目次

実践 PEFT
～ご家庭の GPU で LLM fine-tuning～

1　はじめに

1.1　本書の目的

ChatGPT が登場してから 2 年、Large Language Model (LLM) は世界を席巻しバズワードの代表格となりました。今や AI によるちょっとした推論なら、プログラムを書かずとも自然文で命令するだけで簡単に実行できます。プロンプトエンジニアリング時代の到来です。

しかしだからと言って機械学習が不要になったわけではありません。プロンプトエンジニアリングで達成できる性能は限定的であり、教師データを用いて直接モデルの重みパラメータを最適化する fine-tuning の方がより高い精度を狙えます。またプロンプトエンジニアリングはモデルが変わると最適なプロンプトも変わってくるため再度プロンプトをチューニングする必要がある一方で、fine-tuning であれば学習を実行するためのデータとプログラムは同じものを使いまわすことができます。

一方で LLM は非常に巨大なため、個人の GPU で fine-tuning を行うのは困難です。例えば GPT-3.5 のパラメータ数は約 3550 億で[*1]、これを

[*1] なお GPT-4 のパラメータ数は非公開です。

全て 32bit の浮動小数点数 (float) で扱おうとするとモデルをメモリに載せるだけで 1.4TB も必要になります。これに対し個人で購入できる GPU のメモリは比較的購入しやすいもので 8GB〜12GB、業務用に片足を突っ込んだ価格のハイエンドモデルである NVIDIA の Geforce RTX 4090 でも 24GB しかありません。さすがにこれでは学習どころか推論もままならないため、この本でも取り扱いはしません。

一回りパラメータ数が小さい GPT-2 ならどうでしょうか？　こちらはパラメータ数 15 億で、32bit float なら 6GB で収まります。これなら fine-tuning もできる……とはいきません。推論時と異なり、学習時にはモデル自体が必要とするメモリと同じだけ、**勾配**と呼ばれるパラメータ更新の差分を管理するためのメモリも用意しなければなりません。つまり 6GB × 2 = 12GB が必要です。fine-tuning の手法によっては勾配に加えてそのモーメントもモデルのサイズ分、またはこの 2 倍保持しておく必要があり、これも含めると最大で 6GB × 4 = 24GB も要求します。さらには学習中に行われる勾配順伝播・逆伝播の計算にも GPU メモリは必要となり、庶民的な GPU では動かすのは厳しいです。

またメモリにどうにか載せたとしても、LLM の fine-tuning は精度を出すのが難しいです。事前学習済みなので一から学習するよりはデータが少なくて済むものの、パラメータ数が大きい分 fine-tuning にも大量のデータが必要になるでしょう。

これらの問題を一挙に解決するのが、本書の主題である Parameter Efficient Fine Tuning (PEFT) です。PEFT は LLM のパラメータを固定したまま fine-tuning します。どういうことかというと、LLM の入力であったりレイヤーの間であったりに追加で少数の学習可能な重みを含む計算を挿入し、これだけを学習するのです。どこにどのような重みを組み込むかは手法によって異なり、これらの総称が PEFT です。

学習するパラメータを最小限にすることで勾配やモーメントに必要なメモリはごくわずかで済むため、モデル自体がメモリに載ればあとはバッ

チサイズ次第で fine-tuning を動かすことができます。また学習するパラメータ数が小さいため、データが少ない状況下でも過学習を起こしにくく精度を出しやすいです。

しかし一体どうやって少数のパラメータだけでデータを学習できるのでしょうか？　本書ではそのテクニックを大きく 3 つのカテゴリに分けて紹介していきたいと思います。

1.2 本書の内容

本書はさまざまな PEFT 手法を紹介しつつ、テキスト分類タスクに対してその手法で公開 LLM を fine-tuning し、ある程度の精度が出ることを通して学習の成功を確認します。せっかく LLM の fine-tuning を行うならテキスト生成のタスクに fine-tuning を行いたいところですが、これだときちんと学習できているかどうかを定量的に評価することが難しいです。そこで Accuracy や AUC といった明確な評価指標のあるテキスト分類タスクで実験するというわけです。

構成についてですが、まず 2 章で導入としてタスクやベースとする LLM について紹介し、その fine-tuning がご家庭の GPU では素朴には難しいことを実験で確認します。その後 3〜5 章で大まかに 3 種類ある PEFT 手法について解説しつつ、実際にポエム分類タスクを解かせてその結果を見ていきます。

本書の実験は実際に python のプログラムを書いて実施しており、解説のためその一部を載せることがあります。ただし実装がメインの本ではないので python が読める人向けの挿絵のようなものであり、読み飛ばしてしまっても問題ありません。実装の全体は `https://github.com/jntlnlnd/C105-llm-peft` にアップロードしていますが、こちらも解説というほどの整理はしてないため、あくまでも参考程度となります。もし手元で実行したい場合はあらかじめ以下のライブラリをインストールしてください。バージョンについては多少違っても動くと思われます。

```
pip install \
  torch==2.5.1 \
  transformers==4.46.3 \
  datasets==3.1.0 \
  evaluate==0.4.3 \
  scikit-learn==1.5.2 \
  peft==0.13.2
```

それでは前置きはこれくらいにして、さっそく PEFT を試してみましょう！

2 導入

2.1 fine-tuning するタスクについて

本書で題材に選んだのは BLEACH と COMIC LO のポエム分類タスクです。BLEACH はコミックスの巻頭に、LO は表紙にそれぞれポエムが書いてあるのですが、これを分類していくというものです。例えば以下の 2 つのポエムについて、どちらの作品かわかるでしょうか？

