12feb19
Summary of Concluding Talk on
"Backprop in Neural Nets and Automatic Differentiation"

In the first of these two seminars I left two items undone which were completed in the second talk.

In the second, I first showed that the methodology (presented in the first talk) of keeping track of the component steps in evaluating a function (from many independent variables to fewer dependent variables) as described in the Baydin++ survey paper, "worked" for the case of a neural net with one hidden layer proposed by Trask for the XOR function.

Using the notation in the distilled Trask code (top of the first page of the handout following), Trask iterates the calculation of the mapping

y2 = f(W1,w2) = sigma(sigma(X0 W1) w2)

by back-propagating the gradient descent of the energy function on the output space

y = 1/2 |y0 - y2|^2

to two dynamical systems in the weight spaces of the matrix W1(4,3) and of the vector w2(4,1). Here X0 is the matrix of all four possible truth values of two Booleans, and y0 is the exact value vector of their Boolean XOR product, and sigma(.) is the termwise application of the logistic sigmoid function.

My first attempt to apply the method of backward automatic differentiation to this example, which was a failure, served only as a warning to watch one's notation when a mixture of matrix and Hadamard array products is involved. It should be ignored here.

On the top of the following page of the handhout is the reverse automatic differentiation, and (bottom) the pedantic tracing of the actual value of each auxillary variable, v_i.

Note that I relabelled the adjoints by ai. The first = is a definition of the adjoint, the next ='s are an application of the chain-rule, and only the last = assigns a numerical value the program has already calculated at that step.

The remarkable thing is that I did manage a derivation of Trask's update step for the weights (up to a +/- sign), but could not justify his intermediate substitution into a variable he originally associated with Rumelhart's deltas, and which I mistakenly mislabeled as dy2 and dY1, but now call yy2 and YY1 because they are neither a differential nor a gradient, but some sort of a mixture.

Then, I demonstrated by some simple experiments, a disquieting feature of this very elementary example of a "deep" neural net, namely the instability of the back-propagated dynamical sytem to the weight spaces.

In the choice of the initial values of the weights, Trask follows the industry recommendation of using random values. So outcomes labelled out1 and out10 use such a random initial value, iterated 600, respectively 6000 times. The output 4-vectors and the weights appear to be settling down.

In the second experiment, I perversely choose 0 for all initial weights with the expected failure of NN that all four output values are near 1/2.

In the final experiment I chose all 16 initial weights to be zero, exept two, which were set to a nonzero value. The NN succeeded, with more effort to at least show the right trend by 600 iterations (out3) and 6000 iterations (out30). But unlike the random initialization in the first experiment, the values of the weights do not seem to be settling down to a limiting value, they seem to continue to wander.

