```
#hide
! [ -e /content ] && pip install -Uqq fastbook
import fastbook fastbook.setup_book()
```

In [ ]:

```
#hide
from fastai.vision.all import *
from fastbook import *
matplotlib.rc('image', cmap='Greys')
```

Under the Hood: Training a Digit Classifier

アンダーフード：数字分類器のトレーニング

Having seen what it looks like to actually train a variety of models in Chapter 2, let's now look under the hood and see exactly what is going on. We'll start by using computer vision to introduce fundamental tools and concepts for deep learning.

第 2 章で様々なモデルを実際に学習させるとどのようになるかを見てきましたが、今度はその裏側を覗いて、何が行われているかを正確に見てみましょう。まずはコンピュータビジョンを使って、ディープラーニングの基本的なツールや概念を紹介します。

To be exact, we'll discuss the roles of arrays and tensors and of broadcasting, a powerful technique for using them expressively. We'll explain stochastic gradient descent (SGD), the mechanism for learning by updating weights automatically. We'll discuss the choice of a loss function for our basic classification task, and the role of mini-batches. We'll also describe the math that a basic neural network is actually doing. Finally, we'll put all these pieces together.

具体的には、配列やテンソルの役割と、それらを表現的に使うための強力なテクニックであるブロードキャストについて説明します。また、重みを自動で更新して学習する仕組みである確率的勾配降下法（SGD）を説明します。基本的な分類タスクに対する損失関数の選択と、ミニバッチの役割について説明します。また、基本的なニューラルネットワークが実際に行っている計算についても説明します。最後に、これらのピースをすべてまとめます。

In future chapters we'll do deep dives into other applications as well, and see how these concepts and tools generalize. But this chapter is about laying foundation stones. To be frank, that also makes this one of the hardest chapters, because of how these concepts all depend on each other. Like an arch, all the stones need to be in place for the structure to stay up. Also like an arch, once that happens, it's a powerful structure that can support other things. But it requires some patience to assemble.

今後の章では、他のアプリケーションについても深く掘り下げ、これらの概念やツールがどのように一般化されるかを見ていきたいと思います。しかし、この章は基礎石を積むことが目的です。率直に言って、この章は最も難しい章のひとつでもあります。なぜなら、これらの概念はすべて互いに依存しているからです。アーチのように、すべての石が揃っていないと、構造体は立ち上がりません。また、アーチのように、ひとたび石が揃えば、他のものを支えることができる強力な構造物になります。でも、組み立てるにはちょっと忍耐が必要です。

Let's begin. The first step is to consider how images are represented in a computer.

では、はじめましょう。まず、コンピュータの中で画像がどのように表現されているかを考えることから始めます。

Pixels: The Foundations of Computer Vision

ピクセルのこと コンピュータビジョンの基礎

In order to understand what happens in a computer vision model, we first have to understand how computers handle images. We'll use one of the most famous datasets in computer vision, MNIST, for our experiments. MNIST contains images of handwritten digits, collected by the National Institute of Standards and Technology and collated into a machine learning dataset by Yann Lecun and his colleagues. Lecun used MNIST in 1998 in Lenet-5, the first computer system to demonstrate practically useful recognition of handwritten digit sequences. This was one of the most important breakthroughs in the history of AI.

コンピュータビジョンのモデルで何が起こっているのかを理解するためには、まずコンピュータがどのように画像を扱っているのかを理解する必要があります。今回は、コンピュータビジョンで最も有名なデータセットの 1 つである MNIST を実験に使用することにします。MNIST は、米国国立標準技術研究所が収集した手書きの数字の画像を、Yann Lecun 氏らが機械学習のデータセットとして照合したものです。Lecun は 1998 年、手書きの数字列の実用的な認識を初めて実証したコンピュータシステム Lenet-5 で MNIST を使用しました。これは、AI の歴史において最も重要なブレークスルーの 1 つであった。

Sidebar: Tenacity and Deep Learning

サイドバー：粘り強さとディープラーニング

The story of deep learning is one of tenacity and grit by a handful of dedicated researchers. After early hopes (and hype!) neural networks went out of favor in the 1990's and 2000's, and just a handful of researchers kept trying to make them work well. Three of them, Yann Lecun, Yoshua Bengio, and Geoffrey Hinton, were awarded the highest honor in computer science, the Turing Award (generally considered the "Nobel Prize of computer science"), in 2018 after triumphing despite the deep skepticism and disinterest of the wider machine learning and statistics community.

ディープラーニングのストーリーは、一握りの熱心な研究者による粘り強さと気概の賜物である。初期の期待（と誇大広告！）の後、ニューラルネットワークは 1990 年代から 2000

年代にかけて人気を失い、ほんの一握りの研究者がうまく機能させるために努力を続けました。そのうちの 3 人、ヤン・ルクン、ヨシュア・ベンジオ、ジェフリー・ヒントンは、より広い機械学習や統計学のコミュニティが深い懐疑と無関心だったにもかかわらず勝利し、2018 年にコンピュータサイエンスの最高の名誉であるチューリング賞（一般に「コンピュータサイエンスのノーベル賞」と考えられています）を受賞しました。

Geoff Hinton has told of how even academic papers showing dramatically better results than anything previously published would be rejected by top journals and conferences, just because they used a neural network. Yann Lecun's work on convolutional neural networks, which we will study in the next section, showed that these models could read handwritten text—something that had never been achieved before. However, his breakthrough was ignored by most researchers, even as it was used commercially to read 10% of the checks in the US!

Geoff Hinton は、以前に発表されたものよりも劇的に優れた結果を示す学術論文でさえ、ニューラルネットワークを使用しているというだけで、トップジャーナルやカンファレンスから却下されることを語っています。次のセクションで紹介する Yann Lecun の畳み込みニューラルネットワークの研究は、このモデルが手書きのテキストを読むことができることを示した。しかし、彼の成果は、アメリカの小切手の 10％を読み取るという商業的な利用がなされたにもかかわらず、ほとんどの研究者に無視された！

In addition to these three Turing Award winners, there are many other researchers who have battled to get us to where we are today. For instance, Jurgen Schmidhuber (who many believe should have shared in the Turing Award) pioneered many important ideas, including working with his student Sepp Hochreiter on the long short-term memory (LSTM) architecture (widely used for speech recognition and other text modeling tasks, and used in the IMDb example in <<chapter_intro>>). Perhaps most important of all, Paul Werbos in 1974 invented back-propagation for neural networks, the technique shown in this chapter and used universally for training neural networks (Werbos 1994). His development was almost entirely ignored for decades, but today it is considered the most important foundation of modern AI.

この 3 人のチューリング賞受賞者のほかにも、今日の私たちの地位を築くために戦ってきた研究者はたくさんいます。例えば、ユルゲン・シュミッドフーバー（チューリング賞を受賞すべきと考える人も多い）は、弟子のセップ・ホクライターと共同で長短記憶（LSTM）アーキテクチャ（音声認識やその他のテキストモデリング作業に広く使われており、<>の IMDb 例で使用）を開発するなど、多くの重要なアイデアを先駆的に発表しています。おそらく最も重要なのは、1974 年に Paul Werbos がニューラルネットワークのバックプロパゲーションを発明したことであろう（本章で紹介する技術は、ニューラルネットワークのトレーニングに広く使われている）。彼の開発は数十年間ほとんど無視されていましたが、今日では現代 AI の最も重要な基礎とみなされています。

There is a lesson here for all of us! On your deep learning journey you will face many obstacles, both technical, and (even more difficult) posed by people around you who don't believe you'll be successful. There's one *guaranteed* way to fail, and that's to stop trying. We've seen that the only consistent trait amongst every fast.ai student that's gone on to be a world-class practitioner is that they are all very tenacious.

ここには、私たち全員にとっての教訓があります！ ディープラーニングの旅では、技術的なものから、成功すると信じていない周囲の人々によってもたらされる（さらに難しい）

ものまで、多くの障害に直面することになります。失敗する確実な方法は 1 つ、それは挑戦をやめることです。fast.ai の受講生で世界トップクラスの実践者となった人たちに共通する唯一の特徴は、全員が非常に粘り強いということです。

End sidebar

サイドバーを閉じる

For this initial tutorial we are just going to try to create a model that can classify any image as a 3 or a 7. So let's download a sample of MNIST that contains images of just these digits:

このチュートリアルでは、任意の画像を 3 か 7 に分類できるモデルを作成することを試みます：

```
path = untar_data(URLs.MNIST_SAMPLE)
```

In [ ]:

```
#hide
Path.BASE_PATH = path
```

We can see what's in this directory by using ls, a method added by fastai. This method returns an object of a special fastai class called L, which has all the same functionality of Python's built-in list, plus a lot more. One of its handy features is that, when printed, it displays the count of items, before listing the items themselves (if there are more than 10 items, it just shows the first few):

このディレクトリに何があるかは、fastai が追加した ls というメソッドで確認することができます。このメソッドは、L という fastai の特別なクラスのオブジェクトを返すもので、Python の組み込みリストと同じ機能に加えて、もっと多くの機能を備えています。便利な機能としては、印刷時に、項目そのものを列挙する前に、項目の数を表示することです（項目が 10 個以上ある場合は、最初の数個を表示するだけです）：

In [ ]:

```
path.ls()
```

Out[ ]:

```
(#9) [Path('cleaned.csv'),Path('item_list.txt'),Path('trained_model.pkl'),Path('models'),Path('valid'),Path('labels.csv'),Path('export.pkl'),Path('history.csv'),Path('train')]
```

The MNIST dataset follows a common layout for machine learning datasets: separate folders for the training set and the validation set (and/or test set). Let's see what's inside the training set:

MNIST データセットは、機械学習データセットによくあるレイアウトで、トレーニングセットと検証セット（またはテストセット）のフォルダに分かれています。トレーニングセットの中身を見てみましょう：

In [ ]:

```
(path/'train').ls()
```

Out[ ]:

(#2) [Path('train/7'),Path('train/3')]

There's a folder of 3s, and a folder of 7s. In machine learning parlance, we say that "3" and "7" are the *labels* (or targets) in this dataset. Let's take a look in one of these folders (using `sorted` to ensure we all get the same order of files):

3というフォルダと7というフォルダがある。機械学習の用語では、「3」と「7」がこのデータセットのラベル（またはターゲット）であると言います。これらのフォルダの1つを見てみましょう（ソートを使って、ファイルの順番が同じになるようにしています）：

In [ ]:

```
threes = (path/'train'/'3').ls().sorted()
sevens = (path/'train'/'7').ls().sorted()
threes
```

Out[ ]:

(#6131) [Path('train/3/10.png'),Path('train/3/10000.png'),Path('train/3/10011.png'),Path('train/3/10031.png'),Path('train/3/10034.png'),Path('train/3/10042.png'),Path('train/3/10052.png'),Path('train/3/1007.png'),Path('train/3/10074.png'),Path('train/3/10091.png')...]

As we might expect, it's full of image files. Let's take a look at one now. Here's an image of a handwritten number 3, taken from the famous MNIST dataset of handwritten numbers:

予想通り、画像ファイルでいっぱいです。では、1つ見てみましょう。手書き数字の有名なMNISTデータセットから取った、手書き数字の3の画像です：

In [ ]:

```
im3_path = threes[1]
im3 = Image.open(im3_path)
im3
```

Out[ ]:

Here we are using the `Image` class from the *Python Imaging Library* (PIL), which is the most widely used Python package for opening, manipulating, and viewing images. Jupyter knows about PIL images, so it displays the image for us automatically.

ここでは、Python Imaging Library (PIL)の Image クラスを使用しています。PIL は、画像を開き、操作し、表示するための最も広く使われている Python パッケージです。Jupyter は PIL の画像について知っているので、自動的に画像を表示してくれます。

In a computer, everything is represented as a number. To view the numbers that make up this image, we have to convert it to a *NumPy array* or a *PyTorch tensor*. For instance, here's what a section of the image looks like, converted to a NumPy array:

コンピュータの中では、あらゆるものが数字で表現されます。この画像を構成する数字を表示するには、NumPy の配列や PyTorch のテンソルに変換する必要があります。例えば、画像の一部分を NumPy の配列に変換したものがこちらです：

In [ ]:

```
array(im3)[4:10,4:10]
```

Out[ ]:

```
array([[  0,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0,  29],
       [  0,   0,   0,  48, 166, 224],
       [  0,  93, 244, 249, 253, 187],
       [  0, 107, 253, 253, 230,  48],
       [  0,   3,  20,  20,  15,   0]], dtype=uint8)
```

The `4:10` indicates we requested the rows from index 4 (included) to 10 (not included) and the same for the columns. NumPy indexes from top to bottom and left to right, so this section is located in the top-left corner of the image. Here's the same thing as a PyTorch tensor:

4:10 は、インデックス 4（含まれる）から 10（含まれない）までの行を要求したことを示し、列についても同様である。NumPy は上から下、左から右へとインデックスを付けるので、この部分は画像の左上に位置する。以下は PyTorch の tensor と同じものです：

In [ ]:

```
tensor(im3)[4:10,4:10]
```

Out[ ]:

```
tensor([[  0,   0,   0,   0,   0,   0],
        [  0,   0,   0,   0,   0,  29],
        [  0,   0,   0,  48, 166, 224],
        [  0,  93, 244, 249, 253, 187],
        [  0, 107, 253, 253, 230,  48],
        [  0,   3,  20,  20,  15,   0]], dtype=torch.uint8)
```

We can slice the array to pick just the part with the top of the digit in it, and then use a Pandas DataFrame to color-code the values using a gradient, which shows us clearly how the image is created from the pixel values:

配列をスライスして、数字の先頭が入った部分だけを選び、Pandas DataFrame を使ってグラデーションで色分けすることで、ピクセル値から画像が作られていることがよくわかります：

array(im3)[4:10,4:10]

array([0, 0, 0, 0, 0, 0]、
[0, 0, 0, 0, 0, 29],
[0, 0, 0, 48, 166, 224],
[0, 93, 244, 249, 253, 187],
[0, 107, 253, 253, 230, 48],
[0, 3, 20, 20, 15, 0]], dtype=uint8)

In [ ]:

```
#hide_output
im3_t = tensor(im3)
df = pd.DataFrame(im3_t[4:15,4:22])
df.style.set_properties(**{'font-size':'6pt'}).background_gradient('Greys')
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	29	150	195	254	255	254	176	193	150	96	0	0	0
2	0	0	0	48	166	224	253	253	234	196	253	253	253	253	233	0	0	0
3	0	93	244	249	253	187	46	10	8	4	10	194	253	253	233	0	0	0
4	0	107	253	253	230	48	0	0	0	0	0	192	253	253	156	0	0	0
5	0	3	20	20	15	0	0	0	0	0	43	224	253	245	74	0	0	0
6	0	0	0	0	0	0	0	0	0	0	249	253	245	126	0	0	0	0
7	0	0	0	0	0	0	0	14	101	223	253	248	124	0	0	0	0	0
8	0	0	0	0	0	11	166	239	253	253	253	187	30	0	0	0	0	0
9	0	0	0	0	0	16	248	250	253	253	253	253	232	213	111	2	0	0
10	0	0	0	0	0	0	0	43	98	98	208	253	253	253	253	187	22	0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	29	150	195	254	255	254	176	193	150	96	0	0	0
2	0	0	0	48	166	224	253	253	234	196	253	253	253	253	233	0	0	0
3	0	93	244	249	253	187	46	10	8	4	10	194	253	253	233	0	0	0
4	0	107	253	253	230	48	0	0	0	0	0	192	253	253	156	0	0	0
5	0	3	20	20	15	0	0	0	0	0	43	224	253	245	74	0	0	0
6	0	0	0	0	0	0	0	0	0	0	249	253	245	126	0	0	0	0
7	0	0	0	0	0	0	0	14	101	223	253	248	124	0	0	0	0	0
8	0	0	0	0	0	11	166	239	253	253	253	187	30	0	0	0	0	0
9	0	0	0	0	0	16	248	250	253	253	253	253	232	213	111	2	0	0
10	0	0	0	0	0	0	0	43	98	98	208	253	253	253	253	187	22	0

You can see that the background white pixels are stored as the number 0, black is the number 255, and shades of gray are between the two. The entire image contains 28 pixels across and 28 pixels down, for a total of 784 pixels. (This is much smaller than an image that you would get from a phone camera, which has millions of pixels, but is a convenient size for our initial learning and experiments. We will build up to bigger, full-color images soon.)

背景の白の画素は数字の 0、黒は数字の 255、濃淡はその中間として保存されていることがわかりますね。画像全体では、横 28 ピクセル、縦 28 ピクセル、合計 784 ピクセルが格納されています。(これは、携帯電話のカメラで撮影した数百万画素の画像よりはるかに小さいですが、最初の学習と実験には便利なサイズです。もっと大きなフルカラー画像に近づける予定です)。

So, now you've seen what an image looks like to a computer, let's recall our goal: create a model that can recognize 3s and 7s. How might you go about getting a computer to do that?

さて、画像がコンピュータにどのように見えるかを見てもらったところで、「3 と 7 を認識できるモデルを作る」という目標を思い出してみましょう。コンピュータにそれをさせるには、どうしたらいいのでしょうか?

> Warning: Stop and Think!: Before you read on, take a moment to think about how a computer might be able to recognize these two different digits. What kinds of features might it be able to look at? How might it be able to identify these features? How could it combine them together? Learning works best when you try to solve problems yourself, rather than just reading somebody else's answers; so step away from this book for a few

minutes, grab a piece of paper and pen, and jot some ideas down…

警告 立ち止まって考えてみてください この先を読む前に、コンピュータがどのようにしてこの 2 つの異なる数字を認識することができるのか、少し考えてみてください。どのような特徴に注目すればよいのでしょうか? どのようにこれらの特徴を識別することができるだろうか? どのように組み合わせればいいのだろう? この本から少し離れて、紙とペンを持って、アイデアを書き留めてみてください…。

First Try: Pixel Similarity

初挑戦： 画素の類似性

So, here is a first idea: how about we find the average pixel value for every pixel of the 3s, then do the same for the 7s. This will give us two group averages, defining what we might call the "ideal" 3 and 7. Then, to classify an image as one digit or the other, we see which of these two ideal digits the image is most similar to. This certainly seems like it should be better than nothing, so it will make a good baseline.

そこで、最初のアイデアとして、「3」の各ピクセルの平均値を求め、「7」についても同様に求めるのはどうでしょうか。こうすることで、2 つのグループの平均値が得られ、「理想的な」3 と 7 を定義することができます。そして、この 2 つの理想的な数字のどちらに近いかを判断することで、画像を 1 つの数字に分類します。これは確かに、何もしないよりはましだと思えるので、良い基準値になるでしょう。

jargon: Baseline: A simple model which you are confident should perform reasonably well. It should be very simple to implement, and very easy to test, so that you can then test each of your improved ideas, and make sure they are always better than your baseline. Without starting with a sensible baseline, it is very difficult to know whether your super-fancy models are actually any good. One good approach to creating a baseline is doing what we have done here: think of a simple, easy-to-implement model. Another good approach is to search around to find other people that have solved similar problems to yours, and download and run their code on your dataset. Ideally, try both of these!

専門用語： ベースライン： それなりにうまくいくと確信できるシンプルなモデル。実装が非常に簡単で、テストも簡単であるべきで、改良したアイデアをそれぞれテストし、それらが常にベースラインより優れていることを確認することができます。適切なベースラインから始めなければ、超高級なモデルが実際に優れているかどうかを知ることは非常に困難です。ベースラインを作るための良いアプローチの 1 つは、今回行ったように、シンプルで実

装が簡単なモデルを考えることです。もう一つの良い方法は、あなたと同じような問題を解決した人を探して、その人のコードをダウンロードして、あなたのデータセットで実行することです。理想的には、この 2 つを試してみることです！

Step one for our simple model is to get the average of pixel values for each of our two groups. In the process of doing this, we will learn a lot of neat Python numeric programming tricks!

この単純なモデルのステップ 1 は、2 つのグループそれぞれのピクセル値の平均を得ることです。この過程で、Python の数値プログラミングの巧妙なトリックをたくさん学ぶことになります！

Let's create a tensor containing all of our 3s stacked together. We already know how to create a tensor containing a single image. To create a tensor containing all the images in a directory, we will first use a Python list comprehension to create a plain list of the single image tensors.

では、すべての 3 が積み重なったテンソルを作ってみましょう。1 枚の画像を含むテンソルを作成する方法はすでに知っています。ディレクトリ内のすべての画像を含むテンソルを作成するために、まず Python のリスト内包を使用して、単一画像のテンソルのプレーンリストを作成します。

We will use Jupyter to do some little checks of our work along the way—in this case, making sure that the number of returned items seems reasonable:
この場合、返されるアイテムの数が妥当であることを確認するために、Jupyter を使用して、途中で作業の小さなチェックを行います：

```
seven_tensors = [tensor(Image.open(o)) for o in sevens]
three_tensors = [tensor(Image.open(o)) for o in threes]
len(three_tensors),len(seven_tensors)
```

Out[ ]:

```
(6131, 6265)
```

note: List Comprehensions: List and dictionary comprehensions are a wonderful feature of Python. Many Python programmers use them every day, including the authors of this book—they are part of "idiomatic Python." But programmers coming from other languages may have never seen them before. There are a lot of great tutorials just a web search away, so we won't spend a long time discussing them now. Here is a quick explanation and example to get you started. A list comprehension looks like this: `new_list = [f(o) for o in a_list if o>0]`. This will return every element of `a_list` that is greater than 0, after passing it to the function `f`. There are three parts here: the collection you

are iterating over (a_list), an optional filter (if o>0), and something to do to each element (f(o)). It's not only shorter to write but way faster than the alternative ways of creating the same list with a loop.

note: リスト内包：リストと辞書の内包は、Python の素晴らしい機能です。本書の著者も含め、多くの Python プログラマが毎日使っています。"idiomatic Python "の一部です。しかし、他の言語から来たプログラマは、この機能を見たことがないかもしれません。しかし、他の言語から来たプログラマーにとっては、初めて目にするものでしょう。ここでは、簡単な説明と例を示します。リスト内包は次のようになります： new_list = [f(o) for o in a_list if o>0]. これは、a_list の要素のうち、0 より大きいものを関数 f に渡して返します。この方法は、ループを使って同じリストを作成する方法よりも、記述時間が短いだけでなく、はるかに高速です。

We'll also check that one of the images looks okay. Since we now have tensors (which Jupyter by default will print as values), rather than PIL images (which Jupyter by default will display as images), we need to use fastai's `show_image` function to display it:

また、画像の 1 つが問題なく見えるかどうかもチェックすることにします。PIL 画像（Jupyter のデフォルトでは画像として表示される）ではなく、テンソル（Jupyter のデフォルトでは値として表示される）を持っているので、それを表示するために fastai の show_image 関数を使う必要があります：

In [ ]:

```
show_image(three_tensors[1]);
```

For every pixel position, we want to compute the average over all the images of the intensity of that pixel. To do this we first combine all the images in this list into a single three-dimensional tensor. The most common way to describe such a tensor is to call it a *rank-3 tensor*. We often need to stack up individual tensors in a collection into a single tensor. Unsurprisingly, PyTorch comes with a function called `stack` that we can use for this purpose.

各ピクセルの位置について、そのピクセルの強度の全画像の平均を計算したい。これを行うには、まずこのリストにあるすべての画像を 1 つの 3 次元テンソルに結合する。このようなテンソルを表現する最も一般的な方法は、ランク 3 テンソルと呼ばれるものである。私たちはしばしば、コレクション内の個々のテンソルを積み上げて 1 つのテンソルにする必要があります。当然のことながら、PyTorch には stack という関数が用意されており、この目的のために使うことができます。

Some operations in PyTorch, such as taking a mean, require us to *cast* our integer types to float types. Since we'll be needing this later, we'll also cast our stacked tensor to `float` now. Casting in PyTorch is as simple as typing the name of the type you wish to cast to, and treating it as a method.

PyTorch のいくつかの操作、例えば平均を取るには、整数型を float 型にキャストする必要があります。これは後で必要になるので、スタックされたテンソルも float にキャストすることにします。PyTorch でのキャストは、キャストしたい型の名前を入力し、それをメソッドとして扱うだけでよいのです。

Generally when images are floats, the pixel values are expected to be between 0 and 1, so we will also divide by 255 here:

一般に画像が float の場合、ピクセル値は 0 から 1 の間であることが予想されるので、ここでは 255 で割ることもします：

In [ ]:

```
stacked_sevens = torch.
stack(seven_tensors).float()/255
stacked_threes = torch.stack(three_tensors).float()/255
stacked_threes.shape
```

Out[ ]:

torch.Size([6131, 28, 28])

Perhaps the most important attribute of a tensor is its *shape*. This tells you the length of each axis. In this case, we can see that we have 6,131 images, each of size 28×28 pixels. There is nothing specifically about this tensor that says that the first axis is the number of images, the second is the height, and the third is the width—the semantics of a tensor are entirely up to us, and how we construct it. As far as PyTorch is concerned, it is just a bunch of numbers in memory.

テンソルの最も重要な属性は、おそらくその形状です。これは、各軸の長さを教えてくれます。この場合、6,131 枚の画像があり、それぞれが 28×28 ピクセルの大きさであることがわかる。このテンソルには、第 1 軸が画像の数、第 2 軸が高さ、第 3 軸が幅というような特別なものはありません。テンソルのセマンティクスは、完全に私たち次第で、どのように構築するかが決まります。PyTorch が知る限り、テンソルはメモリ上の数字の束に過ぎません。

The *length* of a tensor's shape is its rank:

テンソルのシェイプの長さは、そのランクです：

In [ ]:

```
len(stacked_threes.shape)
```

Out[ ]:

3

It is really important for you to commit to memory and practice these bits of tensor jargon: *rank* is the number of axes or dimensions in a tensor; *shape* is the size of each axis of a tensor.

ランクとはテンソルの軸や次元の数、形状とはテンソルの各軸の大きさです。

A: Watch out because the term "dimension" is sometimes used in two ways. Consider that we live in "three-dimensonal space" where a physical position can be described by a 3-vector `v`. But according to PyTorch, the attribute `v.ndim` (which sure looks like the "number of dimensions" of `v`) equals one, not three! Why? Because `v` is a vector, which is a tensor of rank one, meaning that it has only one *axis* (even if that axis has a length of three). In other words, sometimes dimension is used for the size of an axis ("space is three-dimensional"); other times, it is used for the rank, or the number of axes ("a matrix has two dimensions"). When confused, I find it helpful to translate all statements into terms of rank, axis, and length, which are unambiguous terms.

A: 「次元」という用語は、2 つの意味で使われることがあるので注意してください。しかし、PyTorch によると、v.ndim という属性（v の「次元数」のように見える）は、3 ではなく 1 である！ なぜか？ なぜなら、v はベクトルであり、ランク 1 のテンソルであるため、軸が 1 つしかない（その軸の長さが 3 であっても）。つまり、次元は軸の大きさを表すこともあれば（「空間は 3 次元」）、ランク、つまり軸の数を表すこともある（「行列は 2 次元」）。混乱したときは、すべての記述をランク、軸、長さという曖昧さのない用語に変換することが有効だと思います。

We can also get a tensor's rank directly with ndim:

テンソルのランクは、ndim で直接求めることもできます：

In [ ]:

```
stacked_threes.ndim
```

Out[ ]:

```
3
```

Finally, we can compute what the ideal 3 looks like. We calculate the mean of all the image tensors by taking the mean along dimension 0 of our stacked, rank-3 tensor. This is the dimension that indexes over all the images.

最後に、理想的な 3 がどのようなものかを計算することができます。すべての画像テンソルの平均を計算するには、積層されたランク 3 テンソルの 0 次元に沿った平均を取ります。この次元は、すべての画像にインデックスを付ける次元です。

In other words, for every pixel position, this will compute the average of that pixel over all images. The result will be one value for every pixel position, or a single image. Here it is:

つまり、すべてのピクセル位置について、そのピクセルの平均をすべての画像について計算することになる。その結果、すべてのピクセル位置、つまり 1 枚の画像に対して 1 つの値が得られることになります。以下はその例です：

In [ ]:

```
mean3 = stacked_threes.mean(0)
show_image(mean3);
```

According to this dataset, this is the ideal number 3! (You may not like it, but this is what peak number 3 performance looks like.) You can see how it's very dark where all the images agree it should be dark, but it becomes wispy and blurry where the images disagree.

このデータセットによると、これが理想的な 3 番であることがわかります！(このデータセットによると、これが理想的な 3 番です！（気に入らないかもしれませんが、これがピーク時の 3 番の性能です）すべての画像が暗くあるべきと同意しているところは非常に暗く、しかし、画像が同意していないところはうっすらとぼやけているのがわかるでしょう。

Let's do the same thing for the 7s, but put all the steps together at once to save some time:

同じことを 7 番でもやってみましょう。しかし、時間を節約するために、すべてのステップを一度にまとめてみましょう：

```
mean7 = stacked_sevens.mean(0)
show_image(mean7);
```

Let's now pick an arbitrary 3 and measure its *distance* from our "ideal digits."

では、任意の 3 を選び、"理想の数字 "からの距離を測ってみましょう。

> stop: Stop and Think!: How would you calculate how similar a particular image is to each of our ideal digits? Remember to step away from this book and jot down some ideas before you move on! Research shows that recall and understanding improves dramatically when you are engaged with the learning process by solving problems, experimenting, and trying new ideas yourself
>
> stop: Stop and Think! ある画像が理想の数字にどれだけ似ているか、どのように計算するのでしょうか？ 次に進む前に、この本から離れ、いくつかのアイデアを書き留めることを忘れないでください！問題を解いたり、実験したり、新しいアイデアを試したりして、学習プロセスに参加すると、記憶と理解が劇的に向上することが研究によって明らかになっています。

Here's a sample 3:

以下、サンプル 3 です：

In [ ]:

```
a_3 = stacked_threes[1]
show_image(a_3);
```

How can we determine its distance from our ideal 3? We can't just add up the differences between the pixels of this image and the ideal digit. Some differences will be positive while others will be negative, and these differences will cancel out, resulting in a situation where an image that is too dark in some places and too light in others might be shown as having zero total differences from the ideal. That would be misleading!

理想的な 3 との距離をどのように求めるか？ この画像のピクセルと理想的な数字の差を足し算すればいいというわけではありません。ある画素はプラス、ある画素はマイナスとなり、その差は相殺されてしまいます。これでは、誤解を招くことになります！

To avoid this, there are two main ways data scientists measure distance in this context:

これを避けるために、データサイエンティストは主に 2 つの方法で距離を測定します：

- Take the mean of the *absolute value* of differences (absolute value is the function that replaces negative values with positive values). This is called the *mean absolute difference* or *L1 norm*
- Take the mean of the *square* of differences (which makes everything positive) and then take the *square root* (which undoes the squaring). This is called the *root mean squared error* (RMSE) or *L2 norm*.

important: It's Okay to Have Forgotten Your Math: In this book we generally assume that you have completed high school math, and remember at least some of it... But everybody forgets some things! It all depends on what you happen to have had reason to practice in the meantime. Perhaps you have forgotten what a *square root* is, or exactly how they work. No problem! Any time you come across a maths concept that is not explained fully in this book, don't just keep moving on; instead, stop and look it up. Make sure you understand the basic idea, how it works, and why we might be using it. One of the best places to refresh your understanding is Khan Academy. For instance, Khan Academy has a great introduction to square roots.

差の絶対値の平均を取る（絶対値とは、負の値を正の値に置き換える関数のこと）。これは平均絶対差または L1 ノルムと呼ばれる。
差の二乗の平均をとり（これはすべてを正にする）、平方根をとる（これは二乗を元に戻す）。これを平均二乗誤差（RMSE）または L2 ノルムと呼びます。
重要：数学を忘れても大丈夫です： 本書では、あなたが高校の数学を修了し、少なくともその一部を覚えていることを前提としています...。しかし、

> 誰でも忘れてしまうことはあるものです！しかし、誰でも忘れてしまうことはあるものです。平方根が何であるか、どのように機能するかを忘れてしまったかもしれません。でも、大丈夫！本書で説明しきれない数学の概念に出会ったら、そのまま先に進まず、立ち止まって調べてみてください。基本的な考え方、仕組み、そしてなぜそれを使うのか、しっかり理解しましょう。理解を深めるのに最適な場所の１つが、カーン・アカデミーです。例えば、Khan Academy には平方根の素晴らしい紹介があります。

Let's try both of these now:

では、この２つを試してみましょう：

In [ ]:

```
dist_3_abs = (a_3 - mean3).abs().mean()
dist_3_sqr = ((a_3 - mean3)**2).mean().sqrt()
dist_3_abs,dist_3_sqr
```

Out[ ]:

(tensor(0.1114), tensor(0.2021))

In [ ]:

```
dist_7_abs = (a_3 - mean7).abs().mean()
dist_7_sqr = ((a_3 - mean7)**2).mean().sqrt()
dist_7_abs,dist_7_sqr
```

Out[ ]:

(tensor(0.1586), tensor(0.3021))

In both cases, the distance between our 3 and the "ideal" 3 is less than the distance to the ideal 7. So our simple model will give the right prediction in this case.

どちらの場合も、私たちの３と「理想的な」３との距離は、理想的な７との距離よりも小さくなります。つまり、この場合、単純なモデルが正しい予測をすることになります。

PyTorch already provides both of these as *loss functions*. You'll find these inside `torch.nn.functional`, which the PyTorch team recommends importing as `F` (and is available by default under that name in fastai):

PyTorch はこの２つを損失関数としてすでに提供しています。これらは `torch.nn.functional` の中にあり、PyTorch チームは `F` としてインポートすることを推奨しています（デフォルトで fastai のこの名前で利用できます）：

In [ ]:

```
F.l1_loss(a_3.float(),mean7), F.mse_loss(a_3,mean7).sqrt()
```

Out[ ]:

(tensor(0.1586), tensor(0.3021))

Here `mse` stands for *mean squared error*, and `l1` refers to the standard mathematical jargon for *mean absolute value* (in math it's called the *L1 norm*).

ここで mse は平均二乗誤差、l1 は平均絶対値を意味する標準的な数学の専門用語（数学では L1 ノルムと呼ばれる）。

S: Intuitively, the difference between L1 norm and mean squared error (MSE) is that the latter will penalize bigger mistakes more heavily than the former (and be more lenient with small mistakes).

S: 直感的には、L1 ノルムと平均二乗誤差（MSE）の違いは、後者は前者よりも大きなミスにペナルティを与える（小さなミスに甘くなる）ということだ。

J: When I first came across this "L1" thingie, I looked it up to see what on earth it meant. I found on Google that it is a *vector norm* using *absolute value*, so looked up *vector norm* and started reading: *Given a vector space V over a field F of the real or complex numbers, a norm on V is a nonnegative-valued any function p: V → [0,+∞) with the following properties: For all a ∈ F and all u, v ∈ V, p(u + v) ≤ p(u) + p(v)...* Then I stopped reading. "Ugh, I'll never understand math!" I thought, for the thousandth time. Since then I've learned that every time these complex mathy bits of jargon come up in practice, it turns out I can replace them with a tiny bit of code! Like, the *L1 loss* is just equal to (a-b).abs().mean(), where a and b are tensors. I guess mathy folks just think differently than me... I'll make sure in this book that every time some mathy jargon comes up, I'll give you the little bit of code it's equal to as well, and explain in common-sense terms what's going on.

J：この「L1」という言葉を初めて目にしたとき、一体どういう意味なのか調べてみたんです。Google で調べると、絶対値を使ったベクトルノルムであることがわかったので、ベクトルノルムを調べて読み始めた： 実数または複素数の場 F 上のベクトル空間 V が与えられたとき、V 上のノルムは、非負値の任意の関数 p である：V → [0,+∞) で、次のような性質を持つ：すべての a∈F とすべての u,v∈V に対して、p(u + v) ≦ p(u) + p(v)...。そして、読むのをやめてしまった。"うっ、数学なんて理解できない！" と、何千回目かの時に思いました。それ以来、私は、このような複雑な数学的専門用語が実際に出てくるたびに、ほんの少しのコードで置き換えられることがわかったのです！ 例えば、L1 の損失は(a-b).abs().mean()と同じで、a と b はテンソルです。マシーな人たちは、私とは考え方が違うんでしょうね...。この本では、数学的な専門用語が出てくるたびに、それが等しいコードも少し出

して、何が起こっているのかを常識的な言葉で説明するようにしようと思います。

We just completed various mathematical operations on PyTorch tensors. If you've done some numeric programming in NumPy before, you may recognize these as being similar to NumPy arrays. Let's have a look at those two very important data structures.

PyTorch のテンソルに対する様々な数学的操作を完了したところです。NumPy で数値プログラミングをしたことがある人なら、これらは NumPy の配列と似ていると認識するかもしれません。この 2 つの非常に重要なデータ構造について見てみましょう。

NumPy Arrays and PyTorch Tensors

NumPy の配列と PyTorch のテンソル

NumPy is the most widely used library for scientific and numeric programming in Python. It provides very similar functionality and a very similar API to that provided by PyTorch; however, it does not support using the GPU or calculating gradients, which are both critical for deep learning. Therefore, in this book we will generally use PyTorch tensors instead of NumPy arrays, where possible.

NumPy は、Python で科学的・数値的プログラミングを行うための最も広く使われているライブラリです。PyTorch が提供するものと非常によく似た機能と API を提供していますが、深層学習に不可欠な GPU の利用や勾配の計算をサポートしていません。そのため、本書では一般的に、可能な限り NumPy の配列の代わりに PyTorch のテンソールを使用することにします。

(Note that fastai adds some features to NumPy and PyTorch to make them a bit more similar to each other. If any code in this book doesn't work on your computer, it's possible that you forgot to include a line like this at the start of your notebook: from fastai.vision.all import *.)

(fastai は NumPy と PyTorch にいくつかの機能を追加して、互いにもう少し似たものにすることに注意してください。この本のコードがあなたのコンピュータで動かない場合、ノートブックの最初に次のような行を入れるのを忘れた可能性があります：from fastai.vision.all import *.).

But what are arrays and tensors, and why should you care?

しかし、配列やテンソルとは何なのか、そしてなぜ気にする必要があるのか？

Python is slow compared to many languages. Anything fast in Python, NumPy, or PyTorch is likely to be a wrapper for a compiled object written (and optimized) in another language—specifically C. In fact, **NumPy arrays and PyTorch tensors can finish computations many thousands of times faster than using pure Python.**

Python は多くの言語と比較して遅いです。Python、NumPy、PyTorch で高速なものは、他の言語、特に C で書かれた（そして最適化された）コンパイルされたオブジェクトのラ

ッパーである可能性が高いです。実際、NumPy 配列と PyTorch テンソルは、純粋な Python を使って計算するより何千回も速く計算を終了できます。

A NumPy array is a multidimensional table of data, with all items of the same type. Since that can be any type at all, they can even be arrays of arrays, with the innermost arrays potentially being different sizes—this is called a "jagged array." By "multidimensional table" we mean, for instance, a list (dimension of one), a table or matrix (dimension of two), a "table of tables" or "cube" (dimension of three), and so forth. If the items are all of some simple type such as integer or float, then NumPy will store them as a compact C data structure in memory. This is where NumPy shines. NumPy has a wide variety of operators and methods that can run computations on these compact structures at the same speed as optimized C, because they are written in optimized C.

NumPy の配列は、データの多次元テーブルで、すべての項目が同じ型である。これは "ギザギザ配列 "と呼ばれるものである。多次元テーブル」とは、例えばリスト（1 次元）、表や行列（2 次元）、「テーブルのテーブル」または「キューブ」（3 次元）などを意味します。もし項目がすべて整数や浮動小数点などの単純な型であれば、NumPy はそれらをコンパクトな C データ構造としてメモリに保存します。ここが NumPy の腕の見せ所です。NumPy には様々な演算子やメソッドがあり、最適化された C 言語で書かれているため、これらのコンパクトな構造に対して、最適化された C 言語と同じ速度で計算を実行することができる。

A PyTorch tensor is nearly the same thing as a NumPy array, but with an additional restriction that unlocks some additional capabilities. It's the same in that it, too, is a multidimensional table of data, with all items of the same type. However, the restriction is that a tensor cannot use just any old type—it has to use a single basic numeric type for all components. For example, a PyTorch tensor cannot be jagged. It is always a regularly shaped multidimensional rectangular structure.

PyTorch のテンソルは、NumPy の配列とほぼ同じものですが、いくつかの追加機能を解除する制約があります。これもまた、すべての項目が同じ型であるデータの多次元テーブルである点では同じです。ただし、テンソルはどんな型でも使えるわけではなく、すべての構成要素に単一の基本的な数値型を使用しなければならないという制約があります。例えば、PyTorch のテンソルはギザギザにすることができません。常に規則正しい多次元直方体構造です。

The vast majority of methods and operators supported by NumPy on these structures are also supported by PyTorch, but PyTorch tensors have additional capabilities. One major capability is that these structures can live on the GPU, in which case their computation will be optimized for the GPU and can run much faster (given lots of values to work on). In addition, PyTorch can automatically calculate derivatives of these operations, including combinations of operations. As you'll see, it would be impossible to do deep learning in practice without this capability.

NumPy がこれらの構造体に対してサポートしているメソッドや演算子の大部分は PyTorch でもサポートされていますが、PyTorch のテンソルには追加の機能があります。その場合、計算が GPU 用に最適化され、はるかに速く実行できます（作業する値がたくさんある場合）。さらに、PyTorch はこれらの演算の導関数（演算の組み合わせも含む）を

自動的に計算することができます。おわかりのように、この機能がなければ、実際にディープラーニングを行うことは不可能でしょう。

> S: If you don't know what C is, don't worry as you won't need it at all. In a nutshell, it's a low-level (low-level means more similar to the language that computers use internally) language that is very fast compared to Python. To take advantage of its speed while programming in Python, try to avoid as much as possible writing loops, and replace them by commands that work directly on arrays or tensors.
>
> S：C 言語が何かわからないという人も、まったく必要ないので心配しないでください。一言で言うと、Python に比べて非常に高速な低レベル（低レベルとは、コンピュータが内部で使っている言語に近いという意味）言語です。Python のプログラミングでその速さを生かすには、ループをできるだけ書かないようにし、配列やテンソルを直接操作するコマンドに置き換えてください。

Perhaps the most important new coding skill for a Python programmer to learn is how to effectively use the array/tensor APIs. We will be showing lots more tricks later in this book, but here's a summary of the key things you need to know for now.

Python プログラマにとって最も重要なコーディングスキルは、配列/テンソル API を効果的に使用する方法でしょう。本書の後半でもっとたくさんのトリックを紹介しますが、ここでは今のところ知っておくべき重要なことをまとめておきます。

To create an array or tensor, pass a list (or list of lists, or list of lists of lists, etc.) to array() or tensor():
配列やテンソルを作るには、array()や tensor()にリスト（またはリストのリスト、リストのリストのリストなど）を渡します：

data = [[1,2,3],[4,5,6]] arr = array (data) tns = tensor(data)

In [ ]:

```
arr # numpy
```

Out[ ]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [ ]:

```
tns # pytorch
```

Out[ ]:

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

All the operations that follow are shown on tensors, but the syntax and results for NumPy arrays is identical.

この後の操作はすべてテンソルに対して示しているが、NumPy の配列に対する構文と結果は同じである。

You can select a row (note that, like lists in Python, tensors are 0-indexed so 1 refers to the second row/column):

行を選択することができます（Python のリストと同様に、テンソルは 0 インデックスなので、1 は 2 行目/列を指すことに注意してください）：

In [ ]:

```
tns[1]
```

Out[ ]:

```
tensor([4, 5, 6])
```

or a column, by using : to indicate *all of the first axis* (we sometimes refer to the dimensions of tensors/arrays as *axes*):

または列を、:を使って最初の軸のすべてを示す（テンソル/配列の次元を軸と呼ぶことがある）：

In [ ]:

```
tns[:,1]
```

Out[ ]:

```
tensor([2, 5])
```

You can combine these with Python slice syntax ([start:end] with end being excluded) to select part of a row or column:

これらを Python のスライス構文（[start:end]で end は除外）と組み合わせることで、行や列の一部を選択することができます：

In [ ]:

```
tns[1,1:3]
```

Out[ ]:

```
tensor([5, 6])
```

And you can use the standard operators such as +, -, *, /:

また、+、-、*、/などの標準的な演算子を使用することができます：

In [ ]:

```
tns+1
```

Out[ ]:

```
tensor([[2, 3, 4],
        [5, 6, 7]])
```

Tensors have a type:

テンソルには型があります：

In [ ]:

```
tns.type()
```

Out[ ]:

'torch.LongTensor'

And will automatically change type as needed, for example from int to float:

また、int から float のように、必要に応じて自動的に型を変更します：

In [ ]:

```
tns*1.5
```

Out[ ]:

```
tensor([[1.5000,  3.0000,  4.5000],
        [6.0000,  7.5000,  9.0000]])
```

So, is our baseline model any good? To quantify this, we must define a metric.

では、ベースラインモデルは良いものなのでしょうか？ これを定量化するためには、指標を定義する必要があります。

Computing Metrics Using Broadcasting

ブロードキャストを利用したメトリクスの計算

Recall that a metric is a number that is calculated based on the predictions of our model, and the correct labels in our dataset, in order to tell us how good our model is. For instance, we could use either of the functions we saw in the previous section, mean squared error, or mean absolute error, and take the average of them over the whole dataset. However, neither of these are numbers that are very understandable to most people; in practice, we normally use *accuracy* as the metric for classification models.

メトリックとは、モデルの予測値とデータセットの正しいラベルに基づいて計算される数値であり、モデルがどの程度優れているかを示すものであることを思い出してください。例えば、前のセクションで見た関数、平均二乗誤差や平均絶対誤差のいずれかを使って、データセット全体の平均を取ることができます。しかし、どちらも多くの人に理解しやすい数値ではありません。実際には、分類モデルの指標として精度を使うのが普通です。

As we've discussed, we want to calculate our metric over a *validation set*. This is so that we don't inadvertently overfit—that is, train a model to work well only on our training data. This is not really a risk with the pixel similarity model we're using here as a first try, since it has no trained components, but we'll use a validation set anyway to follow normal practices and to be ready for our second try later.

これまで述べてきたように、私たちは検証セットに対してメトリックを計算したいと思います。これは、不用意にオーバーフィットを起こさないようにするためです。つまり、訓練データだけでうまく機能するようにモデルを訓練するのです。今回使用するピクセル類似度モデルは、学習済みのコンポーネントがないため、このようなリスクはあまりありま

せんが、通常の慣行に従い、後で 2 回目の試行を行うための準備として、検証セットを使用することにします。

To get a validation set we need to remove some of the data from training entirely, so it is not seen by the model at all. As it turns out, the creators of the MNIST dataset have already done this for us. Do you remember how there was a whole separate directory called *valid*? That's what this directory is for!

検証セットを得るには、いくつかのデータをトレーニングから完全に削除する必要があります。結局のところ、MNIST データセットの作成者がすでにこれをやってくれているのです。valid という別のディレクトリがあったのを覚えていますか？ このディレクトリはそのためのものなのです！

So to start with, let's create tensors for our 3s and 7s from that directory. These are the tensors we will use to calculate a metric measuring the quality of our first-try model, which measures distance from an ideal image:

そこでまず、そのディレクトリから 3S と 7S のテンソルを作ってみましょう。このテンソルを使って、ファーストトライモデルの品質を測る指標、つまり理想的な画像からの距離を測る指標を計算します：

In [ ]:

```
valid_3_tens = torch.stack([tensor(Image.open(o)) for o in (path/'valid'/'3').ls()])
valid_3_tens = valid_3_tens.float()/255
valid_7_tens = torch.stack([tensor(Image.open(o))
    for o in (path/'valid'/'7').ls()]) valid_7_tens = valid_7_tens.float()/255
valid_3_tens.shape,valid_7_tens.shape
```

有効数字 3 桁の形状、有効数字 7 桁の形状

Out[ ]:

```
(torch.Size([1010, 28, 28]), torch.Size([1028, 28, 28]))
```

It's good to get in the habit of checking shapes as you go. Here we see two tensors, one representing the 3s validation set of 1,010 images of size 28×28, and one representing the 7s validation set of 1,028 images of size 28×28.

形状を確認しながら進める習慣をつけるとよいでしょう。ここでは、28×28 サイズの 1,010 枚の画像からなる 3s 検証セットと、28×28 サイズの 1,028 枚の画像からなる 7s 検証セットを表す 2 つのテンソルを示している。

We ultimately want to write a function, is_3, that will decide if an arbitrary image is a 3 or a 7. It will do this by deciding which of our two "ideal digits" this arbitrary image is closer to. For that we need to define a notion of distance—that is, a function that calculates the distance between two images.

最終的には、任意の画像が 3 か 7 かを判定する関数 is_3 を書きたい。この関数は、任意の画像が 2 つの「理想の数字」のどちらに近いかを判定する。そのためには、距離の概念を定義する必要があります。つまり、2 つの画像間の距離を計算する関数を定義します。

We can write a simple function that calculates the mean absolute error using an expression very similar to the one we wrote in the last section:

平均絶対誤差を計算する簡単な関数は、前節で書いた式とよく似たものを使って書くことができます：

```
def mnist_distance(a,b): return (a-b).abs().mean((-1,-2))
mnist_distance(a_3, mean3)
```

Out[ ]:

```
tensor(0.1114)
```

This is the same value we previously calculated for the distance between these two images, the ideal 3 mean3 and the arbitrary sample 3 a_3, which are both single-image tensors with a shape of [28,28].

これは、以前にこの 2 つの画像間の距離について計算した値と同じで、理想 3 mean3 と任意サンプル 3 a_3 は、どちらも形状が[28,28]の単一画像テンソルであることがわかります。

But in order to calculate a metric for overall accuracy, we will need to calculate the distance to the ideal 3 for *every* image in the validation set. How do we do that calculation? We could write a loop over all of the single-image tensors that are stacked within our validation set tensor, valid_3_tens, which has a shape of [1010,28,28] representing 1,010 images. But there is a better way.

しかし、全体的な精度を表す指標を算出するためには、検証セットのすべての画像について、理想 3 との距離を計算する必要があります。その計算はどうすればいいのでしょうか。検証セットのテンソル valid_3_tens は、1,010 枚の画像を表す[1010,28,28]の形状をしており、その中に積み上げられたすべての単一画像テンソルに対してループを書くことができます。しかし、もっと良い方法があります。

Something very interesting happens when we take this exact same distance function, designed for comparing two single images, but pass in as an argument valid_3_tens, the tensor that represents the 3s validation set:

2 枚の画像を比較するために作られたこの距離関数とまったく同じものを、3s 検証セットを表すテンソル valid_3_tens を引数として渡すと、非常に興味深いことが起こります：
で、[ ]内に

In [ ]:

```
valid_3_dist = mnist_distance(valid_3_tens, mean3)
valid_3_dist, valid_3_dist.shape
```

Out[ ]:

(tensor([0.1050, 0.1526, 0.1186, ..., 0.1122, 0.1170, 0.1086]),
 torch.Size([1010]))

Instead of complaining about shapes not matching, it returned the distance for every single image as a vector (i.e., a rank-1 tensor) of length 1,010 (the number of 3s in our validation set). How did that happen?

形状が一致しないと文句を言うのではなく、すべての画像の距離を、長さ 1,010（検証セットの 3 の数）のベクトル（ランク 1 テンソル）として返したのです。なぜそうなったのか?

Take another look at our function mnist_distance, and you'll see we have there the subtraction (a-b). The magic trick is that PyTorch, when it tries to perform a simple subtraction operation between two tensors of different ranks, will use *broadcasting*. That is, it will automatically expand the tensor with the smaller rank to have the same size as the one with the larger rank. Broadcasting is an important capability that makes tensor code much easier to write.

mnist_distance 関数をもう一度見てみると、(a-b)の引き算が行われていることがわかります。このマジックは、PyTorch が異なるランクの 2 つのテンソルの間で単純な引き算を行おうとするとき、ブロードキャストを使うというものです。つまり、ランクの小さい方のテンソルを、ランクの大きい方のテンソルと同じサイズに自動的に拡張するのです。ブロードキャストは、テンソルコードをより書きやすくする重要な機能である。

After broadcasting so the two argument tensors have the same rank, PyTorch applies its usual logic for two tensors of the same rank: it performs the operation on each corresponding element of the two tensors, and returns the tensor result. For instance:

ブロードキャストにより 2 つの引数テンソルが同じランクになった後、PyTorch は同じランクの 2 つのテンソルに対して通常のロジックを適用します：2 つのテンソルの対応する各要素に対して処理を実行し、テンソルの結果を返します。例えば
で、[ ]です：

In [ ]:

```
tensor([1,2,3]) + tensor(1)
```

Out[ ]:

tensor([2, 3, 4])

So in this case, PyTorch treats mean3, a rank-2 tensor representing a single image, as if it were 1,010 copies of the same image, and then subtracts each of those copies from each 3 in our validation set. What shape would you expect this tensor to have? Try to figure it out yourself before you look at the answer below:

つまり、この場合、PyTorch は、1 つの画像を表すランク 2 のテンソル mean3 を、あたかも同じ画像の 1,010 枚のコピーであるかのように扱い、検証セットの各 3 からそのコピーをそれぞれ差し引きます。このテンソルはどのような形をしているのでしょうか? 下の答えを見る前に、自分で考えてみてください：

In [ ]:

```
(valid_3_tens-mean3).shape
```

Out[ ]:

```
torch.Size([1010, 28, 28])
```

We are calculating the difference between our "ideal 3" and each of the 1,010 3s in the validation set, for each of 28×28 images, resulting in the shape [1010,28,28].

28×28 の画像それぞれについて、検証セットの 1,010 個の 3 との差を計算した結果、[1010,28,28]という形状になりました。

There are a couple of important points about how broadcasting is implemented, which make it valuable not just for expressivity but also for performance:

放送の実装方法については、表現力だけでなく性能面でも価値がある重要なポイントがいくつかあります：

- PyTorch doesn't *actually* copy mean3 1,010 times. It *pretends* it were a tensor of that shape, but doesn't actually allocate any additional memory

  PyTorch は実際には mean3 を 1,010 回コピーしない。そのような形のテンソルであるかのように見せかけるが、実際には追加のメモリは割り当てない

- It does the whole calculation in C (or, if you're using a GPU, in CUDA, the equivalent of C on the GPU), tens of thousands of times faster than pure Python (up to millions of times faster on a GPU!).

  計算はすべて C 言語（GPU をお使いの場合は、GPU 上の C 言語に相当する CUDA）で行い、純粋な Python よりも数万倍速く（GPU では最大数百万倍速く！）行います。

This is true of all broadcasting and elementwise operations and functions done in PyTorch. *It's the most important technique for you to know to create efficient PyTorch code.*

これは、PyTorch で行われるすべてのブロードキャストやエレメントワイズ演算、関数に当てはまります。効率的な PyTorch のコードを作成するために知っておくべき最も重要なテクニックです。

Next in mnist_distance we see abs. You might be able to guess now what this does when applied to a tensor. It applies the method to each individual element in the tensor, and returns a tensor of the results (that is, it applies the method "elementwise"). So in this case, we'll get back 1,010 matrices of absolute values.

次に mnist_distance で abs を見ます。テンソルに適用した場合、これが何をするかはもう想像がつくかもしれません。テンソルの各要素にメソッドを適用し、その結果のテンソルを返します（つまり、「要素ごとに」メソッドを適用するのです）。つまり、この場合、絶対値の行列が 1,010 個返されます。

Finally, our function calls `mean((-1,-2))`. The tuple `(-1,-2)` represents a range of axes. In Python, `-1` refers to the last element, and `-2` refers to the second-to-last. So in this case, this tells PyTorch that we want to take the mean ranging over the values indexed by the last two axes of the tensor. The last two axes are the horizontal and vertical dimensions of an image. After taking the mean over the last two axes, we are left with just the first tensor axis, which indexes over our images, which is why our final size was `(1010)`. In other words, for every image, we averaged the intensity of all the pixels in that image.

最後に、この関数は `mean((-1,-2))` を呼び出します。タプル(-1,-2)は、軸の範囲を表します。Python では、-1 が最後の要素、-2 が最後から 2 番目の要素を意味する。つまり、この場合、テンソルの最後の 2 つの軸で示される値の範囲の平均を取りたいことを PyTorch に伝えます。最後の 2 つの軸は、画像の水平方向と垂直方向の寸法を表しています。最後の 2 つの軸で平均を取った後、最初のテンソル軸だけが残り、それが画像のインデックスとなるため、最終的なサイズは(1010)となりました。つまり、すべての画像について、その画像内のすべてのピクセルの強度を平均化したのです。

We'll be learning lots more about broadcasting throughout this book, especially in <<chapter_foundations>>, and will be practicing it regularly too.

本書では、特に<>で放送についてもっとたくさん学び、定期的に練習していく予定です。

We can use `mnist_distance` to figure out whether an image is a 3 or not by using the following logic: if the distance between the digit in question and the ideal 3 is less than the distance to the ideal 7, then it's a 3. This function will automatically do broadcasting and be applied elementwise, just like all PyTorch functions and operators:

`mnist_distance` を使って、ある画像が 3 かどうかを調べるには、次の論理を使います：問題の数字と理想の 3 との距離が、理想の 7 との距離より小さければ、それは 3 です：

In [ ]:

```
def is_3(x): return mnist_distance(x,mean3) < mnist_distance(x,mean7)
```

Let's test it on our example case:

例のケースでテストしてみましょう：

In [ ]:

```
is_3(a_3), is_3(a_3).float()
```

Out[ ]:

```
(tensor(True), tensor(1.))
```

Note that when we convert the Boolean response to a float, we get `1.0` for `True` and `0.0` for `False`. Thanks to broadcasting, we can also test it on the full validation set of 3s:

ブーリアン応答を float に変換すると、True は 1.0、False は 0.0 になることに注意してください。broadcasting のおかげで、3s の完全な検証セットでテストすることもできる：

In [ ]:

```
is_3(valid_3_tens)
```

Out[ ]:

```
tensor([True,  True,  True,  ...,  True,  True,  True])
```

Now we can calculate the accuracy for each of the 3s and 7s by taking the average of that function for all 3s and its inverse for all 7s:

あとは、その関数の平均をすべての 3s について、その逆数をすべての 7s についてとることで、3s と 7s のそれぞれの精度を計算することができます：

In [ ]:

```
accuracy_3s = is_3(valid_3_tens).float() .mean()
accuracy_7s = (1 - is_3(valid_7_tens).float()).mean()
accuracy_3s,accuracy_7s,(accuracy_3s+accuracy_7s)/2
```

Out[ ]:

```
(tensor(0.9168),  tensor(0.9854),  tensor(0.9511))
```

This looks like a pretty good start! We're getting over 90% accuracy on both 3s and 7s, and we've seen how to define a metric conveniently using broadcasting.

これはかなり良いスタートが切れたようです！ 3 桁も 7 桁も 90%以上の精度で計算できていますし、放送を使った便利な指標の定義も確認できました。

But let's be honest: 3s and 7s are very different-looking digits. And we're only classifying 2 out of the 10 possible digits so far. So we're going to need to do better!

しかし、正直に言うと、3 桁と 7 桁は全く違う数字です。しかも、今のところ 10 桁のうち 2 桁しか分類できていない。だから、もっとうまくやる必要があるんです！

To do better, perhaps it is time to try a system that does some real learning—that is, that can automatically modify itself to improve its performance. In other words, it's time to talk about the training process, and SGD.

もっとうまくやるには、そろそろ本格的に学習するシステム、つまり、自動的に自己修正して性能を向上させるシステムを試してみる必要があるのではないでしょうか。つまり、学習プロセスや SGD について話すべき時なのです。

Stochastic Gradient Descent (SGD)

確率的勾配降下法(SGD)

Do you remember the way that Arthur Samuel described machine learning, which we quoted in <<chapter_intro>>?

<> で引用した、アーサー・サミュエルによる機械学習の説明の仕方を覚えていますか？

: Suppose we arrange for some automatic means of testing the effectiveness of any current weight assignment in terms of actual performance and provide a mechanism for altering the weight assignment so as to maximize the performance. We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would "learn" from its experience.

: 例えば、現在の重み付けの有効性を実際のパフォーマンスでテストする自動的な手段を用意し、パフォーマンスを最大化するように重み付けを変更する機構を提供するとしよう。このような手順が完全に自動化できること、そしてそのようにプログラムされた機械がその経験から「学習」することは、詳細を説明するまでもないだろう。

As we discussed, this is the key to allowing us to have a model that can get better and better—that can learn. But our pixel similarity approach does not really do this. We do not have any kind of weight assignment, or any way of improving based on testing the effectiveness of a weight assignment. In other words, we can't really improve our pixel similarity approach by modifying a set of parameters. In order to take advantage of the power of deep learning, we will first have to represent our task in the way that Arthur Samuel described it.

これこそが、より良くなるモデル、つまり学習するモデルを手に入れるための鍵なのです。しかし、私たちの画素類似性アプローチでは、これを実際に行うことはできません。重み付けもなければ、重み付けの有効性を検証して改善する方法もない。つまり、パラメータを変更することで、画素の類似性を改善することはできないのです。ディープラーニングの力を活用するためには、まず、アーサー・サミュエルが説明したような方法でタスクを表現する必要があります。

Instead of trying to find the similarity between an image and an "ideal image," we could instead look at each individual pixel and come up with a set of weights for each one, such that the highest weights are associated with those pixels most likely to be black for a particular category. For instance, pixels toward the bottom right are not very likely to be activated for a 7, so they should have a low weight for a 7, but they are likely to be activated for an 8, so they should have a high weight for an 8. This can be represented as a function and set of weight values for each possible category—for instance the probability of being the number 8:

画像と「理想的な画像」の類似度を求めるのではなく、個々の画素に注目し、特定のカテゴリで最も黒くなりそうな画素に最も高い重みが関連するような、画素ごとの重みを設定することができます。例えば、右下のピクセルは、7 で活性化する可能性はあまり高くないので、7 に対するウェイトは低く、8 で活性化する可能性が高いので、8 に対するウェイトは高くする。これは、例えば、数字が 8 である確率のように、考えられるカテゴリーごとに関数とウェイト値のセットとして表すことができる：

```
def pr_eight(x,w): return (x*w).sum()
```

Here we are assuming that `x` is the image, represented as a vector—in other words, with all of the rows stacked up end to end into a single long line. And we are assuming that the weights are a vector `w`. If we have this function, then we just need some way to update the weights to make them a little bit better. With such an approach, we can repeat that step a number of times, making the weights better and better, until they are as good as we can make them.

ここでは、x をベクトルとして表現された画像、つまり、すべての行が端から端まで積み重なって１本の長い線になっているものと仮定している。この関数があれば、あとは重みを更新して少しずつ良くしていくだけです。このような方法を使えば、何度もこのステップを繰り返して、重みをどんどん良くしていくことができます。

We want to find the specific values for the vector `w` that causes the result of our function to be high for those images that are actually 8s, and low for those images that are not. Searching for the best vector `w` is a way to search for the best function for recognising 8s. (Because we are not yet using a deep neural network, we are limited by what our function can actually do—we are going to fix that constraint later in this chapter.)

この関数の結果が、実際に 8s である画像では高く、そうでない画像では低くなるようなベクトル w の具体的な値を見つけたいのです。最適なベクトル w を探すことは、8 を認識するための最適な関数を探すことでもあります（まだディープニューラルネットワークを使っていないので、実際にできる関数は限られていますが、この章の後半でその制約を解決する予定です）。

To be more specific, here are the steps that we are going to require, to turn this function into a machine learning classifier:

より具体的には、この関数を機械学習の分類器にするために必要なステップを以下に示します：

1. *Initialize* the weights.

   重みを初期化する。

2. For each image, use these weights to *predict* whether it appears to be a 3 or a 7.

   各画像について、この重みを使って「3」に見えるか「7」に見えるか予測する。

3. Based on these predictions, calculate how good the model is (its *loss*).

   これらの予測に基づき、モデルがどの程度優れているか（損失）を計算する。

4. Calculate the *gradient*, which measures for each weight, how changing that weight would change the loss

各重みについて、その重みを変えると損失がどのように変化するかを測定する勾配を計算する。

5. *Step* (that is, change) all the weights based on that calculation.

その計算をもとに、すべての重みをステップ（つまり変更）する。

6. Go back to the step 2, and *repeat* the process.

ステップ 2 に戻り、このプロセスを繰り返す。

7. Iterate until you decide to *stop* the training process (for instance, because the model is good enough or you don't want to wait any longer).

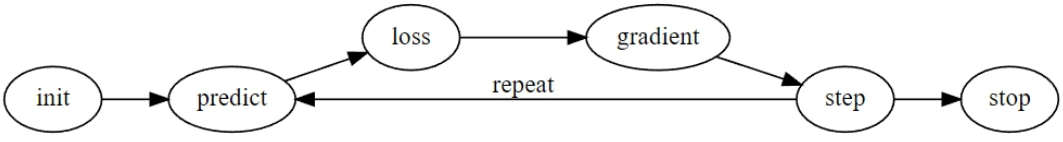
学習プロセスの停止を決定するまで繰り返す（例えば、モデルが十分に良くなった、あるいはこれ以上待ちたくないという理由で）。

These seven steps, illustrated in <<gradient_descent>>, are the key to the training of all deep learning models. That deep learning turns out to rely entirely on these steps is extremely surprising and counterintuitive. It's amazing that this process can solve such complex problems. But, as you'll see, it really does!

gradient_descent "で説明されているこの 7 つのステップは、すべてのディープラーニングモデルのトレーニングの鍵となるものである。ディープラーニングがこれらのステップに完全に依存していることが判明したことは、非常に驚きであり、直感に反しています。このプロセスで、これほど複雑な問題を解決できるのは驚きです。しかし、これからご覧になるように、本当にそうなのです！

In [ ]:

```
#id gradient_descent
#caption The gradient descent process
#alt Graph showing the steps for Gradient Descent
gv('''
init->predict->loss->gradient->step->stop step->predict[label=repeat]
''')
```



There are many different ways to do each of these seven steps, and we will be learning about them throughout the rest of this book. These are the details that make a big difference for deep learning practitioners, but it turns out that the general approach to each one generally follows some basic principles. Here are a few guidelines:

これら 7 つのステップのそれぞれにはさまざまな方法があり、本書の残りの部分を通して学んでいくことになります。これらは、深層学習の実践者にとって大きな違いを生む詳細なものですが、それぞれの一般的なアプローチは、概ねいくつかの基本原則に従っていることがわかります。以下、いくつかのガイドラインを紹介します：

- Initialize:: We initialize the parameters to random values. This may sound surprising. There are certainly other choices we could make, such as initializing them to the percentage of times that pixel is activated for that category—but since we already know that we have a routine to improve these weights, it turns out that just starting with random weights works perfectly well.

初期化する：： パラメータをランダムな値に初期化します。これは意外に聞こえるかもしれません。しかし、この重みを改善するルーチンがあることはすでに知っているので、ランダムな重みで開始するだけで、完全にうまくいくことがわかったのです。

- Loss:: This is what Samuel referred to when he spoke of *testing the effectiveness of any current weight assignment in terms of actual performance*. We need some function that will return a number that is small if the performance of the model is good (the standard approach is to treat a small loss as good, and a large loss as bad, although this is just a convention).

損失：：Samuel が、現在の重み付けの有効性を実際のパフォーマンスでテストすると言ったのは、これのことである。モデルの性能が良ければ小さい数値を返す関数が必要です（標準的なアプローチは、損失が小さいと良い、損失が大きいと悪いと扱うことですが、これは単なる慣習に過ぎません）。

- Step:: A simple way to figure out whether a weight should be increased a bit, or decreased a bit, would be just to try it: increase the weight by a small amount, and see if the loss goes up or down. Once you find the correct direction, you could then change that amount by a bit more, and a bit less, until you find an amount that works well. However, this is slow! As we will see, the magic of calculus allows us to directly figure out in which direction, and by roughly how much, to change each weight, without having to try all these small changes. The way to do this is by calculating *gradients*. This is just a performance optimization, we would get exactly the same results by using the slower manual process as well.

ステップ：： ウェイトを少し増やすべきか、減らすべきかを判断する簡単な方法は、試してみることです。ウェイトを少し増やして、損失が増えるのか減るのかを見てみましょう。方向性が決まったら、その量をもう少し増やしたり、減らしたりして、うまくいく量を見つければいいんです。ただし、これには時間がかかります！ ところが、微積分の魔法を使えば、そんな小さな変化を試さなくとも、それぞれの重さをどの方向に、どのくらい変化させればいいのかが、直接わかるようになるのです。その方法とは、勾配を計算することです。これは単なるパフォーマンスの最適化であり、より遅い手作業

でもまったく同じ結果が得られます。

- Stop:: Once we've decided how many epochs to train the model for (a few suggestions for this were given in the earlier list), we apply that decision. This is where that decision is applied. For our digit classifier, we would keep training until the accuracy of the model started getting worse, or we ran out of time.

  停止：： モデルを何エポック学習させるかを決定したら（このためのいくつかの提案は先のリストで示されました）、その決定を適用します。ここで、その決定が適用されます。数字分類器の場合、モデルの精度が悪くなり始めるか、時間がなくなるまで学習を続けることになります。

Before applying these steps to our image classification problem, let's illustrate what they look like in a simpler case. First we will define a very simple function, the quadratic—let's pretend that this is our loss function, and x is a weight parameter of the function:

この手順を画像分類の問題に適用する前に、もっと単純なケースでどのように見えるかを説明しましょう。まず、2 次関数という非常に単純な関数を定義します。これを損失関数とし、x を関数の重みパラメータとします：

In [ ]:

```
def f(x): return x**2
```

Here is a graph of that function:

その関数をグラフにしたものがこちらです：

In [ ]:

```
plot_function(f, 'x', 'x**2')
```

The sequence of steps we described earlier starts by picking some random value for a parameter, and calculating the value of the loss:

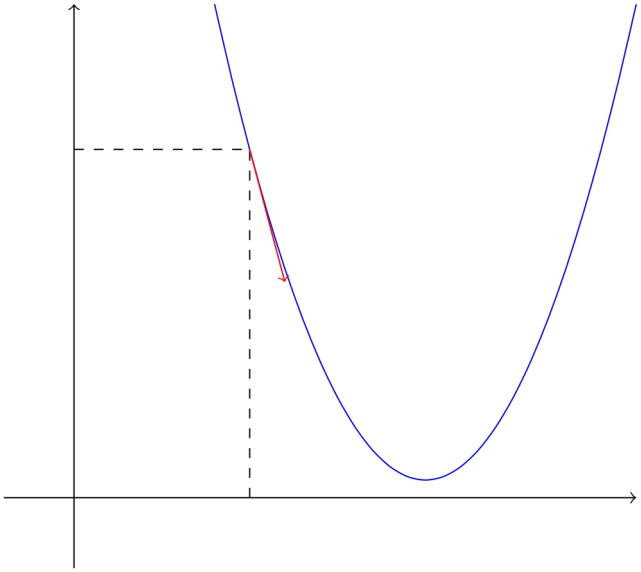
先ほど説明した一連の流れは、まずパラメータに何かランダムな値を選び、損失の値を計算することから始まります：

In [ ]:

```
plot_function(f, 'x', 'x**2') plt.scatter(-1.5, f(-1.5), color='red');
```

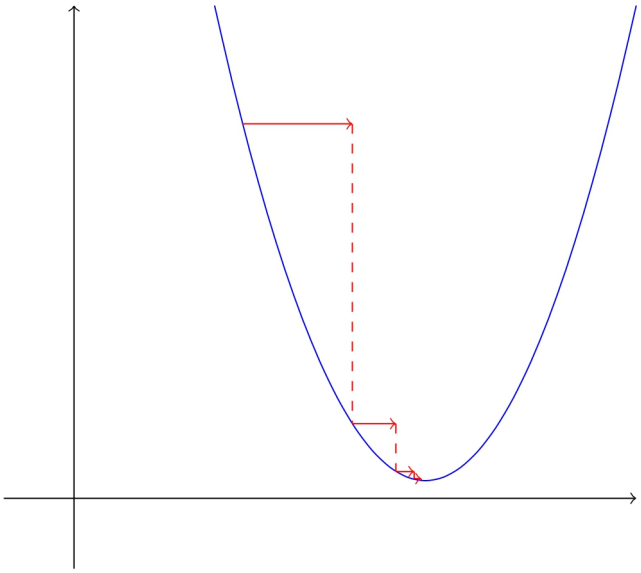
Now we look to see what would happen if we increased or decreased our parameter by a little bit—the *adjustment*. This is simply the slope at a particular point:

ここで、パラメータを少し増やしたり減らしたりするとどうなるかを見てみましょう（調整）。これは、ある点での傾きを表しています：



We can change our weight by a little in the direction of the slope, calculate our loss and adjustment again, and repeat this a few times. Eventually, we will get to the lowest point on our curve:

ある点での傾きを持つ二乗関数を示すグラフ
傾斜の方向に少しづつ重量を変え、また損失と調整を計算し、これを数回繰り返す。最終的には、カーブの一番低いところに到達することになります：



勾配降下法の説明図

This basic idea goes all the way back to Isaac Newton, who pointed out that we can optimize arbitrary functions in this way. Regardless of how complicated our functions become, this basic approach of gradient descent will not significantly change. The only minor changes we will see later in this book are some handy ways we can make it faster, by finding better steps.

この基本的な考え方は、アイザック・ニュートンが「任意の関数をこのように最適化できる」と指摘したことに端を発します。関数がどんなに複雑になっても、この勾配降下法の基本的な考え方は大きく変わることはないでしょう。本書の後半で紹介するのは、より良いステップを見つけることで、より高速に処理するための便利な方法です。

Calculating Gradients

グラデーションの計算

The one magic step is the bit where we calculate the gradients. As we mentioned, we use calculus as a performance optimization; it allows us to more quickly calculate whether our loss will go up or down when we adjust our parameters up or down. In other words, the gradients will tell us how much we have to change each weight to make our model better.

1つの魔法のステップは、勾配を計算するところです。前述したように、私たちはパフォーマンスの最適化のために微積分を使用しています。パラメータを上下に調整したときに、損失が大きくなるか小さくなるかをより迅速に計算することができるのです。つまり、勾配は、モデルをより良くするために、各重量をどれだけ変えればいいかを教えてくれるのです。

You may remember from your high school calculus class that the *derivative* of a function tells you how much a change in its parameters will change its result. If not, don't worry, lots of us forget calculus once high school is behind us! But you will have to have some intuitive understanding of what a derivative is before you continue, so if this is all very fuzzy in your head, head over to Khan Academy and complete the lessons on basic derivatives. You won't have to know how to calculate them yourselves, you just have to know what a derivative is.

高校の微積分の授業で、関数の微分を使うと、パラメータを変えたときに結果がどれだけ変わるかがわかるというのを覚えている人もいるかもしれません。そうでなくても、高校を卒業すると微積分のことは忘れてしまう人が多いので、心配しないでください！しかし、微分とは何かということを直感的に理解した上で学習を進める必要があります。もし、このことが頭の中で非常にあいまいであれば、カーンアカデミーで基本的な微分に関するレッスンを受けてください。微分の計算方法は知らなくても、微分とは何かを知っていればいいのです。

The key point about a derivative is this: for any function, such as the quadratic function we saw in the previous section, we can calculate its derivative. The derivative is another function. It calculates the change, rather than the value. For instance, the derivative of the quadratic function at the value 3 tells us how rapidly the function changes at the value 3. More specifically, you may recall that gradient is defined as *rise/run*, that is, the change in the value of the function, divided by the change in the value of the parameter. When we know how our function will change,

then we know what we need to do to make it smaller. This is the key to machine learning: having a way to change the parameters of a function to make it smaller. Calculus provides us with a computational shortcut, the derivative, which lets us directly calculate the gradients of our functions.

導関数についての重要なポイントは、前節で見た 2 次関数のような任意の関数について、その導関数を計算できることです。導関数とは、もう一つの関数です。値ではなく、変化を計算するのです。例えば、2 次関数の値 3 での導関数は、値 3 での関数の変化の速さを教えてくれます。具体的には、勾配とは「上昇 / 下降」、つまり関数の値の変化をパラメーターの値の変化で割ったものであることを思い出してください。関数がどのように変化するかが分かれば、それを小さくするために何をすべきかが分かるのです。このように、関数のパラメータを変えて小さくする方法を持つことが、機械学習の鍵なのです。微分積分では、関数の勾配を直接計算できる微分という計算上のショートカットが用意されています。

One important thing to be aware of is that our function has lots of weights that we need to adjust, so when we calculate the derivative we won't get back one number, but lots of them—a gradient for every weight. But there is nothing mathematically tricky here; you can calculate the derivative with respect to one weight, and treat all the other ones as constant, then repeat that for each other weight. This is how all of the gradients are calculated, for every weight.

注意しなければならないのは、関数にはたくさんの重みがあり、それを調整する必要があることです。そのため、導関数を計算すると、1 つの数値ではなく、たくさんの数値が返されます。しかし、数学的に難しいことは何もありません。ある重みに関する導関数を計算し、他の重みはすべて一定として扱い、他の重みについてもそれを繰り返すことができます。このようにして、すべての重みについて、すべての勾配が計算されるのです。

We mentioned just now that you won't have to calculate any gradients yourself. How can that be? Amazingly enough, PyTorch is able to automatically compute the derivative of nearly any function! What's more, it does it very fast. Most of the time, it will be at least as fast as any derivative function that you can create by hand. Let's see an example.

先ほど、グラデーションを自分で計算する必要はないといいました。どうしてでしょうか? 驚くべきことに、PyTorch はほとんどすべての関数の微分を自動的に計算することができるのです! しかも、非常に高速に計算することができます。ほとんどの場合、手で作る微分関数と同等以上の速度で計算することができます。例を見てみましょう。

First, let's pick a tensor value which we want gradients at:
まず、勾配をつけたいテンソル値を選びます:

```
xt = tensor(3.).requires_grad_()
```

Notice the special method `requires_grad_`? That's the magical incantation we use to tell PyTorch that we want to calculate gradients with respect to that variable at that value. It is essentially tagging the variable, so PyTorch will remember to keep track of how to compute gradients of the other, direct calculations on it that you will ask for.

requires_grad_という特殊なメソッドにお気づきでしょうか？これは、PyTorch に、その変数に対してその値で勾配を計算したいことを伝えるために使う魔法の呪文です。これは本質的に変数にタグを付けることで、PyTorch はあなたが要求する他の直接的な計算の勾配を計算する方法を覚えておくようになります。

> a: This API might throw you off if you're coming from math or physics. In those contexts the "gradient" of a function is just another function (i.e., its derivative), so you might expect gradient-related APIs to give you a new function. But in deep learning, "gradients" usually means the *value* of a function's derivative at a particular argument value. The PyTorch API also puts the focus on the argument, not the function you're actually computing the gradients of. It may feel backwards at first, but it's just a different perspective.
>
> a: この API は、数学や物理学から来た人なら戸惑うかもしれません。そのような文脈では、関数の「勾配」は単なる別の関数（つまりその微分）なので、勾配関連の API は新しい関数を与えるものと思ってしまうかもしれません。しかし、深層学習では、「勾配」は通常、特定の引数値における関数の導関数の値を意味します。PyTorch API では、実際に勾配を計算する関数ではなく、引数にフォーカスを当てているのです。最初は逆に感じるかもしれませんが、これは単なる視点の違いなのです。

Now we calculate our function with that value. Notice how PyTorch prints not just the value calculated, but also a note that it has a gradient function it'll be using to calculate our gradients when needed:

では、その値で関数を計算してみましょう。PyTorch は計算した値だけでなく、必要なときにグラデーションを計算するために使うグラデーション関数があることを表示していることに注目してください：

In [ ]:

```
yt = f(xt) yt
```

Out[ ]:

```
tensor(9., grad_fn=<PowBackward0>)
```

Finally, we tell PyTorch to calculate the gradients for us:

最後に、PyTorch に勾配を計算するように指示します：

In [ ]:

```
yt.backward()
```

The "backward" here refers to *backpropagation*, which is the name given to the process of calculating the derivative of each layer. We'll see how this is done exactly in chapter <<chapter_foundations>>, when we calculate the

gradients of a deep neural net from scratch. This is called the "backward pass" of the network, as opposed to the "forward pass," which is where the activations are calculated. Life would probably be easier if backward was just called calculate_grad, but deep learning folks really do like to add jargon everywhere they can!

ここでいう「バックワード」とは、バックプロパゲーションのことで、各層の微分を計算するプロセスの名称である。これが具体的にどのように行われるかは、<>の章で、ディープニューラルネットの勾配をゼロから計算するときに確認します。これは、ネットワークの「後方パス」と呼ばれ、活性度を計算する「前方パス」とは対照的である。後方パスが単に calculate_grad と呼ばれるのであれば、もっと簡単なのでしょうが、深層学習の専門家は、できる限り専門用語を追加したがりますね！

We can now view the gradients by checking the grad attribute of our tensor:

テンソルの grad 属性をチェックすることで、勾配を見ることができるようになりました：

In [ ]:

```
xt.grad
```

Out[ ]:

```
tensor(6.)
```

If you remember your high school calculus rules, the derivative of x**2 is 2*x, and we have x=3, so the gradients should be 2*3=6, which is what PyTorch calculated for us!

高校の微積分のルールを覚えているなら、x**2 の微分は 2*x で、x=3 なので、勾配は 2*3=6 となり、PyTorch が計算してくれた通りになるはずです！

Now we'll repeat the preceding steps, but with a vector argument for our function:

では、前のステップを繰り返しますが、関数の引数をベクトルにします：

In [ ]:

```
xt = tensor([3.,4.,10.]).requires_grad_() xt
```

Out[ ]:

```
tensor([ 3.,  4., 10.], requires_grad=True)
```

And we'll add sum to our function so it can take a vector (i.e., a rank-1 tensor), and return a scalar (i.e., a rank-0 tensor):

そして、ベクトル（つまりランク１のテンソル）を受け取り、スカラー（つまりランク０のテンソル）を返すことができるように、この関数に sum を追加します：

In [ ]:

```
def f(x): return (x**2).sum()
yt = f(xt)
yt
```

Out[ ]:

tensor(125., grad_fn=<SumBackward0>)

Our gradients are 2*xt, as we'd expect!

勾配は予想通り 2*xt です！

In [ ]:

```
yt.backward() xt.grad
```

Out[ ]:

tensor([6., 8., 20.])

The gradients only tell us the slope of our function, they don't actually tell us exactly how far to adjust the parameters. But it gives us some idea of how far; if the slope is very large, then that may suggest that we have more adjustments to do, whereas if the slope is very small, that may suggest that we are close to the optimal value.

勾配は関数の傾きを示すだけで、実際にどの程度パラメータを調整すればよいかを教えてくれるわけではありません。勾配が非常に大きい場合は、もっと調整する必要があることを示唆し、逆に勾配が非常に小さい場合は、最適値に近づいていることを示唆します。

Stepping With a Learning Rate

学習率でステップを踏む

Deciding how to change our parameters based on the values of the gradients is an important part of the deep learning process. Nearly all approaches start with the basic idea of multiplying the gradient by some small number, called the *learning rate* (LR). The learning rate is often a number between 0.001 and 0.1, although it could be anything. Often, people select a learning rate just by trying a few, and finding which results in the best model after training (we'll show you a better approach later in this book, called the *learning rate finder*). Once you've picked a learning rate, you can adjust your parameters using this simple function:

勾配の値に基づいてパラメータをどのように変更するかを決定することは、深層学習プロセスの重要な部分である。ほぼすべてのアプローチは、勾配に学習率（LR）と呼ばれる小さな数値を乗じるという基本的な考え方から始まります。学習率は 0.001 から 0.1 の間の数値であることが多いが、何でもよいわけではない。多くの場合、学習率の選択は、いくつか試してみて、訓練後に最も良いモデルになるものを見つけることによって行われる（本書の後半で、学習率ファインダーと呼ばれるより良いアプローチを紹介する）。学習率を選んだら、次のような簡単な関数を使ってパラメータを調整することができる：

```
w -= gradient(w) * lr
```

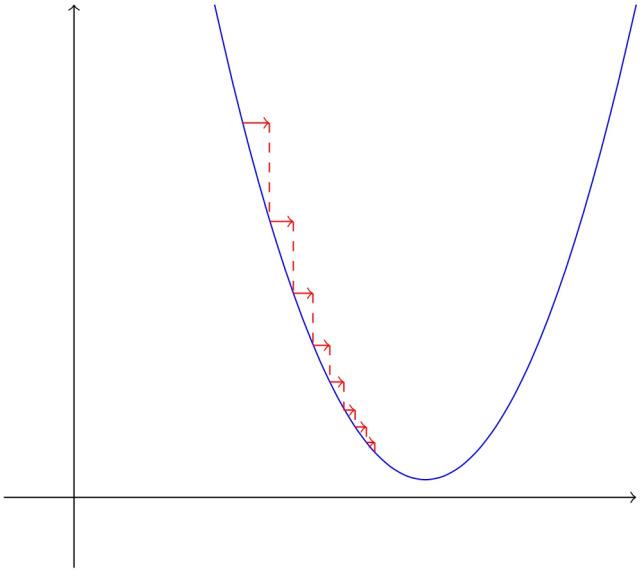
This is known as *stepping* your parameters, using an *optimizer step*. Notice how we *subtract* the gradient * lr from the parameter to update it. This allows us to adjust the parameter in the direction of the slope by increasing

the parameter when the slope is negative and decreasing the parameter when the slope is positive. We want to adjust our parameters in the direction of the slope because our goal in deep learning is to *minimize* the loss.

これは、オプティマイザーのステップを使用して、パラメータをステップ化することとして知られています。パラメータを更新するために、パラメータから勾配 * lr を差し引いていることに注目してください。これにより、傾きが負のときはパラメータを増加させ、正のときはパラメータを減少させることで、傾きの方向にパラメータを調整することができます。パラメータを傾きの方向に調整したいのは、深層学習の目標が損失を最小化することだからです。

If you pick a learning rate that's too low, it can mean having to do a lot of steps. <<descent_small>> illustrates that.

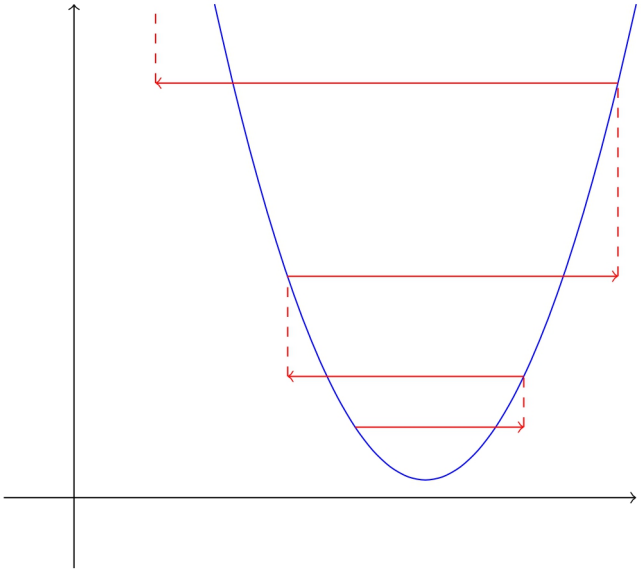
学習率が低すぎるものを選ぶと、多くのステップを踏まなければならないことになりかねません。<>がそれを示しています。



LR が低すぎる勾配降下の説明図

But picking a learning rate that's too high is even worse—it can actually result in the loss getting *worse*, as we see in <<descent_div>>!

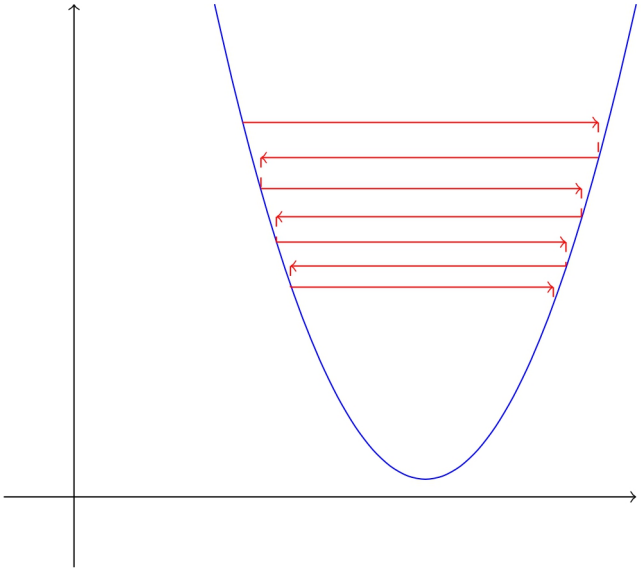
しかし、高すぎる学習率を選ぶと、<>にあるように、損失が悪化してしまうことがあります！



LR が高すぎる勾配降下法の図解

If the learning rate is too high, it may also "bounce" around, rather than actually diverging; <<descent_bouncy>> shows how this has the result of taking many steps to train successfully.

学習率が高すぎると、実際に発散するのではなく、「バウンド」してしまうこともあります。



バウンシー LR を用いた勾配降下法の図解

Now let's apply all of this in an end-to-end example.

では、これらすべてをエンドツーエンドの例で応用してみましょう。

An End-to-End SGD Example

エンドツーエンドの SGD の例

We've seen how to use gradients to find a minimum. Now it's time to look at an SGD example and see how finding a minimum can be used to train a model to fit data better.

ここまで、勾配を利用して最小値を求める方法について見てきました。今度は、SGD の例を見て、最小値を求めることが、データによりよく適合するモデルを訓練するためにどのように使用されるかを見てみましょう。

Let's start with a simple, synthetic, example model. Imagine you were measuring the speed of a roller coaster as it went over the top of a hump. It would start fast, and then get slower as it went up the hill; it would be slowest at the top, and it would then speed up again as it went downhill. You want to build a model of how the speed changes over time. If you were measuring the speed manually every second for 20 seconds, it might look something like this:

まず、簡単な合成モデルの例から始めましょう。ジェットコースターがコブの頂上を通過するときの速度を測定しているとします。最初は速く、坂を上るにつれて遅くなり、頂上で最も遅くなり、下り坂になると再び速くなる。あなたは、時間の経過とともに速度がどのように変化するかというモデルを作りたい。20 秒間、1 秒ごとに手動で速度を測定していた場合、次のようになります：

In [ ]:

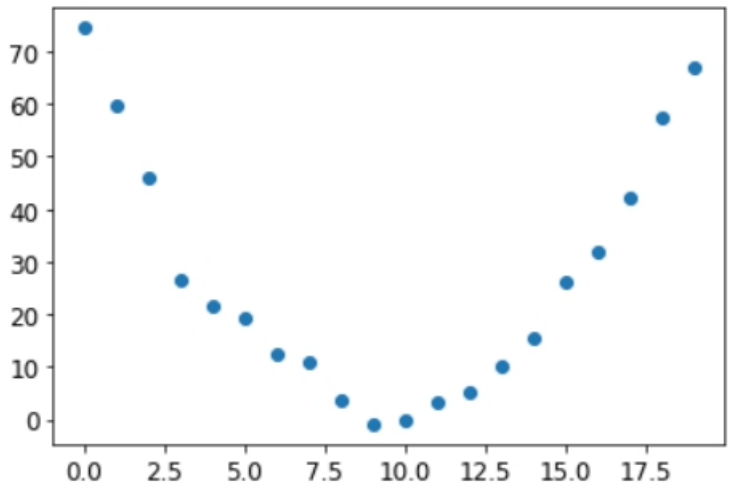
```
time = torch.arange(0,20).float(); time
```

Out[ ]:

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
        15., 16., 17., 18., 19.])
```

In [ ]:

```
speed = torch.randn(20)*3 + 0.75*(time-9.5)**2 + 1
plt.scatter(time,speed);
```



We've added a bit of random noise, since measuring things manually isn't precise. This means it's not that easy to answer the question: what was the roller coaster's speed? Using SGD we can try to find a function that matches our observations. We can't consider every possible function, so let's use a guess that it will be quadratic; i.e., a function of the form `a*(time**2)+(b*time)+c`.

