HW3

2015129053 김형철

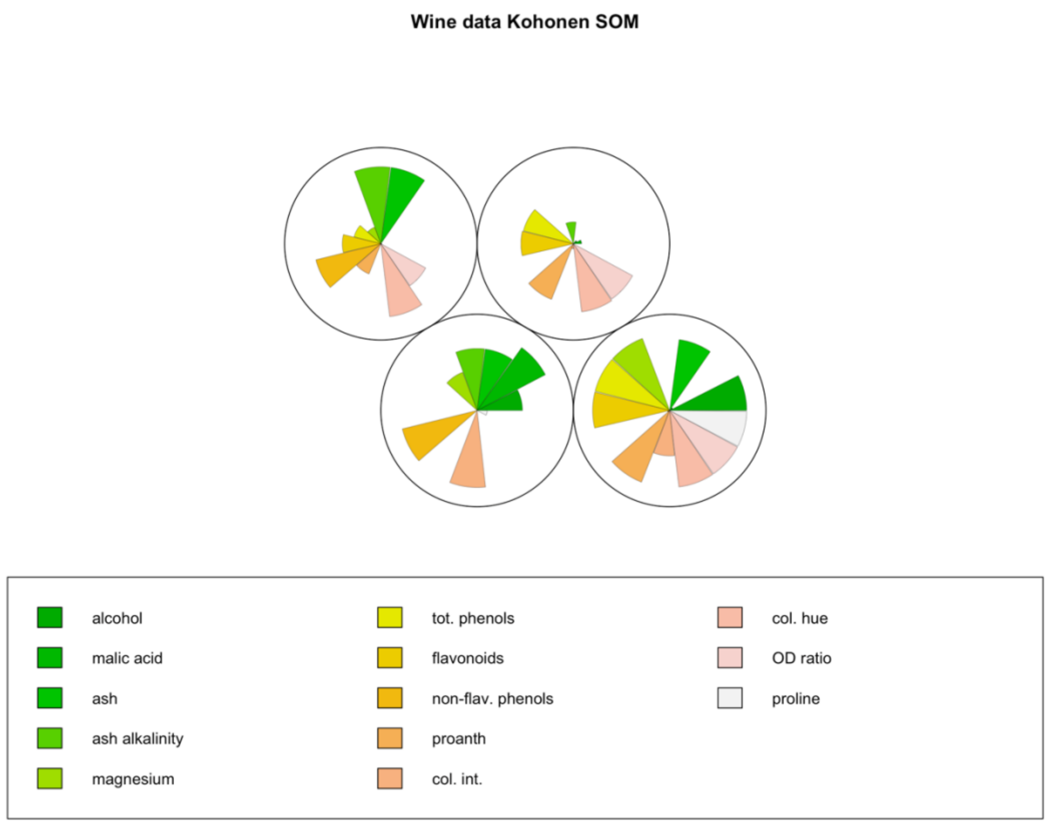
1. Using the wines data, I conducted Kohonen's SOM. I used following codes and I conducted this for varying dimension of output (2 x 2, 4 x 4, 6 x 6, 10 x 10). As the number of grids increase, the data are sorted into more specified groups. If the grid is small, the information given by the data is greatly simplified. On the other hand, as grid number increases, the data's characteristics become more specified. I think the appropriate number of grids is around 6 x 6. This is because 2 x 2 and 4 x 4 are bit too small to take into account all the various characteristics that the data have. That is, it simplifies the data too much into few clusters. On the other hand, 10 x 10 grid is too many. As the data is made up of 177 observations, having 100 clusters is a bit too much. The data will be very specified by their characteristics, but it will not be useful as there are too may number of grids.

Also, it is not possible to fit SOM for 14 x 14 grids for this example because number of grids is more larger than the size of the data. The data only have 177 observations but the grid number is 196. Thus, it is not possible to perform SOM.

