Exercise 2 - Neural network regression

January 12, 2017

2.1

This is fairly trivial. Essentially, we make a meshgrid and use the object from scipy to compute the pdf over this meshgrid. Figure 1 shows the target distribution obtained from this process.

```
In [96]: 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()
```

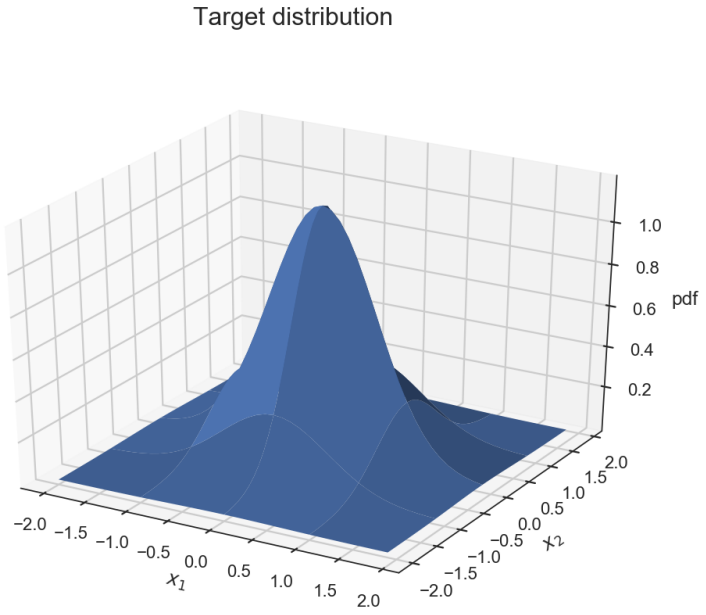

Figure 1: Target distribution for the MLP in the later exercises

2.1

The code provided below allows for plotting the progress over time. The naive output of the network is depicted in figure 2. The algorithm works by performing a forward pass, then feeding the error back through the network.

The forward pass uses bisschop eq 5.62, 5.63, 5.64. The backward pass uses the output to generate a delta (eq 5.65), and update the hidden layer weights for both the output connections and input connections (eq 5.67).

```
In [97]: 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

    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()
```

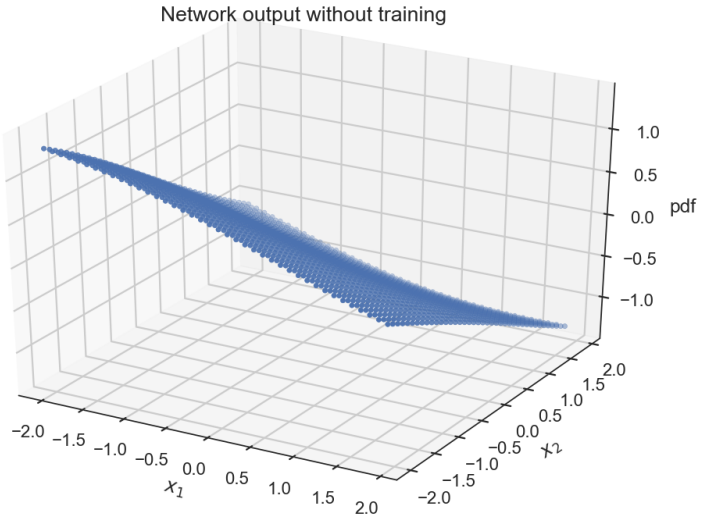

Figure 2: Results for 2.2; output of the network without any training. The output is a 2D plane since the weight have been sampled uniformly.

2.3

If one wants to please parse the correct parameter, i.e. plotProg = (True, interval, [x,y]). As summary of the training progress is depicted in figure 3. It is interesting to see that even after 1 full training cycle (top left plot in 3), we already can see a gaussian shape, it only get more and more refined as a function of training cycles. The MLP learns the most when its output is very wrong compared to the target. Hence, the 2D shee from figure 2, will be adjusted a lot after the first pass, but consecutive passes will learn less at the output is already farely close to the desired output. Please note that cylce = 0 is actually one fully completed training cycle as python starts indexing from 0. The zero training is shown in the previous question, i.e. figure 2.

```
In [98]: # 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()
```

