h5_514

Homework 5 Problem Statement

This assignment will take more time than previous assignments. Function prototypes are not provided since they should be familiar from earlier assignments

In this assignment you will implement a neural-network with 1 hidden layer. The model will have 2 sets of biases and 2 sets of weights.

The code should be configurable so the output layer can support linear, logistic or softmax models. In our solution, we reset global function names based on model type. For example, say your program uses the function name "cost" for the cost function. Also, assume you write "cost.squared.error" for squared error, "cost.negll" for negative log-likelihood and "cost.cross.entropy" for cross-entropy. (You have written each of these in previous assignments)

To configure for a linear model: cost=cost.squared.error To configure for a logistic model: cost=cost.negll To configure for a softmax model: cost=cost.cross.entropy

We found it necessary to configure 4 global functions 1. cost - the cost function 2. f - the hidden layer activation function, used in fwd.prop 3. df - the derivative of the hidden layer activation function, used in bk.prop 4. g - the output layer activation function, used in fwd.prop

There are other ways to make your solution configurable, so please feel free to use a different approach.

Other important functions for this homework will be: - bk.prop - fwd.prop - init.wgts - nnet1.fit - nnet1.fit.batch - fit - predict

Step 1 Generate data

Use the mlbench library to generate a spiral data set for binary classification.

```
setwd("G:\\math\\514")
source('hw5_514.R')
```

```
## Warning: package 'mlbench' was built under R version 3.5.3
```

```
## Loading required package: mvtnorm
```

```
## Loading required package: Matrix
```

```
## Warning: package 'Matrix' was built under R version 3.5.3
```

```
## Loading required package: beepr
```

```
library(mlbench)
data=mlbench.spirals(75,1.5,.07)
plot(data)
X=t(data$x)
Y=matrix(as.integer(data$classes)-1,nrow=1)
```

Step 2 implement 3 cost functions

- Squared error for a linear model
- Negative log-likelihood for a logistic model
- Cross-entropy for a softmax model

```
print.fun('cost.squared.error')
```

```
## [1] "cost.squared.error (X, Y, b, w) "
## {
```

```
##     yhat <- Identity(X, b, w)
##     M = length(yhat)
##     e = matrix(Y - yhat, ncol = 1)
##     (1/(2 * M)) * sum(e^2)
## }
```

```
print.fun('cost.negll')
```

```
## [1] "cost.negll (X, Y, b, w) "
## {
##     (-1/length(as.vector(Y))) * sum(Y * log(Sigmoid(X, b, w)) +
##         (1 - Y) * log(1 - Sigmoid(X, b, w)))
## }
```

```
print.fun('cost.cross.entropy')
```

```
## [1] "cost.cross.entropy (X, Y, b, w) "
## {
##     H <- stable.softmax(X, b, w)
##     (-1/dim(X)[2]) * sum(colSums(one.hot(Y) * log(H)))
## }
```

Step 3 Implement 3 output activation functions

- Use the identity function for a linear model
- Use the sigmoid function for a logistic model
- Use the numerically stable softmax function for a softmax model

```
print.fun('identity')
```

```
## [1] "identity (x) "
## x
```

```
print.fun('sigmoid')
```

```
## [1] "sigmoid (x) "
## 1/(1 + exp(-x))
```

```
print.fun('stable.softmax')
```

```
## [1] "stable.softmax (X, b, w) "
## {
##     Z <- (b %*% matrix(rep(1, dim(X)[2]), nrow = 1) + w %*% X)
##     Z <- exp(sweep(Z, 2, apply(Z, 2, max)))
##     H <- Z %*% (1/colSums(Z) * Diagonal(dim(Z)[2]))
##     return(as.matrix(H))
## }
```

Step 4 Implement 3 hidden layer activation functions

Implement the following activation functions along with their derivatives

- ReLU activation
- tanh activation (built-in R function)
- sigmoid activation

test each of the derivatives using a numerical gradient calculation on activation function

```
print.fun('sigmoid')
```

```
## [1] "sigmoid (x) "
```

```
## 1/(1 + exp(-x))
```

```
print.fun('relu')
```

```
## [1] "relu (X) "
## {
##     max(c(0, X))
## }
```

```
print.fun('tanh')
```

```
## [1] "tanh (x) "
## NULL
```

```
print.fun('dsigmoid')
```

```
## [1] "dsigmoid (X) "
## {
##     sigmoid(X) * (1 - sigmoid(X))
## }
```

```
print.fun('drelu')
```

```
## [1] "drelu (X) "
## {
##     ifelse(X < 0, 0, 1)
## }
```

```
print.fun('dtanh')
```

```
## [1] "dtanh (X) "
## {
##     1 - tanh(X)^2
## }
```

```
# check sigmoid
 d=runif(10)
 (sigmoid(d+10^-8)-sigmoid(d-10^-8))/(2*10^-8)-dsigmoid(d)
```

```
##  [1] -6.839456e-09 -1.177785e-09  4.493420e-11 -4.147017e-09 -2.461071e-09
##  [6]  2.692846e-09  2.062070e-09  8.981876e-09  2.620151e-11  1.883832e-09
```

```
#check relu
 (relu(d+10^-8)-relu(d-10^-8))/(2*10^-8)-drelu(d)
```

```
##  [1] 5.024759e-09 5.024759e-09 5.024759e-09 5.024759e-09 5.024759e-09
##  [6] 5.024759e-09 5.024759e-09 5.024759e-09 5.024759e-09 5.024759e-09
```

```
#check tanh
 (tanh(d+10^-8)-tanh(d-10^-8))/(2*10^-8)-dtanh(d)
```

```
##  [1] -1.998151e-09  1.684647e-09  2.301412e-09  4.658476e-09 -2.481003e-09
##  [6]  6.251919e-09 -1.963393e-09  1.255780e-10 -1.902409e-09 -8.857520e-10
```

