CS 229 - Machine Learning
Xiangliang Zhang
Computer Science (CS)/Statistics (STAT) Program
Computer, Electrical and Mathematical Sciences & Engineering (CEMSE) Division
King Abdullah University of Science and Technology (KAUST)

HOMEWORK VI

Henrique Aparecido Laureano

Spring Semester 2018

Contents

Question 1: Property of derivatives of Error function (Exercise 5.6 and 5.7 of Bishop's book)

(1)

Show the derivative of the error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\} \tag{5.21}$$

with respect to the activation a_k for an output unit having a logistic sigmoid activation function

$$y_k = \frac{1}{1+\exp(-a_k)}$$

satisfies

$$\frac{\partial E}{\partial a_k} = y_k - t_k. \tag{5.18}$$

Solution:

$$\begin{aligned}
\frac{\partial E}{\partial a_k} &= -\left[\frac{t_k}{y_k}\frac{\partial y_k}{\partial a_k} + \frac{1-t_k}{1-y_k}\frac{\partial(-y_k)}{\partial a_k}\right] \\
&= -\left[\frac{t_k}{y_k}\frac{\exp(-a_k)}{(1+\exp(-a_k))^2} - \frac{1-t_k}{1-y_k}\frac{\exp(-a_k)}{(1+\exp(-a_k))^2}\right] \\
&= -\left[t_k(1+\exp(-a_k))\frac{\exp(-a_k)}{(1+\exp(-a_k))^2} - \frac{1-t_k}{\frac{\exp(-a_k)}{1+\exp(-a_k)}}\frac{\exp(-a_k)}{(1+\exp(-a_k))^2}\right] \\
&= -\left[t_k\frac{\exp(-a_k)}{1+\exp(-a_k)} - (1-t_k)\frac{1+\exp(-a_k)}{\exp(-a_k)}\frac{\exp(-a_k)}{(1+\exp(-a_k))^2}\right] \\
&= -\left[t_k\frac{\exp(-a_k)}{1+\exp(-a_k)} - \frac{1-t_k}{1+\exp(-a_k)}\right] \\
&= -\left[t_k(1-y_k) - (1-t_k)y_k\right] \\
&= -\left[t_k - y_k\right] \\
&= \boxed{y_k - t_k.}
\end{aligned}$$

□

(2)

Show the derivative of the error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} \ln y_{nk}(\mathbf{x}_n, \mathbf{w}) \tag{5.24}$$

with respect to the activation a_k for output units having a softmax activation function

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$$

satisfies

$$\frac{\partial E}{\partial a_k} = y_k - t_k. \tag{5.18}$$

Hint : **for each $\mathbf{x}_n$, its true label has $t_{nk} = 0$ or 1, and $\sum_{k=1}^{K} t_{nk} = 1$. That is to say, for the activation a_k activated by an $\mathbf{x}$, the corresponding t_k could be either 1 or 0.**

Solution:

$$\frac{\partial E}{\partial a_k} = -\sum_{n=1}^{N} t_n \frac{\partial \ln y_n(\mathbf{x}_n, \mathbf{w})}{\partial a_k}.$$

$$y_n(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_n(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))} \quad \Rightarrow \quad \ln y_n(\mathbf{x}_n, \mathbf{w}) = a_n(\mathbf{x}, \mathbf{w}) - \ln \sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))$$

we have

$$\frac{\partial \ln y_n(\mathbf{x}_n, \mathbf{w})}{\partial a_k} = \delta_{nk} - \frac{1}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))} \frac{\partial \sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}{\partial a_k},$$

with δ_{nk} being the Kronecker delta (1 or 0).

