following imports:

```
import random
import numpy as np
from PIL import Image

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset, random_split, \
WeightedRandomSampler, SubsetRandomSampler
from torchvision.transforms.v2 import Compose, ToImage, Normalize,\
ToPILImage, RandomHorizontalFlip, Resize, ToDtype

import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
%matplotlib inline

from data_generation.image_classification import generate_dataset
from stepbystep.v0 import StepByStep
```

Classifying Images

Enough already with simple data points: Let's classify **images**! Although the data is different, it is still a classification problem, so we will try to predict **which class an image belongs to**.

First, let's generate some images to work with (so we don't have to use MNIST![41]).

Torchvision

Torchvision is a package containing popular datasets, model architectures, and common image transformations for computer vision.

Datasets

Many of the popular and common **datasets** are included out of the box, like MNIST, ImageNet, CIFAR, and many more. All these datasets inherit from the original `Dataset` class, so they can be naturally used with a `DataLoader` in exactly the same way we've been doing so far.

There is one particular dataset we should pay more attention to: `ImageFolder`. This is *not* a dataset itself, but a **generic dataset** that you can use with your own images, provided that they are properly organized into sub-folders, with each sub-folder named after a class and containing the corresponding images.

We'll get back to it in Chapter 6 when we use *Rock Paper Scissors* images to build a dataset using `ImageFolder`.

Models

PyTorch also includes the most popular **model architectures**, including their **pre-trained weights**, for tackling many tasks like image classification, semantic segmentation, object detection, instance segmentation, person keypoint detection, and video classification.

Among the many models, we can find the well-known AlexNet, VGG (in its many incarnations: VGG11, VGG13, VGG16, and VGG19), ResNet (also in many flavors: ResNet18, ResNet34, ResNet50, ResNet101, ResNet152), and Inception V3.

In Chapter 7, we will load a pre-trained model and fine-tune it to our particular task. In other words, we'll use **transfer learning**.

Transforms

Torchvision has some common image transformations in its **`transforms`** module. It is important to realize there are two main groups of transformations:

- Transformations for **modifying** the images

- Transformations for **converting** between formats

While we can use `ToPILImage()` to convert from a tensor to an actual image, its original counterpart, `ToTensor()`, has been deprecated since the introduction of the second version (V2) of `transforms` in TorchVision v0.15. The conversion from image to tensor has been split into two distinct operations:

- `ToImage()`: it converts a PIL image or *Numpy* array into a **tensor of pixels**, that is, preserving the original pixel values and the integer type (unlike `ToTensor()` which converted the values to float type and scaled them as well).
- `ToDtype()`: it converts the tensor to a different type and, optionally, scales the values to the `[0, 1]` range (we can replicate former `ToTensor()` behavior by calling `ToDtype(torch.float32, scale=True)` as suggested in the deprecation message).

Let's start by using `ToImage()` to convert a *Numpy* array, our example image (#7) in HWC shape, to a PyTorch tensor:

```
image_tensor = ToImage()(example_hwc)
image_tensor, image_tensor.shape
```

Output

```
(Image([[[  0, 255,   0,   0,   0],
         [  0,   0, 255,   0,   0],
         [  0,   0,   0, 255,   0],
         [  0,   0,   0,   0, 255],
         [  0,   0,   0,   0,   0]]], dtype=torch.uint8, ),
 torch.Size([1, 5, 5]))
```

Cool, we got the expected CHW shape, and the pixel values are unchanged.

"Wait a minute, it looks like this is still an image..."

