Below are the transformations performed on the training set. Note how most of them are for data augmentation, while normalization is done to comply with what’s expected by ResNet.

## Image preprocessing pipeline

library(torch) library(torchvision) library(torchdatasets)

library(dplyr) library(pins) library(ggplot2)

device <- if (cuda\_is\_available()) torch\_device("cuda:0") else "cpu"

train\_transforms <- function(img) { img %>%

# first convert image to tensor transform\_to\_tensor() %>%

# then move to the GPU (if available) (function(x) x$to(device = device)) %>% # data augmentation

transform\_random\_resized\_crop(size = c(224, 224)) %>% # data augmentation

transform\_color\_jitter() %>% # data augmentation

transform\_random\_horizontal\_flip() %>%

# normalize according to what is expected by resnet transform\_normalize(mean = c(0.485, 0.456, 0.406), std = c(0.229,

0.224, 0.225))

}

On the validation set, we don’t want to introduce noise, but still need to resize, crop, and normalize the images. The test set should be treated identically.

valid\_transforms <- function(img) { img %>%

transform\_to\_tensor() %>%

(function(x) x$to(device = device)) %>% transform\_resize(256) %>% transform\_center\_crop(224) %>%

transform\_normalize(mean = c(0.485, 0.456, 0.406), std = c(0.229, 0.224, 0.225))

}

test\_transforms <- valid\_transforms

And now, let’s get the data, nicely divided into training, validation and test sets. Additionally, we tell the corresponding R objects what transformations they’re expected to apply:1

train\_ds <- bird\_species\_dataset("data", download = TRUE, transform =

train\_transforms)

valid\_ds <- bird\_species\_dataset("data", split = "valid", transform = valid\_transforms)

test\_ds <- bird\_species\_dataset("data", split = "test", transform = test\_transforms)

Two things to note. First, transformations are part of the *dataset* concept, as opposed to the *data loader* we’ll encounter shortly. Second, let’s take a look at how the images have been stored on disk. The overall directory structure (starting from data, which we specified as the root directory to be used) is this:

data/bird\_species/train data/bird\_species/valid data/bird\_species/test

In the train, valid, and test directories, different classes of images reside in their own folders. For example, here is the directory layout for the first three classes in the test set:

data/bird\_species/test/ALBATROSS/

* data/bird\_species/test/ALBATROSS/1.jpg
* data/bird\_species/test/ALBATROSS/2.jpg
* data/bird\_species/test/ALBATROSS/3.jpg
* data/bird\_species/test/ALBATROSS/4.jpg
* data/bird\_species/test/ALBATROSS/5.jpg

data/test/'ALEXANDRINE PARAKEET'/

* data/bird\_species/test/'ALEXANDRINE PARAKEET'/1.jpg
* data/bird\_species/test/'ALEXANDRINE PARAKEET'/2.jpg
* data/bird\_species/test/'ALEXANDRINE PARAKEET'/3.jpg
* data/bird\_species/test/'ALEXANDRINE PARAKEET'/4.jpg
* data/bird\_species/test/'ALEXANDRINE PARAKEET'/5.jpg

data/test/'AMERICAN BITTERN'/

* data/bird\_species/test/'AMERICAN BITTERN'/1.jpg
* data/bird\_species/test/'AMERICAN BITTERN'/2.jpg
* data/bird\_species/test/'AMERICAN BITTERN'/3.jpg
* data/bird\_species/test/'AMERICAN BITTERN'/4.jpg
* data/bird\_species/test/'AMERICAN BITTERN'/5.jpg

This is exactly the kind of layout expected by torchs image\_folder\_dataset() – and really bird\_species\_dataset() instantiates a subtype of this class. Had we downloaded the data manually, respecting the required directory structure, we could have created the datasets like so:

# e.g.

train\_ds <- image\_folder\_dataset( file.path(data\_dir, "train"), transform = train\_transforms)

Now that we got the data, let’s see how many items there are in each set.

train\_ds$.length()

valid\_ds$.length() test\_ds$.length() 31316

1125

1125

That training set is really big! It’s thus recommended to run this on GPU, or just play around with the provided Colab notebook.

With so many samples, we’re curious how many classes there are.

class\_names <- test\_ds$classes length(class\_names)

225

So we *do* have a substantial training set, but the task is formidable as well: We’re going to tell apart no less than 225 different bird species.

