Intelligent Systems HW4 Part 1

# Problem Description

Given a data set of 16 animals, each labeled with 13 attributes, develop a self-organizing feature map with 100 neurons in a 10 by 10 grid.

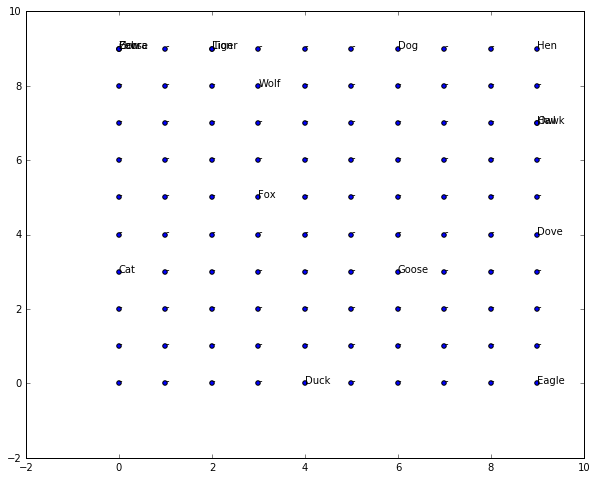
# Approach

The code was developed as a modular python library that can be used in a general manner. The library consists of an SOFMGrid class and a Neuron class. The SOFMGrid has a “train()” function which accepts a set of training data, the number of organizing epochs, and the number of fine-tuning epochs. The SOFM grid uses a time decayed learning rate, as well as a time decayed sigma value for the radial distance function. The radial distance function is implemented as a standard Gaussian function.

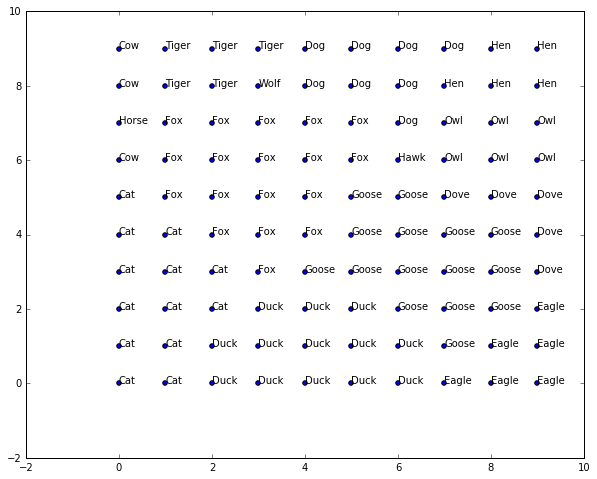
The training of the SOFMGrid occurred in two phases. The first phase was the self-organizing phase, and ran for 2000 epochs over each of the 16 animals. During each epoch, the order in which examples were presented to the network was shuffled. During this phase, the learning rate decayed according to an inverse exponential function defined by , where was .1 and was 2000. These values were selected both from suggestions in the original paper, as well as through experimentation. The sigma function was defined by , where was 5 and was 1243. These values were also selected by suggestions from the original paper. After the self-organizing phase, a fine tuning phase was run for 50,000 epochs. During this phase, was kept at 1, and was kept at 0.001. This allowed for the system to converge, and update only nearest neighbor neurons.

The final SOFM responses were checked in two ways. The first method was used to find the neurons that maximally activated for each neuron. To do this, each animal’s attribute vector was compared to the weight vector of every neuron. The neuron with maximal response was labeled with the animal’s name. This plot is shown below in figure 1. As seen in the plot, a few of the mammals with lots of overlap had the same maximally activating neuron. The second method was used to find what animal each neuron was closest to. To do this, every animal attribute vector was passed through the neuron. The neuron was then labeled with the animal that it was closest to. This plot is shown below in figure 2.

# Plots

**Figure 1: Maximal Neuron Activations For Animal**

**Figure 2: Closest Animal For Each Neuron**

****

# Analysis of Results

The results of the SOFM are similar to the one found in the paper. There are very distinct divisions between the medium sized mammals, large mammals, and birds. The maximally activating neurons in this SOFM are also fairly well separated. The main difference is that the size of the cow, horse, and zebra section is not as well pronounced. In fact, the zebra does not even show up in the figure 2 plot. This could be due to a number of factors. The main factor that I would consider is weight initialization. The weights in this implementation of the SOFM were set to a random value between -0.05 and 0.05. This artificially biases all neurons to activate more for animals that have many attributes close to 0. An alternative would have been to set the values between 0 and 1. This would prevent some of the biasing since the attributes are values distributed as 0’s and 1’s. Overall, the figures shown above confirm that the implementation of the SOFM works, and gives both topologically and conceptually similar to those found in the haykin paper.