Then the gradient of the softmax-denominator is

$$\frac{\partial \sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}{\partial a_k} = \sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))\delta_{jk} = \exp(a_k(\mathbf{x}, \mathbf{w}))$$

which gives

$$\frac{\partial \ln y_n(\mathbf{x}_n, \mathbf{w})}{\partial a_k} = \delta_{nk} - \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))} = \delta_{nk} - y_k.$$

So the gradient of E with respect to a_k is then

$$\frac{\partial E}{\partial a_k} = \sum_{n=1}^{N} t_n (y_k - \delta_{nk}) = y_k \Big(\sum_{n=1}^{N} t_n\Big) - t_k = y_k \cdot 1 - t_k = \boxed{y_k - t_k}.$$

$\square$

Question 2: A different error function (Exercise 5.9 of Bishop's book)

The error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1-t_n)\ln(1-y_n)\} \tag{5.21}$$

for binary classification problems was derived for a network having a logistic-sigmoid output activation function, so that $0 \leq y(x,w) \leq 1$, and data having target values $t \in \{0,1\}$. *Derive the corresponding error function* if we consider a network having an output $-1 \leq y(x,w) \leq 1$ and target values $t=1$ for class C1 and $t=-1$ for class C2. What would be the appropriate *choice of output unit activation function*?

Solution:

Scaling and shifting the binary outputs directly gives the activation function (using the motation from (5.19))

$$y = 2\sigma(a) - 1.$$

The error function is constructed from (5.21) by applying the inverse transformation to y_n and t_n

$$\begin{aligned} E(\mathbf{w}) &= -\sum_{n=1}^{N} \frac{1+t_n}{2}\ln\frac{1+y_n}{2} + \left(1 - \frac{1+t_n}{2}\right)\ln\left(1-\frac{1+y_n}{2}\right) \\ &= \boxed{-\frac{1}{2}\sum_{n=1}^{N}\{(1+t_n)\ln(1+y_n) + (1-t_n)\ln(1-y_n)\}} + N\ln 2 \end{aligned}$$

(the last term can be dropped, since it is independent of $\mathbf{w}$).

To find the activation function we apply the linear transformation to the logistic sigmoid given by (5.19), which gives

$$\begin{aligned} y = 2\sigma(a) - 1 &= \frac{2}{1+\exp(-a)} - 1 \\ &= \frac{1-\exp(-a)}{1+\exp(-a)} \\ &= \frac{\exp(a/2) - \exp(-a/2)}{\exp(a/2)+\exp(-a/2)} \\ &= \boxed{\tanh(a/2).} \qquad \text{(hyperbolic tangent)} \end{aligned}$$

□

Question 3: Implementation of NN (using Back-Propagation)

Data : **Generate a set of data *points* (x, y), by choosing a *nonlinear function* $f(x)$ and evaluating $y = f(x) + \text{noise}$ for *random* x values, where each x is a *real vector of at least 2 elements* and each y *is a real scalar or vector*. As an alternative, you can use a data set you find online or are using in other research. You should clearly state what your data set is, or how you generated it.**

Here I'm using a subset of cereal dataset shared by Carnegie Mellon University (CMU). The details of the dataset are on the following link:
http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html. The objective is to predict rating of the cereals variables such as calories, proteins, fat etc. See the range of values and behavior of the variables in Figure 1). The data is in .csv format and can be downloaded by clicking: cereals.

```
# <r code> ============================================================== #
path <- "~/Dropbox/KAUST/machine_learning/hw6/"                # files directory
df <- read.csv(paste0(path, "cereals.csv"), header = TRUE)      # loading dataset
par(mfrow = c(2, 3), mar = c(4, 4, 0, 1) + .1)          # graphical definitions
                                                             # plotting variables
for (i in 1:5) plot(rating ~ df[ , i], xlab = names(df)[i], df)
# </r code> ============================================================= #
```

