Vol.2019-HPC-168 No.25
2019/3/7

深層ニューラルネットワークにおける訓練高速化のための自動最適化

芹沢 和洋[1,a)]　建部 修見[2]

概要：深層ニューラルネットワークの訓練には大量のデータが必要となり，訓練処理時間の長期化が問題となっている．訓練時間の短縮方法として，複数の訓練データを用いて訓練処理を行うミニバッチ訓練という手法が知られている．本研究では，訓練処理時間と関連性が考えられる，訓練処理中の GPU 利用率を最大化するという最適化手法を用いて．訓練処理時間を可能な限り最短にすることができるミニバッチサイズを決定する方法を提案した．提案手法を深層学習フレームワークである Chainer を用いて実装した．Cifar100 と ImageNet の 2 種類の画像データセットおよび VGG16 と ResNet50 の 2 種類の畳み込みニューラルネットワークを用いて提案手法の評価を行った結果，GPU 利用率のみを最大化するアプローチでは訓練処理速度を最短とするミニバッチサイズを決定することは困難であるという結論となった．一方で，データセットごとに訓練処理中の GPU 利用率とミニバッチサイズとの間の相関性に異なる傾向が観察され，データサイズに起因するボトルネックが GPU 利用を阻害している可能性が発見された．

キーワード：深層ニューラルネットワーク，GPU，Chainer，最適化

1. 研究の背景

深層ニューラルネットワークを用いた機械学習は，2012 年に開催された Imagenet Large Scale Visual Recognition Challenge 2012[1] という物体認識コンペティションにて優勝チームが採用するなどの優れた実績を残しており，現在では画像分類を初め様々な分野での活用が進んでいる．

一方で，深層ニューラルネットワークの訓練には大量の訓練データが必要となり，深層ニューラルネットワークの訓練には数時間から数 10 日間に及ぶ長い処理時間が必要となるという課題が存在する．

機械学習においては，モデルを訓練しそのモデルを用いて評価を行し，その結果をもとにモデルを改善するという試行錯誤を繰り返す方法が一般的である．そのため，訓練時間が長期化すると性能が高いモデルを構築するまでに多くの時間を要し，ビジネスにおいては機会損失などの問題につながる．

一般に GPU を訓練に用いることで訓練処理の高速化が可能であることが知られている．GPU の性能を活かすには，GPU で処理するデータサイズを適切に設定する必要がある．しかし，効率よく適切なサイズを見出す手法は確立されておらず，パラメータを変えながら訓練と評価を繰り返す試行錯誤が必要な状態である．

深層ニューラルネットワークの訓練に用いられる手法の 1 つにミニバッチ訓練が存在する．これは複数の訓練データを用いて一度に訓練を行う手法であり，深層ニューラルネットワークにおいては一般的に使用されている．この手法を用いることで全体の訓練処理回数を削減することが可能である．また，GPU 又は複数のコアを搭載した CPU を使用することにより複数の訓練データの処理を並列に実行することが可能となるため，ミニバッチ訓練を使用することにより訓練処理自体の高速化が期待できる．しかし，一度のミニバッチ訓練で使用する訓練データの個数であるミニバッチサイズは，計算環境，ネットワークの構造，訓練に使用するデータセットなどの組み合わせ毎に最適な値が異なるため，他のパラメータと同様に試行錯誤によって決定されている状態である．

2. 研究の目的

本研究では，訓練速度に最適化したミニバッチサイズを決定する方法を提案する．また，提案する手法を深層学習フレームワークである Chainer[2] に実装し，実際にミニバッチサイズの最適化を行い，任意に決めたミニバッチサ

1 筑波大学大学院システム情報工学研究科コンピュータサイエンス専攻
2 筑波大学計算科学研究センター
a) serizawa@hpcs.cs.tsukuba.ac.jp

ⓒ 2019 Information Processing Society of Japan

イズから訓練速度をどの程度改善できたかを評価する．

3. 関連研究

機械学習モデルのハイパーパラメータを自動で最適化する手法が従来研究において提案されてきた．

Bergstra ら [3] は，ガウス過程と Tree-structured Parzen Estimator を用いたハイパーパラメータ最適化手法を 2 種類提案した．提案手法を用いて，複数レイヤーのパーセプトロンで構成されたニューラルネットワークのパラメータ 10 種類を最適化して画像分類モデルの訓練を行った結果，マニュアルでの最適と場合と比べて推論結果のエラー率が最大で約 5%ほど低くなる結果を示した．Snoek ら [4] は，ベイズ最適化を用いたハイパーパラメータ最適化手法を機械学習に使用する手法を提案した．畳み込みニューラルネットワークのハイパーパラメータ 25 種類の最適化を行った結果，従来手法やマニュアルでの最適化と比較してより速く最適なパラメータを探索できることを示した．これらの研究では訓練された分類モデルの分類精度を最大化する最適化を行っており，処理速度は考慮していない．本研究では分類精度は考慮せずに処理速度を最大化する最適化を目的とする．

また，分散処理を用いて複数のプロセスでミニバッチ訓練を行い，擬似的に大規模なミニバッチ訓練を行う手法が提案されてきた．

Dean ら [5] は，非同期分散処理を用いた深層ニューラルネットワークフレームワークである DistBelief を開発し，ミニバッチ訓練を複数のプロセスを用いて分散実行できることを示した．秋葉ら [6] は，同期分散処理を用いたフレームワークである ChainerMN を開発し，合計 1,024 プロセスで並列に訓練を行い，ImageNet という大規模な画像データセットのクラス分類を行うモデルの訓練を 15 分で完了できることを示した．これらの研究は複数の GPU を用いた分散処理を行うことで処理の高速化を図るものだが，本研究では分散処理を用いずに訓練処理を行うケースにおける高速化手法を検討する．

4. 予備実験

ミニバッチサイズと訓練時間の相関性を確認するため，実際にミニバッチサイズを変えながら訓練処理を実行する．また，訓練処理中の GPU 利用率を測定し，その平均値を求めた．GPU 利用率の測定には，GPU の設定及びモニタリング用の CLI ツールである NVIDIA System Management Interface (nvidia-smi) を使用した．

