マルチモーダル知識融合による適応的反射型神経記号システム（特願2025-077682、出願人：New York General Group, Inc.、発明者：村上 由宇）

New York General Group
2025

1. 技術分野

本発明は人工知能技術、特に神経記号的人工知能（Neuro-Symbolic AI）システムに関し、より具体的には、異なるモダリティからの情報を統合し、動的に適応する反射機構を備えた高効率な神経記号的推論システムに関する。本発明は、ニューラルネットワークによる直感的処理と記号的推論を効率的に組み合わせ、領域知識との整合性を保ちながら複雑な推論タスクを実行する技術を提供するものである。

2. 背景技術

人工知能の分野において、ニューラルネットワークは複雑なパターン認識や予測タスクで優れた性能を示してきた。深層学習の進展により、画像認識、自然言語処理、音声認識などの分野で人間に匹敵する、あるいは人間を上回る性能が達成されている。しかし、ニューラルネットワークは、領域知識との整合性を保証することや、その決定過程の透明性を確保することが困難であるという課題がある。また、少ないデータからの学習や、既存知識の活用という点でも限界がある。

一方、記号的推論システムは明示的な知識表現と論理的推論能力を備えており、推論過程の透明性や知識の再利用性において優れている。しかし、生データからの特徴抽出や不確実性の処理、大規模データからのパターン学習といった点では課題がある。

これらの課題に対処するため、ニューラルネットワークと記号的推論を統合する神経記号的人工知能（NeSy）が提案されてきた。この統合アプローチは、人間の二重過程認知に類似しており、直感的な「システム1」思考をニューラルネットワークでモデル化し、アルゴリズム的な「システム2」思考を記号的推論でモデル化する。これにより、両者の長所を活かしつつ、短所を補完することが期待されている。

既存のNeSyシステムには、以下のようなアプローチがある。
1. ファジー論理を用いるアプローチ：論理規則をファジー化し、ニューラルネットワークで近似する方法
2. 記号的知識をニューラルネットワークの制約として緩和するアプローチ：論理規則を損失関数に組み込む方法
3. 分散表現を用いて論理計算を近似するアプローチ：ベクトル空間上で論理演算を実行する方法

しかし、これらのアプローチは記号的推論の能力を完全に保持できないという限界がある。論理規則の緩和や近似により、厳密な推論能力が損なわれることが多い。

帰納学習（Abductive Learning, ABL）は、機械学習と論理的推論を橋渡しする枠組みとして提案されてきた。ABLでは、機械学習コンポーネントが生データを原始的な記号出力に変換し、記号的推論コンポーネントが領域知識を活用して帰納を実行し、より信頼性の高い出力を生成する。これにより、ニューラルネットワークと記号的推論の両方の表現力を保持しながら統合することが可能となる。

Hu et al.（2025）は、「帰納的反射（Abductive Reflection, ABL-Ref）」を提案し、ニューラルネットワークの出力における潜在的な誤りを検出するための反射ベクトルを生成する方法を示した。この方法では、反射ベ

クトルが直感的な出力のどの部分が領域知識と不整合を引き起こす可能性があるかをフラグ付けし、それらの部分を記号的推論による帰納を通じて修正する。これにより、従来のABLにおける計算ボトルネックであった整合性最適化モジュールが不要となり、効率が大幅に向上した。

しかし、Hu et al.の方法には以下の限界がある。

1. 単一モダリティのデータ処理に限定されており、マルチモーダル情報の統合能力が欠如している。現実世界の多くの問題は、テキスト、画像、音声、構造化データなど、複数のモダリティからの情報を統合して解決する必要がある。

2. 反射メカニズムが静的であり、タスクの複雑さや不確実性に応じて適応的に調整されない。すべてのタスクに対して同じ反射メカニズムを適用するため、簡単なタスクに対しても不必要に計算リソースを消費する可能性がある。

3. 知識ベースが固定されており、新しい情報や経験から学習して拡張する能力がない。これにより、時間の経過とともに変化する環境や問題に対応することが困難となる。

4. 異なるタスク間での知識転移メカニズムが欠如している。類似したタスク間で知識を共有し、転移学習を促進する機能がないため、新しいタスクごとに一からトレーニングする必要がある。

5. 反射プロセスとニューラルネットワークの出力生成プロセスが並列的であり、相互にフィードバックする機構がない。これにより、反射の結果を出力生成に活かすことができず、推論の質を反復的に向上させることが困難である。

これらの限界は、複雑で動的な環境における神経記号的システムの適用可能性と効率性を制限している。特に、マルチモーダルデータを扱う必要がある実世界の問題や、継続的に学習して適応する必要がある長期的なタスクにおいて、これらの限界は顕著となる。

3. 発明が解決しようとする課題

本発明が解決しようとする課題は、以下の通りである。

1. 異なるモダリティ（テキスト、画像、音声、構造化データなど）からの情報を効果的に統合し、マルチモーダル推論を可能にすること。現実世界の多くの問題は、単一のモダリティだけでなく、複数のモダリティからの情報を組み合わせて解決する必要がある。例えば、医療診断では、患者の症状記述（テキスト）、医療画像（画像）、検査結果（構造化データ）を統合して診断を行う必要がある。

2. タスクの複雑さや不確実性に応じて反射メカニズムを動的に適応させ、計算リソースを最適に配分すること。簡単なタスクには少ない計算リソースを割り当て、複雑なタスクにはより多くのリソースを割り当てることで、全体的な効率を向上させる必要がある。

3. 経験から学習して知識ベースを継続的に拡張・更新する能力を提供すること。固定された知識ベースではなく、新しい情報や経験から学習して知識を蓄積し、時間の経過とともにシステムの性能を向上させる機能が必要である。

4. 異なるタスク間での知識転移を促進し、少ないデータでの学習効率を向上させること。類似したタスク間で知識を共有し、新しいタスクに対して効率的に適応する能力が求められる。

5. 反射プロセスとニューラルネットワークの出力生成プロセスの間に双方向フィードバックループを確立し、推論の質を向上させること。反射の結果を出力生成にフィードバックし、出力を反復的に改善する機構が必要である。

これらの課題を解決することで、より柔軟で効率的な神経記号的システムを実現し、複雑な実世界の問題に対応できるようにすることが本発明の目的である。

4. 課題を解決するための手段

本発明は、「マルチモーダル適応的反射（Multimodal Adaptive Reflection, MAR）」メカニズムを提案する。このメカニズムは、Hu et al.の帰納的反射を拡張し、マルチモーダル情報の統合、動的適応、継続的学習、知識転移、双方向フィードバックの能力を追加する。

具体的には、本発明のシステムは以下のコンポーネントから構成される。

1. マルチモーダルエンコーダ（ME）
異なるモダリティ（テキスト、画像、音声など）からの入力を処理し、統一された表現空間に変換するコンポーネントである。各モダリティ専用のエンコーダと、それらの出力を統合するクロスモーダル融合モジュールから構成される。

テキストエンコーダは、トランスフォーマーベースのアーキテクチャを使用して、テキスト入力を処理する。画像エンコーダは、畳み込みニューラルネットワーク（CNN）または視覚トランスフォーマーを使用して、画像入力を処理する。音声エンコーダは、時間的畳み込みネットワーク（TCN）または音声専用トランスフォーマーを使用して、音声入力を処理する。構造化データエンコーダは、グラフニューラルネットワーク（GNN）を使用して、グラフや表形式のデータを処理する。

クロスモーダル融合モジュールは、注意機構を使用して、異なるモダリティからの表現を統合する。このモジュールは、モダリティ間の関連性を捉え、統一された表現を生成する。例えば、テキスト「赤いリンゴ」と、リンゴの画像が与えられた場合、クロスモーダル融合モジュールは、テキストの「赤い」という属性と画像の赤色の部分を関連付け、統合された表現を生成する。

2. コンテキスト認識ネットワーク本体（CAN）
マルチモーダルエンコーダからの統合表現を処理し、タスク固有の文脈を考慮した高次元の埋め込みを生成するコンポーネントである。タスクの複雑さや不確実性に応じて処理深度を動的に調整する適応層を含む。

適応的処理深度機構は、タスクの複雑さに応じて、処理層の数や計算量を動的に調整する。簡単なタスクには浅い処理を、複雑なタスクには深い処理を適用することで、計算効率を向上させる。例えば、単純な分類タスクでは数層の処理で十分であるが、複雑な推論タスクではより多くの層を使用する。

コンテキストメモリは、過去の入力や処理結果を記憶し、現在の処理に活用する機構である。長期依存関係を捉えるために重要であり、時系列データや対話システムなどで特に有用である。

不確実性認識層は、入力や中間表現の不確実性を推定し、後続の処理に反映させる層である。不確実性が高い部分には、より慎重な処理を適用することで、推論の信頼性を向上させる。

3. 出力生成層（OGL）
コンテキスト認識ネットワーク本体の埋め込みから直感的な出力を生成するコンポーネントである。タスクに応じて、分類、回帰、生成など、様々な形式の出力を生成することができる。

4. 階層的反射モジュール（HRM）

コンテキスト認識ネットワーク本体の埋め込みから多層的な反射ベクトルを生成するコンポーネントである。基本層、メタ認知層、不確実性推定層から構成され、出力の異なる側面（論理的整合性、事実的正確性、不確実性など）を評価する。

基本反射層は、出力の論理的整合性を評価し、領域知識との不整合を検出する層である。Hu et al.の反射層に相当するが、マルチモーダル情報を考慮する点で拡張されている。例えば、テキストと画像の両方から得られる情報を統合して、出力の整合性を評価する。

メタ認知層は、システム自体の推論プロセスを監視し、潜在的な推論エラーを検出する層である。過去の推論パターンや一般的な推論バイアスに基づいて評価を行う。例えば、特定のタイプの入力に対して過去に誤った推論を行った場合、類似の入力に対してより慎重な処理を行うようにする。

不確実性推定層は、出力の各部分の不確実性を推定し、高い不確実性を持つ部分を識別する層である。不確実性の種類（認識的不確実性、偶然的不確実性）を区別し、適切な対処を行う。認識的不確実性は知識の不足によるものであり、より多くの情報収集が必要となる。偶然的不確実性は本質的なランダム性によるものであり、確率的な推論が必要となる。

統合層は、上記の層からの評価を統合し、最終的な反射ベクトルを生成する層である。各層の評価に重み付けを行い、タスクや文脈に応じて調整する。例えば、論理的整合性が重要なタスクでは基本反射層の評価に高い重みを与え、不確実性の高いタスクでは不確実性推定層の評価に高い重みを与える。

5. 拡張可能知識ベース（EKB）
領域知識を表現し、記号的推論を実行するとともに、新しい知識を継続的に統合する機能を持つコンポーネントである。知識グラフ、論理規則、確率モデルなど複数の知識表現形式をサポートする。

多様な知識表現は、命題論理、一階論理、確率論理、知識グラフなど、複数の形式で知識を表現できる機能である。これにより、様々な種類の知識を統合し、柔軟な推論を行うことができる。

ハイブリッド推論エンジンは、異なる推論メカニズム（演繹、帰納、アブダクション、確率推論）を統合し、状況に応じて適切な推論方法を選択する機能である。例えば、確実な知識がある場合は演繹推論を、不確実な知識がある場合は確率推論を使用する。

知識獲得モジュールは、新しい経験や観察から知識を抽出し、既存の知識ベースに統合する機能である。矛盾する知識の検出と解決機能を含み、知識ベースの一貫性を維持する。例えば、新しい観察が既存の知識と矛盾する場合、信頼性や適用範囲に基づいて調整を行う。

知識検証モジュールは、知識の信頼性や適用可能性を評価し、不確かな知識に適切な信頼度を割り当てる機能である。これにより、不確実な知識を含む推論でも、信頼性の高い結論を導くことができる。

6. メタ学習コントローラ（MLC）
システム全体の学習プロセスを監視し、タスクの性質や難易度に応じてパラメータ更新戦略を調整するコンポーネントである。異なるタスク間での知識転移を促進する。

タスク特性分析は、入力タスクの特性（複雑さ、不確実性、既存知識との関連性など）を分析し、適切な学習戦略を決定する機能である。例えば、複雑なタスクには低い学習率と強い正則化を、単純なタスクには高い学習率と弱い正則化を適用する。

パラメータ更新制御は、学習率、正則化強度、勾配クリッピングなどのハイパーパラメータを動的に調整する機能である。学習の進行状況や性能指標に基づいて調整を行い、最適な学習を実現する。

知識転移促進は、異なるタスク間での知識の共有と転移を促進するためのパラメータ共有戦略を実装する機能である。類似したタスク間で共通の特徴表現や知識を共有し、効率的な学習を実現する。

学習進捗監視は、学習の進捗を監視し、停滞や過学習の兆候を検出して対応策を講じる機能である。例えば、検証誤差が増加し始めた場合に学習率を下げるなどの調整を行う。

7. フィードバックループ統合器（FLI）
反射プロセスの結果を出力生成プロセスにフィードバックし、出力の質を反復的に向上させるコンポーネントである。

反射結果分析は、階層的反射モジュールからの反射ベクトルを分析し、出力のどの部分が修正を必要とするかを特定する機能である。反射ベクトルの各要素の値や分布を分析し、修正の優先順位を決定する。

出力修正生成は、拡張可能知識ベースを用いて、修正が必要な部分に対する新しい出力候補を生成する機能である。帰納、演繹、確率推論など、適切な推論メカニズムを使用して修正候補を生成する。

整合性評価は、修正された出力の全体的な整合性を評価し、必要に応じてさらなる修正を行う機能である。出力全体が領域知識と整合しているかを確認し、新たな不整合が生じた場合は追加の修正を行う。

反復終了判定は、出力の質が十分に向上したか、または反復回数が上限に達したかを判断し、プロセスを終了するタイミングを決定する機能である。出力の整合性や不確実性の指標に基づいて判断を行い、効率的な推論を実現する。

処理フローにおいては、マルチモーダル入力がマルチモーダルエンコーダによって統合され、コンテキスト認識ネットワーク本体によって処理される。コンテキスト認識ネットワーク本体の埋め込みから出力生成層が直感的な出力を生成し、同時に階層的反射モジュールが階層的な反射ベクトルを生成する。反射ベクトルに基づいて、直感的な出力から潜在的な誤りが識別され、拡張可能知識ベースによる記号的推論を通じて修正される。このプロセスはフィードバックループ統合器によって反復され、最終的な出力が生成される。メタ学習コントローラは全体のプロセスを監視し、学習戦略を最適化する。

5. 発明の効果

本発明の主な効果は以下の通りである。

1. マルチモーダル推論能力
異なるモダリティ（テキスト、画像、音声、構造化データなど）からの情報を効果的に統合し、より豊かな文脈理解と推論を可能にする。これにより、単一モダリティの情報だけでは解決困難な複雑な問題に対応できるようになる。例えば、医療診断において、患者の症状記述（テキスト）、医療画像（画像）、検査結果（構造化データ）を統合して、より正確な診断を行うことができる。

2. 計算効率の向上
タスクの複雑さや不確実性に応じて反射メカニズムを動的に調整し、簡単なタスクには少ない計算リソースを割り当て、複雑なタスクにはより多くのリソースを割り当てる。これにより、全体的な計算効率が向上し、リアルタイム応用や大規模データセットの処理が可能となる。例えば、単純な分類タスクでは浅い処理で迅速に結果を出し、複雑な推論タスクでは深い処理で正確な結果を出すことができる。

3. 継続的学習能力
経験から学習して知識ベースを拡張・更新し、時間の経過とともにシステムの性能を向上させる。これにより、静的な知識ベースに依存するシステムよりも、変化する環境や新しい問題に適応できるようになる。例えば、新しい医学的発見や診断基準の変更に応じて、医療診断システムの知識ベースを自動的に更新することができる。

4. 転移学習の促進
異なるタスク間での知識転移を可能にし、少ないデータでの学習効率を向上させる。これにより、新しいタスクや領域に対して、少ないトレーニングデータでも高い性能を達成できるようになる。例えば、ある疾患の診断に関する知識を、類似した別の疾患の診断に転用することができる。

5. 推論品質の向上
反射プロセスと出力生成プロセスの間の双方向フィードバックにより、推論の質を反復的に向上させる。これにより、初期の出力に含まれる誤りや不整合を検出し、修正することができ、最終的な出力の信頼性が向上する。例えば、初期の診断候補に矛盾がある場合、フィードバックループを通じて修正し、より整合性のある診断結果を生成することができる。

6. 解釈可能性の強化
階層的反射モジュールが出力の異なる側面（論理的整合性、事実的正確性、不確実性など）を評価し、システムの決定過程の透明性を高める。これにより、ユーザーはシステムの推論過程を理解し、信頼性を評価することができる。例えば、診断結果とともに、その診断の根拠となる所見や、不確実性の程度を提示することができる。

7. 適応性の向上
様々なタスクや領域に対して柔軟に適応し、幅広い応用シナリオに対応できる。これにより、同じシステムアーキテクチャを異なる問題領域に適用することができ、開発コストと時間を削減できる。例えば、医療診断、金融リスク分析、自動運転など、様々な領域に同じシステムアーキテクチャを適用することができる。

これらの効果により、本発明は既存の神経記号的システムよりも、より柔軟で効率的、かつ信頼性の高い推論を実現し、複雑な実世界の問題に対応できるようになる。

6. 発明を実施するための形態

以下、本発明の実施形態について詳細に説明する。本発明は、異なるモダリティからの情報を統合し、動的に適応する反射機構を備えた高効率な神経記号的推論システムである「マルチモーダル適応的反射（Multimodal Adaptive Reflection, MAR）」メカニズムを提供する。本発明のシステムは、ニューラルネットワークによる直感的処理と記号的推論を効率的に組み合わせ、領域知識との整合性を保ちながら複雑な推論タスクを実行する。

6.1. システム全体のアーキテクチャ

本発明のシステムは、以下の主要コンポーネントから構成される。

1. マルチモーダルエンコーダ（ME）
2. コンテキスト認識ネットワーク本体（CAN）
3. 出力生成層（OGL）
4. 階層的反射モジュール（HRM）
5. 拡張可能知識ベース（EKB）
6. フィードバックループ統合器（FLI）
7. メタ学習コントローラ（MLC）

これらのコンポーネントは、図1に示すように相互に接続されている。マルチモーダルエンコーダは入力データを受け取り、コンテキスト認識ネットワーク本体に統合された表現を提供する。コンテキスト認識ネットワーク本体は、この表現を処理して高次元の埋め込みを生成し、出力生成層と階層的反射モジュールに送信する。出力生成層は直感的な出力を生成し、階層的反射モジュールは反射ベクトルを生成する。フィード

バックループ統合器は、反射ベクトルに基づいて直感的な出力の誤りを識別し、拡張可能知識ベースを用いて修正する。メタ学習コントローラは、システム全体の学習プロセスを監視し、最適化する。

以下、各コンポーネントの詳細な実装方法と動作原理について説明する。

6.2. マルチモーダルエンコーダ（ME）の実装

マルチモーダルエンコーダは、異なるモダリティからの入力を処理し、統一された表現空間に変換するコンポーネントである。本発明では、テキスト、画像、音声、構造化データなどの異なるモダリティを処理できる柔軟なアーキテクチャを採用している。

6.2.1 テキストエンコーダの実装

テキストエンコーダは、テキスト入力を処理するためのサブコンポーネントである。具体的な実装は以下の通りである。

```python
class TextEncoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, num_layers, dropout=0.1):
        super(TextEncoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(
                d_model=embedding_dim,
                nhead=8,
                dim_feedforward=hidden_dim,
                dropout=dropout
            ),
            num_layers=num_layers
        )
        self.output_projection = nn.Linear(embedding_dim, hidden_dim)

    def forward(self, text_input, attention_mask=None):
        # text_input: [batch_size, seq_length]
        embedded = self.embedding(text_input)  # [batch_size, seq_length, embedding_dim]

        if attention_mask is not None:
            # Convert attention mask to proper format for transformer
            attention_mask = attention_mask.float().masked_fill(
                attention_mask == 0, float('-inf')).masked_fill(
                attention_mask == 1, float(0.0)
            )

        transformed = self.transformer(embedded.transpose(0, 1), src_key_padding_mask=attention_mask).transpose(0, 1)
        output = self.output_projection(transformed)  # [batch_size, seq_length, hidden_dim]

        # Global representation: mean pooling over sequence length
        global_repr = output.mean(dim=1)  # [batch_size, hidden_dim]

        return global_repr, output
```

テキストエンコーダは、単語埋め込み層、トランスフォーマーエンコーダ層、出力投影層から構成される。入力テキストは、まず単語埋め込み層を通じてベクトル表現に変換される。次に、トランスフォーマーエンコーダ層を通じて文脈化された表現が生成される。最後に、出力投影層を通じて、指定された次元の出力表現が生成される。

