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

List of Figures

Preface

This is a book about `torch`, the R interface to PyTorch. PyTorch, as of this writing, is one of the major deep-learning and scientific-computing frameworks, widely used across industries and areas of research. With `torch`, you get to access its rich functionality directly from R, with no need to install, let alone learn, Python. Though still "young" as a project, `torch` already has a vibrant community of users and developers; the latter not just extending the core framework, but also, building on it in their own packages.

In this text, I'm attempting to attain three goals, corresponding to the book's three major sections.

The first is a thorough introduction to core `torch`: the basic structures without whom nothing would work. Even though, in future work, you'll likely go with higher-level syntactic constructs when possible, it is important to know what it is they take care of, and to have understood the core concepts. What's more, from a practical point of view, you just need to be "fluent" in `torch` to some degree, so you don't have to resort to "trial-and-error-programming" too often.

In the second section, basics explained, we proceed to explore various applications of deep learning, ranging from image recognition over time series and tabular data to audio classification. Here, too, the focus is on conceptual explanation. In addition, each chapter presents an approach you can use as a "template" for your own applications. Whenever adequate, I also try to point out the importance of incorporating domain knowledge, as opposed to the not-uncommon "big data, big models, big compute" approach.

The third section is special in that it highlights some of the non-deep-learning things you can do with `torch`: matrix computations (e.g., various ways of solving linear-regression problems), calculating the Discrete Fourier Transform, and wavelet analysis. Here, more than anywhere else, the conceptual approach is very important to me. Let me explain.

For one, I expect that in terms of educational background, my readers will vary quite a bit. With R being increasingly taught, and used, in the natural sciences, as well as other areas close to applied mathematics, there will be those who feel they can't benefit much from a conceptual (though formula-guided!) explanation of how, say, the

Discrete Fourier Transform works. To others, however, much of this may be uncharted territory, never to be entered if all goes its normal way. This may hold, for example, for people with a humanist, not-traditionally-empirically-oriented background, such as literature, cultural studies, or the philologies. Of course, chances are that if you're among the latter, you may find my explanations, though concept-focused, still highly (or: too) mathematical. In that case, please rest assured that, to the understanding of these things (like many others worthwhile of understanding), it is a long way; but we have a life's time.

Secondly, even though deep learning has been "the" paradigm of the last decade, recent developments seem to indicate that interest in mathematical/domain-based foundations is (*again* – this being a recurring phenomenon) on the rise (Consider, for example, the Geometric Deep Learning approach, systematically explained in Bronstein et al. (2021), and conceptually introduced in *Beyond alchemy: A first look at geometric deep learning*[*1].) In the future, I assume that we'll likely see more and more "hybrid" approaches that integrate deep-learning techniques and domain knowledge. The Fourier Transform is not going away.

Last but not least, on this topic, let me make clear that, of course, all chapters have `torch` code. In case of the Fourier Transform, for example, you'll see not just the official way of doing this, using dedicated functionality, but also, various ways of coding the algorithm yourself – in a surprisingly small number of lines, and with highly impressive performance.

This, in a nutshell, is what to expect from the book. Before I close, there is one thing I absolutely need to say, all the more since even though I'd have liked to, I did not find occasion to address it much in the book, given the technicality of the content. In our societies, as adoption of machine/deep learning ("AI") is growing, so are opportunities for misuse, by governments as well as private organizations. Often, harm may not even be intended; but still, outcomes can be catastrophic, especially for people belonging to minorities, or groups already at a disadvantage. Like that, even the inevitable, in most of today's political systems, drive to make profits results in, at the very least, societies imbued with highly questionable features (think: surveillance, and the "quantification of everything"); and most likely, in discrimination, unfairness, and severe harm. Here, I cannot do more than draw attention to this problem, point you to an introductory blog post that perhaps you'll find useful: *Starting to think about AI Fairness*[*2], and just ask you to, please, be actively aware of this problem in public life as well as your own work and applications.

Finally, let me end with saying thank you. There are far too many people to thank that I could ever be sure I haven't left anyone out; so instead I'll keep this short. I'm extremely grateful to my publisher, CRC Press (first and foremost, David Grubbs and

[*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/

Curtis Hill) for the extraordinarily pleasant interactions during all of the writing and editing phases. And *very* special thanks, for their support related to this book as well as their respective roles in the process, go to Daniel Falbel, the creator and maintainer of `torch`, who in-depth reviewed this book and helped me with many technical issues; Tracy Teal, my manager, who supported and encouraged me in every possible way; and Posit (formerly, RStudio), my employer, who lets me do things like this for a living.

