Modules

From other frameworks (Keras, say), you may be used to distinguishing between *models* and *layers*. In `torch`, both are instances of `nn_Module()`, and thus, have some methods in common. For those thinking in terms of "models" and "layers", I'm artificially splitting up this section into two parts. In reality though, there is no dichotomy: New modules may be composed of existing ones up to arbitrary levels of recursion.

Base modules ("layers")

Instead of writing out an affine operation by hand – `x$mm(w1) + b1`, say –, as we've been doing so far, we can create a linear module. The following snippet instantiates a linear layer that expects three-feature inputs and returns a single output per observation:

```
library(torch)
l <- nn_linear(3, 1)
```

The module has two parameters, "weight" and "bias". Both now come pre-initialized:

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

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

Modules are callable; calling a module executes its `forward()` method, which, for a linear layer, matrix-multiplies input and weights, and adds the bias.

Let's try this:

```
data  <- torch_randn(10, 3)
out <- l(data)
```

Unsurprisingly, `out` now holds some data:

```
out$data()
torch_tensor
 0.2711
-1.8151
-0.0073
 0.1876
-0.0930
 0.7498
-0.2332
-0.0428
 0.3849
-0.2618
```

```
[ CPUFloatType{10,1} ]
```

In addition though, this tensor knows what will need to be done, should ever it be asked to calculate gradients:

```
out$grad_fn
AddmmBackward
```

Note the difference between tensors returned by modules and self-created ones. When creating tensors ourselves, we need to pass `requires_grad = TRUE` to trigger gradient calculation. With modules, `torch` correctly assumes that we'll want to perform backpropagation at some point.

By now though, we haven't called `backward()` yet. Thus, no gradients have yet been computed:

```
l$weight$grad
l$bias$grad
torch_tensor
[ Tensor (undefined) ]
torch_tensor
[ Tensor (undefined) ]
```

Let's change this:

```
out$backward()
Error in (function (self, gradient, keep_graph, create_graph)  :
  grad can be implicitly created only for scalar outputs (_make_grads
at ../torch/csrc/autograd/autograd.cpp:47)
```

Why the error? *Autograd* expects the output tensor to be a scalar, while in our example, we have a tensor of size `(10, 1)`. This error won't often occur in practice, where we work with *batches* of inputs (sometimes, just a single batch). But still, it's interesting to see how to resolve this.

To make the example work, we introduce a – virtual – final aggregation step – taking the mean, say. Let's call it `avg`. If such a mean were taken, its gradient with respect to `l$weight` would be obtained via the chain rule:

$$
\begin{equation*}
\frac{\partial \ avg}{\partial w} = \frac{\partial \ avg}{\partial \ out} \ \frac{\partial \ out}{\partial w}
\end{equation*}
$$

Of the quantities on the right side, we're interested in the second. We need to provide the first one, the way it would look *if really we were taking the mean*:

```
d_avg_d_out <- torch_tensor(10)$`repeat`(10)$unsqueeze(1)$t()
out$backward(gradient = d_avg_d_out)
```

Now, `l$weight$grad` and `l$bias$grad` *do* contain gradients:

```
l$weight$grad
l$bias$grad
torch_tensor
```

```
 1.3410  6.4343 -30.7135
[ CPUFloatType{1,3} ]
torch_tensor
 100
[ CPUFloatType{1} ]
```

In addition to `nn_linear()`, `torch` provides pretty much all the common layers you might hope for. But few tasks are solved by a single layer. How do you combine them? Or, in the usual lingo: How do you build *models*?

Container modules ("models")

Now, *models* are just modules that contain other modules. For example, if all inputs are supposed to flow through the same nodes and along the same edges, then `nn_sequential()` can be used to build a simple graph.

