Lab 4

Emilio Dorigatti

2020-11-20

Welcome to the fourth lab. In this lab, we will derive the backpropagation equations, code the training procedure, and test it on our beloved dataset with five points.

Exercise 1

Consider a neural network with L layers and a loss function $\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})$. Call the output of the i-th unit of the ℓ-th layer $\mathbf{z}^{(\ell)}_{i,out} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}_{i,in})$ with $\mathbf{z}^{(\ell)}_{i,in} = \sum_j \mathbf{W}^{(\ell)}_{ji}\mathbf{z}^{(\ell-1)}_{j,out} + \mathbf{b}^{(\ell)}_i$ its pre-activation output. Finally, consider $\delta^{(\ell)}_i = \partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})/\partial\mathbf{z}^{(\ell)}_{i,in}$ the gradient of the loss with respect to the pre-activation outputs of layer ℓ.

Derive the back-propagation algorithm for a network with arbitrary architecture. You might find the results of the previous lab a useful reference, as well as chapter 5 of the book *Mathematics for Machine Learning* (https://mml-book.github.io).

1. Show that

$$\delta^{(L)}_i = \frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{z}^{(L)}_{i,out}} \cdot \sigma'^{(L)}(\mathbf{z}^{(L)}_{i,in}) \tag{1}$$

$$\frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{W}^{(\ell)}_{ji}} = \delta^{(\ell)}_i \cdot \mathbf{z}^{(\ell-1)}_{j,out} \tag{2}$$

$$\frac{\partial\mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out})}{\partial\mathbf{b}^{(\ell)}_i} = \delta^{(\ell)}_i \tag{3}$$

$$\delta^{(\ell-1)}_i = \left(\sum_k \delta^{(\ell)}_k \cdot \mathbf{W}^{(\ell)}_{ik}\right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}^{(\ell-1)}_{i,in}) \tag{4}$$

2. Use vectorized operations (i.e., operations with vectors and matrices) to compute the gradients with respect to a single sample.
3. Extend the vectorized operations to handle data in batches, and show that:

$$\Delta^{(L)} = \nabla_{\mathbf{Z}^{(L)}_{out}}\mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out}) \odot \sigma'^{(L)}(\mathbf{Z}^{(L)}_{in}) \tag{5}$$

$$\nabla_{\mathbf{W}^{(\ell)}}\mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(\ell)}_{out}) = {\mathbf{Z}^{(\ell-1)}_{out}}^T \cdot \Delta^{(\ell)} \tag{6}$$

$$\nabla_{\mathbf{b}^{(\ell)}}\mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out}) = \sum_i {\Delta^{(\ell)}_i}^T \tag{7}$$

$$\Delta^{(\ell-1)} = \Delta^{(\ell)}{\mathbf{W}^{(\ell)}}^T \odot \sigma'^{(\ell-1)}(\mathbf{Z}^{(\ell-1)}_{in}) \tag{8}$$

   where $\Delta^{(\ell)}$, $\mathbf{Y}$ and $\mathbf{Z}^{(\ell)}_{out}$ are matrices whose i-th row contain the respective vectors δ, $\mathbf{y}$ and $\mathbf{z}^{(\ell)}_{\cdot,out}$ for the i-th sample in the batch, and $\odot$ is the element-wise product.

Solution

Question 1 By applying the chain rule, we have, for the last layer:

$$\delta_i^{(L)} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(L)}} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(L)}} \cdot \frac{\partial \mathbf{z}_{i,out}^{(L)}}{\partial \mathbf{z}_{i,in}^{(L)}} = \underbrace{\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(L)}}}_{\text{Loss-dependent}} \cdot \sigma'^{(L)}(\mathbf{z}_{i,in}^{(L)}) \tag{9}$$