Sigrid Keydana

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き、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か、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 NVIDA 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} ]
```

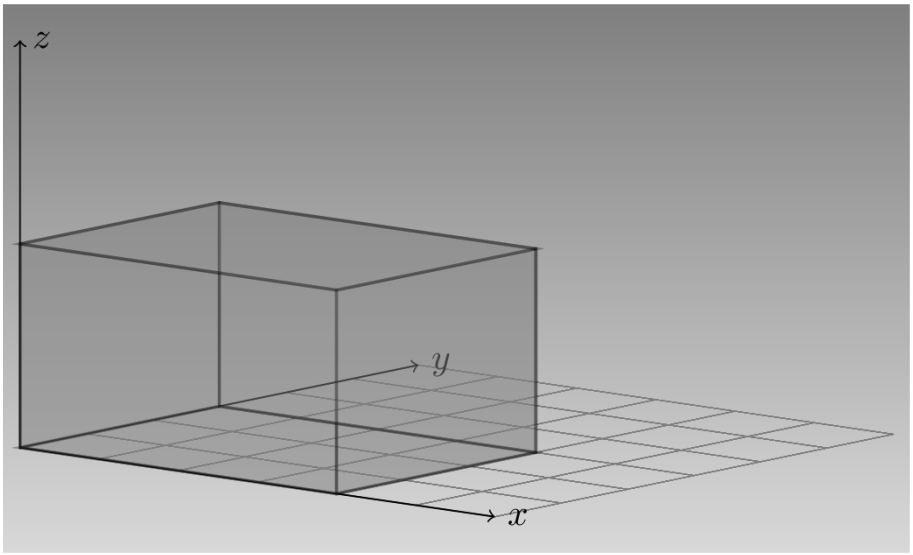
高次元のデータはどうするか。同様の方針に従って、配列を渡すことができる。

```
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 の大口生成函数が便利な状況は、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
-1.0208  0.7889  1.1905
 0.5677 -1.0507 -0.4647
 0.2805 -0.3574  0.4752
[ 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
 0.8186  0.7367  0.0261
 0.7388  0.7528  0.1627
 0.7158  0.6107  0.0052
[ 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函数がある。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
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{59855} ]
```

確かに、技術的にはこれはうまく動作する。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）函数や（$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()）する必要があっただろうか。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} ]
```

同じ函数 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: 個人に亙る。
- 次元 2: 時刻に亙る。
- 次元 3: 特徴に亙る。

テンソルは次のようになる。

```
torch_tensor
(1,.,.) =
  2.5353  0.0382
 -0.7716  1.0223
  0.6837  1.1270

(2,.,.) =
 -1.8819 -0.4631
  1.9013 -1.7242
 -0.0895 -1.7450

(3,.,.) =
 -0.1367  0.8091
  1.7435  0.8960
 -0.6630  1.4432

(4,.,.) =
 -0.9332  0.5071
 -0.4283 -0.5870
 -0.0472  0.2845
[ 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
 0.1594
 0.1340
[ 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
 0.8158  0.7292
-0.0234 -1.3108
 0.3146  1.0494
-0.4695  0.0682
[ 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,.,.) =
  0.0474  0.1396
  0.3702  0.9963
```

```
(2,.,.) =
  0.2790  0.8061
  0.0360  0.5737
[ 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
0.01 *
-3.2092
-32.2895
[ CPUFloatType{2} ]
```

```
torch_tensor
 0.2482 -0.8664
-1.3712  0.2325
[ CPUFloatType{2,2} ]

torch_tensor
(1,.,.) =
  0.1090  0.4185
 -0.5514  1.4651

(2,.,.) =
  0.6289  0.9432
 -0.1097 -1.7407
[ 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
 0.4185  1.4651
 0.9432 -1.7407
[ 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 x 1、12 x 2、6 x 4 などの行列
- 24 x 1 x 1, 12 x 2 x 1 などの 3 次元配列
- その他（24 x 1 x 1 x 1 x 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] "0x127cb6dc0"
```

```
[1] "0x127cb6dc0"
```

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] "0x127eb8ec0"
```

```
[1] "0x127eb8ec0"
```

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 unsqueeeze() 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
-0.4003
-0.2902
-1.4887
[ CPUFloatType{3} ]
```

```
torch_tensor
-0.4003 -0.2902 -1.4887
[ 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して行うと、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] "0x1060b9d80"
```

```
[1] "0x10604c880"
```

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ことになる。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
-0.2970  0.1450  0.2058  0.1768 -0.5657
 0.3524 -0.5062 -0.4432 -0.3545 -0.0800
-0.6479 -0.4224 -0.8937  0.3483  0.6502
[ 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。

ルールを系統だてることにする。

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 x 7 x 1、もう一つは 1 x 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、なんらかの損失函数を最小化と関係している。これは、さらに函数の **微分** の計算を必要とする。ここで、利用者として、個々の微分の函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` は微分の函数表現を作ったり、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、その前に一歩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、**損失函数** 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函数を想像してみる Figure 4.1。

