COMP755-Lect22

November 13, 2018

1 COMP 755

Today:
Fitting neural network's parameters using back-propagation

2 (Deep) Neural networks compute through forward propagation

Deep networks are built up out of layers.

3 (Deep) Neural networks compute through forward propagation

Let f be an activation function and

$$\mathbf{f}(\mathbf{z})_i = f(z_i).$$

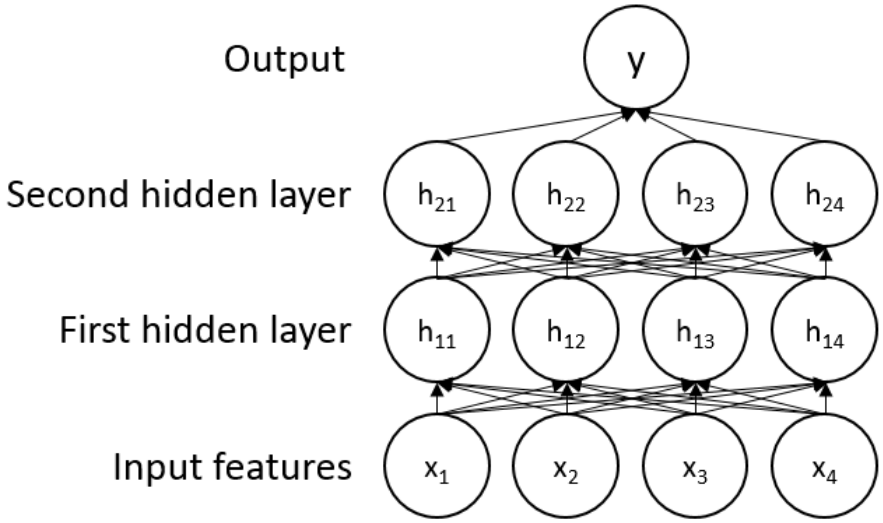
We will denote number of variables in layer l as n_l.
Size of the zeroth layer, n_0, is number of features in a sample.
With each layer, $l > 0$, we assoiate a matrix of size $\mathbf{W}^l : n_l \times n_{l-1}$ and a bias vector $\mathbf{b}^l : n_l$
Recursive specification for forward propagation is given by

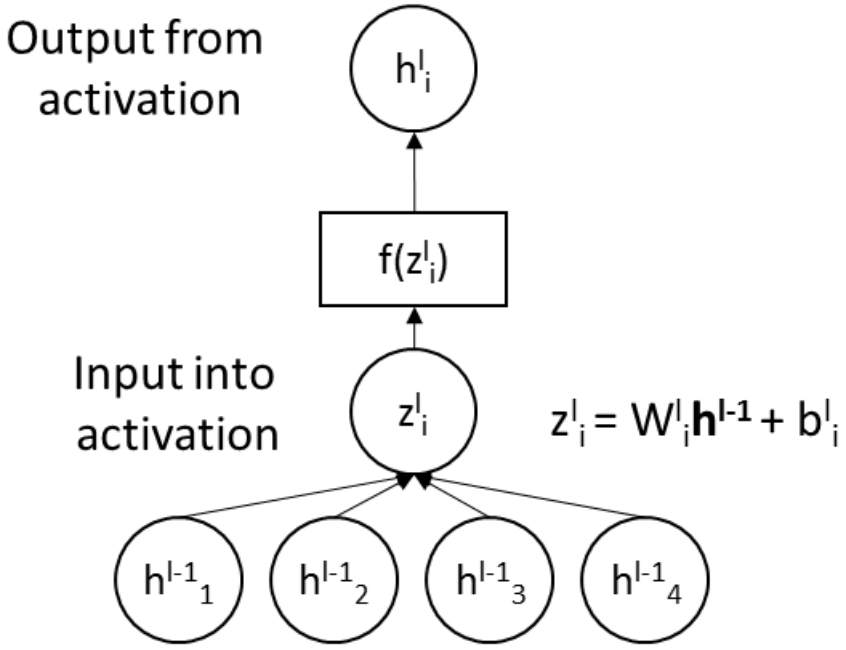
$$\mathbf{h}^l = \mathbf{f}(\mathbf{W}^1\mathbf{h}^{l-1} + \mathbf{b}^l).$$

