CB-005

自律型並列分散処理システム AgentSphere における Tensorflow を用いた機械学習モジュールの実装と評価

Implementation and Evaluation of a Tensorflow based Machine Learning Module in Autonomous Parallel/Distributed Processing System AgentSphere

ディダ ベサリ † 甲斐 宗徳 †

Besar Dida Munenori Kai

1. はじめに

AgentSphere とは並列分散処理を行うためのプラットフォームである。近年のニーズおいてこれらの処理に対する需要が向上しており、なおかつ、使用できるマシンリソースもより手軽に入手できることから専門知識を持たない人も並列処理や分散処理を身近に利用できるようにするために開発中のものである。

AgentSphere はマシン上に展開する Java のバーチャルマシン（VM）であり、これをベースのプラットフォームとして自律的にふるまうことが可能である並列分散処理を主な目的としているモバイルエージェントが展開される。これらのエージェントの内部の処理はユーザが任意にプログラムすることが可能である。Java で実装されたことにより OS の依存性がなくなり、ユーザのプログラミングによってエージェントは一定の自律性（どのマシンで処理してもらうかを判断し、そのマシンへ移動するなどの自律的動作）を持つ。Java のコード上でエージェントとなるコードを記述するために必要な知識は一般的な Java 知識のみであり専門知識を持たないユーザも比較的に手軽に実装することができる。

AgentSphere のモビリティは強マイグレーション技術をベースとしている。強マイグレーションにより各マシンに展開されているエージェントは自らのヒープ及びスタック領域、そしてプログラムカウンタをそのままネットワーク上の別のマシンへ移動できる。移動した後、再び自身の実行状態を移動する前の状態に戻し、処理を再開できる。この方法では Java のオブジェクトの現在の実行状態を保存する API である JavaFlow を使用し、そのデータをネットワーク上に移動できる形にシリアライズしていた。これには Java の既存のシリアライザが今まで使用されていたが、近年の研究によって Google の GSON シリアライザを実装しそれを独自に記述することでオーバヘッドの低下が最近の研究で可能になった。エージェントの移動性能が向上したが、並列分散処理を行うときの移動先の選択を適切に行うなどの自律性はまだ完全であるとは言えない。

そのため、近年で急速に注目を集めている機械学習についての機構を本研究で試験的に導入することにした。上述のとおり、AgentSphere で展開されるエージェントは事実としてコード上の動作しかできないオートマトン程度である。これではユーザが記述できるエージェントの AgentSphere のモビリティを利用してエージェントにより高度な自律性を与えるために AgentSphere のための Tensorflow を用いた機械学習機能の実装を行った。

2. AgentSphere とは

AgentSphere は、本ソフトウェア研究室で開発が行われている自律的にふるまうエージェント達を、ネットワークで接続された AgentSphere と呼ばれる環境を通じて移動させ、並列分散処理を行うプラットフォームである。

AgentSphere とはマシンに展開されている Java 言語で記述されたいわばシェルであり、JavaVM を形成し、その実行環境の中でエージェントを生成・処理する。エージェント達はこの実行環境の中で自身の実行状態とデータを保持しながらネットワーク上に存在する別の AgentSphere に移動することが可能である、この技術をマイグレーション技術と呼ぶ。計算や処理を行う単位がエージェントであるため、これらのエージェントがネットワーク上に存在する限り、いくつかの AgentSphere が使用不能になったとしても、最低でも一つの AgentSphere が残っていれば、プラットフォームとして処理サービスを続行することができ、全体的に高い信頼性と耐故障性を持つ。

プラットフォームとして高い高速性と信頼性を持っていることから、本研究では継続した実行環境の維持が重要である機械学習機構の実装を行うこととした。

2.1 AgentSphere の現状

AgentSphere はモバイルなエージェントを保持し、実行するためのシェルではあるが、これはいくつかのモジュールを内包して実装されている。この論文で実装する機械学習機構（モジュール）のための補足として最近の AgentSphere の変化点を記述する。ほかのモジュールは図 2.1.1 の通りだが、ここではマイグレーションを行うネットワークモジュールと階層型クラスローディングモジュール（ローダ）を抜粋する。

