ワークスペース自動割当による畳み込みニューラルネットワークの高速化

関山 太朗　今井 晴基

畳み込みニューラルネットワーク (CNN) は中間データを保存するためのメモリ (ワークスペース) を割り当てることで高速化されることが知られている．本稿ではニューラルネットワーク (NN) を動的に構築・操作できる深層学習フレームワークにおけるワークスペース管理手法を提案，CNN を自動的に高速化する．NN の動的構築により NN が最低限必要とするメモリ量を推定することが難しくなり，その結果ワークスペースに割けるメモリ量の推測が困難になる．我々は NN が必要とするメモリを空きメモリからだけではなくワークスペースからも割り当てることでワークスペース割当を自動化しつつ NN の構築を妨げない範囲で CNN を高速化する．我々は提案手法を Chainer に実装し，ワークスペースの自動割当により VGGNet などの主要な CNN における伝搬処理が最大 1.60 倍高速化されたことを確認した．

Convolutional neural networks (CNNs) are accelerated by using memory (called *workspace*) in which intermediate data of convolution are stored. This article proposes a workspace management technique for accelerating CNNs automatically in deep learning frameworks that support dynamic construction of neural networks (NNs). Support for dynamic NN construction makes it difficult to estimate the amount of the memory needed by a NN, and, as a result, how much memory we can use as workspace becomes unclear. Our approach allocates memory needed by NN construction from not only free available memory but also workspace, which enables ones to both construct NNs dynamically and allocate workspace automatically. We implemented our workspace management method in Chainer and confirmed that major CNNs such as VGGNet are accelerated 1.60 times.

1 はじめに

深層学習は人工ニューラルネットワークを用いた機械学習技術で，Alex らが物体画像認識コンペティション ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 2012 で優勝 [9] して以来，画像認識，音声認識，自然言語処理といった様々な分野で急速に研究が進められており，その発展にともない多くの深層学習フレームワークが開発されている [8] [1] [14] [2] [3] [5]．これらのフレームワークはニューラルネットワークの構築を簡単にし，微分計算の自動化，GPU による自動並列化といった深層学習の研究開発を促進する多くの機能をサポートしている．

特に最近の TensorFlow, Torch, Chainer, CNTK といった主要なフレームワークではニューラルネットワークの構成要素はプログラミング言語の第一級オブジェクトとして扱われ，実行中にニューラルネットワークを構築・操作することが可能となる．これにより再帰型ニューラルネットワーク [7] のような対象データによって構造が変わるようなネットワークや，GoogLeNet [13] や ResNet [6] といった巨大なニューラルネットワークを簡単に構築・変更することが可能となる．このような柔軟なニューラルネットワークの操作はユーザーにとって便利である一方，フレームワークがニューラルネットワークを解析・高速化することを難しくする一因になる．

本稿では動的ニューラルネットワーク操作をサポートする深層学習フレームワークのための，畳み込み

---

* Accelerating convolutional neural networks by automatic workspace allocation

Taro Sekiyama, Haruki Imai, IBM 東京基礎研究所, IBM Research – Tokyo.

ニューラルネットワークの高速化に向けたメモリ管理技法を提案する．畳み込みニューラルネットワーク [10] は画像認識の畳み込み計算を取り入れたニューラルネットワークで，画像認識分野の他，近年では自然言語処理でも応用されつつある [11]．これまでに複数の畳み込みアルゴリズムが開発されており [4]，その中には中間データを保存するため余剰のメモリを必要とするものもある．そのような高速化のために使われるメモリは**ワークスペース**と呼ばれ，特に GPU のような利用可能なメモリが比較的小さいデバイス上ではワークスペースをどのように割り当てるかが重要となるが，静的にニューラルネットワークを構築するフレームワークに比べ，動的ニューラルネットワーク操作をサポートするフレームワークではワークスペースの自動割当はより難しくなる．

