

言語モデルの仕組みと簡易チャットシステム

スクリプト言語演習

伊澤侑祐

言語モデルを 4 段階に分けて理解する

段階	問い	主な道具
1. 表現する	文字列をどうモデルへ渡すか	トークン、ID
2. 予測する	モデルは何を返すか	logits、softmax
3. 選んでつなぐ	どう文章を伸ばすか	greedy、温度、top-k
4. 会話にする	なぜ対話が続いて見えるか	テンプレート、履歴

モデルの内部をすべて理解する前に、入出力を確かめる。

言語モデルとは

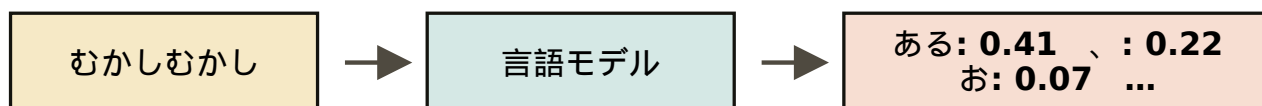
導入

第 2 回の「次の語を引く機械」を「会話」に発展させる

全体像

	マルコフ連鎖	言語モデル
次の候補	<code>table[word]</code>	<code>model(ids)</code>
根拠	数えた回数	学習済みの重み
見る範囲	直前 1~2 語	直前の多くのトークン
選び方	重み付き抽選	greedy、温度、top-k

予測器は大きくなるが「予測 → 選択 → 連結」のループは変わらない



PyTorch と Transformers でモデルを準備する

実行準備

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
```

```
name = "Qwen/Qwen2.5-0.5B-Instruct"
tok = AutoTokenizer.from_pretrained(name)
model = AutoModelForCausalLM.from_pretrained(name)
model.eval()
```

- tok は文字列とトークン ID を相互に変換する
- model は ID 列から logits を計算する
- eval() はモデルを推論用の動作へ切り替える

以降はこの tok と model を準備済みとして進める。

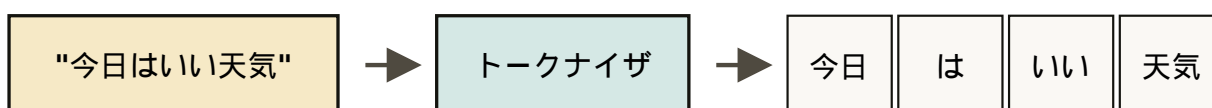
言語モデルの入力: 文字列 (トークン ID の列)

1. 表現する

- トークンは単語とも 1 文字とも限らない
- トークナイザが文字列を分割し、各部分を整数 ID へ変える
- モデルが受け取るのは文字列ではなく ID の列

```
text = " 言語モデルを動かす"
ids = tok(text).input_ids
pieces = tok.convert_ids_to_tokens(ids)

print(ids)
print(pieces)
```

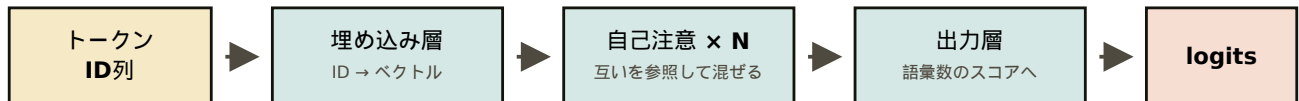


ID はモデルの中で変換されたベクトル

1. 表現する

文字列 → トークン ID → ベクトル → Transformer → logits

- ベクトルは、各トークンを数値の並びで表したもの
- 注意機構は、前にあるトークンの情報を混ぜる
- 最後に、語彙中の各トークンへ 1 つずつスコアを出す



内部の数式より先に、入力と出力の形を押さえる

言語モデルの出力: 次の候補すべてのスコア

2. 予測する

logits は確率になる前の値で、語彙中の各候補に対応する。

```
ids = tok(" むかしむかし", return_tensors="pt").input_ids
```

```
with torch.no_grad():  
    output = model(ids)
```

```
print(output.logits.shape)  
# [バッチ数, 入力の長さ, 語彙数]
```

次のトークンの予測には、バッチ 0 の末尾を使う。[0, -1] はこの位置を指定している。

```
next_logits = output.logits[0, -1]
```

添字と dim: テンソルのどこを操作するか表現

2. 予測する—PyTorch の記法

```
logits = output.logits          # shape: [1, 系列長, 語彙数]  
next_logits = logits[0, -1]    # shape: [語彙数]  
probs = torch.softmax(next_logits, dim=-1)
```

記法	意味
[0, -1]	バッチ 0、系列の末尾を選ぶ

記法	意味
<code>dim=-1</code>	最後の軸、ここでは語彙軸を操作する
<code>.shape</code>	各軸の長さを確認する
<code>.item()</code>	1 要素の Tensor を Python の数へ変える

PyTorch では、値だけでなく `shape` と操作する軸を確認する。

softmax を通すと候補を確率として比較可能

2. 予測する