Step 5a Implement fwd.prop/bk.prop

- fwd.prop: Propagates input data matrix through the network and produces the output activation values needed by the cost functions. Note that bk.prop needs intermediate values computed by fwd.prop.
- bk.prop: Computes analytical gradients of the cost function w.r.t model parameters.

```
print.fun('fwd.prop')
```

```
## [1] "fwd.prop (X, L, W, B, activation, output) "
## {
##     A <- Z <- list()
##     for (i in 1:L) {
##         if (i == 1) {
##             AA <- X
##         }
##         else {
##             AA <- A[[i - 1]]
##         }
##         Z[[i]] <- B[[i]] %*% rep(1, dim(AA)[2]) + (W[[i]]) %*%
##             AA
##         A[[i]] <- apply(Z[[i]], c(1, 2), activation)
##     }
##     Z[[L + 1]] <- B[[L + 1]] %*% rep(1, dim(AA)[2]) + W[[L +
##         1]] %*% A[[L]]
##     A[[L + 1]] <- output(A[[L]], B[[L + 1]], W[[L + 1]])
##     return(list(A = A, Z = Z))
## }
```

```
print.fun('bk.prop')
```

```
## [1] "bk.prop (X, Y, L, W, B, Z, A, Acti) "
## {
##     Act <- as.character(substitute(Acti))
##     if (Act == "relu") {
##         derivative <- drelu
##     }
##     else if (Act == "tanh") {
##         derivative <- dtanh
##     }
##     else {
##         derivative <- dsigmoid
##     }
##     m <- length(as.vector(Y))
##     dZ <- dW <- dB <- list()
##     A[[length(A) + 1]] <- X
##     A <- A[c(length(A), 1:(length(A) - 1))]
##     ell <- L + 1
##     if (length(as.vector(unique(y))) > 2) {
##         dZ[[ell]] <- A[[ell + 1]] - one.hot(Y)
##     }
##     else {
##         dZ[[ell]] <- A[[ell + 1]] - Y
##     }
##     while (ell >= 1) {
##         dW[[ell]] <- (1/m) * dZ[[ell]] %*% t(A[[ell]])
##         dB[[ell]] <- (1/m) * dZ[[ell]] %*% rep(1, m)
##         if (ell > 1) {
##             dZ[[ell - 1]] <- t(W[[ell]]) %*% dZ[[ell]] * apply(Z[[ell -
##                 1]], c(1, 2), derivative)
##         }
```

```
##         ell <- ell - 1
##     }
##     return(list(dZ = dZ, dB = dB, dW = dW))
## }
```

Step 5b Numerically check gradients

-Implement a numerical gradients function and use it to test each of the 3 cost functions. You can used tanh for the hidden activation function for the gradient checking for all 3 cost functions.

- Test gradients for squared error cost with identity output activation using data from data.lawrence.giles function provided
- Test gradients for negative log-likelihood error with sigmoid output activation using data from spiral data
- Test gradients for cross-entropy error with numerically stable softmax output activation using data for 3 class mixture provided below

Step 6 implement a training algorithm

You will code two training algorithms. The first will do simple gradient descent using the full input data to compute gradients. As in previous assignments, you will find it necessary to save information after each epoch. Implement a function that trains using the full input data called nnet1.fit

For training of your neural net the weights and bias are initialized. The init.wgts function is provided in the R file. Feel free to write a more general function that allows experimentation with different types of bias/weight initialization

```
print.fun('init.wgts')
```

```
## [1] "init.wgts (n.in, n.hid, n.out) "
## {
##     b1 = runif(n.hid, -0.1, 0.1)
##     w1 = matrix(rnorm(n.in * n.hid, 0, 0.1), nrow = n.hid, ncol = n.in)
##     b2 = runif(n.out, -0.1, 0.1)
##     w2 = matrix(rnorm(n.out * n.hid, 0, 0.1), nrow = n.out, ncol = n.hid)
##     list(b1 = b1, w1 = w1, b2 = b2, w2 = w2)
## }
```

```
# call init.wgts - data samples are arranged by column
model=init.wgts(n.in=nrow(X),n.hid=1,n.out=nrow(Y))

print.fun('nnet1.fit')
```

```
## [1] "nnet1.fit (X, Y, HL, nodes, Nsim, MaxLR = 1, Activation = relu, "
## {
##     LR <- MaxLR
##     Acts <- as.character(substitute(Activation))
##     Outpt <- as.character(substitute(Output))
##     WB <- init.wgt(HL, nodes, X, Y)
##     W <- WB$W
##     B <- WB$B
##     if (Acts == "relu") {
##         Derivative <- drelu
##     }
##     else if (Acts == "tanh") {
##         Derivative <- dtanh
##     }
##     else {
```

```
##         Derivative <- dsigmoid
##     }
##     C1 <- 0
##     Costs <- rep(NA, Nsim)
##     M <- length(as.vector(Y))
##     ST <- system.time(for (i in 1:Nsim) {
##         FP <- fwd.prop(X, HL, W, B, Activation, Output)
##         C2 <- Costs[i] <- Cost(Y, FP$A[[HL + 1]], Outpt)
##         BP <- bk.prop(X, Y, HL, W, B, FP$Z, FP$A, Activation)
##         B.OLD <- B
##         W.OLD <- W
##         for (j in 1:(HL + 1)) {
##             B[[j]] <- B[[j]] - LR * BP$dB[[j]]
##             W[[j]] <- W[[j]] - LR * BP$dW[[j]]
##             if (is.nan(FP$A[[HL + 1]][1]) == T) {
##                 break
##             }
##         }
##         if (C1 < C2) {
##             i = i - 1
##             LR <- LR * 0.5
##             B <- B.OLD
##             W <- W.OLD
##         }
##         else {
##             LR <- LR * 1.1
##         }
##         C1 <- C2
##         LR <- ifelse(LR > MaxLR, MaxLR, LR)
##     })
##     return(list(st = ST, yhat = FP$A[[HL + 1]], costs = C2, W = W,
##         B = B))
## }
```