For example:

```
model <- nn_sequential(
    nn_linear(3, 16),
    nn_relu(),
    nn_linear(16, 1)
)
```

We can use the same technique as above to get an overview of all model parameters (two weight matrices and two bias vectors):

```
model$parameters
$`0.weight`
torch_tensor
-0.1968 -0.1127 -0.0504
 0.0083  0.3125  0.0013
 0.4784 -0.2757  0.2535
-0.0898 -0.4706 -0.0733
-0.0654  0.5016  0.0242
 0.4855 -0.3980 -0.3434
-0.3609  0.1859 -0.4039
 0.2851  0.2809 -0.3114
-0.0542 -0.0754 -0.2252
-0.3175  0.2107 -0.2954
-0.3733  0.3931  0.3466
 0.5616 -0.3793 -0.4872
 0.0062  0.4168 -0.5580
 0.3174 -0.4867  0.0904
-0.0981 -0.0084  0.3580
 0.3187 -0.2954 -0.5181
[ CPUFloatType{16,3} ]

$`0.bias`
torch_tensor
-0.3714
 0.5603
-0.3791
```

```
 0.4372
-0.1793
-0.3329
 0.5588
 0.1370
 0.4467
 0.2937
 0.1436
 0.1986
 0.4967
 0.1554
-0.3219
-0.0266
[ CPUFloatType{16} ]

$`2.weight`
torch_tensor
Columns 1 to 10-0.0908 -0.1786  0.0812 -0.0414 -0.0251 -0.1961  0.2326
0.0943 -0.0246  0.0748

Columns 11 to 16 0.2111 -0.1801 -0.0102 -0.0244  0.1223 -0.1958
[ CPUFloatType{1,16} ]

$`2.bias`
torch_tensor
 0.2470
[ CPUFloatType{1} ]
```

To inspect an individual parameter, make use of its position in the sequential model. For example:

```
model[[1]]$bias
torch_tensor
-0.3714
 0.5603
-0.3791
 0.4372
-0.1793
-0.3329
 0.5588
 0.1370
 0.4467
 0.2937
 0.1436
 0.1986
 0.4967
 0.1554
-0.3219
-0.0266
[ CPUFloatType{16} ]
```

And just like `nn_linear()` above, this module can be called directly on data:

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

On a composite module like this one, calling `backward()` will backpropagate through all the layers:

```
out$backward(gradient = torch_tensor(10)$`repeat`(10)$unsqueeze(1)$t())

# e.g.
model[[1]]$bias$grad
torch_tensor
  0.0000
-17.8578
  1.6246
 -3.7258
 -0.2515
 -5.8825
 23.2624
  8.4903
 -2.4604
  6.7286
 14.7760
-14.4064
 -1.0206
 -1.7058
  0.0000
 -9.7897
[ CPUFloatType{16} ]
```

And placing the composite module on the GPU will move all tensors there:

```
model$cuda()
model[[1]]$bias$grad
torch_tensor
  0.0000
-17.8578
  1.6246
 -3.7258
 -0.2515
 -5.8825
 23.2624
  8.4903
 -2.4604
  6.7286
 14.7760
-14.4064
 -1.0206
 -1.7058
  0.0000
 -9.7897
[ CUDAFloatType{16} ]
```

Now let's see how using `nn_sequential()` can simplify our example network.

Simple network using modules

```
### 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 ---------------------------------
--------------------------

learning_rate <- 1e-4

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

for (t in 1:200) {

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

  y_pred <- model(x)

  ### -------- compute loss --------
  loss <- (y_pred - y)$pow(2)$sum()
  if (t %% 10 == 0)
    cat("Epoch: ", t, "   Loss: ", loss$item(), "\n")

  ### -------- Backpropagation --------
```

```
  # Zero the gradients before running the backward pass.
  model$zero_grad()

  # compute gradient of the loss w.r.t. all learnable parameters of the
model
  loss$backward()

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

  # Wrap in with_no_grad() because this is a part we DON'T want to
record
  # for automatic gradient computation
  # Update each parameter by its `grad`

  with_no_grad({
    model$parameters %>% purrr::walk(function(param)
param$sub_(learning_rate * param$grad))
  })

}
```

The forward pass looks a lot better now; however, we still loop through the model's parameters and update each one by hand. Furthermore, you may be already be suspecting that `torch` provides abstractions for common loss functions. In the next and last installment of this series, we'll address both points, making use of `torch` losses and optimizers. See you then!