ST340 Lab 8: Artificial neural networks

2019–20

1: OR

Here is an example of using R's `optim` function to learn the OR function. `optim` works best when you provide a function to calculate the gradient, but we will be lazy for now: look at `?optim`.

```
logistic <- function(x) {
  1/(1+exp(-x))
}

(or.x <- matrix(c(0,0,1,1,0,1,0,1),4,2))
(or.y <- c(0,1,1,1))

ann <- function(x,theta) {
  w=theta[1:2]
  b=theta[3]
  logistic( x %*% w + b ) # The coefficient multiplying '1' is often called the 'bias';
  # here denoted b
}

cost=function(theta) {
  o=ann(or.x,theta)
  sum((o-or.y)^2)
}

(theta0=rep(0,3))
(theta=optim(theta0,cost)$par)
ann(or.x,theta) # Should be approximately or.y
```

2: XOR

Adapt the above to learn the XOR function. Hint: you can write a two layer network in the efficient form

```
logistic(logistic(x %*% w + b) %*% w2 + b2)
```

where `w` is a matrix, `b` is a vector, `w2` is a vector, `b2` is a scalar, and the inner `logistic` is applied componentwise to a vector. You will probably find that the architecture for the OR example is not sufficiently big here; try adding a unit in the intermediate layer (then `theta` has length 13), and also randomising `theta0`.

3: ANNs

(a) Read through the following code to see how a basic ANN can be implemented in R.

```
# Used to store the data as it travels through the network
layer <- function() {
  a=new.env()
  # a$x will be used to store the input and hidden layers
```

```
  # a$dx stores the derivatives of the cost function with respect to l$x
  a
}

# Implement a 'layer' to apply the logistic function
logistic<-function() {
  e=new.env()
  e$forward<-function(a,z,train) {
# a is the input to the layer
# z is the output from the layer
# "train" is "true" during training, and false when the network is being tested on
# new data; most layers will ignore the "train" variable, but it is needed for consistency
z$x <- 1/(1+exp(-a$x))
  }
  e$backward<-function(a,z,learning.rate) {
a$dx <- z$dx * z$x * (1-z$x) # Backpropagation just through the logistic function. Check!
  }
  e
}

# Implement a fully connected layer: each output depends on every input.
fully.connected<-function(nIn,nOut) {
  # nIn is size of the input layer
  # nOut is the size of the output layer
  shape=c(nIn,nOut)
  e=new.env()
  e$w=array(runif(prod(shape),-0.1,0.1),shape) # Network parameters - connection weights
  e$mw=array(0,shape) # w-momentum
  shape[1]=1
  e$b=array(0,shape) # Network parameters - bias term
  e$mb=array(0,shape) # b-momentum
  e$forward <- function(a,z,train) {
z$x <- a$x %*% e$w + e$b[rep(1,dim(a$x)[1]),]
  }
  e$backward <- function(a,z,learning.rate) {
a$dx <- z$dx %*% t(e$w) # Backpropagation just through the 'linear combination' step. Check!
dw    <- t(a$x) %*% z$dx
db    <- apply(z$dx,2,sum)

w.new <- e$w - 0.1*learning.rate*dw + 0.9*e$mw
e$mw <- w.new - e$w
e$w <- w.new
b.new <- e$b - 0.1*learning.rate*db + 0.9*e$mb
e$mb <- b.new - e$b
e$b <- b.new
  }
  e
}

softmax.nll.classifier <- function(a,y) {
  weights <- exp(a$x-apply(a$x,1,max))  # (subtract column sums: better numerically)
  C <- apply(weights,1,sum)
  softmax <- weights/C
```

```
  predictions <- apply(softmax,1,which.max)-1
  errors <- sum(predictions!=y)
  target <- diag(dim(softmax)[2])[y+1,] # one-hot encoding of the true label
  a$dx <- softmax - target
  cost <- sum(-target*log(softmax),na.rm=TRUE) # negative log likelihood
  list(errors=errors,cost=cost)
}

train.classification <- function(nn,train.X,train.labels,batch.size,learning.rate) {
  errors <- 0
  cost<- 0
  layers <- replicate(length(nn)+1,layer())
  n=length(nn)
  n.reps=ceiling(dim(train.X)[1]/batch.size)
  for (rep in 1:n.reps) {
p=sample(dim(train.X)[1],batch.size)
if (length(dim(train.X))==2)
  layers[[1]]$x <- train.X[p,,drop=FALSE]
if (length(dim(train.X))==4)
  layers[[1]]$x <- train.X[p,,,,drop=FALSE]
y=train.labels[p]
for (i in 1:n) {
  nn[[i]]$forward(layers[[i]],layers[[i+1]],TRUE)
}
s <- softmax.nll.classifier(layers[[n+1]],y)
errors <- errors + s$errors
cost <- cost + s$cost
for (i in n:1) {
  nn[[i]]$backward(layers[[i]],layers[[i+1]],learning.rate)
}
  }
  print(paste("Training errors:",errors/n.reps/batch.size*100,"%   Cost:",
          cost/n.reps/batch.size))
}

test.classification <- function(nn,test.X,test.labels,batch.size) {
  errors <- 0
  layers <- replicate(length(nn)+1,layer())
  n=length(nn)
  n.test=dim(test.X)[1]
  n.reps=ceiling(n.test/batch.size)
  for (rep in 1:n.reps) {
p=(batch.size*(rep-1)+1):min(batch.size*rep,n.test)
if (length(dim(test.X))==2)
  layers[[1]]$x <- test.X[p,,drop=FALSE]
if (length(dim(test.X))==4)
  layers[[1]]$x <- test.X[p,,,,drop=FALSE]
y=test.labels[p]
for (i in 1:n) {
  nn[[i]]$forward(layers[[i]],layers[[i+1]],FALSE)
}
s <- softmax.nll.classifier(layers[[n+1]],y)
errors <- errors + s$errors
```

```
  }
  print(paste("Test errors:",errors/dim(test.X)[1]*100,"%"))
}
```

