令和 4 年度

修士学位論文

分散強化学習におけるデータ保存ライブラリの設計と実装に関する研究

Design and Implementation of a Data Storage Library for Distributed Reinforcement Learning

1255104　　仮谷 拡晃

指導教員　　松崎 公紀

2023 年 2 月 3 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

分散強化学習におけるデータ保存
ライブラリの設計と実装に関する研究

仮谷 拡晃

強化学習は，環境内のエージェントが状態を観測しながら行動を選択し，行動の結果環境から得られる報酬が最大となるような方策を学習する機械学習の手法である．複雑な環境において強化学習を行う場合は，経験の生成がモデルの訓練に追い付かずにボトルネックとなる可能性がある．そのため，経験の生成などを計算資源のスケールアウトにより分散並列処理する分散強化学習があり，A3C や Ape-X DQN などの手法が考案されている．分散強化学習では複数のアクターが非同期かつ並列に学習データを生成するが，生成された学習データは検証や解析などの追試研究のため保存されていることが望ましい．

そこで，本研究では強化学習向けの分散並列処理フレームワークである Ray 上で動作する分散強化学習向けのデータ保存ライブラリの設計と実装を行う．非同期に生成される学習データを効率よく保存するために，ファイルへの書き込み処理を Ray が提供するアクターによってリモート関数化し非同期かつ並列に処理を行う．実装したライブラリの性能をするために CartPole 環境と Breakout 環境で学習データを保存した際の実行時間を測定した．実験結果から，CartPole 環境では経験の生成を行うアクターが多くなるとアクター間のデータの受け渡しがボトルネックとなり性能が低下した．また，Breakout 環境のように状態空間のデータサイズが大きい環境ではリモート関数実行時に発生する Ray の共有メモリへのオブジェクトのコピーによるオーバーヘッドが 20% 程あるという結果が得られた．これらオーバーヘッドを減らす方法についても議論する．

キーワード　分散強化学習，アクターモデル，Ray，データ保存

Abstract

Design and Implementation of a Data Storage Library for Distributed Reinforcement Learning

Hiroaki Kariya

Reinforcement Learning (RL) is a machine learning technique, in which agents in an environment select actions by observing the environment and try maximize the reward from the environment. When performing RL in a complex environment, experience generation can be a bottleneck. Therefore, Distributed Reinforcement Learning (DRL) was designed to scale out computational resources to perform the experience generation asynchronously in parallel. DRL methods include A3C and Ape-X DQN. The generated training data should be saved for follow-up studies including verification and analysis.

In this study, we design and implement a data storage library for DRL on top of Ray, a distributed parallel processing framework for RL. In order to efficiently save the generated data, we execute the file-writing processes asynchronously by Ray actors. For performance evaluation, we measured the execution time with and without saving the training data for two environments CartPole and Breakout. Experimental results from CartPole show that, as the number of actors generating experiences increases, the transfer of data between actors becomes a bottleneck. Experimental results from Breakout, which has large states, we confirmed an overhead of about 20% in copying objects to Ray's shared memory during remote functions. We discuss future possible methods of reducing these overheads.

key words Distributed Reinforcement Learning, Actor Model, Ray, Data Storage

目次

図目次

表目次

第 1 章

序論

強化学習は，環境の状態を探索し意思決定を行うエージェントが，環境に対して行動を選択，行動の結果得られる報酬の累積が最大となるような方策を学習するという機械学習の手法である [18]．複雑な環境における強化学習では，経験の生成がモデルの訓練に追い付かずボトルネックとなる．このボトルネック解消のための手法に，経験の生成などを計算資源のスケールアウトにより分散並列処理する分散強化学習がある [9]．分散強化学習の手法には A3C [15]，Ape-X DQN [12]，Rainbow [10] などの手法が考案されている．近年，強化学習向けの分散並列処理フレームワークである Ray [16] が開発されたことにより，分散強化学習の実装が比較的容易になった．Ray が提供するアクターモデル [11] では，データの依存関係を把握しながら複数のタスクを並列に処理を行うことが可能である．

分散強化学習では複数のアクターが学習データを非同期かつ並列に生成するが，学習データは検証や解析などの追試研究のために保存されていることが望ましい．しかし，現時点の Ray にはアクターが生成するオブジェクトをファイルへ書き出す機能はなく，ユーザが実装と調整をする必要がある．

そこで，本研究では Python 上で動作する分散強化学習向けのデータ保存ライブラリの設計と実装を行う．提案するデータ保存モデルは 2 種類のプロセスで構成される．それらは，アクターが生成した経験を一時的に保存するプロセスと保存された経験を読み出してファイルへ出力するプロセスである．これらのプロセスを Ray アクターによりリモート化し並列処理することで，性能低下を抑えつつ既存のアプリケーションにデータ保存処理を追加することを目指す．

実装したライブラリの性能を，OpenAI Gym [5] が提供するシミュレーション環境を用い

て評価する．具体的には，状態のデータサイズが異なる CartPole 環境と Breakout 環境のもとで学習を行う Ape-X DQN に対してデータ保存処理を追した際の実行時間を調べることで，経験の一時保存処理のオーバーヘッドを検証する．また，ファイルへの出力タイミングでの実行時間の比較により，ファイル出力処理のオーバーヘッドの原因と大きさについて評価する．

第 2 章

関連研究

本研究で提案するデータ保存モデルの実装に用いた Ray と分散強化学習に関連する技術について記述する．

2.1 Ray

Ray は Philipp Moritz ら [16] によって開発された Python 上で分散並列処理を行うアプリケーションの実装を容易にするフレームワークである．強化学習アプリケーションに求められる性能と柔軟性などのシステム要件を満たすために，分散スケジューラとシステムの制御状態を管理するためにフォールトトレランスなオブジェクトストアを採用している．実験により，1 秒間で 180 万タスクを超えるスケーラビリティと強化学習アプリケーションで既存のシステムよりも優れた性能を発揮している [16].

Ray の特徴としてタスク並列処理とアクターベースの処理を統一したインタフェースで実装することにより，既存のアプリケーションを容易に並列化することが可能となっている．

以下の 1–4 行目のように `@ray.remote` デコレータを記述することでリモート関数を宣言できる（これをタスクと呼ぶ）．リモート関数を関数名`.remote` で呼び出すと object_ref オブジェクトが返される．object_ref オブジェクトにはリモート関数の実行結果が格納される．object_ref オブジェクトはタスクの実行終了を待たずに他のリモート関数に引数として渡すことが可能である．これにより，データの依存関係を保ったまま複数のタスクを並列に実行できる．

アクターは 5–9 行目のようにクラスを用いて定義される．一般にアクターはステートフ

ルな計算であり，8 行目の func() のようなリモートから呼び出すことができる関数を提供する．

```
1    @ray.remote
2    def func(args) {
3        // do something
4    }
5    @ray.remote
6    class Actor:
7        def func(args) {
8            // do something
9        }
```

2.1.1 アクターモデル

アクターモデルは Carl Hewit によって提案された並行計算モデルである [11]．アクターモデルでは，システムはアクターと呼ばれる並行に動作する独立した計算資源によって構成され，アクター間で相互にメッセージを非同期で送信することで目的を達成する．