4 Forward propagation restated

We restate forward propagation to make it easier to compute derivatives

$$\mathbf{h}^0 = \mathbf{x} \tag{1}$$
$$\mathbf{z}^l = \mathbf{W}^l\mathbf{h}^{l-1} + \mathbf{b}^l \tag{2}$$
$$\mathbf{h}^l = \mathbf{f}(\mathbf{z}^l). \tag{3}$$


Output
y
Second hidden layer
h21
h22
h23
h24
First hidden layer
h11
h12
h13
h14
Input features
x1
x2
x3
x4


Output from activation
h^l_i
f(z^l_i)
Input into activation
z^l_i
z^l_i = W^l_i h^l-1 + b^l_i
h^l-1_1
h^l-1_2
h^l-1_3
h^l-1_4

5 Common loss functions for Deep neural nets

Output of the network is $\hat{y}(\mathbf{x}) = \mathbf{h}^L$, where L is the depth of the network.

In order to asses how the network is performing we need an objective.

$$E = \sum_t \text{loss}(\hat{y}(\mathbf{x}^t), y^t)$$

Loss for continuous y:

$$\text{loss}(\hat{y}, y) = \frac{1}{2}(y - \hat{y})^2$$

Loss for a binary y:

$$\text{loss}(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

Q: We can work through a sanity check here. Is the loss larger when $y \neq \hat{y}$ than when $y = \hat{y}$? [Hint: You can assume that $0 \log 0 = 0$]

6 Looking at the objective and forward propagation

We have

$$\mathbf{h}^0 = \mathbf{x} \tag{4}$$

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l \qquad \text{input into the } l\text{th layer activation} \tag{5}$$

$$\mathbf{h}^l = \mathbf{f}(\mathbf{z}^l) \qquad \text{output of the } l\text{th layer activation} \tag{6}$$

$$\hat{y}(\mathbf{x}) = h^L \qquad \text{output is the last layer's state} \tag{7}$$

$$\tag{8}$$

The objective is given by

$$E = \sum_t \text{loss}(\hat{y}(\mathbf{x}), y^t).$$

We are interested in minimizing the loss, so we need partial derviatives

$$\frac{\partial E}{\partial w_{ij}^l} \text{ and } \frac{\partial E}{\partial b_i^l}$$

for all layers $l = 1, ...L$.

7 Frame of mind for back-prop(agation)

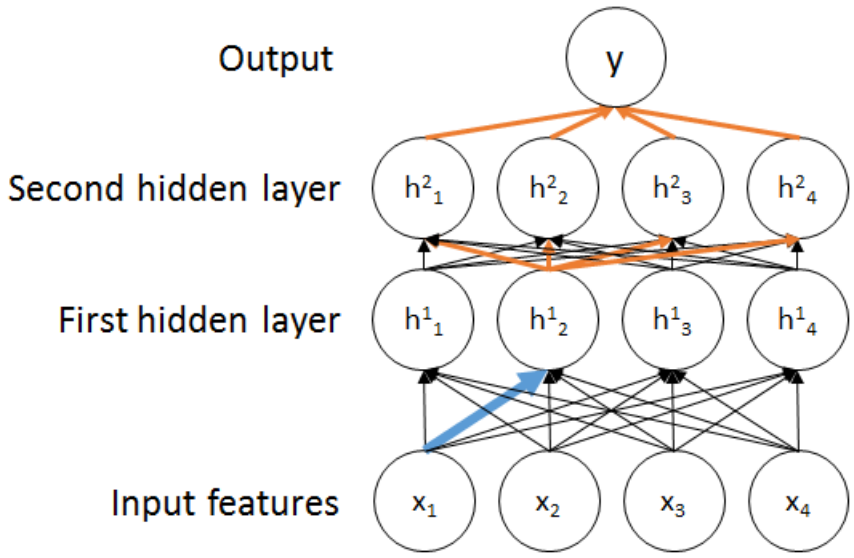
Forward prop is a recursive procedure which propagates information upward to make a prediction.