4.1 実験方法

予備実験ではデータセットとして Cifar100[7]，ネットワークとして VGG16[8] ベースの畳み込みネットワーク (以下 VGG16) を使用した．Cifar100 は 100 種類に分類された 50,000 枚の画像データセットである．データセットに含まれる画像は解像度が 32 × 32 ピクセルの RGB 画像である．本研究で使用する VGG16 の実装は，13 層の畳み込み層，5 層の MAX プーリング層，2 層の全結合層，1 層のバッチ正規化層で構成されている．畳み込み層のフィルタサイズはすべて 3 × 3 である．

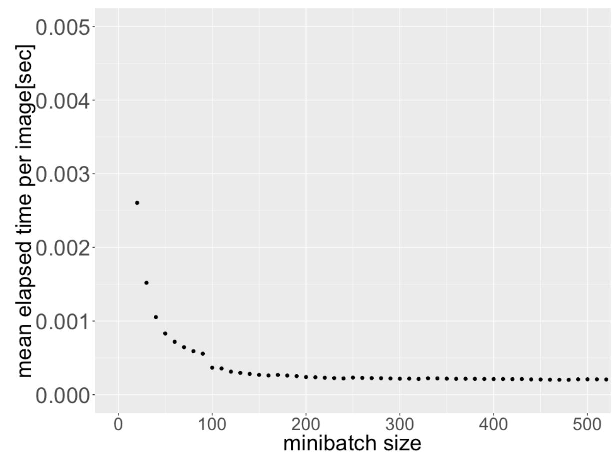

図 1　1 iteration の平均処理時間の推移

実験には，上記のデータセットとネットワークの組み合わせで訓練処理を実行する Chainer のサンプルスクリプトを使用した．このサンプルスクリプトは，Cifar100 の画像データを input とし，input の画像がどのクラスに属するかを判定する画像分類モデルを訓練する処理内容となっている．このスクリプトを使用し，ミニバッチサイズを変えながら 1 epoch 分の訓練を実施し，以下の値を計測した．ミニバッチサイズの値は 10 から 500 まで 10 間隔で増加させた．

- 1 iteration あたりの平均処理時間 [sec]
- 訓練処理中の平均 GPU 利用率 [%]

4.2 実験結果

1 iteration あたりの平均処理時間の推移を図 **1** に示す．このグラフは横軸に訓練時に設定したミニバッチサイズ，縦軸に画像 1 枚あたりの平均訓練処理時間を示す．例えばミニバッチサイズが 100 ならば 1 iteration の算術平均訓練処理時間を 100 で除した値である．

ミニバッチサイズが 10 から 100 前後までの範囲においては，ミニバッチサイズの増加に伴って 1 iteration あたりの平均処理時間が減少し，それ以降はほとんど変化せず収束している様子が観察された．

平均 GPU 利用率の推移を図 **2** に示す．このグラフは横軸に訓練時に設定したミニバッチサイズ，縦軸に訓練時に計測された GPU 利用率の算術平均値を示す．

ミニバッチサイズが 10 から 200 前後までの範囲においてはミニバッチサイズの増加に伴って平均 GPU 利用率が増加し，それ以降は 100%付近で停滞し続ける傾向が見ら

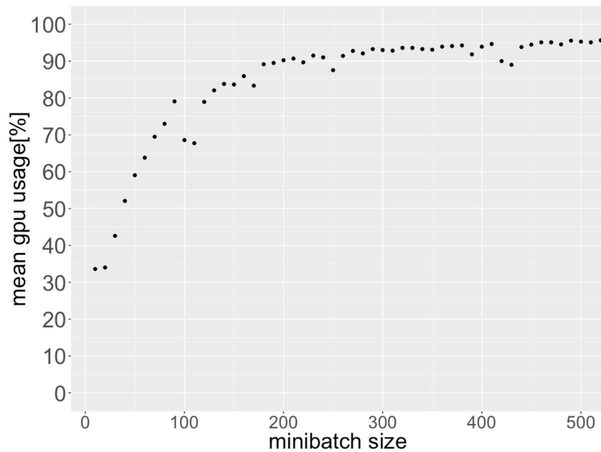

図 2　平均 GPU 利用率の推移

れる．

また，両計測値の傾向を比較すると，どちらもミニバッチサイズが 100 から 200 の同じ範囲で，値変化の傾向が変化していることが確認できる．

以上の実験結果より，訓練処理中の平均 GPU 利用率と 1 iteration あたりの平均処理時間の間には負の相関が確認でき，かつ両者は近い範囲でそれぞれ最大値及び最小値に達する．つまり，平均 GPU 利用率が最大値である 100%付近に到達するタイミングで，1 iteration 当たりの平均処理時間が最小に近い値を示す関係にあることが考えられる．

5. 提案手法

本研究ではミニバッチサイズの自動最適化機能を提案する．最適化手法として以下の 2 種類を提案する．

1 つ目は，先に説明した予備実験の結果に基づき，訓練処理中の GPU 利用率を最大化するミニバッチサイズになるように最適化を行う手法である．

2 つ目は，1 epoch の処理時間を最小化するミニバッチサイズになるように最適化を行う手法である．これは，前者の手法の最適化性能を評価するための対照実験として，実際に訓練時間が最小となるミニバッチサイズを探索するための手法である．これらの最適化手法を実際の訓練処理に適用し，ミニバッチサイズを最適化する機構を実装する．

実装には前述の Chainer を使用する．Chainer を選択した理由は，Python で実装されているためコードの理解が容易であることと，後述する Extension 機能により拡張機能の追加実装が容易である点によるものである．実装は Chainer の github リポジトリの master ブランチをフォークしたブランチをベースに行った．フォークした時点での Chainer のバージョンは 6.0.0.a1 である．