手動で計測するのは正確ではないので、ランダムなノイズを少し加えています。つまり、「ジェットコースターの速度はどれくらいだったのか」という質問に答えるのは、それほど簡単ではないということです。SGD を使用すると、観測結果に一致する関数を見つけることができます。すべての可能な関数を検討することはできないので、2 次関数であろうという推測を使用することにしましょう。

We want to distinguish clearly between the function's input (the time when we are measuring the coaster's speed) and its parameters (the values that define *which* quadratic we're trying). So, let's collect the parameters in one argument and thus separate the input, `t`, and the parameters, `params`, in the function's signature:

関数の入力（コースターの速度を測定している時間）とパラメータ（どの 2 次関数を試すかを定義する値）を明確に区別したい。そこで、パラメータを 1 つの引数に集め、関数のシグネチャで入力である t とパラメータである params を分離することにしましょう：

In [ ]:

```
def f(t, params):
    a,b,c = params
    return a*(t**2) + (b*t) + c
```

In other words, we've restricted the problem of finding the best imaginable function that fits the data, to finding the best *quadratic* function. This greatly simplifies the problem, since every quadratic function is fully defined by the three parameters `a`, `b`, and `c`. Thus, to find the best quadratic function, we only need to find the best values for `a`, `b`, and `c`.

つまり、データに適合する最良の関数を探すという問題を、最良の 2 次関数を探すという問題に限定したのです。なぜなら、すべての 2 次関数は 3 つのパラメータ a、b、c で完全に定義されるからです。したがって、最良の 2 次関数を見つけるには、a、b、c の最適値を見つければよいのです。

If we can solve this problem for the three parameters of a quadratic function, we'll be able to apply the same approach for other, more complex functions with more parameters—such as a neural net. Let's find the parameters for `f` first, and then we'll come back and do the same thing for the MNIST dataset with a neural net.

二次関数の 3 つのパラメータについてこの問題を解くことができれば、より多くのパラメータを持つ他の複雑な関数、例えばニューラルネットについても同じ方法を適用することができるようになる。まずは f のパラメータを求め、その後、MNIST データセットに対してニューラルネットで同じことをやってみましょう。

We need to define first what we mean by "best." We define this precisely by choosing a *loss function*, which will return a value based on a prediction and a target, where lower values of the function correspond to "better" predictions. It is important for loss functions to return *lower* values when predictions are more accurate, as the SGD procedure we defined earlier will try to *minimize* this loss. For continuous data, it's common to use *mean squared error*:

まず、"ベスト "とはどういう意味かを定義する必要があります。損失関数は、予測値と目標値に基づいて値を返すもので、関数の値が小さいほど「より良い」予測に対応します。損失関数は、予測がより正確な場合に低い値を返すことが重要であり、先に定義した SGD 手順ではこの損失を最小化しようとします。連続データでは、平均二乗誤差を使用するのが一般的です：

In [ ]:

```
def mse(preds, targets): return ((preds-targets)**2).mean()
```

Now, let's work through our 7 step process.

では、7 つのステップで作業してみましょう。

Step 1: Initialize the parameters

ステップ 1：パラメータの初期化

First, we initialize the parameters to random values, and tell PyTorch that we want to track their gradients, using requires_grad_:

まず、パラメータをランダムな値に初期化し、requires_grad_を使用して、その勾配を追跡したいことを PyTorch に指示します：

In [ ]:

```
params = torch.randn(3).requires_grad_()
```

In [ ]:

```
#hide
orig_params = params.clone()
```

Step 2: Calculate the predictions

ステップ 2：予測値の算出

Next, we calculate the predictions

次に、予測値を計算する：

:

In [ ]:

```
preds = f(time, params)
```

Let's create a little function to see how close our predictions are to our targets, and take a look:

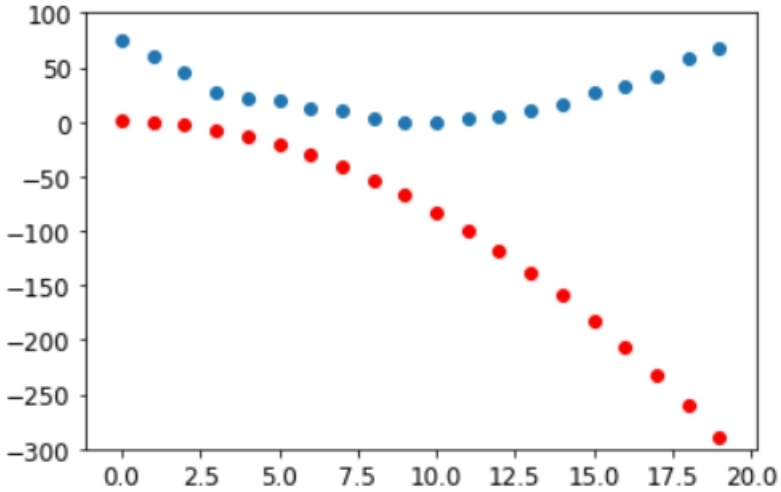
予測値が目標値にどれだけ近いかを見るためのちょっとした関数を作って見てみましょう：

In [ ]:

```
def show_preds(preds, ax=None):
    if ax is None:
    ax=plt.subplots()[1]
    ax.scatter(time, speed)
    ax.scatter(time, to_np(preds), color='red')
    ax.set_ylim(-300,100)
```

In [ ]:

```
show_preds(preds)
```



This doesn't look very close—our random parameters suggest that the roller coaster will end up going backwards, since we have negative speeds!

ランダムなパラメータは、ジェットコースターの速度がマイナスであることから、ジェットコースターが後方に進むことを示唆しています！

Step 3: Calculate the loss

ステップ 3: 損失を計算する

We calculate the loss as follows:

損失は次のように計算します：

In [ ]:

```
loss = mse(preds, speed) loss
```

Out[ ]:

tensor(25823.8086, grad_fn=<MeanBackward0>)

Our goal is now to improve this. To do that, we'll need to know the gradients.

これからは、これを改善することが目標です。そのためには、グラデーションを知っておく必要があります。

Step 4: Calculate the gradients

ステップ 4：グラデーションを計算する

The next step is to calculate the gradients. In other words, calculate an approximation of how the parameters need to change:

次に、勾配を計算します。つまり、パラメータがどのように変化する必要があるのか、その近似値を計算する：

In [ ]:

```
loss.backward()
params.grad
```

Out[ ]:

```
tensor([-53195.8594,  -3419.7146,   -253.8908])
```

In [ ]:

```
params.grad * 1e-5
```

Out[ ]:

```
tensor([-0.5320, -0.0342, -0.0025])
```

We can use these gradients to improve our parameters. We'll need to pick a learning rate (we'll discuss how to do that in practice in the next chapter; for now we'll just use 1e-5, or 0.00001):

この勾配を利用して、パラメーターを改善することができます。学習率を選ぶ必要があります（次の章で実際にどうするか説明します。今は 1e-5、つまり 0.00001 を使います）：

In [ ]:

```
params
```

Out[ ]:

```
tensor([-0.7658, -0.7506,  1.3525], requires_grad=True)
```

Step 5: Step the weights.

ステップ 5：重みをステップ

Now we need to update the parameters based on the gradients we just calculated:

ここで、先ほど計算した勾配をもとにパラメータを更新する必要があります：

In [ ]:

```
lr = 1e-5
params.data -= lr * params.grad.data
```

```
params.grad = None
```

> a: Understanding this bit depends on remembering recent history. To calculate the gradients we call `backward` on the `loss`. But this `loss` was itself calculated by `mse`, which in turn took `preds` as an input, which was calculated using `f` taking as an input `params`, which was the object on which we originally called `requires_grad_`—which is the original call that now allows us to call `backward` on `loss`. This chain of function calls represents the mathematical composition of functions, which enables PyTorch to use calculus's chain rule under the hood to calculate these gradients.

> a: このビットを理解するには、最近の歴史を覚えておくことが必要です。勾配を計算するために、我々は損失について逆算する。しかし、この損失は mse によって計算されたもので、mse は preds を入力とし、f は params を入力として計算されたもので、このオブジェクトはもともと requires_grad_ を呼んだオブジェクトであり、今損失を逆算して呼ぶことができるオリジナルの呼び出しなのです。この関数呼び出しの連鎖は関数の数学的合成を表しており、PyTorch は微積分の連鎖法則を利用してこれらの勾配を計算することができるのです。

Let's see if the loss has improved:

損失が改善されたかどうか見てみましょう：

In [ ]:

```
preds = f(time,params)
mse(preds, speed)
```

Out[ ]:

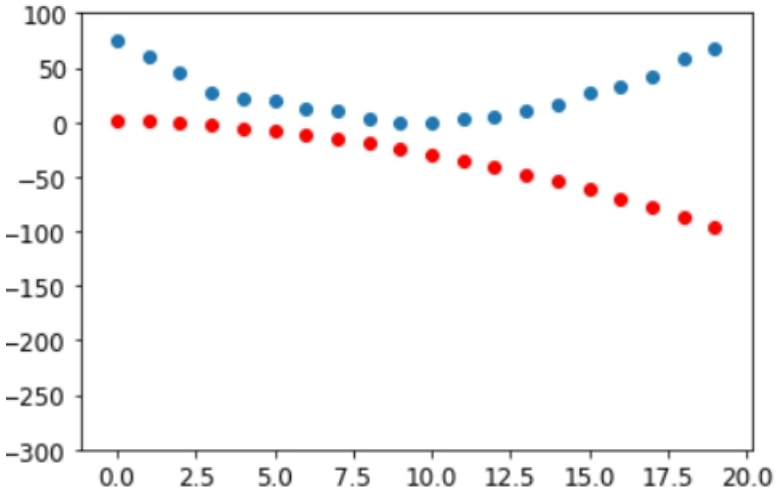
```
tensor(5435.5366, grad_fn=<MeanBackward0>)
```

And take a look at the plot:

そして、プロットを見てみましょう：

In [ ]:

```
show_preds(preds)
```



We need to repeat this a few times, so we'll create a function to apply one step:

これを何度か繰り返す必要があるので、1 ステップを適用する関数を作っておきます：

In [ ]:

```
def apply_step(params, prn=True):
    preds = f(time, params)
    loss = mse(preds, speed)
    loss.backward()
    params.data -= lr * params.grad.data
    params.grad = None
    if prn: print(loss.item())
    return preds
```

Step 6: Repeat the process

Step 6: 繰り返す

Now we iterate. By looping and performing many improvements, we hope to reach a good result:

今度は反復します。ループして何度も改良を行うことで、良い結果にたどり着きたいと考えています：

In [ ]:

```
for i in range(10): apply_step(params)
```

```
5435.53662109375
1577.4495849609375
847.3780517578125
709.22265625
683.0757446289062
```

678.12451171875
677.1839599609375
677.0025024414062
676.96435546875
676.9537353515625

In [ ]:

```
#hide
params = orig_params.detach().requires_grad_()
```

The loss is going down, just as we hoped! But looking only at these loss numbers disguises the fact that each iteration represents an entirely different quadratic function being tried, on the way to finding the best possible quadratic function. We can see this process visually if, instead of printing out the loss function, we plot the function at every step. Then we can see how the shape is approaching the best possible quadratic function for our data:

期待通り、損失は減少しています！しかし、この損失の数値だけを見ていると、各反復は、最良の二次関数を見つけるために、全く異なる二次関数が試されているという事実が見えなくなってしまう。この過程を視覚的に見るには、損失関数をプリントアウトするのではなく、各ステップでの関数をプロットすればよいのです。そして、その形状がデータに対して最適な 2 次関数に近づいていく様子を見ることができます：

In [ ]:

```
_,axs = plt.subplots(1,4,figsize=(12,3))
for ax in axs: show_preds(apply_step(params, False), ax)
plt.tight_layout()
```

Step 7: stop

ステップ 7：停止

We just decided to stop after 10 epochs arbitrarily. In practice, we would watch the training and validation losses and our metrics to decide when to stop, as we've discussed.

10 エポック後に任意に停止することを決めただけです。実際には、これまで説明したように、訓練と検証の損失と我々のメトリクスを見て、停止するタイミングを決定することになります。

Summarizing Gradient Descent

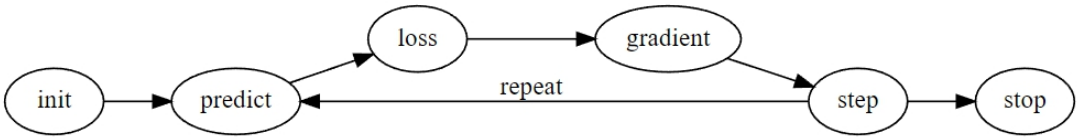
勾配降下法のまとめ

In [ ]:

```
#hide_input
#id gradient_descent
#caption The gradient descent process
```

```
#alt Graph showing the steps for Gradient Descent
gv('''
init->predict->loss->gradient->step->stop step->predict[label=repeat]
''')
```

Out[ ]:



To summarize, at the beginning, the weights of our model can be random (training *from scratch*) or come from a pretrained model (*transfer learning*). In the first case, the output we will get from our inputs won't have anything to do with what we want, and even in the second case, it's very likely the pretrained model won't be very good at the specific task we are targeting. So the model will need to *learn* better weights.

要約すると、最初にモデルの重みをランダムにする（ゼロからトレーニングする）か、事前に学習させたモデルから得る（転移学習）かです。最初の場合、入力から得られる出力は、私たちが望むものとは全く関係ありません。そのため、モデルはより良い重みを学習する必要があります。

We begin by comparing the outputs the model gives us with our targets (we have labeled data, so we know what result the model should give) using a *loss function*, which returns a number that we want to make as low as possible by improving our weights. To do this, we take a few data items (such as images) from the training set and feed them to our model. We compare the corresponding targets using our loss function, and the score we get tells us how wrong our predictions were. We then change the weights a little bit to make it slightly better.

まず、モデルが出す出力と目標値（ラベル付きデータがあるので、モデルが出すべき結果はわかっている）を損失関数で比較します。損失関数は、重みを改善することによって、できるだけ低くしたい数値を返します。そのために、トレーニングセットからいくつかのデータ（画像など）を取り出して、モデルに与えます。損失関数を使って対応するターゲットを比較し、得られたスコアによって、予測がどれだけ間違っていたかを知ることができます。そして、重みを少し変えて、少しでも良くなるようにします。

To find how to change the weights to make the loss a bit better, we use calculus to calculate the *gradients*. (Actually, we let PyTorch do it for us!) Let's consider an analogy. Imagine you are lost in the mountains with your car parked at the lowest point. To find your way back to it, you might wander in a random direction, but that probably wouldn't help much. Since you know your vehicle is at the lowest point, you would be better off going downhill. By always taking a step in the direction of the steepest downward slope, you should eventually arrive at your destination. We use the magnitude of the gradient (i.e., the steepness of the slope) to tell us how big a step to take; specifically, we

multiply the gradient by a number we choose called the *learning rate* to decide on the step size. We then *iterate* until we have reached the lowest point, which will be our parking lot, then we can *stop*.

重みをどう変えれば損失が少し改善されるかを知るために、微積分を用いて勾配を計算します。(実は PyTorch にやらせています！）例えを考えてみましょう。あなたが山で道に迷い、一番低いところに車が止まっているとします。そこに戻るには、適当な方向に歩き回ればいいのですが、それではあまり役に立ちません。自分の車が一番低いところにあることが分かっているのだから、下山したほうがいいに決まっている。常に急な下り坂の方向に一歩ずつ進むことで、最終的に目的地に到着するはずです。具体的には、勾配の大きさに学習率と呼ばれる数値を掛け合わせて、ステップの大きさを決めます。具体的には、勾配に学習率と呼ばれる数値を掛け合わせ、ステップの大きさを決めます。そして、一番低い地点（駐車場）に到達するまで繰り返し、そこでストップします。

All of that we just saw can be transposed directly to the MNIST dataset, except for the loss function. Let's now see how we can define a good training objective.

今見てきたことは、損失関数を除いて、すべて MNIST データセットにそのまま転用することができる。では、どのようにすれば良いトレーニング目的を定義できるかを見てみましょう。

The MNIST Loss Function

MNIST の損失関数

We already have our independent variables `x`—these are the images themselves. We'll concatenate them all into a single tensor, and also change them from a list of matrices (a rank-3 tensor) to a list of vectors (a rank-2 tensor). We can do this using `view`, which is a PyTorch method that changes the shape of a tensor without changing its contents. `-1` is a special parameter to `view` that means "make this axis as big as necessary to fit all the data":

独立変数 x-これらは画像そのものである-はすでに持っている。これらを 1 つのテンソルに連結し、さらに行列のリスト（ランク 3 テンソル）からベクトルのリスト（ランク 2 テンソル）に変更することにします。これは、テンソルの中身を変えずに形を変える PyTorch のメソッドである view を使って行うことができます。-1 というのは view の特別なパラメータで、「この軸をすべてのデータにフィットするために必要なだけ大きくする」という意味です：

In [ ]:

```
train_x = torch.cat([stacked_threes, stacked_sevens]).view(-1, 28*28)
```

We need a label for each image. We'll use `1` for 3s and `0` for 7s:

各画像にラベルが必要です。3 は 1、7 は 0 を使うことにする：

In [ ]:

```
train_y = tensor([1]*len(threes) + [0]*len(sevens)).unsqueeze(1)
train_x.shape,train_y.shape
```

Out[ ]:

```
(torch.Size([12396, 784]), torch.Size([12396, 1]))
```

A Dataset in PyTorch is required to return a tuple of (x,y) when indexed. Python provides a zip function which, when combined with list, provides a simple way to get this functionality:

PyTorch の Dataset は、インデックスを作成すると(x,y)のタプルを返すことが要求されます。Python は zip 関数を提供しており、これを list と組み合わせると、この機能を簡単に手に入れることができます：

In [ ]:

```
dset = list(zip(train_x,train_y)) x,y = dset[0] x.shape,y
```

Out[ ]:

```
(torch.Size([784]), tensor([1]))
```

In [ ]:

```
valid_x = torch.cat([valid_3_tens, valid_7_tens]).view(-1, 28*28)
valid_y = tensor([1]*len(valid_3_tens) + [0]*len(valid_7_tens)).unsqueeze(1)
valid_dset = list(zip(valid_x,valid_y))
```

Now we need an (initially random) weight for every pixel (this is the *initialize* step in our seven-step process):

ここで、各ピクセルの（最初はランダムな）重みが必要になります（これは 7 ステップのプロセスのうち、初期化のステップです）：

In [ ]:

```
def init_params(size, std=1.0): return (torch.randn(size)*std).requires_grad_()
```

In [ ]:

```
weights = init_params((28*28,1))
```

The function weights*pixels won't be flexible enough—it is always equal to 0 when the pixels are equal to 0 (i.e., its *intercept* is 0). You might remember from high school math that the formula for a line is y=w*x+b; we still need the b. We'll initialize it to a random number too:

ピクセルが 0 になると常に 0 になる（つまり切片が 0 になる）ので、weights*pixels という関数は柔軟性に欠ける。高校の数学で、直線の公式が y=w*x+b であることを思い出したかもしれません：

In [ ]:

```
bias = init_params(1)
```

In neural networks, the w in the equation y=w*x+b is called the *weights*, and the b is called the *bias*. Together, the weights and bias make up the *parameters*.