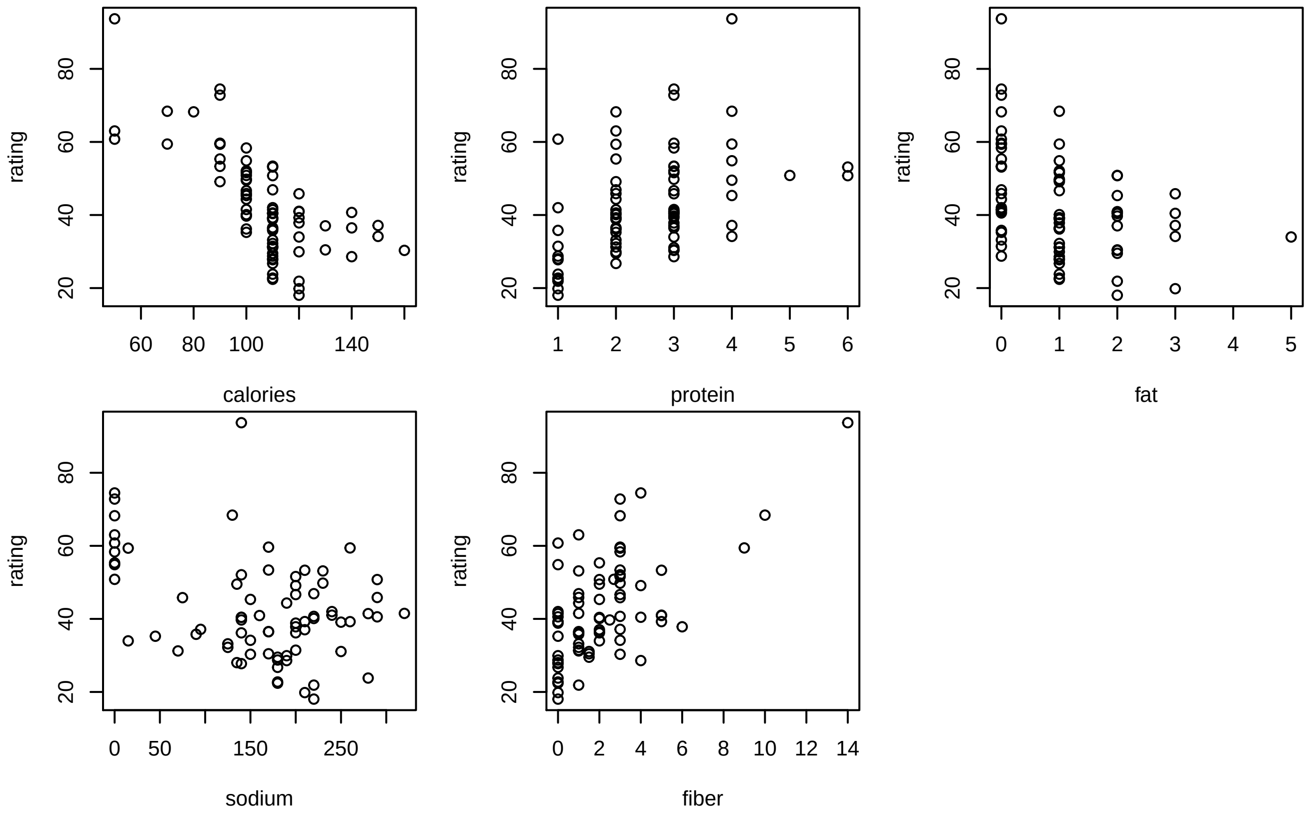

Figure 1: relationships between the rating, y, and the x's.

```r
# <r code> ==================================================================== #
                    # sampling random index to separate the data in train and test
random <- sample(75, 50)                             # 50 values for the train dataset
train <- df[random, ] ; test <- df[-random, ]
                                                      # applying z-score normalization
train.std <- apply(train, 2, function(x) (x - mean(x)) / sd(x))
test.std <- test
for (i in 1:6)
  test.std[ , i] <- (test.std[ , i] - mean(train[ , i])) / sd(train[ , i])
# </r code> =================================================================== #
```

Task : Implement a two-layer neural network with back-propagation.
The network should have *2 or more inputs*. The inputs connect to M neurons in the *hidden layer*, each of which takes a weighted sum of its inputs plus a bias and then applies the *hyperbolic tangent function (as the activation function)*. The network should have 1 or more outputs. Each output of the network is a weighted sum plus bias of the outputs of the hidden layer (need no activation function or use *identity function* $f(\mathrm{x}) = \mathrm{x}$ for the *output* because this is a regression problem).
During learning, both weights and biases change to decrease the mean squared error.

