🤖 GPT-2チャットボット開発のための思考案 - Google Colab版

📋 思考案の構成

- **①～Ⓧ**のテーマ
- **●コードブロック全体**（コード全体をテーマごとに区切った塊）
  - 【コードの小テーマ】
  - 該当部分の抜粋コード
  - 抜粋コードについての説明
- **◆コードブロック全体の解釈**
- **備忘録**

---

① Google Colabのための環境設定

🔧 コードブロック1

```bash
!pip install transformers==4.21.0 torch==1.12.1 datasets==2.4.0 accelerate==0.12.0
!pip install sentencepiece==0.1.97 sacremoses==0.0.53
print("ライブラリのインストールが完了しました。")
```

📦 コード概要

!pip install

transformers==4.21.0 → Hugging FaceのTransformersライブラリ。 GPT-2などの事前学習済みモデルやトークナイザーを使うのに必須。 ※バージョン4.21.0で環境を固定しているのは、コードの互換性を保つため。

torch==1.12.1 → PyTorchのバージョン。深層学習モデルの計算ライブラリ。　GPUを使った高速処理を担う。

datasets==2.4.0 → Hugging FaceのDatasetsライブラリ。データセットの読み込みや加工を簡単に行える。

accelerate==0.12.0 → モデルの分散学習や高速化を簡単に実装するためのユーティリティ。　ColabのGPUを有効活用する際に便利。

---

sentencepiece==0.1.97 → Googleが開発したサブワード分割ツール。　日本語などの言語でトークナイズ（単語分割）するときに使用。

sacremoses==0.0.53 → MosesトークナイザーのPython版。　文章の前処理（トークン分割や正規化）で使われる。　英語のトークナイザーとして多用されているが、日本語モデルでも使うケースあり。

◆ コードブロック1の全体の解釈

- **1行目** ... Transformerモデルや学習処理の基盤ライブラリをインストール
- **2行目** ... 日本語テキストを分割・処理するためのツールを追加でインストール

📝 **【備忘録】**

BPEのはずがなぜトークナイザーに「sentencepiece」や「sacremoses」があるのか

- GPT-2の標準トークナイザーは `GPT2Tokenizer` （BPEベース）で完結
- `sentencepiece` は主に日本語や多言語対応のモデル（T5やmBERTなど）が使う別のトークナイザー
- `sacremoses` は主に英語などの形態素解析やトークナイズ前のテキスト正規化で使われる

🛠️ コードブロック2

```python
```

```
import torch
from transformers import (
    GPT2LMHeadModel,
    GPT2Tokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
from datasets import Dataset
import json
import re
import random
from typing import List, Dict, Optional
import warnings
from datetime import datetime
import os

warnings.filterwarnings('ignore')

if torch.cuda.is_available():
    device = torch.device('cuda')
    print(f"GPU使用: {torch.cuda.get_device_name(0)}")
    print(f"利用可能VRAM: {torch.cuda.get_device_properties(0).total_memory / 1024**3:.1f}GB")
else:
    device = torch.device('cpu')
    print("CPU使用（警告: 学習に時間がかかります）")

torch.manual_seed(42)
random.seed(42)
```

📚 コード概要

【インポート部分】

```
python
```

```
import torch
from transformers import (
    GPT2LMHeadModel,
    GPT2Tokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
from datasets import Dataset
import json
import re
import random
from typing import List, Dict, Optional
import warnings
from datetime import datetime
import os
```

GPT2LMHeadModel → GPT-2の言語モデル本体。

GPT2Tokenizer → GPT-2用のトークナイザー。

TrainingArguments, Trainer → モデルの学習設定と学習ループ管理を簡単にするクラス。

DataCollatorForLanguageModeling → 言語モデル用のデータ整形ユーティリティ。

datasets.Dataset → Hugging Faceのデータセットラッパー。

json, re, random, typing, warnings, datetime, os → それぞれデータ処理、型ヒント、警告管理、日時操作、ファイル操作などで使う一般ライブラリ。

【警告非表示】

```
python

warnings.filterwarnings('ignore')
```

→ 開発時の余計な警告を隠す

【デバイスの判定と設定】

```
python
```

```
if torch.cuda.is_available():
    device = torch.device('cuda')
    print(f"GPU使用: {torch.cuda.get_device_name(0)}")
    print(f"利用可能VRAM: {torch.cuda.get_device_properties(0).total_memory / 1024**3:.1f}GB")
else:
    device = torch.device('cpu')
    print("CPU使用（警告: 学習に時間がかかります）")
```

