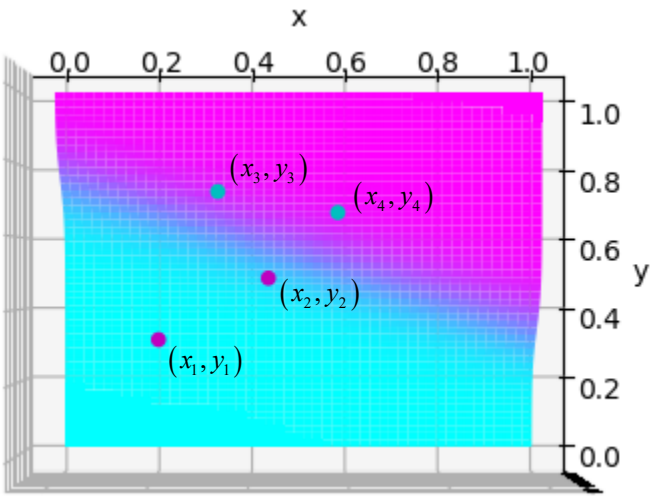
MTRE 2610 – Intermediate Programming for Mechatronics

Homework – Python Applications

1. This problem trains a simple artificial neural network (ANN) with a single neuron to divide the *x*-*y* plane into the classes of blue and yellow points shown below. These four points are saved in the file `data.txt` where the first column contain *x* values and the second column *y*.



Output of the neural network is defined by the equation

$$z = \frac{1}{1+e^{-(ux+vy+b)}}$$

which is computed for any (*x*,*y*) point given the network parameters *u*, *v*, and *b*. These network parameters must be chosen to best classify points 1 and 2 as belonging to one class and points 3 and 4 to another. To accomplish that, it is desired that *z* be near zero in the vicinity of points 1 and 2 (cyan shading in the graph above) while producing near unity values for points 3 and 4 (magenta shading). Accuracy of the ANN is evaluated by the sum squared error

$$E = \left[0 - \frac{1}{1+e^{-(ux_1+vy_1+b)}}\right]^2 + \left[0 - \frac{1}{1+e^{-(ux_2+vy_2+b)}}\right]^2 + \left[1 - \frac{1}{1+e^{-(ux_3+vy_3+b)}}\right]^2 + \left[1 - \frac{1}{1+e^{-(ux_4+vy_4+b)}}\right]^2$$

which judges how close the ANN models the desired values for the four (*x*,*y*) pairs. Optimal values for network parameters *u*, *v*, and *b* are determined through gradient descent, which aims to update them depending on how sensitive the error is to changes in them, i.e. parameters affecting the error more should be changed more. To begin, choose random values for *u*, *v*, and *b*. The three parameters are updated according to

$$u_{i+1} = u_i - \eta \frac{\partial E}{\partial u},\ v_{i+1} = v_i - \eta \frac{\partial E}{\partial v},\ \text{and}\ b_{i+1} = b_i - \eta \frac{\partial E}{\partial b}$$

where η is the learning rate, and the subscript *i* indicates the current value and *i*+1 the new updated value. Note the minus sign in these expressions reflects that derivatives indicate an *increase* in error whereas *decreasing* error is desired. Repeatedly updating these parameters will eventually converge on a solution with a sufficiently low error measurement. ANNs trained with large learning rates will train faster, but may suffer from instability for values too high. For this problem, $\eta = 7$ is reasonable, and updating parameters 50 times should be sufficient. Finding the required

partial derivatives is cumbersome by hand, but can be found easily with the sympy symbolic math module. Remember that x_1 and y_1 through x_4 and y_4 are constants throughout this problem.

Create symbolic variables for E, u, v, and b as well as the three derivatives. Then initialize the network parameters u, v, and b to random values between 0 and 1. Write a loop to iteratively update their values according to the gradient descent equations. To visualize accuracy of the ANN, evaluate z on a `numpy.meshgrid` for the entire domain. Overlay the resulting `surf`ace plot and the four points used for training with their appropriate colors, and use `mpl_toolkits.mplot3d.axes3d.view_init(90,-90)` to look directly down the z axis. The plot can be repeatedly produced inside the loop to view how the solution changes during training (although this is not required), just be sure to include `matplotlib.pyplot.pause` to update the figure every iteration. The final output should appear similar to the figure above. The example in the next problem can be used as a reference for visualizing this problem.