アクターは受信したメッセージに対応する振る舞いとアドレスを持ち，送信先のアドレスを知っている場合に限りメッセージを送信できる．Ray においてアクターの振る舞いはリモート関数によって定義される．他のアクターモデルと同様に，Ray のアクターは実行時にはメッセージ・キューを持ち，受信したメッセージをキューに格納し，格納したメッセージを順次取り出して処理を行う [19]．

2.1.2 Ray アーキテクチャ

Ray のアーキテクチャは API を実装するアプリケーション層と高いスケーラビリティと耐障害性を提供するシステム層で構成される（図 2.1）．

アプリケーション層はドライバ，ワーカー，アクターの 3 種類のプロセスから構成されている．ドライバはユーザープログラムを実行し，ワーカーはドライバや他のワーカーが呼び

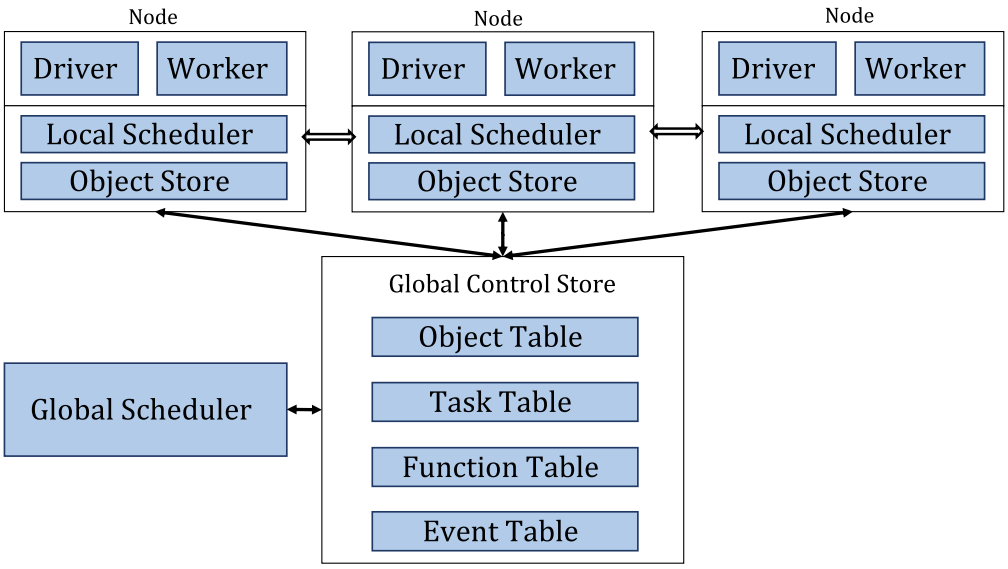

図 2.1 Ray アーキテクチャ[16]

出したリモート関数の実行を行うプロセスである．アクターはステートフルなプロセスであり，アクター内で宣言された関数のみを実行する．

システム層は Global Control Store (GCS)，分散スケジューラ，分散オブジェクトストアから構成されている．

Global Control Store (GCS)

GCS はシステムの制御状態を管理するものであり，Ray が提供する高い耐障害性と低レイテンシを維持するための重要なコンポーネントとなっている．GCS は Publish/Subscribe 機能を持った key-value ストアにより実装されている．Publish/Subscribe 機能はメッセージの生成を行うシステムとメッセージを受信するシステムを切り離して独立させることで，非同期なメッセージの送受信を可能としたモデルである [8]．これにより，システムの実行に必要な情報を全コンポーネント間で共有およびステートレス化が可能となっている．障害発生時にコンポーネントは GCS から必要な情報を読み取れば復旧が可能となるため高い耐障害性を実現している．一方，実装の観点からはリモート関数に引数として渡されたオブジェクトは全て GCS に対してコピーされ，各タスク，アクター間で共有される．

分散スケジューラ

Ray が対象とするアプリケーションでは実行時間が不明なタスクを秒間数百万個スケジューリングする必要がある．要求される要件を満たすために Ray ではグローバルスケジューラとノードごとのローカルスケジューラを用いてスケジューリングを行う．タスクの処理では，ローカルスケジューラが過負荷な場合と計算資源の不足により要件を満たせない場合を除いてタスクを生成したノードでスケジューリングされる．ローカルスケジューラでスケジューリングされないことが決定した場合，グローバルスケジューラが GCS からタスクに関する情報を取得してスケジューリングする．グローバルスケジューラは増設が可能であるため，高いスケジューリング性能の実現が可能となっている．

分散オブジェクトストア

分散オブジェクトストアは，タスクのレイテンシを最小化するために共有メモリを使用したオブジェクトストアを提供する．これにより，同一ノード上で動作するタスク間でゼロコピーのデータ共有を可能としている．具体的には，タスク実行時の入力をローカルオブジェクトストアへ複製し，出力をローカルオブジェクトストアへ書き込むことで実行時間を最小化する．

2.1.3 オブジェクト遷移

Ray でリモート関数に引数 a, b を渡して実行した際のオブジェクトの流れを図 2.2 に示す．リモート関数を実行すると GCS に登録され全てのワーカーに共有され (ステップ 0)，引数 a, b は Node1, Node2 にそれぞれ格納されリモート関数内の処理がローカルスケジューラとグローバルスケジューラに送信される (ステップ 1，2)．グローバルスケジューラは引数 a, b を GCS で検索し (ステップ 3)，引数 b が存在する Node2 にタスクを割り当てる (ステップ 4)．タスクが割り当てられた Node2 にはローカルオブジェクトストアに引数 a は存在しないため (ステップ 5)，GCS で引数 a の場所を調べ (ステップ 6)，Node1 のローカル