The V2 of `transforms` also introduced *wrappers* for tensors representing images, videos, boxes, etc. In the output above, `Image` is actually a tensor, it's *not* a PIL image:

```
isinstance(image_tensor, torch.Tensor)
```

Output

```
True
```

See? It is really a tensor. Now, let's scale its values:

```
example_tensor = ToDtype(torch.float32, scale=True)(image_tensor)
example_tensor
```

Output

```
Image([[[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]]], )
```

That's exactly the output the deprecated transformation, `ToTensor()`, would produce. To make our lives easier, let's define our own `ToTensor()` method that combines the two transformations above and then use it to create a "*tensorizer*" (for lack of a better name). We'll use our example image (#7) in HWC shape once again:

```
def ToTensor():
    return Compose([ToImage(), ToDtype(torch.float32, scale=True)])

tensorizer = ToTensor()
example_tensor = tensorizer(example_hwc)
example_tensor
```

Output

```
Image([[[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]]], )
```

There it is, the same image tensor as before. Moreover, we can "see" the same diagonal line as in the original image. Perhaps you're wondering what that `Compose()` method is doing there, but don't worry, we'll get to it very soon.

"*So, I convert PIL images and Numpy arrays to PyTorch tensors from the start and it is all good?*"

That's pretty much it, yes. It wasn't always like that, though, since earlier versions of Torchvision implemented the **interesting transformations** for **PIL images** only.

"*What do you mean by **interesting transformations**? What do they do?*"

These transformations **modify the training images** in many different ways: rotating, shifting, flipping, cropping, blurring, zooming in, adding noise, or erasing parts of it.

"*Why would I ever want to modify my training images like that?*"

That's what's called **data augmentation**. It is a clever technique to **expand a dataset** (augment it) **without collecting more data**. In general, deep learning models are very **data-hungry**, requiring a massive number of examples to perform well. But collecting large datasets is often challenging, and sometimes impossible.

Enter data augmentation: **Rotate an image** and **pretend it is a brand new image**. Flip an image and do the same. Even better, **do it randomly** during model training, so the model sees many different versions of it.

Let's say we have an **image of a dog**. If we **rotate it**, it is **still a dog**, but from a **different angle**. Instead of taking two pictures of the dog, one from each angle, we

take the picture we already have and use data augmentation to **simulate many different angles**. Not quite the same as the real deal, but close enough to improve our model's performance. Needless to say, data augmentation is **not suited for every task**: If you are trying to perform object detection—that is, detecting the **position of an object** in a picture—you **shouldn't** do anything that changes its position, like flipping or shifting. Adding noise would still be fine, though.

This is just a brief overview of data augmentation techniques so you understand the reasoning behind including this kind of transformation in a training set.

There is also "test-time augmentation," which can be used to improve the performance of a model after it's deployed. This is more advanced, though, and beyond the scope of this book.

The bottom line is, these transformations are important. To more easily visualize the resulting images, we may use `ToPILImage()` to convert a tensor to a PIL image:

```
example_img = ToPILImage()(example_tensor)
print(type(example_img))
```

Output

```
<class 'PIL.Image.Image'>
```

Notice that it is a **real PIL image**, not a *Numpy* array anymore, so we can use Matplotlib to visualize it:

```
plt.imshow(example_img, cmap='gray')
plt.grid(False)
```

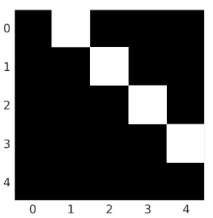

Figure 4.2 - Image #7

`ToPILImage()` can take either a **tensor in PyTorch shape (CHW)** or a ***Numpy* array in PIL shape (HWC)** as inputs.

Transforms on Images

These transforms include the typical things you'd like to do with an image for the purpose of data augmentation: `Resize()`, `CenterCrop()`, `GrayScale()`, `RandomHorizontalFlip()`, and `RandomRotation()`, to name a few. Let's use our example image above and try some **random horizontal flipping**. But, just to make sure we flip it, let's ditch the randomness and make it flip 100% of the time:

```
flipper = RandomHorizontalFlip(p=1.0)
flipped_img = flipper(example_img)
```

OK, the image should be flipped horizontally now. Let's check it out:

```
plt.imshow(flipped_img, cmap='gray')
plt.grid(False)
```

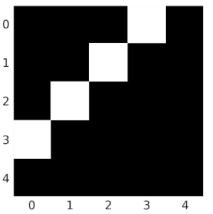

Figure 4.3 - Flipped image #7

Tensor-only Transforms

Some transforms take only tensors (but not PIL images) as inputs, such as `LinearTransformation()`, `Normalize()`, `RandomErasing()` (although I believe this one was a better fit for the other group of transforms), and `ToDtype()`, to name a few.

