GPT-2チャットボット開発ガイド - Google Colab版（改良版）

このガイドでは、Hugging FaceのGPT-2事前学習モデルを使用してGoogle Colabで実用的なチャットボットを作成する方法を詳しく説明します。初心者の方でも理解できるよう、各ステップを丁寧に解説し、実際に動作するコードを提供します。

1. 環境セットアップと理解

Google Colabの設定

まず、Google ColabでGPUを有効化しましょう。メニューの「ランタイム」→「ランタイムのタイプを変更」→「ハードウェアアクセラレータ」を「GPU」に設定します。これにより、モデルの学習と推論が大幅に高速化されます。

必要なライブラリの詳細説明

各ライブラリの役割を理解することで、エラーが発生した際の対処法も分かりやすくなります。

- **transformers**: Hugging Faceのライブラリで、GPT-2モデルとトークナイザーを提供
- **torch**: PyTorchの深層学習フレームワーク
- **datasets**: データセットの効率的な処理を行う
- **accelerate**: 分散学習とGPU使用の最適化

```python
# 必要なライブラリのインストール
# このプロセスには数分かかる場合があります
!pip install transformers==4.21.0 torch==1.12.1 datasets==2.4.0 accelerate==0.12.0

# 日本語テキスト処理用の追加ライブラリ
!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。")
```

基本ライブラリのインポートと設定

各インポートの意味と必要性を説明します。

```python
```

```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

# 警告メッセージを非表示にする（開発時の雑音を減らすため）
warnings.filterwarnings('ignore')

# デバイス設定の詳細確認
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）")

# 再現性のための乱数シード設定
torch.manual_seed(42)
random.seed(42)
```

2. モデルとトークナイザーの詳細設定

GPT-2モデルの選択と特徴

GPT-2には複数のサイズがあります：

- **gpt2** (124M パラメータ): 最小サイズ、Colabで安定動作
- **gpt2-medium** (355M): より高品質だがメモリ使用量大
- **gpt2-large** (774M): さらに高品質だがColab無料版では厳しい

```python
```

```python
# モデルサイズの選択（メモリ使用量を考慮）
model_name = "gpt2"  # Colab無料版に最適

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

try:
    # トークナイザーの読み込み
    tokenizer = GPT2Tokenizer.from_pretrained(model_name)

    # GPT-2は元々パディングトークンがないため、EOSトークンを使用
    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。")

    # モデルの読み込み
    model = GPT2LMHeadModel.from_pretrained(model_name)

    # GPUへの移動
    model.to(device)

    # モデル情報の表示
    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。")
```

3. 高品質な会話データセットの構築

データセット設計の重要性

チャットボットの品質は学習データに大きく依存します。多様で自然な会話例を用意することが重要です。

```python
```

```
# 高品質な日本語会話データセット
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た

        # 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

# データセット作成
conversations = create_comprehensive_dataset()
```

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

# データセットの内容確認
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()
```

外部ファイルからのデータ読み込み機能

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')
```

4. データ前処理とトークン化

高度な前処理機能の実装

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}")
```

5. 学習設定と最適化

詳細な学習パラメータ設定

python

```python
def create_training_arguments(output_dir: str = './gpt2-chatbot') -> TrainingArguments:
    """
    Google Colab環境に最適化された学習設定を作成
    """

    return TrainingArguments(
        # 出力設定
        output_dir=output_dir,
        overwrite_output_dir=True,

        # 学習パラメータ
        num_train_epochs=5,  # エポック数を増加
        per_device_train_batch_size=1,  # Colabメモリ制限対応
        gradient_accumulation_steps=8,  # 実効バッチサイズ = 1 * 8 = 8

        # 最適化設定
        learning_rate=3e-5,  # GPT-2に適した学習率
        weight_decay=0.01,   # 過学習防止
        warmup_steps=200,    # ウォームアップステップ

        # 保存とログ設定
        save_strategy="steps",
        save_steps=100,
        save_total_limit=3,  # 保存モデル数制限（ストレージ節約）

        logging_dir=f'{output_dir}/logs',
        logging_steps=20,

        # 評価設定（今回は訓練データのみのため無効化）
        evaluation_strategy="no",

        # パフォーマンス最適化
        dataloader_pin_memory=True,
        dataloader_num_workers=2,

        # Mixed Precision Training (GPU高速化)
        fp16=torch.cuda.is_available(),

        # その他
        report_to=None,  # WandBなどのログサービス無効化
        seed=42,
    )

# 学習設定作成
training_args = create_training_arguments()
```

```
# データコレーター設定
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False,  # GPT-2はCausal LM（次の単語予測）なのでMLMは無効
    return_tensors="pt"
)

print("学習設定完了:")
print(f"  エポック数: {training_args.num_train_epochs}")
print(f"  バッチサイズ: {training_args.per_device_train_batch_size}")
print(f"  勾配累積: {training_args.gradient_accumulation_steps}")
print(f"  学習率: {training_args.learning_rate}")
print(f"  FP16: {training_args.fp16}")
```

6. モデル学習の実行

堅牢な学習プロセス

python

```
def train_chatbot_model(model, tokenizer, train_dataset, training_args, data_collator):
    """
    エラーハンドリングを含む堅牢な学習関数
    """

    # トレーナーの初期化
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        data_collator=data_collator,
        tokenizer=tokenizer,
    )

    print("=== ファインチューニング開始 ===")
    print(f"学習データ数: {len(train_dataset)}")
    print(f"推定学習時間: {len(train_dataset) * training_args.num_train_epochs // (training_args.per_device_train_batch_siz

    try:
        # 学習実行
        start_time = datetime.now()
        model.train()

        # 学習履歴を取得
        train_result = trainer.train()

        end_time = datetime.now()
        training_duration = end_time - start_time

        print(f"\n=== 学習完了 ===")
        print(f"学習時間: {training_duration}")
        print(f"最終損失: {train_result.training_loss:.4f}")

        # モデル保存
        print("モデルを保存中...")
        trainer.save_model()
        tokenizer.save_pretrained(training_args.output_dir)

        # 学習履歴保存
        train_history = {
            'training_loss': train_result.training_loss,
            'training_duration': str(training_duration),
            'num_epochs': training_args.num_train_epochs,
            'learning_rate': training_args.learning_rate,
        }
```

