛好家にとって、グレイフル・デッドを聴く"}
26 {"prompt": "「日本の文化に根ざした芸術」とは何か？", "target": "「日本の文化に根ざした芸術」は、日本の歴史、伝統、自然、"}
27 {"prompt": "次の文章から、プラム、アプリコット、ピーチの原産地はどこでしょうか？", "target": "プラム、アプリコット、ピー"}
28 {"prompt": "[INST]次の質問の回答はなんですか？日本語で回答してください。\\nQ: 江戸時代以降川遊びなどに用いられた、屋根と座"}
29 {"prompt": "ディーン・デルはどのようなメディアに登場したのですか？箇条書きにし、各行に「{メディア・アウトレット}」({日付"}
30 {"prompt": "国内総生産 (GDP) とは何ですか？", "target": "国内総生産 (GDP) とは、ある国や国が特定の期間に生産・販売したす
```

・Instruction Tuning で重要なのは損失計算のマスク処理とトークナイザの EOS トークンを出力させること、この2つを破壊的忘却が起きないように進めることである。前者は Prompt を損失計算には入れないことで LLM が Prompt を暗記することを防ぎ、学習を効率化させることができる。なおこのプロジェクトの LLM や最近の LLM は Response に Prompt を出力させないように学習させることができないため、チャットボットなどのタスクでは推論処理で Prompt の分のトークン数を測り、出力のみを切り取る処理が必須である。

(↓マスク処理。Prompt など損失計算をさせない箇所は-100 で埋める)

```
75
76 # 1. すべて -100 (計算対象外) で初期化したラベルを作成
77 labels = [-100] * len(input_ids)
78
79 # 2. 応答部分 (prompt_len 以降) だけ、元の input_ids をコピーして学習対象にする
80 for i in range(prompt_len, len(input_ids)):
81     labels[i] = input_ids[i]
82
83 # 3. パディング部分も -100 で埋めて、計算に含まないようにする
84 if padding_len > 0:
85     labels = labels + [-100] * padding_len
86
```

本論文は引用を許可するが、出典 (GitHub の URL) を明記すること。

後者は出力を句点で止めるなど文章をきれいに止めるものである。GPT 系などの次トークン予測は推論で `max_new_tokens` などの引数を設定しないとほぼ無限に出力される。そこで適切な位置でトークナイザの EOS トークンを出力するよう配置することで、出力を EOS トークンで終了させることができる。※2

```
if padding_len > 0:
    ↓ 末尾に eos_token_id を追加することで出力を適切に止めることができる
    input_ids = input_ids + [tokenizer.eos_token_id] * padding_len

    labels = labels + [-100] * padding_len
```

Instruction Tuning は LLM 自体に事前学習での日本語や記号の使い方、知識が一定以上あることが前提なので、破壊的忘却を防ぐためにデータは比率を設定して混ぜる必要がある。

5. 評価のヒント

・Instruction Tuning が完了し、満足する結果が出た後は、インターネットに公開したり、チャットボットの UI に入れたりしても良いのだが、完成したモデルを `lm-evaluation-harness` というライブラリを使用することでスコアとして数値化することもできる。事前学習が途中の LLM を評価した結果は以下の通りである。

Tasks	Version	Filter	n-shot	Metric	Value	Stderr
hellaswag	1	none	0	acc	↑ 0.225 ±	0.0669
		none	0	acc_norm	↑ 0.275 ±	0.0715

タスクは Hellaswag (4 択問題の形式で、全く知識がない状態でランダムに回答した場合のスコアは最低 0.25 になる。) を使用している。学習が進んでいくと `acc_norm` という箇所の数値が上がっていく。

7. Conclusion

本プロジェクトを通じて、以下の結論を得た。

・**個人開発の可能性**: 巨大な企業でなくとも、Kaggle GPU/TPU 等のクラウド資源と適切な実装を組み合わせることで、1B 規模の LLM を制御し、学習させることが可能である。

・**データの重要性**: 単なる知識だけでなく、計算データを適切な比率で混ぜることが、推論力の向上に直結することを実証した。

・**今後の展望**: 今回は強化学習 (RLHF) までは至らなかったが、構築したモデルは指示に従う基礎ができており、さらなるファインチューニングの土台として十分な性能を有している。

※1 破壊的忘却とは新しい知識やタスクを学習する際に、過去に学習した情報を大幅に忘れてしまう現象のこと。

※2 `tokenizer.pad_token = tokenizer.eos_token` とする場合 PAD でも可能