Based on that last state, there are just two more things to do. For one, we still compute the loss by hand. And secondly, even though we get the gradients all nicely computed from *autograd*, we still loop over the model's parameters, updating them all ourselves. You won't be surprised to hear that none of this is necessary.

Losses and loss functions

`torch` comes with all the usual loss functions, such as mean squared error, cross entropy, Kullback-Leibler divergence, and the like. In general, there are two usage modes.

Take the example of calculating mean squared error. One way is to call `nnf_mse_loss()` directly on the prediction and ground truth tensors. For example:

```
x <- torch_randn(c(3, 2, 3))
y <- torch_zeros(c(3, 2, 3))

nnf_mse_loss(x, y)
torch_tensor
0.682362
[ CPUFloatType{} ]
```

Other loss functions designed to be called directly start with `nnf_` as well: `nnf_binary_cross_entropy()`, `nnf_nll_loss()`, `nnf_kl_div()` ... and so on.[2]

The second way is to define the algorithm in advance and call it at some later time. Here, respective constructors all start with `nn_` and end in `_loss`. For example: `nn_bce_loss()`, `nn_nll_loss()`, `nn_kl_div_loss()` ...[3]

```
loss <- nn_mse_loss()

loss(x, y)
torch_tensor
0.682362
[ CPUFloatType{} ]
```

This method may be preferable when one and the same algorithm should be applied to more than one pair of tensors.

Optimizers

So far, we've been updating model parameters following a simple strategy: The gradients told us which direction on the loss curve was downward; the learning rate told us how big of a step to take. What we did was a straightforward implementation of *gradient descent*.

However, optimization algorithms used in deep learning get a lot more sophisticated than that. Below, we'll see how to replace our manual updates using `optim_adam()`, `torch`'s implementation of the Adam algorithm (Kingma and Ba 2017). First though, let's take a quick look at how `torch` optimizers work.

Here is a very simple network, consisting of just one linear layer, to be called on a single data point.

```
data <- torch_randn(1, 3)

model <- nn_linear(3, 1)
model$parameters
$weight
torch_tensor
-0.0385  0.1412 -0.5436
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.1950
[ CPUFloatType{1} ]
```

When we create an optimizer, we tell it what parameters it is supposed to work on.

```
optimizer <- optim_adam(model$parameters, lr = 0.01)
optimizer

  Inherits from:
  Public:
    add_param_group: function (param_group)
    clone: function (deep = FALSE)
    defaults: list
    initialize: function (params, lr = 0.001, betas = c(0.9, 0.999),
eps = 1e-08,
    param_groups: list
    state: list
    step: function (closure = NULL)
    zero_grad: function ()
```

At any time, we can inspect those parameters:

```
optimizer$param_groups[[1]]$params
$weight
torch_tensor
-0.0385  0.1412 -0.5436
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.1950
[ CPUFloatType{1} ]
```

Now we perform the forward and backward passes. The backward pass calculates the gradients, but does *not* update the parameters, as we can see both from the model *and* the optimizer objects:

```
out <- model(data)
out$backward()

optimizer$param_groups[[1]]$params
model$parameters
```

```
$weight
torch_tensor
-0.0385  0.1412 -0.5436
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.1950
[ CPUFloatType{1} ]

$weight
torch_tensor
-0.0385  0.1412 -0.5436
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.1950
[ CPUFloatType{1} ]
```

Calling `step()` on the optimizer actually *performs* the updates. Again, let's check that both model and optimizer now hold the updated values:

```
optimizer$step()

optimizer$param_groups[[1]]$params
model$parameters
NULL
$weight
torch_tensor
-0.0285  0.1312 -0.5536
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.2050
[ CPUFloatType{1} ]

$weight
torch_tensor
-0.0285  0.1312 -0.5536
[ CPUFloatType{1,3} ]

$bias
torch_tensor
-0.2050
[ CPUFloatType{1} ]
```

If we perform optimization in a loop, we need to make sure to call `optimizer$zero_grad()` on every step, as otherwise gradients would be accumulated. You can see this in our final version of the network.

Simple network: final version

```
library(torch)

### generate training data -----------------------------
----------------------

# input dimensionality (number of input features)
d_in <- 3
# output dimensionality (number of predicted features)
d_out <- 1
# number of observations in training set
n <- 100


# create random data
x <- torch_randn(n, d_in)
y <- x[, 1, NULL] * 0.2 - x[, 2, NULL] * 1.3 - x[, 3, NULL] * 0.5 +
torch_randn(n, 1)



### define the network ----------------------------
--------------------------

# dimensionality of hidden layer
d_hidden <- 32

model <- nn_sequential(
  nn_linear(d_in, d_hidden),
  nn_relu(),
  nn_linear(d_hidden, d_out)
)

### network parameters ----------------------------
--------------------------

# for adam, need to choose a much higher learning rate in this problem
learning_rate <- 0.08

optimizer <- optim_adam(model$parameters, lr = learning_rate)

### training loop ----------------------------
--------------------------------

for (t in 1:200) {

  ### -------- Forward pass --------

  y_pred <- model(x)

  ### -------- compute loss --------
  loss <- nnf_mse_loss(y_pred, y, reduction = "sum")
  if (t %% 10 == 0)
```

```
    cat("Epoch: ", t, "   Loss: ", loss$item(), "\n")

  ### -------- Backpropagation --------

  # Still need to zero out the gradients before the backward pass, only
this time,
  # on the optimizer object
  optimizer$zero_grad()

  # gradients are still computed on the loss tensor (no change here)
  loss$backward()

  ### -------- Update weights --------

  # use the optimizer to update model parameters
  optimizer$step()
}
```

And that's it! We've seen all the major actors on stage: tensors, *autograd*, modules, loss functions, and optimizers. In future posts, we'll explore how to use *torch* for standard deep learning tasks involving images, text, tabular data, and more. Thanks for reading!