PyTorch essentials

A primer: reading, running and changing a training script

Who this is for. You know Python, and the ideas of the companion primer *Deep learning for embedded engineers*: tensors, layers, logits, loss, gradient descent, training, validation and test sets. Most of the deep learning code you will work with was written by someone else: you need to read it, find where each step happens, change a setting with confidence and recognise a bug.

Two ways to use this primer

To learn, read Sections 1 to 13 in order, run the complete example of Section 13 and its variations, then check yourself with Section 14.

To look something up while working on a script, start from What do I want to do? below, the quick reference of Section 16 or the cookbook. Every section is a PDF bookmark, every section number is a link, and the footer of each page links back to both tables.

Read	This primer and *Deep learning for embedded engineers*.
Look up	`docs/intro_pytorch_cookbook.md`: short, copyable code for 15 everyday tasks, entries C1 to C15.
Run	`python3 docs/intro_pytorch_example.py`: a complete training run (Section 13). `python3 docs/intro_pytorch_snippets.py --list`: short demonstrations, one per topic, such as `--topic loss`.
Validate	`tests/test_primers.py` checks that all of this stays correct. It is for maintainers and automated checks; you do not need it to learn.

Everything runs with Python 3.8 or later and PyTorch 1.13 or later on a CPU, without any download; the commands are given from the folder that contains `docs/`. To install PyTorch: `https://pytorch.org/get-started/locally/`; online notebooks such as Google Colab already have it. In the snippets, the comment after `#` gives what Python prints, or explains the line.

Contents

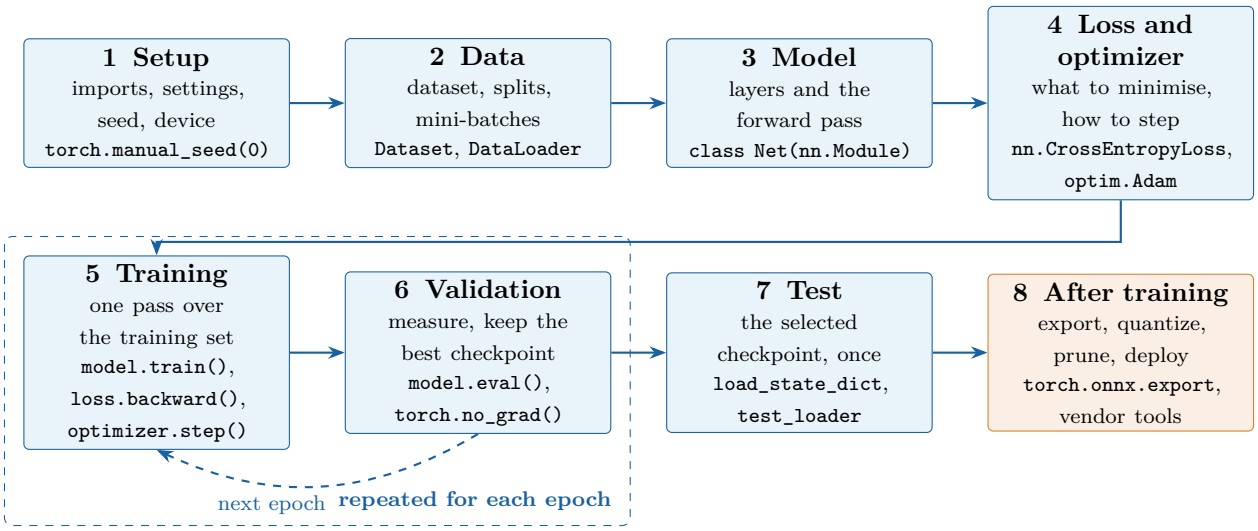
1 The anatomy of a training script
2 Tensors
3 Images and batches: N, C, H, W
4 Models: `nn.Module`
5 Data: `Dataset` and `DataLoader`
6 Logits and the loss
7 The training loop
8 Evaluation: `eval()` and `no_grad()`
9 Saving and loading checkpoints
10 Setup: device, threads and seeds
11 Reading real scripts
12 Debugging shapes and common errors
13 A complete example
14 Which stage is this?
15 To learn more
16 Quick reference, by stage
What do I want to do?

What do I want to do?

