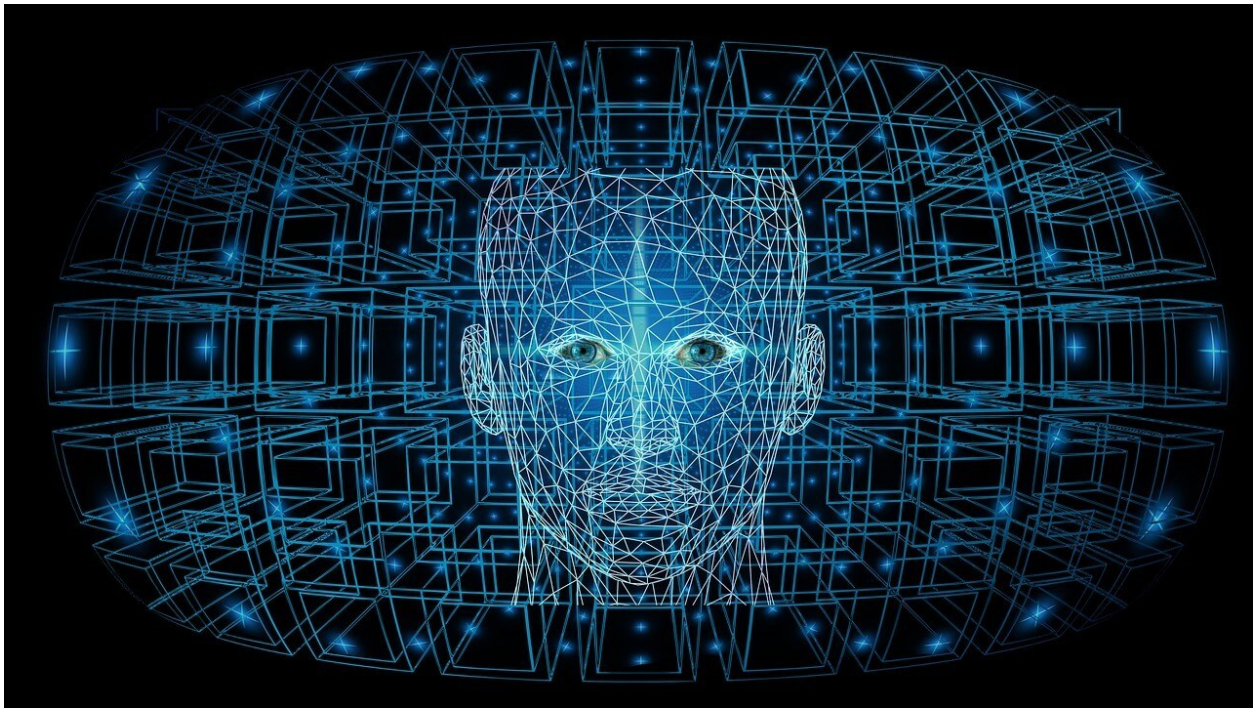


CI HW2

Sohrab Namazinia

97522085

A1.1

I have solved it and plot the result by implementing the Kohonen model. I have used comments for a better explanation of my code. But I also explain each cell of my code here too.

CELL #1:

In this cell, I have implemented all the basic functions and variables that I need later in the algorithm.

The width_lenth is 40 as the question says.

so Data_count is 1600 and I have set neuron count to 1600.

I have set the proper sigma, learning rate, etc by trial and error.

Create_data is a function for creating my color data, some scaled numbers between 0 and 1 (RGB values between 0 and 1).

Plot_map plots a dataset (colormap).

Build_map gives the indices of each cell from 1 to 1600.

Compute_distance_squared computes (distance^2) for a pair of neurons.

CELL #2:

This cell contains the initialization method which randomly initializes the weights with small values.

CELL #3:

This cell contains the competition step which returns the neuron that is the winner, in other words, the neuron that is most similar to the passed data.

CELL #3:

This cell contains the cooperation step that returns the heuristic values regarding their distance to the winner neuron.

CELL #4:

This cell contains the adaptation step which contains updating the weights. I have updated the weights by the Kohonen update formula:

W += learning rate * h * (x - w)

CELL #5:

This cell contains a method for running the Kohonen. It actually does one epoch of the algorithm. This epoch consists of iterating over all the datasets. For each data, doing the above steps that I explained. So total weights are updated by one epoch which contains iteration over all the dataset.

CELL #5:

In this cell, I have created the map, init weights, and created the dataset.