本稿で提案する手法は，ニューラルネットワークの構築・変更にともなって必要になるメモリを空きメモリからだけではなくワークスペースからも割り当てることで，ニューラルネットワークの構築・変更を妨げない範囲で畳み込み計算を最大限高速化するようなワークスペースの自動的割当を実現する．我々はこの手法を Chainer に実装し，AlexNet [9]，GoogLeNet [13]，VGGNet (モデル D) [12]，ResNet-50 [6] の 4 つの畳み込みニューラルネットワークの伝搬処理がそれぞれ最大 1.53 倍，1.09 倍，1.60 倍，1.18 倍高速化されたことを POWER8, NVIDIA Tesla P100 上で確認した．

本稿の以降の構成は次のようになっている．第 2 節では Chainer を例に動的ニューラルネットワーク構築とそれが何故ワークスペースの割当を難しくするのかについて述べる．その後第 3 節で提案手法，第 4 節で提案手法の実装，第 5 節で実験結果について述べ，最後に第 6 節で本論をまとめる．

2 動的ニューラルネットワーク構築のワークスペース割当問題

本節では Chainer を例にニューラルネットワークの動的構築とそれによってワークスペースの割当が難しくなる理由について説明する．

```
class MLP(chainer.Chain):

 def __init__(self, n_units, n_out):
  super(MLP, self).__init__()
  with self.init_scope():
   self.l1 = L.Linear(None, n_units)
   self.l2 = L.Linear(None, n_units)
   self.l3 = L.Linear(None, n_out)

 def __call__(self, x):
  h1 = F.relu(self.l1(x))
  h2 = F.relu(self.l2(h1))
  return self.l3(h2)

model = MLP(1000, 10)
```

ソースコード 1　3 層の全結合をもつニューラルネットワーク

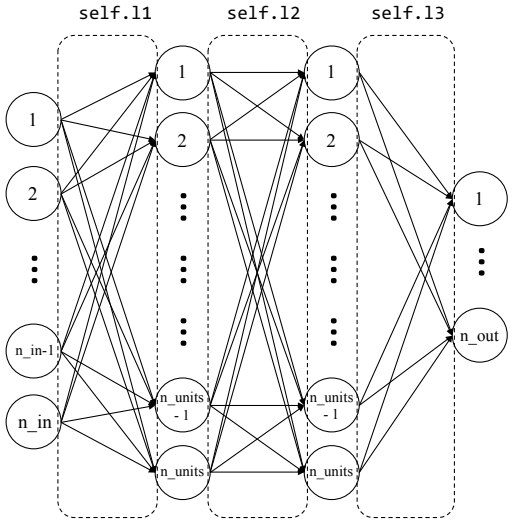

図 1　ソースコード 1 が構築するニューラルネットワーク

2.1 動的ニューラルネットワーク構築

Chainer は Python 上のライブラリとして提供されており，ユーザーが定義するニューラルネットワークも Python 上のオブジェクトとして表現される．ソースコード 1 は Chainer を使ってニューラルネットワークを構成する Python プログラムの例である．[†1] クラ

†1 https://github.com/chainer/chainer/blob/master/examples/mnist/train_mnist.py 2017 年 7 月 14 日アクセス可能であることを確認．

```
def run(epoch, train_data, val_data):
 train_model = MLP(1000, 10)
 val_model = copy_from(train_model)

 for _ in range(epoch):
  train_one_epoch(train_model,
                  train_data)
  validate(val_model, val_data)
```

ソースコード 2　MLP の学習と検証

ス MLP は図 1 のような，2 つの隠れ層をもち，各層が全結合されるニューラルネットワークを表わしており，図中の n_in，n_units，n_out はそれぞれ入力層のベクトルサイズ，隠れ層のニューロン数，出力層のベクトルサイズである．ソースコード 1 では n_units = 1000，n_out = 10 としてニューラルネットワークを構築している．入力層のサイズ n_in は実際に順伝搬処理を行った際に入力データから決定される．MLP オブジェクトを生成する際には，まず __init__ が呼び出されネットワークの各層を全結合する L.Linear オブジェクトが生成される．実際のネットワークの構築はメソッド __call__ で行われる．このメソッドは入力層に相当するベクトル x を受け取ると，各層を全結合し，出力層への結合結果をネットワーク全体の出力として返す．このとき隠れ層のパラメータや隠れ層・出力層の出力 (h1，h2，self.l3(h2)) に対してメモリが割り当てられる．隠れ層の出力はランプ関数 F.relu によって活性化される．