I want to…	Primer	Cookbook	Run `--topic`
understand what an unfamiliar script does	§1, §11		
tell which stage a line of code belongs to	§1, §14, §16		
create, reshape or inspect a tensor	§2	C1	`tensors`
put images in the $N \times C \times H \times W$ layout and normalise them	§3, §5	C2, C6	`images`
define a model, check its shapes and parameter count	§4, §12	C3, C4	`model`
load data in mini-batches and split it into three sets	§5	C5	`data`
compute a classification loss	§6	C7, C8	`loss`
train: one update, then a whole loop	§7, §13	C9, C10	`step`
measure loss and accuracy	§8	C11	`eval`
save or load a checkpoint	§9	C12	`checkpoint`
choose CPU or GPU, fix the seeds, limit the threads	§10	C13, C14	`setup`
understand an error message	§12	C15	`errors`
recognise quantization, pruning, distillation or export	§11		
watch a complete training run	§13	C10	(the example)

1 The anatomy of a training script

Almost every supervised training script, short or long, goes through the same eight stages. Recognising them is the first step in reading one. Stages 1 to 4 prepare; stages 5 and 6 alternate once per epoch; stage 7 happens once, at the end; stage 8 is what happens to the trained model afterwards.



Where am I in the code? Three questions

- **Which data?** `train_loader`: the parameters are being fitted. `val_loader`: a decision is being made (which epoch, which settings, which model). `test_loader`: the final number is being measured, once.
- **Can the parameters change?** `loss.backward()` followed by `optimizer.step()`: yes, this is training. `model.eval()` and `torch.no_grad()`: no, this is a measurement.
- **Which model?** Usually one. Two models in the same loop usually mean that a large, frozen network guides a small one (distillation) or that an optimised copy is compared with the original (Section 11).

2 Tensors

A `torch.Tensor` is an n-dimensional array of numbers with a **shape** and a **data type**. It behaves like a NumPy array, and it can also record the operations applied to it in order to compute gradients.

```
import torch
x = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
x.shape                  # torch.Size([2, 3])
x.dtype                  # torch.float32  (the default for real numbers)
x[1, 2]                  # tensor(6.)     row 1, column 2: indices start at 0
x[:, 0]                  # tensor([1., 4.])
x.reshape(3, 2).shape    # torch.Size([3, 2])  same 6 values, new shape
torch.zeros(2, 3); torch.ones(4); torch.randn(2, 3)   # filled with 0, 1, random normal
w = torch.randn(4, 3)
(w @ x.T).shape          # torch.Size([4, 2])  matrix product
(x * 2 + 1).sum().item() # 48.0  .item() turns a one-value tensor into a Python number
```

Arithmetic works element by element, and shapes **broadcast**: a tensor of shape `(3,)` can be added to one of shape `(2, 3)`, as in NumPy. Class labels are integers of type `torch.int64` (`torch.long`); images and weights are `torch.float32`. NumPy arrays convert with `torch.from_numpy(a)` and `t.numpy()`; NumPy defaults to `float64`, so convert with `.float()` before feeding a model.

Run: `python3 docs/intro_pytorch_snippets.py --topic tensors` **Cookbook:** C1

3 Images and batches: N, C, H, W

PyTorch layers expect a **batch** first. A batch of images has four dimensions in the order **N** (images), **C** (channels), **H** (height), **W** (width):

```
batch = torch.randn(32, 3, 32, 32)     # 32 colour images of 32 x 32 pixels
img = batch[0]                         # one image: torch.Size([3, 32, 32])
img.unsqueeze(0).shape                 # torch.Size([1, 3, 32, 32])  a batch of one
hwc = torch.rand(32, 32, 3)            # an image stored height x width x channels
hwc.permute(2, 0, 1).shape             # torch.Size([3, 32, 32])  reordered to C x H x W
```

Many image libraries store pixels as $H \times W \times C$ integers from 0 to 255; dataset loaders convert them to $C \times H \times W$ floats in $[0, 1]$ and then normalise them. When a model fails on its first input, check the order of the dimensions first.

Run: `python3 docs/intro_pytorch_snippets.py --topic images` **Cookbook:** C2, C6

4 Models: `nn.Module` stage 3: Model

A model is a class derived from `nn.Module`. Its layers are created in `__init__`, and `forward` says how the input flows through them. The layers own their parameters; PyTorch finds them automatically.

```
import torch.nn as nn

class SmallCNN(nn.Module):
    def __init__(self, num_classes=3):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=3, padding=1),    # N x 8 x 16 x 16
            nn.ReLU(),
            nn.MaxPool2d(2),                              # N x 8 x 8 x 8
            nn.Conv2d(8, 16, kernel_size=3, padding=1),   # N x 16 x 8 x 8
            nn.ReLU(),
            nn.MaxPool2d(2),                              # N x 16 x 4 x 4
```

```
        )
        self.classifier = nn.Linear(16 * 4 * 4, num_classes)

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)            # keep dimension 0 (the batch): N x 256
        return self.classifier(x)          # N x 3 logits

model = SmallCNN()
logits = model(torch.randn(5, 1, 16, 16))    # calling the model runs forward()
logits.shape                                 # torch.Size([5, 3])
sum(p.numel() for p in model.parameters())   # 2019
```