That error of that prediction E can be reduced if we change parameter w_{ij}^l according to the gradient

$$\frac{\partial E}{\partial w_{ij}^l}$$

Chain rule tels us that

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial h_i^l} \frac{\partial h_i^l}{\partial w_{ij}^l}$$



8 Frame of mind for back-prop

We compute how much of the error in the prediction is attributable to each parameter.

To do this we will break down computation into two parts: 1. computing how a hidden variable affects the objective -- across all orange paths

$$\frac{\partial E}{\partial h_i^l}$$

2. computing how parameter associated with the hidden variable affects the variable -- blue edge

$$\frac{\partial h_i^l}{\partial w_{ij}^l}$$

9 Total derivative reminder

Total derivative

$$\frac{\partial f(\mathbf{z}(t))}{\partial t} = \sum_i \frac{\partial f}{\partial z_i}\frac{\partial z_i}{\partial t}$$

Let's say

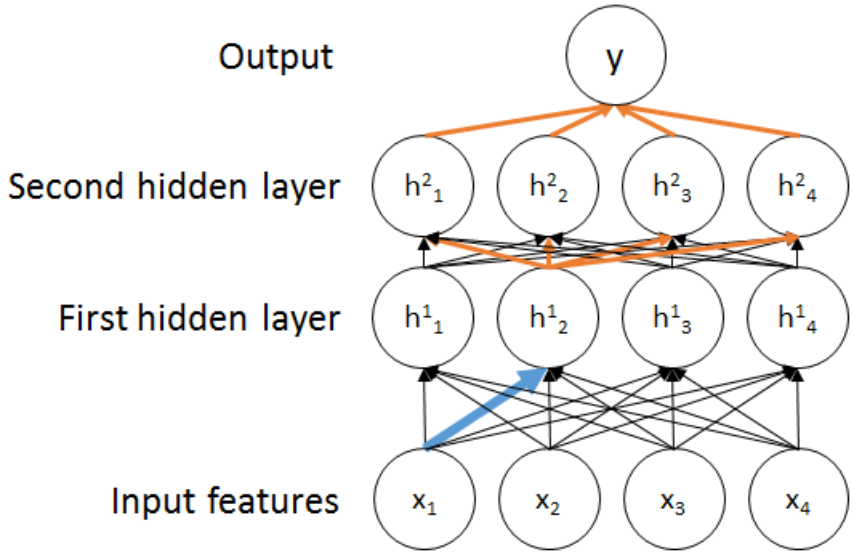
$$f(x,y,z) = x^2 + y^3 + z^4$$

and let's say

$$g(t) = f(t,t,1)$$

then

$$\frac{\partial g(t)}{\partial t} = 2t + 3t^2$$



10 Evaluate-at notation

Also, to be very explicit

$$\frac{\partial f(\mathbf{z})}{\partial z_i}\Big|_{\mathbf{z}=\mathbf{z}_0}$$

this is a partial derivative with respect to z_i evaluated at $\mathbf{z}_0$.

For example,

$$\frac{\partial z_1^2 + z_2}{\partial z_1}\Big|_{\mathbf{z}=[1.0,-1.0]} = 2z_1\Big|_{\mathbf{z}=[1.0,-1.0]} = 2$$

Once we get used to this, I will drop the "evaluate-at" notation.

As a rule of thumb, everything on the left of the vertical bar are variables and functions, **not** values.

Values occur only on the right side of the bar.

11 Using chain rule to compute derivatives -- back propagation

For completness, we are briefly going to be very explicit about the states of the neural network for different samples.