2.2　ワークスペース割当問題

Caffe [8] のような，ニューラルネットワークを静的に構築するフレームワークの場合，ワークスペースは畳み込みニューラルネットワークの構築中に割り当てられることが多い．しかしその方法は動的なニューラルネットワーク構築をサポートするフレームワークには不向きである．

それを示す例として，ソースコード 2 の MLP モデルの学習と検証を行うプログラムについて考えてみる．このプログラムでは学習用・検証用の二つのニューラルネットワーク train_model・val_model を使っている．[†2] ただし val_model は train_model とパラメータを共有するため関数 copy_from によって生成されたものとする．train_model，val_model を生成すると，関数 run は epoch 回だけ学習と検証を繰り返す．

このとき Caffe のようにニューラルネットワークの構築中にワークスペースを割り当てることを考える．Chainer ではニューラルネットワークは順伝搬中に構築されるので最初の train_one_epoch を呼び出す際にワークスペースも割り当てられる．最初の学習が終了すると次は関数 validate によって学習結果の検証が行われる．検証用のニューラルネットワークもこのとき構築され，ソースコード 1 の h1 や h2 といった中間結果を表わすデータにメモリが割り当てられる．このとき，もし学習用ニューラルネットワークとワークスペースがメモリのほとんどを占有し，検証用のニューラルネットワークを構築するのに十分なメモリがない場合，h1 や h2 へのメモリ割当に失敗し実行時エラーが起こってしまう．そのため，動的ニューラルネットワーク構築が可能なフレームワークでは，ニューラルネットワークの構築中にワークスペースを割り当てることは不適切である．

Chainer を含めたいくつかのフレームワークではユーザーがワークスペースのサイズを指定することができ，適切なワークスペースサイズを指定することで上記の問題を解決することができる．しかしこの方法には，ユーザーはニューラルネットワークの構築を妨げず，かつ畳み込み計算を十分に高速化させることのできる適当なワークスペースサイズを事前に知っていなければならないという問題がある．

3　提案手法：ワークスペース管理技法

本稿では動的ニューラルネットワーク構築をサポートするフレームワークのためのワークスペース管理技法を提案する．我々の提案手法は次の 3 つのステップから構成される．

1. ニューラルネットワークの構築時ではなく，伝

†2　Chainer では学習と検証に同じオブジェクトを使うことができるが，ここではわかりやすさのためそれぞれのネットワークを異なるオブジェクトとして与えている..

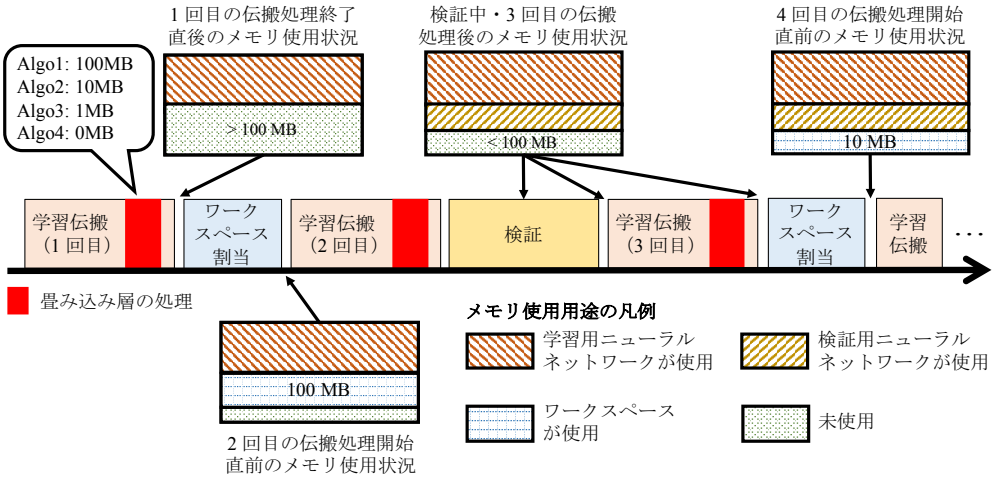

図 2　提案手法を実装したフレームワーク上での学習と検証.

搬計算時に各アルゴリズムが必要とするメモリ量を計算する.

