Lab 2: Applied data science

1) ***Write and, or, not, and nor Boolean operators similar to the XOR operator demonstrated in class***

Part a) Make use of the same code as provided in class and simply change the sample data as given. Does the neural network always get the correct answer? Write a short paragraph explaining how you might improve this.

Without changing any parameters, the iteration result is as follows:

```
the prediction for AND value:
1 AND 1 =  [ 1.]
1 AND 0 =  [  3.33066907e-16]
0 AND 1 =  [  7.77156117e-16]
0 AND 0 =  [ -4.44089210e-16]

the prediction for OR value:
1 OR 1 =  [ 1.01198762]
1 OR 0 =  [ 0.99502403]
0 OR 1 =  [ 0.99447853]
0 OR 0 =  [ 0.00108808]

the prediction for NOT value:
NOT 0  =  [ 1.]
NOT 1  =  [ -8.39536773e-14]

the prediction for NOR value:
1 NOR 1 =  [  6.66133815e-16]
1 NOR 0 =  [ -8.88178420e-16]
```

```
0 NOR 1 =  [ -2.22044605e-16]
0 NOR 0 =  [ 1.]
```

It relatively provide good predictor except for OR function which still show error rate at about 10^{-2}, which is not too good.

Part b) Tune the neural network by playing with the number of hidden layers and the number of hidden nodes in each layer. What improved the effectiveness of your neural network. Provided examples of parameter sets that worked well and parameter sets that did not. These should be python files. Also you should include a write up, with the code merely as a reference for the grader.

On my first attempt, I tried to change the number node since I wanted to focus to improve the OR predictor performance from the previous attempt:

```
net = buildNetwork(2, 10, 1, bias=True)
trainer = BackpropTrainer          learningrate = 0.01, momentum = 0.99)
```

Results:

```
the prediction for AND value:
1 AND 1 =  [ 0.99121504]
1 AND 0 =  [ 0.00750315]
0 AND 1 =  [ 0.00755936]
0 AND 0 =  [-0.03360367]

the prediction for OR value:
1 OR 1 =  [ 1.]
1 OR 0 =  [ 1.]
0 OR 1 =  [ 1.]
0 OR 0 =  [ 0.]
```

```
the prediction for NOT value:
NOT 0  =  [ 1.]
NOT 1  =  [  5.15143483e-14]

the prediction for NOR value:
1 NOR 1 =  [ -5.55111512e-17]
1 NOR 0 =  [ 0.]
0 NOR 1 =  [  5.55111512e-17]
0 NOR 0 =  [ 1.]
```

However, while this generally improve NOT and NOR and significantly improved OR, we could observe that it has reduced the prediction accuracy for AND by the order of 10^{-2}.

Next, I also tried adding the number of nodes to bigger value to 100 :

```
net = buildNetwork(2, 100, 1, bias=True)
trainer = BackpropTrainer        learningrate = 0.01, momentum = 0.99)
```

```
the prediction for AND value:
1 AND 1 =  [ 1.]
1 AND 0 =  [ -5.55111512e-17]
0 AND 1 =  [  5.55111512e-17]
0 AND 0 =  [  5.55111512e-17]

the prediction for OR value:
1 OR 1 =  [ 1.01926925]
1 OR 0 =  [ 0.98782671]
0 OR 1 =  [ 0.99052739]
0 OR 0 =  [ -9.77804790e-05]

the prediction for NOT value:
NOT 0  =  [ 1.]
NOT 1  =  [  1.37667655e-13]

the prediction for NOR value:
```

```
1 NOR 1 =  [-0.0242258]
1 NOR 0 =  [ 0.0149737]
0 NOR 1 =  [ 0.01351082]
0 NOR 0 =  [ 1.00019437]
```

However, the result does **not** improve the result as there were significant decrease in predicition accuracy for OR and NOR.

Now, we could also observe the effect of adding the number of hidden layers by maintaining the number of nodes to be 100:

```
net = buildNetwork(2, 100, 100, 1, bias=True)
trainer = BackpropTrainer(net[illegible]arningrate = 0.01, momentum = 0.99)
```

However, this does **not** improve the result of OR and NOR from a single hidden layer with 100 nodes tested above:

```
the prediction for AND value:
1 AND 1 =  [ 1.]
1 AND 0 =  [  1.77635684e-15]
0 AND 1 =  [  4.44089210e-16]
0 AND 0 =  [ -1.77635684e-15]

the prediction for OR value:
1 OR 1 =  [ 1.01082734]
1 OR 0 =  [ 0.9918763]
0 OR 1 =  [ 0.99943891]
0 OR 0 =  [ 0.00139742]

the prediction for NOT value:
NOT 0  =  [ 1.]
NOT 1  =  [ -7.37188088e-14]
```

```
the prediction for NOR value:
1 NOR 1 =  [-0.01038958]
1 NOR 0 =  [ 0.00583228]
0 NOR 1 =  [ 0.00518549]
0 NOR 0 =  [ 0.99996256]
```