実際の実装では、BERT、RoBERTa、T5などの事前学習済みモデルを基盤として使用することも可能である。その場合、以下のように実装される。

```python
class PretrainedTextEncoder(nn.Module):
    def __init__(self, model_name, hidden_dim):
        super(PretrainedTextEncoder, self).__init__()
        self.model = AutoModel.from_pretrained(model_name)
        self.output_projection = nn.Linear(self.model.config.hidden_size, hidden_dim)

    def forward(self, input_ids, attention_mask=None):
        outputs = self.model(input_ids=input_ids, attention_mask=attention_mask)
        sequence_output = outputs.last_hidden_state  # [batch_size, seq_length, hidden_size]

        # Global representation: use [CLS] token or mean pooling
        global_repr = sequence_output[:, 0, :]  # [batch_size, hidden_size]
        # Alternatively: global_repr = (sequence_output * attention_mask.unsqueeze(-1)).sum(1) /
attention_mask.sum(-1).unsqueeze(-1)

        projected_output = self.output_projection(sequence_output)  # [batch_size, seq_length, hidden_dim]
        projected_global = self.output_projection(global_repr)  # [batch_size, hidden_dim]

        return projected_global, projected_output
```

6.2.2 画像エンコーダの実装

画像エンコーダは、画像入力を処理するためのサブコンポーネントである。具体的な実装は以下の通りである。

```python
class ImageEncoder(nn.Module):
    def __init__(self, hidden_dim, backbone='resnet50', pretrained=True):
        super(ImageEncoder, self).__init__()
        if backbone == 'resnet50':
            self.backbone = models.resnet50(pretrained=pretrained)
            backbone_dim = 2048
        elif backbone == 'efficientnet_b0':
            self.backbone = models.efficientnet_b0(pretrained=pretrained)
            backbone_dim = 1280
        elif backbone == 'vit_b_16':
            self.backbone = models.vit_b_16(pretrained=pretrained)
            backbone_dim = 768
        else:
            raise ValueError(f"Unsupported backbone: {backbone}")

        # Remove the classification head
        if backbone.startswith('resnet') or backbone.startswith('efficientnet'):
            self.backbone = nn.Sequential(*list(self.backbone.children())[:-1])

        self.output_projection = nn.Linear(backbone_dim, hidden_dim)

    def forward(self, image_input):
        # image_input: [batch_size, channels, height, width]
        features = self.backbone(image_input)  # [batch_size, backbone_dim, 1, 1] or [batch_size, backbone_dim]

        # Flatten if necessary
        if len(features.shape) > 2:
            features = features.flatten(1)  # [batch_size, backbone_dim]

        output = self.output_projection(features)  # [batch_size, hidden_dim]

        return output
```

画像エンコーダは、バックボーンネットワーク（ResNet、EfficientNet、ViTなど）と出力投影層から構成される。入力画像は、まずバックボーンネットワークを通じて特徴表現が抽出される。次に、出力投影層を通じて、指定された次元の出力表現が生成される。

より高度な画像理解のために、特徴マップを保持する実装も可能である。

```python
class AdvancedImageEncoder(nn.Module):
    def __init__(self, hidden_dim, backbone='resnet50', pretrained=True):
        super(AdvancedImageEncoder, self).__init__()
        if backbone == 'resnet50':
            # Load backbone without classification head
            self.backbone = models.resnet50(pretrained=pretrained)
            self.backbone = nn.Sequential(*list(self.backbone.children())[:-2])
            backbone_dim = 2048
        else:
            # Similar implementation for other backbones
            pass

        self.global_pool = nn.AdaptiveAvgPool2d(1)
        self.output_projection = nn.Conv2d(backbone_dim, hidden_dim, kernel_size=1)
        self.global_projection = nn.Linear(backbone_dim, hidden_dim)

    def forward(self, image_input):
        # image_input: [batch_size, channels, height, width]
        feature_maps = self.backbone(image_input)  # [batch_size, backbone_dim, h, w]

        # Global representation
        global_features = self.global_pool(feature_maps).flatten(1)  # [batch_size, backbone_dim]
        global_output = self.global_projection(global_features)  # [batch_size, hidden_dim]

        # Local representations
        local_output = self.output_projection(feature_maps)  # [batch_size, hidden_dim, h, w]

        return global_output, local_output
```

6.2.3 音声エンコーダの実装

音声エンコーダは、音声入力を処理するためのサブコンポーネントである。具体的な実装は以下の通りである。

```python
class AudioEncoder(nn.Module):
    def __init__(self, hidden_dim, sample_rate=16000, n_mels=128):
        super(AudioEncoder, self).__init__()
        self.melspec = torchaudio.transforms.MelSpectrogram(
            sample_rate=sample_rate,
            n_fft=400,
            hop_length=160,
            n_mels=n_mels
        )
        self.amplitude_to_db = torchaudio.transforms.AmplitudeToDB()

        # CNN layers
        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        self.pool = nn.AdaptiveAvgPool2d(1)
```

```
        self.output_projection = nn.Linear(256, hidden_dim)

    def forward(self, audio_input):
        # audio_input: [batch_size, time_steps]

        # Convert to mel spectrogram
        mel = self.melspec(audio_input)  # [batch_size, n_mels, time]
        mel_db = self.amplitude_to_db(mel)  # [batch_size, n_mels, time]

        # Add channel dimension
        x = mel_db.unsqueeze(1)  # [batch_size, 1, n_mels, time]

        # CNN layers
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.bn3(self.conv3(x)))

        # Global pooling
        x = self.pool(x).flatten(1)  # [batch_size, 256]

        # Projection
        output = self.output_projection(x)  # [batch_size, hidden_dim]

        return output
```

音声エンコーダは、メルスペクトログラム変換、CNN層、グローバルプーリング層、出力投影層から構成される。入力音声は、まずメルスペクトログラムに変換される。次に、CNN層を通じて特徴表現が抽出される。最後に、グローバルプーリングと出力投影を通じて、指定された次元の出力表現が生成される。

より高度な音声処理のために、事前学習済みモデル（Wav2Vec、HuBERTなど）を使用することも可能である。

```python
class PretrainedAudioEncoder(nn.Module):
    def __init__(self, model_name, hidden_dim):
        super(PretrainedAudioEncoder, self).__init__()
        self.model = AutoModel.from_pretrained(model_name)
        self.output_projection = nn.Linear(self.model.config.hidden_size, hidden_dim)

    def forward(self, audio_input, attention_mask=None):
        outputs = self.model(audio_input, attention_mask=attention_mask)
        sequence_output = outputs.last_hidden_state  # [batch_size, seq_length, hidden_size]

        # Global representation: mean pooling over sequence length
        if attention_mask is not None:
            global_repr = (sequence_output * attention_mask.unsqueeze(-1)).sum(1) /
attention_mask.sum(-1).unsqueeze(-1)
        else:
            global_repr = sequence_output.mean(dim=1)  # [batch_size, hidden_size]

        projected_output = self.output_projection(sequence_output)  # [batch_size, seq_length, hidden_dim]
        projected_global = self.output_projection(global_repr)  # [batch_size, hidden_dim]

        return projected_global, projected_output
```

6.2.4 構造化データエンコーダの実装

構造化データエンコーダは、グラフや表形式のデータを処理するためのサブコンポーネントである。具体的な実装は以下の通りである。

```python
class GraphEncoder(nn.Module):
    def __init__(self, node_dim, edge_dim, hidden_dim, num_layers=3):
        super(GraphEncoder, self).__init__()
        self.node_embedding = nn.Linear(node_dim, hidden_dim)
        self.edge_embedding = nn.Linear(edge_dim, hidden_dim)

        self.gnn_layers = nn.ModuleList()
        for _ in range(num_layers):
            self.gnn_layers.append(GATConv(hidden_dim, hidden_dim, heads=4, concat=False))

        self.global_pool = GlobalAttentionPool(hidden_dim)

    def forward(self, x, edge_index, edge_attr=None, batch=None):
        # x: [num_nodes, node_dim]
        # edge_index: [2, num_edges]
        # edge_attr: [num_edges, edge_dim]
        # batch: [num_nodes]

        # Node embedding
        x = self.node_embedding(x)  # [num_nodes, hidden_dim]

        # Edge embedding (if available)
        if edge_attr is not None:
            edge_attr = self.edge_embedding(edge_attr)  # [num_edges, hidden_dim]

        # GNN layers
        for gnn in self.gnn_layers:
            if edge_attr is not None:
                x = gnn(x, edge_index, edge_attr)
            else:
                x = gnn(x, edge_index)
            x = F.relu(x)

        # Global pooling
        if batch is not None:
            global_repr = self.global_pool(x, batch)  # [batch_size, hidden_dim]
        else:
            global_repr = x.mean(dim=0, keepdim=True)  # [1, hidden_dim]

        return global_repr, x
```

構造化データエンコーダは、ノード埋め込み層、エッジ埋め込み層、GNN層、グローバルプーリング層から構成される。入力グラフのノードとエッジは、まず埋め込み層を通じてベクトル表現に変換される。次に、GNN層を通じてノード表現が更新される。最後に、グローバルプーリングを通じて、グラフ全体の表現が生成される。

表形式データの場合、以下のように実装される。

```python
class TabularEncoder(nn.Module):
    def __init__(self, input_dims, hidden_dim, categorical_cols=None, numerical_cols=None):
        super(TabularEncoder, self).__init__()
        self.categorical_cols = categorical_cols or []
        self.numerical_cols = numerical_cols or []

        # Embeddings for categorical columns
        self.categorical_embeddings = nn.ModuleDict({
            col: nn.Embedding(input_dims[col], min(50, (input_dims[col] + 1) // 2))
            for col in self.categorical_cols
        })
```

```
        # Calculate total embedding dimension
        total_embedding_dim = sum(emb.embedding_dim for emb in self.categorical_embeddings.values())
        total_embedding_dim += len(self.numerical_cols)

        # MLP for feature transformation
        self.mlp = nn.Sequential(
            nn.Linear(total_embedding_dim, hidden_dim * 2),
            nn.BatchNorm1d(hidden_dim * 2),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.ReLU()
        )

    def forward(self, x):
        # x: dictionary with keys as column names and values as tensors

        # Process categorical columns
        categorical_embeddings = []
        for col in self.categorical_cols:
            categorical_embeddings.append(self.categorical_embeddings[col](x[col]))

        # Process numerical columns
        numerical_features = []
        for col in self.numerical_cols:
            numerical_features.append(x[col].unsqueeze(1))

        # Concatenate all features
        if categorical_embeddings and numerical_features:
            features = torch.cat(categorical_embeddings + numerical_features, dim=1)
        elif categorical_embeddings:
            features = torch.cat(categorical_embeddings, dim=1)
        else:
            features = torch.cat(numerical_features, dim=1)

        # Apply MLP
        output = self.mlp(features)  # [batch_size, hidden_dim]

        return output
```

6.2.5 クロスモーダル融合モジュールの実装

クロスモーダル融合モジュールは、異なるモダリティからの表現を統合するためのサブコンポーネントである。具体的な実装は以下の通りである。

```python
class CrossModalFusion(nn.Module):
    def __init__(self, hidden_dim, num_modalities, fusion_type='attention'):
        super(CrossModalFusion, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_modalities = num_modalities
        self.fusion_type = fusion_type

        if fusion_type == 'attention':
            # Cross-modal attention
            self.query_proj = nn.Linear(hidden_dim, hidden_dim)
            self.key_proj = nn.Linear(hidden_dim, hidden_dim)
            self.value_proj = nn.Linear(hidden_dim, hidden_dim)
            self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

            # Output projection
            self.output_proj = nn.Linear(hidden_dim, hidden_dim)
```

```
        elif fusion_type == 'concat':
            # Concatenation followed by projection
            self.fusion_proj = nn.Sequential(
                nn.Linear(hidden_dim * num_modalities, hidden_dim * 2),
                nn.ReLU(),
                nn.Linear(hidden_dim * 2, hidden_dim)
            )

        elif fusion_type == 'gated':
            # Gated fusion
            self.gate_networks = nn.ModuleList([
                nn.Sequential(
                    nn.Linear(hidden_dim * num_modalities, hidden_dim),
                    nn.Sigmoid()
                )
                for _ in range(num_modalities)
            ])
            self.output_proj = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, modality_representations):
        # modality_representations: list of tensors, each with shape [batch_size, hidden_dim]

        if self.fusion_type == 'attention':
            # Stack representations
            stacked = torch.stack(modality_representations, dim=0)  # [num_modalities, batch_size, hidden_dim]

            # Self-attention across modalities
            queries = self.query_proj(stacked)
            keys = self.key_proj(stacked)
            values = self.value_proj(stacked)

            attn_output, _ = self.attention(queries, keys, values)  # [num_modalities, batch_size, hidden_dim]

            # Mean pooling across modalities
            fused = attn_output.mean(dim=0)  # [batch_size, hidden_dim]

            # Output projection
            output = self.output_proj(fused)  # [batch_size, hidden_dim]

        elif self.fusion_type == 'concat':
            # Concatenate representations
            concat = torch.cat(modality_representations, dim=1)  # [batch_size, hidden_dim * num_modalities]

            # Project to output dimension
            output = self.fusion_proj(concat)  # [batch_size, hidden_dim]

        elif self.fusion_type == 'gated':
            # Concatenate for gate computation
            concat = torch.cat(modality_representations, dim=1)  # [batch_size, hidden_dim * num_modalities]

            # Compute gates for each modality
            gates = [gate_net(concat) for gate_net in self.gate_networks]  # list of [batch_size, hidden_dim]

            # Apply gates and sum
            gated_representations = [gate * rep for gate, rep in zip(gates, modality_representations)]
            fused = sum(gated_representations)  # [batch_size, hidden_dim]

            # Output projection
            output = self.output_proj(fused)  # [batch_size, hidden_dim]

        else:
            # Simple average
            output = torch.stack(modality_representations).mean(dim=0)  # [batch_size, hidden_dim]

        return output
```

クロスモーダル融合モジュールは、異なるモダリティからの表現を統合するためのいくつかの方法を提供する。注意機構ベースの融合では、モダリティ間の関係性を捉えるために自己注意機構が使用される。連結ベースの融合では、すべてのモダリティの表現が連結され、線形変換を通じて統合される。ゲートベースの融合では、各モダリティに対してゲート値が計算され、それに基づいて重み付けされた表現が統合される。

より高度なクロスモーダル融合のために、階層的な融合アプローチも実装可能である。

```python
class HierarchicalCrossModalFusion(nn.Module):
    def __init__(self, hidden_dim, modality_pairs, fusion_type='attention'):
        super(HierarchicalCrossModalFusion, self).__init__()
        self.hidden_dim = hidden_dim
        self.modality_pairs = modality_pairs  # List of pairs of modalities to fuse

        # Create fusion modules for each pair
        self.fusion_modules = nn.ModuleDict({
            f"{pair[0]}_{pair[1]}": CrossModalFusion(hidden_dim, 2, fusion_type)
            for pair in modality_pairs
        })

        # Final fusion for all intermediate fusions
        self.final_fusion = CrossModalFusion(hidden_dim, len(modality_pairs), fusion_type)

    def forward(self, modality_representations_dict):
        # modality_representations_dict: dictionary mapping modality names to tensors

        # Intermediate fusions
        intermediate_fusions = {}
        for pair in self.modality_pairs:
            mod1, mod2 = pair
            fusion_key = f"{mod1}_{mod2}"
            fusion_input = [modality_representations_dict[mod1], modality_representations_dict[mod2]]
            intermediate_fusions[fusion_key] = self.fusion_modules[fusion_key](fusion_input)

        # Final fusion
        final_fusion_input = list(intermediate_fusions.values())
        output = self.final_fusion(final_fusion_input)

        return output
```

6.2.6 マルチモーダルエンコーダの統合

上記のサブコンポーネントを統合して、完全なマルチモーダルエンコーダを実装する。

```python
class MultimodalEncoder(nn.Module):
    def __init__(self, hidden_dim, modality_configs, fusion_type='attention'):
        super(MultimodalEncoder, self).__init__()
        self.hidden_dim = hidden_dim
        self.modality_configs = modality_configs

        # Create encoders for each modality
        self.encoders = nn.ModuleDict()
        for modality, config in modality_configs.items():
            if modality == 'text':
                self.encoders[modality] = PretrainedTextEncoder(config['model_name'], hidden_dim)
            elif modality == 'image':
                self.encoders[modality] = ImageEncoder(hidden_dim, config.get('backbone', 'resnet50'))
            elif modality == 'audio':
                self.encoders[modality] = AudioEncoder(hidden_dim, config.get('sample_rate', 16000))
            elif modality == 'graph':
                self.encoders[modality] = GraphEncoder(
```

```
                config['node_dim'], config.get('edge_dim', 0), hidden_dim, config.get('num_layers', 3)
            )
        elif modality == 'tabular':
            self.encoders[modality] = TabularEncoder(
                config['input_dims'], hidden_dim, config.get('categorical_cols'), config.get('numerical_cols')
            )

    # Cross-modal fusion
    self.fusion = CrossModalFusion(hidden_dim, len(modality_configs), fusion_type)

def forward(self, inputs):
    # inputs: dictionary mapping modality names to input tensors

    # Encode each modality
    modality_representations = {}
    for modality, encoder in self.encoders.items():
        if modality in inputs:
            if isinstance(inputs[modality], tuple) or isinstance(inputs[modality], list):
                # Handle complex inputs (e.g., for graphs)
                modality_representations[modality] = encoder(*inputs[modality])[0]
            else:
                # Handle simple inputs
                modality_representations[modality] = encoder(inputs[modality])[0]

    # Fusion
    fused_representation = self.fusion(list(modality_representations.values()))

    return fused_representation, modality_representations
```

このマルチモーダルエンコーダは、異なるモダリティの入力を受け取り、各モダリティに対応するエンコーダを使用して処理し、クロスモーダル融合モジュールを通じて統合された表現を生成する。

6.3. コンテキスト認識ネットワーク本体（CAN）の実装

コンテキスト認識ネットワーク本体は、マルチモーダルエンコーダからの統合表現を処理し、タスク固有の文脈を考慮した高次元の埋め込みを生成するコンポーネントである。

6.3.1 適応的処理深度機構の実装

適応的処理深度機構は、タスクの複雑さに応じて処理層の数や計算量を動的に調整する機構である。具体的な実装は以下の通りである。

```python
class ComplexityEstimator(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(ComplexityEstimator, self).__init__()
        self.estimator = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        # x: [batch_size, input_dim]
        complexity = self.estimator(x)  # [batch_size, 1]
        return complexity

class AdaptiveDepthLayer(nn.Module):
    def __init__(self, dim, ffn_dim, num_heads=8, dropout=0.1):
        super(AdaptiveDepthLayer, self).__init__()
        self.self_attn = nn.MultiheadAttention(dim, num_heads, dropout=dropout)
```

```
        self.norm1 = nn.LayerNorm(dim)

        self.ffn = nn.Sequential(
            nn.Linear(dim, ffn_dim),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(ffn_dim, dim)
        )
        self.norm2 = nn.LayerNorm(dim)

        # Gating mechanism
        self.gate = nn.Parameter(torch.ones(1))

    def forward(self, x, complexity=None):
        # x: [batch_size, dim]
        # complexity: [batch_size, 1] or None

        # Self-attention
        attn_output, _ = self.self_attn(x.unsqueeze(0), x.unsqueeze(0), x.unsqueeze(0))
        attn_output = attn_output.squeeze(0)

        # Apply gating based on complexity
        if complexity is not None:
            gate_value = torch.sigmoid(self.gate * complexity)
            x = x + gate_value * self.norm1(attn_output)
        else:
            x = x + self.norm1(attn_output)

        # Feed-forward network
        ffn_output = self.ffn(x)

        # Apply gating based on complexity
        if complexity is not None:
            gate_value = torch.sigmoid(self.gate * complexity)
            x = x + gate_value * self.norm2(ffn_output)
        else:
            x = x + self.norm2(ffn_output)

        return x

class AdaptiveDepthNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, ffn_dim, num_layers, num_heads=8, dropout=0.1):
        super(AdaptiveDepthNetwork, self).__init__()
        self.complexity_estimator = ComplexityEstimator(input_dim, hidden_dim)

        self.input_projection = nn.Linear(input_dim, hidden_dim)

        self.layers = nn.ModuleList([
            AdaptiveDepthLayer(hidden_dim, ffn_dim, num_heads, dropout)
            for _ in range(num_layers)
        ])

        self.output_projection = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Estimate complexity
        complexity = self.complexity_estimator(x)  # [batch_size, 1]

        # Input projection
        x = self.input_projection(x)  # [batch_size, hidden_dim]

        # Apply adaptive depth layers
        for layer in self.layers:
            x = layer(x, complexity)
```

```
        # Output projection
        output = self.output_projection(x)  # [batch_size, hidden_dim]

        return output
```

適応的処理深度機構は、複雑さ推定器と適応的深度層から構成される。複雑さ推定器は、入力の複雑さを推定し、0から1の値を出力する。適応的深度層は、この複雑さに基づいて、処理の深さを動的に調整する。複雑さが高い場合は、より多くの計算リソースが割り当てられ、複雑さが低い場合は、少ない計算リソースで処理が完了する。

