Automatic differentiation with *autograd*

`torch` uses a module called *autograd* to

1. record operations performed on tensors, and
2. store what will have to be done to obtain the corresponding gradients, once we're entering the backward pass.

These prospective actions are stored internally as functions, and when it's time to compute the gradients, these functions are applied in order: Application starts from the output node, and calculated gradients are successively *propagated back* through the network. This is a form of *reverse mode automatic differentiation*.

Autograd basics

As users, we can see a bit of the implementation. As a prerequisite for this "recording" to happen, tensors have to be created with `requires_grad = TRUE`. For example:

```
library(torch)

x <- torch_ones(2, 2, requires_grad = TRUE)
```

To be clear, `x` now is a tensor *with respect to which* gradients have to be calculated – normally, a tensor representing a weight or a bias, not the input data [1]. If we subsequently perform some operation on that tensor, assigning the result to `y`,

```
y <- x$mean()
```

we find that `y` now has a non-empty `grad_fn` that tells `torch` how to compute the gradient of `y` with respect to `x`:

```
y$grad_fn
MeanBackward0
```

Actual *computation* of gradients is triggered by calling `backward()` on the output tensor.

```
y$backward()
```

After `backward()` has been called, `x` has a non-null field termed `grad` that stores the gradient of `y` with respect to `x`:

```
x$grad
torch_tensor
 0.2500  0.2500
 0.2500  0.2500
[ CPUFloatType{2,2} ]
```

With longer chains of computations, we can take a glance at how `torch` builds up a graph of backward operations. Here is a slightly more complex example – feel free to skip if you're not the type who just *has* to peek into things for them to make sense.

Digging deeper

We build up a simple graph of tensors, with inputs `x1` and `x2` being connected to output `out` by intermediaries `y` and `z`.

```
x1 <- torch_ones(2, 2, requires_grad = TRUE)
x2 <- torch_tensor(1.1, requires_grad = TRUE)

y <- x1 * (x2 + 2)

z <- y$pow(2) * 3

out <- z$mean()
```

To save memory, intermediate gradients are normally not being stored. Calling `retain_grad()` on a tensor allows one to deviate from this default. Let's do this here, for the sake of demonstration:

```
y$retain_grad()

z$retain_grad()
```

Now we can go backwards through the graph and inspect `torch`'s action plan for backprop, starting from `out$grad_fn`, like so:

```
# how to compute the gradient for mean, the last operation executed
out$grad_fn
MeanBackward0
# how to compute the gradient for the multiplication by 3 in z =
y.pow(2) * 3
out$grad_fn$next_functions
[[1]]
MulBackward1
# how to compute the gradient for pow in z = y.pow(2) * 3
out$grad_fn$next_functions[[1]]$next_functions
[[1]]
PowBackward0
# how to compute the gradient for the multiplication in y = x * (x + 2)
out$grad_fn$next_functions[[1]]$next_functions[[1]]$next_functions
[[1]]
MulBackward0
# how to compute the gradient for the two branches of y = x * (x + 2),
# where the left branch is a leaf node (AccumulateGrad for x1)
out$grad_fn$next_functions[[1]]$next_functions[[1]]$next_
functions[[1]]$next_functions
[[1]]
torch::autograd::AccumulateGrad
[[2]]
AddBackward1
# here we arrive at the other leaf node (AccumulateGrad for x2)
out$grad_fn$next_functions[[1]]$next_functions[[1]]$next_
functions[[1]]$next_functions[[2]]$next_functions
[[1]]
torch::autograd::AccumulateGrad
```

If we now call `out$backward()`, all tensors in the graph will have their respective gradients calculated.

```
out$backward()

z$grad
y$grad
x2$grad
x1$grad
torch_tensor
 0.2500  0.2500
 0.2500  0.2500
[ CPUFloatType{2,2} ]
torch_tensor
 4.6500  4.6500
 4.6500  4.6500
[ CPUFloatType{2,2} ]
torch_tensor
 18.6000
[ CPUFloatType{1} ]
torch_tensor
 14.4150  14.4150
 14.4150  14.4150
[ CPUFloatType{2,2} ]
```

After this nerdy excursion, let's see how *autograd* makes our network simpler.

The simple network, now using *autograd*

Thanks to *autograd*, we say good-bye to the tedious, error-prone process of coding backpropagation ourselves. A single method call does it all: `loss$backward()`.

With `torch` keeping track of operations as required, we don't even have to explicitly name the intermediate tensors any more. We can code forward pass, loss calculation, and backward pass in just three lines:

```
y_pred <- x$mm(w1)$add(b1)$clamp(min = 0)$mm(w2)$add(b2)

loss <- (y_pred - y)$pow(2)$sum()

loss$backward()
```

Here is the complete code. We're at an intermediate stage: We still manually compute the forward pass and the loss, and we still manually update the weights. Due to the latter, there is something I need to explain. But I'll let you check out the new version first:

```
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)


### initialize weights ------------------------------
---------------------------

# dimensionality of hidden layer
d_hidden <- 32
# weights connecting input to hidden layer
w1 <- torch_randn(d_in, d_hidden, requires_grad = TRUE)
# weights connecting hidden to output layer
w2 <- torch_randn(d_hidden, d_out, requires_grad = TRUE)

# hidden layer bias
b1 <- torch_zeros(1, d_hidden, requires_grad = TRUE)
# output layer bias
b2 <- torch_zeros(1, d_out, requires_grad = TRUE)

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

learning_rate <- 1e-4

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

for (t in 1:200) {
  ### -------- Forward pass --------

  y_pred <- x$mm(w1)$add(b1)$clamp(min = 0)$mm(w2)$add(b2)

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

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

  # compute gradient of loss w.r.t. all tensors with requires_grad =
TRUE
  loss$backward()

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

```
  # Wrap in with_no_grad() because this is a part we DON'T
  # want to record for automatic gradient computation
   with_no_grad({
     w1 <- w1$sub_(learning_rate * w1$grad)
     w2 <- w2$sub_(learning_rate * w2$grad)
     b1 <- b1$sub_(learning_rate * b1$grad)
     b2 <- b2$sub_(learning_rate * b2$grad)

     # Zero gradients after every pass, as they'd accumulate otherwise
     w1$grad$zero_()
     w2$grad$zero_()
     b1$grad$zero_()
     b2$grad$zero_()
   })

}
```

As explained above, after `some_tensor$backward()`, all tensors preceding it in the graph[2] will have their `grad` fields populated. We make use of these fields to update the weights. But now that *autograd* is "on", whenever we execute an operation we *don't* want recorded for backprop, we need to explicitly exempt it: This is why we wrap the weight updates in a call to `with_no_grad()`.

While this is something you may file under "nice to know" – after all, once we arrive at the last post in the series, this manual updating of weights will be gone – the idiom of *zeroing gradients* is here to stay: Values stored in `grad` fields accumulate; whenever we're done using them, we need to zero them out before reuse.