5.1 Chainer による訓練処理の実装

Chainer で機械学習の訓練処理用のスクリプトを実装する場合，大きく分けて 5 種類のクラスのインスタンスを使用する．

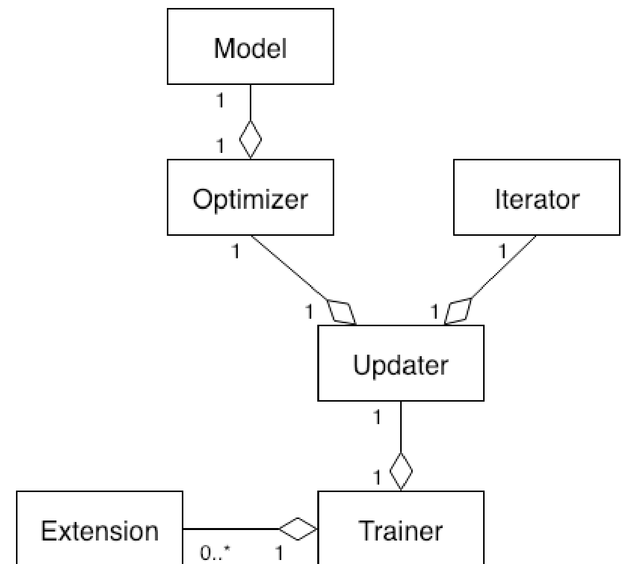

図 3　Chainer 主要クラス図

1 つめは Model クラスである．これは機械学習モデルの内部パラメータ計算ロジックを実装したものであり，Chainer のユーザが要件に応じて実装する．

2 つ目は Optimizer クラスである．これは Model クラスのインスタンスに含まれる機械学習モデルの内部パラメータを更新するクラスである．これを使用したい内部パラメータの最適化手法に応じて選択する．Chainer は複数の内部パラメータの最適化手法に対応した Optimizer クラスの実装を提供しており，例えば *chainer.optimizers.SGD* クラスは確率的勾配降下法を使用して機械学習モデルの内部パラメータを更新する．

3 つ目は Iterator クラスである．これは訓練データへのアクセスを抽象化し，index 指定で訓練データを取り出すことを可能とするクラスである．ミニバッチ訓練を使用する際には，指定したミニバッチのサイズ分だけ Iterator クラスを通じてデータをロードして訓練に使用する実装となっており，ミニバッチサイズを指定するためのパラメータはこのクラスのインスタンス変数として保持される．

4 つ目は Updater クラスである．これは Model クラスのインスタンスと Iterator クラスのインスタンスを使用して実際に訓練処理を実行し，指定された Optimizer クラスを用いて内部パラメータを更新する処理を行うクラスである．

5 つ目は Trainer クラスである．これは，指定された epoch 数分の訓練処理ループを定義し，そのループ内で Updater クラスのインスタンスを使用して訓練処理を実行し，かつその途中経過のレポート出力などを行うクラスである．

この 5 種類のクラスの所有関係を図 **3** に示す．

また，Trainer クラスには訓練処理ループにおいて，訓練処理の開始前，または 1 iteration の訓練処理の前後で任

意の処理を実行するための Extension 機能が存在する．

Extension 機能によって実行される処理は, 関数オブジェクト，または __call__ メソッドを実装したクラスのインスタンスとして実装する．それらのオブジェクトを事前に Trainer クラスのインスタンスに登録しておくことで，訓練処理の前後および，訓練処理ループの開始前に，Trainer クラスが登録されたオブジェクト実行する．

5.2 実装方法

本研究では，Trainer クラスの Extension 機能として，訓練処理ループの開始前にミニバッチサイズを最適化する処理を実装した．この Extension のクラス名を MinibatchSizeOptimizer とした．

また，先に述べた 2 つの最適化手法は，*apply* という名称のパブリックメソッドに最適化処理を実装したクラスとして実装した．最適化手法を実装したクラス名をそれぞれ GpuUsageMaximize，EpochTimeMinimize とした．

これらのクラスのインスタンスを MinibatchSizeOptimizer のコンストラクタで引数として渡して指定できる仕様とし，*strategy* という名称のインスタンス変数として保持するようにした．

MinibatchSizeOptimizer の主要な処理内容をリスト 1 に示す．処理の流れを以下に説明する．まず引数として Trainer オブジェクトと最適化手法を実装したクラスのインスタンスを受け取る．次に Trainer オブジェクトから Updater, Optimizer の各オブジェクトを経由して Model オブジェクトを取得する．次に Model オブジェクトをコピーしたオブジェクトを Updater に付け替え，最適化を実行する．これは最適化によって Model オブジェクトのモデルパラメータを更新してしまうため，最適化完了後の訓練処理に影響を与えないようにするためである．最後に Updater オブジェクトにオリジナルのモデルを付け替えて終了する．

リスト 1 MinibatchSizeOptimizer

```
Input: trainer:Trainer オブジェクト,
    strategy:最適化手法を実装したクラスのインスタンス
 1: updater ⇐ trainer.get_updater()
 2: optimizer ⇐ updater.get_optimizer()
 3:
 4: { 元のモデルをコピーして optimizer にセット }
 5: original_model ⇐ optimizer.get_model()
 6: copied_model ⇐ original_model.copy()
 7: optimizer.setup_model(copied_model)
 8:
 9: { 最適化を実行 }
10: strategy.apply(updater)
11:
12: { オリジナルのモデルを optimizer にセットして終了 }
13: optimizer.setup_model(original_model)
```

GpuUsageMaximize の *apply* メソッドの擬似コードをリスト 2 に示す．処理内容は以下の通りである．まず引数として Updater オブジェクトと，目標とする平均 GPU 利用率の値を受け取る．目標とする平均 GPU 利用率は今回は 99.0%とした．次に，各種変数を初期化する．これらの変数の値については今回は固定とした．次にループ内でミニバッチサイズの変化が収束するか，ループ回数が 100 回に到達するまで最適化処理を実施する．ループ内における最適化処理の詳細を下記に示す．

(1) Updater オブジェクトを用いて訓練処理を任意の回数実行する
(2) 訓練処理の実行中に別スレッドで GPU 利用率を 1 秒ごとに計測する
(3) 計測された GPU 利用率の上位 50%を用いて平均値を求め，この値を「平均 GPU 利用率」として使用する
(4) ループの 5 回に 1 回収束判定を行い，前回の収束判定時のミニバッチサイズとの差分が 5 以下であれば収束したと判定する
(5) 現在のループで求めた平均 GPU 利用率と目標とする平均 GPU 利用率を比較し，上回っていればミニバッチサイズを減少し，下回っていればミニバッチサイズを増加させる

リスト 2 GpuUsageMaximize#apply(updater)

```
Input: updater:Updater オブジェクト, gpu_usage_target:目標と
    する平均 GPU 利用率 (本研究では 99.0%固定)
 1: iteration_count ⇐ 0
 2: norm ⇐ 5
 3: max_iteration ⇐ 100
 4:
 5: while True do
 6:   if iteration_count is over than max_iteration then
 7:     break
 8:   end if
 9:
10:   mean_gpu_usage ⇐ profile_gpu_usage(updater)
11:   if convergent? then
12:     break
13:   end if
14:
15:   if mean_gpu_usage is greater than gpu_usage_target
      then
16:     update_diff ⇐ update_diff + 1
17:   else
18:     update_diff ⇐ update_diff − 1
19:   end if
20:   current_minibatch_size ⇐ current_minibatch_size +
      update_diff
21:   previous_minibatch_size ⇐ current_minibatch_size
22:   iteration_count ⇐ iteration_count + 1
23: end while
```