- お前には一生、勝てない気がする。
- 伏して生きるな、立ちて死すべし

どちらもタイトルしか知らない人には多少難しいでしょうか。もちろんわかりやすいポエムもあるため、両者に詳しくない人でも 8〜9 割くらいは正解できる難易度感です。

なおこのタスクは既刊の『BLEACH・LO ポエム分類でたどる言語処理技術の発展』でも取り扱っています。そちらも合わせてご覧ください。

集めたポエムは学習用の train データ・epoch 毎の評価を行うための valid データ[*2]・最終的な評価の test データに分けました。各区分のデータサイズは以下の通りです。

	BLEACH	LO
train	42	120
valid	16	38
test	16	38

具体的な実装についても少し触れます。ポエムのデータは tsv として保

[*2] 本書では簡便のため EarlyStopping やハイパーパラメータたんさくなどは行っていません

存され、“huggingface/datasets” ライブラリで読み込みます。

tsv は以下のようなものとなります。

```
$ head -5 train.tsv
poem label
誇りを一つ捨てるたび我等は獣に一歩近付く心を一つ殺すたび我等は獣から一歩遠退く 0
初恋は､歳上でした。1
一緒に数えてくれるかい君についた僕の歯型を 0
まっ白いお米は､どろんこからできます。1
"てごわい敵とバトルだ！""マスク少女萌え""とか言ってる場合じゃない!?" 1
```

label は 0 だと BLEACH、1 だと COMIC LO のポエムであることを示しています。

これを “huggingface/datasets” というライブラリを使って読み込みました。

```
from datasets import load_dataset
ds = load_dataset(
  "csv",
  data_files={
      "train": "../data/train.tsv",
      "valid": "../data/valid.tsv",
      "test": "../data/test.tsv",
  },
  delimiter="\t",
).rename_column("label", "labels")
```

非常に簡単に読み込めますね。補足しておくと、label を labels に rename しているのは、この後学習に利用する transformers というライブラリのお作法だと思ってください。

本データについて 1 点注意があります。このデータセットは全体でも

250 行と非常に小さく、本書で紹介する PEFT の各手法を精度比較することはとてもできません。それぞれの手法について学習結果として精度も掲載していますが、あくまできちんと学習が行われたという動作確認以上の意味は持たないことを念頭に置いてください。

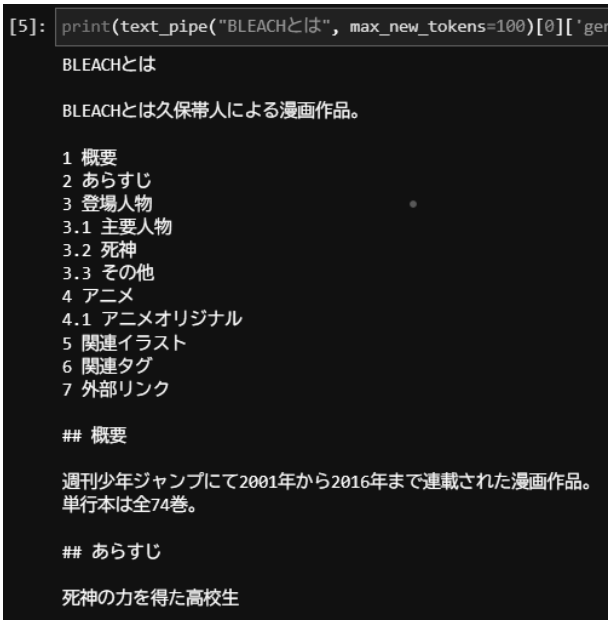
2.2 公開 LLM の用意

本書では fine-tuning のベースモデルとして “llm-jp/llm-jp-3-1.8b”[*3] を用いますというモデル。これは国立情報学研究所が開発した日本語特化の LLM で、パラメータ数は 1.8B（18 億）以上に及びます。3.7B や 13B などより大きなサイズのモデルもありましたが、これらは私の GPU のメモリにはとても乗らないので 1.8B を選定しました。

1.8B と小さめとはいえ、LLM として決して見劣りしない性能は備えています。試しにテキストを与えて続きを生成させてみましょう。

```
1 from transformers import pipeline
2
3 text_pipe = pipeline('text-generation', model="llm-jp/llm-
      jp-3-1.8b", device="cuda")
4
5 print(text_pipe("BLEACH とは", max_length=100)[0]['
      generated_text'])
```

[*3] https://huggingface.co/llm-jp/llm-jp-3-1.8b



文法的にも違和感のない、いい感じの生成結果を得ることができました。目次が挿入されているのは wikipedia か、外部リンクという項目を踏まえると pixiv 大百科あたりの影響を受けている可能性が高いです。事前学習のデータには Common Crawl といった Web ページのテキストデータが使われているので[*4]、この中に pixiv 大百科があってそのテキストを学習したのかもしれません。

Web ページのテキストから BLEACH のことを学習済みなら、ポエムも学習済みなのでしょうか？　もし学習済みだとすると、fine-tuning に成功した時の結果に対してポエムの微妙な表現の違いを学習しているのか、事前学習時に記憶したポエムを思い出しているだけなのかというように考察が変わってきます。

[*4] https://llmc.nii.ac.jp/topics/llm-jp-172b/

試しにテキストを与えてみましょう。

```
print(text_pipe("""以下は
    BLEACH の巻頭のキャッチフレーズです。続きを埋めてください：
伏して生きるな、""", max_length=30)[0]['generated_text'])
```

```
[8]: print(text_pipe("""以下はBLEACHの巻頭のキャッチフレーズです。続きを埋めてください:
     伏して生きるな、""", max_new_tokens=10)[0]['generated_text'])

     以下はBLEACHの巻頭のキャッチフレーズです。続きを埋めてください:
     伏して生きるな、戦え。

     ### 砕蜂
```