6.3.2 コンテキストメモリの実装

コンテキストメモリは、過去の入力や処理結果を記憶し、現在の処理に活用する機構である。具体的な実装は以下の通りである。

```python
class ShortTermMemory(nn.Module):
    def __init__(self, hidden_dim, memory_size=10):
        super(ShortTermMemory, self).__init__()
        self.hidden_dim = hidden_dim
        self.memory_size = memory_size

        # Memory cells
        self.register_buffer('memory', torch.zeros(memory_size, hidden_dim))

        # Attention mechanism for memory access
        self.query_proj = nn.Linear(hidden_dim, hidden_dim)
        self.memory_proj = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, x, update_memory=True):
        # x: [batch_size, hidden_dim]
        batch_size = x.size(0)

        # Project query
        query = self.query_proj(x)  # [batch_size, hidden_dim]

        # Project memory
        memory_proj = self.memory_proj(self.memory)  # [memory_size, hidden_dim]

        # Compute attention scores
        scores = torch.matmul(query, memory_proj.transpose(0, 1))  # [batch_size, memory_size]
        attention = F.softmax(scores, dim=1)  # [batch_size, memory_size]

        # Retrieve from memory
        retrieved = torch.matmul(attention, self.memory)  # [batch_size, hidden_dim]

        # Update memory (if required)
        if update_memory and self.training:
            # Use the batch mean as the new memory entry
            new_memory = x.mean(dim=0, keepdim=True)  # [1, hidden_dim]

            # Shift memory (discard oldest entry)
            self.memory = torch.cat([new_memory, self.memory[:-1]], dim=0)

        return retrieved

class LongTermMemory(nn.Module):
    def __init__(self, hidden_dim, memory_size=100):
        super(LongTermMemory, self).__init__()
        self.hidden_dim = hidden_dim
        self.memory_size = memory_size
```

```
        # Memory cells
        self.register_buffer('memory', torch.zeros(memory_size, hidden_dim))
        self.register_buffer('importance', torch.zeros(memory_size))

        # Memory update mechanism
        self.importance_estimator = nn.Linear(hidden_dim, 1)

        # Attention mechanism for memory access
        self.query_proj = nn.Linear(hidden_dim, hidden_dim)
        self.memory_proj = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, x, update_memory=True):
        # x: [batch_size, hidden_dim]
        batch_size = x.size(0)

        # Project query
        query = self.query_proj(x)  # [batch_size, hidden_dim]

        # Project memory
        memory_proj = self.memory_proj(self.memory)  # [memory_size, hidden_dim]

        # Compute attention scores
        scores = torch.matmul(query, memory_proj.transpose(0, 1))  # [batch_size, memory_size]

        # Apply importance weighting
        weighted_scores = scores * self.importance.unsqueeze(0)  # [batch_size, memory_size]
        attention = F.softmax(weighted_scores, dim=1)  # [batch_size, memory_size]

        # Retrieve from memory
        retrieved = torch.matmul(attention, self.memory)  # [batch_size, hidden_dim]

        # Update memory (if required)
        if update_memory and self.training:
            # Compute importance of new entries
            new_importance = self.importance_estimator(x).squeeze(-1)  # [batch_size]

            # Select most important entry from the batch
            max_idx = new_importance.argmax()
            new_memory = x[max_idx:max_idx+1]  # [1, hidden_dim]
            new_memory_importance = new_importance[max_idx:max_idx+1]  # [1]

            # Find least important entry in memory
            min_idx = self.importance.argmin()

            # Replace if new entry is more important
            if new_memory_importance > self.importance[min_idx]:
                self.memory[min_idx] = new_memory
                self.importance[min_idx] = new_memory_importance

        return retrieved

class ContextMemory(nn.Module):
    def __init__(self, hidden_dim, short_term_size=10, long_term_size=100):
        super(ContextMemory, self).__init__()
        self.hidden_dim = hidden_dim

        # Short-term and long-term memory
        self.short_term = ShortTermMemory(hidden_dim, short_term_size)
        self.long_term = LongTermMemory(hidden_dim, long_term_size)

        # Memory integration
        self.integration = nn.Sequential(
            nn.Linear(hidden_dim * 3, hidden_dim * 2),
            nn.ReLU(),
            nn.Linear(hidden_dim * 2, hidden_dim)
```

```
        )

    def forward(self, x, update_memory=True):
        # x: [batch_size, hidden_dim]

        # Access short-term memory
        short_term_retrieved = self.short_term(x, update_memory)  # [batch_size, hidden_dim]

        # Access long-term memory
        long_term_retrieved = self.long_term(x, update_memory)  # [batch_size, hidden_dim]

        # Integrate current input with memory retrievals
        integrated = torch.cat([x, short_term_retrieved, long_term_retrieved], dim=1)  # [batch_size, hidden_dim * 3]
        output = self.integration(integrated)  # [batch_size, hidden_dim]

        return output
```

コンテキストメモリは、短期メモリと長期メモリから構成される。短期メモリは、最近の入力や処理結果を記憶し、FIFOキューとして動作する。長期メモリは、重要な情報を長期間にわたって記憶し、重要度に基づいて更新される。両方のメモリからの情報は、注意機構を通じて検索され、現在の入力と統合される。

6.3.3 不確実性認識層の実装

不確実性認識層は、入力や中間表現の不確実性を推定し、後続の処理に反映させる層である。具体的な実装は以下の通りである。

```python
class EpistemicUncertaintyEstimator(nn.Module):
    def __init__(self, input_dim, hidden_dim, dropout_rate=0.1, num_samples=10):
        super(EpistemicUncertaintyEstimator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.dropout_rate = dropout_rate
        self.num_samples = num_samples

        # Bayesian neural network
        self.bayesian_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(hidden_dim, hidden_dim)
        )

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Monte Carlo dropout sampling
        samples = []
        for _ in range(self.num_samples):
            samples.append(self.bayesian_net(x))

        # Stack samples
        stacked_samples = torch.stack(samples, dim=0)  # [num_samples, batch_size, hidden_dim]

        # Compute mean and variance
        mean = stacked_samples.mean(dim=0)  # [batch_size, hidden_dim]
        variance = stacked_samples.var(dim=0)  # [batch_size, hidden_dim]

        # Epistemic uncertainty is the variance
        epistemic_uncertainty = variance
```

```
        return mean, epistemic_uncertainty

class AleatoricUncertaintyEstimator(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(AleatoricUncertaintyEstimator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Mean and variance predictors
        self.mean_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim)
        )

        self.var_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.Softplus()  # Ensure positive variance
        )

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Predict mean and variance
        mean = self.mean_predictor(x)  # [batch_size, hidden_dim]
        variance = self.var_predictor(x)  # [batch_size, hidden_dim]

        # Aleatoric uncertainty is the predicted variance
        aleatoric_uncertainty = variance

        return mean, aleatoric_uncertainty

class UncertaintyAwareLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super(UncertaintyAwareLayer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Epistemic uncertainty estimator
        self.epistemic_estimator = EpistemicUncertaintyEstimator(input_dim, hidden_dim)

        # Aleatoric uncertainty estimator
        self.aleatoric_estimator = AleatoricUncertaintyEstimator(input_dim, hidden_dim)

        # Uncertainty-aware transformation
        self.transformation = nn.Sequential(
            nn.Linear(hidden_dim * 3, hidden_dim * 2),
            nn.ReLU(),
            nn.Linear(hidden_dim * 2, hidden_dim)
        )

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Estimate epistemic uncertainty
        epistemic_mean, epistemic_uncertainty = self.epistemic_estimator(x)

        # Estimate aleatoric uncertainty
        aleatoric_mean, aleatoric_uncertainty = self.aleatoric_estimator(x)

        # Combine means and uncertainties
        combined = torch.cat([
            epistemic_mean,
```

```
            epistemic_uncertainty,
            aleatoric_uncertainty
        ], dim=1)  # [batch_size, hidden_dim * 3]

        # Apply uncertainty-aware transformation
        output = self.transformation(combined)  # [batch_size, hidden_dim]

        # Total uncertainty is the sum of epistemic and aleatoric uncertainties
        total_uncertainty = epistemic_uncertainty + aleatoric_uncertainty

        return output, total_uncertainty
```

不確実性認識層は、認識的不確実性推定器と偶然的不確実性推定器から構成される。認識的不確実性は、知識や情報の不足による不確実性であり、モンテカルロドロップアウトサンプリングを通じて推定される。偶然的不確実性は、データの本質的なノイズやランダム性による不確実性であり、直接予測される。両方の不確実性は、後続の処理に反映され、不確実性の高い部分には、より慎重な処理が適用される。

6.3.4 コンテキスト認識ネットワーク本体の統合

上記のサブコンポーネントを統合して、完全なコンテキスト認識ネットワーク本体を実装する。

```python
class ContextAwareNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, ffn_dim, num_layers, short_term_size=10, long_term_size=100):
        super(ContextAwareNetwork, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Input projection
        self.input_projection = nn.Linear(input_dim, hidden_dim)

        # Adaptive depth network
        self.adaptive_depth = AdaptiveDepthNetwork(hidden_dim, hidden_dim, ffn_dim, num_layers)

        # Context memory
        self.context_memory = ContextMemory(hidden_dim, short_term_size, long_term_size)

        # Uncertainty-aware layer
        self.uncertainty_layer = UncertaintyAwareLayer(hidden_dim, hidden_dim)

        # Output projection
        self.output_projection = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, x, update_memory=True):
        # x: [batch_size, input_dim]

        # Input projection
        x = self.input_projection(x)  # [batch_size, hidden_dim]

        # Apply adaptive depth network
        x = self.adaptive_depth(x)  # [batch_size, hidden_dim]

        # Apply context memory
        x = self.context_memory(x, update_memory)  # [batch_size, hidden_dim]

        # Apply uncertainty-aware layer
        x, uncertainty = self.uncertainty_layer(x)  # [batch_size, hidden_dim], [batch_size, hidden_dim]

        # Output projection
        output = self.output_projection(x)  # [batch_size, hidden_dim]

        return output, uncertainty
```

コンテキスト認識ネットワーク本体は、入力投影層、適応的深度ネットワーク、コンテキストメモリ、不確実性認識層、出力投影層から構成される。入力はまず投影され、適応的深度ネットワークを通じて処理される。次に、コンテキストメモリを通じて過去の情報が統合され、不確実性認識層を通じて不確実性が推定される。最後に、出力投影層を通じて、最終的な埋め込みが生成される。

6.4. 出力生成層（OGL）の実装

出力生成層は、コンテキスト認識ネットワーク本体の埋め込みから直感的な出力を生成するコンポーネントである。具体的な実装は以下の通りである。

```python
class OutputGenerationLayer(nn.Module):
    def __init__(self, input_dim, output_dim, task_type='classification', num_classes=None):
        super(OutputGenerationLayer, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.task_type = task_type

        if task_type == 'classification':
            assert num_classes is not None, "Number of classes must be specified for classification tasks"
            self.output_head = nn.Linear(input_dim, num_classes)
        elif task_type == 'regression':
            self.output_head = nn.Linear(input_dim, output_dim)
        elif task_type == 'generation':
            # For sequence generation tasks
            self.output_head = nn.Linear(input_dim, output_dim)
            self.temperature = nn.Parameter(torch.ones(1) * 0.5)
        else:
            raise ValueError(f"Unsupported task type: {task_type}")

    def forward(self, x, temperature=None):
        # x: [batch_size, input_dim]

        if self.task_type == 'classification':
            logits = self.output_head(x)  # [batch_size, num_classes]
            return logits

        elif self.task_type == 'regression':
            output = self.output_head(x)  # [batch_size, output_dim]
            return output

        elif self.task_type == 'generation':
            logits = self.output_head(x)  # [batch_size, output_dim]

            # Apply temperature scaling
            if temperature is None:
                temperature = self.temperature

            scaled_logits = logits / temperature

            return scaled_logits
```

出力生成層は、タスクの種類に応じて異なる出力ヘッドを提供する。分類タスクでは、クラス数に応じたロジットが生成される。回帰タスクでは、指定された次元の出力値が生成される。生成タスクでは、温度スケーリングを適用したロジットが生成される。

6.5. 階層的反射モジュール（HRM）の実装

階層的反射モジュールは、コンテキスト認識ネットワーク本体の埋め込みから多層的な反射ベクトルを生成するコンポーネントである。

6.5.1 基本反射層の実装

基本反射層は、出力の論理的整合性を評価し、領域知識との不整合を検出する層である。具体的な実装は以下の通りである。

```python
class KnowledgeConsistencyChecker(nn.Module):
    def __init__(self, input_dim, hidden_dim, knowledge_base):
        super(KnowledgeConsistencyChecker, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.knowledge_base = knowledge_base

        # Neural consistency estimator
        self.consistency_estimator = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

    def forward(self, x, output):
        # x: [batch_size, input_dim]
        # output: [batch_size, output_dim]

        # Concatenate input and output
        combined = torch.cat([x, output], dim=1)  # [batch_size, input_dim + output_dim]

        # Estimate consistency
        consistency = self.consistency_estimator(combined)  # [batch_size, 1]

        # Convert to inconsistency (1 - consistency)
        inconsistency = 1 - consistency

        return inconsistency

class MultimodalConsistencyEvaluator(nn.Module):
    def __init__(self, input_dim, hidden_dim, modality_dims):
        super(MultimodalConsistencyEvaluator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.modality_dims = modality_dims

        # Pairwise consistency evaluators
        self.pairwise_evaluators = nn.ModuleDict()
        for mod1, dim1 in modality_dims.items():
            for mod2, dim2 in modality_dims.items():
                if mod1 < mod2:  # Avoid duplicates
                    key = f"{mod1}_{mod2}"
                    self.pairwise_evaluators[key] = nn.Sequential(
                        nn.Linear(dim1 + dim2, hidden_dim),
                        nn.ReLU(),
                        nn.Linear(hidden_dim, 1),
                        nn.Sigmoid()
                    )

        # Integration layer
        total_pairs = len(self.pairwise_evaluators)
        self.integration = nn.Sequential(
            nn.Linear(total_pairs, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
```

```
            nn.Sigmoid()
        )

    def forward(self, modality_representations, output):
        # modality_representations: dictionary mapping modality names to tensors
        # output: [batch_size, output_dim]

        # Compute pairwise consistencies
        pairwise_consistencies = []
        for mod1, rep1 in modality_representations.items():
            for mod2, rep2 in modality_representations.items():
                if mod1 < mod2:
                    key = f"{mod1}_{mod2}"
                    combined = torch.cat([rep1, rep2], dim=1)
                    consistency = self.pairwise_evaluators[key](combined)
                    pairwise_consistencies.append(consistency)

        # Stack pairwise consistencies
        stacked = torch.cat(pairwise_consistencies, dim=1)  # [batch_size, total_pairs]

        # Integrate pairwise consistencies
        overall_consistency = self.integration(stacked)  # [batch_size, 1]

        # Convert to inconsistency (1 - consistency)
        inconsistency = 1 - overall_consistency

        return inconsistency

class ReflectionMappingGenerator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(ReflectionMappingGenerator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Reflection mapping generator
        self.generator = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x, inconsistency):
        # x: [batch_size, input_dim]
        # inconsistency: [batch_size, 1]

        # Concatenate input and inconsistency
        combined = torch.cat([x, inconsistency], dim=1)  # [batch_size, input_dim + 1]

        # Generate reflection mapping
        reflection = self.generator(combined)  # [batch_size, output_dim]

        return reflection

class BasicReflectionLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, knowledge_base, modality_dims):
        super(BasicReflectionLayer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Knowledge consistency checker
```

```
        self.knowledge_checker = KnowledgeConsistencyChecker(input_dim + output_dim, hidden_dim,
knowledge_base)

        # Multimodal consistency evaluator
        self.multimodal_evaluator = MultimodalConsistencyEvaluator(input_dim, hidden_dim, modality_dims)

        # Reflection mapping generator
        self.mapping_generator = ReflectionMappingGenerator(input_dim + 2, hidden_dim, output_dim)

    def forward(self, x, output, modality_representations):
        # x: [batch_size, input_dim]
        # output: [batch_size, output_dim]
        # modality_representations: dictionary mapping modality names to tensors

        # Check knowledge consistency
        knowledge_inconsistency = self.knowledge_checker(x, output)  # [batch_size, 1]

        # Evaluate multimodal consistency
        multimodal_inconsistency = self.multimodal_evaluator(modality_representations, output)  # [batch_size, 1]

        # Generate reflection mapping
        combined_inconsistency = torch.cat([knowledge_inconsistency, multimodal_inconsistency], dim=1)  #
[batch_size, 2]
        reflection = self.mapping_generator(x, combined_inconsistency)  # [batch_size, output_dim]

        return reflection, knowledge_inconsistency, multimodal_inconsistency
```

基本反射層は、知識整合性チェッカー、マルチモーダル整合性評価器、反射マッピング生成器から構成される。知識整合性チェッカーは、出力が領域知識と整合しているかを評価する。マルチモーダル整合性評価器は、異なるモダリティからの情報が互いに整合しているかを評価する。反射マッピング生成器は、これらの評価に基づいて、反射ベクトルを生成する。

6.5.2 メタ認知層の実装

メタ認知層は、システム自体の推論プロセスを監視し、潜在的な推論エラーを検出する層である。具体的な実装は以下の通りである。

```python
class ReasoningPatternAnalyzer(nn.Module):
    def __init__(self, input_dim, hidden_dim, pattern_dim, num_patterns=10):
        super(ReasoningPatternAnalyzer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.pattern_dim = pattern_dim
        self.num_patterns = num_patterns

        # Pattern extractor
        self.pattern_extractor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, pattern_dim)
        )

        # Pattern memory
        self.register_buffer('pattern_memory', torch.zeros(num_patterns, pattern_dim))
        self.register_buffer('error_rates', torch.zeros(num_patterns))
        self.register_buffer('usage_counts', torch.zeros(num_patterns))

        # Pattern matcher
        self.pattern_matcher = nn.Sequential(
            nn.Linear(pattern_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
```

```
            nn.Sigmoid()
        )

    def update_memory(self, pattern, error_flag):
        # pattern: [batch_size, pattern_dim]
        # error_flag: [batch_size, 1]

        batch_size = pattern.size(0)

        for i in range(batch_size):
            # Find closest pattern in memory
            distances = torch.norm(self.pattern_memory - pattern[i:i+1], dim=1)
            closest_idx = distances.argmin()

            # Update pattern memory
            if distances[closest_idx] > 0.5:  # If no close match, find least used pattern
                closest_idx = self.usage_counts.argmin()
                self.pattern_memory[closest_idx] = pattern[i]
                self.error_rates[closest_idx] = error_flag[i].float()
                self.usage_counts[closest_idx] = 1
            else:  # Update existing pattern
                self.pattern_memory[closest_idx] = 0.9 * self.pattern_memory[closest_idx] + 0.1 * pattern[i]
                self.error_rates[closest_idx] = 0.9 * self.error_rates[closest_idx] + 0.1 * error_flag[i].float()
                self.usage_counts[closest_idx] += 1

    def forward(self, x, update_memory=False, error_flag=None):
        # x: [batch_size, input_dim]
        # error_flag: [batch_size, 1] or None

        # Extract pattern
        pattern = self.pattern_extractor(x)  # [batch_size, pattern_dim]

        # Update memory if required
        if update_memory and error_flag is not None and self.training:
            self.update_memory(pattern, error_flag)

        # Match pattern with memory
        batch_size = pattern.size(0)
        error_probs = []

        for i in range(self.num_patterns):
            # Expand memory pattern
            mem_pattern = self.pattern_memory[i:i+1].expand(batch_size, -1)  # [batch_size, pattern_dim]

            # Concatenate with current pattern
            combined = torch.cat([pattern, mem_pattern], dim=1)  # [batch_size, pattern_dim * 2]

            # Compute match probability
            match_prob = self.pattern_matcher(combined)  # [batch_size, 1]

            # Compute error probability based on match and error rate
            error_prob = match_prob * self.error_rates[i]
            error_probs.append(error_prob)

        # Combine error probabilities
        stacked_probs = torch.cat(error_probs, dim=1)  # [batch_size, num_patterns]
        max_error_prob, _ = stacked_probs.max(dim=1, keepdim=True)  # [batch_size, 1]

        return max_error_prob, pattern

class BiasDetector(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_biases=5):
        super(BiasDetector, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.num_biases = num_biases
```

```
        # Bias detectors
        self.bias_detectors = nn.ModuleList([
            nn.Sequential(
                nn.Linear(input_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, 1),
                nn.Sigmoid()
            )
            for _ in range(num_biases)
        ])

        # Bias names (for interpretability)
        self.bias_names = [
            "confirmation_bias",
            "anchoring_effect",
            "availability_heuristic",
            "representativeness_heuristic",
            "overconfidence_bias"
        ]

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Detect biases
        bias_probs = []
        for detector in self.bias_detectors:
            bias_prob = detector(x)  # [batch_size, 1]
            bias_probs.append(bias_prob)

        # Stack bias probabilities
        stacked_probs = torch.cat(bias_probs, dim=1)  # [batch_size, num_biases]

        # Compute overall bias probability
        overall_bias_prob = stacked_probs.mean(dim=1, keepdim=True)  # [batch_size, 1]

        return overall_bias_prob, stacked_probs

class SelfMonitoringMechanism(nn.Module):
    def __init__(self, input_dim, hidden_dim, activation_threshold=0.5):
        super(SelfMonitoringMechanism, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.activation_threshold = activation_threshold

        # Activation pattern analyzer
        self.activation_analyzer = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

        # Distribution analyzer
        self.distribution_analyzer = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

    def forward(self, x, activations=None):
```

```
        # x: [batch_size, input_dim]
        # activations: list of activation tensors or None

        # Analyze input distribution
        distribution_anomaly = self.distribution_analyzer(x)  # [batch_size, 1]

        # Analyze activation patterns (if provided)
        if activations is not None:
            # Flatten and concatenate activations
            flat_activations = []
            for act in activations:
                # Apply threshold and flatten
                thresholded = (act > self.activation_threshold).float()
                flat_activations.append(thresholded.flatten(1))

            # Concatenate along feature dimension
            concat_activations = torch.cat(flat_activations, dim=1)

            # Analyze activation pattern
            activation_anomaly = self.activation_analyzer(concat_activations)  # [batch_size, 1]
        else:
            activation_anomaly = torch.zeros_like(distribution_anomaly)

        # Combine anomalies
        combined_anomaly = torch.max(distribution_anomaly, activation_anomaly)

        return combined_anomaly

class MetacognitionLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, pattern_dim, output_dim, num_patterns=10, num_biases=5):
        super(MetacognitionLayer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.pattern_dim = pattern_dim
        self.output_dim = output_dim

        # Reasoning pattern analyzer
        self.pattern_analyzer = ReasoningPatternAnalyzer(input_dim, hidden_dim, pattern_dim, num_patterns)

        # Bias detector
        self.bias_detector = BiasDetector(input_dim, hidden_dim, num_biases)

        # Self-monitoring mechanism
        self.self_monitor = SelfMonitoringMechanism(input_dim, hidden_dim)

        # Integration layer
        self.integration = nn.Sequential(
            nn.Linear(3, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )

        # Reflection mapping generator
        self.mapping_generator = ReflectionMappingGenerator(input_dim + 1, hidden_dim, output_dim)

    def forward(self, x, output, activations=None, update_memory=False, error_flag=None):
        # x: [batch_size, input_dim]
        # output: [batch_size, output_dim]
        # activations: list of activation tensors or None
        # error_flag: [batch_size, 1] or None

        # Analyze reasoning patterns
        pattern_error_prob, pattern = self.pattern_analyzer(x, update_memory, error_flag)  # [batch_size, 1], [batch_size,
pattern_dim]
```

```
        # Detect biases
        bias_prob, detailed_bias_probs = self.bias_detector(x)  # [batch_size, 1], [batch_size, num_biases]

        # Monitor self
        anomaly_prob = self.self_monitor(x, activations)  # [batch_size, 1]

        # Integrate metacognitive assessments
        metacognitive_probs = torch.cat([pattern_error_prob, bias_prob, anomaly_prob], dim=1)  # [batch_size, 3]
        metacognitive_error_prob = self.integration(metacognitive_probs)  # [batch_size, 1]

        # Generate reflection mapping
        reflection = self.mapping_generator(x, metacognitive_error_prob)  # [batch_size, output_dim]

        return reflection, metacognitive_error_prob, detailed_bias_probs
```