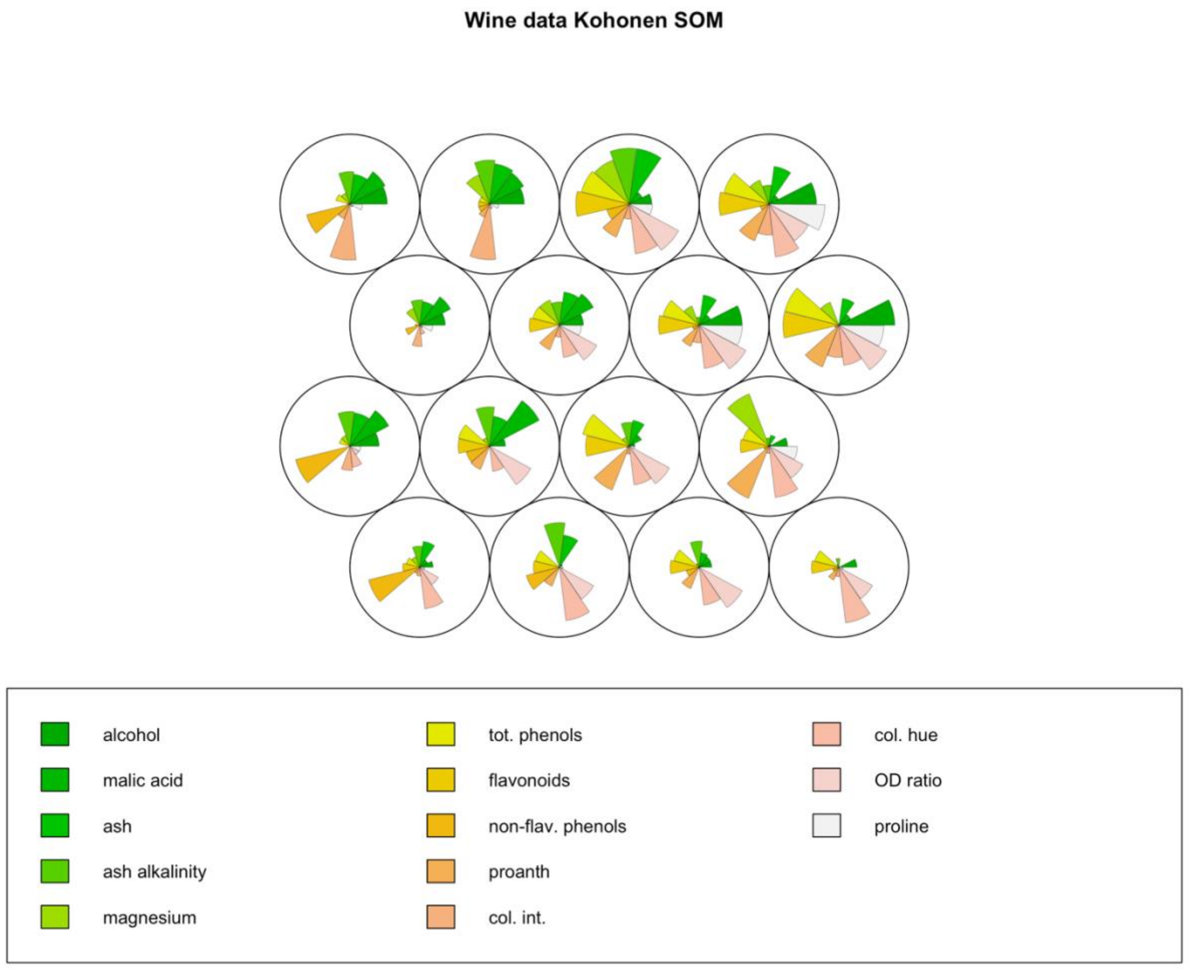
Here are the codes:

```
#1 Using Kohonen SOM on wine data

library("kohonen")
data("wines")
str(wines)
head(wines)
```