First, let's transform our flipped image to a tensor using the `tensorizer()` we've already created:

```
img_tensor = tensorizer(flipped_img)
img_tensor
```

Output

```
Image([[[0., 0., 0., 1., 0.],
        [0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.],
        [1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]], )
```

Normalize Transform

Now we can apply one of the most common transformations: `Normalize()`. In its documentation, we get a brief description of this transformation:

> Normalize a tensor image with mean and standard deviation. Given mean: `(mean[1],...,mean[n])` and std: `(std[1],..,std[n])` for `n` channels, this transform will normalize each channel of the input `torch.*Tensor` i.e., `output[channel] = (input[channel] - mean[channel]) / std[channel]`

Does it look familiar? That's the tensor-based version of the `StandardScaler` commonly used to standardize features, operating independently on each image channel.

*"Why is it called **normalize** then?"*

Unfortunately, there are *many names* for the procedure that involves subtracting the mean value first, and then dividing the result by the standard deviation. In my opinion, it should always be called **standardization**, as in Scikit-Learn, since "normalizing" means something else (transforming the features such that every data point has a unit norm). But, in many cases, and Torchvision is one of those cases, the standardization procedure is called **normalization** (coming from normal distribution, not from the unit norm).

Regardless of its name, standardization or normalization, this transformation modifies the range of values of a given feature or set of features. Having features in well-behaved ranges greatly improves the performance of gradient descent. Moreover, as we'll see shortly, it is better to have features with symmetrical ranges of values (from -1 to 1, for example) when training neural networks.

By definition, **pixel values can only be positive**, usually in the range [0, 255]. We see our **image tensor** has values that are in the **[0, 1] range**, and that we have only **one channel**. We can use the `Normalize()` transform to have its values mapped to a symmetrical range.

But, **instead of computing mean and standard deviation** first, let's **set the mean to 0.5** and **set the standard deviation to 0.5** as well.

"Wait a moment … why?!"

By doing so, we'll effectively be performing a **min-max scaling** (like Scikit-Learn's `MinMaxScaler`) such that the resulting range is [-1, 1]. It is easy to see why, if we compute the resulting values for the extremes of our original range [0, 1].

$$\text{input} = 0 \implies \frac{0 - \text{mean}}{\text{std}} = \frac{0 - 0.5}{0.5} = -1$$
$$\text{input} = 1 \implies \frac{1 - \text{mean}}{\text{std}} = \frac{1 - 0.5}{0.5} = 1$$

Normalizer

There we go: The resulting range is [-1, 1]. Actually, we could set it to *anything* we want. Had we chosen a standard deviation of 0.25, we would get a [-2, 2] range instead. If we had chosen a mean value different than the midpoint of the original range, we would end up with an asymmetrical range as a result.

Now, if we had taken the trouble of **actually computing the real mean and standard deviation of the training data**, we would have achieved an actual **standardization**; that is, our training data would have **zero mean** and **unit standard deviation**.

For now, let's stick with the lazy approach and use the `Normalize()` transformation

as a **min-max scaler** to the [-1, 1] range:

```
normalizer = Normalize(mean=(.5,), std=(.5,))
normalized_tensor = normalizer(img_tensor)
normalized_tensor
```

Output

```
Image([[[-1., -1., -1.,  1., -1.],
        [-1., -1.,  1., -1., -1.],
        [-1.,  1., -1., -1., -1.],
        [ 1., -1., -1., -1., -1.],
        [-1., -1., -1., -1., -1.]]], )
```

Notice that the transformation takes **two tuples** as arguments, one tuple for the means, another one for the standard deviations. Each tuple has **as many values as channels** in the image. Since we have single-channel images, our tuples have a single element each.

