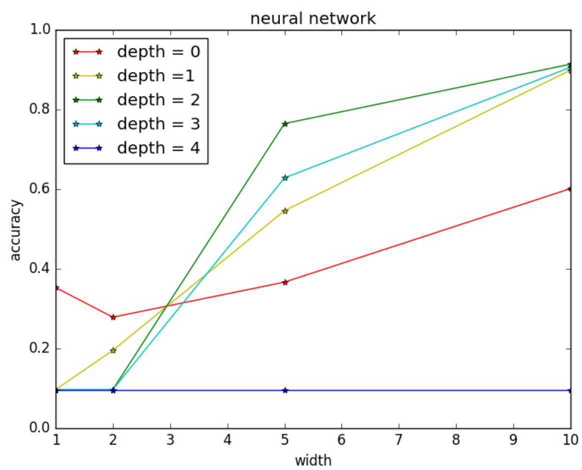
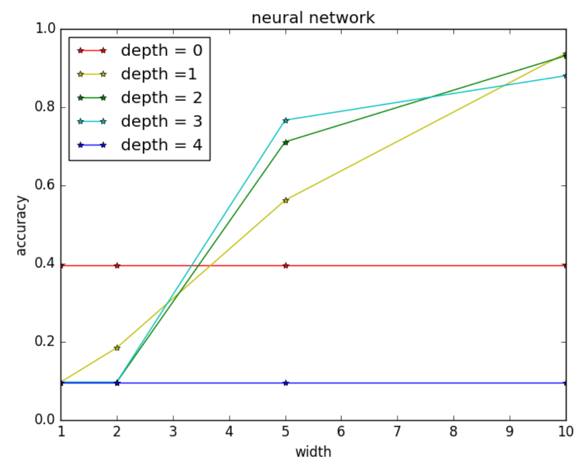
COMP135 pp3 Report

Beibei du





Q: How does the performance vary when we increase d and w?

Answer:

- Fixed d

  When d = 1 or 2 or 4, if increase w, we can see accuracy will also increase;

  When d = 4, accuracy will keep at 10% for all w;

  When d = 0, run four times, since weight initialization will change, accuracy will also not same for these four times according to figure left; Figure right only run d = 0 one time.

- Fixed w

  When w = 1, except d = 0, all other d has same accuracy;

  When w = 2, depth 0 > depth1 > depth 2 = depth3 = depth4 for accuracy;

  When w = 10, depth 2 has higher accuracy than others;

- Conclusion

  Fixed d, if increase w, accuracy will also increase except d = 4 and d = 0, which means increasing width will improve performance for reasonable d.