2. 伝搬処理が終わったときワークスペースが未割当てであれば，アルゴリズムが必要とするメモリのうち，可能な限り大きなものに一致するようワークスペースを割り当てる.
3. ニューラルネットワークの計算中に必要になるメモリ (例えば `MLP` の `h1` や `h2` に割り当てられるメモリ) はまず空きメモリから割り当て，もし十分な空きメモリがない場合にはワークスペースから割り当てる．このときワークスペースは一旦解放する.

図 2 は提案手法によるワークスペースの自動割当の例である．この図は提案手法を実装したフレームワークでの，畳み込みニューラルネットワークを用いて 2 回学習のための伝搬計算を行った後に検証を行い，その後再度学習を行った際のメモリ使用状況の推移を表わしている．提案手法はまず最初の伝搬計算時に畳み込み層で各畳み込みアルゴリズムが使用するワークスペースのサイズを求める．ここでは 4 つの畳み込みアルゴリズムが利用可能だとする．この時点ではまだワークスペースは割り当てられておらず，畳み込み計算にはワークスペースを用いないアルゴリズム Algo4 を使用する．その後 1 回目の伝搬計算が終わると未使用メモリからワークスペースを割り当てる．ここでは 100 MB 以上の未使用メモリがあるため，最も多くのワークスペースを必要とする Algo1 が利用できるように 100 MB のメモリをワークスペースとして割り当てる．そして 2 回目の伝搬計算では割り当てられたワークスペースを使って畳み込み処理を行う．2 回目の伝搬計算終了時には既にワークスペースが割り当てられているため，再度ワークスペースを割り当てる必要はない．次に学習結果の検証を行うが，ここで検証用のニューラルネットワークを構築するためには未使用メモリだけでは不十分だとする．その場合，必要に応じてワークスペースを解放することで検証用ニューラルネットワークの構築と検証を可能にする．検証のためにワークスペースを解放したため検証終了後にはワークスペースが割り当てられていない状態になり，3 回目の伝搬計算ではワークスペースを用いずに畳み込み計算を行う．3 回目の伝搬計算終了後には再度未使用メモリからワークスペースを割り当てる．このとき未使用メモリは 100 MB より小さいため，2 番目に大きなワークスペースである 10 MB を割り当てることで 2 番目に速いであろう Algo2 を使うことが可能となる．このようにして提案手法はワークスペースとしてできるだけ大きなメモリを確保し畳み込み計算を高速化しつつ，柔軟なニューラルネットワークの構築・操作を実現する.

```
# P: Dict[int, List[GPU memory]]
def alloc(self, size):
 if len(P[size]) > 0:
  mem = P[size].pop()
 else:
  mem = cudaMalloc(size)
 return Memory(mem)
```

ソースコード 3　cupy のメモリ割当

4　実装

我々は提案手法を Chainer における CUDA GPU 上のメモリ管理技法として実装し，主に (1) GPU メモリの割当処理，(2) 畳み込み計算の前処理，(3) 伝搬計算の後処理に変更を加えた．以降ではこれらの変更の詳細について述べる．

4.1　GPU メモリの割当処理への変更

Chainer での GPU メモリの割当を始めとした GPU 操作は cupy[†3] という GPU 用の数値演算ライブラリを通して行われる．cupy でのメモリ割当処理を簡易化したものをソースコード 3 に示す．cupy では GPU メモリをメモリプール `P` で管理しており，メモリプールはメモリサイズからそのサイズの GPU メモリのリストへのマップとして実装されている．GPU メモリの割当要求が起こるとまずメモリプール `P` に要求されたメモリサイズ `size` のメモリがないかを調べ，あればそれを返しプールからは削除する．もし目的のサイズのメモリがプールされていなければ，新たにシステムに対しメモリ割当命令 `cudaMalloc(size)` を発行し，その結果を返す．割り当てられた GPU メモリは Python の `Memory` オブジェクトとして振る舞い，参照されなくなると Python のガベージコレクションによって回収される．cupy はその回収作業をフックすることで，参照されなくなった GPU メモリをプールに追加する．

