Vol.2015-CG-161 No.3
Vol.2015-CVIM-199 No.3
2015/11/6

JavaScriptによる環境非依存かつ分散計算可能なDeep Convolutional Neural Networkフレームワークの開発

日高 雅俊[1,a)]　三浦 拳[1,b)]　原田 達也[1,c)]

概要：近年，ビッグデータを用いた機械学習，とりわけ Deep Learning が大きな注目を集めている．既存の Deep Learning フレームワークは専用のコンピュータシステムにセットアップすることが必要であり，十分な計算資源を確保するには構築維持のコストが高いという問題があった．本研究では，一般的なパソコンやスマートフォンに搭載されている Web ブラウザ上で動作するプログラミング言語 JavaScript を用いて Deep Learning フレームワークを実装することにより，特別なソフトウェアをインストールすることなく幅広い環境で Deep Convolutional Neural Network を学習できるようにする．さらに Web ブラウザを計算ノードとして用いることが可能な分散計算フレームワークを実装し，これらを組み合わせる．実験では画像分類で高い精度を示す AlexNet を分散学習させ，提案フレームワークの実用性を示した．

1. はじめに

近年，Web サイト上でのユーザの行動履歴や画像・動画等のビッグデータを用いた機械学習，とりわけ Deep Learning が大きな注目を集めている．従来の機械学習手法は画像・音声など処理対象に応じた特徴量設計を研究者が手動で行う必要があったところを，Deep Learning では生のデータから目的変数までの変換処理を一貫したニューラルネットワークで記述し，誤差逆伝播法により特徴抽出を含めたすべてのパラメータを自動で決定するという点が特長である．実際に，画像認識の分野では画像中の物体を分類するコンペティション ILSVRC2012 [1] において，Hinton らのチーム [2] は Deep Learning を用いて従来の手法を大きく凌駕する成績を残した．その後の同コンペティションでも，Deep Learning ベースの手法がさらなる発展を遂げ好成績を示し続けている [3], [4] .

Deep Learning は従来の手法ではできなかった認識精度を達成した一方，学習すべきパラメータの数は従来手法より多くなるため，過学習の防止にはパラメータの数に見合った大量の学習データが必要となることが問題点として挙げられる．このことはデータの収集コストだけでなく，大量のデータを用いて大量のパラメータを学習するための計算コストも増大させる．さらにタスクごとにネットワーク構造等の試行錯誤を行う必要があり，このコストも大きい．この計算コストの問題を解決し，実用レベルの規模の問題への応用を可能としたのが大量の行列計算を効率的に処理できる GPGPU (General Purpose computing on GPU) 技術である．しかしながら GPGPU は比較的高価であり，またこれを用いる Deep Learning フレームワークを計算機にセットアップすることが必須である．さらに複雑なネットワークを学習する場合は複数台の計算機を連携させる分散計算のためのクラスタシステムが必要となり，このようなシステムを構築・運用することができるのは専門的な IT 事業者に限られるのが現状である．そこで本研究では，一般的なパソコン・スマートフォン等の情報機器に搭載されている Web ブラウザ上で動作するプログラミング言語である JavaScript に注目する．JavaScript は近年の Web 技術の発展により，ワードプロセッサが Google や Microsoft のサービスとして Web ブラウザ上で実現されるようになるなど，従来ネイティブアプリケーションとして提供されていたものと遜色ない性能をもつアプリケーションが JavaScript により記述されている．ただ開発言語が変化したのみでなく，個別のアプリケーションをインストールすることが不要であるという大きな利便性も付加されている．さらに，Google Chrome 等の Web ブラウザは Windows のみならず Mac OS X，Linux，Android・iOS スマートフォンなど多様な環境に対応し，これらの間で互換性の高い JavaScript 実行環境を提供している．さらには IoT (Internet of Things) デバイスの試作に用いられる小型マイコンボードでも Linux が稼働し，この上で JavaScript

[1] 東京大学
The University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] hidaka@mi.t.u-tokyo.ac.jp
[b)] miura@mi.t.u-tokyo.ac.jp
[c)] harada@mi.t.u-tokyo.ac.jp

© 2015 Information Processing Society of Japan

を動作させることも可能である．しかしながら，JavaScript を科学技術計算に用いることはほとんどなされていない．その主要な原因は JavaScript はシングルスレッドを前提としており，科学技術計算に重要となる高速な行列計算ライブラリが存在していないという点である．本研究の目的は，実行速度の問題を克服した上で，多様な計算機を専用ソフトウェアのインストール不要で分散計算ノードとして用いることができる効率的な大規模分散 Deep Learning フレームワークを開発することである．

本研究では JavaScript の実行速度の問題を，JavaScript から WebCL と呼ばれる高速計算プラットフォームを用いることにより解決し，現実的な速度で Deep Learning を行えるフレームワークを開発する．なお，本研究では Deep Learning のうち画像分類に広く使われている Deep Convolutional Neural Network (DCNN) に集中して実装を進める．さらに，多数の Web ブラウザを協調させるための分散計算フレームワークを開発しこれらを組み合わせることにより，実用的な規模の DCNN の並列分散を行えることを，ImageNet 大規模画像データセット [5] を用いた AlexNet [2] の学習実験により実証する．