Forward propagation on each of the samples gives us

$$\mathbf{h}^{0,t} = \mathbf{x}^t \quad \text{sample t, feature vector} \tag{9}$$

$$\mathbf{z}^{l,t} = \mathbf{W}^l \mathbf{h}^{l-1,t} + \mathbf{b}^l \quad \text{sample t, input } l\text{th layer activation} \tag{10}$$

$$\mathbf{h}^{l,t} = \mathbf{f}(\mathbf{z}^{l,t}) \quad \text{sample t, output } l\text{th layer activation} \tag{11}$$

$$\hat{y}(\mathbf{x}^t) = h^{L,t} \quad \text{sample t, network prediction} \tag{12}$$

$$\tag{13}$$

12 Using chain rule to compute derivatives -- back propagation

The objective is given by

$$E = \sum_t \text{loss}(\hat{y}(\mathbf{x}^t), y^t).$$

Partial derivative

$$\frac{\partial E}{\partial w_{ij}^l} = \sum_t \frac{\partial \text{loss}(\hat{y}, y^t)}{\partial \hat{y}}\Big|_{\hat{y}=\hat{y}(\mathbf{x}^t)} \frac{\hat{y}}{\partial h^L}\Big|_{h^L=h^{L,t}} \frac{\partial h^L}{\partial w_{ij}^l} \tag{14}$$

Pausing here for a moment: h^L is a variable, $h^{L,t}$ is a value computed through forward propagation starting with input $\mathbf{x}^t$.

13 Simplifying math

Carrying the sample index, t, and stating where the partial derivatives are evaluated is cumbersome.

From this point on, we will compute partial derivative for a single sample and some parameter θ.

Further, all derivatives will be evaluated at values obtained through forward propagation $(\mathbf{h}^l, \mathbf{z}^l)$.

Hence, this term

$$\frac{\partial \text{loss}(\hat{y}(\mathbf{x}^t), y^t)}{\partial \theta} = \frac{\partial \text{loss}(\hat{y}, y^t)}{\partial \hat{y}}\Big|_{\hat{y}=\hat{y}(\mathbf{x}^t)} \frac{\hat{y}}{\partial h^L}\Big|_{h^L=h^{L,t}} \frac{\partial h^L}{\partial \theta} \tag{15}$$

after dropping sample index and removing evalate-at notation, simplifies to

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial \theta} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \frac{\hat{y}}{\partial h^L} \frac{\partial h^L}{\partial \theta} \tag{16}$$

14 Using chain rule to compute derivatives -- back propagation

We are going to use the facts that $\mathbf{h}^m$ depends on $\mathbf{z}^m$ and $\mathbf{z}^m$ depends on $\mathbf{h}^{m-1}$ to keep expanding the derivative to highlight a pattern

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial \theta} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \sum_{i=1}^{n_L} \frac{\hat{y}}{\partial h_i^L} \frac{\partial h_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial \theta} \tag{17}$$

$$= \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \sum_{i=1}^{n_L} \frac{\hat{y}}{\partial h_i^L} \frac{\partial h^L}{\partial z_i^L} \sum_{j=1}^{n_{L-1}} \frac{\partial z_i^L}{\partial h_j^{L-1}} \frac{\partial h_j^{L-1}}{\partial \theta} \tag{18}$$

$$= \sum_{j=1}^{n_{L-1}} \underbrace{\sum_{i=1}^{n_L} \overbrace{\frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \frac{\hat{y}}{\partial h_i^L}}^{\partial \text{loss}(\hat{y},y)/\partial h_i^L} \frac{\partial h_i^L}{\partial z_i^L} \frac{\partial z_i^L}{\partial h_j^{L-1}}}_{\partial \text{loss}(\hat{y},y)/\partial h_j^{L-1}} \frac{\partial h_j^{L-1}}{\partial \theta} \tag{19}$$

$$\tag{20}$$

Hence, we can see a recursion needed to compute $\frac{\partial \text{loss}(\hat{y},y)}{\partial h_j^{L-1}}$ using $\frac{\partial \text{loss}(\hat{y},y)}{\partial h_i^L}$

