Problem sheet for Optimisation

1. Activation functions

Sketch each of the following functions. Then calculate the derivative, where you can, and sketch the derivative.

1. identity: $f(a) = a$.
2. threshold: $f(a, \theta) = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{otherwise} \end{cases}$
3. sigmoidal: $f(a) = \frac{1}{1+\exp(-ka)}$
4. tanh: $f(a) = \tanh(a)$
5. rectified linear unit (ReLU): $f(a) = \max(0, a)$

2. Perceptron

Appendix A lists the program `per_rosenblatt_broken.py` that learns a mapping between inputs and outputs in the file `eg2d.dat`. Copy the code into a python notebook and run it. You will need to fix one bug to get the algorithm to work correctly. Hint: check closely the "iteration" for loop. (The data file `eg2d.dat` contains three columns: $x1$ and $x2$ are the two input variables and t is the training signal.)

Try your own 2d data set where the data are linearly inseparable. What happens?

If the error function is changed to:

$$E = (t - y)^2 + \beta w^2$$

then how would you change the weight update rule? Try a range of values for β such as $[0.01, 0.1, 1, 10]$ to see its effect.

3. Multi-layer Perceptron

Appendix B lists the program `xor_broken.py` that is an attempt at solving the XOR problem using a multi-layer perceptron. You need to fix five bugs in the code; look closely at:

1. Definition of gprime.
2. Bias.

3. Step 1b.
4. Step 2a.
5. Step 2c.

Refer to the supplementary handout `backprop2.pdf` for the relevant steps.

How does the network solve the problem? Explain in terms of the features that you found.

Advanced: solve the two-moons data set `moons1.dat` with a one-layer MLP.

A. Code for question 2

```
## This file is broken and has one bug in it.

import numpy as np
import matplotlib.pyplot as plt

## read in data

data = np.loadtxt("eg2d.dat", delimiter=",",skiprows=1)

ninputs = data.shape[0]
wts = np.array([1, 1, 1.5])

def show_points(data, wts, plt, title):
    plt.clf()
    colors=np.array(["red", "blue"])
    plt.scatter(data[:,0], data[:,1], c=colors[data[:,2].astype(int)])
    plt.axis('equal')
    intercept = wts[2]/wts[1] # a
    slope = -wts[0]/wts[1]    # b
    plt.axline( (0, intercept), slope=slope)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.8, 1.5])
    plt.title(title)
    plt.show()

plt.ion()
show_points(data, wts, plt, 'start')

epsilon = 0.03
nepochs = 100


x = np.array([0.0, 0.0, -1])
for epoch in range(nepochs):
    error = 0.0
    order = np.random.choice(ninputs, ninputs,replace=False)

    for iteration in range(ninputs):
        i = order[iteration]
        x[0] = data[i,0]
        x[1] = data[i,1]
```

```
        t    = data[i,2]
        a = np.dot(x, wts)
        y = a > 0
        error = error + (0.5 *(t-y)**2)
        dw = epsilon * (y-t) * x
        wts = wts + dw
    title=f"Epoch {epoch} error {error}"
    print(title)
    show_points(data, wts, plt, title)
    plt.pause(0.05)

## Questions, what happens if you use i=iteration?
## What if you use np.heaviside to calculate output y?  (much quicker)
```

B. Code for question 3

```
import math
import numpy as np
import matplotlib.pyplot as plt

def g(x):
    return ( 1.0 /  (1.0 + math.exp(-x) ))

def gprime(x):
    y = g(x)
    return ( y * (1-y) )


## check these are the right shape.

xs = np.linspace(-3, 3, 100)
gs = [g(x) for x in xs]
gps = [gprime(x) for x in xs]

plt.ion()
plt.clf()
plt.xlabel("x")
plt.ylabel("g or gprime")
plt.plot(xs, gs,  label="g, activation")
plt.plot(xs, gps,  label="gprime")
plt.legend()
plt.show()

bias = -1                       # value of bias unit

epsilon = 0.5

data = np.array([[0, 0, bias, 0],
                 [0, 1, bias, 1],
                 [1, 0, bias, 1],
                 [1, 1, bias, 0],
                 ]
                )

targets = data[:,3]
inputs = data[:,0:3]
```

```
ninputs = inputs.shape[0]

I=2                             # number of input units, excluding bias
J=2                             # number of hidden units, excluding bias
K=1                             # only one output unit

## Weight matrices

W1 = np.random.rand(J,I+1)
W2 = np.random.rand(K,J+1)


y_j = np.zeros(J+1)             # outputs of hidden units
delta_j = np.zeros(J)           # delta for hidden units

nepoch = 2000
errors = np.zeros(nepoch)




for epoch in range(nepoch):

    ## accumulate errors for weight matrices
    DW1 = np.zeros(W1.shape)
    DW2 = np.zeros(W2.shape)
    epoch_err = 0.0

    for i in range(ninputs):

        ## Step 1. Forward propagation activity, adding
        ## bias activity along the way.


        ## 1a - input to hidden
        y_i = inputs[i,:]
        a_j = np.matmul(W1, y_i)

        for q in range(J):
            y_j[q] = g( a_j[q] )

        y_j[J] = bias
```

```
        ## 1b - hidden to output
        a_k = np.matmul(W2, y_j)
        y_k = g(a_k)

        ## 1c - compare output to target
        t_k  = targets[i]
        error = np.sum(0.5 * (t_k - y_k)**2 )
        epoch_err += error

        ## Step 2.  Back propagate activity, calculating
        ## errors and dw along the way.


        ## 2a - output to hidden
        delta_k = gprime(a_k) * (t_k - y_k)
        for q in range(J+1):
            ##for r in range(K):
            r=0
            DW2[r,q] += y_j[q] * delta_k


        ## 2b - calculate delta for hidden layer

        for q in range(J):
            delta_j[q] = gprime(a_j[q]) * delta_k * W2[0,q]

        ## 2c - calculate error for input to hidden weights
        for p in range(I+1):
            for q in range(J):
                DW1[q,p] += y_i[p] * delta_j[q]


    ## end of an epoch - now update weights
    errors[epoch] = epoch_err
    if ( epoch % 50)== 0:
        print(f'Epoch {epoch} error {epoch_err:.4f}')

    W1 = W1 + (epsilon*DW1)
    W2 = W2 + (epsilon*DW2)

## how has it worked?
```