```
        with open(f"{training_args.output_dir}/training_history.json", 'w') as f:
            json.dump(train_history, f, indent=2)

        print(f"モデルとトークナイザーを '{training_args.output_dir}' に保存しました")
        return trainer, train_result

    except Exception as e:
        print(f"学習中にエラーが発生: {e}")
        print("メモリ不足の場合は、バッチサイズを小さくするか、max_lengthを短くしてください")
        raise

# 学習実行
trainer, train_result = train_chatbot_model(
    model=model,
    tokenizer=tokenizer,
    train_dataset=train_dataset,
    training_args=training_args,
    data_collator=data_collator
)
```

7. 高度なチャット機能の実装

インテリジェントな応答生成システム

python

```python
class GPT2ChatBot:
    """
    高機能なGPT-2チャットボットクラス
    """

    def __init__(self, model, tokenizer, device, max_history=5):
        self.model = model
        self.tokenizer = tokenizer
        self.device = device
        self.conversation_history = []
        self.max_history = max_history

        # 応答生成パラメータ
        self.generation_params = {
            'max_new_tokens': 80,
            'temperature': 0.8,
            'top_p': 0.9,
            'top_k': 50,
            'repetition_penalty': 1.1,
            'do_sample': True,
            'pad_token_id': tokenizer.pad_token_id,
        }

    def generate_response(self, user_input: str, use_history: bool = True) -> str:
        """
        ユーザー入力に対して応答を生成

        Args:
            user_input: ユーザーからの入力
            use_history: 会話履歴を使用するかどうか

        Returns:
            生成された応答テキスト
        """
        try:
            # 入力のクリーニング
            user_input = user_input.strip()
            if not user_input:
                return "何かお話しください。"

            # 会話文脈の構築
            context = self._build_context(user_input, use_history)

            # トークン化
            inputs = self.tokenizer.encode(context, return_tensors='pt').to(self.device)
```

```
        # 応答生成
        self.model.eval()
        with torch.no_grad():
            outputs = self.model.generate(
                inputs,
                **self.generation_params
            )

        # 応答テキスト抽出
        generated_text = self.tokenizer.decode(outputs[0], skip_special_tokens=False)
        response = self._extract_bot_response(generated_text)

        # フィルタリング
        response = self._filter_response(response)

        # 履歴更新
        self._update_history(user_input, response)

        return response

    except Exception as e:
        print(f"応答生成エラー: {e}")
        return "申し訳ありません。応答の生成に問題が発生しました。"

def _build_context(self, user_input: str, use_history: bool) -> str:
    """会話文脈を構築"""
    context = ""

    if use_history and self.conversation_history:
        # 直近の会話履歴を使用
        recent_history = self.conversation_history[-self.max_history:]
        for turn in recent_history:
            context += f"<|user|>{turn['user']}<|bot|>{turn['bot']}"

    context += f"<|user|>{user_input}<|bot|>"
    return context

def _extract_bot_response(self, generated_text: str) -> str:
    """生成されたテキストからボットの応答を抽出"""
    try:
        # 最後の<|bot|>以降を取得
        if '<|bot|>' in generated_text:
            response = generated_text.split('<|bot|>')[-1]
            # 特殊トークンで終了
            response = response.split('<|endoftext|>')[0]
            response = response.split('<|user|>')[0]
            return response.strip()
```

```
            else:
                return "申し訳ありません。適切な応答を生成できませんでした。"
        except:
            return "応答の処理中にエラーが発生しました。"

    def _filter_response(self, response: str) -> str:
        """応答の品質フィルタリング"""
        # 空または短すぎる応答
        if len(response.strip()) < 2:
            return "もう少し詳しく教えてください。"

        # 長すぎる応答
        if len(response) > 200:
            sentences = response.split('。')
            if len(sentences) > 1:
                response = '。'.join(sentences[:2]) + '。'

        # 重複単語のチェック
        words = response.split()
        if len(words) > 5:
            unique_ratio = len(set(words)) / len(words)
            if unique_ratio < 0.3:
                return "別の方法で説明いたします。"

        return response

    def _update_history(self, user_input: str, bot_response: str):
        """会話履歴を更新"""
        self.conversation_history.append({
            'user': user_input,
            'bot': bot_response
        })

        # 履歴長の制限
        if len(self.conversation_history) > self.max_history:
            self.conversation_history.pop(0)

    def interactive_chat(self):
        """インタラクティブチャット機能"""
        print("=" * 50)
        print("GPT-2 チャットボット（改良版）")
        print("=" * 50)
        print("終了コマンド: quit, exit, bye, q")
        print("履歴クリア: clear")
        print("設定変更: config")
        print("-" * 50)
```

```
    while True:
        try:
            user_input = input("\n👤 あなた: ").strip()

            if not user_input:
                continue

            # 終了コマンド
            if user_input.lower() in ['quit', 'exit', 'bye', 'q']:
                print("\n🤖 ボット: さようなら！また話しましょう。")
                break

            # 履歴クリアコマンド
            if user_input.lower() == 'clear':
                self.conversation_history = []
                print("\n🤖 ボット: 会話履歴をクリアしました。")
                continue

            # 設定変更コマンド
            if user_input.lower() == 'config':
                self._config_menu()
                continue

            # 応答生成
            print("\n🤖 ボット: ", end="", flush=True)
            response = self.generate_response(user_input)
            print(response)

        except KeyboardInterrupt:
            print("\n\n🤖 ボット: チャットを終了します。お疲れさまでした！")
            break
        except Exception as e:
            print(f"\n❌ エラーが発生しました: {e}")
            continue

    return self.conversation_history

def _config_menu(self):
    """設定変更メニュー"""
    print("\n=== 設定変更 ===")
    print(f"1. 温度設定 (現在: {self.generation_params['temperature']})")
    print(f"2. 履歴長 (現在: {self.max_history})")
    print(f"3. 最大トークン数 (現在: {self.generation_params['max_new_tokens']})")
    print("0. 戻る")

    try:
        choice = input("選択してください (0-3): ").strip()
```

```
            if choice == '1':
                temp = float(input("温度 (0.1-2.0): "))
                if 0.1 <= temp <= 2.0:
                    self.generation_params['temperature'] = temp
                    print(f"温度を {temp} に設定しました。")
                else:
                    print("無効な値です。")

            elif choice == '2':
                history_len = int(input("履歴長 (1-10): "))
                if 1 <= history_len <= 10:
                    self.max_history = history_len
                    print(f"履歴長を {history_len} に設定しました。")
                else:
                    print("無効な値です。")

            elif choice == '3':
                max_tokens = int(input("最大トークン数 (20-150): "))
                if 20 <= max_tokens <= 150:
                    self.generation_params['max_new_tokens'] = max_tokens
                    print(f"最大トークン数を {max_tokens} に設定しました。")
                else:
                    print("無効な値です。")

        except ValueError:
            print("無効な入力です。")

