PAMIQ Core: リアルタイム継続学習のための非同期推論・学習フレームワーク

○ 下村 晏弘 (ジー・オー・ピー株式会社)
大澤 弘基 (株式会社インフィニットループ)
中村 建一 (株式会社 Preferred Networks)

本研究では，リアルタイム継続学習のための Python フレームワーク **“PAMIQ Core”**を提案する．本フレームワークは制御・推論・学習のマルチスレッドアーキテクチャにより，環境とのインタラクション（推論）と機械学習モデルの学習を非同期に実行する．モデルパラメータやデータの同期処理において，メモリの参照先を移動する方式を導入することで，推論スレッドへの影響を最小限に抑制した．状態永続化や制御コンソールなどの運用支援機能に加えて，PyTorch や Gymnasium などの統合機能も持つ．我々は PAMIQ Core が，現実的な世界において継続的に成長する複雑かつ自律的な機械知能の実現に向けた，基盤ツールとなることを目指している．

1. はじめに

一般的な深層学習は，大規模なオフライン学習の後にパラメータ更新を行わない静的なパイプラインで実行することが主流である． 強化学習においても，環境とのインタラクション（推論）と学習処理を逐次的に交互実行することが一般的であり，学習中は推論処理が停止する． これらのアプローチは，環境への動的適応や推論処理を停止させることができない現実的な世界における，自律的な機械知能の適用を困難にしている．

リアルタイムかつ継続的にインタラクションが必要な空間の一つにソーシャル VR 空間（VRChat[1]など）がある．我々は好奇心ベースの自律機械知能 **“PAMIQ”** [1] をその空間に実装してきた結果，そのシステムは汎用的に利用できる構造があり，再利用可能なフレームワークとする価値があった．

本研究では，リアルタイム・継続的な推論と学習の非同期実行を実現する Python[2]フレームワーク **“PAMIQ Core”** を提案する[3]． その特徴は次の通りである：

- **マルチスレッドアーキテクチャ**：シンプルな 3 スレッド静的並列（制御・推論・学習）による実装．
- **モジュラー設計**：抽象基底クラスを継承し，拡張実装を行う形式．
- **既存フレームワークとの統合**：深層学習のための PyTorch[2] や強化学習のための環境を定義する Gymnasium[3] との統合機能．
- **運用支援機能**:状態永続化やインタラクティブな制御コンソール，時間進行の調整が可能．
- **段階的なサンプル**：最小実装から実用的なサンプル，さらに VRChat 上への実装サンプルを用意．
- **即時利用性**：Python Package Index へ公開（`pip install pamiq-core` でインストール可能）．

PAMIQ Core は，現実的な環境において継続的に成長し続ける，自律的な機械知能を実現可能にする[4]．

[1]https://hello.vrchat.com
[2]https://www.python.org
[3]https://github.com/MLShukai/pamiq-core
[4]https://youtu.be/hYjet0v17rY

1.1 関連研究

既存の推論と学習を同時実行するフレームワークとして，RLlib [4] などが存在する． このフレームワークは，強化学習エージェントのサンプル効率改善と計算機の使用率最大化を主目的として設計されている．マルチプロセス実装による高いスケーラビリティを実現している点において，PAMIQ Core よりも優れている．

一方，PAMIQ Core の開発目的は現実的な空間での長期継続学習と運用性の実現である．そのため強化学習のみならず画像認識といった一般的な機械学習のオンライン学習も対象とする．またマルチスレッド設計により単一プロセスで完結させることで，並行処理間での通信速度やデータ構造の制限を緩和し，柔軟な実装を可能とする． 軽量なスレッドベースシステムであるため，他の複雑なロボティクスシステムにも組み込みやすいという利点を有している．

2. システムの設計

PAMIQ Core のシステムの設計概要を図 1 に示す．**推論スレッド**は実環境とのインタラクション処理を実行し，学習データの収集も行う． **学習スレッド**は収集されたデータを用いてモデルの内部パラメータを更新し，同期処理を実行する．**制御スレッド**はシステム全体の一時停止，再開，終了といった運用処理を管理する．

推論処理を停止させないための工夫として，同期処理ではデータのコピーではなくメモリの参照先移動を採用した．これにより同期のための処理ロック時間を

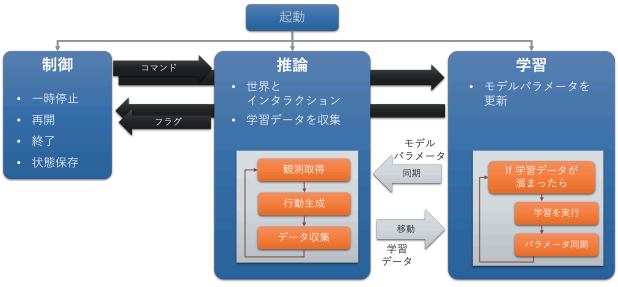

図 1: システム全体の設計概要図．

最小化する． また時間のかかる処理は全て学習スレッド上で実行するように設計し，止まることのない継続的な推論動作を保証する．

システムはコンポーネントとなる抽象基底クラスを継承し拡張実装を行う形式を採用した．開発者はコールバックメソッドをオーバーライドすることで，各コンポーネントの機能を実装する構造である．