Intelligent Systems HW4 Part 2

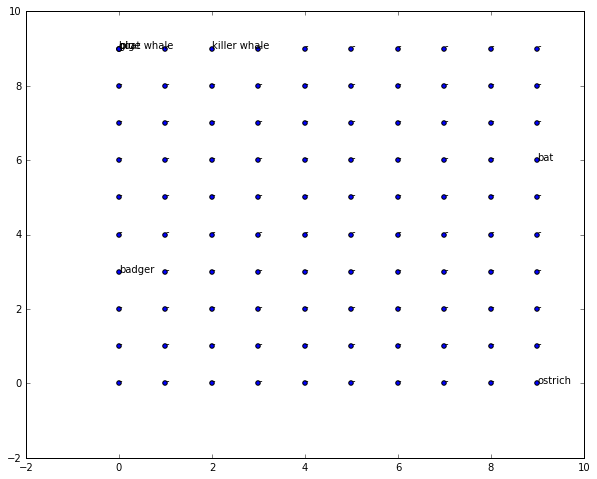
# Problem Description

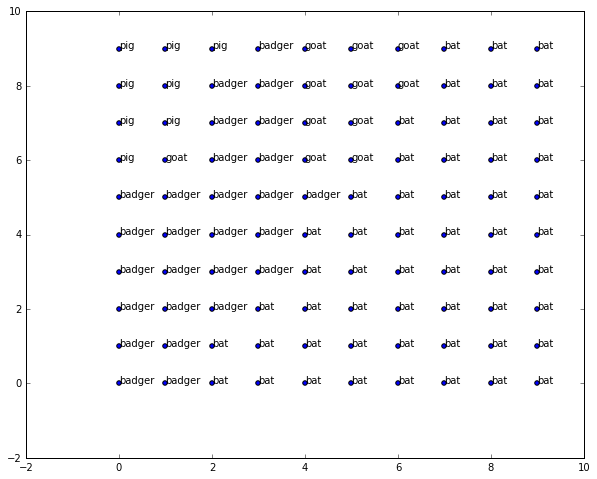
Using the SOFM map created in the previous section, pass attributes for new animals into the map, and determine which neurons they are most similar to. Comment on whether the results make sense, and what this says about the limitations of the system.

# Approach

The SOFM created in part 1 was saved in a pickled binary file, and was loaded from this file for analysis in this part. Therefore, there was no need to re-train the network. The code for this part is almost the exact same as for the previous part. Functions were already written for the SOFMGrid class to take in an arbitrary set of feature vectors and labels, and to return neuron coordinates and a class label for the maximally activating neurons. The goat, pig, badger, ostrich, blue whale, and killer whale system were all hard coded into the python scripts. The plots of the neuron activations are shown below.

# Plots

**Figure 1: Maximal Neuron Activations For Animal**

**Figure 2: Closest Animal For Each Neuron**

# Analysis of Results

The results of the SOFM make sense for a conceptual point of view. The ostrich, and bat are all firmly in regions where animals can fly, and the badger is in a region of a medium sized animal, such as a cat. The maximal neuron for most of the other animals is the neuron in the top left hand corner. While this makes sense for the whale and maybe pig, the goat should not be there. However, when we plot the closest animal for each neuron, we see that the neurons that activated most for dogs also activate a lot for the goat. This means that the neuron in the top left corner of the grid has weights of greater general magnitude than those in the dog/goat region. The killer whale activates more in the region of the tiger, which makes sense because killer whales are hunters like tigers.

These results show three major limitations of the system. First, if there were more neurons in the network, then perhaps we would have been able to capture finer regions of animals with the same number of iterations. Second, if weights aren’t normalized, then a neuron whose inherent weight magnitudes are large may cause other, less similar data elements to maximally activate for that neuron. Finally, the network is inherently limited by which attributes we choose to provide it with. The network only performs linear combinations of attributes. This assumption makes the model simpler, but also means that it is difficult for the model to come up with new attributes that may be better for grouping the data items. Overall, the results shown in the plots provide evidence that the trained SOFM can group data it hasn’t seen before in ways that are topologically and conceptually sensible.