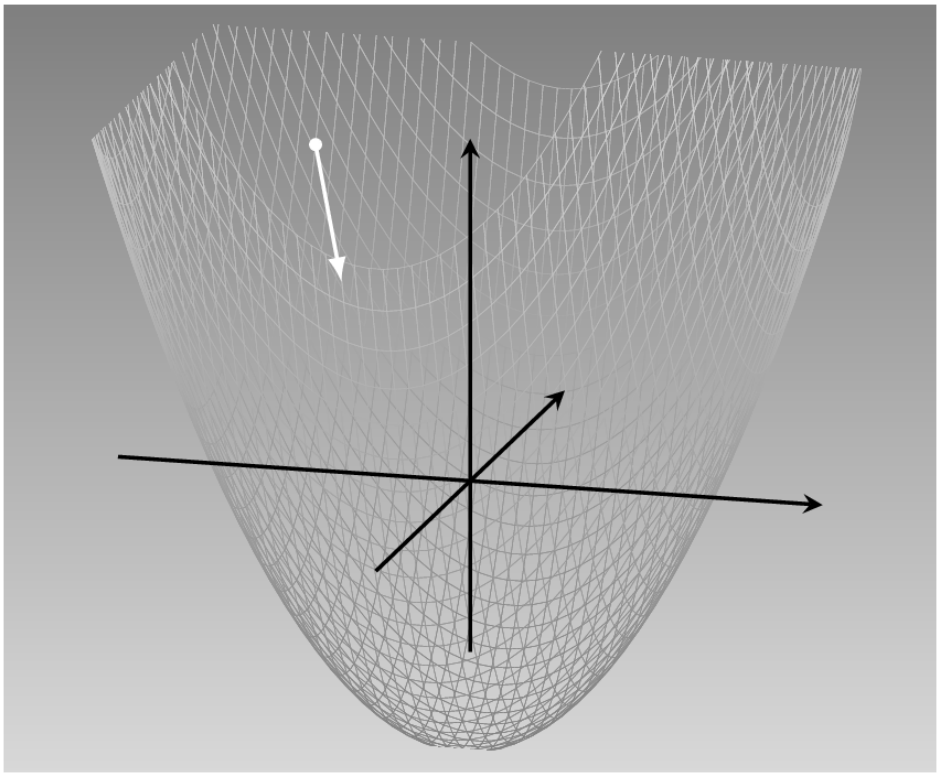


Figure4.1: 仮想的な損失函数（放物面）。

これは二変数の二次函数 `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 に関する函数の微分は、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 $\partial f/x_1 = 0.4x_1$ 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。

この方法は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。

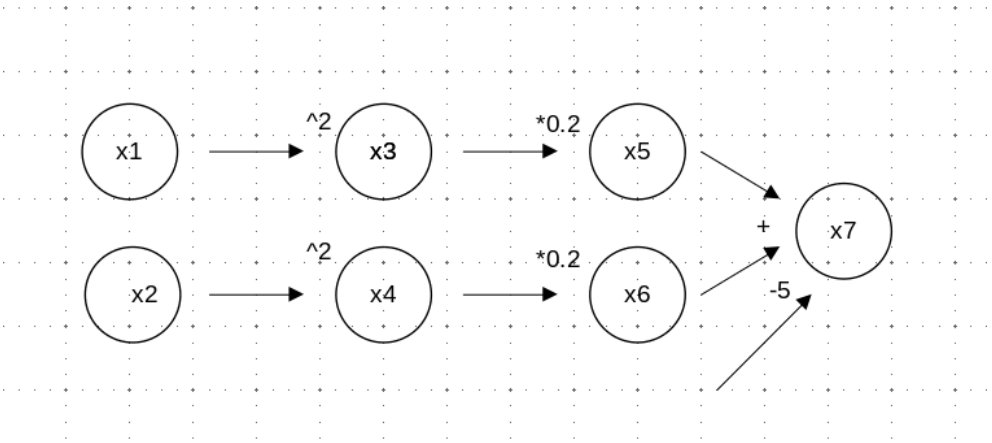


Figure4.2: 計算グラフの例

Figure 4.2 は上の函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。`x1` と `x2` は入力ノードで、対応する函数のパラメタは x_1 と x_2 である。`x7` は函数の出力で、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一種で、まず函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 `x1` と `x2` 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。

- `x7` で、`x5` と `x6` 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。
- `x5` から左に動き、`x3` 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 `x3` 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` 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函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。$\nabla f = (\partial f/\partial x_1, \partial f/\partial x_2)^{\mathrm{T}} = (0.4x_1, 0.4x_2)^{\mathrm{T}}$

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 x1 と x2 を作成する。これは興味のあるパラメタを模している。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 x3 と x6 を作るには、二乗して掛け算をする。最後に x7 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() を呼べぱ、x7 が'x1 と x2'にどのように依存するかが決まる。

この呼び出しにより、x1 と x2 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 x7 の x1 と x2 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 x3 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 retarin_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 を使った函数の最小化

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。

- 函数を最小化する（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、**ローゼンブロック函数** は古典である。この函数は二つの変数をとり、`(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。