だいぶ直球なポエムになりましたね。これをもって学習データにポエムのテキストが含まれていないかという判断は難しいですが、少なくともポエムをはっきり記憶しているわけではなさそうです。

ここまで見てきた通り、日本語能力はそこそこあるがポエム自体の丸暗記まではしていない本モデルがきちんと fine-tuning でポエムの表現の傾向を学習できるか、という点が次章以降のポイントとなります。

2.3 試しに全パラメータの fine-tuning をしてみる

PEFT 手法を試す前に、本モデルを一般的な手法で fine-tuning することが難しいことを確認しましょう。まずモデルを読み込んだ段階での GPU の使用率を “nvidia-smi” コマンドで確認した結果が次の通りです。

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 565.72                 Driver Version: 566.14         CUDA Version: 12.7     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                  Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 2070        On  |   00000000:01:00.0  On |                  N/A |
| 30%   36C    P8             18W /  175W |    7826MiB /   8192MiB |     26%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

GPU には NVIDIA の Geforce RTX 2070 を使っています。メモリサイズは 8GB で庶民的な価格の GPU では一般的ですが、モデルを載せる

だけで GPU メモリの大半を使い切っています。32bit float でパラメータを扱っているのでざっくり 1 パラメータあたり 4Byte、1.8B パラメータのモデルなので単純計算で 7.2GB ほど消費する計算です。本 GPU は画面描画にも用いているためそちらにもメモリを充てていることを踏まえると、8GB のほとんどを使い切っているのは想定通りといったところです。

推論だけなら何とか動きますが、学習しようとするとこれでは足りません。実際に fine-tuning のコードを実装してみましょう。

まずはモデルを読み込みます。これは “huggingface/transformers” のライブラリを使えば簡単です。

```
from transformers import (
    AutoTokenizer,
    AutoModelForSequenceClassification,
)

tokenizer = AutoTokenizer.from_pretrained(
  "llm-jp/llm-jp-3-1.8b"
)
model = AutoModelForSequenceClassification.from_pretrained(
  "llm-jp/llm-jp-3-1.8b"
)
# 後のTrainerで以下を実行しないと失敗するため入れている設定
model.config.pad_token_id = tokenizer.pad_token_id
```

続いて前処理用のクラスを定義します。LLM をはじめとする機械学習モデルは数値計算からなるため、テキストデータはそのままでは扱うことができません。そこでテキストを単語に似たトークンという単位に分割し、それぞれ対応する ID として数値列にする**トークナイズ**という処理を行ってからモデルに入力します。それを行うのが tokenizer で、それを動かすためのクラスになります。

```
# テキストのtokenizeを行うための前処理クラス
```

```
class TokenizeCollator:
    def __init__(self, tokenizer):
        self.tokenizer = tokenizer

    def __call__(self, examples):
        encoding = self.tokenizer(
            [ex["poem"] for ex in examples],
            padding="longest",
            truncation=True,
            max_length=200,
            return_tensors="pt",
        )
        return {
            "input_ids": encoding["input_ids"],
            "attention_mask": encoding["attention_mask"],
            "labels": torch.tensor([
              ex["labels"] for ex in examples
            ]),
        }
```

次は metrics を計算する関数です。これは学習途中での valid データに対する精度や、学習完了後の test データに対する精度を出す関数となっています。“huggingface/evaluate” というライブラリを使って実装しています。

```
import evaluate

roc_auc_evaluate = evaluate.load("roc_auc")
acc_evaluate = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    logits, labels = map(torch.tensor, eval_pred)
    probs = torch.nn.functional.softmax(logits, dim=1)[:,
        1] # label=1の確率
```

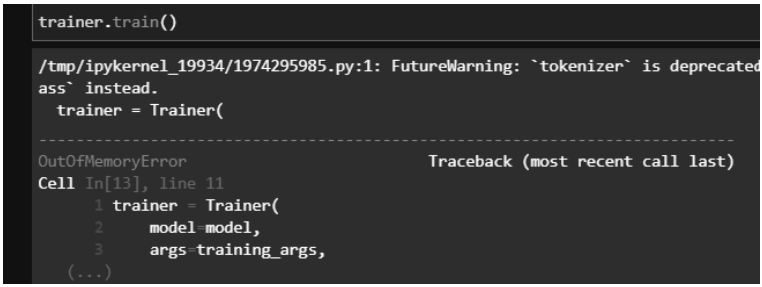
```
    pred_labels = torch.argmax(logits, dim=1) # 予測ラベル
    return {
      **roc_auc_evaluate.compute(prediction_scores=probs,
          references=labels),
      **acc_evaluate.compute(predictions=pred_labels,
          references=labels),
    }
```

必要なものを一通り定義したので学習を実行します。transformers の Trainer を使っています。

```
from transformers import (
    TrainingArguments,
    Trainer,
)

training_args = TrainingArguments(
    output_dir=f"../results/",
    num_train_epochs=10,
    learning_rate=1e-4,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
    weight_decay=1.0,
    evaluation_strategy="epoch",
    logging_strategy="epoch",
    remove_unused_columns=False,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=ds["train"],
    eval_dataset=ds["valid"],
```

```
    tokenizer=tokenizer,
    data_collator=TokenizeCollator(tokenizer),
    compute_metrics=compute_metrics,
)

trainer.train()
```

学習を始めてすぐに、案の定 GPU のメモリがあふれてエラーになってしまいました。



transformers は非常に便利なライブラリで、学習の実装についてはかなりシンプルにできます。ですが計算リソースだけはどうしようもありません。それを解決するため、次章から PEFT 手法を一つ一つ見ていきます。