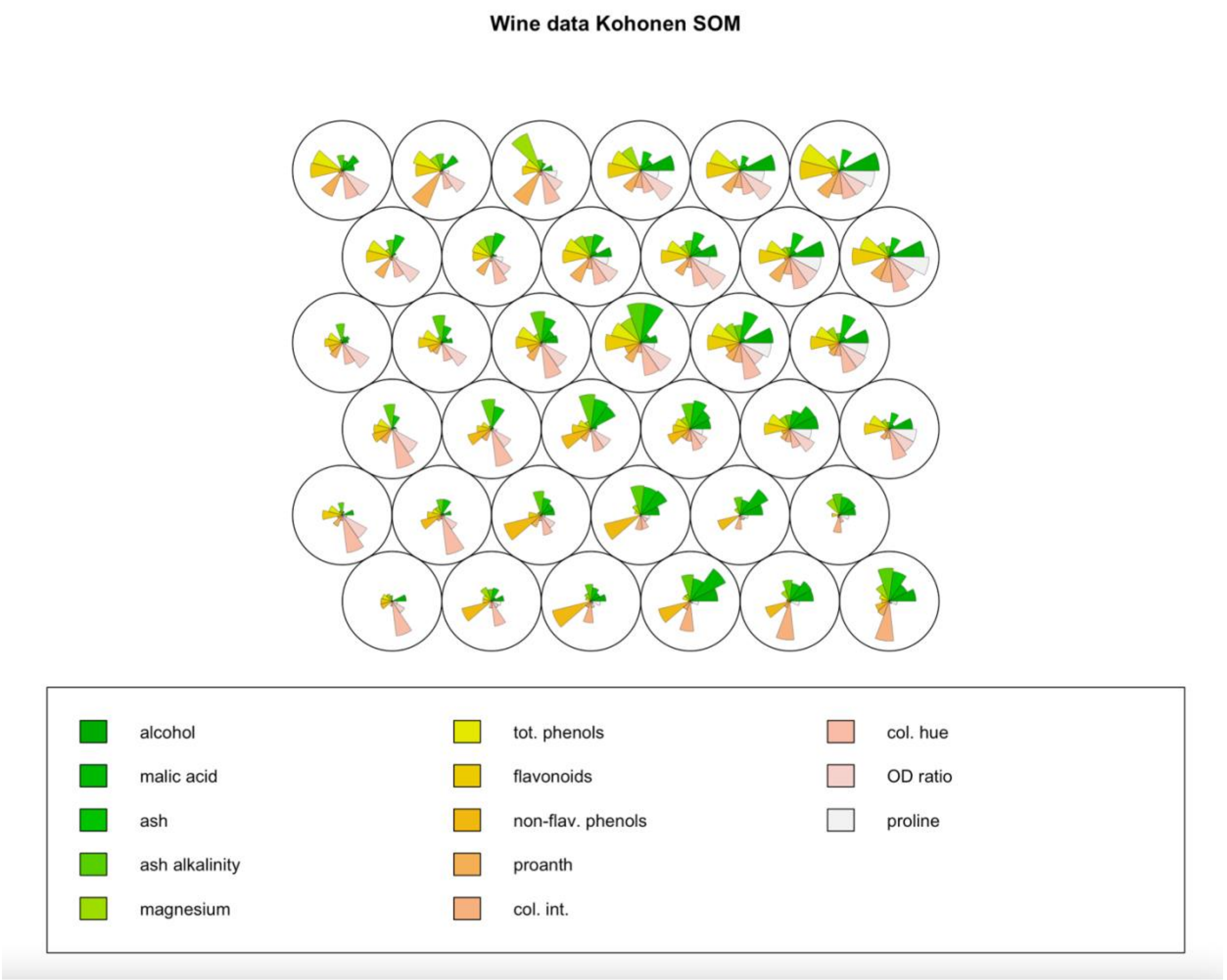
```
#for 2x2
set.seed(10)
som.wines = som(scale(wines), grid=somgrid(2,2,"hexagonal"))
som.wines
dim(getCodes(som.wines))

plot(som.wines, main="Wine data Kohonen SOM")

par(mfrow=c(1,1))
plot(som.wines, type="changes", main="Wine data: SOM")
```