# チャットボットインスタンス作成
chatbot = GPT2ChatBot(model, tokenizer, device)

print("チャットボット準備完了！")
```

8. モデルの評価と性能測定

包括的評価システム

python

```
class ChatBotEvaluator:
    """
    チャットボットの性能を多角的に評価するクラス
    """

    def __init__(self, chatbot: GPT2ChatBot):
        self.chatbot = chatbot
        self.evaluation_results = {}

    def evaluate_comprehensive(self) -> Dict:
        """包括的な評価を実行"""

        print("=== チャットボット総合評価開始 ===\n")

        # 1. 応答品質評価
        self._evaluate_response_quality()

        # 2. 一貫性評価
        self._evaluate_consistency()

        # 3. 多様性評価
        self._evaluate_diversity()

        # 4. 応答速度評価
        self._evaluate_response_time()

        # 5. エラーハンドリング評価
        self._evaluate_error_handling()

        # 結果サマリー
        self._generate_evaluation_summary()

        return self.evaluation_results

    def _evaluate_response_quality(self):
        """応答品質の評価"""
        print("1. 応答品質評価")

        test_cases = [
            {"input": "おはよう", "expected_keywords": ["おはよう", "こんにちは", "今日"]},
            {"input": "ありがとう", "expected_keywords": ["どういたしまして", "お役", "嬉しい"]},
            {"input": "疲れました", "expected_keywords": ["お疲れ", "休憩", "大変"]},
            {"input": "映画を見たい", "expected_keywords": ["映画", "ジャンル", "おすすめ"]},
            {"input": "勉強について", "expected_keywords": ["勉強", "学習", "効率"]},
        ]
```

```
        quality_scores = []

        for i, case in enumerate(test_cases, 1):
            response = self.chatbot.generate_response(case["input"], use_history=False)

            # キーワード関連性チェック
            keyword_score = self._calculate_keyword_relevance(
                response, case["expected_keywords"]
            )

            # 応答長チェック
            length_score = self._calculate_length_score(response)

            # 自然さチェック
            naturalness_score = self._calculate_naturalness_score(response)

            total_score = (keyword_score + length_score + naturalness_score) / 3
            quality_scores.append(total_score)

            print(f"  テスト {i}: 「{case['input']}」")
            print(f"    応答: 「{response}」")
            print(f"    品質スコア: {total_score:.2f}/1.0")
            print()

        avg_quality = sum(quality_scores) / len(quality_scores)
        self.evaluation_results['response_quality'] = {
            'average_score': avg_quality,
            'individual_scores': quality_scores,
            'grade': self._score_to_grade(avg_quality)
        }

        print(f"応答品質平均: {avg_quality:.2f}/1.0 ({self._score_to_grade(avg_quality)})")
        print("-" * 50)

    def _evaluate_consistency(self):
        """一貫性評価"""
        print("2. 一貫性評価")

        # 同じ質問を複数回実行
        test_input = "こんにちは"
        responses = []

        for i in range(5):
            response = self.chatbot.generate_response(test_input, use_history=False)
            responses.append(response)
            print(f"  試行 {i+1}: 「{response}」")
```

```
        # 応答の多様性と一貫性のバランスを評価
        consistency_score = self._calculate_consistency_score(responses)

        self.evaluation_results['consistency'] = {
            'score': consistency_score,
            'responses': responses,
            'grade': self._score_to_grade(consistency_score)
        }

        print(f"一貫性スコア: {consistency_score:.2f}/1.0 ({self._score_to_grade(consistency_score)})")
        print("-" * 50)

    def _evaluate_diversity(self):
        """多様性評価"""
        print("3. 応答多様性評価")

        diverse_inputs = [
            "天気について", "音楽について", "料理について", "旅行について", "読書について"
        ]

        responses = []
        for inp in diverse_inputs:
            response = self.chatbot.generate_response(inp, use_history=False)
            responses.append(response)
            print(f"  「{inp}」→「{response[:50]}...」")

        diversity_score = self._calculate_diversity_score(responses)

        self.evaluation_results['diversity'] = {
            'score': diversity_score,
            'grade': self._score_to_grade(diversity_score)
        }

        print(f"多様性スコア: {diversity_score:.2f}/1.0 ({self._score_to_grade(diversity_score)})")
        print("-" * 50)

    def _evaluate_response_time(self):
        """応答速度評価"""
        print("4. 応答速度評価")

        import time

        test_inputs = ["こんにちは", "元気ですか", "今日は何をしていましたか"]
        response_times = []

        for inp in test_inputs:
            start_time = time.time()
```

```
            response = self.chatbot.generate_response(inp, use_history=False)
            end_time = time.time()

            response_time = end_time - start_time
            response_times.append(response_time)

            print(f"  「{inp}」: {response_time:.2f}秒")

        avg_time = sum(response_times) / len(response_times)
        speed_score = max(0, min(1, (3.0 - avg_time) / 3.0))  # 3秒以下が満点

        self.evaluation_results['response_time'] = {
            'average_time': avg_time,
            'individual_times': response_times,
            'score': speed_score,
            'grade': self._score_to_grade(speed_score)
        }

        print(f"平均応答時間: {avg_time:.2f}秒")
        print(f"速度スコア: {speed_score:.2f}/1.0 ({self._score_to_grade(speed_score)})")
        print("-" * 50)

    def _evaluate_error_handling(self):
        """エラーハンドリング評価"""
        print("5. エラーハンドリング評価")

        error_cases = [
            "",  # 空入力
            "a" * 500,  # 極端に長い入力
            "🎉🎊✨🌟💫",  # 絵文字のみ
            "1234567890",  # 数字のみ
            "!@#$%^&*()",  # 記号のみ
        ]

        error_handling_scores = []

        for i, case in enumerate(error_cases, 1):
            try:
                response = self.chatbot.generate_response(case, use_history=False)

                # 応答が適切に生成されたかチェック
                if response and "エラー" not in response and "申し訳" in response:
                    score = 1.0
                elif response:
                    score = 0.7
                else:
                    score = 0.3
```

