SML_assignment4_ex2

January 10, 2017

1 Exercise 2 - Neural network regression

2 1

```
In [74]: import seaborn as sb
         from scipy.stats import multivariate_normal as mv
         from mpl_toolkits.mplot3d import Axes3D
         from pylab import *
         from numpy import *
         # q1
         # create the target distribution
         # sample at .1 interval
         tmp = arange(-2, 2, .1)
         x, y = meshgrid(tmp, tmp)

         mu = [0,0]
         sigma = eye(len(mu)) * 2/5
         dist = mv(mu, sigma)

         X = vstack((x.flatten(), y.flatten())).T
         Y = dist.pdf(X).reshape(x.shape)  * 3
         targets = array(Y.flatten(), ndmin = 2).T

         fig, ax = subplots(1, 1, subplot_kw = {'projection': '3d'})
         ax.plot_surface(x, y, targets.reshape(x.shape))
         ax.set_xlabel('$x_1$', labelpad = 20)
         ax.set_ylabel('$x_2$', labelpad = 20)
         ax.set_zlabel('pdf', labelpad =20)
         sb.set_context('poster')
         sb.set_style('white')
         fig.suptitle('Target distribution')
         savefig('../Figures/2.1.png')
         show()
```

3 2

In [88]:
```
class mlp(object):
    '''
    Multi-layered-pereptron
    K = output nodes
    M = hidden nodes
    Assumes the input data X is samples x feature dimension
    Returns:
        prediction and error
    '''
    def __init__(self, X, t,\
                 eta = 1e-1,\
                 gamma = .0,\
                 M = 8,\
                 K = 1):
        # learning rate / momentum rate
        self.eta          = eta
        self.gamma        = gamma
        # Layer dimensions; input, hidden, output
        self.D            = D =  X.shape[1] + 1
        self.M            = M
        self.K            = K
        # add bias node to input
        self.X            = hstack( (X, ones(( X.shape[0], 1 ) ) ) )
        self.targets      = t
        # weights; hidden and output
        wh                = random.rand(D, M) - 1/2
        wo                = random.rand(M, K) - 1/2

        self.layers       = [wh, wo]
        # activation functions:
        self.func         = lambda x: tanh(x)
        self.dfunc        = lambda x: 1 - x**2


    def forwardSingle(self, xi):
        ''' Performs a single forward pass in the network'''
        layerOutputs = [ []  for j in self.layers ]
        #forward pass
        a = xi.dot(self.layers[0])
        z = self.func(a)
        y = z.dot(self.layers[1])

        # save output
        layerOutputs[0].append(z);
        layerOutputs[1].append(y)
        return layerOutputs
```

```python
def backwardsSingle(self, ti, xi, forwardPass):
    '''Backprop + update of weights'''
    # prediction error
    dk = forwardPass[-1][0] - ti
    squaredError = dk**2
    # compute hidden activation; note elementwise product!!
    dj = \
    self.dfunc(forwardPass[0][0]) * (dk.dot(self.layers[-1].T))

    # update the weights
    E1 = forwardPass[0][0].T.dot(dk)
    E2 = xi.T.dot(dj)

    # update weights of layers
    self.layers[-1] -= \
    self.eta * E1 + self.gamma * self.layers[-1]

    self.layers[0]  -= \
    self.eta * E2 + self.gamma * self.layers[0]
    return squaredError

def train(self, num, plotProg = (False,)):
    #set up figure
    if plotProg[0]:
        fig, ax = subplots(subplot_kw = {'projection':'3d'})


    num   = int(num) # for scientific notation
    SSE   = zeros(num) # sum squared error
    preds = zeros((num, len(self.targets))) # predictions per run
    for iter in range(num):
        error = 0 # sum squared error
        for idx, (ti, xi) in enumerate(zip(self.targets, self.X)):
            xi = array(xi, ndmin = 2)

            forwardPass = self.forwardSingle(xi)
            error += self.backwardsSingle(ti, xi, forwardPass)
            preds[iter, idx] = forwardPass[-1][0]
        # plot progress
        if plotProg[0]:
            if not iter % plotProg[1]:
                x, y = plotProg[2]
                ax.cla() # ugly workaround
                ax.plot_surface(x, y, preds[iter, :].reshape(x.shape))
                ax.set_xlabel('$x_1$', labelpad = 20)
                ax.set_ylabel('$x_2$', labelpad = 20)
                ax.set_zlabel('pdf', labelpad =20)
```

```
                    ax.set_title('Cycle = {0}'.format( iter ))
                    pause(1e-10)
            SSE[iter] = .5 * error
        return SSE, preds

# perform a single forward pass and show the results
model = mlp(X, targets)
# perform a single pass
preds = array([\
              model.forwardSingle(\
              array(hstack( ( xi, 1) ),\
                    ndmin = 2))[-1]\
              for xi in X]).flatten()
# plot the results
fig, ax = subplots(subplot_kw  = {'projection': '3d'})
ax.scatter(x, y, preds.reshape(x.shape))
ax.set_xlabel('$x_1$', labelpad = 20)
ax.set_ylabel('$x_2$', labelpad = 20)
ax.set_zlabel('pdf', labelpad =20)
ax.set_title('Network output without training')
sb.set_context('poster')
savefig('../Figures/2.2.png')
show()
```

