特　集「ニューラルネットワーク研究のフロンティア」

ニューラルネットワークの実装
—深層学習フレームワークの構成とChainerの設計思想—

Implementing Neural Networks
— Structure of Deep Learning Frameworks and Design Concept of Chainer —

得居　誠也　株式会社 Preferred Networks
Seiya Tokui　Preferred Networks, Inc.
tokui@preferred.jp

Keywords: deep learning, open source software, neural networks, software design.

1. は　じ　め　に

深層学習は機械学習の一分野である．ニューラルネットワーク全体を同時に学習する技術は，2010年頃から音声や画像の認識においてめざましい成果を上げ，一躍注目の的となった．

深層学習において最も成功しているアプローチは，複雑なニューラルネットの全体を同時に最適化するというものである．このアプローチでは，特徴抽出とそれに基づく予測を同じように扱い，両方をニューラルネットで記述することで，予測部分のみならず特徴抽出の方法も同時に学習する．例えば画像認識において，従来は画像や人間の視覚の特性を反映した特徴抽出手法を構築していた．この特徴抽出を畳込みネットで記述し，その後段の線形予測器と合わせて学習することで，従来の手法に比べて大幅に性能を改善した．

認識タスクに限らず，予測ルーチンを微分可能な処理の連鎖で書くことができれば，誤差逆伝播法によって全体を同時に最適化することができる．最近では，認識のような順伝播型の処理だけでなく，言語の理解や生成，意思決定のような複雑な処理過程も微分可能な処理で記述し，ニューラルネットとして学習する研究が盛んである．

深層学習が成果を上げ注目を浴びるにつれ，リソースも割かれるようになり，研究は加速している．今，深層学習の（特にモデル設計に関わる）研究は，解きたい問題や解決したい課題をいかに微分可能な処理に落とし込み，最適化のための工夫を加えるかが問題となっている．このような研究をするにあたって，新しいネットワーク構造を素早く実装し，実験することは，試行錯誤のサイクルを高速化するために極めて重要である．実装と実験のサイクルを高速化するのは，研究のみならず，深層学習の技術を実用化する開発の成功においても重要な因子である．試行錯誤の高速化において鍵となるのは，ニューラルネットを実装するためのフレームワークの設計である．

本稿では，ニューラルネットの実装の一般的な構成をまず紹介する．その中で，ニューラルネットの実装において最も重要な概念である計算グラフについて詳しく述べる．この計算グラフの実装や扱いにおけるさまざまな選択肢に触れた後，我々が開発しているフレームワークChainer [Tokui 15] の設計を紹介する．

なお，本稿ではニューラルネットを構成する変数は実数値，実ベクトル，実行列，それを一般化した実数値の多次元配列を値としてもつものとする．

2.　ニューラルネット実装の基本構成

ニューラルネットの学習は，損失関数の最適化として定式化される．損失関数は，入力データに対するニューラルネットの出力が正しい出力データからどれくらいずれているかを計算する．損失関数の出力（損失と呼ぶ）を最小化するようにパラメータを調節することで，ニューラルネットが学習される．

最適化には確率的勾配降下法を用いるのが一般的である．その実行には，損失関数のパラメータに対する勾配を計算する必要がある．ニューラルネットは，単純な関数を多数組み合わせてつくられる複雑な関数とみなせるが，その勾配は誤差逆伝播法によって計算できる．

誤差逆伝播法について，人工的で簡単な例を用いておさらいする．

入力変数 x_1, x_2, x_3 を用いて，次のように出力値 y を計算する過程を考える．

$$z = f(x_1, x_2), \quad y = g(z, x_3). \tag{1}$$

ここで f, g はともに二つの配列を取る関数である．誤差逆伝播法の目標は，y の各入力に対する勾配を求めることである．関数 f, g の各入力に対する勾配（ヤコビ行列）は計算できるとする．よって，この例では x_3 に対する y の勾配 $\partial y/\partial x_3$ は直接計算できる．一方，x_1, x_2 に対する y の勾配を求めるには，連鎖律を用いる必要がある．連鎖律によれば，これらは次式で計算できる．

$$\frac{\partial y}{\partial x_1} = \frac{\partial z}{\partial x_1}\frac{\partial y}{\partial z}, \quad \frac{\partial y}{\partial x_2} = \frac{\partial z}{\partial x_2}\frac{\partial y}{\partial z}. \tag{2}$$