15 Using chain rule to compute derivatives -- back propagation

Start of recursion at L:

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^L} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \frac{\hat{y}}{\partial h_i^L} \tag{21}$$

Recursive rule, going **back** from layer l to $l-1$:

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_j^{l-1}} = \sum_{i=1}^{n_l} \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} \frac{\partial h_i^l}{\partial z_i^l} \frac{\partial z_i^l}{\partial h_j^{l-1}} \tag{22}$$

$$= \sum_{i=1}^{n_l} \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) w_{ij}^l \tag{23}$$

16 Recap

Given a neural network we know how to use **forward propagation** to obtain **h** and **z**.

Using these **h** and **z** we can use a **back propagation** to compute $\frac{\partial \text{loss}(\hat{y},y)}{\partial h_i^l}$

We can plug-in computed $\frac{\partial \text{loss}(\hat{y},y)}{\partial h_i^l}$ to obtain derivatives of the loss with respect to parameters

$$\frac{\partial E}{\theta} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} \frac{\partial h_i^L}{\partial \theta} \tag{24}$$

where θ is either w_{ij}^l or b_i^l

17 Computing derivatives within a layer

$$\frac{\partial E}{\theta} = \underbrace{\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l}}_{\text{know how to compute}} \underbrace{\frac{\partial h_i^L}{\partial \theta}}_{\text{need to figure out}} \tag{25}$$

Hence

$$\frac{\partial h_i^L}{\partial \theta} = \frac{\partial h_i^L}{\partial z_i^l} \frac{\partial z_i^L}{\partial \theta}$$

$$\frac{\partial h_i^L}{\partial \theta} = \frac{\partial h_i^L}{\partial z_i^l} \frac{\partial z_i^L}{\partial \theta}$$

Gives us

$$\frac{\partial h_i^l}{\partial w_{ij}^l} = f'(z_i^l) h_j^{l-1} \tag{26}$$

$$\frac{\partial h_i^l}{\partial b_i^l} = f'(z_i^l) \tag{27}$$

18 Back-propagation: putting it together

1. Run forward prop

$$\mathbf{h}^0 = \mathbf{x} \tag{28}$$

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l \qquad \text{input into the } l\text{th layer activation} \tag{29}$$

$$\mathbf{h}^l = \mathbf{f}(\mathbf{z}^l) \qquad \text{output of the } l\text{th layer activation} \tag{30}$$

$$\hat{y}(\mathbf{x}) = h^L \qquad \text{output is the last layer's state} \tag{31}$$

$$\tag{32}$$

2. Run backward prop

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^L} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \frac{\hat{y}}{\partial h_i^L} \tag{33}$$

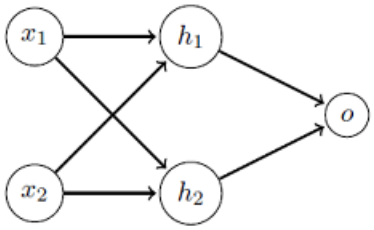
$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_j^{l-1}} = \sum_{i=1}^{n_l} \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) w_{ij}^l \tag{34}$$

3. Compute gradients

$$\frac{\partial E}{w_{ij}^l} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) h_j^{l-1} \tag{35}$$

$$\frac{\partial E}{b_i^l} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) \tag{36}$$

$$\tag{37}$$



An XOR Network

x_1	0	1	0	1
x_2	0	0	1	1
$h_1 = \sigma(F(-0.5 + x_1 - x_2))$	0	1	0	0
$h_2 = \sigma(F(-0.5 - x_1 + x_2))$	0	0	1	0
$y = \sigma(F(-0.5 + h_1 + h_2))$	0	1	1	0