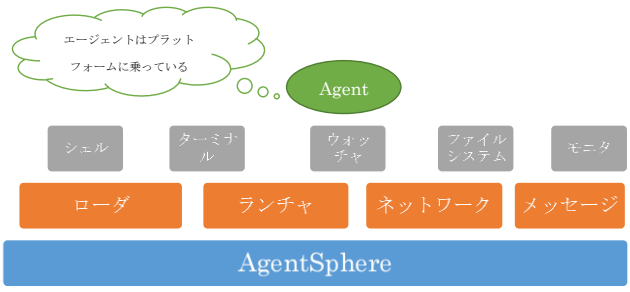

図 2.1.1 AgentSphere のモジュール

† 成蹊大学理工学研究科理工学専攻 Graduate School of Science and Technology, Seikei University

AgentSphere が使用しているマイグレーション技術を可能とするのが階層型クラスローディングおよびネットワークモジュールである。

階層型クラスローディングモジュール[1]では一つの AgentSphere からマイグレーションの命令受けて他の AgentSphere に移動したエージェントは未知クラス、すなわちクラスパス上に存在しないクラスとなるので、これらの未知クラスに対応するために専用のクラスローダを実装している。実行中のエージェントの実行状態を保存し別マシンに移動させてもこのエージェントが未知クラスの場合には Java 標準のクラスローダでは例外が発生する。AgentSphere の階層型クラスローダは、移動してきた未知エージェントの処理を再開させる時に、そのエージェントが使用しているクラスを順にロードしていきそこでの JVM 内のインスタンスとする。

次に、重要なモジュールがネットワークモジュールである。このモジュールは一つの AgentSphere のネットワークに関するマイグレーションやメッセージングをすべて総括する。これがなくては、すべてのモジュールとエージェントのために独自のネットワーク機構をその都度生成しなくてはならないので余計なオーバヘッドとなる。ネットワーク上のすべてのエージェントは DHT[2]でお互いの位置を把握している。

ネットワークモジュールでは、Google の Gson を用いている。Google-Gson[3]とは、Java オブジェクトを JSON 形式にシリアライズ・デシリアライズするために使用されるものである。例として、事前に Car や Person などのクラスを定義していたならば、コード 2.1.1 のように変換される。

```
Gson gson = new Gson();
Car audi = new Car("Audi", "A4", 1.8, false);
Car skoda = new Car("Škoda", "Octavia", 2.0, true);
Car[] cars = {audi, skoda};
Person johnDoe = new Person("John", "Doe", 245987453, 35,
cars);
System.out.println(gson.toJSON(johnDoe));
```

```
   "name": "John",
   "surname": "Doe",
   "cars": [
       {
           "manufacturer": "Audi",
           "model": "A4",
           "capacity": 1.8,
           "accident": false
       },
       {
           "manufacturer": "Škoda",
           "model": "Octavia",
           "capacity": 2,
           "accident": true
       }
```

コード 2.1.1　Gson の変換例

Gson の主な特徴として独自のシリアライザ・デシリアライザを記述することで、ソースコードのないクラスのインスタンスもシリアライズ・デシリアライズ可能となる。

インスタンスクリエイタを記述することで本来シリアライズ可能ではなかったクラスおよび未知のクラスを復元できる。AgentSphere では独自のインスタンスクリエイタおよびデシリアライザを記述することで階層型クラスローダの内部で行われていた未知クラスのパス取得およびインスタンスのキャスティング部分を省略することで高速化と安定性が増している。AgentSphere では Gson を用いた JSON 出力をベースに、Java のインスタンスオブジェクトに適用させることで、本来の Java のバイトコード変換ではなくより柔軟に使用できる JSON 形式のマイグレーションを実装してある。コード内に存在するそれぞれのメソッドやメンバはすべて JSON 形式でその型、中身、値、引数などがすべて渡される。