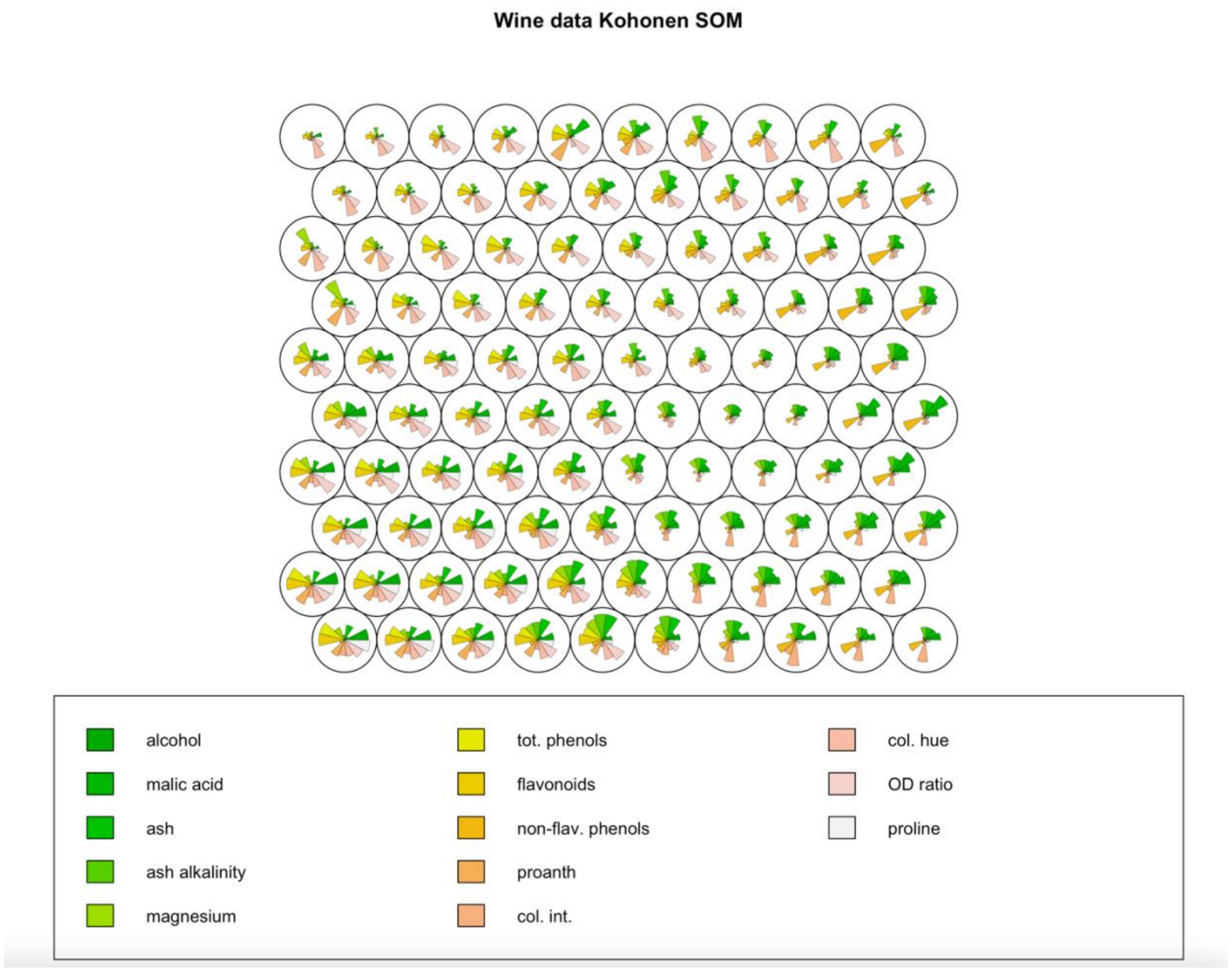
```
#for 4x4
set.seed(10)
som.wines = som(scale(wines), grid=somgrid(4,4,"hexagonal"))
som.wines
dim(getCodes(som.wines))

plot(som.wines, main="Wine data Kohonen SOM")

par(mfrow=c(1,1))
plot(som.wines, type="changes", main="Wine data: SOM")
```



```
#for 6x6
set.seed(10)
som.wines = som(scale(wines), grid=somgrid(6,6,"hexagonal"))
som.wines
dim(getCodes(som.wines))

plot(som.wines, main="Wine data Kohonen SOM")

par(mfrow=c(1,1))
plot(som.wines, type="changes", main="Wine data: SOM")
```



```
#for 10x10
set.seed(10)
som.wines = som(scale(wines), grid=somgrid(10,10,"hexagonal"))
som.wines
dim(getCodes(som.wines))

plot(som.wines, main="Wine data Kohonen SOM")

par(mfrow=c(1,1))
plot(som.wines, type="changes", main="Wine data: SOM")
```



```
#for 14x14 this gives us error as there are more grids than the number of observations

# set.seed(10)
# som.wines = som(scale(wines), grid=somgrid(14,14,"hexagonal"))
# som.wines
# dim(getCodes(som.wines))
#
# plot(som.wines, main="Wine data Kohonen SOM")
#
# par(mfrow=c(1,1))
# plot(som.wines, type="changes", main="Wine data: SOM")
```