我々は cupy の GPU メモリ管理を次のように変更した．

†3　https://github.com/cupy/cupy

```
# P: Dict[int, List[GPU memory]]
# WS: GPU memory
def alloc(self, size):
 if len(P[size]) > 0:
  mem = P[size].pop()
 else:
  try:
   mem = cudaMalloc(size)
  except CUDARuntimeError:
   cudaFree(WS)
   WS = None
   mem = cudaMalloc(size)
 return Memory(mem)
```

ソースコード 4　ワークスペース導入後の cupy のメモリ割当

- 指定したサイズのワークスペース `WS` の割当を明示的に行う関数 `alloc_WS` を追加．この関数により割り当てられたワークスペースはメモリプールによって管理されない．
- メモリ割当処理をソースコード 4 のように変更．cupy へ一般の GPU メモリ割当要求が起こったとき，まずは通常通りメモリプールに要求に適うメモリがないかを確認し，なければ新たにシステムに対しメモリ割当命令を発行する．このときシステムへのメモリ割当命令が失敗，すなわち十分な空きメモリがなければ (これは例外 CUDARuntimeError として通知される)，ワークスペースに割り当てられたメモリを解放してから再度システムへのメモリ割当命令を発行する．ワークスペースの解放後，ワークスペースにメモリが割り当てられていないことを示すため `None` を代入する．この変更によってワークスペースを割り当てつつ，ニューラルネットワークの構築を妨げないメモリ管理が可能になる．

4.2　畳み込み計算の前処理への変更

次に畳み込み計算の変更部分について説明する．ここでは順伝搬方向の畳み込み計算についてだけ述べるが，逆伝搬についても同様の変更を行う．変更前の畳み込みの順伝搬はソースコード 5 のように，畳み込み処理を行うたびに指定されたサイズのワークス

```
def conv_forward(inputs, WS_size):
 WS = cupy.alloc(WS_size)

 # WSとinputsから畳み込みアルゴリズム
 # を選択し，畳み込み計算を実行
 ...
```

ソースコード 5　ワークスペース自動割当実装前の畳み込み処理

```
def conv_forward(inputs):
 WS = cupy.WS

 if WS is None:
  register_WS_size(inputs)

 # WSとinputsから畳み込みアルゴリズム
 # を選択し，畳み込み計算を実行
 ...
```

ソースコード 6　ワークスペース自動割当実装後の畳み込み処理

ペースを割り当てるのに対し，[†4] 変更後はソースコード 6 のように既に割り当てられているワークスペースを使用する．ただし，特に初回の伝搬計算のように，ワークスペースがまだ割り当てられていない場合は，入力 inputs に対して各畳み込みアルゴリズムを実行するのに必要なメモリを計算し，それを保存する (register_WS_size(inputs))．このとき計算されたワークスペースサイズに従って，伝搬処理終了後にワークスペースの割当を行う．

4.3　伝搬計算の後処理への変更

伝搬計算の終了時にはワークスペースを割り当てる．これは Chainer の訓練用関数を変更することで実現する．ソースコード 7 の train_one_epoch は Chainer の訓練用関数を簡易化したもので，ソースコード 2 のように使用される．train_one_epoch は学習用データを全て一度だけ使って学習を行う関数で，9 行目から 13 行目以外は通常通り学習用データをバッチと呼ばれる単位に分割し各バッチごとにパラメータを更新する．9 行目から 13 行目ではワークスペースの割当を行っている．WS_size_list は register_WS_size によって追加されたワークスペースサイズが降順に並べられたリストで，大きなワークスペースサイズから順に割当可能であるか調べる．10 行目の条件式 cupy.WS is not None はワークスペースが既に割り当てられているかを確認しており，既に割り当てられていれば新たなワークスペースの割当は行わない．これはワークスペース割当に利用できる空きメモリは最初の伝搬処理直後が最も多く，それ以降は (一度割り当てられたメモリはプールされるため) 減っていくため，ワークスペースが既に割り当てられている場合，それが畳み込み処理の高速化に利用できる最大のワークスペースとなるからである．ただしこれは通常の畳み込みニューラルネットワークのように，全ての畳み込み計算が初回の伝搬計算で必ず起こる場合にのみ最適なワークスペースを割り当てる．もし初回の伝搬計算で実行されない畳み込み計算があった場合には，その伝搬計算に使用されるワークスペースサイズが登録されず，最適なワークスペースが割り当てられない可能性がある．11 行目では割り当てようとしているワークスペースのサイズが未使用の GPU メモリ量 free_GPU_memory_size 以下であることを確認しており，もしそうなら実際にそのサイズのメモリをワー

```
# WS_size_list: register_WS_size で
# 登録されたサイズの降順リスト
def train_one_epoch(model,train_data):
 optimizer = model.optimizer()
 for data in split_batch(train_data):
  loss = model(data)
  loss.backward()
  optimizer.update()
  for size in WS_size_list:
   if cupy.WS is not None or
      free_GPU_memory_size < size:
    break
   cupy.alloc_WS(size)
```