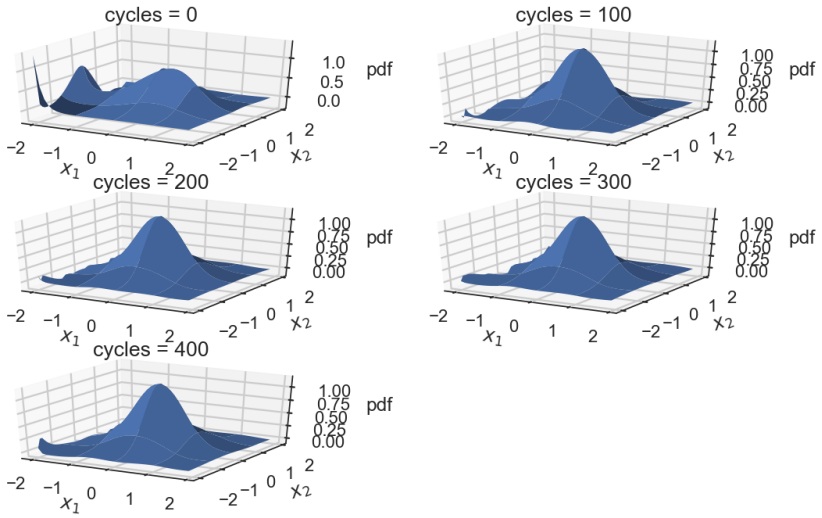

Figure 3: This figure shows the output of the network as a function of training cycles. Note that cycle = 0 is after 1 full pass of the network. Python starts indexing from 0. This if for the non-shuffled data.

2.4

Permute the indices and plot the sum squared error for the training. We expect the shuffled condition to have faster convergence. Results are shown in figure 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()
```

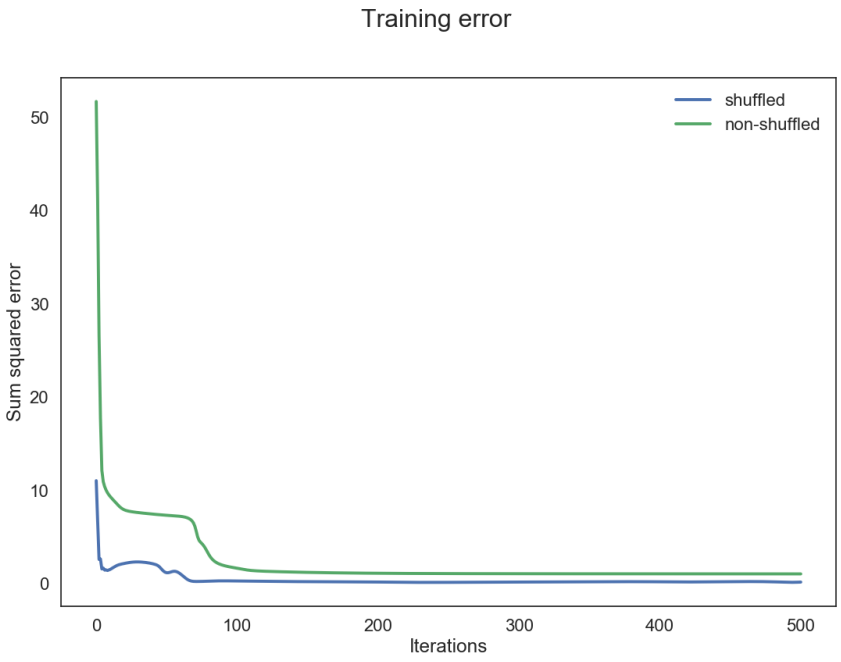

Figure 4: The sum squared error as a function of training cycles. Note a full cycle is after seeing all the training samples, weights are updated after seeing a datapoint. Noticeably, we see that the shuffled data is faster in converging, for explanation see text.

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. To summarize, the linearly spaced inputs will yield smaller gradients and thus slower learning as the network 'locally' overtrains. Randomizing inputs will yield larger gradients and thus the network will be better able to adjust the weights such that the prediction is generalized over the entire dataset.

2.5

Same as in section 1. The target distribution is shown in figure 5.

