Computational Intelligence *Second* assignment

Mohammad ali farahat

97521423

1.1) Kohonen-SOFM

in this part we have 1600 input data that are RGB, we want to map them into 40x40 Kohonen map.

first of all we initialize weights and inputs by random

after that in each iteration we have to find best matching unit (bmu) and this neuron is winner:

The Euclidian distance: $i(x)=\arg\min \|X-W_j\|$

then find distance from each neuron from chosen neuron (bmu).

Then update the weights by this formula:

$$h_{j,i(x)} = e^{\frac{-d_{j,i}^2}{2\sigma^2}}$$

$$\Delta w_j = \eta h_{j,i(x)}(x - w_j)$$

* d is distance, sigma is radius of neighborhood

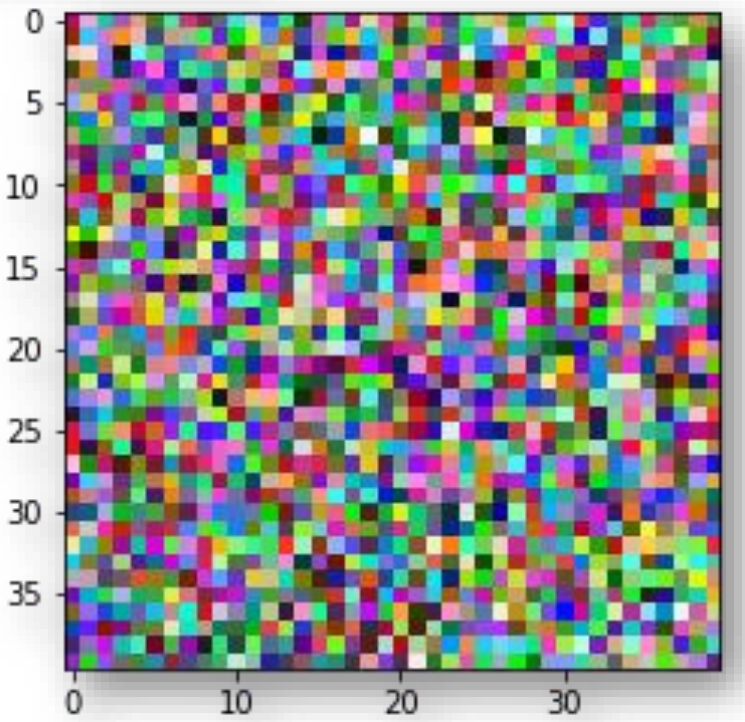
we do this work for enough iterations (I did 100)

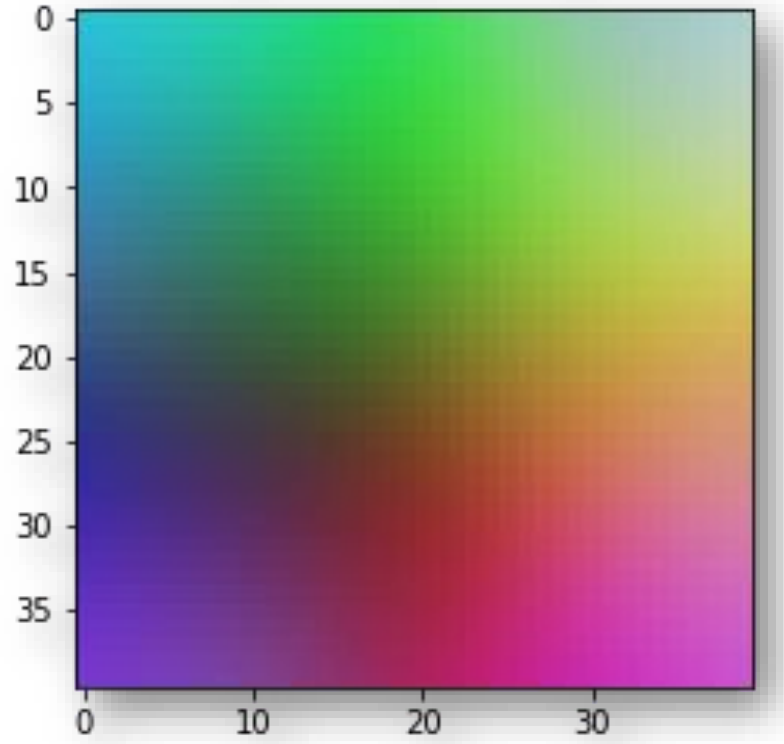
at the end we plot images from first random numbers and final weights to see the effect of this work

