Lab 5

Hüseyin Anil Gündüz

2022-05-31

Welcome to the fifth lab. We will first implement a simple scalar automatic differentiation engine to compute partial derivatives for us, then do a theoretical exercise about L2 regularization.

Exercise 1

Modern deep learning frameworks compute gradients automatically, so that you only need to define how to perform the forward pass in your code. Under the hood, the framework constructs a computational graph based on the operations you used. For example, consider the node:

$$4xy + e^{-y} \tag{1}$$

It can be translated into a graph that looks like this:

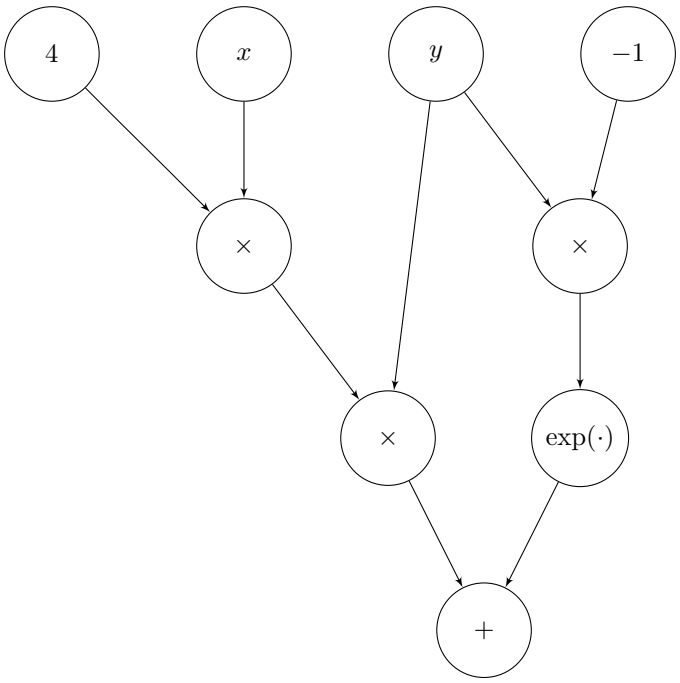


Figure 1: Computational graph representing Eq. 1.

Where we have "leaf" nodes at the top for variables and constants, and "internal" nodes for operations. To make things simpler, in this exercise we will only work with scalar operations and scalar variables, but what we are going to create could, in principle, be extended to work with vectors and matrices. Section 6 of chapter 5 of the *Mathematics for Machine Learning* book (https://mml-book.github.io/) is a good supplementary read.

A node is represented as a list with an attribute `op` to indicate what kind of node it is. Leaf nodes are denoted as `op="const"` or `op="var"`, with attributes to indicate their value or name. Internal nodes have `op` set to the operation they represent, and one or two attributes `x` and `y` to denote their argument(s):

- `op="sum"` for Addition $x + y$
- `op="sub"` for Subtraction $x - y$
- `op="mul"` for Product $x \cdot y$
- `op="div"` for Division x/y
- `op="exp"` for Exponentiation e^x
- `op="tanh"` for Hyperbolic tangent $\tanh(x)$
- `op="log"` for Logarithm $\log(x)$

We first define some utility functions to easily create nodes:

```
# node representing a constant
n.const = function(value) list(op = "const", value = value)

# node representing a variable
n.var = function(name) list(op = "var", name = name)

# nodes for binary operations
n.sum = function(x, y) list(op = "sum", x = x, y = y)
n.sub = function(x, y) list(op = "sub", x = x, y = y)
n.mul = function(x, y) list(op = "mul", x = x, y = y)
n.div = function(x, y) list(op = "div", x = x, y = y)

# nodes for functions
n.exp = function(x) list(op = "exp", x = x)
n.log = function(x) list(op = "log", x = x)
n.tanh = function(x) list(op = "tanh", x = x)

# now define the graph computing Eq. 1 and show in Fig. 1
x = n.var("x")
y = n.var("y")

z = n.sum(
  n.mul(
     n.mul(
        n.const(4),
        x),
     y),
  n.exp(
     n.mul(
        n.const(-1),
        y)))

z
```

```
## $op
## [1] "sum"
##
## $x
## $x$op
## [1] "mul"
##
## $x$x
## $x$x$op
## [1] "mul"
##
## $x$x$x
## $x$x$x$op
## [1] "const"
##
## $x$x$x$value
```

```
## [1] 4
##
##
## $x$x$y
## $x$x$y$op
## [1] "var"
##
## $x$x$y$name
## [1] "x"
##
##
##
## $x$y
## $x$y$op
## [1] "var"
##
## $x$y$name
## [1] "y"
##
##
##
## $y
## $y$op
## [1] "exp"
##
## $y$x
## $y$x$op
## [1] "mul"
##
## $y$x$x
## $y$x$x$op
## [1] "const"
##
## $y$x$x$value
## [1] -1
##
##
## $y$x$y
## $y$x$y$op
## [1] "var"
##
## $y$x$y$name
## [1] "y"
```