EpochTimeMinimize の *apply* メソッドの擬似コードをリスト 3 に示す．処理内容は以下の通りである．まず引数

として Updater オブジェクトを受け取る．次に，各種変数を初期化する．これらの変数の値については今回は固定とした．次にループ内で 1 epoch あたりの訓練処理時間の改善率が収束するか，ループ回数が 100 回に到達するまで最適化処理を実施する．ループ内における最適化処理の詳細を下記に示す．

(1) Updater オブジェクトを用いて訓練処理を任意の回数実行する
(2) 実行にかかった前後の時間を記録し，実行回数とデータサイズを用いて 1 epoch あたりの訓練処理時間を導出する
(3) ループの初回であれば，得られた 1 epoch あたりの訓練処理時間をベースラインとして別変数に保存する
(4) ベースラインに対する計測された 1 epoch あたりの訓練処理時間の割合を計算し，この値を「1 epoch あたりの訓練処理時間改善率」として使用する．
(5) ループの 5 回に 1 回収束判定を行い，前回の収束判定時の 1 epoch あたりの訓練処理時間改善率との差分が 0.05 以下であれば収束したと判定する
(6) ミニバッチサイズを増加させる

リスト 3 EpochTimeMinimize#apply(updater)

```
Input: updater:Updater クラスのインスタンス
 1: iteration_count ⇐ 0
 2: norm ⇐ 0.05
 3: max_iteration ⇐ 100
 4:
 5: while True do
 6:   if iteration_count is over than max_iteration then
 7:     break
 8:   end if
 9:
10:   current_elapsed_time ⇐ calc_elapsed_time(updater)
11:   { 初回の計測値をベースラインとする }
12:   if iteration_count is 0 then
13:     base_elapsed_time ⇐ current_elapsed_time
14:   end if
15:   elapsed_time_improvement_ratio ⇐
16:   if convergent? then
17:     break
18:   end if
19:
20:   update_diff ⇐ update_diff + 1
21:   current_minibatch_size ⇐ current_minibatch_size + update_diff
22:   iteration_count ⇐ iteration_count + 1
23: end while
```

6. 評価

6.1 評価方法

評価に使用するデータセットは予備実験で使用した Cifar100 に加え，画像コンペティションである ILSVRC2012 で使用された画像データセット [9](以下 ImageNet-1k) を使用する．これは 1,000 種類のラベル付けがされた RGB 画像で構成されている．前処理として，すべての画像の中心の 224 × 224 ピクセルのみを切り出し，カラーの画像のみを抽出した．前処理を行った結果の画像の合計枚数は 1,261,197 枚である．

評価に使用するネットワークは予備実験で使用した VGG16 に加え，ResNet[10] をベースしたネットワーク (以下 ResNet50) を使用する．ResNet50 は VGG16 同様に，45 層の畳み込み層を持つ畳み込みネットワークの一種として実装されている．VGG16 との相違点として，3 層の畳み込み層を 1 ブロックとして，前のブロックの出力結果を現在のブロックの出力に合計した行列を出力とする構造になっているという特徴がある．

ResNet50 での評価には Chainer 付属のサンプルスクリプトを使用したが，Iterator を MultiprocessIterator から SerialIterator に変更している．これは，Cifar100 を使用するサンプルスクリプトと仕様を合わせ，データセットの画像解像度の差異による影響を確認するためである．

これら 2 種類のモデルとデータセットをそれぞれ組み合わせて 4 パターンの組み合わせを使用する．これらの組み合わせ一覧を**表 1** に示す．

1 epoch 分の訓練を行い，収束するまでのミニバッチサイズの推移と，実際のミニバッチサイズと訓練処理時間の関係性について評価する．

表 1 評価パターン

評価パターン	データセット	ネットワーク
パターン 1	Cifar100	VGG16
パターン 2	Cifar100	ResNet50
パターン 3	ImageNet-1k	VGG16
パターン 4	ImageNet-1k	ResNet50

評価環境として，筑波大学計算科学研究センターに設置されている Pre-PACS-X の計算ノード 1 台を使用した．評価環境の計算ノード仕様を**表 2** に示す

表 2 評価環境の計算ノード仕様

CPU	Xeon(R) CPU E5-2690 v4 @ 2.60GHz × 2
Memory	64GB
GPU	NVIDIA Tesla V100 16GB × 1
Python	3.6.6

6.2 GpuUsageMaximize の評価結果

評価結果の一覧を**表 3**, **表 4** に示す．どの評価パターンにおいても，ミニバッチサイズの増加に正比例して GPU 利用率が増加し，最終的に平均 GPU 利用率がターゲットである 99.0%に近い数値で収束している．この結果より意図した通りに最適化を実現できたものと考えられる．

次に，データセット別の評価結果を以下に示す．なおグ

表 3　GpuUsageMaximize の最適化結果.1

評価パターン	収束に要した最適化回数	初期ミニバッチサイズ	収束時ミニバッチサイズ
パターン 1	10	32	54
パターン 2	15	32	70
パターン 3	10	32	46
パターン 4	20	32	116

表 4　GpuUsageMaximize の最適化結果.2

評価パターン	初期平均 GPU 利用率 [%]	収束時平均 GPU 利用率 [%]
パターン 1	73.0	98.23
パターン 2	55.71	98.35
パターン 3	90.96	98.26
パターン 4	63.08	97.61