The layers match the vocabulary of the deep learning primer: `nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)` is a convolution with weights of shape `(out, in, k, k)`; `nn.Linear(in_features, out_features)` is a fully connected layer; `nn.ReLU` and `nn.MaxPool2d` have no parameters. The count above checks the formulas: $8 \cdot 1 \cdot 9 + 8 = 80$, $16 \cdot 8 \cdot 9 + 16 = 1\,168$ and $256 \cdot 3 + 3 = 771$ parameters. `nn.Sequential` simply chains layers. The last layer has no activation: the model returns logits. `print(model)` lists the layers, which is the quickest way to see the architecture of a model you did not write.

Run: `python3 docs/intro_pytorch_snippets.py --topic model` **Cookbook:** C3, C4

5 Data: `Dataset` and `DataLoader` stage 2: Data

A `Dataset` answers two questions: how many examples (`__len__`) and which example has index i (`__getitem__`, returning an input and its label). A `DataLoader` draws examples from a dataset, groups them into mini-batches and, for training, reshuffles them at every epoch.

```
from torch.utils.data import Dataset, DataLoader, TensorDataset, random_split

class PairsDataset(Dataset):               # the general form; TensorDataset does exactly this
    def __init__(self, images, labels):
        self.images, self.labels = images, labels
    def __len__(self):
        return len(self.labels)
    def __getitem__(self, i):
        return self.images[i], self.labels[i]

images = torch.randn(1500, 1, 16, 16)            # random stand-ins for real images
labels = torch.randint(0, 3, (1500,))            # class indices 0, 1, 2 (int64)
data = TensorDataset(images, labels)
train_set, val_set, test_set = random_split(data, [1000, 250, 250],
                                            generator=torch.Generator().manual_seed(0))
train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
x, y = next(iter(train_loader))
x.shape                                          # torch.Size([32, 1, 16, 16])
y.shape                                          # torch.Size([32])
len(train_loader)                                # 32 batches: 1000 / 32, rounded up
```

`torchvision.datasets` provides ready-made datasets such as MNIST, FashionMNIST and CIFAR-10, and `torchvision.transforms` the usual conversions (`ToTensor`, `Normalize`) and augmentations (random crops and flips, for the training set only). Such a dataset reads its files from the folder given as `root`; with `download=True` it fetches them first. On shared machines the files are usually prepared once in a common folder, and scripts receive that folder as a setting.

Run: `python3 docs/intro_pytorch_snippets.py --topic data` **Cookbook:** C5

6 Logits and the loss stage 4: Loss and optimizer

`nn.CrossEntropyLoss` expects the **raw logits** of shape `(N, K)` and the labels as class **indices** of shape `(N,)` and type `int64`. It applies the softmax itself, in a numerically safe way, and returns the mean loss of the batch. Classes are numbered from 0 to $K-1$; the deep learning primer, like most mathematical texts, numbers them from 1, so its class 1 is index 0 here. The values below are its worked example:

```
loss_fn = nn.CrossEntropyLoss()
logits = torch.tensor([[2.0, 1.0, 0.1]])   # one example, three classes
target = torch.tensor([0])                 # its label: class 0
loss_fn(logits, target)                    # tensor(0.4170)
torch.softmax(logits, dim=1)               # tensor([[0.6590, 0.2424, 0.0986]])
logits.argmax(dim=1)                       # tensor([0])  the predicted class
```

Do not apply a softmax before the loss

`CrossEntropyLoss(softmax(z), y)` raises no error, but it applies the softmax twice: the loss becomes flat, the gradients small, and training slow or stuck. Return logits from `forward`; call `torch.softmax` only when you need probabilities for display. The predicted class is `logits.argmax(dim=1)`, with or without a softmax.

Other tasks use other losses, called the same way, `loss_fn(outputs, targets)`: `nn.BCEWithLogitsLoss` for yes/no labels, `nn.MSELoss` or `nn.L1Loss` for numbers.