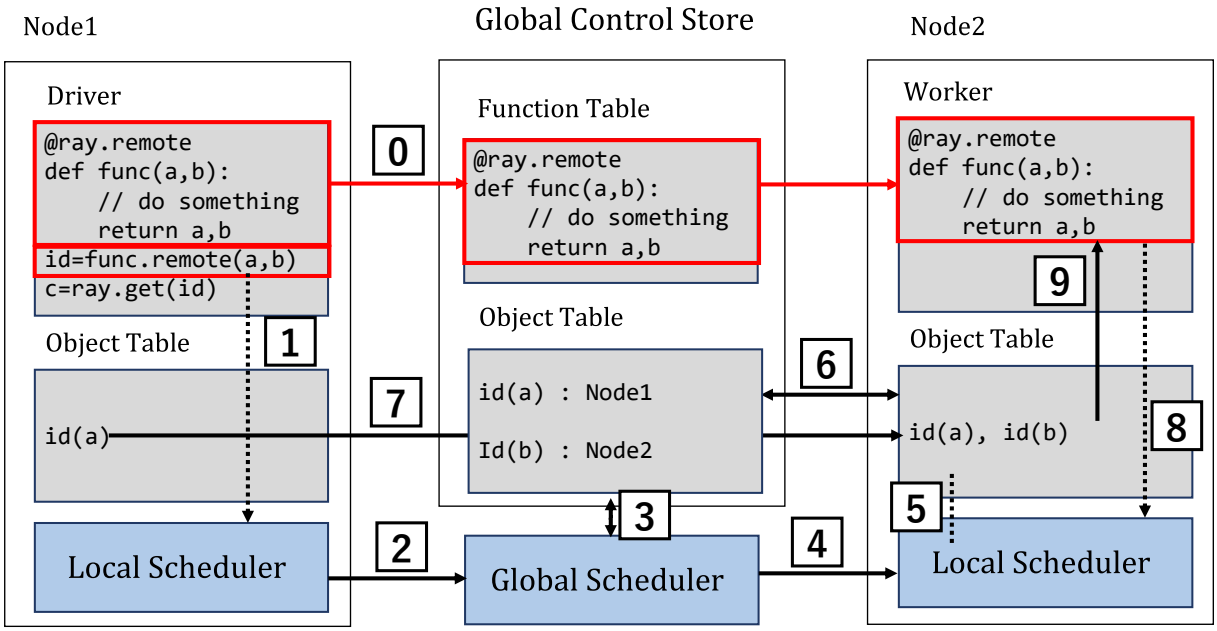

図 2.2 リモート関数実行時のオブジェクトの流れ [16]

オブジェクトストアから引数 a を複製する (ステップ 7)．全てのローカルオブジェクトストア内に引数が保存されたため，ローカルスケジューラはローカルワーカーでリモート関数を起動し (ステップ 8)，共有メモリを介して引数にアクセスする (ステップ 9)．

2.1.4 主要な API

以下に本稿で提案するモデルを実装するために使用した Ray の主要な API について記述する．

- `ray.init()`

  Ray クラスタの起動と起動済みの既存クラスタへの接続を行う．引数無しでこの関数を実行した際は起動済みのクラスタを自動検出し接続する．クラスタが起動していない場合は新しい Ray インスタンスを生成しクラスタを起動する．引数 `address` に `auto` を指定すると起動済みのローカルクラスタへと明示的に接続し，IP アドレスを指定すると既存のリモートクラスタへ接続する．

- `ray.get()`

  リモート関数を実行した際に返される object_ref に対応するオブジェクトをオブジェクトストアから取得する．この関数は，対応するオブジェクトが取得できるまで処理をブロックする．

- `ray.wait()`

  引数として渡された object_ref のリスト内で利用可能な k 個の結果を返す．この関数では `ray.get()` と異なり，オプションで指定した数の結果が得られると，結果と実行中のタスクの 2 つのリストを返す．すなわち，実行時間の異なる複数のタスクを並列に扱うことができる．

- `ray.put()`

  前述したようにリモート関数の引数として与えられたオブジェクトは暗黙の内に GCS へとコピーされるが，この関数を呼び出すことで明示的に GCS に対してコピーすることもできる．`ray.put()` が返す object_ref をタスクやアクターに渡すことで，object_ref に対応したオブジェクトを GCS 経由でリモート関数に与えることができる．

2.2 Experience Replay

本研究の実験で用いた Distributed Prioritized Experience Replay の前提となる Experience Replay と Prioritized Experience Replay について記述する．

経験再生（Experience Replay）[14] は学習を安定させるために，時刻 t において (s_t, a, r, s_{t+1}) で表わされる状態，行動，行動の結果得られる報酬と行動後の状態を Replay Buffer に保存し，ランダムに取り出して学習に用いる．

2.2.1 Prioritized Experience Replay

Experience Replay では，ランダムに経験を Replay Buffer から取り出して学習に用いるため，学習に有用かどうか関係なく再生していた．そこで，より効率的に学習を行うた

めの手法として優先度付き経験再生（Prioritized Experience Replay）[17] が提案されている．この手法では，式 2.1 で定義される遷移の重要度を表す TD 誤差 δ（の絶対値）と経験を Replay Buffer に保存し，式 2.2 で表わされる確率 $P(i)$ に従ってサンプリングする．指数 α は優先度をつける度合いを表し，$\alpha = 0$ で一様なサンプリングとなる．

$$\delta = \alpha\{r_t + \gamma \max Q(s_{t+1}, a_t) - Q(s_t, a_t)\} \tag{2.1}$$

$$P(i) = \frac{(p_i)^\alpha}{\sum_k (p_k)^\alpha} \tag{2.2}$$

非一様なサンプリングでは経験再生により生じる歪みを調整するため式 2.3 で定義される重要度サンプリングを用いる．

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta \tag{2.3}$$

2.3 Distributed Prioritized Experience Replay

Distributed Experience Replay（または Ape-X） [12] は優先度付き経験再生を大規模な深層強化学習に対応させた手法である．この手法は Deep Q-Networks（DQN）と Deep Deterministic Policy Gradient（DDPG） [13] に対応しているが，本稿では DQN に対応させた Ape-X DQN を中心に説明を行う．この手法では深層強化学習における行動と学習を分離し非同期かつ並列に処理することで優先度付き経験再生を大規模な学習に対応させている．行動は環境を観測することで，ニューラルネットワークとして実装された方策を評価し，観測した経験を Replay Buffer に保存することを指す．学習は保存した Replay Buffer から経験のミニバッチを取り出し再生することでニューラルネットワークのパラメータを更新する．

Ape-X のアーキテクチャは図 2.3 で示すように，CPU 上で動作する複数の Actor が経験を生成し，生成した経験と優先度を Replay Buffer に保存する．Learner は Replay Buffer から経験をサンプリングし，ネットワークと保存されている優先度の更新を行う．各 Actor

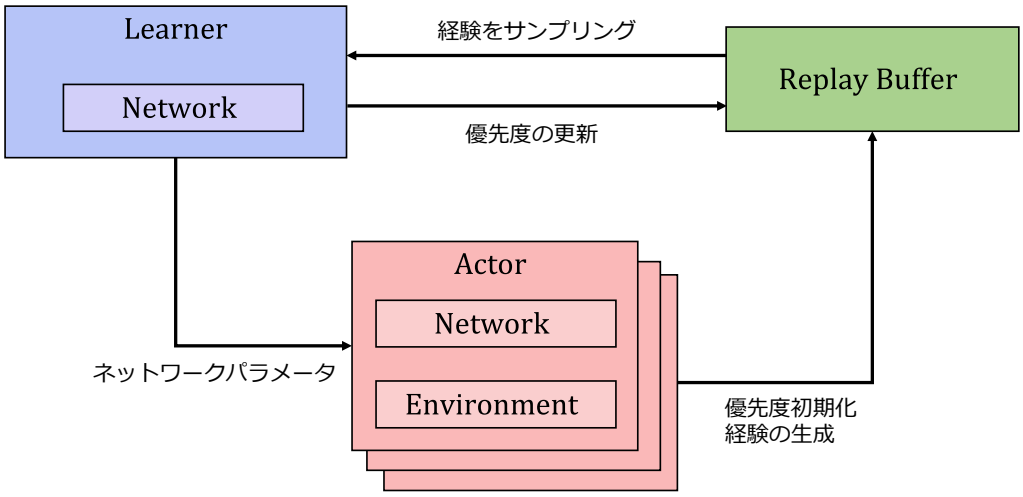

図 2.3 Ape-X アーキテクチャ[12]