```
            error_handling_scores.append(score)

            case_display = case[:20] + "..." if len(case) > 20 else case
            print(f"  ケース {i}: 「{case_display}」 → スコア: {score:.1f}")

        except Exception as e:
            error_handling_scores.append(0.0)
            print(f"  ケース {i}: エラー発生 - {e}")

    avg_error_handling = sum(error_handling_scores) / len(error_handling_scores)

    self.evaluation_results['error_handling'] = {
        'average_score': avg_error_handling,
        'individual_scores': error_handling_scores,
        'grade': self._score_to_grade(avg_error_handling)
    }

    print(f"エラーハンドリングスコア: {avg_error_handling:.2f}/1.0 ({self._score_to_grade(avg_error_handling)})")
    print("-" * 50)

def _calculate_keyword_relevance(self, response: str, keywords: List[str]) -> float:
    """キーワード関連性スコア計算"""
    response_lower = response.lower()
    matches = sum(1 for keyword in keywords if keyword in response_lower)
    return min(1.0, matches / len(keywords))

def _calculate_length_score(self, response: str) -> float:
    """適切な応答長スコア計算"""
    length = len(response)
    if 10 <= length <= 100:
        return 1.0
    elif 5 <= length <= 150:
        return 0.7
    else:
        return 0.3

def _calculate_naturalness_score(self, response: str) -> float:
    """自然さスコア計算（簡易版）"""
    # 基本的なヒューリスティック
    if not response or len(response) < 2:
        return 0.0

    # 日本語の基本的な文章構造チェック
    has_hiragana = any('ひ' <= c <= 'ん' for c in response)
    has_punctuation = any(c in response for c in '。 ！ ？')
```

```
    score = 0.5
    if has_hiragana:
        score += 0.3
    if has_punctuation:
        score += 0.2

    return min(1.0, score)

def _calculate_consistency_score(self, responses: List[str]) -> float:
    """一貫性スコア計算"""
    if len(responses) < 2:
        return 0.0

    # 応答の類似度を簡易計算
    total_similarity = 0
    comparisons = 0

    for i in range(len(responses)):
        for j in range(i + 1, len(responses)):
            similarity = self._simple_similarity(responses[i], responses[j])
            total_similarity += similarity
            comparisons += 1

    avg_similarity = total_similarity / comparisons if comparisons > 0 else 0

    # 適度な一貫性（0.3-0.7）が理想
    if 0.3 <= avg_similarity <= 0.7:
        return 1.0
    else:
        return max(0.0, 1.0 - abs(avg_similarity - 0.5) * 2)

def _calculate_diversity_score(self, responses: List[str]) -> float:
    """多様性スコア計算"""
    if len(responses) < 2:
        return 0.0

    # 全応答間の類似度を計算
    total_similarity = 0
    comparisons = 0

    for i in range(len(responses)):
        for j in range(i + 1, len(responses)):
            similarity = self._simple_similarity(responses[i], responses[j])
            total_similarity += similarity
            comparisons += 1

    avg_similarity = total_similarity / comparisons if comparisons > 0 else 0
```

```
        # 低い類似度（高い多様性）が良い
        return max(0.0, 1.0 - avg_similarity)

    def _simple_similarity(self, text1: str, text2: str) -> float:
        """簡易文字列類似度計算"""
        words1 = set(text1.split())
        words2 = set(text2.split())

        if not words1 and not words2:
            return 1.0
        if not words1 or not words2:
            return 0.0

        intersection = words1.intersection(words2)
        union = words1.union(words2)

        return len(intersection) / len(union)

    def _score_to_grade(self, score: float) -> str:
        """スコアをグレードに変換"""
        if score >= 0.9:
            return "優秀"
        elif score >= 0.7:
            return "良好"
        elif score >= 0.5:
            return "普通"
        elif score >= 0.3:
            return "改善要"
        else:
            return "要改良"

    def _generate_evaluation_summary(self):
        """評価結果のサマリー生成"""
        print("=== 総合評価結果 ===")

        categories = [
            ('応答品質', 'response_quality'),
            ('一貫性', 'consistency'),
            ('多様性', 'diversity'),
            ('応答速度', 'response_time'),
            ('エラー処理', 'error_handling')
        ]

        total_score = 0
        for name, key in categories:
            score = self.evaluation_results[key]['score'] if key == 'response_time' else self.evaluation_results[key].get('averag
```

```
            grade = self.evaluation_results[key]['grade']
            print(f"{name}: {score:.2f}/1.0 ({grade})")
            total_score += score

        overall_score = total_score / len(categories)
        overall_grade = self._score_to_grade(overall_score)

        print(f"\n総合スコア: {overall_score:.2f}/1.0 ({overall_grade})")

        # 改善提案
        print(f"\n=== 改善提案 ===")
        self._generate_improvement_suggestions()

        self.evaluation_results['overall'] = {
            'score': overall_score,
            'grade': overall_grade
        }

    def _generate_improvement_suggestions(self):
        """改善提案の生成"""
        suggestions = []

        if self.evaluation_results['response_quality']['average_score'] < 0.7:
            suggestions.append("• より多様で高品質な学習データを追加")
            suggestions.append("• 学習エポック数の調整")

        if self.evaluation_results['consistency']['score'] < 0.6:
            suggestions.append("• 温度パラメータの調整（より安定した出力のため）")
            suggestions.append("• 応答フィルタリングの強化")

        if self.evaluation_results['diversity']['score'] < 0.5:
            suggestions.append("• top_pやtop_kパラメータの調整")
            suggestions.append("• より多様な学習データの使用")

        if self.evaluation_results['response_time']['score'] < 0.7:
            suggestions.append("• max_new_tokensの削減")
            suggestions.append("• GPUの使用確認")

        if suggestions:
            for suggestion in suggestions:
                print(suggestion)
        else:
            print("現在の性能は良好です。継続的な監視をお勧めします。")

# 評価実行
```

```
evaluator = ChatBotEvaluator(chatbot)
evaluation_results = evaluator.evaluate_comprehensive()
```

9. 実用的なデモとテスト

インタラクティブデモシステム

python

```python
def run_comprehensive_demo():
    """
    包括的なデモとテストを実行
    """

    print("=" * 60)
    print("GPT-2チャットボット 包括的デモ")
    print("=" * 60)

    # 1. 基本機能テスト
    print("\n1. 基本応答テスト")
    print("-" * 30)

    basic_tests = [
        "こんにちは",
        "今日の調子はどうですか？",
        "ありがとうございます",
        "映画について教えて",
        "さようなら"
    ]

    for test_input in basic_tests:
        response = chatbot.generate_response(test_input, use_history=False)
        print(f"入力: {test_input}")
        print(f"応答: {response}")
        print()