ニューラルネットワークでは、y=w*x+b という方程式の w を重み、b をバイアスと呼びます。重みとバイアスを合わせて、パラメータを構成します。

> jargon: Parameters: The *weights* and *biases* of a model. The weights are the w in the equation w*x+b, and the biases are the b in that equation.
>
> 専門用語： パラメーターのこと： モデルの重みとバイアスのこと。重みは方程式 w*x+b の w で、バイアスはその方程式の b である。

We can now calculate a prediction for one image:

これで、1 枚の画像に対する予測値を計算できるようになった：

In [ ]:

```
(train_x[0]*weights.T).sum() + bias
```

Out[ ]:

```
tensor([20.2336], grad_fn=<AddBackward0>)
```

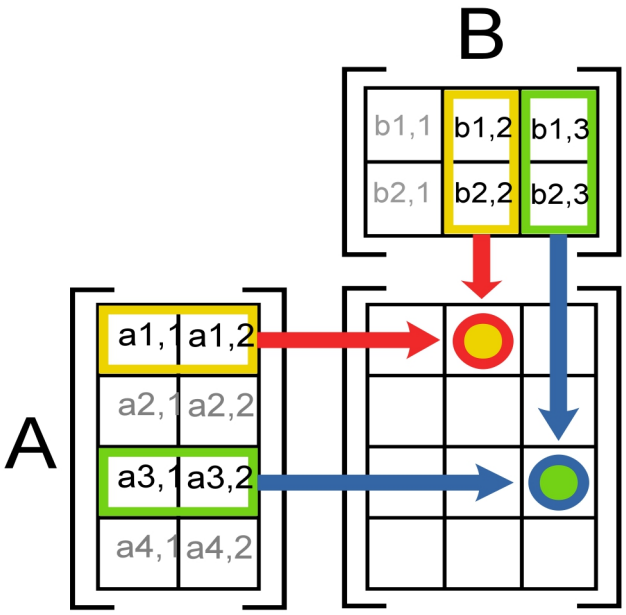
While we could use a Python for loop to calculate the prediction for each image, that would be very slow. Because Python loops don't run on the GPU, and because Python is a slow language for loops in general, we need to represent as much of the computation in a model as possible using higher-level functions.

Python の for ループを使って各画像の予測値を計算することもできますが、それでは非常に遅くなってしまいます。Python のループは GPU 上で動作しないし、Python は一般的にループが遅い言語なので、モデル内の計算をできるだけ高位の関数で表現する必要があります。

In this case, there's an extremely convenient mathematical operation that calculates w*x for every row of a matrix—it's called *matrix multiplication*. <> shows what matrix multiplication looks like.

この場合、行列の各行に対して w*x を計算する非常に便利な数学演算があり、行列の乗算と呼ばれています。<>は、行列の乗算の様子を示しています。

行列の乗算



This image shows two matrices, A and B, being multiplied together. Each item of the result, which we'll call AB, contains each item of its corresponding row of A multiplied by each item of its corresponding column of B, added together. For instance, row 1, column 2 (the yellow dot with a red border) is calculated as $a_{1,1} * b_{1,2} + a_{1,2} * b_{2,2}$. If you need a refresher on matrix multiplication, we suggest you take a look at the Intro to Matrix Multiplication on *Khan Academy*, since this is the most important mathematical operation in deep learning.

この画像は、2 つの行列 A、B を掛け合わせたものである。AB と呼ぶ結果の各項目は、A の対応する行の各項目と B の対応する列の各項目を掛け合わせたものを足したものです。例えば、1 行 2 列（赤枠の黄色い点）は次のように計算されます。
. 行列の掛け算はディープラーニングで最も重要な数学的操作なので、行列の掛け算について再確認したい場合は、Khan Academy の Intro to Matrix Multiplication をご覧になることをお勧めします。

In Python, matrix multiplication is represented with the @ operator. Let's try it:

Python では、行列の掛け算は@演算子で表します。試してみましょう：

In [ ]:

```
def linear1(xb): return xb@weights + bias
preds = linear1(train_x)
preds
```

Out[ ]:

```
tensor([[20.2336],
        [17.0644],
```

[15.2384],
…,
[18.3804],
[23.8567],
[28.6816]], grad_fn=<AddBackward0>)

The first element is the same as we calculated before, as we'd expect. This equation, batch@weights + bias, is one of the two fundamental equations of any neural network (the other one is the *activation function*, which we'll see in a moment).

最初の要素は、予想通り、前に計算したものと同じです。この式、batch@weights + bias は、あらゆるニューラルネットワークの 2 つの基本方程式のうちの 1 つです（もう 1 つは活性化関数で、これは後で見ていきます）。

Let's check our accuracy. To decide if an output represents a 3 or a 7, we can just check whether it's greater than 0.0, so our accuracy for each item can be calculated (using broadcasting, so no loops!) with:

精度を確認してみましょう。出力が 3 か 7 かを判断するには、0.0 より大きいかどうかをチェックすればよいので、各項目の精度は次のように計算できます（ブロードキャストを使っているのでループはありません！）：

In [ ]:

```
corrects = (preds>0.0).float() == train_y corrects
```

Out[ ]:

```
tensor([[ True],
        [ True],
        [ True],
        …,
        [False],
        [False],
        [False]])
```

In [ ]:

```
corrects.float().mean().item()
```

Out[ ]:

```
0.4912068545818329
```

Now let's see what the change in accuracy is for a small change in one of the weights (note that we have to ask PyTorch not to calculate gradients as we do this, which is what with torch.no_grad() is doing here):

では、重みの１つを少し変えたときの精度の変化を見てみましょう（このとき、PyTorch に勾配を計算しないよう依頼する必要があることに注意してください：

In [ ]:

```
with torch.no_grad(): weights[0] *= 1.0001
```

In [ ]:

```
preds = linear1(train_x) ((preds>0.0).float() == train_y).float().mean().item()
```

Out[ ]:

```
0.4912068545818329
```

As we've seen, we need gradients in order to improve our model using SGD, and in order to calculate gradients we need some *loss function* that represents how good our model is. That is because the gradients are a measure of how that loss function changes with small tweaks to the weights.

これまで見てきたように、SGD を使ってモデルを改良するには勾配が必要です。勾配を計算するためには、モデルの良し悪しを表す損失関数が必要です。なぜなら、勾配は、重みを少し調整したときに、その損失関数がどのように変化するかを示す尺度だからです。
So, we need to choose a loss function. The obvious approach would be to use accuracy, which is our metric, as our loss function as well. In this case, we would calculate our prediction for each image, collect these values to calculate an overall accuracy, and then calculate the gradients of each weight with respect to that overall accuracy.

そこで、損失関数を選択する必要があります。わかりやすいのは、指標である「精度」を損失関数として使うことです。この場合、各画像の予測値を計算し、その値を集めて総合精度を算出し、その総合精度に対する各重みの勾配を計算することになります。

Unfortunately, we have a significant technical problem here. The gradient of a function is its *slope*, or its steepness, which can be defined as *rise over run*—that is, how much the value of the function goes up or down, divided by how much we changed the input. We can write this in mathematically as: `(y_new - y_old) / (x_new - x_old)`. This gives us a good approximation of the gradient when `x_new` is very similar to `x_old`, meaning that their difference is very small. But accuracy only changes at all when a prediction changes from a 3 to a 7, or vice versa. The problem is that a small change in weights from `x_old` to `x_new` isn't likely to cause any prediction to change, so `(y_new - y_old)` will almost always be 0. In other words, the gradient is 0 almost everywhere.

しかし、ここで重大な技術的問題が発生します。関数の勾配とは、その傾き、つまり急勾配のことで、rise over run、つまり関数の値がどれだけ上がったか下がったかを、入力をどれだけ変えたかで割ったものと定義することができる。これを数学的に書くと、次のようになります： `(y_new - y_old) / (x_new - x_old)` です。これは、x_new が x_old と非常に似ているとき、つまり両者の差が非常に小さいときに、勾配の良い近似値を与えてくれます。しかし、精度がまったく変化しないのは、予測が 3 から 7 に変化したとき、あるいはその逆のときだけです。問題は、x_old から x_new への重みの小さな変化では、どの予測も変化しないので、（y_new - y_old）はほとんど常に 0 になることである。

A very small change in the value of a weight will often not actually change the accuracy at all. This means it is not useful to use accuracy as a loss function—if we do, most of the time our gradients will actually be 0, and the model will not be able to learn from that number.

重みの値をごく小さく変更しても、実際には精度がまったく変わらないことが多い。つまり、精度を損失関数として使うのはあまり意味がない。そうすると、ほとんどの場合、勾配は実際には 0 になり、モデルはその数値から学習することができなくなる。

> S: In mathematical terms, accuracy is a function that is constant almost everywhere (except at the threshold, 0.5), so its derivative is nil almost everywhere (and infinity at the threshold). This then gives gradients that are 0 or infinite, which are useless for updating the model.
>
> S: 数学的に言うと、accuracy はほぼどこでも（閾値の 0.5 を除いて）一定なので、その微分はほぼどこでも（閾値では無限大）ゼロになる関数なんだ。この場合、勾配は 0 か無限大となり、モデルを更新するのに役に立ちません。

Instead, we need a loss function which, when our weights result in slightly better predictions, gives us a slightly better loss. So what does a "slightly better prediction" look like, exactly? Well, in this case, it means that if the correct answer is a 3 the score is a little higher, or if the correct answer is a 7 the score is a little lower.

その代わりに、重み付けをした結果、少し良い予測ができたときに、少し良い損失を与えてくれる損失関数が必要です。では、「少し良い予測」とは、具体的にどのようなものでしょうか。この場合、正解が「3」であればスコアが少し高くなり、「7」であればスコアが少し低くなるということです。

Let's write such a function now. What form does it take?

では、そんな関数を書いてみましょう。どのような形になるのでしょうか。

The loss function receives not the images themselves, but the predictions from the model. Let's make one argument, prds, of values between 0 and 1, where each value is the prediction that an image is a 3. It is a vector (i.e., a rank-1 tensor), indexed over the images.

損失関数は、画像そのものではなく、モデルからの予測値を受け取ります。prds は 0 から 1 までの値からなる引数で、各値は画像が 3 であるという予測値である。これはベクトル（つまりランク 1 のテンソル）で、画像の上にインデックスが付けられている。

The purpose of the loss function is to measure the difference between predicted values and the true values — that is, the targets (aka labels). Let's make another argument, trgts, with values of 0 or 1 which tells whether an image actually is a 3 or not. It is also a vector (i.e., another rank-1 tensor), indexed over the images.

損失関数の目的は、予測値と真の値、つまりターゲット（別名ラベル）の差を測定することです。もう一つの引数、trgts を作ろう。0 か 1 の値で、画像が実際に 3 であるか否かを指示する。これもベクトル（ランク 1 テンソル）であり、画像にインデックスを付けます。

So, for instance, suppose we had three images which we knew were a 3, a 7, and a 3. And suppose our model predicted with high confidence (0.9) that the first was a 3, with slight confidence (0.4) that the second was a 7, and with fair confidence (0.2), but incorrectly, that the last was a 7. This would mean our loss function would receive these values as its inputs:

例えば、3、7、3 であることが分かっている 3 つの画像があったとします。 そして、モデルが、最初の画像は 3 であると高い信頼性（0.9）で予測し、2 番目の画像は 7 であるとわずかな信頼性（0.4）で予測し、最後の画像は 7 であるとかなりの信頼性（0.2）だが間違っているとします。この場合、損失関数はこれらの値を入力として受け取ることになります：

In [ ]:

```
trgts = tensor([1,0,1]) prds = tensor([0.9, 0.4, 0.2])
```

Here's a first try at a loss function that measures the distance between predictions and targets:

ここでは、予測とターゲットの間の距離を測定する損失関数の最初の試みを紹介します：

In [ ]:

```
def mnist_loss(predictions, targets):
return torch.where(targets==1, 1-predictions, predictions).mean()
```

We're using a new function, torch.where(a,b,c). This is the same as running the list comprehension [b[i] if a[i] else c[i] for i in range(len(a))], except it works on tensors, at C/CUDA speed. In plain English, this function will measure how distant each prediction is from 1 if it should be 1, and how distant it is from 0 if it should be 0, and then it will take the mean of all those distances.

新しい関数、torch.where(a,b,c)を使っています。これはリスト内包 [b[i] if a[i] else c[i] for i in range(len(a))] を実行するのと同じですが、テンソルに対して C/CUDA の速度で動作することが異なります。平たく言えば、この関数は、各予測が 1 であるべきなら 1 からどれだけ離れているか、0 であるべきなら 0 からどれだけ離れているかを測定し、それらの距離の平均を取ります。

> note: Read the Docs: It's important to learn about PyTorch functions like this, because looping over tensors in Python performs at Python speed, not C/CUDA speed! Try running `help(torch.where)` now to read the docs for this function, or, better still, look it up on the PyTorch documentation site.
>
> note: ドキュメントを読もう： Python のテンソルループは C/CUDA の速度ではなく、Python の速度で実行されるため、このような PyTorch 関数について学ぶことは重要です！ 今すぐ help(torch.where) を実行してこの関

数のドキュメントを読むか、もっといいのは PyTorch のドキュメントサイトで調べることです。

Let's try it on our prds and trgts:

それでは、prd と trgts で試してみましょう：

In [ ]:

```
torch.where(trgts==1, 1-prds, prds)
```

Out[ ]:

tensor([0.1000, 0.4000, 0.8000])

You can see that this function returns a lower number when predictions are more accurate, when accurate predictions are more confident (higher absolute values), and when inaccurate predictions are less confident. In PyTorch, we always assume that a lower value of a loss function is better. Since we need a scalar for the final loss, mnist_loss takes the mean of the previous tensor:

この関数は、予測がより正確な場合、正確な予測がより信頼できる（絶対値が高い）場合、不正確な予測がより信頼できない場合に、より低い数値を返すことがわかりますね。PyTorch では、常に損失関数の値が小さい方が良いと仮定しています。最終的な損失にはスカラーが必要なので、mnist_loss は前のテンソルの平均をとります：

In [ ]:

```
mnist_loss(prds,trgts)
```

Out[ ]:

tensor(0.4333)

For instance, if we change our prediction for the one "false" target from 0.2 to 0.8 the loss will go down, indicating that this is a better prediction:

例えば、1 つの「偽」のターゲットに対する予測を 0.2 から 0.8 に変更すると、損失は減少し、この方が良い予測であることを示す：

In [ ]:

```
mnist_loss(tensor([0.9, 0.4, 0.8]),trgts)
```

Out[ ]:

tensor(0.2333)

One problem with mnist_loss as currently defined is that it assumes that predictions are always between 0 and 1. We need to ensure, then, that this is actually the case! As it happens, there is a function that does exactly that—let's take a look.

現在定義されている mnist_loss の問題点の一つは、予測値が常に 0 と 1 の間であると仮定していることです！そこで、それを実現する関数があります。

Sigmoid

シグモイド

The sigmoid function always outputs a number between 0 and 1. It's defined as follows:

シグモイド関数は常に 0 から 1 の間の数を出力する関数で、次のように定義されています：

In [ ]:

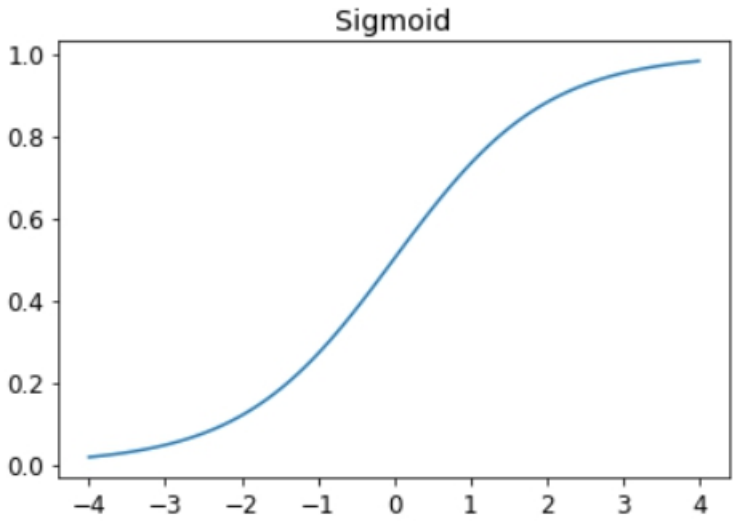
```
def sigmoid(x): return 1/(1+torch.exp(-x))
```

Pytorch defines an accelerated version for us, so we don't really need our own. This is an important function in deep learning, since we often want to ensure values are between 0 and 1. This is what it looks like:

Pytorch は高速化されたバージョンを定義してくれているので、私たち自身のものは必要ありません。これは深層学習において重要な関数で、値が 0 から 1 の間であることを確認したいことがよくあるからです：

In [ ]:

```
plot_function(torch.sigmoid, title='Sigmoid', min=-4, max=4)
```



As you can see, it takes any input value, positive or negative, and smooshes it onto an output value between 0 and 1. It's also a smooth curve that only goes up, which makes it easier for SGD to find meaningful gradients.

見てわかるように、これは正負を問わずどんな入力値でも、0 から 1 の間の出力値に合成します。また、上昇するだけの滑らかな曲線であるため、SGD は意味のある勾配を見つけやすくなっています。

Let's update mnist_loss to first apply sigmoid to the inputs:

mnist_loss を更新して、まず入力にシグモイドを適用してみましょう：

In [ ]:

```
def mnist_loss(predictions, targets):
    predictions = predictions.sigmoid()
    return torch.where(targets==1, 1-predictions, predictions).mean()
```

Now we can be confident our loss function will work, even if the predictions are not between 0 and 1. All that is required is that a higher prediction corresponds to higher confidence an image is a 3.

これで、予測値が０と１の間でなくても、損失関数が機能することを確信することができます。

Having defined a loss function, now is a good moment to recapitulate why we did this. After all, we already had a metric, which was overall accuracy. So why did we define a loss?

損失関数を定義したところで、なぜこのようなことをしたのか、もう一度整理してみましょう。というのも、私たちはすでに総合的な精度という指標を持っているからです。では、なぜ損失関数を定義したのでしょうか?

The key difference is that the metric is to drive human understanding and the loss is to drive automated learning. To drive automated learning, the loss must be a function that has a meaningful derivative. It can't have big flat sections and large jumps, but instead must be reasonably smooth. This is why we designed a loss function that would respond to small changes in confidence level. This requirement means that sometimes it does not really reflect exactly what we are trying to achieve, but is rather a compromise between our real goal and a function that can be optimized using its gradient. The loss function is calculated for each item in our dataset, and then at the end of an epoch the loss values are all averaged and the overall mean is reported for the epoch.

その大きな違いは、指標は人間の理解を促すものであり、損失は自動学習を促すものであるということです。自動学習を促進するためには、損失は意味のある微分を持つ関数でなければなりません。大きな平坦部や大きなジャンプがあるわけではなく、適度に滑らかでなければなりません。そのため、信頼度の小さな変化にも反応するような損失関数を設計しました。この要件は、時に私たちが達成しようとしていることを正確に反映していないことを意味しますが、むしろ私たちの真の目標とその勾配を使用して最適化できる関数との間の妥協であることを意味します。損失関数はデータセットの各項目について計算され、エポックの終了時に損失値がすべて平均化され、そのエポックの全体平均が報告される。

Metrics, on the other hand, are the numbers that we really care about. These are the values that are printed at the end of each epoch that tell us how our model is really doing. It is important that we learn to focus on these metrics, rather than the loss, when judging the performance of a model.

一方、メトリクスは、私たちが本当に気になる数値です。各エポックの終了時に表示される値で、モデルが実際にどのように機能しているのかを知ることができます。モデルの性能を判断する際には、損失ではなく、これらのメトリクスに注目することを学ぶことが重要です。

SGD and Mini-Batches

SGD とミニバッチ

Now that we have a loss function that is suitable for driving SGD, we can consider some of the details involved in the next phase of the learning process, which is to change or update the weights based on the gradients. This is called an *optimization step*.

SGD の駆動に適した損失関数ができたので、学習プロセスの次の段階、つまり勾配に基づいて重みを変更または更新することに関係するいくつかの詳細を検討することができます。これは最適化ステップと呼ばれます。

In order to take an optimization step we need to calculate the loss over one or more data items. How many should we use? We could calculate it for the whole dataset, and take the average, or we could calculate it for a single data item. But neither of these is ideal. Calculating it for the whole dataset would take a very long time. Calculating it for a single item would not use much information, so it would result in a very imprecise and unstable gradient. That is, you'd be going to the trouble of updating the weights, but taking into account only how that would improve the model's performance on that single item.

最適化ステップを行うには、1 つ以上のデータ項目に対する損失を計算する必要があります。何個使えばいいのでしょうか? データセット全体で計算して平均を取ることもできるし、1 つのデータ項目で計算することもできる。しかし、どちらも理想的ではありません。データセット全体について計算すると、非常に長い時間がかかる。1 つの項目に対して計算すると、あまり情報を使わないので、非常に不正確で不安定な勾配になります。つまり、せっかく重みを更新しても、それがその 1 つの項目に対するモデルのパフォーマンスをどのように向上させるかだけを考慮することになります。

So instead we take a compromise between the two: we calculate the average loss for a few data items at a time. This is called a *mini-batch*. The number of data items in the mini-batch is called the *batch size*. A larger batch size means that you will get a more accurate and stable estimate of your dataset's gradients from the loss function, but it will take longer, and you will process fewer mini-batches per epoch. Choosing a good batch size is one of the decisions you need to make as a deep learning practitioner to train your model quickly and accurately. We will talk about how to make this choice throughout this book.

そこで、両者の妥協点として、一度にいくつかのデータ項目の平均損失を計算することにします。これはミニバッチと呼ばれるものです。ミニバッチに含まれるデータアイテムの数をバッチサイズと呼びます。バッチサイズが大きいと、損失関数からデータセットの勾配をより正確に安定して推定することができますが、時間がかかり、エポックあたりのミニバッチの処理数は少なくなります。良いバッチサイズを選択することは、モデルを迅速かつ正確に訓練するために、ディープラーニングの実践者として行うべき決定の 1 つです。この選択方法については、本書を通じて説明します。

Another good reason for using mini-batches rather than calculating the gradient on individual data items is that, in practice, we nearly always do our training on an accelerator such as a GPU. These accelerators only perform well if they have lots of work to do at a time, so it's helpful if we can give them lots of data items to work on. Using

mini-batches is one of the best ways to do this. However, if you give them too much data to work on at once, they run out of memory—making GPUs happy is also tricky!

