

R torch による深層学習と科学計算

Sigrid Keydana 著 榎本剛 訳

2024-12-02

Table of contents

第 I 部	torch に慣れる	11
第 1 章	概要	13
第 2 章	torch とその入手方法	15
2.1	torch の世界	15
2.2	torch のインストールと実行	16
第 3 章	テンソル	17
3.1	テンソルとは何か	17
3.2	テンソルの作成	18
3.2.1	値からテンソル	18
3.2.2	指定からテンソル	21
3.2.3	データセットからテンソル	22
3.3	テンソルに対する操作	27
3.3.1	集計	29
3.4	テンソルの部分参照	31
3.4.1	「R 思考」	31
3.4.2	R を越える	32
3.5	テンソルの変形	33
3.5.1	複製なし変形と複製あり変形	34
3.6	拡張	36
3.6.1	拡張のルール	38

第 4 章	自動微分	41
4.1	なぜ微分を計算するのか	41
4.2	自動微分の例	43
4.3	<code>torch autograd</code> による自動微分	44
第 5 章	<code>autograd</code> を使った関数の最小化	47
5.1	最適化の古典	47
5.2	最小化を白紙から	48
第 6 章	ニューラルネットワークを白紙から	53
6.1	ネットワークの考え方	53
6.2	ネットワークの層	54
6.3	活性化関数	55
6.4	損失関数	56
6.5	ネットワークの実装	56
6.5.1	ランダムデータの生成	57
6.5.2	ネットワークの構築	57
6.5.3	ネットワークの訓練	57
第 7 章	モジュール	59
7.1	組込の <code>nn_moudle()</code>	59
7.2	モデルの構築	62
7.2.1	層の列としてのモデル: <code>nn_sequential()</code>	62
7.2.2	独自処理のモデル	63
第 8 章	最適化器	65
8.1	何のための最適化器か	65
8.2	<code>torch</code> 組込最適化器の利用	66
8.3	パラメタの更新手法	67
8.3.1	勾配降下法	67
8.3.2	問題となること	69
8.3.3	軌道に留まる: 慣性付勾配降下法	70
8.3.4	Adagrad	71

8.3.5	RMSProp	72
8.3.6	Adam	73
第 9 章	損失関数	75
9.1	torch の損失関数	75
9.2	どの損失関数を選択すべきか。	76
9.2.1	最尤法	76
9.2.2	回帰	76
第 10 章	L-BFGS を用いた関数の最適化	77
10.1	L-BFGS 見参	77
10.1.1	変化する傾き	77
10.1.2	厳密ニュートン法	78
10.2	ニュートンの近似: BFGS と L-BFGS	78
10.2.1	直線探索	79
10.3	optim_lbfgs() を使ったローゼンブロッック関数の最小化	79
10.3.1	optim_lbfgs() の既定の動作	81
10.3.2	直線探索付 optim_lbfgs()	81
第 11 章	ニューラルネットワークのモジュール化	83
11.1	データ	83
11.2	ネットワーク	83
11.3	訓練	84
第 II 部	他にも torch でできること: 行列計算・フーリエ変換・ウェーブレット	85
第 12 章	概要	87
第 13 章	行列演算: 最小二乗問題	89
13.1	最小二乗を求める五つの方法	89
13.2	天気予報についての回帰	90
13.2.1	最小二乗法 (I): lm() による参照値	92
13.2.2	最小二乗法 (II): linalg_lstsq() の利用	93

13.2.3	一休み: 標準化しなかった場合	93
13.2.4	最小二乗法 (III): 正規方程式	97
13.2.5	最小二乗法 (IV): コレスキー分解	98
13.2.6	最小二乗法 (V): LU 分解	100
13.2.7	最小二乗法 (VI): QR 分解	101
13.2.8	最小二乗法 (VII): 特異値分解 (SVD)	102
13.2.9	実行時間の比較	103
13.3	簡単な数値安定性の確認	103
第 14 章	行列計算: 畳み込み	105
14.1	畳み込みの意義	105
14.2	一次元の畳み込み	106
14.3	畳み込みの二つの考え方	107
14.3.1	実装	109
14.4	二次元での畳み込み	110
14.4.1	仕組み (出力からの見方)	110
14.4.2	実装	112

List of Figures

4.1	仮想的な損失関数（放物面）。	42
4.2	計算グラフの例	43
5.1	ローゼンブロック関数	48
8.1	わずかに異なる学習率を用いた非等方的な放物面上の最急降下	69
8.2	慣性付 SGD（白）と素の SGD（灰）との比較	71
8.3	Adagrad（白）と素の SGD（灰）との比較	72
8.4	RMSProp（白）と素の SGD（灰）との比較	73
8.5	Adam（白）と素の SGD（灰）	74

はじめに

この本は `torch`、PyTorch の R インターフェースについて書かれている。PyTorch は執筆時点で、主要な機械学習と科学計算フレームワークの一つで、産業や研究の分野で広く使われている。`torch` を使うと、R から直接その豊富な機能を使うことができる。Python をインストールすることも、学ぶことすら必要ない。まだ **若い** プロジェクトだが、`torch` は既に活発なユーザと開発者のコミュニティがある。開発者はフレームワーク本体だけでなく、その上に自前のパッケージを構築している。

本書では、三つの部に対応して、三つの目的を達成することを目指している。

最初の目的は `torch` の基礎について丁寧に紹介することである。基礎とは、それなしには何も動作しない基礎的な構造を指す。先々では、できるだけ高度な文法構造を使うことになるが、基礎的な機能の動作や、概念について知ることは重要である。さらに、実用的な観点からは、必要以上の試行錯誤をしなくて済むように `torch` についてある程度慣れることは重要である。

第二部では、説明した基礎の上に、画像認識、時系列データ、表形式データ、音声分類など、様々な深層学習への応用を探索する。ここでも、概念的な説明に重点を置いている。さらに、各章では、自分のアプリケーションに使える「テンプレート」としてのアプローチを紹介している。適切な場合には、稀ではない「ビッグデータ、巨大モデル、大量計算」の方法とは一線を画して、専門知識の組み込みの重要性についても指摘している。

第三部は、`torch` でできるディープラーニング以外のことを紹介している。行列計算（例えば、線型回帰問題の解法）、離散フーリエ変換、ウェーブレット解析などである。ここで、本書の他のどの部分よりも、概念的な記述が非常に重要であると考えている。その理由について説明する。

一つには、読者の教育的背景が様々であると想定しているからである。R が自然科学や応用数学に近い分野で教えられ、使われるようになっているため、例えば、離散フーリエ変換がどのように動作するかなどの概念的な説明には、あまり恩恵を受けられないと感じる人もいるかもしれない。一方で、これらのことが未知の領域である可能性がある人もいるだろう。例えば、文学、文化研究、言語学など、伝統的に経験主義的でない背景を持つ人々である。もちろん、もし読者が後者に属している場合、本書の説明が概念に焦点を合わせているにもかかわらず、それでも非常に（または非常に）数学的であると感じるかもしれない。その場合でも、安心してほしい。確かに、（他の多くの有意義なことを理解すると同様に）数学的な概念を理解するには長い道のりが必要だが、人生をかけてゆっくり進めばよい。

二つ目に、深層学習が過去 10 年間のパラダイムであったにもかかわらず、最近の動向は、数学的または専門分野に根差した基盤への関心が再び高まっていることを示している（例えば、Bronstein et al. (2021) にある体系的に説明や *Beyond alchemy: A first look at geometric deep learning*^{*1}（錬金術を超えて – 幾何学的深層学習ことはじめ）が概念を紹介している幾何学的ディープラーニングアプローチ参照）。将来、深層学習技術と専門知識を統合する「ハイブリッド」アプローチがますます増えると予想される。フーリエ変換は消えない。

最後にこの話題に関して重要なことは、全ての章には `torch` のコードを付けたということと言及しておきたい。例えば、フーリエ変換の場合、専用の機能を使った公式の方法だけでなく、アルゴリズムを自分でコーディングする様々な方法も習得できる。自作のコードは、驚くほど少ない行数で、非常に印象的なパフォーマンスで実装できる。

簡単に言えば、これが本書から得られるものである。最後に、一つだけ言及しておきたいことがある。述べようとしたが、技術的な内容のため、本編ではあまり触れることができなかったことだ。私たちの社会では、機械/ディープラーニング（「AI」）の採用が増えるにつれ、政府や民間組織による悪用の機会も増えている。しばしば、悪意がなくても、結果は壊滅的になることがある。特に、少数派や既に不利な立場にあるグループにとっては、そのような結果は致命的である。そのため、現在の政治体制では不可避だが、利益を追求することが、最悪の場合疑問の余地のある機能（監視、すべての量的化）を持つ社会を生み出し、最も不公平、差別、深刻な害をもたらす可能性が高い。ここで、この問題に注意を向けることをお願いすべく、ブログ記事 *Starting to think about AI Fairness*^{*2}（AI 公平性についての考察の試み）を参照して、社会生活や自分の仕事や応用において、この問題に積極的に注意を払っていただきたい。

最後に、感謝の言葉で締めくくりたい。感謝すべき人々は多すぎるので、誰かを落としてしまうかもしれないが、それでも短く書くことにする。出版社である CRC Press（最初に、David Grubbs と Curtis Hill）には、執筆と編集のすべての段階で非常に快適なやり取りをしていただき、感謝している。そして、以下の人々に対し、この本に関連する支援とそれぞれの役割に対する特別な感謝を表明する。Daniel Falbel、`torch` の作成者兼メンテナー、この本を詳細にレビューし、多くの技術的問題を助けてくれた。Tracy Teal、私の上司で、すべての可能な方法で支援し、励ましてくれた。Posit（旧 RStudio）は、雇用主であり、このような仕事で生計を立てることを許してくれる。

Sigrid Keydana

^{*1} <https://blogs.rstudio.com/ai/posts/2021-08-26-geometric-deep-learning/>

^{*2} <https://blogs.rstudio.com/ai/posts/2021-07-15-ai-fairness/>

第 I 部

torch に慣れる

第 1 章

概要

本書は三部からなる。第二部と第三部は、それぞれ深層学習の様々な応用例と基本的な科学計算技術を調査する。その前に、第一部では `torch` の基本的な構成要素である、テンソルや自動微分、最適化法、モジュールについて学ぶ。この部分を「`torch` の基礎」、よくある雛型に従って「`torch` を始める」にすることもできたが、これらが誤った印象を与えると考えた。これらは確かに基礎ではあるが、基盤としての基礎である。これらの章を勉強すれば、`torch` がどのように動いているかしっかりとした概念が身につく、後の節に現れるより複雑な例をいじるのに支障がない程度までコードに馴染むことができる。言い換えれば、ある程度 `torch` に **堪能** になれる。

また、ニューラルネットワークを一度や二度、白紙から書くことになる。最初は、単なるテンソルそのものとそれが備える機能を使う次は、ニューラルネットワークの学習に不可欠な機能をオブジェクト指向でカプセル化した、`torch` の専用の構造を利用する。結果として、第二部に進む準備が周到に整い、そこで深層学習を様々な問題や分野に適用する。

第 2 章

torch とその入手方法

2.1 torch の世界

torch は PyTorch を R に移植したものである。PyTorch は、(執筆時点で) 産業や研究で最もよく使われている二つの深層学習フレームワークのうちの一つである。その設計から、様々な科学計算の問題（その一部を本書の最終部で扱う）で有用なすばらしい道具でもある。torch は R と C++（少しの C を含む）だけで書かれており、これを使うために Python をインストールする必要はない。

Python (PyTorch) の側では、エコシステムは同心円状になっている。中に PyTorch 自体、これなしでは何も動作しない中心ライブラリがある。これを取り囲むのは、フレームワークライブラリと呼ばれる内側の円があり、特別なデータの種類の種類に特化しているか、運用のような仕事の流れの問題を中心据えている。さらに、より広範なアドオンや特化したコード、ライブラリのエコシステムがあり、それは PyTorch を構成要素やツールとして使っている。

R の側では同一の「心臓」を用いている。全ては torch のコアに依存している。R でも類似のライブラリがあるが、カテゴリの「円」は互いに明確に定められておらず、境界ははっきりとしていない。活発な開発者のコミュニティがあり、出自や目的も様々だが、さらに torch を開発して拡張するために共同で活動し、より多くの人々がそれぞれの目的の達成を手助けしている。エコシステムは急速に成長しており、個々のパッケージに言及することは控えるが、torch のウェブサイト^{*1}を訪れば、その一部が掲示されている。

三つのパッケージについては、名前をあげ、本書で利用する。それらは torchvision、torchaudio と luz である。最初の二つは用途を特定した変換、深層学習モデル、データセット、それぞれ画像（動画を含む）及び音声データのユーティリティを集めたものである。三番目は torch の高水準で直感的、使いやすいインターフェースで、ニューラルネットワークの定義、訓練、評価がわずか数行でできる。torch 自体のように三つのパッケージは CRAN からインストールできる。

^{*1} <https://torch.mlverse.org/>

2.2 torch のインストールと実行

torch は Windows、macOS と Linux で利用できる。対応する GPU と必要な NVIDIA のソフトウェアがインストールされていれば、訓練されたモデルの種類次第で、かなりの高速化の恩恵を得られる。本書の全ての例は、CPU で実行できるものを選び、読者が辛抱強く待つ必要がないようにした。

適合性の問題は一時的なものであるため、本書ではここに記さない。同様に具体的なインストールの手順を羅列することも控える。いつでも最新情報は vignette^{*2}から得られる。問題や質問があれば、torch の GitHub リポジトリに遠慮なく issue を立ててほしい。

^{*2} <https://cran.r-project.org/web/packages/torch/vignettes/installation.html>

第 3 章

テンソル

3.1 テンソルとは何か

`torch` で何か役に立つことをするには、テンソルについて知る必要がある。数学や物理の意味のテンソルではない。TensorFlow や (Py-)Torch のような深層学習フレームワークでは、テンソルは「単なる」多次元配列で、CPU だけでなく、GPU や TPU のような専用の装置上での高速計算に最適化されたものだ。

実際、`torch` の `tensor` は、R の `array` に同様に任意の次元を取れる。R の `array` とは異なり、高速かつ大規模に計算を実行するために、GPU に移すことができる（おまけに、自動微分ができるので、大変有用だ）。

`tensor` は R6 オブジェクトに類似していて、\$によりフィールドやメソッドを利用できる。

```
torch_tensor
1
[ CPUFloatType{1} ]
```

これは単一の値 1 だけを格納したテンソルだ。CPU に「生息」しており、その型は `Float`。次に波括弧の中の `1{1}` に着目する。これはテンソルの値を改めて示したものではない。これはテンソルの形状、つまりそれが生息する空間と次元の長さである。Base R と同様にベクトルは単一の要素だけでもよい（base R は 1 と `c(1)` を区別しないことを思い出してほしい）。

前述の \$ 記法を使って、一つ一つ関連するフィールドを参照することで、個別に以上の属性を確認できる。

```
torch_Float

torch_device(type='cpu')

[1] 1
```

テンソルの `$to()` を使うと、メソッドいくつかの属性は直接変更できる。

```
torch_Int
```

```
torch_device(type='mps', index=0)
```

形状の変更はどのようにするのか。これは別途扱うに値する話題だが、手始めにいじってみることにする。値の変更なしに、この1次元の「ベクトルテンソル」を2次元の「行列テンソル」にできる。

```
[1] 1 1
```

概念的には、Rで1要素のベクトルや行列を作るのに似ている。

```
[1] 1
```

```
[,1]
```

```
[1,] 1
```

テンソルがどのようなものか分かったところで、いくつかのテンソルを作る方法について考えてみる。

3.2 テンソルの作成

既に見たテンソルを作る一つの方法は `torch_tensor()` を呼び出し、Rの値を渡すというものだった。この方法は多次元オブジェクトに適用でき、以下にいくつかの例を示す。

しかし、多くの異なる値を渡す必要があるときは効率が悪くなる。ありがたいことに、値が全て同一であるべき場合や、明示的なパターンに従うときに適用できる別の方法がある。この節ではこの技についても説明する。

3.2.1 値からテンソル

前の例では単一要素のベクトルを `torch_tensor()` に渡したが、より長いベクトルを同様に渡すことができる。

```
torch_tensor
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
[ CPULongType{5} ]
```

同様に規定のデバイスは CPU だが、最初から GPU/MPS に配置するテンソルを作成することもできる。

```
torch_tensor
 1
 2
 3
 4
 5
[ MPSLongType{5} ]
```

これまで作ってきたのはベクトル。行列、つまり 2 次元テンソルはどうやって作るのか。

R の行列を同様に渡せばよい。

```
torch_tensor
 1 2 3 4 5 6 7 8 9
[ CPULongType{1,9} ]
```

結果を見てほしい。1 から 9 までの数字は、列ごとに表示されている。これは意図通りかもしれないし、そうではないかもしれない。意図と異なる場合は `matrix()` に `byrow = TRUE` を渡せばよい。

```
torch_tensor
 1 2 3
 4 5 6
 7 8 9
[ CPULongType{3,3} ]
```

高次元のデータはどうするか。同様の方針に従って、配列を渡すことができる。

```
torch_tensor
(1,...) =
 1 13
 5 17
 9 21

(2,...) =
 2 14
 6 18
10 22

(3,...) =
```

```

3 15
7 19
11 23

```

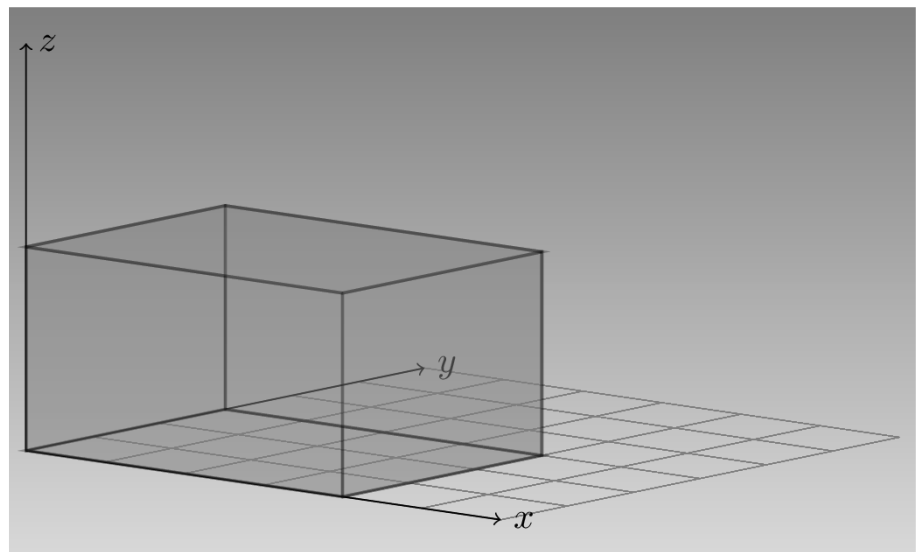
```

(4,.,.) =
4 16
8 20
12 24
[ CPULongType{4,3,2} ]

```

この場合でも、結果は R の埋め方に沿ったものとなる。これが求めるものではないなら、テンソルを構築するプログラムを書いた方が簡単かもしれない。

慌てる前に、その必要が非常に稀であることを考えてみてほしい。実際は、R のデータセットからテンソルを作ることがほとんどだ。「データセットからテンソル」の最後の小節で詳しく確認する。その前に、少し時間をとって最後の出力を少し吟味しよう。



渡したテンソルは以下のように印字される。

```

, , 1

      [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

```

```

      [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24

```

上のテンソルの印字と比較しよう。Array と tensor は異なる方向にオブジェクトを切っている。テンソルは値を 3x2 の上向きと奥に向かう広がる長方形に切り、4つの x のそれぞれの値に対して一つ示している。一方、配列は z の値で分割し、二つの奥向きと右向きに進む大きな 4x3 の部分を示す。

言い換えれば、テンソルは左/「外側」から、配列は右/「内側」から思考を始めているとも言えるだろう。

3.2.2 指定からテンソル

torch の大口生成関数が便利な状況は、おおまかに二つある。一つは、テンソルの個々の値は気にせず、分布のみに興味がある場合だ。もう一つは、ある一定のパターンに従う場合だ。

要素の値の代わりに、大口生成函数を使うときは、取るべき形状を指定する。例えば、3x3 のテンソルを生成し、標準正規分部の値で埋める場合は次のようにする。

```

torch_tensor
-0.0565  0.6605  0.6548
 0.2333 -1.4885 -0.2260
-1.3467 -0.3453 -0.0587
[ CPUFloatType{3,3} ]

```

次に示すのは、0 と 1 の間の一様分布に対する同様なもの。

```

torch_tensor
 0.4506  0.9286  0.7268
 0.1708  0.2191  0.9547
 0.1523  0.0266  0.3447
[ CPUFloatType{3,3} ]

```

全て 1 や 0 からなるテンソルが必要となることがよくある。

```

torch_tensor
 0  0  0  0  0
 0  0  0  0  0
[ CPUFloatType{2,5} ]

```

```
torch_tensor
  1  1
  1  1
[ CPUFloatType{2,2} ]
```

他にも多くの大口生成関数がある。最後に線型代数で一般的ないくつかの行列を作る方法を見ておく。これは単位行列。

```
torch_tensor
  1  0  0  0  0
  0  1  0  0  0
  0  0  1  0  0
  0  0  0  1  0
  0  0  0  0  1
[ CPUFloatType{5,5} ]
```

そしてこれは対角行列。

```
torch_tensor
  1  0  0
  0  2  0
  0  0  3
[ CPUFloatType{3,3} ]
```

3.2.3 データセットからテンソル

さて、R のデータセットからテンソルを作る方法を見ていこう。データセットによっては、この過程は「自動」であったり、考慮や操作が必要になったりする。

まず、base R についてくる `JohnsonJohnson` を試してみる。これは、Johnson & Johnson の一株あたりの四半期利益の時系列である。

	Qtr1	Qtr2	Qtr3	Qtr4
1960	0.71	0.63	0.85	0.44
1961	0.61	0.69	0.92	0.55
1962	0.72	0.77	0.92	0.60
1963	0.83	0.80	1.00	0.77
1964	0.92	1.00	1.24	1.00
1965	1.16	1.30	1.45	1.25
1966	1.26	1.38	1.86	1.56
1967	1.53	1.59	1.83	1.86
1968	1.53	2.07	2.34	2.25

1969	2.16	2.43	2.70	2.25
1970	2.79	3.42	3.69	3.60
1971	3.60	4.32	4.32	4.05
1972	4.86	5.04	5.04	4.41
1973	5.58	5.85	6.57	5.31
1974	6.03	6.39	6.93	5.85
1975	6.93	7.74	7.83	6.12
1976	7.74	8.91	8.28	6.84
1977	9.54	10.26	9.54	8.73
1978	11.88	12.06	12.15	8.91
1979	14.04	12.96	14.85	9.99
1980	16.20	14.67	16.02	11.61