This structure of nested lists contains the computational graph for the node above. Now, we can write code to manipulate this expression as we please. In the course of this exercise, we will see how:

1. Print an expression,
2. Compute its value, given the values of the variables involved,
3. Differentiate it to automatically find partial derivatives with respect to any given variable,
4. Transform it into simpler expressions that are cheaper to handle, and
5. Write code to train a neural network without getting our hands dirty with derivatives ever again.

Printing an expression

First, since it is quite hard to understand the node from the representation above, let us write a function to convert a computational graph into a string representation that is easier to understand. For example, the expression $x + 2y$ should be converted to

```
c("(", "x", "+", "(", "2", "*", "y", ")", ")")
```

Which can be printed easily using `cat`, resulting in `( x + ( 2 * y ) )`.

Such a function should be *recursive*. This means that when simplifying a complicated expression it will call itself on each constituting piece of that expression, and "assemble" the results together. Conceptually, the procedure is similar to the factorial operation, which is recursively defined in terms of the factorial of a smaller number:

$$n! = \begin{cases} 1 & \text{if } n < 1 \\ n \cdot (n-1)! & \text{otherwise} \end{cases} \tag{2}$$

This definition can be converted into R as:

```
factorial = function(n) {
  if(n < 1) {
    1
  }
  else {
    n * factorial(n - 1)
  }
}

factorial(4)
```

```
## [1] 24
```

In a similar way, the function `to.string` below should call itself to "stringify" the operands on an operation, then merge these representations into a single string for the whole operation. This basic skeleton of recursively navigating the computational graph will be used throughout the exercise.

```
to.string = function(node) {
  # return a vector of strings representing each parts of the node

  if(node$op == "const") {
    # the string representation of a constant is its value
    c(node$value)
  }
  else if(node$op == "var") {
    # the string representation of a variable is its name
    c(node$name)
  }
  else if(node$op %in% c("sum", "sub", "mul", "div")) {
    # find the string representation of the operation
    operator = list(
      sum = "+", sub = "-", mul = "*", div = "/"
    )[[node$op]]

    string_x = (
      to.string(node$x)
    )

    string_y = (
      to.string(node$y)
    )

    c(
      "(",
      string_x, operator, string_y,
```

```
      ")"
    )
  }
  else if(node$op %in% c("tanh", "exp", "log")) {
    c(
      node$op, "(", to.string(node$x), ")"
    )
  }
  else {
    stop(c("unknown node: ", node$op))
  }
}

print.node = function(node) {
  cat(to.string(node), "\n")
}

print.node(z)
```

```
## ( ( ( 4 * x ) * y ) + exp ( ( -1 * y ) ) )
```

This is much simpler to read!

Computing the value of an expression

We can now write a function to compute the value of an expression given values for the variables. This function should be recursive too, like `to.string` above.

```
compute = function(node, var_values) {
  # compute the numerical result of the node using the provided variable values

  if(node$op == "const") {
    # the value of a constant is its value
    node$value
  }
  else if(node$op == "var") {
    # read the value of the variable from the list
    val = var_values[[node$name]]
    if(is.null(val)) {
      stop(c("value not defined or NULL for variable ", node$name))
    }
    else {
      val
    }
  }
  else if(node$op == "sum") {
    value_x = (
      compute(node$x, var_values)
    )

    value_y = (
      compute(node$y, var_values)
    )

    # add the values and return the result
    value_x + value_y
  }
  else if(node$op == "sub") {
    compute(node$x, var_values) - compute(node$y, var_values)
```

```
  }
  else if(node$op == "mul") {
    compute(node$x, var_values) * compute(node$y, var_values)
  }
  else if(node$op == "div") {
    compute(node$x, var_values) / compute(node$y, var_values)
  }
  else if(node$op == "tanh") {
    tanh(compute(node$x, var_values))
  }
  else if(node$op == "exp") {
    exp(compute(node$x, var_values))
  }
  else if(node$op == "log") {
    log(compute(node$x, var_values))
  }
  else {
    stop(c("unknown node: ", node$op))
  }
}

compute(z, list(x = 2, y = 3))
```

```
## [1] 24.04979
```

The result that we expect is, of course:

```
4 * 2 * 3 + exp(-3)
```

```
## [1] 24.04979
```

Differentiating an expression

We can finally see how to differentiate an expression with respect to a variable. We do this again through a recursive function that differentiates each argument and merges the result. Note that this function should return a new computational graph that contains the operations necessary to compute the partial derivative we are interested in.