    # 2. 会話履歴機能テスト
    print("\n2. 会話履歴機能テスト")
    print("-" * 30)

    # 履歴をクリア
    chatbot.conversation_history = []

    conversation_flow = [
        "私の名前は田中です",
        "今日は良い天気ですね",
        "私の名前を覚えていますか？"
    ]

    for turn in conversation_flow:
        response = chatbot.generate_response(turn, use_history=True)
        print(f"入力: {turn}")
        print(f"応答: {response}")
        print(f"履歴数: {len(chatbot.conversation_history)}")
        print()
```

```
    # 3. 設定変更テスト
    print("\n3. 生成パラメータ比較テスト")
    print("-" * 30)

    test_input = "おすすめの本を教えて"

    # 元の設定を保存
    original_temp = chatbot.generation_params['temperature']

    temperatures = [0.3, 0.7, 1.0]

    for temp in temperatures:
        chatbot.generation_params['temperature'] = temp
        response = chatbot.generate_response(test_input, use_history=False)
        print(f"温度 {temp}: {response}")
        print()

    # 設定を復元
    chatbot.generation_params['temperature'] = original_temp

    # 4. エラーハンドリングテスト
    print("\n4. エラーハンドリングテスト")
    print("-" * 30)

    error_tests = ["", "???", "1234567890"]

    for error_input in error_tests:
        try:
            response = chatbot.generate_response(error_input, use_history=False)
            print(f"入力: '{error_input}' → 応答: {response}")
        except Exception as e:
            print(f"入力: '{error_input}' → エラー: {e}")

    print("\n" + "=" * 60)
    print("デモ完了！")
    print("=" * 60)

# デモ実行
run_comprehensive_demo()
```

10. 運用とメンテナンス

継続的改善システム

python

```python
class ChatBotMaintenanceSystem:
    """
    チャットボットの運用・メンテナンスを支援するクラス
    """

    def __init__(self, chatbot: GPT2ChatBot):
        self.chatbot = chatbot
        self.usage_stats = {
            'total_conversations': 0,
            'total_responses': 0,
            'average_response_length': 0,
            'common_inputs': {},
            'error_count': 0,
            'session_start': datetime.now()
        }
        self.feedback_log = []

    def log_interaction(self, user_input: str, bot_response: str, error: bool = False):
        """インタラクション情報をログ"""
        self.usage_stats['total_responses'] += 1

        if error:
            self.usage_stats['error_count'] += 1

        # 入力パターンの統計
        if user_input in self.usage_stats['common_inputs']:
            self.usage_stats['common_inputs'][user_input] += 1
        else:
            self.usage_stats['common_inputs'][user_input] = 1

        # 平均応答長の更新
        current_avg = self.usage_stats['average_response_length']
        total = self.usage_stats['total_responses']
        new_avg = ((current_avg * (total - 1)) + len(bot_response)) / total
        self.usage_stats['average_response_length'] = new_avg

    def collect_feedback(self, user_input: str, bot_response: str, rating: int, comment: str = ""):
        """ユーザーフィードバックを収集"""
        feedback = {
            'timestamp': datetime.now(),
            'user_input': user_input,
            'bot_response': bot_response,
            'rating': rating,  # 1-5
            'comment': comment
        }
        self.feedback_log.append(feedback)
```

```
    def generate_usage_report(self) -> str:
        """使用状況レポートを生成"""
        duration = datetime.now() - self.usage_stats['session_start']

        report = f"""
=== チャットボット使用状況レポート ===

セッション期間: {duration}
総応答数: {self.usage_stats['total_responses']}
エラー数: {self.usage_stats['error_count']}
エラー率: {(self.usage_stats['error_count'] / max(1, self.usage_stats['total_responses'])) * 100:.1f}%
平均応答長: {self.usage_stats['average_response_length']:.1f}文字

=== よく使われる入力 TOP5 ===
"""

        sorted_inputs = sorted(
            self.usage_stats['common_inputs'].items(),
            key=lambda x: x[1],
            reverse=True
        )[:5]

        for i, (input_text, count) in enumerate(sorted_inputs, 1):
            report += f"{i}. '{input_text}' ({count}回)\n"

        if self.feedback_log:
            avg_rating = sum(f['rating'] for f in self.feedback_log) / len(self.feedback_log)
            report += f"\n=== フィードバック統計 ===\n"
            report += f"フィードバック数: {len(self.feedback_log)}\n"
            report += f"平均評価: {avg_rating:.1f}/5.0\n"

        return report

    def suggest_improvements(self) -> List[str]:
        """改善提案を生成"""
        suggestions = []

        error_rate = (self.usage_stats['error_count'] / max(1, self.usage_stats['total_responses'])) * 100

        if error_rate > 10:
            suggestions.append("エラー率が高いです。入力検証の強化を検討してください。")

        if self.usage_stats['average_response_length'] > 150:
            suggestions.append("応答が長すぎる可能性があります。max_new_tokensの調整を検討してください。")
        elif self.usage_stats['average_response_length'] < 20:
            suggestions.append("応答が短すぎる可能性があります。生成パラメータの調整を検討してください。")
```

```
        if self.feedback_log:
            avg_rating = sum(f['rating'] for f in self.feedback_log) / len(self.feedback_log)
            if avg_rating < 3.0:
                suggestions.append("ユーザー満足度が低いです。学習データの見直しを推奨します。")

        # よく使われる入力パターンの分析
        if self.usage_stats['common_inputs']:
            most_common = max(self.usage_stats['common_inputs'].items(), key=lambda x: x[1])
            total_responses = self.usage_stats['total_responses']
            if most_common[1] / total_responses > 0.3:
                suggestions.append(f"'{most_common[0]}'のような入力が多いです。この分野の応答品質向上を検討してく

        return suggestions if suggestions else ["現在のパフォーマンスは良好です。"]

    def export_data(self, filepath: str):
        """統計データをエクスポート"""
        export_data = {
            'usage_stats': self.usage_stats,
            'feedback_log': [
                {
                    **feedback,
                    'timestamp': feedback['timestamp'].isoformat()
                }
                for feedback in self.feedback_log
            ],
            'export_timestamp': datetime.now().isoformat()
        }

        with open(filepath, 'w', encoding='utf-8') as f:
            json.dump(export_data, f, ensure_ascii=False, indent=2)

        print(f"データを {filepath} にエクスポートしました。")