メタ認知層は、推論パターン分析器、バイアス検出器、自己モニタリング機構から構成される。推論パターン分析器は、過去の推論プロセスのパターンを分析し、エラーが発生しやすい状況や条件を特定する。バイアス検出器は、認知バイアスや推論の偏りを検出し、それらが結果に与える影響を評価する。自己モニタリング機構は、システム自体の推論プロセスをリアルタイムで監視し、異常や不整合を検出する。これらの評価に基づいて、メタ認知層は反射ベクトルを生成する。

6.5.3 不確実性推定層の実装

不確実性推定層は、出力の各部分の不確実性を推定し、高い不確実性を持つ部分を識別する層である。具体的な実装は以下の通りである。

```python
class BayesianUncertaintyEstimator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.1, num_samples=10):
        super(BayesianUncertaintyEstimator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.dropout_rate = dropout_rate
        self.num_samples = num_samples

        # Bayesian neural network
        self.bayesian_net = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Enable dropout at inference time
        self.bayesian_net.train()

        # Monte Carlo dropout sampling
        samples = []
        for _ in range(self.num_samples):
            samples.append(self.bayesian_net(x))

        # Stack samples
        stacked_samples = torch.stack(samples, dim=0)  # [num_samples, batch_size, output_dim]

        # Compute mean and variance
```

```
        mean = stacked_samples.mean(dim=0)  # [batch_size, output_dim]
        variance = stacked_samples.var(dim=0)  # [batch_size, output_dim]

        return mean, variance

class EnsembleUncertaintyEstimator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_models=5):
        super(EnsembleUncertaintyEstimator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.num_models = num_models

        # Ensemble of models
        self.models = nn.ModuleList([
            nn.Sequential(
                nn.Linear(input_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.Linear(hidden_dim, output_dim)
            )
            for _ in range(num_models)
        ])

    def forward(self, x):
        # x: [batch_size, input_dim]

        # Get predictions from each model
        predictions = []
        for model in self.models:
            predictions.append(model(x))

        # Stack predictions
        stacked_predictions = torch.stack(predictions, dim=0)  # [num_models, batch_size, output_dim]

        # Compute mean and variance
        mean = stacked_predictions.mean(dim=0)  # [batch_size, output_dim]
        variance = stacked_predictions.var(dim=0)  # [batch_size, output_dim]

        return mean, variance

class UncertaintyDecompositionAnalyzer(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(UncertaintyDecompositionAnalyzer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Mean predictor
        self.mean_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )

        # Aleatoric uncertainty predictor
        self.aleatoric_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
            nn.Softplus()  # Ensure positive variance
        )

    def forward(self, x, epistemic_variance):
        # x: [batch_size, input_dim]
```

```
        # epistemic_variance: [batch_size, output_dim]

        # Predict mean
        mean = self.mean_predictor(x)  # [batch_size, output_dim]

        # Predict aleatoric uncertainty
        aleatoric_variance = self.aleatoric_predictor(x)  # [batch_size, output_dim]

        # Total variance is the sum of epistemic and aleatoric variances
        total_variance = epistemic_variance + aleatoric_variance

        return mean, aleatoric_variance, total_variance

class UncertaintyEstimationLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.1, num_samples=10, num_models=5):
        super(UncertaintyEstimationLayer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Bayesian uncertainty estimator
        self.bayesian_estimator = BayesianUncertaintyEstimator(input_dim, hidden_dim, output_dim, dropout_rate,
num_samples)

        # Ensemble uncertainty estimator
        self.ensemble_estimator = EnsembleUncertaintyEstimator(input_dim, hidden_dim, output_dim, num_models)

        # Uncertainty decomposition analyzer
        self.decomposition_analyzer = UncertaintyDecompositionAnalyzer(input_dim, hidden_dim, output_dim)

        # Reflection mapping generator
        self.mapping_generator = nn.Sequential(
            nn.Linear(input_dim + output_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x, output):
        # x: [batch_size, input_dim]
        # output: [batch_size, output_dim]

        # Estimate uncertainty using Bayesian approach
        bayesian_mean, bayesian_variance = self.bayesian_estimator(x)  # [batch_size, output_dim], [batch_size,
output_dim]

        # Estimate uncertainty using ensemble approach
        ensemble_mean, ensemble_variance = self.ensemble_estimator(x)  # [batch_size, output_dim], [batch_size,
output_dim]

        # Combine epistemic uncertainties
        epistemic_variance = (bayesian_variance + ensemble_variance) / 2  # [batch_size, output_dim]

        # Decompose uncertainty
        mean, aleatoric_variance, total_variance = self.decomposition_analyzer(x, epistemic_variance)  # [batch_size,
output_dim], [batch_size, output_dim], [batch_size, output_dim]

        # Generate reflection mapping
        combined = torch.cat([x, total_variance], dim=1)  # [batch_size, input_dim + output_dim]
        reflection = self.mapping_generator(combined)  # [batch_size, output_dim]

        return reflection, epistemic_variance, aleatoric_variance, total_variance
```

不確実性推定層は、ベイズ不確実性推定器、アンサンブル不確実性推定器、不確実性分解分析器から構成される。ベイズ不確実性推定器は、ベイズ推論の枠組みを使用して、出力の確率分布とその不確実性を推定する。アンサンブル不確実性推定器は、複数のモデルからの予測を集約し、その分散や不一致を不確実性の指標として使用する。不確実性分解分析器は、全体の不確実性を認識的不確実性と偶然的不確実性に分解し、それぞれに適した対処法を適用する。これらの評価に基づいて、不確実性推定層は反射ベクトルを生成する。

6.5.4 統合層の実装

統合層は、上記の層からの評価を統合し、最終的な反射ベクトルを生成する層である。具体的な実装は以下の通りである。

```python
class WeightedIntegrator(nn.Module):
    def __init__(self, input_dim, hidden_dim, num_layers):
        super(WeightedIntegrator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        # Weight predictor
        self.weight_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_layers),
            nn.Softmax(dim=1)
        )

    def forward(self, x, layer_outputs):
        # x: [batch_size, input_dim]
        # layer_outputs: list of tensors, each with shape [batch_size, output_dim]

        # Predict weights
        weights = self.weight_predictor(x)  # [batch_size, num_layers]

        # Apply weights
        weighted_outputs = []
        for i, output in enumerate(layer_outputs):
            weighted = output * weights[:, i:i+1]
            weighted_outputs.append(weighted)

        # Sum weighted outputs
        integrated = sum(weighted_outputs)  # [batch_size, output_dim]

        return integrated, weights

class NonlinearIntegrator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers):
        super(NonlinearIntegrator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.num_layers = num_layers

        # Nonlinear integrator
        self.integrator = nn.Sequential(
            nn.Linear(input_dim + output_dim * num_layers, hidden_dim * 2),
            nn.ReLU(),
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )
```

```
    def forward(self, x, layer_outputs):
        # x: [batch_size, input_dim]
        # layer_outputs: list of tensors, each with shape [batch_size, output_dim]

        # Concatenate input and layer outputs
        concat = [x]
        for output in layer_outputs:
            concat.append(output)

        combined = torch.cat(concat, dim=1)  # [batch_size, input_dim + output_dim * num_layers]

        # Apply nonlinear integration
        integrated = self.integrator(combined)  # [batch_size, output_dim]

        return integrated

class ContextConditionedIntegrator(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers, num_contexts=3):
        super(ContextConditionedIntegrator, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.num_layers = num_layers
        self.num_contexts = num_contexts

        # Context predictor
        self.context_predictor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_contexts),
            nn.Softmax(dim=1)
        )

        # Context-specific integrators
        self.context_integrators = nn.ModuleList([
            WeightedIntegrator(input_dim, hidden_dim, num_layers)
            for _ in range(num_contexts)
        ])

    def forward(self, x, layer_outputs):
        # x: [batch_size, input_dim]
        # layer_outputs: list of tensors, each with shape [batch_size, output_dim]

        # Predict context weights
        context_weights = self.context_predictor(x)  # [batch_size, num_contexts]

        # Apply context-specific integrators
        context_outputs = []
        all_weights = []
        for i, integrator in enumerate(self.context_integrators):
            output, weights = integrator(x, layer_outputs)
            context_outputs.append(output)
            all_weights.append(weights)

        # Stack context outputs
        stacked_outputs = torch.stack(context_outputs, dim=1)  # [batch_size, num_contexts, output_dim]

        # Apply context weights
        expanded_weights = context_weights.unsqueeze(-1)  # [batch_size, num_contexts, 1]
        integrated = (stacked_outputs * expanded_weights).sum(dim=1)  # [batch_size, output_dim]

        return integrated, context_weights, all_weights

class IntegrationLayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, integration_type='weighted'):
```

```
        super(IntegrationLayer, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.integration_type = integration_type

        # Number of reflection layers
        self.num_layers = 3  # Basic, Metacognition, Uncertainty

        if integration_type == 'weighted':
            self.integrator = WeightedIntegrator(input_dim, hidden_dim, self.num_layers)
        elif integration_type == 'nonlinear':
            self.integrator = NonlinearIntegrator(input_dim, hidden_dim, output_dim, self.num_layers)
        elif integration_type == 'context':
            self.integrator = ContextConditionedIntegrator(input_dim, hidden_dim, output_dim, self.num_layers)
        else:
            raise ValueError(f"Unsupported integration type: {integration_type}")

    def forward(self, x, basic_reflection, metacognition_reflection, uncertainty_reflection):
        # x: [batch_size, input_dim]
        # basic_reflection: [batch_size, output_dim]
        # metacognition_reflection: [batch_size, output_dim]
        # uncertainty_reflection: [batch_size, output_dim]

        # Collect layer outputs
        layer_outputs = [basic_reflection, metacognition_reflection, uncertainty_reflection]

        # Apply integration
        if self.integration_type == 'weighted':
            integrated, weights = self.integrator(x, layer_outputs)
            return integrated, weights
        elif self.integration_type == 'nonlinear':
            integrated = self.integrator(x, layer_outputs)
            return integrated
        elif self.integration_type == 'context':
            integrated, context_weights, layer_weights = self.integrator(x, layer_outputs)
            return integrated, context_weights, layer_weights
        else:
            # Simple average
            integrated = sum(layer_outputs) / len(layer_outputs)
            return integrated
```

統合層は、重み付け統合器、非線形統合器、コンテキスト条件付き統合器から構成される。重み付け統合器は、各層からの評価に対して、タスクや文脈に応じた重みを割り当て、重み付け平均を計算する。非線形統合器は、各層からの評価を非線形関数を通じて統合し、複雑な相互作用を捉える。コンテキスト条件付き統合器は、タスクや入力の文脈に応じて、統合方法自体を動的に変更する。これらの統合方法に基づいて、統合層は最終的な反射ベクトルを生成する。

6.5.5 階層的反射モジュールの統合

上記のサブコンポーネントを統合して、完全な階層的反射モジュールを実装する。

```python
class HierarchicalReflectionModule(nn.Module):
    def __init__(self, input_dim, hidden_dim, pattern_dim, output_dim, knowledge_base, modality_dims,
integration_type='weighted'):
        super(HierarchicalReflectionModule, self).__init__()
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.pattern_dim = pattern_dim
        self.output_dim = output_dim

        # Basic reflection layer
```

```
        self.basic_layer = BasicReflectionLayer(input_dim, hidden_dim, output_dim, knowledge_base, modality_dims)

        # Metacognition layer
        self.metacognition_layer = MetacognitionLayer(input_dim, hidden_dim, pattern_dim, output_dim)

        # Uncertainty estimation layer
        self.uncertainty_layer = UncertaintyEstimationLayer(input_dim, hidden_dim, output_dim)

        # Integration layer
        self.integration_layer = IntegrationLayer(input_dim, hidden_dim, output_dim, integration_type)

    def forward(self, x, output, modality_representations, activations=None, update_memory=False, error_flag=None):
        # x: [batch_size, input_dim]
        # output: [batch_size, output_dim]
        # modality_representations: dictionary mapping modality names to tensors
        # activations: list of activation tensors or None
        # error_flag: [batch_size, 1] or None

        # Apply basic reflection layer
        basic_reflection, knowledge_inconsistency, multimodal_inconsistency = self.basic_layer(x, output,
modality_representations)

        # Apply metacognition layer
        metacognition_reflection, metacognitive_error_prob, detailed_bias_probs = self.metacognition_layer(x, output,
activations, update_memory, error_flag)

        # Apply uncertainty estimation layer
        uncertainty_reflection, epistemic_variance, aleatoric_variance, total_variance = self.uncertainty_layer(x, output)

        # Apply integration layer
        if self.integration_layer.integration_type == 'weighted':
            integrated_reflection, weights = self.integration_layer(x, basic_reflection, metacognition_reflection,
uncertainty_reflection)
            return integrated_reflection, {
                'basic_reflection': basic_reflection,
                'metacognition_reflection': metacognition_reflection,
                'uncertainty_reflection': uncertainty_reflection,
                'knowledge_inconsistency': knowledge_inconsistency,
                'multimodal_inconsistency': multimodal_inconsistency,
                'metacognitive_error_prob': metacognitive_error_prob,
                'detailed_bias_probs': detailed_bias_probs,
                'epistemic_variance': epistemic_variance,
                'aleatoric_variance': aleatoric_variance,
                'total_variance': total_variance,
                'layer_weights': weights
            }
        elif self.integration_layer.integration_type == 'nonlinear':
            integrated_reflection = self.integration_layer(x, basic_reflection, metacognition_reflection,
uncertainty_reflection)
            return integrated_reflection, {
                'basic_reflection': basic_reflection,
                'metacognition_reflection': metacognition_reflection,
                'uncertainty_reflection': uncertainty_reflection,
                'knowledge_inconsistency': knowledge_inconsistency,
                'multimodal_inconsistency': multimodal_inconsistency,
                'metacognitive_error_prob': metacognitive_error_prob,
                'detailed_bias_probs': detailed_bias_probs,
                'epistemic_variance': epistemic_variance,
                'aleatoric_variance': aleatoric_variance,
                'total_variance': total_variance
            }
        elif self.integration_layer.integration_type == 'context':
            integrated_reflection, context_weights, layer_weights = self.integration_layer(x, basic_reflection,
metacognition_reflection, uncertainty_reflection)
            return integrated_reflection, {
                'basic_reflection': basic_reflection,
```

```
            'metacognition_reflection': metacognition_reflection,
            'uncertainty_reflection': uncertainty_reflection,
            'knowledge_inconsistency': knowledge_inconsistency,
            'multimodal_inconsistency': multimodal_inconsistency,
            'metacognitive_error_prob': metacognitive_error_prob,
            'detailed_bias_probs': detailed_bias_probs,
            'epistemic_variance': epistemic_variance,
            'aleatoric_variance': aleatoric_variance,
            'total_variance': total_variance,
            'context_weights': context_weights,
            'layer_weights': layer_weights
        }
    else:
        integrated_reflection = self.integration_layer(x, basic_reflection, metacognition_reflection,
uncertainty_reflection)
        return integrated_reflection, {
            'basic_reflection': basic_reflection,
            'metacognition_reflection': metacognition_reflection,
            'uncertainty_reflection': uncertainty_reflection,
            'knowledge_inconsistency': knowledge_inconsistency,
            'multimodal_inconsistency': multimodal_inconsistency,
            'metacognitive_error_prob': metacognitive_error_prob,
            'detailed_bias_probs': detailed_bias_probs,
            'epistemic_variance': epistemic_variance,
            'aleatoric_variance': aleatoric_variance,
            'total_variance': total_variance
        }
```

階層的反射モジュールは、基本反射層、メタ認知層、不確実性推定層、統合層から構成される。入力と出力を受け取り、各層を通じて反射ベクトルを生成し、統合層を通じて最終的な反射ベクトルを生成する。また、各層からの詳細な評価情報も提供する。

6.6. 拡張可能知識ベース（EKB）の実装

拡張可能知識ベースは、領域知識を表現し、記号的推論を実行するとともに、新しい知識を継続的に統合する機能を持つコンポーネントである。

6.6.1 多様な知識表現の実装

多様な知識表現は、命題論理、一階論理、確率論理、知識グラフなど、複数の形式で知識を表現できる機能である。具体的な実装は以下の通りである。

```python
class PropositionalLogicRepresentation:
    def __init__(self):
        self.rules = []

    def add_rule(self, rule):
        """
        Add a propositional logic rule.

        Args:
            rule: A string representing a propositional logic rule.
        """
        self.rules.append(rule)

    def get_rules(self):
        """
        Get all propositional logic rules.

        Returns:
            A list of strings representing propositional logic rules.
```

```
        """
        return self.rules

    def to_cnf(self):
        """
        Convert rules to conjunctive normal form (CNF).

        Returns:
            A list of clauses in CNF.
        """
        # Implementation depends on the specific propositional logic library
        pass

class FirstOrderLogicRepresentation:
    def __init__(self):
        self.facts = []
        self.rules = []

    def add_fact(self, fact):
        """
        Add a first-order logic fact.

        Args:
            fact: A string representing a first-order logic fact.
        """
        self.facts.append(fact)

    def add_rule(self, rule):
        """
        Add a first-order logic rule.

        Args:
            rule: A string representing a first-order logic rule.
        """
        self.rules.append(rule)

    def get_facts(self):
        """
        Get all first-order logic facts.

        Returns:
            A list of strings representing first-order logic facts.
        """
        return self.facts

    def get_rules(self):
        """
        Get all first-order logic rules.

        Returns:
            A list of strings representing first-order logic rules.
        """
        return self.rules

class ProbabilisticLogicRepresentation:
    def __init__(self):
        self.facts = []
        self.rules = []

    def add_fact(self, fact, probability):
        """
        Add a probabilistic fact.

        Args:
            fact: A string representing a fact.
            probability: A float representing the probability of the fact.
```

```
        """
        self.facts.append((fact, probability))

    def add_rule(self, rule, probability):
        """
        Add a probabilistic rule.

        Args:
            rule: A string representing a rule.
            probability: A float representing the probability of the rule.
        """
        self.rules.append((rule, probability))

    def get_facts(self):
        """
        Get all probabilistic facts.

        Returns:
            A list of tuples (fact, probability).
        """
        return self.facts

    def get_rules(self):
        """
        Get all probabilistic rules.

        Returns:
            A list of tuples (rule, probability).
        """
        return self.rules

class KnowledgeGraphRepresentation:
    def __init__(self):
        self.triples = []

    def add_triple(self, subject, predicate, object):
        """
        Add a knowledge graph triple.

        Args:
            subject: A string representing the subject entity.
            predicate: A string representing the predicate relation.
            object: A string representing the object entity.
        """
        self.triples.append((subject, predicate, object))

    def get_triples(self):
        """
        Get all knowledge graph triples.

        Returns:
            A list of tuples (subject, predicate, object).
        """
        return self.triples

    def get_entities(self):
        """
        Get all entities in the knowledge graph.

        Returns:
            A set of strings representing entities.
        """
        entities = set()
        for subject, _, object in self.triples:
            entities.add(subject)
            entities.add(object)
```

```
        return entities

    def get_relations(self):
        """
        Get all relations in the knowledge graph.

        Returns:
            A set of strings representing relations.
        """
        relations = set()
        for _, predicate, _ in self.triples:
            relations.add(predicate)
        return relations

class OntologyRepresentation:
    def __init__(self):
        self.concepts = {}
        self.relations = {}

    def add_concept(self, concept, parent=None):
        """
        Add a concept to the ontology.

        Args:
            concept: A string representing the concept.
            parent: A string representing the parent concept, or None.
        """
        if concept not in self.concepts:
            self.concepts[concept] = {
                'parent': parent,
                'children': [],
                'attributes': {}
            }

        if parent is not None and parent in self.concepts:
            self.concepts[parent]['children'].append(concept)

    def add_attribute(self, concept, attribute, value):
        """
        Add an attribute to a concept.

        Args:
            concept: A string representing the concept.
            attribute: A string representing the attribute.
            value: The value of the attribute.
        """
        if concept in self.concepts:
            self.concepts[concept]['attributes'][attribute] = value

    def add_relation(self, relation, domain, range):
        """
        Add a relation to the ontology.

        Args:
            relation: A string representing the relation.
            domain: A string representing the domain concept.
            range: A string representing the range concept.
        """
        self.relations[relation] = {
            'domain': domain,
            'range': range
        }

    def get_concepts(self):
        """
        Get all concepts in the ontology.
```

```
        Returns:
            A dictionary mapping concept names to concept information.
        """
        return self.concepts

    def get_relations(self):
        """
        Get all relations in the ontology.

        Returns:
            A dictionary mapping relation names to relation information.
        """
        return self.relations

    def get_parents(self, concept):
        """
        Get all parents of a concept.

        Args:
            concept: A string representing the concept.

        Returns:
            A list of strings representing parent concepts.
        """
        parents = []
        current = concept
        while current in self.concepts and self.concepts[current]['parent'] is not None:
            current = self.concepts[current]['parent']
            parents.append(current)
        return parents

    def is_a(self, concept1, concept2):
        """
        Check if concept1 is a subclass of concept2.

        Args:
            concept1: A string representing the first concept.
            concept2: A string representing the second concept.

        Returns:
            A boolean indicating whether concept1 is a subclass of concept2.
        """
        return concept2 in self.get_parents(concept1)
```