In [293]:
```
import numpy as np

def loss(yhat,y):
    # loss(yhat,y)
    l = 0.5*(yhat - y)**2.0
    # d loss(yhat,y) / d yhat
    g = (yhat - y)
    return l,g

def sigmoid(z):
    # if z is a vector
    # h is a vector of outputs
    # g is derivative of f at each z
    h = 1.0/(1.0 + np.exp(-z))
    g = h*(1.0 - h)
    return h,g

def compute_loss(loss,predict,x,y):
    yhat = predict(x)
    val,_ = loss(yhat,y)
    return val.squeeze()
```

1. Run forward prop

$$\mathbf{h}^0 = \mathbf{x} \tag{38}$$

$$\mathbf{z}^l = \mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l \quad \text{input into the } l\text{th layer activation} \tag{39}$$

$$\mathbf{h}^l = \mathbf{f}(\mathbf{z}^l) \quad \text{output of the } l\text{th layer activation} \tag{40}$$

$$\hat{y}(\mathbf{x}) = h^L \quad \text{output is the last layer's state} \tag{41}$$

$$\tag{42}$$

```
In [206]: def forward_prop(x,f,Ws,bs):
              # simple implementation of forward prop
              hs = [x]                # store h^0
              zs = [np.zeros(len(x))] # store z^0
              h = x                   # current layer is 0
              L = len(Ws)
              for l in range(L):
                  Wl = Ws[l]              # get layer weights
                  bl = bs[l]              # get layer biases
                  nl = Wl.shape[0]        # number of variables in
                                          # layer l
                  z = np.dot(Wl,h) + bl # input into layer l
                  h,_ = f(z)              # activation for each z
                  zs.append(z)
                  hs.append(h)
              return hs,zs

          # XOR net
          Ws = []
          bs = []
          F = 5.0
          # first layer
          Ws.append(F*np.asarray([[1.0,-1.0],
                                  [-1.0,1.0]]))
          bs.append(F*np.asarray([[-0.5],
                                  [-0.5]]))
          # second layer weights
          Ws.append(F*np.asarray([[1.0,1.0]]))
          bs.append(F*np.asarray([[-0.5]]))

          for x1 in [0.0,1.0]:
              for x2 in [0.0,1.0]:
                  x = np.asarray([[x1],[x2]])
                  hs,zs = forward_prop(x,sigmoid,Ws,bs)
                  print "Input: ",x.T,"Output: ",hs[-1].squeeze()

Input:  [[ 0.  0.]] Output:  0.149132886381
Input:  [[ 0.  1.]] Output:  0.893163783089
Input:  [[ 1.  0.]] Output:  0.893163783089
Input:  [[ 1.  1.]] Output:  0.149132886381
```

2. Run backward prop

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^L} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial \hat{y}} \frac{\hat{y}}{\partial h_i^L} \tag{43}$$

$$\frac{\partial \text{loss}(\hat{y}, y)}{\partial h_j^{l-1}} = \sum_{i=1}^{n_l} \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) w_{ij}^l \tag{44}$$

```
In [208]: def backward_prop(loss,f,Ws,bs,zs,y):
              _,dldh = loss(hs[-1],y)
              L = len(Ws)
              dldhs = [[]]*(L+1) # one per layer
              dldhs[L] = dldh  # derivative of loss with respect to
                               # prediction (yhat)
              for l in reversed(range(L)):
                  Wl = Ws[l]
                  bl = bs[l]
                  nl = Wl.shape[0]
                  n = Wl.shape[1]
                  dldhs[l] = np.zeros((n,1))
                  for i in range(nl):
                      _,df = f(zs[l+1][i])
                      m = dldhs[l+1][i]*df
                      for j in range(n):
                          dldhs[l][j] = m*Wl[i,j]
              return dldhs
```