4 3

```
In [89]: # run at atleast 500 cycles
         num = int(5e2) + 1
         model = mlp(X, targets)
         # train the model
         SSE, preds = model.train(num)
```

```
In [82]: # plot every 250 cycles
         cycleRange = arange(0,  num, num // 5)

         # use at most 3 rows
         nRows = 3
         nCols = int(ceil(len(cycleRange)/nRows))
         nCols = 2

         fig, axes  = subplots(nRows,\
                               nCols,\
                               subplot_kw = {'projection': '3d'})


         for axi, i in enumerate(cycleRange):
             ax = axes.flatten()[axi]
```

```
        ax.plot_surface(\
                        x,\
                        y,\
                        preds[i,:].reshape(x.shape),\
                        cstride = 1,\
                        rstride = 1)
        # formatting of plot
        ax.set_xlabel('$x_1$', labelpad = 20)
        ax.set_ylabel('$x_2$', labelpad = 20)
        ax.set_zlabel('pdf', labelpad =20)
        ax.set_title('cycles = {0}'.format(i))
        sb.set_style('white')
    fig.delaxes(axes.flatten()[-1])
    fig.suptitle('Output of MLP as a function of complete cycles')
    subplots_adjust(top=0.8)
    # fig.tight_layout()
    savefig('../Figures/2.3.png')
    show()
```

5 4

```
In [87]: # shuffle the indices
         idx = random.permutation(len(targets))

         # shuffle the data
         shuffleX = X[idx,:]
         # shuffle the targets as well (same indices)
         shuffleTargets = targets[idx, :]
         shuffleModel = mlp(shuffleX, shuffleTargets)
         shuffleSSE, shufflePreds = shuffleModel.train(num)
```

```
In [90]: # plot the results: compare with error from [3]
         fig, ax = subplots()
         ax.plot( range(num),SSE,\
                 range(num), shuffleSSE)

         ax.set_xlabel('Iterations')
         ax.set_ylabel('Sum squared error')
         ax.legend(['shuffled','non-shuffled'],loc = 0)
         fig.suptitle('Training error')
         savefig('../Figures/2.4.png')
         show()
```

Since the grid is linearly spaced, this will mean that nearby points will yield the same gradient in the error. This 'local' correlation will yield that the algorithm will change the weights in similar direction for a while, hence shuffling the data removes this 'local' correlation structure, yielding more likely to move in the different directions, constraining the algorithm, yielding faster convergence.

6 5

In [91]:
```
# load the data
X, Y, target = list(np.loadtxt('../Data/a017_NNpdfGaussMix.txt').T)
tmp = int(np.sqrt(target.shape[0]))
# convert in shape for it to be plottable
x = X.reshape(tmp,tmp)

y = Y.reshape(tmp, tmp)
# stack to create input data
X = np.vstack((X,Y)).T

# target vector
target = np.array(target, ndmin = 2).T
```

In [92]:
```
# visualize the target distribution
fig, ax = subplots(1,1, subplot_kw = {'projection': '3d'})
ax.plot_surface(x, y, target.reshape(tmp, tmp))
ax.set_xlabel('$x_1$', labelpad = 20)
ax.set_ylabel('$x_2$', labelpad = 20)
ax.set_zlabel('pdf', labelpad = 20)
fig.suptitle('Target distribution')
savefig('../Figures/2.5.png')
show()
```

7 6

In [93]:
```
#randomly permute indices
idx = np.random.permutation(range(len(target)))
# keep track of the changes
shuffX = X[idx,:]; shuffTarget = target[idx]
# run mlp with eta = .01
model = mlp(shuffX, shuffTarget,\
                    eta = 1e-2,\
                    M = 40)

# run for 2000 complete cylcles
num = int(2e3)
errors, preds = model.train(num = num)
# map back to original space
orgIdx = argsort(idx)
# get final prediction
finalPred = preds[-1, orgIdx]
```