多様な知識表現は、命題論理表現、一階論理表現、確率論理表現、知識グラフ表現、オントロジー表現などのクラスを提供する。これらのクラスは、それぞれの形式で知識を表現し、操作するためのメソッドを提供する。

6.6.2 ハイブリッド推論エンジンの実装

ハイブリッド推論エンジンは、異なる推論メカニズム（演繹、帰納、アブダクション、確率推論）を統合し、状況に応じて適切な推論方法を選択する機能である。具体的な実装は以下の通りである。

```python
class DeductiveReasoningEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def reason(self, query):
        """
        Perform deductive reasoning.
```

```
        Args:
            query: A query to be answered.

        Returns:
            The result of the deductive reasoning.
        """
        # Implementation depends on the specific deductive reasoning library
        pass

class InductiveReasoningEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def reason(self, examples):
        """
        Perform inductive reasoning.

        Args:
            examples: A list of examples to learn from.

        Returns:
            The result of the inductive reasoning.
        """
        # Implementation depends on the specific inductive reasoning library
        pass

class AbductiveReasoningEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def reason(self, observation):
        """
        Perform abductive reasoning.

        Args:
            observation: An observation to be explained.

        Returns:
            The result of the abductive reasoning.
        """
        # Implementation depends on the specific abductive reasoning library
        pass

class ProbabilisticReasoningEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def reason(self, query, evidence=None):
        """
        Perform probabilistic reasoning.

        Args:
            query: A query to be answered.
            evidence: Evidence to condition on, or None.

        Returns:
            The result of the probabilistic reasoning.
        """
        # Implementation depends on the specific probabilistic reasoning library
        pass

class HybridReasoningEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base
        self.deductive_engine = DeductiveReasoningEngine(knowledge_base)
        self.inductive_engine = InductiveReasoningEngine(knowledge_base)
```

```
        self.abductive_engine = AbductiveReasoningEngine(knowledge_base)
        self.probabilistic_engine = ProbabilisticReasoningEngine(knowledge_base)

    def reason(self, query, reasoning_type=None, **kwargs):
        """
        Perform reasoning.

        Args:
            query: A query to be answered.
            reasoning_type: The type of reasoning to perform, or None to automatically select.
            **kwargs: Additional arguments for the specific reasoning engine.

        Returns:
            The result of the reasoning.
        """
        if reasoning_type == 'deductive':
            return self.deductive_engine.reason(query, **kwargs)
        elif reasoning_type == 'inductive':
            return self.inductive_engine.reason(query, **kwargs)
        elif reasoning_type == 'abductive':
            return self.abductive_engine.reason(query, **kwargs)
        elif reasoning_type == 'probabilistic':
            return self.probabilistic_engine.reason(query, **kwargs)
        else:
            # Automatically select the most appropriate reasoning type
            # This is a simplified implementation
            if 'examples' in kwargs:
                return self.inductive_engine.reason(query, **kwargs)
            elif 'observation' in kwargs:
                return self.abductive_engine.reason(query, **kwargs)
            elif 'evidence' in kwargs:
                return self.probabilistic_engine.reason(query, **kwargs)
            else:
                return self.deductive_engine.reason(query, **kwargs)
```

ハイブリッド推論エンジンは、演繹推論エンジン、帰納推論エンジン、アブダクション推論エンジン、確率推論エンジンから構成される。これらのエンジンは、それぞれの推論メカニズムを実装し、ハイブリッド推論エンジンは、状況に応じて適切な推論エンジンを選択する。

6.6.3 知識獲得モジュールの実装

知識獲得モジュールは、新しい経験や観察から知識を抽出し、既存の知識ベースに統合する機能である。具体的な実装は以下の通りである。

```python
class PatternExtractor:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def extract_patterns(self, data):
        """
        Extract patterns from data.

        Args:
            data: The data to extract patterns from.

        Returns:
            A list of extracted patterns.
        """
        # Implementation depends on the specific pattern extraction algorithm
        pass

class KnowledgeIntegrator:
```

```
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def integrate_knowledge(self, new_knowledge):
        """
        Integrate new knowledge into the knowledge base.

        Args:
            new_knowledge: The new knowledge to integrate.

        Returns:
            The updated knowledge base.
        """
        # Implementation depends on the specific knowledge integration algorithm
        pass

class ConflictDetector:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def detect_conflicts(self, new_knowledge):
        """
        Detect conflicts between new knowledge and existing knowledge.

        Args:
            new_knowledge: The new knowledge to check for conflicts.

        Returns:
            A list of detected conflicts.
        """
        # Implementation depends on the specific conflict detection algorithm
        pass

class ConflictResolver:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def resolve_conflicts(self, conflicts):
        """
        Resolve conflicts in the knowledge base.

        Args:
            conflicts: A list of conflicts to resolve.

        Returns:
            The updated knowledge base.
        """
        # Implementation depends on the specific conflict resolution algorithm
        pass

class KnowledgeAcquisitionModule:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base
        self.pattern_extractor = PatternExtractor(knowledge_base)
        self.knowledge_integrator = KnowledgeIntegrator(knowledge_base)
        self.conflict_detector = ConflictDetector(knowledge_base)
        self.conflict_resolver = ConflictResolver(knowledge_base)

    def acquire_knowledge(self, data):
        """
        Acquire knowledge from data.

        Args:
            data: The data to acquire knowledge from.

        Returns:
```

```
        The updated knowledge base.
        """
        # Extract patterns from data
        patterns = self.pattern_extractor.extract_patterns(data)

        # Detect conflicts with existing knowledge
        conflicts = self.conflict_detector.detect_conflicts(patterns)

        # Resolve conflicts
        if conflicts:
            self.conflict_resolver.resolve_conflicts(conflicts)

        # Integrate new knowledge
        self.knowledge_integrator.integrate_knowledge(patterns)

        return self.knowledge_base
```

知識獲得モジュールは、パターン抽出器、知識統合器、矛盾検出器、矛盾解決器から構成される。パターン抽出器は、データからパターンや規則性を抽出する。知識統合器は、抽出されたパターンや規則を既存の知識ベースに統合する。矛盾検出器は、新しい知識と既存の知識の間の矛盾を検出する。矛盾解決器は、検出された矛盾を解決する。

6.6.4 知識検証モジュールの実装

知識検証モジュールは、知識の信頼性や適用可能性を評価し、不確かな知識に適切な信頼度を割り当てる機能である。具体的な実装は以下の通りである。

```python
class ReliabilityEvaluator:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def evaluate_reliability(self, knowledge):
        """
        Evaluate the reliability of knowledge.

        Args:
            knowledge: The knowledge to evaluate.

        Returns:
            A float representing the reliability score.
        """
        # Implementation depends on the specific reliability evaluation algorithm
        pass

class ScopeAnalyzer:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def analyze_scope(self, knowledge):
        """
        Analyze the scope of applicability of knowledge.

        Args:
            knowledge: The knowledge to analyze.

        Returns:
            A description of the scope of applicability.
        """
        # Implementation depends on the specific scope analysis algorithm
        pass

class ConfidenceAssigner:
```

```
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def assign_confidence(self, knowledge, reliability, scope):
        """
        Assign a confidence score to knowledge.

        Args:
            knowledge: The knowledge to assign confidence to.
            reliability: The reliability score of the knowledge.
            scope: The scope of applicability of the knowledge.

        Returns:
            The knowledge with an assigned confidence score.
        """
        # Implementation depends on the specific confidence assignment algorithm
        pass

class KnowledgeVerificationModule:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base
        self.reliability_evaluator = ReliabilityEvaluator(knowledge_base)
        self.scope_analyzer = ScopeAnalyzer(knowledge_base)
        self.confidence_assigner = ConfidenceAssigner(knowledge_base)

    def verify_knowledge(self, knowledge):
        """
        Verify knowledge.

        Args:
            knowledge: The knowledge to verify.

        Returns:
            The verified knowledge with an assigned confidence score.
        """
        # Evaluate reliability
        reliability = self.reliability_evaluator.evaluate_reliability(knowledge)

        # Analyze scope
        scope = self.scope_analyzer.analyze_scope(knowledge)

        # Assign confidence
        verified_knowledge = self.confidence_assigner.assign_confidence(knowledge, reliability, scope)

        return verified_knowledge
```

知識検証モジュールは、信頼性評価器、適用範囲分析器、信頼度割当器から構成される。信頼性評価器は、知識の信頼性を評価するための指標や基準を提供する。適用範囲分析器は、知識が適用可能な条件や範囲を特定する。信頼度割当器は、評価された信頼性と適用範囲に基づいて、知識に適切な信頼度を割り当てる。

6.6.5 拡張可能知識ベースの統合

上記のサブコンポーネントを統合して、完全な拡張可能知識ベースを実装する。

```python
class ExtensibleKnowledgeBase:
    def __init__(self):
        # Knowledge representations
        self.propositional_logic = PropositionalLogicRepresentation()
        self.first_order_logic = FirstOrderLogicRepresentation()
        self.probabilistic_logic = ProbabilisticLogicRepresentation()
        self.knowledge_graph = KnowledgeGraphRepresentation()
        self.ontology = OntologyRepresentation()
```

```
        # Reasoning engine
        self.reasoning_engine = HybridReasoningEngine(self)

        # Knowledge acquisition module
        self.acquisition_module = KnowledgeAcquisitionModule(self)

        # Knowledge verification module
        self.verification_module = KnowledgeVerificationModule(self)

    def add_knowledge(self, knowledge, representation_type):
        """
        Add knowledge to the knowledge base.

        Args:
            knowledge: The knowledge to add.
            representation_type: The type of knowledge representation to use.

        Returns:
            None
        """
        # Verify knowledge
        verified_knowledge = self.verification_module.verify_knowledge(knowledge)

        # Add to appropriate representation
        if representation_type == 'propositional_logic':
            if isinstance(verified_knowledge, list):
                for rule in verified_knowledge:
                    self.propositional_logic.add_rule(rule)
            else:
                self.propositional_logic.add_rule(verified_knowledge)
        elif representation_type == 'first_order_logic':
            if isinstance(verified_knowledge, tuple) and len(verified_knowledge) == 2:
                if verified_knowledge[0] == 'fact':
                    self.first_order_logic.add_fact(verified_knowledge[1])
                elif verified_knowledge[0] == 'rule':
                    self.first_order_logic.add_rule(verified_knowledge[1])
            elif isinstance(verified_knowledge, list):
                for item in verified_knowledge:
                    if isinstance(item, tuple) and len(item) == 2:
                        if item[0] == 'fact':
                            self.first_order_logic.add_fact(item[1])
                        elif item[0] == 'rule':
                            self.first_order_logic.add_rule(item[1])
        elif representation_type == 'probabilistic_logic':
            if isinstance(verified_knowledge, tuple) and len(verified_knowledge) == 3:
                if verified_knowledge[0] == 'fact':
                    self.probabilistic_logic.add_fact(verified_knowledge[1], verified_knowledge[2])
                elif verified_knowledge[0] == 'rule':
                    self.probabilistic_logic.add_rule(verified_knowledge[1], verified_knowledge[2])
            elif isinstance(verified_knowledge, list):
                for item in verified_knowledge:
                    if isinstance(item, tuple) and len(item) == 3:
                        if item[0] == 'fact':
                            self.probabilistic_logic.add_fact(item[1], item[2])
                        elif item[0] == 'rule':
                            self.probabilistic_logic.add_rule(item[1], item[2])
        elif representation_type == 'knowledge_graph':
            if isinstance(verified_knowledge, tuple) and len(verified_knowledge) == 3:
                self.knowledge_graph.add_triple(*verified_knowledge)
            elif isinstance(verified_knowledge, list):
                for triple in verified_knowledge:
                    if isinstance(triple, tuple) and len(triple) == 3:
                        self.knowledge_graph.add_triple(*triple)
        elif representation_type == 'ontology':
            if isinstance(verified_knowledge, tuple) and len(verified_knowledge) >= 2:
```

```
            if verified_knowledge[0] == 'concept':
                if len(verified_knowledge) == 2:
                    self.ontology.add_concept(verified_knowledge[1])
                elif len(verified_knowledge) == 3:
                    self.ontology.add_concept(verified_knowledge[1], verified_knowledge[2])
            elif verified_knowledge[0] == 'attribute':
                if len(verified_knowledge) == 4:
                    self.ontology.add_attribute(verified_knowledge[1], verified_knowledge[2], verified_knowledge[3])
            elif verified_knowledge[0] == 'relation':
                if len(verified_knowledge) == 4:
                    self.ontology.add_relation(verified_knowledge[1], verified_knowledge[2], verified_knowledge[3])
        elif isinstance(verified_knowledge, list):
            for item in verified_knowledge:
                if isinstance(item, tuple) and len(item) >= 2:
                    if item[0] == 'concept':
                        if len(item) == 2:
                            self.ontology.add_concept(item[1])
                        elif len(item) == 3:
                            self.ontology.add_concept(item[1], item[2])
                    elif item[0] == 'attribute':
                        if len(item) == 4:
                            self.ontology.add_attribute(item[1], item[2], item[3])
                    elif item[0] == 'relation':
                        if len(item) == 4:
                            self.ontology.add_relation(item[1], item[2], item[3])

    def reason(self, query, reasoning_type=None, **kwargs):
        """
        Perform reasoning.

        Args:
            query: A query to be answered.
            reasoning_type: The type of reasoning to perform, or None to automatically select.
            **kwargs: Additional arguments for the specific reasoning engine.

        Returns:
            The result of the reasoning.
        """
        return self.reasoning_engine.reason(query, reasoning_type, **kwargs)

    def acquire_knowledge_from_data(self, data):
        """
        Acquire knowledge from data.

        Args:
            data: The data to acquire knowledge from.

        Returns:
            None
        """
        self.acquisition_module.acquire_knowledge(data)

    def get_knowledge(self, representation_type=None):
        """
        Get knowledge from the knowledge base.

        Args:
            representation_type: The type of knowledge representation to get, or None to get all.

        Returns:
            The requested knowledge.
        """
        if representation_type == 'propositional_logic':
            return self.propositional_logic.get_rules()
        elif representation_type == 'first_order_logic':
            return {
```

```
            'facts': self.first_order_logic.get_facts(),
            'rules': self.first_order_logic.get_rules()
        }
    elif representation_type == 'probabilistic_logic':
        return {
            'facts': self.probabilistic_logic.get_facts(),
            'rules': self.probabilistic_logic.get_rules()
        }
    elif representation_type == 'knowledge_graph':
        return self.knowledge_graph.get_triples()
    elif representation_type == 'ontology':
        return {
            'concepts': self.ontology.get_concepts(),
            'relations': self.ontology.get_relations()
        }
    else:
        return {
            'propositional_logic': self.propositional_logic.get_rules(),
            'first_order_logic': {
                'facts': self.first_order_logic.get_facts(),
                'rules': self.first_order_logic.get_rules()
            },
            'probabilistic_logic': {
                'facts': self.probabilistic_logic.get_facts(),
                'rules': self.probabilistic_logic.get_rules()
            },
            'knowledge_graph': self.knowledge_graph.get_triples(),
            'ontology': {
                'concepts': self.ontology.get_concepts(),
                'relations': self.ontology.get_relations()
            }
        }
```

拡張可能知識ベースは、多様な知識表現、ハイブリッド推論エンジン、知識獲得モジュール、知識検証モジュールから構成される。知識の追加、推論の実行、データからの知識獲得、知識の取得などの機能を提供する。

6.7. フィードバックループ統合器（FLI）の実装

フィードバックループ統合器は、反射プロセスの結果を出力生成プロセスにフィードバックし、出力の質を反復的に向上させるコンポーネントである。

6.7.1 反射結果分析の実装

反射結果分析は、階層的反射モジュールからの反射ベクトルを分析し、出力のどの部分が修正を必要とするかを特定する機能である。具体的な実装は以下の通りである。

```python
class ReflectionVectorAnalyzer:
    def __init__(self, threshold=0.5):
        self.threshold = threshold

    def analyze(self, reflection_vector):
        """
        Analyze a reflection vector.

        Args:
            reflection_vector: A tensor of shape [batch_size, output_dim].

        Returns:
            A binary mask of the same shape, indicating which elements need modification.
        """
```

```
        # Apply threshold
        mask = (reflection_vector > self.threshold).float()

        return mask

class ErrorPatternClassifier:
    def __init__(self, input_dim, hidden_dim, num_patterns=5):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.num_patterns = num_patterns

        # Pattern classifier
        self.classifier = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, num_patterns),
            nn.Softmax(dim=1)
        )

        # Pattern names (for interpretability)
        self.pattern_names = [
            "logical_inconsistency",
            "factual_error",
            "high_uncertainty",
            "bias",
            "incomplete_information"
        ]

    def classify(self, reflection_info):
        """
        Classify error patterns.

        Args:
            reflection_info: A dictionary containing reflection information.

        Returns:
            A tensor of shape [batch_size, num_patterns] representing pattern probabilities.
        """
        # Extract relevant information
        features = []

        if 'knowledge_inconsistency' in reflection_info:
            features.append(reflection_info['knowledge_inconsistency'])

        if 'multimodal_inconsistency' in reflection_info:
            features.append(reflection_info['multimodal_inconsistency'])

        if 'metacognitive_error_prob' in reflection_info:
            features.append(reflection_info['metacognitive_error_prob'])

        if 'detailed_bias_probs' in reflection_info:
            features.append(reflection_info['detailed_bias_probs'])

        if 'epistemic_variance' in reflection_info:
            # Take mean across output dimension
            features.append(reflection_info['epistemic_variance'].mean(dim=1, keepdim=True))

        if 'aleatoric_variance' in reflection_info:
            # Take mean across output dimension
            features.append(reflection_info['aleatoric_variance'].mean(dim=1, keepdim=True))

        # Concatenate features
        if features:
            combined = torch.cat(features, dim=1)

            # Classify patterns
```

```
            pattern_probs = self.classifier(combined)

            return pattern_probs
        else:
            # Return uniform distribution if no features are available
            batch_size = next(iter(reflection_info.values())).size(0)
            return torch.ones(batch_size, self.num_patterns) / self.num_patterns

class ModificationImpactPredictor:
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Impact predictor
        self.predictor = nn.Sequential(
            nn.Linear(input_dim + output_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
```

