査読論文 本文構成案：「トランスフォーマーにおける解釈可能な意味概念と非可換性処理の導入効果」

エコツーラボ合同会社
猪澤也寸志
polyp@ webman.jp
（2025年5月26日 現地時間JST）

アブストラクト (Abstract / 抄録)

- (以前の骨子案と同様の内容を、より具体的に記述)
    - 近年成功を収めているトランスフォーマーモデルは、多くのタスクで高い性能を示す一方で、その判断プロセスの不透明性（ブラックボックス性）や、文脈における順序性やより深い意味構造の扱いに課題を残している。本研究では、これらの課題に対処するため、トランスフォーマーの表現力を拡張する二つの異なるモジュール型アプローチを提案する。第一のアプローチは「意味アテンション機構」であり、学習可能な意味概念プロトタイプ群とアテンションメカニズムを利用して、入力テキストの文脈に応じた意味概念を特徴量に明示的に統合する。これにより、モデルの解釈可能性向上と意味文脈の最適化を目指す。第二のアプローチは「直接的非可換処理モジュール」であり、トランスフォーマーが出力するトークンシーケンス間の順序依存性や非可換な関係性を陽にモデル化し、より構造的な文脈理解を促進する。標準的なテキスト分類タスク（例：IMDbを用いた感情分析）において、これらの機構をそれぞれ独立してベースラインのトランスフォーマーモデルに組み込み、性能評価（Accuracy, F1スコア等）と比較を行った。実験の結果、[ここに主な定量的結果を簡潔に記述。例：意味アテンション機構は解釈性を大幅に向上させつつ、特定の条件下で性能向上を示した。非可換処理モジュールは、特に○○な性質を持つデータに対して有効であった。など]。さらに、各機構の内部動作に関する定性的な分析を通じて、その有効性と限界を考察する。本研究は、トランスフォーマーの能力を多角的に拡張し、より高度な意味理解と解釈可能性を有するAIモデルへの道筋を示す基礎的知見を提供する。

キーワード: トランスフォーマー、意味理解、解釈可能性、アテンション機構、非可換性、テキスト分類

1. 序論 (Introduction) * **1.1. 背景と研究の動機** * 近年におけるトランスフォーマーモデル (BERT, GPTなど) の自然言語処理分野での顕著な成功とその応用範囲の拡大。 * その一方で顕在化してきた課題： * ブラックボックス性：モデルがなぜそのような判断を下したのか、人間にとって理解することが困難。 * 深い意味理解の限界：表面的なパターンマッチングに留まり、人間のような深い意味構造やニュアンス、文脈の順序性を十分に捉えきれていない可能性。 * これらの課題を克服し、より信頼性が高く、人間と協調できるAIシステムを構築するための、解釈可能性と性能を両立させるアプローチの必要性。 * **1.2. 問題提起と本研究の目的** * **問題点1（意味の明示性）：** 標準的なトランスフォーマーは文脈情報を暗黙的に符号化するが、人間が理解・操作しやすい明示的な「意味概念」をどのようにモデルに導入し、活用できるか。 * **問題点2（順序性の扱い）：** テキストにおける単語や情報の順序は意味を理解する上で極めて重要であるが、トランスフォーマーの自己注意機構がこれをどの程度効果的に捉え、特に「非可換な」関係性（順序が入れ替わると意味が変わる関係）をどうモデル化できるか。 * **本研究の目的：** 上記の問題点に対し、(1) 解釈可能な意味概念を導入する「意味アテンション機構」、および (2) テキストの順序性と非可換性を陽にモデル化する「直接的非可換処理モジュール」という、二つの異なるアプローチを提案し、それぞれの有効性をトランス

フォーマーのベースラインと比較して検証する。特に、下流タスクの性能向上（意味文脈最適化）と、モデルの内部動作の理解しやすさ（ホワイトボックス効果）の観点から評価する。 * **1.3. 提案手法の概要とコントリビューション** * **意味アテンション機構の概要:** 学習可能な意味概念プロトタイプを用い、入力テキストの各部分がどの概念に注目すべきかをアテンションで動的に決定し、その情報を特徴量に統合する仕組み。 * **直接的非可換処理モジュールの概要:**隣接するトークン（またはその特徴量）間の順序性を考慮し、順方向と逆方向の処理結果の差異から非可換な特徴を抽出する仕組み。 * **本論文の主な貢献:** 1. 上記二つの新しいモジュールの具体的なアーキテクチャ設計と、既存のトランスフォーマーモデルへの統合方法の提案。 2. 標準的なテキスト分類タスクにおける、各モジュールの単独での効果をベースラインと比較・評価した実験結果の提示（定量的評価）。 3. 各モジュールの内部動作（意味概念の活性化パターン、非可換効果の現れ方など）を分析し、解釈可能性向上への寄与を考察する定性的評価。 4. これらの結果を通じて、トランスフォーマーの能力を補強・拡張するための異なるアプローチの特性と可能性を明らかにする。 * **1.4. 論文構成** * 第2章で関連研究を概観する。第3章で提案する二つのモジュールの詳細な設計を述べる。第4章で実験設定を説明し、第5章で実験結果とそれに基づく考察を示す。最後に第6章で結論と今後の展望を述べる。

2. 関連研究 (Related Work) * **2.1. トランスフォーマーの解釈可能性向上手法** * アテンション機構の可視化とその限界。 * 影響関数、LIME、SHAPなどの汎用的な説明手法のトランスフォーマーへの適用。 * プロービングタスクによる内部表現の分析。 * TCAV (Testing with Concept Activation Vectors) や類似の概念ベースの説明手法。 * 本研究の位置づけ：学習可能な概念プロトタイプを導入することで、よりモデル内部に根ざした解釈性を目指す点。 * **2.2. トランスフォーマーへの外部知識・構造情報の導入** * 知識グラフ埋め込みの統合。 * シンボリックなルールや制約の組み込み。 * メモリーネットワークの活用。 * 本研究の位置づけ：外部知識に直接依存するのではなく、データから「意味概念」や「順序性パターン」を学習・内部化するアプローチ。 * **2.3. 潜在概念・潜在構造の学習** * 変分オートエンコーダ（VAE）や敵対的生成ネットワーク（GAN）における潜在空間の解釈。 * ディープラーニングモデルが獲得する表現から、人間が理解可能な潜在的な要因や概念を発見しようとする研究。 * 本研究の位置づけ：「意味概念プロトタイプ」という形で、ある程度構造化された潜在概念の学習を促す点。 * **2.4. ニューラルネットワークにおける順序性・非可換性のモデル化 (★拡充)** * RNN、LSTM、GRUといった系列モデルの基本的な特性（順序情報の処理）。 * トランスフォーマーにおける位置エンコーディングの役割と限界。 * より明示的に順序や構造を捉えるためのアテンション機構の改良（例：Transformer-XLの相対位置エンコーディング、順序を意識した畳み込み層の導入など）。 * グラフニューラルネットワーク（GNN）における有向グラフの扱い。 * 自然言語処理における構文解析や意味役割ラベリングなど、構造的情報を活用するタスクでの取り組み。 * 本研究の位置づけ：トークンレベルの特徴量間の局所的な非可換関係を直接モデル化し、それを特徴量として活用する新しい試み。

3. 提案手法 (Proposed Methods) * **3.1. ベースラインモデルアーキテクチャ** * 本研究で比較の基準となるベースラインのトランスフォーマーモデルについて説明。 * 使用する事前学習済みモデル（例: BERT `bert-base-uncased`）。 * シーケンス分類タスクの場合、`[CLS]` トークンの最終層隠れ状態の上に線形分類層とソフトマックス関数を配置する標準的な構成。数式や図で示す。 * **3.2. モジュールA：意味アテンション機構 (`SemanticConceptModule`)** * **3.2.1. 設計思想と目的:** なぜこのモジュールが必要か。どのような「意味概念」を捉え、どのようにモデルの判断を助けることを目指すか。解釈可能性への期待。 * **3.2.2. アーキテクチャ詳細:**(以前の骨子の内容を数式や図を交えて詳細化) * 意味概念プロトタイプ (`N_concepts` 個の `D_concept` 次元ベクトル) の定義と学習可能性。 * ベースモデル出力 `H` (形状 `B, L, D_hidden`) から `H_proj` (形状 `B, L, D_concept`) への射影。 * アテンションスコア `S = H_proj @ P.T` (形状 `B, L, N_concepts`, `P` は概念プロトタイプ行列)。 * アテンションウェイト `A = softmax(S, dim=-1)`。 * 文脈化された意味概念表現 `C = A @ P` (形状 `B, L, D_concept`)。 * `H` と `C` の統合方法（例：`H_fused = H + Linear(C)` や `H_fused = Linear(torch.cat([H, C], dim=-1))`）と、最終的な出力次元 `D_output` の決定。 * **3.2.3. 期待される動作と解釈:** 学習後に `A` や `P` を分析することで何が分かるか。 * **3.3.**

モジュールB：直接的非可換処理モジュール (`DirectNonCommutativeModule`) * **3.3.1. 設計思想と目的:** なぜ非可換性に着目するのか。テキストのどのような順序依存情報を捉えたいか（例：隣接トークン間の非対称な関係性、論理的な流れなど）。 * **3.3.2. アーキテクチャ詳細:** (以前の提案コードを元に数式や図を交えて詳細化) * 入力 `H` (形状 `B, L, D_hidden`) から隣接トークンペア `(h_i, h_{i+1})` を作成。 * 順方向結合 `f_i = [h_i, h_{i+1}]` と逆方向結合 `b_i = [h_{i+1}, h_i]`。 * 非可換プロセッサ `NCP`(例: MLP) を用いた処理: `p_f_i = NCP(f_i)`, `p_b_i = NCP(b_i)`。 * 非可換効果 `nc_effect_i = \lambda * (p_f_i - p_b_i)` の計算 (`\lambda` は学習可能なパラメータ)。 * 計算された `nc_effect` を元の `H` にどう統合するか（例：`h'_i = h_i + nc_effect_i` のような形で、シーケンスの各位置に影響マップとして蓄積・適用後、必要なら最終射影）。出力次元 `D_output`。 * **3.3.3. 期待される動作と解釈:** `nc_effect` がどのような場合に大きな値を示すか。`\lambda` の学習結果。 * **3.4. 統合モデル (`EnhancedTransformerForSequenceClassification`) におけるモジュール選択** * `tech_flags` と `module_configs` を用いて、ベースライン、モジュールA搭載、モジュールB搭載の各モデル構成をどのように実現するか。`__init__` と `forward` メソッドにおけるモジュールの排他的な呼び出しロジックの概要。 * **3.5. 学習目的** * 全てのモデル構成において、下流タスク（シーケンス分類）の損失関数（例：クロスエントロピー損失）を用いたエンドツーエンド学習を行うことを明記。