3. Compute gradients

$$\frac{\partial E}{w_{ij}^l} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) h_j^{l-1} \tag{45}$$

$$\frac{\partial E}{b_i^l} = \frac{\partial \text{loss}(\hat{y}, y)}{\partial h_i^l} f'(z_i^l) \tag{46}$$

$$\tag{47}$$

```
In [ ]: def compute_gradients(dldhs,f,zs,hs):
            L = len(dldhs)-1
            dWs = [[]]*L
            dbs = [[]]*L
            for l in range(L):
                nl = len(hs[l+1])
                n = len(hs[l])
                dWs[l] = np.zeros((nl,n))
                for i in range(nl):
                    _,g = f(zs[l+1][i])
                    hl = hs[l].squeeze()
                    dWs[l][i,:] = dldhs[l+1][i]*g*hl
```

```
                dbs[l] = dldhs[l+1][i]*g


        return dWs,dbs
```

In [329]:
```
#
# Gradient Checking code using finite diffrences
#
x = np.asarray([[1.0],[1.0]])
y = np.asarray([[0.0]])
Ws[0] = 0.1*np.random.randn(2,2)
Ws[1] = 0.1*np.random.randn(1,2)
bs[0] = 0.1*np.random.randn(2,1)
bs[1] = 0.1*np.random.randn(1,1)
hs,zs = forward_prop(x,sigmoid,Ws,bs)
dldhs = backward_prop(loss,sigmoid,Ws,bs,zs,y)
dWs,dbs = compute_gradients(dldhs,sigmoid,zs,hs)

L = len(dWs)
eps = 1e-7
fddWs = [[]]*L


for l in range(L):
    fddWs[l] = np.zeros(Ws[l].shape)
    for i in range(Ws[l].shape[0]):
        for j in range(Ws[l].shape[1]):
            Wc = Ws[l].copy()
            Ws[l][i,j] = Wc[i,j] + 0.5*eps
            predict = lambda x: forward_prop(x,sigmoid,Ws,bs)[0][-1].squeeze()
            v1 = compute_loss(loss,predict,x,y)
            Ws[l][i,j] = Wc[i,j] - 0.5*eps
            predict = lambda x: forward_prop(x,sigmoid,Ws,bs)[0][-1].squeeze()
            v2 = compute_loss(loss,predict,x,y)
            Ws[l] = Wc
            fddWs[l][i,j] = (v1 - v2)/eps
            print(fddWs[l][i,j],dWs[l][i,j])
```

```
(0.0031136092315531272, 0.0031136087408337918)
(0.0031136092315531272, 0.0031136087408337918)
(-0.0047451058360348242, -0.0047451063432885328)
(-0.0047451058360348242, -0.0047451063432885328)
(0.05932591112811636, 0.059325910360931711)
(0.053831228363776518, 0.053831228615190692)
```