# Code

## main.py

'''

Author: Daniel Griffin

Date: 4/15/2016

Description: This module is used to implement answers to the questions given in

the intelligent systems class at the university of cincinnati on self organizing

feature maps. It uses a "Library" that has all of the code related to building and

training the SOFM core.

'''

import numpy as np

import pandas as pd

import scipy as scipy

import pickle

from SOFM\_Lib import SOFM\_Core

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def getOneHotDict(nameSet, scale):

mDict = {}

currentIndex = 0

for name in nameSet:

oneHotArray = np.zeros((len(nameSet),))

oneHotArray[currentIndex] = 1\*scale

mDict[name] = oneHotArray

currentIndex += 1

return mDict

def main():

global mSOFM

print("Running Main.")

# Read data.

data = pd.read\_csv('./animals.csv')

# Make a 'one-hot' encoding of each animal class label.

oneHotDict = getOneHotDict(data['name'], 0.2)

oneHotLabelsList = []

for index, row in data.iterrows():

oneHotLabelsList.append(oneHotDict[row['name']])

# Parse data, scale it, and rebuild it so that the bit vectors are normalized.

dataToScale = data[data.columns[:-1]].values

animals = data[data.columns[-1]].values

i = 0

dataToScale = dataToScale.astype(np.float32)

for row in dataToScale[:]:

mean = row.mean()

dataToScale[i] = dataToScale[i] \* mean

i += 1

data = pd.DataFrame(dataToScale, columns=data.columns[:-1])

data['name'] = animals

# Create SOFm grid.

mSOFM = SOFM\_Core.SOFMGrid(input\_size=29 , row\_size=10, col\_size=10)

# Train grid.

dataValues = data[data.columns[:-1]].values

dataValues = np.hstack((dataValues, np.array(oneHotLabelsList)))

mSOFM.train(dataValues, organize\_epochs=2000, finetune\_epochs=50000)

# Create a data set for probing, where each animal label has attributes 0.

zeroAttrData = np.zeros((data[data.columns[:-1]].values.shape))

zeroAttrData = np.hstack((zeroAttrData, np.array(oneHotLabelsList)))

tupList = mSOFM.getMaxActivations(zeroAttrData, data['name'])

tups = list(map(lambda item: (item[0], item[1]), tupList))

for row in range(0, 10):

for col in range(0, 10):

if (row, col) not in tups:

tupList.append((row, col, '-'))

exWeights = mSOFM.neurons[0].weights

print("Example Weights: ")

print(exWeights)

xVals = list(map(lambda item: item[0], tupList))

yVals = list(map(lambda item: item[1], tupList))

fig = plt.figure()

ax = fig.add\_subplot(111)

fig.set\_size\_inches(10, 8, forward=True)

ax.scatter(xVals, yVals)

for tup in tupList:

ax.text(tup[0], tup[1], tup[2])

plt.show()

# Find the closest class for each neuron.

tupList = mSOFM.getAllNeuronActivations(zeroAttrData, classLabels=data['name'])

xVals = list(map(lambda item: item[0], tupList))

yVals = list(map(lambda item: item[1], tupList))

fig = plt.figure()

ax = fig.add\_subplot(111)

fig.set\_size\_inches(10, 8, forward=True)

ax.scatter(xVals, yVals)

for tup in tupList:

ax.text(tup[0], tup[1], tup[2])

plt.show()

tupList = mSOFM.getGridResponses(dataValues[0])

xVals = list(map(lambda item: item[0], tupList))

yVals = list(map(lambda item: item[1], tupList))

zVals = list(map(lambda item: item[2], tupList))

fig = plt.figure()

ax = fig.gca(projection='3d')

ax.plot\_trisurf(xVals, yVals, zVals)

plt.show()

# Save the SOFM in a file.

with open('./mSOFM.pkl', 'wb') as f:

pickle.dump(mSOFM, f)

def part2():

global mSOFM

labels = ['goat', 'pig', 'badger', 'ostrich', 'bat', 'blue whale', 'killer whale']

newAnimals = np.array([[0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],

[0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0],

[1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0],

[0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0],

[1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0],

[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],

[0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1]])

newAnimals = np.hstack((newAnimals, np.zeros((newAnimals.shape[0], 16) )))