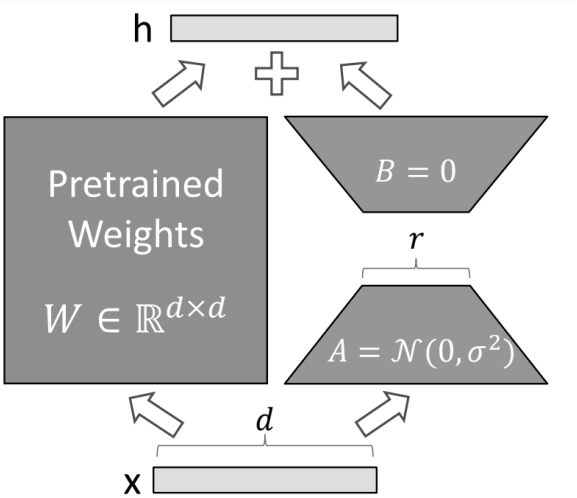
3　低ランク近似により学習するパラメータを減らす

3.1　LoRA: 低ランクの差分テンソルを学習

PEFT の代表格が Low-Rank Adaptation (LoRA) です[*5]。LoRA というと画像生成 AI で良くも悪くも取りざたされがちですが、画像に限らずニューラルネットワークの fine-tuning 一般に応用可能な技術です。というのも LoRA はテンソルパラメータの fine-tuning を省メモリで行う技術なのですが、ニューラルネットワークは基本的にテンソル演算の繰り返しなのでニューラルネットワークなら LoRA 適用可能と言って差し支えないためです。

もう少し詳しく書きます。一般の fine-tuning では事前学習済みモデルのテンソルのパラメータ自体を更新していきます。それに対し LoRA では元のモデルのパラメータは学習中も固定してしまい、そこからの差分となるテンソルを別途用意してそちらだけを学習します。これだけだとテンソルが増える分無駄に GPU メモリを消費するだけなのですが、追加の工夫として低ランク近似を行います。例えば元のモデルが 1024×1024 のところ、1024×8 のテンソルと 8×1024 のテンソルの積に近似するといった具合です。この場合パラメータ数は $1024^2 = 1048576$ 個から $1024 \times 8 \times 2 = 16384$ と 1/64 まで小さくすることができます。学習時に勾配としてメモリが必要になるのは学習対象のパラメータ分のみなので、1.8B モデルならもともと勾配に 7.2GB 必要とするところ、LoRA により約 110MB まで圧縮できます。モーメントに必要なメモリも同様です。

[*5] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” arXiv preprint arXiv:2106.09685, 2021.



PEFT の他手法と比べた LoRA の利点としては、学習後は差分テンソルを元のテンソルと足し合わせて一つのモデルにマージすることで、推論時の計算量を元のモデルと同じにすることができるという点です。後述する他の手法ではモデルの中に追加の計算処理を入れて、その処理におけるパラメータを学習するパターンが多いです。そのケースでは推論時にも追加の計算処理を実行する必要があるため、どうしても元のモデルより計算量は増えてしまいます。それを回避できる点で LoRA は魅力的です。

また LoRA の低ランク近似は、元のモデルの知識が fine-tuning によって失われる**破滅的忘却**という現象を回避する良い正則化になっているという議論もあります[*6]。これが事実であれば、省メモリなだけでなく LLM の知識を引き出し精度を出す fine-tuning 手法としても理にかなっているということになります。

[*6] D. Biderman, J. Portes, J. J. G. Ortiz, M. Paul, P. Greengard, C. Jennings, D. King, S. Havens, V. Chiley, J. Frankle, C. Blakeney, and J. P. Cunningham, “LoRA Learns Less and Forgets Less,” arXiv preprint. arXiv:2405.09673 [cs.LG] (2024).

説明はこれくらいにして、実際に実行してみましょう！　PEFT 手法は “huggingface/peft” というライブラリにまとめられていて、LoRA を含む様々な手法を統一された書き方で簡単に実装することができます。

```
from peft import LoraConfig, TaskType
from peft.peft_model import
    PeftModelForSequenceClassification

# PeftModelForSequenceClassification にバグがあるため
# 関数の差し替えをしている
class PatchedPeftModelForSequenceClassification(
  PeftModelForSequenceClassification):
    def add_adapter(
      self,
      adapter_name,
      peft_config,
      low_cpu_mem_usage=False):
        super().add_adapter(adapter_name, peft_config)

peft_config = LoraConfig(
  task_type=TaskType.SEQ_CLS,
  r=1,
)

peft_model = PatchedPeftModelForSequenceClassification(
  model, peft_config)
```

たったこれだけの実装で元のモデルの重みを固定し、LoRA の低ランク差分テンソルと連携させることができます。実際に学習するパラメータがどれくらい減ったか見てみましょう。

```
peft_model.print_trainable_parameters()
# 出力: trainable params: 200,704 || all params:
```

```
1,663,870,976 || trainable%: 0.0121
```

全パラメータが 16.6 億[*7]に対して、学習するパラメータは 20 万と、実に 0.01 %程度まで減らせています。今回の設定では低ランク近似のランクを 1 にまで圧縮しているので、学習対象のパラメータも 4 桁も下げることができているのです。これにより、学習中に必要な勾配などのために保持しておくメモリ容量は誤差程度となるため、最低限の GPU メモリ消費で学習を動かせます。

それでは実際に学習させてみましょう。学習スクリプトは 2 章 3 節のスクリプトの Trainer に渡す model という変数を peft_model に置き換えるだけです。