本研究で開発するフレームワークの特徴は以下の通りである．

- GPGPU の利用による，Web ブラウザで動作するライブラリとしては最速の行列計算・Deep Convolutional Neural Network フレームワーク
  - Web ブラウザとサーバ環境双方で同じモデルを学習・利用することが可能
  - ImageNet のような大規模画像データを学習可能なスケーラビリティ
- Deep Learning に限らず多目的に利用できる分散計算フレームワーク
  - 分散計算ノードは，一般のコンピュータの Web ブラウザで特定のページを開いておくだけの容易なセットアップ
- これらの組み合わせにより，計算リソースを CPU と GPU で切り替えるかのように，手軽にサーバ 1 台での学習を複数の Web ブラウザに分散させることが可能

本研究で実装したフレームワークは，MIT ライセンスにて GitHub 上に公開 [*1] されており，自由に利用することが可能である．

2. 関連研究

本章ではまず，科学技術計算向けでない一般のコンピュータを用いた分散処理に関する研究を述べる．SETI@home は，地球外生命体を発見するプロジェクトであり，電波の解析をインターネット上の有志のコンピュータで分散して計算した [6]．専用のソフトウェアをインストールする必要があったものの，300 万台以上のコンピュータが参加し大きな計算リソースを得ることに成功している．Merelo-Guervos らや Klein らは，遺伝的アルゴリズム (GA) を Web ブラウザを計算ノードとして分散計算した [7], [8]．GA はその主要な部分である個体の適応度の計算が完全に並列に行うことができ，有効な分散計算を行うことができている．

次に Deep Learning の分散処理に関する関連研究を述べる．Dean らは，ニューラルネットワークを複数の部分ニューロンのまとまりごとに分割し，それぞれを異なるコンピュータで学習する DistBelief という枠組みを提案した [9]．分割されたネットワークの境界では多数のデータの転送があり，多対多の通信が必要となるため，計算ノードが同一の LAN 上に存在することを前提としない環境では難しい．deeplearning4j は，分散計算フレームワークである Hadoop 環境で Deep Learning を分散するライブラリである [*2]．分散計算に利用する計算ノードすべてに Hadoop のセットアップが必要であり，構築維持のコストが高い．本研究ではその構築維持のコストを大幅に削減するため，Web ブラウザがインストールされている以外の前提条件を設けないという違いがある．Meeds らは Web ブラウザを使った分散 Deep Learning システム MLitB を開発した [10]．しかしネイティブな JavaScript のみで実装され，速度面から大規模なデータの学習はきわめて困難である．また，すべての重みパラメータの更新量をサーバ・計算ノード間で転送しなければならないため，通信のオーバーヘッドが大きい．本研究では行列計算部分を高速化したのみでなく，それに対応して可能になった大規模計算に耐えるデータ転送機構・転送量削減機構等も実装している．

3. 実装

本研究で開発するシステムの設計指針を示す．

- JavaScript が科学技術計算に適さない大きな原因である実行速度の問題を解消するため，数値計算部分においては，WebCL と呼ばれる並列計算プラットフォームを活用する．
- DCNN の並列計算において計算ノード間で共有すべきパラメータが多く，Web ブラウザを介した通信では通信コストが大きいため極力通信量を減らす．
- 計算ノードとなる Web ブラウザでは，容易に分散計算に参加できることを重視し，特定の Web ページに接続する以外の操作を不要とする．

本研究では，行列計算・Deep Learning・分散計算など，すべて JavaScript で実装し，計算速度がクリティカルな面では WebCL を活用したフレームワーク MILJS を開発した．MILJS は複数のフレームワークから構成されており，

[*1] https://github.com/mil-tokyo

[*2] http://deeplearning4j.org

© 2015 Information Processing Society of Japan

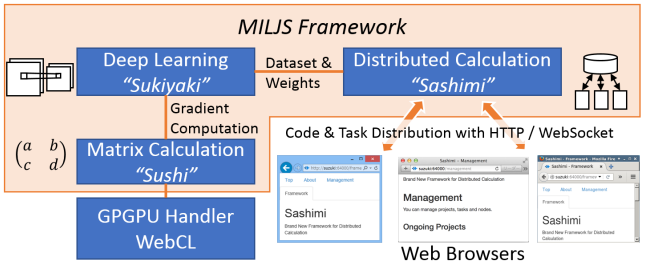

図 1 JavaScript による科学技術計算フレームワーク MILJS の概要

今回は**図 1** のように連携させることにより，DCNN を Web ブラウザを用いて分散計算する．本研究では，JavaScript のボトルネックである計算速度の問題を克服した行列計算ライブラリ”Sushi”，これをバックエンドとした Deep Learning フレームワーク”Sukiyaki”，Web ブラウザを計算ノードとして用いることができる分散計算フレームワーク”Sashimi”を提案する．本研究で提案するこれらのフレームワークを協調動作させることにより，Web ブラウザを計算ノードとした分散計算が可能となる．Sushi, Sashimi は Deep Learning に特化したものではなく，単独で他の用途にも使用可能な設計となっている．その一つとして，我々は Sushi を用いて SVM，K-means 等の機械学習を行えるライブラリ Tempura を MILJS の一部として GitHub 上で公開している．