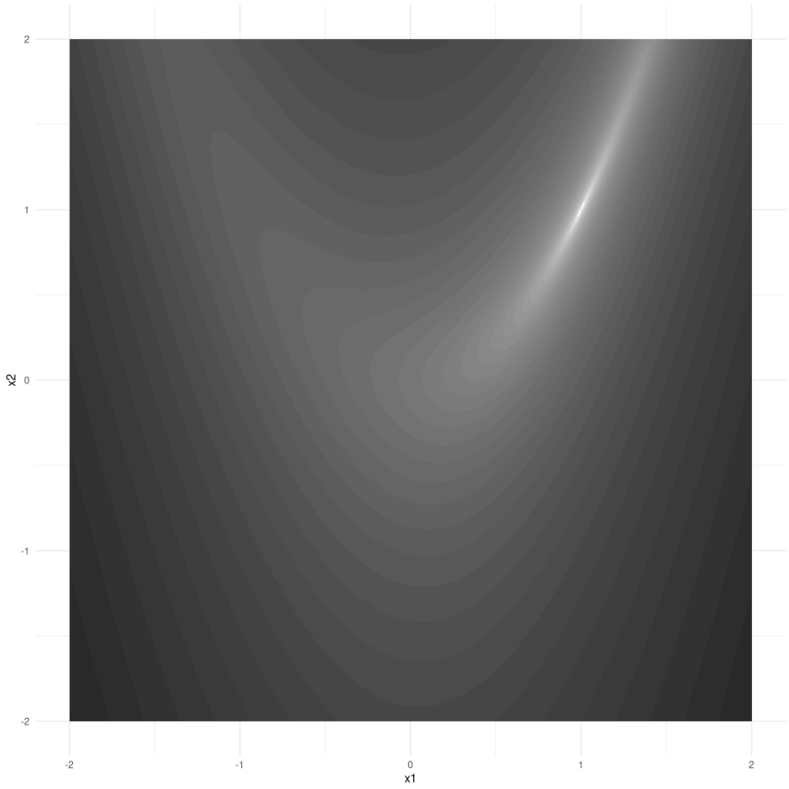


Figure5.1: ローゼンブロック函数

函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 (x1, x2) から出発し、ローゼンブロック函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函数の勾配を計算し、それを使って逆方向にに進む。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) {

  # 函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、つまり函数の入力であり、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函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$grad$zer_()
})
```

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、函数値は 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 **損失函数**、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、lm ではない。実際にニューラネ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。適切な函数が求めるものだ。

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 x 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 x 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
-0.4802
-0.3520
 0.0824
 0.6544
-0.9871
 0.9157
 0.8466
-1.0116
 1.0486
 0.1496
... [the output was truncated (use n=-1 to disable)]
[ CPUFloatType{100,1} ][ grad_fn = <AddBackward0> ]
```

数学の表記では、ここでは次の函数を実装した。

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

これ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、つまり層が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、上に示した函数 f を適用する。そして、別の函数を適用する。この函数は隠れそう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函数の合成である。二番目の函数 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、`w2` は次のようになる。

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

バイアス `b2` は `b1` 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、今の構造では函数を連鎖、あるいは合成しているのはよいが、これらの函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 活性化函数

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。

ここで活性化函数、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函数は **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函数は $ReLU$、Rectified Linear Unit（正規化線型函数）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、層を重ねて活性化函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。その概念とは損失函数だ。

6.4　損失函数

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。函数を最小化するとき、前の章で行ったように、これは現在の函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函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
9.99998e-05
[ CPUFloatType{} ]
```

損失函数が決まれば、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。

函数の最小化のように、適切な学習率（差し引く勾配の比率）は実験的によって決める。

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:  84.46025
Epoch:  20    Loss:  75.19603
Epoch:  30    Loss:  67.27222
Epoch:  40    Loss:  60.45836
Epoch:  50    Loss:  54.57803
Epoch:  60    Loss:  49.43273
Epoch:  70    Loss:  44.94943
Epoch:  80    Loss:  41.02125
Epoch:  90    Loss:  37.56583
Epoch:  100    Loss:  34.51753
Epoch:  110    Loss:  31.81525
```

```
Epoch:  120    Loss:  29.41616
Epoch:  130    Loss:  27.28504
Epoch:  140    Loss:  25.36717
Epoch:  150    Loss:  23.62668
Epoch:  160    Loss:  22.06767
Epoch:  170    Loss:  20.66031
Epoch:  180    Loss:  19.39462
Epoch:  190    Loss:  18.25359
Epoch:  200    Loss:  17.22379
```

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。この例はすばらしい性能を示すために作られたのでなく、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ら粗削りで、（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、自作の損失函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、簡単な函数呼び出し `nnf_relu()` で表されていた。活性化函数は、函数であり、与えられた入力 **x** に対して出力 **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 x 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
-0.0147  0.3684  0.2736 -0.2089  0.4193
-0.4468  0.2964  0.3191 -0.1244 -0.2027
-0.1079 -0.2307 -0.1835 -0.3296 -0.0354
 0.3423 -0.1737  0.2863  0.3886  0.2197
 0.2553 -0.2489 -0.4259 -0.1407 -0.2224
 0.3528  0.2467  0.1707  0.3188  0.0825
 0.1874 -0.2990 -0.3938 -0.2101 -0.1987
-0.0030 -0.1157  0.0010  0.1355  0.1953
 0.2743  0.3365  0.3691  0.2878  0.2355
-0.3686 -0.2526 -0.4133  0.2018 -0.0871
-0.1309 -0.1604  0.3630  0.0697 -0.3561
-0.3172 -0.3863  0.2955 -0.1978  0.3080
 0.0052 -0.0893 -0.3018  0.1636  0.2563
 0.3366  0.2008  0.0968 -0.4063  0.0166
-0.0800  0.4313  0.1775  0.0939  0.3849
-0.1033  0.4067 -0.0720  0.0061 -0.4168
[ CPUFloatType{16,5} ][ requires_grad = TRUE ]

torch_tensor
 0.2320
-0.3196
-0.0459
-0.3535
-0.0767
 0.0786
 0.1897
 0.2338
 0.0281
 0.0692
-0.2815
 0.1223
```