`torch_tensor()` に渡すだけで、魔法のようにほしいものが手に入るだろうか。

`torch_tensor`

0.7100
0.6300
0.8500
0.4400
0.6100
0.6900
0.9200
0.5500
0.7200
0.7700
0.9200
0.6000
0.8300
0.8000
1.0000
0.7700
0.9200
1.0000
1.2400
1.0000
1.1600
1.3000
1.4500
1.2500
1.2600
1.3800

```

1.8600
1.5600
1.5300
1.5900
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{84} ]

```

うまくいっているようだ。値は希望通り四半期ごとに並んでいる。

魔法？ いや、そうではない。torch ができるのは与えられたものに対して動作することだ。ここでは、与えられたのは実は四半期順に並んだ double のベクトル。データは ts クラスなので、その通りに印字されたただけだ。

```

[1] 0.71 0.63 0.85 0.44 0.61 0.69 0.92 0.55 0.72 0.77 0.92 0.60
[13] 0.83 0.80 1.00 0.77 0.92 1.00 1.24 1.00 1.16 1.30 1.45 1.25
[25] 1.26 1.38 1.86 1.56 1.53 1.59 1.83 1.86 1.53 2.07 2.34 2.25
[37] 2.16 2.43 2.70 2.25 2.79 3.42 3.69 3.60 3.60 4.32 4.32 4.05
[49] 4.86 5.04 5.04 4.41 5.58 5.85 6.57 5.31 6.03 6.39 6.93 5.85
[61] 6.93 7.74 7.83 6.12 7.74 8.91 8.28 6.84 9.54 10.26 9.54 8.73
[73] 11.88 12.06 12.15 8.91 14.04 12.96 14.85 9.99 16.20 14.67 16.02 11.61
attr(,"tsp")
[1] 1960.00 1980.75 4.00

```

これはうまくいった。別なものを試そう。

```
[1] 35 3
```

```

Tree age circumference
1 1 118 30
2 1 484 58
3 1 664 87
4 1 1004 115
5 1 1231 120
6 1 1372 142

```

```
Error in torch_tensor_cpp(data, dtype, device, requires_grad, pin_memory): R type
```

どの型が処理されないのか。「元凶」は順序付き因子の列 Tree に違いないのは明らかだ。先に torch が因子を扱えるか確認する。

```

torch_tensor
1
2

```


3

[CPULongType{3}]

これは問題なく動作した。他に何がありうるか。ここでの問題は含まれている構造 `data.structure` である。`as.matrix()` を先に作用させる必要がある。でも、因子が存在するので、全て文字列の配列になってしまい、希望通りにならない。したがって、基礎となるレベル（整数）を抽出してから、`data.frame` から行列に変換する。

```
torch_tensor
  2   118   30
  2   484   58
  2   664   87
  2  1004  115
  2  1231  120
  2  1372  142
  2  1582  145
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{35,3} ]
```

同じことを別の `data.frame`、`modeldata` の `okc` でしてみよう。

Caution

`okc` は `modeldata` の 0.1.1 で廃止となり、0.1.2 以降は削除された。

	age	diet	height	location	date	Class
1	22	strictly anything	75	south san francisco	2012-06-28	other
2	35	mostly other	70	oakland	2012-06-29	other
3	38	anything	68	san francisco	2012-06-27	other
4	23	vegetarian	71	berkeley	2012-06-28	other
5	29	<NA>	66	san francisco	2012-06-27	other
6	29	mostly anything	67	san francisco	2012-06-29	stem

[1] 59855 6

二つある整数の列は問題なく、一つある因子の列の扱い方は学んだ。`character` と `date` の列はどうだろう。個別に `date` の列からテンソルを作ってみる。

```
torch_tensor
15519
15520
15518
15519
```

```

15518
15520
15516
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{59855} ]

```

これはエラーを投げなかったが、何を意味するのか。これらは R の `Date` に格納されている実際の値、つまり 1970 年 1 月 1 日からの日数である。すなわち、技術的には動作する変換だ。結果が実際に意味をなすかは、どのようにそれを使うつもりかという問題だ。言い換えれば、おそらく計算に使う前に、これらのデータを追加の処理する必要がある。どのようにするかは文脈次第。

次に `location` を見る。これは、`character` 型の列のうちの一つだ。そのまま `torch` に渡すとどうなるか。

```
Error in torch_tensor_cpp(data, dtype, device, requires_grad, pin_memory): R type
```

実際 `torch` には文字列を格納するテンソルはない。これらを `numeric` 型に変換する何らかの方法を適用する必要がある。この例のような場合、個々の観測が単一の実体（例えば文やパラグラフではなく）を含む場合、最も簡単な方法は R で `factor` に変換し、`numeric`、そして `tensor` にすることだ。

```

torch_tensor
120
74
102
10
102
102
102
102
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{59855} ]

```

確かに、技術的にはこれらうまく動作する。しかしながら、情報が失われる。例えば、最初と 3 番目の場所はそれぞれ "south san francisco" と "san francisco" だ。一度因子に変換されると、これらは意味の上で "san francisco" や他の場所と同じ距離になる。繰り返しになるが、これが重要かはデータの詳細と目的次第だ。これが重要なら、例えば、観測をある基準でまとめたり、緯度/経度に変換したりすることを含めさまざまな対応がありうる。これらの考慮は全く `torch` に特有ではないが、ここで述べたのは `torch` の「データ統合フロー」に影響するからだ

最後に実際のデータ科学の世界に挑むには、`NA` を無視するわけにはいかない。確認しよう。

```
torch_tensor
  1
nan
  3
[ CPUFloatType{3} ]
```

R の NA は NaN に変換された。これを扱えるだろうか。いくつかの torch のかんすうでは可能だ。例えば、`torch_nanquantile()` は単に NaN を無視する。

```
torch_tensor
  2
[ CPUFloatType{1} ]
```

ただし、ニューラルネットワークを訓練するなら、欠損値を意味のあるように置き換える方法を考える必要があるが、この話題は後回しにする。

3.3 テンソルに対する操作

テンソルに対する数学的操作は全て可能だ。和、差、積など。これらの操作は（`torch_`で始まる）関数や（\$記法で呼ぶ）オブジェクトに対するメソッドとして利用可能だ。次の二つは同じだ。

```
torch_tensor
  4
  6
[ CPUFloatType{2} ]
```

```
torch_tensor
  4
  6
[ CPUFloatType{2} ]
```

どちらも新しいオブジェクトが生成され、`t1` も `t2` も変更されない。オブジェクトをその場で変更する別のメソッドもある。

```
torch_tensor
  4
  6
[ CPUFloatType{2} ]
```

```
torch_tensor
  4
```

```
6
[ CPUFloatType{2} ]
```

実は、同じパターンは他の演算にも適用される。アンダスコアが後についているのを見たら、オブジェクトはその場で変号とされる。

当然、科学計算の場面では行列演算は特に重要だ。二つの一次元構造、つまりベクトルの内積から始める。

```
torch_tensor
32
[ CPULongType{} ]
```

これは動かないはずだと考えただろうか。テンソルの一つを転置 (`torch_t()`) する必要があるだろうか。これも動作する。

```
torch_tensor
32
[ CPULongType{} ]
```

最初の呼び出しも動いたのは、`torch` が行ベクトルと列ベクトルを区別しないからだ。結果として、`torch_matmul()` を使ってベクトルを行列にかけるときも、ベクトルの向きを心配する必要はない。

```
torch_tensor
14
32
50
68
[ CPULongType{4} ]
```

同じ関数 `torch_matmul()` は二つの行列をかけるときにも使う。これが `torch_multiply()` の動作、つまり引数のスカラ積とどのように異なるかよく見てほしい。

```
torch_tensor
4
10
18
[ CPULongType{3} ]
```

テンソル演算は他にも多数あり、勉強の途中、いくつかに出会うことになるか、特に述べておく必要な集まりが一つある。

3.3.1 集計

R 行列に対して和を計算する場合、それは次の三つのうちの一つを意味する。総和、行の和、もしくは列の和。これら三つを見てみよう（訳あって `apply()` を使う）。

```
[1] 126
```

```
[1] 21 42 63
```

```
[1] 6 12 18 24 30 36
```

それでは `torch` で同じことをする。総和から始める。

```
torch_tensor
126
[ CPULongType{} ]
```

行と列の和は面白くなる。`dim` 引数は `torch` にどの次元の和をとるか伝える。`dim = 1` を渡すと次のようになる。

```
torch_tensor
6
12
18
24
30
36
[ CPULongType{6} ]
```

予想外にも列の和になった。結論を導く前に、`dim = 2` だとどうなるか。

```
torch_tensor
21
42
63
[ CPULongType{3} ]
```

今度は行の和である。`torch` の次元の順序を誤解したのだろうか。そうではない。`torch` では、二つの次元があれば行が第一で列が第二である（すぐに示すように、添え字は R で一般的なのと同じで 1 から始まる）。

むしろ、概念の違いは集計にある。R における集計は、頭の中にあるものをよく特徴づけている。行（次元 1）ごとに集計して行のまとめを得て、列（次元 2）ごとに集計した列

のまとめを得る。`torch` では考え方が異なる。列（次元 2）を圧縮して行のまとめを計算し、行（次元 1）で列のまとめを得る。

同じ考え方がより高い次元に対しても適用される。例えば、4 人の時系列データを記録しているとする。二つの特徴量を 3 回計測する。再帰型ニューラルネットワーク（詳しくは後ほど）を訓練する場合、測定を次のように並べる。

- 次元 1: 個人に互る。
- 次元 2: 時刻に互る。
- 次元 3: 特徴に互る。

テンソルは次のようになる。

```
torch_tensor
(1,.,.) =
  0.3689  1.1017
 -0.8657 -0.2362
 -1.9824  0.4815

(2,.,.) =
 -1.1152 -0.7635
 -1.1341 -0.6355
  0.4823  0.2772

(3,.,.) =
  0.1911  0.3335
  0.2906 -0.5625
  0.7638  1.8530

(4,.,.) =
  0.9650  1.0710
  0.0767  1.0040
 -1.5226 -0.0091
[ CPUFloatType{4,3,2} ]
```

二つの特徴量についての平均は、対象と時刻に独立で、次元 1 と 2 を圧縮する。

```
torch_tensor
-0.2901
 0.3263
[ CPUFloatType{2} ]
```

一方、特徴量について平均を求めるが、各個人に対するものは次のように計算する。

```
torch_tensor
-0.8264  0.4490
-0.5890 -0.3739
 0.4151  0.5413
-0.1603  0.6887
[ CPUFloatType{4,2} ]
```

ここで圧縮されたは時刻である。

3.4 テンソルの部分参照

テンソルを使っていると、計算のある部分が入力テンソルの一部にのみに対する演算であることはよくある。その部分が単一の実体（値、行、列など）なら添字参照、このような実体の範囲なら切り出しと呼ばれる。

3.4.1 「R 思考」

添字参照も切り出しも基本的には R と同じように働く。いくつかの拡張された記法を続く節で示すが、総じてふるまいは直感に反しない。

なぜなら R と同じように、`torch` でも添字は 1 から始まるし、1 要素になった次元は落とされるからだ。

下の例では、2 次元テンソルの最初の行を求め、その結果 1 次元つまりベクトルを得る。

```
torch_tensor
1
2
3
[ CPULongType{3} ]
```

ただし、`drop = FALSE` を指定すると次元は保持される。

```
torch_tensor
1 2 3
[ CPULongType{1,3} ]
```

切り出しの時は、1 要素となる次元はないので、他に考慮すべきことはない。

```
torch_tensor
(1,...) =
 0.6499  0.9941
 0.4647  0.7137
```

```
(2,...) =
  0.6969  0.6480
  0.4530  0.6746
[ CPUFloatType{2,2,2} ]
```

まとめると、添字参照と切り出しはほぼ R と同じように働く。次に、前に述べた、さらに使いやすくする拡張について見る。

3.4.2 R を越える

拡張の一つはテンソルの最後の要素の参照だ。利便性のため、torch では-1 を使ってそれができる。

```
torch_tensor
4
[ CPULongType{} ]
```

注意すべきは、R では負の添字はかなり異なった効果を持ち、対応する位置の要素は取り除かれることだ。

もう一つの便利な機能は、切り出しの記法で刻み幅を二つ目のコロンの後に指定できることだ。ここでは、一つ目から八つ目の列を一つおきに取り出している。

```
torch_tensor
  1   3   5   7
 11  13  15  17
[ CPULongType{2,4} ]
```

最後に示すのは、同じコードを異なる次元のテンソルに対して動作させる方法だ。この場合、.. を使って明示的に参照されていない、存在する次元全てをまとめて指定できる。

例えば、行列、配列、もしくは高次元の構造など、どんなテンソルが渡されても最初の次元の添字参照をしたいとする。次の

```
t[1, ..]
```

は全てに対して機能する。

```
torch_tensor
-0.2555
  0.8151
[ CPUFloatType{2} ]
```



```
torch_tensor
-0.2098  0.8114
-0.9975 -1.2008
[ CPUFloatType{2,2} ]
```

```
torch_tensor
(1,...) =
  0.01 *
    6.9539 -98.7315
   -19.1422 -41.5001

(2,...) =
  0.2623  0.8591
 -0.5575  0.3014
[ CPUFloatType{2,2,2} ]
```

最後の次元の添字参照がしたければ、代わりに `t[..., 1]` と書けばよい。両方を組み合わせることもできる。

```
torch_tensor
-0.9873 -0.4150
  0.8591  0.3014
[ CPUFloatType{2,2} ]
```

次の話題は、添字参照や切り出しと同じくらい重要なテンソルの変形だ。

3.5 テンソルの変形

24 要素のテンソルがあるとする。形状はどうなっているか。次の可能性がある。

- 長さ 24 のベクトル
- 24×1 、 12×2 、 6×4 などの行列
- $24 \times 1 \times 1$ 、 $12 \times 2 \times 1$ などの 3 次元配列
- その他 ($24 \times 1 \times 1 \times 1 \times 1$ という可能性もありうる)

値をお手玉しなくても、`view()` メソッドでテンソルの形状を変更できる。最初のテンソルは長さ 24 のベクトルとする。

```
torch_tensor
0
0
0
```

```
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{24} ]
```

同じベクトルを横長の行列に変形する。

新しいテンソル `t2` を得たが、興味深いことに（そして性能の上で重要なことに）、`torch` はその値に対して新たに記憶域を割り付ける必要がなかったことだ。自分で確認することができる。二つのテンソルはデータを同じ場所に格納している。

```
[1] "0x11eb73f00"
```

```
[1] "0x11eb73f00"
```

どのように実現されているか少し議論する。

3.5.1 複製なし変形と複製あり変形

`torch` にテンソルの変形をさせると、テンソルの中身に対して新たな記憶域を割り付けずに要求を達成しようとする。これが実現可能なのは、同じデータ、究極的には同じバイト列は異なる方法で読み出すことができるからだ。必要なのはメタデータの記憶域だけだ。

`torch` はどのようにしているか。具体的な例を見てみる。3 x 5 行列から始める。

```
torch_tensor
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
[ CPULongType{3,5} ]
```

テンソルには `stride()` メソッドがあり、各次元に対して次の要素にたどり着くまでにいくつの要素を越えたか追跡する。上記のテンソル `t` に対して、次の行に進むには5要素飛ばす必要があるが、次の列には一つだけ飛ばせばよい。

```
[1] 5 1
```

ここで、テンソルを変形して、今度は5行3列にする。データ自体は変化しないことを思い出してほしい。

```
torch_tensor
  1   2   3
  4   5   6
  7   8   9
 10  11  12
 13  14  15
```

```
[ CPULongType{5,3} ]
```

今回は次の行には、5 要素ではなく 3 要素だけ飛ばせば次の行に到達する。次の列に進むのは、ここでも 1 要素だけ「跳べ」ばよい。

```
[1] 3 1
```

ここで、要素の順序を変えることができるか考えてみよう。例えば、行列の転置はメタデータの方法で可能だろうか。

```
torch_tensor
  1   6  11
  2   7  12
  3   8  13
  4   9  14
  5  10  15
[ CPULongType{5,3} ]
```

元のテンソルとその転置はメモリ上の同じ場所を指しているので、実際に可能であるはずだ。

```
[1] "0x11ebaf240"
```

```
[1] "0x11ebaf240"
```

これは道理にかなっている。次の行に到達するのに、1 要素だけ跳び、次の列には 5 要素跳べばよいから、うまくいくだろう。確認する。

```
[1] 1 5
```

その通りだ。

可能な限り、`torch` は形状を変更する演算をこの方法で扱おうとする。

このような（今後多数見ることになる）複製なし演算の一つは `squeeze()` とその対義語 `unsqueeze()` だ。後者は指定位置に単一要素の次元を付け加え、前者は取り除く。例を挙げる。

```
torch_tensor
-0.5961
-0.4916
 0.5372
[ CPUFloatType{3} ]
```

```
torch_tensor
```

```
-0.5961 -0.4916  0.5372
[ CPUFloatType{1,3} ]
```

ここでは単一要素の次元を前につけた。代わりに、`t$unsqueeze(2)` を使えば末尾につけることもできた。

さて、複製なしの技法は失敗することがあるか。そのような例を示す。

```
Error in (function (self, size) :
view size is not compatible with input tensor's size and
stride (at least one dimension spans across two contiguous subspaces).
Use .reshape(...) instead.
```

ストライドを変える演算を連続で行うと、二つ目は失敗する可能性が高い。失敗するかどうか決める方法はあるが、簡単な方法は `view()` の代わりに `reshape()` を使うことだ。後者は魔法のように機能し、可能であればメタデータで、そうでなければ複製する。

```
[1] "0x125e602c0"
```

```
[1] "0x125e5da40"
```

想像通り、二つのテンソルは今度は異なる場所に格納されている。

この長い章の終わりに取り上げる内容は、一見手に余るように見える機能だが、性能の上で極めて重要なものである。多くのもののように、慣れるには時間が掛かるが、安心してほしい。この本や `torch` を使った多くのプロジェクトでたびたび目にするようになる。この機能 **拡張** (ブロードキャスト) と呼ばれている。

3.6 拡張

形状が厳密に一致しないテンソルに対する演算をすることが多い。

もちろん、長さ 2 のベクトルに長さ 5 のベクトルを足すようなことはしないかもしれない。でもやってみたいこともありうる。例えば、全ての要素にスカラを掛けることがあるが、これはできる。

```
torch_tensor
 0.0275  0.6155 -0.3108 -0.2054  0.1803
 0.2123  0.2620  0.9127 -0.2798  1.1626
-0.3124 -0.8470 -0.3082  0.5310 -0.1626
[ CPUFloatType{3,5} ]
```

これはおそらく大したことではなかっただろう。R で慣れている。しかし、次は R では動作しない。同じベクトルを行列の全ての行に加えようとしている。

m2 をベクトルに代えてもうまくいかない。

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    2    6    5    9    8
[2,]    8   12   11   10   14
[3,]   14   13   17   16   20
```

文法としては動いたが、意味の上では意図した通りではない。

ここで、上の二つを `torch` で試してみる。まず二つのテンソルが 2 次元の場合（概念的には一つは行ベクトルだが）から。

```
[1] 3 5
```

```
[1] 1 5
```

```
torch_tensor
  2  4  6  8 10
  7  9 11 13 15
12 14 16 18 20
[ CPULongType{3,5} ]
```

次に足すものが 1 次元テンソルの場合。

```
[1] 5
```

```
torch_tensor
  2  4  6  8 10
  7  9 11 13 15
12 14 16 18 20
[ CPULongType{3,5} ]
```

`torch` ではどちらも意図通りにうまくいった。理由を考えてみよう。上の例でテンソルの形状をあえて印字した。3 x 5 のテンソルには、形状 3 のテンソルも形状 1 x 5 のテンソルも足すことができた。これらは、拡張がどのようにされるか示している。簡単にいうと、起きたのは次の通りだ。

1. 1 x 5 テンソルが加数として使われると、実際は拡張される。つまり、同じ 3 行あるかのように扱われる。このような拡張は一致しない次元が単一で一番左にある場合にのみ実行される。
2. 形状 3 のテンソルも同様だが、先に手順が追加される。大きさが 1 の主要な次元が実質上左に追加される。これにより 1 と同様になり、そこから手順が続く。

重要なのは、物理的な拡張はされないことだ。

ルールを系統だててことにする。

3.6.1 拡張のルール

ルールは次の通り。まず、一つ目は目を引くものではないか、全ての基礎になる。

1. テンソルの形状を右に揃える。

二つのテンソル、一つのサイズは $3 \times 7 \times 1$ 、もう一つは 1×5 、があるとする。これらを右に揃える。

t1, 形状: 3 7 1
t2, 形状: 1 5

2. 右から始めて、揃えた軸に沿う大きさが厳密に一致するか、一つが1でなければならない。後者の場合、単一要素次元のテンソルは単一でないものに対して **拡張**される。

上の例では、拡張は各テンソルに1回ずつ2回発生する。結果は実質的に以下のようになる。

t1, 形状: 3 7 5
t2, 形状: 7 5

3. もしテンソルの一つが一つ（もしくは1以上）余分な軸があれば、実質的に拡張される。

t1, 形状: 3 7 5 t2, 形状: 1 7 5

そして拡張が発生する。

t1, 形状: 3 7 5
t2, 形状: 3 7 5

この例では、拡張が両方のテンソルに同時に発生していることを見た。覚えておくべきことは、常に右から見るということだ。次の例は、どんな拡張をしてもうまくいかない。