が持つネットワークのパラメータは定期的に Learner からパラメータ受け取ることで更新する．

2.4 RLlib

本研究では Ray を用いて分散強化学習を行った際に生成される学習データを保存することを目的としたライブラリの設計と開発を行っているが，Ray には RLlib [7] という強化学習用サブパッケージが開発されている．RLlib には本研究中の実験に用いた Ape-X DQN をはじめとして A3C, AlphaZero, Rainbow など多くのアルゴリズムが実装されており，パラメータを与えるだけでそれらのアルゴリズムを使用できる．RLlib では学習のログや結果，パラメータなどが tensorboard ファイルに保存されているため，RLlib を用いて学習を行うことで追試に必要と考えられる情報を保存することができる．RLlib ではモデルやシミュレーション環境のカスタマイズが可能であるが，アルゴリズムに関しては `https://docs.ray.io/en/latest/rllib/rllib-algorithms.html` で公開されているものを用いる必要があるため，新規のアルゴリズムや既存手法に手を加えたアルゴリズムを用いる場合は自身で実装する必要がある．

第 3 章

提案モデル

本研究で提案するデータ保存モデルの概要と実装について記述する．

3.1 概要

本研究で提案するデータ保存ライブラリは経験と一時保存するプロセスと保存した経験をファイルへ書き込むプロセスで構成される（図 3.1）．以降，これらのプロセスをそれぞれ LocalManager と GlobalManager と呼ぶ．LocalManager はアクター $\{Actor_1, Actor_2, \ldots, Actor_n\}$ が生成する時刻 t における，状態，行動，報酬，行動後の状態，終了判定 (s_t, a, r, s_{t+1}, d) をバッファ $\{buffer_1, buffer_2, \ldots, buffer_n\}$ へと一時的に保存する．GlobalManager は LocalManager のバッファに保存された経験を時系列順にソートして *FILE* に書き込む．データ保存処理を既存のアプリケーションに追加することによる性能の低下を抑制するため，LocalManager と GlobalManager は Ray によってアクター化して非同期に処理を行う．

3.2 実装

GlobalManager は 1 つのみ生成され，LocalManager のインスタンスと ID の一覧を内部に持つ．また，GlobalManager は Ray クラスタ構築時に head ノードに指定した計算機上に配置する．LocalManager はアクターと 1 : 1 に対応するように生成する．

Ray アクターのリモート関数の呼び出しには**関数名**`.remote()` と記述する必要がある．これに対し，ユーザの利便性向上のためのリモート関数呼び出し用ラッパークラス

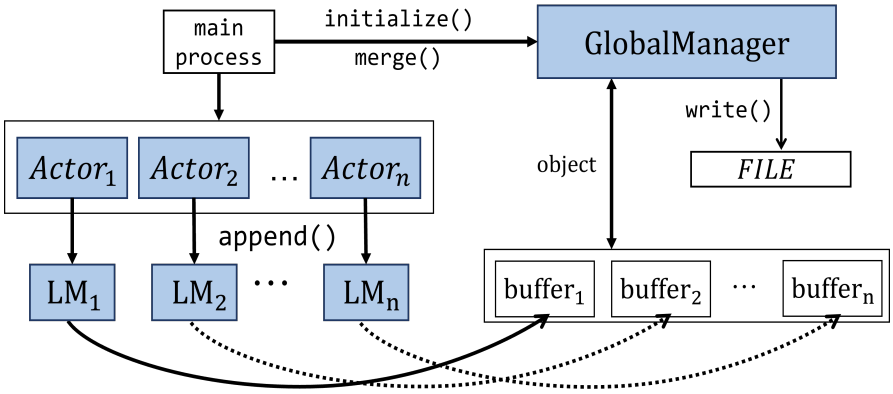

図 3.1 モデル概要

`LocalWrapper`, `GlobalWrapper` を実装する．LocalManager と GlobalManager の起動，バッファへの学習データの追加，*FILE* への書き込みおよび *FILE* からの読み込みは以下の関数を用いて行う．

- `initialize`

  GlobalManager の起動と初期化を行う．引数には後述する `merge()` で学習データを書き込むファイル名を渡し，GlobalManager 内のリモート関数を実行するための `GlobalWrapper` を返す．また，Ray インスタンスが起動していない場合は `ray.init()` を用いて起動と初期化を行う．Ray をローカルモードで使用する場合は GlobalManager をこの関数を実行した計算機上に配置する．Ray クラスタを使用する場合は，起動時に head ノードに指定した計算機上で起動するように制御する．

- `activate_manager`

  LocalManager の起動・初期化を行い，`LocalWrapper` インスタンスを返す．そして，LocalManager 内のバッファから学習データを取得できるように GlobalManager に対してオブジェクト ID とインスタンスを渡す．

- `append`

  各アクターが生成する経験を LocalManager が持つバッファに対して追加する．この関数が呼び出されると `LocalWrapper` 内で圧縮とシリアライズし，`ray.put()` で GCS に対してコピーを行う．`ray.put()` の結果得られる object_ref を 引数として

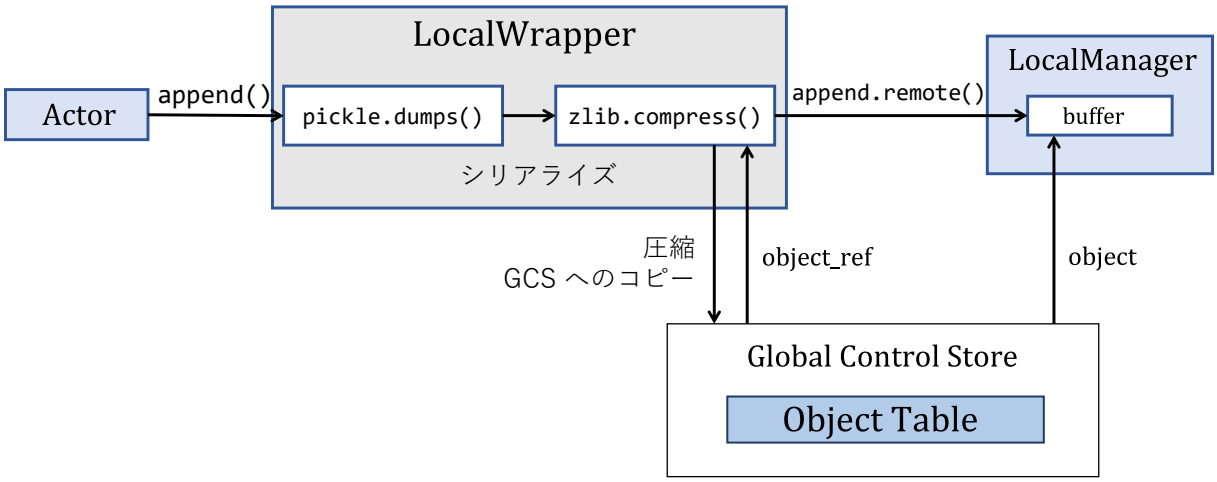

図 3.2　`append()` の処理の流れ

LocalManager に与える (図 3.2)．LocalManager 内では object_ref オブジェクトと紐づけられた経験が GCS から読み出される．データを追加する際は datetime でタイムスタンプを取得してから追加する．