Remember to use the chain rule where appropriate!

```
differentiate = function(node, variable) {
  # differentiate the given expression with respect to the given variable
  #
  # VERY IMPORTANT: this function returns a graph, which can only contain nodes.
  # Therefore, you must use the functions n.const, n.sum, etc., instead of normal
  # numbers and operations.

  if(node$op == "const") {
    # derivative of a constant is always zero
    n.const(0)
  }
  else if(node$op == "var") {
    if(node$name == variable) {
      # derivative is one if we are differentiating with respect to this variable
      n.const(1)
    }
    else {
      # or zero if we are differentiating with respect to a different variable
      n.const(0)
    }
```

```
  }
  # call the right function depending on what type of node we are processing
  else if(node$op == "sum") {
    differentiate.sum(node, variable)
  }
  else if(node$op == "sub") {
    differentiate.sub(node, variable)
  }
  else if(node$op == "mul") {
    differentiate.mul(node, variable)
  }
  else if(node$op == "div") {
    differentiate.div(node, variable)
  }
  else if(node$op == "tanh") {
    differentiate.tanh(node, variable)
  }
  else if(node$op == "exp") {
    differentiate.exp(node, variable)
  }
  else if(node$op == "log") {
    differentiate.log(node, variable)
  }
  else {
    stop(c("unknown node: ", node$op))
  }
}

differentiate.sum = function(node, variable) {
  diff_x = (
    differentiate(node$x, variable)
  )
  diff_y = (
    differentiate(node$y, variable)
  )

  # return a new node that sums the derivatives of the left and right part
  #
  # note that we are returning a new graph node that connects the two
  # graphs representing the derivatives of the left and right parts
  n.sum(diff_x, diff_y)
}

differentiate.sub = function(node, variable) {
  n.sub(
     differentiate(node$x, variable),
     differentiate(node$y, variable))
}

differentiate.mul = function(node, variable) {
  n.sum(
    n.mul(differentiate(node$x, variable), node$y),
    n.mul(node$x, differentiate(node$y, variable))
  )
}

differentiate.div = function(node, variable) {
```

```
  # we differentiate only once and re-use this part of the graph
  dy = differentiate(node$y, variable)

  n.div(
    n.sub(
       n.mul(
          differentiate(node$x, variable),
          node$y),
       n.mul(
          node$x,
          dy)),
    n.mul(
       node$y,
       node$y))
}

differentiate.tanh = function(node, variable) {
  # (1 - tanh(x)^2) * differentiate(x)
  tx = n.tanh(node$x)
  n.mul(
    n.sub(
       n.const(1),
       n.mul(
          tx,
          tx)),
    differentiate(node$x, variable))
}

differentiate.exp = function(node, variable) {
  n.mul(
     n.exp(node$x),
     differentiate(node$x, variable))
}


differentiate.log = function(node, variable) {
  # (1 / x) * differentiate(x)
   n.div(
      differentiate(node$x, variable),
      node$x)
}

dz = differentiate(z, "x")
print.node(dz)
```

```
## ( ( ( ( ( 0 * x ) + ( 4 * 1 ) ) * y ) + ( ( 4 * x ) * 0 ) ) + ( exp ( ( -1 * y ) ) * ( ( 0 * y )
```

This looks a bit complicated, but by applying some trivial simplifications we see it is correct:

$$
\begin{aligned}
&(((((0 \cdot x) + (4 \cdot 1)) \cdot y) + ((4 \cdot x) \cdot 0)) + (exp((-1 \cdot y)) \cdot ((0 \cdot y) + (-1 \cdot 0)))) \\
&\quad = (((0 + 4) \cdot y) + 0) + (exp((-1 \cdot y)) \cdot (0 + 0))) \\
&\quad = (4 \cdot y) + (exp((-1 \cdot y)) \cdot 0)) \\
&\quad = (4 \cdot y) + 0 \\
&\quad = 4 \cdot y \\
&\quad = \frac{\mathrm{d}}{\mathrm{d}x} \left(4xy + e^{-y}\right)
\end{aligned}
$$