```
In [99]: # 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()
```

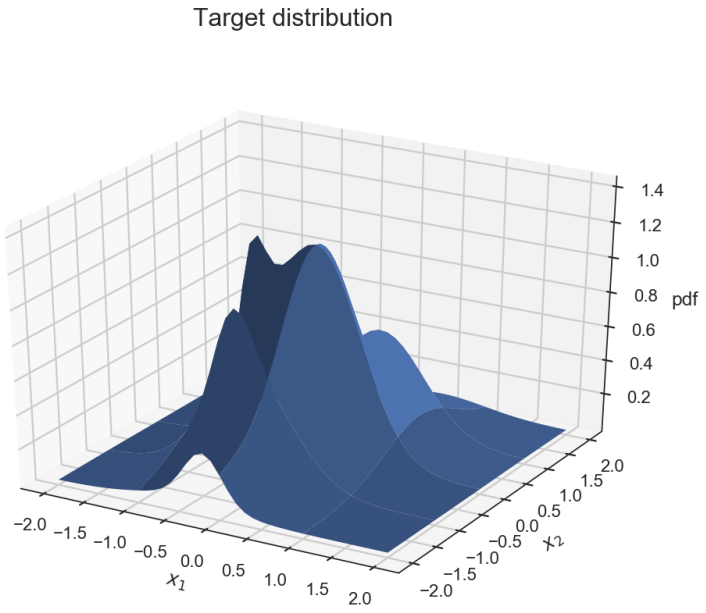

Figure 5: Target distribution loaded from the provided data files

2.6

The training error and final prediction are shown in figures 7, 6. Explanation is below the code block.

```
In [100]: #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()
```

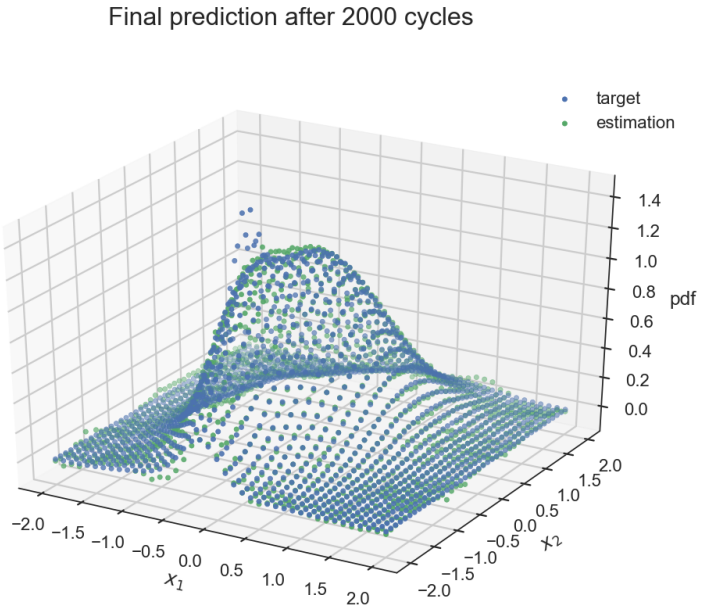

Figure 6: In green we gave the prediction generated from our network, for parameters please see the code block in this exercise. The performance of training is shown in figure 7. Note that the little hump is not captured, but the sum squared error remains low.

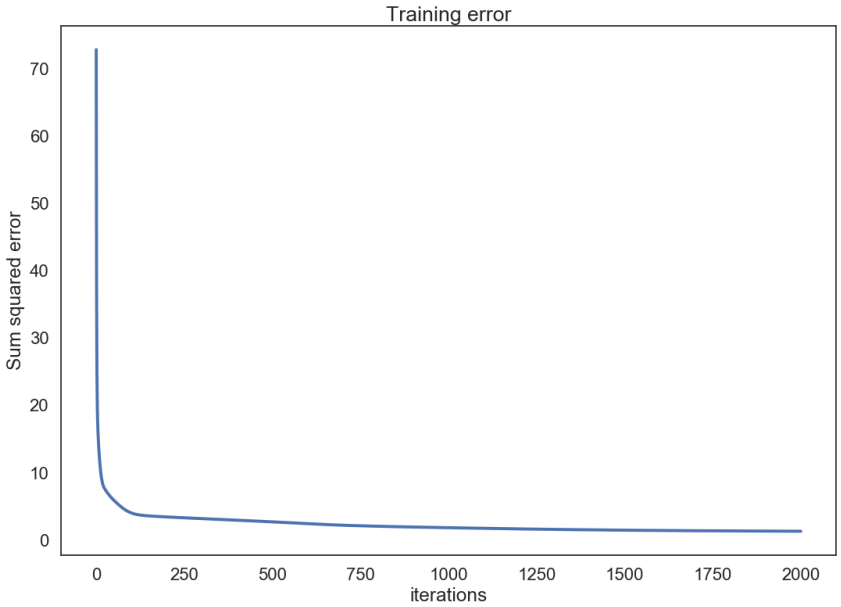

Figure 7: Sum squared error as a function of completed training cycles.

One might improve the performance by adding more hidden nodes to the network. The hid-

den 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).

2.7

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. The final output for the neurolab is represented in figure 8. Comparisons of the training errors are displayed in figure 9. Note, that the conjugate gradient stopped after 25 cycles, the output stated that the algorithm didn't converge. Hence, we zoomed in on the plot. The error was lower than our implementation however.