## Data loaders

While *datasets* know what to do with each single item, *data loaders* know how to treat them collectively. How many samples make up a batch? Do we want to feed them in the same order always, or instead, have a different order chosen for every epoch?

batch\_size <- 64

train\_dl <- dataloader(train\_ds, batch\_size = batch\_size, shuffle = TRUE)

valid\_dl <- dataloader(valid\_ds, batch\_size = batch\_size) test\_dl <- dataloader(test\_ds, batch\_size = batch\_size)

Data loaders, too, may be queried for their length. Now length means: How many batches?

train\_dl$.length() valid\_dl$.length() test\_dl$.length() 490

18

18

## Some birds

Next, let’s view a few images from the test set. We can retrieve the first batch – images and corresponding classes – by creating an iterator from the dataloader and calling next() on it:

# for display purposes, here we are actually using a batch\_size of 24 batch <- train\_dl$.iter()$.next()

batch is a list, the first item being the image tensors:

batch[[1]]$size() [1] 24 3 224 224

And the second, the classes:

batch[[2]]$size() [1] 24

Classes are coded as integers, to be used as indices in a vector of class names. We’ll use those for labeling the images.

classes <- batch[[2]] classes

torch\_tensor 1

1

1

1

1

2

2

2

2

2

3

3

3

3

3

4

4

4

4

4

5

5

5

5

[ GPULongType{24} ]

The image tensors have shape batch\_size x num\_channels x height x width. For plotting using as.raster(), we need to reshape the images such that channels come last. We also undo the normalization applied by the dataloader.

Here are the first twenty-four images:

library(dplyr)

images <- as\_array(batch[[1]]) %>% aperm(perm = c(1, 3, 4, 2)) mean <- c(0.485, 0.456, 0.406

std <- c(0.229, 0.224, 0.225)

images <- std \* images + mean images <- images \* 255 images[images > 255] <- 255

images[images < 0] <- 0

par(mfcol = c(4,6), mar = rep(1, 4))

images %>% purrr::array\_tree(1) %>%

purrr::set\_names(class\_names[as\_array(classes)]) %>% purrr::map(as.raster, max = 255) %>% purrr::iwalk(~{plot(.x); title(.y)})



# Model

The backbone of our model is a pre-trained instance of ResNet.

model <- model\_resnet18(pretrained = TRUE

But we want to distinguish among our 225 bird species, while ResNet was trained on 1000 different classes. What can we do? We simply replace the output layer.

The new output layer is also the only one whose weights we are going to train – leaving all other ResNet parameters the way they are. Technically, we *could* perform backpropagation through the complete model, striving to fine-tune ResNet’s weights as well. However, this would slow down training significantly. In fact, the choice is not all-or-none: It is up to us how many of the original parameters to keep fixed, and how many to “set free” for fine tuning. For the task at hand, we’ll be content to just train the newly added output layer: With the abundance of animals, including birds, in ImageNet, we expect the trained ResNet to know a lot about them!

model$parameters %>% purrr::walk(function(param) param$requires\_grad\_(FALSE))

To replace the output layer, the model is modified in-place:

num\_features <- model$fc$in\_features

model$fc <- nn\_linear(in\_features = num\_features, out\_features = length(class\_names))

Now put the modified model on the GPU (if available):

model <- model$to(device = device

# Training

For optimization, we use cross entropy loss and stochastic gradient descent.

criterion <- nn\_cross\_entropy\_loss()

optimizer <- optim\_sgd(model$parameters, lr = 0.1, momentum = 0.9)

## Finding an optimally efficient learning rate

We set the learning rate to 0.1, but that is just a formality. As has become widely known due to the excellent lectures by fast.ai, it makes sense to spend some time upfront to determine an efficient learning rate. While out-of-the-box, torch does not provide a tool like fast.ai’s learning rate finder, the logic is straightforward to implement.

losses <- c() log\_lrs <- c()

find\_lr <- function(init\_value = 1e-8, final\_value = 10, beta = 0.98) {

num <- train\_dl$.length()

mult = (final\_value/init\_value)^(1/num) lr <- init\_value optimizer$param\_groups[[1]]$lr <- lr avg\_loss <- 0

best\_loss <- 0

batch\_num <- 0

for (b in enumerate(train\_dl)) {

batch\_num <- batch\_num + 1 optimizer$zero\_grad()

output <- model(b[[1]]$to(device = device))

loss <- criterion(output, b[[2]]$to(device = device))

#Compute the smoothed loss

avg\_loss <- beta \* avg\_loss + (1-beta) \* loss$item() smoothed\_loss <- avg\_loss / (1 - beta^batch\_num) #Stop if the loss is exploding

if (batch\_num > 1 && smoothed\_loss > 4 \* best\_loss) break #Record the best loss

if (smoothed\_loss < best\_loss || batch\_num == 1) best\_loss <- smoothed\_loss

#Store the values

losses <<- c(losses, smoothed\_loss) log\_lrs <<- c(log\_lrs, (log(lr, 10)))

loss$backward() optimizer$step()

#Update the lr for the next step lr <- lr \* mult

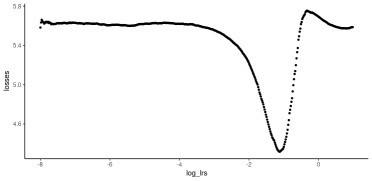
optimizer$param\_groups[[1]]$lr <- lr

}

}

find\_lr()

df <- data.frame(log\_lrs = log\_lrs, losses = losses) ggplot(df, aes(log\_lrs, losses)) + geom\_point(size = 1) + theme\_classic()



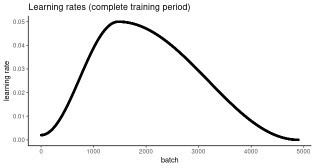
The best learning rate is not the exact one where loss is at a minimum. Instead, it should be picked somewhat earlier on the curve, while loss is still decreasing. 0.05 looks like a sensible choice.