ここで，$\partial z/\partial x_1$ や $\partial z/\partial x_2$ はともに f のヤコビ行列の一部である．

計算の過程と勾配を図 1 に示す．図のように一連の演算をグラフで表したものを計算グラフと呼ぶ．上式と図を見比べると，x_i に対する y の勾配はグラフ上のこれらのノードをつなぐパスに沿って順にヤコビ行列をかけたものである．

式（2）を計算する際に，$\partial y/\partial z$ は共通しているため，1 回だけ計算して使い回せばよい．勾配を取りたい目的変数が一つの場合，このように勾配を求めるパスが出力変数側で合流する．また，ニューラルネットの学習においては出力変数 y はスカラであり，一方で入力変数は大きなパラメータベクトルとなる．よって，y のほうから順にヤコビ行列をかけたほうが計算効率が良い．そこで図 1 の点線のように，出力変数から入力変数へのパスに沿って勾配を計算する．このようにして勾配を計算する方法を誤差逆伝播法という．

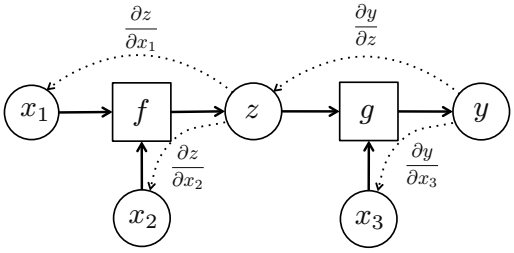

図 1　式（1）の計算グラフとその勾配

もう一つ簡単な例として，分岐がある計算グラフにも触れておく．関数 f_1, f_2, g を用いて入力 x から出力 y を次の手順で計算する．

$$z_1 = f_1(x), \quad z_2 = f_2(x), \quad y = g(z_1, z_2). \tag{3}$$

勾配 $\partial y/\partial x$ は次式で表される．

$$\frac{\partial y}{\partial x} = \frac{\partial z_1}{\partial x}\frac{\partial y}{\partial z_1} + \frac{\partial z_2}{\partial x}\frac{\partial y}{\partial z_2}. \tag{4}$$

対応する計算グラフを図 2 に示す．式（4）の右辺の二つの項は，計算グラフ上で y から x へ至る二通りのパスにそれぞれ対応する．よって，誤差逆伝播のアルゴリズムで式（4）を捉えると，分岐点 x では伝播してきた

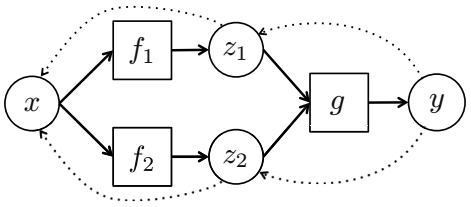

図 2　式（3）の計算グラフ

複数の勾配を足し合わせることになる．これはより一般の計算グラフに一般化される．つまり，計算グラフの分岐点では，各パスから伝播してきた勾配の総和を計算する必要がある．

さて，ここまでで誤差逆伝播法について駆け足ではあるが一通り述べた．誤差逆伝播法を実装する方法はいろいろ考えられるが，基本的には計算グラフをつくり，それを逆向きにたどるということになる．

3. フレームワーク設計における選択肢

深層学習のフレームワークを設計するにあたって，さまざまな選択肢がある．そこでどのような選択を行うかで，各フレームワークは異なる特性を有することとなる．本章ではいくつかの選択肢を紹介し，それがフレームワークにどのような特性を与えるかを見ていく．

3·1　計算グラフの記述方法

深層学習フレームワークに求められる最も重要な機能は，計算グラフによる誤差逆伝播の自動化である．ユーザから見たとき，この機能を使うには計算グラフを記述する必要があるが，いくつかの記述方式がある．