```
torch_zeros(4, 3, 2, 1)$add(torch_ones(4, 3, 2)) # error
```

おそらく、この本の中で、この章は最も長く、最も応用から離れたように見えるものだった。しかし、テンソルに慣れることは、`torch` をすらすら書くための前提であると言っておく。同様のことは次の章で扱う話題、自動微分についても言える。違いは、`torch` が大

変な仕事を我々の代わりにしてくれるということだ。我々は何をしているか理解すればよいだけだ。

第 4 章

自動微分

前の章では、テンソルをどのように扱うかを学び、それに対して行うことができる数学的な演算の例を示した。そのような演算が多数あったとしても、それらが主要な `torch` の全てだったら、この本は読む必要がない。`torch` のようなフレームワークの人気は、それらを使ってできること、一般的には深層学習、機械学習、最適化、大規模科学計算にある。これらの応用分野の多くは、なんらかの損失関数を最小化と関係している。これは、さらに関数の **微分** の計算を必要とする。ここで、利用者として、個々の微分の関数形を自分で指定しなくてはならないということを想像してみよう。特にニューラルネットワークでは、すぐに面倒になるだろう。

実は、`torch` は微分の関数表現を作ったり、保存したりしない。代わりに、**自動微分** と呼ばれるものを実装している。自動微分では、より具体的には逆モード形では、微分はテンソル演算のグラフを逆向き走査で計算され、結合される。この後すぐに例を示すが、その前に一步引いてなぜ微分を計算する必要があるのか、簡単に議論しておこう。

4.1 なぜ微分を計算するのか

教師あり機械学習では、訓練集合が使えて、予測したい変数は既知である。これが目的変数で真値である。今予測アルゴリズムを開発し、これを入力変数、予測変数に基づいて訓練する。この訓練あるいは学習過程は、アルゴリズムの予測と真値とを比べ、現在の予測がどれくらいよいか悪いか捉える数値が出てくるような比較に基づいている。この数値を与えるのは、**損失関数** の仕事だ。

一度現在の損失が分かったら、アルゴリズムはパラメタ、つまりニューラルネットワークの重みを調整して、もっとよい予測にする。アルゴリズムはどの方向に調整するか知る必要がある。この情報は、**勾配** つまり微分のベクトルから得られる。

例として次のような損失関数を想像してみる Figure 4.1。

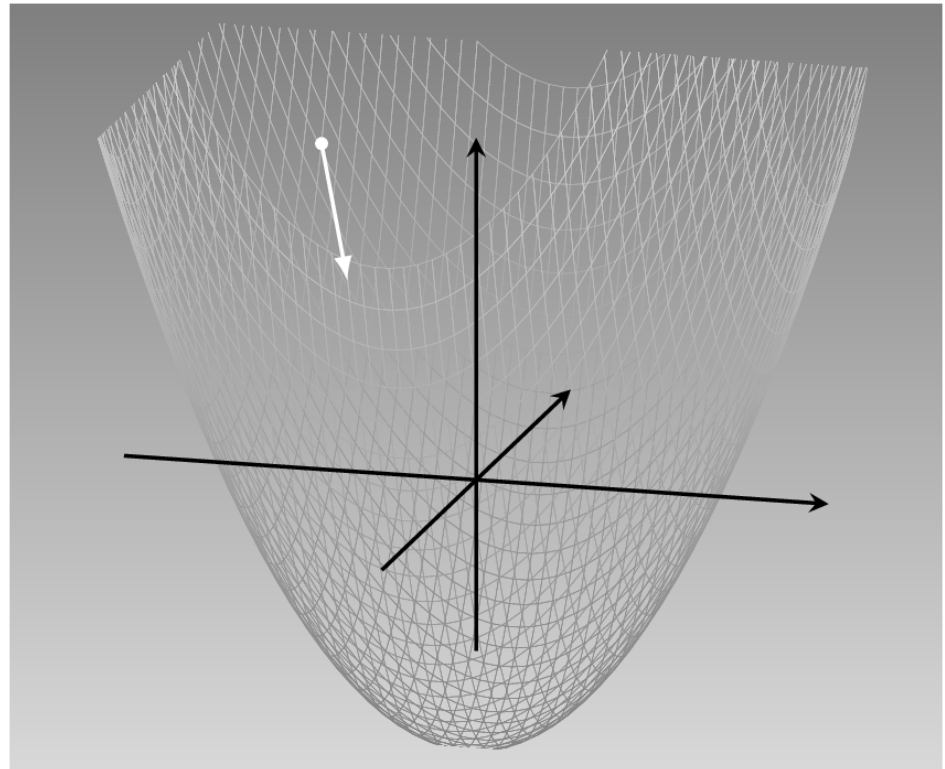


Figure4.1: 仮想的な損失関数（放物面）。

これは二変数の二次関数 $f(x_1, x_2) = 0.2x_1^2 + 0.2x_2^2 - 5$ である。最小値は $(0,0)$ で、この点を求める。白い点で示した点に立ち、風景を眺めれば、坂を速く降る方法は明確に分かる（坂を下るのを恐れないとする）。でも、最良の方向を計算で見つけるには、勾配を計算する。

x_1 の方向を取り上げる。 x_1 に関する関数の微分は、函数値が x_1 とともにどのように変化するかを示す。計算すると $\partial f / \partial x_1 = 0.4x_1$ となる。これは x_1 が増えると損失が増えることと、それがどの程度かを示している。でも損失を減らす必要があるので、逆方向に進む必要がある。

同じことが x_2 軸に対しても成り立つ。微分を計算すると、 $\partial f / \partial x_2 = 0.4x_2$ を得る。再び、微分が示す向きと逆方向を選ぶ。全体では、降下方向は

$$\begin{bmatrix} -0.4x_1 \\ -0.4x_2 \end{bmatrix}$$

である。

この方法是最急降下と呼ばれている。一般的に **勾配降下** と呼ばれ、機械学習で最も基本的な最適化アルゴリズムである。おそらく直感に反して、最も効率の良い方法ではない。さらに別の問いがある。出発点で計算されたこの方向は降下中にずっと最適なのか。代わりに、定期的に方向を計算し直した方が良いのかもしれない。このような質問は後の章で検討する。

4.2 自動微分の例

微分がなぜ必要か分かったところで、自動微分（AD: automatic differentiation）がどのように計算しているか見てみよう。

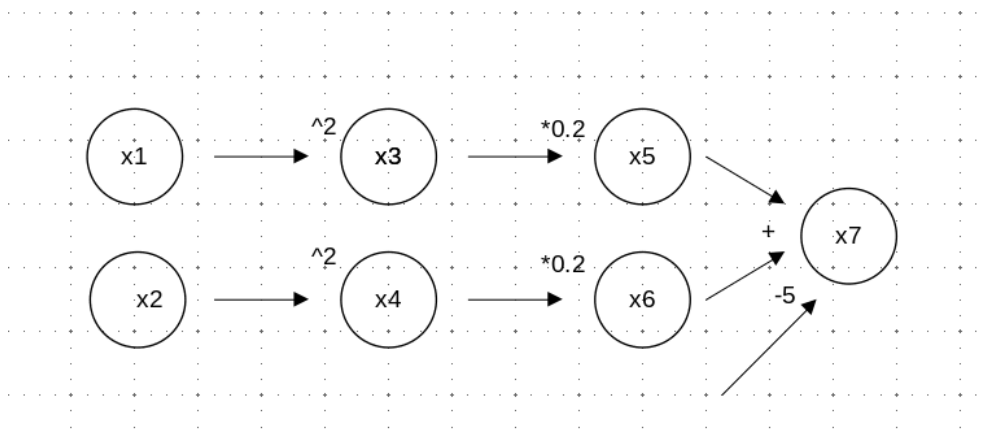


Figure4.2: 計算グラフの例

Figure 4.2 は上の関数が計算グラフにどのように表すことができるかを示している。 x_1 と x_2 は入力ノードで、対応する関数のパラメタは x_1 と x_2 である。 x_7 は関数の出力で、他は全て中間ノードであり正しい順序で実行するために必要である（定数-5、0.2 及び 2 をノードとすることもできるが、定数なので特に気にせずに、簡潔なグラフを選んだ）。

逆モード AD は、`torch` が実装している自動微分の種類で、まず関数の出力を計算する。これはグラフの順方向伝播である。次に逆伝播を行い、両方の入力 x_1 と x_2 に関する出力の勾配を計算する。この過程で、右から情報が利用可能となり、積み重なっていく。

- x_7 で、 x_5 と x_6 に関する偏微分を計算する。つまり、微分する式は $f(x_5, x_6) = x_5 + x_6 - 5$ なので、偏微分は両方とも 1 である。
- x_5 から左に動き、 x_3 にどのように依存しているか確認すると、 $\partial x_5 / \partial x_3 = 0.2$ である。微積分の連鎖律を用いると、出力がどのように x_3 に依存するか分かるので、 $\partial f / \partial x_3 = 0.2 \times 1 = 0.2$ と計算できる。
- x_3 から、 x_1 に最後の段階を踏む。 $\partial x_3 / \partial x_1 = 2x_1$ なので、連鎖律を再度用いて関数が最初の入力にどのように依存するか定式化できる。つまり $\partial f / \partial x_1 = 2x_1 \times 0.2 \times 0.1 = 0.4x_1$ となる。
- 同様に二番目の偏微分も計算し、勾配を求める。 $\nabla f = (\partial f / \partial x_1, \partial f / \partial x_2)^T = (0.4x_1, 0.4x_2)^T$

これが原理である。実際には、フレームワークによって逆モード自動微分の実装は異なる。次の節で `torch` がどのように実装しているか簡潔に示す。

4.3 torch autograd による自動微分

まず、用語について注意しておく。torchではADエンジンは *autograd* と呼ばれ、本書の残りの多くの部分でもそのように記す。それでは説明に戻る。

上述の計算グラフを torch で構築するには、入力テンソル x_1 と x_2 を作成する。これは興味のあるパラメータを模している。これまでしてきたように「いつも通り」テンソルを作成すると、torchはAD向けの準備をしない。そうせずに、これらのテンソルを作るときに `requires_grad = TRUE` を渡す必要がある。

(ところで二つのテンソルに 2 という値を選んだのは完全に任意である。)

次に「隠れた」ノード x_3 と x_6 を作るには、二乗して掛け算をする。最後に x_7 に最終出力を格納する。

```
torch_tensor
-3.4000
[ CPUFloatType{1} ][ grad_fn = <SubBackward1> ]
```

`requires_grad = TRUE` を追加しなければならなかったのは、入力テンソルを作るときだけであることに注目してほしい。グラフの依存するノードは全てこの属性を継承する。確認してみよう。

```
[1] TRUE
```

これまでに自動微分が動作するために必要な前提が全て満たされた。あとは `backward()` を呼べば、 x_7 が x_1 と x_2 にどのように依存するかが決まる。

この呼び出しにより、 x_1 と x_2 の `$grad` フィールドが埋まる。

```
torch_tensor
0.8000
[ CPUFloatType{1} ]
```

```
torch_tensor
0.8000
[ CPUFloatType{1} ]
```

これらは、それぞれ x_7 の x_1 と x_2 に関する偏微分である。上記の手計算を確認すると、どちらも 0.8 つまり 0.4 にテンソル値 2 及び 2 をかけたものになっている。

すでに述べた、端から端の微分を積み上げるのに必要な積算過程はどうなっているのか。積み上げられるに従って端から端までの微分を「追跡」することはできるのだろうか。例えば、最終出力がどのように x_3 に依存しているか見ることはできるだろうか。

```
torch_tensor
[ Tensor (undefined) ]
```

このフィールドは埋まっていないようである。実は、これらを計算することは必要だが、torch は不要になったら中間集計を捨て、メモリを節約する。しかしながら、保存する `retain_grad = TRUE` を渡して保存を指示することも可能だ。

それでは、`x3` の `grad` フィールドが埋まっているか確認してみよう。

```
torch_tensor
  0.2000
[ CPUFloatType{1} ]
```

`x4`、`x5`、`x6` についても同様だ。

```
torch_tensor
  0.2000
[ CPUFloatType{1} ]
```

```
torch_tensor
  1
[ CPUFloatType{1} ]
```

```
torch_tensor
  1
[ CPUFloatType{1} ]
```

もう一つ気になることがある。勾配の蓄積過程を「実行中の勾配」の観点から理解したが、蓄積を進めるのに必要な個々の微分はどのように計算されるのか。例えば `x3$grad` が示しているのは出力が中間状態 `x3` にどのように依存しているかであるが、ここから実際の入力ノードである `x1` にどのように到達するのか。

この面についても、確認できる。順伝播で torch はすべきことを書き残しておいて、後で個々の微分を計算する。この「レシピ」はテンソルの `grad_fn` フィールドに格納される。これが `x3` に対して `x1` への「失われたつながり」を追加する。

```
PowBackward0
```

`x4`、`x5`、`x6` についても同様。

```
PowBackward0
```

```
MulBackward1
```

```
MulBackward1
```

これでおしまい。この節では、`torch` がどのように微分を計算するかを見た上で、それをどのように行っているかの概要を示した。ここで、自動微分を応用した最初の二つの課題に取り組む準備が整った。

第 5 章

autograd を使った関数の最小化

前の二つの章ではテンソルと自動微分について学んだ。これから二つの章では `torch` の仕組みについての勉強は休んで、その代わりすでに学んだことで何ができるか試してよう。テンソルだけを使い、自動微分だけの力を借りるだけで、既に二つのことができる。

- 関数を最小化する（つまり、数値最適化を行う）、そして
- ニューラルネットワークを構築して訓練する。

この章では、最小化からはじめ、ネットワークは次章に回す。

5.1 最適化の古典

最適化研究において、**ローゼンブロッック関数** は古典である。この関数は二つの変数を取り、 $(1, 1)$ で最小となる。等値線を眺めると、最小は伸びた細い谷の中にあることが分かる。

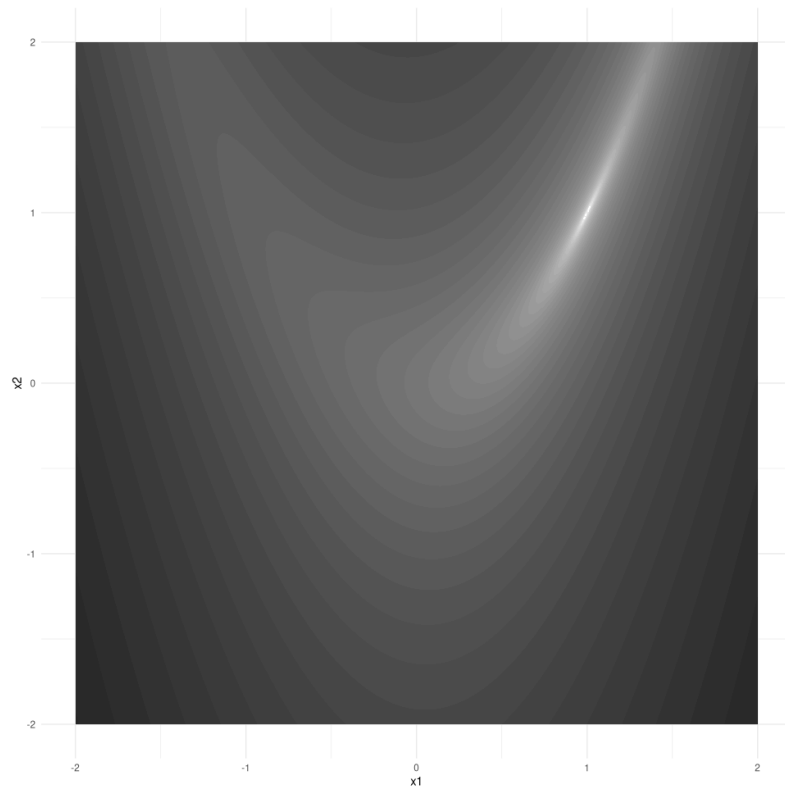


Figure5.1: ローゼンブロッック関数

関数の定義は次の通りだ。a と b は自由に定めてよいパラメタだが、よく使われる値を用いる。

5.2 最小化を白紙から

シナリオは次の通り。与えられた点 (x_1 , x_2) から出発し、ローゼンブロッック関数が最小になる場所を見つける。

前の章で説明した方法に従い、現在の位置における関数の勾配を計算し、それを使って逆方向に進む。どれくらい遠くまで行けばよいかは分からない。大きく進みすぎると、簡単に行き過ぎてしまう。(等値線図を再度確認すると最小値の西または東の急な崖に立っていると、これがすぐに生じることが分かる。)

つまり、最良の方法は反復して進むことで、妥当な幅を取り、毎回勾配を再評価することだ。

まとめると、最適化の手順は次のようになる。

```
library(torch)
```

```
# これはまだ正しい手順ではない!
```



```
for (i in 1:num_iterations) {  
  
  # 関数を呼び現在のパラメタ値を渡す。  
  value <- rosenbrock(x)  
  
  # パラメタについての勾配を計算する。  
  value$backward()  
  
  # 手動でパラメタを更新し、勾配に比例した一部を引く。  
  # ここはまだ正しくない。  
  x$sub_(lr * x$grad)  
}
```

書かれている通り、コード片は考えを示したもので、(まだ) 正しくない。また、いくつかの必要なものが欠けている。テンソル `x` も変数 `lr` や `num_iterations` も定義されていない。まず、これらを準備しよう。学習率 `lr` は毎回引く勾配に比例した一部で、`num_iterations` は反復する回数である。これらは実験パラメタである。

```
lr <- 0.01
```

```
num_iterations <- 1000
```

`x` は最適化するパラメタ、つまり関数の入力であり、最適化の最後に可能な限り最小の函数値に近い値を与える位置であることが望まれる。このテンソルについて関数の微分を計算するので、`requires_grad = TRUE` をつけて作る必要がある。

```
x <- torch_tensor(c(-1, 1), requires_grad = TRUE)
```

初期位置 `(-1, 1)` は任意に選択した。さて、残っているのは最適化ループを少し修正することだ。`autograd` が `x` について有効化されていると、`torch` はこのテンソルに対して行われるすべての演算を記録する。そのため `backward()` を呼ぶたびに、すべての必要な微分を計算しようとすることになる。しかし、勾配の一部を引くときは、微分を計算する必要はない。`torch` にこれを記録しないように指示するために、`with_no_grad()` を囲む。

ここで説明しておかなければならないことがある。既定で `torch` は `grad` フィールドに格納された勾配を蓄積する。新しい計算をするたびに `grad$zero_()` を使ってゼロに消去する必要がある。

これらを考慮すると、パラメタの更新は次のように書ける。

```
with_no_grad({  
  x$sub_(lr * x$grad)  
  x$grad$zero_()  
})
```

完成したコードは次のようになる。ログをとる文を追加して何が起きているか分かるようにしてある。

```
Iteration: 100
Value is : 0.3502924
Gradient is : -0.667685 -0.5771312
Iteration: 200
Value is : 0.07398106
Gradient is : -0.1603189 -0.2532476
Iteration: 300
Value is : 0.02483024
Gradient is : -0.07679074 -0.1373911
Iteration: 400
Value is : 0.009619333
Gradient is : -0.04347242 -0.08254051
Iteration: 500
Value is : 0.003990697
Gradient is : -0.02652063 -0.05206227
Iteration: 600
Value is : 0.001719962
Gradient is : -0.01683905 -0.03373682
Iteration: 700
Value is : 0.0007584976
Gradient is : -0.01095017 -0.02221584
Iteration: 800
Value is : 0.0003393509
Gradient is : -0.007221781 -0.01477957
Iteration: 900
Value is : 0.0001532408
Gradient is : -0.004811743 -0.009894371
Iteration: 1000
Value is : 6.962555e-05
Gradient is : -0.003222887 -0.006653666
```

1000 回の反復後、関数値は 0.0001 よりも小さくなった。対応する (x1,x2) の位置はどこになったか。

```
torch_tensor
0.9918
0.9830
[ CPUFloatType{2} ][ requires_grad = TRUE ]
```

これは真の最小 $(1, 1)$ にかなり近い。気が向いたら、学習率がどのような違いを生じるか試してみよう。例えば、0.001 と 0.1 をそれぞれ使ってみるとよい。

次の章では、白紙からニューラルネットワークを構築する。そこで最小化するのは **損失関数**、つまり回帰問題から現れる平均二乗誤差である。

第 6 章

ニューラルネットワークを白紙から

この章では、回帰問題を解く。とはいっても、1m ではない。実際にニューラルネットワークを構築が、テンソルだけを使う（言うまでもなく *autograd* を有効にしたもの）。もちろん、今後このやり方で *torch* を使うわけではないが、無駄な努力にはならない。むしろその逆だ。基本的な仕組みを見ることで、*torch* が多くの手間を省いてくれることを認識できるだろう。それに、基本を理解することは、驚くほどよくある、深層学習をある種の「魔法」と考えてしまう罠に対する有効な対策になる。それは行列演算の繰り返しに過ぎない。学ぶべきはこれらをまとめ上げることだ。

回帰を行うネットワークに必要なものから始めよう。

6.1 ネットワークの考え方

簡単にいうと、ネットワークは入力から出力への関数だ。適切な関数が求めるものだ。

これを決めるには、会期として線型回帰を考えてみよう。線型回帰がしているのは、掛け算と足し算だ。独立変数一つ一つに対して、これに掛ける係数がある。それから、**バイアス**と呼ばれる項を末尾に加える。（二次元では、回帰係数とバイアスは回帰直線の傾きと y 切片である。）

これらを考慮すると、乗算と加算はテンソルでできることで、テンソルはこれらの演算のためにあると言ってもいいくらいだ。例として、入力が百個の観測からなり、特徴量がそれぞれ三つあるものを示す。

[1] 100 3

x に掛ける特徴量ごとの係数を格納するために、特徴量の数である長さ 3 の列ベクトルが必要だ。あるいは、この後すぐ行う修正に備えて、列の長さが三の行列にしてもよい。つまり、三行を持つ行列である。列はいくつ必要か。単一の特徴量を予測したいとすれば、行列は大きさ 3×1 となる。

よさそうな候補を乱数で初期化する。テンソルは `requires_grad = TRUE` をつけて作成されていることに留意する。この変数はネットワークに学習てもらいたいパラメタを表し

ているからだ。

バイアステンソルは大きさ 1×1 となる。

これで、データに重み w をかけ、バイアス b を加えて「予測」を得ることができる。

```
torch_tensor
-0.2856
-1.3165
-0.2054
-0.4098
-0.2833
-0.9032
-0.6822
 0.3961
-0.3243
-0.1730
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{100,1} ][ grad_fn = <AddBackward0> ]
```

数学の表記では、ここでは次の関数を実装した。

$$f(\mathbf{X}) = \mathbf{XW} + \mathbf{b}$$

このどこがニューラルネットワークなのか。

6.2 ネットワークの層

ニューラルネットワークの用語に戻れば、ここでしたのは単層のネットワーク、出力層の動作の原型である。単層ネットワークは構築したい類とはとてもいえない。単に線型内挿すれば済むのに、なぜネットワークを作るのか。実際、ニューラルネットワークの特徴は、(理論的に) 無数の層を連鎖できることになある。もちろん出力層以外は「隠れ」層と呼ばれるが。深層学習フレームワークを使う上では全く隠れている訳ではない。

例えば、隠れ層が一つあるネットワークを作りたいとしよう。その大きさ、つまり層を持つユニットの数はネットワークの力を決める重要な要因になる。この数は作成する重み行列の大きさに反映される。八つのユニットを持つ層には八つの列を持つ行列が必要である。