Where the first term depends on the loss function. Using the chain rule again, the derivatives of the weights of a generic layer ℓ are:

$$\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{W}_{ji}^{(\ell)}} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{i,in}^{(\ell)}}{\partial \mathbf{W}_{ji}^{(\ell)}} \tag{10}$$

$$= \delta_i^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{W}_{ji}^{(\ell)}} \underbrace{\left(\sum_k \mathbf{W}_{ki}^{(\ell)} \mathbf{z}_{k,out}^{(\ell-1)} + \mathbf{b}_i^{(\ell)} \right)}_{\mathbf{z}_{i,in}^{(\ell)}} \tag{11}$$

$$= \delta_i^{(\ell)} \cdot \mathbf{z}_{j,out}^{(\ell-1)} \tag{12}$$

And, as for the bias:

$$\frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{b}_i^{(\ell)}} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{i,in}^{(\ell)}}{\partial \mathbf{b}_i^{(\ell)}} \tag{13}$$

$$= \delta_i^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{b}_i^{(\ell)}} \underbrace{\left(\sum_k \mathbf{W}_{ki}^{(\ell)} \mathbf{z}_{k,out}^{(\ell-1)} + \mathbf{b}_i^{(\ell)} \right)}_{\mathbf{z}_{i,in}^{(\ell)}} \tag{14}$$

$$= \delta_i^{(\ell)} \tag{15}$$

Finally, the deltas of the previous layer are:

$$\delta_i^{(\ell-1)} = \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,in}^{(\ell-1)}} \tag{16}$$

$$= \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \cdot \frac{\partial \mathbf{z}_{i,out}^{(\ell-1)}}{\partial \mathbf{z}_{i,in}^{(\ell-1)}} \tag{17}$$

$$= \left(\sum_k \frac{\partial \mathcal{L}(\mathbf{y}, \mathbf{z}_{\cdot,out}^{(L)})}{\partial \mathbf{z}_{k,in}^{(\ell)}} \cdot \frac{\partial \mathbf{z}_{k,in}^{(\ell)}}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)}) \tag{18}$$

$$= \left(\sum_k \delta_k^{(\ell)} \cdot \frac{\partial}{\partial \mathbf{z}_{i,out}^{(\ell-1)}} \left(\sum_l \mathbf{W}_{lk}^{(\ell)} \mathbf{z}_{l,out}^{(\ell-1)} + \mathbf{b}_k^{(\ell)} \right) \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)}) \tag{19}$$

$$= \left(\sum_k \delta_k^{(\ell)} \cdot \mathbf{W}_{ik}^{(\ell)} \right) \cdot \sigma'^{(\ell-1)}(\mathbf{z}_{i,in}^{(\ell-1)}) \tag{20}$$

Question 2 The trick to find vectorized formulas is to see how to compute the previous equations all at the same time via matrix multiplication. Always check the dimensionality of the matrices and vectors involved, to make sure the shape of the result matches what it should be.

For the last deltas, each neuron is treated independently from the others, therefore an element-wise multiplication between the two vectors does the job:

$$\begin{aligned} \delta^{(L)} &= \nabla_{\mathbf{z}^{(L)}_{\cdot,out}} \mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out}) \odot \sigma'^{(L)}(\mathbf{z}^{(L)}_{\cdot,in}) \\ (N^{(L)} \times 1) &= (N^{(L)} \times 1) \odot (N^{(L)} \times 1) \end{aligned} \tag{21}$$

Where the second row indicates the dimensionality, rows times columns, of the elements involved.

To compute the gradients for the weights in Eq. 2, we multiply every activation of the previous layer by every delta of the current layer, resulting into a matrix which contains all combinations of $\mathbf{z}^{(\ell)}_{i,out}$ times $\delta^{(\ell)}_j$. This is computed as an "outer product":

$$\begin{aligned} \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out}) &= \mathbf{z}^{(\ell-1)}_{\cdot,out} \cdot \delta^{(\ell)^T} \\ (N^{(\ell-1)} \times N^{(\ell)}) &= (N^{(\ell-1)} \times 1) \cdot (N^{(\ell)} \times 1)^T \end{aligned} \tag{22}$$

The gradient for the biases is easy to compute:

$$\nabla_{\mathbf{b}^{(\ell)}} \mathcal{L}(\mathbf{y}, \mathbf{z}^{(L)}_{\cdot,out}) = \delta^{\ell} \tag{23}$$

Finally, the deltas for the previous layer:

$$\begin{aligned} \delta^{(\ell-1)} &= \left(\mathbf{W}^{(\ell)} \cdot \delta^{(\ell)}\right) \odot \sigma'^{(\ell-1)}(\mathbf{z}^{(\ell-1)}_{\cdot,in}) \\ (N^{(\ell-1)} \times 1) &= \left((N^{(\ell-1)} \times N^{(\ell)}) \cdot (N^{(\ell)} \times 1)\right) \odot (N^{(\ell-1)} \times 1) \end{aligned} \tag{24}$$

Which follows because the sum in Eq. 4 is the dot-product of the i-th row of $\mathbf{W}^{(\ell)}$ with $\delta^{(\ell)}$. Doing this separately for each row results in the matrix-vector multiplication $\mathbf{W}^{(\ell)} \cdot \delta^{(\ell)}$.