- `merge`

  この関数を呼び出すと GlobalManager が LocalManager のリモート関数 `pull.remote()` を実行してバッファから経験を取得する (図 3.3)．取得した経験をタイムスタンプでソートしてから GlobalManager 初期化時に指定したファイルへ書き込む．LocalManager から経験を取得する際は経験を重複して書き込むことを防ぐために排他制御を行う．

- `finalize`

  `merge()` を行う際に生成されたロックファイルなど実行中に生成された中間ファイルの削除，処理中のタスクの完了を待つ．

- `pull`

  FILE から末尾数件の経験のリストを取得する．本研究では 100 を規定値とする．

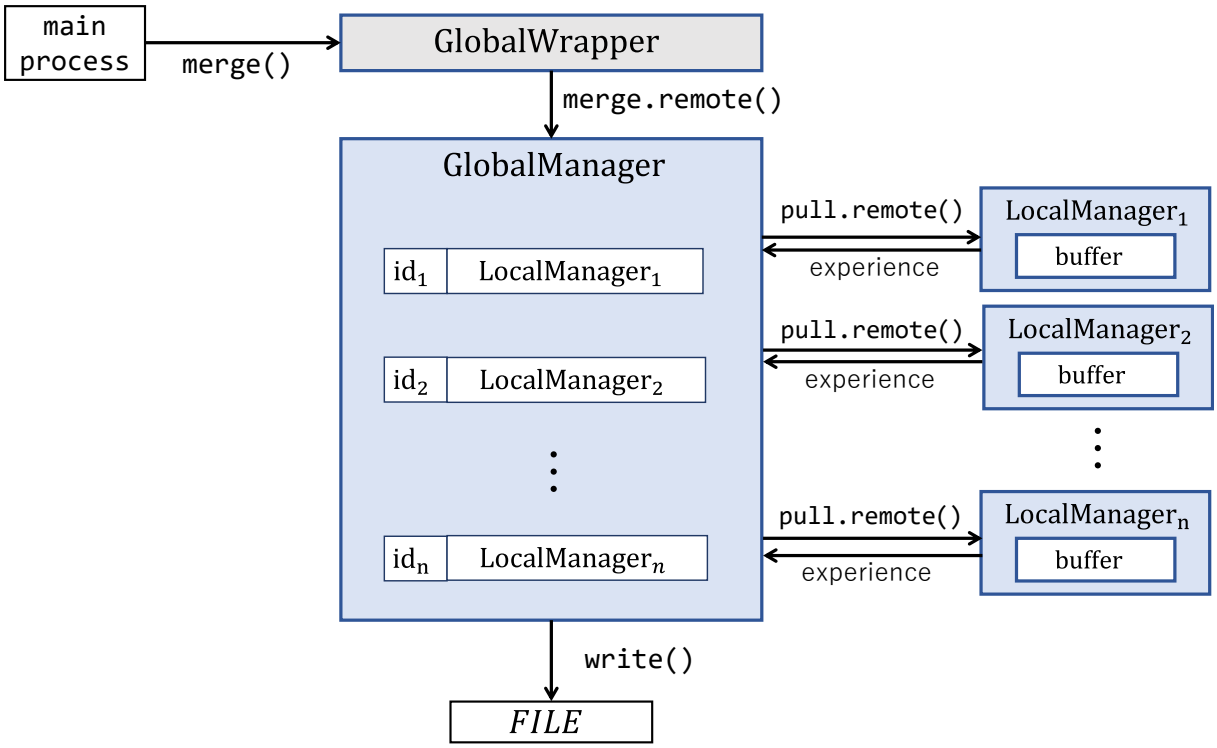

図 3.3　`merge()` の処理の流れ

3.3　保存データの圧縮

学習データは膨大になるため，適切に圧縮して保存する必要がある．この圧縮処理がボトルネックになることを避けるため，`LocalWrapper` 内で経験ごとに圧縮する．

具体的には，学習データを GCS に対してコピーする前に `pickle.dumps()` によるシリアライズと `zlib.compress()` による圧縮を行う．`pickle.dumps()` のプロトコルと `zlib.compress()` の圧縮レベルはそれぞれ Python 3.7.7 における規定値を用いた．

第 4 章

実験

データ保存モデルの性能を検証するために行った実験の環境と手順，結果について記述する．

4.1 シミュレーション環境

本実験では OpenAI Gym が提供するシミュレーション環境である CartPole-v1 [2] と BreakoutDeterministic-v4 [1] を使用する．以下に各環境の説明を記述する．

4.1.1 CartPole-v1

CartPole は図 4.1 のように摩擦のない平面上を移動する台車の上に棒が直立に取り付けられており，台車を左右に動かすことによって棒が倒れないようにバランスを取ることを目標としたシミュレーション環境ある．この環境における行動空間は $\{0, 1\}$ の値であり，表 4.1 に示す処理を行う．状態は台車の位置，速度，棒の角度，角速度である．

表 4.1 CartPole-v1 の行動空間

Num	Action
0	台車を左へ押す
1	台車を右へ押す

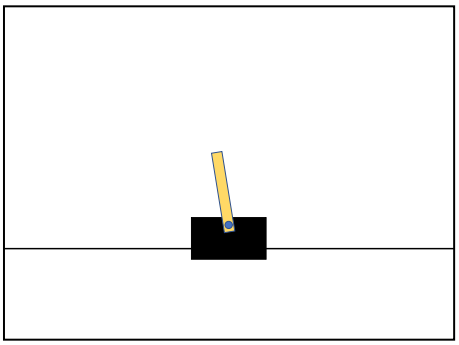

図 4.1　CartPole のシミュレーション画面 [2]

4.1.2　BreakoutDeterministic-v4

Breakout は Atari 環境の一部であり，図 4.2 に示すように画面下部にあるパネルを操作してボールを画面上部にあるブロックに当て，全てのブロックを消すことを目標としたゲームである．この環境の行動空間は $\{0, 1, 2, 3\}$ の値であり，表 4.2 に示す処理を行う．この環境で観測される状態は縦 210 ピクセル，横 160 ピクセルの RGB 画像であり，ブロックを消すと報酬が得られる．得られる報酬は消したブロックの色によって変化する．

4.2　実験内容

本研究で提案するデータ保存モデルの性能を評価するために CartPole 環境と Breakout 環境のもとで学習を行う Ape-X DQN に対してデータ保存処理を追加した際の実行時間を計測する．本実験は表 4.3 で示す計算機を 2 台用いて構築した Ray クラスタ上で学習を 10 回行い，実行時間の中央値を測定する．Ray クラスタは `ray start --head` と `ray start`