4. 実験設定 (Experimental Setup) * **4.1. データセット:** * 使用データセット (例: IMDb) の詳細な説明（タスク、データソース、統計量：訓練/検証/テストのサンプル数、平均シーケンス長、クラス分布など）。 * テキストの前処理手順（クリーニング、小文字化など、もしあれば）。 * トークナイゼーションの詳細（使用トークナイザー、`max_length`、パディング・切り捨て戦略）。 * **4.2. 評価指標:** * **定量的指標:** 正解率 (Accuracy)、F1スコア（マクロ平均、必要ならクラスごと）、適合率、再現率を主要指標として使用することを明記。これらの指標の定義式も簡単に示すと丁寧。 * **定性的評価の方法の概要:** モジュールAについてはアテンションウェイトの可視化と概念プロトタイプの解釈、モジュールBについては非可換効果マップの可視化など、どのような分析を行うか。 * **4.3. 実験構成とハイパーパラメータ:** * **E0 (ベースライン):** `BASE_MODEL_NAME` を使用した標準的なトランスフォーマー分類器。 * **E1 (モジュールA搭載):** `SemanticConceptModule` のハイパーパラメータ (`N_concepts`, `D_concept`, 統合方法、出力次元など) の具体的な設定値。 * **E2 (モジュールB搭載):**`DirectNonCommutativeModule` のハイパーパラメータ（内部MLPの構造、`\lambda` の初期値、統合方法、出力次元など）の具体的な設定値。 * **共通の訓練ハイパーパラメータ:** オプティマイザの種類（例: AdamW）、学習率、バッチサイズ、総エポック数、ウォームアップ（もしあれば）、weight decay、乱数シード。 * **4.4. 実装詳細:** * 使用した主要ライブラリとバージョン（PyTorch, Transformers, Datasets, Accelerate, scikit-learnなど）。 * 実験に使用した計算環境（例: Google Colab (CPU/GPUの種類)、あるいは特定のハードウェア構成）。

5. 結果と考察 (Results and Discussion) * (このセクションは実際の実験結果に基づいて記述されるため、ここでは記述の「型」を示す) *

5.1. 定量的結果と分析 * **表1：主要評価指標の比較:** E0, E1, E2 の各構成におけるAccuracy, F1スコアなどをまとめた表を提示。平均値と標準偏差（複数回実行した場合）も示すと良い。 * **結果の記述:** 表の結果に基づき、どのモデルがどの指標で優れていたか、その差は統計的に有意か（もし検定していれば）などを客観的に記述。 * **「意味文脈最適化」に関する考察:** * モジュールAがベースラインより性能向上した場合、それは「意味概念」の導入が有効だったことを示唆。どのような側面で向上したか（例：特定のクラスの精度が上がったなど）。 * モジュールBがベースラインより性能向上した場合、それは「非可換性」の考慮が有効だったことを示唆。どのような側面で向上したか。 * もし性能が向上しなかった、あるいは低下した場合、その考えられる理由（学習不足、ハイパーパラメータの不適切さ、モジュール設計の課題、タスクとの相性など）を率直に議論する。 * **

する。 **5.2. モジュールA（意味アテンション機構）の定性的分析 (ホワイトボックス効果)** **図X：アテンションウェイトの可視化例:** いくつかの入力文を選び、各トークンがどの意味概念プロトタイプを強く活性化させたかを示すヒートマップなどを提示。 **分析:** 可視化結果から、モデルが文のどの部分に注目し、どのような「意味概念」を判断の手がかりにしているように見えるかを解釈・説明する。例えば、「この文では『悲しい』という単語が『ネガティブ感情』概念を強く活性化し、それが最終的な分類に繋がった」など。 **図Y：意味概念プロトタイプの解釈:** 学習された代表的な概念プロトタイプについて、それぞれを強く活性化する単語群や文の例を提示し、各プロトタイプがどのような意味を獲得したかを推定・記述する。 これらの分析が、モデルの透明性や信頼性の向上にどう繋がるかの考察。 **5.3. モジュールB（直接的非可換処理モジュール）の定性的分析 (順序性理解)** **図Z：非可換効果の分析例:** 語順を入れ替えると意味が変わる例文（例：能動態と受動態）や、特定の順序が重要な文を入力した際に、`non_commutative_effects_map` がどのような反応を示すか、あるいは `lambda_param` がどのように学習されたかを分析。 **分析:** モジュールBが、人間が直感的に「順序が重要だ」と感じる箇所で、実際に非可換性を捉えているか、あるいは予期せぬパターンを学習しているかなどを考察。 **5.4. エラー分析と比較** E0, E1, E2 がそれぞれどのようなタイプの文で間違いやすいか、あるいは得意とするかを比較。具体的な誤分類例を挙げて分析。 モジュール導入によって改善されたエラー、あるいは新たに発生したエラーの傾向。 **5.5. 総合的な議論** 提案した二つのモジュールは、それぞれどのような特性を持ち、どのような場合に有効性を示したか（あるいは示さなかったか）。 「意味概念の導入」と「非可換性の陽なモデル化」というアプローチの比較と、それぞれの限界。 今回の実験結果から得られた、トランスフォーマーの能力拡張に関する知見。

6. 結論 (Conclusion) 本研究の目的、提案手法、主要な実験結果、およびそこから得られた結論を簡潔に要約。 「意味アテンション機構」と「直接的非可換処理モジュール」が、トランスフォーマーの性能や解釈可能性、あるいは順序性理解に対してどのような貢献をしたか（あるいは課題を残したか）を明確に述べる。 本研究の意義と、今後のAI研究における位置づけ（もしあれば）。

7. 今後の課題 (Future Work) **7.1. モジュールの改良と拡張:** 今回の実験結果を踏まえ、各モジュールのアーキテクチャや学習方法の改善点。 ハイパーパラメータのより詳細な探索。 **7.2. モジュールの組み合わせと階層化（シリーズ論文の次段階へ）:** 「意味アテンション機構」と「非可換処理モジュール」を組み合わせたモデルの設計と評価。 お客様の最終目標である「断絶検出」と「意味創発ラベル生成」を含む、より高次の「断絶創発モジュール」の具体的な設計案と、その実現に向けたステップ。 これらのモジュールを階層的に組み合わせた場合の相乗効果や、新たな課題の探求。 **7.3. 他のタスクやデータセットへの適用:** 提案手法の汎用性を検証するため、異なる種類のタスク（例：質疑

応答、自然言語推論、要約など）や、異なる言語・ドメインのデータセットへの適用。 * **7.4.「動的辞書」や「正解データに基づく学習」の導入:** * お客様のビジョンにあるこれらの要素を、今後の研究でどのように具体化していくか。

謝辞 (Acknowledgements) (もしあれば) **参考文献 (References) 付録 (Appendix)** (詳細な実験設定、追加の分析結果、可視化例など)