# メンテナンスシステムの初期化
maintenance = ChatBotMaintenanceSystem(chatbot)

print("メンテナンスシステムが初期化されました。")
```

11. 最終的な使用方法とベストプラクティス

完全な実行例

python

```
def main_execution_example():
    """
    完全な実行例とベストプラクティスの実演
    """

    print("=" * 70)
    print("GPT-2チャットボット 完全実行ガイド")
    print("=" * 70)

    # ステップ1: モデルの状態確認
    print("\n📋 ステップ1: モデル状態確認")
    print(f"モデル: {model.__class__.__name__}")
    print(f"デバイス: {next(model.parameters()).device}")
    print(f"学習モード: {model.training}")

    # ステップ2: 基本動作確認
    print("\n🧪 ステップ2: 基本動作確認")
    test_response = chatbot.generate_response("動作テストです", use_history=False)
    print(f"テスト応答: {test_response}")

    # ステップ3: 設定確認と最適化
    print("\n⚙️ ステップ3: 設定最適化")
    print("現在の生成パラメータ:")
    for key, value in chatbot.generation_params.items():
        print(f"  {key}: {value}")

    # Colab環境に最適化された設定を適用
    optimized_params = {
        'max_new_tokens': 60,  # メモリ効率のため削減
        'temperature': 0.7,    # バランス重視
        'top_p': 0.9,
        'top_k': 50,
        'repetition_penalty': 1.1,
        'do_sample': True,
    }

    chatbot.generation_params.update(optimized_params)
    print("最適化されたパラメータを適用しました。")

    # ステップ4: 実際の使用例
    print("\n💬 ステップ4: 実使用例")

    demo_conversation = [
        "こんにちは、初めまして",
        "今日はプログラミングについて学びたいです",
        "Pythonから始めるのが良いでしょうか？",
```

```
        "ありがとうございました"
    ]

    print("デモ会話:")
    chatbot.conversation_history = []  # 履歴クリア

    for user_msg in demo_conversation:
        bot_response = chatbot.generate_response(user_msg, use_history=True)
        print(f"👤 ユーザー: {user_msg}")
        print(f"🤖 ボット: {bot_response}")
        print()

        # メンテナンス情報の記録
        maintenance.log_interaction(user_msg, bot_response)

    # ステップ5: パフォーマンス確認
    print("\n📊 ステップ5: パフォーマンス確認")
    print(maintenance.generate_usage_report())

    # ステップ6: 改善提案
    print("\n💡 ステップ6: 改善提案")
    improvements = maintenance.suggest_improvements()
    for suggestion in improvements:
        print(f"  • {suggestion}")

    print("\n✅ 実行完了!")
    print("チャットボットは使用準備が整いました。")

    return chatbot, maintenance

# メイン実行
chatbot_ready, maintenance_system = main_execution_example()
```

12. トラブルシューティングガイド

よくある問題と解決策

python

```python
class TroubleshootingGuide:
    """
    よくある問題の診断と解決策を提供
    """

    @staticmethod
    def diagnose_system():
        """システム診断を実行"""
        print("=" * 50)
        print("システム診断開始")
        print("=" * 50)

        issues = []

        # GPU確認
        print("\n🔍 GPU状況確認")
        if torch.cuda.is_available():
            gpu_name = torch.cuda.get_device_name(0)
            gpu_memory = torch.cuda.get_device_properties(0).total_memory / 1024**3
            gpu_allocated = torch.cuda.memory_allocated(0) / 1024**3
            gpu_free = gpu_memory - gpu_allocated

            print(f"  ✅ GPU利用可能: {gpu_name}")
            print(f"  📊 総メモリ: {gpu_memory:.1f}GB")
            print(f"  📈 使用中: {gpu_allocated:.1f}GB")
            print(f"  💾 空き: {gpu_free:.1f}GB")

            if gpu_free < 2.0:
                issues.append("GPU메모리 不足の可能性")
        else:
            print("  ❌ GPU未使用（CPU実行）")
            issues.append("GPU未使用による性能低下")

        # メモリ使用量確認
        print("\n🔍 メモリ使用量確認")
        try:
            import psutil
            memory = psutil.virtual_memory()
            print(f"  📊 システムメモリ使用率: {memory.percent}%")
            print(f"  💾 利用可能メモリ: {memory.available / 1024**3:.1f}GB")

            if memory.percent > 85:
                issues.append("システムメモリ使用率が高すぎます")
        except ImportError:
            print("  ⚠️ メモリ監視ツール未インストール")
```

```
        # モデル状態確認
        print("\n🔍 モデル状態確認")
        try:
            model_device = next(model.parameters()).device
            model_dtype = next(model.parameters()).dtype
            model_size = sum(p.numel() for p in model.parameters()) * 4 / 1024**2  # MB概算

            print(f"  📍 モデルデバイス: {model_device}")
            print(f"  🔢 データ型: {model_dtype}")
            print(f"  📏 推定サイズ: {model_size:.0f}MB")

            if str(model_device) == 'cpu' and torch.cuda.is_available():
                issues.append("モデルがGPUに移動されていません")

        except Exception as e:
            issues.append(f"モデル状態確認エラー: {e}")

        # 問題レポート
        print("\n📋 診断結果")
        if issues:
            print("  発見された問題:")
            for issue in issues:
                print(f"    ❌ {issue}")
        else:
            print("  ✅ 問題は検出されませんでした")

        return issues

    @staticmethod
    def suggest_solutions(issues):
        """問題に対する解決策を提案"""
        if not issues:
            print("\n✨ システムは正常に動作しています")
            return

        print("\n🔧 推奨解決策:")

        solutions = {
            "GPUメモリ不足": [
                "バッチサイズを1に削減",
                "max_lengthを128に短縮",
                "勾配累積ステップを増加",
                "ランタイムを再起動してメモリクリア"
            ],
            "GPU未使用": [
                "ランタイム→ランタイムのタイプを変更→GPU選択",
                "model.to(device)でGPUに移動を確認"
```

```
            ],
            "システムメモリ使用率": [
                "不要な変数をdel文で削除",
                "ガベージコレクション実行: import gc; gc.collect()",
                "Colabランタイムを再起動"
            ],
            "モデルがGPUに移動されていない": [
                "model.to(device)を再実行",
                "device設定を確認"
            ]
        }

        for issue in issues:
            for problem_type, solution_list in solutions.items():
                if problem_type in issue:
                    print(f"\n 🔸 {issue}")
                    for solution in solution_list:
                        print(f"    💡 {solution}")
                    break