表 4.2　BreakoutDeterministic-v4 の行動空間

Num	Action
0	何もしない
1	ボールを射出する
2	パネルを右に動かす
3	パネルを左に動かす

`--address="ヘッドノードの IP アドレス"` でそれぞれヘッドノードとワーカーノードを起動して構築する．

実験は各環境について学習終了時に経験を保存する方法（一括出力と呼ぶ）とネットワーク更新時に保存する方法（都度出力と呼ぶ）の 2 通り行い実行時間を比較する．学習データの生成回数は 100 ステップを 1 ロールアウトとして，ロールアウトを 2000，4000，6000，8000，10000 回とする経験の生成を行うアクターの数は CartPole 環境では 2，4，6，8，10，20，Breakout 環境では 20 で行い，Replay Buffer のサイズは 2^{14} とする．

実装したライブラリでは pickle [6] で Python オブジェクトをシリアライズしファイルへ書き込むことで学習データを保存している．これらの処理によるオーバーヘッドを検証するために，pickle で逐次的に経験を保存した際の実行時間を調べる．

また，既存ライブラリとの性能を比較するために，データ保存機能を持った Replay Buffer を実装したライブラリである cpprb [3] を用いる．cpprb を用いたデータ保存には，このライブラリが提供する Replay Buffer が持つ関数 `save_transitions()` を使用する．

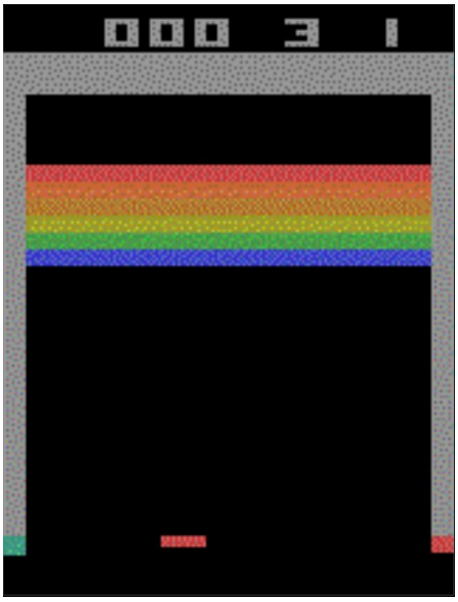

図 4.2　Breakout のプレイ画面 [1]

表 4.3　計算機の構成

名称	バージョン情報
CPU	AMD Ryzen 7 3800X 8-Core Processor
メモリ	128GB
GPU	GeForce RTX 2060 SUPER
OS	Ubuntu 20.04.2 LTS
Python	3.7.7
Ray	1.13.0

4.3　実験結果

4.3.1　CartPole-v1

表 4.4 はアクター数を 2 ～ 20 としたときの CartPole 環境における実行時間である．学習終了時に経験を保存する一括出力の性能低下率は約 4% で，アクター数の変化による差は見られなかった．一方，ネットワーク更新時に経験を保存する都度出力では 5.3 ～ 16.8% 実行速度が低下しており，アクター数 20 において最も性能低下率が大きくなるという結果が得られた．

また，pickle を用いた場合ではアクター数を増やすと性能が低下しており，アクター数 20 において最大で 11.7% 実行速度が低下することを確認した．

表 4.4 CartPole 環境における実行時間 [sec]

アクター数	rollout	データ保存なし	一括出力	都度出力	pickle
2	2000	63.27	66.06	67.38	66.76
	4000	124.72	131.61	131.05	131.63
	6000	187.06	194.84	196.50	188.72
	8000	246.00	258.98	259.61	259.32
	10000	313.96	320.09	327.25	321.79
4	2000	38.08	38.35	39.73	38.26
	4000	73.30	76.34	78.37	71.70
	6000	110.78	115.21	115.34	112.99
	8000	142.68	115.21	115.34	148.61
	10000	180.91	188.59	195.26	182.57
6	2000	28.73	30.27	30.86	29.36
	4000	54.52	56.13	58.56	55.63
	6000	82.36	86.57	88.02	83.08
	8000	106.79	111.33	118.68	113.15
	10000	135.24	140.94	143.62	135.83
8	2000	24.53	24.85	26.02	24.05
	4000	45.13	47.25	48.57	46.95
	6000	65.85	70.14	71.81	69.42
	8000	88.91	90.58	94.42	92.19
	10000	111.70	113.50	120.70	111.52
10	2000	20.70	21.81	23.00	21.72
	4000	40.19	40.11	42.53	40.88
	6000	57.96	60.10	63.18	60.86
	8000	77.53	79.45	83.16	81.91
	10000	99.95	99.73	103.61	98.00
20	2000	15.21	15.71	16.92	15.50
	4000	25.99	27.21	30.68	27.48
	6000	38.36	40.10	44.60	41.30
	8000	48.18	51.76	59.12	53.82
	10000	61.98	63.16	71.81	67.31

4.3.2 CartPole-v1 (cpprb)

表 4.5 は cpprb が提供する Replay Buffer を用いた学習に対して，同ライブラリが提供するデータ保存機能と提案手法でデータを保存した際の実行時間である．cpprb が提供する機能でデータを保存すると，都度出力では約 10%性能が低下していたが一括出力では処理を追加したことによる性能の低下は確認できなかった．提案手法によりデータを保存すると，一括出力では約 3%，都度出力では 3 ～ 14% の性能低下が確認できた．アクター数 20 において cpprb と提案手法の両方で性能低下率が最も性能低下率が大きくなるという結果が得られた．

4.3.3 BreakoutDeterministic-v4

表 4.6 はロールアウトの回数を 2000 ～ 10000 としたときの Breakout 環境での実行時間である．提案手法 (一括出力) では 16.5 ～ 21.7%，提案手法 (都度出力) では 15.8 ～ 19.6% の性能低下となっていた．ロールアウトの回数とデータを保存するタイミングの違いによる性能低下率に差は確認できなかった．

また，pickle を用いたデータ保存では性能の低下は確認できなかった．この結果から，pickle による Python オブジェクトのシリアライズと，シリアライズしたオブジェクトのバイナリファイルへの書き込みによる性能への影響が少ないということがわかる．

4.3.4 BreakoutDeterministic-v4 (cpprb)

表 4.7 は Breakout 環境で cpprb が提供する Replay Buffer を用いた学習に対して，同ライブラリが提供するデータ保存機能と提案手法でデータを保存した際の実行時間である．一括出力ではロールアウトの回数を 2000, 4000, 10000 としたときに cpprb を用いたデータ保存が提案手法より効率的であった．都度出力では cpprb の機能を用いたデータ保存は大幅に性能が低下した．提案手法ではロールアウトの回数が 2000, 4000, 8000 の時に 37 ～ 60% 性能が低下していたが，cpprb を用いた場合と比較すると性能低下率は抑えられていた．

表 4.5　CartPole 環境における cpprb を用いた実行時間 [sec]