(b) Run this code on the MNIST data using a small, fully-connected network.

```
load("mnist.RData")
train.X <- train.X/255
test.X <- test.X/255
ls()
input.dim <- dim(train.X)[2] #784
n.classes <- max(train.labels)+1 #10
hidden.layer.size <- 100
batch.size <- 100
learning.rate <- 0.001
nn=list(
  fully.connected(input.dim,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,n.classes)
)
for (i in 1:100) { # <- Increase this if you have time
  train.classification(nn,train.X,train.labels,batch.size,learning.rate)
  # (Expect 90% error initially)
  test.classification(nn,test.X,test.labels,batch.size)
}
```

(c) Run this code on the CIFAR-10 subset using a small, fully-connected network.

```
load("frog-horse.RData")
train.X <- train.X/255
test.X <- test.X/255
ls()
input.dim <- dim(train.X)[2] #3072
n.classes <- max(train.labels)+1 #2
hidden.layer.size <- 100
batch.size <- 100
learning.rate <- 0.001
nn=list(
  fully.connected(input.dim,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  logistic(),
  fully.connected(hidden.layer.size,n.classes)
)
for (i in 1:100) { # <- Increase this if you have time
  train.classification(nn,train.X,train.labels,batch.size,learning.rate)
  # (Expect 50% error rate initially)
  test.classification(nn,test.X,test.labels,batch.size)
}
```

4: Alternative activation functions

Fill in the gaps below to create two functions that can be used instead of the logistic function defined above.

```
Tanh<-function() { # tanh nonlinearity
  e=new.env()
  e$forward<-function(a,z,train) {
    z$x <- tanh(-a$x)  # Use tanh instead of the logistic function
  }
  e$backward<-function(a,z,learning.rate) {
    a$dx <- # [use the chain rule to calculate a$dx in terms of z$dx, a$x and z$x]
  }
  e
}

relu<-function() { # Rectified Linear Units -- positive part function nonlinearity
  e=new.env()
  e$forward<-function(a,z,train) {
    z$x <- # [apply the positive part function to a$x; see
      # http://en.wikipedia.org/wiki/Positive_and_negative_parts]
  }
  e$backward<-function(a,z,learning.rate) {
    a$dx <- # [use the chain rule to calculate a$dx in terms of z$dx, a$x and z$x]
  }
  e
}

load("mnist.RData")
train.X <- train.X/255
test.X <- test.X/255
ls()
input.dim <- dim(train.X)[2] #784
n.classes <- max(train.labels)+1 #10
hidden.layer.size <- 100
batch.size <- 100
learning.rate <- 0.001
nn=list(
  fully.connected(input.dim,hidden.layer.size),
  relu(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  relu(),
  fully.connected(hidden.layer.size,hidden.layer.size),
  relu(),
  fully.connected(hidden.layer.size,n.classes)
)
for (i in 1:100) { # <- Increase this if you have time
  train.classification(nn,train.X,train.labels,batch.size,learning.rate)
  # (Expect 90% error initially)
  test.classification(nn,test.X,test.labels,batch.size)
}
```