6.7.2 出力修正生成の実装

出力修正生成は、拡張可能知識ベースを用いて、修正が必要な部分に対する新しい出力候補を生成する機能である。具体的な実装は以下の通りである。

```python
class KnowledgeBaseQueryEngine:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base

    def query(self, error_pattern, input_data, output_data, modification_mask):
        """
        Query the knowledge base for relevant knowledge.

        Args:
            error_pattern: The type of error pattern.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            Relevant knowledge from the knowledge base.
        """
        # Extract key information from input and output
        key_info = self._extract_key_info(input_data, output_data, modification_mask)

        # Query knowledge base based on error pattern and key information
        if error_pattern == "logical_inconsistency":
            # Query logical rules
            return self.knowledge_base.reason(key_info, reasoning_type="deductive")
        elif error_pattern == "factual_error":
            # Query facts
            return self.knowledge_base.reason(key_info, reasoning_type="deductive")
        elif error_pattern == "high_uncertainty":
            # Query probabilistic knowledge
            return self.knowledge_base.reason(key_info, reasoning_type="probabilistic")
        elif error_pattern == "bias":
            # Query bias correction rules
            return self.knowledge_base.reason(key_info, reasoning_type="deductive")
        elif error_pattern == "incomplete_information":
            # Query for additional information
            return self.knowledge_base.reason(key_info, reasoning_type="abductive")
        else:
```

```
            # Default to deductive reasoning
            return self.knowledge_base.reason(key_info, reasoning_type="deductive")

    def _extract_key_info(self, input_data, output_data, modification_mask):
        """
        Extract key information from input and output data.

        Args:
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            Key information for knowledge base query.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated information extraction

        # Extract parts of output that need modification
        masked_output = output_data * modification_mask

        # Combine with input data
        key_info = {
            "input": input_data,
            "output": output_data,
            "masked_output": masked_output,
            "modification_mask": modification_mask
        }

        return key_info

class CandidateGenerator:
    def __init__(self, knowledge_base):
        self.knowledge_base = knowledge_base
        self.query_engine = KnowledgeBaseQueryEngine(knowledge_base)

    def generate_candidates(self, error_pattern, input_data, output_data, modification_mask):
        """
        Generate modification candidates.

        Args:
            error_pattern: The type of error pattern.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            A list of modification candidates.
        """
        # Query knowledge base for relevant knowledge
        relevant_knowledge = self.query_engine.query(error_pattern, input_data, output_data, modification_mask)

        # Generate candidates based on relevant knowledge
        candidates = self._generate_from_knowledge(relevant_knowledge, input_data, output_data, modification_mask)

        return candidates

    def _generate_from_knowledge(self, relevant_knowledge, input_data, output_data, modification_mask):
        """
        Generate candidates from relevant knowledge.

        Args:
            relevant_knowledge: Relevant knowledge from the knowledge base.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.
```

```
        Returns:
            A list of modification candidates.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated candidate generation

        # Initialize candidates list
        candidates = []

        # Generate candidates based on reasoning type
        if isinstance(relevant_knowledge, dict) and "reasoning_type" in relevant_knowledge:
            if relevant_knowledge["reasoning_type"] == "deductive":
                # Generate candidates based on deductive reasoning
                deductive_candidates = self._generate_deductive_candidates(relevant_knowledge, input_data, output_data,
modification_mask)
                candidates.extend(deductive_candidates)
            elif relevant_knowledge["reasoning_type"] == "abductive":
                # Generate candidates based on abductive reasoning
                abductive_candidates = self._generate_abductive_candidates(relevant_knowledge, input_data, output_data,
modification_mask)
                candidates.extend(abductive_candidates)
            elif relevant_knowledge["reasoning_type"] == "probabilistic":
                # Generate candidates based on probabilistic reasoning
                probabilistic_candidates = self._generate_probabilistic_candidates(relevant_knowledge, input_data,
output_data, modification_mask)
                candidates.extend(probabilistic_candidates)

        # If no candidates were generated, create a default candidate
        if not candidates:
            default_candidate = output_data.clone()
            # Set modified parts to zero (placeholder)
            default_candidate[modification_mask > 0.5] = 0.0
            candidates.append(default_candidate)

        return candidates

    def _generate_deductive_candidates(self, relevant_knowledge, input_data, output_data, modification_mask):
        """
        Generate candidates based on deductive reasoning.

        Args:
            relevant_knowledge: Relevant knowledge from the knowledge base.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            A list of candidates.
        """
        # This is a simplified implementation
        candidates = []

        # Create a modified output
        modified_output = output_data.clone()

        # Apply modifications based on deductive reasoning
        if "conclusions" in relevant_knowledge:
            for i, conclusion in enumerate(relevant_knowledge["conclusions"]):
                # Apply conclusion to modified output
                # This is a placeholder implementation
                modified_output[i][modification_mask[i] > 0.5] = conclusion

        candidates.append(modified_output)

        return candidates
```

```
    def _generate_abductive_candidates(self, relevant_knowledge, input_data, output_data, modification_mask):
        """
        Generate candidates based on abductive reasoning.

        Args:
            relevant_knowledge: Relevant knowledge from the knowledge base.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            A list of candidates.
        """
        # This is a simplified implementation
        candidates = []

        # Create multiple modified outputs
        if "explanations" in relevant_knowledge:
            for explanation in relevant_knowledge["explanations"]:
                modified_output = output_data.clone()

                # Apply explanation to modified output
                # This is a placeholder implementation
                for i in range(len(modified_output)):
                    modified_output[i][modification_mask[i] > 0.5] = explanation

                candidates.append(modified_output)

        return candidates

    def _generate_probabilistic_candidates(self, relevant_knowledge, input_data, output_data, modification_mask):
        """
        Generate candidates based on probabilistic reasoning.

        Args:
            relevant_knowledge: Relevant knowledge from the knowledge base.
            input_data: The input data.
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.

        Returns:
            A list of candidates.
        """
        # This is a simplified implementation
        candidates = []

        # Create multiple modified outputs with different probabilities
        if "distributions" in relevant_knowledge:
            for distribution in relevant_knowledge["distributions"]:
                modified_output = output_data.clone()

                # Apply distribution to modified output
                # This is a placeholder implementation
                for i in range(len(modified_output)):
                    # Sample from distribution
                    sample = torch.tensor(np.random.choice(distribution["values"], p=distribution["probabilities"]))
                    modified_output[i][modification_mask[i] > 0.5] = sample

                candidates.append(modified_output)

        return candidates

class CandidateRanker:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
```

```
        self.hidden_dim = hidden_dim

        # Ranking model
        self.ranking_model = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1)
        )

    def rank_candidates(self, candidates, input_data, original_output, reflection_info):
        """
        Rank modification candidates.

        Args:
            candidates: A list of modification candidates.
            input_data: The input data.
            original_output: The original output.
            reflection_info: Information from the reflection process.

        Returns:
            A list of ranked candidates.
        """
        # Compute features for each candidate
        candidate_features = []
        for candidate in candidates:
            # Compute consistency with knowledge base
            consistency = self._compute_consistency(candidate, input_data)

            # Compute similarity to original output
            similarity = self._compute_similarity(candidate, original_output)

            # Compute uncertainty reduction
            uncertainty_reduction = self._compute_uncertainty_reduction(candidate, reflection_info)

            # Combine features
            features = torch.cat([consistency, similarity, uncertainty_reduction], dim=1)
            candidate_features.append(features)

        # Stack features
        if candidate_features:
            stacked_features = torch.cat(candidate_features, dim=0)

            # Compute scores
            scores = self.ranking_model(stacked_features)

            # Sort candidates by score
            sorted_indices = torch.argsort(scores, dim=0, descending=True).squeeze()
            ranked_candidates = [candidates[i] for i in sorted_indices]

            return ranked_candidates
        else:
            return candidates

    def _compute_consistency(self, candidate, input_data):
        """
        Compute consistency of candidate with knowledge base.

        Args:
            candidate: A modification candidate.
            input_data: The input data.

        Returns:
            A tensor representing consistency.
        """
```

```
        # This is a simplified implementation
        # In practice, this would involve more sophisticated consistency evaluation

        # Placeholder: return random consistency score
        batch_size = candidate.size(0)
        return torch.rand(batch_size, self.hidden_dim // 3)

    def _compute_similarity(self, candidate, original_output):
        """
        Compute similarity between candidate and original output.

        Args:
            candidate: A modification candidate.
            original_output: The original output.

        Returns:
            A tensor representing similarity.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated similarity computation

        # Compute cosine similarity
        normalized_candidate = F.normalize(candidate, p=2, dim=1)
        normalized_original = F.normalize(original_output, p=2, dim=1)
        similarity = torch.sum(normalized_candidate * normalized_original, dim=1, keepdim=True)

        # Expand to required dimension
        batch_size = candidate.size(0)
        expanded_similarity = similarity.expand(batch_size, self.hidden_dim // 3)

        return expanded_similarity

    def _compute_uncertainty_reduction(self, candidate, reflection_info):
        """
        Compute uncertainty reduction achieved by candidate.

        Args:
            candidate: A modification candidate.
            reflection_info: Information from the reflection process.

        Returns:
            A tensor representing uncertainty reduction.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated uncertainty evaluation

        # Placeholder: return random uncertainty reduction
        batch_size = candidate.size(0)
        return torch.rand(batch_size, self.hidden_dim // 3)

class OutputModificationGenerator:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Components
        self.candidate_generator = CandidateGenerator(knowledge_base)
        self.candidate_ranker = CandidateRanker(knowledge_base, hidden_dim)

    def generate_modifications(self, error_patterns, input_data, output_data, modification_mask, reflection_info):
        """
        Generate output modifications.

        Args:
            error_patterns: The types of error patterns.
            input_data: The input data.
```

```
            output_data: The output data.
            modification_mask: A binary mask indicating which parts of the output need modification.
            reflection_info: Information from the reflection process.

        Returns:
            A list of ranked modification candidates.
        """
        all_candidates = []

        # Generate candidates for each error pattern
        for error_pattern in error_patterns:
            candidates = self.candidate_generator.generate_candidates(
                error_pattern, input_data, output_data, modification_mask
            )
            all_candidates.extend(candidates)

        # Rank candidates
        ranked_candidates = self.candidate_ranker.rank_candidates(
            all_candidates, input_data, output_data, reflection_info
        )

        return ranked_candidates
```

出力修正生成は、知識ベース照会エンジン、候補生成器、候補ランキング器から構成される。知識ベース照会エンジンは、修正が必要な部分に関連する知識を知識ベースから検索する。候補生成器は、検索された知識に基づいて、修正候補を生成する。候補ランキング器は、生成された修正候補を評価し、最も適切な候補を選択する。

6.7.3 整合性評価の実装

整合性評価は、修正された出力の全体的な整合性を評価し、必要に応じてさらなる修正を行う機能である。具体的な実装は以下の通りである。

```python
class GlobalConsistencyChecker:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Consistency evaluation model
        self.consistency_model = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def check_consistency(self, output, input_data):
        """
        Check global consistency of output with knowledge base.

        Args:
            output: The output to check.
            input_data: The input data.

        Returns:
            A consistency score and a list of inconsistencies.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated consistency checking
```

```
        # Compute features for consistency evaluation
        features = self._compute_features(output, input_data)

        # Evaluate consistency
        consistency_score = self.consistency_model(features)

        # Detect inconsistencies
        inconsistencies = self._detect_inconsistencies(output, input_data)

        return consistency_score, inconsistencies

    def _compute_features(self, output, input_data):
        """
        Compute features for consistency evaluation.

        Args:
            output: The output to check.
            input_data: The input data.

        Returns:
            Features for consistency evaluation.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated feature computation

        # Concatenate input and output
        combined = torch.cat([input_data, output], dim=1)

        # Project to feature space
        features = combined  # Placeholder

        return features

    def _detect_inconsistencies(self, output, input_data):
        """
        Detect inconsistencies in output.

        Args:
            output: The output to check.
            input_data: The input data.

        Returns:
            A list of detected inconsistencies.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated inconsistency detection

        # Placeholder: return empty list
        return []

class LocalConsistencyChecker:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Consistency evaluation model
        self.consistency_model = nn.Sequential(
            nn.Linear(hidden_dim * 3, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def check_consistency(self, output, input_data, modification_mask):
```

```
        """
        Check local consistency of modified parts with surrounding context.

        Args:
            output: The output to check.
            input_data: The input data.
            modification_mask: A binary mask indicating which parts were modified.

        Returns:
            A consistency score and a list of inconsistencies.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated consistency checking

        # Compute features for consistency evaluation
        features = self._compute_features(output, input_data, modification_mask)

        # Evaluate consistency
        consistency_score = self.consistency_model(features)

        # Detect inconsistencies
        inconsistencies = self._detect_inconsistencies(output, input_data, modification_mask)

        return consistency_score, inconsistencies

    def _compute_features(self, output, input_data, modification_mask):
        """
        Compute features for consistency evaluation.

        Args:
            output: The output to check.
            input_data: The input data.
            modification_mask: A binary mask indicating which parts were modified.

        Returns:
            Features for consistency evaluation.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated feature computation

        # Concatenate input, output, and modification mask
        combined = torch.cat([input_data, output, modification_mask], dim=1)

        # Project to feature space
        features = combined  # Placeholder

        return features

    def _detect_inconsistencies(self, output, input_data, modification_mask):
        """
        Detect inconsistencies in output.

        Args:
            output: The output to check.
            input_data: The input data.
            modification_mask: A binary mask indicating which parts were modified.

        Returns:
            A list of detected inconsistencies.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated inconsistency detection

        # Placeholder: return empty list
        return []
```

```
class ConsistencyScorer:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Global and local consistency checkers
        self.global_checker = GlobalConsistencyChecker(knowledge_base, hidden_dim)
        self.local_checker = LocalConsistencyChecker(knowledge_base, hidden_dim)

        # Weights for global and local consistency
        self.global_weight = nn.Parameter(torch.tensor(0.5))
        self.local_weight = nn.Parameter(torch.tensor(0.5))

    def score_consistency(self, output, input_data, modification_mask=None):
        """
        Score the consistency of output.

        Args:
            output: The output to score.
            input_data: The input data.
            modification_mask: A binary mask indicating which parts were modified, or None.

        Returns:
            A consistency score and a list of inconsistencies.
        """
        # Check global consistency
        global_score, global_inconsistencies = self.global_checker.check_consistency(output, input_data)

        # Check local consistency if modification mask is provided
        if modification_mask is not None:
            local_score, local_inconsistencies = self.local_checker.check_consistency(output, input_data,
modification_mask)

            # Combine global and local scores
            combined_score = self.global_weight * global_score + self.local_weight * local_score

            # Combine inconsistencies
            combined_inconsistencies = global_inconsistencies + local_inconsistencies
        else:
            combined_score = global_score
            combined_inconsistencies = global_inconsistencies

        return combined_score, combined_inconsistencies

class ConsistencyEvaluator:
    def __init__(self, knowledge_base, hidden_dim, consistency_threshold=0.8):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim
        self.consistency_threshold = consistency_threshold

        # Consistency scorer
        self.consistency_scorer = ConsistencyScorer(knowledge_base, hidden_dim)

    def evaluate_consistency(self, output, input_data, modification_mask=None):
        """
        Evaluate the consistency of output and determine if further modification is needed.

        Args:
            output: The output to evaluate.
            input_data: The input data.
            modification_mask: A binary mask indicating which parts were modified, or None.

        Returns:
            A tuple (is_consistent, consistency_score, inconsistencies, new_modification_mask).
        """
        # Score consistency
```

```
        consistency_score, inconsistencies = self.consistency_scorer.score_consistency(output, input_data,
modification_mask)

        # Determine if output is consistent
        is_consistent = (consistency_score >= self.consistency_threshold).all()

        # Create new modification mask if further modification is needed
        if not is_consistent and inconsistencies:
            new_modification_mask = self._create_modification_mask(output, inconsistencies)
        else:
            new_modification_mask = None

        return is_consistent, consistency_score, inconsistencies, new_modification_mask

    def _create_modification_mask(self, output, inconsistencies):
        """
        Create a modification mask based on detected inconsistencies.

        Args:
            output: The output.
            inconsistencies: A list of detected inconsistencies.

        Returns:
            A binary mask indicating which parts need further modification.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated mask creation

        # Initialize mask with zeros
        mask = torch.zeros_like(output)

        # Set mask to 1 for each inconsistency
        for inconsistency in inconsistencies:
            if "indices" in inconsistency:
                for idx in inconsistency["indices"]:
                    mask[idx] = 1.0

        return mask
```

整合性評価は、グローバル整合性チェッカー、ローカル整合性チェッカー、整合性スコアリングから構成される。グローバル整合性チェッカーは、修正後の出力全体が領域知識と整合しているかを確認する。ローカル整合性チェッカーは、修正された部分と周囲の部分の間の整合性を確認する。整合性スコアリングは、出力の整合性を定量的に評価し、スコアを割り当てる。

6.7.4 反復終了判定の実装

反復終了判定は、出力の質が十分に向上したか、または反復回数が上限に達したかを判断し、プロセスを終了するタイミングを決定する機能である。具体的な実装は以下の通りである。

```python
class ImprovementEvaluator:
    def __init__(self, improvement_threshold=0.01):
        self.improvement_threshold = improvement_threshold

    def evaluate_improvement(self, current_score, previous_score):
        """
        Evaluate improvement in consistency score.

        Args:
            current_score: The current consistency score.
            previous_score: The previous consistency score.

        Returns:
```

```
            A boolean indicating whether significant improvement was achieved.
        """
        # Compute improvement
        improvement = current_score - previous_score

        # Check if improvement is significant
        is_significant = (improvement > self.improvement_threshold).all()

        return is_significant

class ConvergenceDetector:
    def __init__(self, convergence_threshold=0.001):
        self.convergence_threshold = convergence_threshold

    def detect_convergence(self, current_output, previous_output):
        """
        Detect convergence in output.

        Args:
            current_output: The current output.
            previous_output: The previous output.

        Returns:
            A boolean indicating whether convergence has been achieved.
        """
        # Compute change in output
        change = torch.norm(current_output - previous_output, dim=1)

        # Check if change is below threshold
        has_converged = (change < self.convergence_threshold).all()

        return has_converged

class ResourceManager:
    def __init__(self, max_iterations=10, time_limit=None):
        self.max_iterations = max_iterations
        self.time_limit = time_limit
        self.start_time = None

    def start(self):
        """
        Start resource management.
        """
        self.start_time = time.time()

    def should_continue(self, current_iteration):
        """
        Determine if iteration should continue based on resource constraints.

        Args:
            current_iteration: The current iteration number.

        Returns:
            A boolean indicating whether iteration should continue.
        """
        # Check iteration limit
        if current_iteration >= self.max_iterations:
            return False

        # Check time limit
        if self.time_limit is not None and self.start_time is not None:
            elapsed_time = time.time() - self.start_time
            if elapsed_time > self.time_limit:
                return False

        return True
```

```
class IterationTerminator:
    def __init__(self, max_iterations=10, time_limit=None, improvement_threshold=0.01,
convergence_threshold=0.001):
        self.improvement_evaluator = ImprovementEvaluator(improvement_threshold)
        self.convergence_detector = ConvergenceDetector(convergence_threshold)
        self.resource_manager = ResourceManager(max_iterations, time_limit)

    def start(self):
        """
        Start iteration termination management.
        """
        self.resource_manager.start()

    def should_terminate(self, current_iteration, current_output, previous_output, current_score, previous_score):
        """
        Determine if iteration should terminate.

        Args:
            current_iteration: The current iteration number.
            current_output: The current output.
            previous_output: The previous output.
            current_score: The current consistency score.
            previous_score: The previous consistency score.

        Returns:
            A boolean indicating whether iteration should terminate.
        """
        # Check resource constraints
        if not self.resource_manager.should_continue(current_iteration):
            return True

        # Check convergence
        if previous_output is not None and self.convergence_detector.detect_convergence(current_output,
previous_output):
            return True

        # Check improvement
        if previous_score is not None and not self.improvement_evaluator.evaluate_improvement(current_score,
previous_score):
            return True

        return False
```

反復終了判定は、改善度評価器、収束検出器、リソース管理器から構成される。改善度評価器は、各反復での出力の質の改善度を評価する。収束検出器は、出力が安定状態に収束したかどうかを検出する。リソース管理器は、利用可能な計算リソースや時間制約に基づいて、反復を継続するか終了するかを決定する。

6.7.5 フィードバックループ統合器の統合

上記のサブコンポーネントを統合して、完全なフィードバックループ統合器を実装する。

```python
class FeedbackLoopIntegrator:
    def __init__(self, knowledge_base, hidden_dim, output_dim, max_iterations=10, time_limit=None,
consistency_threshold=0.8, improvement_threshold=0.01, convergence_threshold=0.001):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Components
        self.reflection_analyzer = ReflectionVectorAnalyzer()
        self.error_classifier = ErrorPatternClassifier(hidden_dim, hidden_dim)
```

```
        self.modification_generator = OutputModificationGenerator(knowledge_base, hidden_dim)
        self.consistency_evaluator = ConsistencyEvaluator(knowledge_base, hidden_dim, consistency_threshold)
        self.iteration_terminator = IterationTerminator(max_iterations, time_limit, improvement_threshold,
convergence_threshold)

    def integrate(self, input_data, initial_output, reflection_vector, reflection_info):
        """
        Integrate feedback from reflection to improve output.

        Args:
            input_data: The input data.
            initial_output: The initial output.
            reflection_vector: The reflection vector.
            reflection_info: Additional information from the reflection process.

        Returns:
            The improved output.
        """
        # Start iteration termination management
        self.iteration_terminator.start()

        # Initialize variables
        current_output = initial_output
        previous_output = None
        current_score = None
        previous_score = None
        current_iteration = 0

        # Analyze reflection vector
        modification_mask = self.reflection_analyzer.analyze(reflection_vector)

        # Classify error patterns
        error_pattern_probs = self.error_classifier.classify(reflection_info)
        error_patterns = self._select_error_patterns(error_pattern_probs)

        # Iterative improvement
        while True:
            # Evaluate consistency of current output
            is_consistent, consistency_score, inconsistencies, new_modification_mask =
self.consistency_evaluator.evaluate_consistency(
                current_output, input_data, modification_mask
            )

            # Update scores
            previous_score = current_score
            current_score = consistency_score

            # Check termination conditions
            if is_consistent or self.iteration_terminator.should_terminate(
                current_iteration, current_output, previous_output, current_score, previous_score
            ):
                break

            # Update modification mask if new inconsistencies were detected
            if new_modification_mask is not None:
                modification_mask = new_modification_mask

            # Generate modification candidates
            candidates = self.modification_generator.generate_modifications(
                error_patterns, input_data, current_output, modification_mask, reflection_info
            )

            # Select best candidate
            if candidates:
                best_candidate = candidates[0]  # First candidate is the highest ranked
                previous_output = current_output
```

```
                current_output = best_candidate
            else:
                break

            # Increment iteration counter
            current_iteration += 1

        return current_output

    def _select_error_patterns(self, error_pattern_probs, top_k=2):
        """
        Select top-k error patterns based on probabilities.

        Args:
            error_pattern_probs: Probabilities for each error pattern.
            top_k: Number of top patterns to select.

        Returns:
            A list of selected error pattern names.
        """
        # Get indices of top-k patterns
        _, indices = torch.topk(error_pattern_probs, min(top_k, error_pattern_probs.size(1)), dim=1)

        # Map indices to pattern names
        pattern_names = []
        for i in range(indices.size(0)):
            batch_patterns = []
            for j in range(indices.size(1)):
                pattern_idx = indices[i, j].item()
                pattern_name = self.error_classifier.pattern_names[pattern_idx]
                batch_patterns.append(pattern_name)
            pattern_names.append(batch_patterns)

        return pattern_names
```

