CSE 5523: Lecture Notes 14
(Multi-layer) Neural Networks

Contents

Logistic regression can only find linear separators, which are not always appropriate.

Adding L layers of logistic or other non-linear transfer functions lets us learn ***any*** function:

$$f_L(\mathbf{x}) \stackrel{\text{def}}{=} \frac{e^{\mathbf{W}_L\, f_{L-1}(\mathbf{x})}}{\sum_{y'} e^{\delta_{y'}^\top \mathbf{W}_L\, f_{L-1}(\mathbf{x})}}$$

$$f_0(\mathbf{x}) \stackrel{\text{def}}{=} \mathbf{x}$$

(with the disadvantage that they are no longer guaranteed to be globally optimal).

14.1 (Multi-layer) Neural Networks

We again fit parameters using a negative log loss function to make low probabilities linearly bad:

$$\mathsf{L}_{\text{NL}}(y, f_L(\mathbf{x})_{[y]}) = -\ln f_L(\mathbf{x})_{[y]}$$

We can then find the slope (derivative) of the expected loss to update our gradient descent:

$$\frac{\partial}{\partial(\mathbf{W}_m)_{[p,q]}} - \ln f_L(\mathbf{x})_{[y]} = -\frac{1}{f_L(\mathbf{x})_{[y]}} \frac{\partial f_L(\mathbf{x})_{[y]}}{\partial(\mathbf{W}_m)_{[p,q]}} \qquad \text{derivative of natural log}$$

14.2 Jacobian matrices for transfer/activation functions

Now, let's assume each $f_\ell(\mathbf{x}) = \frac{e^{\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}$ is a softmax:

$$\begin{aligned}
\frac{\partial f_\ell(\mathbf{x})_{[i]}}{\partial(\mathbf{W}_m)_{[p,q]}} &= \frac{\partial}{\partial(\mathbf{W}_m)_{[p,q]}} \frac{e^{\delta_i^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}} \\
&= [\![i{=}j]\!] \frac{1}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}} e^{\delta_j^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})} \frac{\partial(\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x}))_{[j]}}{\partial(\mathbf{W}_m)_{[p,q]}} \\
&\quad - e^{\delta_i^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})} \frac{1}{(\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})})^2} e^{\delta_j^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})} \frac{\partial(\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x}))_{[j]}}{\partial(\mathbf{W}_m)_{[p,q]}} && \text{product rule} \\
&= \left([\![i{=}j]\!] - \frac{e^{\delta_i^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}\right) \frac{e^{\delta_j^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}} \frac{\partial(\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x}))_{[j]}}{\partial(\mathbf{W}_m)_{[p,q]}} && \text{distributive axiom}
\end{aligned}$$

So $\left(\text{diag}(\mathbf{1}) - \frac{e^{\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}} \mathbf{1}^\top\right) \text{diag}\left(\frac{e^{\mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}{\sum_k e^{\delta_k^\top \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}}\right) = \frac{\partial f_\ell(\mathbf{x})}{\partial \mathbf{W}_\ell\, f_{\ell-1}(\mathbf{x})}$ is a **Jacobian matrix** of derivatives.

14.3 Jacobian matrices for weights

Now, if $m = \ell$ then $\frac{\partial(\mathbf{W}_\ell f_{\ell-1}(\mathbf{x}))_{[i]}}{\partial(\mathbf{W}_m)_{[p,q]}} = [\![i{=}p]\!] f_{\ell-1}(\mathbf{x})_{[q]}$.

And if $m \neq \ell$, then by the multivariable chain rule for derivatives:

$$\frac{\partial}{\partial z} f(g_1(x), \ldots, g_J(x)) = \sum_{j \in \{1,\ldots,J\}} \frac{\partial f(g_1(x), \ldots, g_J(x))}{\partial g_j(x)} \frac{\partial g_j(x)}{\partial z}$$

This gives us:

$$\frac{\partial(\mathbf{W}_\ell f_{\ell-1}(\mathbf{x}))_{[i]}}{\partial(\mathbf{W}_m)_{[p,q]}} = \sum_j (\mathbf{W}_\ell)_{[i,j]} \frac{\partial f_{\ell-1}(\mathbf{x})_{[j]}}{\partial(\mathbf{W}_m)_{[p,q]}} \qquad \text{multivariable chain rule}$$

$$\frac{\partial \mathbf{W}_\ell f_{\ell-1}(\mathbf{x})}{\partial(\mathbf{W}_m)_{[p,q]}} = \mathbf{W}_\ell \frac{\partial f_{\ell-1}(\mathbf{x})}{\partial(\mathbf{W}_m)_{[p,q]}} \qquad \text{definition of inner product}$$

So $\mathbf{W}_\ell = \frac{\partial \mathbf{W}_\ell f_{\ell-1}(\mathbf{x})}{\partial f_{\ell-1}(\mathbf{x})}$. This means $\mathbf{W}_\ell$ is another Jacobian matrix.