It is also easy to see that we achieved the desired range of values: The transformation simply converted *zeros* into negative ones and preserved the original *ones*. Good for illustrating the concept, but surely not exciting.

In Chapter 6, we'll use `Normalize()` to **standardize real (three-channel) images**.

Composing Transforms

No one expects you to run these transformations one by one; that's what **`Compose()`** can be used for: **composing several transformations** into a single, big, composed transformation. Also, I guess I could have composed a better sentence to explain it (pun intended).

It is quite simple, actually: Just line up all desired transformations in a list. This works pretty much the same way as a pipeline in Scikit-Learn. We only need to make sure the **output** of a given **transformation** is an **appropriate input** for the **next one**.

Let's compose a new transformation using the following list of transformations:

- First, let's **flip an image** using `RandomHorizontalFlip()`.
- Next, let's perform some **min-max scaling** using `Normalize()`.

In code, the sequence above looks like this:

```
composer = Compose([RandomHorizontalFlip(p=1.0),
                    Normalize(mean=(.5,), std=(.5,))])
```

If we use the composer above to transform the example **tensor**, we should get the **same normalized tensor as output**. Let's double-check it:

```
composed_tensor = composer(example_tensor)
(composed_tensor == normalized_tensor).all()
```

Output

```
tensor(True)
```

Great! We can use a single composed transformation from now on!

Notice that we have not used the original `example`, a *Numpy* array already in PyTorch shape (CHW), as input. To understand why, let's briefly compare it to the `example_tensor` we used as the actual input (a PyTorch tensor, also in CHW shape):

```
print(example)
print(example_tensor)
```

Output

```
[[[  0 255   0   0   0]
  [  0   0 255   0   0]
  [  0   0   0 255   0]
  [  0   0   0   0 255]
  [  0   0   0   0   0]]]
Image([[[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 1.],
        [0., 0., 0., 0., 0.]]], )
```

As you can see, the only differences between them are the scale (255 vs one) and the type (integer and float).

We can convert the former into the latter using a one-liner:

```
example_tensor = torch.as_tensor(example / 255).float()
```

Moreover, we can use this line of code to convert our whole *Numpy* dataset into tensors so they become an appropriate input to our composed transformation.

Data Preparation

The first step of data preparation is to convert our features and labels from *Numpy* arrays to PyTorch tensors:

```
# Builds tensors from numpy arrays BEFORE split
x_tensor = torch.as_tensor(images / 255).float()
y_tensor = torch.as_tensor(labels.reshape(-1, 1)).float()
```

The only difference is that we scaled the images to get them into the expected [0.0, 1.0] range.

Dataset Transforms

Next, we use both tensors to build a `Dataset`, but not a simple `TensorDataset`. We'll build our own **custom dataset** that is capable of **handling transformations**. Its code

is actually quite simple:

Transformed Dataset

```
class TransformedTensorDataset(Dataset):
    def __init__(self, x, y, transform=None):
        self.x = x
        self.y = y
        self.transform = transform

    def __getitem__(self, index):
        x = self.x[index]

        if self.transform:
            x = self.transform(x)

        return x, self.y[index]

    def __len__(self):
        return len(self.x)
```

It takes **three arguments**: a tensor for **features (*x*)**, another tensor for **labels (*y*)**, and an **optional transformation**. These arguments are then stored as **attributes** of the class. Of course, if *no transformation* is given, it will behave similarly to a regular `TensorDataset`.

The main difference is in the `__getitem__()` method: Instead of simply returning the elements corresponding to a given index in both tensors, it **transforms the features**, if a transformation is defined.