Question 3 We now extend these formulas to handle batched data. Vectors become matrices where each row contains the vector for the corresponding sample in the batch:

- The sample labels become a matrix $\mathbf{Y}$, with $\mathbf{Y}_{ij}$ the label for the j-th output of the i-th sample;
- The hidden activations become $\mathbf{Z}^{(\ell)}_{out}$, with $\mathbf{Z}^{(\ell)}_{ij,out}$ the activation of the j-th unit in the ℓ-th layer for the i-th sample;
- The deltas become a matrix $\Delta^{(\ell)}$, where row i contains $\delta^{(\ell)}$ for the i-th example in the batch.

Remember, the first thing you should do to understand these formulas is to think at the dimensionality of the vectors and matrices involved and make sure they match.

The delta for the output layer is:

$$\Delta^{(L)} = \nabla_{\mathbf{Z}^{(L)}_{out}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out}) \odot \sigma'^{(L)}(\mathbf{Z}^{(L)}_{in}) \tag{25}$$

which looks the same as Eq. 21 above, except that now we are using matrices instead of vectors. But the operation is the same: element-wise multiplication.

The gradient with respect to $\mathbf{W}^{(\ell)}$ is a bit more involved to compute, as it includes a three-dimensional tensor: the first dimension is for the samples, the second dimension is for the neurons of the $(\ell-1)$-th layer, and the third dimension for the neurons of the ℓ-th layer. In other words, we are taking the gradients in Eq. 22, which are matrices, for each sample, and "stacking" them one on top of each other to get a "cube" of gradients. The element indexed by i, j, k is the derivative of the loss of the i-th sample in the batch with respect to $\mathbf{W}^{(\ell)}_{jk}$.

$$\left(\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out})\right)_{ijk} = \left(\frac{\partial \mathcal{L}(\mathbf{Y}, \mathbf{Z}^{(L)}_{out})}{\partial \mathbf{W}^{(\ell)}_{jk}}\right)_i = \mathbf{Z}^{(\ell-1)}_{ij,out} \cdot \Delta^{(\ell)}_{ik} \tag{26}$$

To find the gradient of the weights with respect to the whole batch, we need to average this on the first dimension (the samples in the batch) to get the gradient:

$$\frac{\partial \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(L)})}{\partial \mathbf{W}_{jk}^{(\ell)}} = \sum_i \mathbf{Z}_{ij,out}^{(\ell-1)} \cdot \Delta_{ik}^{(\ell)} \tag{27}$$

If you look closely, you should realize that this is just a matrix product. Let's use a simpler notation to make it clear:

$$A_{jk} = \sum_i B_{ij} \cdot C_{ik} = \sum_i \left(B^T\right)_{ji} \cdot C_{ik} \tag{28}$$

Therefore, after much pain:

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(\ell)}) = {\mathbf{Z}_{out}^{(\ell-1)}}^T \cdot \Delta^{(\ell)} \tag{29}$$

The biases are straightforward, we just have to sum over the deltas of each sample:

$$\nabla_{\mathbf{b}^{(\ell)}} \mathcal{L}(\mathbf{Y}, \mathbf{Z}_{out}^{(L)}) = \sum_i {\Delta_i^{(\ell)}}^T \tag{30}$$

Finally, the deltas of the previous layer. From Eq. 4, each element is:

$$\Delta_{ij}^{(\ell-1)} = \left(\sum_k \Delta_{ik}^{(\ell)} \cdot \mathbf{W}_{jk}^{(\ell)} \right) \cdot \sigma'^{(\ell-1)}(\mathbf{Z}_{ij,in}^{(\ell-1)}) \tag{31}$$

The sum is again a matrix product, therefore:

$$\Delta^{(\ell-1)} = \Delta^{(\ell)} {\mathbf{W}^{(\ell)}}^T \odot \sigma'^{(\ell-1)}(\mathbf{Z}_{in}^{(\ell-1)}) \tag{32}$$

Exercise 2

In this exercise, we will code the backpropagation algorithm and apply it to our five-points dataset.