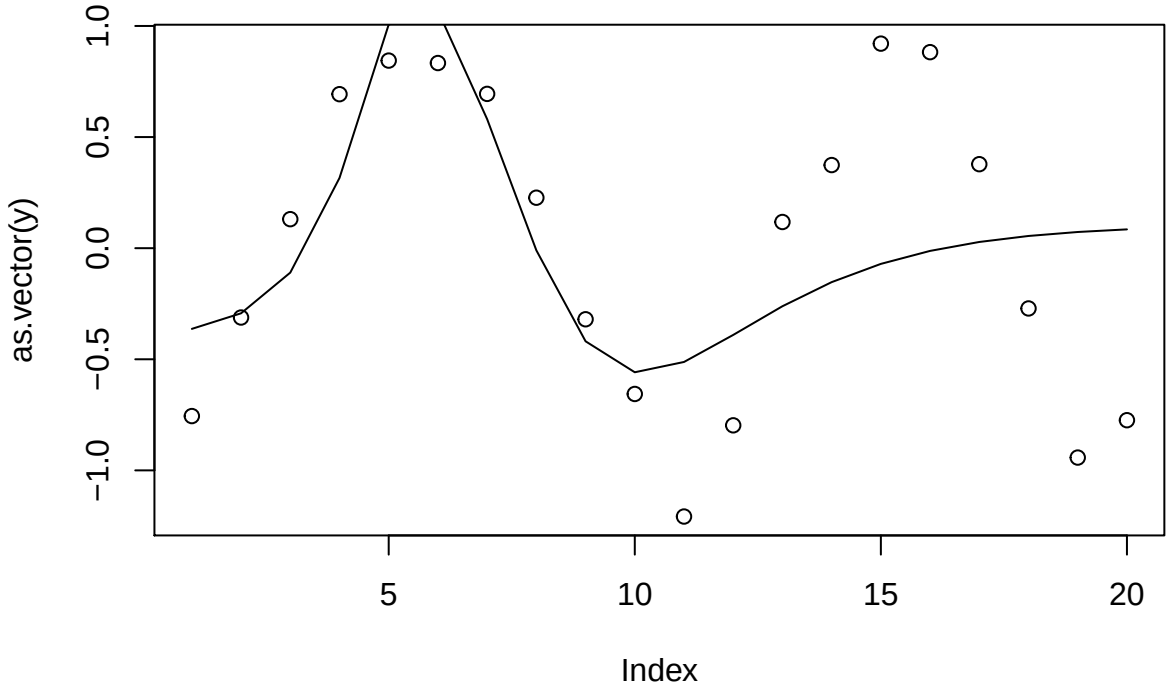
Step 7 Train linear model using Giles/Lawrence data

Fit 3 linear models (squared error) to the Lawrence/Giles data with 2, 5 and 100 hidden nodes respectively. Use the tanh hidden layer activation function. Plot the resulting fits and compare them to an order 15 polynomial fit. You should find that adding lots of hidden nodes does not cause overfitting.

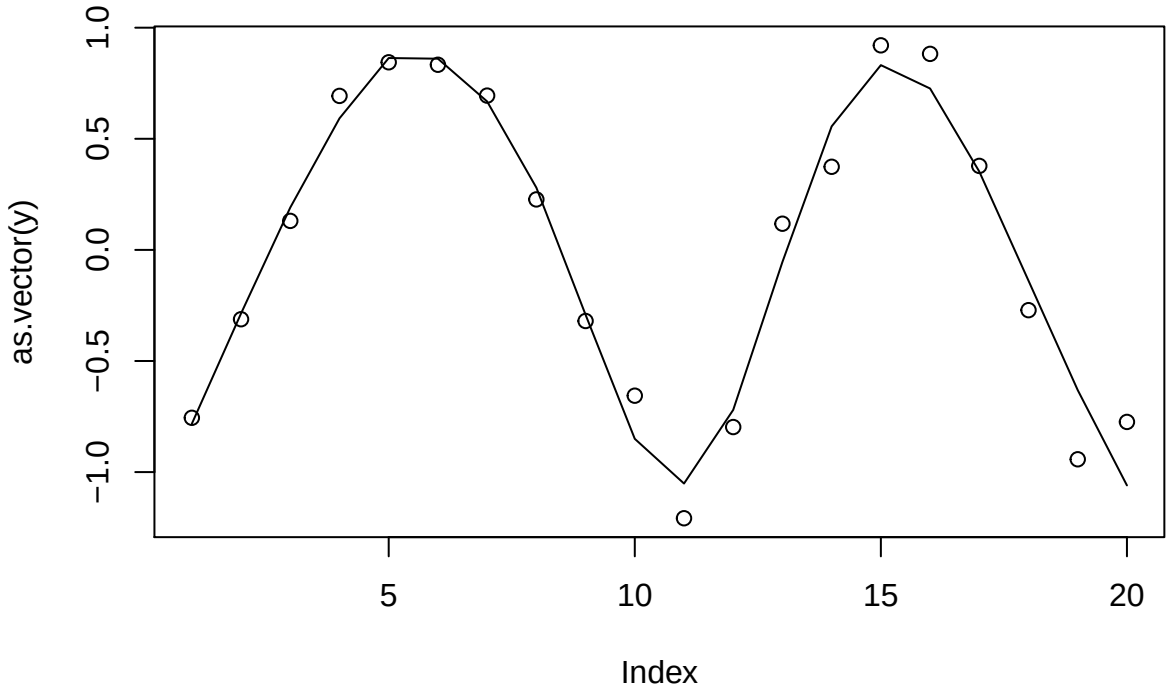
For our runs we used 150,000 iterations with a learning rate of .1. Feel free to experiment with other parameters and other activation functions.

```
  c(x,y,xgrid,ygrid) %<-% data.lawrence.giles(12345)

  np=length(X)

x.set=c(X)
y.set=c(Y)
degree=15
lm.fit = lm(y ~ poly(x,degree,raw=FALSE), data=data.frame(y=y.set,x=x.set))
#y = predict.lm(lm.fit,data.frame(x=xgrid))
#points(xgrid,y,type="l",col="black",lwd=2)
#legend("topright", legend = c(num_hidden,paste("degree=",degree)), col = colors,lwd=2 )
```

```
fit.LG1 <- nnet1.fit( x, y, 1 , 2 , 150000, 1.5, 1.5, Activation = tanh, Output = Identity );  #beep("c
plot(as.vector(y))
lines(as.vector(fit.LG1$yhat ))
```