最も大きなくくりとして，計算グラフを静的に記述するか動的に構築するかによる違いがある．前者のアプローチは，Caffe [Jia 14] や Pylearn2 [Goodfellow 13] などが代表的だが，何らかのデータ記述言語を用いて計算グラフをデータとして記述する．プログラマでないユーザにも受け入れられやすい方式であるが，用いられる言語は高度な抽象化をサポートしない場合が多く，複雑なネットワークを書こうとすると煩雑になりがちである．一方，後者のアプローチでは汎用のプログラミング言語を用いて，計算グラフを出力するプログラムを書く．Theano [Bergstra 10] や Torch7 [Collobert 11] などをベースにしたものを始め，プログラマ向けの多くのフレームワークがこのグループに当てはまる．プログラミング言語のもつ抽象化能力を利用して，複雑なネットワークも簡潔に書くことができる．

さて，計算グラフを動的に生成する方法にも二通りのやり方がある．

一つは，学習ループの前にあらかじめ計算グラフ全体をつくっておき，順伝播や逆伝播のような操作をその計算グラフ上で実行するというものである．Theano ベー

表1 計算グラフの記述・構築方法による違い

記述方法	宣言的	命令的	
構築方法	事前に構築		順伝播時に構築
可搬性	✓	✓	
最適化	✓	✓	
処理系との親和性			✓
可変なネットワーク構造			✓
フレームワーク	Caffe	Theano ベース, Torch.nn, Neon, MXNet, MXNet, TensorFlow	Chainer, autograd

スのフレームワークや Torch.nn, Neon[*1], TensorFlow [Abadi 15] などはこの枠組みに収まる．この方法では，学習の実行前に計算グラフが定まっているため，後述する最適化がしやすい．一方で，順伝播の処理に複雑な条件分岐やループが必要な場合，計算グラフにそれらをすべて表現しなければならないため，フレームワークの仕様が大きくなりがちである．

もう一つの方法は，順伝播する際に実際にどのような計算が行われたかをグラフとして記録し，誤差逆伝播だけをそのグラフ上で実行するというものである．つまり，順伝播の処理と計算グラフの構築が同時に行われる．Chainer や autograd がこの枠組みに収まる．この方式では，データごとに異なるグラフを構築したり，ベースとなるプログラミング言語の条件分岐やループを順伝播の中で自由に使えるといった利点がある．一方で，データごとに計算グラフが構築されるオーバヘッドがかかったり，順伝播を最適化しづらいといった効率面での問題がある．この方式については次節でより詳しく述べる．

グラフの記述方法による違いを表1にまとめる．現状では，あらかじめ計算グラフを構築しておくフレームワークがよく使われている．もっとも，順伝播と計算グラフの構築を同時に行うフレームワークは2015年に出始めたばかりのアプローチであり，今後の発展が期待される．

3·2 勾配計算の実装方法

前節で述べたとおり，勾配の計算は計算グラフ上の誤差逆伝播で記述できる．その実装方法は大きく二通りに分類される．

一つ目の方法は，各演算ノードに逆伝播の処理も定義しておき，逆伝播の際にはその処理を実行するというものである．この場合，前節の説明どおり，計算グラフを逆向きにたどりながら逆伝播を実行する．

もう一つの方法は，計算グラフを逆向きにたどりながら，誤差逆伝播に相当する計算グラフを付け加えていくというものである．例えば図3は，式 $z = x_1 \odot x_2, y = z - x_3$ の計算グラフに，その誤差逆伝播の計算グラフを付け加えたものである．ここで $\odot$ は要素積とする．この式で使われる二つの演算 $\odot$, $-$ に対して，誤差逆伝播の処理がまた計算グラフを構築するコードとして書かれていれば，図のような展開が可能となる．

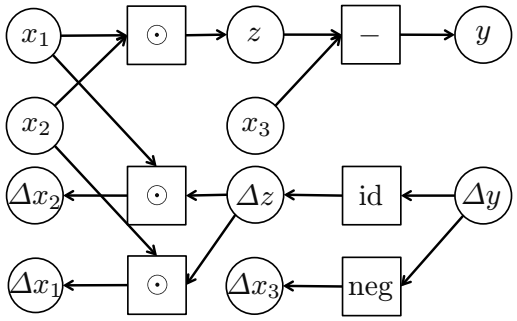

図3 式 $z = x_1 \odot x_2, y = z - x_3$ とその勾配の計算グラフ．neg は負号を付ける関数，id は恒等関数である．また，Δ は勾配を表す