- GPUが使えるかどうかチェックして、使えるならGPU（cuda）、なければCPUを選択。
- GPUなら名前とVRAM容量を表示してくれるので、環境の確認に便利。
- CPUのみの場合は「時間かかります」と警告。

【乱数シード固定】

```
python

torch.manual_seed(42)
random.seed(42)
```

① **torch.manual_seed(X)** → PyTorchの乱数生成器。モデルの重み初期化、テンソル生成(torch.rand/torch.randnなど)

② **random.seed(42)** → Pythonの標準乱数生成器。random.randint, random.shuffleなどPythonの標準乱数

◆ 2つ必要な意味

データ関連 → Pythonのrandomを使う（例：学習データをシャッフル）

モデルやテンソル関連 → PyTorchの乱数を使う（例：重み初期化）

両方で乱数が使われるから、両方のシードを固定しないと完全再現できない。

- `torch.manual_seed(x)` → モデル内部のランダム性を固定
- `random.seed(x)` → Python側のランダム性を固定

◆ コードブロック2の全体の解釈

土台つくりの部分。GPU/CPU判定で計算環境を設定し、結果のブレを防ぐために乱数シードを固定する。

📝 **【備忘録】**

乱数シードを固定する意味

深層学習では、重みの初期化やデータシャッフルなどに乱数が使われる

もし、シード値を固定しないと...

1. **毎回モデルの初期状態が違う** → 学習結果が毎回微妙に変わる
2. **結果の再現性がなくなる** → 他人や未来の自分が同じ結果を再現できない

といった問題が起こる

シード値は「乱数列を作る際のスタート番号」みたいなもの
同じシード値を使えば、同じ乱数列が生成されるので結果も同じになる

🌌 【42の由来】

元ネタは、ダグラス・アダムスのSF小説
『銀河ヒッチハイク・ガイド』（The Hitchhiker's Guide to the Galaxy）

物語の中で、超スーパーコンピュータ「ディープ・ソート」に 「生命、宇宙、そして万物の究極の疑問の答え」を計算させたら、 答えが **"42"** だった というシュールなオチ。

その意味は最後まで明かされず、「答えは42」というジョークとして有名に。

---

② モデルとトークナイザーの詳細設定

🎯 GPT-2のサイズ

1. **GPT-2 (124M パラメータ)** : 最小サイズ、Colabで安定動作
2. **GPT-2-medium (355M パラメータ)** : より高品質だがメモリ使用量大
3. **GPT-2-large (774M パラメータ)** : さらに高品質だがColab無料版では厳しい

🤖 コードブロック3

```python
```

```
model_name = "gpt2"
print(f"モデル '{model_name}' を読み込み中...")

try:
    tokenizer = GPT2Tokenizer.from_pretrained(model_name)

    if tokenizer.pad_token is None:
        tokenizer.pad_token = tokenizer.eos_token
        print("パディングトークンをEOSトークンに設定しました。")

    model = GPT2LMHeadModel.from_pretrained(model_name)
    model.to(device)

    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    print(f"モデル読み込み完了!")
    print(f"総パラメータ数: {total_params:,}")
    print(f"学習対象パラメータ数: {trainable_params:,}")

except Exception as e:
    print(f"モデル読み込みエラー: {e}")
    print("インターネット接続またはHugging Faceのサーバー状況を確認してください。")
```

📊 コード概要

【モデルサイズの選択】

```
python

model_name = "gpt2"
print(f"モデル '{model_name}' を読み込み中...")
```

→ `model_name` に "gpt2" を指定 Hugging Faceのモデル名で、この場合は一番小さいGPT-2（約1.24億パラメータ）を使う 他には"gpt2-medium", "gpt2-large", "gpt2-xl" などがある

【トークナイザーの読み込み】

```
python

tokenizer = GPT2Tokenizer.from_pretrained(model_name)
```

→ Hugging FaceからGPT-2用のBPEトークナイザーをロード `from_pretrained`...ネットからモデルや辞書ファイルをダウンロードしてキャッシュに保存