デッドロック等の問題が発生しうる非同期処理の制御は，開発が難航する要因となりうる． ユーザーから制御処理を隠蔽し，開発を簡単にすることに努めた．

3. 主要な使い方

PAMIQ Core を使用する際は，パッケージをインストールした後，**pamiq_core** モジュールから必要なコンポーネントを取得する．取得可能な主要コンポーネントを次に示す：

- **launch**，**LaunchConfig**：システムの起動関数とその設定を行うためのクラス．
- **Interaction**，**Agent**，**Environment**：強化学習的な観測行動ループを実装する．
- **TrainingModel**，**InferenceModel**：機械学習モデルの定義や推論処理，同期処理の実装を行う．
- **DataBuffer**，**DataCollector**，**DataUser**：学習データの貯蔵と収集，利用を行う．
- **Trainer**：学習処理や実行可能条件を記述する．

これらの詳細な扱い方はリファレンスページ [5] を参照されたい．

3.1 実行の流れ

PAMIQ Core システムは，以下のように launch 関数を呼び出すことで起動する．

```
launch(
  interaction=Interaction(agent, environment),
  models={"model_name": training_model},
  buffers={"buffer_name": data_buffer},
  trainers={"trainer_name": trainer},
  config=LaunchConfig(
    ... # 設定値を記述
  )
)
```

launch 関数には，ユーザーが定義した Interaction（Agent と Environment），TrainingModel，DataBuffer，Trainer，および起動設定である LaunchConfig を引数として渡す． 名前付きで渡されたモデルおよびデータバッファは，Agent と Trainer に受け渡される．Agent には推論用のモデル InferenceModel とデータ収集用の DataCollector が渡され，Trainer には学習用のモデル TrainingModel と収集したデータを使うための DataUser が渡される． その後推論スレッドと学習スレッドがバックグラウンドで動作を開始し，制御スレッドがメインスレッドで処理を開始する．

3.2 環境とエージェントの相互作用

一般的な強化学習の観測・行動ループを実現するためには，Agent, Environment, Interaction クラス扱う．

抽象基底クラス Agent は，モデルによる行動生成およびデータ収集処理を記述する step メソッドの実装を求める． InferenceModel と DataCollector は基底クラスのイベントフックメソッド：on_inference_models_attached, on_data_collectors_attached をオーバーライドし，それらの中でメソッド：get_inference_model("name"), get_data_collector("name")を用いて取得する．この時に指定する"name"は launch 関数に与えたものである．

抽象基底クラス Environment では観測を取得する observe および行動を実環境に作用させる affectメソッドの実装を要求する．

Interaction クラスは Environment と Agent のインスタンスを受け取り，これらを結びつけて観測・行動ループを実現する．

3.3 機械学習モデルの定義

PAMIQ Core フレームワークには，Agent 内で推論処理に用いるための InferenceModel クラスと，Trainer 内で学習処理に用いるための TrainingModel クラスが存在する． 抽象基底クラス InferenceModel は推論処理 infer メソッドの実装，抽象基底クラス TrainingModel は推論用モデルを返す_create_inference_model，順伝播処理 forward，同期処理 sync_impl メソッドの実装を要求する． これらの基底クラスは非同期処理の同期ロック機構を提供しないため，推論処理中にパラメータが書き換えられないよう同期処理を実装する必要がある：

```
class CustomInferenceModel(InferenceModel):
  def __init__(self):
    self.lock = Lock()

  def infer(self, ... ):
    with self.lock:
      ... # 推論処理

class CustomTrainingModel(TrainingModel):
  def sync_impl(self, inference_model):
    with inference_model.lock:
      ... # 同期処理
```

3.4 学習データの収集，貯蔵と利用

DataBuffer，DataCollector，DataUser クラスは学習データの収集，貯蔵，そして利用を行う際に用いる．

抽象基底 DataBuffer クラスはデータを貯める機能を実装し，データを蓄積する add，データを返す get_data，および蓄積されたデータ数を返す__len__の実装を求

[5] https://mlshukai.github.io/pamiq-core/

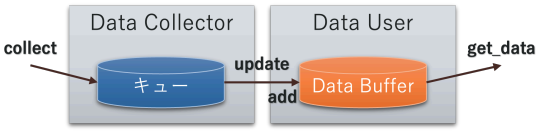

図 2: クラスの接続とデータの流れ.