2.2 JSON を用いたマイグレーション

Gson ではエージェントをシリアライズ・デシリアライズし、専用の階層型クラスローダへ組み込む手順は過去の研究で使用された既存のシリアライズと大まかな流れは同じだが、シリアライズとデシリアライズに関するところだけが変更された。流れとしては、新たな Gson のためのインスタンスを作成し、現在のエージェントとしての記述をインプリメントした型である AbstractAgent が何なのかを記述させるために、すなわち Json としてどのように出力させるのかを定めるために Type トークンとして AbstractAgent を与える。

これにより Gson はエージェントを Gson.toJson でそれを実際に Json 形式に変換させる。本来ここで AgentSphere のモジュールの一つであるネットワークモジュールを用いたメッセージパッシング機能で Network Sender に直列化されたバイトデータが対象のマイグレーション先に運ばれるのだが、代わりに Json 形式のデータが運ばれる。

InputStream から Json 形式を受け取るのだが、この時点ではまだ Java のバイトデータと同等に扱う、つまり、マイグレーションが終了した後デシリアライズが発生し、階層型クラスローダによってクラスパス外のクラスとして扱われる。続くステップでは Gson.toJson の逆である Gson.fromJson を用いて復元し新たな実行可能なインスタンスを JVM 内で作成し、実行させる。しかし、最後ではインスタンスを作るときにマイグレーションしてきた Json データをどのように処理すればよいのかがわからずインスタンスの作成がうまくいかない。そのため、Gson はユーザに独自のインスタンスクリエイタの記述を可能としている、コード 4.2-1.のように、このクリエイターを用いれば Json の中身をどのようにオブジェクトとして生成するのかを学習させる。

```
final class AgentDeserializer
      implements JsonDeserializer<AbstractAgent> {

   private static final JsonDeserializer<AbstractAgent> AgentDesirializer = new
AgentDesirializer();
   private AgentDesirializer() {
   }
   static JsonDeserializer< AbstractAgent > getAgentDesirializer() {
      return AgentDesirializer;
   }
   @Override
   public AbstractAgent deserialize(final JsonElement jsonElement, final Type
AbstractAgent, final JsonDeserializationContext obj) {
      final JsonObject root = jsonElement.getAsJsonObject();
      final AbstractAgent aa = (AbstractAgent) obj;
      aa.key = root.get("key").getAsJsonPrimitive().getAsInt();
      aa.obj = GsonAgentCreator(aa.key).fromJson(root.get("obj"), obj.class);
      return aa;
   }
   private static Gson GsonAgentCreator(final int key) {
      return new GsonBuilder().registerTypeAdapter(obj.class,
(InstanceCreator<(AbstractAgent) obj>) type -> new obj(key)).create();
```

コード 2.1.1　AgentSphere 用カスタムシリアライザ

3. 機械学習機構

3.1 AgentSphere における機械学習の必要性

AgentSphere で展開されているエージェントは自律的に行動可能だが、その行動の範囲はコードを記述する側の技量に左右される。エージェントのふるまいは現状のままではいかなる手法を用いてもあらかじめ定められた記述にのみ従った振る舞い限定される、今後の AgentSphere の自律性を向上させるための土台として今回の研究を行う。新たに実装されたマイグレーション方式によりエージェントは高速化し、シリアライズの範囲が拡張されたことによって大量のデータの処理が求められ、さらに近年ではトレンドとして流行している機械学習を実現するための機構の実装が現実的となり AgentSphere の利用可能範囲を拡大させるためにこの機構の開発を行うこととした。機械学習のための環境を整える目的として著者らは適切な機械学習のためのライブラリを調査した。

この中でも深層学習（ディープラーニング）が主流である。ディープラーニングを可能とするソフトウェア [4] の数は多くあるが、この論文における AgentSphere のための機械学習機構の実装には Tensorflow [5] を用いることとした。

AgentSphere に適したライブラリとなるには、効率的であり、スケーラブルであり、オープンソースである必要性がある。Tensorflow を選抜した理由は、ディープラーニングライブラリの中でも AgentSphere に必要な仕様に耐えうるほどの柔軟性を持っており、なおかつ、スケーラブルであるのは Tensorflow だと考えた。