Obviously, this isn't a very persuasive, and certainly not systematic set of five experiments. But it suggests that back propagation of a totally elementary gradient dynamical system, while is seems to work, does not do so in a very predicatable way.

```
#5feb19 from playInits 28jan19                                PAGE 0

import numpy as np ## Python library for array operations
X0 = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]]) ## X0(4,3)=4rows of 3vcrs
y0 = np.array([[0,1,1,0]]).T ## XOR, y0(4,1) column vector
W1= 2*np.random.random((3,4))-1 ## W1(3,4) signed fractions
w2= 2*np.random.random((4,1))-1 ## w2(4,1) signed fractions

for jj in xrange(600):
    Y1 = 1/(1+np.exp(-(np.dot(X0,W1)))) ## Y1(4,4)=sigma(X0(4,3)W1(3,4))
    y2 = 1/(1+np.exp(-(np.dot(Y1,w2)))) ## y2(4,1)=sigma(Y1(4,4)w2(4,1))

    dy2= (y0-y2)*y2*(1-y2)              ## dy2(4,1)*Hadamard y2*(1-y2)
    dY1= dy2.dot(w2.T)*Y1*(1-Y1)        ## dY1(4,4) = dy2(4,1)w2.T(1,4)*Had Y1*(1-Y1)

    w2 += Y1.T.dot(dy2)                 ## w2(4,1) += Y1.T(4,4)dy2(4,1)
    W1 += X0.T.dot(dY1)                 ## W1(4,4) += X0.T(3,4)dY1(4,4)



================================================================================

#5feb19 from playInits 28jan19                            PAGE 1 IGNORE

import numpy as np ## Python library for array operations
X0 = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]]) ## X0(4,3)=4rows of 3vcrs
y0 = np.array([[0,1,1,0]]).T ## XOR, y0(4,1) column vector
W1= 2*np.random.random((3,4))-1 ## W1(3,4) signed fractions
w2= 2*np.random.random((4,1))-1 ## w2(4,1) signed fractions

for jj in xrange(600):
    Y1 = 1/(1+np.exp(-(np.dot(X0,W1)))) ## Y1(4,4)=sigma(X0(4,3)W1(3,4))
    y2 = 1/(1+np.exp(-(np.dot(Y1,w2)))) ## y2(4,1)=sigma(Y1(4,4)w2(4,1))

    dy2= (y0-y2)*y2*(1-y2)              ## dy2(4,1)*Hadamard y2*(1-y2)
    dY1= dy2.dot(w2.T)*Y1*(1-Y1)        ## dY1(4,4) = dy2(4,1)w2.T(1,4)*Had Y1*(1-Y1)

    w2 += Y1.T.dot(dy2)                 ## w2(4,1) += Y1.T(4,4)dy2(4,1)
    W1 += X0.T.dot(dY1)                 ## W1(4,4) += X0.T(3,4)dY1(4,4)


    X0, y0:
    v00= W1
    v0 = w2
    v1 = X0 W1 = X0 V00
    v2 = Y2 = sigma(v1) ;; dv2/dv1 = *v2*(1-v2)
    v3 = Y1 w2 =v2 v0   ;; dv3/dress-upv2 h = h v0 but dv3/dv2 k = v2 k -- Matrix product
    v4 = y2 = sigma(v3)
    v5 = y = 1/2 |y0 - y2|^2 = 1/2 |y0 - v4|^2

    av5 = dy/dv5 = dy/dy = 1                    ;; av5(1)
    av4 = dy/dv4 = av5 dv5/dv4 = (v4 - y0)T     ;; av4(1,4) chain rule
    av3 = dy/dv3 = av4 dv4/dv3 = av4*(v4)*(1-v4)  ;; s' = s(1-s) Hadamard
    av2 = dy/dv2 = av3 dv3/dv2 = av3 v0         ;; I hope that's right
    av1 = dy/dv1 = av2 dv2/dv1 = av2*v2*(1-v2)  ;;
    av0 = dy/dv0 = av3 dv3/dv0 = av3 v2         ;;

    dy/dw2    derailed
```

```
------------------------------Trax's 1-layer deep NN for XOR-------------
 X0(4,3) 4rows of 3vcrs
 y0(4,1) column vector

 W1(3,4) signed fractions
 w2(4,1) signed fractions

 Y1(4,4)=sigma(X0(4,3)W1(3,4))
 y2(4,1)=sigma(Y1(4,4)w2(4,1))

 yy2(4,1) = (y0-y2)*Hadamard y2*(1-y2)      ->  w2(4,1) += Y1T(4,4)yy2(4,1)
 YY1(4,4) = yy2(4,1)w2T(1,4)*Hadd Y1*(1-Y1) ->  W1(4,4) += X0T(3,4)YY1(4,4)

---------------------------------reverse auto diff -----------------------
    X0, y0
    v- = W1
    v0 = w2
    v1 = X0 W1 = X0 v-
    v2 = Y2 = sigma(v1) ;; dv2/dv1 = *v2*(1-v2)
    v3 = Y1 w2 =v2 v0   ;; dv3/dv2 h = h v0 but dv3/dv2 k = v2 k -- Matrix product
    v4 = y2 = sigma(v3)
    v5 = y = 1/2 |y0 - y2|^2 = 1/2 |y0 - v4|^2

    a5 = dy/dv5 = dy/dy = 1                    ;; av5 scalar
    a4 = dy/dv4 = a5 dv5/dv4 = (v4 - y0)T      ;; chainrule + a4(1,4)
    a3 = dy/dv3 = a4 dv4/dv3 = a4*v4*(1-v4)    ;; s' = s(1-s) + Hadamard
    a2 = dy/dv2 = a3 dv3/dv2 = a3 (. v0)       ;; (.v0)h = (h v0)
    a1 = dy/dv1 = a2 dv2/dv1 = a2*v2*(1-v2)    ;; s' = s(1-s) + Hadamard
    a0 = dy/dv0 = a3 dv3/dv0 = a3 (v2 .)       ;; (v2.)h = (v2 h)
    a- = dy/dv- = a1 dv1/dv- = a1 (X0 .)

================================ now find out the current and next value================
```



```
------------------------------Trax's 1-layer deep NN for XOR-------------
 X0(4,3) 4rows of 3vcrs
 y0(4,1) column vector

 W1(3,4) signed fractions
 w2(4,1) signed fractions

 Y1(4,4)=sigma(X0(4,3)W1(3,4))
 y2(4,1)=sigma(Y1(4,4)w2(4,1))

 yy2(4,1) = (y0-y2)*Hadamard y2*(1-y2)      ->  w2(4,1) += Y1T(4,4)yy2(4,1)
 YY1(4,4) = yy2(4,1)w2T(1,4)*Hadd Y1*(1-Y1) ->  W1(4,4) += X0T(3,4)YY1(4,4)

---------------------------------reverse auto diff -----------------------
    X0, y0
    v- = W1
    v0 = w2            ;; current value of var
    v1 = X0 W1 = X0 v-  = X0W1
    v2 = Y2 = sigma(v1) = sigma(X0W1) = Y1
    v3 = Y1 w2 =v2 v0   = Y1w2
    v4 = y2 = sigma(v3) = sigma(Y1w2 = y2
    v5 = y              = 1/2 |y0 - y2|^2

    a5 = dy/dv5 = dy/dy = 1
    a4 = dy/dv4 = a5 dv5/dv4 = (v4 - y0)T   = (y2-y0)T
    a3 = dy/dv3 = a4 dv4/dv3 = a4*v4*(1-v4) = (y2-y0)T*y2*(1-y2)  ~~ -yy2T ??
    a2 = dy/dv2 = a3 dv3/dv2 = a3 (. v0)    = -yy2T (. w2)        ~~
    a1 = dy/dv1 = a2 dv2/dv1 = a2*v2*(1-v2) = -yy2T(.w2)*Y1*(1-Y1)~~ YY1T ??
    a0 = dy/dv0 = a3 dv3/dv0 = a3 (v2 .)    = -yy2T(Y1.)          ~~
    a- = dy/dv- = a1 dv1/dv- = a1 (X0 .)    = YY1T (X0.)          ~~

    a- = dy/dW1 -> W1 += - dy/dW1 T  = -(a-)T = (-YY1T (X0.))T = -X0T YY1 ;;bingo
    a0 = dy/dw2 => w2 += - dy/dw2 T  = -(a0)T = (yy2T (Y1 .))T = -Y1T yy2 ;;bingo
                                                      except for a sign problem
```