```
w1 <- torch_randn(3, 8, requires_grad = TRUE)
```

各ユニットにはそれぞれバイアス値もある。

```
b1 <- torch_zeros(1, 8, requires_grad = TRUE)
```

以前見たように、隠れ層は受け取った入力に重みを掛けてバイアスを加える。つまり、上に示した関数 f を適用する。そして、別の関数を適用する。この関数は隠れ層から入力を受け取り、最終出力を生成する。結局、ここで行われているのは関数の合成である。二番目の関数 g を呼び出すと、全体の変換は $g(f(\mathbf{X}))$ つまり $g \circ f$ となる。

上述の単層構造に類似した出力を g がするには、その重み行列は八つの列の隠れ層を取り、単一の列にしなければならない。つまり、 \mathbf{w}_2 は次のようになる。

```
w2 <- torch_randn(8, 1, requires_grad = TRUE)
```

バイアス \mathbf{b}_2 は \mathbf{b}_1 のように単一の値である。

```
b2 <- torch_randn(1, 1, requires_grad = TRUE)
```

もちろん、単一の隠れ層に止める理由はなく、仕組みを完成させたら、自由にコードをいじってほしい。でも最初に、追加しなければならない部品がいくつかある。その一つは、今の構造では関数を連鎖、あるいは合成しているのはよいが、これらの関数がしているのは加算と乗算であり、線型である。しかし、ニューラルネットワークの力は通常 **非線型性** にある。なぜか。

6.3 活性化関数

ここで、三つの層からなるネットワークがあったとして、各層では入力に重みを掛けただけだとする。(バイアス項を考慮することで変わるものはないにもないが、例がややこしくなるだけなので、省略する。)

このネットワークは行列積の連鎖 $f(\mathbf{X}) = ((\mathbf{X}\mathbf{W}_1)\mathbf{W}_2)\mathbf{W}_3$ になる。さて、この式を変形するとすべての重み行列を掛け合わせてから \mathbf{X} に作用させ $f(\mathbf{X}) = \mathbf{X}(\mathbf{W}_1\mathbf{W}_2\mathbf{W}_3)$ と書くことができる。つまり三つの層のネットワークは単層のものに簡略化され、 $f(\mathbf{X}) = \mathbf{X}\mathbf{W}_4$ となる。こうなると、深層ニューラルネットワークの利点がすべて失われてしまう。

ここで活性化関数、時に「非線型性」が登場する。これは非線型演算を導入するもので、行列演算でモデル化することはできない。歴史的には典型的な活性化関数は **シグモイド** であり、現在でも極めて重要である。その本質的な作用は入力を 0 と 1 との間に圧縮して確率と解釈できる量を与える。回帰では、これが求めるものではなく、ほとんどの隠れ層についても同様だ。

代わりに、ネットワークの中で最も利用されている活性化関数は *ReLU*、Rectified Linear Unit (正規化線型関数) と呼ばれるものだ。名前は長いが単に全ての負の値を 0 するだけだ。torch では `relu()` でこれができる。

なぜこれが非線型なのか。線型である規準の一つは、二つの入力に対し先に加えてから変換しても、先に個々の入力を変換してから加えても結果が同じになることだ。ReLU ではそのようにはならない。

```
torch_tensor
  2
  0
  6
[ CPUFloatType{3} ]
```

```
torch_tensor
  2
  2
  6
[ CPUFloatType{3} ]
```

結果は同じではない。

これまでをまとめると、層を重ねて活性化関数を作用されることを説明した。あと一つ概念を示してからネットワークを完成される。その概念とは損失関数だ。

6.4 損失関数

抽象的にいえば、損失とは目的からどれくらい離れているかという尺度だ。関数を最小化するとき、前の章で行ったように、これは現在の関数値と取りうる最小の値との差である。ニューラルネットワークでは、目的に適合している限り損失関数を自由に選ぶことができる。回帰問題では、平均二乗誤差（MSE: mean square error）がよく用いられるが、それには限らない。平均絶対誤差を代わりに用いるべきこともあるだろう。

`torch` では、平均二乗誤差は一行で書ける。

```
torch_tensor
9.99999e-05
[ CPUFloatType{} ]
```

損失関数が決まれば、重みの更新は勾配に比例した一部を引くことでできる。やり方は既に前の章で見たが、すぐにまた行うことになる。

それでは説明した部品を集めて組み立てよう。

6.5 ネットワークの実装

実装は三つの部分に分かれる。こうしておくと、後で `torch` の高レベルの機能を利用してそれぞれの部分を改訂する際に、カプセル化やモジュール化が行われる部分が分かりやすい。

6.5.1 ランダムデータの生成

例として使うデータは百個の観測からなる。入力 x は三つの特徴量を持つ。目的 y は一つの値で、 y は x から生成されるが、ノイズが加えられる。

6.5.2 ネットワークの構築

ネットワークは隠れ層と出力層の二層とする。つまり二つの重み行列と二つのバイアステンソルが必要である。特に理由はないが、隠れ層には三十二ユニット配置する。

現在の値、乱数初期化の結果は、重みとバイアスは有用ではない。ネットワークを訓練する時が来た。

6.5.3 ネットワークの訓練

ネットワークの訓練は入力を層に伝播させ、損失を計算し、パラメタ（重みとバイアス）を調整して、予測を向上するようにする事である。性能が十分（実際の応用では、非常に注意深く定義する必要がある）と思われるまで、これらの計算を繰り返す。技術的には、これらの手順を反復して適用する各回を **エポック** と呼ぶ。

関数の最小化のように、適切な学習率（差し引く勾配の比率）は実験的に決める。

以下に示す訓練ループを見ると、必然的に四つの部分に分かれることが分かる。

- 順伝播してネットワークの予測を得る（一行に書くのが好みでないなら、分けて書いてもよい）。
- 損失を計算する（これも一行で、いくらかのログ出力を追加しただけだ）。
- *autograd* に損失のパラメタに対する勾配を計算させる。
- パラメタを適切に更新する（ここでも全ての演算を `with_no_grad()` で囲み、`grad` フィールドを反復ごとにゼロにしている）。

```
Epoch: 10    Loss: 26.19279
Epoch: 20    Loss: 24.31503
Epoch: 30    Loss: 22.62099
Epoch: 40    Loss: 21.08653
Epoch: 50    Loss: 19.69379
Epoch: 60    Loss: 18.42898
Epoch: 70    Loss: 17.27875
Epoch: 80    Loss: 16.23043
Epoch: 90    Loss: 15.27229
Epoch: 100   Loss: 14.39839
Epoch: 110   Loss: 13.59888
```

Epoch:	120	Loss:	12.86548
Epoch:	130	Loss:	12.19497
Epoch:	140	Loss:	11.58117
Epoch:	150	Loss:	11.01724
Epoch:	160	Loss:	10.49894
Epoch:	170	Loss:	10.02205
Epoch:	180	Loss:	9.582516
Epoch:	190	Loss:	9.177781
Epoch:	200	Loss:	8.804156

損失は最初急速に減少し、その後それほど速く減少なくなる。この例は素晴らしい性能を示すために作られたのではなく、わずかな行のコードで「本物の」ニューラルネットワークが構築できることを示すことが意図である。

層や、損失、パラメタ更新はまだかなり粗削りで、(文字通り)単なるテンソルである。このような小さなネットワークには問題だが、より複雑な設計ではすぐに面倒になる。以下の二つの章では、重みとバイアスをネットワークに抽象化し、自作の損失関数を組込のものに取り替え、冗長なパラメタ更新部分を取り除く。

第 7 章

モジュール

前の章では、ニューラルネットワークを構築して回帰の問題に適用した。演算には、線型と非線型の二種類があった。

非線型に分類される ReLU 活性化は、簡単な関数呼び出し `nnf_relu()` で表されていた。活性化関数は、関数であり、与えられた入力 \mathbf{x} に対して出力 \mathbf{y} を呼び出す毎に返す。つまり、決定論的であるが、線型部分は異なる。

回帰ネットワークの線型部分は、重みを表す行列との積とバイアスを表すベクトルの和で実装されていた。このような操作では、結果は当然関係するテンソルに格納されている実際の値に依存する。言い換えれば、演算は状態依存である。

状態が関係する場合は、それをオブジェクトにカプセル化して、ユーザが直接管理する必要がないようにするとよい。これが torch のモジュールが行なっていることだ。

モジュールという用語に注意してほしい。torch では、モジュールの複雑さの程度はさまざまである。この後すぐに導入する `nn_linear()` のような基本的な層から、多層からなる完成されたモデルまでが含まれる。コード上では、「層」と「モデル」との違いはない。そのため、「モジュール」だけが使われることもある。本書では、一般的な層とモデルの使い方に従い、概念との対応を重視する。

なぜモジュールかという話に戻る。カプセル化に加えて、層オブジェクトが提供されているのには、別の理由がある。よく用いられる層が全部 `nn_linear()` のように軽量ではない。次の節の最後で、簡単に他のモジュールについて述べ、本書の後の章で詳しく説明する。

7.1 組込の `nn_moudle()`

torch では、線型層は `nn_linear()` で作る。`nn_linear()` は、`in_features` と `out_features` の（最低）二つの引数が必要である。入力データに観測が 50 あり、それぞれ五つの特徴量がある場合、大きさは 50×5 となる。潜在層には、16 のユニットを置く。この場合、`in_features` は 5、`out_features` は 16 である。（自分で重み行列を作る場合、同じ 5 と 16 が行と列の数である。）

一度作られれば、モジュールのパラメタの情報は簡単に得られる。

```
An `nn_module` containing 96 parameters.
```

```
-- Parameters -----
* weight: Float [1:16, 1:5]
* bias: Float [1:16]
```

カプセル化されていても、重みとバイアステンソルを確認することができる。

```
torch_tensor
 0.3601  0.3305  0.0571 -0.0935 -0.4446
-0.3451  0.3687 -0.3867  0.2148 -0.0612
-0.2116 -0.3804  0.3172  0.3535 -0.3074
 0.2177  0.3526  0.0578  0.4434  0.3242
 0.4128 -0.3968 -0.2295  0.2118 -0.2304
-0.4357  0.3819  0.3139 -0.4431  0.3186
 0.0095 -0.1954 -0.3746  0.4162 -0.4187
 0.2530  0.4090 -0.3707 -0.2952  0.4327
-0.3186  0.1896  0.1117 -0.3897 -0.3525
-0.3242 -0.4133 -0.3338 -0.0303  0.1018
 0.3727 -0.0078  0.3368  0.0862 -0.1240
-0.2469 -0.4330  0.3333 -0.0802  0.4338
 0.4260 -0.2566 -0.3499 -0.1909  0.3346
 0.2963 -0.1849  0.2168  0.0848 -0.2071
 0.1666  0.3163 -0.3893 -0.4065  0.2939
-0.0929 -0.1595 -0.0711  0.2656  0.1287
[ CPUFloatType{16,5} ][ requires_grad = TRUE ]
```

```
torch_tensor
-0.2969
 0.2343
-0.2089
-0.1634
-0.0267
-0.2164
-0.0394
-0.0418
-0.2697
 0.1209
-0.0360
-0.0271
```



```
4.4854 -18.2068  9.3530 -6.6302  9.5229
[ CPUFloatType{16,5} ]
```

つまり、`nn_module` を使うと、`torch` は自動的に勾配計算が必要だとみなされる。

`nn_linear` は簡単に見えるが、ほとんど全てのモデル構成において必須の要素だ。

- `nn_conv1d()`、`nn_conv2d()`、および `nn_conv3d()`、いわゆる畳み込み層は入力要素に対するフィルタを異なる次元で適用する。
- `nn_lstm()` と `nn_gru()` は状態を引き継ぐ再帰層。
- `nn_embedding()` は高次元の質的データにつ分かれる。
- その他多数

7.2 モデルの構築

組込の `nn_module()` は普通の呼び方では層だが、どのようにモデルに組み上げるのか。「工場函数」`nn_module()` を使って、モデルを任意の複雑さで定義できる。でも、必ずしもそうする必要はない。

7.2.1 層の列としてのモデル: `nn_sequential()`

モデルが単に層を伝播するだけであれば、モデルを作るときに `nn_sequential()` が使える。線型層だけからなるモデルは多層パーセプトロン (MPL: Multi-Layer Perceptron) として知られている。

関係する層を詳しく見てみよう。ReLU 活性化を実装した函数をであった。(nnf_のfは汎函数であることを示す。) 以下は `nn_relu` は `nn_linear()` と同様にモジュール、つまりオブジェクトで、引数は全てモジュールでなければならない。

組込モジュール同様、このモデルをデータに適用するには呼び出せばよい。

```
torch_tensor
-0.2208
-0.1484
-0.1933
-0.1302
-0.0229
[ CPUFloatType{5,1} ][ grad_fn = <AddmmBackward0> ]
```

一回の呼び出しによりネットワークを通じた順伝播が始動した。同様に、`backward()` を呼び出せば全ての層を通じて逆伝播される。

7.2.2 独自処理のモデル

既に暗示されているように、ここが `nn_module` の使いどころだ。

`nn_module|()` は独自に作成された R6 オブジェクトに対するコンストラクタを作る。以下、`my_linear()` はそのようなコンストラクタだ。呼び出させると、組込の `nn_linear()` に似た線型モジュールを返す。

コンストラクタの定義の中で、二つのメソッドが実装されなければならない。`initialize()` と `forward()` である。`initialize()` はモジュールのオブジェクトフィールドを作る。すなわちそれが持つオブジェクトまたは値でどのメソッドの中からも見える。`forward()` はモジュールが入力があったときの動作を定義する。

`nn_parameter()` の使い方を見てほしい。`nn_parameter()` は渡されたテンソルかモジュールの **パラメタ** として登録されていることを確認する、つまり逆伝播されるのが既定だ。

新たに定義されたモジュールのインスタンスを作るには、コンストラクタを呼び出す。

```
An `nn_module` containing 8 parameters.
```

```
-- Parameters -----  
* w: Float [1:7, 1:1]  
* b: Float [1:1]
```

この例には、モジュールを定義するために必要な独自の計算手順はないが、ここではどんな利用形態にも適用できる雛型である。後に、より複雑な `initialize()` と `forward()` を検討したが、モジュールに定義される追加のメソッドを示す。基本的な仕組みは一緒だ。

ここでは、前の章でモジュールを使ったニューラルネットワークを書き換えることができると感じるかもしれない。自由に取り組んでもよいし、最適化手法と組込損失関数を学ぶ次の章まで待ってもよい。それが済んだら、関数最小化と回帰ネットワークの二つの例に戻ることができる。そして、`torch` により自作した余計なものを取り除くことができる。

第 8 章

最適化器

これまで、テンソルや、自動微分、モジュールについて詳しく学んできた。この章では、`torch` にある主要な概念の締めくくりである最適化手法について調べる。モジュールは層とモデルの計算手順をカプセル化するが、最適化手法は最適化手法を包容する。

なぜ最適化オブジェクトがとても便利なのか考えるところから始めよう。

8.1 何のための最適化器か

この質問には、主に二つの答えがある。一つは技術的なものだ。最初のニューラルネットワークのコードをどのように書いたか思い出すと、次のように進めていたことが分かる。

- 予測（順伝播）を計算し、
- 損失を算出し、
- `autograd` を使って (`loss.backward()` を呼びだし) 偏微分を計算して、
- パラメタを更新するため、勾配の一定の割合をそれぞれ引く。

最後の部分を再掲する。

```
library(torch)

# requires_grad = TRUE がついた
# 全てのテンソルに関して損失の勾配を計算
loss.backward()

### ----- 重みを更新 -----

# この部分は自動勾配計算に記録したくないので、
# `with_no_grad()` で包む。
with_no_grad({
  w1 = w1$sub_(learning_rate * w1$grad)
```

```

w2 = w2$sub_(learning_rate * w2$grad)
b1 = b1$sub_(learning_rate * b1$grad)
b2 = b2$sub_(learning_rate * b2$grad)

# 毎回勾配を 0 にする。
# さもないと積み上がってしまう。
w1$grad$zero_()
w2$grad$zero_()
b1$grad$zero_()
b2$grad$zero_()
})

```

この例は小さなネットワークだったが、このような計算手順を数十、数百層を持つ構造についてコードを書かなければならないとしたら大変だ。当然、それは深層学習フレームワークの開発者がユーザにしてほしいことではない。だから、重みの更新は専用のオブジェクト、当該の最適化器が担当する。

つまり、技術面の答えは使い勝手と利便性に関係する。しかし、他のことも関係している。以上の方法では、良好な学習率は試行錯誤により決めるしかない。そしておそらく、最適な学習率が訓練過程を通じて一定であることもない。ありがたいことに、これまでのたくさんの研究で確立した更新方法がいくつも存在する。これらの方法は、通常演算の間の状態に依存する。これが、torch において最適化器がモジュールのようにオブジェクトである、もう一つの理由だ。

これらの方法を詳しく見る前に、手動の重み更新過程を最適化器を使った版に置き換える方法を示す。

8.2 torch 組込最適化器の利用

最適化器は何を最適化するか知る必要がある。ニューラルネットワークの文脈では、それはネットワークのパラメタである。とは言っても、「モデルのモジュール」と「層のモジュール」に実質的な差はないので、どのように動作するかを `nn_linear` のような単一の組込モジュールで説明する。

まず勾配降下最適化器をある線型モジュールのパラメタに対して働くように作る。

いつも必要である最適化対象のテンソルへの参照に加えて、`optim_sgd()` は一つだけ任意でないパラメタ学習率 `lr` がある。

一度最適化器オブジェクトを作れば、パラメタの更新は `step()` を呼び出すことで実行される。でも変わってないことが一つある。訓練の反復に互って勾配が積み上がらないようにしなければならない。つまり、`zero_grad()` を呼び出す必要があるが、今度は最適化器オブジェクトに対して呼べばよい。

以下に示す完全なコードは、上述の手動の手順を置き換えたものだ。

```
# requires_grad = TRUE がついた
# 全てのテンソルに関して損失の勾配を計算
# ここは変化なし
loss.backward()