実験コード 1

```
# セル1 (セットアップとカーネル再起動用)

# --------------------------------

# 関連ライブラリを一度アンインストールしてクリーンな状態にする

!pip uninstall -y transformers accelerate peft sentence-transformers datasets -q


# 指定バージョンでライブラリをインストール

!pip install transformers==4.36.2 -q

!pip install datasets -q

!pip install peft==0.7.1 -q

!pip install accelerate==0.25.0 -q  # <--- accelerateのバージョンを0.25.0に固定 (または0.26.1)


# 変更を適用するためにカーネルを再起動

import os

os.kill(os.getpid(), 9)
```

```
# --- セル2: ライブラリのインポート ---

import torch
import torch.nn as nn
import torch.nn.functional as F # SemanticConceptModule で F.softmax を使用するため

from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
    AutoConfig,
    PreTrainedModel,
    AutoModel
)
from transformers.modeling_outputs import SequenceClassifierOutput # モデルの出力形式用
from torch.nn import CrossEntropyLoss # 損失関数用

from datasets import load_dataset, Dataset # ★ Dataset クラスをインポート

import numpy as np
from sklearn.metrics import accuracy_score # 評価指標計算用
import itertools # 実験構成生成用 (今回は手動定義なので厳密には不要ですが、将来的に使う可能性も)

# (もし他に必要な標準ライブラリがあればここに追加してください)
```

```
# import os # (セル1で使用済みなので、ここでは必須ではない)
# import json
# import time
# import datetime

# (もしグラフ描画などを行う場合は、matplotlibなどもここでインポート)
# import matplotlib.pyplot as plt
# import seaborn as sns
# import pandas as pd

print("Cell 2: Libraries imported successfully.")
print(f"  PyTorch version: {torch.__version__}")
try:
    import transformers
    print(f"  Transformers version: {transformers.__version__}")
    import datasets
    print(f"  Datasets version: {datasets.__version__}")
    import accelerate
    print(f"  Accelerate version: {accelerate.__version__}")
    import peft
    print(f"  PEFT version: {peft.__version__}")
except ImportError:
    print("Warning: Could not print all Hugging Face library versions. Ensure they are installed.")


# --- セル3: 基本設定・共通パラメータ ---
```

```
from transformers import AutoTokenizer, AutoConfig # 必要なものをインポート (セル2でインポート済みなら重複を避けてもOK)


# --- 基本的な設定値 ---
BASE_MODEL_NAME = "bert-base-uncased"  # ベースとなる事前学習済みモデルの名前
NUM_LABELS = 2                # 分類タスクのラベル数 (例: IMDbならポジティブ/ネガティブの2)
OUTPUT_DIR_BASE = "./experiment_results/" # 実験結果の出力先ベースディレクトリ


# --- 訓練に関するハイパーパラメータ ---
LEARNING_RATE = 2e-5          # 学習率
BATCH_SIZE = 1                # バッチサイズ (GPUメモリやデータに応じて調整)
                              # CPU実行の場合は小さめ(例: 1や2)にしないと非常に遅い可能性があります
NUM_EPOCHS = 1                # 訓練エポック数 (最初は1などでテストし、問題なければ増やす)


# --- トークナイザーとモデル設定のロード ---
try:
    print(f"Loading tokenizer for {BASE_MODEL_NAME}...")
    tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL_NAME)
    print(f"Tokenizer for {BASE_MODEL_NAME} loaded successfully.")
except Exception as e:
    print(f"Error loading tokenizer for {BASE_MODEL_NAME}: {e}")
    raise # エラーが発生したら処理を停止
```

```
try:
    print(f"Loading Hugging Face model config for {BASE_MODEL_NAME}...")
    hf_model_config = AutoConfig.from_pretrained(
        BASE_MODEL_NAME,
        num_labels=NUM_LABELS
        # 必要に応じて他のAutoConfigのパラメータもここで設定可能
        # 例: finetuning_task='text-classification' など
    )
    print(f"Model config for {BASE_MODEL_NAME} loaded successfully.")
except Exception as e:
    print(f"Error loading model config for {BASE_MODEL_NAME}: {e}")
    raise

print("\n--- Cell 3: Basic/Common Settings Defined ---")
print(f"BASE_MODEL_NAME: {BASE_MODEL_NAME}")
print(f"NUM_LABELS: {NUM_LABELS}")
print(f"OUTPUT_DIR_BASE: {OUTPUT_DIR_BASE}")
print(f"LEARNING_RATE: {LEARNING_RATE}")
print(f"BATCH_SIZE: {BATCH_SIZE}")
print(f"NUM_EPOCHS: {NUM_EPOCHS}")
print(f"Tokenizer type: {type(tokenizer).__name__}")
print(f"HF Model Config type: {type(hf_model_config).__name__}")
```

```
# --- セル4: データセット準備 と compute_metrics 関数の定義 (カラム名修正をここで行う最終版) ---

from datasets import load_dataset
from sklearn.metrics import accuracy_score
import numpy as np
import torch

# --- グローバル変数 (セル3で定義されているはずのもの) の確認 ---
if 'BASE_MODEL_NAME' not in globals():
    raise NameError("Global variable 'BASE_MODEL_NAME' is not defined. Please ensure Cell 3 has been executed.")
if 'tokenizer' not in globals():
    raise NameError("Global variable 'tokenizer' is not defined. Please ensure Cell 3 has been executed.")

print(f"Starting dataset preparation using BASE_MODEL_NAME: {BASE_MODEL_NAME} and tokenizer: {type(tokenizer).__name__}")

# 1. データセットのロード
try:
    dataset_dict = load_dataset("imdb") # DatasetDictオブジェクトとしてロード
    print("IMDb dataset loaded successfully.")
except Exception as e:
    print(f"Error loading IMDb dataset: {e}")
    raise

# 2. トークナイズ関数の定義
```

```
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=256)

# 3. データセットのトークナイズ (必要なsplitのみを対象にすることも検討)
try:
    print("Tokenizing datasets...")
    # 'train' と 'test' splitのみをトークナイズする例 (unsupervisedは使わない想定)
    # もし他のsplitも使うなら、dataset_dict全体をmapするか、個別に処理
    tokenized_train = dataset_dict["train"].map(tokenize_function, batched=True,
remove_columns=["text"])
    tokenized_test = dataset_dict["test"].map(tokenize_function, batched=True,
remove_columns=["text"])
    print("Tokenization complete for train and test splits.")
except Exception as e:
    print(f"Error during tokenization: {e}")
    raise

# 4. 訓練用・評価用データセットの準備とカラム名修正
try:
    # 'label' カラムを 'labels' にリネーム
    if 'label' in tokenized_train.column_names and 'labels' not in tokenized_train.column_names:
        print("Renaming 'label' to 'labels' in tokenized_train...")
        final_tokenized_train = tokenized_train.rename_column("label", "labels")
    else:
        final_tokenized_train = tokenized_train
        if 'labels' not in final_tokenized_train.column_names:
            print("Warning: 'labels' column expected but not found in final_tokenized_train.")
```

```
    if 'label' in tokenized_test.column_names and 'labels' not in tokenized_test.column_names:
        print("Renaming 'label' to 'labels' in tokenized_test...")
        final_tokenized_test = tokenized_test.rename_column("label", "labels")
    else:
        final_tokenized_test = tokenized_test
        if 'labels' not in final_tokenized_test.column_names:
            print("Warning: 'labels' column expected but not found in final_tokenized_test.")

    # グローバル変数として train_dataset と eval_dataset を定義
    # (デバッグ用に縮小したサイズ)
    train_dataset = final_tokenized_train.shuffle(seed=42).select(range(20))
    eval_dataset = final_tokenized_test.shuffle(seed=42).select(range(10))

    print(f"Train dataset created. Number of samples: {len(train_dataset)}")
    print(f"  Train dataset FINAL column names: {train_dataset.column_names}")
    print(f"Evaluation dataset created. Number of samples: {len(eval_dataset)}")
    print(f"  Eval dataset FINAL column names: {eval_dataset.column_names}")

    if len(eval_dataset) > 0:
        print(f"First sample of EVAL_dataset (for column check): {eval_dataset[0]}")

except Exception as e:
    print(f"Error creating or renaming train/eval datasets: {e}")
    raise
```

```
# 5. 評価指標計算関数の定義 (デバッグプリント有効)

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    if labels is None or len(labels) == 0:
        print("--- Inside compute_metrics: Received no labels or labels is None. Returning empty metrics. ---")
        return {}
    predictions = np.argmax(logits, axis=-1)
    accuracy = accuracy_score(labels, predictions)
    metrics_dict = {"eval_accuracy": accuracy}
    print(f"--- Inside compute_metrics ---")
    print(f"  eval_pred logits type: {type(logits)}, labels type: {type(labels)}")
    if hasattr(logits, 'shape'): print(f"  eval_pred logits shape: {logits.shape}")
    if hasattr(labels, 'shape'): print(f"  eval_pred labels shape: {labels.shape}")
    print(f"  Predictions example (first 5): {predictions[:5]}")
    print(f"  Labels example (first 5): {labels[:5]}")
    print(f"  Calculated accuracy: {accuracy}")
    print(f"  Returning metrics_dict: {metrics_dict}")
    return metrics_dict


print("\nDataset preparation cell (Cell 4) complete.")
print(f"Defined global variables: 'train_dataset' (size: {len(train_dataset)}), 'eval_dataset' (size: {len(eval_dataset)}), 'compute_metrics'")
if callable(globals().get('compute_metrics')):
    print("'compute_metrics' function is defined and callable.")
else:
    print("Warning: 'compute_metrics' function is NOT defined or not callable.")
```

```
# --- セル4.1: ダミーデータセットによる上書き (デバッグ用) ---

from datasets import Dataset # Datasetクラスをインポート (セル2でインポート済みなら不要)

import numpy as np          # (セル2でインポート済みなら不要)

# tokenizer はセル3で定義済みのはず


print("--- Cell 4.1: Overwriting train_dataset and eval_dataset with super simple dummy data ---")


# 必要なグローバル変数の確認

if 'tokenizer' not in globals():

    raise NameError("Global variable 'tokenizer' is not defined. "

                    "Please ensure Cell 3 (Basic/Common Settings) has been executed correctly and defines tokenizer.")


try:

    # ダミーテキストとラベル

    dummy_train_texts = ["this is a training sentence for debug.", "another training sentence here for debug."]

    dummy_train_labels = [0, 1] # ラベルは整数

    dummy_eval_texts = ["a test sentence for debug.", "another one here for eval debug."]

    dummy_eval_labels = [1, 0] # ラベルは整数


    # 手動でトークナイズ (訓練データ用)

    encoded_train_inputs = tokenizer(dummy_train_texts, padding="max_length", truncation=True, max_length=32, return_tensors="pt")

    dummy_train_data_dict = {

        'input_ids': encoded_train_inputs['input_ids'].tolist(),
```

```
    'attention_mask': encoded_train_inputs['attention_mask'].tolist(),
    'labels': dummy_train_labels # Trainerは 'labels' を期待
}
if 'token_type_ids' in encoded_train_inputs: # BERT系なら token_type_ids も
    dummy_train_data_dict['token_type_ids'] = encoded_train_inputs['token_type_ids'].tolist()

# グローバル変数を上書き
train_dataset = Dataset.from_dict(dummy_train_data_dict)

# 手動でトークナイズ (評価データ用)
encoded_eval_inputs = tokenizer(dummy_eval_texts, padding="max_length", truncation=True, max_length=32, return_tensors="pt")
dummy_eval_data_dict = {
    'input_ids': encoded_eval_inputs['input_ids'].tolist(),
    'attention_mask': encoded_eval_inputs['attention_mask'].tolist(),
    'labels': dummy_eval_labels # Trainerは 'labels' を期待
}
if 'token_type_ids' in encoded_eval_inputs:
    dummy_eval_data_dict['token_type_ids'] = encoded_eval_inputs['token_type_ids'].tolist()

# グローバル変数を上書き
eval_dataset = Dataset.from_dict(dummy_eval_data_dict)

print(f"Super simple dummy train_dataset created and assigned. Samples: {len(train_dataset)}, Columns: {train_dataset.column_names}")