【パディングトークンの設定】

```python
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
    print("パディングトークンをEOSトークンに設定しました。")
```

→ GPT-2にはパディング用トークンが存在しない（BERTみたいな[PAD]なし）
バッチ学習で長さを揃えるため、代わりに文末トークン（eos_token）を流用する

【モデル本体の読み込み】

```python
model = GPT2LMHeadModel.from_pretrained(model_name)
```

→ GPT-2の「言語モデル」部分をロード（LMHeadは単語予測用の最終層を含む構造）
事前学習済みの重みも一緒に読み込む

【デバイスに移動】

```python
model.to(device)
```

→ GPUがあればGPUメモリへ転送して計算を高速化。CPUの場合はメモリ上で計算（遅くなる）。

【パラメータ数の表示】

```python
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
```

→ **総パラメータ数**: モデル全体の重みの数。

学習対象パラメータ数: 勾配計算が有効（requires_grad=True）なパラメータ数。

転移学習や微調整（fine-tuning）の時、固定されてる層があるとこの値は減る。

【例外処理】

```python
except Exception as e:
    print(f"モデル読み込みエラー: {e}")
    print("インターネット接続またはHugging Faceのサーバー状況を確認してください。")
```

→ モデルやトークナイザーのロードが失敗したときにエラー原因を表示。
　ネット接続やHugging Face側の障害もここで検知。

◆ コードブロック3の全体の解釈

全体的に初期のセットアップ工程

1. **モデルサイズ選択**
2. **トークナイザー準備**（パディング設定含む）
3. **モデル読み込み**
4. **デバイス転送 & パラメータ数確認**

📝 **【備忘録】**

パディングについて

```python
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
```

GPT-2はpad_tokenを持ってない

- GPT-2は言語生成モデルなので、「途中で埋める」という発想がなく、`pad_token` が None（未定義）。なので、このまま`padding=True`で使おうとするとエラーになる

EOSトークンを代わりに使う理由

- GPT-2では**`<eos>`（End Of Sequence）**が「文の終わり」を表す唯一の特別トークン。
- バッチ入力でパディングが必要な場合、`pad_token` の代わりに `eos_token` を使うのが一般的な対処法
- 言語モデルは基本的に「終わった後の空白部分」は学習に影響しにくい（attention_maskで無視できる）
- pad専用トークンを作るより、既存のEOSを使ったほうがシンプルで安全

まとめ

- GPT-2には`pad_token`がない → 代わりに`eos_token`を設定
- これで`padding=True`が使えるようになり、Trainerやデータローダーがバッチ化できる
- 実際の計算では`attention_mask`がpad部分を無視する

パラメーター数の表示について

```python
```

```
total_params = sum(p.numel() for p in model.parameters())
trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
```

① **model.parameters()とは** → PyTorchの nn.Module（= GPT-2モデル）が持っているすべての学習可能な重み（nn.Parameter）を順番に返すジェネレータ。

つまり、`sum(p.numel() for p in model.parameters())`はGPT-2モデルに含まれるすべての重みやバイアスの数字の総和を足し合わせたもの＝そのモデルの規模感がわかる。

② **trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)** → 今回のこの学習ではこのくらいの重み、バイアスを使用するという規模感を表示する

```
python

print(f"モデル読み込み完了!")
print(f"総パラメータ数: {total_params:,}")
print(f"学習対象パラメータ数: {trainable_params:,}")
```