誤差逆伝播を計算グラフとして展開する方式の利点は，最終的な計算グラフにおいて順伝播と逆伝播の処理が同じ形式で記述されるという点である．これによって，計算グラフを最適化しやすくなる．

3·3 計算の最適化

深層学習において，最も時間のかかる工程が学習である．深層学習は膨大なデータセットと複雑なニューラルネットにおいて最も効果を発揮する．そのため，そのほかの機械学習手法と比べても速度やメモリ効率が重要となる．深層学習のフレームワークにおいて，ある程度以上の規模の問題を扱うにはまず GPU による計算が可能である必要がある．そのうえで，GPU での計算が効率的になるような工夫が必要となる．

一つのアプローチは，ユーザがいくらでも効率の良いコードをかけるようにすることである．Torch.nn や Chainer では，本体に手を加えることなく，ユーザが演算ノードを自由に定義することができる．この点については次項にて改めて触れる．

もう一つのアプローチは，コンパイラのように計算グラフを最適化することである．Theano や CGT がこのアプローチによって，計算グラフを最適化する．GPU を用いる場合，最適化の際に重要となるのはメモリ使用

*1 http://neon.nervanasys.com/

量, 演算数, およびメモリ帯域である. 例えば, 複数のループを一つに統合することで, 大きな一時変数を確保せずに済ませることができ, また一連の計算を GPU コア内で完結させることでメモリとの通信量を節約できる.

計算グラフの実行を効率化するもう一つの方法に, 並列化がある. 計算グラフのうち, 依存関係のないパス同士は並列に実行することができる. ユーザにどのパスを並列実行するか記述させようとすると煩雑になりがちである. そこで, 計算グラフを解析して, 並列化できる部分を自動的に見つけ出しスケジューリングすることで, 記述を簡潔に保ちながら計算を並列化することができる. 例えば MXNet や TensorFlow はこの方法で計算の自動並列化を行う.

計算グラフの並列実行には, 単一 GPU 内での複数カーネルの並列実行, 複数 GPU での並列実行, そして複数マシンでの分散並列実行がある. これらは並列化の単位が異なるだけでなく, メモリの扱い方が異なる. 単一 GPU 内での並列化は, CPU におけるマルチスレッドプログラミングのように, メモリを共有しながら実行することができる. 一方, マシン内外の複数 GPU を用いる場合, メモリの内容を転送しながら協調して実行する必要がある. 特に複数マシンでの並列実行ではメモリの転送コストが非常に高い. そのため, 特にメモリの転送と別の計算を並列に実行することで転送コストを隠ぺいすることが重要になる.

ここまで主に計算自体についての最適化について述べたが, GPU における計算ではメモリの最適化も重要である. GPU の計算においてはメモリの確保・解放はすべての計算を同期してしまい, 非常にコストが高い. よって, 基本的にメモリの管理はフレームワークの仕事となる. 最適化を行わないフレームワークは基本的にすべての配列をあらかじめメモリ上に確保するが, 一時変数の中には計算のごく一部でしか利用しないものがある. 使用期間が重複しない一時変数同士で同じメモリ領域を共有することで, 最大メモリ消費量を抑えることができる. このようなメモリの共有は, 単純な実装ではメモリプールによって実現されるが, サイズが一致しない一時変数同士ではメモリ領域が再利用できないなど欠点もある. MXNet ではメモリ領域を最適化することで, より高速に大きなネットワークを学習することを可能としている.

学習スクリプトの実行時に最適化を行うアプローチは, 計算グラフをあらかじめ構築する場合と相性が良い. 一方で, 前述のとおり, データごとに計算グラフを構築したほうがコンパクトな枠組みで高い表現力を得ることができる. 将来的に, これらのアプローチを組み合わせたフレームワークが出てくれば, 柔軟でかつ効率的な実装ができるだろう.