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	1.541700	0.446591	0.980263	0.944444
2	0.169300	0.767902	0.980263	0.944444
3	0.050000	0.548937	0.986842	0.944444
4	0.000100	0.699652	0.983553	0.944444
5	0.000000	0.677101	0.981908	0.944444
6	0.000000	0.666380	0.981908	0.944444
7	0.000000	0.658808	0.981086	0.944444
8	0.000000	0.653420	0.981086	0.944444
9	0.000000	0.650459	0.981908	0.944444
10	0.000000	0.649708	0.981908	0.944444

まず、メモリのエラーを起こさずにきちんと学習が進行しました。これだけでも PEFT 手法を使う意義があります。また学習に伴う valid データへの精度を見ると 1 epoch で AUC 0.98 を超え、Accuracy は 0.944 に収束しています。また test データへの Accuracy は 0.963 でこちらも良好

[*7] パラメータ数が 1.8B よりも小さいのは Classification モデルとして読み込むことにより不要なテンソルが除外されていることなどが理由だと思いますが、詳細は調べ切れていません……。

です。

2 章 1 節で触れた通り、今回使うデータセットは PEFT 手法ごとの性能を比較するにはサイズが小さすぎるため、指標の値から LoRA の精度面における優位性について論じることはできません。しかし少なくとも意味のある学習ができているとは言ってよいでしょう。

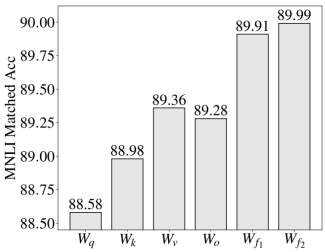
3.2　AdaLoRA: LoRA のランクをいい感じに決める

さて、LoRA で効率よく LLM の fine-tuning ができることが確認できたのは良いのですが、賢い人は次のように考えました。「LLM の全てのテンソルで同じランクの近似を行っているが、テンソルごとにランクを変えた方が良いのではないか」と。

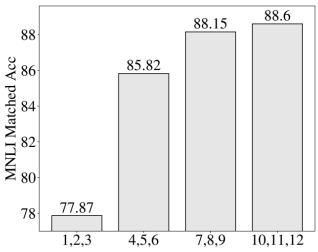
LLM は Transformer ブロックと呼ばれる Self Attention と全結合を組み合わせたブロックを何層も重ねています。その層は深さに応じて役割が異なると言われており、それは浅いブロックでは単語同士の関係や短いフレーズなどの特徴を抽出する一方、深いほど広い文脈や抽象的な概念の情報を処理する、といった具合です。実際にこれを問題提起した AdaLoRA という手法の提案論文[*8]では、LLM の層別に LoRA を適用すると層が深いテンソルへの LoRA モデルの方が fine-tuning 後の精度が上がることを確認しています。

[*8] Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen, and T. Zhao, “Adaptive budget allocation for parameter-efficient fine-tuning,” arXiv preprint arXiv:2303.10512, 2023.

.



(a) Selected weight matrix



(b) Selected layers

であればレイヤーごとに異なるランクのテンソルを学習しよう！ ……となるのですが、たくさんあるレイヤーそれぞれランクを調整すると組み合わせ爆発して探索が大変になってしまいます。そこで AdaLoRA では、学習中に層ごとの重要度を計測し、その重要度に応じてランクを割り当てるというテクニックが開発・導入されました。テクニックの詳細は原著をご参照いただければと思いますが、ざっくり説明すると、

1. 低ランク近似を特異値分解の形に変換したうえで
2. 学習が進むごとに、中央の対角テンソルの各値およびそれに紐づく左右のテンソルの要素に対する損失関数の勾配から重要度を計算
3. それが小さいものは精度への影響が小さい要素ということで対角テンソルの要素を 0 にしてしまう

という流れにより、最初は全体的に高めにランクを割り当てたところから間引いていくことで最終的に重要度に合わせたランクに落とし込んでいきます。

と、長々書きましたが、このような細かいロジックを知らなくてもとりあえず動かすことはできます。先ほどの LoRAConfig を AdaLoRAConfig に書き換えるだけです。

```
from peft import AdaLoraConfig, TaskType
peft_config = AdaLoraConfig(
```

```
    task_type=TaskType.SEQ_CLS,
    init_r=2,
    target_r=0.5,
)
```

最初は 1 テンソルあたりランク 2 からスタートして、最終的には 1 テンソルあたりランク 0.5 まで間引くような設定を書きました。実際には対角テンソルの非ゼロの要素が 0 個のテンソル、1 個のテンソル、2 個のテンソルが重要度に応じて決定されるという挙動を期待しています。

学習した結果は次の通りです。

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	2.363700	0.832683	0.960526	0.833333
2	0.739400	0.371368	0.996711	0.907407
3	0.440800	0.254095	0.998355	0.925926
4	0.321500	0.302767	0.998355	0.925926
5	0.281900	0.259691	0.998355	0.925926
6	0.239800	0.262338	0.998355	0.925926
7	0.224800	0.211605	0.998355	0.925926
8	0.192500	0.179823	0.998355	0.925926
9	0.174700	0.169674	0.998355	0.925926
10	0.169900	0.173275	0.998355	0.925926

LoRA に比べると少し収束までステップを要していますが、最終的には Accuracy 0.926 に落ち着きました。test データへの Accuracy は 0.944 となり、良い精度が出ています。

LoRA の派生手法は AdaLoRA 以外にも数多く存在します。一部はより高精度を求め、一部はより高速・省メモリになるように発展させ、“huggingface/peft” ライブラリで利用できるものも数多くあります。このように派生が数多くあることが LoRA という手法の優秀さを物語っているといえるでしょう。