# 依然逆伝播の前に勾配を 0 にする必要がある。
# 今度は最適化器オブジェクトに対してする。
optimizer.zero_grad()

# 最適化器を使ってモデルパラメタを更新
optimizer.step
```

使いやすくなったことは納得してもらえらると思う。これはすごい改善だ。ここで、元の質問「何のための最適化器か」に戻り、二つ目の手法面での答えについてもっと議論する。

8.3 パラメタの更新手法

よい学習率を試行錯誤で探すのは大変だ。それに学習率だけがはっきりしないものではない。学習率が指定するのは、歩幅の大きさだけだ。だが、それだけが未解決の問題ではない。

これまで、勾配が与える最急降下方向が最良の方向だと仮定してきた。それはいつも正しいわけではない。そのため、パラメタ更新の大きさと方向の両方とも不明ということになる。

ありがたいことに、ここ十年間でニューラルネットワークにおける重みの更新に関する研究が大きく進展した。ここでは、関係する主要な問題について述べ、`torch` により提供されている最もよく使われる最適化器のいくつかを位置付ける。

比較対象となる基準は **勾配降下** あるいは **最急降下** で、このアルゴリズムを関数最適化やニューラルネットワークの訓練を手動で実装するとき用いてきた。簡単にその指導原理をおさらいしよう。

8.3.1 勾配降下法

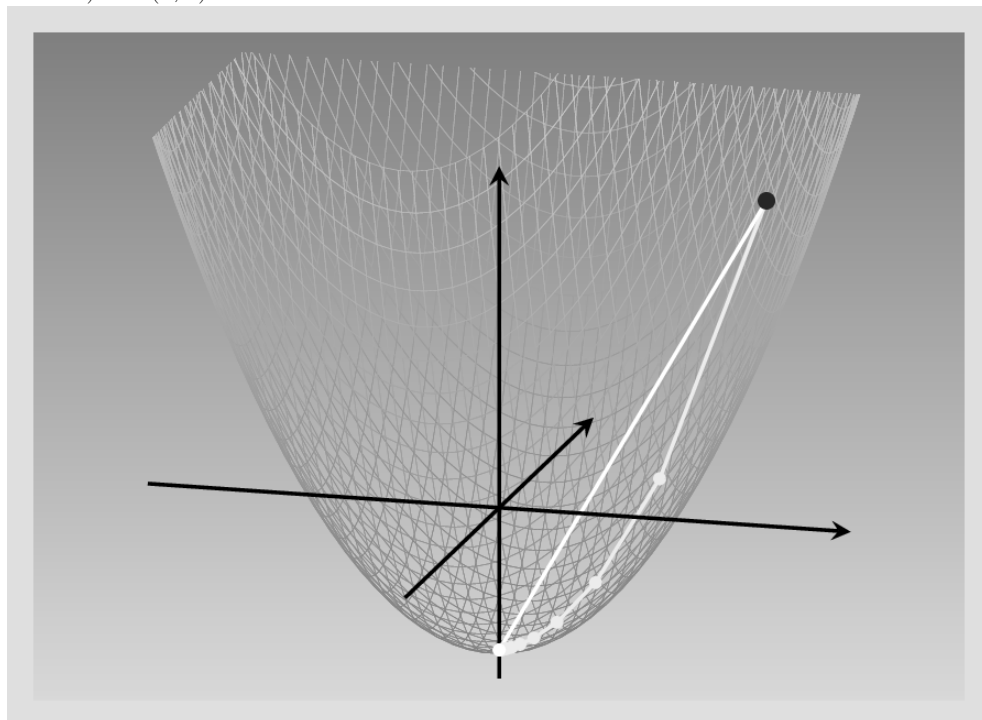
最急降下や確率的勾配降下法 (SGD: stochastic gradient descent) としても知られている。勾配は偏微分のベクトルで、各入力特徴量に対して一つの要素がある。これは、関数が最も増加する方向を表す。その反対方向に進めば、最も速く降下できる。そうだろうか。

残念ながら、そう簡単ではない。取り囲む地形、より技術的には最小化したい関数の等値線に依存する。説明のため、二つの状況を考える。

一つは初めて自動微分を学んだときに出てきたものだ。その例では二次元の二次関数があった。そのときは指摘しなかったが、この特定の関数の重要な点は勾配が二つの次元とも同じだった。そのような条件では、最急降下は最適である。

確認しよう。関数は $f(x_1, x_2) = 0.2x_1^2 + 0.2x_2^2 - 5$ で勾配は $\begin{bmatrix} 0.4 \\ 0.4 \end{bmatrix}$ だった。

例えば点 $(x_1, x_2) = (6, 6)$ にいるとする。それぞれの座標軸について、現在の値の 0.4 倍を引く。これは学習率 1 の場合だが、その必要はない。もし学習率 2.5 を使えば、一步で最小に到達できる。 $(x_1, x_2) = (6 - 2.5 * 0.4 * 6, 6 - 2.5 * 0.4 * 6) = (0, 0)$ 。次の図を見ると、それぞれの場合に何が起きているか分かる。



まとめると、このような等方的な関数は二つの方向の分散が等しいので、学習率を正しく決める「だけ」の問題になる。

次に両方の方向の傾きが大きく異なるとどうなるか、これと比較してみる。

今度は x_2 の係数が x_1 の十倍大きく、 $f(x_1, x_2) = 0.2x_1^2 + 2x_2^2 - 5$ である。つまり x_2 方向に進むと函数値は大きくなるが、 x_1 方向にはもっとゆっくり上昇する。すなわち、勾配降下の間、一つの方向が他の方向よりも大きく進む。

また異なる学習率を使うとどうなるか調べる。以下では三つの異なる設定を比較する。最も小さい学習率を使うと、最終的に最小に到達するものの、対称な場合よりもかなり遅い。わずかに大きな学習率を使うと、ジグザクを繰り返して、より影響の大きい x_2 が正と負の値の間で振動する。最後に、学習率をもう少しだけ大きくすることは最悪の効果をもたらす。函数値は発散し、ジグザクに無限大に大きくなる。

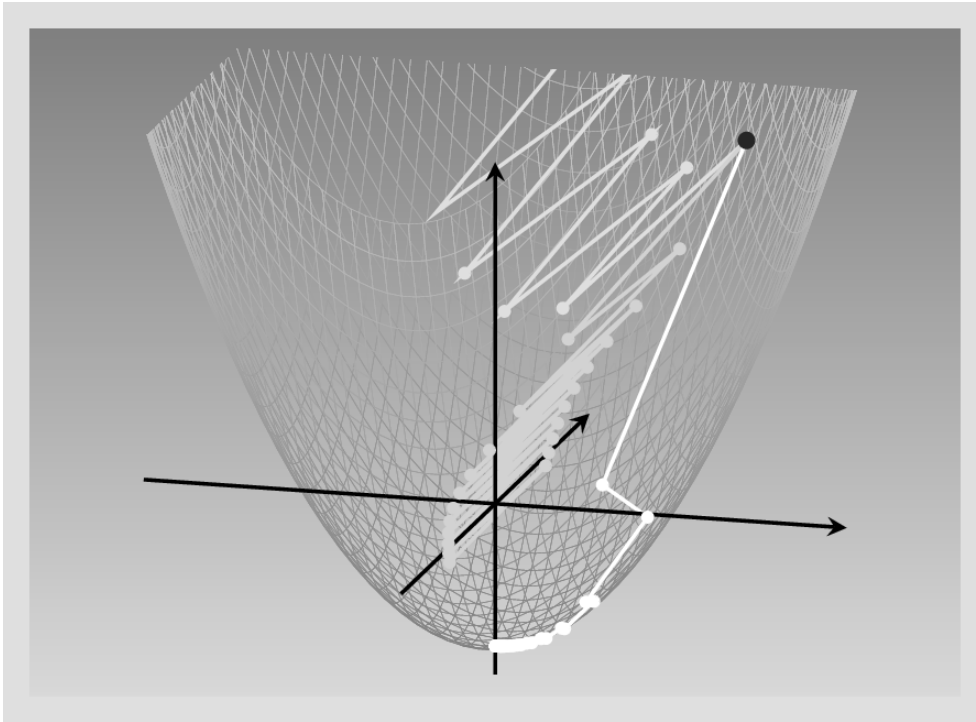


Figure8.1: わずかに異なる学習率を用いた非等方的な放物面上の最急降下

これは非常に説得力がある。二変数だけのよくある関数でも最急降下は万能からは程遠い。そして深層学習では、損失関数は **もっと** 質が悪い。ここがより洗練されたアルゴリズムが必要になるところである。最適化器に再度登場願う。

8.3.2 問題となること

概念から見ると、最急降下法の主な改良は、駆動の考え方、つまり解こうとしている問題により分類できる。

まず、毎回勾配を再計算する度に、全く新しい方向から始める代わりに、古い方向、技術的には慣性を残したい。これは上の例で見た非効率なジグザクの防止に役立つはずである。

次に非対称関数の最小化の例では、本当に全ての変数に対して同じ学習率を使わなければならないのか。明らかに全ての変数が同程度に変化しないことが明らかであるとき、それらを個別に適切な方法で更新したらよいのではないか。

三つ目は、過度に影響が大きい特徴量に対して学習率を下げようとした場合にだけ生じる問題に対する修正で、学習が進み、パラメタが更新されることを確実にしたい。

これらの考慮すべき点は、最適化アルゴリズムの中でいくつかの古典に示されている。

8.3.3 軌道に留まる: 慣性付勾配降下法

慣性付勾配降下法では、勾配を **直接** 重みの更新には使用しない。代わりに、軌道上の粒子として重みの更新を考える。粒子は進んでいる方向を維持しようとする、物理で言う慣性が働くが、衝突により向きが変化する。この「衝突」により **現在** の位置での勾配を考慮するように突かれる。これらの力学から二段階の更新手順が得られる。

以下の式で、記号の選択は物理の類比によるものだ。 \mathbf{x} は位置で、パラメタ空間で現在いるところ、より簡単には現在のパラメタ値である。時間変化は上付添字で表し、 $\mathbf{y}^{(k)}$ は変数 \mathbf{y} の現在時刻 k での状態である。時刻 k における瞬間速度は勾配 $\mathbf{g}^{(k)}$ で測る。位置の更新にはこれを直接用いない。代わりに各反復で、更新された速度は古い速度に慣性パラメタ m で重み付された値と新たに計算した勾配（学習率で重み付けする）の組み合わせとする。二段階の最初はこの方法を表している。

$$\mathbf{v}^{(k+1)} = m\mathbf{v}^{(k)} + lr\mathbf{g}^{(k)} \quad (8.1)$$

二段階目は \mathbf{x} の更新を「折衷」された速度でする。

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{v}^{(k+1)} \quad (8.2)$$

物理の類比以外にも、有用かもしれないのは、時系列解析で有名な概念だ。 m と lr の和が 1 であるように選べば、結果は**指数函数的加重移動平均** となる。（この概念の適用は理解の助けにはなるが、実際は m と lr との和を 1 にする必要はない。）

では、非等方的な放物面に戻り、慣性なしとありの SGD を比較しよう。後者（明るい曲線）では、 $lr = 0.5$ と $m = 0.1$ を用いた。SGD（暗い曲線）で学習率は上の「よい値」を用いた。慣性付 SDG はかなり少ない歩数で最小に到達する Figure 8.2。

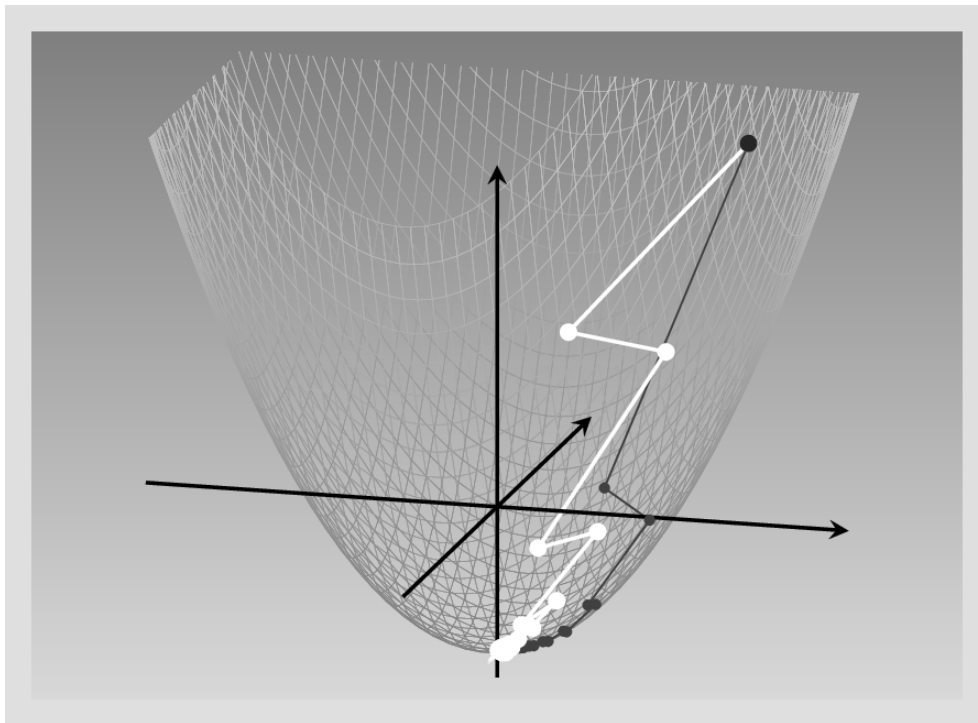


Figure8.2: 慣性付 SGD（白）と素の SGD（灰）との比較

8.3.4 Adagrad

まだ改善できるだろうか。用いている例では、一つの特徴量がもう一つよりもかなり速く変化することとが最適化を遅くしている。異なる学習率をパラメタ毎に使うべきなのは明白だ。実際、深層学習でよく使われる最適化手法はパラメタ毎に学習率を変えている。でもどのように決めるのか。

この点にアルゴリズムの違いが現れる。例えば、Adagrad は個々のパラメタの更新をその偏微分の（正確には二乗）積算和で割っている。ここで、「積算」は最初の反復から積み上げる。「積算変数」 s の対象とするパラメタ i 、反復回 k の値は、次の式で更新する。

$$s_i^{(k)} = \sum_{j=1}^k (g_i^{(j)})^2 \quad (8.3)$$

（ところで、数式が苦手なら読み飛ばして構わない。できるだけ言葉で伝えるようにしているので、基本的な情報を失うことはない。）

ここで、個々の変数に対する更新では、勾配の一部割合を引くのは素の最急降下と同じだが、割合は（大域的な）学習率だけでなく、前述の二乗した偏微分の積算和も用いて決める。その和が大きければ、つまり訓練中の勾配が大きければ大きいほど、調整は小さくなる。^{*1}

^{*1} 引数 `type = "open"` を渡すことにより、単調で周期的でない畳み込みにすることを指定できる。

$$x_i^{(k+1)} = x_i^{(k)} - \frac{lr}{\epsilon + \sqrt{s_i^{(k)}}} g_i^{(k)} \quad (8.4)$$

この手法の全体的な効果は、パラメタの勾配が一貫して大きいと影響は抑制される。ずっと勾配が小さいパラメタが変化すると、十分に考慮される。このアルゴリズムでは、大域的な学習率 lr の重要性は低い。現在の例で最もよい結果に対して、非常に大きな学習率 3.7 を使える（使う必要がある）。今回も素の勾配降下（灰の曲線）と比較した結果を Figure 8.3 に示す。

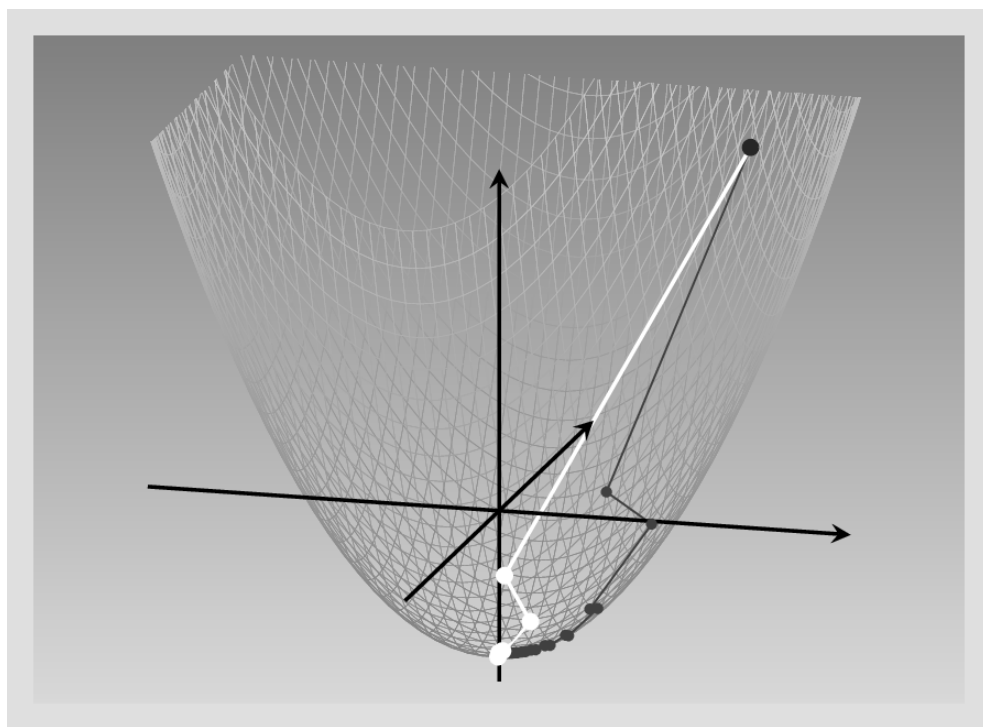


Figure8.3: Adagrad（白）と素の SGD（灰）との比較

この例では、Adagrad は非常によい性能を示す。でもニューラルネットワークの訓練では反復をたくさん行う。その場合、勾配が積算されていくので、実質的な学習率は次第に減少し、行き止まりに達する。

他の方法で学習率を個別、パラメタ毎にできないだろうか。

8.3.5 RMSProp

RMSProp は Adagrad の積算勾配の方法を重み付き平均に置き換える。各点で、「簿記」されたパラメタ毎の変数 s_i は前の値と前の（二乗）勾配の加重平均とする。

$$s_i^{(k+1)} = \gamma s_i^{(k)} + (1 - \gamma)(g_i^{(k)})^2 \quad (8.5)$$

更新は Adagrad と同様にする。

$$x_i^{(k+1)} = x_i^{(k)} - \frac{lr}{\epsilon + \sqrt{s_i^{(k)}}} g_i^{(k)} \quad (8.6)$$

この方法では、各パラメタは適切な重みを得られ、全体として学習が遅くなることがない。
基準である SGD との比較を示す Figure 8.4。

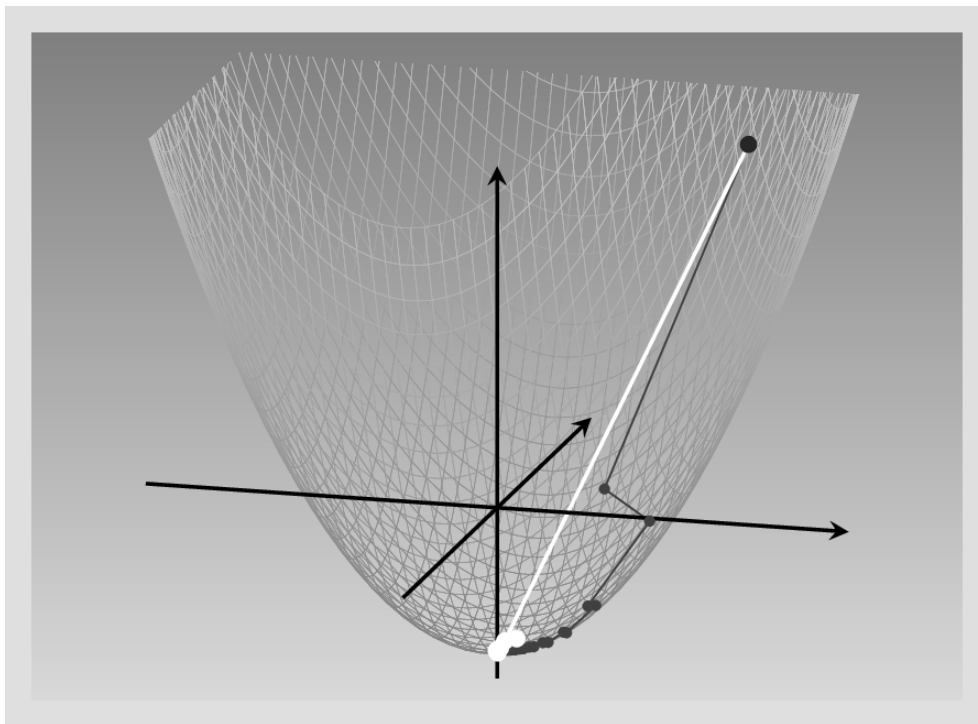


Figure8.4: RMSProp（白）と素の SGD（灰）との比較

現時点で、RMSProp は次に述べる Adam に次いで深層学習で最もよく使われている手法である。

8.3.6 Adam

Adam はこれまでに見た二つの概念を組み合わせている。慣性で「軌道」に留まり、パラメタ依存の更新で速く変化するパラメタへの過剰な依存を回避している。Adam の手順は次の通りである。^{*2}

まず、慣性付 SGD のように、勾配の指数関数的加重平均を維持する。ここでは加重係数 γ_v は通常 0.9 に設定される。

$$v_i^{(k+1)} = \gamma_v v_i^{(k)} + (1 - \gamma_v) g_i^{(k)} \quad (8.7)$$

^{*2} クロネッカーのデルタは、 $i = j$ のときに 1、そうでないときに 0 となる。

また、RMSProp のように二乗勾配の指数関数的加重平均を求め、加重係数 γ_s は通常 0.999 を使う。

$$s_i^{(k+1)} = \gamma_s s_i^{(k)} + (1 - \gamma_s)(g_i^{(k)})^2 \quad (8.8)$$

$$x_i^{(k+1)} = x_i^{(k)} - \frac{lr v_i^{(k+1)}}{\epsilon + \sqrt{s_i^{(k+1)}}} g_i^{(k)} \quad (8.9)$$

この章の締めくくりに、Adam を同じ例で試してみる Figure 8.5。

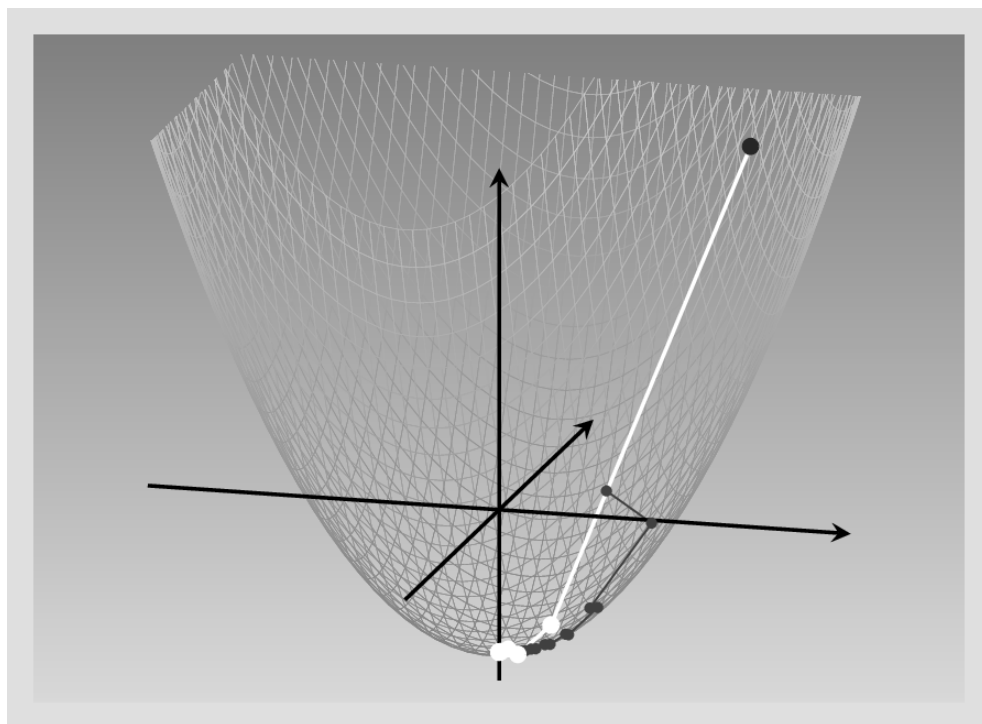


Figure8.5: Adam (白) と素の SGD (灰)

次の章で扱う損失関数は、取り上げる最後の構成要素で、回帰ネットワークと最小化の例を書き直し、`torch` のモジュールと最適化器の恩恵を受ける。

第 9 章

損失関数

損失関数の概念は機械学習に重要なものだ。どの反復においても、現在の損失値は推定値が目的からどれくらい離れているかを示す。そして、その値は損失が減少する方向にパラメタを更新するのに用いられる。

これまでの応用例でも、損失関数を使ってきた。それは平均二乗誤差で手作業で次のように計算した。

```
loss <- (y_pred - y)$pow(2)$sum()
```

想像通り、ここでもこのような手作業は不要だ。

この概念に関する最後の章で、現在の例を見直す前に、二つのことを述べたい。一つは、torch 組込の損失関数の作り方である。もう一つは、どの損失関数を選択するかについてだ。

9.1 torch の損失関数

torch では、損失関数は `nn_` または `nnf_` で始まる。

`nnf_` を使う場合、直接関数呼び出しを行う。これに対応して、その（推定と目的の）引数はどちらもテンソルだ。例えば、`nnf_mse_loss()` という組込関数は手作業でコードを書いたものに類似している。

```
torch_tensor  
0.81  
[ CPUFloatType{} ]
```

一方 `nn_` では、まずオブジェクトを作成する。

このオブジェクトはテンソルに対して呼び出すと、求める損失が得られる。