Run: `python3 docs/intro_pytorch_snippets.py --topic loss` **Cookbook:** C7, C8

7 The training loop stage 5: Training

An **optimizer** holds references to the parameters and applies the update rule. One training epoch is a loop over the mini-batches with five essential lines:

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
# alternative: torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

model.train()                          # training mode (dropout, batch normalisation)
for x, y in train_loader:
    optimizer.zero_grad()              # 1. clear the previous gradients
    logits = model(x)                  # 2. forward pass
    loss = loss_fn(logits, y)          # 3. mean loss of the mini-batch
    loss.backward()                    # 4. backpropagation: fills p.grad
    optimizer.step()                   # 5. update the parameters
```

- `loss.backward()` works because PyTorch recorded every operation of the forward pass. It *adds* the new gradients to `p.grad`; without `zero_grad()` the gradients of all previous steps would accumulate.
- `optimizer.step()` applies $\theta \leftarrow \theta - \eta \nabla_\theta L$ for SGD, or Adam's adaptive version. The learning rate is the `lr` argument. Only the parameters given to the optimizer change.
- These five lines are the signature of training: wherever you find them, parameters are being updated. The loop is repeated for several epochs, with a validation after each one.

Run: `python3 docs/intro_pytorch_snippets.py --topic step` **Cookbook:** C9, C10

8 Evaluation: `eval()` and `no_grad()` stage 6: Validation stage 7: Test

```
model.eval()                    # evaluation mode (dropout, batch normalisation)
correct = total = 0
with torch.no_grad():           # no gradients: faster, less memory
    for x, y in val_loader:
        pred = model(x).argmax(dim=1)
        correct += (pred == y).sum().item()
        total += y.numel()
print("validation accuracy: %.3f" % (correct / total))
```

`model.eval()` and `torch.no_grad()` do different things, and evaluation needs both. Remember to call `model.train()` again before the next training epoch. The same loop on `test_loader` is stage 7; the only difference is which data it reads, and that its result must not be used to make a choice.

Run: `python3 docs/intro_pytorch_snippets.py --topic eval` **Cookbook:** C11

9 Saving and loading checkpoints

Save the **state dict**, a dictionary from parameter names to tensors, rather than the whole Python object:

```
torch.save(model.state_dict(), "smallcnn_best.pt")
model = SmallCNN()                                  # same architecture, fresh weights
model.load_state_dict(torch.load("smallcnn_best.pt", map_location="cpu"))
model.eval()
```

`map_location="cpu"` loads a checkpoint saved on a GPU machine. The file contains only numbers: the code that defines the architecture must be available when loading. Keep the checkpoint that is best *on validation*; the test set is evaluated once, on that checkpoint.

Run: `python3 docs/intro_pytorch_snippets.py --topic checkpoint` **Cookbook:** C12

10 Setup: device, threads and seeds stage 1: Setup

Tensors and models live on a **device**. Many teaching and embedded machines have no GPU; the usual portable line is

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)
x, y = x.to(device), y.to(device)       # inputs must be on the same device as the model
```

On a shared machine, `torch.get_num_threads()` shows how many CPU threads PyTorch uses, and `torch.set_num_threads(n)` limits it; record it with your timings, since it changes the training time.

Initial weights, shuffling and augmentation are random. Fix the seeds at the start of a script:

```
import random, numpy as np
random.seed(0); np.random.seed(0); torch.manual_seed(0)
loader = DataLoader(train_set, batch_size=32, shuffle=True,
                    generator=torch.Generator().manual_seed(0))   # reproducible shuffling
```

With the same seed, code, PyTorch version and number of threads, a CPU run repeats; another version or machine can give slightly different numbers. A fixed seed makes a run *repeatable*, not *representative*: when you compare two models, give them the same data splits, the same training budget and the same selection rule (best validation checkpoint), and run more than one seed before concluding that one is better.

Run: `python3 docs/intro_pytorch_snippets.py --topic setup` **Cookbook:** C13, C14

11 Reading real scripts

Real scripts wrap the same stages with code that makes them configurable and traceable. Find the stages first, then the extras; the extras rarely change what the model learns.

You will also see	What it does	Stage
`argparse`, `args.lr`, `--epochs 20`	Settings read from the command line, with defaults	1
`transforms.RandomCrop`, `RandomHorizontalFlip`	Augmentation, applied to the training set only	2
`scheduler.step()` (`torch.optim.lr_scheduler`)	Changes the learning rate during training, usually once per epoch	5
`patience`, early stopping	Stops when validation has not improved for a few epochs	6
Confusion matrix, per-class recall	More detail than accuracy alone	6, 7
`logging`, progress bars, CSV or JSON files, one folder per run	Records the settings and results of each run	all

When the model is optimised for a device (stage 8). The techniques are taught on their own; here is only how to recognise them in code.