```
probs = torch.softmax(next_logits, dim=-1)
top = torch.topk(probs, 5)
```

```
for token_id, prob in zip(top.indices, top.values):
    print(tok.decode([token_id]), prob.item())
```

- softmax 後の値は 0 以上で、合計が 1 になる
- topk は確率の高い候補だけを取り出す
- まず候補を観察すると、選び方の違いを理解しやすい

モデルは文章ではなく、まず「次の 1 トークンの分布」を返す

greedy: 最大の候補だけを選ぶ方法

3. 選んでつなぐ

```
next_id = torch.argmax(next_logits, dim=-1)
next_id = next_id.view(1, 1)
ids = torch.cat([ids, next_id], dim=1)
```

- `argmax(..., dim=-1)` は語彙軸で最大の ID を選ぶ
- `view(1, 1)` は [バッチ数, 系列長] の形へ合わせる
- `cat(..., dim=1)` は系列長の軸へ新しい ID を連結する

1. 現在の ID 列をモデルへ渡す
2. 最大の logit を持つ ID を選ぶ
3. 選んだ ID を末尾へ追加する
4. 終端 ID が出るまで繰り返す

同じ入力なら毎回同じ結果になるため、動作確認に向いている

greedy は局所的な最善を選ぶため単調になりやすい

3. 選んでつなぐ

入力: 好きな食べ物は

出力: 好きな食べ物は何ですか? 好きな食べ物は何ですか? ...

- 各時点で 1 位の候補しか選ばない
- いったん反復へ入ると、同じ選択が続くやすい
- 文章全体としてよいかは、1 トークン先の最大値だけでは決まらない

そこで、複数の候補を残し、確率に応じて 1 つを選ぶ

温度は候補間の差を、top-k は候補の数を変える

3. 選んでつなぐ

temperature = 0.8

k = 50

```
probs = torch.softmax(next_logits / temperature, dim=-1)
top = torch.topk(probs, k)
index = torch.multinomial(top.values, 1)
next_id = top.indices[index]
```

操作	小さくする	大きくする
温度	上位へ強く偏る	候補間の差が縮む
k	安全な候補に絞る	低確率の候補も残す

第 2 回の温度と役割は同じで、logits では割り算として現れる

top-p は確率分布に合わせて候補数を変える

3. 選んでつなぐ—発展

- 候補を確率の高い順に並べる
- 確率の合計が p に達するまで候補を残す
- 1 つが突出すれば少数、拮抗すれば多数の候補が残る

分布 A: 0.80 0.10 0.05 ... → p=0.9 なら 2 候補

分布 B: 0.25 0.20 0.18 ... → p=0.9 なら多くの候補

```
sorted_probs, sorted_ids = torch.sort(probs, descending=True)
cumulative = torch.cumsum(sorted_probs, dim=-1)
keep = cumulative - sorted_probs < p
```

```
kept_probs = sorted_probs[keep]
kept_ids = sorted_ids[keep]
index = torch.multinomial(kept_probs, 1)
next_id = kept_ids[index]
```

top-k は候補数を固定し、top-p は確率の合計を固定する

生成は「予測・選択・連結」の反復

ここまでの接続を確認する

文字列

↓ tokenizer

トークン ID 列

↓ model

次の候補の logits

↓ greedy または sampling

次のトークン ID

↓ 末尾へ連結して繰り返す

長くなったトークン ID 列

↓ decode

生成された文字列

途中の値を順に確認すれば生成結果だけを見るより原因を探しやすい

会話では役割を示す記号も入力へ含める

4. 会話にする

```
messages = [
    {"role": "system", "content": " 簡潔に答えてください。"},
    {"role": "user", "content": " 日本の首都は？"},
]
```

```
prompt = tok.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True,
)
```

apply_chat_template は役割と発言をモデル用の文字列へ変換する。

モデルは会話を記憶せず履歴を毎回読み直す

4. 会話にする

system: 簡潔に教えてください。

user: 日本の首都は？

assistant: 東京です。

user: 人口は？

assistant:

- 過去の発言と新しい質問をまとめてモデルへ渡す
- 履歴に前の回答があるため、「人口」が東京を指すと解釈できる
- 履歴を削れば、その情報を使えなくなる

「覚えている」ように見える動作は入力に履歴が含まれるために起こる

入力長と出力品質の限界

4. 会話にする

現象	理由	扱い方
会話が次第に遅くなる	履歴が長くなる	古い履歴を削る・要約する
同じ表現を繰り返す	選択が狭い	温度や候補数を調整する
もっともらしく誤る	次の語の予測器である	事実を別の資料で確認する
素朴なループが遅い	毎回すべて再計算する	<code>generate</code> や KV キャッシュを使う

0.7B モデルは仕組みの観察には向くが実応用には向かない

まとめ

段階	押さえること
1. 表現する	文字列はトークン ID 列へ変換される
2. 予測する	モデルは次の候補の logits を返す
3. 選んでつなぐ	選択した ID を連結し、再び予測する
4. 会話にする	役割付きの履歴を毎回入力へ含める

言語モデルは巨大でも外側のプログラムは変換とループで組み立てられる