3·4 演算ノードの粒度

フレームワーク設計のもう一つの観点として, どれくらいの粒度で演算ノードを定義するかという点がある. これは, 前節の最適化との兼合いによっても理想的な粒度が変わってくる.

計算グラフの最適化が強く効くフレームワークでは, 演算ノードをできるだけ細かい単位で用いる. 複雑な計算はすべて複雑な計算グラフとして記述しておき, 計算の効率はすべて最適化によって解決する. Theano ベースのフレームワークはおおむねこの考え方で構築されている.

一方, 最適化がない, もしくはあまり期待できないフレームワークでは, ある程度大きな処理のかたまりを一つの演算ノードとして定義し, その実装を工夫することで効率を上げる必要がある. Torch.nn や Chainer はこのアプローチで書かれている. この場合, フレームワーク本体に変更を加えることなく新しい演算ノードを定義できることが重要である. これは, ユーザが手でコードを効率化するコストを下げることにつながる.

演算ノードには順伝播と逆伝播の実装が必要である. これらは, Torch.nn では Lua の, Chainer では Python のメソッドとして記述できる. 特に Chainer の場合, CUDA カーネルを Python のインタフェースからコンパイル・実行することができるため, 効率的な演算ノードをユーザが実装する際の敷居が低い.

3·5 パラメータの扱い

前節にも関連することだが, 学習の対象となるパラメータを計算グラフ上でどう扱うかという点においても, フレームワークによって違いが見られる.

本稿で紹介した誤差逆伝播の定義に則れば, パラメータは入力と同様に計算グラフ上のデータノードとして書かれるべきであろう. Theano ベースのフレームワークや Chainer（v1.5 以降）はこの方式をとる.

もう一つ, より古典的なニューラルネットの定式化に近い方法として, 一部の演算ノードの中にパラメータを含めてしまう方法がある. 例えば, 全結合層という演算ノードを考えよう. 全結合層は, 入力に重み行列をかけてバイアスベクトルを足す, という処理である. 一般に, 重み行列とバイアスベクトルがパラメータとして用いられる. そこで, この方式では重み行列とバイアスベクトルが付随した演算ノードとして全結合層を定義し, 演算ノードとしては 1 入力 1 出力の関数として定義する. 誤差逆伝播の際には, 逆伝播の処理をしつつ, 各パラメータに対する勾配も副作用として計算, 蓄積する. このアプローチは, ニューラルネットの古典的な表現と類似しており, 理解しやすいという利点がある. 一方で, どの変数をパラメータとして扱うかをユーザが選べないため, 拡張性に乏しい. また, 演算ノードの処理が副作用を伴うことになるため, 最適化がしづらくなる. Torch.nn

やCaffeなど多くのフレームワークがこのような実装を行っている．Chainerもv1.4まではこの方式で実装されていた．

4. Chainerの設計思想

Chainerは Pythonで書かれたニューラルネットのフレームワークである．前章の分類に従うと，Chainerは以下のような特徴をもつ（それぞれ前章の各節に対応する）．

（1）Pythonによって計算グラフを順伝播時に動的に構築する．
（2）勾配は計算グラフを逆向きにたどることで実行する．
（3）計算の最適化は行わないが，ユーザが自由に演算ノードを定義できる．
（4）細かい粒度の演算ノードと，最適化された大きな演算ノードの両方を提供している．
（5）パラメータは変数ノードとして扱う．

本章では，Chainerの設計とその背景についてより詳しく述べる．

4·1 計算グラフとその処理系

Chainerの中身に入る前に，計算グラフについてもう少しだけ考える．

計算グラフとは，関数（演算ノード）を組み合わせてつくった計算過程をグラフの形で表したものであった．ここで，一つの関数にはいくらでも複雑なことをさせられる．どのような関数を用意するかによって，計算グラフの表現力は変わってくる．例えば，何らかの処理を系列データに対して反復適用するような演算ノード，というものを定義することもできる．実際にTheanoではこの操作がscanという名前で提供されており，可変長の入力に対するリカレントネットを実装するのに用いられる．