- **Post-training quantization**: a *calibration* loop that looks like an evaluation loop. The model runs on a few hundred images, without a loss or a backward pass, while the quantization tool records the range of every tensor; the tool then produces a quantized model, which is evaluated like any other.
- **Pruning** and **quantization-aware training**: the model is modified first (weights masked, channels removed, or quantization simulated), then the ordinary training loop of stage 5 runs again for a few epochs: this is fine-tuning.
- **Knowledge distillation**: two models in the training loop. The teacher is already trained, in `eval()` mode and under `no_grad()`; only the student's parameters are given to the optimizer, and the loss adds to the usual cross-entropy a term that compares the student's outputs with the teacher's.
- **Export**: `torch.onnx.export`, `torch.jit.trace` or a vendor tool turns the model into a file for another runtime or compiler. From there on, the model is no longer Python code.

Go further: PyTorch, *Pruning Tutorial*; the quantization and distillation papers listed in the deep learning primer.

12 Debugging shapes and common errors

Most first errors are shape errors. Follow the shape through the layers:

```
x = torch.zeros(1, 1, 16, 16)
for layer in model.features:
    x = layer(x)
    print(type(layer).__name__, tuple(x.shape))
# Conv2d (1, 8, 16, 16)
# ReLU (1, 8, 16, 16)
# MaxPool2d (1, 8, 8, 8)
# Conv2d (1, 16, 8, 8)
# ReLU (1, 16, 8, 8)
# MaxPool2d (1, 16, 4, 4)       -> flatten: 16 * 4 * 4 = 256 inputs for nn.Linear
```

Message or symptom	Usual cause and fix
`mat1 and mat2 shapes cannot be multiplied`	The `in_features` of an `nn.Linear` does not match the flattened size. Trace the shapes as above.
`Expected 3D (unbatched) or 4D (batched) input`	An image without its channel or batch dimension; use `unsqueeze(0)`.
`expected scalar type Long`, or another error about the target	Labels stored as floats; `CrossEntropyLoss` wants `int64` class indices: `y.long()`.
`must have the same dtype`, or a message with `Double` and `Float`	Input in `float64`, often from NumPy; convert with `.float()`.
`Expected all tensors to be on the same device`	Model and batch on different devices: move both with `.to(device)` (Section 10).
Loss stays near $\ln K$	No learning: learning rate too high or too low, labels shuffled independently of the images, or a softmax before the loss.
Loss becomes `nan`	Learning rate too high, or invalid input values; lower `lr` and check the data.
Validation accuracy varies between two evaluations	`model.eval()` missing, or the evaluation loader shuffles and augments.
Training slower and slower, memory grows	Storing `loss` instead of `loss.item()` in a list keeps every computation graph alive.

The wording of the messages varies between PyTorch versions; the `errors` demonstration prints those of your version.

Run: `python3 docs/intro_pytorch_snippets.py --topic errors` **Cookbook:** C15

13 A complete example

The program below puts everything together: synthetic data, three splits, the model of Section 4 (its width is a setting), the training loop with the mean training loss of each epoch, validation after each epoch, checkpoint selection on validation and one final test. Its comments mark the stages of Section 1. Read `main()` first, then the functions it calls. Run it from the folder that contains `docs/`:

```
python3 docs/intro_pytorch_example.py
python3 docs/intro_pytorch_example.py --help        # the settings you can change
```

