Lab 3

Emilio Dorigatti

2020-11-20

Welcome to the third lab. The first exercise is an implementation of gradient descent on a bivariate function. The second exercise is about computing derivatives of the weights of a neural network, and the third exercise combines the previous two.

Exercise 1

This exercise is about gradient descent. We will use the function $f(x_1, x_2) = (x_1 - 6)^2 + x_2^2 - x_1 x_2$ as a running example:

1. Use pen and paper to do three iterations of gradient descent:
   - Find the gradient of f;
   - Start from the point $x_1 = x_2 = 6$ and use a step size of $1/2$ for the first step, $1/3$ for the second step and $1/4$ for the third step;
   - What will happen if you keep going?
2. Write a function that performs gradient descent:
   - For simplicity, we use a constant learning rate.
   - Can you find a way to prematurely stop the optimization when you are close to the optimum?

```
func.value = function(x) {
  # TODO compute the value of f at x
}

func.gradient = function(x) {
  c(
    # TODO compute the gradienf of f at x
  )
}

func.value(c(6, 6))
func.gradient(c(6, 6))
```

Does it match what you computed?

```
gradient_descent_optimizer = function(x0, func, grad, max_steps, alpha) {
  # x0 is the initial point
  # func computes the value of the function at a given point
  # grad computes the gradient of the function at a given point
  # max_steps is the maximum number of gradient descent steps
  # alpha is the learning rate

  # TODO use a for loop to do gradient descent
}

gradient_descent_optimizer(c(6, 6), func.value, func.gradient, 10, 0.1)
```

Play a bit with the starting point and learning rate to get a feel for its behavior; how close can you get to the minimum?

Exercise 2

This exercise is about computing gradients with the chain rule, with pen and paper. We will work with a neural network with a single hidden layer with two neurons and an output layer with one neuron (see Figure 1).

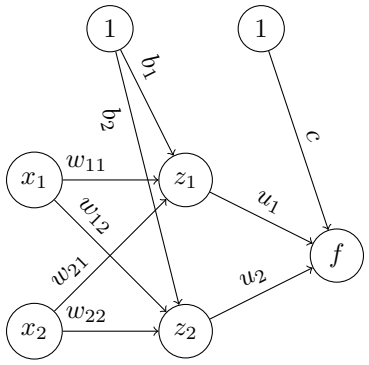

Figure 1: Neural network used in Exercise 2.

The neurons in the hidden layer use the tanh activation, while the output neuron uses the sigmoid. The loss used in binary classification is the *binary cross-entropy*:

$$\mathcal{L}(y, f_{out}) = -y \log f_{out} - (1 - y) \log(1 - f_{out})$$

where $y \in \{0, 1\}$ is the true label and $f_{out} \in (0, 1)$ is the predicted probability that $y = 1$.

1. Compute $\partial \mathcal{L}(y, f_{out}) / \partial f_{out}$
2. Compute $\partial f_{out} / \partial f_{in}$
3. Show that $\partial \sigma(x) / \partial x = \sigma(x)(1 - \sigma(x))$
4. Show that $\partial \tanh(x) / \partial x = 1 - \tanh(x)^2$ (Hint: $\tanh(x) = (e^x - e^{-x})(e^x + e^{-x})^{-1}$)
5. Compute $\partial f_{in} / \partial c$
6. Compute $\partial f_{in} / \partial u_1$
7. Compute $\partial \mathcal{L}(y, f_{out}) / \partial c$
8. Compute $\partial \mathcal{L}(y, f_{out}) / \partial u_1$
9. Compute $\partial f_{in} / \partial z_{2,out}$
10. Compute $\partial z_{2,out} / \partial z_{2,in}$
11. Compute $\partial z_{2,in} / \partial b_2$
12. Compute $\partial z_{2,in} / \partial w_{12}$
13. Compute $\partial z_{2,in} / \partial x_1$
14. Compute $\partial \mathcal{L}(y, f_{out}) / \partial b_2$
15. Compute $\partial \mathcal{L}(y, f_{out}) / \partial w_{12}$
16. Compute $\partial \mathcal{L}(y, f_{out}) / \partial x_1$

You will notice that there are lots of redundancies. We will see how to improve these computations in the lecture and in the next lab. Luckily, modern deep learning software computes gradients automatically for you.

Exercise 3

Now that we know how to do gradient descent and how to compute the derivatives of the weights of a simple network, we can try to do these steps together and train our first neural network! We will use the small dataset with five points we studied in the first lab.