ソースコード 7　学習関数でのワークスペース割当

†4　cupy は割り当てられた GPU メモリをプールしているため cupy.alloc によるワークスペースメモリ割当によって実際にシステムへメモリ割当命令が発行されるのは最初の順伝搬計算時だけである．

クスペースとして割り当てる．これにより最も大きくかつ割当可能なサイズのワークスペースが割り当てられることになる．

5 実験

我々は提案手法を Chainer に実装し，更に提案手法をより効果的にするために畳み込みアルゴリズムの選択処理にも変更を加えた．まず第 5.1 節でその変更について述べた後，第 5.2 節で実験内容を，そして第 5.3 節で実験結果を示す．

5.1 畳み込みアルゴリズム選択への変更

多くの深層学習フレームワークがそうであるように，Chainer も CUDA GPU 上での畳み込みのために NVIDIA が提供する cuDNN ライブラリ [4] を利用している．cuDNN は深層学習の伝搬処理を高速に行うための CUDA ライブラリで，畳み込み計算の他に再帰型ニューラルネットワークの伝搬計算や活性化関数の CUDA 実装を提供している．cuDNN はいくつかの畳み込みアルゴリズムを実装しており，Chainer はその中から適切なアルゴリズムを選ぶために `cudnnGetConvolution` 関数群 †5 を利用している．`cudnnGetConvolution` はヒューリスティックに畳み込みアルゴリズムを決定するため高速であるが，選択されたアルゴリズムが必ずしも最速なものとは限らないという問題がある．cuDNN バージョン 3.0 からは `cudnnFindConvolution` という関数群が提供されており，これは実際に各畳み込み処理にかかった時間を測定することで最速の (あるいはそれに近い) アルゴリズムを選ぶことのできる畳み込みアルゴリズム選択関数である．ただし，選択中に各畳み込みアルゴリズムを実行するため，選択にかかる時間自体は `cudnnGetConvolution` よりも大きくなってしまう．

我々は畳み込みアルゴリズムの選択に `cudnnFindConvolution` を使うように変更した．Chainer では `cudnnGetConvolution` を伝搬処理を行う毎に呼び出していたが，実際にはワークスペース (と畳み込み層への入力ベクトルのサイズ) が同じである限りその結果は変わらない．そのため我々は `cudnnFindConvolution` をワークスペースが割り当てられてから初めての伝搬処理時にのみ呼び出すことで，畳み込みアルゴリズムの選択にかかる負担を軽減させた．

5.2 実験内容

提案手法により畳み込みニューラルネットワークの伝搬処理が高速化されることを確認するため，本稿では 4 つの畳み込みニューラルネットワーク AlexNet [9]，GoogLeNet [13]，VGGNet (モデル D) [12]，ResNet-50 [6] の学習における伝搬処理速度を比較した．実験は比較対象である Chainer 3.0.0b1 (PFN Chainer)†6 と，Chainer 3.0.0b1 に提案手法と前節の変更を適用した TRL Chainer を用いて行う．PFN Chainer ではワークスペースはソースコード 5 のように割り当てられ，そのサイズはユーザーが設定することができるが自動調整はされない．デフォルトのワークスペースサイズは 8 MB であり，本稿の実験でもこの値を用いた．実験に用いる画像として ImageNet [9] から無作為に選んだカテゴリの画像 (総数 1299) を使用した．実験は POWER8 2.86 GHz 10 コア × 2 (RAM 256 GB)，NVIDIA Tesla P100 (RAM 16 GB; NVLink 1.0) 上で，CUDA 8.0，cuDNN 6.0.20 を用いて行った．

