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.

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
  # TODO perform the forward pass and return the predictions
}
```

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$weights)
  activations = list(x)
  pre_activations = list(x)
  # TODO perform the forward pass, storing all activations
  loss = nnet$loss(y, activations[[length(activations)]])

  weights_gradients = list()
  biases_gradients = list()
  # TODO compute the gradients

  # 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)
```

We now need to implement gradient descent:

```
nnet.gradient_descent_step = function(nnet, gradients, learning_rate) {
  # TODO perform one step of gradient descent,
  # modifying the parameters of the network
  nnet  # return the modified parameters
}


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

  # TODO iterate over the dataset for the given number of epochs and
  # modify the weights at each epoch.
  # use all the data to compute the gradients

  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)
```

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.