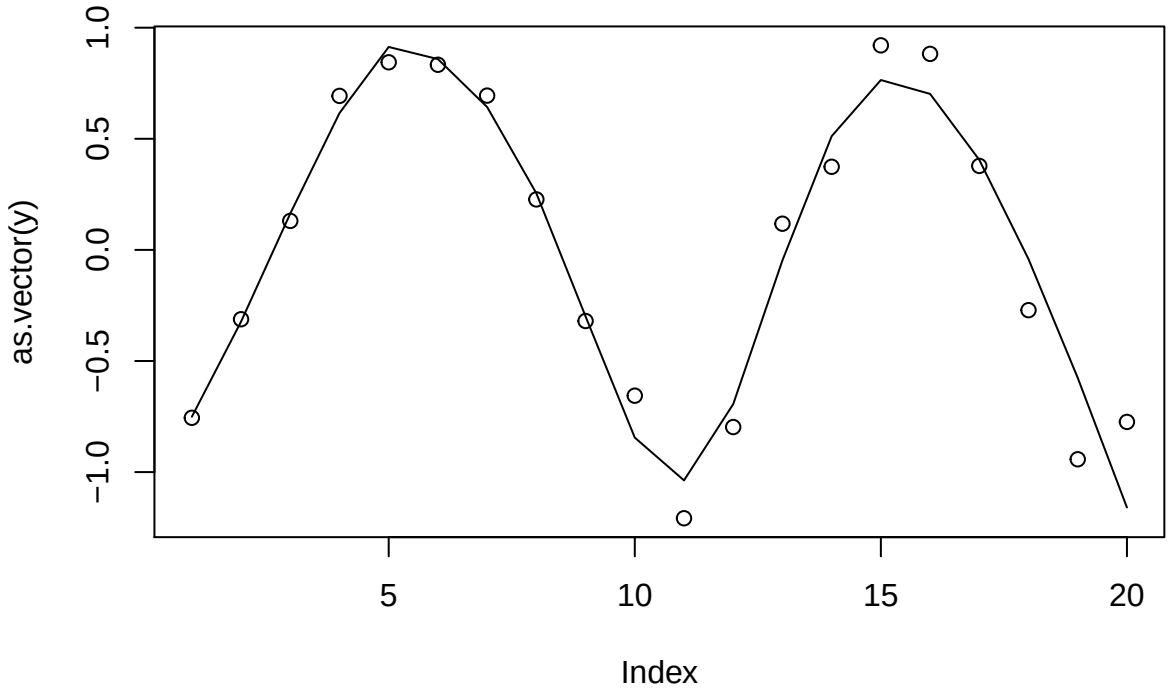



```
fit.LG2 <- nnet1.fit( x, y, 1 , 5 , 30000, 1.5, 1.5, Activation = tanh, Output = Identity );  #beep("co
plot(as.vector(y))
lines(as.vector(fit.LG2$yhat ))
```




```
fit.LG3 <- nnet1.fit( x, y, 1 , 100 , 60000, 1.5, 1.5, Activation = tanh, Output = Identity );  #beep("
```

```
plot(as.vector(y))
lines(as.vector(fit.LG3$yhat ))
```



Step 8 Train binary classifier using spiral data

Train a neural net to predict binary classes for the mlbench.spirals data introduced earlier. Using the neg log likelihood cost function compare performance of the 3 activation functions (tanh, sigmoid, and relu) for 5 vs 100 hidden units.