柔軟性の面では PyTorch [6] のほうが優秀だが AgentSphere が用いる理論上制限の無い数のエージェントを用いて実現したいスケーラビリティを確保したいのであればオーバヘッドが大きいため、本研究で求める仕様を満足することが不可能である。同様に、スケーラビリティの点では満足する Caffe [5] も挙げられるが、著者ら必要とする記述面での柔軟性を確保するのが難しい、これでは AgentSphere の目的である専門知識を持たないユーザによる容易な記述を確保することは不可能である。

3.2 Tensorflow を用いた AgentSphere の機械学習機構

改めて今回の研究の機械学習機構のためのライブラリとして Tensorflow を選んだ利点を以下に挙げる。

・Python が提供するすべての API を使用することが可能。
・多様な OS に対応している。OS に依存しない AgentSphere と適合する。
・機械学習のためのモデルを保存・読み込みすることが可能。
・現状でもっともポピュラであり開発が旺盛に進んでいる API。

本研究では Tensorflow の API をエージェントを介してそのマシン上に呼び出すという方法をとる。本研究を行うにあたって様々な実装方法を調査したが、AgentSphere とより適合する方法をとることにした。

最初に調査したのは、エージェントから各マシンの Python インタプリタをネイティブに呼び出しコードを実行させる方法である。もう一つはエージェント自身に Java 版の Tensorflow API を持たせて、JVM 上で Tensorflow を用いた機械学習を行う方法である。それぞれの優劣および実装方法については詳細に述べることとする。

3.3 エージェントから Python を呼び出す手法

特定のマシンにすでに Python のインタプリタの環境が構築されているのであれば Java よりランタイム単位で呼び出すことが可能である [7]。先に Java で Python ファイルの中身を BufferedWriter で書き出し、そのファイルを Python インタプリタに読み込ませれば実行することは可能である。

```
import java.io.*;

class test1{
public static void main(String a[]){
try{

String prg = "import sys\nprint int(sys.argv[1])+int(sys.argv[2])\n";
BufferedWriter out = new BufferedWriter(new FileWriter("test1.py"));
out.write(prg);
out.close();
int number1 = 10;
int number2 = 32;
Process p = Runtime.getRuntime().exec("python test1.py "+number1+" "+number2);
BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));
int ret = new Integer(in.readLine()).intValue();
System.out.println("value is : "+ret);
}catch(Exception e){}
}
}
```

図 3.3.1　Python のランタイムでの実行方法

しかし、最新の Java のバージョンでは ProcessBuilder [8] にて Python のインタプリタに特定のファイルをリダイレクトした状態で開始することができる。Java 内部においてオブジェクトとして保存することが可能であるとともに、実行ファイルの同期を行うことができればリモートからでも実行が可能である。

```
import java.io.*;

class test2{
public static void main(String a[]){
try{

String prg = "import sys\nprint int(sys.argv[1])+int(sys.argv[2])\n";
BufferedWriter out = new BufferedWriter(new FileWriter("test1.py"));
out.write(prg);
out.close();
int number1 = 10;
int number2 = 32;

ProcessBuilder pb = new ProcessBuilder("python","test1.py",""+number1,""+number2);
Process p = pb.start();

BufferedReader in = new BufferedReader(new InputStreamReader(p.getInputStream()));
int ret = new Integer(in.readLine()).intValue();
System.out.println("value is : "+ret);
}catch(Exception e){System.out.println(e);}
}
```

図 3.3.2　Python の Processbuilder での実行方法

上述の二つの方法では単一マシン上または同 OS 内でのマシンにおいて実行するのであれば問題は特に認められない。しかし、AgentSphere とはモバイルエージェントシステムとして OS に依存させる記述は極めて好ましくない。そこで考慮されたのが Jython [9] ライブラリを用いた Python の実行法である。