5.3 実験結果

実験結果を図 3 に示す．これらのグラフではワークスペース自動割当を有効にした TRL Chainer (TRL Chainer w/ auto-tune)，ワークスペース自動割当を無効にした (すなわち畳み込みアルゴリズム選択の変更部分だけを有効にした) TRL Chainer (TRL Chainer w/o auto-tune)，そして PFN Chainer の画像処理速度がそれぞれ示されている．“BS” はバッチサイズを表わしており，例えば “32 BS” は 1 バッチの画像数が 32 であることを意味している．バッチサイズが大きくなるほど処理速度も向上しており，これは伝搬計算が CUDA によって並列に行われるため

†5 順伝搬，逆伝搬用にそれぞれ実装されている．

†6 https://github.com/chainer/chainer/tree/v3.0.0b1

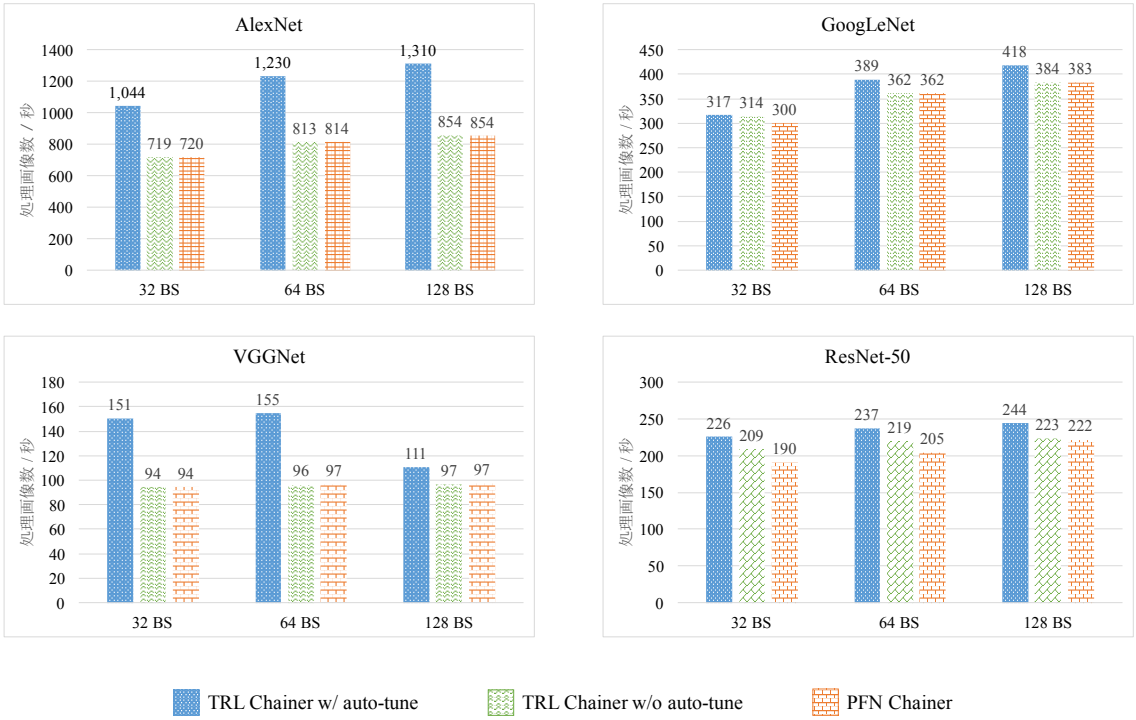

図 3　100 バッチ目から 200 バッチ目までの TRL Chainer と PFN Chainer の画像処理速度.

である.