"*Do I **have to** create a custom dataset to perform transformations?*"

Not necessarily, no. The **`ImageFolder` dataset**, which you'll likely use for handling real images, **handles transformations out of the box**. The mechanism is essentially the same: If a transformation is defined, the dataset applies it to the images. The **purpose** of using a custom dataset here is to **illustrate this mechanism**.

So, let's redefine our composed transformations (so it *actually* flips the image *randomly* instead of every time) and create our dataset:

```
composer = Compose([RandomHorizontalFlip(p=0.5),
                    Normalize(mean=(.5,), std=(.5,))])

dataset = TransformedTensorDataset(x_tensor, y_tensor, composer)
```

Cool! But we still have to **split** the dataset as usual. But we'll do it a *bit differently* this time.

SubsetRandomSampler

Typically, when creating a data loader for the training set, we set its argument `shuffle` to `True` (since shuffling data points, in most cases, improves the performance of gradient descent). This is a very convenient way of **shuffling** the data that is **implemented using a `RandomSampler`** under the hood. Every time a new mini-batch is requested, it **samples** some indices **randomly**, and the data points corresponding to those indices are returned.

Even when there is **no shuffling** involved, as in the data loader used for the validation set, a **`SequentialSampler`** is used. In this case, whenever a new mini-batch is requested, this sampler simply returns a sequence of **indices**, **in order**, and the data points corresponding to those indices are returned.

In a nutshell, a **sampler** can be used to **return sequences of indices** to be used for data loading. In the two examples above, each sampler would take a `Dataset` as an argument. But not all samplers are like that.

The `SubsetRandomSampler` samples indices **from a list**, given as argument, without replacement. As in the other samplers, these indices are used to load data from a dataset. If an **index is not on the list**, the corresponding **data point will never be used**.

So, if we have **two disjoint lists of indices** (that is, no intersection between them, and they cover all elements if added together), we can create **two samplers** to effectively **split a dataset**. Let's put this into code to make it more clear.

First, we need to generate **two shuffled lists of indices**, one corresponding to the points in the **training set**, the other to the points in the **validation set**. We have done this already using *Numpy*. Let's make it a bit more *interesting* and *useful* this time by assembling **Helper Function #4**, aptly named `index_splitter()`, to split the indices:

Helper Function #4

```
def index_splitter(n, splits, seed=13):
    idx = torch.arange(n)
    # Makes the split argument a tensor
    splits_tensor = torch.as_tensor(splits)
    total = splits_tensor.sum().float()
    # If the total does not add up to one
    # divide every number by the total
    if not total.isclose(torch.ones(1)[0]):
        splits_tensor = splits_tensor / total
    # Uses PyTorch random_split to split the indices
    torch.manual_seed(seed)
    return random_split(idx, splits_tensor)
```

The function above takes three arguments:

- `n`: The **number of data points** to generate indices for.
- `splits`: A list of values representing the **relative weights** of the split sizes.
- `seed`: A random seed to ensure **reproducibility**.

It always bugged me a little that PyTorch's `random_split()` needed a list with the *exact* number of data points in each split. Then, since version 1.13, it started accepting proportions, as long as they add up to one. I still wish I could give it any kind of **proportions**, like `[80, 20]` or even `[4, 1]`, and then it would **figure out how many points** go into each split on its own. That's the main reason `index_splitter()` exists: We can give it relative weights, even if they do not add up to one, and it figures the number of points out.

Sure, it still calls `random_split()` to split a tensor containing a list of indices (which can also be used to split `Dataset` objects). The resulting splits are `Subset` objects:

```
train_idx, val_idx = index_splitter(len(x_tensor), [80, 20])
train_idx
```

Output

```
<torch.utils.data.dataset.Subset at 0x7fc6e7944290>
```

Helper Function #4

```
def index_splitter(n, splits, seed=13):
    idx = torch.arange(n)
    # Makes the split argument a tensor
    splits_tensor = torch.as_tensor(splits)
    total = splits_tensor.sum().float()
    # If the total does not add up to one
    # divide every number by the total
    if not total.isclose(torch.ones(1)[0]):
        splits_tensor = splits_tensor / total
    # Uses PyTorch random_split to split the indices
    torch.manual_seed(seed)
    return random_split(idx, splits_tensor)
```