I also tried some other parameter tweak attempt by changing the type of layer, for instance changing hidden layer to the type *thanh*, however, it **fails** entirely for AND operation:

```
net = buildNetwork(2, 4, 1,hiddenclass=TanhLayer,  bias=True)
trainer = BackpropTrainer(net, learningrate = 0.01, momentum = 0.99)
```

```
the prediction for AND value:
1 AND 1 =  [ 0.39923562]
1 AND 0 =  [ 0.39923694]
0 AND 1 =  [ 0.3992522]
0 AND 0 =  [ 0.39948195]

the prediction for OR value:
1 OR 1 =  [ 1.]
1 OR 0 =  [ 1.]
0 OR 1 =  [ 1.]
0 OR 0 =  [ -2.22044605e-16]

the prediction for NOT value:
NOT 0  =  [ 1.]
NOT 1  =  [ -7.64943664e-14]

the prediction for NOR value:
1 NOR 1 =  [-0.0113836]
1 NOR 0 =  [ 0.0018139]
```

```
0 NOR 1 =  [ 0.00076514]

0 NOR 0 =  [ 0.99760806]
```

In conclusion, the parameter tuning that provides the easiest to understand effect is to adjust number of hidden nodes by maintaining number of layer and type of layer remain the same.

Part c) Write boolean functions for composite boolean operators:

With the concept learned in 1a and 1b, the figure below shows the results compared to the theoretical calculations:

1) (A AND B) OR C :

Input	Theoritical value	Test result
0 0 0	0	-0.00210791
0 0 1	1	0.99684276
0 1 0	0	-0.00167214
0 1 1	1	0.99328039
1 0 0	0	-0.00144703
1 0 1	1	0.99246093
1 1 0	1	0.98752535
1 1 1	1	1.0216004

2)(NOT (A OR B)) AND C

Input	Theoritical value	Test result
0 0 0	0	2.22E-16
0 0 1	1	1
0 1 0	0	2.22E-16
0 1 1	0	-6.66E-16
1 0 0	0	0
1 0 1	0	2.22E-16
1 1 0	0	-4.44E-16
1 1 1	1	1

3) NOT ((A OR B) AND C)

Input	Theoritical value	Test result
0 0 0	1	1.00573696
0 0 1	1	0.99622717
0 1 0	1	0.98631449
0 1 1	0	0.00112076
1 0 0	1	0.98639846
1 0 1	0	0.00428197
1 1 0	1	1.00668135
1 1 1	0	-0.01270967

It shows accurate prediction of the theoretical values with error rates at around 1-5 x 10^{-2}**.**

2) ***Premise: Teach py brains to recognize your face. Make use of https://github.com/EricSchles/neuralnet/blob/master/facial_recognition.py Take 10 pictures of your own face. Then find 10 other faces, that are not your own and are human. Make use of the code presented in class for doing facial recognition. You will need to process the images first. You can make use of cleaner.py to do this. However you will have to figure out how to make use of the different functions available to you.***

Sample data used in this number is as follows:

A> Self picture

drp1.png	Drp2.png	Drp3.png	Drp4.png
Drp5.png	Drp6.png	Drp7.png	Drp8.png

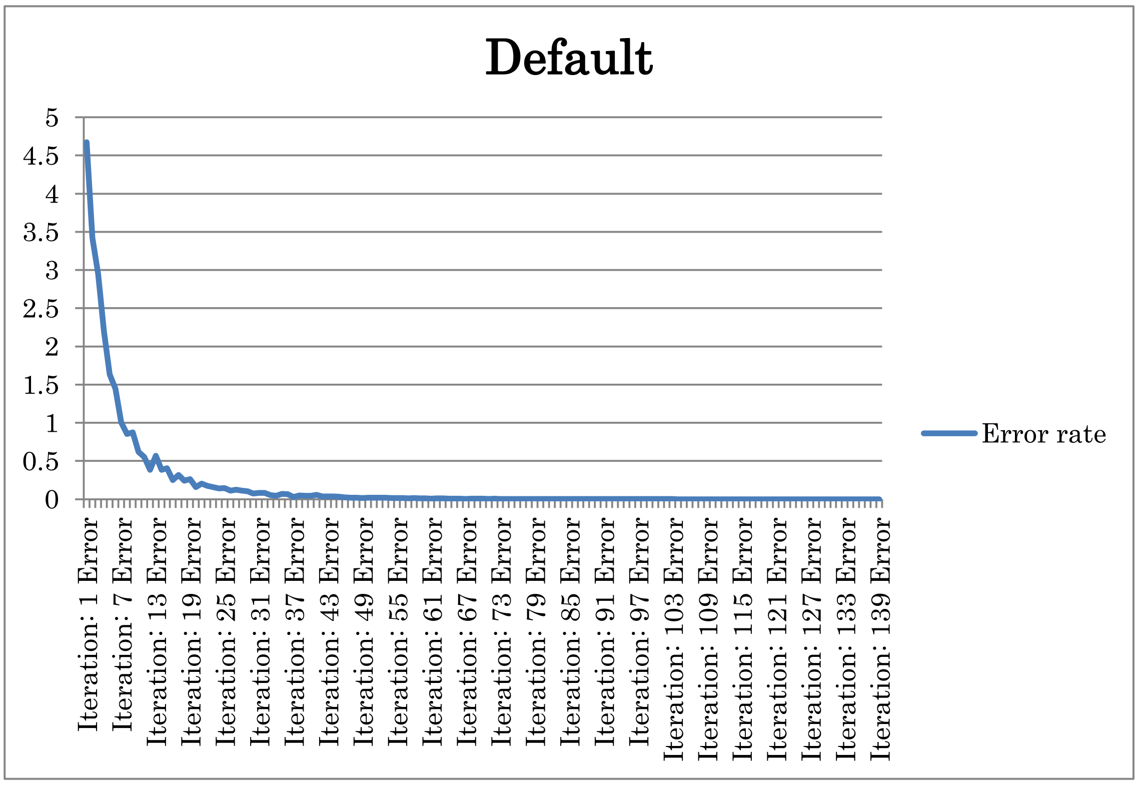
B> Other people picture

other1.png	other2.png	other3.png	other4.png
other5.png	other6.png	other7.png	other8.png

These pictures then resized to 40x40 using cleaner.py and name.sh (attached)

Part a) Did the tuning I provided in class work for your face? Why do you think this worked or didn't work? If you were not able to successfully use my tuning what did you have to change to get the neural network to work with your face?

By using tuning standard that was covered in class, the result was as follows:



In details:

```
                    :
Iteration: 132 Error 0.000157001164145
Iteration: 133 Error 0.000111149350062
Iteration: 134 Error 0.000105412812902
Iteration: 135 Error 0.000124883074953
Iteration: 136 Error 0.000117474717495
Iteration: 137 Error 0.000121240459356
Iteration: 138 Error 0.00012846163411
Iteration: 139 Error 7.69712585017e-05
```

Theoritical value: [2]

Result: [1.27042941]

Theoritical value: [1]

Result: [-0.2942844]

Theoritical value: [2]

Result: [0.77439851]

Theoritical value: [1]

Result: [-2.89474243]

It can be observed from the iteration process that the training process experienced a lot of iteration process and during the first phase encountered difficulties in reducing the number of errors. In addition, regarding the test data, although it could consistently get low score for self-pictures, the results deviated from the theoretical value (2 and 1) hence difficult to observe.

Therefore, I changed some parameters as follows:

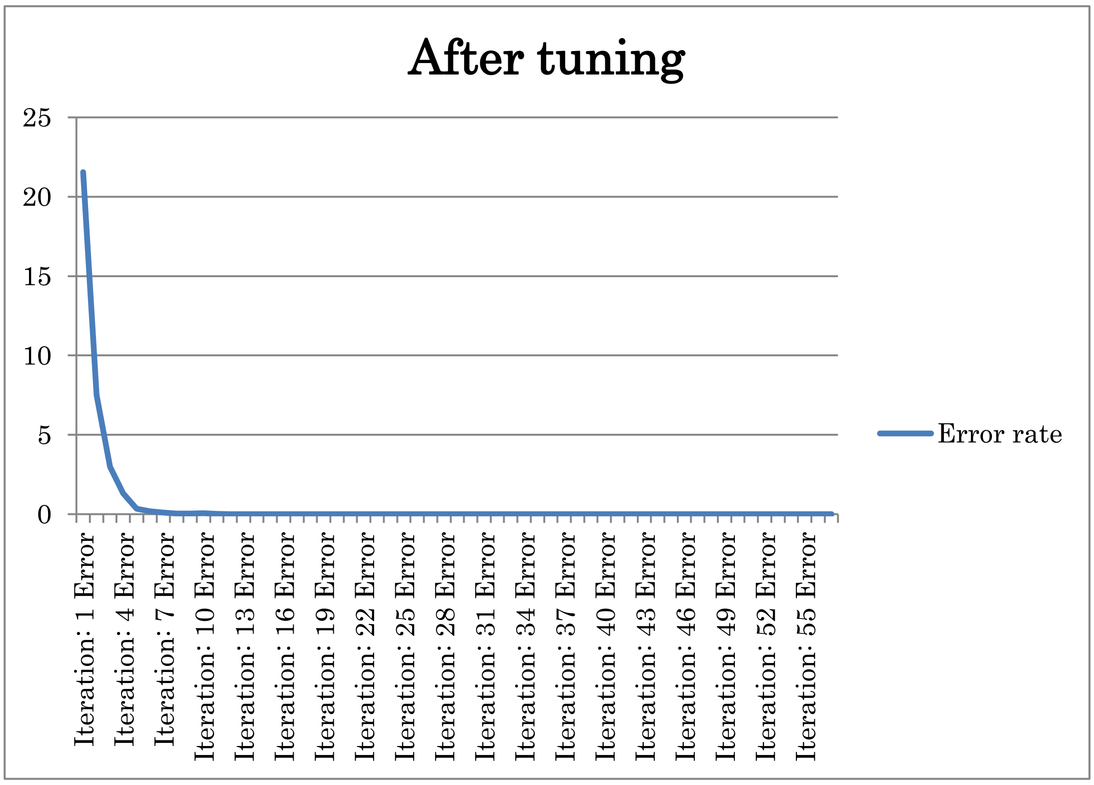
1. Added hidden layers:

```
net = buildNetwork(len(t), .03*len(t),.03*len(t),.03*len(t), 1
```