These simplification rules are trivial arithmetic identities:

- $0 + x = x$
- $0 \cdot x = 0$
- $1 \cdot x = x$
- $0/x = 0$

Let us write a function that uses these identities to automatically simplify `dz` in the same way we just did. As with differentiation, this function should return a new computational graph.

```
is.zero = function(node) {
  # returns TRUE iff the node is the constant "0"
  node$op == "const" && node$value == 0
}

is.one = function(node) {
  # returns TRUE iff the node is the constant "1"
  node$op == "const" && node$value == 1
}

simplify = function(node) {
  # simplifies the provided node, returning a new computational graph

  if(node$op %in% c("const", "var")) {
    # constants and variables cannot be simplified
    node
  }
  # call the right function depending on what type of node we are processing
  else if(node$op == "sum") {
    simplify.sum(node)
  }
  else if(node$op == "sub") {
    simplify.sub(node)
  }
  else if(node$op == "mul") {
    simplify.mul(node)
  }
  else if(node$op == "div") {
    simplify.div(node)
  }
  else if(node$op == "tanh") {
    simplify.tanh(node)
  }
  else if(node$op == "exp") {
    simplify.exp(node)
  }
  else if(node$op == "log") {
    simplify.log(node)
  }
  else {
    stop(c("unknown node: ", node$op))
  }
}

simplify.sum = function(node) {
  simple_x = (
    simplify(node$x)
  )
  simple_y = (
```

```
    simplify(node$y)
  )

  if(is.zero(simple_x)) {
    # rule: 0 + y = y
    simple_y
  }
  else if(is.zero(simple_y)) {
    # rule: x + 0 = x
    simple_x
  }
  else if(simple_x$op == "const" && simple_y$op == "const") {
    # if both arguments are constants we can perform the sum immediately
    n.const(simple_x$value + simple_y$value)
  }
  else {
    # cannot simplify further; return a new sum node with the simplified operands
    n.sum(simple_x, simple_y)
  }
}

simplify.sub = function(node) {
  sx = simplify(node$x)
  sy = simplify(node$y)

  if(is.zero(sx)) {
    # 0 - y = -1 * y
    n.mul(n.const(-1), sy)
  }
  else if(is.zero(sy)) {
    # x - 0 = 0
    sx
  }
  else if(sx$op == "const" && sy$op == "const") {
    # perform the operation if possible
    const(sx$value - sy$value)
  }
  else {
    # cannot simplify further
    n.sub(sx, sy)
  }
}

simplify.mul = function(node) {
  sx = simplify(node$x)
  sy = simplify(node$y)

  if(is.zero(sx) || is.zero(sy)) {
    # 0 * y = x * 0 = 0
    n.const(0)
  }
  else if(is.one(sx)) {
    # 1 * y = y
    sy
  }
  else if(is.one(sy)) {
    # x * 1 = x
```

```
    sx
  }
  else if(sx$op == "const" && sy$op == "const") {
    # perform the operation if possible
    n.const(sx$value * sy$value)
  }
  else {
    # cannot simplify further
    n.mul(sx, sy)
  }
}

simplify.div = function(node) {
  sx = simplify(node$x)
  sy = simplify(node$y)

  if(is.zero(sx)) {
    # 0 / y = 0 (even when y = 0)
    n.const(0)
  }
  else if(is.zero(sy)) {
    # cannot do x / 0
    stop("division by zero")
  }
  else if(is.one(sy)) {
    # x / 1 = x
    sx
  }
  else if(sx$op == "const" && sy$op == "const") {
    # perform the operation if possible
    n.const(sx$value / sy$value)
  }
  else {
    # cannot simplify further
    n.div(sx, sy)
  }
}

simplify.tanh = function(node) {
  sx = simplify(node$x)
  if(is.zero(sx)) {
    # tanh(0) = 0
    n.const(0)
  }
  else if(sx$op == "const") {
    # perform the operation if possible
    n.const(tanh(sx$value))
  }
  else {
    # cannot simplify further
    n.tanh(sx)
  }
}

simplify.exp = function(node) {
  sx = simplify(node$x)
  if(is.zero(sx)) {
    # exp(0) = 1
```

```
    n.const(1)
  }
  else if(sx$op == "const") {
    # perform the operation if possible
    n.const(exp(sx$value))
  }
  else {
    # cannot simplify further
    n.exp(sx)
  }
}

simplify.log = function(node) {
  sx = simplify(node$x)
  if(sx$op == "const" && sx$value <= 0) {
    # cannot compute log(x) for x <= 0
    stop("logarithm of non-positive number")
  }
  else if(sx$op == "const") {
    # perform the operation if possible
    n.const(log(sx$value))
  }
  else {
    # cannot simplify further
    n.log(sx)
  }
}

dz = simplify(dz)
print.node(dz)
```

```
## ( 4 * y )
```

The result matches what we showed above, $4y$. Simplifying the graph with these and other, more advanced tricks, can greatly speed up code.