める．DataCollector クラスは Agent 内でデータを収集するインターフェイスであり，collect メソッドを利用してデータを一時的なキューに貯める． DataUser クラスは Trainer 内で収集されたデータを利用するためのインターフェイスであり，update メソッドで DataCollector の内部キューを受け取り，DataBuffer の add メソッドをへ渡す．（図 2）

3.5 モデルの学習処理

学習処理は抽象基底 Trainer クラスを継承し，train メソッドを実装することで記述する．

継続学習を安定的に実行するには，十分な学習データが蓄積されており，かつ重複した学習を行わないよう一定以上データが更新されたことを確認する必要がある． そのため Trainer クラスのコンストラクタには十分な学習データが蓄積されていることを確認する min_buffer_size パラメータと，重複した学習を避けるため一定以上データが更新されたことを確認する min_new_data_count パラメータが用意されている． 次の例では，名称“data_buffer” というバッファが，データを 128 個以上蓄積し，かつ新規データ数が 32 個以上集まった時に学習処理を実行する設定を示す：

```
class CustomTrainer(Trainer):
  def __init__(self):
    super().__init__(
      "data_buffer",
      min_buffer_size=128,
      min_new_data_count=32
    )
```

TrainingModel と DataUser の取得は基底クラスのイベントフックメソッド：on_training_models_attached, on_data_users_attached をオーバーライドし，それらの中でメソッド：get_training_model("name"), get_data_users("name")を用いて取得する．この時に指定する"name"は launch 関数に与えたものである．

4. 同期システム

4.1 モデルパラメータ

学習スレッドと推論スレッドのモデルパラメータを同期する際に，パラメータコピーを行うと$O(\text{size})$の時間計算量が必要となる．これは大規模モデルの同期処理が推論処理の妨げになり，利用しにくくなる可能性がある．

PAMIQ Core ではこれを解決するために，参照スワップによる擬似同期処理を採用している（図 3）．この方式では参照スワップ自体は$O(1)$時間で完了し，実

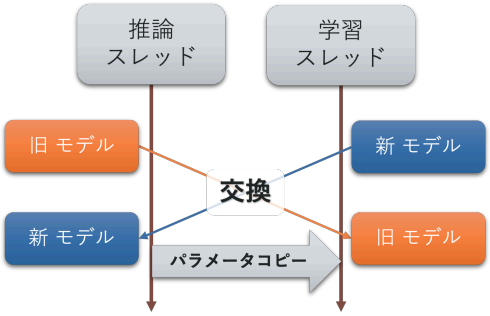

図 3: モデルの同期機構.

際のパラメータコピー処理は学習スレッドで非同期に実行される．推論時はモデルのパラメータが固定されているため，安全に読み出すことが可能である．

参照スワップ方式にはいくつかの制約がある．モデルが推論処理に依存する内部状態を持つことができないため，RNN などの隠れ状態は Agent クラスで保持する必要がある．また，PyTorch のオプティマイザなどパラメータ参照を持つものは，同期を行うたびにその参照を更新する必要がある．そのため PyTorch 統合機能の中で実装された Trainer（TorchTrainer）の内部では，自動的に参照を復元する機能を実装した．

性能ベンチマーク

参照スワップ方式の有効性を検証するため，PyTorch を用いて性能ベンチマークを実施した．実験環境は CPU: Ryzen9 7950X を使用し，モデルとしてバイアス無しの torch.nn.Linear を使い，1MiB（行列サイズ512×512）および 1GiB（行列サイズ 16384×16384）の float32 パラメータで構成した．比較のためにコピー方式と参照スワップ方式の実行時間を測定した．

```
# コピー方式
m2.load_state_dict(m1.state_dict())
# 参照スワップ
m1, m2 = m2, m1
```

表 1: モデル同期処理の性能比較結果.

モデルサイズ	コピー時間	参照スワップ時間
1MiB	11.6µs	62.6ns
1GiB	63.3ms	177.3ns

表 1 に示すように，1MiB モデルでは 185 倍，1GiB モデルでは 357,000 倍の高速化を実現した． この結果は大規模モデルを使用する継続学習システムにおいて，参照スワップ方式が推論性能に与える影響を大幅に軽減できることを示している．

4.2 データ

推論処理を停止させないために，データ移動においても参照移動を行う（図 4）．推論スレッドで収集されるデータは一時的にキューに保存され，学習スレッドへの受け渡し時にはデータのコピーではなくキュー

