Lab 4

Welcome to the fourth lab. In this lab, we will derive the backpropagation equations, code the training procedure, and test it on the XOR problem. Additionally, there will be a couple of theoretical questions about weight decay (or L2 regularization) that should give you some intuition on how it works.

Exercise 1

Derive the back-propagation algorithm. You might find the results of the second exercise of the previous lab a useful reference.

- Consider a neural network with L layers and a loss function $\mathcal{L}$ composed of a generic error term $\mathcal{E}(\mathbf{y}^L, \hat{\mathbf{y}})$ and weight decay term $\mathcal{R}_\lambda(\mathbf{W})$. Call the output of the ℓ-th layer $\mathbf{y}^\ell = \phi_\ell(\mathbf{z}^\ell)$ with $\mathbf{z}^\ell = \mathbf{W}^\ell \mathbf{y}^{\ell-1} + \mathbf{b}^\ell$ its pre-activation output. Finally, consider a vector $\delta^\ell = \nabla_{\mathbf{z}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ containing the gradient of the loss with respect to the pre-activation outputs of layer ℓ.
- Compute δ^L.
- Compute $\nabla_{\mathbf{W}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ in terms of $\mathbf{y}^{\ell-1}$ and δ^ℓ.
- Compute $\nabla_{\mathbf{b}^\ell} \mathcal{L}(\mathbf{y}^L, \hat{\mathbf{y}})$ in terms of $\mathbf{y}^{\ell-1}$ and δ^ℓ.
- Compute $\delta^{\ell-1}$ from δ^ℓ.
- Use vectorized operations (i.e. operations with vectors and matrices): they will make your code in the next exercise *much* faster!
- Optional: Extend the vectorized operations to handle data in batches. Call $\mathbf{Y}^\ell$ and Δ^ℓ the matrices whose i-th rows contains the activations $\mathbf{y}^\ell$ and deltas δ^ℓ of the i-th training example in the batch.
- Hint: make sure that the results have the right shape. The deltas should be vectors, and the gradients should have the same shape as the respective parameters.

Exercise 2

In this exercise, we will code the backpropagation algorithm and apply it to a very simple example: the XOR problem. We will use reference classes to keep the code modular and organized.

1. Create a class that computes the binary cross entropy and its derivative.
2. Create a class that computes the ReLU activation and its derivative.
3. Create a class that computes the sigmoid activation and its derivative.
4. Create a class that computes the tanh activation and its derivative (which is $1 - \tanh(x)^2$)
5. Create a class that performs one step of gradient descent.
6. Write a function for the forward pass and for backpropagation.
7. Create a neural network with a suitable architecture and train it on the XOR problem.
8. Visualize the weights and biases of the trained network. Can you explain how the network makes predictions?
9. Test different activation functions, learning rates, initializations, number of layers and their sizes.
10. Optional: implement the softmax activation, the categorical cross entropy loss function, and train a network on the MNIST dataset.

Hint: you can check that the gradients you compute are correct by comparing them with the "empirical" gradients obtained through the finite differences method.

Hint: If you do this exercise in a R script, you will be able to debug the code and see where things are not going according to plan.