First, let's define a function to quickly create a neural network with layers of given size. It will use tanh activation in the hidden layers and sigmoid for the output layer. Although we will use it for classification, we use the mean squared error loss for a change.

```
# activations, losses, and their gradient
sigmoid = function(x) { 1 / (1 + exp(-x)) }
sigmoid_derivative = function(x) { sigmoid(x) * (1 - sigmoid(x)) }
tanh_derivative = function(x) { 1 - tanh(x)^2 }
mse = function(ytrue, ypred) { mean((ytrue - ypred)^2) }
mse_derivative = function(ytrue, ypred) { 2 * (ypred - ytrue) / length(ytrue) }


nnet.new = function(layer_sizes) {
  # all information about the network is stored in a list
  nnet = list(
    weights = list(),
    biases = list(),
    activations = list(),
    activations_derivatives = list(),
    loss = mse,
    loss_derivative = mse_derivative
  )

  # create random weight matrices and bias vectors
```

```
  last_size = layer_sizes[1]  # the first element is the number of inputs
  for(l in 2:length(layer_sizes)) {
    this_size = layer_sizes[l]

    # weights are initialize using the the famous "Glorot" initialization
    b = sqrt(6 / (this_size + last_size))
    nnet$weights[[l - 1]] = matrix(
      runif(last_size * this_size, -b, b), ncol = this_size
    )

    # biases are initialized to zero
    nnet$biases[[l - 1]] = rep(0, this_size)

    # set the activation
    nnet$activations[[l - 1]] = tanh
    nnet$activations_derivatives[[l - 1]] = tanh_derivative

    last_size = this_size
  }

  # change the output activation to sigmoid
  nnet$activations[[length(nnet$activations)]] = sigmoid
  nnet$activations_derivatives[[length(nnet$activations)]] = sigmoid_derivative

  nnet
}

nnet = nnet.new(c(2, 5, 3, 1))
```

Let us now write the forward pass:

```
nnet.predict = function(nnet, data.x) {
  # data.x is a matrix with samples on rows
  n_layers = length(nnet$weights)
  zout = data.x
  for(l in 1:n_layers) {
    zin = t(t(zout %*% nnet$weights[[l]]) + nnet$biases[[l]])
    zout = nnet$activations[[l]](zin)
  }
  zout
}
```

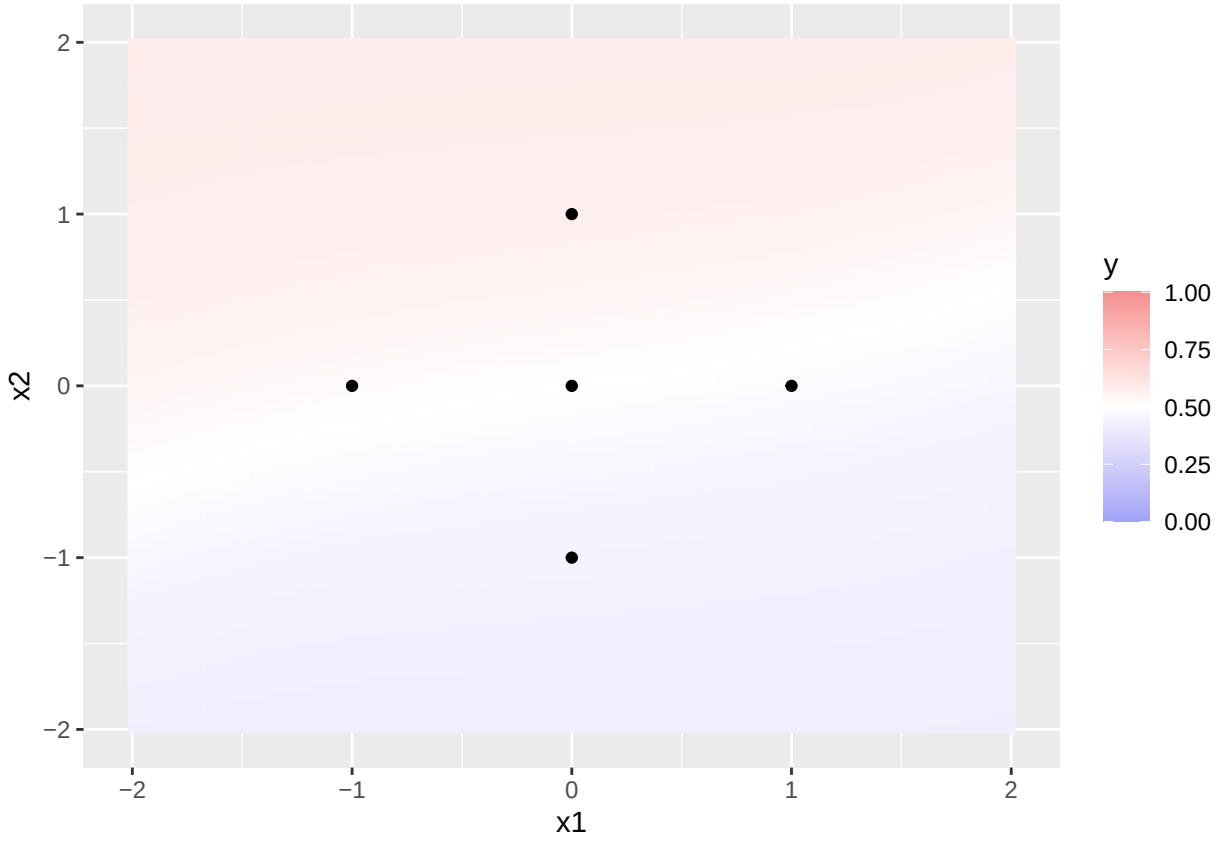
As in the previous labs, let us visualize the output for a randomly initialized network:

```
library(scales)
library(ggplot2)

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

# run this a few times to see what different random networks predict
nnet = nnet.new(c(2, 5, 3, 1))
plot_grid(nnet.predict(nnet, grid))
```