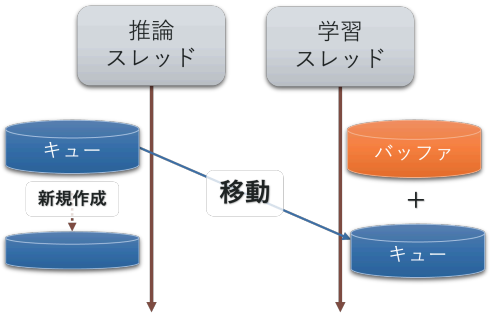

図 4: データ移動の仕組み．

オブジェクトの参照を移動，新規のキューと入れ替え，バッファと結合する．

データの物理的な移動は$O(\text{size})$の時間計算量を要するが，参照移動や新規キューの作成は$O(1)$で完了するため，推論処理への影響を最小限に抑制できる．この機構により，大量のデータが蓄積された状況においても，推論スレッドのロック時間は最小限に保たれる．

5. サンプル実装

PAMIQ Core の実用性を示すため，複数の実装サンプルを用意した． 最小実装例はリポジトリの **`samples/minimum.py`** に配置し，起動後の挙動確認を目的とした． より実践的なサンプルとして **`samples/vae-torch/`** に Variational AutoEncoder [5] の継続学習例を配置した． このサンプルでは PyTorch 統合機能を使用し，教師なし学習を継続的に実行するシステムの実装方法を示す．現実的な世界への応用例として，VRChat 環境での自律機械知能実装サンプルを独立したリポジトリ **`pamiq-vrchat`**[6] に公開した． このサンプルではソーシャル VR 空間における複雑な環境インタラクションと継続学習の統合実装を確認できる．

6. 将来性

6.1 GIL の制約と対策

PAMIQ Core のマルチスレッドアーキテクチャは，Python の Global Interpreter Lock（GIL）[7] による制約を受ける可能性がある． GIL はマルチスレッドプログラムにおいて真の並列実行を制限し，CPU 集約的なタスクでは性能向上が期待できない場合がある．

この制約に対する主要な回避策として，PyTorch の GPU 演算を活用することが挙げられる． GPU 演算は GIL の影響を受けないため，GPU 上で実行することで CPU 時間を削減できる．また Python 3.13 では実験的な GIL 無効化オプション [8] が導入されており，将来的にこの制約は解消される見込みである．

[6] https://github.com/MLShukai/pamiq-vrchat
[7] https://docs.python.org/3/glossary.html#term-global-interpreter-lock
[8] https://docs.python.org/3/whatsnew/3.13.html

6.2 既存フレームワーク統合性

PAMIQ Core の設計思想は他の機械学習フレームワークとの統合を考慮している． JAX [6] との統合は，PyTorch 統合で確立されたパターンを踏襲することで実現可能である． 適切な同期処理機構を実装することで，基本的にどのような機械学習ライブラリも PAMIQ Core システムに組み込むことができる．

7. まとめ

本論文では，リアルタイム継続学習のための非同期学習・推論フレームワーク PAMIQ Core を提案した．本フレームワークは，推論処理を停止させず継続的な学習を実現するためのものである．これは一般的な機械学習の継続学習システムにも利用可能であり，今後登場すると思われる，現実的な世界で動的に成長する複雑な機械知能の実現を支援するだろう．

PAMIQ Core の開発は，オープンソースプロジェクトとして進行しており，コントリビューションは大いに歓迎する．本フレームワークが現実的な世界において，継続的に成長し続ける自律機械知能の実現に向けた基盤ツールとして機能し，この分野の研究発展に寄与することを願う．

謝辞

PAMIQ Core の開発，論文の執筆にご協力いただいた廣瀬 裕様，杉山 聡様，堀 颯新様，早瀬 友裕様に，深く感謝いたします．誠にありがとうございました．

参 考 文 献

[1] GesonAnko et al.: “ソーシャル VR 空間に適用可能な好奇心ベースの自律機械知能”, バーチャル学会発表概要集, vol. 2024, pp. 19–22, 2024,

[2] A. Paszke et al.: “PyTorch: An Imperative Style, High-Performance Deep Learning Library”, in Advances in Neural Information Processing Systems (NeurIPS), Curran Associates, Inc., pp. 8024–8035, 2019.

[3] M. Towers et al.: “Gymnasium: A Standard Interface for Reinforcement Learning Environments”, arXiv preprint arXiv:2407.17032, 2024.

[4] E. Liang et al.: “RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem”, in Advances in Neural Information Processing Systems (NeurIPS), Curran Associates, Inc., pp. 5506–5517, 2021.

[5] D. P. Kingma and M. Welling: “Auto-Encoding Variational Bayes”, in 2nd International Conference on Learning Representations, 2014.

[6] J. Bradbury et al.: “JAX: composable transformations of Python+NumPy programs”, http://github.com/jax-ml/jax