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

[*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/

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 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

テンソル

1.1 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。

1.2　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。

1.2.1　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} ]
```

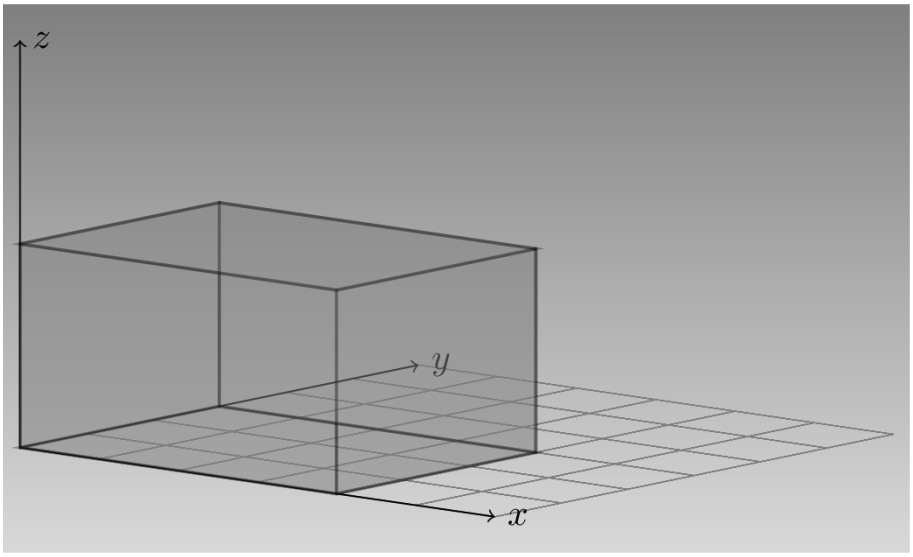
高次元のデータはどうするか。同様の方針に従って、配列を渡すことができる。

```
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。



{#fig-tensor-432} 私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。

1.2.2 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
 0.6775 -0.5989  0.5278
-1.6457 -1.4489 -0.4530
 0.6508 -1.4873  0.3159
[ 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
 0.7194  0.1030  0.4637
 0.6942  0.8178  0.5947
 0.3235  0.5861  0.3299
[ 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} ]
```

1.2.3　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。こられは 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 not handled
```

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 型に本管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_nanquantil() 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。

1.3　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() が行う、引数のスカラ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。

1.3.1 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,.,.) =
  0.3435  0.0081
 -1.1450 -0.1330
 -0.9007 -0.2518

(2,.,.) =
  1.9531 -0.5289
  0.8893  0.9942
 -0.3371 -0.2754

(3,.,.) =
 -0.7762 -0.7614
 -0.1577  1.5952
 -0.0018  1.2559

(4,.,.) =
 0.01 *
  6.9891  0.3290
 -9.5083 -68.2955
```

```
  -50.6621  9.0358
[ 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
-0.0554
 0.1095
[ 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
-0.5674 -0.1256
 0.8351  0.0633
-0.3119  0.6965
-0.1773 -0.1964
[ CPUFloatType{4,2} ]
```

ここで圧縮されたは時刻である。

1.4　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。

1.4.1　「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,.,.) =
  0.2994  0.0438
  0.0402  0.6197

(2,.,.) =
  0.6779  0.7232
  0.6813  0.7698
[ 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。

1.4.2 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
-2.5471
-1.6286
[ CPUFloatType{2} ]

torch_tensor
 0.4041  1.7152
 1.0903 -0.5575
[ CPUFloatType{2,2} ]

torch_tensor
(1,.,.) =
  1.6793  0.4921
 -0.6997 -0.0203

(2,.,.) =
 -1.5741  0.2728
 -0.2063  0.1640
[ 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
 0.4921 -0.0203
 0.2728  0.1640
[ 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。

1.5　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ほ同じ場所に格納している。

```
[1] "0x12e2d5f80"
```

```
[1] "0x12e2d5f80"
```

どのように実現されているか少し議論する。

1.5.1 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] "0x1671b8540"
```

```
[1] "0x1671b8540"
```

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
-0.1978
-0.2463
-0.2585
[ CPUFloatType{3} ]
```

```
torch_tensor
-0.1978 -0.2463 -0.2585
[ 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] "0x1670e6980"
```