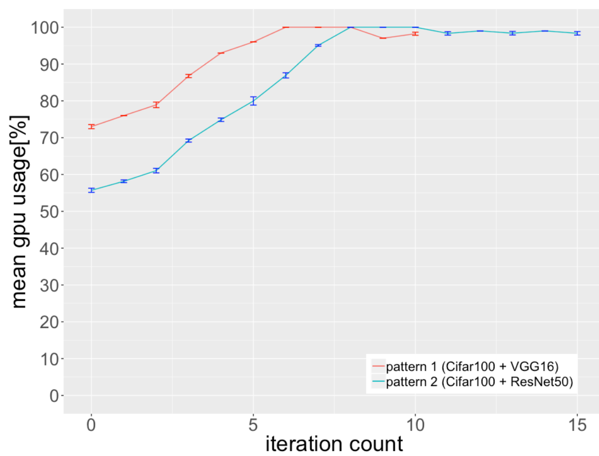

図 4　GpuUsageMaximize/Cifar100 における最適化実行回数ごとの平均 GPU 利用率の推移

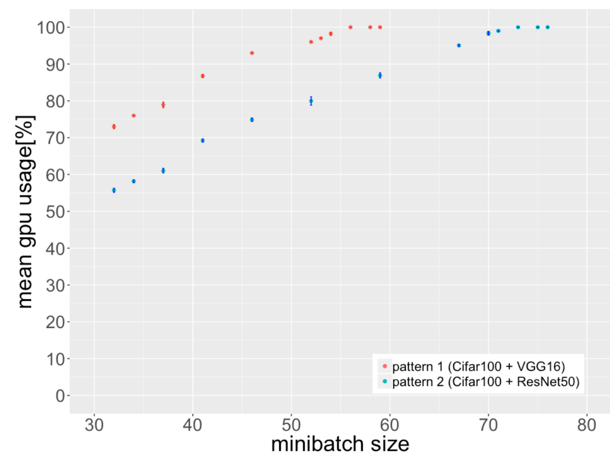

図 5　GpuUsageMaximize/Cifar100 におけるミニバッチサイズと平均 GPU 利用率の関係

ラフ中のエラーバーは標準偏差を示す.

6.2.1　Cifar100 を用いた評価結果

最適化実行回数ごとの平均 GPU 利用率の推移を**図 4** に示す．このグラフは横軸に最適化実行回数，縦軸に平均 GPU 利用率を示す．VGG16 においては iteration count が 6 回，ResNet50 においては 8 回の最適化の実行により，平均 GPU 利用率がほぼ 100%に達し，その後に収束している．

ミニバッチサイズごとの平均 GPU 利用率の関係を**図 5** に示す．このグラフは横軸にミニバッチサイズ，縦軸に平均 GPU 利用率を示す．両パターンともに，ミニバッチサイズの増加に伴って平均 GPU 利用率が増加する正の相関が見られる．その一方で，平均 GPU 利用率が 100%に最初に到達した際のミニバッチサイズは，VGG16 が 60 前後，ResNet50 が 75 前後と差が見られる．これは両ネットワークの計算量の差による影響と考えられる．

6.2.2　ImageNet-1k を用いた評価結果

最適化実行回数ごとの平均 GPU 利用率の推移を**図 6** に示す．VGG16 においては iteration count が 5 回，ResNet50 においては 16 回の最適化の実行により，平均 GPU 利用率がほぼ 100%に達し，その後に収束している．一方で，両ネットワークともにエラーバーが示す範囲が Cifar100 と比べると広く，同じミニバッチサイズを用いて計測された GPU 利用率の分散が大きいことが確認できる．

ミニバッチサイズごとの平均 GPU 利用率の関係を**図 7** に示す．両パターンともにともにミニバッチサイズの増加に伴って平均 GPU 利用率が増加する正の相関が見られる．しかし，各平均 GPU 利用率の分散の大きさを考慮すると，Cifar100 の評価結果と比較して相関性は弱いと考えられる．

平均 GPU 利用率が 100%に到達するミニバッチサイズに差異が見られるのは，先に示した Cifar100 の評価結果と同様の理由によるものと考えられる．

6.3　EpochTimeMinimize の評価結果

最適化結果を**表 5**, **表 6** に示す．

評価結果をデータセット別に以下に示す．

6.3.1　Cifar100

最適化実行回数ごとの改善率の推移を**図 8** に示す．このグラフは横軸に最適化実行回数，縦軸に 1 epoch あたりの処理時間改善率を示す．どちらのネットワークについても最適化の回数に比例して 1 epoch あたりの処理時間改善率

表 5　EpochTimeMinimize の最適化結果.1

評価パターン	収束に要した最適化回数	初期ミニバッチサイズ	収束時ミニバッチサイズ
パターン 1	20	32	196
パターン 2	20	32	196
パターン 3	10	8	72
パターン 4	10	32	96

表 6　EpochTimeMinimize の最適化結果.2

評価パターン	収束時の 1 epoch あたりの処理時間改善率 [%]
パターン 1	73.40
パターン 2	60.31
パターン 3	13.50
パターン 4	8.48

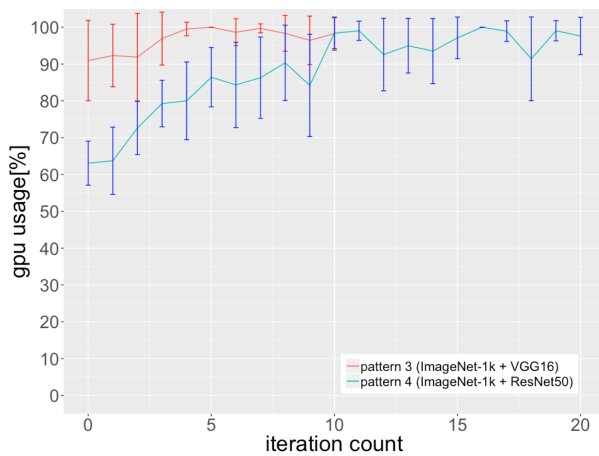

図 6　GpuUsageMaximize/ImageNet-1k における最適化実行回数ごとの平均 GPU 利用率の推移

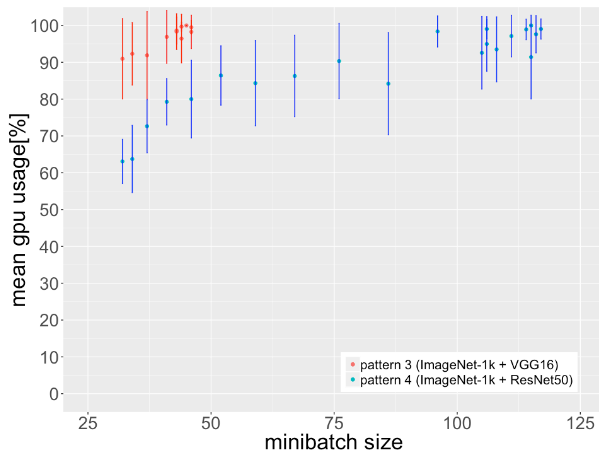

図 7　GpuUsageMaximize/ImageNet-1k におけるミニバッチサイズと平均 GPU 利用率の関係

が向上し，最終的に収束していることが確認できる．

ミニバッチサイズごとの 1 epoch あたりの処理時間改善率の関係を図 **9** に示す．このグラフは横軸にミニバッチサイズ，縦軸に 1 epoch あたりの処理時間改善率を示す．どの評価パターンにおいても，ミニバッチサイズと 1 epoch あたりの処理時間改善率が正比例していることが確認できる．これは GpuUsageMaximize を用いた結果と同じ傾向を示している．一方で，各評価パターンにおける収束時ミニバッチサイズには，両手法間で差異が見られる．