* note that weight <u>are not</u> exactly the inputs, because we can map more than one input to each neuron, but in this case we didn't .

see the results in the next page:

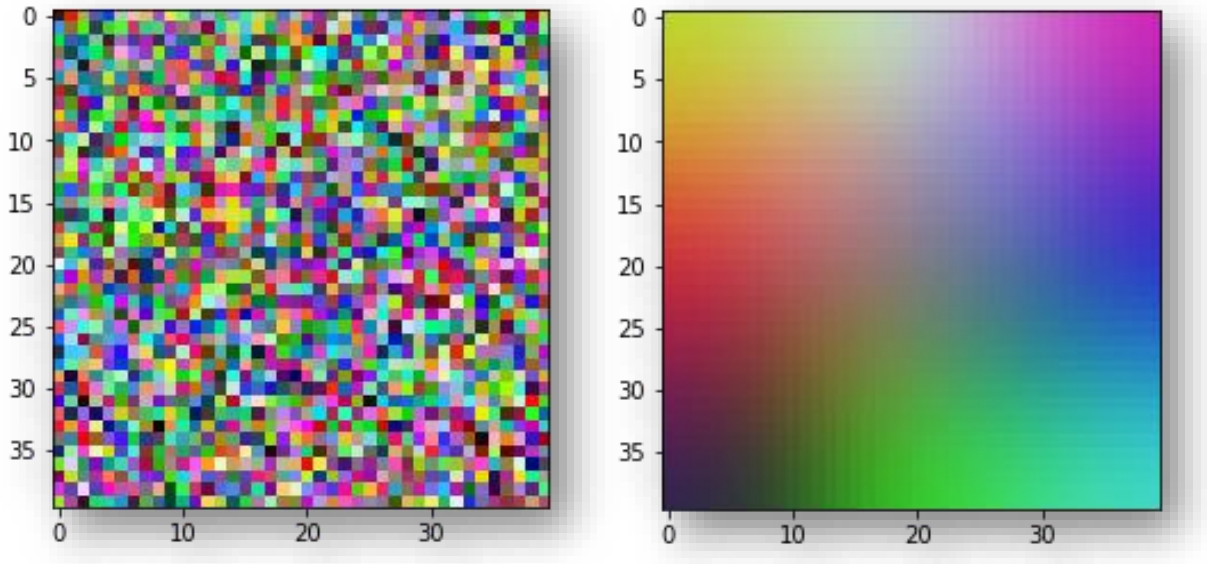
```
learning_rate = 0.1
sigma = 5
epochs = 100
```



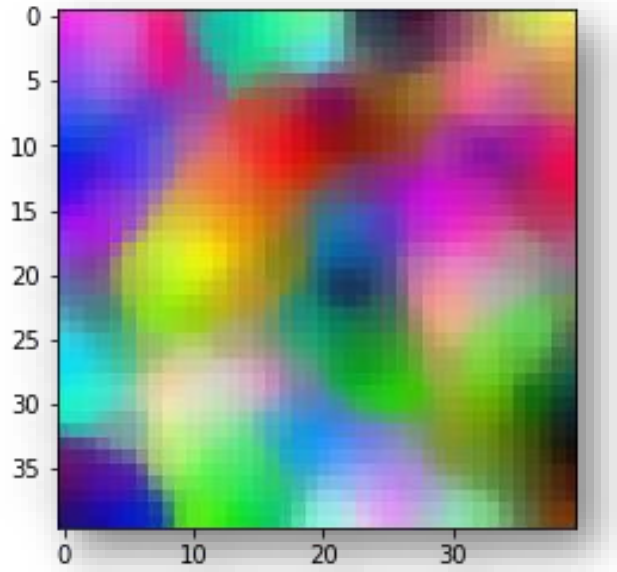


1.2) if the learning rate does not change over time, we can't ensure that network converged and many neurons change each time, and sometimes it gets stuck, we call it Map distortions.
for solving that problem we decrease learning rate each time by multiply it by a number close to 1 but smaller than 1, like 0.95