First, let's define the 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);
```

Next, a function to compute the output of the network:

```
sigmoid = function(x) {
  # TODO compute the sigmoid on the input
}

nnet.predict = function(params) {
  # params is a vector, we first unpack for clarity
  b1 = params[1]; b2 = params[2];
  w11 = params[3]; w12 = params[4];
  w21 = params[5]; w22 = params[6];
  c = params[7]; u1 = params[8]; u2 = params[9];

  # TODO compute and return the output of the network
}

# this should return the predictions for the five points in the datasets
params = rnorm(9)
nnet.predict(params)
```

Since gradient descent is done on the loss function, we need a function to compute it:

```
nnet.loss = function(params) {
  # TODO compute the predictions and return the average loss
}

nnet.loss(params)
```

Now, we need to compute the gradient of each parameter:

```
nnet.gradient = function(params) {
  # params is a vector, we first unpack for clarity
  b1 = params[1]; b2 = params[2];
  w11 = params[3]; w12 = params[4];
  w21 = params[5]; w22 = params[6];
  c = params[7]; u1 = params[8]; u2 = params[9];

  # TODO compute all the partial derivatives

  # return the derivatives in the same order as the parameters vector
  c(
    dL_db1, dL_db2,
    dL_dw11, dL_dw12,
    dL_dw21, dL_dw22,
    dL_dc, dL_du1, dL_du2
  )
}

nnet.gradient(params)
```

Finite differences are a useful way to check that the gradients are computed correctly:

```
# first, compute the analytical gradient of the parameters
gradient = nnet.gradient(params);

eps = 1e-9
for(i in 1:9) {
  # compute loss when subtracting eps to parameter i
  neg_params = c(params);
  neg_params[i] = neg_params[i] - eps;
  neg_value = nnet.loss(neg_params);
```

```
  # compute loss when adding eps to parameter i
  pos_params = c(params);
  pos_params[i] = pos_params[i] + eps;
  pos_value = nnet.loss(pos_params);

  # compute the "empirical" gradient of parameter i
  fdiff_gradient = mean((pos_value - neg_value) / (2 * eps));

  # error if difference is too large
  stopifnot(abs(gradient[i] - fdiff_gradient) < 1e-5);
}

print("Gradients are correct!")
```

We can finally train our network. Since the network is so small compared to the dataset, the training procedure is very sensitive to the way the weights are initialized and the step size used in gradient descent.

Try to play around with the learning rate and the random initialization of the weights and find reliable values that make training successful in most cases.

```
min_loss = 10
best_params = NULL

for(i in 1:10) {
  params = rnorm(9, sd = 1)
  optimized_params = gradient_descent_optimizer(
    params, nnet.loss, nnet.gradient, max_steps = 100, alpha = 1
  )
  final_loss = nnet.loss(optimized_params)
  cat("Loss", final_loss, ifelse(final_loss < 0.1, "*", ""), "\n")

  if(final_loss < min_loss) {
    min_loss = final_loss
    best_params = optimized_params
  }
}
```

We can use the function in the previous lab to visualize the decision boundary of the best network:

```
library(scales)
library(ggplot2)

data = as.matrix(expand.grid(
  x0 = 1:1,
  x1 = seq(-2, 2, 1 / 25),
  x2 = seq(-2, 2, 1 / 25)
))

data.x1 = data[,2]
data.x2 = data[,3]

plot_grid = function(predictions) {
  # plots the predicted value for each point on the grid;
  # the predictions should have one column and
  # the same number of rows (10,201) as the data
  df = cbind(as.data.frame(data), y = predictions)
  ggplot() +
    geom_tile(aes(x = x1, y = x2, fill = y, color = y), df) +
    scale_color_gradient2(low = muted("blue", 70), mid = "white",
                          high = muted("red", 70), limits = c(0, 1),
```

```
                          midpoint = 0.5) +
    scale_fill_gradient2(low = muted("blue", 70), mid = "white",
                         high = muted("red", 70), limits = c(0, 1),
                         midpoint = 0.5) +
    geom_point(aes(x=c(0, 1, 0, -1, 0), y=c(0, 0, -1, 0, 1)))
}

plot_grid(nnet.predict(best_params))
```

Also try to visualize the decision boundary of network with random parameters:

```
plot_grid(nnet.predict(rnorm(9, sd=5)))
```