4 入力プロンプトを最適化する

4.1 Prompt tuning: 仮想プロンプトを学習

LLM と言えばプロンプトエンジニアリング、すなわち LLM の出力を見ながらプロンプトを調整することで目的のタスクを解かせるような使い方が定番です。この枠組みを fine-tuning に応用したのが仮想プロンプト系の手法で、Prompt tuning もその一つです[*9]。

Prompt tuning の趣旨はプロンプトエンジニアリングと比較するとわかりやすいです。プロンプトエンジニアリングでは、人が様々にプロンプトを入力します。例えば以下のようなものです。

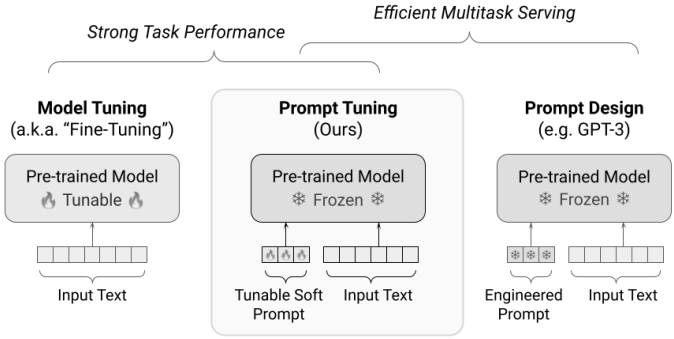
```
1 次に挙げるポエムがBLEACH のものか COMIC
      LO のものか分類してください：{ポエム}
```

この入力に対して LLM はまず『次に挙げるポエムが BLEACH のものか COMIC LO のものか分類してください:』というプロンプトをまずトークン分割したうえで、Embedding 層を通してトークンごとに埋め込みベクトルに変換し横につなげます。結果として、もしプロンプトが 20 個のトークンに分割された場合、出力は (20, ベクトル次元) の shape を持つテンソルになります。ポエムも同様にトークン分割・埋め込みベクトルに変換し、つなげてテンソルとします。得られたテンソル同士も横につなげて一つのテンソルとしたのち、多層 Transformer でトークン間の関係を解釈していきます。

Prompt tuning では実際のプロンプトのテンソルの代わりに、例えば (20, ベクトル次元) の学習可能なテンソルを用意し、ポエムに対応するテ

[*9] B. Lester, R. Al-Rfou, and N. Constant, “The power of scale for parameter-efficient prompt tuning,” arXiv preprint arXiv:2104.08691, 2021

ンソルの横につなげます。ここで 20 という数字はハイパーパラメータです。学習時にはこの仮想プロンプトの埋め込みテンソルに対して勾配逆伝播を行いパラメータを調整していきます。



*10

通常のプロンプトエンジニアリングがトークンを離散的に選びながら調整する一方、Prompt tuning では連続的に値をとれるパラメータを勾配法で調整することから、前者をハードプロンプト・後者をソフトプロンプトと呼ぶことがあります。

Prompt tuning の趣旨を説明したところで実際に動かしてみましょう。前章の低ランク近似とは全く異なる手法ですが、"huggingface/peft" ライブラリでは全く同じ書き方で OK です。

```
from peft import PromptTuningConfig, TaskType

peft_config = PromptTuningConfig(
    task_type=TaskType.SEQ_CLS,
    num_virtual_tokens=10,
)
```

*10 図の出典: https://research.google/blog/guiding-frozen-language-models-with-learned-soft-prompts/

この設定で学習した結果が以下の通りです。

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	1.188200	0.617633	0.899671	0.925926
2	0.728800	0.637780	0.931743	0.925926
3	0.650000	0.667039	0.935033	0.925926
4	0.477200	0.578857	0.942434	0.925926
5	0.331600	0.523367	0.945724	0.925926
6	0.278300	0.506315	0.952303	0.925926
7	0.266800	0.436548	0.952303	0.925926
8	0.204600	0.524743	0.955592	0.925926
9	0.207300	0.444462	0.955592	0.925926
10	0.182600	0.457476	0.955592	0.925926

前章の LoRA・AdaLoRA とは異なり Training Loss からして少し不安定です。一応 valid データの Accuracy はそこそこ出ているものの、test データへの Accuracy は 0.815 と比較的小さめに出てしまいました。

LoRA と異なり Prompt tuning は入力という勾配逆伝播の終点部分のみを学習する[*11]ため、勾配を伝える間にノイズが乗りやすく学習が難しいのかもしれません。とはいえデータが少ないですから、手法の優劣についてここで論じるのは早計であることは改めて書いておきます。

4.2 P-Tuning: 仮想プロンプトの埋め込みにニューラルネットワークを挟む

ソフトプロンプトを勾配法で学習するのが素朴には難しいということは知られており、仮想プロンプトの埋め込みを直接ポエムのテンソルと結合するのではなく、ニューラルネットワークを通してから結合する

[*11] 一応クラスごとの logits を計算する最後の全結合層も学習しますが。

P-Tuning という手法も提唱されています[*12]。具体的には以下の通りで、トークン埋め込みにあたるテンソル $h_{0:m}$ を i 番目で分割し、ニューラルネットワークの学習可能なレイヤーの 1 つである LSTM と Multi Layer Perceptron(MLP) に通しています。

$$h_i = \mathrm{MLP}\left([\overrightarrow{h}_i : \overleftarrow{h}_i]\right) = \mathrm{MLP}\left([\mathrm{LSTM}(h_{0:i}) : \mathrm{LSTM}(h_{i:m})]\right)$$

これにより学習結果にどのような影響があるのでしょうか？　設定は以下の通りです。

```
from peft import PromptEncoderConfig, TaskType

peft_config = PromptEncoderConfig(
    task_type=TaskType.SEQ_CLS,
    num_virtual_tokens=10,
)
```

そして学習した結果が次の通りです。

[*12] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, “GPT understands, Too,” arXiv preprint arXiv:2103.10385, 2021.