print(f"Super simple dummy eval_dataset created and assigned. Samples: {len(eval_dataset)}, Columns: {eval_dataset.column_names}")
```

```
    if len(eval_dataset) > 0:
        print(f"First sample of new eval_dataset (dummy): {eval_dataset[0]}")

except Exception as e:
    print(f"Error creating super simple dummy dataset in Cell 4.1: {e}")
    import traceback
    traceback.print_exc()
    raise

print("--- Cell 4.1: Dummy dataset assignment complete ---")



import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import PreTrainedModel, AutoModel, AutoConfig, Trainer
from transformers.modeling_outputs import SequenceClassifierOutput
from torch.nn import CrossEntropyLoss

# --- モジュール1: 意味アテンション機構 (SemanticConceptModule) ---
class SemanticConceptModule(nn.Module):
    def __init__(self, input_dim, output_dim,
                 num_concepts=32, concept_dim=128,
                 model_config=None,
                 module_specific_config=None
                 ):
        super().__init__()
```

```
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.num_concepts = num_concepts
        self.concept_dim = concept_dim

        self.concept_prototypes = nn.Parameter(torch.randn(self.num_concepts, self.concept_dim))
        nn.init.xavier_uniform_(self.concept_prototypes)

        self.hidden_to_concept_proj = nn.Linear(self.input_dim, self.concept_dim)
        self.concepts_to_integrate_proj = nn.Linear(self.concept_dim, self.input_dim)
        self.activation = nn.ReLU()

        if self.input_dim != self.output_dim:
            self.final_projection = nn.Linear(self.input_dim, self.output_dim)

    def forward(self, hidden_states, attention_mask=None, **kwargs):
        projected_hidden = self.activation(self.hidden_to_concept_proj(hidden_states))
        attention_scores = torch.matmul(projected_hidden, self.concept_prototypes.t())

        if attention_mask is not None:
            expanded_attention_mask = attention_mask.unsqueeze(-1).float()
            attention_scores = attention_scores.masked_fill(expanded_attention_mask == 0, -1e9)

        attention_weights = F.softmax(attention_scores, dim=-1)
        contextual_concepts = torch.matmul(attention_weights, self.concept_prototypes)
        projected_contextual_concepts =
self.activation(self.concepts_to_integrate_proj(contextual_concepts))
```

```
        fused_hidden_states = hidden_states + projected_contextual_concepts

        if self.input_dim != self.output_dim:
            output = self.final_projection(fused_hidden_states)
        else:
            output = fused_hidden_states

        return {"last_hidden_state": output, "attention_weights": attention_weights}

    def get_output_dim(self):
        return self.output_dim

# --- ★★★ 新しいモジュール: 直接的非可換処理モジュール (DirectNonCommutativeModule) ★★★ ---
class DirectNonCommutativeModule(nn.Module):
    def __init__(self, input_dim, output_dim, # このモジュールの主要な入出力次元
                 dropout=0.2, # モジュール固有のドロップアウト率
                 model_config=None, # ベースモデルのconfig (オプション)
                 module_specific_config=None # このモジュール特有の設定 (オプション)
                 ):
        super().__init__()
        self.input_dim = input_dim  # ベースモデルからのhidden_statesの次元 D_hidden
        self.output_dim = output_dim # このモジュール全体の出力次元

        # 非可換処理器: 入力は input_dim * 2 (隣接する2つのhidden_stateを結合)
        # 出力次元はこのモジュールの output_dim と一致させる
```

```
        internal_processing_dim = module_specific_config.get("internal_processing_dim", input_dim * 2) # 中間層の次元


        self.non_commutative_processor = nn.Sequential(
            nn.Linear(input_dim * 2, internal_processing_dim),
            nn.LayerNorm(internal_processing_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            # 出力は、非可換効果の表現として、例えば input_dim にする
            nn.Linear(internal_processing_dim, input_dim)
        )
        self.lambda_param = nn.Parameter(torch.tensor(0.5)) # 学習可能な重み


        # 元のhidden_statesと統合後の最終的な出力次元が異なる場合の射影層
        if self.input_dim != self.output_dim: # input_dim は非可換効果の次元 (input_dimのまま)
            self.final_integration_projection = nn.Linear(self.input_dim, self.output_dim)


        print(f"DirectNonCommutativeModule initialized: input_dim={input_dim}, output_dim={output_dim}")


    def forward(self, hidden_states, attention_mask=None, **kwargs):
        batch_size, seq_len, local_input_dim = hidden_states.shape # local_input_dim は self.input_dim と同じはず
        device = hidden_states.device


        if seq_len < 2:
```

```
        print("Warning: Seq_len < 2 in DirectNonCommutativeModule. Returning original hidden_states (or projecting if output_dim differs).")
        if self.input_dim == self.output_dim:
            return {"last_hidden_state": hidden_states, "non_commutative_effects_map": None}
        else:
            if not hasattr(self, 'final_integration_projection'): # 念のため
                # output_dim に合わせるためのフォールバック射影 (input_dim -> output_dim)
                # このケースでは非可換効果は加味されない
                temp_proj = nn.Linear(self.input_dim, self.output_dim).to(device)
                return {"last_hidden_state": temp_proj(hidden_states), "non_commutative_effects_map": None}
            return {"last_hidden_state": self.final_integration_projection(hidden_states), "non_commutative_effects_map": None}

    non_commutative_effects_at_each_step = torch.zeros_like(hidden_states)

    for i in range(seq_len - 1):
        current_h = hidden_states[:, i, :]
        next_h = hidden_states[:, i + 1, :]

        forward_concat = torch.cat([current_h, next_h], dim=-1)
        backward_concat = torch.cat([next_h, current_h], dim=-1)

        forward_processed = self.non_commutative_processor(forward_concat)
        backward_processed = self.non_commutative_processor(backward_concat)

        non_comm_effect = self.lambda_param * (forward_processed - backward_processed)
```

```
            # 計算された非可換効果を、元のシーケンスの対応する位置 (i番目) に加算
            # (あるいは、i番目とi+1番目の両方に影響を与えるなど、統合方法は要検討)
            # ここでは、i番目のトークンに効果を蓄積する例
            non_commutative_effects_at_each_step[:, i, :] += non_comm_effect
            # もし (h_i, h_{i+1}) の関係性が h_i と h_{i+1} の両方に影響すると考えるなら、
            # non_commutative_effects_at_each_step[:, i+1, :] にも何らかの形で加算/代入する

        # 元のhidden_statesに非可換効果を加算
        fused_hidden_states = hidden_states + non_commutative_effects_at_each_step

        # 最終的な出力次元への射影 (必要な場合)
    if self.input_dim != self.output_dim: # ここでのinput_dimは非可換効果を加算後の次元（つまり元のinput_dimと同じ）
            output = self.final_integration_projection(fused_hidden_states)
        else:
            output = fused_hidden_states

        return {"last_hidden_state": output, "non_commutative_effects_map": non_commutative_effects_at_each_step}

    def get_output_dim(self):
        return self.output_dim

# --- EnhancedTransformerForSequenceClassification (DirectNonCommutativeModule を組み込み) ---
class EnhancedTransformerForSequenceClassification(PreTrainedModel):
```

```
    def __init__(self, hf_config, base_model_name, num_labels, tech_flags=None,
module_configs=None):
        super().__init__(hf_config)
        self.num_labels = num_labels
        self.config = hf_config

        self.base_model_prefix = self.config.model_type
        core_model = AutoModel.from_pretrained(base_model_name, config=self.config)
        setattr(self, self.base_model_prefix, core_model)

        current_dim = self.config.hidden_size
        self.tech_modules = nn.ModuleDict()
        self.tech_flags = tech_flags if tech_flags is not None else {}
        self.module_configs = module_configs if module_configs is not None else {}

        # モジュール1: 意味アテンション機構 (SemanticConceptModule)
        if self.tech_flags.get("use_semantic_attention_module", False):
            scm_config = self.module_configs.get("semantic_attention_module", {})
            module1_output_dim = scm_config.get("output_dim", current_dim)
            self.tech_modules['semantic_attention_module'] = SemanticConceptModule(
                input_dim=current_dim,
                output_dim=module1_output_dim,
                num_concepts=scm_config.get("num_concepts", 32),
                concept_dim=scm_config.get("concept_dim", 128),
                model_config=self.config
            )
            current_dim = module1_output_dim # モジュールの出力次元で更新
```

```
            print("SemanticAttentionModule enabled.")


        # ★★★ モジュールX: 直接的非可換処理モジュール (DirectNonCommutativeModule) ★★★
        # SemanticConceptModule とは排他的に有効化されることを想定
        elif self.tech_flags.get("use_direct_non_commutative_module", False): # elif に変更
            dnc_config = self.module_configs.get("direct_non_commutative_module", {})
            moduleX_output_dim = dnc_config.get("output_dim", current_dim) # 通常はcurrent_dimと同じ

            self.tech_modules['direct_non_commutative_module'] = DirectNonCommutativeModule(
                input_dim=current_dim, # ベースモデルの出力次元
                output_dim=moduleX_output_dim,
                dropout=dnc_config.get("dropout", 0.2),
                module_specific_config=dnc_config # モジュール固有の設定を渡す
            )
            current_dim = moduleX_output_dim # モジュールの出力次元で更新
            print("DirectNonCommutativeModule enabled.")

        self.dropout = nn.Dropout(self.config.hidden_dropout_prob if hasattr(self.config, 'hidden_dropout_prob') else 0.1)
        self.classifier = nn.Linear(current_dim, self.num_labels)
        print(f"EnhancedTransformerForSequenceClassification initialized. Classifier input dim: {current_dim}")

    def forward(
        self,
```

```
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        labels=None,
        target_semantic_labels=None,
        return_dict=None,
        **kwargs
    ):
        print(f"--- EnhancedTransformer.forward CALLED (is_training: {self.training}) ---")
        # ... (入力確認のデバッグプリントは省略、必要なら追加してください) ...

        base_model_kwargs = {
            k: v for k, v in kwargs.items()
            if k in ["position_ids", "head_mask", "inputs_embeds", "output_attentions",
"output_hidden_states"]
        }

        transformer_outputs = self.base_model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            return_dict=True,
            **base_model_kwargs
        )
        hidden_states = transformer_outputs.last_hidden_state

        module_outputs_dict = {}
```

```
        if self.tech_flags.get("use_semantic_attention_module", False) and