この結果から，全てのケースで提案手法を適用した TRL Chainer が PFN Chainer よりも高速に動作することが確認された．提案手法が最も効果的だったのは VGGNet で，特に 64 バッチサイズのケースで PFN Chainer に比べて最大の †7 1.60 倍の高速化が達成された．ただしバッチサイズが 128 の場合はバッチサイズ 32，64 のケースよりも効果が小さくなっている．これはバッチサイズが大きくなったことで畳み込み計算以外で必要とされるメモリ量が大きくなってしまい，その結果ワークスペースに使用できるメモリが少なくなり，バッチサイズ 32，64 のときに使用できた最も高速な畳み込みアルゴリズムが使用できなくなってしまったためだと考えられる．また，提案手法の効果が最も小さかったのは GoogLeNet で，特にバッチサイズ 32 の場合，PFN Chainer に対し 1.06 倍しか高速化されなかった．これは GoogLeNet の畳み込み層が他のニューラルネットワークの畳み込み層に比べて小さく，畳み込み処理を高速に行うために必要なワークスペースも小さくすみ，デフォルトのワークペースサイズで十分高速に畳み込み処理が行えたためだと考えられる．ResNet-50 も同様の理由のため AlexNet，VGGNet に比べて提案手法の恩恵が大きくなかったと思われる．

さらに，全てのケースで TRL Chainer w/ auto-tune が TRL Chainer w/o auto-tune よりも高速に動作した．このことから，畳み込みアルゴリズム選択方法の変更だけではなくワークスペースの自動割当が高速化に貢献していることが確認できた．

6　おわりに

Chainer は非常に柔軟な深層学習フレームワークであるが，学習や検証プロセスをまとめたライブラリ関数を提供し，それをユーザーに使ってもらうことでニューラルネットワークの学習・検証をある程度制御

†7　図 3 の数字からは VGGNet 32 BS のケースが最も高速化されているように見えるが，丸め込み前の値を使うと VGGNet 64 BS が最も高速化されていることがわかる.

することができる．我々は GPU 上のワークスペース管理技法を Chainer が提供する学習・検証機能と組み合わせることで，ユーザーコードに手を加えることなく畳み込み処理が高速化できることを確認した．今後の方向性としては TensorFlow などの他のフレームワークでの提案手法の実装や，特にヘテロな分散環境への提案手法ないしはその他の高速化技法の導入が考えられる．

参 考 文 献

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[2] Agarwal, A., Akchurin, E., Basoglu, C., Chen, G., Cyphers, S., Droppo, J., Eversole, A., Guenter, B., Hillebrand, M., Hoens, T. R., Huang, X., Huang, Z., Ivanov, V., Kamenev, A., Kranen, P., Kuchaiev, O., Manousek, W., May, A., Mitra, B., Nano, O., Navarro, G., Orlov, A., Parthasarathi, H., Peng, B., Radmilac, M., Reznichenko, A., Seide, F., Seltzer, M. L., Slaney, M., Stolcke, A., Wang, H., Wang, Y., Yao, K., Yu, D., and andGeoffrey Zweig, Y. Z.: An Introduction to Computational Networks and the Computational Network Toolkit, Technical Report MSR-TR-2014-112, 2014.

[3] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z.: MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems, *arXiv preprint arXiv:1512.01274*, (2014).

[4] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning, *arXiv preprint arXiv:1410.0759*, (2014).

[5] Collobert, R., Kavukcuoglu, K., and Farabet, C.: Torch7: A Matlab-like Environment for Machine Learning, *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The 25th Annual Conference on Neural Information Processing Systems (NIPS)*, 2011.

[6] 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 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778.

[7] Hochreiter, S. and Schmidhuber, J.: Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8(1997), pp. 1735–1780.

[8] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T.: Caffe: Convolutional Architecture for Fast Feature Embedding, *arXiv preprint arXiv:1408.5093*, (2014).

[9] Krizhevsky, A., Sutskever, I., and Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012, pp. 1106–1114.

[10] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P.: Gradient-Based Learning Applied to Document Recognition, *Proceedings of the IEEE*, Vol. 86, No. 11(1998), pp. 2278–2324.

[11] Shi, X., Chen, Z., Wang, H., Yeung, D.-Y., kin Wong, W., and chun Woo, W.: Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting, *arXiv preprint arXiv:1506.04214*, (2015).

[12] Simonyan, K. and Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition, *arXiv preprint arXiv:1409.1556*, (2014).

[13] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A.: Going deeper with convolutions, *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, 2015, pp. 1–9.

[14] 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 29th Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.