3.1 行列計算ライブラリ Sushi

Sushi は，JavaScript ベースの新しい行列計算ライブラリである．JavaScript が科学技術計算に適さない大きな原因である実行速度の問題を解消するため，WebCL と呼ばれる並列計算プラットフォームを活用する．WebCL は OpenCL と呼ばれる並列計算プラットフォームの JavaScript 用ラッパである．OpenCL は Chronos Group により標準化された並列計算プラットフォームであり，マルチコア CPU や GPGPU 等を統一的に扱えるクロスプラットフォームな仕組みとなっている．残念ながら現時点で WebCL を標準搭載した Web ブラウザは存在しないが，Firefox のアドオンや Chromium の WebCL 統合版等が公開されており容易にセットアップが可能である．また，JavaScript の実行環境は Web ブラウザに限らず，node.js と呼ばれるサーバサイドの環境も普及しており，これに対応した WebCL モジュールが利用可能である．Sushi は，2 つのソースファイルからなっており，sushi.js をロードすることによりネイティブな JavaScript による行列計算ができる．さらに sushi_cl.js をロードすることにより，WebCL が利用可能であれば計算量の大きい演算が WebCL を用いた実装に置換されるようになっている．例として，行列にスカラーを掛ける処理 times の呼び出し例を**図 2** に，ネイティブな JavaScript による実装を**図 3** に，WebCL による実装のカーネル部分を**図 4** に示す．カーネルは，C 言語で記述され WebCL により動的に機種依存コードにコンパイル・GPU 上に展開されるプログラムである．また，JavaScript の配列と GPU 上のデータの転送は必要最小限にとどまるよう配慮されている．この実装により，Sylvester [11] 等の JavaScript 行列計算ライブラリを大幅に越える速度で計算を行うことができる．

```
1 var a = new $M(10, 10);
2 a.times(2.0);
```

図 2 行列にスカラーを掛ける処理 (times) の呼び出し方法

```
1 $P.times = function(times) {
2   this.syncData();
3   for (var i = 0; i < this.length; i++) {
4     this.data[i] *= times;
5   }
6   return this;
7 };
```

図 3 times のネイティブ JavaScript 実装

```
1 __kernel void kernel_func(__global float *a, float b, uint
      iNumElements)
2 {
3   size_t i = get_global_id(0);
4   if(i >= iNumElements) return;
5   a[i] *= b;
6 }
```

図 4 times の WebCL カーネル

Sushi には行列の作成，要素操作のほか，四則演算，逆行列計算などの基本的な線形代数の関数が実装されている．また，Sushi に実装されていない行列操作を WebCL を用いて実装することも容易にできるよう，ラッパ関数が提供されている．

3.2 Deep Learning フレームワーク Sukiyaki

Sukiyaki は，Sushi をバックエンドに据えた新しい Deep Learning ライブラリである．画像分類の分野で広く活用されている DCNN を学習することを主要な目標とし，畳み込み層，全結合層，Pooling 層等必要なレイヤーをすべて実装しており，また新規レイヤーも容易に実装することが可能である．Web ブラウザで動作するフレームワークとして，Karpathy らによる ConvNetJS [12] が存在するが，Sukiyaki では WebCL をサポートすることにより高速な計算を行える点が大きく異なる．WebCL が利用できない場合は代替となる通常の JavaScript コードが使用され，完全に互換性が保たれるようになっている．

ネットワーク構造は JSON (JavaScript Object Notation) で記述され，さらに重みパラメータ数が少ないネットワー

© 2015 Information Processing Society of Japan

```
{"layer_params":[
{"type":"conv","name":"conv1","params":{"input_rows":28,"input_cols":28,"input_depth":1,"output_depth":20,"window_size":5,"padding":0}},
{"type":"pool","name":"pool1","params":{"input_rows":24,"input_cols":24,"input_depth":20,"window_size":2,"stride":2}},
{"type":"conv","name":"conv2","params":{"input_rows":12,"input_cols":12,"input_depth":20,"output_depth":50,"window_size":5,"padding":0}},
{"type":"pool","name":"pool2","params":{"input_rows":8,"input_cols":8,"input_depth":50,"window_size":2,"stride":2}},
{"type":"fc","name":"fc3","params":{"input_size":800,"output_size":500}},
{"type":"act","name":"act3","params":{"activation_type":"relu"}},
{"type":"fc","name":"fc4","params":{"input_size":500,"output_size":10}},
{"type":"act","name":"act4","params":{"activation_type":"softmax"}}
]}
```

図 5 ネットワーク構造の定義例．28 × 28px の画像を 10 クラスに分類する DCNN を表す．

クであれば，重みも含めて JSON に格納が可能であり，学習済みネットワークの配布が容易である．ネットワーク構造の定義例を**図 5** に示す．レイヤーの種類およびパラメータを配列として記述する．JSON への格納の際，浮動小数点数は Base64 形式に変換され精度を損なわないよう実装されている．より大きなネットワークに対しては，tar フォーマットのバイナリ形式が利用可能で，100MB を超えるパラメータを持つネットワークについても効率的な保存・転送が可能である．