Now, we code backpropagation to compute the gradients. Use the vectorized formulas in Equations 5-8 to make your code much faster.

```
nnet.gradients = function(nnet, x, y) {
  # x is be a matrix with samples on rows
  # y is a vector with the labels

  n_layers = length(nnet$weights)
  activations = list(x)
  pre_activations = list(x)
  for(l in 1:n_layers) {
    zin = t(t(activations[[l]] %*% nnet$weights[[l]]) + nnet$biases[[l]])
    zout = nnet$activations[[l]](zin)
    pre_activations[[l + 1]] = zin
    activations[[l + 1]] = zout
  }
  loss = nnet$loss(y, activations[[length(activations)]])

  weights_gradients = list()
  biases_gradients = list()
  # Eq. 5
  deltas = nnet$loss_derivative(
    y, activations[[length(activations)]]
```

```
  ) * nnet$activations_derivatives[[n_layers]](pre_activations[[length(activations)]])
  for(l in n_layers:1) {
    weights_gradients[[l]] = t(activations[[l]]) %*% deltas  # Eq. 6
    biases_gradients[[l]] = colSums(deltas)  # Eq. 7

    if(l > 1) {
      # Eq. 8
      deltas = deltas %*% t(
        nnet$weights[[l]]
      ) * nnet$activations_derivatives[[l - 1]](pre_activations[[l]])
    }
  }

  # make sure the gradients have the correct size
  for(l in 1:n_layers) {
    stopifnot(dim(nnet$weights[[l]]) == dim(weights_gradients[[l]]))
    stopifnot(length(nnet$biases[[l]]) == length(biases_gradients[[l]]))
  }

  # return gradients as a list
  list(
    loss = loss,
    weights_gradients = weights_gradients,
    biases_gradients = biases_gradients
  )
}

data.x = matrix(c(
  0, 1, 0, -1, 0,
  0, 0, -1, 0, 1
), ncol = 2)
data.y = c(1, 0, 0, 0, 0)
nnet.gradients(nnet, data.x, data.y)
```

```
## $loss
## [1] 0.2519268
##
## $weights_gradients
## $weights_gradients[[1]]
##               [,1]         [,2]          [,3]         [,4]          [,5]
## [1,]  0.0002038037 -0.001168316 -0.0009001711  0.001033912 -0.0009998162
## [2,] -0.0004942026  0.002546084  0.0016914546 -0.002360452  0.0011475807
##
## $weights_gradients[[2]]
##               [,1]          [,2]        [,3]
## [1,] -0.0004482070 -6.256298e-06 0.001553982
## [2,] -0.0004780603 -8.758309e-06 0.001405281
## [3,] -0.0009479815 -1.812323e-05 0.002695238
## [4,] -0.0004711807 -7.830662e-06 0.001482008
## [5,] -0.0013276378 -2.602887e-05 0.003696342
##
## $weights_gradients[[3]]
##             [,1]
## [1,] 0.004321912
## [2,] 0.010468581
## [3,] 0.008273765
##
##
```

```
## $biases_gradients
## $biases_gradients[[1]]
## [1] -0.004688995  0.032946858  0.022790616 -0.028361400  0.019991625
##
## $biases_gradients[[2]]
## [1] -0.0198761400 -0.0002447866  0.0608179051
##
## $biases_gradients[[3]]
## [1] 0.1480732
```