It prints the device, the number of parameters and the file of the best checkpoint (`out/intro_pytorch_example/smallcnn_best.pt` under the current folder, or `--out`), then one line per epoch and the final test result.

```
"""PyTorch primer: a complete training run, from data to a tested checkpoint.

The data are generated on the fly, so nothing is downloaded: 16x16 greyscale images that contain
one short stroke, horizontal (class 0), vertical (class 1) or diagonal (class 2), at a random
position, plus noise. Runs on the CPU (Python 3.8, PyTorch 1.13 or later). From the repository root:

    python3 docs/intro_pytorch_example.py                # default settings
    python3 docs/intro_pytorch_example.py --lr 0.1       # one change at a time

The best checkpoint is written to out/intro_pytorch_example/ under the current directory (--out);
invalid settings, such as --epochs 0, are rejected before anything is written or read.
The "Stage" comments refer to the eight stages of a training script described in the primer.
"""
# Stage 1 (Setup): imports and settings
import argparse
import math
import os

import torch
import torch.nn as nn
```

```
from torch.utils.data import DataLoader, TensorDataset, random_split


def checked(convert, low, strict=False, high=None):
    """argparse type: convert the text, then reject values outside the allowed range."""
    rule = (">" if strict else ">=") + " %s" % low + ("" if high is None else " and <= %s" % high)

    def parse(text):
        value = convert(text)
        if (not math.isfinite(value) or value < low or (strict and value == low)
                or (high is not None and value > high)):
            raise argparse.ArgumentTypeError("%s is invalid: the value must be %s" % (text, rule))
        return value
    parse.__name__ = convert.__name__                    # argparse names the type in its errors
    return parse


def parse_args(argv=None):
    p = argparse.ArgumentParser(description="Train a small CNN on synthetic stroke images.")
    p.add_argument("--epochs", type=checked(int, 1), default=8)
    p.add_argument("--lr", type=checked(float, 0, strict=True), default=1e-3,
                   help="learning rate of Adam")
    p.add_argument("--noise", type=checked(float, 0), default=0.3, help="std of the pixel noise")
    p.add_argument("--width", type=checked(int, 1, high=256), default=8,
                   help="channels of the first convolution")
    p.add_argument("--seed", type=checked(int, 0, high=2**32 - 1), default=0)
    p.add_argument("--softmax-before-loss", action="store_true",
                   help="deliberate mistake: return probabilities instead of logits")
    p.add_argument("--out", default=os.path.join("out", "intro_pytorch_example"))
    return p.parse_args(argv)


# Stage 2 (Data): a synthetic dataset (a real script reads image files instead)
def make_strokes(n, generator, noise=0.3):
    """Return n images (n x 1 x 16 x 16) and their labels 0, 1 or 2."""
    images = noise * torch.randn(n, 1, 16, 16, generator=generator)
    labels = torch.randint(0, 3, (n,), generator=generator)
    for i in range(n):
        r, c = torch.randint(2, 9, (2,), generator=generator).tolist()
        for k in range(6):
            if labels[i] == 0:
                images[i, 0, r, c + k] += 1.0        # horizontal
            elif labels[i] == 1:
                images[i, 0, r + k, c] += 1.0        # vertical
            else:
                images[i, 0, r + k, c + k] += 1.0    # diagonal
    return images, labels


# Stage 3 (Model): two convolution blocks and a linear classifier
class SmallCNN(nn.Module):
    def __init__(self, num_classes=3, width=8, softmax_output=False):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, width, kernel_size=3, padding=1),          # N x w x 16 x 16
            nn.ReLU(),
            nn.MaxPool2d(2),                                        # N x w x 8 x 8
            nn.Conv2d(width, 2 * width, kernel_size=3, padding=1),  # N x 2w x 8 x 8
            nn.ReLU(),
            nn.MaxPool2d(2),                                        # N x 2w x 4 x 4
        )
        self.classifier = nn.Linear(2 * width * 4 * 4, num_classes)
        self.softmax_output = softmax_output

    def forward(self, x):
```

```
        x = self.features(x)
        x = torch.flatten(x, 1)                                    # N x (2w * 16)
        logits = self.classifier(x)                                # N x 3
        if self.softmax_output:                                    # the deliberate mistake
            return torch.softmax(logits, dim=1)
        return logits


# Stages 6 (Validation) and 7 (Test): evaluation without gradients
def evaluate(model, loader, loss_fn, device):
    """Sample-weighted mean loss and accuracy on a data loader."""
    model.eval()
    loss_sum, correct, total = 0.0, 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            loss_sum += loss_fn(outputs, y).item() * y.numel()
            correct += (outputs.argmax(dim=1) == y).sum().item()
            total += y.numel()
    return loss_sum / total, correct / total


def main(argv=None):
    # Stage 1 (Setup): settings, seed, device, output directory
    args = parse_args(argv)
    torch.manual_seed(args.seed)                           # fixes the initial weights
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    os.makedirs(args.out, exist_ok=True)
    checkpoint = os.path.join(args.out, "smallcnn_best.pt")
    if os.path.exists(checkpoint):
        os.remove(checkpoint)                              # never test the checkpoint of an earlier run

    # Stage 2 (Data): three disjoint splits and their loaders
    gen = torch.Generator().manual_seed(args.seed)
    images, labels = make_strokes(1500, gen, args.noise)
    train_set, val_set, test_set = random_split(TensorDataset(images, labels),
                                                [1000, 250, 250], generator=gen)
    train_loader = DataLoader(train_set, batch_size=32, shuffle=True,
                              generator=torch.Generator().manual_seed(args.seed))
    val_loader = DataLoader(val_set, batch_size=250)
    test_loader = DataLoader(test_set, batch_size=250)

    # Stages 3 (Model) and 4 (Loss and optimizer)
    model = SmallCNN(width=args.width, softmax_output=args.softmax_before_loss).to(device)
    loss_fn = nn.CrossEntropyLoss()                        # expects raw logits and class indices
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
    print("device: %s  parameters: %d  checkpoint: %s"
          % (device, sum(p.numel() for p in model.parameters()), os.path.abspath(checkpoint)))

    best_acc = -1.0
    for epoch in range(1, args.epochs + 1):
        # Stage 5 (Training): one pass over the training set
        model.train()
        train_loss_sum, seen = 0.0, 0
        for x, y in train_loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad()                          # forget the previous gradients
            outputs = model(x)                             # forward pass: N x 3
            loss = loss_fn(outputs, y)                     # mean loss of the mini-batch
            loss.backward()                                # backpropagation: fills p.grad
            optimizer.step()                               # update the parameters
            train_loss_sum += loss.item() * y.numel()      # weighted by the batch size
            seen += y.numel()
        # Stage 6 (Validation): measure, keep the best checkpoint
```

```
        val_loss, val_acc = evaluate(model, val_loader, loss_fn, device)
        print("epoch %2d  train loss %.3f  val loss %.3f  val acc %.3f"
              % (epoch, train_loss_sum / seen, val_loss, val_acc))
        if val_acc > best_acc:                          # select on validation only
            best_acc = val_acc
            torch.save(model.state_dict(), checkpoint)

    # Stage 7 (Test): the selected checkpoint, evaluated once
    model.load_state_dict(torch.load(checkpoint, map_location=device))
    test_loss, test_acc = evaluate(model, test_loader, loss_fn, device)
    print("selected checkpoint: val acc %.3f  test loss %.3f  test acc %.3f"
          % (best_acc, test_loss, test_acc))
    # Stage 8 (After training) would follow: export the model, or optimise it for a device
    return best_acc, test_acc


if __name__ == "__main__":
    main()
```