アクター数	rollout	データ保存なし	cpprb（一括出力）	cpprb（都度出力）	提案手法（一括出力）	提案手法（都度出力）
2	2000	53.16	53.12	54.88	55.70	55.28
	4000	102.66	103.05	106.49	106.83	106.11
	6000	150.56	151.57	157.68	157.01	156.91
	8000	200.62	200.58	206.06	207.06	208.76
	10000	248.88	249.25	255.27	256.82	260.08
4	2000	31.15	31.17	32.60	32.54	32.38
	4000	57.00	57.06	60.39	60.33	60.39
	6000	83.07	83.27	88.78	88.12	88.90
	8000	108.79	109.80	116.82	115.70	118.68
	10000	136.50	135.83	144.81	143.78	145.30
6	2000	23.37	23.30	25.22	24.94	24.94
	4000	42.82	41.90	44.57	44.69	45.77
	6000	60.28	60.83	66.52	65.05	65.15
	8000	79.61	80.80	89.49	84.12	85.98
	10000	99.57	97.94	105.22	104.59	106.23
8	2000	20.47	20.33	21.59	21.18	21.90
	4000	36.43	37.33	38.55	37.43	38.27
	6000	52.22	51.57	56.92	53.85	56.38
	8000	67.55	67.09	72.16	70.52	73.35
	10000	84.80	82.81	90.83	87.92	90.03
10	2000	18.35	18.55	19.96	18.96	19.76
	4000	32.10	32.38	35.69	33.19	34.65
	6000	46.05	46.16	50.87	47.45	49.82
	8000	59.92	59.83	65.80	61.60	64.45
	10000	74.40	73.64	81.61	76.24	79.81
20	2000	15.04	14.93	16.28	15.38	16.99
	4000	26.03	25.13	28.10	25.74	29.56
	6000	35.86	35.26	41.05	36.03	41.37
	8000	46.35	45.87	51.79	46.63	53.49
	10000	56.63	55.46	61.64	57.62	64.81

表 4.6 Breakout 環境における実行時間 [sec]

アクター数	rollout	データ保存なし	一括出力	都度出力	pickle
20	2000	92.46	112.53	110.11	91.16
	4000	185.03	215.58	214.24	178.15
	6000	268.95	321.77	321.13	265.92
	8000	357.00	427.45	421.66	358.06
	10000	444.50	530.07	531.69	438.17

表 4.7 Breakout 環境における cpprb を用いた実行時間 [sec]

アクター数	rollout	データ保存なし	cpprb（一括出力）	cpprb（都度出力）	提案手法（一括出力）	提案手法（都度出力）
20	2000	512.78	602.49	1284.64	715.91	704.22
	4000	745.58	714.37	2463.63	1466.09	1020.43
	6000	1995.48	927.24	4218.83	1107.03	1665.02
	8000	1402.95	2952.28	5199.22	2125.71	2241.80
	10000	3568.27	1739.95	6870.91	2313.01	3682.60

4.4 LocalManager のノード配置による影響

上記で行った実験は GlobalManager のみヘッドノードに配置するように制御しているが，その他のアクターは Ray のスケジューラにより配置ノードを決定している．そのため，経験を生成するアクターと LocalManager が同一ノード上に存在せずノード間通信が発生する場合がある．そこで，アクターと LocalManager を同一ノード上に配置するようにプログラムを変更し実行時間を調べることで，Ray のスケジューリングによるオーバーヘッドを検証する．

LocalManager とアクターを同一ノードへの配置は，Ray の API を用いてプロセス ID から動作しているノードを特定し，`ray.remote()` のオプションに与えることで行っている．

表 4.8 はアクター数 2 ～ 20 で CartPole 環境のもとで学習を行った際の実行時間である．一括出力，都度出力ともに LocalManager の配置するノードを制御しない場合と比較して差は確認できなかった．

表 4.8 アクターのノード配置を変えた場合の実行時間の比較 [sec]

アクター数	rollout	一括出力（制御なし）	都度出力（制御なし）	一括出力（制御あり）	都度出力（制御あり）
2	2000	66.06	67.38	64.24	67.69
	4000	131.61	131.05	130.55	129.46
	6000	194.84	196.50	194.21	196.21
	8000	258.98	259.61	257.74	260.13
	10000	320.09	327.25	316.78	322.93
4	2000	38.35	39.73	39.99	39.75
	4000	76.34	78.37	77.37	78.00
	6000	115.21	115.34	117.06	116.41
	8000	115.21	115.34	155.10	155.32
	10000	188.59	195.26	191.02	189.59
6	2000	30.27	30.86	29.40	30.26
	4000	56.13	58.56	57.51	59.56
	6000	86.57	88.02	84.40	85.83
	8000	111.33	118.68	112.15	116.94
	10000	140.94	143.62	141.29	144.49
8	2000	24.85	26.02	24.48	25.41
	4000	47.25	48.57	46.19	49.05
	6000	70.14	71.81	70.31	72.24
	8000	90.58	94.42	91.50	96.31
	10000	113.50	120.70	115.27	120.03
10	2000	21.81	23.00	21.82	22.44
	4000	40.11	42.53	40.50	42.33
	6000	60.10	63.18	59.00	64.59
	8000	79.45	83.16	78.05	81.59
	10000	99.73	103.61	99.20	105.63
20	2000	15.71	16.92	15.31	17.00
	4000	27.21	30.68	26.80	30.09
	6000	40.10	44.60	38.74	44.53
	8000	51.76	59.12	53.20	57.22
	10000	63.16	71.81	63.19	72.97

第 5 章

考察

実験結果から，CartPole 環境の一括出力以外のテストケースで約 20% 実行速度が低下することが分かった．LocalManager のノード配置による影響を調べた実験の結果から，Ray のスケジューリング機能のオーバーヘッドによる実行速度への影響は小さいと考えられる．以上のことを踏まえて，CartPole 環境と Breakout 環境での性能低下の要因と既存ライブラリとの比較について考察した内容を述べる．

5.1 CartPole 環境

CartPole 環境では，ネットワーク更新時に経験を保存する都度出力のアクター数 20 において 11.2 ～ 22.7% 性能が低下するという結果が得られた．一括出力と都度出力で異なる点は `merge()` の呼び出し回数のみであるため，性能低下の要因と考えられる．`merge()` では GlobalManager が LocalManager から経験を取得し，バイナリファイルへ書き込む処理を行っている．経験を受け取る処理は GlobalManager が LocalManager とメッセージを送受信することで行っている。ここで，Breakout 環境の都度出力ではロールアウト 2000 回につき平均 10 回 `merge()` を呼び出していたのに対して，CartPole 環境では平均で 73 回呼び出していた．この結果から，CartPole 環境における経験の生成とネットワーク更新の計算コストは Braekout 環境と比較して小さいと考えられる．そのため，LocalManager から経験を受け取る処理のオーバーヘッドが経験の生成やネットワーク更新のコストと比較して大きくなり実行速度が低下したと考えられる．LocalManager と経験を生成するアクターは 1 : 1 に対応しているため，アクターが多くなるほどメッセージの送受信によるオーバーヘッ