```
In [101]: 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)]
```

```
Warning: Warning: CG iterations didn't converge.  The Hessian is not positive defir
         Current function value: 2.973533
         Iterations: 25
         Function evaluations: 36
         Gradient evaluations: 6893
         Hessian evaluations: 0
```

In [105]:
```
            # 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(errorNL[-1]))
            print(\
            'final SSE our alogorithm:\n{0}'.format(errors[-1]))
            savefig('../Figures/2.7.png')
            show()

            # plot training errors
            fig, ax = subplots()
            ax.plot(errorNL, label = 'conjugate gradient')
            ax.plot(errors, label = 'our MLP')
            ax.set_xlabel('Training cycles')
            ax.set_ylabel('Sum squared error')
            fig.suptitle('Conjugate gradient convergence')
            # savefig('../Figures/2.71.png')
            show()
```

```
final SSE neruolab :
 2.9735334308131867
```

```
final SSE our alogorithm:
1.3464563919674712
```

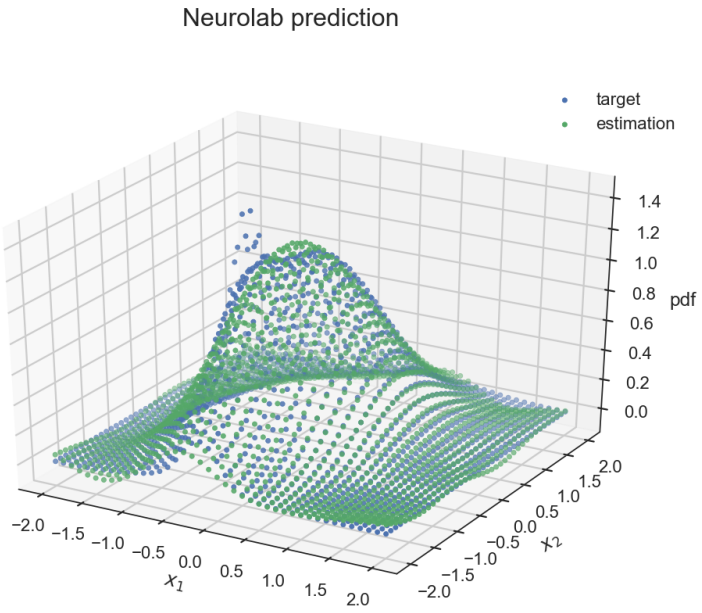

Figure 8: Output of conjugate gradient from neurolab toolbox.

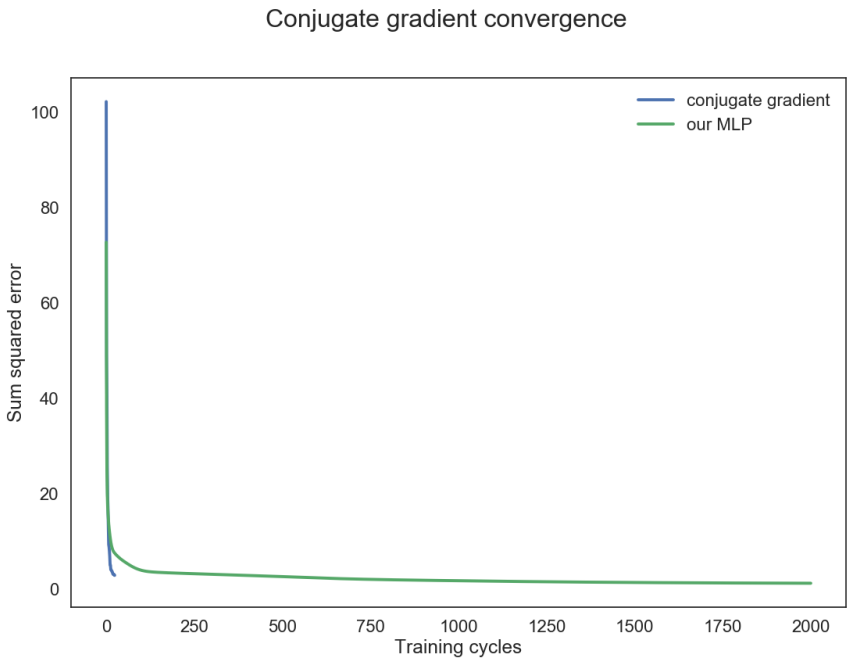

Figure 9: Training error of the neurolab implementation versus ours. Note that the rate of convergence if faster than our implementation. However, it did halt after 25 iterations due to a failure to converge. It uses the methods from scipy.