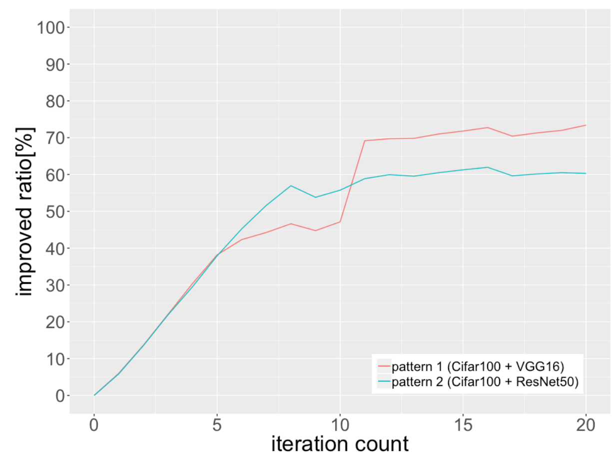

図 8　EpochTimeMinimize/Cifar100 における最適化実行回数ごとの訓練処理時間改善率の推移

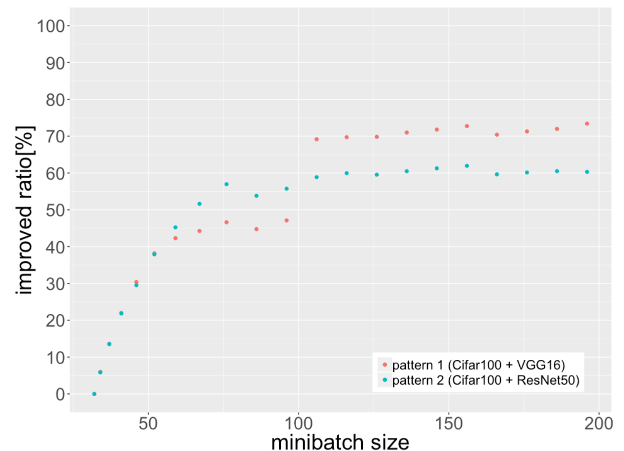

図 9　EpochTimeMinimize/Cifar100 におけるミニバッチサイズと訓練処理時間改善率の関係

6.3.2　ImageNet-1k

最適化実行回数ごとの改善率の推移を図 **10** に，ミニバッチサイズごとの 1 epoch あたりの処理時間改善率の関係を

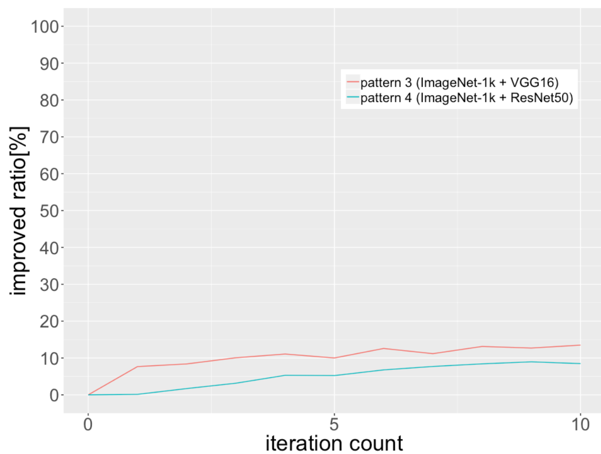

図 10　EpochTimeMinimize/ImageNet-1k における最適化実行回数ごとの訓練処理時間改善率の推移

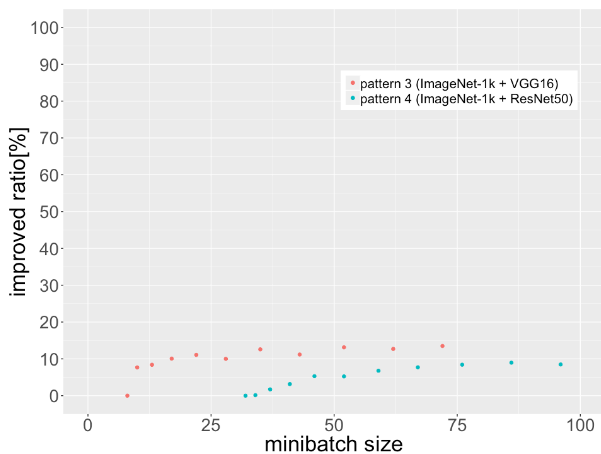

図 11　EpochTimeMinimize/ImageNet-1k におけるミニバッチサイズと訓練処理時間改善率の関係

図 11 にそれぞれ示す．どちらも Cifar100 を用いたパターン 1,2 と同様の傾向を示している．

7. 考察

7.1 手法間の結果の差異について

GpuUsageMaximize と EpochTimeMinimize の手法間の最適化の結果を比較すると，収束時ミニバッチサイズの値に乖離が見られる．手法別の収束時ミニバッチサイズを**表 7** に示す．

特に乖離が大きいのは，Cifar100 を使用したパターン 1 とパターン 2 であり，いずれも EpochTimeMinimize による最適化の収束時ミニバッチサイズが，もう一方の結果を大きく上回る結果となった．

図 9 において，GpuUsageMaximize における収束時ミニバッチサイズ (パターン 1 は 54, パターン 2 は 70) における，1 epoch あたりの処理時間改善率を比較すると，パターン 1 についてはミニバッチサイズ 52 で 1 epoch あたりの処理時間改善率が 38.18%, 収束時の同計測値は 73.40% であり, 実際に GpuUsageMaximize によって最適化されたミニバッチサイズからさらに増加させることで 1 epoch あたりの処理改善率が上昇していることが確認できる．特にミニバッチサイズ 100 を境に大きく性能が向上していることが確認される．今回実際に計測された値としては, ミニバッチサイズ 96 と 106 とで 47.13%から 69.17%へと大きく向上している．この理由については今回は明らかにできなかったため，今後の課題としたい．