```
-0.2371
-0.4155
-0.0058
 0.2757
[ CPUFloatType{16} ][ requires_grad = TRUE ]
```

この時点で、ちょっと立ち止まってほしい。torch が返した重み行列の大きさは 16 x 5 で、白紙からコードを書いたときの 5 x 16 ではない。これは基礎となる C++ 実装 libtorch に由来する。性能のため、libtorch の線型モジュールは重みとバイアスを転置で格納している。R からは、この点を指摘しておくことくらいしかできない。混乱が軽減されることを望む。

続けよう。このモジュールにデータを適用するには函数のように「呼び出す」だけだ。

```
[1] 50 16
```

これが順伝播だ。勾配計算はどうするのか。以前は、勾配計算の「入力」として必要なテンソルを作るとき、torch に明示的に requires_grad = TRUE を渡す必要があった。組込 nn_module() ではその必要はない。すぐに output が backward() でどのような計算をすべきか調べることができる。

```
AddmmBackward0
```

確認のため、output に基づいて、適当な損失を計算して、backward() を読んでみよう。線型モジュールの weight テンソルには埋められた grad フィールドがあることが分かる。

```
torch_tensor
0.01 *
 1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
  1.8323  0.3640  0.2694  1.6031 -0.5630
```

```
  1.8323  0.3640  0.2694  1.6031 -0.5630
[ 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() と同様でモジュール、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
0.01 *
 2.3016
  0.9121
  4.5638
  3.0168
 -9.3430
[ 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。呼び出させれると、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函数を学ぶ次の章まで待ってもよい。それが済んだら、函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して調べる。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。

いつも必要であるを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。訓練の反復に亙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$bacward()

# 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函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に用いてきた。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

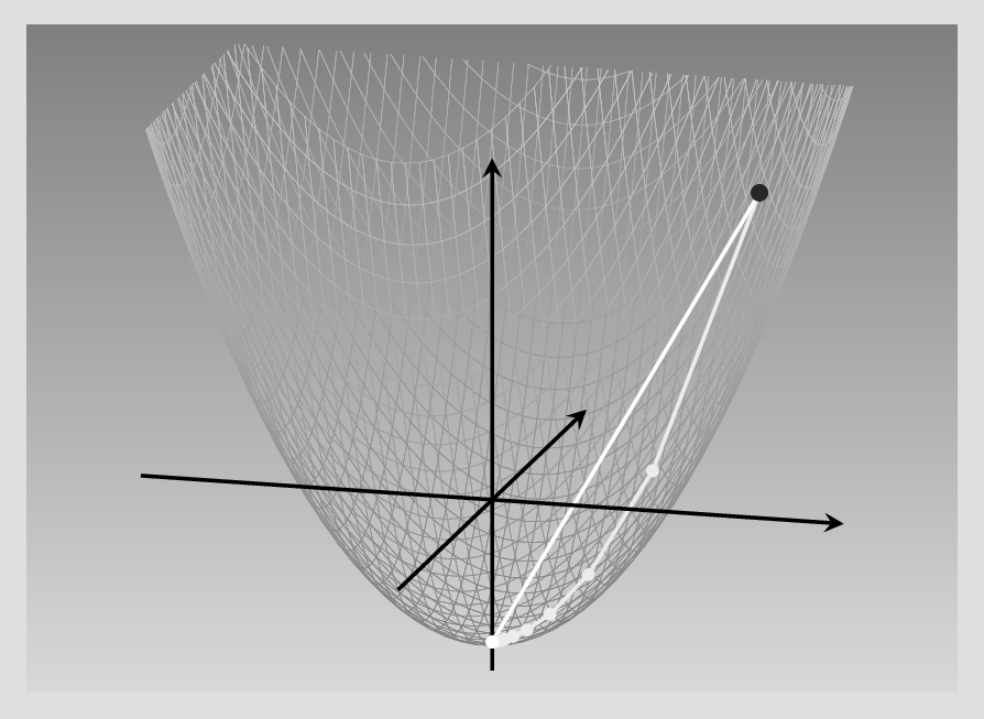
最急降下や確率的勾配降下法（SGD: stochastic gradient descent）としても知られている。勾配は偏微分のベクトルで、各入力特徴量に対して一つの要素がある。これは、関数が最も増加する方向を表す。その反対方向に進めば、最も速く降下できる。そうだろうか。

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函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函数があった。そのときは指摘しなかったが、この特定の函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。函数は $f(x_1, x_2) = 0.2x_1^2 + 0.2x_2^2 - 5$ 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、一歩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函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。