Try it. Change one setting at a time, keep the others and the seed, and compare the printed lines with those of the default run. Below is what usually happens; on such a small problem the details depend on the seed, so observe rather than expect.

- `--lr 0.1` and `--lr 1e-5`: a learning rate far too large usually makes the loss jump or stay high; a very small one makes it decrease slowly, and 8 epochs may not be enough.
- `--epochs 2` and `--epochs 30`: too few epochs usually leave the validation accuracy lower; beyond some point more epochs bring little on validation, even while the training loss keeps falling.
- `--noise 0.8` and `--noise 1.5`: noisier images hide the stroke; expect a lower accuracy and slower progress.
- `--width 2` and `--width 32`: the parameter count printed at the start changes (check it with the formulas of Section 4); a narrow network may learn less, a wide one costs more computation for little gain on such an easy task.
- `--softmax-before-loss`: the model returns probabilities, so the softmax is applied twice. The loss can no longer approach 0 (its minimum is $\ln(1+2/e) \approx 0.55$ for three classes) and its gradients are smaller, so learning is usually slower; the predicted classes are computed as before.
- `--seed 1`, `--seed 2`: the same settings with other random choices show how much the numbers vary on their own, before you attribute a difference to a setting.

The chance level, a model that guesses, is an accuracy of $1/3$ and a loss of $\ln 3 \approx 1.10$.

14 Which stage is this?

Give the stage of each line (1 Setup, 2 Data, 3 Model, 4 Loss and optimizer, 5 Training, 6 Validation, 7 Test, 8 After training), as if you met it in a script you did not write. Answers are below.

(a) `optimizer.step()`
(b) `random_split(data, [1000, 250, 250])`
(c) `nn.Conv2d(8, 16, 3, padding=1)`
(d) `torch.manual_seed(0)`
(e) `if val_acc > best:  torch.save(...)`
(f) `loss_fn = nn.CrossEntropyLoss()`
(g) `acc = evaluate(model, test_loader)`
(h) `torch.onnx.export(model, x, "net.onnx")`
(i) `model.train()`
(j) `DataLoader(train_set, shuffle=True)`

Two more: (k) a script picks its number of epochs by looking at the accuracy on `test_loader`; (l) an evaluation function calls `model(x)` under `torch.no_grad()` but never calls `model.eval()`. What is wrong in each?

Answers. (a) 5 Training; (b) 2 Data; (c) 3 Model; (d) 1 Setup; (e) 6 Validation; (f) 4 Loss and optimizer; (g) 7 Test; (h) 8 After training; (i) 5 Training; (j) 2 Data. (k) The test set is used to make a choice, so the reported test accuracy is optimistic: choose on `val_loader`. (l) Layers such as dropout and batch normalisation stay in training mode, so the results are wrong or vary between runs; `no_grad()` only turns off gradient recording.