Useful functions:

- `numpy.loadtxt, numpy.random.random, numpy.exp`
- `sympy.symbols, sympy.diff, sympy.exp, sympy.subs`
- `mpl_toolkits.mplot3d.axes3d.plot_surface, mpl_toolkits.mplot3d.axes3d.scatter`

2. In problem 1, the ANN and its optimization was coded from scratch. This is possible for small ANNs, but becomes increasingly difficult for deep learning projects. PyTorch is a Python module supporting training of ANNs. In order to use PyTorch, a custom class must be created that inherits from `torch.nn.Module`. The default constructor must be overloaded and the `forward` function defined to take in the ANN input data and return the ANN output.

When training, data is organized with one variable for the input data, and another for the desired output data (also called ground truth labels). These variables must be of the torch tensor data type, with functions to convert between numpy arrays and torch tensors (for example, see lines 30 and 61 in the sample code). The data in most deep learning problems is so large that it cannot be processed at once, requiring it be handled in batches. In this problem, a single batch is reasonable although Torch still requires the single batch be organized. For this reason, the input training data (see line 26) is a 1×4×2 three-dimensional array (technically a Torch tensor) where the first dimension has size 1 (for the one batch), the second dimension is size 4 (for the four data points), and the third dimension is size 2 (for the two inputs, x and y). The output labels (see line 29) has size 1×4×1 corresponding to 1 batch, 4 data points, and 1 output. When evaluating the ANN over the entire domain, the input data (see line 40) has dimension 1×2500×2 where the 50×50 `meshgrid` contains 2500 data points.

Before training, an instance of the class is initialized (line 43). How ANN performance is measured (line 44) is often defined using the mean squared error (MSE) which is the same as the definition of E in the previous problem, except dividing by the number of data points (four in this case) to give a *mean* squared error instead of a *sum* squared error. Finally, the strategy for updating parameters is defined (line 45). The stochastic gradient descent (SGD) used here is a more complicated version of the gradient descent equations used in the previous problem that, among other things, supports training in multiple batches. Use this same code for the current problem, although the learning rate (LR) in line 45 will need to be modified.

Like the previous problem, training occurs in a loop (line 47), and the code for performing an iteration of updating ANN parameters (lines 52-56) is the same for every instance of an ANN class (assuming the same variable names are used). Output for inputs different from the training data can be evaluated (line 60) and then visualized (lines 64-73).