Now we are also equipped to perform differentiation of any order, for example $\partial z/\partial x \partial y$ is simply:

```
simplify(differentiate(differentiate(z, "x"), "y"))
```

```
## $op
## [1] "const"
##
## $value
## [1] 4
```

Training a network

Let us now define a computational graph that performs the forward pass of a simple network, and use the functions above to compute the gradients of the parameters. We will use the same network we used in the third lab, reproduced below, and, as usual, we will test the code on the five points dataset. Since the functions we have written so far only work with scalar values, we will perform stochastic gradient descent using one sample at a time.

```
# the two input nodes
x1 = n.var("x1")
x2 = n.var("x2")

# parameters for the first hidden neuron
b1 = n.var("b1")
```

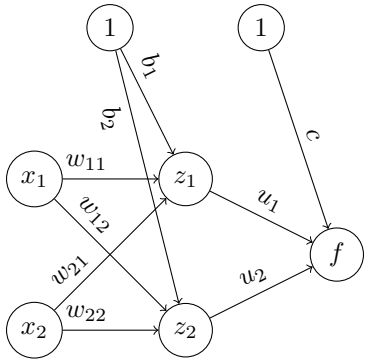


Figure 2: Structure of the neural network.

```
w11 = n.var("w11")
w21 = n.var("w21")

# compute the output of the first hidden neuron
z1in = n.sum(
    b1,
    n.sum(
       n.mul(x1, w11),
       n.mul(x2, w21)))

z1out = n.tanh(z1in)

print.node(z1out)
```

```
## tanh ( ( b1 + ( ( x1 * w11 ) + ( x2 * w21 ) ) ) )
```

Now, complete the remaining part of the network:

```
b2 = n.var("b2")
w12 = n.var("w12")
w22 = n.var("w22")

z2out = (
  n.tanh(
    n.sum(
      b2,
      n.sum(
         n.mul(x1, w12),
         n.mul(x2, w22))))
)

c = n.var("c")
u1 = n.var("u1")
u2 = n.var("u2")

fin = (
  n.sum(
    c,
    n.sum(
      n.mul(z1out, u1),
      n.mul(z2out, u2)))
)

fout = (
```

```
  n.div(
     n.const(1),
     n.sum(
        n.const(1),
        n.exp(
           n.mul(
              n.const(-1),
              fin))))
)

print.node(fout)
```

```
## ( 1 / ( 1 + exp ( ( -1 * ( c + ( ( tanh ( ( b1 + ( ( x1 * w11 ) + ( x2 * w21 ) ) ) ) * u1 ) + ( t
```

And this defines the forward pass.

We can now compute the predictions of the network by evaluating `fout`, providing values for the inputs and weights. For example:

```
compute(z1out, list(
  # values for weights and biases
  b1 = 1.543385, w11 = 3.111573, w12 = -2.808800,
  b2 = 1.373085, w21 = 3.130452, w22 = -2.813466,
  c = -4.241453, u1 = 4.036489, u2 = 4.074885,

  # values for the input
  x1 = 1, x2 = -1
))
```

```
## [1] 0.9094797
```

Which should be about 0.9. We now have to compute the cross-entropy loss. For numerical stability, we will compute the loss using f_{in} instead of f_{out}. Therefore, first, show that:

$$-y \cdot \log(f_{out}) - (1-y) \cdot \log(1-f_{out}) = f_{in} - f_{in} \cdot y + \log(1+e^{-f_{in}}) \tag{3}$$