```
torch_tensor
```

0.81

[CPUFloatType{ }]

オブジェクトまたは関数のどちらを選択するかは、主に好みと文脈次第だ。大きなモデルでは、いくつかの損失関数を合わせてつかうことになる。その際、損失オブジェクトを作る方が部品として扱いやすく、維持しやすいコードになる。本書では、最初の方法を主に用いるが、特別な理由があれば別の方法を取ることにする。

9.2 どの損失関数を選択すべきか。

深層学習や機械学習全般では、ほとんどの応用は数値の予測または確率の推定のどちらか一つ、または両方を行う。現在の例の回帰問題は前者を行なっている。実際の応用では、気温や従業員の離職率を推定したり、売り上げを予測したりする。後者では、典型的な問題は分類だ。例えば画像を最も顕著な内容に基づいて分類するには、実際にはそれぞれの確率を計算する。そして「犬」の確率が0.7で、「猫」の確率が0.3なら犬と判定する。

9.2.1 最尤法

分類も回帰も最も使われている損失関数は最尤原理に基づいている。最尤とは、モデルのパラメタの選択がデータ、つまり観測したものや観測できなかったかもしれないことが最大限起こりやすいようにする。この原理は基本的である「だけ」でなく、直感に訴えるものだ。簡単な例を考えよう。

例えば、7.1, 22.14, 11.3 という値があり、生じさせる過程が正規分布に則っているものとする。その場合、これらのデータは平均14、標準偏差7の分布により生じたものである可能性が、平均20、標準偏差1よりもはるかに可能性が高い。

9.2.2 回帰

回帰（目的の分布が正規分布であるという暗黙の仮定をおいたもの¹）では、尤度を最大化するには、これまで計算してきた損失である、平均二乗誤差を引き続き用いればよい。最尤推定値は望まれる全ての統計的な性質を持つ。しかしながら、特定の用途では他の損失を使う理由があるかもしれない。¹: 仮定があり得ない場合、分布適合損失関数が提供されている（例: Poisson 負対数は `nnf_poisson_nll_loss()`）。

例えば、データセットに外れ値があり、何らかの理由で予測と目的がかなりずれていることがある。平均二乗誤差は外れ値に大きな重みを置く。そのような場合、代替となりうるのは平均絶対誤差（`nnf_l1_loss()`）と滑らかな L1 損失（`nn_smooth_l1_loss()`）である。後者は混合した型で、既定では絶対（L1）誤差を計算するが二乗（L2）に絶対誤差が非常に小さいところで切り替える。

第 10 章

L-BFGS を用いた関数の最適化

`torch` のモジュールと最適化器になじんだところで、別々に取り組んだ二つの課題、関数最小化とニューラルネットワークの訓練に戻る。再び、最小化から始め、ネットワークは次章で扱う。

ローゼンブロッック関数を最小化したときに何をしたら思い返すと、最も重要なことは、

1. 最適化すべきパラメタを格納するテンソルを定義する。つまり、 \mathbf{x} が最小をとるところを見つける。
2. 反復してパラメタを更新するため、現在の勾配の一定の割合を差し引く。

方法としては、これは簡単だが、問題が残っている。どのくらいの割合を差し引くべきか。まさにこれが最適化器が有用な点である。

10.1 L-BFGS 見参

これまで、深層学習でよく使われる最適化器、確率的勾配降下法 (SGD) やモーメンタム付 SGD に加えて、適応学習率を使う種類、RMSProp や、Adadelta、Adagrad、Adam のような古典的な手法について議論した。これらには一つの共通点がある。それは、勾配、つまり一階微分のベクトルだけを使うことだ。従って、これらは一次アルゴリズムである。しかしながら、これらの手法はヘシアン、二階微分の行列から得られる有用な情報が欠けていることを意味している。

10.1.1 変化する傾き

一階微分からは地形の **勾配** が得られる。上がっているか、下がっているか。それはどれくらいか。一歩先に進めると、二階微分は傾きがどれくらい変化するかを表す。

なぜこれが重要なのか。

現在 \mathbf{x}_n にいて、適切な降下方向を決めたとする。事前に定めた学習率で決めた幅の一歩を進んで、 \mathbf{x}_{n+1} に到達して完了する。分からないのは、傾きが到着するまでにどれくら

い変化したかだ。途中でかなり平らになったからかもしれない。その場合、行き過ぎて遠いところに行ってしまい、途中で（傾きが再び上に転じることも含めて）いろいろなことが起きていたかもしれない。

これを単変数の関数で説明できる。放物線、例えば

$$y = 10x^2$$

を考える。その微分は $dy/dx = 20x$ だ。現在の x が例えば 3 で、学習率が 0.1 であれば、 $20 * 3 * 0.1 = 6$ を引いて、-3 に至る。

でも 2 で減速して、そこでの傾きを調べると、緩やかになっているので、 $20 * 2 * 0.1 = 4$ を差し引くことになる。

運を天に任せ、「目を閉じて飛び込む」という方法がうまくいくのは、問題となっている関数に対する適切な学習率を用いた場合に限る。（選んだ学習率だと、別の放物線 $y = 5x^2$ の場合が該当する。）しかし、最初から判断に二階微分を含めておくほうが賢明ではないだろうか。

この形を実践するアルゴリズムがニュートン法の一群である。最初に最も「純粋な」種類を見る。これは、原理を説明するには最適だが、実際に使うのが容易であることは稀である。

10.1.2 厳密ニュートン法

高次元で厳密ニュートン法は勾配にヘシアン逆行列をかけ、降下方向を座標毎に伸縮する。我々の例は独立変数は一つだけなので、一階微分を二階微分で割るということを意味する。

勾配の伸縮をしたところで、どの位の割合を差し引くべきか。原型では、厳密ニュートン法は学習率を使用しないので、慣れ親しんだ試行錯誤からは解放される。例えば、我々の例では二階微分は 20 なので、 $(20 * 3) / 20 = 3$ を引く。確かに、最小値の場所である 0 に一歩でたどり着く。

この結果が素晴らしいのに、なぜいつも使わないのか。一つ理由は、示した例のような二次函数には完璧に働くが、通常は何らかの「調整」、例えば学習率を使うことなども必要となることだ。

しかし、主要な理由は別にある。もっと現実的な応用で、機械学習と深層学習の分野では、ヘシアンの逆行列を毎回計算するのが高くつくからだ。（そもそも可能ではないかもしれない。）そこで、準ニュートン法として知られる近似が使われる。

10.2 ニュートンの近似: BFGS と L-BFGS

ニュートン法の近似のうち、最もよく使われているのは、Broyden-Fletcher-Goldfarb-Shanno アルゴリズム、*BFGS* である。ヘシアンの厳密な逆行列を連続して計算する代わ

りに、反復更新された逆行列の近似を保持する。BFGS はメモリに優しい派生型、メモリ制約 BFGS (*L-BFGS*) で実装されることが多い。これが torch 最適化器の一部として提供されている。

試してみる前に、もう一つのことを説明しておく。

10.2.1 直線探索

厳密な形式同様に、近似ニュートン法は学習率なしで使うことができる。その場合、降下方向を計算して伸縮した勾配にそのまま従う。問題となる関数次第で、これはうまく行ったり行かなかったりする。うまく行かない場合は、二つの対応がある。一つは、短い幅をとる、つまり学習率を導入する。もう一つは直線探索だ。

直線探索を使うと、どこまで降下方向に沿うのかの評価に時間にかける。これを行う二つの主要な方法がある。

最初は、厳密な直線探索で、もう一つの最適化問題を伴う。現在の位置で、降下方向を計算し、これを学習率だけに依存する **第二の関数** として書く。そしてこの関数を微分して、その最小を見つける。解は、歩幅を最適化する学習率となる。

代替の方法は、近似探索だ。もう驚かないだろうが、厳密ニュートン法よりも近似ニュートン法が現実的に実行可能であるように、近似探索は厳密直線探索よりも実行がしやすい。

直線探索では、最適解の近似は以下の折り紙付きの経験則に基づいている。基本的に、十分であるものを探す。確立した経験則の最たるものが **強ウルフ** 条件であり、これが torch の optim_lbfgs() に実装されている。次の節では、optim_lbfgs() の使い方を学び、ローゼンブロッック関数を最適化で、直線探索ありとなしの両方を使う。

10.3 optim_lbfgs() を使ったローゼンブロッック関数の最小化

ローゼンブロッック関数を再掲する。

自作で最小化を試みた時は、次の手順で行った。最初だけ、まずパラメータテンソルを定義し、現在の **x** を格納した。

```
x <- torch_tensor(c(-1, 1), requires_grad = TRUE)
```

そして、反復して次の演算を実行した。

1. 現在の **x** での関数値を計算する。
2. 考えている場所におけるその値の勾配を計算する。
3. 現在の **x** から勾配の一定の割合を差し引く。

どのようにこの青写真が変わるのか。

最初の手順は変わらない。

```
value <- rosenbrock(x)
```

二つ目の手順も変わらず、直接出力テンソルに対して `backward()` を呼ぶ。

```
value$backward()
```

その理由は最適化器は勾配を **計算**せず、ひとたび計算されたら勾配をどうするか決めている。

変わるのは、三番目の手順で、最も面倒でもあった。

更新を適用するのは最適化器である。それを可能にするには、前提がある。ループを始める前に、最適化器はどのパラメタに対して作用するか決められている必要がある。実は、これはとても重要であり、そのパラメタを渡すことなく、最適化器を作ることすらできない。

```
opt <- otpim_lbfgs(x)
```

ループでは最適化オブジェクトに対して `step()` メソッドを呼んでパラメタを最適化する。自作での手順のうち新しいやり方に引き継がなければならないことが一つだけある。依然として各反復で勾配を 0 にしなければならない。今回は、パラメタ `x` に対してではなく、最適化オブジェクトそのものに対してである。基本的に、各反復して次の動作を実行することになる。

```
value <- rosenbrock(x)
```

```
opt$zero_grad()
```

```
value$backward()
```

```
opt$step()
```

なぜ「基本的に」なのか。実は、これは `optimizer_lbdgs()` **以外** に書かなければならないものだ。

`optim_lbfgs()` については、`step()` は無名関数、クロージャに渡す必要がある。前の勾配を 0 にするもの、関数呼び出し、勾配計算が全てクロージャの中で行われる。

```
calc_loss <- function() {  
  optimizer$zero_grad()  
  value <- rosenbrock(x_star)  
  value$backward()  
  value  
}
```


これらの操作を実行したら、クロージャは函数値を返す。step() でどのように呼ぶか示す。

```
for (i in 1:num_iterations) {  
  optimizer$step(calc_loss)  
}
```

組み上がったので、少しログ出力を加えて、直線探索ありとなしで何が起きるか比較してみよう。

10.3.1 optim_lbfgs() の既定の動作

基準として、まず直線探索なしで走らせる。反復 2 回で十分だ。以下の出力で、各反復でクロージャは何回か評価される。これがまずクロージャを書いた技術的な理由だ。

```
Iteration: 1  
Value is: 4
```

```
Iteration: 2  
Value is: 4
```

最小が見つかったか確認するするには、x を印字してみる。

```
torch_tensor  
-1  
 1  
[ CPUFloatType{2} ][ requires_grad = TRUE ]
```

10.3.2 直線探索付 optim_lbfgs()

```
Iteration: 1  
Value is: 4  
Value is: 6  
Value is: 3.802412  
Value is: 3.680712  
Value is: 2.883048  
Value is: 2.5165  
Value is: 2.064779  
Value is: 1.383838
```

```
Value is: 1.073062
Value is: 0.8844348
Value is: 0.5554548
Value is: 0.2501073
Value is: 0.8948812
Value is: 0.1619076
Value is: 0.06823033
Value is: 0.01653571
Value is: 0.004060294
Value is: 0.003537784
Value is: 0.0003913814
Value is: 4.303527e-06
Value is: 2.033371e-08
Value is: 6.870948e-12
```

```
Iteration: 2
Value is: 6.870948e-12
```

直線探索を使うと、一回の反復で十分に最小値に到達する。順に損失を確認すると、アルゴリズムは関数を調べる度にほぼ毎回損失を減少させるが、使わない場合はそうではないことが分かる。

第 11 章

ニューラルネットワークのモジュール化

少し前の章で構築したネットワークを思い出そう。その目的は回帰だったが、その手法は線型ではなかった。その代わり、活性化函数（ReLU、rectified linear unit）により非線型性を導入し、単一の隠れ層と出力層との間に配置した。「層」の元の実装では、単なるテンソルで重みとバイアスを表していた。これらが **モジュール** に置き換えると聞いても驚かないだろう。

どのように訓練の過程は変わるだろうか。概念的には、四つの段階に分けることができる。順伝播、損失の計算、勾配の逆伝播、そして最後に重みの更新である。新しい道具がどこに使われるかを考えてみよう。

- 順伝播では、テンソルに対して函数を呼ぶ代わりに、モデルを呼ぶ。
- 損失計算では、`torch` の `nnf_mse_loss()` を利用する。
- 勾配の逆伝播は唯一変わらない。
- 重みの更新は、最適化器が担当する。

これらの変更を加えたら、コードはよりモジュール化され、より読みやすくなるだろう。

11.1 データ

準備として、データを前のように生成する。

11.2 ネットワーク

二つの線型層が ReLU 活性化函数により接続するには、`sequential` モジュールを使うのが最も簡単である。これは、モジュールを導入したときに見たものによく似ている。

11.3 訓練

更新された訓練の過程を示す。よく選ばれている、Adam 最適化器を使う。

エポック:	10	損失:	2.355549
エポック:	20	損失:	2.18379
エポック:	30	損失:	2.022035
エポック:	40	損失:	1.868386
エポック:	50	損失:	1.722124
エポック:	60	損失:	1.582844
エポック:	70	損失:	1.452155
エポック:	80	損失:	1.331241
エポック:	90	損失:	1.222415
エポック:	100	損失:	1.126599
エポック:	110	損失:	1.045161
エポック:	120	損失:	0.9775077
エポック:	130	損失:	0.9229671
エポック:	140	損失:	0.8812787
エポック:	150	損失:	0.85098
エポック:	160	損失:	0.829643
エポック:	170	損失:	0.8147446
エポック:	180	損失:	0.8031414
エポック:	190	損失:	0.7943742
エポック:	200	損失:	0.7871546

コードが短くなり、効率的になっただけでなく、変更により性能も大きく向上した。

第 II 部

他にも torch でできること: 行列計算・フーリエ変換・ウェーブレット

第 12 章

概要

これまで機械学習について多くの議論を重ねてきた。しかし `torch` は他の目的の科学的な応用においても効果的に利用されている。例えば、パターンや関係、構造を発見する数学的な手法に依存する例が挙げられる。

この部では、三つの話題に絞る。最初は行列計算で、その重要性は疑うことが困難である。実際、科学計算と機械学習の全ての計算は行列計算（テンソルは高次元の行列）である。具体的には、最小二乗法を行列分解により解く。その際に、`linalg_cholesky()`、`linalg_qr()`、`linalg_svd()` のような関数を使う。また、どのように（本来の信号処理の意味での）畳み込みが効率的に実装されているかについても触れる。

次に、有名な数学的手法で、既に（間接的だが非常に便利に）使ったことがある離散フーリエ変換（DST: Discrete Fourier Transform）今回はただ使うだけではなく、なぜそしてどのように動作するか理解することを目的とする。理解できたら、簡単な実装はわずか数行できる。二番目の章では、高速フーリエ変換（FFT: Fast Fourier Transform）を使って、DFT を効率的に実装することに専念する。自作の一つが驚くほど性能の上で `torch_fft_fft()` に近いことを確認することになる。

最後に探求するのは、フーリエ法よりもはるかに新しい考え方、ウェーブレット変換である。この変換はデータ解析において広く用いられているが、その理由が明らかになる。`torch` にはウェーブレット変換を計算する専用の方法はないが、`torch_fft_fft()` を繰り返し使うことで、効率的な実装ができる。

第 13 章

行列演算: 最小二乗問題

本章と次の章では、`torch` が提供する行列演算を調べる。この章では、最小二乗問題を解く様々な方法を見ていく。目的は二つある。

まず、この問題はすぐに技術上、計算上の内容になりがちだ。読者の素養（や目的）により、これが望むものであるかもしれない。基礎的な概念について、よく知っているかもしれないし、あまり興味がないかもしれない。純粋に技術的な、つまり同時に概念や問題の基礎となる抽象的な考え方に基づかない説明だと、一部の読者はこの問題が持つ魅力を感じ、知的な関心を持つことができないかもしれない。そのため、この章では、主な考え方が「コンピュータ科学的」な詳細（多くの素晴らしい本に書いてある）により覆い隠されないような方法で説明することに努める。

13.1 最小二乗を求める五つの方法

線型最小二乗回帰をどのように計算するか。R では、`lm()` を用いる。`torch` には、`linalg_lstsq()` がある。R が利用者からややこしいことを隠蔽するのに対し、`torch` のような高性能計算フレームワークではより明示的な作業を求める傾向がある。作業とは、説明書を丁寧に読むことや、少し試すこと、もしくはその両方だ。例えば、次に示す `linalg_lstsq()` の説明の主要部は `driver` パラメタが詳しく記述されている。

`driver` chooses the LAPACK/MAGMA function that will be used.

For CPU inputs the valid values are 'gels', 'gelsy', 'gelsd', 'gelss'. For CUDA input, the only valid driver is 'gels', which assumes that A is full-rank.

To choose the best driver on CPU consider:

- If A is well-conditioned (its condition number^{*1} is not too large), or you do not mind some precision loss.
- For a general matrix: 'gelsy' (QR with pivoting) (default)
- If A is full-rank: 'gels' (QR)

^{*1} <https://pytorch.org/docs/master/linalg.html#torch.linalg.cond>

- If A is not well-conditioned.
- 'gelss' (tridiagonal reduction and SVD)
- But if you run into memory issues: 'gelss' (full SVD).

See also the full description of these drivers^{*2}

これを知る必要があるかどうかは、解こうとしている問題次第だ。でも必要なら、ここで何が議論されているか理解することは、抽象的であっても、役立つだろう。

以下の例では、幸運にもどのドライバを用いても同一の結果が得られる。ただし、「秘訣」の類を適用することが必要だ。それでも、`linalg_lstsq()` が使っている様々な手法やよく使われる他の方法を深掘りする。具体的には、以下の方法で最小二乗問題を解く。

1. いわゆる **正規方程式**を用いた、最も直接的な方法、その意味は問題の数学的な記述から直ちに得られることを指す。
2. ここでも正規方程式から出発するが、**コレスキー分解**を用いて問題を解く。
3. またまた正規方程式を出発点とし、**LU 分解**を用いて進める。
4. 四つ目と最後の方法では、異なる種類の分解 **QR** を用いる。この分解は「現実」に適用される主要なものだ。QR 分解を用いる場合、アルゴリズムは正規方程式から出発しない。
5. 最後の五つ目は、**特異値分解 (SVD: Singular Value Decomposition)** を用いる。ここでも、正規方程式は不要である。

全ての方法は、現実のデータセット、次に安定性に欠けることが知られているベンチマーク問題に適用される。

13.2 天気予報についての回帰

用いる データセット^{*3}は UCI 機械学習リポジトリ^{*4}から得られる。Zip を展開して、作業ディレクトリ（ここでは `data`）に配置する。ここでの利用方法は、本来の目的とは異なる。機械学習で気温を予測する代わりに、原論文 (Cho et al. 2020) では数値天気予報モデルの予測データのバイアス修正について述べられている。本来の用途を気にする必要はない。ここでの主眼は行列演算にあり、データセット自体はここで行う調査に非常に適している。

	station	Date	Present_Tmax	Present_Tmin	LDAPS_RHmin	LDAPS_RHmax
1	1	2013-06-30	28.7	21.4	58.25569	91.11636
2	2	2013-06-30	31.9	21.6	52.26340	90.60472
3	3	2013-06-30	31.6	23.3	48.69048	83.97359
4	4	2013-06-30	32.0	23.4	58.23979	96.48369

^{*2} <https://netlib.org/lapack/lug/node27.html>

^{*3} <https://doi.org/10.24432/C59K76>

^{*4} <https://archive.ics.uci.edu/>

```

5      5 2013-06-30      31.4      21.9      56.17410      90.15513
6      6 2013-06-30      31.9      23.5      52.43713      85.30725
  LDAPS_Tmax_lapse LDAPS_Tmin_lapse LDAPS_WS  LDAPS_LH LDAPS_CC1 LDAPS_CC2
1      28.07410      23.00694 6.818887  69.45181 0.2339475 0.2038957
2      29.85069      24.03501 5.691890  51.93745 0.2255082 0.2517714
3      30.09129      24.56563 6.138224  20.57305 0.2093437 0.2574694
4      29.70463      23.32618 5.650050  65.72714 0.2163720 0.2260024
5      29.11393      23.48648 5.735004 107.96554 0.1514069 0.2499953
6      29.21934      23.82261 6.182295  50.23139 0.1852788 0.2808180
  LDAPS_CC3 LDAPS_CC4 LDAPS_PPT1 LDAPS_PPT2 LDAPS_PPT3 LDAPS_PPT4      lat
1 0.1616969 0.1309282      0      0      0      0 37.6046
2 0.1594441 0.1277273      0      0      0      0 37.6046
3 0.2040915 0.1421253      0      0      0      0 37.5776
4 0.1611574 0.1342487      0      0      0      0 37.6450
5 0.1788925 0.1700210      0      0      0      0 37.5507
6 0.2328410 0.1463629      0      0      0      0 37.5102
      lon      DEM  Slope Solar.radiation Next_Tmax Next_Tmin
1 126.991 212.3350 2.7850      5992.896      29.1      21.2
2 127.032  44.7624 0.5141      5869.312      30.5      22.5
3 127.058  33.3068 0.2661      5863.556      31.1      23.9
4 127.022  45.7160 2.5348      5856.965      31.7      24.3
5 127.135  35.0380 0.5055      5859.552      31.2      22.5
6 127.042  54.6384 0.1457      5873.781      31.5      24.0

```

ここでの問題の組み立て方では、基本的にデータセット内の全てが予測変数である。(データセットに残しておけば予測因子になりうる。詳しくは以下で説明する) 目的変数としては、Next_Tmax 翌日に取った最高気温を用いる。この場合、Next_Tmin を予測変数から取り除く必要がある。さもないと、これが強力すぎる手掛かりになる。同様に station 測候所の識別番号と Date も削除する。結果として、21 個の予測変数が得られ、これらには実際の気温 (Prexent_Tmax、Present_Tmin)、様々な変数に対するモデルの予報 (LDAPS_*)、そして補助的な情報 (lat や、lon Solar radiation など) がある。

ここで予測変数を **標準化** する `scale()` を追加したことに目を向けてほしい。これは、上述した「秘訣」である。なぜそのようなするかは、すぐに議論する。

`torch` には、データを二つのテンソルに分け、全ての予測変数を行列 `A` に、目的変数を格納した `b` に格納する。

```
[1] 7588    21
```

それでは、最初に期待される出力を求めよう。

13.2.1 最小二乗法 (I): `lm()` による参照値

「信用できる」最小二乗法の実装があるとすれば、もちろん `lm()` に違いない。

Call:

```
lm(formula = Next_Tmax ~ ., data = weather_df)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-1.94439	-0.27097	0.01407	0.28931	2.04015

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.901e-12	5.390e-03	0.000	1.000000
Present_Tmax	1.456e-01	9.049e-03	16.089	< 2e-16 ***
Present_Tmin	4.029e-03	9.587e-03	0.420	0.674312
LDAPS_RHmin	1.166e-01	1.364e-02	8.547	< 2e-16 ***
LDAPS_RHmax	-8.872e-03	8.045e-03	-1.103	0.270154
LDAPS_Tmax_lapse	5.908e-01	1.480e-02	39.905	< 2e-16 ***
LDAPS_Tmin_lapse	8.376e-02	1.463e-02	5.726	1.07e-08 ***
LDAPS_WS	-1.018e-01	6.046e-03	-16.836	< 2e-16 ***
LDAPS_LH	8.010e-02	6.651e-03	12.043	< 2e-16 ***
LDAPS_CC1	-9.478e-02	1.009e-02	-9.397	< 2e-16 ***
LDAPS_CC2	-5.988e-02	1.230e-02	-4.868	1.15e-06 ***
LDAPS_CC3	-6.079e-02	1.237e-02	-4.913	9.15e-07 ***
LDAPS_CC4	-9.948e-02	9.329e-03	-10.663	< 2e-16 ***
LDAPS_PPT1	-3.970e-03	6.412e-03	-0.619	0.535766
LDAPS_PPT2	7.534e-02	6.513e-03	11.568	< 2e-16 ***
LDAPS_PPT3	-1.131e-02	6.058e-03	-1.866	0.062056 .
LDAPS_PPT4	-1.361e-03	6.073e-03	-0.224	0.822706
lat	-2.181e-02	5.875e-03	-3.713	0.000207 ***
lon	-4.688e-02	5.825e-03	-8.048	9.74e-16 ***
DEM	-9.480e-02	9.153e-03	-10.357	< 2e-16 ***
Slope	9.402e-02	9.100e-03	10.331	< 2e-16 ***
Solar.radiation	1.145e-02	5.986e-03	1.913	0.055746 .