```
[1] "0x1670cdbc0"
```

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。

1.6 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
-0.4556 -0.3412  0.1051  0.9877 -0.0622
-0.7055 -0.3447 -0.2255 -0.5051  0.6169
 0.1258 -0.0381 -0.1058 -1.1243  0.0016
[ 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。

1.6.1　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 2 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。

2.1 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 2.1。

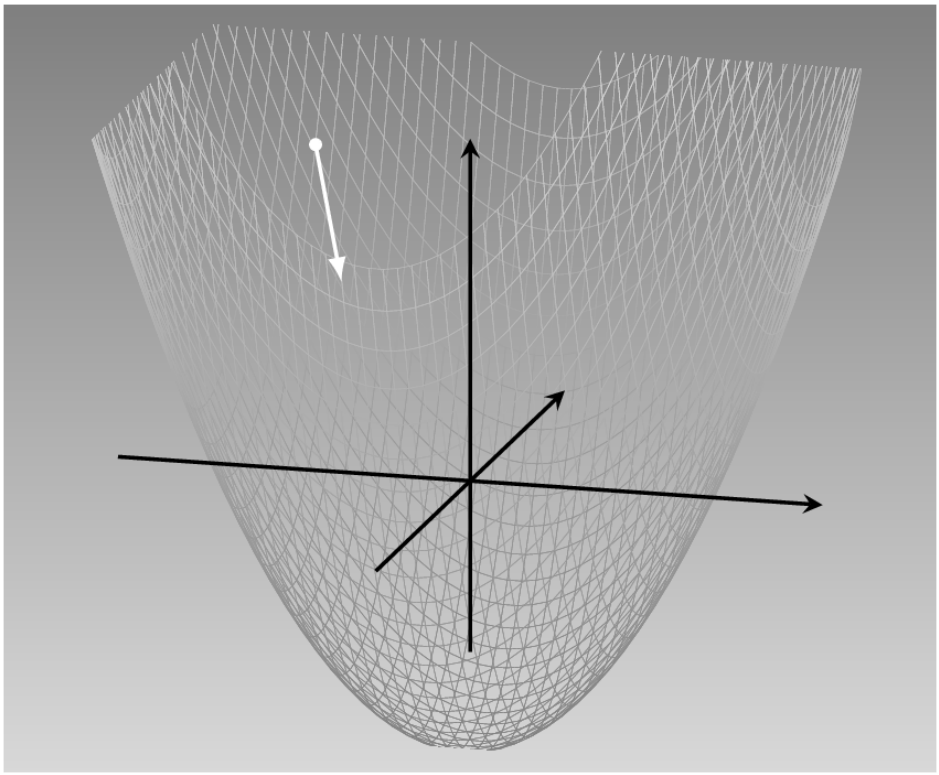


Figure2.1: 仮想的な損失函数（放物面）。

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。

2.2 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。

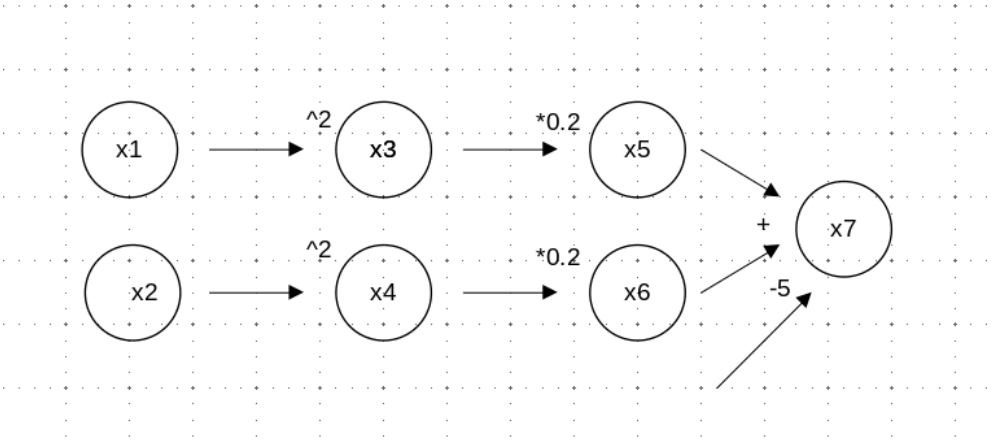


Figure2.2: 計算グラフの例

Figure 2.2 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。`xu` 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。

2.3　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。これは興味のあるバ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。

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.