2. Changed the range of predictor boundaries:

```
ds = SupervisedDataSet(len(t), 1)
ds.addSample(loadImage('pic/drp1.png'),(1,))
ds.addSample(loadImage('pic/drp2.png'),(1,))
ds.addSample(loadImage('pic/drp3.png'),(1,))
ds.addSample(loadImage('pic/drp4.png'),(1,))
ds.addSample(loadImage('pic/drp5.png'),(1,))
ds.addSample(loadImage('pic/drp6.png'),(1,))
ds.addSample(loadImage('pic/drp7.png'),(1,))
ds.addSample(loadImage('pic/drp8.png'),(1,))
ds.addSample(loadImage('pic/other1.png'),(15,))
ds.addSample(loadImage('pic/other2.png'),(15,))
ds.addSample(loadImage('pic/other3.png'),(15,))
ds.addSample(loadImage('pic/other4.png'),(15,))
ds.addSample(loadImage('pic/other5.png'),(15,))
ds.addSample(loadImage('pic/other6.png'),(15,))
ds.addSample(loadImage('pic/other7.png'),(15,))
ds.addSample(loadImage('pic/other8.png'),(15,))
```

This setting resulted the following:



:

Iteration: 48 Error 0.000147696352253

```
Iteration: 49 Error 0.000137449677568
Iteration: 50 Error 0.000156478862141
Iteration: 51 Error 0.000150692125433
Iteration: 52 Error 0.000158173416096
Iteration: 53 Error 0.000151655173991
Iteration: 54 Error 0.00013917356801
Iteration: 55 Error 0.000155068469612
Iteration: 56 Error 0.000109737860283
Iteration: 57 Error 9.90877038239e-05
```

Theoritical value: [15]

Result: [17.22002369]

Theoritical value: [1]

Result: [8.77882067]

Theoritical value: [15]

Result: [17.38340649]

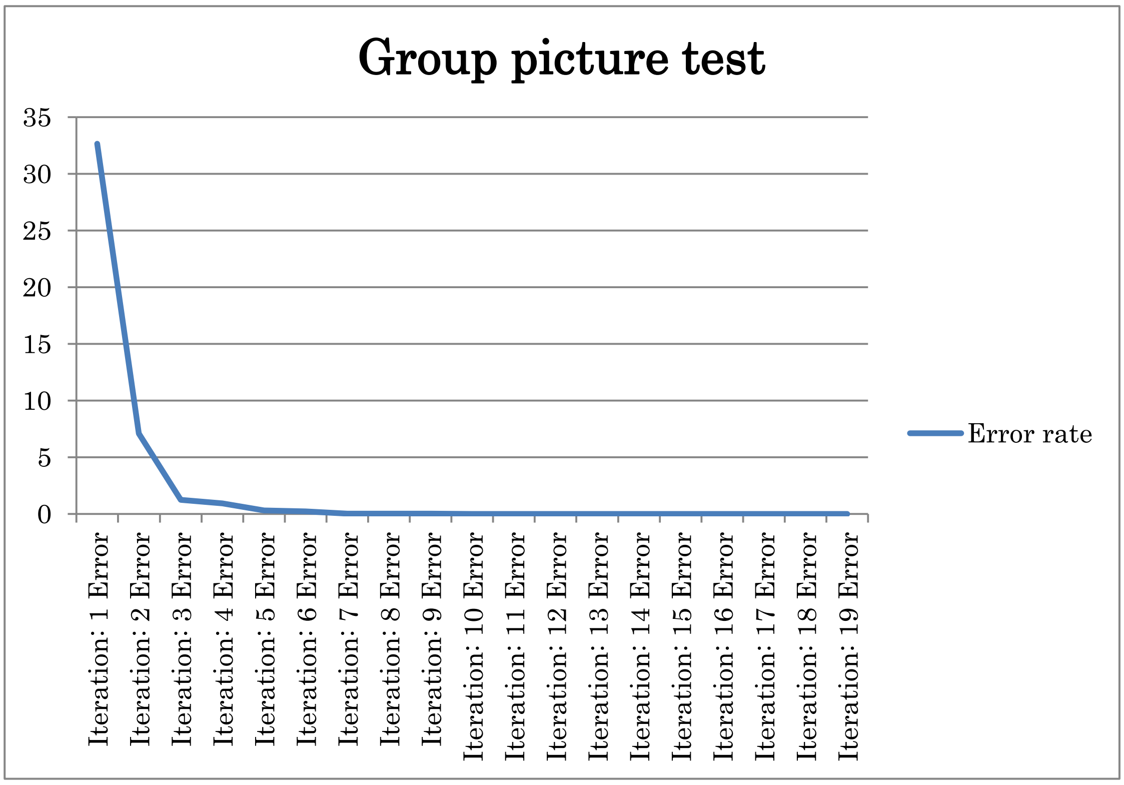
Theoritical value: [1]

Result: [7.29167152]

It can be observed that the accuracy of the facial recognition model improved compared to the last method. Another important point is that in this particular estimation, adding number of hidden layers did improve the performance.

Part b) Now that you have tuned the code to work with your face, try the code when all of the pictures have your face, some of them should be the ten original faces and the other ten should be group photos. Note there should be at least five other faces in all group photos you use. Did the tuning you used in part (a) still work? What did you have to change to get the neural network to recognize your face in all of the pictures. Was it able to recognize your face EVEN with multiple faces?