Sukiyaki の実行環境は Web ブラウザだけでなく，サーバサイド JavaScript 実行環境である node.js でも動作するように設計されている．セキュリティ上の制約から入出力機能が制約されている Web ブラウザと異なり，node.js であればファイルシステム，ネットワークに自由にアクセスできるため，学習データのロード方法に柔軟性をもたせることができる．また node.js を用いた Web サービスの一部として Sukiyaki によるデータ分類などの機能を組み込むことも容易である．

データセットの格納は，MySQL データベースを用いる．Web ブラウザからは仲介となる HTTP サーバを経由し Ajax により取得，node.js からは直接データベースにアクセスして取得するようにインターフェースが実装されており，共通のデータセットを利用できるようになっている．インターフェースは独自に実装することも可能である．将来的に，Web 上の画像や動画を学習する場合は Web ブラウザから直接その画像を取得することが効率的と考えられるものの，現在のブラウザのセキュリティ仕様上難しく [*3]，フレームワークとしての汎用性も低いため，データセットを集約したサーバからダウンロードするインターフェースを優先して実装している．画像を処理する場合，データは jpeg 圧縮した状態で格納・転送することにより通信時間を削減する．複数枚の画像は，個別に HTTP リクエストを行うのではなく DataURL 形式に変換した上で JSON 内の配列に格納して 1 つのファイルとして転送する．Web ブラウザではこれを canvas オブジェクトに直接描画することができるため，JavaScript での文字列処理を極力回避し効率的にピクセルデータに復元できる．Web ブラウザではローカルに 5MB を超える大きなデータを保存することができないため，学習した重みを保存するサーバプログラムおよびその Ajax インターフェースも実装した．保存した重みにはユニークな ID が自動で割り振られる．後で述べる分散学習では，重みサーバを経由してサーバと分散計算ノード間の重みを共有する．

3.3 分散計算フレームワーク Sashimi

Sashimi は，Web ブラウザを計算ノードにすることができる分散計算フレームワークである．分散計算は通常，計算ノードにジョブを受信および処理させるための専用ソフトウェアのインストールが必要であり，これがノードを増やす際の大きな障害となる．Sashimi では，計算ノードとなるコンピュータでは Web ブラウザで特定のページを開いておくだけで準備は完了する．Sashimi は Deep Learning とは独立に動作するフレームワークとして設計されており，機械学習に限らず JavaScript で処理可能な任意の計算を Web ブラウザに分散させることができる画期的なシステムである．

C 言語でマルチコア CPU による並列計算を行うライブラリとして OpenMP が有名である．OpenMP では，for ループを並列に計算する際，”#pragma for”と記述するだけで，ループの各インデックスについて並列に実行させることができる．Sashimi においてもこのように全体の処理の一部を手軽に並列化できるよう設計されている．Sashimi のモジュールの 1 つである CalculationFramework は，分散して複数回実行すべきメソッドおよび各回での引数を配列として与えることにより，そのメソッドが計算ノードにより並列に実行されるようになっている．素数のリストを求めるタスクにおいて，ユーザが記述すべきコードの例を**図 7** に示す．タスク作成側では，計算ノードが計算すべきタスクのクラス名を指定し (createTask)，各実行の引数を

[*3] Web ページが存在するサーバ以外から画像をロードし内容を処理するためには CORS と呼ばれるヘッダが必要だが，現在ほとんどのサーバがこのヘッダを送出していない．

© 2015 Information Processing Society of Japan

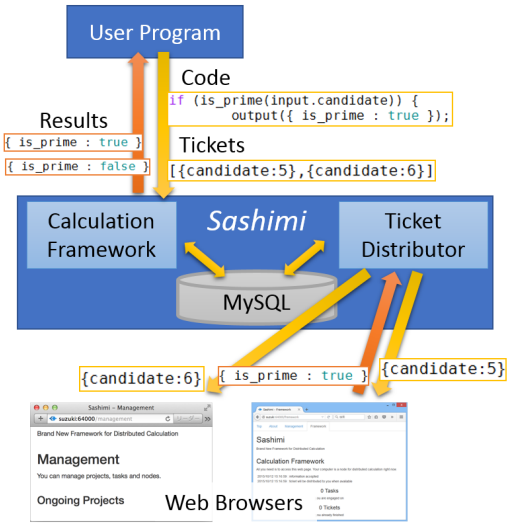

図 6 Web ブラウザを計算ノードとする分散計算フレームワーク Sashimi の概要

```
1  var ProjectBase = require('../project_base');
2  var IsPrimeTask = require('./is_prime_task');
3  var PrimeListMakerProject = function() {
4    this.name = 'PrimeListMakerProject';
5    this.run = function() {
6      var task = this.createTask(IsPrimeTask);
7      var inputs = [];
8      for (var i = 1; i <= 1000; i++) {
9        inputs.push({ candidate : i });
10     }
11     console.log('distributing tickets to clients
           ');
12     task.calculate(inputs);
13     console.log('waiting all tickets to be
           finished');
14     task.block(function(results) {
15       console.log('all tickets have been finished
             ');
16       for (var i = 0; i < results.length; i++) {
17         if (results[i].output.is_prime) {
18           console.log(i + ' is a prime number.');
19         }
20       }
21     });
22   };
23 };
24 PrimeListMakerProject.prototype = new ProjectBase
       ();
25 module.exports = PrimeListMakerProject;
```