CELL #6:

This cell has a for loop with the size of my epoch count, in each for loop I have shuffled the dataset and ran Kohonen, and updated the weights.

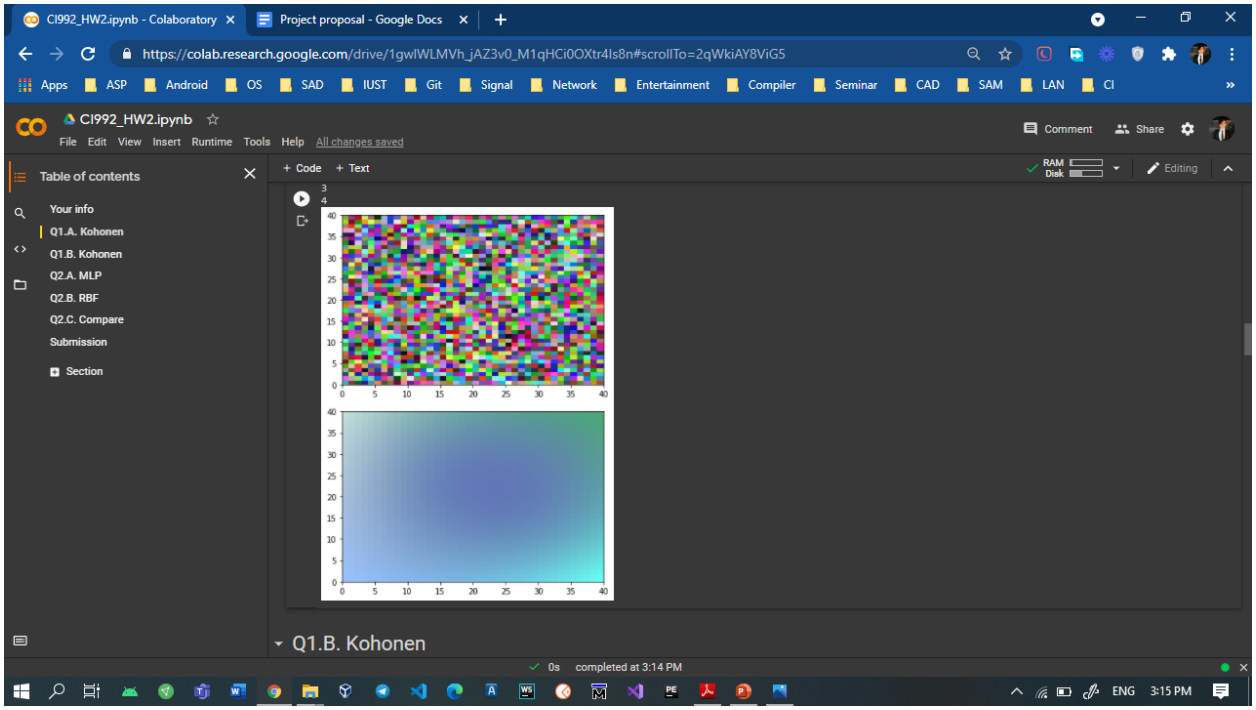
Finally, the weights are well updated and I have plotted both the data and output.

A1.2

My code is like the last part, but this time with a variant learning rate. If we keep the learning rate fixed during the algorithm, it may cause that we **overshoot** the utilized weights. It means that we may pass the utilized weight with our big steps. So, the **divergence** of getting to the best point (the best weights) happens.

The reason for this problem is that we keep the learning rate a fixed very small value, our steps to get to the best point are really slow and may take lots of epochs so that we get to our goal.

Even if we set the learning rate to a value that is not small, we should make it smaller during the algorithm. Because in each step we are closer to the goal. When you are closer to the goal, you should take smaller steps towards it, because you may **overshoot** and pass the goal. Again in the next update when you want to get closer to the goal, you undershoot it. This is what happens that is called **divergence**. Even if we converge, the convergence may happen later and takes a long time. In the previous question, I have solved the problem of fixed learning rate by setting the proper sigma. Here is a demo of how my output looks like with a fixed learning rate (but I solved it by setting proper sigma in the previous question).



in this part of the question, My learning rate is variant with the following formula for avoiding what I just explained:

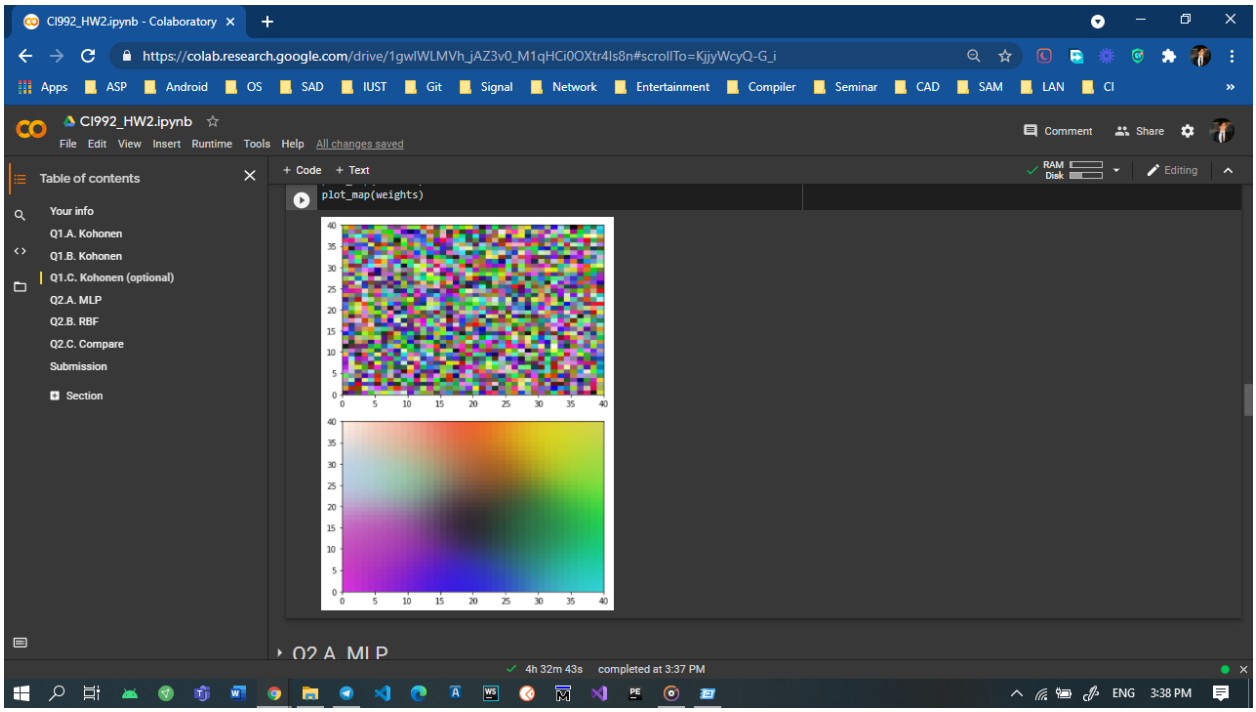
Alpha = 1 / (epoch_index + 1), it start from 1, moves towards zero.

A1.3 (optional)

Here I have set sigma to a bit initial value, then decrease it by the following formula:

Sigma = Sigma * 0.99

Finally, the output looks like the following image. That is because this formula decreases sigma smoothly, so no problem happens and sigma will be finally around 0.6 of its initial value. But I also tried it with sigma = sigma * 0.9 and that caused the partition colors to be very small, not enough effect to change the neighbors in iterations.



So finally i think it is good to decrease sigma and learning rate over epochs, but we should pay attention to the number of epochs to, meaning that we should consider epoch count in our formulas too.

A2.1

I have explained the cells of my code here (also commented in code):

CELL #1: import cell

CELL #2:

I have defined the domain and codomain of y = sin(x) function by NumPy.

CELL #3:

I have defined my NN model with 10 hidden neurons in the hidden layer.

The output layer has 1 neuron.

I have compiled the model. But our goal is metrics is mean squared error.

Then I have to fit the model with the specified batch size and epoch count.

Finally, I have stored the prediction result in the variable MLP_result

CELL #4:

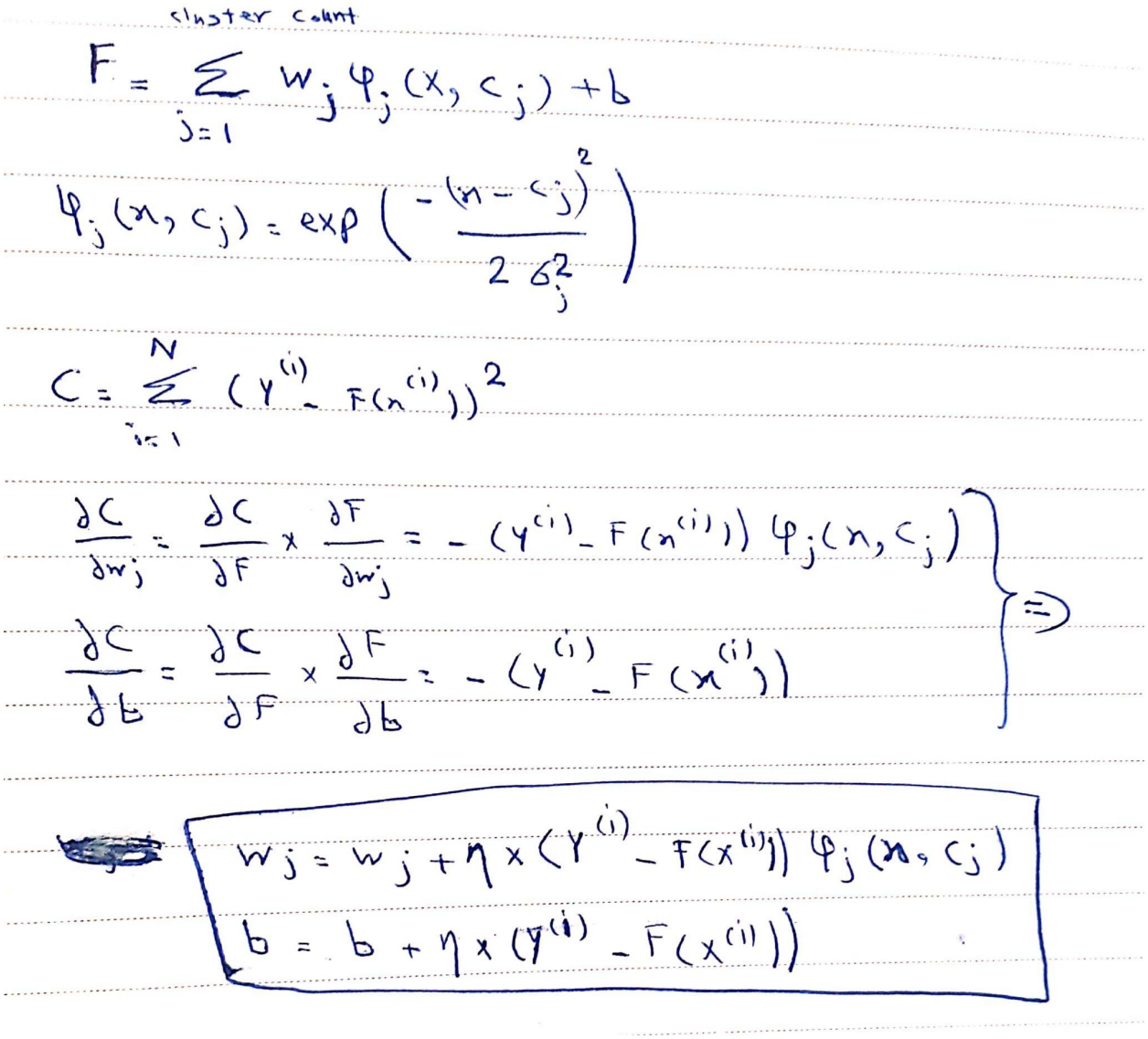
I have plotted both MLP_result and real sin(x) in the specified range for comparison.

A2.2

I have commented on my code completely but I do explain it here to cell by cell:
CELL #1: import cell
CELL #2: define some useful functions:
Get cluster data: returns the cluster for each data
Get points in cluster: returns all the data in the cluster
Get RBF centers & sigmas: returns the RBFs for starting algorithm

Gaussian function: returns a value in respect of distance to the center of the cluster

CELL #3: train RBF
It initializes the weights and b. Then there is a for loop with the size of epoch count, inside that another for loop on data to get the output of output neuron and compare it to the real result to get the error value for updating the weights and b by my following formulas. All the logic of my training (RBF) is in the image (next page).



CELL #4: predict RBF

It simply predicts the result of a set of X values by forwarding propagation.

CELL #5: run RBF

It does the whole actually! First computes the RBF centers and sigmas, then trains the network and finally predicts the results and returns them.

CELL #6: main

It sets the necessary coefficients and calls the run RBF method in the previous section and gets the predicted result.

CELL #6: plot

It plots predicted result and sin(x) both together in [-3, 3] for easy comparison.

A2.3

It is just one section which plots the results of sin(x), RBF sin(x) and MLP sin(x) all together. You can compare the results very easily. But setting different coefficients for each of MLP and RBF algorithms can result in a bit different graphs, so it just a demo.

"Thanks"