In [94]:
```
# plot the final prediction and the target distribution
fig, ax = subplots(subplot_kw = {'projection': '3d'})
ax.scatter(x, y, target.reshape(x.shape), label = 'target')
ax.scatter(x, y, finalPred.reshape(x.shape), label = 'estimation')
```

```
ax.set_xlabel('$x_1$', labelpad = 20)
ax.set_ylabel('$x_2$', labelpad = 20)
ax.set_zlabel('pdf', labelpad   = 20)
ax.legend(loc = 0)
fig.suptitle('Final prediction after {0} cycles'.format(num))
savefig('../Figures/2.61.png')

fig, ax = subplots()
ax.plot(errors)
ax.set_xlabel('iterations')
ax.set_ylabel('Sum squared error')
ax.set_title('Training error')
savefig('../Figures/2.62.png')

show()
```

One might improve the performance by adding more hidden nodes to the network. The hidden nodes essentially represent the degrees of freedom in the model. Increasing the number of hidden nodes might increase the fit on a trainingset, however it will also increase the modelling noise (i.e overfitting). Improvements might also be found in presenting the inputs / targets in a different feature space. Multi-layered perceptrons are notorious for being sensitive to how the data is represented. Another might be instead of taking a global learning rate, is make it adaptive (see conjugate gradient descent, momentum etc).

8 7

We are using python hence netlab toolbox is not available to us. We opted for the neurolab toolbox which has a conjugate gradient method. The same parameters were used as in 7 to improve comparison.

```
In [16]: import neurolab as nl
         from scipy.optimize import fmin_ncg
         # inputs and targets
         inp = shuffX
         tar = shuffTarget

         # specify same network structure as we have
         # i.e. M = 40, D = 3, K = 1
         # this function takes min/max of input space

         # Specify activation functions;
         # input - to hidden  is hyperbolic tangent
         # hidden- to out is linear
         tranfs = [nl.trans.TanSig(), nl.trans.PureLin()]
         net = nl.net.newff(\
                             [ [np.min(X), np.max(X)] ] * inp.shape[1],\
                             [40, 1],\
                             transf= tranfs,\
```

```
                     )
        # use conjugate gradient as method
        net.trainf = nl.train.train_ncg # conjugate gradient
        # net.trainf = fmin_ncg
        net.errorf  = nl.error.SSE()     # same as above

        # init weight matrix between -.5,.5
        for l in net.layers:
            l.initf = nl.init.InitRand([-.5, .5], 'wb')
        net.init()

        # Train network; show output ever 500 iterations
        errorNL = net.train(inp, tar, epochs= num, show = 500)

        # Simulate network
        outNL = net.sim(inp)
        # sort back to original indices
        outNL = outNL[argsort(idx)]
```