'semantic_attention_module' in self.tech_modules:
            scm_results = self.tech_modules['semantic_attention_module'](
                hidden_states,
                attention_mask=attention_mask
            )
            hidden_states = scm_results["last_hidden_state"]
            module_outputs_dict["attention_weights"] = scm_results["attention_weights"]

        # ★★★ DirectNonCommutativeModule の呼び出し ★★★
        elif self.tech_flags.get("use_direct_non_commutative_module", False) and
'direct_non_commutative_module' in self.tech_modules:
            dnc_results = self.tech_modules['direct_non_commutative_module'](
                hidden_states,
                attention_mask=attention_mask # このモジュールでは使わないかもしれないが渡しておく
            )
            hidden_states = dnc_results["last_hidden_state"]
            # module_outputs_dict["non_commutative_effects_map"] =
dnc_results["non_commutative_effects_map"] # 必要なら

        pooled_output = hidden_states[:, 0]
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)

        loss = None
        if labels is not None:
```

```
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
            print(f"  Forward CALCULATED loss: {loss.item() if loss is not None else 'None'}")
        else:
            print(f"  Forward did NOT calculate loss (labels were None).")

        print(f"--- EnhancedTransformer.forward RETURNING (explicit (loss, logits) tuple for final debug) ---")

        other_outputs_tuple_elements = ()
        if hasattr(transformer_outputs, 'hidden_states') and transformer_outputs.hidden_states is not None:
            other_outputs_tuple_elements += (transformer_outputs.hidden_states,)
        if hasattr(transformer_outputs, 'attentions') and transformer_outputs.attentions is not None:
            other_outputs_tuple_elements += (transformer_outputs.attentions,)
        # (モジュール固有の出力を追加する場合はここに)

        if loss is not None:
            print(f"  Returning loss: {loss.item()}")
            print(f"  Returning logits (tensor) shape: {logits.shape}")
            return (loss, logits) + other_outputs_tuple_elements
        else:
            print(f"  Returning logits (tensor) shape: {logits.shape} (loss is None)")
            return (None, logits) + other_outputs_tuple_elements

# --- DebugTrainer クラスの定義 (変更なし) ---
```

```
class DebugTrainer(Trainer):
    def prediction_step(
        self,
        model: nn.Module,
        inputs: dict[str, torch.Tensor | nn.utils.rnn.PackedSequence],
        prediction_loss_only: bool,
        ignore_keys: list[str] | None = None,
    ) -> tuple[torch.Tensor | None, torch.Tensor | None, torch.Tensor | None]:

        print(f"--- DebugTrainer.prediction_step CALLED ---")
        # ... (以前のDebugTrainerのコードはそのまま) ...
        print(f"  prediction_loss_only: {prediction_loss_only}")
        if inputs is not None and isinstance(inputs, dict):
            print(f"  Input keys: {list(inputs.keys())}")
            if "labels" in inputs:
                print(f"  'labels' found in inputs. Shape: {inputs['labels'].shape}")
            else:
                print(f"  Warning: 'labels' NOT found in inputs for prediction_step.")
        else:
            print(f"  Warning: inputs to prediction_step is None or not a dict.")

        model.eval()
        with torch.no_grad():
            model_outputs = model(**inputs)

        loss = model_outputs[0] if model_outputs[0] is not None else None
```

```
        logits = model_outputs[1] if len(model_outputs) > 1 and model_outputs[1] is not None else None

        labels_out = inputs.get("labels")


        print(f"--- DebugTrainer.prediction_step after manual model call ---")

        if loss is not None:

            loss_val_to_print = loss.item() if isinstance(loss, torch.Tensor) and loss.numel() == 1 else loss

            loss_shape_to_print = loss.shape if isinstance(loss, torch.Tensor) else type(loss)

            print(f"  Manually extracted loss shape: {loss_shape_to_print}, loss value: {loss_val_to_print}")

        else:

            print(f"  Manually extracted loss is None.")

        if logits is not None:

            if isinstance(logits, torch.Tensor):

                print(f"  Manually extracted logits shape: {logits.shape}")

            else:

                print(f"  Manually extracted logits type is {type(logits)}. Value: {logits}")

        else:

            print(f"  Manually extracted logits is None.")

        if labels_out is not None:

            print(f"  Extracted labels_out shape: {labels_out.shape}")

        else:

            print(f"  Extracted labels_out is None (or not in inputs).")


        if prediction_loss_only:

            return (loss, None, None)

        return loss, logits, labels_out
```

```
print("Cell 5: Custom model classes (including DirectNonCommutativeModule and updated EnhancedTransformer) and DebugTrainer defined.")


# --- セル6: アブレーション実験の構成定義と実行ループ ---


# --- グローバル変数 (セル3, 4, 5 で定義されているはずのもの) の確認 ---
required_globals_cell6 = [
    'hf_model_config', 'BASE_MODEL_NAME', 'NUM_LABELS', 'OUTPUT_DIR_BASE',
    'LEARNING_RATE', 'BATCH_SIZE', 'NUM_EPOCHS',
    'EnhancedTransformerForSequenceClassification',
    'train_dataset', 'eval_dataset', 'tokenizer', 'compute_metrics',
    'Dataset',
    'DebugTrainer'
]
for var_name in required_globals_cell6:
    if var_name not in globals():
        raise NameError(
            f"Global variable '{var_name}' is not defined. "
            f"Please ensure all prerequisite cells have been executed correctly and define this variable."
        )
print("All required global variables for Cell 6 are defined.")


# --- 実験構成の定義 ---
experiment_configurations = [
    {
        "name": "0_baseline_transformer",
```

```
    "tech_flags": {},
    "module_configs": {}
  },
  {
    "name": "1_semantic_attention_end_to_end",
    "tech_flags": {"use_semantic_attention_module": True}, # SemanticConceptModuleを有効化
    "module_configs": {
      "semantic_attention_module": { # SemanticConceptModule の設定
        "num_concepts": 32,
        "concept_dim": 128,
        "output_dim": hf_model_config.hidden_size # 出力次元はベースと同じ
      }
    }
  },
  { # ★★★ 新しい実験構成の追加 ★★★
    "name": "2_direct_non_commutative",
    "tech_flags": {"use_direct_non_commutative_module": True}, # DirectNonCommutativeModuleを有効化
    "module_configs": {
      "direct_non_commutative_module": { # DirectNonCommutativeModule の設定
        "output_dim": hf_model_config.hidden_size, # 出力次元はベースと同じ
        "dropout": 0.2, # モジュール内のドロップアウト率
        "internal_processing_dim": hf_model_config.hidden_size # 例: 中間層の次元をhidden_sizeと同じに
      }
    }
```

```
    }
]

all_results = {}
all_trained_models = {}

print(f"Generated {len(experiment_configurations)} experiment configurations.")

# --- カラム名リネーム処理 (念のためループ直前にも配置) ---
# (この部分はセル4で確実に行われていれば、ここでは実際には何も起こらないはずです)
if 'train_dataset' in globals() and isinstance(train_dataset, Dataset) and \
   'label' in train_dataset.column_names and 'labels' not in train_dataset.column_names:
    print("Cell 6 (pre-loop): Renaming 'label' column to 'labels' in train_dataset...")
    train_dataset = train_dataset.rename_column("label", "labels")
if 'eval_dataset' in globals() and isinstance(eval_dataset, Dataset) and \
   'label' in eval_dataset.column_names and 'labels' not in eval_dataset.column_names:
    print("Cell 6 (pre-loop): Renaming 'label' column to 'labels' in eval_dataset...")
    eval_dataset = eval_dataset.rename_column("label", "labels")
print(f"  Train dataset column names (pre-loop check): {train_dataset.column_names if 'train_dataset' in globals() else 'Not defined'}")
print(f"  Eval dataset column names (pre-loop check): {eval_dataset.column_names if 'eval_dataset' in globals() else 'Not defined'}")
# --- ここまでリネーム処理 ---

# --- 実験ループ ---
for exp_config_setting in experiment_configurations:
```

```
config_name = exp_config_setting["name"]
tech_flags = exp_config_setting["tech_flags"]
module_cfgs = exp_config_setting["module_configs"]

print(f"\n--- Running Experiment: {config_name} ---")
print(f"Technology Flags: {tech_flags}")
print(f"Module Configs: {module_cfgs}")

model = EnhancedTransformerForSequenceClassification(
    hf_config=hf_model_config,
    base_model_name=BASE_MODEL_NAME,
    num_labels=NUM_LABELS,
    tech_flags=tech_flags,
    module_configs=module_cfgs
)

current_output_dir = f"{OUTPUT_DIR_BASE}{config_name}"

# --- TrainingArguments (本格運用向けだが、NUM_EPOCHS=1でまずテスト) ---
training_args = TrainingArguments(
    output_dir=current_output_dir,
    learning_rate=LEARNING_RATE, # セル3で定義
    per_device_train_batch_size=BATCH_SIZE, # セル3で定義 (例: 1)
    per_device_eval_batch_size=BATCH_SIZE,  # セル3で定義 (例: 1)
    num_train_epochs=NUM_EPOCHS,            # セル3で定義 (例: 1)
    weight_decay=0.01,
```

```
        evaluation_strategy="epoch",
        logging_strategy="epoch",
        save_strategy="epoch",

        load_best_model_at_end=True,
        metric_for_best_model="accuracy",

        report_to="none",
        do_eval=True,
        remove_unused_columns=False,  # カラム削除しない (安全のため)
        # save_total_limit=1, # 保存するチェックポイントの最大数を1に制限 (ディスク節約のため)
    )

    trainer = DebugTrainer( # 引き続き DebugTrainer を使用
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer,
        compute_metrics=compute_metrics,  # compute_metrics (セル4で定義、デバッグprint有効)
    )

    print(f"Starting training for {config_name}...")
    try:
```

```
        trainer.train()
        print(f"Training finished for {config_name}.")

        print(f"Starting final evaluation for {config_name} (with best model)...")
        eval_results = trainer.evaluate()
        print(f"Evaluation results for {config_name}: {eval_results}")

        all_results[config_name] = eval_results

    except Exception as e:
        print(f"!!! An error occurred during training or evaluation for {config_name}: {e} !!!")
        import traceback
        traceback.print_exc()
        break # エラー発生時はループを抜ける

print("\n\n--- Experiment Summary ---")
if all_results:
    for name, res in sorted(all_results.items()):
        accuracy = res.get('eval_accuracy', 'N/A')
        loss = res.get('eval_loss', 'N/A')
        print(f"Configuration: {name:<50} Accuracy: {accuracy:<10} Loss: {loss:<10}")
else:
    print("No results to summarize. Experiments might have failed or were not run.")

print("\nAll experiments complete (or stopped due to an error).")
```