The iteration process is as follows:



:

Iteration: 15 Error 0.000323038824108

Iteration: 16 Error 0.000380796649312

Iteration: 17 Error 0.000204303751397

Iteration: 18 Error 0.000141164238314

Iteration: 19 Error 5.29297825704e-05

Theoritical value: [15]

Result: [11.98200559]

Theoritical value: [15]

Result: [15.09102171]

Theoritical value: [1]

Result: [9.37929336]

Theoritical value: [1]

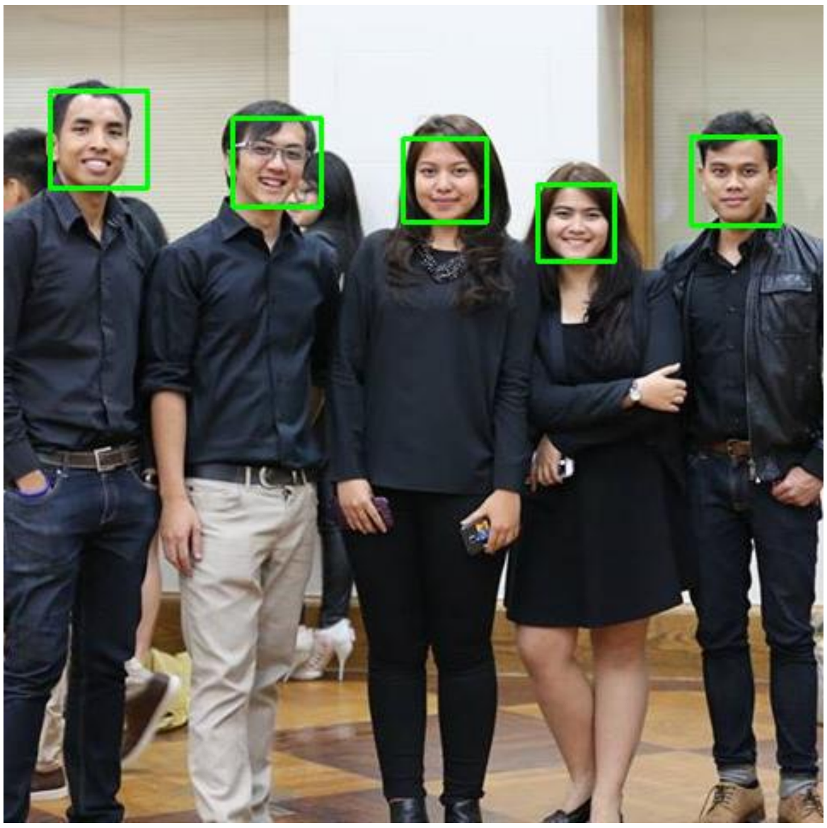
Result: [9.09625699]

It can be observed above that while it is still cannot perfectly detect the group picture as the same value as self-pictures, compared to the previous comparison with other people pictures, it shows narrower gap in the result value.

Part c) - optional extra credit

For this question you'll need to use faceFind to crop all the images in the group photos down to separate single faces (ideally in their own files). Then you'll have the neural network use the 10 original faces of you to predict which face the group picture is yours.)

In this number, face_detect.py was modified to achieve detecting the faces in the photo group below to get the all faces and crop them into separate files:



Breakdown into:

Group1_0.png Group1_1.png Group1_2.png Group1_3.png Group1_4.png

The neural network iteration shows that the last picture, Group1_4.png, get the closest to zero, meaning it can detect that last picture is me.

```
Result:    [ 5.37861873]
Result:    [ 1.32018669]
Result:    [ 1.85514706]
Result:    [ 4.18817045]
Result:    [ 0.99099169]
```

Modified applied to the face_detect.py :

```
imagePath = sys.argv[1]
cascPath = sys.argv[2]

def face_detect(scale_factor,min_neighbors,minSize_x,minSize_y):
# Get user supplied values
    global imagePath
    global cascPath

# Create the haar cascade
    faceCascade = cv2.CascadeClassifier(cascPath)

# Read the image
    image = cv2.imread(imagePath)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect faces in the image
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=scale_factor,
        minNeighbors=min_neighbors,
        minSize=(minSize_x, minSize_y),
        flags = cv2.cv.CV_HAAR_SCALE_IMAGE
        )

    return len(faces),faces,image
```

```
def tuning():
    #trying the default values
    num_faces,faces,image = face_detect(1.1,5,30,30)

    scale_factors = [1.1,1.2,1.3]
    collected_faces = []
    for min_neighbor in xrange(1,2):
        for minSize in xrange(10,30,10):
            for scale_factor in scale_factors:
                num_faces,faces,image =
face_detect(scale_factor,min_neighbor,minSize,minSize)
                if num_faces > 0:
                    parameters = {}
                    parameters['scale_factor'] = scale_factor
                    parameters['min_neighbors'] = min_neighbor
                    parameters['minSize'] = minSize

collected_faces.append([num_faces,faces,image,parameters])

    possible_faces = []
    for collected in collected_faces:
        if len(possible_faces) == 5: break
        for (x, y, w, h) in collected[1]:
            coordinates = [x,y,x+w,y+h]
            possible_faces.append(coordinates)
            if len(possible_faces) == 5: break

    print "len of possible face: %d" % len(possible_faces)
    return possible_faces,image

def face_find(potential_faces,epsilon,tolerance_level):
    matches = [0 for x in range(len(potential_faces))]
    for ind,candidate in enumerate(potential_faces):
        for coordinate_set in potential_faces:
            if coordinate_set[0] - epsilon < candidate[0] <
```