Jython は Java で記述された Python のインタプリタであり Java コードに Python を埋め込むために使用されているものである。今回の研究の検証で使用した Jython の Jar バージョンは 2.7.0。この場合、Java から Python を呼び出すこと自体は可能だったが、我々が使用した Tensorflow には C++のランタイムが含まれているため上記の OS 依存において問題が残る。それを解消するために Java 言語で書かれた Tensorflow を用いることとする。

```
import org.python.util.PythonInterpreter;
import org.python.core.*;

class test3{
public static void main(String a[]){

PythonInterpreter python = new PythonInterpreter();

int number1 = 10;
int number2 = 32;

python.set("number1", new PyInteger(number1));
python.set("number2", new PyInteger(number2));
python.exec("number3 = number1+number2");
PyObject number3 = python.get("number3");
System.out.println("val : "+number3.toString());
}
}
```

図 3.3.1　Python の Jython での実行方法

3.4 Java 版 Tensorflow を用いた機構

Java 版の Tensorflow[]とは Java の API であり、主に Python 内で生成されたモデルデータを Java で読み込むことを目的としている。Java 版の Tensorflow を使用するには AgentSphere が有しているすべてのエージェントに等しく適用されるエージェント化のためのテンプレートファイル AbstractAgent にてそのふるまいを追加した。

```
public void ImportGraph(String Path, Def){
boolean checkpointExists = Files.exists(Paths.get(Path));
byte[] GraphDef = Files.readAllBytes(Paths.get(Def));
Graph graph = new Graph();
Tensor<String> chec Session sess = new Session(graph);
kpoint =
            Tensors.create(Paths.get(Path,"ckpt").toString()))
            {graph.importGraphDef(GrapthDef);
}
```

図 3.4.1　AbstractAgent のグラフインポータ

読み込まれるモデルデータを含むファイルには中断されたときに自身の状態を保存する場所とモデルデータをどう定義するのかについての情報が入っているデータがある。この場合のモデルデータとは Tensorflow 内部においてデータフローの流れが記述されているグラフとして処理される、以降のグラフとはデータフローのグラフである。現段階では読み込まれたグラフファイルはあらかじめ Python で作成する必要があるが、一度作成されれば Java の実行できるプラットフォーム上であれば実行は可能である。

簡単なモデルである $Y = W * x + b$ を使用するとすれば Python で作成されるグラフデータにはあらかじめモデルを定義するための tf.placeholder などの宣言と使用するオプティマイザやロス関数が定義されている。これらの定義はすべて f.write より SerializeToString 関数を呼び出すことによって Java 上で読み込める String としてシリアライズされる。現段階の AgentSphere ではこのようなグラフデータを読み込んだエージェントを数体生成し、それぞれのグラフデータ内部のオプティマイザの種類、学習率、ロス関数の定義などを徐々にランダムにシフトすることで同じモデルに対する複数の学習方法を並列的に生成する。

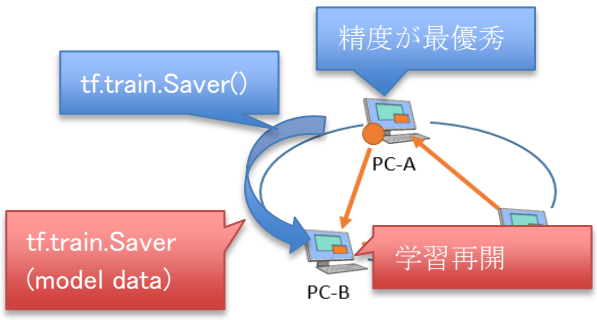

図 3.4.2　最良のモデルの配布方法

それぞれのエージェントは任意設定可能な時間間隔毎にお互いと交信し、現時点におけるモデルの学習で最も良い予測率を持ったモデルを最優秀モデルと定義する。最優秀モデルと定義されたものはいったん自身の処理を中断しそのモデルを保存する。これは Tensorflow の tf.train.Saver クラス内部に含まれる。保存されたモデルデータはネットワーク上で共有され学習しているすべてのエージェントに分配される。これによりほかのエージェントはすべて均一な優秀なモデルを読み込み、現在のランダムで分配されたパラメータでさらに学習を続ける。