実験コード2

```
# セル1 (セットアップとカーネル再起動用)
# ---------------------------------
# 関連ライブラリを一度アンインストールしてクリーンな状態にする
!pip uninstall -y transformers accelerate peft sentence-transformers datasets -q

# 指定バージョンでライブラリをインストール
!pip install transformers==4.36.2 -q
!pip install datasets -q
!pip install peft==0.7.1 -q
!pip install accelerate==0.25.0 -q  # <--- accelerateのバージョンを0.25.0に固定 (または0.26.1)

# 変更を適用するためにカーネルを再起動
import os
os.kill(os.getpid(), 9)
```

```
# --- セル2: ライブラリのインポート ---
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F # SemanticConceptModule で F.softmax を使用するため

from transformers import (
    AutoModelForSequenceClassification,
    AutoTokenizer,
    Trainer,
    TrainingArguments,
    AutoConfig,
    PreTrainedModel,
    AutoModel
)
from transformers.modeling_outputs import SequenceClassifierOutput # モデルの出力形式用
from torch.nn import CrossEntropyLoss # 損失関数用

from datasets import load_dataset, Dataset # ★ Dataset クラスをインポート

import numpy as np
from sklearn.metrics import accuracy_score # 評価指標計算用
import itertools # 実験構成生成用 (今回は手動定義なので厳密には不要ですが、将来的に使う可能性も)

# (もし他に必要な標準ライブラリがあればここに追加してください)
# import os # (セル1で使用済みなので、ここでは必須ではない)
# import json
```

```
# import time
# import datetime

# (もしグラフ描画などを行う場合は、matplotlibなどもここでインポート)
# import matplotlib.pyplot as plt
# import seaborn as sns
# import pandas as pd

print("Cell 2: Libraries imported successfully.")
print(f"  PyTorch version: {torch.__version__}")
try:
    import transformers
    print(f"  Transformers version: {transformers.__version__}")
    import datasets
    print(f"  Datasets version: {datasets.__version__}")
    import accelerate
    print(f"  Accelerate version: {accelerate.__version__}")
    import peft
    print(f"  PEFT version: {peft.__version__}")
except ImportError:
    print("Warning: Could not print all Hugging Face library versions. Ensure they are installed.")


# --- セル3: 基本設定・共通パラメータ ---

from transformers import AutoTokenizer, AutoConfig # 必要なものをインポート (セル2でインポート済みなら重複を避けてもOK)
```

```
# --- 基本的な設定値 ---
BASE_MODEL_NAME = "bert-base-uncased"  # ベースとなる事前学習済みモデルの名前
NUM_LABELS = 2                # 分類タスクのラベル数 (例: IMDbならポジティブ/ネガティブの2)
OUTPUT_DIR_BASE = "./experiment_results/" # 実験結果の出力先ベースディレクトリ

# --- 訓練に関するハイパーパラメータ ---
LEARNING_RATE = 2e-5          # 学習率
BATCH_SIZE = 1                # バッチサイズ (GPUメモリやデータに応じて調整)
                              # CPU実行の場合は小さめ(例: 1や2)にしないと非常に遅い可能性があります
NUM_EPOCHS = 1                # 訓練エポック数 (最初は1などでテストし、問題なければ増やす)

# --- トークナイザーとモデル設定のロード ---
try:
    print(f"Loading tokenizer for {BASE_MODEL_NAME}...")
    tokenizer = AutoTokenizer.from_pretrained(BASE_MODEL_NAME)
    print(f"Tokenizer for {BASE_MODEL_NAME} loaded successfully.")
except Exception as e:
    print(f"Error loading tokenizer for {BASE_MODEL_NAME}: {e}")
    raise # エラーが発生したら処理を停止

try:
    print(f"Loading Hugging Face model config for {BASE_MODEL_NAME}...")
```

```
    hf_model_config = AutoConfig.from_pretrained(
        BASE_MODEL_NAME,
        num_labels=NUM_LABELS
        # 必要に応じて他のAutoConfigのパラメータもここで設定可能
        # 例: finetuning_task='text-classification' など
    )
    print(f"Model config for {BASE_MODEL_NAME} loaded successfully.")
except Exception as e:
    print(f"Error loading model config for {BASE_MODEL_NAME}: {e}")
    raise

print("\n--- Cell 3: Basic/Common Settings Defined ---")
print(f"BASE_MODEL_NAME: {BASE_MODEL_NAME}")
print(f"NUM_LABELS: {NUM_LABELS}")
print(f"OUTPUT_DIR_BASE: {OUTPUT_DIR_BASE}")
print(f"LEARNING_RATE: {LEARNING_RATE}")
print(f"BATCH_SIZE: {BATCH_SIZE}")
print(f"NUM_EPOCHS: {NUM_EPOCHS}")
print(f"Tokenizer type: {type(tokenizer).__name__}")
print(f"HF Model Config type: {type(hf_model_config).__name__}")



# --- セル4: データセット準備 と compute_metrics 関数の定義 (カラム名修正をここで行う最終版) ---

from datasets import load_dataset
```

```
from sklearn.metrics import accuracy_score

import numpy as np

import torch


# --- グローバル変数 (セル3で定義されているはずのもの) の確認 ---

if 'BASE_MODEL_NAME' not in globals():

    raise NameError("Global variable 'BASE_MODEL_NAME' is not defined. Please ensure Cell 3 has been executed.")

if 'tokenizer' not in globals():

    raise NameError("Global variable 'tokenizer' is not defined. Please ensure Cell 3 has been executed.")


print(f"Starting dataset preparation using BASE_MODEL_NAME: {BASE_MODEL_NAME} and tokenizer: {type(tokenizer).__name__}")


# 1. データセットのロード

try:

    dataset_dict = load_dataset("imdb") # DatasetDictオブジェクトとしてロード

    print("IMDb dataset loaded successfully.")

except Exception as e:

    print(f"Error loading IMDb dataset: {e}")

    raise


# 2. トークナイズ関数の定義

def tokenize_function(examples):

    return tokenizer(examples["text"], padding="max_length", truncation=True, max_length=256)
```

```
# 3. データセットのトークナイズ (必要なsplitのみを対象にすることも検討)
try:
    print("Tokenizing datasets...")
    # 'train' と 'test' splitのみをトークナイズする例 (unsupervisedは使わない想定)
    # もし他のsplitも使うなら、dataset_dict全体をmapするか、個別に処理
    tokenized_train = dataset_dict["train"].map(tokenize_function, batched=True, remove_columns=["text"])
    tokenized_test = dataset_dict["test"].map(tokenize_function, batched=True, remove_columns=["text"])
    print("Tokenization complete for train and test splits.")
except Exception as e:
    print(f"Error during tokenization: {e}")
    raise

# 4. 訓練用・評価用データセットの準備とカラム名修正
try:
    # 'label' カラムを 'labels' にリネーム
    if 'label' in tokenized_train.column_names and 'labels' not in tokenized_train.column_names:
        print("Renaming 'label' to 'labels' in tokenized_train...")
        final_tokenized_train = tokenized_train.rename_column("label", "labels")
    else:
        final_tokenized_train = tokenized_train
        if 'labels' not in final_tokenized_train.column_names:
            print("Warning: 'labels' column expected but not found in final_tokenized_train.")

    if 'label' in tokenized_test.column_names and 'labels' not in tokenized_test.column_names:
```

```
        print("Renaming 'label' to 'labels' in tokenized_test...")
        final_tokenized_test = tokenized_test.rename_column("label", "labels")
    else:
        final_tokenized_test = tokenized_test
        if 'labels' not in final_tokenized_test.column_names:
            print("Warning: 'labels' column expected but not found in final_tokenized_test.")

    # グローバル変数として train_dataset と eval_dataset を定義
    # (デバッグ用に縮小したサイズ)
    train_dataset = final_tokenized_train.shuffle(seed=42).select(range(20))
    eval_dataset = final_tokenized_test.shuffle(seed=42).select(range(10))

    print(f"Train dataset created. Number of samples: {len(train_dataset)}")
    print(f"  Train dataset FINAL column names: {train_dataset.column_names}")
    print(f"Evaluation dataset created. Number of samples: {len(eval_dataset)}")
    print(f"  Eval dataset FINAL column names: {eval_dataset.column_names}")

    if len(eval_dataset) > 0:
        print(f"First sample of EVAL_dataset (for column check): {eval_dataset[0]}")

except Exception as e:
    print(f"Error creating or renaming train/eval datasets: {e}")
    raise

# 5. 評価指標計算関数の定義 (デバッグプリント有効)
def compute_metrics(eval_pred):
```

```
    logits, labels = eval_pred
    if labels is None or len(labels) == 0:
        print("--- Inside compute_metrics: Received no labels or labels is None. Returning empty metrics. ---")
        return {}
    predictions = np.argmax(logits, axis=-1)
    accuracy = accuracy_score(labels, predictions)
    metrics_dict = {"eval_accuracy": accuracy}
    print(f"--- Inside compute_metrics ---")
    print(f"  eval_pred logits type: {type(logits)}, labels type: {type(labels)}")
    if hasattr(logits, 'shape'): print(f"  eval_pred logits shape: {logits.shape}")
    if hasattr(labels, 'shape'): print(f"  eval_pred labels shape: {labels.shape}")
    print(f"  Predictions example (first 5): {predictions[:5]}")
    print(f"  Labels example (first 5): {labels[:5]}")
    print(f"  Calculated accuracy: {accuracy}")
    print(f"  Returning metrics_dict: {metrics_dict}")
    return metrics_dict

print("\nDataset preparation cell (Cell 4) complete.")
print(f"Defined global variables: 'train_dataset' (size: {len(train_dataset)}), 'eval_dataset' (size: {len(eval_dataset)}), 'compute_metrics'")
if callable(globals().get('compute_metrics')):
    print("'compute_metrics' function is defined and callable.")
else:
    print("Warning: 'compute_metrics' function is NOT defined or not callable.")

# --- セル4.1: ダミーデータセットによる上書き (デバッグ用) ---
from datasets import Dataset # Datasetクラスをインポート (セル2でインポート済みなら不要)
```

```
import numpy as np          # (セル2でインポート済みなら不要)

# tokenizer はセル3で定義済みのはず


print("--- Cell 4.1: Overwriting train_dataset and eval_dataset with super simple dummy data ---")


# 必要なグローバル変数の確認

if 'tokenizer' not in globals():

    raise NameError("Global variable 'tokenizer' is not defined. "

                    "Please ensure Cell 3 (Basic/Common Settings) has been executed correctly and defines tokenizer.")


try:

    # ダミーテキストとラベル

    dummy_train_texts = ["this is a training sentence for debug.", "another training sentence here for debug."]

    dummy_train_labels = [0, 1] # ラベルは整数

    dummy_eval_texts = ["a test sentence for debug.", "another one here for eval debug."]

    dummy_eval_labels = [1, 0] # ラベルは整数


    # 手動でトークナイズ (訓練データ用)

    encoded_train_inputs = tokenizer(dummy_train_texts, padding="max_length", truncation=True, max_length=32, return_tensors="pt")

    dummy_train_data_dict = {

        'input_ids': encoded_train_inputs['input_ids'].tolist(),

        'attention_mask': encoded_train_inputs['attention_mask'].tolist(),

        'labels': dummy_train_labels # Trainerは 'labels' を期待

    }
```

```
    if 'token_type_ids' in encoded_train_inputs: # BERT系なら token_type_ids も
        dummy_train_data_dict['token_type_ids'] = encoded_train_inputs['token_type_ids'].tolist()

    # グローバル変数を上書き
    train_dataset = Dataset.from_dict(dummy_train_data_dict)

    # 手動でトークナイズ (評価データ用)
    encoded_eval_inputs = tokenizer(dummy_eval_texts, padding="max_length", truncation=True, max_length=32, return_tensors="pt")
    dummy_eval_data_dict = {
        'input_ids': encoded_eval_inputs['input_ids'].tolist(),
        'attention_mask': encoded_eval_inputs['attention_mask'].tolist(),
        'labels': dummy_eval_labels # Trainerは 'labels' を期待
    }
    if 'token_type_ids' in encoded_eval_inputs:
        dummy_eval_data_dict['token_type_ids'] = encoded_eval_inputs['token_type_ids'].tolist()

    # グローバル変数を上書き
    eval_dataset = Dataset.from_dict(dummy_eval_data_dict)

    print(f"Super simple dummy train_dataset created and assigned. Samples: {len(train_dataset)}, Columns: {train_dataset.column_names}")