---

③ 高品質な会話データセットの構築

コードブロック4

```
python
```

```python
def create_comprehensive_dataset():
    """
    多様なシナリオをカバーする会話データセットを作成
    各カテゴリーから自然な会話パターンを収集
    """

    conversations = [
        # 挨拶・基本会話
        {"user": "おはようございます", "bot": "おはようございます！今日も良い一日をお過ごしください。何かお手伝い
        {"user": "こんにちは", "bot": "こんにちは！お疲れさまです。今日はどのようなことについてお話しましょうか？
        {"user": "こんばんは", "bot": "こんばんは！一日お疲れさまでした。リラックスしてお話しましょう。"},

        # 感情・状態確認
        {"user": "元気ですか？", "bot": "はい、とても元気です！あなたはいかがですか？今日は調子はどうでしょう？"
        {"user": "疲れました", "bot": "お疲れさまです。少し休憩を取られてはいかがでしょうか？何か気分転換になるこ
        {"user": "嬉しいことがありました", "bot": "それは素晴らしいですね！よろしければ、どのような嬉しいことがお

        # 日常会話
        {"user": "今日は何をしていましたか？", "bot": "私はAIなので特別な活動はしていませんが、多くの方とお話しす
        {"user": "天気はどうですか？", "bot": "申し訳ありませんが、リアルタイムの天気情報は取得できません。お住ま
        {"user": "時間を教えて", "bot": "申し訳ありませんが、現在時刻は表示できません。デバイスの時計をご確認くだ

        # 趣味・娯楽
        {"user": "おすすめの映画を教えて", "bot": "映画のジャンルによりますね。アクション、ドラマ、コメディ、アニ
        {"user": "音楽は好きですか？", "bot": "音楽について話すのは楽しいです！クラシック、ポップ、ロック、ジャズ
        {"user": "読書について", "bot": "読書は知識を広げる素晴らしい趣味ですね。小説、ビジネス書、自己啓発書など

        # 学習・仕事
        {"user": "勉強について相談があります", "bot": "勉強についてお手伝いできることがあれば嬉しいです。どのよう
        {"user": "プログラミングを学びたい", "bot": "プログラミング学習は素晴らしい挑戦ですね！Python、JavaScript
        {"user": "仕事が大変です", "bot": "仕事でお疲れのようですね。具体的にどのような点が大変でしょうか？時間管

        # 感謝・終了
        {"user": "ありがとうございます", "bot": "どういたしまして！お役に立てて嬉しいです。他にも何かご質問やお話
        {"user": "また今度話しましょう", "bot": "はい、ぜひまたお話しましょう！いつでもお待ちしています。素晴らし
        {"user": "さようなら", "bot": "さようなら！今日はお話しできて楽しかったです。また次回お会いできることを楽

        # エラーハンドリング用
        {"user": "よくわからない質問", "bot": "申し訳ありませんが、質問の内容をもう少し具体的に教えていただけます
    ]

    return conversations


conversations = create_comprehensive_dataset()
print(f"会話データセット作成完了: {len(conversations)}件の会話ペア")
```

```
print("\n=== データセット例 ===")
for i, conv in enumerate(conversations[:3]):
    print(f"例 {i+1}:")
    print(f"  ユーザー: {conv['user']}")
    print(f"  ボット: {conv['bot']}")
    print()
```

📈 コード概要

【関数定義とドキュメンテーション】

```
python

def create_comprehensive_dataset():
```

→ 会話データセットを作成する関数を定義。

【会話データの構造】

```
python

conversations = [
    {"user": "おはようございます", "bot": "おはようございます！..."},
    {"user": "こんにちは", "bot": "こんにちは！..."},
    ...
]
```

→ 会話のリストの各要素(辞書)は`{"user": ユーザー発話, "bot": ボット応答}`の形式

カテゴリー別に会話を分けており、コード内コメントで整理（挨拶、日常会話、趣味、学習、感謝など）。

多様な入力に対して適切な応答を生成しやすいデータ構造を予測

【データセット作成と件数確認】

```
python

conversations = create_comprehensive_dataset()
print(f"会話データセット作成完了: {len(conversations)}件の会話ペア")
```

→ 関数を呼び出して、データセットを生成。　`len(conversations)`で会話ペアの件数を表示　出力例: 会話データセット作成完了: 19件の会話ペア

【データセットの内容確認】

```python
print("\n=== データセット例 ===")
for i, conv in enumerate(conversations[:3]):
    print(f"例 {i+1}:")
    print(f"  ユーザー: {conv['user']}")
    print(f"  ボット: {conv['bot']}")
    print()
```

`enumerate(conversations[:3])` → データセットの先頭3件だけを確認するループ。 `conv['user']` と `conv['bot']` でユーザー発話とボット応答を表示。

デバッグや動作確認、データの品質チェックに便利。

◆ コードブロック4の全体の解釈

日本語の会話データセットをまとめて作る関数を定義 `create_comprehensive_dataset()` が呼ばれると、挨拶・雑談・タスク依頼・感情表現・トラブル対応など、多様なシナリオに対応した ユーザー発話とボット返答のペア をリスト形式で返す