1)

Describe all the parameters you chose, including the number of inputs, outputs, and hidden neurons, the sizes of the initial random weights, learning rate etc.

Solution/Implementation :

```r
# <r code> ==================================================================== #
                                                          # hyperbolic tangent function
htan <- function(x) (exp(x) - exp(-x)) / (exp(x) + exp(-x))

prop.for <- function(x, w1, w2) {                                # forward propagation
  z1 = cbind(1, x) %*% w1
  h = htan(z1)
  z2 = h %*% w2
  list(output = z2, h = h)
}
                                                                    # back-propagation
backpropagate <- function(x, target, y, h, w1, w2, learn.rate) {
  delta = y - target
  dw1 = t(t(delta %*% t(w2) %*% (1 - crossprod(h))) %*% cbind(1, x))
  dw2 = t(delta) %*% h

  w1 = w1 - learn.rate * dw1
  w2 = w2 - learn.rate * t(dw2)

  list(w1 = w1, w2 = w2)
}
      # neural network: 10 hidden neurons, learn rate of 0.001 and 2000 iterations
neuralnet <- function(x, target, hidden = 10, learn.rate = 1e-3, iter = 2e3) {
  d = ncol(x) + 1
  w1 = matrix(rnorm(d * hidden), d, hidden)
  w2 = as.matrix(rnorm(hidden))
  error = matrix(NA, 2, iter)
  for (i in 1:iter) {
    pf = prop.for(x, w1, w2)
    bp = backpropagate(x, target, y = pf$output, h = pf$h, w1, w2
                       , learn.rate = learn.rate)
    w1 <- bp$w1 ; w2 <- bp$w2
    error[1, i] = (1/length(target)) * sum((pf$output - target)**2)
    error[2, i] = sd(abs(pf$output - target))
  }
  list(output = pf$output, w1 = w1, w2 = w2, error = error)
}

x <- data.matrix(train.std[ , -6]) ; target <- train.std[ , 6]
                                          # five inputs of size 50 plus an intercept
                                                          # vector output of size 50
                  # initial random weights generated by a normal of mean 0 and sd 1
                         # w_{1}: matrix of dim 6 x 10, w_{2}: matrix of dim 1 x 10
nn.train <- neuralnet(x, target)
# </r code> ============================================================= #
```

□

2)

Find a learning rate that allows it to learn to a small mean squared error. ***Plot a figure of how the error decreases during learning.***

Solution :

The biggest learning rate that allows learning consecutive times is with 0.001.

```
# <r code> ============================================================== #
plot(nn.train$error[1, ], type = "l", col = 2                 # plotting the MSE
     , main = paste("MSE:", round(nn.train$error[1, 2e3], 3))
     , xlab = "Iteration", ylab = "MSE")
# </r code> ============================================================= #
```