ドが大きくなりボトルネックになったと考えられる．

また，表 4.4 に示した結果から pickle によるシリアライズとバイナリファイルへの書き込み処理も性能低下の要因であると考えられるため検証が必要がある．

5.2 Breakout 環境

Breakout 環境では，一括出力，都度出力ともに 20% 程の性能低下を確認した．一括出力と都度出力で実行時間に差がなかったことから，LocalManager 内のメモリに経験を書き込む `append()` がボトルネックになっていると考えられる．`append()` は LocalManager が提供するリモート関数であり，保存する経験を引数として実行する．Ray のリモート関数が実行されると，Global Control Store (GCS) に対して引数として入力したオブジェクトのコピーが発生する．引数として入力されるオブジェクトのデータサイズは，CartPole 環境では平均で約 2446 バイト，Breakout 環境では平均で約 216160 バイトであった．そのため，状態空間のデータサイズが大きい Breakout 環境では GCS へのコピーがボトルネックとなり実行速度が低下したと考えられる．また，一括出力と都度出力で `merge()` の呼び出し回数が異なっているにも関わらず実行時間に差が見られなかったのは，Breakout 環境では経験の生成とネットワーク更新のオーバーヘッドが大きくなるため `merge()` 処理のオーバーヘッドが無視できる大きさになったためと考えられる．

5.3 既存ライブラリ (cpprb) との比較

cpprb を用いた経験の保存では，Breakout 環境の都度出力で著しく性能が低下した．提案手法では，ある時刻で保存されたデータとの差分のみを保存する．これに対し cpprb の `save_transitions()` では，Replay Buffer 内の全データを保存するため，1 度に処理するデータ量が提案手法と比較して大きくなり実行速度の低下を招いたと考えられる．また，cpprb で 1 度に保存するデータ数は本実験の Replay Buffer のサイズに設定した 2^{14} 個が上限であるため，Breakout 環境における一括出力で提案手法より優れた結果を示したと考

えられる．しかし，cpprb を用いた Breakout 環境でのデータ保存の実行時間は振れ幅が大きかったため実行回数を増やし再度検証を行う必要がある．

5.4　ボトルネックの解消

実験結果と上記の考察から LocalManager のメモリ経験を書き込む `append()` 実行時に発生する GCS へのコピーと GlobalManager が LocalManager のメモリから経験を取得する `merge()` がボトルネックになっていると考えられる．

`append()` によるボトルネックを解消するためには，`LocalWrapper` 内で経験を圧縮しつつファイルへ書き出し，ファイル名を `append.remote()` の引数として LocalManager へ与えることで GCS へコピーするオブジェクトを最小限するなどの方法が考えられる．ファイルへの書き込みには圧縮率が高く処理速度に優れた `numpy.savez_compressed` [4] が候補として考えられる．

また，`merge()` による性能低下を解消するためには `append()` 実行時に書き込んだファイル名を GlobalManager に共有することで，ファイルから経験を直接読み出せるようにする方法が考えられる．しかし，複数ノードで学習を行う場合は GlobalManager の動作しているノード上にファイルが存在せず読み込みができない可能性があるため，その場合は従来通り LocalManager を介して経験を取得する必要がある．

第 6 章

結論

本研究では，分散強化学習でアクターが非同期に生成する学習データを Ray を用いて保存するライブラリの設計と開発を行った．生成される経験を一時的に保存するプロセスと保存された経験を読み出してファイルへ書き込むプロセスを Ray アクターによりリモート化し非同期に並列処理することで，既存のアプリケーションにデータ保存処理を追加した場合の性能低下を抑えることを目指した．

提案手法を用いて学習データを保存した結果．CartPole 環境では学習終了時に保存する場合に最も効率よく学習データを保存でき，ネットワーク更新時に保存する場合は約 20% のオーバーヘッドがあるという結果が得られた．Breakout 環境ではデータ保存のタイミングに関係なく 20% 程性能が低下することを確認した．また，他フレームワークを利用した場合との比較実験から，オブジェクトのシリアライズとバイナリファイルへの書き込み，Ray のスケジューリングによるオーバーヘッドは，提案手法全体におけるオーバーヘッドの中で比較的小さいということが分かった．

実行速度の低下の要因として LocalManager のメモリに経験を書き込む `append()` 実行時に発生する GCS へのオブジェクトのコピーと GlobalManager が LocalManager のメモリから経験を取得する処理がボトルネックとになったと考えられる．上記のボトルネックを解消するために，`append()` でリモート関数に渡すオブジェクトのデータサイズを小さくするか，GCS へのコピーが発生しない処理へ変更するなどの改良が必要である．また，本研究では経験の書き込みに関する API は実装しているが，ファイルから読み込む API は最低限のものしかないため追加で実装する必要がある．

謝辞

本研究を進めるにあたり多くの指導をしてくださった松崎公紀教授に御礼申し上げます．研究の方向性が定まらず研究が中々進まなかった際に，松崎先生が方向性やその手法に関する相談に乗ってくださったことにより最後まで本研究に取り組むことができました．大変お世話になりました．また，副査を引き受けてくださった岩田誠教授と横山和俊教授にも心より感謝申し上げます．セミナーなど報告会の場で，研究を進めるにあたり貴重なご意見を頂き大変お世話になりました．

参考文献

[1] "Breakout," https://www.gymlibrary.dev/environments/atari/breakout/#breakout.

[2] "CartPole," https://www.gymlibrary.dev/environments/classic_control/cart_pole/.

[3] "cpprb," https://ymd_h.gitlab.io/cpprb/.

[4] "numpy.savez_compressed — NumPy v1.24 Manual,"https://numpy.org/doc/stable/reference/generated/numpy.savez_compressed.html.

[5] "OpenAI Gym," https://github.com/openai/gym.

[6] "pickle — Python オブジェクトの直列化 — Python 3.7.10," https://docs.python.org/ja/3.7/library/pickle.html.

[7] "RLlib," https://docs.ray.io/en/latest/rllib/index.html.

[8] P. Th. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, Vol. 35, No. 2, pp. 114–131 (2003).

[9] Y. Fujita, K. Uenishi, A. Ummadisingu, P. Nagarajan, S. Masuda, M. Y.Castro, "Distributed reinforcement learning of targeted grasping with active vision for mobile manipulators," *International Conference on Intelligent Robots and Systems 2020* (2020).

[10] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *Association for the Advancement of Artificial Intelligence* (2018).

[11] C. Hewitt, P. Bishop, R. Steiger, "A universal modular actor formalism for artificial intelligence," *the 3rd international joint conference on Artificial intelligence*,

pp. 235–245 (1973).

[12] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, D. Silver, "Distributed prioritized experience replay," *International Conference on Learning Representations* (2018).

[13] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, "Continuous control with deep reinforcement learning," *International Conference on Learning Representations* (2016).

[14] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, Vol. 8, pp. 293–321 (1992).

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning 2016* (2016).

[16] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I.Jordan, I. Stoica, "Ray: A distributed framework for emerging AI applications," *13th USENIX Symposium on Operating Systems Design and Implementation*, pp. 561–577 (2018).

[17] T. Schaul, J. Quan, I. Antonoglou, D. Silver, "Prioritized experience replay," *International Conference on Learning Representations* (2016).

[18] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction,"The MIT Press (1998), 三上 貞芳, 皆川 雅章 共訳, "強化学習," 森北出版 (2000).

[19] 竹野 創平, 渡部 卓雄, "アクターモデルに基づく並行文脈指向プログラミング機構の実装と評価," コンピュータ ソフトウェア, 33 巻, 1 号, pp. 167–180 (2016).