📝 **【備忘録】**

create_comprehensive_dataset（総合的なデータセットを作成する）

ここに入れ込んでいく。
リスト形式で、要素はすべて 辞書（{}）。

各辞書は

"user" → ユーザーの発話（質問やメッセージ）

"bot" → それに対応するボットの返答

用意しているカテゴリ：

1. 挨拶・基本会話
2. 感情・状態確認
3. 日常会話
4. 趣味・娯楽
5. 学習・仕事
6. 感謝・終了
7. エラーハンドリング

```python
```

```
return conversations
```

→ この関数を呼び出すと、上で作ったリストが返されます。

```python
conversations = create_comprehensive_dataset()
print(f"会話データセット作成完了: {len(conversations)}件の会話ペア")
```

→ 関数を呼び出し、返ってきたリストを conversations に代入。 `len(conversations)` で会話ペアの総数を表示。

サンプルの表示

```python
for i, conv in enumerate(conversations[:3]):
    print(f"例 {i+1}:")
    print(f"  ユーザー: {conv['user']}")
    print(f"  ボット: {conv['bot']}")
    print()
```

最初の3件だけサンプルとして表示。 `enumerate` で番号を付けながら user と bot の内容を整形して出力。

📁 コードブロック5

```python

```

```python
def load_conversation_data(file_path: str) -> List[Dict[str, str]]:
    """
    JSONファイルから会話データを読み込む関数

    JSONファイル形式例:
    [
        {"user": "質問1", "bot": "回答1"},
        {"user": "質問2", "bot": "回答2"}
    ]
    """
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)

        # データ形式の検証
        if not isinstance(data, list):
            raise ValueError("データは配列形式である必要があります")

        for item in data:
            if not isinstance(item, dict) or 'user' not in item or 'bot' not in item:
                raise ValueError("各要素にはuserとbotキーが必要です")

        print(f"外部ファイルから {len(data)} 件の会話データを読み込みました")
        return data

    except FileNotFoundError:
        print(f"ファイル '{file_path}' が見つかりません。デフォルトデータを使用します。")
        return create_comprehensive_dataset()
    except json.JSONDecodeError as e:
        print(f"JSONファイルの解析エラー: {e}")
        return create_comprehensive_dataset()
    except Exception as e:
        print(f"データ読み込みエラー: {e}")
        return create_comprehensive_dataset()

# 外部ファイルがある場合は読み込み、なければデフォルトデータを使用
conversations = load_conversation_data('chatbot_data.json')
```

📊 コード概要

【関数定義部分】

```python
def load_conversation_data(file_path: str) -> List[Dict[str, str]]:
```

→ (file_path: str)　引数に読み込むJSONのパスを入れて文字列型で渡すように指定　-> List[Dict[str, str]]:

→ 会話のリストを戻り値として設定

【ファイル読み込み処理】

```python
with open(file_path, 'r', encoding='utf-8') as f:
    data = json.load(f)
```

→ with を使うと、処理が終わった後に自動でcloseされる (file_path, 'r', encoding='utf-8') ①file_path → 読み込みたいファイルのパス、'r' → モード指定 "r" は読み込み専用 ②encoding='utf-8' → UTF-8 文字コードで読み込み

指定されたパスのJSONファイルをUTF-8で開き、その内容をPythonのリスト・辞書形式に変換して data に代入する

【データ形式の検証】

```python
if not isinstance(data, list):
    raise ValueError("データは配列形式である必要があります")

for item in data:
    if not isinstance(item, dict) or 'user' not in item or 'bot' not in item:
        raise ValueError("各要素にはuserとbotキーが必要です")
```

- if not isinstance(data, list): → dataという変数がリスト型かどうか確認する
- isinstance(変数, 型) → 変数の型が一致しているか調べる関数
- not が付くので、もしリスト型でなければ True になり、次の行（raise）が実行される。
- ValueError は「値が想定と違うとき」に使うPythonの例外クラス
- 「配列じゃない」という警告を出して、後続処理を止める

(itemの条件) ①dict（辞書型）であること ②'user' というキーがあること ③'bot' というキーがあること

↑会話データの最低限の形を確保する

【正常処理】

```python
```

```
print(f"外部ファイルから {len(data)} 件の会話データを読み込みました")
return data
```