フィードバックループ統合器は、反射結果分析、エラーパターン分類器、出力修正生成器、整合性評価器、反復終了判定器から構成される。これらのコンポーネントを組み合わせて、反射ベクトルに基づいて出力を反復的に改善するプロセスを実装する。

6.8. メタ学習コントローラ（MLC）の実装

メタ学習コントローラは、システム全体の学習プロセスを監視し、タスクの性質や難易度に応じてパラメータ更新戦略を調整するコンポーネントである。

6.8.1 タスク特性分析の実装

タスク特性分析は、入力タスクの特性（複雑さ、不確実性、既存知識との関連性など）を分析し、適切な学習戦略を決定する機能である。具体的な実装は以下の通りである。

```python
class TaskComplexityEvaluator:
    def __init__(self, hidden_dim):
        self.hidden_dim = hidden_dim

        # Complexity evaluation model
        self.complexity_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
```

```
        )

    def evaluate_complexity(self, task_embedding):
        """
        Evaluate the complexity of a task.

        Args:
            task_embedding: An embedding representing the task.

        Returns:
            A complexity score between 0 and 1.
        """
        complexity = self.complexity_model(task_embedding)
        return complexity

class UncertaintyAnalyzer:
    def __init__(self, hidden_dim):
        self.hidden_dim = hidden_dim

        # Uncertainty analysis model
        self.uncertainty_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 3)  # 3 types of uncertainty
        )

    def analyze_uncertainty(self, task_embedding):
        """
        Analyze the uncertainty in a task.

        Args:
            task_embedding: An embedding representing the task.

        Returns:
            A tensor with scores for different types of uncertainty.
        """
        uncertainty_scores = self.uncertainty_model(task_embedding)

        # Split into different types of uncertainty
        data_uncertainty = uncertainty_scores[:, 0:1]
        model_uncertainty = uncertainty_scores[:, 1:2]
        environment_uncertainty = uncertainty_scores[:, 2:3]

        return data_uncertainty, model_uncertainty, environment_uncertainty

class KnowledgeRelevanceMapper:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Relevance mapping model
        self.relevance_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def map_relevance(self, task_embedding):
        """
        Map the relevance of existing knowledge to the task.
```

```
        Args:
            task_embedding: An embedding representing the task.

        Returns:
            A relevance score between 0 and 1.
        """
        relevance = self.relevance_model(task_embedding)
        return relevance

    def identify_knowledge_gaps(self, task_embedding):
        """
        Identify gaps in existing knowledge for the task.

        Args:
            task_embedding: An embedding representing the task.

        Returns:
            A list of identified knowledge gaps.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated gap analysis

        # Placeholder: return empty list
        return []

class TaskCharacteristicsAnalyzer:
    def __init__(self, knowledge_base, hidden_dim):
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim

        # Components
        self.complexity_evaluator = TaskComplexityEvaluator(hidden_dim)
        self.uncertainty_analyzer = UncertaintyAnalyzer(hidden_dim)
        self.relevance_mapper = KnowledgeRelevanceMapper(knowledge_base, hidden_dim)

        # Task embedding model
        self.task_embedding_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim)
        )

    def analyze_task(self, task_data):
        """
        Analyze the characteristics of a task.

        Args:
            task_data: Data representing the task.

        Returns:
            A dictionary of task characteristics.
        """
        # Generate task embedding
        task_embedding = self.task_embedding_model(task_data)

        # Evaluate complexity
        complexity = self.complexity_evaluator.evaluate_complexity(task_embedding)

        # Analyze uncertainty
        data_uncertainty, model_uncertainty, environment_uncertainty =
self.uncertainty_analyzer.analyze_uncertainty(task_embedding)

        # Map knowledge relevance
        knowledge_relevance = self.relevance_mapper.map_relevance(task_embedding)

        # Identify knowledge gaps
```

```
        knowledge_gaps = self.relevance_mapper.identify_knowledge_gaps(task_embedding)

        # Combine characteristics
        characteristics = {
            "task_embedding": task_embedding,
            "complexity": complexity,
            "data_uncertainty": data_uncertainty,
            "model_uncertainty": model_uncertainty,
            "environment_uncertainty": environment_uncertainty,
            "knowledge_relevance": knowledge_relevance,
            "knowledge_gaps": knowledge_gaps
        }

        return characteristics
```

タスク特性分析は、タスク複雑性評価器、不確実性分析器、知識関連性マッパーから構成される。タスク複雑性評価器は、タスクの複雑さを評価するための指標や基準を提供する。不確実性分析器は、タスクに含まれる不確実性の種類と程度を分析する。知識関連性マッパーは、タスクと既存知識の関連性を評価する。

6.8.2 パラメータ更新制御の実装

パラメータ更新制御は、学習率、正則化強度、勾配クリッピングなどのハイパーパラメータを動的に調整する機能である。具体的な実装は以下の通りである。

```python
class AdaptiveLearningRateScheduler:
    def __init__(self, base_lr=0.001, min_lr=1e-6, max_lr=0.1):
        self.base_lr = base_lr
        self.min_lr = min_lr
        self.max_lr = max_lr

    def schedule_learning_rate(self, task_characteristics, learning_progress):
        """
        Schedule learning rate based on task characteristics and learning progress.

        Args:
            task_characteristics: Characteristics of the task.
            learning_progress: Information about learning progress.

        Returns:
            The scheduled learning rate.
        """
        # Extract relevant characteristics
        complexity = task_characteristics["complexity"]

        # Extract relevant progress information
        current_epoch = learning_progress["current_epoch"]
        total_epochs = learning_progress["total_epochs"]

        # Compute base learning rate based on complexity
        # Higher complexity -> lower learning rate
        complexity_factor = 1.0 - complexity
        base_lr = self.base_lr * complexity_factor

        # Apply cosine annealing schedule
        progress = current_epoch / total_epochs
        cosine_factor = 0.5 * (1.0 + math.cos(math.pi * progress))

        # Compute final learning rate
        lr = self.min_lr + (base_lr - self.min_lr) * cosine_factor

        # Clip to range
        lr = max(self.min_lr, min(self.max_lr, lr))
```

```
        return lr

class RegularizationStrengthController:
    def __init__(self, base_strength=0.001, min_strength=1e-6, max_strength=0.1):
        self.base_strength = base_strength
        self.min_strength = min_strength
        self.max_strength = max_strength

    def control_regularization(self, task_characteristics, learning_progress):
        """
        Control regularization strength based on task characteristics and learning progress.

        Args:
            task_characteristics: Characteristics of the task.
            learning_progress: Information about learning progress.

        Returns:
            The controlled regularization strength.
        """
        # Extract relevant characteristics
        complexity = task_characteristics["complexity"]
        data_uncertainty = task_characteristics["data_uncertainty"]

        # Extract relevant progress information
        train_loss = learning_progress["train_loss"]
        val_loss = learning_progress["val_loss"]

        # Compute overfitting factor
        # Higher difference between train and val loss -> higher overfitting
        overfitting_factor = max(0.0, (train_loss - val_loss) / train_loss)

        # Compute complexity factor
        # Higher complexity -> higher regularization
        complexity_factor = complexity

        # Compute uncertainty factor
        # Higher data uncertainty -> lower regularization
        uncertainty_factor = 1.0 - data_uncertainty

        # Combine factors
        combined_factor = (overfitting_factor + complexity_factor + uncertainty_factor) / 3.0

        # Compute final regularization strength
        strength = self.base_strength * combined_factor

        # Clip to range
        strength = max(self.min_strength, min(self.max_strength, strength))

        return strength

class GradientStabilizer:
    def __init__(self, base_clip=1.0, min_clip=0.1, max_clip=10.0):
        self.base_clip = base_clip
        self.min_clip = min_clip
        self.max_clip = max_clip

        # Gradient statistics
        self.grad_mean = None
        self.grad_var = None
        self.grad_count = 0

    def stabilize_gradients(self, task_characteristics, gradients):
        """
        Stabilize gradients based on task characteristics and gradient statistics.
```

```
        Args:
            task_characteristics: Characteristics of the task.
            gradients: The gradients to stabilize.

        Returns:
            The stabilized gradients and the clip value.
        """
        # Extract relevant characteristics
        complexity = task_characteristics["complexity"]
        model_uncertainty = task_characteristics["model_uncertainty"]

        # Update gradient statistics
        if self.grad_mean is None:
            self.grad_mean = torch.mean(torch.stack([torch.norm(g) for g in gradients]))
            self.grad_var = torch.var(torch.stack([torch.norm(g) for g in gradients]))
        else:
            new_mean = torch.mean(torch.stack([torch.norm(g) for g in gradients]))
            new_var = torch.var(torch.stack([torch.norm(g) for g in gradients]))
            self.grad_mean = 0.9 * self.grad_mean + 0.1 * new_mean
            self.grad_var = 0.9 * self.grad_var + 0.1 * new_var

        self.grad_count += 1

        # Compute complexity factor
        # Higher complexity -> lower clip value
        complexity_factor = 1.0 - complexity

        # Compute uncertainty factor
        # Higher model uncertainty -> lower clip value
        uncertainty_factor = 1.0 - model_uncertainty

        # Compute stability factor
        # Higher gradient variance -> lower clip value
        stability_factor = 1.0 / (1.0 + self.grad_var / (self.grad_mean + 1e-8))

        # Combine factors
        combined_factor = (complexity_factor + uncertainty_factor + stability_factor) / 3.0

        # Compute clip value
        clip_value = self.base_clip * combined_factor

        # Clip to range
        clip_value = max(self.min_clip, min(self.max_clip, clip_value))

        # Apply gradient clipping
        stabilized_gradients = [torch.clamp(g, -clip_value, clip_value) for g in gradients]

        return stabilized_gradients, clip_value

class ParameterUpdateController:
    def __init__(self, base_lr=0.001, base_reg=0.001, base_clip=1.0):
        self.base_lr = base_lr
        self.base_reg = base_reg
        self.base_clip = base_clip

        # Components
        self.lr_scheduler = AdaptiveLearningRateScheduler(base_lr)
        self.reg_controller = RegularizationStrengthController(base_reg)
        self.grad_stabilizer = GradientStabilizer(base_clip)

    def control_update(self, task_characteristics, learning_progress, gradients):
        """
        Control parameter update based on task characteristics and learning progress.

        Args:
            task_characteristics: Characteristics of the task.
```

```
            learning_progress: Information about learning progress.
            gradients: The gradients for parameter update.

        Returns:
            A dictionary of update parameters.
        """
        # Schedule learning rate
        lr = self.lr_scheduler.schedule_learning_rate(task_characteristics, learning_progress)

        # Control regularization strength
        reg_strength = self.reg_controller.control_regularization(task_characteristics, learning_progress)

        # Stabilize gradients
        stabilized_gradients, clip_value = self.grad_stabilizer.stabilize_gradients(task_characteristics, gradients)

        # Combine update parameters
        update_params = {
            "learning_rate": lr,
            "regularization_strength": reg_strength,
            "clip_value": clip_value,
            "stabilized_gradients": stabilized_gradients
        }

        return update_params
```

パラメータ更新制御は、適応的学習率スケジューラ、正則化強度コントローラ、勾配安定化器から構成される。適応的学習率スケジューラは、タスクの複雑さや学習の進捗に応じて、学習率を動的に調整する。正則化強度コントローラは、過学習のリスクに応じて、正則化の強度を調整する。勾配安定化器は、勾配の爆発や消失を防ぐために、勾配クリッピングやスケーリングを適用する。

6.8.3 知識転移促進の実装

知識転移促進は、異なるタスク間での知識の共有と転移を促進するためのパラメータ共有戦略を実装する機能である。具体的な実装は以下の通りである。

```python
class SharedRepresentationLearner:
    def __init__(self, hidden_dim, num_tasks):
        self.hidden_dim = hidden_dim
        self.num_tasks = num_tasks

        # Shared representation model
        self.shared_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim)
        )

        # Task-specific heads
        self.task_heads = nn.ModuleList([
            nn.Linear(hidden_dim, hidden_dim)
            for _ in range(num_tasks)
        ])

    def forward(self, x, task_id):
        """
        Forward pass through the shared representation model.

        Args:
            x: The input data.
            task_id: The ID of the task.

        Returns:
```

```
            The output of the model.
        """
        # Shared representation
        shared_repr = self.shared_model(x)

        # Task-specific head
        output = self.task_heads[task_id](shared_repr)

        return output, shared_repr

class TransferabilityEvaluator:
    def __init__(self, hidden_dim):
        self.hidden_dim = hidden_dim

        # Transferability evaluation model
        self.transferability_model = nn.Sequential(
            nn.Linear(hidden_dim * 2, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def evaluate_transferability(self, source_embedding, target_embedding):
        """
        Evaluate the transferability from source to target task.

        Args:
            source_embedding: An embedding representing the source task.
            target_embedding: An embedding representing the target task.

        Returns:
            A transferability score between 0 and 1.
        """
        # Concatenate embeddings
        combined = torch.cat([source_embedding, target_embedding], dim=1)

        # Evaluate transferability
        transferability = self.transferability_model(combined)

        return transferability

class SelectiveFineTuningController:
    def __init__(self, model, hidden_dim):
        self.model = model
        self.hidden_dim = hidden_dim

        # Parameter importance model
        self.importance_model = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1),
            nn.Sigmoid()
        )

    def compute_parameter_importance(self, task_embedding):
        """
        Compute the importance of each parameter for a task.

        Args:
            task_embedding: An embedding representing the task.

        Returns:
```

```
            A dictionary mapping parameter names to importance scores.
        """
        # This is a simplified implementation
        # In practice, this would involve more sophisticated importance computation

        # Initialize importance dictionary
        importance = {}

        # Compute importance for each parameter
        for name, param in self.model.named_parameters():
            # Generate parameter embedding
            param_embedding = torch.mean(param.view(-1, 1).expand(-1, self.hidden_dim), dim=0, keepdim=True)

            # Concatenate with task embedding
            combined = torch.cat([task_embedding, param_embedding], dim=1)

            # Compute importance
            importance[name] = self.importance_model(combined)

        return importance

    def control_fine_tuning(self, source_task_embedding, target_task_embedding, transferability):
        """
        Control which parameters to freeze and which to update during fine-tuning.

        Args:
            source_task_embedding: An embedding representing the source task.
            target_task_embedding: An embedding representing the target task.
            transferability: The transferability score from source to target.

        Returns:
            A dictionary mapping parameter names to freeze flags.
        """
        # Compute parameter importance for source and target tasks
        source_importance = self.compute_parameter_importance(source_task_embedding)
        target_importance = self.compute_parameter_importance(target_task_embedding)

        # Initialize freeze dictionary
        freeze = {}

        # Determine which parameters to freeze
        for name in source_importance.keys():
            # Parameters important for source but not for target should be frozen
            if source_importance[name] > 0.5 and target_importance[name] < 0.5:
                freeze[name] = True
            # Parameters important for both source and target should be fine-tuned
            elif source_importance[name] > 0.5 and target_importance[name] > 0.5:
                freeze[name] = False
            # Parameters not important for source but important for target should be reinitialized
            elif source_importance[name] < 0.5 and target_importance[name] > 0.5:
                freeze[name] = False
            # Parameters not important for either task can be frozen
            else:
                freeze[name] = True

        return freeze

class KnowledgeTransferPromoter:
    def __init__(self, model, hidden_dim, num_tasks):
        self.model = model
        self.hidden_dim = hidden_dim
        self.num_tasks = num_tasks

        # Components
        self.shared_learner = SharedRepresentationLearner(hidden_dim, num_tasks)
        self.transferability_evaluator = TransferabilityEvaluator(hidden_dim)
```

```
        self.fine_tuning_controller = SelectiveFineTuningController(model, hidden_dim)

        # Task embeddings
        self.task_embeddings = nn.Parameter(torch.randn(num_tasks, hidden_dim))

    def promote_transfer(self, source_task_id, target_task_id, source_data, target_data):
        """
        Promote knowledge transfer from source to target task.

        Args:
            source_task_id: The ID of the source task.
            target_task_id: The ID of the target task.
            source_data: Data from the source task.
            target_data: Data from the target task.

        Returns:
            A dictionary of transfer parameters.
        """
        # Get task embeddings
        source_embedding = self.task_embeddings[source_task_id]
        target_embedding = self.task_embeddings[target_task_id]

        # Evaluate transferability
        transferability = self.transferability_evaluator.evaluate_transferability(source_embedding, target_embedding)

        # Control fine-tuning
        freeze = self.fine_tuning_controller.control_fine_tuning(source_embedding, target_embedding, transferability)

        # Learn shared representation
        _, source_repr = self.shared_learner(source_data, source_task_id)
        _, target_repr = self.shared_learner(target_data, target_task_id)

        # Compute representation similarity
        similarity = F.cosine_similarity(source_repr, target_repr, dim=1).mean()

        # Combine transfer parameters
        transfer_params = {
            "transferability": transferability,
            "freeze": freeze,
            "similarity": similarity
        }

        return transfer_params
```

知識転移促進は、共有表現学習器、転移可能性評価器、選択的ファインチューニング制御器から構成される。共有表現学習器は、複数のタスクに共通する特徴表現を学習する。転移可能性評価器は、ソースタスクからターゲットタスクへの知識転移の可能性を評価する。選択的ファインチューニング制御器は、転移学習の際に、どのパラメータを凍結し、どのパラメータを更新するかを決定する。