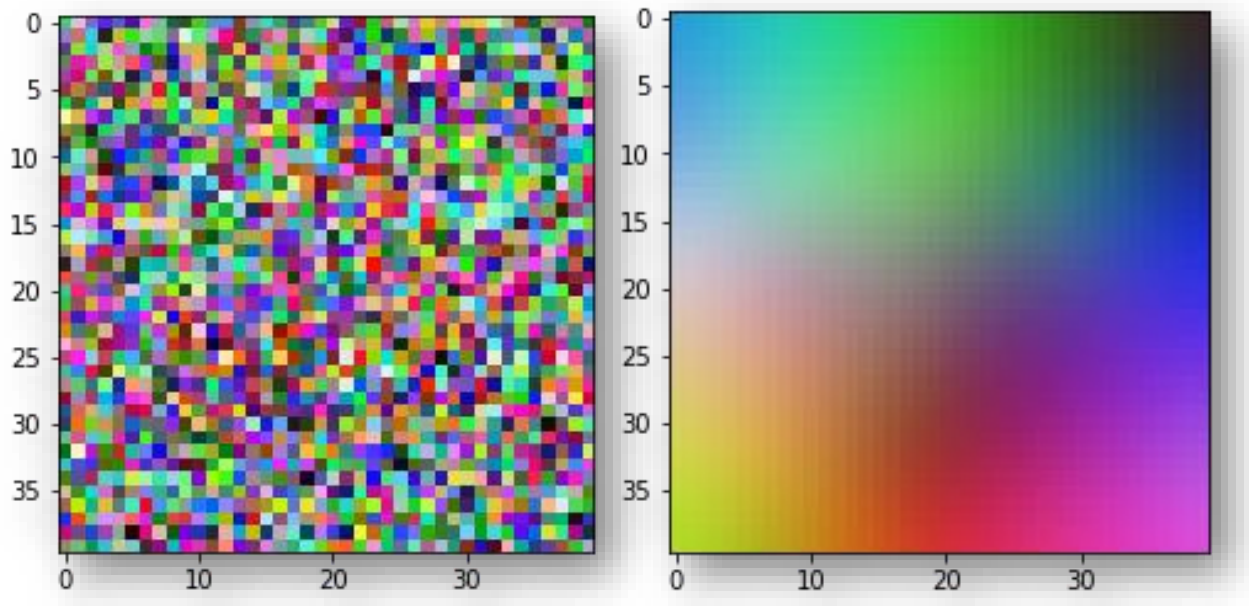
first I initialize learning rate by 1. so, after 100 iterations learning rate become 0.005
but we still have another problem here, the sigma.
I choose it manually and by try and error. But in the next part we'll figure it out.
output for this part is like:



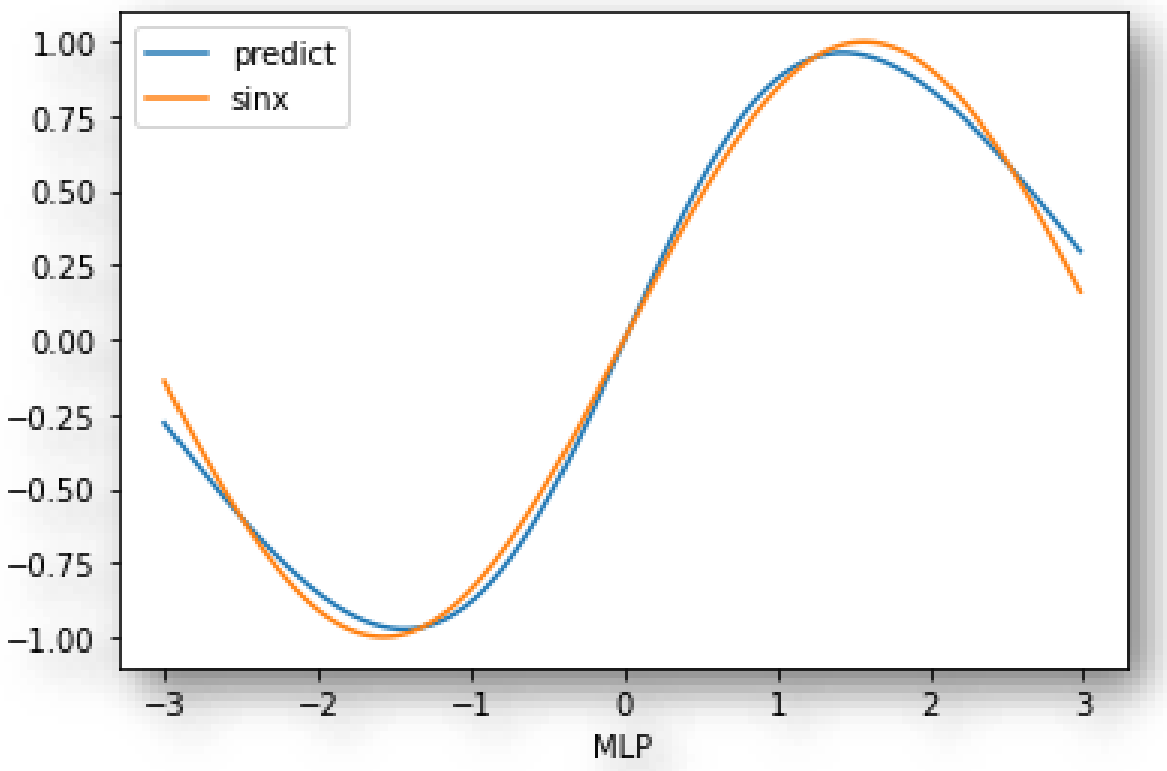
1.3) Decreasing sigma can be useful, because we don't have to choose it manually by try and error. First, we initialize it by something big, and in each iteration, we decrease it.
if we don't do this, and cannot choose a good sigma ourself, it maybe causes some neurons cannot see each other and for example we will have 2 different blue part in our map, something like this:



but now, we choose a big sigma and it decrease during time output looks like this:

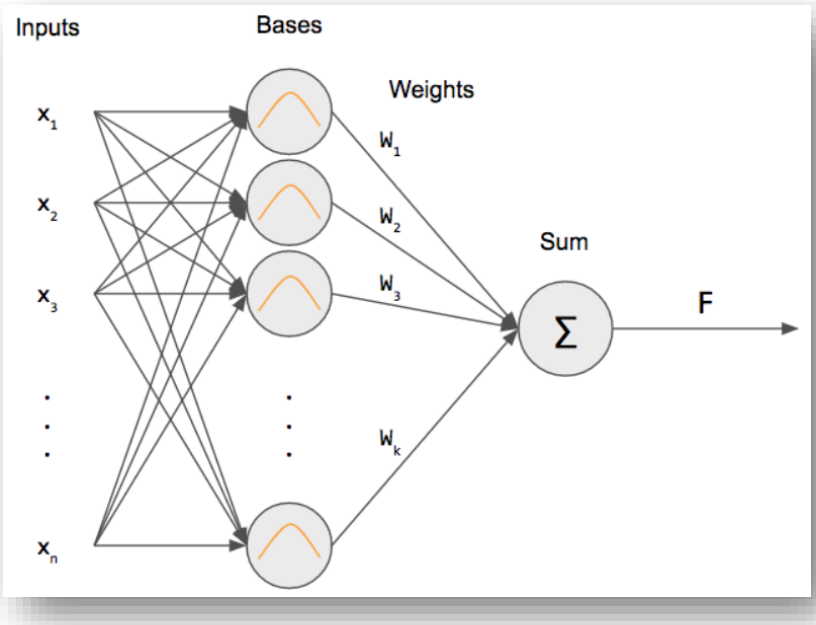


2.1) for this question, I used Keras library to train an MLP.
my MLP has 3 layers with 30, 20 and one neuron.
activation functions are sigmoid (except output layer that has no activation function)
I plot both sin(x) and prediction of MLP in [-3,3]
there is nothing more to say here :)
result is:



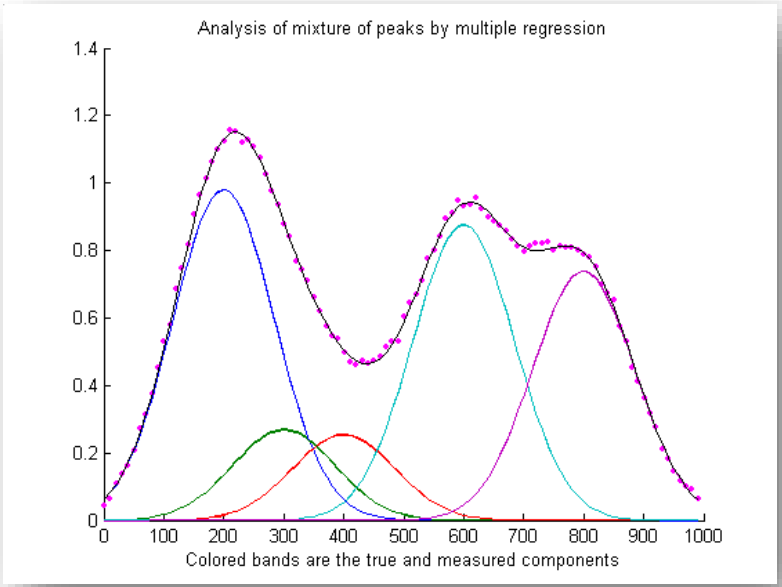
2.2) in this part we have to train RBF for sin(x)

RBF or Radial Basis Function is actually 2-layer neural network that activation function of middle layer is radial.

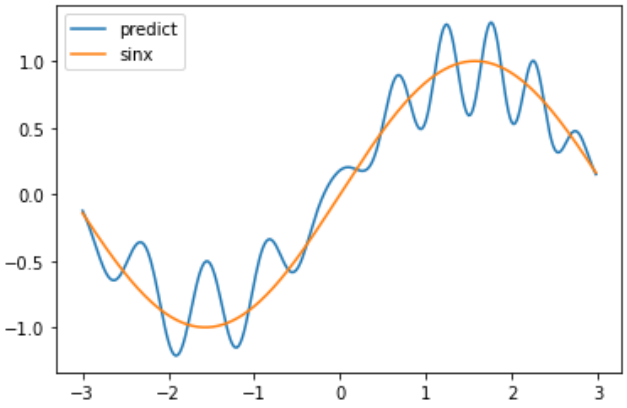


Gaussians RBF:

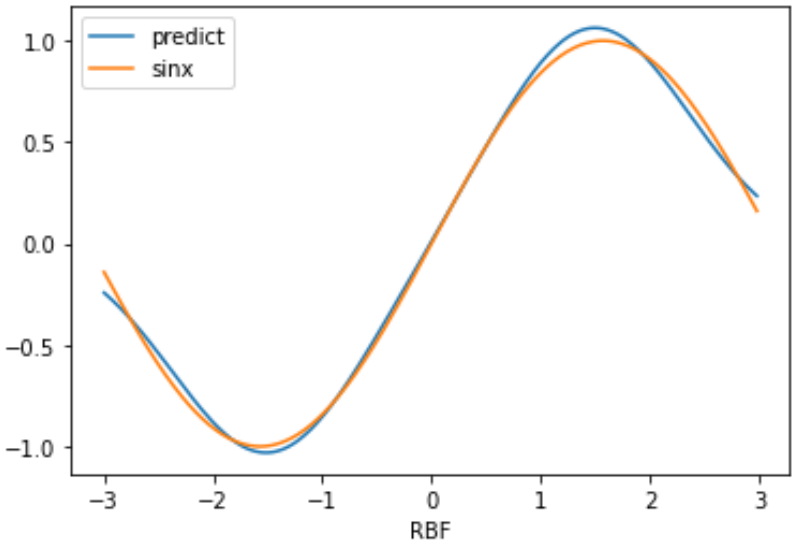
by Gaussians Distribution we can create any function we want. this is the main idea of RBF.