- Train a neural net on the spiral binary classification data using a sigmoid activation function for the output layer and compare tanh, sigmoid and relu results for 5 vs 100 hidden units. For relu and tanh use a learning rate of 0.5 and for sigmoid use a learning rate of 3
- Plot the performance histories of all 6 model combinations in one plot
- look at the decision boundary, cost history and performance history of relu with 100 hidden units and sigmoid models with 100 hidden units with a learning rate of 3. You will notice that relu model converges faster with fewer hidden units than relu

```
spirals <- spiralpred <- mlbench.spirals(75,1.5,.07)

y <- as.numeric(spirals$classes) - 1
x <- t(spirals$x )
```

```
#fit.Sp1 <- nnet1.fit( x, y , 1 , 5 ,  100000 ,  .5 ,  .5 , Activation = tanh, Output = Sigmoid ); #bee
```

```
fit.sp2 <- nnet1.fit( x, y , 1 , 5 ,  200000 ,  .5 ,  .5 ,  Activation = relu, Output = Sigmoid ); #bee
```

```
fit.sp3 <- nnet1.fit( x, y , 1 , 5 ,  200000 ,  .5 ,  .5 ,  Activation = sigmoid, Output = Sigmoid ); #
```

```
#fit.sp4 <- nnet1.fit( x, y , 1 , 100 ,  100000 ,  .5 ,  .5 ,   Activation = tanh, Output = Sigmoid );
```

```
fit.sp5 <- nnet1.fit( x, y , 1 , 100 ,  200000 ,  .5 ,  .5 ,    Activation = relu, Output = Sigmoid );
```

```
fit.sp6 <- nnet1.fit( x, y , 1 , 100 ,  200000 ,  .5 ,  .5 ,    Activation = sigmoid, Output = Sigmoid
```

```
spfit.pred2 <- ifelse( fit.sp2$yhat  > .5 , 1 , 0    )
sum(( y  ==   as.numeric( spfit.pred2  ) ) * 1 ) / length(y)
```

```
## [1] 0.6533333
```

```
spfit.pred3 <- ifelse( fit.sp3$yhat  > .5 , 1 , 0    )
sum(( y  ==   as.numeric( spfit.pred3  ) ) * 1 ) / length(y)
```

```
## [1] 0.8
```

```
spfit.pred5 <- ifelse( fit.sp5$yhat  > .5 , 1 , 0    )
sum(( y  ==   as.numeric( spfit.pred5  ) ) * 1 ) / length(y)
```

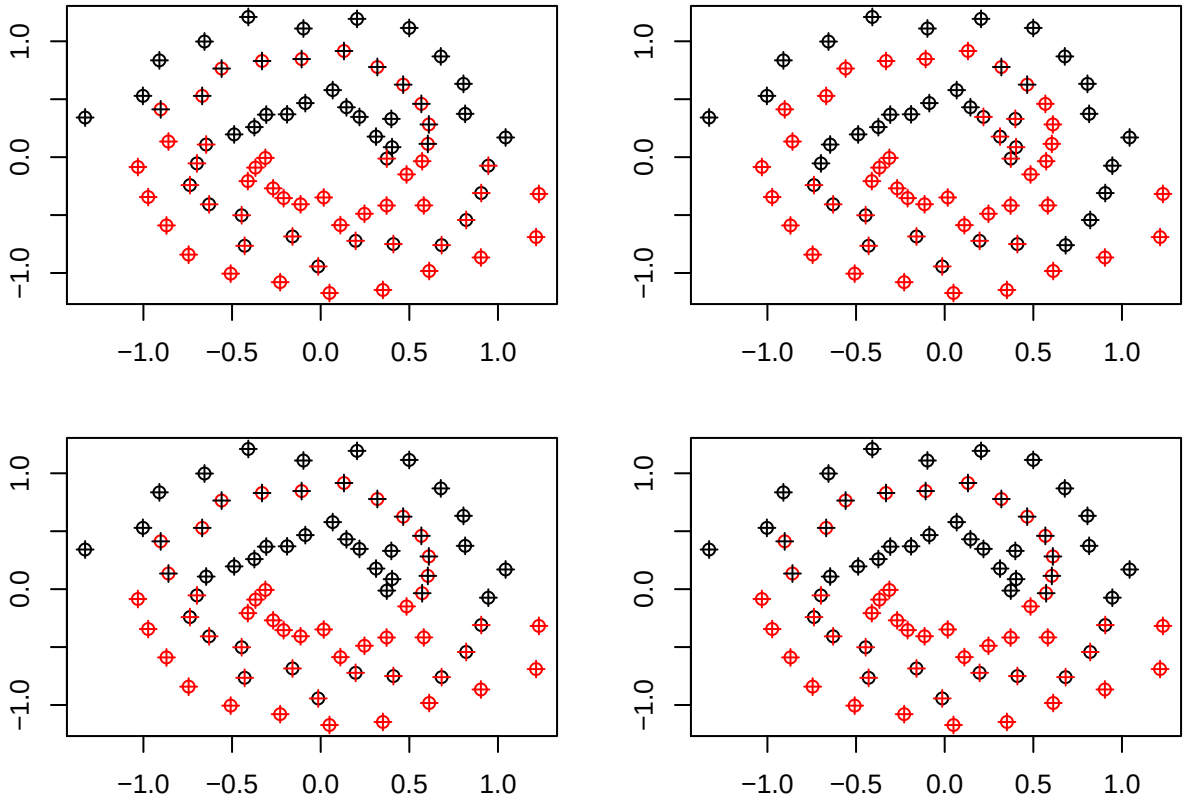
```
## [1] 0.6666667
```

```
spfit.pred6 <- ifelse( fit.sp6$yhat  > .5 , 1 , 0    )
sum(( y  ==   as.numeric( spfit.pred6  ) ) * 1 ) / length(y)
```

```
## [1] 0.6666667
```

```
#fit.sp5 <- nnet1.fit( x, y , 3 , c(25,25) ,  200000 ,  1.5 ,       Activation = sigmoid, Output = Sigm
```

```
par(mfrow=c(2,2) , mar=c(2.1,2.1,2.1,2.1) )
plot(spirals)
spiralpred$classes   <- as.factor( spfit.pred2 + 1 )
points( spiralpred$x , col = spiralpred$classes    , pch = 3 )
plot(spirals)
spiralpred$classes   <- as.factor( spfit.pred3 + 1 )
points( spiralpred$x , col = spiralpred$classes    , pch = 3 )
plot(spirals)
spiralpred$classes   <- as.factor( spfit.pred5 + 1 )
points( spiralpred$x , col = spiralpred$classes    , pch = 3 )
plot(spirals)
spiralpred$classes   <- as.factor( spfit.pred6 + 1 )
points( spiralpred$x , col = spiralpred$classes    , pch = 3 )
```



Step 9 Train a softmax model on MNIST data

Train a neural network on the 60k MNIST training data and measure performance on test set using cross-entropy cost function and relu activation. Look at performance and plot of performance history. Use a learning rate of 0.15 and 30 hidden units and 50 epochs.

