…I introduced `torch`, an R package that provides the native functionality that is brought to Python users by PyTorch. In that post, I assumed basic familiarity with TensorFlow/Keras. Consequently, I portrayed `torch` in a way I figured would be helpful to someone who "grew up" with the Keras way of training a model: Aiming to focus on differences, yet not lose sight of the overall process.

This post now changes perspective. We code a simple neural network "from scratch", making use of just one of `torch`'s building blocks: *tensors*. This network will be as "raw" (low-level) as can be. (For the less math-inclined people among us, it may serve as a refresher of what's actually going on beneath all those convenience tools they built for us. But the real purpose is to illustrate what can be done with tensors alone.)

Subsequently, three posts will progressively show how to reduce the effort – noticeably right from the start, enormously once we finish. At the end of this mini-series, you will have seen how automatic differentiation works in `torch`, how to use `module`s (layers, in `keras` speak, and compositions thereof), and optimizers. By then, you'll have a lot of the background desirable when applying `torch` to real-world tasks.

This post will be the longest, since there is a lot to learn about tensors: How to create them; how to manipulate their contents and/or modify their shapes; how to convert them to R arrays, matrices or vectors; and of course, given the omnipresent need for speed: how to get all those operations executed on the GPU. Once we've cleared that agenda, we code the aforementioned little network, seeing all those aspects in action.

Tensors

Creation

Tensors may be created by specifying individual values. Here we create two one-dimensional tensors (vectors), of types `float` and `bool`, respectively:

```
library(torch)
# a 1d vector of length 2
t <- torch_tensor(c(1, 2))
t

# also 1d, but of type boolean
t <- torch_tensor(c(TRUE, FALSE))
t
torch_tensor
 1
 2
[ CPUFloatType{2} ]

torch_tensor
 1
 0
[ CPUBoolType{2} ]
```

And here are two ways to create two-dimensional tensors (matrices). Note how in the second approach, you need to specify `byrow = TRUE` in the call to `matrix()` to get values arranged in row-major order.

```
# a 3x3 tensor (matrix)
t <- torch_tensor(rbind(c(1,2,0), c(3,0,0), c(4,5,6)))
t

# also 3x3
t <- torch_tensor(matrix(1:9, ncol = 3, byrow = TRUE))
t
torch_tensor
 1  2  0
 3  0  0
 4  5  6
[ CPUFloatType{3,3} ]

torch_tensor
 1  2  3
 4  5  6
 7  8  9
[ CPULongType{3,3} ]
```

In higher dimensions especially, it can be easier to specify the type of tensor abstractly, as in: "give me a tensor of <…> of shape n1 x n2", where <…> could be "zeros"; or "ones"; or, say, "values drawn from a standard normal distribution":

```
# a 3x3 tensor of standard-normally distributed values
t <- torch_randn(3, 3)
t

# a 4x2x2 (3d) tensor of zeroes
t <- torch_zeros(4, 2, 2)
t
torch_tensor
-2.1563  1.7085  0.5245
 0.8955 -0.6854  0.2418
 0.4193 -0.7742 -1.0399
[ CPUFloatType{3,3} ]

torch_tensor
(1,.,.) =
  0  0
  0  0

(2,.,.) =
  0  0
  0  0

(3,.,.) =
  0  0
  0  0

(4,.,.) =
  0  0
  0  0
[ CPUFloatType{4,2,2} ]
```

Many similar functions exist, including, e.g., `torch_arange()` to create a tensor holding a sequence of evenly spaced values, `torch_eye()` which returns an identity matrix, and `torch_logspace()` which fills a specified range with a list of values spaced logarithmically.