個々のデータ項目で勾配を計算するのではなく、ミニバッチを使用するもう一つの良い理由は、実際には、ほとんど常に GPU などのアクセラレータでトレーニングを行うことです。このようなアクセラレータは、一度にたくさんの仕事をこなさないとうまく機能しないので、たくさんのデータ項目を与えることができれば便利です。ミニバッチの使用は、そのための最良の方法の 1 つです。しかし、一度にたくさんのデータを与えると、メモリ不足になりますし、GPU を喜ばせるのも難しい！

As we saw in our discussion of data augmentation in <<chapter_production>>, we get better generalization if we can vary things during training. One simple and effective thing we can vary is what data items we put in each mini-batch. Rather than simply enumerating our dataset in order for every epoch, instead what we normally do is randomly shuffle it on every epoch, before we create mini-batches. PyTorch and fastai provide a class that will do the shuffling and mini-batch collation for you, called `DataLoader`.

<>のデータ増強の説明で見たように、トレーニング中に様々な変化を与えることができれば、より良い汎化を実現することができます。単純かつ効果的な変化として、各ミニバッチに入れるデータ項目があります。エポックごとにデータセットを順番に並べるのではなく、ミニバッチを作成する前に、エポックごとにランダムにシャッフルするのが普通です。PyTorch と fastai は DataLoader と呼ばれるシャッフルやミニバッチの照合を行うクラスを提供しています。

A `DataLoader` can take any Python collection and turn it into an iterator over mini-batches, like so:
DataLoader は、Python のコレクションを受け取り、ミニバッチに対するイテレータにすることができます（以下のように）：

```
coll = range(15)
dl = DataLoader(coll, batch_size=5, shuffle=True)
list(dl)
```

Out[ ]:

```
[tensor([ 3, 12,  8, 10,  2]),
 tensor([ 9,  4,  7, 14,  5]),
 tensor([ 1, 13,  0,  6, 11])]
```

For training a model, we don't just want any Python collection, but a collection containing independent and dependent variables (that is, the inputs and targets of the model). A collection that contains tuples of independent and dependent variables is known in PyTorch as a `Dataset`. Here's an example of an extremely simple `Dataset`:

モデルの学習には、Python のコレクションであれば何でも良いわけではなく、独立変数と従属変数（つまり、モデルの入力とターゲット）を含むコレクションが必要です。独立変数と従属変数のタプルを含むコレクションは、PyTorch ではデータセットと呼ばれていま

す。以下は、極めて単純なデータセットの例です：

In [ ]:

```
ds = L(enumerate(string.ascii_lowercase))
ds
```

Out[ ]:

```
(#26) [(0, 'a'),(1, 'b'),(2, 'c'),(3, 'd'),(4, 'e'),(5, 'f'),(6, 'g'),(7, 'h'),(8, 'i'),(9, 'j
')...]
```

When we pass a Dataset to a DataLoader we will get back mini-batches which are themselves tuples of tensors representing batches of independent and dependent variables:

データセットを DataLoader に渡すと、それ自体が独立変数と従属変数のバッチを表すテンソルのタプルであるミニバッチが返ってきます：

In [ ]:

```
dl = DataLoader(ds, batch_size=6, shuffle=True)
list(dl)
```

Out[ ]:

```
[(tensor([17, 18, 10, 22,  8, 14]), ('r', 's', 'k', 'w', 'i', 'o')),
 (tensor([20, 15,  9, 13, 21, 12]), ('u', 'p', 'j', 'n', 'v', 'm')),
 (tensor([ 7, 25,  6,  5, 11, 23]), ('h', 'z', 'g', 'f', 'l', 'x')),
 (tensor([ 1,  3,  0, 24, 19, 16]), ('b', 'd', 'a', 'y', 't', 'q')),
 (tensor([2, 4]), ('c', 'e'))]
```

We are now ready to write our first training loop for a model using SGD!

これで、SGD を使ったモデルの最初のトレーニングループを書く準備ができました！

Putting It All Together

すべてをまとめる

It's time to implement the process we saw in <<gradient_descent>>. In code, our process will be implemented something like this for each epoch:

gradient_descent "で見た処理を実装する時が来たようです。コード上では、各エポックに対して次のような処理を行うことになる：

```
for x,y in dl:
    pred = model(x)
    loss = loss_func(pred, y)
```

loss.backward()

parameters -= parameters.grad * lr

First, let's re-initialize our parameters:

まず、パラメータを再初期化しましょう：

In [ ]:

```
weights = init_params((28*28,1))
bias = init_params(1)
```

A DataLoader can be created from a Dataset:

DataLoader は、Dataset から作成することができます：

In [ ]:

```
dl = DataLoader(dset, batch_size=256)
xb,yb = first(dl)
xb.shape,yb.shape
```

Out[ ]:

```
(torch.Size([256, 784]), torch.Size([256, 1]))
```

We'll do the same for the validation set:

バリデーションセットも同じようにします：

In [ ]:

```
valid_dl = DataLoader(valid_dset, batch_size=256)
```

Let's create a mini-batch of size 4 for testing:

テスト用にサイズ 4 のミニバッチを作成しよう：

In [ ]:

```
batch = train_x[:4] batch.shape
```

Out[ ]:

```
torch.Size([4, 784])
```

In [ ]:

```
preds = linear1(batch)
preds
```

Out[ ]:

```
tensor([[-11.1002],
        [  5.9263],
        [  9.9627],
        [ -8.1484]], grad_fn=<AddBackward0>)
```

In [ ]:

```
loss = mnist_loss(preds, train_y[:4])
loss
```

Out[ ]:

```
tensor(0.5006, grad_fn=<MeanBackward0>)
```

Now we can calculate the gradients:

これで勾配を計算できるようになりました：

In [ ]:

```
loss.backward() weights.grad.shape,weights.grad.mean(),bias.grad
```

Out[ ]:

```
(torch.Size([784, 1]), tensor(-0.0001), tensor([-0.0008]))
```

Let's put that all in a function:

それをすべて関数にまとめてみましょう：

In [ ]:

```
def calc_grad(xb, yb, model):
    preds = model(xb)
    loss = mnist_loss(preds, yb)
    loss.backward()
```

and test it:

を作成し、それをテストする：

In [ ]:

```
calc_grad(batch, train_y[:4], linear1) weights.grad.mean(),bias.grad
```

Out[ ]:

```
(tensor(-0.0002), tensor([-0.0015]))
```

But look what happens if we call it twice:

しかし、これを 2 回呼び出すとどうなるか見てみましょう：

In [ ]:

```
calc_grad(batch, train_y[:4], linear1) weights.grad.mean(),bias.grad
```

Out[ ]:

```
(tensor(-0.0003), tensor([-0.0023]))
```

The gradients have changed! The reason for this is that loss.backward actually *adds* the gradients of loss to any gradients that are currently stored. So, we have to set the current gradients to 0 first:

グラデーションが変更されました！この理由は、loss.backward が実際に loss のグラデーションを現在保存されているグラデーションに追加しているからです。そこで、まず現在の勾配を 0 にする必要があります：

In [ ]:

```
weights.grad.zero_() bias.grad.zero_();
```

> note: Inplace Operations: Methods in PyTorch whose names end in an underscore modify their objects *in place*. For instance, bias.zero_() sets all elements of the tensor bias to 0.
>
> note: インプレイス操作： PyTorch のメソッドで、名前がアンダースコアで終わるものは、その場でオブジェクトを変更します。例えば、bias.zero_() はテンソル bias のすべての要素を 0 にします。

Our only remaining step is to update the weights and biases based on the gradient and learning rate. When we do so, we have to tell PyTorch not to take the gradient of this step too—otherwise things will get very confusing when we try to compute the derivative at the next batch! If we assign to the data attribute of a tensor then PyTorch will not take the gradient of that step. Here's our basic training loop for an epoch:

残りのステップは、勾配と学習率に基づいて重みとバイアスを更新することだけです。そうしないと、次のバッチで導関数を計算しようとしたときに非常に混乱することになります！もしテンソルのデータ属性に代入すると、PyTorch はそのステップの勾配を取らないようにします。以下は、1 つのエポックに対する基本的な学習ループです：

In [ ]:

```
def train_epoch(model, lr, params):
    for xb,yb in dl:
        calc_grad(xb, yb, model)
```

```
    for p in params:
        p.data -= p.grad*lr
        p.grad.zero_()
```

We also want to check how we're doing, by looking at the accuracy of the validation set. To decide if an output represents a 3 or a 7, we can just check whether it's greater than 0. So our accuracy for each item can be calculated (using broadcasting, so no loops!) with:

また、検証セットの精度を見ることで、自分たちがどのようなことをしているかをチェックしたい。出力が 3 か 7 かを判断するには、出力が 0 より大きいかどうかをチェックすればよい：

In [ ]:

```
(preds>0.0).float() == train_y[:4]
```

Out[ ]:

```
tensor([[False],
        [ True],
        [ True],
        [False]])
```

That gives us this function to calculate our validation accuracy:

これで、検証精度を計算するための関数ができました：

In [ ]:

```
def batch_accuracy(xb, yb):
    preds = xb.sigmoid()
    correct = (preds>0.5) == yb
    return correct.float().mean()
```

We can check it works:

動作確認ができる：

In [ ]:

```
batch_accuracy(linear1(batch), train_y[:4])
```

Out[ ]:

```
tensor(0.5000)
```

and then put the batches together:

を行い、バッチをまとめる：

In [ ]:

```
def validate_epoch(model):
    accs = [batch_accuracy(model(xb), yb) for xb,yb in valid_dl]
    return round(torch.stack(accs).mean().item(), 4)
```

In [ ]:

```
validate_epoch(linear1)
```

Out[ ]:

0.5219

That's our starting point. Let's train for one epoch, and see if the accuracy improves:

これが出発点です。1 エポック分訓練して、精度が上がるかどうか見てみましょう：

In [ ]:

```
lr = 1. params = weights,bias train_epoch(linear1, lr, params) validate_epoch(linear1)
```

Out[ ]:

0.6883

Then do a few more:

In [ ]:

```
for i in range(20):
    train_epoch(linear1, lr, params)
    print(validate_epoch(linear1), end=' ')
```

0.8314 0.9017 0.9227 0.9349 0.9438 0.9501 0.9535 0.9564 0.9594 0.9618 0.9613 0.9638 0.9643 0.9652 0.9662 0.9677 0.9687 0.9691 0.9691 0.9696

Looking good! We're already about at the same accuracy as our "pixel similarity" approach, and we've created a general-purpose foundation we can build on. Our next step will be to create an object that will handle the SGD step for us. In PyTorch, it's called an *optimizer*.

いい感じですね！すでに「ピクセルの類似性」アプローチとほぼ同じ精度で、汎用的な基盤ができました。次のステップは、SGD のステップを処理するオブジェクトを作成することです。PyTorch では、これをオプティマイザーと呼びます。

Creating an Optimizer

オプティマイザーを作成する

Because this is such a general foundation, PyTorch provides some useful classes to make it easier to implement. The first thing we can do is replace our `linear1` function with PyTorch's `nn.Linear` module. A *module* is an object

of a class that inherits from the PyTorch nn.Module class. Objects of this class behave identically to standard Python functions, in that you can call them using parentheses and they will return the activations of a model.

このように一般的な基礎であるため、PyTorch はこれを簡単に実装するための便利なクラスをいくつか提供しています。最初にできることは、linear1 関数を PyTorch の nn.Linear モジュールに置き換えることです。モジュールとは、PyTorch の nn.Module クラスを継承したクラスのオブジェクトのことです。このクラスのオブジェクトは標準的な Python 関数と同じように動作し、括弧を使って呼び出すと、モデルの活性度を返します。

nn.Linear does the same thing as our init_params and linear together. It contains both the *weights* and *biases* in a single class. Here's how we replicate our model from the previous section:

nn.Linear は、init_params と linear を一緒にしたものと同じ働きをします。これは、重みとバイアスの両方を 1 つのクラスに含んでいます。以下は、前節のモデルを再現する方法です：

In [ ]:

```
linear_model = nn.Linear(28*28,1)
```

Every PyTorch module knows what parameters it has that can be trained; they are available through the parameters method:

PyTorch の各モジュールは、学習可能なパラメータを知っています：

In [ ]:

```
w,b = linear_model.parameters() w.shape,b.shape
```

Out[ ]:

```
(torch.Size([1, 784]), torch.Size([1]))
```

We can use this information to create an optimizer:

この情報を使ってオプティマイザを作ることができる：

In [ ]:

```
class BasicOptim:
    def __init__(self,params,lr): self.params,self.lr = list(params),lr
    def step(self, *args, **kwargs):
        for p in self.params: p.data -= p.grad.data * self.lr
    def zero_grad(self, *args, **kwargs):
        for p in self.params: p.grad = None
```

We can create our optimizer by passing in the model's parameters:

トレーニングループは次のように簡略化できます：

In [ ]:

```
opt = BasicOptim(linear_model.parameters(), lr)
```

Our training loop can now be simplified to:

トレーニングループは次のように簡略化できます：

In [ ]:

```
def train_epoch(model): for xb,yb in dl: calc_grad(xb, yb, model) opt.step() opt.zero_grad()
```

Our validation function doesn't need to change at all:

検証関数は全く変更する必要がない：

In [ ]:

```
validate_epoch(linear_model)
```

Out[ ]:

```
0.4157
```

Let's put our little training loop in a function, to make things simpler:

この小さな学習ループを、よりシンプルにするために、関数の中に入れてみましょう：

In [ ]:

```
def train_model(model, epochs):
    for i in range(epochs):
        train_epoch(model)
        print(validate_epoch(model), end=' ')
```

The results are the same as in the previous section:

という結果になる：

In [ ]:

```
train_model(linear_model, 20)
```

```
0.4932 0.8618 0.8203 0.9102 0.9331 0.9468 0.9555 0.9629 0.9658 0.9673
0.9687 0.9707 0.9726 0.9751 0.9761 0.9761 0.9775 0.978 0.9785 0.9785
```

fastai provides the SGD class which, by default, does the same thing as our BasicOptim:

fastai は SGD クラスを提供しており、デフォルトでは私たちの BasicOptim と同じことをする：

In [ ]:

```
linear_model = nn.Linear(28*28,1) opt = SGD(linear_model.parameters(), lr) train_model(linear_model, 20)
```

0.4932 0.852 0.8335 0.9116 0.9326 0.9473 0.9555 0.9624 0.9648 0.9668
0.9692 0.9712 0.9731 0.9746 0.9761 0.9765 0.9775 0.978 0.9785 0.9785

fastai also provides Learner.fit, which we can use instead of train_model. To create a Learner we first need to create a DataLoaders, by passing in our training and validation DataLoaders:

fastai は Learner.fit も提供しており、train_model の代わりに使用することができます。Learner を作成するには、まず DataLoaders を作成する必要があり、トレーニングと検証用の DataLoaders を渡します：

In [ ]:

```
dls = DataLoaders(dl, valid_dl)
```

To create a Learner without using an application (such as vision_learner) we need to pass in all the elements that we've created in this chapter: the DataLoaders, the model, the optimization function (which will be passed the parameters), the loss function, and optionally any metrics to print:

アプリケーション（vision_learner など）を使わずに Learner を作成するには、この章で作成したすべての要素、つまり DataLoaders、モデル、最適化関数（パラメータを渡す）、損失関数、そしてオプションとして出力するメトリクスを渡す必要があります：

In [ ]:

```
learn = Learner(dls, nn.Linear(28*28,1), opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)
```

Now we can call fit:

これで fit を呼び出すことができます：

In [ ]:

```
learn.fit(10, lr=lr)
```

epoch	train_loss	valid_loss	batch_accuracy	time
0	0.636857	0.503549	0.495584	00:00
1	0.545725	0.170281	0.866045	00:00
2	0.199223	0.184893	0.831207	00:00
3	0.086580	0.107836	0.911187	00:00
4	0.045185	0.078481	0.932777	00:00
5	0.029108	0.062792	0.946516	00:00
6	0.022560	0.053017	0.955348	00:00
7	0.019687	0.046500	0.962218	00:00
8	0.018252	0.041929	0.965162	00:00
9	0.017402	0.038573	0.967615	00:00

As you can see, there's nothing magic about the PyTorch and fastai classes. They are just convenient pre-packaged pieces that make your life a bit easier! (They also provide a lot of extra functionality we'll be using in future chapters.)

ご覧のように、PyTorch と fastai のクラスには何の魔法もありません。これらは、あなたの生活を少し楽にするために、便利なパッケージ化された部品に過ぎないのです！(また、今後の章で使用する多くの追加機能も提供されています)。

With these classes, we can now replace our linear model with a neural network.

これらのクラスを使って、線形モデルをニューラルネットワークに置き換えることができるようになりました。

Adding a Nonlinearity

非線形性を追加する

So far we have a general procedure for optimizing the parameters of a function, and we have tried it out on a very boring function: a simple linear classifier. A linear classifier is very constrained in terms of what it can do. To make it a bit more complex (and able to handle more tasks), we need to add something nonlinear between two linear classifiers—this is what gives us a neural network.

ここまでで、関数のパラメータを最適化する一般的な手順がわかりましたが、今回は非常につまらない関数である単純な線形分類器で試してみました。線形分類器は、できること

が非常に限定されています。これをもう少し複雑にする（より多くのタスクを処理できるようにする）には、2 つの線形分類器の間に非線形なものを加える必要があるのです。

Here is the entire definition of a basic neural network:

以下に、基本的なニューラルネットワークの定義全体を示します：

In [ ]:

```
def simple_net(xb):
    res = xb@w1 + b1 res = res.max(tensor(0.0))
    res = res@w2 + b2
    return res
```

That's it! All we have in `simple_net` is two linear classifiers with a `max` function between them.

これで終わりです！simple_net にあるのは、2 つの線形分類器とその間の max 関数だけです。

Here, `w1` and `w2` are weight tensors, and `b1` and `b2` are bias tensors; that is, parameters that are initially randomly initialized, just like we did in the previous section:

ここで、w1 と w2 は重みテンソル、b1 と b2 はバイアステンソルです。つまり、前のセクションでやったように、最初はランダムに初期化されるパラメータです：

In [ ]:

```
w1 = init_params((28*28,30))
b1 = init_params(30)
w2 = init_params((30,1))
b2 = init_params(1)
```

The key point about this is that `w1` has 30 output activations (which means that `w2` must have 30 input activations, so they match). That means that the first layer can construct 30 different features, each representing some different mix of pixels. You can change that `30` to anything you like, to make the model more or less complex.

ここで重要なのは、w1 の出力活性度が 30 であることです（つまり、w2 の入力活性度も 30 でなければならないので、両者は一致します）。つまり、第 1 層は 30 種類の特徴を構築することができ、それぞれが異なるピクセルの組み合わせを表現している。この 30 を好きなように変えて、モデルをより複雑にしたり、より少なくしたりすることができます。

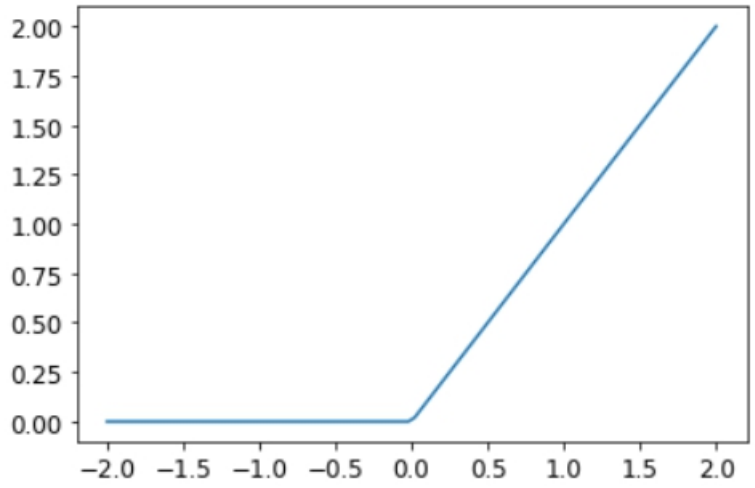
That little function `res.max(tensor(0.0))` is called a *rectified linear unit*, also known as *ReLU*. We think we can all agree that *rectified linear unit* sounds pretty fancy and complicated... But actually, there's nothing more to it than `res.max(tensor(0.0))`—in other words, replace every negative number with a zero. This tiny function is also available in PyTorch as `F.relu`:

この小さな関数 res.max(tensor(0.0)) は、rectified linear unit、別名 ReLU と呼ばれています。整流線形ユニットというと、かなり派手で複雑な響きがあるのは、誰もが認めると

ころでしょう。しかし実際には、res.max(tensor(0.0))、つまり負の数をすべて 0 に置き換えるだけなのです。この小さな関数は、PyTorch でも F.relu として利用できます：

In [ ]:

```
plot_function(F.relu)
```



J: There is an enormous amount of jargon in deep learning, including terms like *rectified linear unit*. The vast vast majority of this jargon is no more complicated than can be implemented in a short line of code, as we saw in this example. The reality is that for academics to get their papers published they need to make them sound as impressive and sophisticated as possible. One of the ways that they do that is to introduce jargon. Unfortunately, this has the result that the field ends up becoming far more intimidating and difficult to get into than it should be. You do have to learn the jargon, because otherwise papers and tutorials are not going to mean much to you. But that doesn't mean you have to find the jargon intimidating. Just remember, when you come across a word or phrase that you haven't seen before, it will almost certainly turn out to be referring to a very simple concept.

J：ディープラーニングには、整流線形ユニットなどの用語を含め、膨大な量の専門用語が存在します。この専門用語の大部分は、この例で見たように、短いコード行で実装できるほど複雑なものではありません。現実には、学者が論文を出版するためには、できるだけ印象的で洗練された響きにする必要があります。その方法のひとつが、専門用語の導入です。しかし、残念ながら、その結果、その分野は必要以上に威圧的で、入り込むのが難しくなってしまうのです。専門用語を学ばなければ、論文やチュートリアルはあまり意

味をなさないからです。しかし、だからといって、専門用語に怯える必要はありません。見たことのない言葉やフレーズを目にしたとき、それはほとんど間違いなく、非常にシンプルなコンセプトを指していることがわかるからです。

The basic idea is that by using more linear layers, we can have our model do more computation, and therefore model more complex functions. But there's no point just putting one linear layer directly after another one, because when we multiply things together and then add them up multiple times, that could be replaced by multiplying different things together and adding them up just once! That is to say, a series of any number of linear layers in a row can be replaced with a single linear layer with a different set of parameters.

基本的な考え方は、線形レイヤーを増やすことで、モデルにより多くの計算をさせることができ、より複雑な関数をモデル化することができるというものです。というのも、あるものを掛け合わせて何度も足し算をすると、別のものを掛け合わせて１度だけ足し算をすることに置き換わってしまうからです！ つまり、いくつもの線形レイヤーを並べたものは、異なるパラメータのセットを持つ１つの線形レイヤーに置き換えることができる。

But if we put a nonlinear function between them, such as `max`, then this is no longer true. Now each linear layer is actually somewhat decoupled from the other ones, and can do its own useful work. The `max` function is particularly interesting, because it operates as a simple `if` statement.