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	2.614700	0.785152	0.914474	0.777778
2	1.283100	1.140250	0.960526	0.777778
3	1.083600	0.856120	0.985197	0.740741
4	1.246300	1.332329	0.986842	0.740741
5	1.005300	0.236901	0.986842	0.907407
6	0.678600	0.492552	0.990132	0.944444
7	0.555200	0.167653	0.993421	0.962963
8	0.680300	0.314980	0.990132	0.944444
9	0.461400	0.268396	0.993421	0.944444
10	0.335200	0.223721	0.993421	0.944444

valid AUC などは少し Prompt tuning よりも良くなっています。test データへの Accuracy は 0.852 とこちらも少し改善しました。が、データ数を踏まえると誤差の範囲とも言えるかもしれません。

なおあたかも P-Tuning を Prompt tuning の発展手法のように書いていますが、実際には P-Tuning の提案論文は Prompt tuning よりも 1 カ月早く世に出ています。明確な優劣もついておらずタスク・データセットごとに相性があるという報告もあり[*13]、どちらがより良い手法かについては実験的に探索するのが望ましいでしょう。

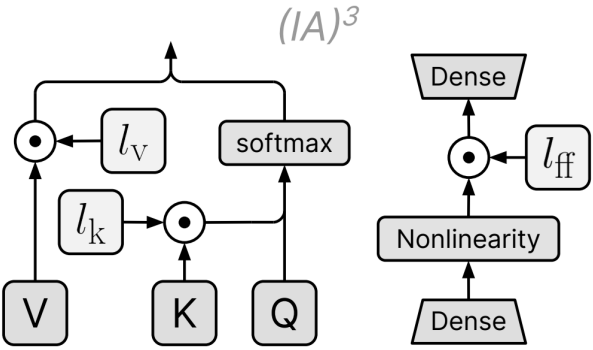
[*13] X. Liu, K. Ji, Y. Fu, W. L. Tam, Z. Du, Z. Yang, and J. Tang, "P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks," arXiv preprint arXiv:2110.07602, 2021.

5　内部に追加のアダプターを導入する

5.1　$(IA)^3$: 各層の出力をベクトルで補正する

低ランクの差分テンソル・仮想プロンプトに続く第 3 の PEFT 手法が内部的な出力を変換するアダプターを挟むという手法です。その中でも $(IA)^3$ は LoRA よりさらに fine-tuning するパラメータ数が小さい手法として知られています[*14]。

$(IA)^3$ では Transformer ブロックにおけるテンソルの一部（一般的には Self Attention の Key, Value とそれに続く Feed Forward Network(FFN) の 1 層目）の出力に対して学習可能なベクトルを定義し、各トークンに対応するベクトルに対して要素ごとに積を取ることでその出力を補正します。



LoRA の場合はざっくり（トークンごとの埋め込み次元 × 低ランク近似のランク数 × 2 × 近似するテンソルの数 × Transformer ブロックの

[*14] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel, “Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning,” Advances in Neural Information Processing Systems, vol. 35, pp. 1950–1965, 2022.

数）分のパラメータを学習することになりますが[*15]、$(IA)^3$ だと（トークンごとの埋め込み次元 × 学習可能なベクトルで補正するテンソルの数 × Transformer ブロックの数）分のパラメータを学習するだけで済みます。ただ LoRA でもかなり学習するパラメータ数を減らせることから GPU メモリの節約効果は絶対値的には大差なく、学習速度もほとんど変わらないことが多いです。

なお $(IA)^3$ は LoRA と同様に学習後にはベクトルと各テンソルをかけ合わせてマージすることで、推論は元のモデルと同じ計算量で済みます。LoRA のランクのようなハイパーパラメータもないため、$(IA)^3$ で十分に精度が出る場合にはこちらを選ぶのも良いでしょう。

それでは動かしてみましょう。IA3 はハイパーパラメータもほとんどなく設定もかなりシンプルです。

```
1 from peft import IA3Config, TaskType
2
3 peft_config = IA3Config(task_type=TaskType.SEQ_CLS)
```

[*15] 厳密には FFN の 1 層目の埋め込み次元のみ異なる点を考慮する必要があります。

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	0.822000	0.471093	0.986842	0.870370
2	0.380800	0.148605	0.998355	0.925926
3	0.209400	0.229504	0.998355	0.907407
4	0.167800	0.175166	1.000000	0.925926
5	0.119700	0.229367	1.000000	0.925926
6	0.112200	0.233600	1.000000	0.925926
7	0.106500	0.210052	1.000000	0.925926
8	0.088100	0.194159	1.000000	0.925926
9	0.082500	0.195634	1.000000	0.925926
10	0.078700	0.199589	1.000000	0.925926

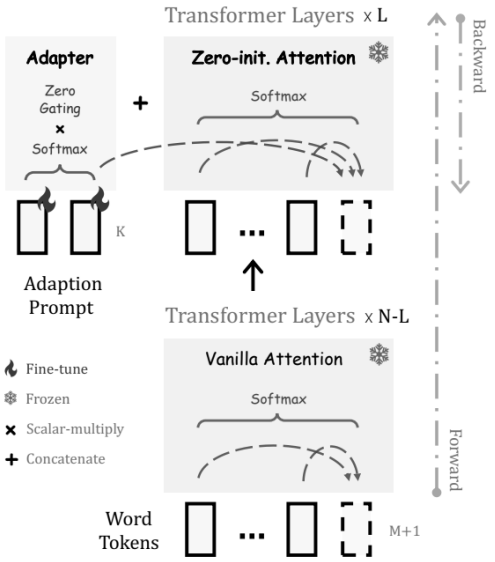
学習は比較的速く進み、valid データへの精度も安定しています。test データの Accuracy は 0.926 とこちらも良好でした。

