Deepayan Bhadra - hmwk2 Solutions

Solution to Q1(a) to create a classification

```
def create_classification_problem(Nd,Nf,kappa):
    D = buildmat(Nd,Nf,kappa)
    w = np.random.randn(Nf,1);
    c = np.sign(D.dot(w));
    flipper = np.ones((Nd,1)) - 2*(np.random.rand(Nd,1)>.9);
    c = c*flipper;
    return D,c

[ D,c ] = create_classification_problem(100, 2, 1)
valid1 = D[np.where(c==1)[0]]
valid2 = D[np.where(c==-1)[0]]
plt.scatter(valid1[:, 0], valid1[:, 1])
plt.scatter(valid2[:, 0], valid2[:, 1])
plt.title("Scatter plot with condition no = 1")
plt.show()

[ D,c ] = create_classification_problem(100, 2, 10)
valid1 = D[np.where(c==1)[0]]
valid2 = D[np.where(c==-1)[0]]
plt.scatter(valid1[:, 0], valid1[:, 1])
plt.scatter(valid2[:, 0], valid2[:, 1])
plt.title("Scatter plot with condition no = 10")
plt.show()
```

Solution to Q1(b) to evaluate logistic regression

```
def logreg_objective(x,D,c):
    z = np.diagflat(c).dot(D).dot(x)
    idxN, idxP = z<0, z>=0
    y1 = np.sum(-z[idxN]) + np.sum([math.log(np.exp(x)+1) for x in z[idxN]])
    y2 = np.sum([math.log(1+np.exp(-x)) for x in z[idxP]])
    y = y1+y2
    return y

Nd, Nf, kappa = 1000,100,10
[D,c] = create_classification_problem(Nd,Nf,kappa)
x = np.random.randn(Nf,1);
```

```
print("The logistic regression objective is")
y = logreg_objective(x,D,c)
print(y)

input("Press Enter to continue...")

print("The logistic regression objective for a large x is");
x = 10000*np.random.randn(Nf,1)
#Calling on a large input to verify that we don't get any NaNs
y = logreg_objective(x,D,c)
print(y)

input("Press Enter to continue...")
```

Solution to Q2(a): to load the dataset

```
def load_mnist():
    import h5py
    f = h5py.File("mnist.h5")
    x_test = f["x_test"]
    x_train = f["x_train"]
    y_test = f["y_test"]
    y_train = f["y_train"]

    with x_test.astype("float"):
        x_test = x_test[:]
    with y_test.astype("float"):
        y_test = y_test[:]
    with x_train.astype("float"):
        x_train = x_train[:]
    with y_train.astype("float"):
        y_train = y_train[:]
    return x_test,x_train,y_test,y_train

x_test,x_train,y_test,y_train = load_mnist()
```

Solution to Q2(b):to implement SoftPlus

```
def softplus(X):
    y = np.zeros(X.shape)
    idxN, idxP = X<0, X>=0
    xn,xp = X[idxN],X[idxP]
    y[X<0] = [math.log(1+np.exp(x)) for x in xn]
    y[X>=0] = xp+[math.log(np.exp(-x)+1) for x in xp];
    return y
```

Solution to Q2(c):to implement cross-entropy

```
def cross_entropy(X,Y):
    Xm = X - np.max(X,axis = 1, keepdims = True)
    softmax = np.divide(np.exp(Xm),np.sum(np.exp(Xm),axis = 1,keepdims = True))
    y = -math.log(np.mean(np.sum(softmax*Y,axis=1,keepdims = True)))
    return y
```

Solution to Q2(d): to implement neural loss

```
def net_objective(W,D,L):
    temp = D
    for i in range(0,len(W)):
        temp = softplus(np.dot(temp,W[i]))
    y = cross_entropy(temp,L);
    return y
```

Solution to Q2(e): to evaluate objective function on MNIST data

```
D = x_train
W = [None] * 2
W[0] = np.random.randn(784,100)
W[1] = np.random.randn(100,10)
y_train += np.ones((60000,))

# One HOT Encoding
s = pd.Series(y_train)
L = pd.get_dummies(s).values
```

```
print("""The objective value using the random weights and MNIST
      training data is {r:5.3f}""".format(r=net_objective(W,D,L)))

input("Press Enter to continue...")
```

Solution to Q3(a): to compute image gradient

```
def grad2d(X):

    # Computing the x-differences
    kernel = np.zeros(X.shape)
    kernel[0,0] = 1
    kernel[0,1] = -1
    Dx = np.fft.fft2(kernel)
    Gx = np.fft.ifft2(Dx*np.fft.fft2(X))

    #Computing the y-differences
    kernel = np.zeros(X.shape)
    kernel[0,0] = 1
    kernel[1,0] = -1
    Dy = np.fft.fft2(kernel)
    Gy = np.fft.ifft2(Dy*np.fft.fft2(X))

    return np.stack([Gx, Gy], axis=2)
```

Solution to Q3(b): to compute (-ve) divergence

```
def div2d(G):
    # Computing the x-differences
    #  extract the x- and y- derivatives
    Gx = G[:,:,0]
    Gy = G[:,:,1]

# We must convolve with the FLIPPED stencil to get the linear filtering we want
    kernel = np.zeros(Gx.shape)
    kernel[0,0]=1
    kernel[0][-1] =-1

#  create diagonal matrix in decomposition K=F'DF
    Dx = np.fft.fft2(kernel)

#  Use the eigen-decomposition to convolve the stencil with X, and get the
```

```
#  differences in the horizontal direction.
    Divx = np.fft.ifft2(Dx*np.fft.fft2(Gx))

    kernel = np.zeros(Gy.shape)
    kernel[0,0]=1
    kernel[-1][0]=-1

    Dy = np.fft.fft2(kernel)

    Divy = np.fft.ifft2(Dy*np.fft.fft2(Gy))

    return Divx+Divy
```

Solution to Q3(c): to verify adjointness using $< x, Ay > \quad = \quad < A'x, y >$ where A = G or d

```
print('\nTesting adjoints\n')
# random inputs
x = np.random.randn(100,100);
y = np.random.randn(100,100,2)
Ay = grad2d(y)
x_Ay = np.sum(np.conj(x)*Ay)
#  Compute Ax, where A is the gradient operator
Ax = grad2d(x)
#  Compute Aty, where A is the gradient operator
Aty = div2d(y)

left = np.inner(Ax.flatten(),y.flatten(1))
right = np.inner(x.flatten(),Aty.flatten(1))
print('first inner product = \t',left,'\n')
print('second inner product = \t',right)
```