Solution:

$$\begin{aligned}
&-y \cdot \log(f_{out}) - (1-y) \cdot \log(1-f_{out}) && (4)\\
&= -y \cdot \log \frac{1}{1+e^{-f_{in}}} - (1-y) \cdot \log\left(1 - \frac{1}{1+e^{-f_{in}}}\right) && (5)\\
&= -y \cdot -\log\left(1+e^{-f_{in}}\right) - (1-y) \cdot \left(-f_{in} - \log\left(1+e^{-f_{in}}\right)\right) && (6)\\
&= y \cdot \log\left(1+e^{-f_{in}}\right) + f_{in} + \log\left(1+e^{-f_{in}}\right) - y \cdot f_{in} - y \cdot \log\left(1+e^{-f_{in}}\right) && (7)\\
&= f_{in} - f_{in} \cdot y + \log(1+e^{-f_{in}}) && (8)
\end{aligned}$$

```
# this variable contains the label for the sample the network is predicting
y = n.var("y")

loss = (
  n.sum(
     n.sub(
        fin,
        n.mul(
           fin,
           y)),
     n.log(
        n.sum(
```

```
            n.const(1),
            n.exp(
               n.mul(
                  n.const(-1),
                  fin)))))
)

print.node(loss)
```

```
## ( ( ( c + ( ( tanh ( ( b1 + ( ( x1 * w11 ) + ( x2 * w21 ) ) ) ) * u1 ) + ( tanh ( ( b2 + ( ( x1 *
```

This is starting to look complicated! Luckily, this time, we do not have to get our hands dirty with derivatives; let us find the graphs for the derivatives of each parameter of the network

```
param_names = c("b1", "w11", "w12", "b2", "w21", "w22", "c", "u1", "u2")

gradient_graphs = lapply(param_names, function(p) {
  # each item contains a computational graph that computes
  # the gradient of the loss with respect to a parameter
  simplify(differentiate(loss, p))
})

names(gradient_graphs) = param_names

print.node(gradient_graphs$w11)
```

```
## ( ( ( ( ( 1 - ( tanh ( ( b1 + ( ( x1 * w11 ) + ( x2 * w21 ) ) ) ) * tanh ( ( b1 + ( ( x1 * w11 )
```

As you can see, there is a great deal of repetition in this expression. The repetitions could be removed by storing, in each node, its current value and gradient, so that we would not need to re-compute them every time. Modern deep learning frameworks indeed do this, and are able to compute the gradient of the loss with respect to all parameters in a single pass, but here we accept these inefficiencies for the sake of simplicity.

We are now ready to train this network:

```
# dataset
data.x1 = c(0, 1, 0, -1, 0)
data.x2 = c(0, 0, -1, 0, 1)
data.y = c(1, 0, 0, 0, 0)

# Glorot initializtion for the parameters
b = sqrt(6 / 4)
values = as.list(sapply(param_names, function(p) {
  if(p %in% c("b1", "b2", "c")) {
    0.0
  }
  else {
    runif(1, -b, b)
  }
}))

# training loop
losses = list()
for(e in 0:250) {
  epoch_loss = 0.0

  for(j in 1:5) {
    # set the correct values for the inputs and label
    values$x1 = data.x1[j]
    values$x2 = data.x2[j]
```

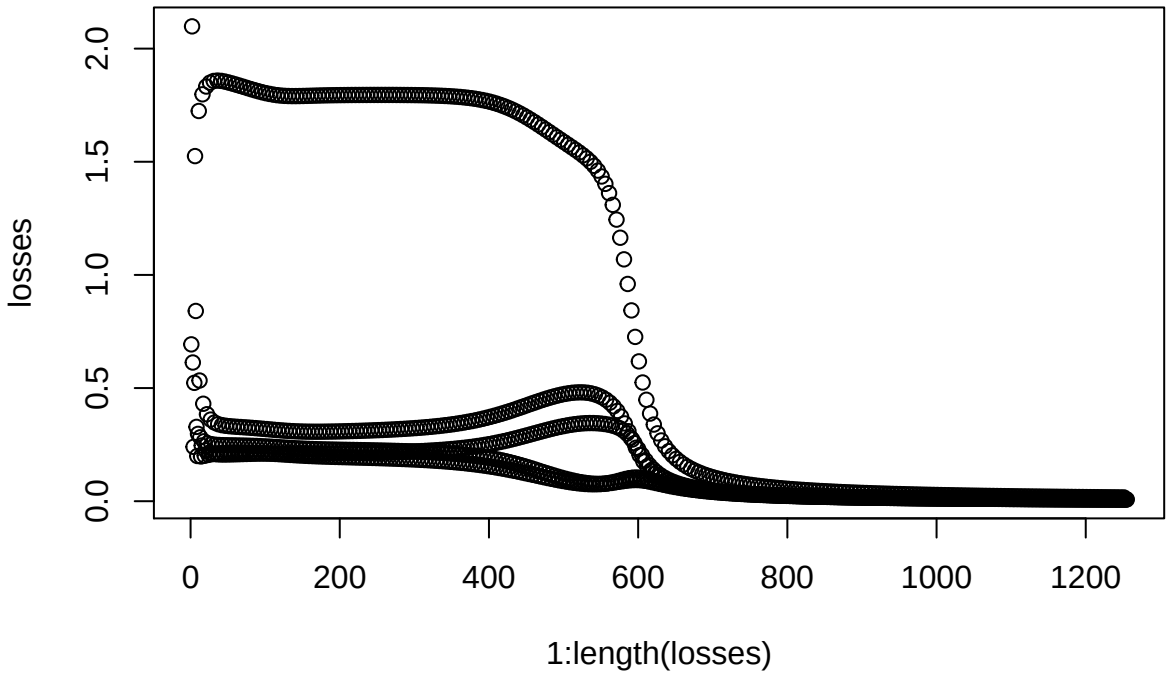
```
    values$y = data.y[j]

    losses[[e * 5 + j]] = (
      compute(loss, values)
    )

    gradients = sapply(param_names, function(p) {
      compute(gradient_graphs[[p]], values)
    })

    values = as.list(sapply(param_names, function(p) {
      values[[p]] - 0.5 * gradients[[p]]
    }))
  }
}

stopifnot(mean(unlist(tail(losses))) < 0.05) # convergence check (sometimes fails)
plot(1:length(losses), losses)
```