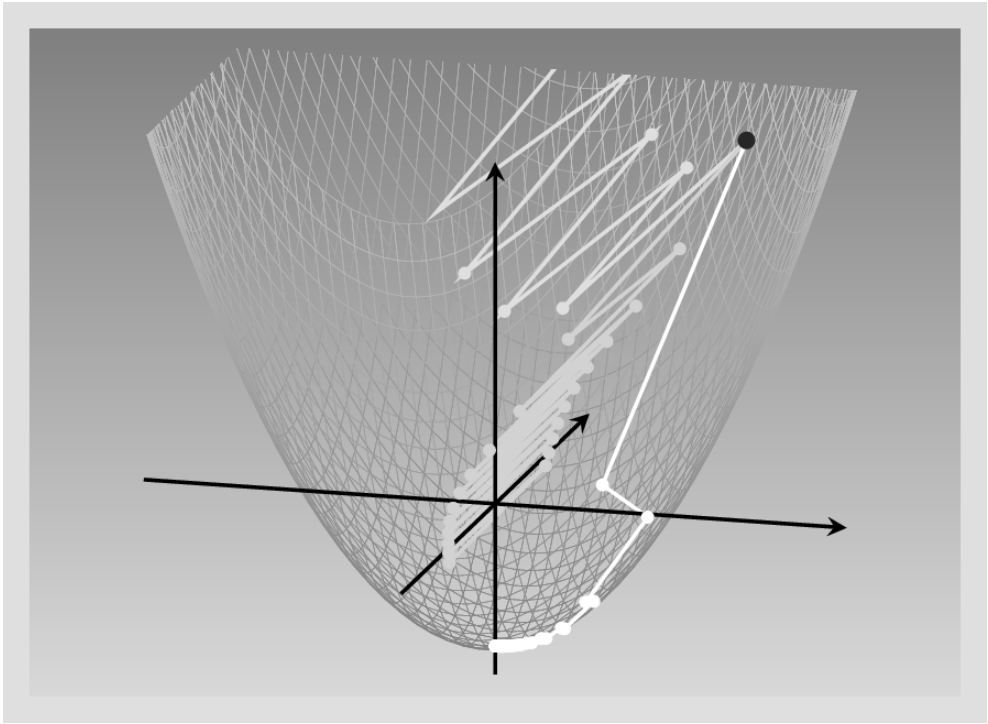


Figure8.1: わずかに異なる学習率を用いた非等方的な放物面上の最急降下

これは非常に説得力がある。二変数だけのよくある函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、損失函数は **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。

次に非対称函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。

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

二段階目は $\mathbf{x}$ の更新を「折衷」された速度でする。

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

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。

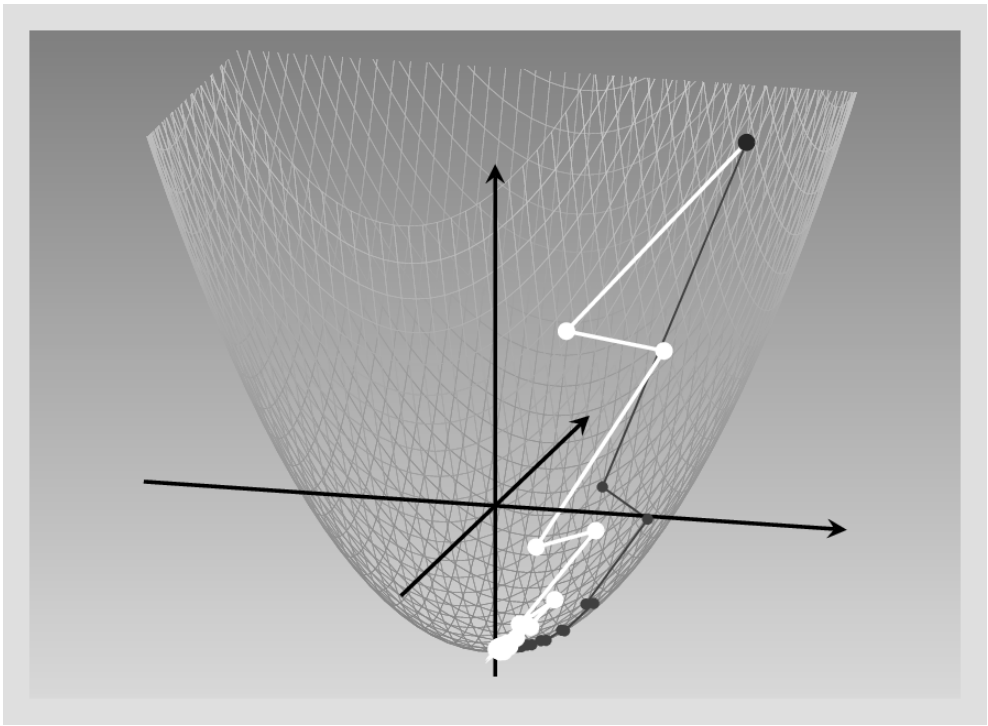


Figure8.2: 慣性付 SGD（白）と素の SGD（灰）との比較

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。

$$\mathbf{s}_i^{(k)} = \sum_{j=1}^{k} (g_i^{(j)})^2 \tag{8.3}$$

（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] ここで ϵ は小さな値で 0 で割ることを防止する。