# Find the closest class for each neuron.

tupList = mSOFM.getMaxActivations(newAnimals, classLabels=labels)

tups = list(map(lambda item: (item[0], item[1]), tupList))

for row in range(0, 10):

for col in range(0, 10):

if (row, col) not in tups:

tupList.append((row, col, '-'))

xVals = list(map(lambda item: item[0], tupList))

yVals = list(map(lambda item: item[1], tupList))

fig = plt.figure()

ax = fig.add\_subplot(111)

fig.set\_size\_inches(10, 8, forward=True)

ax.scatter(xVals, yVals)

for tup in tupList:

ax.text(tup[0], tup[1], tup[2])

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

#main()

part2()

## main.py

'''

Author: Daniel Griffin

Date: 4/15/2016

Description: This represents the core code for building and

training SOFM nets. The code was made into a module initially

so that it would be easier to incorporate it as an actual

library, should the need or desire ever arise.

'''

import math

import numpy as np

class SOFMGrid:

def \_\_init\_\_(self, input\_size=1, row\_size=1, col\_size=1):

# Eta, learning rate. Initially, set as constant.

self.eta0 = .1

self.eta\_tau = 2000

# Time constant for the decaying sigma.

self.sigma0 = 5

self.sigma\_tau = 1243

#self.sigma\_tau = 50

# SOFM Grid size.

self.row\_size = row\_size

self.col\_size = col\_size

self.neurons = []

self.input\_size = input\_size

# Make a list of neuorns, where the positions are set based on the grid and row size.

for row in range(0, row\_size):

for col in range(0, col\_size):

self.neurons.append(Neuron(row, col, input\_size=self.input\_size, weight\_lbound=-0.05, weight\_ubound=0.05))

'''

This function expects a 2 dimensional numpy array, with

each row representing a data example.

Training Steps:

1) Select a data point xj.

2) Find position of max neuron, i\* = argmin\_i(||xj - wi||)

3) Update the weights in the network over all i as dwij = n\*N(i,i\*,t)\*(xj - wij)

N(i, i\*, t) = exp(-1\*( ( -(||ri - ri\*||)^2 )/(2sigma(t)^2) )

-ri is the position of neuron i in the grid.

-ri\* is the position of the winning neuron i\* in the grid.

sigma(t) = sigmo\_0\*exp(-t/tau)

-Tau is a time constant (Typically around 1000ish).

'''

def train(self, train\_data, organize\_epochs=2000, finetune\_epochs=10000):

# Time variable for the decaying sigma.

time = 0

# Get a copy of the training data that can be shuffled.

mTrainData = train\_data.copy()

# Organizing phase.

for epoch in range(0, organize\_epochs):

print("Epoch: " + str(epoch))

# Shuffle the training data.

#np.random.shuffle(mTrainData)

# Loop over each datapoint.

for dataPoint in mTrainData[:]:

# Get the row and column for the closest neuron.

row, col, index = self.findMaxActivatingNeuron(dataPoint)

# Update all weights.

self.updateWeights(index, dataPoint, time)

#Increment the time parameter.

time += 1

# Fine tuning phase.

self.eta0 = 0.001

self.sigma0 = 1

time = 0

for epoch in range(0, finetune\_epochs):

print("Fine tune epoch: " + str(epoch))

# Shuffle the training data.

np.random.shuffle(mTrainData)

# Loop over each datapoint.

for dataPoint in mTrainData[:]:

# Get the row and column for the closest neuron.

row, col, index = self.findMaxActivatingNeuron(dataPoint)

# Update all weights.

self.updateWeights(index, dataPoint, time)

'''

Update the weights in the network using a radial distance function.

'''

def updateWeights(self, maxNeuronPos, x, t):

for neuronIndex in range(0, len(self.neurons)):

neuron = self.neurons[neuronIndex]

for i in range(0, neuron.weights.shape[0]):

# Calculate delta wi using the time decaying radial distance function.

dwi = (self.eta(t) \* self.radialDist(neuronIndex, maxNeuronPos, t) \* (x[i] - neuron.weights[i]))

# Update the weights.

neuron.weights[i] = neuron.weights[i] + dwi

'''

Calculate the radial distance function.