    @staticmethod
    def performance_optimization():
        """パフォーマンス最適化のヒント"""
        print("\n ⚡ パフォーマンス最適化のヒント")

        tips = [
            "学習時はmodel.train()、推論時はmodel.eval()を確実に設定",
            "推論時はwith torch.no_grad():を使用してメモリ節約",
            "バッチサイズは小さく、勾配累積で実効バッチサイズを調整",
            "定期的にtorch.cuda.empty_cache()でGPUメモリをクリア",
            "長いシーケンスは避け、適切なmax_lengthを設定",
            "FP16を使用してメモリ使用量を半減",
            "データローダーのnum_workersを調整",
            "不要な出力やログを削減"
        ]

        for i, tip in enumerate(tips, 1):
            print(f"  {i}. {tip}")

# 診断実行
troubleshooter = TroubleshootingGuide()
detected_issues = troubleshooter.diagnose_system()
troubleshooter.suggest_solutions(detected_issues)
troubleshooter.performance_optimization()
```

13. 実用的なユーティリティ関数

便利な補助機能

python

```python
class ChatBotUtils:
    """
    チャットボット開発に役立つユーティリティ関数集
    """

    @staticmethod
    def save_conversation_history(history: List[Dict], filename: str = None):
        """会話履歴をJSONファイルに保存"""
        if filename is None:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            filename = f"conversation_history_{timestamp}.json"

        try:
            with open(filename, 'w', encoding='utf-8') as f:
                json.dump(history, f, ensure_ascii=False, indent=2)
            print(f"会話履歴を {filename} に保存しました")
            return filename
        except Exception as e:
            print(f"保存エラー: {e}")
            return None

    @staticmethod
    def load_conversation_history(filename: str) -> List[Dict]:
        """JSONファイルから会話履歴を読み込み"""
        try:
            with open(filename, 'r', encoding='utf-8') as f:
                history = json.load(f)
            print(f"会話履歴を {filename} から読み込みました ({len(history)}件)")
            return history
        except Exception as e:
            print(f"読み込みエラー: {e}")
            return []

    @staticmethod
    def analyze_conversation_patterns(history: List[Dict]):
        """会話パターンを分析"""
        if not history:
            print("分析する会話履歴がありません")
            return

        print("\n📊 会話パターン分析")
        print("-" * 30)

        # 基本統計
        total_turns = len(history)
        avg_user_length = sum(len(turn['user']) for turn in history) / total_turns
```

```
    avg_bot_length = sum(len(turn['bot']) for turn in history) / total_turns

    print(f"総ターン数: {total_turns}")
    print(f"ユーザー入力平均長: {avg_user_length:.1f}文字")
    print(f"ボット応答平均長: {avg_bot_length:.1f}文字")

    # よく使われる単語
    user_words = []
    bot_words = []

    for turn in history:
        user_words.extend(turn['user'].split())
        bot_words.extend(turn['bot'].split())

    from collections import Counter

    user_common = Counter(user_words).most_common(5)
    bot_common = Counter(bot_words).most_common(5)

    print("\nユーザーによく使われる単語:")
    for word, count in user_common:
        print(f"  {word}: {count}回")

    print("\nボット応答によく含まれる単語:")
    for word, count in bot_common:
        print(f"  {word}: {count}回")

@staticmethod
def benchmark_response_time(chatbot, test_inputs: List[str], runs: int = 3):
    """応答時間のベンチマーク"""
    print(f"\n⏱️ 応答時間ベンチマーク ({runs}回実行)")
    print("-" * 40)

    import time

    results = {}

    for test_input in test_inputs:
        times = []

        for run in range(runs):
            start_time = time.time()
            response = chatbot.generate_response(test_input, use_history=False)
            end_time = time.time()
            times.append(end_time - start_time)

        avg_time = sum(times) / len(times)
```

```
        min_time = min(times)
        max_time = max(times)

        results[test_input] = {
            'average': avg_time,
            'min': min_time,
            'max': max_time,
            'response_length': len(response)
        }

        print(f"'{test_input[:20]}...':")
        print(f"  平均: {avg_time:.2f}s")
        print(f"  最小: {min_time:.2f}s")
        print(f"  最大: {max_time:.2f}s")
        print(f"  応答長: {len(response)}文字")
        print()

    # 全体統計
    all_avg_times = [r['average'] for r in results.values()]
    overall_avg = sum(all_avg_times) / len(all_avg_times)
    print(f"全体平均応答時間: {overall_avg:.2f}秒")

    return results

@staticmethod
def export_model_info(model, tokenizer, filepath: str = "model_info.json"):
    """モデル情報をエクスポート"""
    try:
        model_info = {
            'model_name': model.config.name_or_path if hasattr(model.config, 'name_or_path') else 'unknown',
            'model_type': model.config.model_type if hasattr(model.config, 'model_type') else 'unknown',
            'vocab_size': tokenizer.vocab_size,
            'max_position_embeddings': model.config.max_position_embeddings if hasattr(model.config, 'max_position_e
            'num_parameters': sum(p.numel() for p in model.parameters()),
            'trainable_parameters': sum(p.numel() for p in model.parameters() if p.requires_grad),
            'device': str(next(model.parameters()).device),
            'dtype': str(next(model.parameters()).dtype),
            'export_timestamp': datetime.now().isoformat()
        }

        with open(filepath, 'w', encoding='utf-8') as f:
            json.dump(model_info, f, indent=2, ensure_ascii=False)

        print(f"モデル情報を {filepath} にエクスポートしました")

        # コンソールにも表示
        print("\n📋 モデル情報サマリー:")
```

```
            for key, value in model_info.items():
                if key != 'export_timestamp':
                    print(f"  {key}: {value:,}" if isinstance(value, int) else f"  {key}: {value}")

            return model_info

        except Exception as e:
            print(f"モデル情報エクスポートエラー: {e}")
            return None

    @staticmethod
    def memory_cleanup():
        """メモリクリーンアップ"""
        print("🧹 メモリクリーンアップ実行中...")

        import gc

        # Python ガベージコレクション
        collected = gc.collect()
        print(f"  Python GC: {collected}オブジェクト解放")

        # PyTorch GPU メモリクリア
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
            allocated = torch.cuda.memory_allocated(0) / 1024**3
            print(f"  GPU メモリクリア完了")
            print(f"  現在のGPU使用量: {allocated:.2f}GB")

        print("✅ クリーンアップ完了")