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

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。

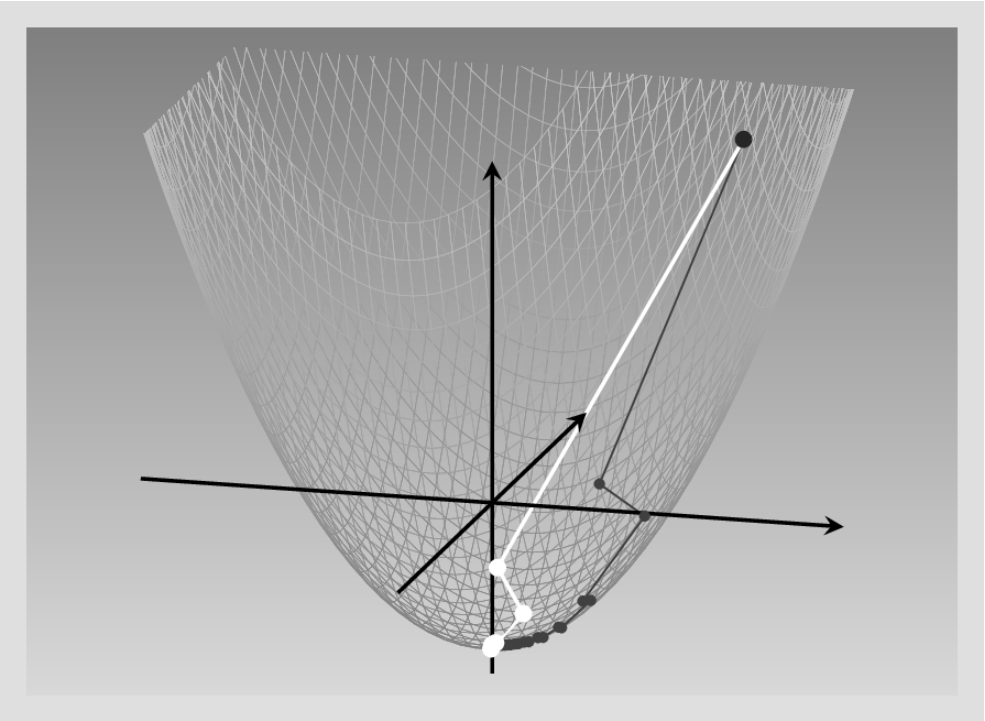


Figure8.3: Adagrad（白）と素の SGD（灰）との比較

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。

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

更新は Adagrad と同様にする。

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

この方法では、各パラメタは適切な重みが得られ、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。

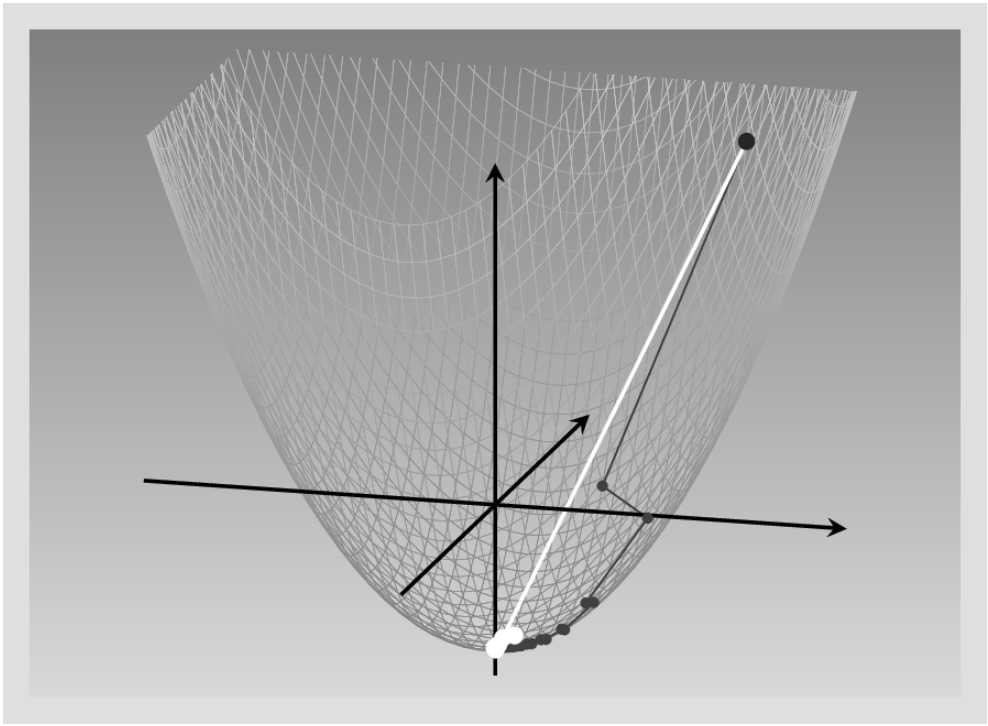


Figure8.4: RMSProp（白）と素の SGD（灰）との比較

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、勾配の指数函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。