→ 読み込んだ会話データが何件あるかを表示する `len(data)` → リストの要素数（件数）

動作の流れ：

1. JSONファイルを開いてPythonのデータ構造に変換。
2. 型やキーの存在チェックで安全性を確保。
3. 件数を表示してデータを返す。

【例外処理（フォールバック機構）】

■ ファイルが存在しない場合

```
python

except FileNotFoundError:
    print(f"ファイル '{file_path}' が見つかりません。デフォルトデータを使用します。")
    return create_comprehensive_dataset()
```

→ `FileNotFoundError` → 指定されたパスのファイルが存在しないときに発生　この場合はエラーメッセージを表示し、代わりに `create_comprehensive_dataset()` で作ったデフォルトデータを返す 「ファイルがないから、用意してある安全な初期データで代用する」パターン

■ JSON形式エラー

```
python

except json.JSONDecodeError as e:
    print(f"JSONファイルの解析エラー: {e}")
    return create_comprehensive_dataset()
```

→ `json.JSONDecodeError` → ファイルが見つかっても、中身が正しいJSON形式じゃないと発生 例：カンマ抜け、{} や [] の閉じ忘れ

■ その他エラー

```
python

except Exception as e:
    print(f"データ読み込みエラー: {e}")
    return create_comprehensive_dataset()
```

→ `Exception` → Pythonの全ての例外の親クラス　上の2つのexceptではカバーできない想定外のエラー（例：権限エラー、エンコード不一致など）もここで捕まえる

【外部呼び出し部分】

```python
conversations = load_conversation_data('chatbot_data.json')
```

→「会話データを準備する入り口」
外部ファイルがあればそちらを優先、なければ内部のデフォルトを自動採用する仕組み

◆ コードブロック5の全体の解釈

コードブロック5は、会話データを安全に読み込むための仕組みを実装しています。
まずJSON形式の外部ファイルを開き、正しい形式かを検証します。
ファイルが無い・壊れている・形式が不正な場合は、例外処理によってエラーを防ぎ、
代わりに内部で用意したデフォルトデータ（create_comprehensive_dataset()）を返すことで常にconversations変数に有効な会話データが確保されるようになっています。

📝 **【備忘録】**

JSONファイル→文字列として保存が可能。Pythonだけでなくほぼ全言語で読み書き可能。

- Web APIやアプリ間通信のデータ形式として定番
- データは"キー":"値"のペアで表現。

コードブロック5ではチャットの「質問（user）」と「回答（bot）」をセットにした会話履歴をJSONで保存しておき、Pythonで読み込んでチャットボットのデータセットにしている。

🔄 コードブロック6

```python
```

```
def preprocess_conversations(conversations: List[Dict[str, str]],
                             tokenizer,
                             max_length: int = 256) -> Dict:
    """
    会話データを学習用に前処理する高度な関数

    Args:
        conversations: 会話データのリスト
        tokenizer: GPT-2トークナイザー
        max_length: 最大シーケンス長

    Returns:
        トークン化されたデータセット
    """

    processed_texts = []
    valid_conversations = 0

    print("会話データの前処理を開始...")

    for i, conv in enumerate(conversations):
        try:
            # テキストのクリーニング
            user_text = conv['user'].strip()
            bot_text = conv['bot'].strip()

            # 空文字チェック
            if not user_text or not bot_text:
                continue

            # 特殊トークンを使用した会話フォーマット
            # <|user|>と<|bot|>で明確に区別
            formatted_text = f"<|user|>{user_text}<|bot|>{bot_text}<|endoftext|>"

            # 長すぎるテキストの事前チェック
            if len(formatted_text) > max_length * 4:  # 大まかな文字数チェック
                print(f"警告: 会話 {i+1} が長すぎるため省略されました")
                continue

            processed_texts.append(formatted_text)
            valid_conversations += 1

        except Exception as e:
            print(f"会話 {i+1} の処理中にエラー: {e}")
            continue
```

```
    print(f"前処理完了: {valid_conversations}/{len(conversations)} 件の会話が有効")

    # バッチトークン化（効率的）
    try:
        encodings = tokenizer(
            processed_texts,
            truncation=True,
            padding=True,
            max_length=max_length,
            return_tensors="pt"
        )

        print(f"トークン化完了: {len(processed_texts)} 件のデータ")
        print(f"平均トークン数: {encodings['input_ids'].shape[1]}")

        return encodings

    except Exception as e:
        print(f"トークン化エラー: {e}")
        raise

# 前処理実行
max_length = 256  # Colabのメモリ制限を考慮
encodings = preprocess_conversations(conversations, tokenizer, max_length)

# データセット作成
train_dataset = Dataset.from_dict({
    'input_ids': encodings['input_ids'],
    'attention_mask': encodings['attention_mask'],
    'labels': encodings['input_ids'].clone()  # 言語モデリングではlabels=input_ids
})

print(f"学習用データセット作成完了: {len(train_dataset)} 件")

# データセットサンプルの確認
sample_idx = 0
sample_tokens = train_dataset[sample_idx]['input_ids']
sample_text = tokenizer.decode(sample_tokens, skip_special_tokens=False)
print(f"\nデータセットサンプル:\n{sample_text}")
```