You can clearly see how the loss of each individual training sample evolves over time. This also explains the "saddle" you might have noticed in the loss curve from the previous lab.

And these are the predictions for the five points:

```
for(j in 1:5) {
  values$x1 = data.x1[j]
  values$x2 = data.x2[j]
  values$y = data.y[j]

  pred = compute(fout, values)
  cat("Sample", j, "-", "label:", data.y[j], "- predicted: ", pred, "\n")
}
```

```
## Sample 1 - label: 1 - predicted:  0.9834447
## Sample 2 - label: 0 - predicted:  0.007633603
## Sample 3 - label: 0 - predicted:  0.007548883
## Sample 4 - label: 0 - predicted:  0.007641578
## Sample 5 - label: 0 - predicted:  0.007750374
```

Conclusion

What we did in this exercise is (a simplification of) how deep learning frameworks evaluate the code you write. You only need to define how to compute the output of the network, and the framework figures out the necessary gradients on its own. They provide a much better user interface, allowing you to use `+`, `-`, `/`, `*` etc. as you normally would instead of the clumsy node constructors we defined here, but there is always a computational graph hidden behind the curtains.

Exercise 2

This exercise should improve your understanding of weight decay (or L2 regularization).

1. Consider a quadratic error function $E(\mathbf{w}) = E_0 + \mathbf{b}^T\mathbf{w} + 1/2 \cdot \mathbf{w}^T\mathbf{H}\mathbf{w}$ and its regularized counterpart $E'(\mathbf{w}) = E(\mathbf{w}) + \tau/2 \cdot \mathbf{w}^T\mathbf{w}$, and let $\mathbf{w}^*$ and $\tilde{\mathbf{w}}$ be the minimizers of E and E' respectively. We want to find a node to express $\tilde{\mathbf{w}}$ as a function of $\mathbf{w}^*$, i.e. find the displacement introduced by weight decay.
   - Find the gradients of E and E'. Note that, at the global minimum, we have $\nabla E(\mathbf{w}^*) = \nabla E'(\hat{\mathbf{w}}) = 0$.
   - In the equality above, express $\mathbf{w}^*$ and $\tilde{\mathbf{w}}$ as a linear combination of the eigenvectors of $\mathbf{H}$.
   - Through algebraic manipulation, obtain $\tilde{\mathbf{w}}_i$ as a function of $\mathbf{w}_i^*$.
   - Interpret this result geometrically.
   - Note: $\mathbf{H}$ is square, symmetric, and positive definite, which means that its eigenvectors are pairwise orthogonal and its eigenvalues are positive (spectral theorem).
2. Consider a linear network of the form $y = \mathbf{w}^T\mathbf{x}$ and the mean squared error as a loss function. Assume that every observation is corrupted with Gaussian noise $\epsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$. Compute the expectation of the gradient under ϵ and, show that adding gaussian noise to the inputs has the same effect of weight decay.