さて，計算グラフは計算過程をグラフとして表現するが，我々は普段，計算過程を記述するのにグラフは使わない．普通はプログラミング言語を使うだろう．ソースコードは例えば文の列として解釈され，インタプリタがそれを順に実行したり，コンパイラによって実行可能コードに変換される．ソースコードを解釈して実行したり，実行可能なコードを出力するプログラムは，プログラミング言語の処理系と呼ばれる．

計算グラフを用いたフレームワークは，グラフとして表現された計算過程を実行する処理系である．入力の形態こそ異なるが，本質的な役割はプログラミング言語の処理系とほとんど同じである．ただし，プログラミング言語の処理系が汎用的な操作の処理を目的としているのに対し，深層学習のフレームワークはニューラルネットを学習・実行することが主な目的である．

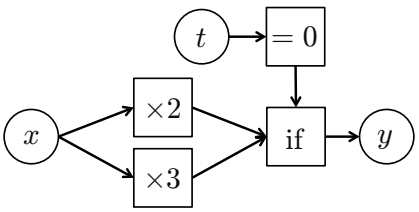

図4 計算前にグラフを構築する場合の，式（5）の計算グラフ

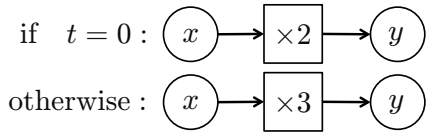

図5 計算時にグラフを構築する場合の，式（5）の計算グラフ

本稿でこれまで見てきたように，計算グラフの最も大きな目的は，誤差逆伝播の自動導出である．よって，深層学習のフレームワークは，少なくとも誤差逆伝播を表現する計算グラフの処理系を含まなければならない．それでは，順伝播についてはどうだろうか．

例えば次の関数を考えよう．

$$f(x,t)=\begin{cases} 2x & \text{if} \quad t=0 \\ 3x & \text{otherwise.} \end{cases} \tag{5}$$

この関数fは，tの値によって挙動を変えている．ここでは，フレームワークが事前に定義している演算ノードの組合せでこの関数fを表現することを考える．このとき，tの値が決まる前に計算グラフを構築しなければならない場合，条件分岐をグラフ上で表現しなければならない．例えば，真偽値によって二つの入力のうち一方を出力する関数ifを使うと，図4のように計算グラフで表すことができる．このグラフには，tの値によらず両方のパス（この場合，$2x$と$3x$）を実行してしまう欠点がある．むだな計算を省くには，計算グラフの処理自体を拡張しなければならない．ここまで述べてきた計算グラフには変数ノードと演算ノードしかなかったが，新しい仕掛けが必要になる．これはプログラミング言語に構文糖ではない本質的な機能を追加するようなものである．

では，計算グラフの構築を遅らせて，tの値が決まるのを待てばどうだろうか．この場合，tの値による条件分岐は計算グラフの外側，すなわちユーザが用いるプログラミング言語上で行うことができる．計算グラフは図5のように，tの値に応じて異なるものがつくられる．計算グラフに新たな仕掛けは必要なく，変数ノードと演算ノードの単純な組合せで済む．この計算グラフは一見関数fを表現できていないように見えるが，実際にはこれで誤差逆伝播が実行できる．なぜなら，関数fのヤコビ行列には，f自体の定義と全く同じ条件分岐が現れるからである．より一般に，勾配の計算に必要な情報は，入力に対して実際にどのような微分可能な操作がなされ

たかだけである．つまり，条件分岐を含めた計算の定義は必要なくて，実際に通った実行パスの情報だけが必要となる．計算グラフの構築を計算時まで遅らせることができれば，このようなプログラミング上の構造物は計算グラフに含める必要がなくなる．

Chainerはこの考え方に基づき，計算グラフを順伝播の計算時に動的に構築する．構築した計算グラフは，順伝播で行われた微分可能な演算の履歴を表している．Chainerの世界では，ニューラルネットはプログラムであり，Chainerはその実行履歴を管理する．こうすることで，条件分岐や反復処理をプログラミング言語の処理系にまかせて，コンパクトな設計でニューラルネットを自由に記述することができる．