-Plot in sample and out of sample performance history of the model

```
mtrain <- read.csv("G:\\math\\mnist_train.csv" , header = F)
x <- unname( mtrain[,-1] )
x <- t(x)
x <- x / 255
y <-  as.matrix( unname(  mtrain[,1] ))
```

```
mnist.fit <- nnet1.fit( x, y, 1, 30, 50 ,  MaxLR = .5,
                 Activation = relu,   Output = stable.softmax )
```

```
yhat.mnist <- ( apply( mnist.fit$yhat , 2 , function( M )  which( M == max(M) ) )  - 1)
sum( 1 * (yhat.mnist == y ) ) / length(as.vector(y))
```

```
## [1] 0.6900833
```

Step 10 Implement a mini-batch gradient descent training algorithm

Mini-batch gradient descent converges faster than gradient descent and is a necessity when data sets are large. Re-implement your training function from step 6 as nnet1.fit.batch to break each epoch into a set of mini-batches. nnet1.fit.batch will need an additional batch.size parameter.

Mini-batch GD adds a loop inside the epoch/iterations loop. Your mini-batches should divide the dataset into randomly chosen samples of size batch.size. It is also best practices to use different random samples for

each epoch.

```
print.fun('nnet1.fit.batch')
```

```
## [1] "nnet1.fit.batch (X, Y, HL, Batches, nodes, Nsim, MaxLR = 1, Activation, "
## {
##     LR <- MaxLR
##     WB <- init.wgt(HL, nodes, X)
##     W <- WB$W
##     B <- WB$B
##     Acts <- as.character(substitute(Activation))
##     Outpt <- as.character(substitute(Output))
##     if (Acts == "relu") {
##         Derivative <- drelu
##     }
##     else if (Acts == "tanh") {
##         Derivative <- dtanh
##     }
##     else {
##         Derivative <- dsigmoid
##     }
##     C1 <- 0
##     Costs <- list()
##     M <- length(as.vector(Y))
##     Xt <- Yt <- list()
##     BS <- matrix(0, round(M/Batches), Batches)
##     BP <- list()
##     ij <- 1
##     Costs <- rep(NA, Nsim * Batches)
##     db <- dw <- list()
##     C2 <- 100
##     sm <- 0.001
##     Perf <- rep(NA, Nsim)
##     ST <- system.time(for (i in 1:Nsim) {
##         BS[1:M] <- sample(M)
##         for (k in 1:Batches) {
##             Xt[[k]] <- X[, BS[BS[, k] > 0, k]]
##             Yt[[k]] <- Y[BS[BS[, k] > 0, k]]
##         }
##         for (v in 1:length(Xt)) {
##             FP <- fwd.prop(Xt[[v]], HL, W, B, Activation, Output)
##             C2 <- Costs[ij] <- Cost(Yt[[v]], FP$A[[HL + 1]],
##                 Outpt)
##             ij <- ij + 1
##             BP <- bk.prop(Xt[[v]], Yt[[v]], HL, W, B, FP$Z, FP$A,
##                 Activation)
##             for (j in 1:(HL + 1)) {
##                 B[[j]] <- B[[j]] - (LR) * BP$dB[[j]]
##                 W[[j]] <- W[[j]] - (LR) * BP$dW[[j]]
##             }
##         }
##         Perf[i] <- nnet.Predict(Outpt, X, Y, HL, W, B, Activation)
##         if (Perf[i] == 1) {
##             break
##         }
```

```
##     })
##     return(list(performance = Perf[!is.na(Perf)], costs = Costs[!is.na(Costs)],
##         ST = ST, W = W, B = B, iter = i))
## }
```

** Step 11 Try 3 mini-batch sizes on spiral data **

- Compare cost histories and performance histories using MNIST and nnet1.fit.batch for mini-batch sizes of 32, 64 and 128
- how does this compare against the full gradient descent neural network model from step 9

```
spirals <- spiralpred <- mlbench.spirals(75,1.5,.07)

y <- as.numeric(spirals$classes) - 1
x <- t(spirals$x )

sp.fit.b1 <- nnet1.fit.batch( x  , y , 1 , 3 ,  35, 50000, .5,
    Activation = sigmoid, Output = Sigmoid); beep("coin")
```

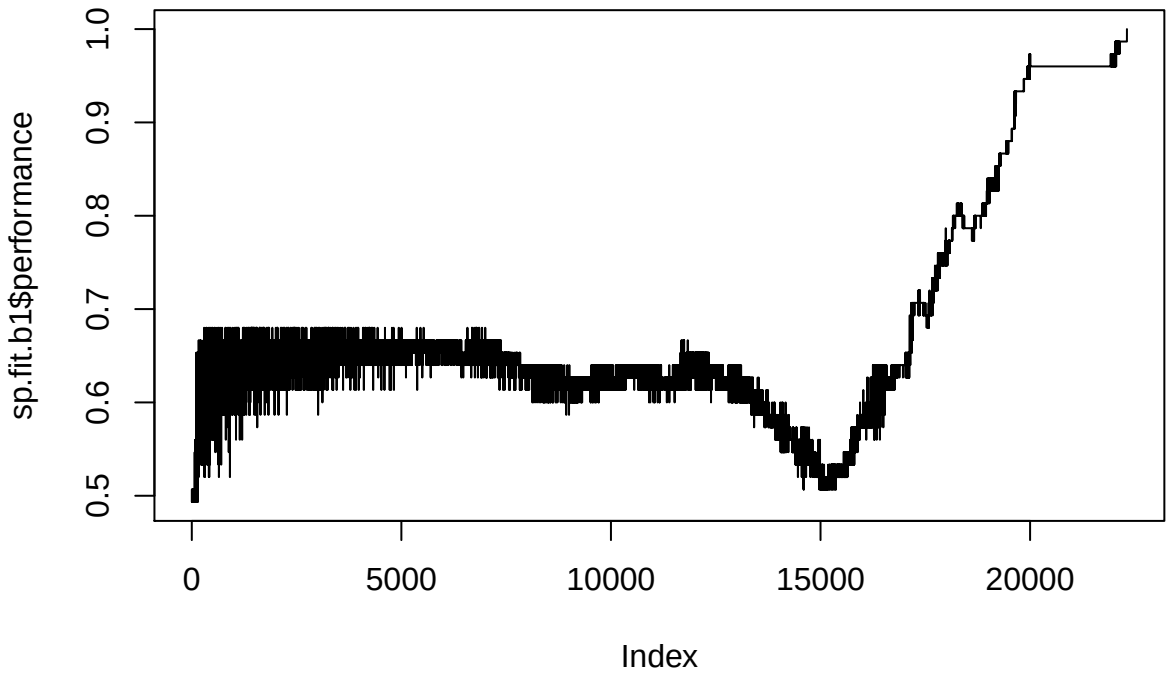
```
sp.fit.b1$iter
```

```
## [1] 22311
```