パターン 2 については, ミニバッチサイズ 76 で 1 epoch あたりの処理時間改善率が 57.00%となり, 収束時の同計測値である 60.31%に近い値を示している．一方でミニバッチサイズ 76 から 196 までの範囲においては変化は 5%程度の範囲に収まっている．このことから，収束判定の閾値設定に問題があったために結果的に収束するタイミングが遅れ，最終的に過剰にミニバッチサイズを増加し続ける結果となり，結果に乖離が生じたものと考えられる．全体の傾向としては，ミニバッチサイズの増加に伴って 1 epoch あたりの処理時間改善率の上昇幅が小さくなる傾向にあることが計測の結果から明らかになったので，上昇幅を元に動的に収束判定の閾値を決定するロジック追加するなどの改善により，より適切なタイミングで収束判定を行うことができる可能性が考えられる．

ImageNet-1k を使用したパターン 3,4 においても，パターン 1,2 よりは小さいが乖離が見られる．しかし，パターン 3,4 については GpuUsageMaximize で計測された平均 GPU 利用率は分散が大きく，再現性の低い結果であることが想定されるため，この乖離については現時点の情報だけでは原因の解明が困難である．正確に原因を解明するには，1 回の訓練処理における処理時間の内訳をより細かくプロファイルする機能の実装が必要である．また，GPU 上の処理のプロファイリング用 CLI である nvprof を用いる等の手法により，実際に GPU によって処理が行われている時間帯の把握も必要である．

以上の結果より，実際の訓練処理時間の最速化という観点においては GpuUsageMaximize よりも EpochTimeMinimize の方がより適切な最適化手法と考えられる．

表 7　収束時ミニバッチサイズ比較

評価パターン	GpuUsageMaximize	EpochTimeMinimize
パターン 1	54	196
パターン 2	70	196
パターン 3	46	72
パターン 4	116	96

7.2 データセット間の結果の差異について

Cifar100 を用いたパターン 1,2 と ImageNet-1k を用いたパターン 3,4 との計測結果を比較すると，ある程度のミニバッチサイズまでは正比例して増加する傾向は共通している．その一方で，GpuUsageMaximize で計測された平均

Vol.2019-HPC-168 No.25
2019/3/7

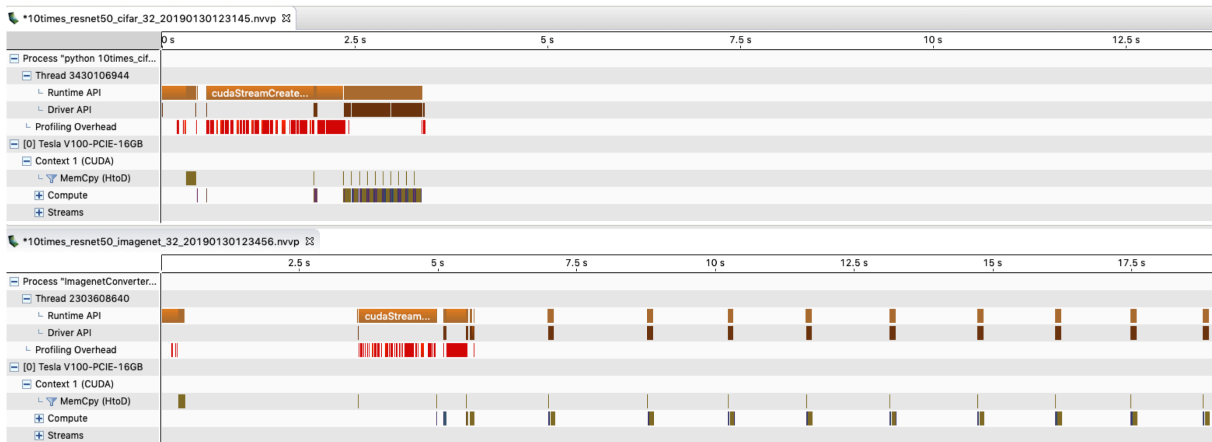

図 **12**　ResNet50 における GPU 利用状況

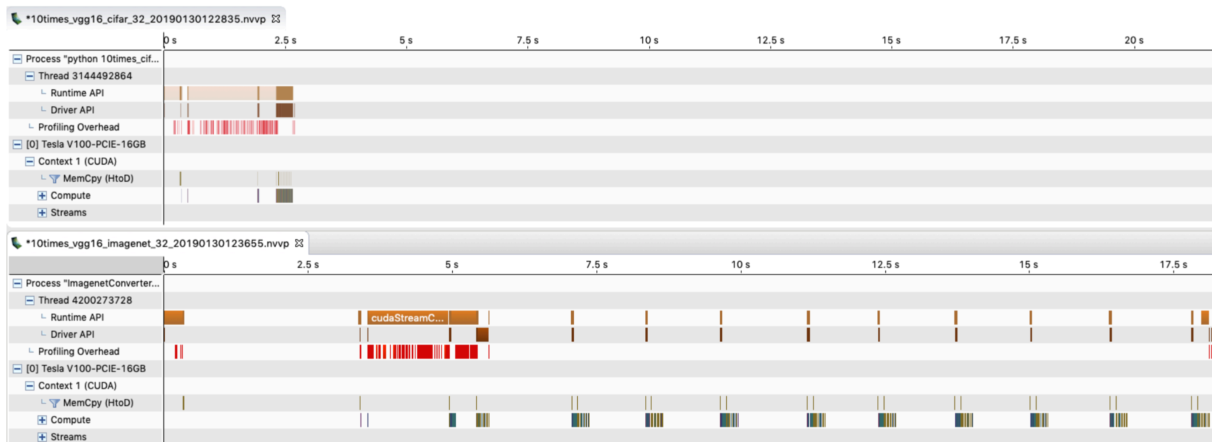

図 **13**　VGG16 における GPU 利用状況

GPU 利用率の値を比較すると，図 7 に示す ImageNet を使用した評価パターンの方が全体的に平均 GPU 利用率の分散が大きくなる傾向となっている．

これは，ImageNet-1k データセットに含まれる画像 1 枚のファイルサイズが Cifar100 と比較して大きいため，GPU を使用しないデータロード等に要する処理の時間が Cifar100 と比較して長期化していることが考えられる．したがって，1 回の訓練処理時間に占める GPU が使用されていない時間帯の比率が，Cifar100 と比較して増加していることが考えられる．今回の実装では，GPU 利用率の計測は 1.0 秒間隔で定期実行されている．GPU が使用されてない時間帯に計測された GPU 利用率は極端に低い値となるため，結果として，GPU 利用率の計測値の母集団の分散が大きくなっていると考えられる．