Note: all these functions should process the data in batches, i.e. matrices where every row is a different sample of the same batch.

```
loss_function = setRefClass(  # base class for loss functions
  "loss_function",
```

```r
  methods = list(
    forward = function(y_true, y_pred) NA,
    backward = function(y_true, y_pred) NA
  )
)

binary_crossentropy = setRefClass(
  "binary_crossentropy",
  contains = "loss_function",
  methods = list(
    forward = function(y_true, y_pred) {
      # TODO compute the binary cross entropy loss
    },
    backward = function(y_true, y_pred) {
      # TODO compute the gradient of the cross entropy loss
    }
  )
)

y_true = matrix(c(0, 0, 0, 1, 1, 0), ncol=3)
y_pred = matrix(c(0.2, 0.5, 0.3, 0.8, 0.1, 0.1), ncol=3)
loss = binary_crossentropy()

loss$forward(y_true, y_pred)
loss$backward(y_true, y_pred)
```

```r
activation = setRefClass(  # base class for activation functions
  "activation",
  methods = list(
    forward = function(x) NA,
    backward = function(x) NA
  )
)


relu = setRefClass(
  "relu",
  contains = "activation",
  methods = list(
    forward = function(x) {
      # TODO compute the relu activation on x
    },
    backward = function(x) {
      # TODO compute the gradient of the relu activation
    }
  )
)

x = matrix(c(-0.1, 0.3, 0.7, 0.5, -1.0, 0.7), ncol=3)
act = relu()

act$forward(x)
act$backward(x)
```

```r
sigmoid = setRefClass(
  "sigmoid",
  contains = "activation",
  methods = list(
    forward = function(x) {
      # TODO compute the sigmoid activation
    },
    backward = function(x) {
      # TODO compute the gradient of the sigmoid activation
    }
  )
)

x = matrix(c(2, 0, -2, 0.5, -0.25, 0.25), ncol=3)
act = sigmoid()

act$forward(x)
act$backward(x)
```

```r
htan = setRefClass(
  "htan",
  contains = "activation",
  methods = list(
    forward = function(x) {
      # TODO compute the tanh activation
    },
    backward = function(x) {
      # TODO compute the gradient of the tanh activation
    }
  )
)

act = htan()

act$forward(x)
act$backward(x)
```

```r
gradient_descent_optimizer = setRefClass(
  "gradient_descent_optimizer",
  fields = list(
    learning_rate = "numeric"
  ),
  methods = list(
    step = function(x, gradient) {
      # TODO perform one step of gradient descent on x
    }
  )
)

opt = gradient_descent_optimizer(learning_rate = 0.1)
opt$step(10, 10)
```

```r
dense_neural_network = setRefClass(
  "dense_neural_network",
```

```
  fields = list(
    weights = "list",
    biases = "list",
    activations = "list",
    loss = "loss_function",
    optimizer = "gradient_descent_optimizer"
  ),
  methods = list(

    predict = function(batch_x) {
      result = batch_x

      # TODO perform the forward pass to get the predictions

      result
    },

    train_on_batch = function(batch_x, batch_y, iter, lrate) {
      intermediate_activations = list(batch_x)

      # TODO perform the forward pass to get the predictions
      # and put the intermediate activations in the list

      batch_loss = loss$forward(
        batch_y,
        intermediate_activations[[length(intermediate_activations)]]
      )

      weight_gradients = list()
      bias_gradients = list()

      # TODO use backpropagation to compute the gradients,
      # accumulate them in the lists, then use the optimizer
      # to apply the gradients to the parameters

      batch_loss
    }
  )
)


# just an utility function to create networks
build_dense_neural_network = function(input_size, layers, loss, optimizer) {
  weights = list()
  biases = list()
  activations = list()

  last_layer_size = input_size
  for(i in 1:length(layers)) {
    if(class(layers[[i]]) == "numeric") {
      sd = sqrt(2 / (last_layer_size + layers[[i]]))
      vals = rnorm(n = last_layer_size * layers[[i]], mean = 0, sd = sd)
      vals = ifelse(vals > 2 * sd, 2 * sd, vals)
```

```r
      vals = ifelse(vals < -2 * sd, -2 * sd, vals)
      weights[[length(weights) + 1]] = matrix(
        vals, ncol = layers[[i]], nrow = last_layer_size
      )
      biases[[length(biases) + 1]] = rep(0, layers[[i]])
      last_layer_size = layers[[i]]
    }
    else {
      activations[[length(activations) + 1]] = layers[[i]]
    }
  }

  dense_neural_network(
    weights = weights,
    biases = biases,
    activations = activations,
    loss = loss,
    optimizer = optimizer
  )
}


data.x = matrix(c(
  -1, -1,
  -1, 1,
  1, -1,
  1, 1
), nrow = 4, ncol = 2, byrow = TRUE)


data.y = matrix(c(
  0, 1, 1, 0
), nrow = 4, ncol = 1)


network = build_dense_neural_network(
  input_size = 2,
  layers = list(2, htan(), 1, sigmoid()),
  loss = binary_crossentropy(),
  optimizer = gradient_descent_optimizer(learning_rate = 0.25)
)


losses = lapply(1:250, function(i) network$train_on_batch(data.x, data.y))
network$predict(data.x)
plot(1:length(losses), losses)
```

```r
network$weights
network$biases
```

Exercise 3

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 formula 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.
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.