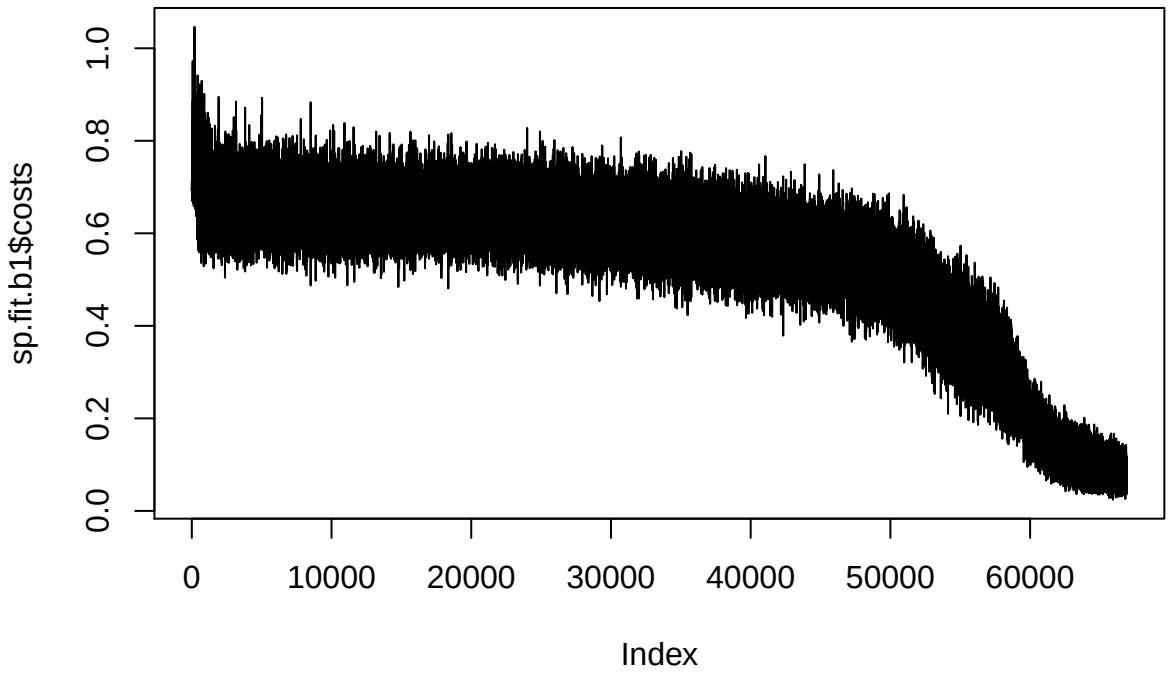
```
tail(sp.fit.b1$performance,1)
```

```
## [1] 1
```

```
plot(sp.fit.b1$performance, type="l")
```



```
plot(sp.fit.b1$costs, type="l" )
```




```
sp.fit.b2 <- nnet1.fit.batch( x  , y , 1 , 3 ,  35, 50000, .5,
    Activation = relu, Output = Sigmoid); beep("coin")
```

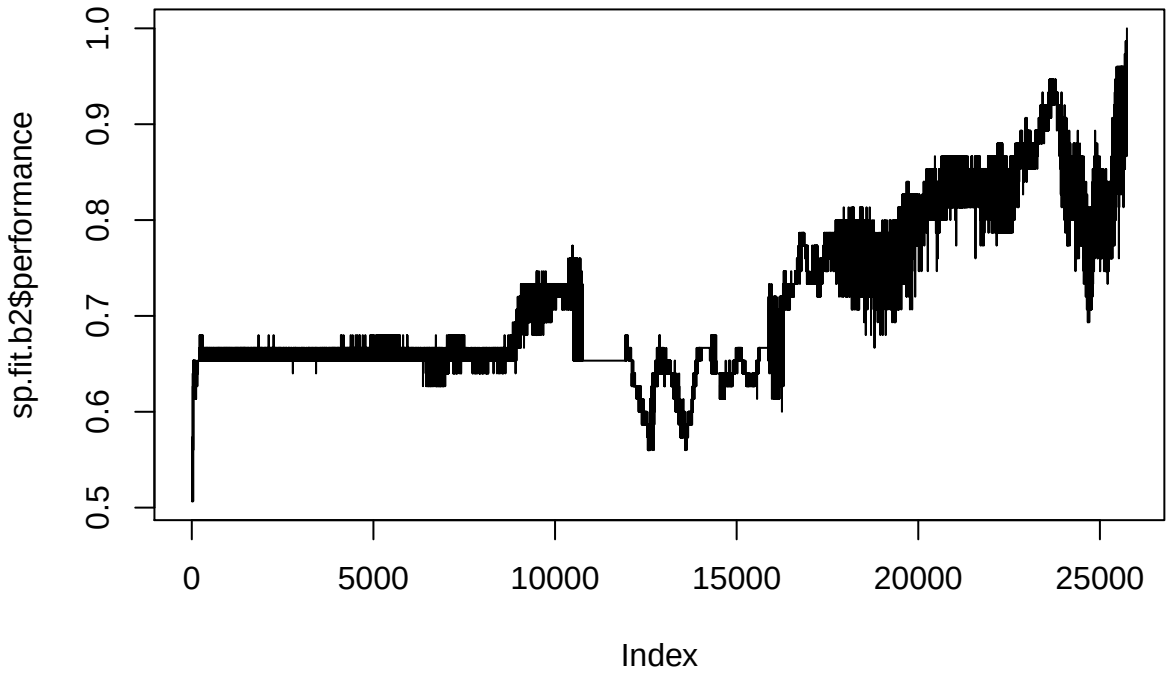
```
sp.fit.b2$iter
```

```
## [1] 25745
```