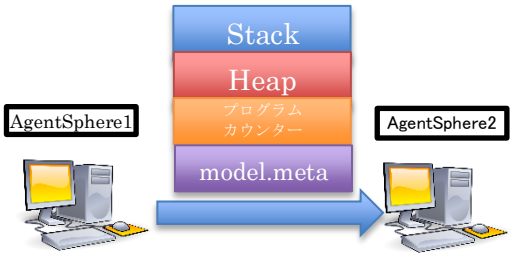

図 3.4.3　マイグレーション時に移動されるデータ

上記の図 3.4.3 から、AgentSphere からエージェントがマイグレーションを行うときもこれらのモデルデータも一緒にアペンドできる、そのため、どこにいても一律に自分の所有しているモデルデータから学習を再開することが可能となっている。

この手法により、本来であればユーザが単一モデルを学習しきった後に手動でモデルの改良を行う手間を省略し、AgentSphere のモビリティを用いてネットワーク上の別の計算リソースに余裕を持っているほかのマシンに移動することで安価な学習のためのリソースを確保することが可能となった。

4. 機械学習機構の評価

ここでは実装した機械学習機構の評価を行う。AgentSphere は並列分散処理を行うのが主な役割であるため、それに見合った試験的な評価を行う。

評価で使用するマシンはソフトウェア研究室のサーバおよび AgentSphere の研究室にある一般オフィス用マシンである。

使用する学習モデルは単純な 1 層のみの MNIST データベースを読み込んで文字認識を行うためのものである。各エージェントは Java 版 Tensorflow を用いてこのモデルを読み込み、それぞれのマシンへ展開されてから処理を開始する。

評価の主な目的は、MNIST のデータベースを用いて文字認識を学習している各エージェントがどの程度の速度で学習し、ほかのマシンから良い結果が送られてきたときにどの程度の性能の向上があるかどうかを確認するためである。

各エージェントにはそれぞれ学習するときのパラメータはある程度ランダムに設定してある。ここではそれらのパラメータを活性化関数、オプティマイザの学習率、バッチサイズ等と定義し、各エージェントはそれぞれ違うパラメータで学習に挑むこととなる。

学習率	活性化関数
0.005~0.001 の間でランダム	Sigmoid
	TanH
	ReLU
	Softmax

表 4.1　主にランダマイズされるパラメータ

4.1 評価結果

同じネットワークに接続されている AgentSphere 環境を四つ用意した。それぞれの AgentSphere には別途五つ目の AgentSphere があらかじめエージェントを用意し、それぞれに AgentSphere に一つずつマイグレーションさせる。

このとき、それぞれのエージェントが何も学習していない状態で MNIST のデータセットを読み込み、それぞれがランダムで記述された活性化関数と学習率を用いて学習を行ったときに記録した精度のグラフは以下の通り。

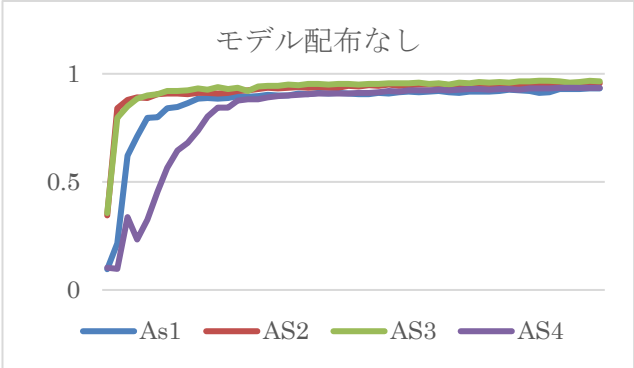

図 4.1.1　モデル配布なしの時の学習結果

活性化関数についてはランダムだが、一回一つの活性化関数を行うエージェントが現れたらそれ以外のエージェントはそれを使用しないように設定した。本来であればここは完全にランダムだが、データを取る上での一律性を保つためにこの設定を施した。単純な学習であるため速度を考慮せず、ステップの数でいかに高い学習結果を残せるかを確認するため、タイムリミットを 5 秒と設定し、それ以降で精度が一番高い AgentSphere のモデルが保存され、配布される。上記のグラフでは 3 番目の AgentSphere がより優れていることがわかるため、そのモデルが使用される。