2.

(a) I created forwardProp function based on the neural network in this HW problem as follows:

```
forwardProp = function(input, w){
    # input to hidden layer
    neth1 = w[1]*input[1]
    neth2 = w[2]*input[1] + w[3]*input[2]
    outh1 = sigmoid(neth1)
    outh2 = sigmoid(neth2)

    # hidden layer to output layer
    neto1 = w[4]*outh1
    neto2 = w[5]*outh1 + w[6]*outh2
    outo1 = sigmoid(neto1)
    outo2 = sigmoid(neto2)

    res = c(outh1, outh2, outo1, outo2)
    return(res)
}
```

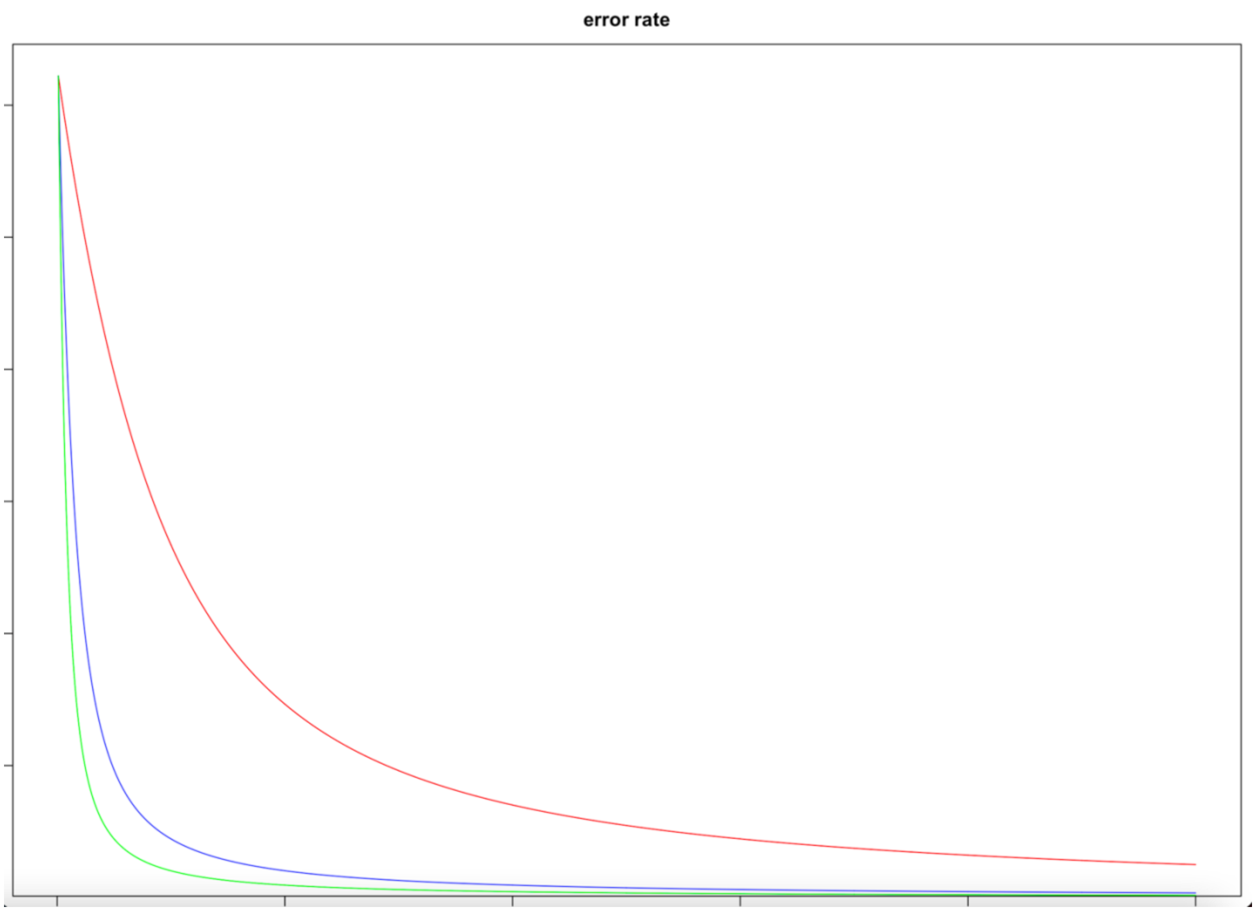
(b) I also created backpropagation update code based on the neural network in this HW problem as follows:

```
numIter = 1000

### Initial settings

# Initialize parameters
w1 = 0.7
w2 = 0.9
w3 = 0.5
w4 = 0.3
w5 = 0.8
w6 = 0.5

w = c(w1, w2, w3, w4, w5, w6)

# input and target values
input1 = 2.5
input2 = 0.5
input = c(input1, input2)

out1 = 1
out2 = 1
out = c(out1, out2)


### define sigmoid activation functions
sigmoid = function(z){
    return( 1/(1+exp(-z)) )
}

forwardProp = function(input, w){
    # input to hidden layer
    neth1 = w[1]*input[1]
    neth2 = w[2]*input[1] + w[3]*input[2]
    outh1 = sigmoid(neth1)
    outh2 = sigmoid(neth2)

    # hidden layer to output layer
    neto1 = w[4]*outh1
    neto2 = w[5]*outh1 + w[6]*outh2
    outo1 = sigmoid(neto1)
    outo2 = sigmoid(neto2)

    res = c(outh1, outh2, outo1, outo2)
    return(res)
}

error = function(res, out){
    err = 0.5*(out[1] - res[3])^2 + 0.5*(out[2] - res[4])^2
    return(err)
}


# set learning rate

gamma = 0.1
gamma = 0.6
gamma = 1.2

### Implement Forward-backward propagation
err = c()
err1 = c()
err2 = c()

#for gamma = 0.1
for(i in 1:numIter){

    ### forward
    res = forwardProp(input, w)
    outh1 = res[1]; outh2 = res[2]; outo1 = res[3]; outo2 = res[4]

    ### compute error
    err[i] = error(res, out)

    ### backward propagation
    ## update w_4, w_5, w_6

    # compute dE_dw4
    dE_douto1 = -( out[1] - outo1 )
    douto1_dneto1 = outo1*(1-outo1)
    dneto1_dw4 = outh1
    dE_dw4 = dE_douto1*douto1_dneto1*dneto1_dw4

    # compute dE_dw5
    dE_douto2 = -( out[2] - outo2 )
    douto2_dneto2 = outo2*(1-outo2)