また，表 6 に示すように，収束時の 1 epoch あたりの処理時間改善率は，パターン 1,2 と比べてパターン 3,4 はかなり低い結果となっている．この理由についても GPU を使用しない処理時間の増加が律速している可能性が考えられる．

これらの仮説を検証するため，パターン 1 からパターン 4 までの組み合わせにおいて，10 回連続で訓練処理を行う処理を対象として，訓練処理中の GPU 利用状況のプロファイリングを行った．このプロファイリングには，GPU 用プロファイリング用の CLI ツールである nvprof を使用した．その後，nvprof の出力ファイルを，可視化用ツールである NVIDIA Visual Profiler を用いて GPU 利用状況をヒートマップとして可視化を行った．それぞれの可視化結果を図 **12**，図 **13** に示す．この図は，横軸が処理開始からの経過時間を示し，縦軸が GPU 上の処理内容の種別を示している．それぞれの図の上段に Cifar100，下段が ImageNet-1k の計測結果を示す．

GPU 上における演算処理の実行状況を示す Compute の行を見ると，Cifar100 は GPU 上で iteration 間に処理を行っていない時間帯がほとんどないが，ImageNet-1k は iteration 間に 1 秒程度の GPU 上で演算を行っていない時間帯が存在している．この「GPU 上で演算が行われていない時間帯」に計測された GPU 利用率が，先の評価実験において低い数値の GPU 利用率として記録されていると考えられるため，先に述べた仮説と矛盾しない結果が nvprof によるプロファイリングによって確認された．

ⓒ 2019 Information Processing Society of Japan

7.3 ネットワーク間の結果の差異について

図 5 と図 7 に示す平均 GPU 利用率において VGG16 と ResNet50 の結果を比較すると，同じミニバッチサイズに同士で比較すると全体的に VGG16 の方が GPU 利用率が高い傾向が見られる．これはネットワークごとの計算量の差によるものと考えられる．

8. 結論

本研究では，深層ニューラルネットワークにおける訓練処理中の平均 GPU 利用率と，1 epoch あたりの処理時間に注目したミニバッチサイズの最適化機構を提案し，深層学習フレームワークである Chainer に実装して評価を行った．

評価の結果，訓練処理中の平均 GPU 利用率を最大化する手法では訓練処理速度が最速となるミニバッチサイズを得ることは困難であると言わざるを得ないという結論となった．訓練処理速度を最速にするという観点では，1 epoch あたりの処理時間改善率が一定の範囲に収束するまでミニバッチサイズを増加させ続けるという手法の方が，1 epoch あたりの処理時間がより最速になるミニバッチサイズが得られた．一方で，この手法についても収束判定を行うロジックについては改善の余地があることが判明した．

評価実験の結果からは，データセットごとの訓練処理中の GPU 利用率の分散の傾向に相違が見られ，何らかのボトルネックが GPU 利用率に影響を与えていること，及びミニバッチサイズの小さな差によって訓練処理時間が大きく改善する現象の発見があった．これらの発見を活用し，今後さらなる提案手法の改善に努める所存である．

9. 謝辞

本研究の一部は，JST CREST JPMJCR1303，JST CREST JPMJCR1414，JSPS 科研費 17H01748，国立研究開発法人新エネルギー・産業技術総合開発機構（NEDO）および富士通研究所との共同研究の助成を受けたものです．

参考文献

[1] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C. and Fei-Fei, L.: ImageNet Large Scale Visual Recognition Challenge, *International Journal of Computer Vision (IJCV)*, Vol. 115, No. 3, pp. 211–252 (online), DOI: 10.1007/s11263-015-0816-y (2015).

[2] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: a Next-Generation Open Source Framework for Deep Learning, *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, (online), available from ⟨http://learningsys.org/papers/LearningSys_2015_paper_33.pdf⟩ (2015).

[3] Bergstra, J., Bardenet, R., Bengio, Y. and Kégl, B.: Algorithms for Hyper-parameter Optimization, *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, USA, Curran Associates Inc., pp. 2546–2554 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2986459.2986743⟩ (2011).

[4] Snoek, J., Larochelle, H. and Adams, R. P.: Practical Bayesian Optimization of Machine Learning Algorithms, *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, USA, Curran Associates Inc., pp. 2951–2959 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2999325.2999464⟩ (2012).

[5] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K. and Ng, A. Y.: Large Scale Distributed Deep Networks, *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, USA, Curran Associates Inc., pp. 1223–1231 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2999134.2999271⟩ (2012).

[6] Akiba, T., Suzuki, S. and Fukuda, K.: Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes, *CoRR*, Vol. abs/1711.04325 (online), available from ⟨http://arxiv.org/abs/1711.04325⟩ (2017).

[7] Krizhevsky, A., Nair, V. and Hinton, G.: CIFAR-100 (Canadian Institute for Advanced Research), (online), available from ⟨http://www.cs.toronto.edu/ kriz/cifar.html⟩.

[8] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *CoRR*, Vol. abs/1409.1556 (online), available from ⟨http://arxiv.org/abs/1409.1556⟩ (2014).

[9] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L.: ImageNet: A Large-Scale Hierarchical Image Database, *CVPR09* (2009).

[10] He, K., Zhang, X., Ren, S. and Sun, J.: Deep Residual Learning for Image Recognition, *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778 (2016).

ⓒ 2019 Information Processing Society of Japan

正誤表

下記の箇所に誤りがございました．お詫びして訂正いたします．

訂正箇所	誤	正
4 ページ リスト 2 15 行目	mean_gpu_usage is greater than	mean_gpu_usage is less than
4 ページ リスト 2 16 行目	update_diff <= update_diff + 1	update_diff <= calc_positive_update_diff()
4 ページ リスト 2 18 行目	update_diff <= update_diff - 1	update_diff <= calc_negative_update_diff()
5 ページ リスト 3 15 行目	elapsed_time_improvement_ratio <=	elapsed_time_improvement_ratio <= calc_improvement_ratio()
5 ページ リスト 3 20 行目	update_diff <= update_diff + 1	update_diff <= calc_positive_update_diff()

ⓒ2019 Information Processing Society of Japan