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 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。

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。

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。

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。

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。

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。

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。

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。

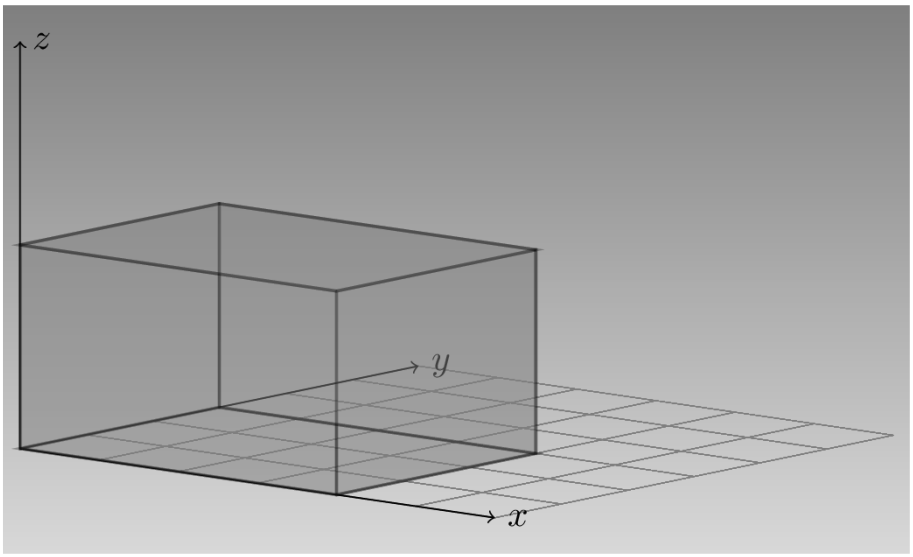
R の行列を同様に渡せばよい。

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。

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。

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。

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。

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。

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。

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。

そしてこれは対角行列。

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。

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。

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。

これはうまくいった。別なものを試そう。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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）。

それでは torch で同じことをする。総和から始める。

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。

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。

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。

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。

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。

ここで圧縮されたは時刻である。

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.