    print(f"Super simple dummy eval_dataset created and assigned. Samples: {len(eval_dataset)}, Columns: {eval_dataset.column_names}")
    if len(eval_dataset) > 0:
        print(f"First sample of new eval_dataset (dummy): {eval_dataset[0]}")
```

```
except Exception as e:
    print(f"Error creating super simple dummy dataset in Cell 4.1: {e}")
    import traceback
    traceback.print_exc()
    raise

print("--- Cell 4.1: Dummy dataset assignment complete ---")



import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import PreTrainedModel, AutoModel, AutoConfig, Trainer
from transformers.modeling_outputs import SequenceClassifierOutput
from torch.nn import CrossEntropyLoss

# --- ★★★ モジュール: 断絶創発モジュール (DiscontinuityEmergenceModule) ★★★ ---
class DiscontinuityEmergenceModule(nn.Module):
    def __init__(self, input_dim, output_dim, # このモジュールの主要な入出力次元
                 semantic_dim_for_discontinuity=12, # 断絶検出に使う内部的な意味表現の次元
                 emergent_label_dim=24,           # 創発意味ラベルの次元 (例: semantic_dim * 2)
                 dropout=0.2,
                 model_config=None, # ベースモデルのconfig (オプション)
                 module_specific_config=None # このモジュール特有の設定 (オプション)
                 ):
        super().__init__()
```

```
        self.input_dim = input_dim   # 前の階層からの特徴量の次元 (例: 768)
        self.output_dim = output_dim # このモジュール全体の出力次元 (例: 768)
        self.semantic_dim_for_discontinuity = module_specific_config.get("semantic_dim_for_discontinuity", semantic_dim_for_discontinuity)
        self.emergent_label_dim = module_specific_config.get("emergent_label_dim", emergent_label_dim)
        internal_dropout = module_specific_config.get("dropout", dropout)

        # 1. 断絶検出のための特徴変換器 (入力全体を一度変換)
        # input_features を断絶検出しやすい表現 (semantic_dim_for_discontinuity) に変換
        self.feature_to_semantic_for_discontinuity = nn.Sequential(
            nn.Linear(input_dim, input_dim // 2),
            nn.ReLU(),
            nn.Linear(input_dim // 2, self.semantic_dim_for_discontinuity),
            nn.Tanh()
        )

        # 2. 断絶検出器 (Discontinuity Detector)
        # 隣接する2つの意味表現を比較して断絶スコアを出す
        self.discontinuity_detector = nn.Sequential(
            nn.Linear(self.semantic_dim_for_discontinuity * 2, 64),
            nn.ReLU(),
            nn.Dropout(internal_dropout),
            nn.Linear(64, 1),
            nn.Sigmoid()
        )
```

```
        # 3. 創発意味ラベル生成器 (Emergent Label Generator)
        # 断絶前後の意味表現 (semantic_dim_for_discontinuity * 2) を入力とする例
        self.emergent_label_generator = nn.Sequential(
            nn.Linear(self.semantic_dim_for_discontinuity * 2, input_dim // 4), # お客様のFourthOrderModel参考
            nn.LayerNorm(input_dim // 4),
            nn.ReLU(),
            nn.Dropout(internal_dropout),
            nn.Linear(input_dim // 4, self.emergent_label_dim),
            nn.Tanh()
        )

        # 4. 統合層 (元のhidden_statesと創発ラベルを統合)
        # 元のinput_dim と emergent_label_dim を結合し、output_dimに射影する例
        # または、創発ラベルをinput_dimに射影して加算するなど、設計による
        # ここでは、元の特徴量に「創発ラベルの影響を加えたもの」を生成し、次元をoutput_dimに合わせる
        # 簡単のため、創発ラベルはシーケンス全体で1つとし、それを各トークンにブロードキャストして加算後、射影
        if self.input_dim + self.emergent_label_dim != self.output_dim : # 結合して射影する場合
            self.integration_projection = nn.Linear(self.input_dim + self.emergent_label_dim, self.output_dim)
        elif self.input_dim != self.output_dim: # 加算後に射影する場合 (創発ラベルがinput_dimに射影される前提)
            self.integration_projection = nn.Linear(self.input_dim, self.output_dim)
```

```
        print(f"DiscontinuityEmergenceModule initialized: input_dim={input_dim},
output_dim={output_dim}, "
              f"semantic_dim_disc={self.semantic_dim_for_discontinuity},
emergent_dim={self.emergent_label_dim}")


    def forward(self, hidden_states, attention_mask=None, **kwargs):
        batch_size, seq_len, _ = hidden_states.shape
        device = hidden_states.device

        # 1. 断絶検出のための意味表現に変換
        # (batch_size, seq_len, semantic_dim_for_discontinuity)
        semantic_features_for_discontinuity =
self.feature_to_semantic_for_discontinuity(hidden_states)

        # 2. 断絶スコアの計算 (シーケンスの各隣接点に対して)
        discontinuity_scores_list = []
        if seq_len > 1:
            # テンソル操作で効率化 (ループを避ける)
            # (batch_size, seq_len-1, semantic_dim_for_discontinuity)
            sfd_t = semantic_features_for_discontinuity[:, :-1, :]
            sfd_t_plus_1 = semantic_features_for_discontinuity[:, 1:, :]
            # (batch_size, seq_len-1, semantic_dim_for_discontinuity * 2)
            combined_for_discontinuity = torch.cat([sfd_t, sfd_t_plus_1], dim=-1)
            # (batch_size, seq_len-1, 1)
            discontinuity_scores = self.discontinuity_detector(combined_for_discontinuity)
            discontinuity_scores = discontinuity_scores.squeeze(-1) # (batch_size, seq_len-1)
        else:
```

```
        discontinuity_scores = torch.zeros(batch_size, 0, device=device)


    # 3. 創発意味ラベルの生成 (シーケンスの各隣接点に対して)
    emergent_labels_sequence = torch.zeros(batch_size, 0, self.emergent_label_dim,
device=device) # 初期化
    if seq_len > 1:
        # 上記 combined_for_discontinuity を創発ラベル生成器の入力として再利用する例
        # (batch_size, seq_len-1, emergent_label_dim)
        emergent_labels_sequence = self.emergent_label_generator(combined_for_discontinuity)


    # 4. 元の特徴量との統合
    #   ここでは、各トークンiに、(i-1,i)間と(i,i+1)間の創発ラベルの平均を影響させる、
などの複雑な処理も可能
    #   最も簡単なのは、全創発ラベルの平均/最大を取り、それを全トークンにブロード
キャスト加算
    output_features = hidden_states # まずは元の特徴量
    if emergent_labels_sequence.numel() > 0 and emergent_labels_sequence.shape[1] > 0: # 創発
ラベルが生成された場合
        # 例: 全創発ラベルの平均を計算 (batch_size, emergent_label_dim)
        pooled_emergent_label = emergent_labels_sequence.mean(dim=1)


        # 元のhidden_statesの各トークンに結合して射影するアプローチ
        # (batch_size, seq_len, input_dim + emergent_label_dim)
        expanded_emergent_label = pooled_emergent_label.unsqueeze(1).expand(-1, seq_len, -1)
        combined_features = torch.cat([hidden_states, expanded_emergent_label], dim=-1)
```

```
        if hasattr(self, 'integration_projection'):
            output_features = self.integration_projection(combined_features)
        elif self.input_dim + self.emergent_label_dim == self.output_dim: # 結合だけで次元が合う場合
            output_features = combined_features
        elif self.input_dim == self.output_dim: # 加算モデルで次元が合う場合 (この例では結合なので該当しにくい)
            # project_emergent = nn.Linear(self.emergent_label_dim, self.input_dim).to(device)
            # output_features = hidden_states + project_emergent(pooled_emergent_label).unsqueeze(1)
            print("Warning: Integration strategy for DiscontinuityEmergenceModule needs review if output_dim == input_dim and using concat style.")
            pass # このケースの統合戦略を明確にする必要あり
        else: # フォールバック（何もしない、または単純な射影）
            print(f"Warning: Fallback integration in DiscontinuityEmergenceModule. output_dim might not be as expected.")
            if self.input_dim != self.output_dim and hasattr(self, 'final_projection_fallback'): # 万が一のための射影
                output_features = self.final_projection_fallback(hidden_states)
            else: # 何もしない
                output_features = hidden_states
    elif self.input_dim != self.output_dim : # 創発ラベルがないが、出力次元が異なる場合
        if not hasattr(self, 'final_projection_fallback'):
            self.final_projection_fallback = nn.Linear(self.input_dim, self.output_dim).to(device)
        output_features = self.final_projection_fallback(hidden_states)


    return {
```

```
            "last_hidden_state": output_features,
            "discontinuity_scores": discontinuity_scores,
            "emergent_labels_sequence": emergent_labels_sequence
        }

    def get_output_dim(self):
        return self.output_dim


# --- EnhancedTransformerForSequenceClassification (DiscontinuityEmergenceModule を組み込み) ---
class EnhancedTransformerForSequenceClassification(PreTrainedModel):
    def __init__(self, hf_config, base_model_name, num_labels, tech_flags=None, module_configs=None):
        super().__init__(hf_config)
        self.num_labels = num_labels
        self.config = hf_config

        self.base_model_prefix = self.config.model_type
        core_model = AutoModel.from_pretrained(base_model_name, config=self.config)
        setattr(self, self.base_model_prefix, core_model)

        current_dim = self.config.hidden_size
        self.tech_modules = nn.ModuleDict()
        self.tech_flags = tech_flags if tech_flags is not None else {}
        self.module_configs = module_configs if module_configs is not None else {}
```

```
        # モジュールは排他的に有効化されることを想定 (tech_flagsで制御)
        active_module_name = None

        if self.tech_flags.get("use_semantic_attention_module", False):
            active_module_name = 'semantic_attention_module'
            scm_config = self.module_configs.get(active_module_name, {})
            module_output_dim = scm_config.get("output_dim", current_dim)
            self.tech_modules[active_module_name] = SemanticConceptModule( # これは以前定義したモジュール
                input_dim=current_dim, output_dim=module_output_dim,
                num_concepts=scm_config.get("num_concepts", 32),
                concept_dim=scm_config.get("concept_dim", 128), model_config=self.config
            )
            current_dim = module_output_dim
            print("SemanticAttentionModule enabled.")

        elif self.tech_flags.get("use_direct_non_commutative_module", False):
            active_module_name = 'direct_non_commutative_module'
            dnc_config = self.module_configs.get(active_module_name, {})
            module_output_dim = dnc_config.get("output_dim", current_dim)
            self.tech_modules[active_module_name] = DirectNonCommutativeModule( # これは前回定義したモジュール
                input_dim=current_dim, output_dim=module_output_dim,
                dropout=dnc_config.get("dropout", 0.2), module_specific_config=dnc_config
            )
            current_dim = module_output_dim
            print("DirectNonCommutativeModule enabled.")
```

```
    elif self.tech_flags.get("use_discontinuity_emergence_module", False): # ★新しいモジュールのフラグ

        active_module_name = 'discontinuity_emergence_module'

        dem_config = self.module_configs.get(active_module_name, {})

        module_output_dim = dem_config.get("output_dim", current_dim) # 通常はcurrent_dimと同じ

        self.tech_modules[active_module_name] = DiscontinuityEmergenceModule(

            input_dim=current_dim,

            output_dim=module_output_dim,

            semantic_dim_for_discontinuity=dem_config.get("semantic_dim_for_discontinuity", 12),

            emergent_label_dim=dem_config.get("emergent_label_dim", 24),

            dropout=dem_config.get("dropout", 0.2),