6.8.4 学習進捗監視の実装

学習進捗監視は、学習の進捗を監視し、停滞や過学習の兆候を検出して対応策を講じる機能である。具体的な実装は以下の通りである。

```python
class PerformanceMetricTracker:
    def __init__(self, window_size=10):
        self.window_size = window_size
        self.metrics = {}

    def update_metric(self, name, value):
        """
        Update a performance metric.
```

```
        Args:
            name: The name of the metric.
            value: The value of the metric.
        """
        if name not in self.metrics:
            self.metrics[name] = []

        self.metrics[name].append(value)

        # Keep only the most recent values
        if len(self.metrics[name]) > self.window_size:
            self.metrics[name] = self.metrics[name][-self.window_size:]

    def get_metric(self, name):
        """
        Get the values of a performance metric.

        Args:
            name: The name of the metric.

        Returns:
            A list of metric values.
        """
        return self.metrics.get(name, [])

    def get_latest_metric(self, name):
        """
        Get the latest value of a performance metric.

        Args:
            name: The name of the metric.

        Returns:
            The latest metric value, or None if the metric doesn't exist.
        """
        values = self.get_metric(name)
        return values[-1] if values else None

    def get_metric_trend(self, name):
        """
        Get the trend of a performance metric.

        Args:
            name: The name of the metric.

        Returns:
            The trend of the metric (positive, negative, or neutral).
        """
        values = self.get_metric(name)

        if len(values) < 2:
            return "neutral"

        # Compute linear regression
        x = np.arange(len(values))
        y = np.array(values)
        slope, _ = np.polyfit(x, y, 1)

        # Determine trend
        if slope > 0.001:
            return "positive"
        elif slope < -0.001:
            return "negative"
        else:
            return "neutral"
```

```
class StagnationDetector:
    def __init__(self, patience=5, min_improvement=0.001):
        self.patience = patience
        self.min_improvement = min_improvement
        self.best_value = float('inf')
        self.stagnation_counter = 0

    def detect_stagnation(self, value, minimize=True):
        """
        Detect stagnation in learning.

        Args:
            value: The current value of the monitored metric.
            minimize: Whether the metric should be minimized (True) or maximized (False).

        Returns:
            A boolean indicating whether stagnation has been detected.
        """
        # Adjust value based on optimization direction
        if not minimize:
            value = -value

        # Check if value has improved
        if value < self.best_value - self.min_improvement:
            self.best_value = value
            self.stagnation_counter = 0
            return False
        else:
            self.stagnation_counter += 1
            return self.stagnation_counter >= self.patience

    def reset(self):
        """
        Reset the stagnation detector.
        """
        self.best_value = float('inf')
        self.stagnation_counter = 0

class OverfittingMonitor:
    def __init__(self, patience=5, threshold=0.05):
        self.patience = patience
        self.threshold = threshold
        self.overfitting_counter = 0

    def detect_overfitting(self, train_loss, val_loss):
        """
        Detect overfitting.

        Args:
            train_loss: The current training loss.
            val_loss: The current validation loss.

        Returns:
            A boolean indicating whether overfitting has been detected.
        """
        # Compute gap between train and validation loss
        gap = val_loss - train_loss

        # Check if gap exceeds threshold
        if gap > self.threshold:
            self.overfitting_counter += 1
            return self.overfitting_counter >= self.patience
        else:
            self.overfitting_counter = 0
            return False
```

```
    def reset(self):
        """
        Reset the overfitting monitor.
        """
        self.overfitting_counter = 0

class LearningProgressMonitor:
    def __init__(self, patience=5, min_improvement=0.001, overfitting_threshold=0.05):
        self.patience = patience
        self.min_improvement = min_improvement
        self.overfitting_threshold = overfitting_threshold

        # Components
        self.metric_tracker = PerformanceMetricTracker()
        self.stagnation_detector = StagnationDetector(patience, min_improvement)
        self.overfitting_monitor = OverfittingMonitor(patience, overfitting_threshold)

    def update_metrics(self, metrics):
        """
        Update performance metrics.

        Args:
            metrics: A dictionary of metric names and values.
        """
        for name, value in metrics.items():
            self.metric_tracker.update_metric(name, value)

    def monitor_progress(self):
        """
        Monitor learning progress and detect issues.

        Returns:
            A dictionary of monitoring results.
        """
        # Get latest metrics
        train_loss = self.metric_tracker.get_latest_metric("train_loss")
        val_loss = self.metric_tracker.get_latest_metric("val_loss")

        # Detect stagnation
        stagnation_detected = False
        if val_loss is not None:
            stagnation_detected = self.stagnation_detector.detect_stagnation(val_loss)

        # Detect overfitting
        overfitting_detected = False
        if train_loss is not None and val_loss is not None:
            overfitting_detected = self.overfitting_monitor.detect_overfitting(train_loss, val_loss)

        # Get metric trends
        metric_trends = {}
        for name in self.metric_tracker.metrics.keys():
            metric_trends[name] = self.metric_tracker.get_metric_trend(name)

        # Combine monitoring results
        results = {
            "stagnation_detected": stagnation_detected,
            "overfitting_detected": overfitting_detected,
            "metric_trends": metric_trends,
            "latest_metrics": {name: self.metric_tracker.get_latest_metric(name) for name in
self.metric_tracker.metrics.keys()}
        }

        return results

    def reset(self):
```

```
        """
        Reset the learning progress monitor.
        """
        self.stagnation_detector.reset()
        self.overfitting_monitor.reset()
```

学習進捗監視は、性能指標トラッカー、停滞検出器、過学習モニターから構成される。性能指標トラッカーは、精度、損失、F値などの性能指標を追跡し、学習の進捗を評価する。停滞検出器は、学習の停滞（性能の向上が見られない状態）を検出し、対応策を提案する。過学習モニターは、過学習（トレーニングセットでの性能は向上するが、検証セットでの性能は低下する状態）を検出し、対応策を提案する。

6.8.5 メタ学習コントローラの統合

上記のサブコンポーネントを統合して、完全なメタ学習コントローラを実装する。

```python
class MetaLearningController:
    def __init__(self, model, knowledge_base, hidden_dim, num_tasks, base_lr=0.001, base_reg=0.001, base_clip=1.0, patience=5, min_improvement=0.001, overfitting_threshold=0.05):
        self.model = model
        self.knowledge_base = knowledge_base
        self.hidden_dim = hidden_dim
        self.num_tasks = num_tasks

        # Components
        self.task_analyzer = TaskCharacteristicsAnalyzer(knowledge_base, hidden_dim)
        self.update_controller = ParameterUpdateController(base_lr, base_reg, base_clip)
        self.transfer_promoter = KnowledgeTransferPromoter(model, hidden_dim, num_tasks)
        self.progress_monitor = LearningProgressMonitor(patience, min_improvement, overfitting_threshold)

        # Task ID mapping
        self.task_id_mapping = {}
        self.next_task_id = 0

    def get_task_id(self, task_name):
        """
        Get the ID for a task.

        Args:
            task_name: The name of the task.

        Returns:
            The ID of the task.
        """
        if task_name not in self.task_id_mapping:
            if self.next_task_id >= self.num_tasks:
                raise ValueError(f"Maximum number of tasks ({self.num_tasks}) exceeded")
            self.task_id_mapping[task_name] = self.next_task_id
            self.next_task_id += 1

        return self.task_id_mapping[task_name]

    def analyze_task(self, task_data, task_name):
        """
        Analyze a task.

        Args:
            task_data: Data representing the task.
            task_name: The name of the task.

        Returns:
            A dictionary of task characteristics.
        """
```

```
        # Get task ID
        task_id = self.get_task_id(task_name)

        # Analyze task
        characteristics = self.task_analyzer.analyze_task(task_data)

        # Add task ID to characteristics
        characteristics["task_id"] = task_id

        return characteristics

    def control_update(self, task_characteristics, learning_progress, gradients):
        """
        Control parameter update.

        Args:
            task_characteristics: Characteristics of the task.
            learning_progress: Information about learning progress.
            gradients: The gradients for parameter update.

        Returns:
            A dictionary of update parameters.
        """
        return self.update_controller.control_update(task_characteristics, learning_progress, gradients)

    def promote_transfer(self, source_task_name, target_task_name, source_data, target_data):
        """
        Promote knowledge transfer from source to target task.

        Args:
            source_task_name: The name of the source task.
            target_task_name: The name of the target task.
            source_data: Data from the source task.
            target_data: Data from the target task.

        Returns:
            A dictionary of transfer parameters.
        """
        # Get task IDs
        source_task_id = self.get_task_id(source_task_name)
        target_task_id = self.get_task_id(target_task_name)

        return self.transfer_promoter.promote_transfer(source_task_id, target_task_id, source_data, target_data)

    def update_metrics(self, metrics):
        """
        Update performance metrics.

        Args:
            metrics: A dictionary of metric names and values.
        """
        self.progress_monitor.update_metrics(metrics)

    def monitor_progress(self):
        """
        Monitor learning progress and detect issues.

        Returns:
            A dictionary of monitoring results.
        """
        return self.progress_monitor.monitor_progress()

    def adjust_strategy(self, task_characteristics, monitoring_results):
        """
        Adjust learning strategy based on task characteristics and monitoring results.
```

```
        Args:
            task_characteristics: Characteristics of the task.
            monitoring_results: Results from progress monitoring.

        Returns:
            A dictionary of strategy adjustments.
        """
        adjustments = {}

        # Adjust learning rate if stagnation is detected
        if monitoring_results["stagnation_detected"]:
            adjustments["learning_rate_multiplier"] = 0.5

        # Increase regularization if overfitting is detected
        if monitoring_results["overfitting_detected"]:
            adjustments["regularization_multiplier"] = 2.0

        # Adjust based on metric trends
        if "val_loss" in monitoring_results["metric_trends"]:
            if monitoring_results["metric_trends"]["val_loss"] == "positive":
                # Validation loss is increasing
                adjustments["early_stopping"] = True
            elif monitoring_results["metric_trends"]["val_loss"] == "neutral":
                # Validation loss is plateauing
                adjustments["learning_rate_multiplier"] = 0.7

        # Adjust based on task complexity
        complexity = task_characteristics["complexity"]
        if complexity > 0.8:
            # High complexity task
            adjustments["batch_size_multiplier"] = 0.5
            adjustments["model_capacity_multiplier"] = 2.0
        elif complexity < 0.2:
            # Low complexity task
            adjustments["batch_size_multiplier"] = 2.0
            adjustments["model_capacity_multiplier"] = 0.5

        return adjustments
```

メタ学習コントローラは、タスク特性分析器、パラメータ更新制御器、知識転移促進器、学習進捗監視器から構成される。これらのコンポーネントを組み合わせて、システム全体の学習プロセスを監視し、タスクの性質や難易度に応じてパラメータ更新戦略を調整する。

6.9. システム全体の統合

上記のコンポーネントを統合して、完全なマルチモーダル適応的反射システムを実装する。

```python
class MultimodalAdaptiveReflectionSystem:
    def __init__(self, hidden_dim, output_dim, modality_configs, knowledge_base=None, num_tasks=10,
integration_type='weighted'):
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim

        # Create knowledge base if not provided
        if knowledge_base is None:
            knowledge_base = ExtensibleKnowledgeBase()
        self.knowledge_base = knowledge_base

        # Components
        self.multimodal_encoder = MultimodalEncoder(hidden_dim, modality_configs)
        self.context_aware_network = ContextAwareNetwork(hidden_dim, hidden_dim, hidden_dim * 4, 6)
        self.output_generation_layer = OutputGenerationLayer(hidden_dim, output_dim)
```

```
        # Create modality dimensions dictionary for reflection module
        modality_dims = {modality: hidden_dim for modality in modality_configs.keys()}
        self.hierarchical_reflection_module = HierarchicalReflectionModule(
            hidden_dim, hidden_dim, hidden_dim // 2, output_dim, knowledge_base, modality_dims, integration_type
        )

        self.feedback_loop_integrator = FeedbackLoopIntegrator(knowledge_base, hidden_dim, output_dim)
        self.meta_learning_controller = MetaLearningController(self, knowledge_base, hidden_dim, num_tasks)

    def forward(self, inputs, task_name=None, update_memory=True):
        """
        Forward pass through the system.

        Args:
            inputs: A dictionary mapping modality names to input tensors.
            task_name: The name of the task, or None.
            update_memory: Whether to update memory.

        Returns:
            The final output.
        """
        # Analyze task if task name is provided
        task_characteristics = None
        if task_name is not None:
            # Extract task data from inputs
            task_data = torch.cat([input.mean(dim=1) for input in inputs.values()], dim=1)
            task_characteristics = self.meta_learning_controller.analyze_task(task_data, task_name)

        # Encode inputs
        fused_representation, modality_representations = self.multimodal_encoder(inputs)

        # Process through context-aware network
        context_embedding, uncertainty = self.context_aware_network(fused_representation, update_memory)

        # Generate initial output
        initial_output = self.output_generation_layer(context_embedding)

        # Generate reflection vector
        reflection_vector, reflection_info = self.hierarchical_reflection_module(
            context_embedding, initial_output, modality_representations, None, update_memory, None
        )

        # Integrate feedback
        final_output = self.feedback_loop_integrator.integrate(
            context_embedding, initial_output, reflection_vector, reflection_info
        )

        return final_output, {
            "initial_output": initial_output,
            "reflection_vector": reflection_vector,
            "reflection_info": reflection_info,
            "context_embedding": context_embedding,
            "uncertainty": uncertainty,
            "task_characteristics": task_characteristics
        }

    def train_step(self, inputs, targets, task_name=None, optimizer=None):
        """
        Perform a training step.

        Args:
            inputs: A dictionary mapping modality names to input tensors.
            targets: The target outputs.
            task_name: The name of the task, or None.
            optimizer: The optimizer to use, or None to create a new one.
```

```
        Returns:
            The loss value.
        """
        # Forward pass
        outputs, info = self.forward(inputs, task_name)

        # Compute loss
        loss = F.mse_loss(outputs, targets)

        # Update metrics
        metrics = {
            "train_loss": loss.item()
        }
        self.meta_learning_controller.update_metrics(metrics)

        # Monitor progress
        monitoring_results = self.meta_learning_controller.monitor_progress()

        # Adjust strategy if needed
        if info["task_characteristics"] is not None:
            adjustments = self.meta_learning_controller.adjust_strategy(info["task_characteristics"], monitoring_results)

            # Apply adjustments
            # This is a simplified implementation
            # In practice, this would involve more sophisticated adjustment application
            if "learning_rate_multiplier" in adjustments and optimizer is not None:
                for param_group in optimizer.param_groups:
                    param_group['lr'] *= adjustments["learning_rate_multiplier"]

        # Backward pass
        if optimizer is not None:
            optimizer.zero_grad()
            loss.backward()

            # Get gradients
            gradients = [p.grad for p in self.parameters() if p.grad is not None]

            # Control update if task characteristics are available
            if info["task_characteristics"] is not None:
                learning_progress = {
                    "current_epoch": 0,  # Placeholder
                    "total_epochs": 0,  # Placeholder
                    "train_loss": metrics["train_loss"],
                    "val_loss": self.meta_learning_controller.progress_monitor.metric_tracker.get_latest_metric("val_loss") or
0.0
                }
                update_params = self.meta_learning_controller.control_update(info["task_characteristics"],
learning_progress, gradients)

                # Apply controlled update
                # This is a simplified implementation
                # In practice, this would involve more sophisticated update application
                if "stabilized_gradients" in update_params:
                    for p, g in zip(self.parameters(), update_params["stabilized_gradients"]):
                        if p.grad is not None:
                            p.grad = g

            optimizer.step()

        return loss.item()

    def transfer_knowledge(self, source_task_name, target_task_name, source_data, target_data):
        """
        Transfer knowledge from source to target task.
```

```
        Args:
            source_task_name: The name of the source task.
            target_task_name: The name of the target task.
            source_data: Data from the source task.
            target_data: Data from the target task.

        Returns:
            A dictionary of transfer parameters.
        """
        return self.meta_learning_controller.promote_transfer(source_task_name, target_task_name, source_data,
target_data)

    def parameters(self):
        """
        Get the parameters of the model.

        Returns:
            An iterator over the parameters.
        """
        for component in [self.multimodal_encoder, self.context_aware_network, self.output_generation_layer,
self.hierarchical_reflection_module]:
            for param in component.parameters():
                yield param
```

マルチモーダル適応的反射システムは、マルチモーダルエンコーダ、コンテキスト認識ネットワーク本体、出力生成層、階層的反射モジュール、フィードバックループ統合器、メタ学習コントローラから構成される。これらのコンポーネントを組み合わせて、異なるモダリティからの情報を統合し、動的に適応する反射機構を備えた高効率な神経記号的推論システムを実現する。

7. 産業上の利用可能性

本発明は、以下のような幅広い産業分野で利用可能である。

7.1. 医療診断支援

複数のモダリティ（患者の症状記述、医療画像、検査結果など）を統合し、信頼性の高い診断候補と推論過程を提供するシステム。医師の診断精度向上と業務効率化に貢献する。特に、複雑な症例や希少疾患の診断、遠隔医療などの分野で有用である。

具体的な応用例：
- 放射線画像（X線、MRI、CT）と患者の臨床データを統合した診断支援
- 病理画像と遺伝子データを組み合わせたがん診断
- 電子カルテデータと医療画像を統合した疾患予測
- 複数の専門分野にまたがる複合疾患の診断支援

7.2. 金融リスク分析

テキストデータ（ニュース、報告書）、数値データ（市場指標、財務諸表）、時系列データ（株価推移）を統合し、投資リスクを評価するシステム。投資判断の精度向上とリスク管理の強化に貢献する。

具体的な応用例：
- 企業の財務データ、ニュース記事、ソーシャルメディアの感情分析を統合した信用リスク評価
- 市場データ、マクロ経済指標、地政学的イベントを考慮した投資戦略の最適化
- 取引パターン、顧客プロファイル、通信データを統合した不正検出
- 複数の情報源からのデータを統合した市場予測

7.3. 自動運転

視覚データ（カメラ映像）、センサーデータ（LiDAR、レーダー）、地図データを統合し、安全な運転判断を行うシステム。自動運転車の安全性と信頼性の向上に貢献する。

具体的な応用例：
- カメラ映像、LiDARデータ、レーダー情報を統合した障害物検出と回避
- 交通標識認識、車線検出、歩行者追跡を統合した運転環境理解
- 地図データ、GPS情報、交通状況を考慮した経路計画
- 天候条件、道路状態、交通ルールを考慮した運転戦略の適応

7.4. スマート製造

生産ラインのセンサーデータ、品質検査画像、作業指示書を統合し、製造プロセスを最適化するシステム。生産効率の向上と品質管理の強化に貢献する。

具体的な応用例：
- センサーデータ、検査画像、製品仕様を統合した品質管理
- 生産ラインの状態監視、故障予測、メンテナンス計画の最適化
- 材料特性、加工パラメータ、環境条件を考慮した製造プロセスの最適化
- 作業指示書、作業者のパフォーマンスデータ、生産スケジュールを統合した作業計画

7.5. 教育テクノロジー

学習者の行動データ、テスト結果、テキスト回答を統合し、パーソナライズされた学習支援を提供するシステム。教育の効率化と学習成果の向上に貢献する。

具体的な応用例：
- 学習履歴、テスト結果、学習スタイルを考慮したパーソナライズされた学習コンテンツの提供
- 学習者の回答、問題解決プロセス、参照資料の使用パターンを分析した学習困難の診断
- 学習進捗、興味・関心、学習目標に基づいた最適な学習経路の提案
- 複数の学習者のデータを統合した協調学習の促進と教育効果の評価

7.6. 科学的発見

科学文献、実験データ、シミュレーション結果を統合し、新しい仮説を生成するシステム。科学研究の加速と新しい知見の発見に貢献する。

具体的な応用例：
- 科学論文、実験データ、分子構造情報を統合した新薬候補の探索
- 気象データ、地理情報、歴史的記録を統合した気候変動予測
- 遺伝子データ、タンパク質構造、代謝経路情報を統合した生物学的機能の予測
- 物理実験データ、理論モデル、シミュレーション結果を統合した新しい物理現象の探索

7.7. カスタマーサービス

顧客の問い合わせテキスト、音声、購入履歴を統合し、適切な対応を提供するシステム。顧客満足度の向上と業務効率化に貢献する。

具体的な応用例：
- 顧客の問い合わせ内容、感情分析、過去の対応履歴を統合したパーソナライズされた対応
- 音声通話、チャットログ、顧客プロファイルを考慮した問題解決の最適化
- 購入履歴、閲覧行動、顧客フィードバックを統合した商品推奨
- 複数のチャネル（電話、メール、SNS）からの顧客情報を統合したシームレスな対応

7.8. セキュリティ監視
監視カメラ映像、アクセスログ、異常検知アラートを統合し、セキュリティ脅威を特定するシステム。セキュリティ対策の強化と迅速な対応に貢献する。

具体的な応用例：
- 監視カメラ映像、入退室記録、ネットワークアクセスログを統合した不審行動の検出
- システムログ、ユーザー行動、ネットワークトラフィックを分析したサイバー攻撃の早期検出
- 物理セキュリティシステム、ITセキュリティシステム、人的セキュリティ対策を統合した包括的な脅威管理
- 過去のセキュリティインシデント、現在の脅威情報、リスク評価を統合した予防的セキュリティ対策

本発明は、特に複数のデータソースからの情報統合と、不確実性の高い状況での信頼性の高い推論が求められる分野で大きな価値を提供する。また、継続的学習と知識転移の能力により、限られたデータでも効果的に機能し、時間の経過とともに性能が向上するシステムの構築を可能にする。

8. 要約

【課題】異なるモダリティからの情報を統合し、動的に適応する反射機構を備えた高効率な神経記号的推論システムを提供する。既存の神経記号的システムは単一モダリティの処理に限定され、反射メカニズムが静的で、知識ベースの拡張能力や知識転移機能が欠如しており、反射プロセスと出力生成プロセス間のフィードバック機構がない。
【解決手段】マルチモーダルエンコーダ、コンテキスト認識ネットワーク本体、出力生成層、階層的反射モジュール、拡張可能知識ベース、フィードバックループ統合器、メタ学習コントローラを備える神経記号的人工知能システム。マルチモーダルエンコーダが異なるモダリティからの入力を統一された表現に変換し、コンテキスト認識ネットワーク本体がタスク固有の文脈を考慮した埋め込みを生成する。階層的反射モジュールが反射ベクトルを生成し、フィードバックループ統合器が反射ベクトルに基づいて誤りを識別し、拡張可能知識ベースを用いて修正候補を生成する。

9. 先行技術文献

【非特許文献1】Garcez, A. D., Lamb, L. C. "Neurosymbolic AI: The 3rd Wave", arXiv preprint arXiv:2012.05876, 2020
【非特許文献2】Mao, J., Gan, C., Kohli, P., Tenenbaum, J. B., Wu, J. "The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision", arXiv preprint arXiv:1904.12584, 2019
【非特許文献3】Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., Tenenbaum, J. "Neural-symbolic VQA: Disentangling reasoning from vision and language understanding", Advances in Neural Information Processing Systems, vol. 31, 2018
【非特許文献4】Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., De Raedt, L. "DeepProbLog: Neural probabilistic logic programming", Advances in Neural Information Processing Systems, vol. 31, 2018
【非特許文献5】Evans, R., Grefenstette, E. "Learning explanatory rules from noisy data", Journal of Artificial Intelligence Research, vol. 61, pp. 1-64, 2018
【非特許文献6】Hu, W., Zhao, Y., Zhang, Y., Qiao, J., Xie, Y., & Zhou, Z. (2025). ABL-Refl: Augmented Bayesian Learning with Reflection. In Proceedings of the 39th AAAI Conference on Artificial Intelligence (AAAI-25). Association for the Advancement of Artificial Intelligence.