We now need to implement gradient descent:

```
nnet.gradient_descent_step = function(nnet, gradients, learning_rate) {
  for(l in 1:length(nnet$weights)) {
    nnet$weights[[l]] = (
      nnet$weights[[l]] - learning_rate * gradients$weights_gradients[[l]]
    )

    nnet$biases[[l]] = (
      nnet$biases[[l]] - learning_rate * gradients$biases_gradients[[l]]
    )
  }
  nnet  # return the modified parameters
}


nnet.train = function(nnet, x, y, n_epochs, learning_rate) {
  losses = list()

  for(e in 1:n_epochs) {
    gradients = nnet.gradients(nnet, x, y)
    nnet = nnet.gradient_descent_step(nnet, gradients, learning_rate)
    losses[[length(losses) + 1]] = gradients$loss
  }

  list(
    losses = unlist(losses),
    nnet = nnet
  )
}
```

Finally, let us train the network on the small dataset:

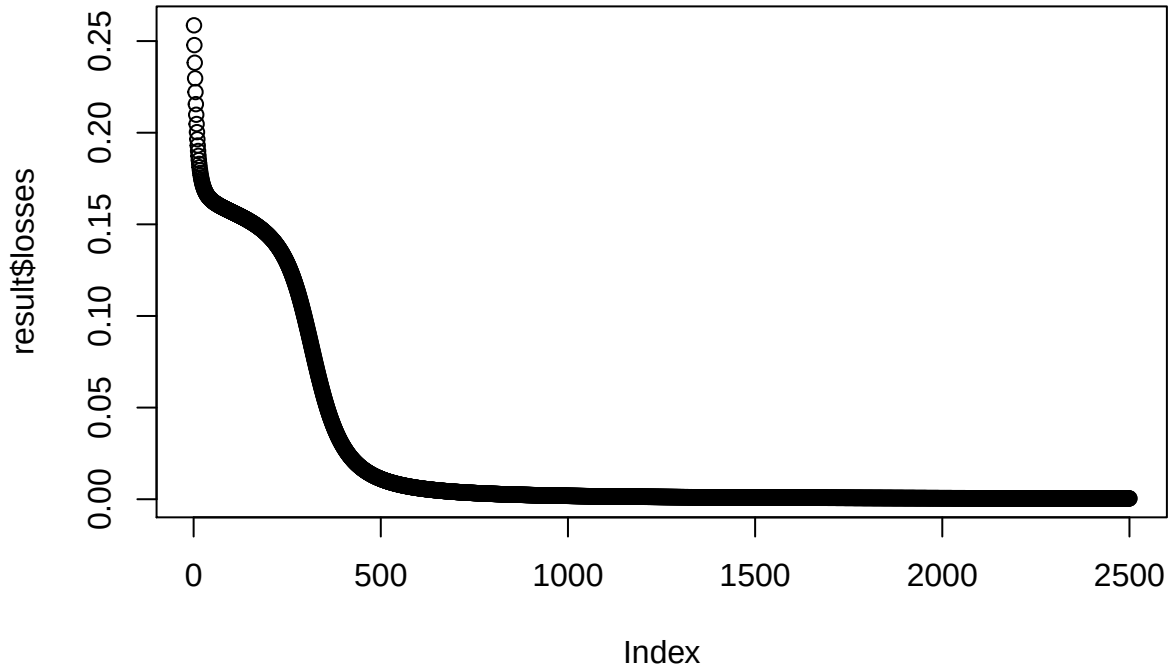
```
data.x = matrix(c(
  0, 1, 0, -1, 0,
  0, 0, -1, 0, 1
), ncol = 2)
data.y = c(1, 0, 0, 0, 0)

nnet = nnet.new(c(2, 5, 3, 1))
result = nnet.train(nnet, data.x, data.y, 2500, 0.25)
nnet.predict(result$nnet, data.x)
```

```
##             [,1]
## [1,] 0.97242010
## [2,] 0.01992840
## [3,] 0.01979303
## [4,] 0.01823799
## [5,] 0.01801450
```