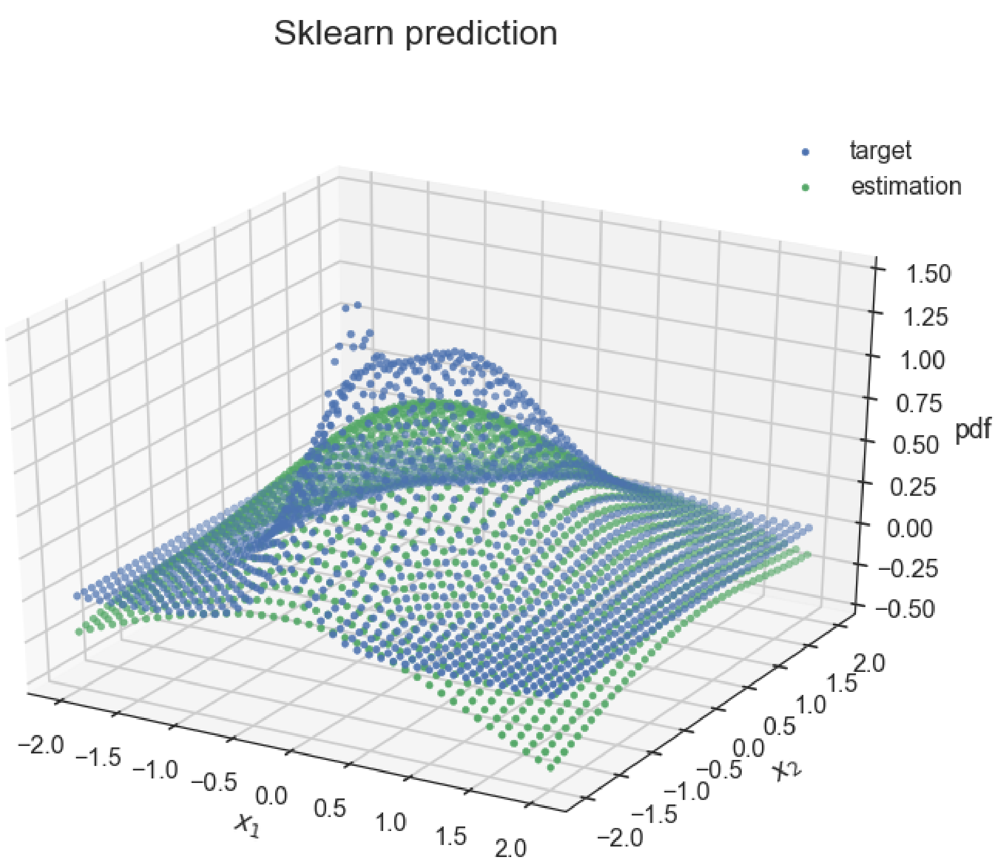
```
Epoch: 10; Error: 5.887266430718493;
Epoch: 20; Error: 3.517431483655753;
Epoch: 30; Error: 2.55336675093663;
Epoch: 40; Error: 2.247242834589139;
Epoch: 50; Error: 1.8303922893463396;
Epoch: 60; Error: 1.5999082835264171;
Epoch: 70; Error: 1.4111868109144738;
Epoch: 80; Error: 1.3008636235167548;
Epoch: 90; Error: 1.2518343611218838;
Optimization terminated successfully.
         Current function value: 1.233727
         Iterations: 94
         Function evaluations: 134
         Gradient evaluations: 5239
         Hessian evaluations: 0
4.49810048069 23.3882092042
```

```
In [138]: # plot the performance versus our algorithm
          # plot the final prediction and the target distribution
          fig, ax = subplots(subplot_kw = {'projection': '3d'})
          ax.scatter(x, y,\
                     target.reshape(x.shape), label = 'target')
          ax.scatter(x, y,\
                     outNL.reshape(x.shape), label = 'estimation')

          ax.set_xlabel('$x_1$', labelpad = 20)
          ax.set_ylabel('$x_2$', labelpad = 20)
          ax.set_zlabel('pdf', labelpad = 20)
```

```
        ax.legend(loc = 0)
        fig.suptitle('Neurolab prediction')
        print(\
        'final SSE neruolab :\n {0}'\
        .format(errorSklearn))
        print(\
        'final SSE our alogorithm:\n{0}'.format(errors[-1]))
        savefig('../Figures/2.7.png')
        show()
```

```
(1681, 1) (41, 41)
final SSE sklearn :
 2.45558501298368
final SSE our alogorithm:
2.383194490516197
```



In [36]:

```
C:\Program Files\Anaconda3\lib\site-packages\matplotlib\backend_bases.py:2442: Matp
  warnings.warn(str, mplDeprecation)
```

```
    ---------------------------------------------------------------------------

    TclError                                  Traceback (most recent call last)

    <ipython-input-36-cc32c1419b3d> in <module>()
     15
     16 for i in range(0, 20):
---> 17     plt.pause(1)
     18
     19     Y = np.random.rand(100, 3)*5


    C:\Program Files\Anaconda3\lib\site-packages\matplotlib\pyplot.py in pause(
    297                 canvas.draw()
    298             show(block=False)
--> 299             canvas.start_event_loop(interval)
    300             return
    301


    C:\Program Files\Anaconda3\lib\site-packages\matplotlib\backends\backend_tk
    515
    516     def start_event_loop(self,timeout):
--> 517         FigureCanvasBase.start_event_loop_default(self,timeout)
    518     start_event_loop.__doc__=FigureCanvasBase.start_event_loop_default.
    519


    C:\Program Files\Anaconda3\lib\site-packages\matplotlib\backend_bases.py in
   2448         self._looping = True
   2449         while self._looping and counter * timestep < timeout:
-> 2450             self.flush_events()
   2451             time.sleep(timestep)
   2452             counter += 1


    C:\Program Files\Anaconda3\lib\site-packages\matplotlib\backends\backend_tk
    512
    513     def flush_events(self):
--> 514         self._master.update()
    515
    516     def start_event_loop(self,timeout):


    C:\Program Files\Anaconda3\lib\tkinter\__init__.py in update(self)
   1023     def update(self):
   1024         """Enter event loop until all pending events have been processe
```

```
-> 1025         self.tk.call('update')
   1026     def update_idletasks(self):
   1027         """Enter event loop until all idle callbacks have been called.

    TclError: can't invoke "update" command: application has been destroyed
```