図 7 タスク作成側のコード全文

指定し計算を開始する (calculate)．その結果が得られるのを待ち (block)，集計を行う．タスクのクラスには，引数に基づく処理を行い結果を返すコードを記述する．

フレームワークの動作の概要を**図 6** に示す．フレームワークは，計算すべき内容を作成・結果を統合するユーザプログラムに分散計算のための API を提供する CalculationFramework と，計算ノードとなる Web ブラウザと通信を行う TicketDistributor の 2 モジュールからなる．ユーザプログラムでは，計算ノードが読み込むべき JavaScript コードを指定して「タスク」を作成し，タスクのコードが呼び出される各回の引数である「チケット」を作成する．チケットは MySQL サーバに格納され，TicketDistributor によってタスクのコードと共に Web ブラウザに配布される．チケットの処理が完了すると，TicketDistributor に結果が返答される．チケットにはタイムアウトが設定されており，実行中に Web ブラウザが不意に終了した場合は他の計算ノードに割り振り直されるようになっており，多少信頼性が低くても問題ない．TicketDistributor と Web ブラウザの間の通信の大部分には WebSocket が用いられ，通常の HTTP よりも低レイテンシでチケットの配布が可能となっている．

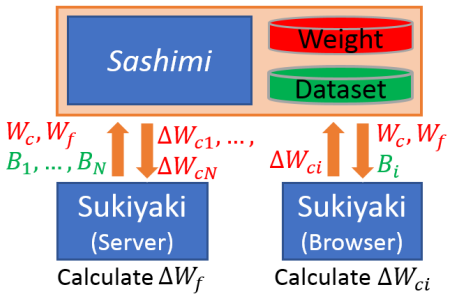

図 8 Deep Learning フレームワーク Sukiyaki と分散計算フレームワーク Sashimi を組み合わせた分散学習システム．Sashimi により制御情報を交換し，重みおよびデータセットサーバを介して重み W_*，勾配 ΔW_*，バッチ内入力サンプル B_* が転送される．

3.4 Sukiyaki と Sashimi による DCNN の分散学習

ここまでで述べた Deep Learning フレームワーク Sukiyaki と，分散計算フレームワーク Sashimi を協調させ，DCNN の並列分散学習システムを構築する．システムの概略を**図 8** に示す．データセットの配布，重みの転送等の分散処理はシステム内で自動的に行われるため，Web ブラウザでサーバに接続するだけで，サーバ 1 台での計算と Web ブラウザも利用した分散計算を容易に切り替えることができる優れたシステムを提供する．

重みの最適化はミニバッチを用いた Stochastic Gradient Descent (SGD) によって行う．ミニバッチ SGD では，数十から数千入力サンプルについて DCNN の出力を計算し，正しい出力との誤差に基づき重みの勾配を求め，ミニバッチ内の平均を用いて重みを更新する．分散計算の際は，重みを全計算ノードで共有し，計算ノードごとに異なる入力サンプルを用いて勾配を計算，サーバで平均を計算する．最適化を担当する optimizer クラスでは，各レイヤーで計算された勾配を SGD により適用する update メソッド，全

© 2015 Information Processing Society of Japan

Algorithm 1 サーバ側の 1 バッチの処理

Send W_c, W_f to weight server
Split B into N subsets $B_1, ..., B_N$
Send tickets $B_1, ..., B_N$ to web browsers
Calculate ΔW_f using data B
Gather results $\Delta W_{c1}, ..., \Delta W_{cN}$ from weight server
$W_c \leftarrow W_c - \eta \frac{1}{|B|} \sum_1^N (\Delta W_{ci})$
$W_f \leftarrow W_f - \eta \frac{1}{|B|} \Delta W_f$

Algorithm 2 Web ブラウザ側の 1 バッチの処理

Get weight W_c, W_f from weight server
Calculate gradient ΔW_{ci} using data B_i
Send gradient ΔW_{ci} to weight server

レイヤーの勾配をバッファに tar 形式でまとめて出力する gatherTar メソッド，この内容を各レイヤーの勾配に加算する addTar メソッドが実装されており，これらを組み合わせることにより分散計算された勾配を利用する仕組みになっている．分散計算により，1 台で計算するより大きなバッチで勾配を計算でき，学習率を上げても安定性が増すためより速く学習を進めることができる．バッチサイズの適切な値はデータセットおよび通信速度により異なるため，ユーザが設定可能である．なお，学習の初期段階ではバッチサイズが小さい方が学習の効率がよいため，まずはじめにサーバのみで一定精度が得られるまで学習を行い，その後分散計算に移行する．