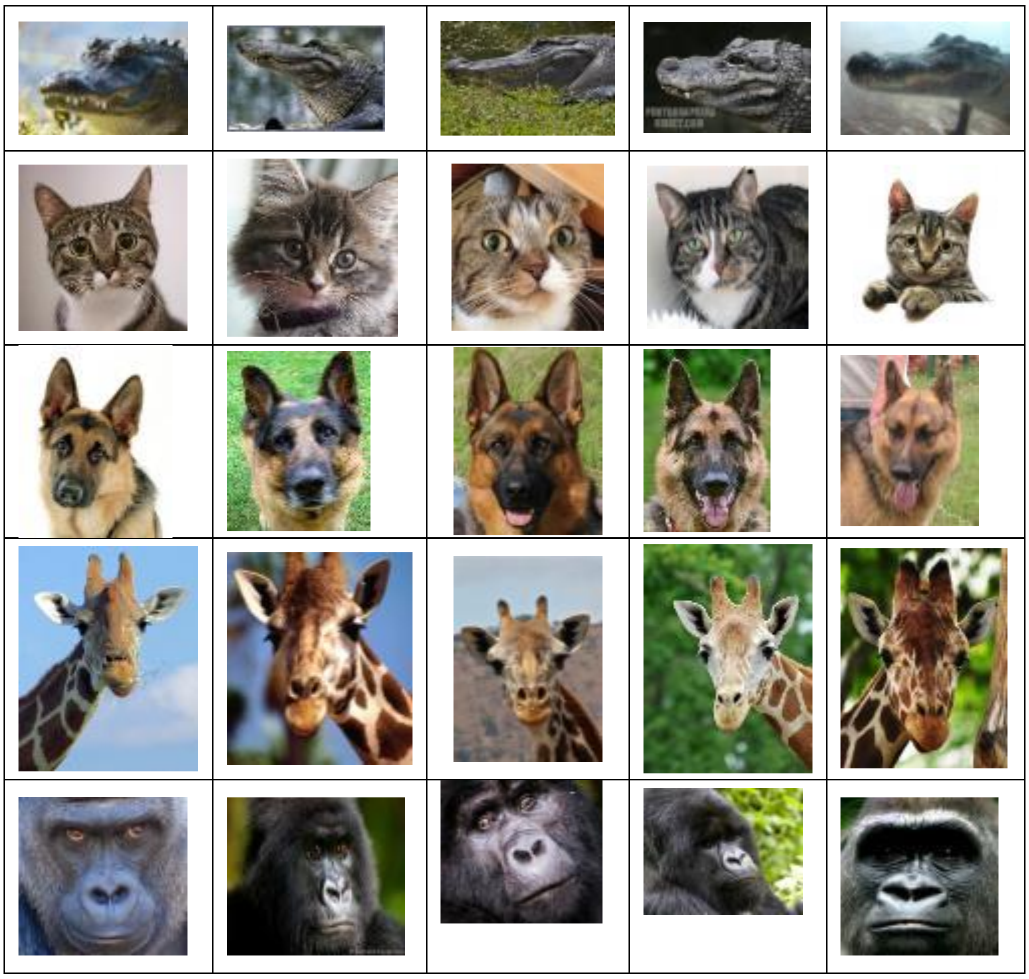
```
coordinate_set[0] + epsilon:
                if coordinate_set[1] - epsilon < candidate[1] <
coordinate_set[1] + epsilon:
                    matches[ind] += 1
    faces = []
    for ind,match in enumerate(matches):
        if match/float(len(matches)) > tolerance_level:
            faces.append(potential_faces[ind])
    return faces

potential_faces,image = tuning()
faces = face_find(potential_faces,5,0.01)

counter=0
for face in faces:
    cv2.rectangle(image, (face[0],face[1]),(face[2],face[3]),
(0,255,0),2)
    cv2.imshow("Faces found", image)
    cv2.waitKey(3000)
    cv2.imwrite("faces_found.png", image)
    crop=image[face[1]:face[3], face[0]:face[2]]
    file_name = imagePath.replace('.png','_')
    file_name = file_name + str(counter) + '.png'
    file_name = file_name.replace('\n','')
    print 'Writing ' + file_name
    cv2.imwrite(str(file_name), crop)
    counter+=1
```

3) ***Find faces of giraffe's, gorillas, dogs, cats, and alligators. Use py brains to classify each of these pictures. Your code should be able to consistently tell which animal is which.***