every Gaussian has one center and one derivation.
now, for training our RBF we have to find best number of Gaussian function and find their centers and derivation.
in this case we want to train sin(x) in [-3,3]
and we need 2 Gaussian function.
we can see here if we choose 10, something like this will happen:

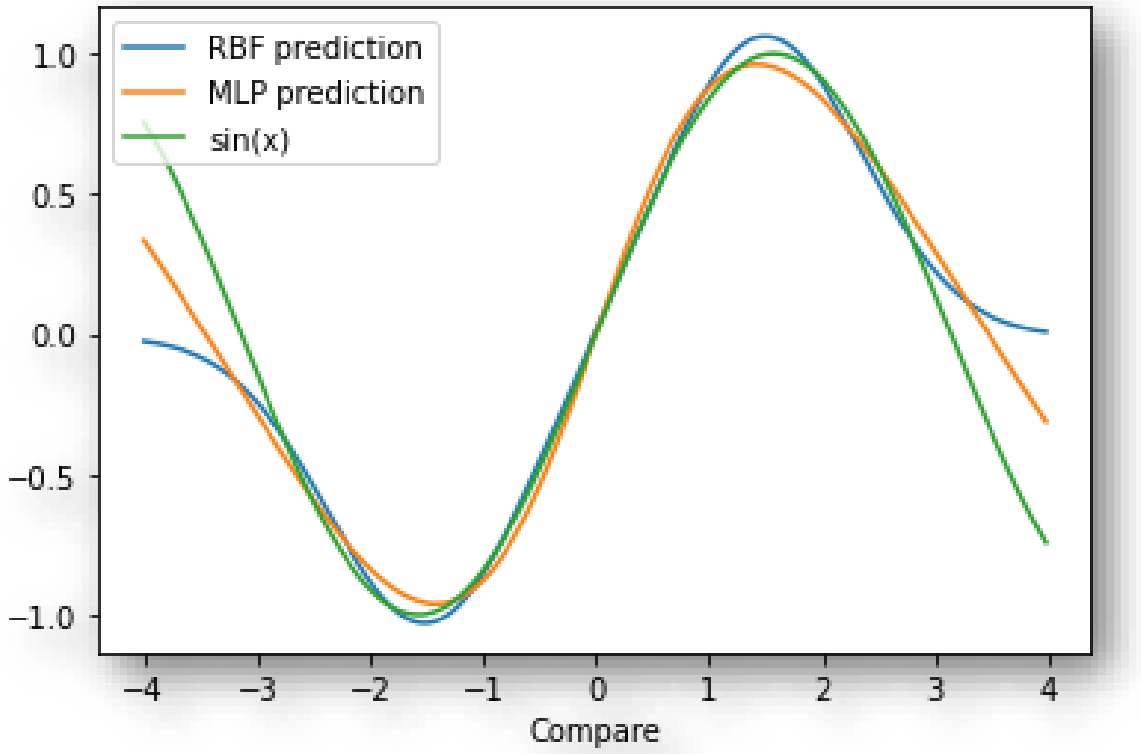


in training process first we find centers and derivations, after that like normal neural network, we update the weights and continues.
by learning rate of 0.01 and 1000 iterations output is like:



* I used this link for leaning this question implementations

2.3) in this part we have to compare MLP and RBF.



MLP accuracy is more than RBF, but of course RBF is so much faster than MLP, the whole process takes 1 minute.
MLP takes more than 40 sec, and RBF takes 20 sec.
both has 1000 iteration on same input data.
so, in this case maybe RBF is better choice.