Helper Function #5

```
def make_balanced_sampler(y):
    # Computes weights for compensating imbalanced classes
    classes, counts = y.unique(return_counts=True)
    weights = 1.0 / counts.float()
    sample_weights = weights[y.squeeze().long()]
    # Builds sampler with compute weights
    generator = torch.Generator()
    sampler = WeightedRandomSampler(
        weights=sample_weights,
        num_samples=len(sample_weights),
        generator=generator,
        replacement=True
    )
    return sampler
```

following imports:

```
import random
import numpy as np
from PIL import Image

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset
from torchvision.transforms.v2 import Compose, Normalize

from data_generation.image_classification import generate_dataset
from helpers import index_splitter, make_balanced_sampler
from stepbystep.v1 import StepByStep
```

Convolutions

In Chapter 4, we talked about **pixels as features**. We considered each pixel as an individual, independent feature, thus **losing information** while **flattening** the image. We also talked about **weights as pixels** and how we could interpret the weights used by a neuron as an **image**, or, more specifically, a **filter**.

Now, it is time to take that one step further and learn about **convolutions**. A convolution is "*a mathematical operation on two functions (f and g) that produces a third function (f * g) expressing how the shape of one is modified by the other.*"[52] In image processing, a **convolution matrix** is also called a **kernel** or **filter**. Typical image processing operations—like *blurring*, *sharpening*, *edge detection*, and more, are accomplished by performing a **convolution between a kernel and an image**.

Filter / Kernel

Simply put, one defines a **filter** (or *kernel*, but we're sticking with "filter" here) and **applies** this filter to an image (that is, convolving an image). Usually, the filters are **small square matrices**. The convolution itself is performed by **applying the filter on the image repeatedly**. Let's try a *concrete example* to make it more clear.

Since the **softmax** is computed using **odds ratios** instead of **log odds ratios** (logits), we need to **exponentiate the logits**!

$$z = \text{logit}(p) \quad = \log \text{ odds ratio } (p) \quad = \log\left(\frac{p}{1-p}\right)$$

$$e^z = e^{\text{logit}(p)} \quad = \text{odds ratio } (p) \qquad = \left(\frac{p}{1-p}\right)$$

Equation 5.5 - Logit and odds ratio

The softmax formula itself is quite simple:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{c=0}^{C-1} e^{z_c}}$$

Equation 5.6 - Softmax function

In the equation above, *C* stands for the **number of classes** and *i* corresponds to the index of a particular class. In our example, we have **three classes**, so our model needs to **output three logits** (z_0, z_1, z_2). Applying **softmax** to these logits, we would get the following:

$$\text{softmax}(z) = \left[\frac{e^{z_0}}{e^{z_0}+e^{z_1}+e^{z_2}}, \frac{e^{z_1}}{e^{z_0}+e^{z_1}+e^{z_2}}, \frac{e^{z_2}}{e^{z_0}+e^{z_1}+e^{z_2}}\right]$$

Equation 5.7 - Softmax for a three-class classification problem

Simple, right? Let's see it in code now. Assuming our model produces this tensor containing three logits:

```
logits = torch.tensor([ 1.3863,  0.0000, -0.6931])
```

We **exponentiate the logits** to get the corresponding **odds ratios**:

```
odds_ratios = torch.exp(logits)
odds_ratios
```

Imports

For the sake of organization, all libraries needed throughout the code used in any given chapter are imported at its very beginning. For this chapter, we'll need the following imports:

```
import numpy as np
from PIL import Image
from copy import deepcopy

import torch
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

from torch.utils.data import DataLoader, TensorDataset, random_split
from torchvision.transforms.v2 import Compose, ToImage, Normalize, \
ToPILImage, Resize, ToDtype
from torchvision.datasets import ImageFolder
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau, \
MultiStepLR, CyclicLR, LambdaLR

from stepbystep.v2 import StepByStep
from data_generation.rps import download_rps
```

Rock, Paper, Scissors...