しかし、その間に max のような非線形関数を置くと、これはもはや真実ではありません。これで、それぞれの線形層は、実際には他の線形層から多少切り離され、独自の有用な仕事をすることができる。max 関数は、単純な if 文として動作するので、特に興味深い。

S: Mathematically, we say the composition of two linear functions is another linear function. So, we can stack as many linear classifiers as we want on top of each other, and without nonlinear functions between them, it will just be the same as one linear classifier.

S: 数学的には、２つの線形関数の合成は別の線形関数であると言います。つまり、線形分類器をいくらでも積み重ねることができ、その間に非線形関数がなければ、１つの線形分類器と同じになるんだ。

Amazingly enough, it can be mathematically proven that this little function can solve any computable problem to an arbitrarily high level of accuracy, if you can find the right parameters for `w1` and `w2` and if you make these matrices big enough. For any arbitrarily wiggly function, we can approximate it as a bunch of lines joined together; to make it closer to the wiggly function, we just have to use shorter lines. This is known as the *universal approximation theorem*. The three lines of code that we have here are known as *layers*. The first and third are known as *linear layers*, and the second line of code is known variously as a *nonlinearity*, or *activation function*.

驚くべきことに、この小さな関数が、w1 と w2 のパラメータを適切に設定し、これらの行列を十分に大きくすれば、あらゆる計算可能な問題を任意の高い精度で解くことができ

ることが数学的に証明される。任意のくねくねした関数に対しては、それを何本もの線をつないだものとして近似することができます。くねくねした関数に近づけるには、より短い線を使えばよいのです。これは万能近似定理と呼ばれるものです。ここにある 3 行のコードはレイヤーと呼ばれるものです。1 番目と 3 番目は線形層と呼ばれ、2 番目の行は非線形性、または活性化関数として様々に知られているコードです。

Just like in the previous section, we can replace this code with something a bit simpler, by taking advantage of PyTorch:

前節と同じように、PyTorch を活用することで、このコードをもう少しシンプルなものに置き換えることができます：

In [ ]:

```
simple_net = nn.Sequential( nn.Linear(28*28,30), nn.ReLU(), nn.Linear(30,1) )
```

nn.Sequential creates a module that will call each of the listed layers or functions in turn.

nn.Sequential は、リストアップされた各レイヤーまたは関数を順番に呼び出すモジュールを作成します。

nn.ReLU is a PyTorch module that does exactly the same thing as the F.relu function. Most functions that can appear in a model also have identical forms that are modules. Generally, it's just a case of replacing F with nn and changing the capitalization. When using nn.Sequential, PyTorch requires us to use the module version. Since modules are classes, we have to instantiate them, which is why you see nn.ReLU() in this example.

nn.ReLU は F.relu 関数と全く同じことをする PyTorch のモジュールです。モデルに登場することができるほとんどの関数は、モジュールである同一のフォームも持っています。一般的には、F を nn に置き換えて、大文字を変えるだけです。nn.Sequential を使う場合、PyTorch はモジュール版を使うことを要求します。モジュールはクラスなので、インスタンス化する必要があり、この例で nn.ReLU() が出てくるのはそのためです。

Because nn.Sequential is a module, we can get its parameters, which will return a list of all the parameters of all the modules it contains. Let's try it out! As this is a deeper model, we'll use a lower learning rate and a few more epochs.

nn.Sequential はモジュールなので、そのパラメータを取得することができ、それが含むすべてのモジュールのパラメータのリストを返します。試しにやってみましょう！これはより深いモデルなので、学習率を低くし、エポックを数回多くします。

In [ ]:

```
learn = Learner(dls, simple_net, opt_func=SGD,
                loss_func=mnist_loss, metrics=batch_accuracy)
```

In [ ]:

```
#hide_output
learn.fit(40, 0.1)
```

epoch	train_loss	valid_loss	batch_accuracy	time
0	0.305828	0.399663	0.508341	00:00
1	0.142960	0.225702	0.807655	00:00
2	0.079516	0.113519	0.919529	00:00
3	0.052391	0.076792	0.943081	00:00
4	0.039796	0.060083	0.956330	00:00
5	0.033368	0.050713	0.963690	00:00
6	0.029680	0.044797	0.965653	00:00
7	0.027290	0.040729	0.968106	00:00
8	0.025568	0.037771	0.968597	00:00
9	0.024233	0.035508	0.970559	00:00
10	0.023149	0.033714	0.972031	00:00
11	0.022242	0.032243	0.972522	00:00
12	0.021468	0.031006	0.973503	00:00
13	0.020796	0.029944	0.974485	00:00
14	0.020207	0.029016	0.975466	00:00
15	0.019683	0.028196	0.976448	00:00
16	0.019215	0.027463	0.976448	00:00
17	0.018791	0.026806	0.976938	00:00
18	0.018405	0.026212	0.977920	00:00
19	0.018051	0.025671	0.977920	00:00
20	0.017725	0.025179	0.977920	00:00
21	0.017422	0.024728	0.978410	00:00
22	0.017141	0.024313	0.978901	00:00
23	0.016878	0.023932	0.979392	00:00
24	0.016632	0.023580	0.979882	00:00
25	0.016400	0.023254	0.979882	00:00
26	0.016181	0.022952	0.979882	00:00
27	0.015975	0.022672	0.980864	00:00
28	0.015779	0.022411	0.980864	00:00
29	0.015593	0.022168	0.981845	00:00
30	0.015417	0.021941	0.981845	00:00
31	0.015249	0.021728	0.981845	00:00
32	0.015088	0.021529	0.981845	00:00
33	0.014935	0.021341	0.981845	00:00
34	0.014788	0.021164	0.981845	00:00
35	0.014647	0.020998	0.982336	00:00
36	0.014512	0.020840	0.982826	00:00
37	0.014382	0.020691	0.982826	00:00
38	0.014257	0.020550	0.982826	00:00
39	0.014136	0.020415	0.982826	00:00

We're not showing the 40 lines of output here to save room; the training process is recorded in learn.recorder, with the table of output stored in the values attribute, so we can plot the accuracy over training as:

学習過程は learn.recorder に記録され、values 属性に出力表が格納されているので、学習に対する精度をプロットすることができます：

In [ ]:

```
plt.plot(L(learn.recorder.values).itemgot(2));
```

And we can view the final accuracy:

そして、最終的な精度を見ることができます：

In [ ]:

```
learn.recorder.values[-1][2]
```

Out[ ]:

0.982826292514801

At this point we have something that is rather magical:

この時点で私たちは、むしろ魔法のようなものを手に入れたのです：

1. A function that can solve any problem to any level of accuracy (the neural network) given the correct set of parameters
   正しいパラメータのセットがあれば、どんな問題でもどんな精度でも解決できる関数（ニューラルネットワーク）です。
2. A way to find the best set of parameters for any function (stochastic gradient descent)
   任意の関数に対して最適なパラメータセットを見つける方法（確率的勾配降下法）。

This is why deep learning can do things which seem rather magical, such fantastic things. Believing that this combination of simple techniques can really solve any problem is one of the biggest steps that we find many students have to take. It seems too good to be true—surely things should be more difficult and complicated than this? Our recommendation: try it out! We just tried it on the MNIST dataset and you have seen the results. And since we are doing everything from scratch ourselves (except for calculating the gradients) you know that there is no special magic hiding behind the scenes.

そのため、ディープラーニングは、一見魔法のように見える、素晴らしいことをすることができるのです。このシンプルな技術の組み合わせが、どんな問題でも本当に解決できると信じることは、多くの学生にとって最大のステップの１つです。物事はもっと難しくて複雑なはずなのに……。私たちのお勧めは、「試してみる」ことです！ MNIST データセットで試してみて、その結果をご覧ください。そして、私たちは（グラデーションの計算を除いて）すべて自分たちでゼロからやっているので、舞台裏に特別な魔法が隠れているわけではないことがお分かりいただけると思います。

Going Deeper

さらに深く

There is no need to stop at just two linear layers. We can add as many as we want, as long as we add a nonlinearity between each pair of linear layers. As you will learn, however, the deeper the model gets, the harder it is to optimize the parameters in practice. Later in this book you will learn about some simple but brilliantly effective techniques for training deeper models.

2 つのリニアレイヤーだけで止める必要はありません。各直線層の間に非直線性を追加すれば、いくらでも追加できます。しかし、これから学ぶように、モデルが深くなればなるほど、実際にパラメータを最適化するのは難しくなります。本書の後半では、より深いモデルをトレーニングするための、シンプルだが非常に効果的なテクニックについて学ぶことができる。

We already know that a single nonlinearity with two linear layers is enough to approximate any function. So why would we use deeper models? The reason is performance. With a deeper model (that is, one with more layers) we do

not need to use as many parameters; it turns out that we can use smaller matrices with more layers, and get better results than we would get with larger matrices, and few layers.

私たちはすでに、2 つの線形層を持つ単一の非線形性で、あらゆる関数を近似するのに十分であることを知っています。では、なぜより深いモデルを使うのでしょうか? その理由は、性能です。より深いモデル（つまり、より多くの層を持つモデル）では、それほど多くのパラメータを使用する必要はありません。より大きな行列と少ない層で得られる結果よりも、より小さな行列と多くの層を使用し、より良い結果を得ることができることが判明しています。

That means that we can train the model more quickly, and it will take up less memory. In the 1990s researchers were so focused on the universal approximation theorem that very few were experimenting with more than one nonlinearity. This theoretical but not practical foundation held back the field for years. Some researchers, however, did experiment with deep models, and eventually were able to show that these models could perform much better in practice. Eventually, theoretical results were developed which showed why this happens. Today, it is extremely unusual to find anybody using a neural network with just one nonlinearity.

つまり、より早くモデルを学習させることができ、メモリ消費量も少なくて済むのです。1990 年代、研究者は普遍的な近似定理に集中し、複数の非線形性を実験している人はほとんどいませんでした。このような理論的な基礎はあっても実用的な基礎がないため、この分野は何年も足踏み状態でした。しかし、一部の研究者はディープモデルを使った実験を行い、最終的には、これらのモデルが実用上はるかに優れた性能を発揮できることを示すことができました。やがて、なぜこのようなことが起こるのかを示す理論的な結果も出てきました。現在では、非線形性が 1 つしかないニューラルネットワークを使う人は、非常に珍しい存在になっています。

Here is what happens when we train an 18-layer model using the same approach we saw in <<chapter_intro>>:

以下は、<>で見たのと同じアプローチで 18 層のモデルを学習させた場合の結果です：

In [ ]:

```
dls = ImageDataLoaders.from_folder(path)
learn = vision_learner(dls, resnet18, pretrained=False,
    loss_func=F.cross_entropy, metrics=accuracy)
learn.fit_one_cycle(1, 0.1)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.082089	0.009578	0.997056	00:11

Nearly 100% accuracy! That's a big difference compared to our simple neural net. But as you'll learn in the remainder of this book, there are just a few little tricks you need to use to get such great results from scratch

yourself. You already know the key foundational pieces. (Of course, even once you know all the tricks, you'll nearly always want to work with the pre-built classes provided by PyTorch and fastai, because they save you having to think about all the little details yourself.)

ほぼ 100%の精度を実現！これは、私たちのシンプルなニューラルネットと比べると、大きな違いです。しかし、本書の残りの部分で学ぶように、あなた自身がゼロからこのような素晴らしい結果を得るために必要な小さなコツはほんの少しです。あなたはすでに、基礎となる重要な部分を知っています。(もちろん、すべてのコツを知っても、PyTorch や fastai が提供するビルド済みのクラスで作業したいと思うことがほとんどでしょう。）

Jargon Recap

専門用語のまとめ

Congratulations: you now know how to create and train a deep neural network from scratch! We've gone through quite a few steps to get to this point, but you might be surprised at how simple it really is.

おめでとうございます。これで、ディープニューラルネットワークをゼロから作成し、訓練する方法がわかりました！ここまで来るのにかなりのステップを踏んできましたが、本当に簡単なことに驚かれるかもしれません。

Now that we are at this point, it is a good opportunity to define, and review, some jargon and key concepts.

ここまで来たからには、専門用語や重要な概念を定義し、復習する良い機会だと思います。

A neural network contains a lot of numbers, but they are only of two types: numbers that are calculated, and the parameters that these numbers are calculated from. This gives us the two most important pieces of jargon to learn:

ニューラルネットワークにはたくさんの数値が含まれていますが、それらは 2 種類しかありません。計算される数値と、その数値が計算されるパラメータです。つまり、計算される数値と、その数値を計算するパラメータの 2 つです：

- Activations:: Numbers that are calculated (both by linear and nonlinear layers)
  アクティベーション：： 計算される数値（線形層と非線形層の両方で）。
- Parameters:: Numbers that are randomly initialized, and optimized (that is, the numbers that define the model)
  パラメータ：： ランダムに初期化され、最適化される数値（つまり、モデルを定義する数値）。

We will often talk in this book about activations and parameters. Remember that they have very specific meanings. They are numbers. They are not abstract concepts, but they are actual specific numbers that are in your model. Part of becoming a good deep learning practitioner is getting used to the idea of actually looking at your activations and parameters, and plotting them and testing whether they are behaving correctly.

本書では、アクティベーションとパラメータについてよく説明します。これらは非常に特殊な意味を持っていることを忘れないでください。これらは数字なのです。抽象的な概念ではなく、モデルの中にある実際の具体的な数値なのです。優れたディープラーニングの実践者になるためには、実際に活性度とパラメータを見て、プロットして、正しく動作しているかどうかをテストするという考え方に慣れることが必要です。

Our activations and parameters are all contained in *tensors*. These are simply regularly shaped arrays—for example, a matrix. Matrices have rows and columns; we call these the *axes* or *dimensions*. The number of dimensions of a tensor is its *rank*. There are some special tensors:

活性化とパラメータは、すべてテンソルに含まれています。テンソルとは、例えば行列のような、規則的な形をした配列のことです。行列には行と列があり、これを軸または次元と呼びます。テンソルの次元数はランクと呼ばれます。いくつかの特殊なテンソルがあります：

- Rank zero: scalar

ランクゼロ：スカラー

- Rank one: vector

1 位：ベクトル

- Rank two: matrix

2 位：マトリクス

A neural network contains a number of layers. Each layer is either *linear* or *nonlinear*. We generally alternate between these two kinds of layers in a neural network. Sometimes people refer to both a linear layer and its subsequent nonlinearity together as a single layer. Yes, this is confusing. Sometimes a nonlinearity is referred to as an *activation function*.

ニューラルネットワークは、いくつかの層を含んでいます。各レイヤーは線形または非線形のどちらかである。一般に、ニューラルネットワークではこれら 2 種類のレイヤーを交互に使用します。また、線形層とそれに続く非線形層の両方をまとめて 1 つの層と呼ぶこともあります。そう、これは紛らわしいのです。また、非線形性を活性化関数と呼ぶこともあります。

<> summarizes the key concepts related to SGD.

<>は、SGD に関連する重要な概念をまとめたものです。

asciidoc

[[dljargon1]]

.Deep learning vocabulary

ディープラーニングボキャブラリー

[options="header"]

|=====

| Term | Meaning

| 用語解説 | 意味

|ReLU | Function that returns 0 for negative numbers and doesn't change positive numbers.

|ReLU | 負の数には 0 を返し、正の数には変化を与えない関数。

|Mini-batch | A small group of inputs and labels gathered together in two arrays. A gradient descent step is updated on this batch (rather than a whole epoch).

|ミニバッチ | 入力とラベルを 2 つの配列にまとめた小さなグループ。勾配降下ステップはこのバッチで更新される（全エポックではなく）。

|Forward pass | Applying the model to some input and computing the predictions.

|フォワードパス | ある入力にモデルを適用し、予測値を計算すること。

|Loss | A value that represents how well (or badly) our model is doing.

|損失 | モデルがどの程度うまくいっているか（あるいはうまくいっていないか）を表す値。

|Gradient | The derivative of the loss with respect to some parameter of the model.

|勾配（Gradient） | モデルのパラメータに対する損失の微分値。

|Backward pass | Computing the gradients of the loss with respect to all model parameters.

|バックワードパス：すべてのモデルパラメータに対する損失の勾配を計算する。

|Gradient descent | Taking a step in the directions opposite to the gradients to make the model parameters a little bit better.

|勾配降下（Gradient Descent） 勾配とは逆の方向に一歩ずつ進み、モデルパラメータを少しずつ良くしていくこと。

|Learning rate | The size of the step we take when applying SGD to update the parameters of the model.

|学習率 | SGD を適用してモデルのパラメータを更新する際に取るステップの大きさです。

|=====

> note: *Choose Your Own Adventure* Reminder: Did you choose to skip over chapters 2 & 3, in your excitement to peek under the hood? Well, here's your reminder to head back to chapter 2 now, because you'll be needing to know that stuff very soon!
>
> note:「Choose Your Own Adventure」のリマインダーです： 第 2 章と第 3 章をスキップして、ボンネットを覗くことに興奮したのでしょうか? 今すぐ第 2 章に戻りましょう！ もうすぐその知識が必要になります！

Questionnaire

1. How is a grayscale image represented on a computer? How about a color image?
グレースケールの画像は、コンピューター上でどのように表現されるのでしょうか? カラー画像はどうでしょう?

2. How are the files and folders in the MNIST_SAMPLE dataset structured? Why?

MNIST_SAMPLE データセットのファイルやフォルダーはどのような構造になっていますか? なぜか?

3. Explain how the "pixel similarity" approach to classifying digits works.
数字を分類する「画素の類似性」アプローチの仕組みを説明しなさい。

4. What is a list comprehension? Create one now that selects odd numbers from a list and doubles them.
リスト内包とは何ですか? リストから奇数を選び、それを 2 倍にするものを今すぐ作ってください。

5. What is a "rank-3 tensor"?
ランク 3 テンソル」とは何ですか?

6. What is the difference between tensor rank and shape? How do you get the rank from the shape?
テンソルのランクと形はどう違うのか? 形状からランクを求めるにはどうすればいい?

7. What are RMSE and L1 norm?

RMSE と L1 ノルムとは何ですか？

8. How can you apply a calculation on thousands of numbers at once, many thousands of times faster than a Python loop?

Python のループより何千倍も速く、一度に何千もの数に対して計算を適用するにはどうしたらいいのか？

9. Create a 3×3 tensor or array containing the numbers from 1 to 9. Double it. Select the bottom-right four numbers.

1〜9 の数字を含む 3×3 のテンソルまたは配列を作成します。それを 2 倍にする。右下の 4 つの数字を選択する。

10. What is broadcasting?
放送とは何ですか？

11. Are metrics generally calculated using the training set, or the validation set? Why?
メトリクスは一般的にトレーニングセットとバリデーションセットのどちらを使って計算するのか？ なぜか？

12. What is SGD?
SGD とは何ですか？

13. Why does SGD use mini-batches?
なぜ SGD ではミニバッチを使用するのか？

14. What are the seven steps in SGD for machine learning?
機械学習における SGD の 7 つのステップとは？

15. How do we initialize the weights in a model?
モデルの重みはどのように初期化するのか？

16. What is "loss"?
損失」とは何ですか？

17. Why can't we always use a high learning rate?
なぜ常に高い学習率を使うことができないのか？

18. What is a "gradient"?
勾配」とは何ですか？

19. Do you need to know how to calculate gradients yourself?
勾配の計算方法を自分で知る必要があるのか？

20. Why can't we use accuracy as a loss function?
なぜ損失関数として精度を使うことができないのか？

21. Draw the sigmoid function. What is special about its shape?
シグモイド関数を描いてみてください。その形は何が特別なのか？

22. What is the difference between a loss function and a metric?
損失関数とメトリックの違いは何ですか？

23. What is the function to calculate new weights using a learning rate?
学習率を使って新しい重みを計算する関数は何ですか？

24. What does the DataLoader class do?

DataLoader クラスは何をするクラスですか？

25. Write pseudocode showing the basic steps taken in each epoch for SGD.
SGD の各エポックで行われる基本的なステップを示す疑似コードを書いてください。

26. Create a function that, if passed two arguments [1,2,3,4] and 'abcd', returns [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')].

What is special about that output data structure?
2 つの引数［1,2,3,4］と 'abcd' を渡されると、[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')］を返す関数を作成しなさい。その出力データ構造は何が特別なのでしょうか？

27. What does view do in PyTorch?

PyTorch で view は何をするのですか？

28. What are the "bias" parameters in a neural network? Why do we need them?
ニューラルネットワークの「バイアス」パラメータとは何ですか？ なぜそれが必要なのか？

29. What does the @ operator do in Python?

Python で@演算子は何をするのですか？

30. What does the backward method do?

バックワードメソッドって何をするんですか？

31. Why do we have to zero the gradients?
なぜ勾配をゼロにする必要があるのでしょうか？

32. What information do we have to pass to Learner?

Learner に渡す情報は何ですか？

33. Show Python or pseudocode for the basic steps of a training loop.
学習ループの基本的なステップを Python または擬似コードで示しなさい。

34. What is "ReLU"? Draw a plot of it for values from -2 to +2.

ReLU "とは何ですか？ 2〜+2 までの値に対するプロットを描け。

35. What is an "activation function"?
活性化関数」とは何ですか？

36. What's the difference between F.relu and nn.ReLU?

F.relu と nn.ReLU は何が違うのですか？

37. The universal approximation theorem shows that any function can be approximated as closely as needed using just one nonlinearity. So why do we normally use more?
普遍的近似定理は、どんな関数もたった 1 つの非線形性を使って必要なだけ近似できることを示しています。では、なぜ私たちは通常より多くの非線形性を使用するのでしょうか？

Further Research

さらなる研究

1. Create your own implementation of Learner from scratch, based on the training loop shown in this chapter.

本章で示した学習ループを基に、Learner の実装をゼロから作成する。

2. Complete all the steps in this chapter using the full MNIST datasets (that is, for all digits, not just 3s and 7s). This is a significant project and will take you quite a bit of time to complete! You'll need to do some of your own research to figure out how to overcome some obstacles you'll meet on the way.

MNIST の全データセットを使って、本章のすべてのステップを完了する（つまり、3s と 7s だけでなく、すべての桁について）。これは重要なプロジェクトであり、完了するまでにかなりの時間がかかるでしょう！ 途中で出会ういくつかの障害を克服する方法を見つけるために、自分で研究する必要があります。