    dneto2_dw5 = outh1
    dE_dw5 = dE_douto2*douto2_dneto2*dneto2_dw5

    # compute dE_dw6
    dneto2_dw6 = outh2
    dE_dw6 = dE_douto2*douto2_dneto2*dneto2_dw6

    ## update w_1, w_2, w_3
    # compute dE_douth1 first
    dneto1_douth1 = w4
    dneto2_douth1 = w5
    dE_douth1 = dE_douto1*douto1_dneto1*dneto1_douth1 +
dE_douto2*douto2_dneto2*dneto2_douth1

    # compute dE_douth2 first
    dneto2_douth2 = w6
    dE_douth2 = dE_douto2*douto2_dneto2*dneto2_douth2

    # compute dE_dw1
    douth1_dneth1 = outh1*(1-outh1)
    dneth1_dw1 = input[1]
    dE_dw1 = dE_douth1*douth1_dneth1*dneth1_dw1

    # compute dE_dw2
    douth2_dneth2 = outh2*(1-outh2)
    dneth2_dw2 = input[1]
    dE_dw2 = dE_douth2*douth2_dneth2*dneth2_dw2

    # compute dE_dw3
    dneth2_dw3 = input[2]
    dE_dw3 = dE_douth2*douth2_dneth2*dneth2_dw3

    ### update all parameters via a gradient descent
    w1 = w1 - gamma*dE_dw1
    w2 = w2 - gamma*dE_dw2
    w3 = w3 - gamma*dE_dw3
    w4 = w4 - gamma*dE_dw4
    w5 = w5 - gamma*dE_dw5
    w6 = w6 - gamma*dE_dw6

    w = c(w1, w2, w3, w4, w5, w6)

    print(i)
```

```
}

#for gamma = 0.6
for(i in 1:numIter){

    ### forward
    res = forwardProp(input, w)
    outh1 = res[1]; outh2 = res[2]; outo1 = res[3]; outo2 = res[4]

    ### compute error
    err1[i] = error(res, out)

    ### backward propagation
    ## update w_4, w_5, w_6

    # compute dE_dw4
    dE_douto1 = -( out[1] - outo1 )
    douto1_dneto1 = outo1*(1-outo1)
    dneto1_dw4 = outh1
    dE_dw4 = dE_douto1*douto1_dneto1*dneto1_dw4

    # compute dE_dw5
    dE_douto2 = -( out[2] - outo2 )
    douto2_dneto2 = outo2*(1-outo2)
    dneto2_dw5 = outh1
    dE_dw5 = dE_douto2*douto2_dneto2*dneto2_dw5

    # compute dE_dw6
    dneto2_dw6 = outh2
    dE_dw6 = dE_douto2*douto2_dneto2*dneto2_dw6

    ## update w_1, w_2, w_3
    # compute dE_douth1 first
    dneto1_douth1 = w4
    dneto2_douth1 = w5
    dE_douth1 = dE_douto1*douto1_dneto1*dneto1_douth1 +
dE_douto2*douto2_dneto2*dneto2_douth1

    # compute dE_douth2 first
    dneto2_douth2 = w6
    dE_douth2 = dE_douto2*douto2_dneto2*dneto2_douth2
    # compute dE_dw1
    douth1_dneth1 = outh1*(1-outh1)
    dneth1_dw1 = input[1]