> ...Lizard, Spock! The "extended" version of the game was displayed in the "The Lizard-Spock Expansion" episode of *The Big Bang Theory* series, and was developed by Sam Kass and Karen Bryla. To learn more about the extended version, visit Sam Kass' page[59] about the game.

Trivia aside, I guess you're probably a bit *bored* with the image dataset we've been using so far, right? Well, at least, it wasn't MNIST! But it is time to use a **different dataset**: ***Rock Paper Scissors*** (unfortunately, no lizard or Spock).

Rock Paper Scissors Dataset

This dataset was created by Laurence Moroney (lmoroney @ gmail.com / laurencemoroney.com) and can be found on his site: *Rock Paper Scissors* Dataset (*https://bit.ly/3F6qp88*).

The dataset is licensed as Creative Commons (CC BY 2.0). No changes were made to the dataset.

The dataset contains 2,892 images of diverse hands in the typical *rock*, *paper*, and *scissors* poses against a white background. This is a **synthetic dataset** as well since the images were generated using CGI techniques. Each image is 300x300 pixels in size and has four channels (RGBA).

RGBA stands for Red-Green-Blue-Alpha, which is the traditional RGB color model together with an alpha channel indicating how opaque each pixel is. Don't mind the alpha channel, it will be removed later.

The **training set** (2,520 images) can be downloaded at *https://storage.googleapis.com/download.tensorflow.org/data/rps.zip* and the **test set** (372 images) can be downloaded at *https://storage.googleapis.com/download.tensorflow.org/data/rps-test-set.zip*. In the notebook, the datasets will be downloaded and extracted to `rps` and `rps-test-set` folders, respectively.

Here are some examples of its images, one for each pose.

Figure 6.1 - Rock, paper, scissors

There are **three classes** once again, so we can use what we learned in Chapter 5.

Data Preparation

The data preparation step will be a bit more demanding this time since we'll be **standardizing the images** (for real this time—no min-max scaling anymore!). Besides, we can use the `ImageFolder` dataset now.

ImageFolder

This is *not* a dataset itself, but a **generic dataset** that you can use with your own images provided that they are properly organized into sub-folders, with each sub-folder named after a class and containing the corresponding images.

The *Rock Paper Scissors* dataset **is** organized like that: Inside the **`rps` folder of the training set**, there are three sub-folders named after the three classes (rock, paper, and scissors).

```
rps/paper/paper01-000.png
rps/paper/paper01-001.png

rps/rock/rock01-000.png
rps/rock/rock01-001.png

rps/scissors/scissors01-000.png
rps/scissors/scissors01-001.png
```

The dataset is also **perfectly balanced**, with each sub-folder containing 840 images of its particular class.

The `ImageFolder` dataset requires only the **root folder**, which is the `rps` folder in our case. But it can take another **four optional arguments**:

- `transform`: You know that one already; it tells the dataset which transformations should be applied to each image, like the data augmentation transformations we've seen in previous chapters.
- `target_transform`: So far, our targets have always been integers, so this argument wouldn't make sense; it starts making sense if your target is also an image (for instance, in a segmentation task).
- `loader`: A function that loads an image from a given path, in case you're using weird or atypical formats that cannot be handled by PIL.

- `is_valid_file`: A function that checks if a file is corrupted or not.

Let's create a dataset then:

Temporary Dataset

```
temp_transform = Compose([Resize(28), ToImage(),
                          ToDtype(torch.float32, scale=True)])
temp_dataset = ImageFolder(root='rps', transform=temp_transform)
```

We're using only the `transform` optional argument here, and keeping transformations to a minimum. First, images are **resized to 28x28 pixels** (and automatically transformed to the RGB color model by the PIL loader, thus losing the alpha channel), and then are **converted to PyTorch tensors**. Smaller images will make our models faster to train, and more "CPU-friendly." Let's take the first image of the dataset and check its shape and corresponding label:

```
temp_dataset[0][0].shape, temp_dataset[0][1]
```

Output

```
(torch.Size([3, 28, 28]), 0)
```

Perfect!