            module_specific_config=dem_config

        )

        current_dim = module_output_dim

        print("DiscontinuityEmergenceModule enabled.")


    self.active_module_name = active_module_name # どのモジュールが有効か保持 (forwardで使う)



    self.dropout = nn.Dropout(self.config.hidden_dropout_prob if hasattr(self.config, 'hidden_dropout_prob') else 0.1)

    self.classifier = nn.Linear(current_dim, self.num_labels)

    print(f"EnhancedTransformerForSequenceClassification initialized. Classifier input dim: {current_dim}, Active module: {self.active_module_name}")



  def forward(
```

```
        self, input_ids=None, attention_mask=None, token_type_ids=None, labels=None,
        return_dict=None, **kwargs ):
        # print(f"--- EnhancedTransformer.forward CALLED (is_training: {self.training}) ---") # デバッグprintは適宜
        # if labels is not None: print(f"  Forward RECEIVED labels, shape: {labels.shape}")
        # else: print(f"  Forward did NOT receive labels this call.")

        base_model_kwargs = { k: v for k, v in kwargs.items() if k in ["position_ids", "head_mask", "inputs_embeds", "output_attentions", "output_hidden_states"]}
        transformer_outputs = self.base_model(input_ids=input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids, return_dict=True, **base_model_kwargs)
        hidden_states = transformer_outputs.last_hidden_state

        module_specific_outputs = {} # モジュール固有の出力を格納

        if self.active_module_name and self.active_module_name in self.tech_modules:
            module_output = self.tech_modules[self.active_module_name](hidden_states, attention_mask=attention_mask)
            hidden_states = module_output["last_hidden_state"] # 全てのモジュールはこのキーで加工済みhidden_statesを返す前提
            # 他の出力（例: attention_weights, discontinuity_scoresなど）も module_specific_outputs に格納
            for key, value in module_output.items():
                if key != "last_hidden_state":
                    module_specific_outputs[key] = value

        pooled_output = hidden_states[:, 0]
        pooled_output = self.dropout(pooled_output)
```

```
        logits = self.classifier(pooled_output)

        loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
            # if loss is not None: print(f"  Forward CALCULATED loss: {loss.item()}") # デバッグprintは適宜
            # else: print(f"  Forward did NOT calculate loss (labels were None).")

        # print(f"--- EnhancedTransformer.forward RETURNING ... ---") # デバッグprintは適宜

        # 返り値のタプル構成 (loss, logits, ベースモデルの出力, モジュール固有の出力)
        # Trainerが主に使うのは loss と logits
        final_outputs = (logits,)
        if transformer_outputs.hidden_states is not None: final_outputs += (transformer_outputs.hidden_states,)
        if transformer_outputs.attentions is not None: final_outputs += (transformer_outputs.attentions,)
        if module_specific_outputs: final_outputs += (module_specific_outputs,) # モジュール出力をタプルに追加

        if loss is not None:
            return (loss,) + final_outputs
        else:
            return (None,) + final_outputs
```

```
# --- DebugTrainer クラスの定義 (変更なし) ---

class DebugTrainer(Trainer):

    def prediction_step(

        self, model: nn.Module, inputs: dict[str, torch.Tensor | nn.utils.rnn.PackedSequence],

        prediction_loss_only: bool, ignore_keys: list[str] | None = None,

    ) -> tuple[torch.Tensor | None, torch.Tensor | None, torch.Tensor | None]:

        # print(f"--- DebugTrainer.prediction_step CALLED ---") # デバッグprintは適宜

        # ... (以前のDebugTrainerのコードはそのまま) ...

        model.eval()

        with torch.no_grad(): model_outputs = model(**inputs)

        loss = model_outputs[0] if model_outputs[0] is not None else None

        logits = model_outputs[1] if len(model_outputs) > 1 and model_outputs[1] is not None else None

        labels_out = inputs.get("labels")

        # ... (詳細なprintデバッグは適宜) ...

        if prediction_loss_only: return (loss, None, None)

        return loss, logits, labels_out


print("Cell 5: Custom model classes (including DiscontinuityEmergenceModule) and DebugTrainer defined.")




# --- セル6: アブレーション実験の構成定義と実行ループ (デバッグプリント強化・最小実行版) ---
```

```
print("--- Cell 6 Execution START ---") # ★セル実行開始の確認


# --- グローバル変数 (セル3, 4, 5 で定義されているはずのもの) の確認 ---
required_globals_cell6 = [
    'hf_model_config', 'BASE_MODEL_NAME', 'NUM_LABELS', 'OUTPUT_DIR_BASE',
    'LEARNING_RATE', 'BATCH_SIZE', 'NUM_EPOCHS',
    'EnhancedTransformerForSequenceClassification',
    'train_dataset', 'eval_dataset', 'tokenizer', 'compute_metrics',
    'Dataset',
    'DebugTrainer'
]
missing_globals_check = False
for var_name in required_globals_cell6:
    if var_name not in globals():
        print(f"ERROR: Global variable '{var_name}' is not defined in Cell 6!")
        missing_globals_check = True

if missing_globals_check:
    raise NameError("One or more required global variables are not defined. Please check previous cells.")
else:
    print("All required global variables for Cell 6 seem to be defined.")


# --- 実験構成の定義 (まずはベースラインのみでテスト) ---
print("Defining experiment configurations (debug: baseline only)...")
experiment_configurations = [
```

```
    {
        "name": "0_baseline_transformer_DEBUG_RUN", # 出力先を分ける
        "tech_flags": {},
        "module_configs": {}
    }
]
all_results = {}
print(f"Generated {len(experiment_configurations)} experiment configurations for debug.")

# --- カラム名リネーム処理 (念のためループ直前にも配置) ---
print("Checking and renaming dataset columns if necessary (pre-loop)...")
try:
    if 'train_dataset' in globals() and isinstance(train_dataset, Dataset) and \
    'label' in train_dataset.column_names and 'labels' not in train_dataset.column_names:
        print("  Renaming 'label' to 'labels' in train_dataset...")
        train_dataset = train_dataset.rename_column("label", "labels")
    if 'eval_dataset' in globals() and isinstance(eval_dataset, Dataset) and \
    'label' in eval_dataset.column_names and 'labels' not in eval_dataset.column_names:
        print("  Renaming 'label' to 'labels' in eval_dataset...")
        eval_dataset = eval_dataset.rename_column("label", "labels")
    print(f"  Train dataset columns: {train_dataset.column_names if 'train_dataset' in globals() else 'Not defined'}")
    print(f"  Eval dataset columns: {eval_dataset.column_names if 'eval_dataset' in globals() else 'Not defined'}")
except Exception as e:
    print(f"Error during pre-loop column renaming: {e}")
    raise
```

```
# --- ここまでリネーム処理 ---


print("\nStarting experiment loop...")
# --- 実験ループ (最初の構成のみ実行) ---
for exp_config_setting in experiment_configurations: # ループは回るが、中身は1回だけ
    config_name = exp_config_setting["name"]
    tech_flags = exp_config_setting["tech_flags"]
    module_cfgs = exp_config_setting["module_configs"]

    print(f"\n--- Running Experiment: {config_name} ---")
    print(f"  Technology Flags: {tech_flags}")
    print(f"  Module Configs: {module_cfgs}")

    try:
        print("  Instantiating model...")
        model = EnhancedTransformerForSequenceClassification(
            hf_config=hf_model_config,
            base_model_name=BASE_MODEL_NAME,
            num_labels=NUM_LABELS,
            tech_flags=tech_flags,
            module_configs=module_cfgs
        )
        print(f"  Model '{type(model).__name__}' instantiated successfully.")

        current_output_dir = f"{OUTPUT_DIR_BASE}{config_name}"
        print(f"  Output directory set to: {current_output_dir}")
```

```
    print("  Defining TrainingArguments...")
    training_args = TrainingArguments(
        output_dir=current_output_dir,
        num_train_epochs=1, # 最小限
        per_device_train_batch_size=BATCH_SIZE, # セル3で定義 (例: 1)
        per_device_eval_batch_size=BATCH_SIZE,  # セル3で定義 (例: 1)

        evaluation_strategy="epoch",
        logging_strategy="epoch",
        save_strategy="no", # 保存しない

        report_to="none",
        do_eval=True,
        load_best_model_at_end=False,
        remove_unused_columns=False,
    )
    print(f"  TrainingArguments defined. Evaluation strategy:
{training_args.evaluation_strategy}")

    print("  Instantiating DebugTrainer...")
    trainer = DebugTrainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=eval_dataset,
        tokenizer=tokenizer,
```

```
        compute_metrics=compute_metrics,
    )
    print(f"  DebugTrainer instantiated. Compute_metrics function: {trainer.compute_metrics.__name__ if trainer.compute_metrics else 'None'}")

    print(f"  Starting trainer.train() for {config_name}...")
    trainer.train() # 訓練と評価の実行
    print(f"  Training finished for {config_name}.")

    # trainer.train() の中で評価が行われ、結果がログに出るはず
    # 最終評価を明示的に行いたい場合は以下も実行
    # print(f"  Starting final evaluation for {config_name}...")
    # eval_results = trainer.evaluate()
    # print(f"  Final evaluation results for {config_name}: {eval_results}")
    # if eval_results:
    #     all_results[config_name] = eval_results

    # Trainerのログから最新の評価結果を取得しようと試みる (より頑健な方法)
    # 訓練中に保存されたログからメトリクスを取得
    if trainer.state.log_history:
        print("  Extracting eval metrics from trainer state log history...")
        final_eval_metrics = {}
        for log_entry in reversed(trainer.state.log_history): # 最新のログから探す
            is_eval_log = True
            # eval_loss と eval_accuracy が両方あるか確認
            if 'eval_loss' not in log_entry or 'eval_accuracy' not in log_entry:
```

```
                is_eval_log = False


            if is_eval_log : # これが評価ログのエントリだと仮定
                final_eval_metrics = {k: v for k, v in log_entry.items() if k.startswith("eval_") or k ==
"epoch"}
                print(f"    Found eval metrics in log: {final_eval_metrics}")
                all_results[config_name] = final_eval_metrics
                break # 最新の評価ログを見つけたら終了

          if not final_eval_metrics:
              print("    No evaluation metrics found in trainer log history with both eval_loss and
eval_accuracy.")
       else:
          print("  Trainer log history is empty. No metrics to extract.")



    except Exception as e:
       print(f"!!! An error occurred during experiment {config_name}: {e} !!!")
       import traceback
       traceback.print_exc()
       # エラーが発生しても、次の実験構成があればループは継続する (breakしない)
       # ただし、このデバッグ版では実験構成は1つだけ


print("\n\n--- Experiment Summary ---")
if all_results:
    for name, res in sorted(all_results.items()):
       accuracy = res.get('eval_accuracy', 'N/A')
       loss = res.get('eval_loss', 'N/A')
```

```
        print(f"Configuration: {name:<50} Accuracy: {accuracy:<10} Loss: {loss:<10}")
else:
    print("No results to summarize. Experiments might have failed or did not produce eval metrics in log_history.")


print("\n--- Cell 6 Execution END ---") # ★セル実行終了の確認
```