This value is nothing but an anchor, however. *Learning rate schedulers* allow learning rates to evolve according to some proven algorithm. Among others, torch implements one-cycle learning [@abs-1708-07120], cyclical learning rates (Smith 2015), and cosine annealing with warm restarts (Loshchilov and Hutter 2016).

Here, we use lr\_one\_cycle(), passing in our newly found, optimally efficient, hopefully, value

0.05 as a maximum learning rate. lr\_one\_cycle() will start with a low rate, then gradually ramp up until it reaches the allowed maximum. After that, the learning rate will slowly, continuously decrease, until it falls slightly below its initial value.

All this happens not per epoch, but exactly once, which is why the name has one\_cycle in it. Here’s how the evolution of learning rates looks in our example:



Before we start training, let’s quickly re-initialize the model, so as to start from a clean slate:

model <- model\_resnet18(pretrained = TRUE model$parameters %>% purrr::walk(function(param)

param$requires\_grad\_(FALSE)) num\_features <- model$fc$in\_features

model$fc <- nn\_linear(in\_features = num\_features, out\_features = length(class\_names))

model <- model$to(device = device criterion <- nn\_cross\_entropy\_loss()

optimizer <- optim\_sgd(model$parameters, lr = 0.05, momentum = 0.9)

And instantiate the scheduler:

num\_epochs = 10

scheduler <- optimizer %>%

lr\_one\_cycle(max\_lr = 0.05, epochs = num\_epochs, steps\_per\_epoch = train\_dl$.length())

## Training loop

Now we train for ten epochs. For every training batch, we call scheduler$step() to adjust the learning rate. Notably, this has to be done *after* optimizer$step().

train\_batch <- function(b) {

optimizer$zero\_grad() output <- model(b[[1]])

loss <- criterion(output, b[[2]]$to(device = device)) loss$backward()

optimizer$step() scheduler$step() loss$item()

}

valid\_batch <- function(b) {

output <- model(b[[1]])

loss <- criterion(output, b[[2]]$to(device = device)) loss$item()

}

for (epoch in 1:num\_epochs) {

model$train() train\_losses <- c()

for (b in enumerate(train\_dl)) { loss <- train\_batch(b)

train\_losses <- c(train\_losses, loss)

}

model$eval() valid\_losses <- c()

for (b in enumerate(valid\_dl)) { loss <- valid\_batch(b)

valid\_losses <- c(valid\_losses, loss)

}

cat(sprintf("\nLoss at epoch %d: training: %3f, validation: %3f\n", epoch, mean(train\_losses), mean(valid\_losses)))

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Loss | at | epoch | 1: | training: | 2.662901, | validation: | 0.790769 |
| Loss | at | epoch | 2: | training: | 1.543315, | validation: | 1.014409 |
| Loss | at | epoch | 3: | training: | 1.376392, | validation: | 0.565186 |
| Loss | at | epoch | 4: | training: | 1.127091, | validation: | 0.575583 |
| Loss | at | epoch | 5: | training: | 0.916446, | validation: | 0.281600 |
| Loss | at | epoch | 6: | training: | 0.775241, | validation: | 0.215212 |
| Loss | at | epoch | 7: | training: | 0.639521, | validation: | 0.151283 |
| Loss | at | epoch | 8: | training: | 0.538825, | validation: | 0.106301 |
| Loss | at | epoch | 9: | training: | 0.407440, | validation: | 0.083270 |

Loss at epoch 10: training: 0.354659, validation: 0.080389

It looks like the model made good progress, but we don’t yet know anything about classification accuracy in absolute terms. We’ll check that out on the test set.

# Test set accuracy

Finally, we calculate accuracy on the test set:

model$eval(

test\_batch <- function(b) { output <- model(b[[1]])

labels <- b[[2]]$to(device = device) loss <- criterion(output, labels)

test\_losses <<- c(test\_losses, loss$item())

# torch\_max returns a list, with position 1 containing the values # and position 2 containing the respective indices

predicted <- torch\_max(output$data(), dim = 2)[[2]] total <<- total + labels$size(1)

# add number of correct classifications in this batch to the aggregate

correct <<- correct + (predicted == labels)$sum()$item()

}

test\_losses <- c() total <- 0

correct <- 0

for (b in enumerate(test\_dl)) { test\_batch(b)

}

mean(test\_losses [1] 0.03719

test\_accuracy <- correct/total test\_accuracy

[1] 0.98756

An impressive result, given how many different species there are!

# Wrapup

Hopefully, this has been a useful introduction to classifying images with torch, as well as to its non-domain-specific architectural elements, like datasets, data loaders, and learning-rate schedulers. Future posts will explore other domains, as well as move on beyond “hello world” in image recognition. Thanks for reading!