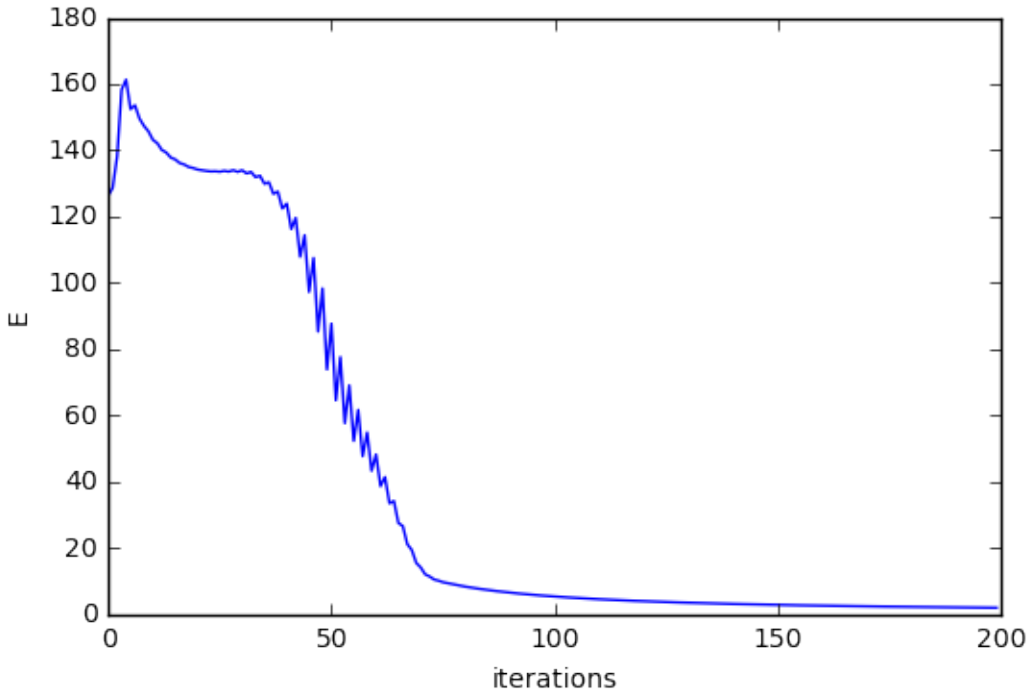
In [327]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

```
np.random.seed(1)
Ws[0] = 0.5*np.random.randn(2,2)
Ws[1] = 0.5*np.random.randn(1,2)
bs[0] = 0.5*np.random.randn(2,1)
bs[1] = 0.5*np.random.randn(1,1)


N = 1000
x = (np.random.rand(2,N)>0.5).astype('float')
x = x + 0.1*np.random.randn(2,N)

y = np.logical_xor(x[0,:]>0.5,x[1,:]>0.5).astype('float')

L = 2
eta = 3e-2
Es = []
for it in range(200):
    E = 0
    for l in range(L):
        dWs[l] = np.zeros(Ws[l].shape)
        dbs[l] = np.zeros(bs[l].shape)

    for t in range(N):
        xt = x[:,[t]]
        yt = y[t]
        hs,zs = forward_prop(xt,sigmoid,Ws,bs)
        dldhs = backward_prop(loss,sigmoid,Ws,bs,zs,yt)
        dtWs,dtbs = compute_gradients(dldhs,sigmoid,zs,hs)
        for l in range(L):
            dWs[l] = dWs[l] + dtWs[l]
            dbs[l] = dbs[l] + dtbs[l]
        Et = loss(hs[-1][0][0],yt)[0]
        E = E + Et
    for l in range(L):
        Ws[l] = Ws[l] - eta*dWs[l]
        bs[l] = bs[l] - eta*dbs[l]
    Es.append(E)
```

In [328]:
```
plt.plot(Es)
plt.xlabel('iterations')
plt.ylabel('E')
print "First layer weights:\n",Ws[0]
print "First layer biases:\n",bs[0]
print "Second layer weights:\n",Ws[1]
print "Second layer biases:\n",bs[1]

for x1 in [0.0,1.0]:
```

```
            for x2 in [0.0,1.0]:
                x = np.asarray([[x1],[x2]])
                hs,zs = forward_prop(x,sigmoid,Ws,bs)
                print "Input: ",x.T,"Output: ",hs[-1].squeeze()
```

```
First layer weights:
[[-3.2933533  -3.3736506 ]
 [-6.57363912 -6.7182171 ]]
First layer biases:
[[ 4.72145779]
 [ 3.46844846]]
Second layer weights:
[[ 9.89952122 -9.51866265]]
Second layer biases:
[[-4.20933528]]
Input:  [[ 0.  0.]] Output:  0.0258780857113
Input:  [[ 0.  1.]] Output:  0.96419601442
Input:  [[ 1.  0.]] Output:  0.966669564212
Input:  [[ 1.  1.]] Output:  0.0487062060114
```



19 Recap

We introduced back-propagation.

It decomposed computation of partial derivatives of the loss wrt parameters into two tasks 1. derivative of loss with respect to hidden variables 2. derivative of the hidden variable with respect to parameter

First task was accomplished using chain rule to yield a recursive formula which works backwards from the loss -- back-propagation.

Second task was accomplished by taking derivatives within a layer.