    dE_dw1 = dE_douth1*douth1_dneth1*dneth1_dw1

    # compute dE_dw2
    douth2_dneth2 = outh2*(1-outh2)
    dneth2_dw2 = input[1]
    dE_dw2 = dE_douth2*douth2_dneth2*dneth2_dw2

    # compute dE_dw3
    dneth2_dw3 = input[2]
    dE_dw3 = dE_douth2*douth2_dneth2*dneth2_dw3

    ### update all parameters via a gradient descent
    w1 = w1 - gamma*dE_dw1
    w2 = w2 - gamma*dE_dw2
    w3 = w3 - gamma*dE_dw3
    w4 = w4 - gamma*dE_dw4
    w5 = w5 - gamma*dE_dw5
    w6 = w6 - gamma*dE_dw6

    w = c(w1, w2, w3, w4, w5, w6)

    print(i)

}


#for gamma = 1.2
for(i in 1:numIter){

    ### forward
    res = forwardProp(input, w)
    outh1 = res[1]; outh2 = res[2]; outo1 = res[3]; outo2 = res[4]

    ### compute error
    err2[i] = error(res, out)

    ### backward propagation
    ## update w_4, w_5, w_6

    # compute dE_dw4
    dE_douto1 = -( out[1] - outo1 )
    douto1_dneto1 = outo1*(1-outo1)
    dneto1_dw4 = outh1
    dE_dw4 = dE_douto1*douto1_dneto1*dneto1_dw4

    # compute dE_dw5
    dE_douto2 = -( out[2] - outo2 )
    douto2_dneto2 = outo2*(1-outo2)
    dneto2_dw5 = outh1
    dE_dw5 = dE_douto2*douto2_dneto2*dneto2_dw5

    # compute dE_dw6
    dneto2_dw6 = outh2
    dE_dw6 = dE_douto2*douto2_dneto2*dneto2_dw6

    ## update w_1, w_2, w_3
    # compute dE_douth1 first
    dneto1_douth1 = w4
    dneto2_douth1 = w5
    dE_douth1 = dE_douto1*douto1_dneto1*dneto1_douth1 +
dE_douto2*douto2_dneto2*dneto2_douth1

    # compute dE_douth2 first
    dneto2_douth2 = w6
    dE_douth2 = dE_douto2*douto2_dneto2*dneto2_douth2

    # compute dE_dw1
    douth1_dneth1 = outh1*(1-outh1)
    dneth1_dw1 = input[1]
    dE_dw1 = dE_douth1*douth1_dneth1*dneth1_dw1

    # compute dE_dw2
    douth2_dneth2 = outh2*(1-outh2)
    dneth2_dw2 = input[1]
    dE_dw2 = dE_douth2*douth2_dneth2*dneth2_dw2

    # compute dE_dw3
    dneth2_dw3 = input[2]
    dE_dw3 = dE_douth2*douth2_dneth2*dneth2_dw3

    ### update all parameters via a gradient descent
    w1 = w1 - gamma*dE_dw1
    w2 = w2 - gamma*dE_dw2
    w3 = w3 - gamma*dE_dw3
    w4 = w4 - gamma*dE_dw4
    w5 = w5 - gamma*dE_dw5
    w6 = w6 - gamma*dE_dw6

    w = c(w1, w2, w3, w4, w5, w6)

    print(i)

}
```