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.4695 on 7566 degrees of freedom

Multiple R-squared: 0.7802, Adjusted R-squared: 0.7796

F-statistic: 1279 on 21 and 7566 DF, p-value: < 2.2e-16

分散説明率 78% は、予報がうまくいっていることを示している。これを基準として、他の手法を比較調査する。そのためにそれぞれの予測と予測誤差を保存する（後で平均二乗誤差 RMSE: root mean squared error を計算するため）。今は、`lm()` に対する入力だけがある。

```
lm
1 40.8369
```

13.2.2 最小二乗法 (II): `linalg_lstsq()` の利用

	b	lm	lstsq
7745	-1.1380931	-1.3544620	-1.3544625
7746	-0.8488721	-0.9040997	-0.9040998
7747	-0.7203294	-0.9675286	-0.9675287
7748	-0.6239224	-0.9044044	-0.9044047
7749	-0.5275154	-0.8738639	-0.8738641
7750	-0.7846007	-0.8725795	-0.8725797

予測は `lm()` と非常によく似ている。実際、想像されるのは、わずかな差が数値誤差がそれぞれの呼び出しスタックの奥から表面化したものだけではないかということで、すなわち、同じはずだということだ。

```
lm    lstsq
1 40.8369 40.8369
```

確かに。これは満足のいく結果だ。しかし、これはあの「秘訣」、標準化に基づく結果だ。もちろん、「秘訣」というほどのものではない。データの標準化は通常の操作で、特にニューラルネットワークでは日常的に使われており、訓練を高速化している。強調したい点は、`torch` のような高性能計算のためのフレームワークは適用される領域の知識や、事前の解析がより多く利用者側に求められることが多いということだ。

13.2.3 一休み: 標準化しなかった場合

素早く比較するために、今度はデータを標準化 **せず** に別の予測変数の行列を作ってみる。

```
lm
1 127.0765
```

次に、前のように既定の引数とともに `linalg_lstsq()` を呼ぶ。

```
lm      lstsq
1 127.0765 217.7987
```

なんと、何が起きたのか。引用した説明書の一部を思い出すと、今度は既定の引数がうまくいかなかったのかもしれない。理由を見つけよう。

13.2.3.1 問題の調査

最小二乗問題を効率的に解くため、`torch` は LAPACK という Fortran のルーチン一式を呼んでいる。LAPACK は効率的かつ規模の拡大に対応しつつ、線型代数においてよくある問題を解けるように設計されており、線型方程式の解や固有ベクトルと固有値、特異値を求めることができる。

`linalg_lstsq()` で利用できる `driver` は LAPACK の異なる手続^{*5}に対応し、これらの手続は皆異なるアルゴリズムで問題を解く。これは、今後行っていくことに類似している。

つまり、何が起きているか調べるには、第一歩はどの手法がなぜ利用されたのか、（可能ならば）なぜ結果が満足いかないのか、代わりに使いたい LAPACK のルーチンを定め、実際に用いたときに何が起こるか確認することである。（もちろん、わずかな手間がかかるだけなら、全ての方法を試せばよい。）

13.2.3.2 概念 (I): 行列の階数

「ちょっと待て！」上述の説明書の一部から、最初にすべきことは階数ではなく、**条件数**、つまり行列が「良条件」であると思うかもしれない。確かに条件数は重要で、この後すぐにこの問題に戻る。でも、もっと基本的にことが起きている。実際に「目に飛び込んでくる」ものではないことが。

`linalg_lstsq()` について参照してる、LAPACK の説明書の一部に重要な情報がある。四つのルーチン `GELS`、`GELSY`、`GELSD`、`GELSS` の間で、違いは実装に限らない。最適化の目的も異なっている。根拠は次の通りだ。一貫して、行列は行が列よりも多い（観測が特徴量より多い通常の場合）と仮定する。

- 行列が完全階数、つまり列が線型独立なら「完璧な」回は存在しない。問題は優決定である。できることは、最良の近似を見つけることだ。これは予測誤差を最小化することである。この点は正規方程式について議論するときに再検討する。予測誤差の最小化が `GELS` が行うことで、`GELS` が予測変数の完全階数の行列があるときに使うべきルーチンである。
- 行列が完全階数でない場合、問題は劣決定である。このとき、解は無数にある。残りの全てのルーチン `GELSY`、`GELSD`、`GELSS` はこの解に適している。計算の進め方は異なっているが、同一の方法をとる。これは `GELS` のものとは異なる。予測誤

^{*5} 引数 `type = "open"` を渡すことにより、単調で周期的でない量み込みにすることを指定できる。

差だけではなく、**加えて**係数のベクトルも最小化する。これは最小ノルム最小二乗解を見つけることである。

まとめると GELS（完全階数行列向け）と GELSY、GELSD、GELSS の三種（階数不足の向け行列向け）は意図的に異なる最適化の基準に従っている。

さて、`linalg_lstsq()` の説明書に従って、`driver` が明示的に渡されていないときは GELSY が呼び出される。これは行列が階数不足のときに適切なはずだが、そうになっているだろうか。

```
torch_tensor
21
[ CPU LongType{ } ]
```

行列は 21 列あり、階数は 21 なら、完全階数である。呼び出すべきは確実に GELS ルーチンだ。

13.2.3.3 正しい `linalg_lstsq()` の呼び出し方

何を `driver` に渡すか分かったので、呼び出しを修正する。

```
lm      lstsq
1 127.0765 127.9489
```

今度はそれぞれの RMSE が非常に近くなった。標準化された行列を使ったときは、なぜ Fortran のルーチンを指定する必要がなかったのか疑問に思うだろう。

13.2.3.4 標準化が役立つ理由

用いた行列に対して、標準化がしたことは特異値が範囲をかなり狭めたということだ。標準化された行列 A では、特異値の最大値は最小値の約 10 倍である。

```
[1] 1.0000000 0.7473212 0.5929527 0.5233989 0.5188765 0.4706139 0.4391665
[8] 0.4249272 0.4034658 0.3815900 0.3621314 0.3557949 0.3297923 0.2707912
[15] 0.2489560 0.2229860 0.2175170 0.1852890 0.1627083 0.1553169 0.1075778
```

一方 A_alt では約 100 万倍大きい。

```
[1] 1.000000e+00 1.014368e-02 6.407313e-03 2.881967e-03 2.236537e-03
[6] 9.633786e-04 6.678379e-04 3.988165e-04 3.584047e-04 3.137257e-04
[11] 2.699152e-04 2.383504e-04 2.234152e-04 1.803385e-04 1.625245e-04
[16] 1.300102e-04 4.312534e-05 3.463854e-05 1.964121e-05 1.689914e-05
[21] 8.419588e-06
```

これがなぜ重要なのか。ここでついに **条件数** に話が戻る。

13.2.3.5 概念 (II): 条件数

いわゆる **条件数** が大きいほど、計算の際に数値安定性の問題が生じる可能性が高くなる。`torch` では、`linalg_cond()` を用いて条件数を計算できる。`A` と `A_alt` の条件数をそれぞれ比較しよう。

```
torch_tensor
9.2956
[ CPUFloatType{ } ]
```

```
torch_tensor
118771
[ CPUFloatType{ } ]
```

かなりの差だ。どこから来たのか。

条件数は `A` の行列ノルムをその逆行列のノルムで割ったものとして定義される。様々な種類のノルムが用いられるが、既定は 2 ノルムである。その場合、条件数は行列の特異値はから計算できる。つまり、`A` の 2 ノルムは最大の特異値であり、その逆行列のものは最小値で与えられる。

以前のように `linalg_svdvals()` でこれを確認できる。

```
torch_tensor
9.2956
[ CPUFloatType{ } ]
```

```
torch_tensor
118771
[ CPUFloatType{ } ]
```

繰り返しになるが、これはかなりの違いだ。同時に `A_alt` の場合、`linalg_lstsq()` に対する RMSE が `lm()` に対するものよりもわずかに悪くなっていることをご記憶だろうか。適切なルーチン GELS を使ったにもかかわらず。どちらも基本的に同一のアルゴリズム（すぐに説明する QR 分解）を用いているとすると、数値誤差が `A_alt` の高い条件数により生じている可能性が高い。

ここまでで、`torch` の `linalg` の部分を使うと、最もよく使われている最小二乗アルゴリズムがどのように動いているか理解することに役立つごとに納得してもらえたと思う。これらに馴染もう。

13.2.4 最小二乗法 (III): 正規方程式

目的を示すことから始める。特徴量を列に観測を行に格納した行列 \mathbf{A} 及び観測された結果のベクトル \mathbf{b} が与えられたとき、各特徴量について \mathbf{b} を最もよく近似する回帰係数一つ求めたい。回帰係数のベクトルを \mathbf{x} とする。求めるには、連立方程式を解く必要があり、その行列表記は次のようになる。

$$\mathbf{Ax} = \mathbf{b}$$

\mathbf{A} が正方行列なら、解を直接求めることができ、 $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ となる。しかし、これはほとんど不可能である。予測変数よりも多くの観測が（おそらく）常にある。別の方法が必要である。

\mathbf{A} の列を \mathbf{b} の近似に使う場合、この近似は \mathbf{A} の列空間に存在しなければならない。 \mathbf{b} は通常そうではない。これらは可能な限り近づいてほしい。つまり、これらの距離を最小化したい。距離として 2 ノルムを使うと、目的は次のようになる。

$$\text{minimize} \|\mathbf{Ax} - \mathbf{b}\|$$

距離は（二乗された）予測誤差のベクトルの長さである。このベクトルは \mathbf{A} と直交していなければならない。

$$\mathbf{A}^T(\mathbf{Ax} - \mathbf{b}) = 0$$

これを並び替えるといわゆる **正規方程式** が得られる。

$$\mathbf{A}^T = \mathbf{A}^T\mathbf{b}$$

これを \mathbf{x} について解くため、 $\mathbf{A}^T\mathbf{A}$ の逆行列を計算する。

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$$

$\mathbf{A}^T\mathbf{A}$ は正方行列である。逆行列が求まらないかもしれないが、その場合は擬逆行列を代わりに計算すればよい。例としている問題はその必要はない。 \mathbf{A} は完全階数なので、 $\mathbf{A}^T\mathbf{A}$ も完全階数である。

このように、正規方程式から \mathbf{b} を予測する方法が導出された。これを使って、`lm()` や `linalg_lstsq()` を用いて得られたものと比較しよう。

```
lm    lstsq    neq
1 40.8369 40.8369 40.8369
```

直説法がうまくいくことを確かめたので、より洗練された手法を試す。四つの行列分解が現れる。コレスキー、LU、QR、そして特異値分解だ。目的は、どの場合でも、(擬)逆行列の重い計算を回避することだ。この計算は全ての手法に共通する。しかしながら、行列の分解の仕方「だけ」ではなく、どの行列を分解するかも異なる。これは様々な手法が課す制約と関係している。大まかに述べると、上に並べた順序は前提条件の程度を反映している。別の言い方をすれば、後ろほど一般性が高くなる。関係する制約に応じて、最初の二つ（コレスキーと LU 分解）は $\mathbf{A}^T \mathbf{A}$ に対して行い、後の二つ（QR と SVD）は \mathbf{A} に直接作用させる。これらを使うと $\mathbf{A}^T \mathbf{A}$ を計算する必要がない。

13.2.5 最小二乗法 (IV): コレスキー分解

コレスキー分解では、行列は二つの同じ大きさの三角行列に分解される。一方は他方の転置となっている。通常次のように表す。

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

または

$$\mathbf{A} = \mathbf{R}^T \mathbf{R}$$

ここで記号 \mathbf{L} 及び \mathbf{R} はそれぞれ下三角及び上三角行列である。

コレスキー分解が可能であるためには、行列は対称かつ正定値でなければならない。これはかなり強い条件で、実際に満たされることは多くない。用いている例では \mathbf{A} は対称ではないので、代わりに $\mathbf{A}^T \mathbf{A}$ に作用される必要があることが直ち示唆される。また \mathbf{A} はすでに正定値なので、 $\mathbf{A}^T \mathbf{A}$ も正定値であることが分かっている。

`torch` では、コレスキー分解を得るには `linalg_cholesky()` を用いる。既定では、呼び出すと下三角行列 \mathbf{L} が返ってくる。

\mathbf{A} が \mathbf{L} から再構築できるか確認する。

```
torch_tensor
0.00372015
[ CPUFloatType{ } ]
```

ここでは、元の行列と再構築したものとの差に対してフロベニウスノルムを計算した。フロベニウスノルムは全ての行列の要素について和を取り、平方根を返す。理論的には、ここでは零を見たいが、数値誤差の存在の下では、結果は分解が十分にうまくいったことを示している。

$\mathbf{L}\mathbf{L}^T$ が $\mathbf{A}^T \mathbf{A}$ の代わりに得られたことがどのように役立つのか。ここが魔法が起きるところで、同様な魔法が残りの三つの手法でも働くことを見出すことになる。着想は、ある分解のために、問題を構成する連立方程式を解くためのより性能の良い方法が得られるということにある。それはよく分かるのは小さな例だ。

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 11 \\ 15 \end{bmatrix}$$

最初の行から始めると、直ちに x_1 が 1 であることが分かる。これが分かると、二番目の行から x_2 は 3 に違いないと計算することは容易だ。最後の量は x_3 は 0 であることを示している。

コードでは、`torch_triangular_solve()` を使うと効率的に予測変数の行列が下または上三角行列である連立方程式を効率的に解くことができる。追加の条件は行列が対称であることであるが、これはコレスキー分解を使うために既に満たされている。

既定では、`torch_triangular_solve()` は上（下ではなく）三角行列を想定するが、関数のパラメタに `upper` があり、想定を修正することができる。返り値はリストで、最初の要素が求める解である。説明のため、`torch_triangular_solve()` を、上で暗算で解いたおもちゃの例に適用したものを示す。

```
torch_tensor
1
3
0
[ CPUFloatType{3,1} ]
```

現在の例に戻ると、正規方程式は次のようになる。

$$\mathbf{L}\mathbf{L}^T\mathbf{x} = \mathbf{A}^T\mathbf{b}$$

新しい変数 \mathbf{y} を $\mathbf{L}^T\mathbf{x}$ を表すために導入する。

$$\mathbf{L}\mathbf{y} = \mathbf{A}^T\mathbf{b}$$

そして **この** 系の解を求める。

\mathbf{y} がもとまったので、これがどのように定義されていたか振り返る。

$$\mathbf{y} = \mathbf{L}^T\mathbf{x}$$

\mathbf{x} を定めるには、また `torch_triangular_solve()` を用いる。

これで完了。

通常通り、誤差を計算する。

```
lm    lstsq    neq    chol
1 40.8369 40.8369 40.8369 40.8369
```

コレスキー分解の理論的根拠を理解した。既に述べたが、この考え方は他の全ての分解にも適用される。実は、専用の便利な関数 `torch_cholesky_solve` を利用すると作業を節約できる。これを使うと、二つの `torch_triangular_solve()` の呼び出しが不要になる。

```
lm  lstsq  neq  chol  chol2
1 40.8369 40.8369 40.8369 40.8369 40.8369
```

次の手法、つまり次の分解に進もう。

13.2.6 最小二乗法 (V): LU 分解

LU 分解という名前は、分解により得られる二つの因子に由来している。下三角行列 \mathbf{L} と上三角行列 \mathbf{U} である。理論的には、LU 分解に制約はない。つまり、行の交換を許して、実質的に $\mathbf{A} = \mathbf{LU}$ を $\mathbf{A} = \mathbf{PLU}$ (ここで \mathbf{P} は置換行列) とすれば、どんな行列も分解できる。

実際には、`torch_triangular_solve()` を使う場合は、入力する行列は対称でなければならない。そのため、 \mathbf{A} ではなく、 $\mathbf{A}^T \mathbf{A}$ を使うことになる。(そのため、LU 分解をコレスキーの直後に説明している。これらは使い方が似ているものの、考え方は全く異なる。)

$\mathbf{A}^T \mathbf{A}$ を使うということは、この方法も正規方程式から出発することを意味する。 $\mathbf{A}^T \mathbf{A}$ を分解し、二つの三角行列を解いて、最終的な解を得る。ここで手順を示す。常に必要とは限らない置換行列 \mathbf{P} も含まれている。

$$\begin{aligned}\mathbf{A}^T \mathbf{A} \mathbf{x} &= \mathbf{A}^T \mathbf{b} \\ \mathbf{PLU} \mathbf{x} &= \mathbf{A}^T \mathbf{b} \\ \mathbf{L} \mathbf{y} &= \mathbf{P}^T \mathbf{A}^T \mathbf{b} \\ \mathbf{y} &= \mathbf{U} \mathbf{x}\end{aligned}$$

\mathbf{P} が必要な場合、追加の計算が生じる。コレスキーで使った方法と同様に \mathbf{P} を左辺から右辺に移したい。幸いにも、大変そうな逆行列の計算は重くない。置換行列に対しては、その転置が逆演算となる。

コード上は、しなければならないことは既に馴染みのあるものだ。てできてないのは `torch_lu()` だけだ。`torch_lu()` は二つのテンソルのリストを返す。最初の要素は三つの行列 \mathbf{P} 、 \mathbf{L} 、 \mathbf{U} で、二つ目はピボットだ。`torch_lu_unpack()` で三つの行列に展開できる。

\mathbf{P} を右辺に移す。

あとは、二つの三角行列を解けば完了する。

```
lm  lstsq  neq  chol  chol2  lu
1 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369
```

コレスキー分解と同様に、`torch_triangular_solve()` を二回呼び出す手間を省ける。`torch_lu_solve()` は分解を取り、直接最終的な解を返す。

```
      lm  lstsq    neq   chol   chol2      lu    lu2
1 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369
```

次に、 $\mathbf{A}^T \mathbf{A}$ の計算が必要ない二つの方法を見る。

13.2.7 最小二乗法 (VI): QR 分解

どんな行列も直交行列 \mathbf{Q} と上三角行列 \mathbf{R} に分解できる。QR 分解は、最小二乗問題の解法としておそらく最もよく使われており、 \mathbf{R} の `lm()` にも採用されている。どのようにして、問題を簡単にしているのか。

\mathbf{F} については、どのように便利かを既に述べた。三角行列により、代入だけで順に解くことができる連立方程式が得られる。 \mathbf{Q} はもっと便利である。直交行列は、列が直交するものを指す。これが意味するのは、相互内積が全て 0 で、ノルムが 1 であるということだ。そのため、このような行列はその逆行列が転置に等しい。一般的には、逆行列は計算が困難である一方、転置は容易である。逆行列を計算し、 $\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$ を解くことが最小二乗法の中心的な問題であることを考慮すると、その重要性は明らかだ。

これまでの方法と比較して、手順は少し短くなる。ダミー変数 \mathbf{y} はもう不要だ。代わりに \mathbf{Q} を右辺に移し、転置(逆行列)を求める。残りは、後退代入だけだ。また、全ての行列に対して QR 分解が存在するので、 \mathbf{A} から直接始め、 $\mathbf{A}^T \mathbf{A}$ は不要だ。

$$\begin{aligned}\mathbf{Ax} &= \mathbf{b} \\ \mathbf{QRx} &= \mathbf{b} \\ \mathbf{Rx} &= \mathbf{Q}^T \mathbf{b}\end{aligned}$$

`torch` では `linalg_qr()` が行列 \mathbf{Q} と \mathbf{R} を計算する。

右辺に $\mathbf{A}^T \mathbf{b}$ を格納した「便宜的な」変数を用いてきたがこの手順は省略でき、代わりに「すぐに役立つ」ことをする。つまり \mathbf{Q} を右辺に移す。

残りの手順は、三角行列を解くことだけだ。

```
      lm  lstsq    neq   chol      lu    qr
1 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369
```

ここまで読んでくると、節の終わりに「専用の関数が `torch / torch_linalg` にある...