Solution

Question 1 The error is computed as:

$$E(\mathbf{w}) = E_0 + \sum_i w_i b_i + \frac{1}{2}\sum_i\sum_j w_i w_j h_{ij}$$

The derivative with respect to w_i is, then:

$$\frac{\partial E}{\partial w_i} = b_i + \sum_j w_j h_{ij}$$

Where the factor $1/2$ was removed since the pair w_i and w_j is multiplied together twice, and $h_{ij} = h_{ji}$. In vector form:

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w}$$

The same reasoning applied to E' yields:

$$\nabla_{\mathbf{w}} E'(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w} + \tau\mathbf{w}$$

Now let $\mathbf{u}_i$ and λ_i be the eigenvectors and eigenvalues of $\mathbf{H}$, so that $\mathbf{H}\mathbf{u}_i = \lambda_i\mathbf{u}_i$. Any vector $\mathbf{v}$ can then be expressed as $\mathbf{v} = \sum_i \gamma_i\mathbf{u}_i$. Now, note that

$$\mathbf{H}\mathbf{v} = \sum_i \gamma_i\mathbf{H}\mathbf{u}_i = \sum_i \gamma_i\lambda_i\mathbf{u}_i$$

Moreover, at the global minimum, both gradients equal zero, hence:

$$\underbrace{\mathbf{b} + \sum_i \alpha_i \lambda_i \mathbf{u}_i}_{\mathbf{H}\mathbf{w}^*} = \underbrace{\mathbf{b} + \sum_i \beta_i \lambda_i \mathbf{u}_i}_{\mathbf{H}\tilde{\mathbf{w}}} + \tau \underbrace{\sum_i \beta_i \mathbf{u}_i}_{\tilde{\mathbf{w}}} \iff \sum_i \left(\alpha_i \lambda_i - \beta_i \lambda_i - \tau \beta_i\right) \mathbf{u}_i = \mathbf{0}$$

Since the eigenvectors are linearly independent, the above expression is zero only when each term inside the sum is zero, i.e.

$$\alpha_i \lambda_i - \beta_i \lambda_i - \tau \beta_i = 0 \iff \beta_i = \frac{\lambda_i}{\lambda_i + \tau} \alpha_i$$

Now, by replacing this into the expression for $\hat{\mathbf{w}}$, we get:

$$\tilde{\mathbf{w}} = \beta^T \mathbf{u} = \sum_i \beta_i \mathbf{u}_i = \sum_i \frac{\lambda_i}{\lambda_i + \tau} \alpha_i \mathbf{u}_i$$

The eigenvalues of $\mathbf{H}$ indicate how much the error changes by moving in the direction of the corresponding eigenvector, with larger changes associated to smaller eigenvalues. In light of this, the node above is saying that the largest changes are applied to the weights that have little influence on the error, while "important" weights are not perturbed much.

Question 2 The prediction for $\tilde{\mathbf{x}} = \mathbf{x} + \epsilon$ is:

$$\tilde{y} = \mathbf{w}^T \left(\mathbf{x} + \epsilon\right) = \mathbf{w}^T \mathbf{x} + \mathbf{w}^T \epsilon$$

The error of this sample is

$$\tilde{E} = \frac{1}{2} \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right)^2$$

And its gradient with respect to a single weight is

$$\begin{aligned} \frac{\partial \tilde{E}}{\partial w_i} &= \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right) \left(-x_i - \epsilon_i\right) \\ &= -x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right) - \epsilon_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right) \end{aligned}$$

The expectation with respect to ϵ is

$$\begin{aligned} \mathbb{E}\left[\frac{\partial \tilde{E}}{\partial w_i}\right] &= \mathbb{E}\left[-x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right)\right] + \mathbb{E}\left[-\epsilon_i \left(\hat{y} - \mathbf{w}^T \mathbf{x} - \mathbf{w}^T \epsilon\right)\right] \\ &= -x_i \left(\hat{y} - \mathbf{w}^T \mathbf{x}\right) + \mathbb{E}\left[-\epsilon_i \hat{y} + \epsilon_i \mathbf{w}^T \mathbf{x} + \epsilon_i \mathbf{w}^T \epsilon\right] \\ &\overset{*}{=} \frac{\partial E}{\partial w_i} + \sum_j w_j \mathbb{E}\left[\epsilon_i \epsilon_j\right] \\ &= \frac{\partial E}{\partial w_i} + w_i \sigma^2 \end{aligned}$$

Where we used $\partial E / \partial w_i$ to denote the gradient of the error of the de-noised sample, and the step marked with $*$ follows because $\mathbb{E}[\epsilon_i \epsilon_j] = \mathrm{Cov}\left[\epsilon_i, \epsilon_j\right] = \delta_{ij} \sigma^2$.

Clearly, the gradient is the same that results from weight decay.