(c) As I wrote in question (b), I performed forward/back propagation 1000 times using three different learning rates. Using it, I was able to get error rate figure with the following code:

```
ts.plot( err, col="red", main="error rate" )
lines(err1, col="blue")
lines(err2, col="green")
```

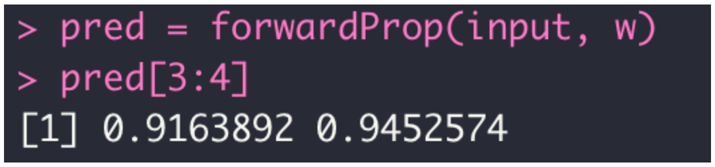


Red line is learning rate = 0.1, Blue line is learning rate = 0.6, Green line is learning rate = 1.2

Also, I calculated prediction results for 3 learning rates using following codes:

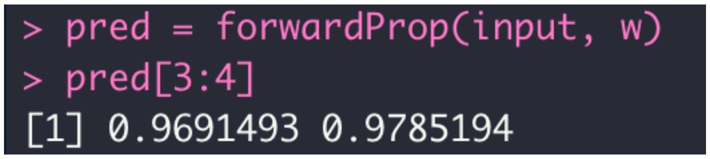
pred = forwardProp(input, w)
pred[3:4]

For learning rate = 0.1, this gave me:



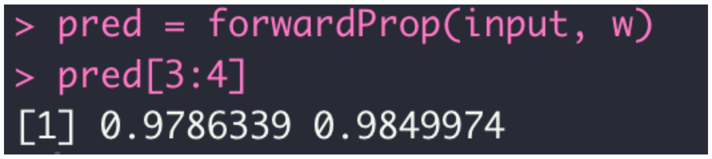
```
> pred = forwardProp(input, w)
> pred[3:4]
[1] 0.9163892 0.9452574
```

For learning rate = 0.6, this gave me:



```
> pred = forwardProp(input, w)
> pred[3:4]
[1] 0.9691493 0.9785194
```

For learning rate = 1.2, this gave me:



```
> pred = forwardProp(input, w)
> pred[3:4]
[1] 0.9786339 0.9849974
```

As mentioned in the lecture, bigger learning rate makes the neural network to converge more quickly to the actual target values. This is because having larger learning rate allows the back propagation to adjust in a large amount in the process of gradient descent. If one has a smaller learning rate, 1000 iteration might not be enough to fully adjust to the target value using gradient descent. However, large learning rate do have some problem. As the adjustment made by a larger learning rate is lot bigger, it can lead to overshooting. That is, it can make gradient descent process to jump over the minima.

To summarize, larger learning rate allows the neural network to converge more faster to the target value. But larger learning rate could lead to overshooting. Thus, it might be best to first start with a large learning rate and then later reduce it to prevent overshooting.