15 To learn more

Official PyTorch material

- *Learn the Basics*: the official step-by-step tutorial on tensors, datasets, models, automatic differentiation, optimisation and saving.
  https://docs.pytorch.org/tutorials/beginner/basics/index.html
- *Training a Classifier*: a complete CIFAR-10 example with the same stages as this primer.
  https://docs.pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- J. Howard, *What is torch.nn really?*: writes a training loop with raw tensors, then replaces each piece by its `torch.nn` equivalent; the best way to see what the library does for you.
  https://docs.pytorch.org/tutorials/beginner/nn_tutorial.html
- The API reference, to look up any class or function you meet in a script:
  https://docs.pytorch.org/docs/stable/
- *Pruning Tutorial*, for later:
  https://docs.pytorch.org/tutorials/intermediate/pruning_tutorial.html

Books and courses

- D. Bourke, *Learn PyTorch for Deep Learning: Zero to Mastery*: a free online book with notebooks, from tensors to complete computer-vision projects. https://www.learnpytorch.io/
- *Dive into Deep Learning*: every chapter with runnable PyTorch code.
  https://classic.d2l.ai/chapter_convolutional-neural-networks/index.html
- A. Karpathy, *Neural Networks: Zero to Hero*: videos that write backpropagation from scratch, then the same networks in PyTorch. https://karpathy.ai/zero-to-hero.html

16 Quick reference: the calls you will meet, by stage

Grouped by the stages of Section 1. *See* gives the section of this primer (§) and the cookbook entry (C). Back to What do I want to do?

Call	What it does	See
1 Setup		
`torch.manual_seed(0)`	Fixes the initial weights and every random choice that follows	§10, C14
`torch.device("cpu")`, `model.to(device)`, `x.to(device)`	Where tensors and the model live; both on the same device	§10, C13
`torch.set_num_threads(n)`	Limits the CPU threads PyTorch uses	§10, C14
`argparse`, `args.lr`, `--epochs 20`	Settings read from the command line	§11
2 Data		
`class D(Dataset)` with `__len__`, `__getitem__`	A dataset: its size, and example i with its label	§5, C5
`TensorDataset(x, y)`, `random_split(data, sizes)`	A dataset from tensors; disjoint splits	§5, C5
`DataLoader(ds, batch_size=32, shuffle=True)`	Mini-batches, reshuffled at every epoch	§5, C5
`transforms.ToTensor()`, `transforms.Normalize(mean, std)`	Image to $C \times H \times W$ floats, then normalised	§3, C6
3 Model		
`class Net(nn.Module)` with `__init__` and `forward`	The model: its layers, and how data flows through them	§4, C3
`nn.Conv2d(c_in, c_out, k, stride, padding)`	Convolution; weights $c_{\text{out}} \times c_{\text{in}} \times k \times k$	§4
`nn.Linear(n_in, n_out)`	Fully connected layer	§4
`nn.ReLU()`, `nn.MaxPool2d(2)`, `nn.Sequential(...)`	Activation; pooling; a chain of layers	§4
`torch.flatten(x, 1)`	$N \times C \times H \times W$ to $N \times CHW$, before a linear layer	§4, C4
`print(model)`, `sum(p.numel() for p in model.parameters())`	The layers; the number of parameters	§4, C4
4 Loss and optimizer		
`nn.CrossEntropyLoss()`	Classification loss, from raw logits and `int64` labels	§6, C8
`torch.optim.SGD(model.parameters(), lr, momentum)`, `torch.optim.Adam(...)`	The update rule, and the parameters it may change	§7
5 Training		
`model.train()`	Training mode (dropout and batch normalisation active)	§7
`optimizer.zero_grad()`, `loss.backward()`, `optimizer.step()`	Clear the gradients, compute them, update the parameters	§7, C9
`scheduler.step()`	Changes the learning rate, usually once per epoch	§11
6 Validation and 7 Test		
`model.eval()`, `with torch.no_grad():`	Evaluation mode; no gradient recording	§8, C11
`logits.argmax(dim=1)`, `loss.item()`	Predicted classes; a one-value tensor as a Python number	§6, C7
`torch.save(model.state_dict(), path)`	Saves a checkpoint (parameters only)	§9, C12
`model.load_state_dict(torch.load(path, map_location="cpu"))`	Loads a checkpoint into the same architecture	§9, C12
8 After training		
`torch.onnx.export(model, x, path)`, `torch.jit.trace(model, x)`	Exports the model for another runtime or a compiler	§11