🔧 コード概要

【関数定義と型ヒント】

python

```
def preprocess_conversations(conversations: List[Dict[str, str]],
                             tokenizer,
                             max_length: int = 256) -> Dict:
```

→ 引数の型指定

- `conversations: List[Dict[str, str]]` → 会話データのリスト（辞書のリスト）
- `tokenizer` → GPT-2トークナイザー
- `max_length: int = 256` → 最大シーケンス長（デフォルト256）
- `-> Dict:` → 戻り値は辞書形式

【前処理ループ】

```python
processed_texts = []
valid_conversations = 0

for i, conv in enumerate(conversations):
    try:
        # テキストのクリーニング
        user_text = conv['user'].strip()
        bot_text = conv['bot'].strip()

        # 空文字チェック
        if not user_text or not bot_text:
            continue
```

→ テキストクリーニング

- `.strip()` → 前後の空白・改行を削除
- 空文字チェックで無効なデータをスキップ

【会話フォーマット】

```python
# 特殊トークンを使用した会話フォーマット
# <|user|>と<|bot|>で明確に区別
formatted_text = f"<|user|>{user_text}<|bot|>{bot_text}<|endoftext|>"
```

→ 特殊トークンによる構造化

- `<|user|>` → ユーザー発話の開始マーカー

- `<|bot|>` → ボット応答の開始マーカー
- `<|endoftext|>` → 会話の終了マーカー

【長さチェック】

```python
# 長すぎるテキストの事前チェック
if len(formatted_text) > max_length * 4:  # 大まかな文字数チェック
    print(f"警告: 会話 {i+1} が長すぎるため省略されました")
    continue
```

→ 効率的な事前フィルタリング

- 文字数ベースで大まかにチェック
- `max_length * 4` → トークン数の概算（1文字≈0.25トークン）

【バッチトークン化】

```python
encodings = tokenizer(
    processed_texts,
    truncation=True,
    padding=True,
    max_length=max_length,
    return_tensors="pt"
)
```

→ 効率的なバッチ処理

- `truncation=True` → 長いテキストを切り詰め
- `padding=True` → 短いテキストをパディング
- `return_tensors="pt"` → PyTorchテンソル形式で返す

【データセット作成】

```python
train_dataset = Dataset.from_dict({
    'input_ids': encodings['input_ids'],
    'attention_mask': encodings['attention_mask'],
    'labels': encodings['input_ids'].clone()  # 言語モデリングではlabels=input_ids
})
```

→ Hugging Face Datasets形式

- `input_ids` → トークン化された入力
- `attention_mask` → パディング部分を無視するマスク
- `labels` → 学習ラベル（言語モデルでは入力と同じ）

◆ コードブロック6の全体の解釈

会話データを学習可能な形式に変換する前処理パイプラインです。テキストのクリーニング、特殊トークンによる構造化、バッチトークン化、Hugging Face Dataset形式への変換を効率的に行い、GPT-2の学習に適したデータセットを作成します。

📝 **【備忘録】**

特殊トークンの役割

- GPT-2に「どこからどこまでがユーザー発話で、どこからがボット応答か」を教える
- `<|user|>` と `<|bot|>` で明確に区別することで、より自然な会話学習が可能

バッチ処理の効率性

- 一つずつトークン化するより、リスト全体を一度に処理する方が高速
- GPUメモリの効率的な利用にもつながる

labels=input_idsの意味

- 言語モデルの学習では「次の単語を予測する」タスク
- 入力シーケンスを1つ右にシフトしたものが正解ラベルになる