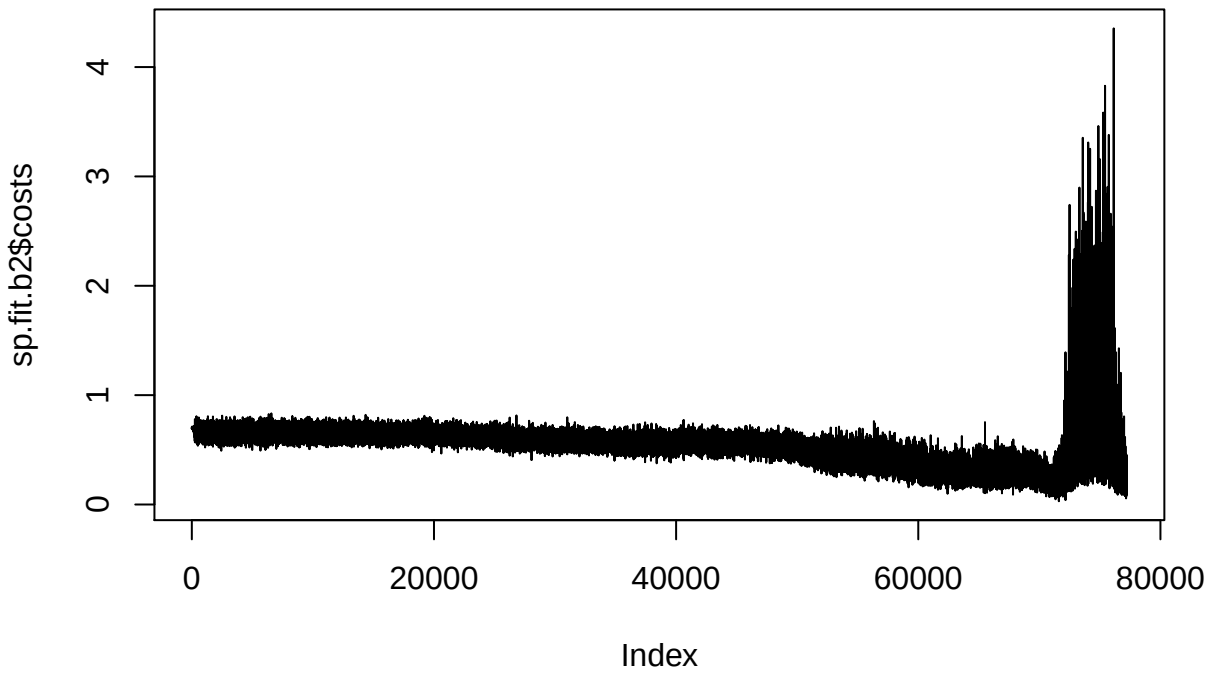
```
tail(sp.fit.b2$performance,1)
```

```
## [1] 1
```

```
plot(sp.fit.b2$performance, type="l")
```




```
plot(sp.fit.b2$costs , type="l")
```



** Step 12 train using mini-batch gradient descent on MNIST**

- Train a neural network using nnet1.fit.batch using a batch size of 30 with learning rate of 0.15 compare out of sample performance of this model against out of sample performance of neural net trained on full gradient descent. For the mini-batch model use 10 epochs and for the full model use 50

```
#mnist = list(train=load_image_file("train-images-idx3-ubyte"),test=load_image_file("t10k-images-idx3-u
#mnist[["train"]]$y = load_label_file("train-labels-idx1-ubyte")
#mnist[["test"]]$y = load_label_file("t10k-labels-idx1-ubyte")


train_set=sample(1:dim(mnist$train$x)[1], size = 60000)
X=t(mnist[["train"]]$x[train_set,])/255.
y=mnist[["train"]]$y[train_set]
Y=one.hot(y)

testX=t(mnist[["test"]]$x)/255.
testY=mnist[["test"]]$y

M=ncol(X)
b=numeric(10)
w=matrix(rnorm(10*784),nrow=10,ncol=784)

n = nrow(X)  # number of input nodes
n.out = nrow(Y)
M = ncol(X)  # number of samples
```

```
# it would be nice to add split cost/accuracy plot
nh=30
lr=.15
```

Step 13 Compare impact of hidden units on mini-batch

- Using minibatch of size 32 and learning rate of 0.15 compare the performance of the model for 5, 10,30 hidden units out of sample

```
lr=.15
perf=vector(mode='list')
counter=1
num_hidden =c(5,10,30)
```