という記述を期待しているかもしれない。文字通りのものはないが、実質上はある。`linalg_lstsq()` を呼び出すときに `driver = "gels"` を渡すと、QR 分解が使われる。

13.2.8 最小二乗法 (VII): 特異値分解 (SVD)

まさに、頂点に向かう順序で、最後に説明する分解手法は万能で、用途が広く、意味論において価値がある、特異値分解 (SVD: singular value decomposition) である。三つ目の側面は、興味深いものの、現在の問題に関係しないので省略する。ここでは、汎用性が重要である。全ての行列は、SVD の形式に分解して構成することができる。

特異値分解は、入力 \mathbf{A} を二つの直交行列 \mathbf{U} と \mathbf{V}^T と対角行列 Σ に分解し、 $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T$ という形にする。ここで、 \mathbf{U} と \mathbf{V}^T は左 と 右 特異ベクトルであり、 Σ には特異値が格納されている。

$$\begin{aligned}\mathbf{Ax} &= \mathbf{b} \\ \mathbf{U}\Sigma\mathbf{V}^T\mathbf{x} &= \mathbf{b} \\ \Sigma\mathbf{V}^T\mathbf{x} &= \mathbf{U}^T\mathbf{b} \\ \mathbf{V}^T\mathbf{x} &= \mathbf{y}\end{aligned}$$

行列の分解を `linalg_svd()` を用いて求めることから始める。引数 `full_matrices = FALSE` は `torch` に \mathbf{U} が \mathbf{A} と同じ次元であることを伝え、7588 x 7588 に展開されるのを防ぐ。

```
[1] 7588 21
```

```
[1] 21
```

```
[1] 21 21
```

\mathbf{U} を右辺に移す。これは \mathbf{U} が直交行列なので軽い計算である。

$\mathbf{U}^T\mathbf{b}$ と Σ は同じ長さのベクトルなので、要素毎の積を使って Σ を右辺に移すことができる。一時変数 \mathbf{y} に結果を保存する。

最後に残ったのは、方程式系 $\mathbf{V}^T\mathbf{x} = \mathbf{y}$ を解くことだが、今回は行列 \mathbf{V} に関して再び直交性が役立つ。

まとめて予測値と予測誤差を計算しよう。

```
lm lstsq neq chol lu qr svd
1 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369 40.8369
```

これで、重要な最小二乗アルゴリズムが一巡した。この例の仕上げに、簡単に性能を見ておこう。

13.2.9 実行時間の比較

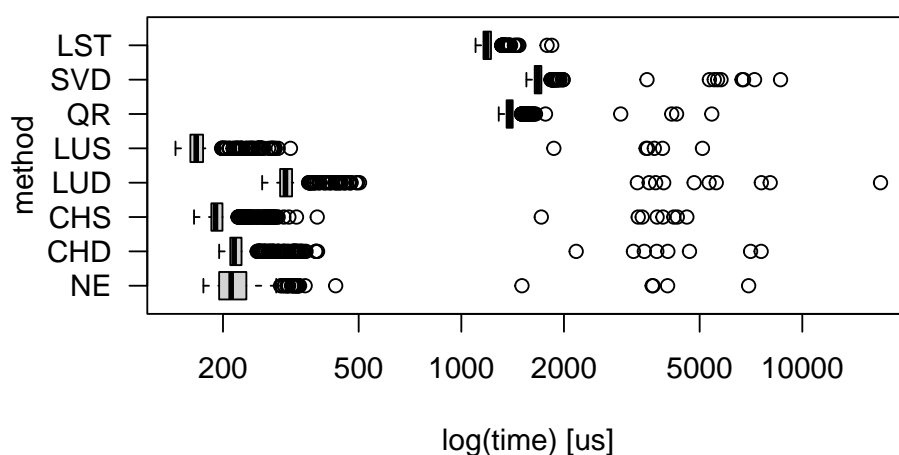
既に述べたように、本章の主眼は概念であり、性能ではない。しかしより大きなデータセットを扱い始めると、スピードが気になることは避けられない。また、これらの手法がどのくらい速いか確認すること自体は、興味深い。ということで、簡単な性能比較をしよう。ただし、結果を外挿しないように。扱うデータに対して、類似のコードを実行すること。

時間計測のために各アルゴリズムを対応する関数にまとめる。

microbenchmark パッケージを使って、これらの手法を計測する。

```
Warning in microbenchmark(ls_normal_eq(A, b), ls_cholesky_diy(A, b),
ls_cholesky_solve(A, : less accurate nanosecond times to avoid potential
integer overflows
```

Least Squares Benchmark



結論として、様々な方法で行列を分解することが、最小二乗問題を解くために役立つことが分かった。また、これらの手法の時間を計測する方法を簡単に示したが、スピードだけが重要ではない。同時に、解が信頼できるものであることも望まれる。これを専門用語では**数値安定性**と呼ぶ。

13.3 簡単な数値安定性の確認

既に条件数について議論した。数値安定性の概念は、考え方としては類似しているが、行列ではなくアルゴリズムを参照している。どちらの場合も、考え方は入力における微小な

変化は出力においても微小であるべきだというものだ。この話題について一冊全体割いた本がいくつもあるので、詳細に立ち入ることは避ける。^{*6}

代わりに、悪条件な最小二乗問題の例を用いる。つまり、行列が悪条件となっている。この例を用いて、既に議論したアルゴリズムの数値安定性について理解する。^{*7}

予測変数の行列は、悪条件になるように作られた 100 x 15 のファンデルモンド行列である。

条件数はとても大きい。

```
torch_tensor
2.27178e+10
[ CPUDoubleType{} ]
```

次に、予測対象を作成する。

上記の最小二乗問題では全ての手法で同じ RMSE を得た。今回の問題では興味深いことが起きる。以前に示したもののうち、「自作」関数だけに絞る。便宜上、再掲する。

それでは実行してみよう。

	algorithm	rmse
1	ls_normal_eq	4.730447e-04
2	ls_cholesky_diy	3.013914e-06
3	ls_lu_diy	1.170186e-07
4	ls_qr	3.436748e-08
5	ls_svd	3.436749e-08

これはかなり興味深い！ 正規方程式は簡単だが、問題が良条件でなくなった場合には良い選択ではないかもしれないことが明らかだ。コレスキーも LU 分解もかなり改善されているが、明らかな「勝者」は QR 分解と SVD だ。これら二つ（とそれぞれの派生型）が `linalg_lstsq()` で使われているのは当然だ。

^{*6} クロネッカーのデルタは、 $i = j$ のときに 1、そうでないときに 0 となる。

^{*7} n を t の代わりに異なる位置として使っている。信号その他離散化された値は、時間（あるいは空間）の離散点においてのみ「存在」するからである。文脈によって、少しきこえなく見えるかもしれないが少なくとも一貫している。

第 14 章

行列計算: 畳み込み

機械学習では、畳み込みや畳み込み層、畳み込みニューラルネットワークが話題となる。しかしながら、画像処理の章で説明したように、そのように呼んで行っているものは実際は少し異なり、相互相関である。

形式的には、差は重要ではなく、符号が逆になるだけである。意味の上では、これらは全く異なる。既に見たように、相互相関は類似性を見つけ、特徴抽出器として働く。畳み込みは、抽象的に特徴づけるのは難しい。信号処理において果たしている傑出した役割やその数学的な意義については、本が何冊も書けるほどである。ここでは、深く基礎を掘り下げることはしない。その代わりに、その作用について何らかの洞察を得ることにする。まず、関わる手順について考察し示すこと、そしてコードとして実装することによりこれを行う。前の章と同様に、読者が興味を持てるよう、焦点は理解と更なる探究のための基礎作りに合わせる。

14.1 畳み込みの意義

信号処理では、**フィルタ**を用いることにより、信号を変えて目的の形、例えば高振動数が除去されたものにする。手元に、時系列をフーリエ変換した表現、つまり振動数毎に大きさと位相があるとする。ある閾値より高い振動数をを除去したいとしよう。最も簡単な方法は、振動数のデータに 1 と 0 の列を掛けることだ。このようにすると、フィルタの作用は振動数領域で生じ、これが断然最も便利な方法だ。

もし、同様の結果を時間領域で得たい、つまり生の時系列データを扱うとしたらどうするか。その場合、フィルタの時間領域での表現（逆フーリエ変換から得られる）を見つけなくてはならない。そして、この表現を時系列に **畳み込む**。言い換えれば、時間領域における畳み込みは振動数領域における掛け算に対応する。この基本的な事実は常に利用される。

それでは、畳み込みは何にをするものかや、どのように実装されているかについて、より詳しく理解しよう。単一の次元から始めて、二次元の場合にどのようなになるか、少し探究する。

14.2 一次元の畳み込み

簡単な信号 x と簡単なフィルタ h を作る場所から取り掛かる。変数の名前は思いつきで決めたのではない、信号処理で h は、この後すぐに現れる用語 **インパルス応答**を表すのに通常使われる記号だ。

さて、`torch_conv1d()` があるので、これと呼んで何が起こるか見てみよう。畳み込みの定義によると、出力の長さは入力 of 長さ とフィルタの長さとの和より一つ少ない。`torch_conv1d()` を使って、長さ 6 の出力を得るには、フィルタの長さが 3 なので、両側に二つずつ広げる必要がある。

次のコードで、`view()` の呼び出しに気を取られてはいけない。これはただ `torch` が 3 次元の入力を必要としているためだ。最初と次の次元は、いつも通りそれぞれバッチとチャンネルを表す。

```
torch_tensor
(1,.,.) =
  1  2  2  2 -3 -4
[ CPUFloatType{1,1,6} ]
```

`torchconv1d()` は畳み込みではなく相互相関を計算するのだったのではないかと考えているかもしれない。R には `convolve()` があるので確認しよう^{*1}

```
[1]  1  2  2  2 -3 -4
```

結果は同じである。しかしながら、`convolve()` の説明を読むと

二つの列 x と y の畳み込みに対する通常の定義は次で与えられることに注意する。
`convolve(x, rev(y), type="o")`

明らかに、フィルタの要素の順序を逆転させる必要がある。

```
[1] -1 -2 -2 -2  3  4
```

確かに、今度の結果は異なっている。同様なことを `torch_conv1d()` でやってみよう。

```
torch_tensor
(1,.,.) =
 -1 -2 -2 -2  3  4
[ CPUFloatType{1,1,6} ]
```

^{*1} 引数 `type = "open"` を渡すことにより、単調で周期的でない畳み込みにすることを指定できる。

ここでも結果は `torch` と `R` で同じである。`convolve()` の説明の中の “Details” 部に
ある、あの簡潔な記述は相互総観と畳み込みの違いを完璧に捉えている。畳み込みでは、二
つ目の引数は逆順になっている。信号処理の言葉では、**反転** である。（「反転」の方がかな
り良い用語だ。高次元にも一般化できるからだ。）

技術的にはさはわずかだ。単に符号が逆転するだけである。しかし、数学的には、フィル
タが何であり、何をするかに関わるという意味で重要である。これは、この後説明する。

畳み込みの基礎となる演算は二つの方法で示すことができる。

14.3 畳み込みの二つの考え方

一つは、単一の出力に着目しどのように得られたか定めるものだ。つまり、どの入力要素
がその値に寄与したか、どのように組み合わせられたかを問う。これは「出力からの見方」
と呼ぶことができ、相互相関で馴染みのあるものである。

相互相関と同様に、次のようにも述べることもできる。フィルタが画像を「移動」し、個々
の画像中の位置（画素）で、周囲の入力画素と対応する「重ねられた」フィルタ値とを掛
け算し、全ての画素を足し合わせる。つまり、個々の出力画素は入力とフィルタ値の組の
間の内積の計算から得られる。

二つ目の考え方は、入力の観点から見たもの（「入力からの見方」と呼ぶ）だ。問いは、ど
のように各入力が出力に寄与するかである。この見方は、最初の見方よりも慣れを要する
が、単に馴染み、ニューラルネットワークの文脈でのよくある説明のされ方の問題かもし
れない。どちらにしても、入力からの見方は、畳み込みの数学的な意味について学ぶとい
う点で非常に有益である。

両方の見方を学ぶ。まず、馴染みのある、出力からの見方から始める。

14.3.0.1 出力からの見方

出力からの見方では、`torch_conv1d()` で `padding = 2` を指定したときのように、まず
入力信号の両側を延長する。指定通りに、インパルス応答を反転し、1, 0, -1 とする。
次に、「移動」を示す。

以下は、表形式でこれを示したものだ（Table 14.1）。

Table14.1: 畳み込み: 出力からの見方

信号	反転	IR			
0	1				
0	0	1			
1	-1	0	1		
2		-1	0	1	
3			-1	0	1

信号	反転 IR					
4				-1	0	1
0					-1	0
0						-1
結果		-1	-2	-2	-2	3 4

結局この話題について既に述べたように、書き出してみた結果に驚くことは何もない。次に入力からの見方を考える。

14.3.0.2 入力からの見方

入力からの考え方で重要なことは、入力信号をどのように考えるかである。個々の要素は、拡大縮小され、移動されたインパルスと見なされる。

インパルスは単位標本（またはインパルス）関数、デルタ（ δ ）で与えられる。この関数は値が 1 となる 0 を除いたすべての位置において 0 である。

$$\delta[n] = \begin{cases} 1, & n = 0 \\ 0, & n \neq 0 \end{cases}$$

これは、クロネッカーのデルタ δ_{ij} ^{*2} に似ており、一つの添字を 0 に固定したものである。

$$\delta[n] = \delta_{n0} = \delta_{0n}$$

つまり、この関数 $\delta[n]$ だけがあれば、 n が、例えば離散時間を表すとする、厳密に一つの信号値、時刻 $n = 0$ ^{*3}における値を表すことができ、その唯一とりうる値は 1 である。今、この操作に拡大縮小と移動を加える。

- 拡大縮小により、 $n = 0$ でのいかなる値も作れる。例えば $x_0 = 0 * \delta[n]$ 。
- 移動により、他の時刻での値に影響を与えることができる。例えば、時刻 $n = 3$ は $n - 3 = 0$ なので $\delta[n - 3] = \delta_{n3}$ で与えられる。
- 両方を組み合わせることにより、どんな時刻のどんな値でも表すことができる。例えば $x_5 = 1.11 * \delta[n - 5]$ 。

これまで、信号についてだけ考えてきた。フィルタはどうなるか。インパルスの特徴づけたと同様に、フィルタも * インパルス応答* で完全に記述できる [4]。インパルス応答は、定義により、入力がインパルスである（つまり、時刻 $n = 0$ で生じる）ときに現れるものである。信号に対して用いた表記にならい、 h をインパルス応答とすると次のように書ける。

^{*2} クロネッカーのデルタは、 $i = j$ のときに 1、そうでないときに 0 となる。

^{*3} n を t の代わりに異なる位置として使っている。信号その他離散化された値は、時間（あるいは空間）の離散点においてのみ「存在」するからである。文脈によって、少しきこえなく見えるかもしれないが少なくとも一貫している。

$$h[n] = h[n-0] \equiv h(\delta[n-0])$$

[4]: この章ではどこでも、フィルタについて話題にするときは、線型時間不変系だけを考えている。時間不変系への制約は、畳み込み操作に内在している。

扱っている例では、数列 $-1, 0, 1$ となるだろう。でも信号は 0 だけでなく、他の時刻においても表すことが必要であるように、フィルタも他の位置でも適用できなければならない。その目的のためにも、移動操作が用いられ、似た方法で定式化される。例えば、 $h[n-1]$ はフィルタが時刻 $1, n-1$ が 0 となる時刻に適用されることを意味する。このような移動は、俗に「ずらし」と呼んでいるものに相当する。

さて、残されている仕事は部品を組み合わせることだ。時刻 $n=0$ において、移動なしのインパルス応答を受け取り、信号の振幅で拡大縮小する。例においては、その値は 1 なので、 $1 * h[n-0] = 1 * [-1, 0, 1] = [-1, 0, 1]$ となる。他の時刻においては、インパルスオス等を考えている入力位置に移動し掛け算する。最後に、すべての入力位置からすべての寄与を得たら、足し合わせて畳み込まれた出力を得る。

次の表は、それを説明するものである Table 14.2。

Table14.2: 畳み込み: 入力からの見方

信号	インパルス入力	積
1	$h[n-0]$	$-1 \ 0 \ 1 \ 0 \ 0 \ 0$
2	$h[n-1]$	$0 \ -2 \ 0 \ 2 \ 0 \ 0$
3	$h[n-2]$	$0 \ 0 \ -3 \ 0 \ 3 \ 0$
4	$h[n-3]$	$0 \ 0 \ 0 \ -4 \ 0 \ 4$
和		$-1 \ -2 \ -2 \ -2 \ 3 \ 4$

個人的には、出力からの見方の方が理解しやすいものの、入力からの見方から得られる洞察が多いように感じる。特に、避けられない問い、なぜインパルス応答を反転させるのかに答えている。

謎の力のようなものからは程遠く、負号は信号の表現のされ方から生じた機械的な結果にすぎない。時刻 $n=2$ において測定された信号は $\delta[n-2]$ (2 から 2 を引くと 0) と表され、その信号に適用されるフィルタも $h[n-2]$ となる。

14.3.1 実装

出力からの見方についての説明から、どのようにコードにするか述べることはあまりないと思うかもしれない。入力ベクトルに互ってループし、期待される出力位置において内積を計算するのは容易に見える。しかし、これは多数のベクトル積を計算することを意味し、さらに入力が増えれば長いほどその計算が増加する。

幸運にも、もっと良い方法がある。単一次元（線型）畳み込みは、対角成分がいくつかの定数で、他の要素が 0 である Toeplitz 行列により計算される。フィルタが Toeplitz 行列

で定式化れれば、行われる積は Toeplitz 行列と入力との間の一つだけである。行列の列数は入力の長さと同じだけ必要である（さもなくば積が計算できない）といっても、行列は「ほぼ空」なので計算量は少ない。

扱っている例に対して、Toeplitz 行列を作ったものを示す。

```
torch_tensor
-1  0  0  0
 0 -1  0  0
 1  0 -1  0
 0  1  0 -1
 0  0  1  0
 0  0  0  1
[ CPUFloatType{6,4} ]
```

例の入力との積が期待通りの結果になるか確認しよう。

```
torch_tensor
-1
-2
-2
-2
 3
 4
[ CPUFloatType{6} ]
```

期待通りになった。次に二次元の場合を考えよう。概念的には、差はないが、実際の計算（「自作」と行列の両方）はもっと複雑になる。つまり、手計算の（一般化可能な）一部を示すことで満足し、計算の部分ではいちいち詳細を示さないことにする。

14.4 二次元での畳み込み

一次元と二次元の畳み込みがどのように類似しているか外見的に示すために、出力からの見方を考える。

14.4.1 仕組み（出力からの見方）

今回の例では、入力は二次元で、次のようなものである。

$$\begin{bmatrix} 1 & 4 & 1 \\ 2 & 5 & 3 \end{bmatrix}$$

フィルタについても同様である。ありうる例を示す。

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

出力からの見方では、フィルタを入力に対して「ずらす」。しかし、見やすくするために、一つの出力値（「画素」）に着目する。入力が $m1 \times n1$ で出力が $m2 \times n2$ なら出力の大きさは $(m + m2 - 1) \times (n1 + n2 - 1)$ である。つまり、例では 3×4 となる。 $(0, 1)$ の値を選ぶ。画像処理では、通常行を下から数える。

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & y_{01} & \cdot & \cdot \end{bmatrix}$$

存在しない（負）位置を含めた表で入力を示す。

位置 (x/y)	-1	0	1	2
1		1	4	1
0		2	5	3
-1				

対応する位置に値を置いたフィルタは次のように表される。

位置 (x/y)	-1	0	1	2
1		1	1	
0		1	-1	
-1				

一次元の場合のように、まずはフィルタを反転させる。反転は 180° の回転である。

位置 (x/y)	-1	0	1	2
1				
0		-1	1	
-1		1	1	

次にフィルタを必要ない出力位置に移動する。ここでは右に一つ移動して、上下には移動しない。

位置 (x/y)	-1	0	1	2
1				
0		-1	1	
-1		1	1	

(0, 1) での出力を計算する準備が完了した。計算は重なっている画像とフィルタ値の内積である。

位置 (x/y)	-1	0	1	2
1				
0		-1*2=-2	1*5=5	
-1				

最終結果は $-2 + 5 = 3$ である。

$$\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & 3 & \cdot & \cdot \end{bmatrix}$$

まだ値のない残りについても同様に計算できる。その計算は省略して、どのように実際の計算が行われるか見ることにする。

14.4.2 実装

二次元の畳み込みの実装にも Toeplitz 行列を使う。すでに述べたように、なぜすべての手順がそのなるか **厳密な形**では示さない。ここでの意図は扱っている例を示すことであり、その例を踏まえて、必要ならいろいろと探究してほしい。

14.4.2.1 手順 1: フィルタ行列の準備

フィルタを出力の大きさに合わせる。

```
0 0 0 0
1 1 0 0
1 -1 0 0
```

次にフィルタの全ての行について、下から Toeplitz 行列を作成する。

```
# H0
1 0 0
-1 1 0
0 -1 1
0 0 -1
```

```
# H1
1 0 0
1 1 0
```



```
0  1  1
0  0  1
```

```
# H2
0  0  0
0  0  0
0  0  0
```

コードは次のように書ける。

次に、これら三つの行列を組み合わせて、次のように**二重ブロック Toeplitz 行列**を作る。

```
H0  0
H1  H0
H2  H1
```

これをコードにする一つの方法は `torch_block_diag()` を二回使って二つの非 0 ブロックを作り、で結合することだ。

```
torch_tensor
 1  0  0  0  0  0
-1  1  0  0  0  0
 0 -1  1  0  0  0
 0  0 -1  0  0  0
 1  0  0  1  0  0
 1  1  0 -1  1  0
 0  1  1  0 -1  1
 0  0  1  0  0 -1
 0  0  0  1  0  0
 0  0  0  1  1  0
 0  0  0  0  1  1
 0  0  0  0  0  1
[ CPUFloatType{12,6} ]
```

```
torch_tensor
 1  0  0  0  0  0
-1  1  0  0  0  0
 0 -1  1  0  0  0
 0  0 -1  0  0  0
 1  0  0  1  0  0
 1  1  0 -1  1  0
 0  1  1  0 -1  1
 0  0  1  0  0 -1
```

```

0 0 0 1 0 0
0 0 0 1 1 0
0 0 0 0 1 1
0 0 0 0 0 1
[ CPUFloatType{12,6} ]

```

最終的な行列は二つの非 0 の「バンド」を持ち、二つの全てが 0 の対角線で分けられている。この行列は、行列積のために必要なフィルタの最終的な形である。

14.4.2.2 手順 2: 入力の前準備

この 12 x 6 行列と掛け算できるように、入力をベクトルに平坦化する必要がある。再び、行を下から順に進める。

```

torch_tensor
2
5
3
1
4
1
[ CPUFloatType{6} ]

```

14.4.2.3 手順 3: 行列積

ここまでで、畳み込みは単純な行列積に変わった。

```

torch_tensor
2
3
-2
-3
3
10
5
2
1
5
5
1
[ CPUFloatType{12} ]

```

残るは、出力を正しい二次元の構造に変形することだけだ。行を（下から）順に組み立てると、次のようになる。

$$\begin{bmatrix} 1 & 5 & 5 & 1 \\ 3 & 10 & 5 & 2 \\ 2 & 3 & -2 & -3 \end{bmatrix}$$

(0, 1) の要素に着目する、手計算の結果と一致していることがわかる。

ここで、`torch` の行列計算の話題を終える。しかし、次の話題であるフーリエ変換は、行列計算から余り離れていない。フーリエ変換は、時間領域の畳み込みが振動数領域の掛け算に対応することを覚えているだろうか。

この対応は、計算の高速化によく利用される。入力データをフーリエ変換し、結果をフィルタで掛け算し、フィルタされた振動数領域の表現を再び逆変換する。R の `colvolve()` の説明を見てみよう。出だしにはっきりと書いてある。

フーリエ変換を使って、二つの数列の畳み込みを計算する。

それでは、フーリエ変換に進もう。

参考文献

- Bronstein, M. M., J. Bruna, T. Cohen, and P. Velickovic, 2021: Geometric deep learning: Grids, groups, graphs, geodesics, and gauges^{*4}. *CoRR*, **abs/2104.13478**.
- Cho, D., C. Yoo, J. Im, and D.-H. Cha, 2020: Comparative assessment of various machine learning-based bias correction methods for numerical weather prediction model forecasts of extreme air temperatures in urban areas. *Earth and Space Science*, **7**, e2019EA000740, <https://doi.org/10.1029/2019EA000740>.

^{*4} <https://arxiv.org/abs/2104.13478>