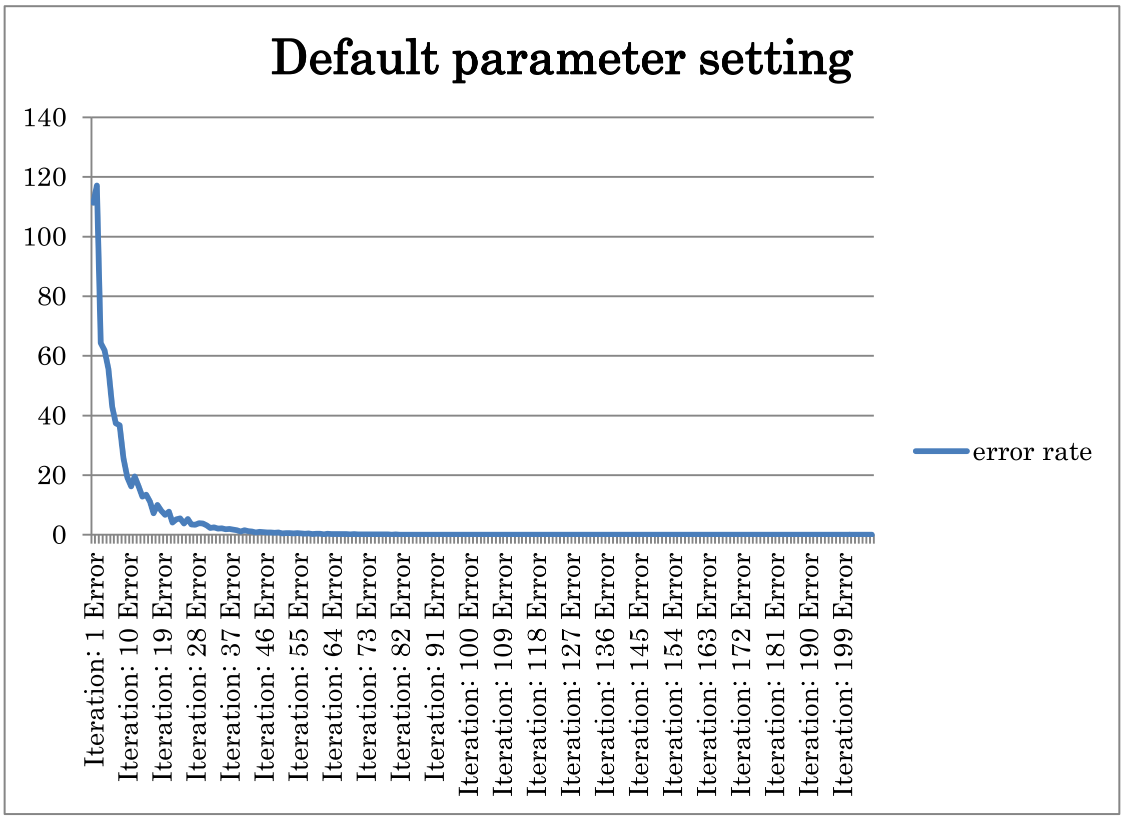
Sample data used in this number is as follows:



The pictures were then renamed animal1-5.jpg, converted to png files and then resized equally to 40x40.

Part a) Initially just use the code from class with the settings I provided. How well does this do? Are you able to identify each of the animals successfully? Your code should print out the following output:

While testing it using the standard parameter given in class, the result is as following:



===RESULT====

Picture pic/giraffe6.png has a dog

Picture pic/dog6.png has a alligator

Picture pic/gorilla6.png has a

gorilla

Picture pic/cat6.png has a gorilla

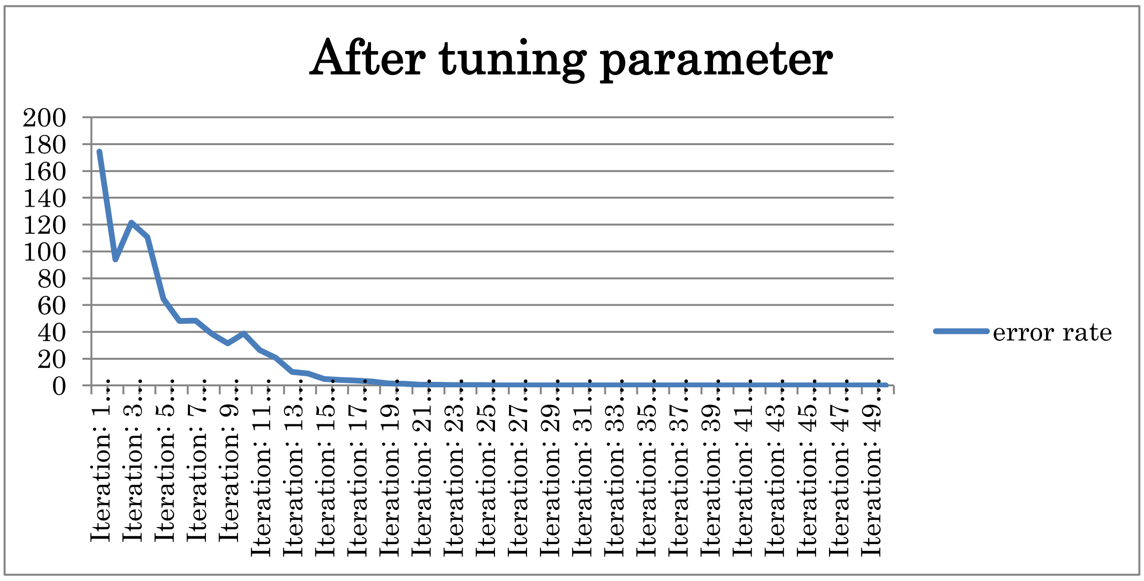
Picture pic/alligator6.png has a dog

Part b) Now tune your code so that every animal is guessed correctly, or as many as possible. (Note that a best effort is expected here, if your code does not do well here you will lose points).