"Wait, where is the standardization you promised?"

Standardization

To standardize data points, we need to **learn their mean and standard deviation** first. What's the mean pixel value of our rock paper scissors images? And standard deviation? To compute these, we need to **load the data**. The good thing is, we have a (temporary) dataset with the resized images already! We're only missing a **data loader**.

Temporary DataLoader

```
temp_loader = DataLoader(temp_dataset, batch_size=16)
```

The Real Datasets

It's time to build our *real* datasets using the `Normalize()` transform with the statistics we learned from the (temporary) training set. The data preparation step looks like this:

Data Preparation

```
composer = Compose([Resize(28),
                    ToImage(),
                    ToDtype(torch.float32, scale=True),
                    normalizer])

train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)

# Builds a loader of each set
train_loader = DataLoader(
    train_data, batch_size=16, shuffle=True
)
val_loader = DataLoader(val_data, batch_size=16)
```

Even though the second part of the dataset was named `rps-test-set` by its author, we'll be using it as our validation dataset. Since **each dataset**, both training and validation, corresponds to a **different folder**, there is no need to split anything.

Next, we use both datasets to create the corresponding data loaders, remembering to **shuffle the training set**.

Let's take a peek at some images from the *real* training set.

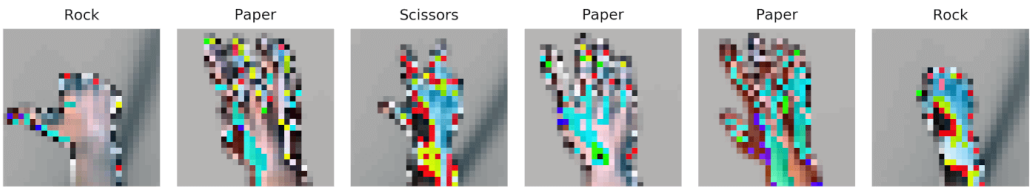

Figure 6.2 - Training set (normalized)

Data Preparation

```
# Loads temporary dataset to build normalizer
temp_transform = Compose([Resize(28), ToImage(),
                          ToDtype(torch.float32, scale=True)])
temp_dataset = ImageFolder(root='rps', transform=temp_transform)
temp_loader = DataLoader(temp_dataset, batch_size=16)
normalizer = StepByStep.make_normalizer(temp_loader)

# Builds transformation, datasets, and data loaders
composer = Compose([Resize(28), ToImage(),
                    ToDtype(torch.float32, scale=True),
                    normalizer])
train_data = ImageFolder(root='rps', transform=composer)
val_data = ImageFolder(root='rps-test-set', transform=composer)
# Builds a loader of each set
train_loader = DataLoader(
    train_data, batch_size=16, shuffle=True
)
val_loader = DataLoader(val_data, batch_size=16)
```

In the model configuration part, we can use **SGD with Nesterov's momentum** and a **higher dropout probability** to increase regularization:

Model Configuration

```
torch.manual_seed(13)
model_cnn3 = CNN2(n_feature=5, p=0.5)
multi_loss_fn = nn.CrossEntropyLoss(reduction='mean')
optimizer_cnn3 = optim.SGD(
    model_cnn3.parameters(), lr=1e-3, momentum=0.9, nesterov=True
)
```

Before the actual training, we can run an LR Range Test:

Learning Rate Range Test

```
sbs_cnn3 = StepByStep(model_cnn3, multi_loss_fn, optimizer_cnn3)
tracking, fig = sbs_cnn3.lr_range_test(
    train_loader, end_lr=2e-1, num_iter=100
)
```