次のグラフではモデルの配布が行われた後に再学習をさせた結果を示す。

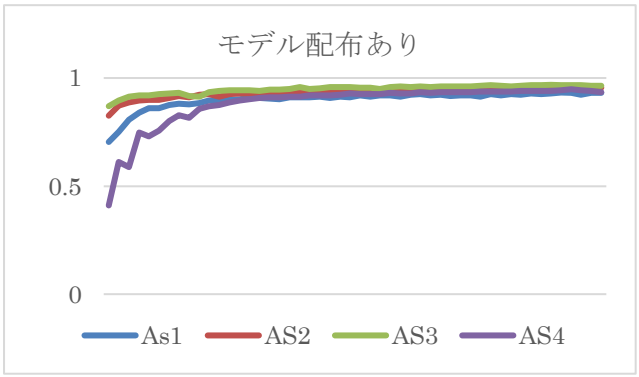

図 4.1.2　モデル配布ありの時の学習結果

これにより、AgentSphere が行った学習の初期精度が上昇し、より早く高い学習結果へ到達することが可能となった。しかし、もともと優秀な活性化関数と学習率を持っていた AgentSphere にとってはそこまで大きな恩恵はないが、マシン間を頻繁に移動するエージェントが必要以上に処理リソースを使うという事態を避けられるため、モバイルエージェントシステムである AgentSphere にとっては利点の一つとなる。

5. おわり

AgentSphere はプラットフォーム依存の無い、専門的な知識を持っていなくても容易で安価な並列分散のための環境を提供するものである。この研究では AgentSphere の使

用幅を広めるための試験的な機械学習を行うための機構とそれの評価を行った。

現在の Tensorflow の実装はまだ新規でありできること自体はまだ幼稚ではあるが AgentSphere が近年のニーズに追いつけるための寄与になったと考える。

Tensorflow を用いることができたことによって、今後はその API を用いて解ける様々な問題を AgentSphere の並列性と分散性を利用して解くことができ、加えて AgentSphere の高い耐故障性を利用することで長期的に、安定して機械学習を行うことができるプラットフォームとして利用できるようになる。今回の手法の利点としては本来であればユーザが考えるべきパラメータさえも、機械学習のモデルに委任することにより、同系列の学習を行っているモデル同士がお互いの学習結果を共有しあうことでより優れた学習結果を残せることである。

近年では GP-GPU を用いることで高速な機械学習を行うことができるサーバはある一方で、それらのサーバを複数用いて並列・分散を用いる機械学習は容易ではない。今後はこのシステムをベースとし、本来であれば計算が難しい問題を、AgentSphere および GP-GPU 搭載のサーバ複数台を用いた並列分散型の機械学習機構を用いて計算をすることが課題である。

参考文献

[1] **赤井雄樹、** *“JavaVM 上での非手続オブジェクト転送を可能とする直列化方式の構築”*、成蹊大学理工学部情報科ソフトウェア研究室学士論文、2010 年

[2] **長塩征幸**、“DHT に基づく検索機能を用いた AgentMonitor の改良”、成蹊大学理工学部情報科ソフトウェア研究室学士論文、2012 年

[3] **Google-Gson**, https://github.com/Google-Gson

[4] **Comparison of deep-learning software**, https://en.wikipedia.org/wiki/Comparison_of_deep-learning_software, 2019 年

[5] **Google Tensorflow**, https://www.tensorflow.org/, 2019 年

[6] **PyTorch**, https://pytorch.org/, 2019

[7] **Berkeley Caffe**, https://caffe.berkeleyvision.org/, 2019.

[8] **Oracle, Java Class Runtime,** https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html, 2019.

[9] **Oracle, Java Process Builder,** https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html, 2019.

[10] Jython, https://www.jython.org/, 2019.