ニューラルネットはプログラムであるという考え方は，ほかにも利点を生む．例えばデバッグについて考えよう．ニューラルネットの構成にバグがあり（例えば途中の計算で次元が合わなくなり），エラーが発生したとする．このとき，順伝播も計算グラフ上で実行するフレームワークでは，エラーのハンドリングもその処理系が行う必要がある．ユーザはベースとなるプログラミング言語におけるエラーハンドリングとは別に，そのフレームワークにおけるエラーの処理方法やエラーメッセージの読み方を習得しなければならない．一方，Chainerでは順伝播中のエラーは単純にPythonのスクリプトにおけるエラーなので，ユーザはPythonのエラー処理方法を知っていればそれで事足りる．Pythonが例外によって終了する場合，スタックトレースが出力されるが，Chainerではそのスタックトレースがまさにネットワークのどこで終了したかを示してくれる．

また，Chainerでは計算グラフは動的に構築されることが前提となっているため，計算の途中でグラフを編集することも可能である．編集されたグラフ上で誤差逆伝播を行うと，厳密な勾配計算ではなくなる．例えば，非常に長い系列に対するリカレントネットの学習において，打切り型誤差逆伝播（truncated backpropagation）を考える．これは，適当なステップ数sに対して，ある時刻に被った損失の逆伝播をsステップ前までだけ行い，それより以前の計算には逆伝播しないというものである．Chainerにはある変数から逆伝播で到達可能な部分グラフをグラフ上から削除するAPI（unchain backward）がある．このAPIを使いsステップより前の部分グラフを削除してから誤差逆伝播することで，打切り型誤差逆伝播が実装できる．

このようなフレームワークをつくった動機として，ニューラルネットのさらなる複雑化がある．深層学習が流行した当初は音声認識と画像認識における成功が著しく，多層パーセプトロンや畳込みネットなどフィードフォワードなニューラルネットが主な対象であった．分岐や反復は必要なく，単純な非線形関数の合成としてニューラルネットが書けたため，Caffeのように単純な仕組みで扱いやすく，高速に動作するフレームワークが人気を集めた．一方で，深層学習の主戦場はより複雑なモデルを必要とする問題に移りつつある．例えば自然言語処理では可変長の入出力系列を扱う必要があり，そこでは主にループを使ったリカレントネットが用いられる．画像認識の分野でも，注意のモデル化ではループ構造が用いられる．これらにとどまらず，ニューラルネットの研究では予測の過程を微分可能に保ちながらいかに事前知識を入れ込むかが重要になっており，その探求にはできるだけ自由かつ簡単に計算が記述できることが求められると考えたのである．

もう一つの側面として研究の加速がある．認識課題で優秀な成績を収めただけでなく，部品が組み合わせやすく，かつそのうえで全体最適化が可能であるという点が評価され，今や機械学習のさまざまな応用分野で新しいニューラルネットが提案されている．新しい手法がすさまじい頻度で提案されるようになり，最先端の研究についていくためには査読を待たずにプレプリントサーバに上げられた文書を読む必要がある．特にニューラルネットは実験してみないとわからない点が多い．そこで，このような技術を利用するには，素早く追試を行い有用性を確認して，自らの課題における試行錯誤のサイクルを多く回すことが成功への鍵となる．Chainerはこの点を意識してつくられており，特に柔軟性が重視されている．

動的にネットワークを構築するアプローチの欠点は，最適化の難しさである．あらかじめ計算グラフが決まっている場合と異なり，Chainerでは実行するまで計算グラフが定まらない．よって，計算を最適化するのが難しい．実際，Chainerは現状計算の自動最適化は行わない．最適化を行うにはどうしても事前にコードについての知識を得る必要がある．Pythonで書かれた順伝播コードからどうやって最適化を行うかは，今後の課題である．

4·2 モジュール化されたモデル定義

ニューラルネットの最も重要な特徴の一つとして，モジュール性がある．ニューラルネットは，非線形な関数を組み合わせてつくられる．この部品として用いられる非線形関数自体が，別のニューラルネットでもよいわけである．この点が，ニューラルネットによって表現されるモデルの豊かさを導いている．そこでニューラルネットのフレームワークに対しても，このようにネットワークを組み合わせて大きなモデルを構築する能力が求められる．