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

[*2] 実装は通常追加の手順があるが、ここではその詳細は必要ない。

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

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

$$x_i^{(k+1)} = x_i^{(k)} - \frac{lrv_i^{(k+1)}}{\epsilon + \sqrt{s_i^{(k+1)}}} g_i^{(k)} \tag{8.9}$$

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

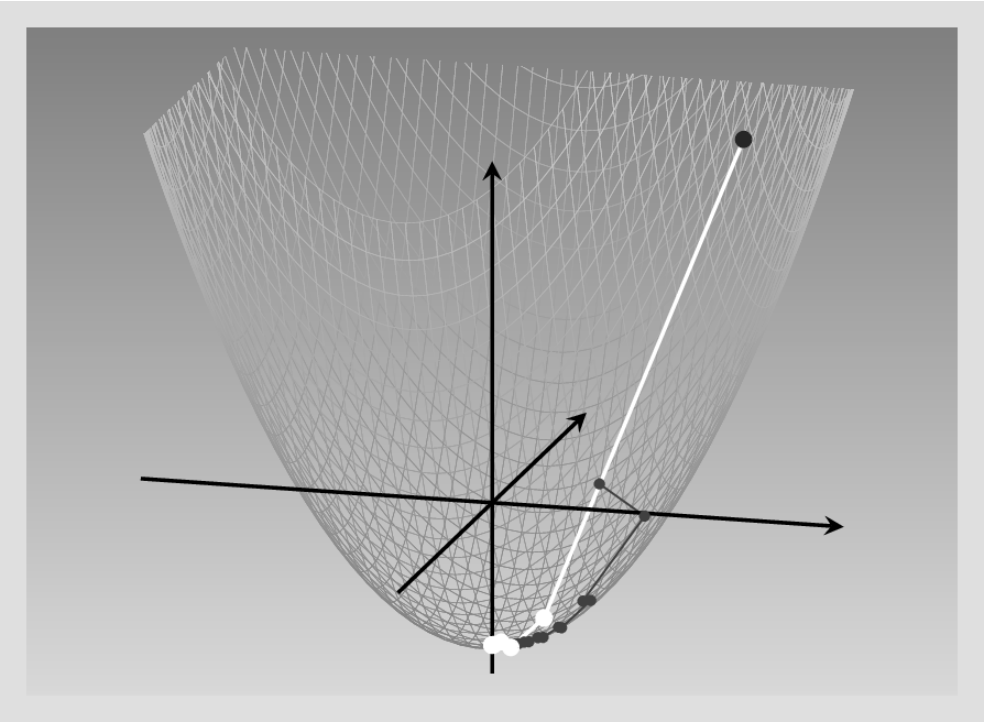


Figure8.5: Adam（白）と素の SGD（灰）

次の章で扱う損失函数は、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

損失函数

損失函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、損失函数を使ってきた。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 組込の損失函数の作り方である。もう一つは、どの損失函数を選択するかについてだ。

9.1 torch の損失函数

torch では、損失函数は nn_または nnf_で始まる。

nnf_を使う場合、直接函数呼び出しを行う。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() という組込函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函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、いくつかの損失函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　どの損失函数を選択すべきか。

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函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、分布適合損失函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 を用いた函数の最適化

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、函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。

ローゼンブロック函数を最小化したときに何をしたか思い返すと、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（傾きが再び上り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 $\mathrm{d}y/\mathrm{d}x = 20x$ 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、問題となっている函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、もう一つの概念を説明しておく。

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。問題となる函数次第で、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をかける。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**函数として書く。そしてこの函数を微分して、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、ローゼンブロック函数を最適化で、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ク函数の最小化

ローゼンブロック函数を再掲する。

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** での函数値を計算する。
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。自作での手続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() は無名函数、クロージャに渡す必要がある。前の勾配を 0 にするもの、函数呼び出し、勾配計算が全てクロージャの中で行われる。

```
calc_loss <- function() {
  optimzer$zero_grad()
  value <- rosenbrokc(x_star)
  value$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、アルゴリズムは函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:  2.478375
エポック:  20  損失:  2.316162
エポック:  30  損失:  2.161446
エポック:  40  損失:  2.014008
エポック:  50  損失:  1.874141
エポック:  60  損失:  1.741656
エポック:  70  損失:  1.617546
エポック:  80  損失:  1.50241
エポック:  90  損失:  1.397426
エポック:  100  損失:  1.303001
エポック:  110  損失:  1.220005
エポック:  120  損失:  1.149338
エポック:  130  損失:  1.090693
エポック:  140  損失:  1.04337
エポック:  150  損失:  1.006247
エポック:  160  損失:  0.9770314
エポック:  170  損失:  0.9545125
エポック:  180  損失:  0.9365191
エポック:  190  損失:  0.9216542
エポック:  200  損失:  0.9091917
```

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。

参考文献

Bronstein, Michael M., Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. “Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.” *CoRR* abs/2104.13478. https://arxiv.org/abs/2104.13478.