600 respectively 6000 iterations starting with random weights

```
ITERATION 598
Weight1
[[2.364 2.566 -5.388 -3.306]
 [-0.332 -4.660 -5.675 -1.717]
 [-0.158 -1.165 1.980 3.426]]
weight2
[[-3.682]
 [4.581]
 [-7.633]
 [4.731]]
output
[[0.061]
 [0.919]
 [0.932]
 [0.097]]

ITERATION 599
Weight1
[[2.366 2.567 -5.389 -3.306]
 [-0.334 -4.661 -5.676 -1.719]
 [-0.158 -1.167 1.981 3.428]]
weight2
[[-3.685]
 [4.583]
 [-7.636]
 [4.734]]
output
[[0.061]
 [0.920]
 [0.932]
 [0.097]]

ITERATION 5997
Weight1
[[3.870 3.643 -6.001 -3.828]
 [-1.710 -5.426 -6.315 -3.030]
 [0.449 -1.775 2.376 5.037]]
weight2
[[-5.723]
 [6.162]
 [-9.414]
 [6.614]]
output
[[0.010]
 [0.988]
 [0.989]
 [0.014]]
ITERATION 5998
Weight1
[[3.870 3.643 -6.001 -3.828]
 [-1.710 -5.426 -6.315 -3.030]
 [0.449 -1.775 2.376 5.038]]
weight2
[[-5.723]
 [6.162]
 [-9.414]
 [6.614]]
output
[[0.010]
 [0.988]
 [0.989]
 [0.014]]
ITERATION 5999
Weight1
[[3.870 3.643 -6.001 -3.828]
 [-1.710 -5.426 -6.315 -3.030]
 [0.450 -1.775 2.376 5.038]]
weight2
[[-5.723]
 [6.162]
 [-9.414]
 [6.614]]
output
[[0.010]
 [0.988]
 [0.989]
 [0.014]]
```