Chainerでは，Chainと呼ばれる仕組みを用いてモジュール化されたニューラルネットを記述する．Chainは，パラメータと順伝播コードを組み合わせるための仕掛けである．オブジェクト指向プログラミングにおいてデータと振舞いをひも付けるのと同じように捉えればよい．Chainはそれだけでなく，ほかのChainを中にもつことで，ニューラルネットを組み合わせてより大

きなニューラルネットをつくることを可能にする．例えば，次の例はデコーダとエンコーダとして外から任意のニューラルネットを与えられるセルフデコーダの実装である．

```
class AutoEncoder (Chain)：
  def ___ init __ (self, encoder, decoder) :
    super (AutoEncoder, self) . ___ init __ (
      encoder = encoder,
      decoder = decoder,
    )

  def __ call __ (self, x) :
    h = self .encoder (x)
    x _ hat = self .decoder (h)
    return F.mean _ squared _ error (x, x _ hat)
```

非常にシンプルな例だが，このAutoEncoderクラスは最適化を行うOptimizerとの連携やシリアライゼーションの機能，CPU/GPU間のパラメータ転送などを自動的にサポートする．encoderやdecoderの部分には，多層パーセプトロンや畳込みネットなど好きなChainを実装して使うことができる．F.mean_squared_errorというのは平均二乗誤差関数である．計算された二乗誤差は，これまで述べたどおり計算履歴を保持していて，誤差逆伝播を実行することができる．

モデル定義をモジュール化する一定の方法を提供することで，ユーザは定義したニューラルネットの断片を使い回し，ほかのユーザと共有することができる．

5. ま と め

本稿では，一般的な深層学習フレームワークの構成からその具体例であるChainerについて，設計における選択肢と背景を紹介した．深層学習の進歩は速く，研究開発するにあたっては素早い試行錯誤が欠かせない．また，画像認識などの技術はコモディティ化も始まっており，使いやすいインタフェースを用意することですそ野を広げることも重要である．

今，ニューラルネットのフレームワークがたくさん提案されている．その背景には，設計における多様な選択肢の存在がある．本稿がその理解を促し，目的に応じて適切なフレームワークを使い分ける助けになれば幸いである．

謝 辞

本稿をまとめるにあたって阿部 厳氏，岡野原大輔氏，大野健太氏，海野裕也氏の助けをいただいたので，ここに感謝の意を表す．

◇ 参 考 文 献 ◇

[Abadi 15] 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., Yangqing, J., 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.: Tensor Flow: Large-Scale Machine Learning on Heterogeneous Systems（2015）

[Bergstra 10] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D. and Bengio, Y.: Theano: a CPU and GPU Math Expression Compiler, *Proc. Python for Scientific Computing Conference*（*SciPy*）（2010）

[Collobert 11] Collobert, R., Kavukcuoglu, K. and Farabet, C.: Torch7: A Matlab-like environment for machine learning, *Big Learn, NIPS Workshop*（2011）

[Goodfellow 13] Goodfellow, I. J., Warde-Farley, D., Lamblin, P., Dumoulin,V., Mirza, M., Pascanu, R., Bergstra, J., Bastien, F. and Bengio, Y.: Pylearn2: A machine learning research library, arXiv preprint arXiv:1308.4214（2013）

[Jia 14] 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）

[Tokui 15] Tokui, S., Oono, K., Hido, S. and Clayton, J.: Chainer: A next-generation open source framework for deep learning, *Workshop on Machine Learning Systems at Neural Information Processing Systems*（*NIPS*）（2015）

2016年1月20日 受理

著 者 紹 介

得居 誠也

株式会社Preferred Networksリサーチャー．東京大学理学部数学科卒業，同大学院情報理工学系研究科数理情報学専攻にて修士号取得．2012年に株式会社Preferred Infrastructureに入社，2014年より現職．共著に「オンライン機械学習」（講談社，2015），「データサイエンティスト養成読本機械学習入門編」（技術評論社，2015）．