# ユーティリティの使用例
utils = ChatBotUtils()

# モデル情報のエクスポート
model_info = utils.export_model_info(model, tokenizer)

# ベンチマークテスト
benchmark_inputs = [
    "こんにちは",
    "今日は良い天気ですね",
    "プログラミングについて教えて",
    "ありがとうございます"
]

benchmark_results = utils.benchmark_response_time(chatbot, benchmark_inputs)
```

```
# メモリクリーンアップ
utils.memory_cleanup()
```

14. 最終チェックリストと運用開始

運用前最終チェック

```
python
```

```python
def final_system_check():
    """
    運用開始前の最終システムチェック
    """

    print("=" * 60)
    print("🔍 最終システムチェック")
    print("=" * 60)

    checks = []

    # 1. モデル動作確認
    print("\n1️⃣ モデル動作確認")
    try:
        test_response = chatbot.generate_response("システムチェック")
        if test_response and len(test_response) > 5:
            print("  ✅ モデル正常動作")
            checks.append(True)
        else:
            print("  ❌ モデル応答異常")
            checks.append(False)
    except Exception as e:
        print(f"  ❌ モデル動作エラー: {e}")
        checks.append(False)

    # 2. GPU使用確認
    print("\n2️⃣ GPU使用確認")
    if torch.cuda.is_available() and next(model.parameters()).is_cuda:
        print("  ✅ GPU使用中")
        checks.append(True)
    else:
        print("  ⚠️ CPU使用（性能低下の可能性）")
        checks.append(False)

    # 3. メモリ状態確認
    print("\n3️⃣ メモリ状態確認")
    if torch.cuda.is_available():
        memory_usage = torch.cuda.memory_allocated(0) / torch.cuda.max_memory_allocated(0) * 100
        if memory_usage < 80:
            print(f"  ✅ GPUメモリ 使用率: {memory_usage:.1f}%")
            checks.append(True)
        else:
            print(f"  ⚠️ GPUメモリ 使用率高: {memory_usage:.1f}%")
            checks.append(False)
    else:
        print("  ⚠️ GPUメモリ 確認不可")
```

```
    checks.append(False)

# 4. 応答品質確認
print("\n4️⃣ 応答品質確認")
quality_tests = [
    ("挨拶テスト", "こんにちは"),
    ("質問応答テスト", "元気ですか？"),
    ("感謝応答テスト", "ありがとう")
]

quality_scores = []
for test_name, test_input in quality_tests:
    response = chatbot.generate_response(test_input, use_history=False)
    # 簡易品質評価
    if response and 10 <= len(response) <= 150 and "エラー" not in response:
        quality_scores.append(1)
        print(f"  ✅ {test_name}: 合格")
    else:
        quality_scores.append(0)
        print(f"  ❌ {test_name}: 不合格")

quality_rate = sum(quality_scores) / len(quality_scores)
if quality_rate >= 0.7:
    print(f"  ✅ 応答品質: {quality_rate*100:.0f}%")
    checks.append(True)
else:
    print(f"  ❌ 応答品質不足: {quality_rate*100:.0f}%")
    checks.append(False)

# 5. エラーハンドリング確認
print("\n5️⃣ エラーハンドリング確認")
try:
    error_response = chatbot.generate_response("", use_history=False)
    if error_response:
        print("  ✅ エラー処理正常")
        checks.append(True)
    else:
        print("  ❌ エラー処理異常")
        checks.append(False)
except Exception as e:
    print(f"  ❌ エラー処理失敗: {e}")
    checks.append(False)

# 最終結果
print("\n" + "=" * 60)
passed_checks = sum(checks)
total_checks = len(checks)
```

```
    success_rate = passed_checks / total_checks * 100

    print(f"📊 最終結果: {passed_checks}/{total_checks} 項目合格 ({success_rate:.0f}%)")

    if success_rate >= 80:
        print("🎉 システム運用準備完了！")
        status = "READY"
    elif success_rate >= 60:
        print("⚠️ 一部問題がありますが使用可能です")
        status = "CAUTION"
    else:
        print("❌ 重大な問題があります。修正が必要です")
        status = "ERROR"

    return status, checks

# 最終チェック実行
system_status, check_results = final_system_check()

# 運用開始メッセージ
if system_status == "READY":
    print("\n" + "=" * 60)
    print("🚀 GPT-2チャットボット運用開始！")
    print("=" * 60)
    print()
    print("利用可能な機能:")
    print("  • chatbot.interactive_chat() - インタラクティブチャット")
    print("  • chatbot.generate_response(text) - 単発応答生成")
    print("  • evaluator.evaluate_comprehensive() - 性能評価")
    print("  • maintenance.generate_usage_report() - 使用状況報告")
    print("  • utils.memory_cleanup() - メモリクリーンアップ")
    print()
    print("使用例:")
    print("  chatbot.interactive_chat()  # チャット開始")
    print()
    print("楽しいチャットをお楽しみください！ 🤖✨")

elif system_status == "CAUTION":
    print("\n⚠️ 注意事項:")
    print("  • 応答速度が遅い場合があります")
    print("  • メモリ不足でエラーが発生する可能性があります")
    print("  • 定期的にutils.memory_cleanup()を実行してください")

else:
    print("\n❌ 修正が必要な項目:")
    print("  • GPU設定の確認")
```

```
print("・モデル読み込みの再実行")
print("・メモリ不足の解消")
```

15. まとめと今後の発展

このガイドで学んだこと

このガイドでは、以下の重要な要素をカバーしました：

基礎技術

- GPT-2モデルの理解と活用
- Hugging Face Transformersライブラリの使用
- Google Colab環境での効率的な開発

実装技術

- 高品質なデータセット設計
- 効果的なファインチューニング手法
- 応答生成の最適化
- エラーハンドリングとフィルタリング

運用技術

- 性能評価と品質管理
- メンテナンスとモニタリング
- トラブルシューティング
- 継続的改善

今後の発展可能性

1. **より大きなモデルへの移行**
   - GPT-2 medium/large
   - GPT-3.5やGPT-4等の最新モデル
2. **専門分野への特化**
   - カスタマーサポート
   - 教育支援
   - 創作支援
3. **多言語対応**
   - 英語やその他言語への展開
   - 多言語間翻訳機能

4. **外部システム連携**
   - API統合
   - データベース連携
   - ウェブアプリケーション化

このガイドが、あなたのチャットボット開発の成功につながることを願っています。継続的な学習と改善を通じて、より優れたAIシステムを構築してください！

最終更新: 2025年8月版 **対応環境**: Google Colab, Python 3.7+, PyTorch 1.12+ **ライセンス**: 教育・研究目的での自由利用可