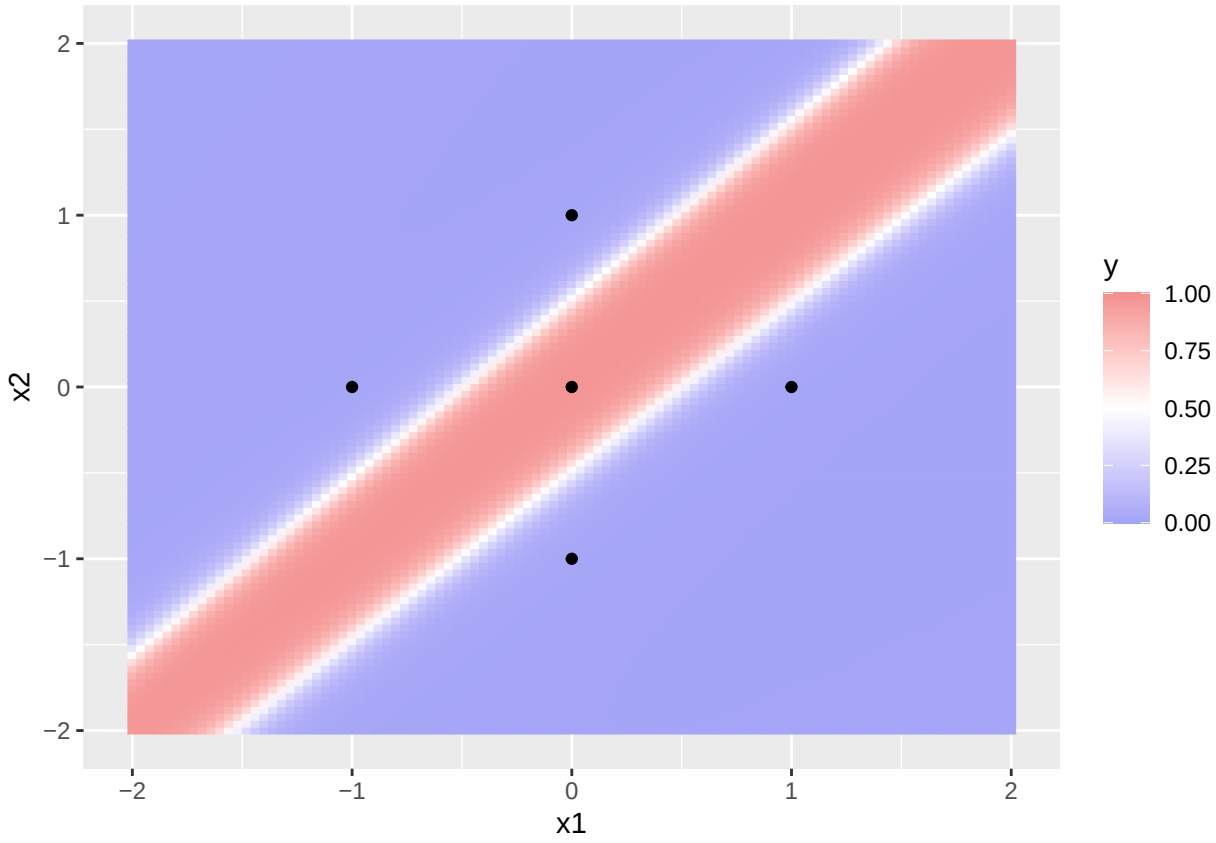
By plotting the loss after each parameter update, we can be sure that the network converged:

```
plot(result$losses)
```



And the decision boundary of the network is:

```
plot_grid(nnet.predict(result$nnet, grid))
```



Try to train a few randomly initialized network to discover different decision boundaries. Try to modify the learning rate and see how it affects the convergence speed. Finally, try different ways to initialize the weights and note how the trainability of the network is affected.