14.4 Backpropagation

We can now chain up these two kinds of Jacobian matrices to update any parameter:

$$\mathbf{W}_m^{(i)} = \mathbf{W}_m^{(i-1)} - \underbrace{\frac{\partial \mathsf{L}_{\text{NL}}(y, f_L(\mathbf{x})_{[y]})}{\partial f_L(\mathbf{x})}}_{\text{loss fn}} \underbrace{\frac{\partial f_L(\mathbf{x})}{\partial \mathbf{W}_L f_{L-1}(\mathbf{x})}}_{\text{transfer fn}} \underbrace{\frac{\partial \mathbf{W}_L f_{L-1}(\mathbf{x})}{\partial f_{L-1}(\mathbf{x})}}_{\text{weights}} \underbrace{\frac{\partial f_{L-1}(\mathbf{x})}{\partial \mathbf{W}_{L-1} f_{L-2}(\mathbf{x})}}_{\text{transfer fn}} \cdots \underbrace{\frac{\partial f_m(\mathbf{x})}{\partial \mathbf{W}_m f_{m-1}(\mathbf{x})}}_{\text{transfer fn}} f_{m-1}(\mathbf{x})$$

This is called **backpropagation**.

14.5 Sample multi-layer neural network code

Sample multi-layer neural network code in pandas:

```
import sys
import numpy
import pandas

def logistic( Wx ):
  return numpy.exp( Wx ).div ( numpy.ones(len(Wx)) @ numpy.exp( Wx ) )

YX = pandas.read_csv( sys.argv[1] )          ## read data

Y = pandas.get_dummies( YX[YX.columns[0]] )   ## transform data
X = YX[YX.columns[1:]]
N = len(YX)
X['line'] = numpy.ones((N,1))

J = len(Y.columns)  ## output layer
K = 5               ## hidden layer
V = len(X.columns)  ## input layer
L = 2               ## number of layers
```

```
                                                            ## random initial weights
W = [ None, pandas.DataFrame( numpy.random.rand(K,V), range(K),  X.columns ),
            pandas.DataFrame( numpy.random.rand(J,K), Y.columns, range(K)  ) ]
f      = {}                                                 ## estimation functions
f[0] = lambda x : x
f[1] = lambda x : logistic( W[1] @ f[0](x) )
f[2] = lambda x : logistic( W[2] @ f[1](x) )
fx     = {}                                                 ## partial estimates
df_dWf = {}                                                 ## jacobians for transfer functions
dC_dWf = {}                                                 ## propagated cost at each layer

for i in range(100):                                        ## for each epoch
  for n in range(N):                                        ## for each training example
    for l in range(L+1):                                    ## for each layer (forward pass)
      fx[l] = f[l]( X.iloc[[n]].T )                         ## get predictions at each layer
      if l>0: df_dWf[l] = ( ( numpy.eye(len(W[l]))          ## update transfer fn jacobians
                              - logistic( W[l] @ fx[l-1] ) @ numpy.ones((1,len(W[l]))) )
                            @ numpy.diagflat( logistic( W[l] @ fx[l-1] ).values ) )
    for l in range(L,0,-1):                                 ## for each layer (backward pass)
      if l==L: dC_dWf[l] = ( logistic( W[l] @ fx[l-1] ) - Y.iloc[[n]].T ).T
      else:    dC_dWf[l] = dC_dWf[l+1] @ W[l+1] @ df_dWf[l]    ## update propagated costs
      W[l] = W[l] - 1/N * dC_dWf[l].T @ fx[l-1].T               ## update weights

for Wl in W:                                                ## print weight matrices
  print( Wl )

for n in range(len(YX)):                                    ## print predictions
  print( f[L]( X.iloc[[n]].T ) )
```

Sample input data file 'YX.csv':

```
y,x1,x2
ya,-1,-1
ya,-1,1
ya,1,-1
ya,1,1
ya,0,0
no,-2,-2
no,-2,2
no,2,-2
no,2,2
```

Output trained weights and predictions:

```
         x1        x2      line
0  0.357868  0.533448  3.563356
1  2.517051  1.594430 -0.345190
2 -1.283607  2.195032 -0.533848
3 -0.996177 -1.221073 -0.134422
4  1.821077 -1.197364 -0.245445
           0         1         2         3         4
no -2.473382  1.751548  1.758151  1.477024  1.695649
ya  3.156034 -0.914642 -1.000134 -0.985230 -0.686480
```

```
           0
no  0.069736
ya  0.930264
           1
no  0.050798
ya  0.949202
           2
no  0.076214
ya  0.923786
           3
no  0.061119
ya  0.938881
           4
no  0.006705
ya  0.993295
           5
no  0.865514
ya  0.134486
           6
no  0.893557
ya  0.106443
           7
no  0.861823
ya  0.138177
           8
no  0.886587
ya  0.113413
```