5.2　LLaMa-Adapter: Attention の Key と Value を外部からも与える

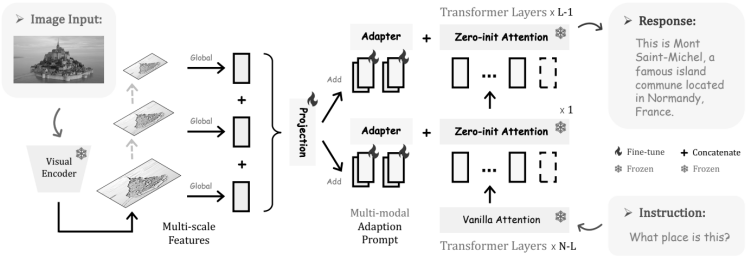
Adapter を入れる手法としてもう一つ、LLaMa-Adapter を紹介しておきましょう[*16]。

[*16] R. Zhang, J. Han, A. Zhou, X. Hu, S. Yan, P. Lu, H. Li, P. Gao, and Y. Qiao, “Llama-adapter: Efficient fine-tuning of language models with zero-init attention,” arXiv preprint arXiv:2303.16199, 2023.

LLaMa-Adapter は Transformer ブロックの Self Attention の前方に、仮想的な埋め込みにあたるテンソルから計算した Attention の Key と Value をつなげる手法です。学習可能なテンソルを前方につなげるという意味では仮想プロンプトを学習する手法に近いですが、Transformer ブロックのうち出力層に近い数ブロックを対象にする点が違いとなります。主なハイパーパラメータとしては仮想的なプロンプトのトークン数と出力から何ブロックまでさかのぼって適用するかの 2 つがあります。



LLaMa-Adapter の特徴としては、仮想的なプロンプトの代わりに画像などを埋め込んで渡しても良いという点です。これにより、画像を入力しつつ「これに写っている標識はどういう意味ですか？」と自然文で尋ねるなどのマルチモーダルなタスクに答えることができます。学習時には画像をテキストと同じ意味空間に落とし込むような学習をすることで、LLM のパラメータは固定したまま画像入力を受け取ることができるようになります[*17]。



LLaMa-Adapter はその名前のとおり LLaMa と呼ばれる Meta 社が公開する LLM のネットワーク構造に依存した手法で、他のネットワーク構造の LLM に適用するのは難しい場合があります。しかし本書で利用する llm-jp/llm-jp-3-1.8b モデルは LLaMa と同じ構造を採用しており、LLaMa-Adapter も利用可能です。さっそく試してみましょう。

```
from peft import AdaptionPromptConfig, TaskType

peft_config = AdaptionPromptConfig(
    task_type=TaskType.SEQ_CLS,
    adapter_layers=8,
    adapter_len=10,
```

[*17] 察しのいい人は気づいたかもしれませんが、同じ理屈で Prompt Tuning や P-Tuning もマルチモーダル化することは可能で、そういった研究もあります。

7)

Epoch	Training Loss	Validation Loss	Roc Auc	Accuracy
1	0.998600	0.403320	0.970395	0.925926
2	0.523000	0.297817	0.975329	0.944444
3	0.426800	0.258351	0.976974	0.962963
4	0.336500	0.323799	0.984375	0.962963
5	0.283300	0.295028	0.985197	0.962963
6	0.233800	0.300153	0.986842	0.962963
7	0.191900	0.243756	0.985197	0.962963
8	0.147500	0.257128	0.985197	0.962963
9	0.130600	0.249951	0.985197	0.962963
10	0.117500	0.248799	0.985197	0.962963

学習は非常に安定して進みました。仮想プロンプト系の手法と異なり出力層に近い部分の勾配のみが考慮されることが安定に寄与している……かもしれません。test データへの Accuracy は 0.963 でこちらも良好でした。

なお LLaMa-Adapter については $(IA)^3$ などとは異なりレイヤーをマージして推論時には元のモデルと同じ推論時間というわけにはいきません。ただし計算量が増えるのは Attention の部分だけですので、仮想トークン数が同じなら仮想プロンプト系の手法よりは計算量は抑えられると考えられます。

6　おわりに

以上が代表的な PEFT 手法になります。いずれも興味深い工夫で数少ないパラメータからモデルの性能を引き出していましたね。

今や様々な企業・機関が LLM を公開しており、その性能も日進月歩です。せっかく自由にパラメータをいじれる公開 LLM をプロンプトエンジニアリングだけで扱うのはもったいないことで、目的に合わせたラベル付きデータセットで fine-tuning することでその可能性を引き出すことができます。学習するハードウェアの確保が難しいとなった場合には、ぜひ本書の内容を思い出してみてください。

誰でもプロンプトで AI を活用できる AI 民主化の時代になっても、精度を突き詰めるシーンで機械学習エンジニアの役割がなくなることはありません。AI に使われず、AI を使いこなす気概で次の時代を切り拓いていきましょう。

実践 PEFT 〜ご家庭の GPU で LLM fine-tuning〜

著者　お椀の底の玉
発行日　2024/12/30 (初版)
発行元　ゆるふわ数理研究所
X ID　@yurufuwasuuri
印刷所　ちょ古っ都製本工房