If no `dtype` argument is specified, `torch` will infer the data type from the passed-in value(s). For example:

```
t <- torch_tensor(c(3, 5, 7))
t$dtype

t <- torch_tensor(1L)
t$dtype
torch_Float
torch_Long
```

But we can explicitly request a different `dtype` if we want:

```
t <- torch_tensor(2, dtype = torch_double())
t$dtype
torch_Double
```

`torch` tensors live on a *device*. By default, this will be the CPU:

```
t$device
torch_device(type='cpu')
```

But we could also define a tensor to live on the GPU:

```
t <- torch_tensor(2, device = "cuda")
t$device
torch_device(type='cuda', index=0)
```

We'll talk more about devices below.

There is another very important parameter to the tensor-creation functions: `requires_grad`. Here though, I need to ask for your patience: This one will prominently figure in the follow-up post.

Conversion to built-in R data types

To convert `torch` tensors to R, use `as_array()`:

```
t <- torch_tensor(matrix(1:9, ncol = 3, byrow = TRUE))
as_array(t)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Depending on whether the tensor is one-, two-, or three-dimensional, the resulting R object will be a vector, a matrix, or an array:

```
t <- torch_tensor(c(1, 2, 3))
as_array(t) %>% class()

t <- torch_ones(c(2, 2))
as_array(t) %>% class()

t <- torch_ones(c(2, 2, 2))
as_array(t) %>% class()
[1] "numeric"

[1] "matrix" "array"

[1] "array"
```

For one-dimensional and two-dimensional tensors, it is also possible to use `as.integer()` / `as.matrix()`. (One reason you might want to do this is to have more self-documenting code.)

If a tensor currently lives on the GPU, you need to move it to the CPU first:

```
t <- torch_tensor(2, device = "cuda")
as.integer(t$cpu())
[1] 2
```

Indexing and slicing tensors

Often, we want to retrieve not a complete tensor, but only some of the values it holds, or even just a single value. In these cases, we talk about *slicing* and *indexing*, respectively.

In R, these operations are 1-based, meaning that when we specify offsets, we assume for the very first element in an array to reside at offset `1`. The same behavior was implemented for `torch`. Thus, a lot of the functionality described in this section should feel intuitive.

The way I'm organizing this section is the following. We'll inspect the intuitive parts first, where by intuitive I mean: intuitive to the R user who has not yet worked with Python's NumPy. Then come things which, to this user, may look more surprising, but will turn out to be pretty useful.

Indexing and slicing: the R-like part

None of these should be overly surprising:

```
t <- torch_tensor(rbind(c(1,2,3), c(4,5,6)))
t

# a single value
t[1, 1]

# first row, all columns
t[1, ]

# first row, a subset of columns
t[1, 1:2]
torch_tensor
 1  2  3
 4  5  6
```

```
[ CPUFloatType{2,3} ]

torch_tensor
1
[ CPUFloatType{} ]

torch_tensor
 1
 2
 3
[ CPUFloatType{3} ]

torch_tensor
 1
 2
[ CPUFloatType{2} ]
```

Note how, just as in R, singleton dimensions are dropped:

```
t <- torch_tensor(rbind(c(1,2,3), c(4,5,6)))

# 2x3
t$size()

# just a single row: will be returned as a vector
t[1, 1:2]$size()

# a single element
t[1, 1]$size()
[1] 2 3

[1] 2

integer(0)
```

And just like in R, you can specify `drop = FALSE` to keep those dimensions:

```
t[1, 1:2, drop = FALSE]$size()

t[1, 1, drop = FALSE]$size()
[1] 1 2

[1] 1 1
```

Indexing and slicing: What to look out for

Whereas R uses negative numbers to remove elements at specified positions, in `torch` negative values indicate that we start counting from the end of a tensor – with `-1` pointing to its last element:

```
t <- torch_tensor(rbind(c(1,2,3), c(4,5,6)))

t[1, -1]
```

```
t[ , -2:-1]
torch_tensor
3
[ CPUFloatType{} ]

torch_tensor
 2  3
 5  6
[ CPUFloatType{2,2} ]
```

This is a feature you might know from NumPy. Same with the following.

When the slicing expression `m:n` is augmented by another colon and a third number – `m:n:o` –, we will take every `o`th item from the range specified by `m` and `n`:

```
t <- torch_tensor(1:10)
t[2:10:2]
torch_tensor
  2
  4
  6
  8
 10
[ CPULongType{5} ]
```

Sometimes we don't know how many dimensions a tensor has, but we do know what to do with the final dimension, or the first one. To subsume all others, we can use `..`:

```
t <- torch_randint(-7, 7, size = c(2, 2, 2))
t

t[.., 1]

t[2, ..]
torch_tensor
(1,.,.) =
  2 -2
 -5  4

(2,.,.) =
  0  4
 -3 -1
[ CPUFloatType{2,2,2} ]

torch_tensor
 2 -5
 0 -3
[ CPUFloatType{2,2} ]

torch_tensor
 0  4
-3 -1
```

```
[ CPUFloatType{2,2} ]
```

Now we move on to a topic that, in practice, is just as indispensable as slicing: changing tensor *shapes*.

Reshaping tensors

Changes in shape can occur in two fundamentally different ways. Seeing how “reshape” really means: *keep the values but modify their layout*, we could either alter how they’re arranged physically, or keep the physical structure as-is and just change the “mapping” (a semantic change, as it were).

In the first case, storage will have to be allocated for two tensors, source and target, and elements will be copied from the latter to the former. In the second, physically there will be just a single tensor, referenced by two logical entities with distinct metadata.

Not surprisingly, for performance reasons, the second operation is preferred.

Zero-copy reshaping

We start with zero-copy methods, as we’ll want to use them whenever we can.

A special case often seen in practice is adding or removing a singleton dimension.

`unsqueeze()` adds a dimension of size `1` at a position specified by `dim`:

```
t1 <- torch_randint(low = 3, high = 7, size = c(3, 3, 3))
t1$size()

t2 <- t1$unsqueeze(dim = 1)
t2$size()

t3 <- t1$unsqueeze(dim = 2)
t3$size()
[1] 3 3 3

[1] 1 3 3 3

[1] 3 1 3 3
```

Conversely, `squeeze()` removes singleton dimensions:

```
t4 <- t3$squeeze()
t4$size()
[1] 3 3 3
```

The same could be accomplished with `view()`. `view()`, however, is much more general, in that it allows you to reshape the data to any valid dimensionality. (Valid meaning: The number of elements stays the same.)

Here we have a `3x2` tensor that is reshaped to size `2x3`:

```
t1 <- torch_tensor(rbind(c(1, 2), c(3, 4), c(5, 6)))
t1
```

```
t2 <- t1$view(c(2, 3))
t2
torch_tensor
 1  2
 3  4
 5  6
[ CPUFloatType{3,2} ]

torch_tensor
 1  2  3
 4  5  6
[ CPUFloatType{2,3} ]
```

(Note how this is different from matrix transposition.)

Instead of going from two to three dimensions, we can flatten the matrix to a vector.

```
t4 <- t1$view(c(-1, 6))

t4$size()

t4
[1] 1 6

torch_tensor
 1  2  3  4  5  6
[ CPUFloatType{1,6} ]
```

In contrast to indexing operations, this does not drop dimensions.

Like we said above, operations like `squeeze()` or `view()` do not make copies. Or, put differently: The output tensor shares storage with the input tensor. We can in fact verify this ourselves:

```
t1$storage()$data_ptr()

t2$storage()$data_ptr()
[1] "0x5648d02ac800"

[1] "0x5648d02ac800"
```

What’s different is the storage *metadata* `torch` keeps about both tensors. Here, the relevant information is the *stride*:

A tensor’s `stride()` method tracks, *for every dimension*, how many elements have to be traversed to arrive at its next element (row or column, in two dimensions). For `t1` above, of shape `3x2`, we have to skip over 2 items to arrive at the next row. To arrive at the next column though, in every row we just have to skip a single entry:

```
t1$stride()
[1] 2 1
```

For `t2`, of shape `3x2`, the distance between column elements is the same, but the distance between rows is now 3:

```
t2$stride()
[1] 3 1
```

While zero-copy operations are optimal, there are cases where they won't work.

With `view()`, this can happen when a tensor was obtained via an operation – other than `view()` itself – that itself has already modified the *stride*. One example would be `transpose()`:

```
t1 <- torch_tensor(rbind(c(1, 2), c(3, 4), c(5, 6)))
t1
t1$stride()

t2 <- t1$t()
t2
t2$stride()
torch_tensor
 1  2
 3  4
 5  6
[ CPUFloatType{3,2} ]

[1] 2 1

torch_tensor
 1  3  5
 2  4  6
[ CPUFloatType{2,3} ]

[1] 1 2
```

In `torch` lingo, tensors – like `t2` – that re-use existing storage (and just read it differently), are said not to be "contiguous"[1]. One way to reshape them is to use `contiguous()` on them before. We'll see this in the next subsection.

Reshape with copy

In the following snippet, trying to reshape `t2` using `view()` fails, as it already carries information indicating that the underlying data should not be read in physical order.

```
t1 <- torch_tensor(rbind(c(1, 2), c(3, 4), c(5, 6)))

t2 <- t1$t()

t2$view(6) # error!
Error in (function (self, size)  :
  view size is not compatible with input tensor's size and stride (at
least one dimension spans across two contiguous subspaces).
  Use .reshape(...) instead. (view at ../aten/src/ATen/native/
TensorShape.cpp:1364)
```

However, if we first call `contiguous()` on it, a *new tensor* is created, which may then be (virtually) reshaped using `view()`.[2]

```
t3 <- t2$contiguous()

t3$view(6)
torch_tensor
 1
 3
 5
 2
 4
 6
[ CPUFloatType{6} ]
```

Alternatively, we can use `reshape()`. `reshape()` defaults to `view()`-like behavior if possible; otherwise it will create a physical copy.

```
t2$storage()$data_ptr()

t4 <- t2$reshape(6)

t4$storage()$data_ptr()
[1] "0x5648d49b4f40"

[1] "0x5648d2752980"
```

Operations on tensors

Unsurprisingly, `torch` provides a bunch of mathematical operations on tensors; we'll see some of them in the network code below, and you'll encounter lots more when you continue your `torch` journey. Here, we quickly take a look at the overall tensor method semantics.

Tensor methods normally return references to new objects. Here, we add to `t1` a clone of itself:

```
t1 <- torch_tensor(rbind(c(1, 2), c(3, 4), c(5, 6)))
t2 <- t1$clone()

t1$add(t2)
torch_tensor
  2   4
  6   8
 10  12
[ CPUFloatType{3,2} ]
```

In this process, `t1` has not been modified:

```
t1
torch_tensor
 1  2
 3  4
 5  6
[ CPUFloatType{3,2} ]
```

Many tensor methods have variants for mutating operations. These all carry a trailing underscore:

```
t1$add_(t1)

# now t1 has been modified
t1
torch_tensor
  4   8
 12  16
 20  24
[ CPUFloatType{3,2} ]

torch_tensor
  4   8
 12  16
 20  24
[ CPUFloatType{3,2} ]
```

Alternatively, you can of course assign the new object to a new reference variable:

```
t3 <- t1$add(t1)

t3
torch_tensor
  8  16
 24  32
 40  48
[ CPUFloatType{3,2} ]
```

There is one thing we need to discuss before we wrap up our introduction to tensors: How can we have all those operations executed on the GPU?

Running on GPU

To check if your GPU(s) is/are visible to torch, run

```
cuda_is_available()

cuda_device_count()
[1] TRUE

[1] 1
```

Tensors may be requested to live on the GPU right at creation:

```
device <- torch_device("cuda")

t <- torch_ones(c(2, 2), device = device)
```

Alternatively, they can be moved between devices at any time:

```
t2 <- t$cuda()
t2$device
torch_device(type='cuda', index=0)
t3 <- t2$cpu()
t3$device
```

```
torch_device(type='cpu')
```

That's it for our discussion on tensors — almost. There is one `torch` feature that, although related to tensor operations, deserves special mention. It is called broadcasting, and "bilingual" (R + Python) users will know it from NumPy.

Broadcasting

We often have to perform operations on tensors with shapes that don't match exactly.

Unsurprisingly, we can add a scalar to a tensor:

```
t1 <- torch_randn(c(3,5))

t1 + 22
torch_tensor
 23.1097  21.4425  22.7732  22.2973  21.4128
 22.6936  21.8829  21.1463  21.6781  21.0827
 22.5672  21.2210  21.2344  23.1154  20.5004
[ CPUFloatType{3,5} ]
```

The same will work if we add tensor of size `1`:

```
t1 <- torch_randn(c(3,5))

t1 + torch_tensor(c(22))
```

Adding tensors of different sizes normally won't work:

```
t1 <- torch_randn(c(3,5))
t2 <- torch_randn(c(5,5))

t1$add(t2) # error
Error in (function (self, other, alpha)  :
  The size of tensor a (2) must match the size of tensor b (5) at non-
singleton dimension 1 (infer_size at ../aten/src/ATen/ExpandUtils.
cpp:24)
```

However, under certain conditions, one or both tensors may be virtually expanded so both tensors line up. This behavior is what is meant by *broadcasting*. The way it works in `torch` is not just inspired by, but actually identical to that of NumPy.

The rules are:

1. We align array shapes, *starting from the right*.

   Say we have two tensors, one of size `8x1x6x1`, the other of size `7x1x5`.

   Here they are, right-aligned:

```
# t1, shape:     8  1  6  1
# t2, shape:        7  1  5
```

2. *Starting to look from the right*, the sizes along aligned axes either have to match exactly, or one of them has to be equal to `1`: in which case the latter is *broadcast* to the larger one.

In the above example, this is the case for the second-from-last dimension. This now gives

```
# t1, shape:     8  1  6  1
# t2, shape:        7  6  5
```

, with broadcasting happening in `t2`.

3. If on the left, one of the arrays has an additional axis (or more than one), the other is virtually expanded to have a size of `1` in that place, in which case broadcasting will happen as stated in (2).

   This is the case with `t1`'s leftmost dimension. First, there is a virtual expansion

```
# t1, shape:     8  1  6  1
# t2, shape:     1  7  1  5
```

and then, broadcasting happens:

```
# t1, shape:     8  1  6  1
# t2, shape:     8  7  1  5
```

According to these rules, our above example

```
t1 <- torch_randn(c(3,5))
t2 <- torch_randn(c(5,5))

t1$add(t2)
```

could be modified in various ways that would allow for adding two tensors.

For example, if `t2` were `1x5`, it would only need to get broadcast to size `3x5` before the addition operation:

```
t1 <- torch_randn(c(3,5))
t2 <- torch_randn(c(1,5))

t1$add(t2)
torch_tensor
-1.0505  1.5811  1.1956 -0.0445  0.5373
 0.0779  2.4273  2.1518 -0.6136  2.6295
 0.1386 -0.6107 -1.2527 -1.3256 -0.1009
[ CPUFloatType{3,5} ]
```

If it were of size `5`, a virtual leading dimension would be added, and then, the same broadcasting would take place as in the previous case.