従来の Deep Learning の並列計算では同じコンピュータ内の GPU 間で通信が行われており，重みの共有は短時間で行うことができたが，本研究では Web ブラウザとサーバ間での通信となり通信コストが大きい．Krizhevsky の研究 [13] では，一般的な DCNN において畳み込み層は計算量の 95%を使用する一方重みの 5%のみを持ち，全結合層は計算量の 5%を使用する一方重みの 95%を持っている．そこで，Web ブラウザに計算を行わせている間，サーバでも学習を行う．以下，畳み込み層の重みを W_c，全結合層の重みを W_f と表記する．サーバ側では全結合層の勾配 ΔW_f のみを求め，i 番目の分割されたバッチを割り当てられた Web ブラウザでは畳み込み層の勾配 ΔW_{ci} のみを求めて送信する．この非対称な割当により，サーバ側は畳み込み層を計算する場合と比べ倍以上サンプルを処理でき，また Web ブラウザは送信する勾配のサイズを大幅に減少させることができ，効率よく学習を進めることが可能となる．どのレイヤーの計算を Web ブラウザで行うかは，ネットワークの特性に応じて設定を変えることが可能である．重みの本体は Sashimi のチケットには格納せず，重みサーバ上の ID を格納する．これにより，同一の Web ブラウザが同じ重みをもとに複数のデータの勾配を求める場合は，重みを再度転送しないよう実装されている．

以上で述べた，1 回のミニバッチに対応する勾配を分散計算し重みを更新するアルゴリズムを Algorithm 1, 2 に示す．

表 1 Sushi ベンチマークのタスク

タスク 1	1000 × 1000 行列同士の和
タスク 2	1000 × 100 の行列と 100 × 10 の行列の積
タスク 3	1000 × 1000 の行列と 1000 × 1000 の行列の積
タスク 4	((200 × 500 の行列と 500 × 200 の行列の積) と 200×1 の行列の和 (broadcast)) 及び (200×500 の行列の転置行列と 200 × 500 の行列の積)

表 2 Sushi ベンチマークの結果 (単位:ms)

処理系	ライブラリ	Task1	Task2	Task3	Task4
Firefox	Sushi+CL	33.0	41.6	**52.0**	**107.8**
	Sushi	**5.4**	**5.8**	1962.6	152.4
	Sylvester	130.2	8.4	9376.2	292.4
	Math.js	22.8	43.8	23973.0	1232.0
node.js	Sushi+CL	**5.2**	**1.8**	**13.8**	**3.2**
	Sushi	24.2	6.0	2172.8	80.2
	Sylvester	111.4	24.0	83690.8	1085.0
	Math.js	111.6	90.6	72364.4	1539.2

4. 実験

4.1 行列計算

本節では行列計算ライブラリ Sushi と，既存の JavaScript 行列計算ライブラリとのベンチマークを行い，WebCL を使用することによる速度向上を確認する．

比較項目を**表 1** に示す．タスク 4 は，DCNN の全結合層の forward および backward の計算を想定した一連の処理である．実行するシステム環境は CPU Intel i7-5960X 3.00GHz，GPU NVIDIA Geforce TITAN Z，OS Ubuntu 14.04 とし，JavaScript の処理系は Web ブラウザ Firefox (v32.0.3) および node.js (v0.10.29) での実行を比較した．入力データはランダム生成し，この時間は含まない．WebCL を使用する場合，入力データの GPU への転送および出力データの GPU からの転送時間も計測対象とした．比較する行列計算ライブラリは，Sylvester [11] と Math.js [14] を用いた．Sylvester は 2007 年から存在する行列計算ライブラリで，空間幾何計算の機能が標準搭載されている．Math.js は行列のみでなく多倍長整数や有理数も扱うことができ，現在も活発に機能拡充や速度向上が図られている．

実験結果を**表 2** に示す．Sushi+CL は WebCL を利用した場合，それ以外は WebCL を使用しない場合である．ほぼすべてのタスクにおいて Sushi は他のライブラリより高速であった．WebCL を使用しない場合においても，Sylvester・Math.js は行列表現を入れ子にした配列で行う一方 Sushi はメモリが連続した TypedArray で行ってお

© 2015 Information Processing Society of Japan

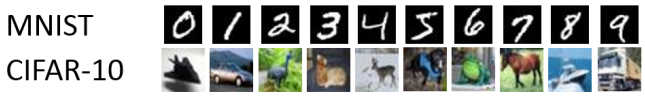

図 9 MNIST，CIFAR-10 データセット

表 3 Sukiyaki ベンチマークの DCNN.
Conv (kernel_size, output_channels, padding): 畳み込み層, FC (output_channels): 全結合層, MaxPool (kernel_size, stride): Max Pooling 層, ReLU: Activation (ReLU) 層

MNIST	CIFAR-10
Conv(5, 20)	Conv(5, 16, 2)
MaxPool(2, 2)	ReLU
Conv(5, 50)	MaxPool(2, 2)
MaxPool(2, 2)	Conv(5, 20, 2)
FC(500)	ReLU
ReLU	MaxPool(2, 2)
FC(10)	Conv(5, 20, 2)
	ReLU
	MaxPool(2,2)
	FC(10)