By applying the same approach in number 2, the hidden layers were added and the range were changed:

```
net = buildNetwork(len(t), .03*len(t),.03*len(t),1)
```

```
ds = SupervisedDataSet(len(t), 1)
ds.addSample(loadImage('pic/giraffe1.png'),(0,))
ds.addSample(loadImage('pic/giraffe2.png'),(0,))
ds.addSample(loadImage('pic/giraffe3.png'),(0,))
ds.addSample(loadImage('pic/giraffe4.png'),(0,))
ds.addSample(loadImage('pic/giraffe5.png'),(0,))
ds.addSample(loadImage('pic/dog1.png'),(10,))
ds.addSample(loadImage('pic/dog2.png'),(10,))
ds.addSample(loadImage('pic/dog3.png'),(10,))
ds.addSample(loadImage('pic/dog4.png'),(10,))
ds.addSample(loadImage('pic/dog5.png'),(10,))
ds.addSample(loadImage('pic/gorilla1.png'),(20,))
ds.addSample(loadImage('pic/gorilla2.png'),(20,))
ds.addSample(loadImage('pic/gorilla3.png'),(20,))
ds.addSample(loadImage('pic/gorilla4.png'),(20,))
ds.addSample(loadImage('pic/gorilla5.png'),(20,))
ds.addSample(loadImage('pic/alligator1.png'),(30,))
ds.addSample(loadImage('pic/alligator2.png'),(30,))
ds.addSample(loadImage('pic/alligator3.png'),(30,))
ds.addSample(loadImage('pic/alligator4.png'),(30,))
ds.addSample(loadImage('pic/alligator5.png'),(30,))
ds.addSample(loadImage('pic/cat1.png'),(40,))
ds.addSample(loadImage('pic/cat2.png'),(40,))
ds.addSample(loadImage('pic/cat3.png'),(40,))
ds.addSample(loadImage('pic/cat4.png'),(40,))
ds.addSample(loadImage('pic/cat5.png'),(40,))
```



===RESULT====

Picture pic/giraffe6.png has a gorilla

Picture pic/alligator6.png has a **alligator**

Picture pic/dog6.png has a alligator

Picture pic/cat6.png has a gorilla

Picture pic/gorilla6.png has a **gorilla**

However, the method **only** improved reducing number of iteration needed during the training process and a bit of the accuracy. It is **still failed** to be able to detect all of the pictures (only improved the accuracy of alligator).

.

4) ***Part a) Now that you are well versed in playing with neural networks compare and contrast neural networks with linear regression and other classical statistics techniques we have covered so far in the course. What's different about them? What's the same? What kinds of information are different?***

In my opinion, below are advantages and some characteristics of neural network:

1. Neural Network technique is designed more towards certain datasets to which we know minimum knowledge about. Since choosing a perfect model for the datasets that we know nothing is very difficult, iterative method such as neural network could for the best approach.
2. The hardware architecture development, such as development of CMOS in computer architecture, has been a major boost in implementing Neural Network for computer simulation. Neural Network process is highly parallel and therefore the use of parallel processors is possible and cuts down the necessary time for calculations.
3. The proper use of Neural Network could replace the needs of manual mathematical modelling, hence, made automation more possible.
4. Robust in respect of (unknown) random noise and fault tolerant.
5. Possibility of on-the-fly adjustments.

Some limitations of Neural Networks:

1. It might take too much time only to train the machine to understand simple relation (the use of classical statistic would save time).
2. It might lead the user (human) to be undermine the relation of variables, start jumping into tuning the parameters without understanding the basic correlation of each predictor, for instance.
3. When the tuning method is not working, Neural Network is not more than a black box. It is very difficult to trace back and learn from it manually.

With this characteristic, I would argue that Neural Network might work best in some dataset environment in which we know nothing about mathematical model and we want the machine to learn and get better every single iteration:

- Speech recognition
- Face recognition
- Augmented reality
- Recommender engine (Movies, online shopping, targeted advertisement)

On the other hand, it has also some shared similarities with classical statistic method, such as both are used for future prediction / estimator.

Part b) Explain where you think machine learning is heading. Also, do you think classical statistics will always have a place in data science? Why or why not? The next method is to use pearson test to compare both variables:

As mentioned above, Neural Network is getting more popularity especially on pattern recognition and recommender engine because it has the capability to adapt to new dataset trends on-the-fly, and can be implemented in the form of automatic task.

However, I would argue that classical statistics will always have place for the type of datasets analysis that is driven by domain knowledge and does not require online parameter adjustment. For instance, to understand stock exchange in certain period of time, the use of classical statistics by finance and economic expert might lead to better result. They could extract the relevancy of statistic correlation between multiple predictors (not only using math/statistic approach, but general domain knowledge) and manually choose statistical model that fit best.