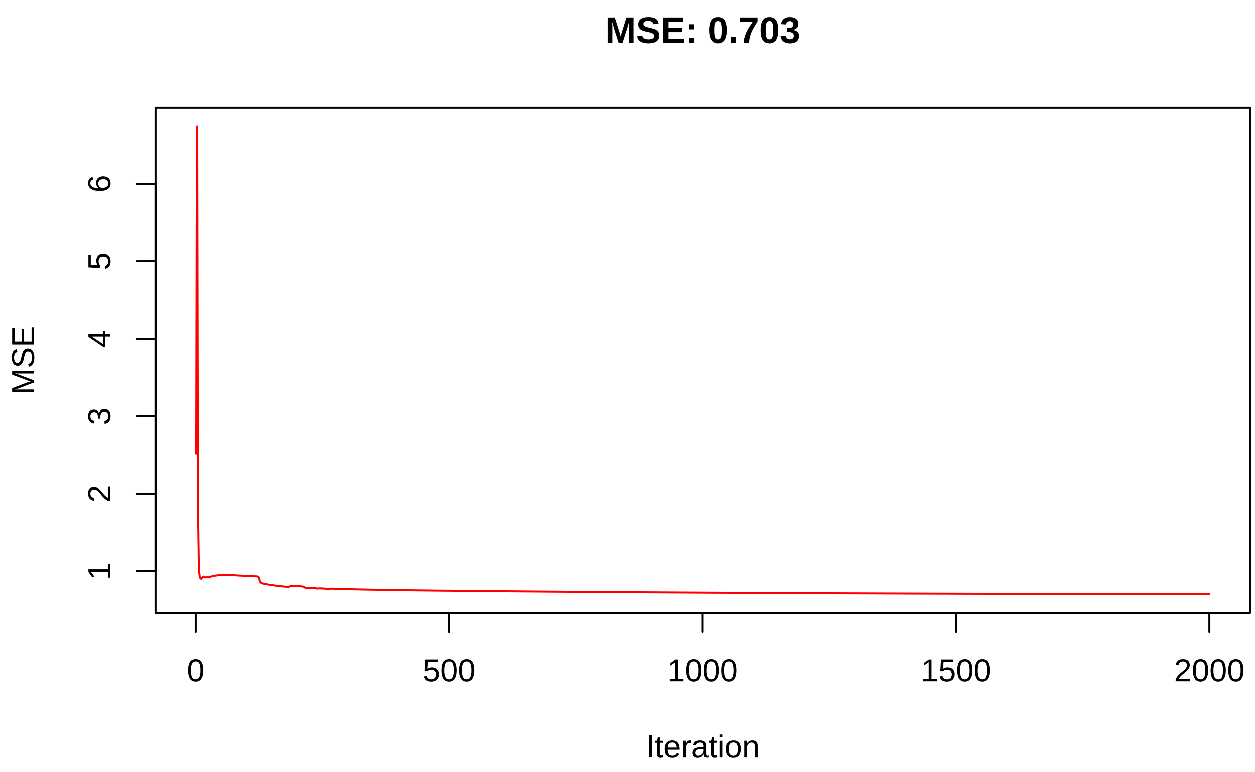

Figure 2: Neural Network MSE during learning.

□

3)

Test **the NN you learned by a** ***different*** **set of data** ***points*** (x, y) **(different from training set, but y is still generated by** $f(\mathrm{x}) +$ **noise), what's the error when comparing the** ***predicted*** y_n **with the** ***true target*** y**? Give the mean and standard deviation of errors.**

Solution :

```
# <r code> =================================================================== #
x <- data.matrix(test.std[ , -6]) ; target <- test.std[ , 6]        # test dataset
                                        # five inputs of size 25 plus an intercept
                                                        # vector output of size 25
                                                         # testing neural network
target.test <- prop.for(x, w1 = nn.train$w1, w2 = nn.train$w2)$output

(1/length(target)) * sum( (target.test - target)**2 )                  # MSE error

[1] 0.3310582
```

```
mean( abs(target.test - target) )                                  # error mean

[1] 0.4757085

sd( abs(target.test - target) )                          # error standard deviation

[1] 0.3303402

# </r code> ================================================================== #
```

□

4)

How will the training error and testing error be different if you re-train the NN by different initializations of weights? And how will they change if you set M (the number of hidden units) to be different values? *Plot the error bars* (as examples of http://www.mathworks.com/help/techdoc/ref/errorbar.html) *for both training and testing.*

Solution :

How will the training error and testing error be different if you re-train the NN by different initializations of weights?
Maybe a little different, but in average very similar. Doing this we only put different start points. The ideia of the algorithm (gradient descent) is to find the same/desired region, even with different start points (the method is robust), if the same data is used.