り，そのアクセス速度に違いがあったものと考えられる．WebCL を利用することにより，特に大きな行列計算が圧倒的に高速化されている．Task3 においては，Sylvester の 180 倍の速度を達成している．一方計算量の小さな処理では，タスク 1・2 のように WebCL 呼び出しのオーバーヘッドによりむしろ時間がかかってしまう場合もみられた．処理系の間では，node.js が概して Firefox より高速であった．同様の結果は MacBook Pro (Mac OS X) でも確認することができた．

4.2 DCNN の学習

本節では DCNN の例として，画像分類データセット MNIST [15] ，CIFAR-10 [16] の学習を行いベンチマーク結果を提示する．MNIST は，手書き数字 10 クラスのデータセットで，28×28 ピクセルのグレースケール画像 60,000 枚を学習データとする．CIFAR-10 は，動物・乗り物の写真 10 クラスのデータセットで，32×32 ピクセルのカラー画像 50,000 枚を学習データとする．データセットの例を**図 9** に示す．

学習する DCNN を**表 3** に示す．バッチサイズは 64 枚とし，損失関数は Softmax + Cross-entropy loss である．これらの DCNN を WebCL を有効にした Sukiyaki，JavaScript による既存の Deep Learning ライブラリ ConvNetJS [12] で学習し速度を比較する．実行環境は Sushi のベンチマークと同様のものを用いる．

1 分あたりに処理したバッチ数を**表 4** に示す．Firefox，node.js 環境双方において，GPU が使用可能な Sukiyaki は ConvNetJS と比較し高速であった．特に，node.js 環境では 10 倍以上の速度向上を達成している．なお，データセットのロードは大規模化を想定し動的に読み込む仕組みにしている．DCNN の構造が比較的単純なため，ロード部分のレイテンシが比較的大きいという問題が見受けられた．

表 4 MNIST, CIFAR-10 の学習速度 (単位: バッチ数/分)

処理系	ライブラリ	MNIST	CIFAR-10
Firefox	Sukiyaki	75	86
	ConvNetJS	42	24
node.js	Sukiyaki	857	461
	ConvNetJS	60	42

表 5 ImageNet の学習速度 (単位: バッチ数/分)

処理系	ライブラリ	速度
Firefox	Sukiyaki	9.1
node.js	Sukiyaki	17.8
-	Caffe	120.0

さらに，本研究のシステムの実用レベルでのスケーラビリティを示すため，ImageNet データセットを用いて，物体認識コンペティション ILSVRC2012 における優勝チームの使用した DCNN である AlexNet [2] の学習を行う [*4]．実験には ImageNet のサブセットである ILSVRC2012 データセットを用いる．これは 120 万枚 1000 クラスの画像からなる大規模なデータセットである．AlexNet は 5 層の畳み込み層と 3 層の全結合層からなり，5,000 万個の重みパラメータを持つ DCNN である．ConvNetJS のような CPU での処理は極めて時間がかかるため，Sukiyaki および，C++で実装され広く用いられているライブラリ Caffe [17] で GPU を使用した場合のみの結果を**表 5** 示す．バッチサイズは 128 枚に設定した．

AlexNet を完全に学習するためには約 900,000 バッチを処理する必要があり，node.js 環境であれば 35 日で学習が可能である．しかしながら，C++で実装された Caffe には速度が及ばなかった．これは言語の実行効率および jpeg 画像のデコードがマルチスレッド化できない点にある．JavaScript 部分の影響が少ない GPU 計算部分においても，行列計算部分で Caffe は NVIDIA GPU に高度にチューニングされた線形代数ライブラリ cuBLAS および cuDNN を使用しており，速度の差につながったと考えられる．OpenCL に対しても AMD が提供する clBLAS というライブラリが存在するが，本研究の実験環境である NVIDIA GPU では使用できないため比較はできなかった．

4.3 ImageNet の分散学習

本節では，Sukiyaki と Sashimi を組み合わせて AlexNet の Web ブラウザによる分散学習を試みる．

分散計算環境としてサーバおよび 2 台のクライアントを使用した．バッチサイズは 1024 枚とし，学習率は 0.028 とした．バッチを 128 枚ずつに分割し，8 つのチケットを作

[*4] 実際には，わずかに変更された版である CaffeNet を学習した．

© 2015 Information Processing Society of Japan

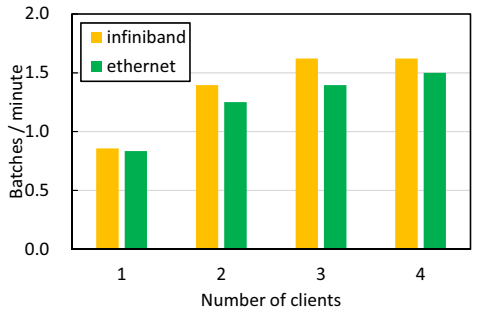

図 10 AlexNet の分散学習におけるバッチの処理速度

成するようにした．分散計算ノードにはそれぞれ NVIDIA K40 が搭載されており，Firefox ブラウザを 2 つ立ち上げて別々の GPU を使用するようにし，最大 4 つの分散計算ノードとして使用し，ノード数と 1 バッチあたりの処理時間を計測した．サーバと分散計算ノード間の通信経路として，Ethernet (1Gbps) と Inifiniband (10Gbps) を使用し，比較を行った．