'''

def radialDist(self, i, i\_max, t):

# Form the positions into vectors since they are stored as individual values.

i\_pos = np.array([self.neurons[i].row, self.neurons[i].col])

i\_max\_pos = np.array([self.neurons[i\_max].row, self.neurons[i\_max].col])

# Calculate the time-shrinking, radial distance function.

radDist = math.exp( -1.0\*(math.pow(np.linalg.norm(i\_pos - i\_max\_pos),2)) / (2.0\*math.pow(self.sigma(t),2)) )

return radDist

'''

Calculate the time decaying sigma value.

'''

def sigma(self, t):

return float(self.sigma0) \* math.exp(float(-t)/self.sigma\_tau)

'''

Calculate the time decaying eta (learning rate) value.

'''

def eta(self, t):

return float(self.eta0)\*math.exp(float(-t)/self.eta\_tau)

'''

Find the neuron whose weight vector is closest to the input vector x.

'''

def findClosestNeuron(self, x):

maxPos = (0, 0)

minDist = 1000000000

maxIndex = 0

for i in range(0, len(self.neurons)):

dist = np.linalg.norm(self.neurons[i].weights - x)

if dist < minDist:

minDist = dist

maxPos = (self.neurons[i].row, self.neurons[i].col)

maxIndex = i

return maxPos[0], maxPos[1], maxIndex

def findMaxActivatingNeuron(self, x):

maxIndex = 0

maxOutput = 0

pos = (0, 0)

for i in range(0, len(self.neurons)):

output = self.neurons[i].output(x)

if output > maxOutput:

maxOutput = output

maxIndex = i

pos = (self.neurons[i].row, self.neurons[i].col)

return pos[0], pos[1], maxIndex

'''

'''

def getGridResponses(self, x):

responseTupleList = []

for neuron in self.neurons:

responseTupleList.append((neuron.row, neuron.col, neuron.output(x)))

return responseTupleList

'''

Returns a list of tuples, where each tuple is a neuron with:

(<row>, <col>, <class>).

'''

def getMaxActivations(self, data, classLabels):

# Find Max Activations and label them.

tupList = []

for index in range(0, data.shape[0]):

row, col, i = self.findMaxActivatingNeuron(data[index])

tupList.append((row, col, classLabels[index]))

return tupList

'''

Get the grid of all neurons, and which animal class they are closest to.

'''

def getAllNeuronActivations(self, data, classLabels):

# Find data point for which each neuron is closest to.

tupList = []

for neuron in self.neurons:

maxOutput = 0

closestLabel = None

# Loop through all animal points, and find the closest one.

for index in range(0, data.shape[0]):

output = neuron.output(data[index])

if output > maxOutput:

closestLabel = classLabels[index]

maxOutput = output

# Append the closest animal label tuple to the tupList.

tupList.append((neuron.row, neuron.col, closestLabel))

return tupList

'''

Return x, y of time response of the sigma function over time.

'''

def plotSigmaOverTime(self, time\_steps=1000):

x = np.arange(0, time\_steps, 1)

y = np.zeros(time\_steps)

for step in range(0, time\_steps):

y[step] = self.sigma0 \* math.exp(-float(step)/self.sigma\_tau)

return x, y

'''

Return x, y of time response of the eta function over time.

'''

def plotEtaOverTime(self, time\_steps=1000):

x = np.arange(0, time\_steps, 1)

y = np.zeros(time\_steps)

for step in range(0, time\_steps):

y[step] = self.eta0 \* math.exp(-float(step)/self.eta\_tau)

return x, y

'''

SOFM neuron abstraction.

'''

class Neuron:

def \_\_init\_\_(self, row, col, input\_size=1, weight\_lbound = -0.05, weight\_ubound = 0.05):

self.weights = np.random.uniform(weight\_lbound, weight\_ubound, input\_size)

self.row = row

self.col = col

'''

Return the distance, or magnitude between a point x and this neuron's weight vector.

'''

def dist(self, x):

return scipy\_dist.euclidean(x, self.weights)

'''

Return the dot product of the neuron weights and the input data.

'''

def output(self, x):

return np.dot(self.weights, x)