```
t1 <- torch_randn(c(3,5))
t2 <- torch_randn(c(5))

t1$add(t2)
torch_tensor
-1.4123  2.1392 -0.9891  1.1636 -1.4960
 0.8147  1.0368 -2.6144  0.6075 -2.0776
-2.3502  1.4165  0.4651 -0.8816 -1.0685
[ CPUFloatType{3,5} ]
```

Here is a more complex example. Broadcasting how happens both in `t1` and in `t2`:

```
t1 <- torch_randn(c(1,5))
t2 <- torch_randn(c(3,1))

t1$add(t2)
torch_tensor
 1.2274  1.1880  0.8531  1.8511 -0.0627
 0.2639  0.2246 -0.1103  0.8877 -1.0262
-1.5951 -1.6344 -1.9693 -0.9713 -2.8852
[ CPUFloatType{3,5} ]
```

As a nice concluding example, through broadcasting an outer product can be computed like so:

```
t1 <- torch_tensor(c(0, 10, 20, 30))

t2 <- torch_tensor(c(1, 2, 3))

t1$view(c(4,1)) * t2
torch_tensor
  0   0   0
 10  20  30
 20  40  60
 30  60  90
[ CPUFloatType{4,3} ]
```

And now, we really get to implementing that neural network!

A simple neural network using `torch` tensors

Our task, which we approach in a low-level way today but considerably simplify in upcoming installments, consists of regressing a single target datum based on three input variables.

We directly use `torch` to simulate some data.

Toy data

```
library(torch)

# 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
# input
x <- torch_randn(n, d_in)
# target
y <- x[, 1, drop = FALSE] * 0.2 -
  x[, 2, drop = FALSE] * 1.3 -
  x[, 3, drop = FALSE] * 0.5 +
  torch_randn(n, 1)
```

Next, we need to initialize the network's weights. We'll have one hidden layer, with `32` units. The output layer's size, being determined by the task, is equal to `1`.

Initialize weights

```
# dimensionality of hidden layer
d_hidden <- 32

# weights connecting input to hidden layer
w1 <- torch_randn(d_in, d_hidden)
# weights connecting hidden to output layer
w2 <- torch_randn(d_hidden, d_out)

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

Now for the training loop proper. The training loop here really *is* the network.

Training loop

In each iteration ("epoch"), the training loop does four things:

- runs through the network, computing predictions (*forward pass)*
- compares those predictions to the ground truth and quantify the loss
- runs backwards through the network, computing the gradients that indicate how the weights should be changed
- updates the weights, making use of the requested learning rate.

Here is the template we're going to fill:

```
for (t in 1:200) {

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

    # here we'll compute the prediction


    ### -------- compute loss --------

    # here we'll compute the sum of squared errors


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

    # here we'll pass through the network, calculating the required
gradients
```

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

    # here we'll update the weights, subtracting portion of the
gradients
}
```

The forward pass effectuates two affine transformations, one each for the hidden and output layers. In-between, ReLU activation is applied:

```
# compute pre-activations of hidden layers (dim: 100 x 32)
# torch_mm does matrix multiplication
h <- x$mm(w1) + b1

# apply activation function (dim: 100 x 32)
# torch_clamp cuts off values below/above given thresholds
h_relu <- h$clamp(min = 0)

# compute output (dim: 100 x 1)
y_pred <- h_relu$mm(w2) + b2
```

Our loss here is mean squared error:

```
loss <- as.numeric((y_pred - y)$pow(2)$sum())
```

Calculating gradients the manual way is a bit tedious[3], but it can be done:

```
# gradient of loss w.r.t. prediction (dim: 100 x 1)
grad_y_pred <- 2 * (y_pred - y)
# gradient of loss w.r.t. w2 (dim: 32 x 1)
grad_w2 <- h_relu$t()$mm(grad_y_pred)
# gradient of loss w.r.t. hidden activation (dim: 100 x 32)
grad_h_relu <- grad_y_pred$mm(w2$t())
# gradient of loss w.r.t. hidden pre-activation (dim: 100 x 32)
grad_h <- grad_h_relu$clone()

grad_h[h < 0] <- 0

# gradient of loss w.r.t. b2 (shape: ())
grad_b2 <- grad_y_pred$sum()

# gradient of loss w.r.t. w1 (dim: 3 x 32)
grad_w1 <- x$t()$mm(grad_h)
# gradient of loss w.r.t. b1 (shape: (32, ))
grad_b1 <- grad_h$sum(dim = 1)
```

The final step then uses the calculated gradients to update the weights:

```
learning_rate <- 1e-4

w2 <- w2 - learning_rate * grad_w2
b2 <- b2 - learning_rate * grad_b2
w1 <- w1 - learning_rate * grad_w1
```

```
  b1 <- b1 - learning_rate * grad_b1
```

Let’s use these snippets to fill in the gaps in the above template, and give it a try!

Putting it all together

```
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)
# weights connecting hidden to output layer
w2 <- torch_randn(d_hidden, d_out)

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

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

learning_rate <- 1e-4

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

for (t in 1:200) {
  ### -------- Forward pass --------
  # compute pre-activations of hidden layers (dim: 100 x 32)
  h <- x$mm(w1) + b1
  # apply activation function (dim: 100 x 32)
  h_relu <- h$clamp(min = 0)
  # compute output (dim: 100 x 1)
  y_pred <- h_relu$mm(w2) + b2

  ### -------- compute loss --------

  loss <- as.numeric((y_pred - y)$pow(2)$sum())

  if (t %% 10 == 0)
    cat("Epoch: ", t, "    Loss: ", loss, "\n")

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

  # gradient of loss w.r.t. prediction (dim: 100 x 1)
  grad_y_pred <- 2 * (y_pred - y)
  # gradient of loss w.r.t. w2 (dim: 32 x 1)
  grad_w2 <- h_relu$t()$mm(grad_y_pred)
  # gradient of loss w.r.t. hidden activation (dim: 100 x 32)
  grad_h_relu <- grad_y_pred$mm(
    w2$t())
  # gradient of loss w.r.t. hidden pre-activation (dim: 100 x 32)
  grad_h <- grad_h_relu$clone()

  grad_h[h < 0] <- 0

  # gradient of loss w.r.t. b2 (shape: ())
  grad_b2 <- grad_y_pred$sum()

  # gradient of loss w.r.t. w1 (dim: 3 x 32)
  grad_w1 <- x$t()$mm(grad_h)
  # gradient of loss w.r.t. b1 (shape: (32, ))
  grad_b1 <- grad_h$sum(dim = 1)

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

  w2 <- w2 - learning_rate * grad_w2
  b2 <- b2 - learning_rate * grad_b2
  w1 <- w1 - learning_rate * grad_w1
  b1 <- b1 - learning_rate * grad_b1

}
Epoch:  10     Loss:  352.3585
Epoch:  20     Loss:  219.3624
Epoch:  30     Loss:  155.2307
Epoch:  40     Loss:  124.5716
Epoch:  50     Loss:  109.2687
Epoch:  60     Loss:  100.1543
Epoch:  70     Loss:  94.77817
Epoch:  80     Loss:  91.57003
```

```
Epoch:  90     Loss:  89.37974
Epoch:  100    Loss:  87.64617
Epoch:  110    Loss:  86.3077
Epoch:  120    Loss:  85.25118
Epoch:  130    Loss:  84.37959
Epoch:  140    Loss:  83.44133
Epoch:  150    Loss:  82.60386
Epoch:  160    Loss:  81.85324
Epoch:  170    Loss:  81.23454
Epoch:  180    Loss:  80.68679
Epoch:  190    Loss:  80.16555
Epoch:  200    Loss:  79.67953
```

This looks like it worked pretty well! It also should have fulfilled its purpose: Showing what you can achieve using `torch` tensors alone. In case you didn't feel like going through the backprop logic with too much enthusiasm, don't worry: In the next installment, this will get significantly less cumbersome. See you then!