実験で得られた計算速度を**図 10** に示す．分散せずに学習した場合，1 分間に 1.33 バッチ処理することができた．分散計算した場合，分散計算ノード 1 台では，サーバのみで学習する場合よりも速度が低下した．これは，重みの共有に時間がかかること，またサーバの node.js 環境と比べ分散計算ノードの Firefox 環境は JavaScript の処理速度が遅いためである．分散計算ノード数を増加させることにより，サーバのみでの学習より高速に学習を進めることができた．しかしながら，使用した計算リソースに比して速度向上はまだ改善の余地があるといえる．主な理由としてはまだ重みの共有にかかる時間が比較的長いことである．Web ブラウザの通信処理速度に限度があるようで，Infiniband による速度向上はわずかにとどまった．バッチサイズを大きくしても学習率を増加させることには限界があるため，より通信頻度を下げることのできる新しい学習アルゴリズムの開発および実装が今後の効率化に必要である．

5. 結論

本研究では，特別なクラスタコンピューティングシステムがなくても Deep Learning の分散計算を容易に行えることを目標とし，JavaScript で記述され Web ブラウザ上で動作するフレームワーク群 MILJS を提案した．このフレームワークでは，WebCL を用いることによる従来の JavaScript で達成しえなかった効率的な行列計算を軸として DCNN の学習を現実的な速度で行うことを可能とした．また，MILJS は本研究のように組み合わせて使うだけでなく行列計算・分散計算のライブラリ単独でも十分活用可能に作られており，様々な応用が可能である．DCNN のライブラリはすでに複数提案されているが，本研究で提案したものはサーバ 1 台での学習と複数の分散計算ノードを用いた計算を容易に切り替えることができるという大きな特徴がある．現在のところ WebCL の利用のためには Web ブラウザのアドオンのインストールが必要となっているが，WebCL が Web ブラウザに標準搭載されれば，直ちに学校や会社等のパソコンを数百台規模で分散計算ノードにすることができるようになる．さらにはスマートフォンの性能が向上すれば，その計算能力を活用するだけでなく，カメラ等のセンサデータを用いた新しいアプリケーションも視野に入れることができる．

実験では，行列計算ライブラリ Sushi においては行列積を既存の JavaScript ライブラリの 180 倍，Deep Learning ライブラリ Sukiyaki においては小規模な DCNN で 10 倍以上の速度で処理することができた．さらに，大規模画像データセット ImageNet を用いた DCNN の学習を 4 つの分散計算ノードに分散し現実的な速度で行うことができた．Web ブラウザによりこの規模の DCNN の学習を行うことができるシステムは，我々の知るかぎり唯一のものである．

学習速度面において，現時点ではバッチごとにモデルをサーバとやりとりする必要があり，学習データより重みパラメータのほうが帯域を消費してしまっているためインターネットを経由する場合は回線速度が問題となる．今後はより通信速度が低速な場合においても効率的に学習が行えるような新しいアルゴリズムの考案を行っていく．また，DCNN に限らずより多くの種類のニューラルネットワークの学習ができるよう，機能の拡充も進めていく予定である．

6. 謝辞

本研究は，JST CREST における研究領域「ビッグデータ統合利活用のための次世代基盤技術の創出・体系化」の研究課題「膨大なマルチメディアデータの理解・要約・検索基盤の構築」の支援により行った．

参考文献

[1] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, pp. 1–42, April 2015.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[3] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arxiv:1409.1556*, 2014.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv:1409.4842*, 2014.

[5] Jia Deng and Wei Dong and Richard Socher and Li-Jia

© 2015 Information Processing Society of Japan

Li and Kai Li and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 2009.

[6] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, Vol. 45, pp. 56–61, 2002.

[7] J.J. Merelo-Guervos, P.A. Castillo, J.L.J. Laredo, A. Mora Garcia, and A. Prieto. Asynchronous distributed genetic algorithms with javascript and json. In *CEC*, 2008.

[8] Jon Klein and Lee Spector. Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML. In *GECCO*, 2007.

[9] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc' AurelioRanzato, Andrew Senior, Paul Tucker, Ke Yang, Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[10] Edward Meeds, Remco Hendriks, Said al Faraby, Magiel Bruntink, and Max Welling. Mlitb: Machine learning in the browser. *arxiv:1412.2432*, 2014.

[11] James Coglan. Sylvester. `http://sylvester.jcoglan.com/`.

[12] Andrej Karpathy. ConvNetJS. `http://cs.stanford.edu/people/karpathy/convnetjs/index.html`.

[13] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arxiv:1404.5997*, 2014.

[14] Jos de Jong. Math.js. `http://mathjs.org/`.

[15] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`.

[16] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images, 2009. Master's Thesis, Department of Computer Science, University of Toronto.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv:1408.5093*, 2014.

© 2015 Information Processing Society of Japan