  Fixed w, if increase d, accuracy will not increase monotonously, d = 2 has highest accuracy when w = 10; d = 4 has lowest accuracy for all w; we can get the conclusion that increasing depth doesn't help for performance, this maybe because increasing the depth of a neural network will let the approximate functions with increased non-linearity, increasing the chance of over-fitting.

```
# Code
import numpy as np
import sys
import math
from operator import itemgetter
import random
import matplotlib.pyplot as plt

eta = 0.1

# Reading Data Files in arff format, return a 2D matrix which store the data
def read_file(filename):
	file_data = []
	with open(filename, 'r') as f:
		for line in f:
			if line[0].isdigit():
				features = []
				for data in line.split(','):
					try:
						features.append(int(data))
					except:
						features.append(data)
				file_data.append(features)
	return file_data

# figure number of labels and output units to be used
def split_data(train_data):
	d_label = dict()
	### not finished
	for tr in train_data:
		label = tr[-1]
		if label in d_label.keys():
			d_label[label] += 1
		else:
			d_label[label] = 1
	n = len(d_label.keys())
	y = np.zeros((n, n))
	for j in range(n):
		y[j][j] = 1
	return len(d_label), y
```

```
def initial_weights(w, d, feature):
    weights = []
    d += 1
    for k in range(d):
        if d == 1:
            weight = np.array([random.uniform(-0.1,0.1) for i in
range(feature*10)])
            weight = weight.reshape(10, feature)
            weights.append(weight)
        else:
            if k == 0:
                weight = np.array([random.uniform(-0.1,0.1) for i in
range(feature*w)])
                weight = weight.reshape(w, feature)
                weights.append(weight)
            elif k == (d-1):
                # number of output layer is 10
                weight = np.array([random.uniform(-0.1,0.1) for i in
range(w * 10)])
                weight = weight.reshape(10,w)
                weights.append(weight)
            else:
                weight = np.array([random.uniform(-0.1,0.1) for i in
range(w * w)])
                weight = weight.reshape(w,w)
                weights.append(weight)
    return weights

# compute X using sigmoid function
def sigmoid(s):
    res = []
    for a in s:
        if a > 50:
            res.append(1 - 10**(-50))
        elif a < -50:
            res.append(10**(-50))
        else:
            res.append(1 / (1 + np.exp(-a)))

    res = np.array(res)
    return res

# compute x
def compute_x(weights, d, tr):
```

```
        X = []
        X.append(tr[:-1])
        # forwards compute X
        for di in range(d):
                s_hidden = np.dot(weights[di], X[-1])
                x_hidden = sigmoid(s_hidden)
                X.append(x_hidden)
        return X

# compute DELTA
def compute_delta(L, X, weights):
        depth = len(weights)
        d = depth
        DELTA = []
        while d > 0 :
                if d == depth:
                        x = X[d]
                        n = len(x)
                        delta = -(L - x) * x * (np.ones(n)-x)
                        DELTA.append(delta)
                        d -= 1
                else:
                        last = DELTA[-1]
                        x = X[d]
                        n = len(x)
                        weight = weights[d]
                        delta = x * (np.ones(n) - x) * np.dot(np.transpose(weight), last)
                        DELTA.append(delta)
                        d -= 1
        return DELTA

# backpropagation algorithm
def learn(w, d, train_data, test_data, y):
        global eta
        # construct network with w, d and initialize weights
        feature = len(train_data[0]) - 1
        weights = initial_weights(w, d, feature)
        # Repeat 200 times
        d += 1
        for i in range(200):
                for tr in train_data:
                        X = compute_x(weights, d, tr)
                        # backwards compute DELTA
                        index = tr[-1]
```

```python
                L = y[:,index] # L is vector
                DELTA = compute_delta(L, X, weights)

                # update weights
                for di in range(d):
                    x = X[di]
                    delta = np.matrix(DELTA[d-di-1]).T
                    g = eta * delta * x
                    g = np.array(g)
                    weights[di] -= g

        # test data
        accu = 0
        for te in test_data:
            X = compute_x(weights, d, te)
            y = X[-1]
            if np.argmax(y) == te[-1]:
                accu += 1
        te_len = len(test_data)
        accuracy = float(accu) / te_len
        return accuracy


    def main():

        file = ['optdigits_train.arff.txt', 'optdigits_test.arff.txt']
        train_data = read_file(file[0])
        test_data = read_file(file[1])
        depth = [1,2,3,4]
        width = [1,2,5,10]

        # number of lables(d) and output units (y)
        d_label, y = split_data(train_data)
        accu_list = []
        accuracy = learn(0, 0, train_data,test_data,y)
        acc = []
        acc.append(accuracy)
        accu_list.append(acc * 4)

        for d in depth:
            acc = []
            for w in width:
                accuracy = learn(w, d, train_data, test_data, y)
                acc.append(accuracy)
```

```
            accu_list.append(acc)

        print len(accu_list)
        print accu_list

        # plot figure
        print "start to plot:"
        plt.plot(width, accu_list[0], 'r', marker = '*')
        plt.plot(width, accu_list[1], 'y', marker = '*')
        plt.plot(width, accu_list[2], 'g', marker = '*')
        plt.plot(width, accu_list[3], 'c', marker = '*')
        plt.plot(width, accu_list[4], 'b', marker = '*')
        plt.title('neural network')
        plt.xlabel('width')
        plt.ylabel('accuracy')
        plt.legend(['depth = 0', 'depth = 1', 'depth = 2', 'depth = 3', 'depth = 4'], loc = 0)
        plt.savefig("neural.png")
        plt.clf()

main()
```