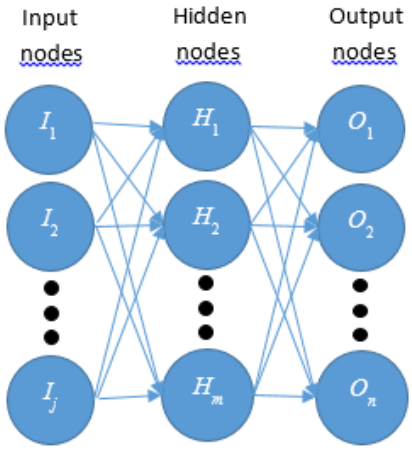
```
import numpy as np
import torch
import matplotlib.pyplot as plot
from mpl_toolkits.mplot3d import Axes3D

class Net(torch.nn.Module): # Custom ANN inherits from base class
    def __init__(self):                    # Default constructor
        super(Net, self).__init__()        # Run base class constructor
        self.fc1 = torch.nn.Linear(2, 1)   # Two inputs and one output
        self.sigmoid = torch.nn.Sigmoid() # Sigmoid function for output z
    def forward(self, x):                  # Function to return ANN output
        return self.sigmoid(self.fc1(x))   # Return output for input x

# Load four data points for training
data = np.loadtxt('data.txt')
x = data[:,0] # Pull out x values for plotting later
y = data[:,1] # Pull out y values for plotting later
#### Convert numpy data into torch tensor format required for training
data = np.reshape(data,(1,4,2)) # Reshape 4x2 matrix of data into one batch
data   = torch.tensor(data,dtype=torch.float) # Convert numpy to torch tensor
labels = [0,0,1,1] # Ground truth labels ANN is to be trained with
labels = np.reshape(labels,(1,4,1)) # Reshape array:1 batch, 4 values, 1 output
labels = torch.tensor(labels,dtype=torch.float) # Convert numpy to torch tensor
# Create data for entire x-y plan for evaluation
res = 50 # Resolution of meshgrid
# Meshgrid of all x and y values in a grid in the x-y plane
xGrid, yGrid = np.meshgrid(np.linspace(0,1,res),np.linspace(0,1,res))
# Take the two matrices in the meshgrid and place all x values in first column
# and all y values in second column
domainData = np.column_stack((np.reshape(xGrid,(res*res,)),
                              np.reshape(yGrid,(res*res,))))
# Convert numpy data into torch data where the first dimension has only 1 batch
domainData = torch.tensor(np.reshape(domainData,(1,res*res,2)),
                          dtype=torch.float)
#### Declare and set up ANN
model = Net()                                          # Create ANN object
criterion = torch.nn.MSELoss()                         # Set loss function
optimizer = torch.optim.SGD(model.parameters(),lr=10) # Set learning rule
#### Loop to train ANN
for i in range(100):
    #### Standard code for iteration to update ANN parameters giving inputs
    #### stored in variable data and ground truth labels stored in variable
    #### labels. If you use the same variable names in your code, then this
    #### section will remain unchanged in your solution.
    optimizer.zero_grad()
    prediction = model(data)
    loss = criterion(prediction, labels)
    loss.backward()
    optimizer.step()
    #### End of standard code
```

```
    print(loss.item()) # Display error in training data
    #### Get ANN output for entire x,y domain and convert to numpy meshgrid
    output = model(domainData)  # Predict output for all points in x,y plane
    output = output.detach().numpy() # Convert torch tensor to numpy array
    output = np.reshape(output,(res,res)) # Reshape to be shape of meshgrid
    #### Plot ANN output and training data
    fig = plot.figure(1)          # Open figure
    plot.clf()                    # Clear Figure
    ax = fig.gca(projection="3d") # Set axis to be 3D
    ax.plot_surface(xGrid,yGrid,output,cmap='cool') # Surface plot ANN output
    ax.scatter(x[0:2],y[0:2],[2,2],color='m') # Training data for one class
    ax.scatter(x[2:4],y[2:4],[2,2],color='c') # Training data for other class
    plot.xlabel('x')      # x axis label
    plot.ylabel('y')      # y axis label
    ax.view_init(90,-90) # Rotate to look directly down on surface
    plot.pause(0.001)     # Pause so that figure will update
print(prediction) # Print output, near zero for class 1, near unity for class 2
```

The solution to this problem, will train an ANN to recognize whether an image contains a face or not. The grayscale images are reasonably small, 19 pixels by 19 pixels, so each image produces 19×19 = 361 input values. Like previously, there will be a single output where zero indicates "not a face" and unity "is a face". Use a more complicated architecture than previously, where additional "hidden" nodes are inserted between the input and output nodes, see the figure below. In this problem $j = 361$ for the inputs, $m = 50$ for the hidden nodes, and $n = 1$ for a single output node. Use the example given above as a template for building the solution, and more information is available in the online post PyTorch: Introduction to Neural Network – Feedforward / MLP. Especially useful is the class inheritance under the heading "Feedforward Neural Network", where `input_size` is 361 and `hidden_size` is 50 in our case (this online example, like our situation, has a single output). Use either sigmoid (as all examples here have) or ReLU (as the online post does) for the hidden nodes, but be sure of the sigmoid on the output node (as in all examples).



The input data is organized as two folders, one with 50 images of faces and another folder of not face images. The `glob.glob` function returns a list containing strings of all the filenames in a given directory. Loop through each filename in both of the directories to read in each image, and reshape the 19×19 image into a single row to store the input data. The result will be two Torch tensors, one for the pixel inputs values and another for the ground truth outputs for each image. The input data will be of size 1×100×361 (1 batch, 100 images, 361 pixel values) and the output 1×100×1 (1 batch, 100 images, 1 output). The ground truth output values should be 0 (not a face) or 1 (is a face). Run a sufficient number of iterations so that the ANN correctly classifies all 100 images, i.e. the output is greater than 0.5 for faces, and less than 0.5 for not faces. **Important**: use a learning rate of 0.04 when defining `torch.optim.SGD` as the optimizer.