6000 interations, starting with both weight matrices all zeros
respectively with "contamination" of a single nonzero value in each matrix

```
ITERATION 5997
Weight1
[[3.105 3.105 3.105 3.105]
 [3.105 3.105 3.105 3.105]
 [-1.027 -1.027 -1.027 -1.027]]
weight2
[[0.100]
 [0.100]
 [0.100]
 [0.100]]
output
[[0.526]
 [0.588]
 [0.588]
 [0.598]]
ITERATION 5998
Weight1
[[3.105 3.105 3.105 3.105]
 [3.105 3.105 3.105 3.105]
 [-1.028 -1.028 -1.028 -1.028]]
weight2
[[0.100]
 [0.100]
 [0.100]
 [0.100]]
output
[[0.526]
 [0.588]
 [0.588]
 [0.598]]
ITERATION 5999
Weight1
[[3.106 3.106 3.106 3.106]
 [3.106 3.106 3.106 3.106]
 [-1.029 -1.029 -1.029 -1.029]]
weight2
[[0.100]
 [0.100]
 [0.100]
 [0.100]]
output
[[0.526]
 [0.588]
 [0.588]
 [0.598]]
```

Left:
converging on logical indecision.

```
ITERATION 5997
Weight1
[[6.950 1.555 6.950 6.950]
 [6.950 1.555 6.950 6.950]
 [-3.281 -1.567 -3.281 -3.281]]
weight2
[[4.926]
 [-21.877]
 [4.926]
 [4.926]]
output
[[0.038]
 [0.972]
 [0.972]
 [0.037]]
ITERATION 5998
Weight1
[[6.950 1.555 6.950 6.950]
 [6.950 1.555 6.950 6.950]
 [-3.281 -1.567 -3.281 -3.281]]
weight2
[[4.926]
 [-21.877]
 [4.926]
 [4.926]]
output
[[0.038]
 [0.972]
 [0.972]
 [0.037]]
ITERATION 5999
Weight1
[[6.950 1.555 6.950 6.950]
 [6.950 1.555 6.950 6.950]
 [-3.281 -1.567 -3.281 -3.281]]
weight2
[[4.926]
 [-21.878]
 [4.926]
 [4.926]]
output
[[0.038]
 [0.972]
 [0.972]
 [0.037]]
```

Right:
Good outcome at 6000 iterations.
At 600, trend is at least apparent with
Outcome= [0.231,0.818,0.818,0.234]