And how will they change if you set M (the number of hidden units) to be different values?
Also maybe a little different, but in average very similar. Generally, a M bigger than the number of input nodes is already good enough to reach similar results.

```
# <r code> =================================================================== #
                                                         # plotting the error bars
par(mfrow = c(1, 2), mar = c(4, 4, 2, 1) + .1)            # graphical definitions

plot(nn.train$error[1, ], type = "l", col =  2          # plotting the training MSE
     , main = "Training", xlab = "Iteration", ylab = "MSE", ylim = c(.25, 6.65))

xs <- seq(250, 2000, 250) ; ys <- nn.train$error[1, xs]
arrows(xs, ys - nn.train$error[2, xs], xs, ys + nn.train$error[2, xs] # error bars
       , length = .05, angle = 90, code = 3, col = 2)

                                              # plotting the testing error/error bar
```

```
error.mean <- mean(abs(target.test - target))
error.sd <- sd(abs(target.test - target))

plot(error.mean, pch = 19, col = 2, ylim = c(.1, .85), axes = FALSE
     , xlab = NA, ylab = "Error", main = "Test")

Axis(side = 2, at = seq(.1, .85, length.out = 4))
arrows(1, error.mean - error.sd, 1, error.mean + error.sd
       , length = .05, angle = 90, code = 3, col = 2)
# </r code> ==================================================================== #
```

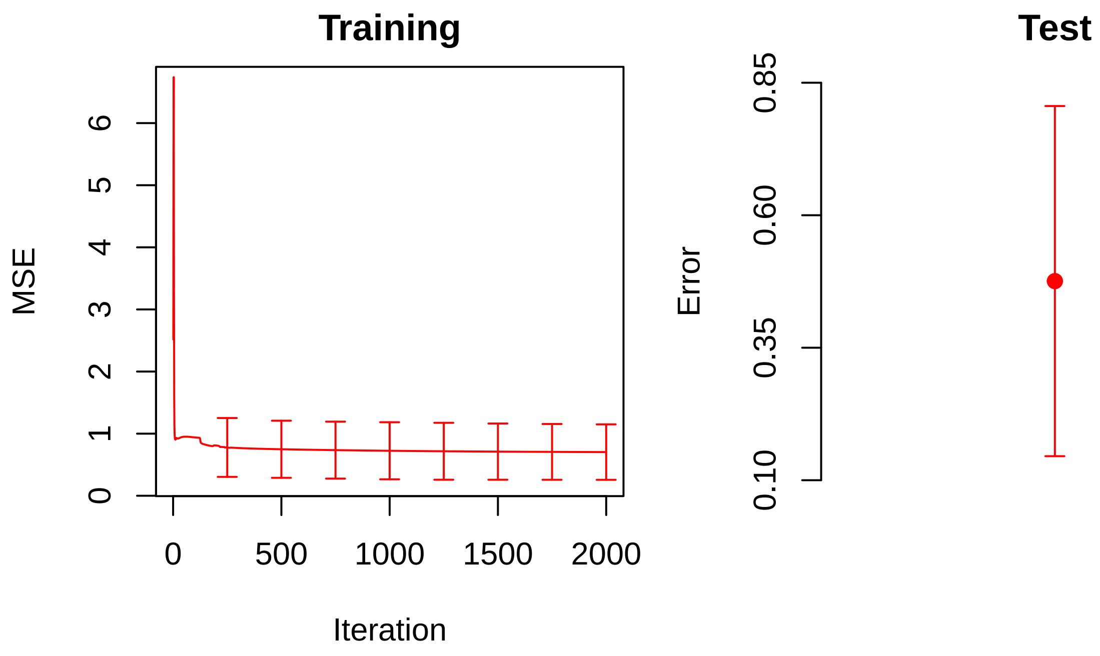

Figure 3: Left: Error bars (plus/minus standard deviation of errors at different iterations) for the MSE in the training. Right: Error mean with standard deviation error bar for the test.

□

An Additional Exercise

(*a bonus* for those who still have time, energy and interest to work)

Implement the *momentum* discussed in class.
Run it and see if it learns faster, or learns a better solution (in terms of testing error). Describe whether it improved speed, accuracy, or both, and why you think that occurred. (It's OK if you discover your "improvement" had no effect or even made it

worse. The important thing is to test it and explain the results).

Solution:

■