ENPM690: HW2

Link to code: https://github.com/gauharbains/CMAC-Cerebellar-model-articulation-controller

1. **Program a Discrete CMAC and train it on a 1-D function (ref: Albus 1975, Fig. 5) Explore effect of overlap area on generalization and time to convergence. Use only 35 weights for your CMAC and sample your function at 100 evenly spaced points. Use 70 for training and 30 for testing. Report the accuracy of your CMAC network using only the 30 test points.**

I've programmed a discrete CMAC using python. It is trained to predict f(x)= sin(x). A dataset of 100 evenly spaced points was constructed between the range (0,2π). Following this, the dataset was split into test and train in the ratio of 30:70.

The algorithm loops over each training example and associate's weights to each input depending on its value and the generalization factor. To do this, the inputs are converted to a scaled input of 0-1 and then multiplied by 35(number of weights) to get the central weight corresponding to each input. Following this, neighboring weights are also linked to the input depending on the generalization factor (no. of overlapping weights). For example, If the generalization factor is 5, then 2 weights on either side of the central weight are associated to that input. Note that in Discrete CMAC, no partial cell overlap takes place. The output is calculated by summing up the associated weights, which is then compared to the expected value to calculate the error. Following this each weight involved with that input is updated using the weight update equations. During each iteration over the training the dataset, the error for each training input is summed up to find the cumulative error. The training concludes when the cumulative error falls below a specified threshold.

Results:

The performance of the discrete CMAC was analyzed for generalization factor ranging between 1 to 34. To evaluate the performance of the network, two parameters have been used:

- MAPE : Mean Absolute Percentage Error
- RMSE: Root Mean Square Error

In addition to this, the time to convergence was also analyzed.

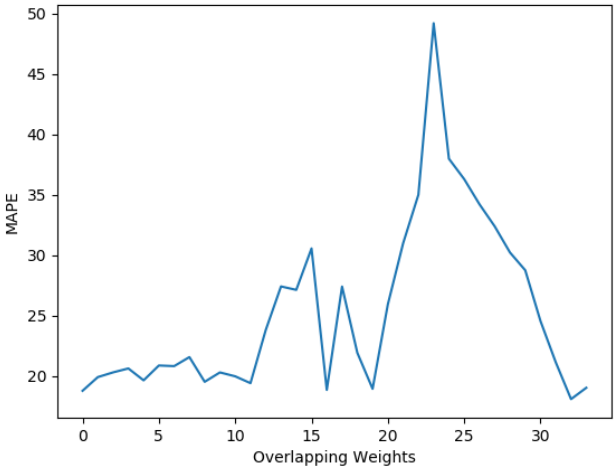

Figure 1 Root Mean Square Error - Discrete CMAC

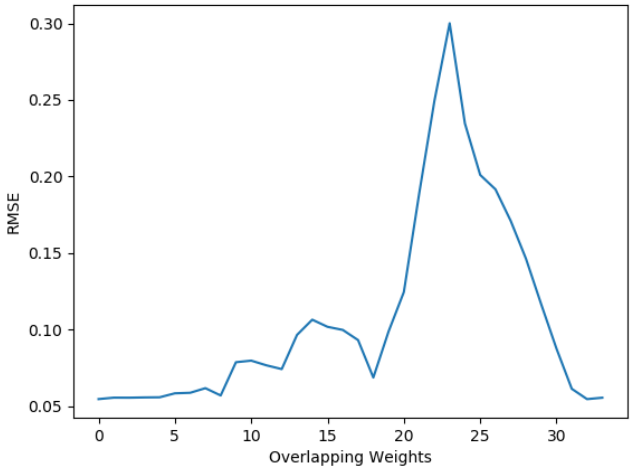

Figure 2 Mean Absolute Percentage Error - Discreet CMAC

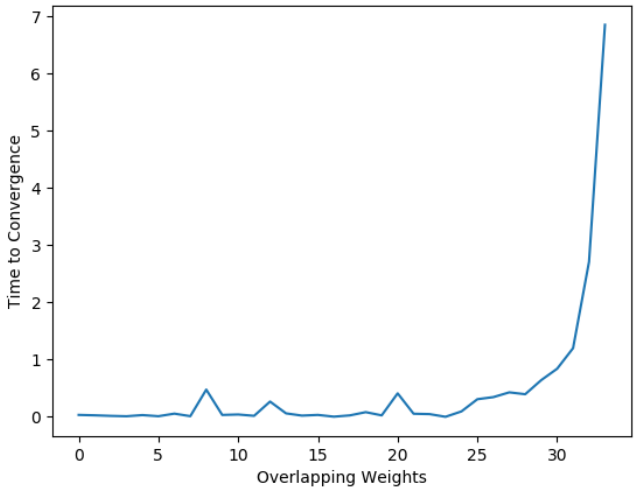

Figure 3 Time to Convergence for Discreet CMAC

After analyzing the RMSE and MAPE, it was found that the CMAC had the highest error (worst performance) when the generalization factor was 23. The performance was best for generalization factor between 0-5. In general, the model performs well for a lower generalization value, this is because when the amount of overlap increases, a change in value of a single weight affects a wider input range. Thus, it becomes difficult to have a higher accuracy. Also, it can be seen in

figure 3 that the time to convergence stays around the same when the generalization value is between 0-20, but after that it increases at a rapid pace.

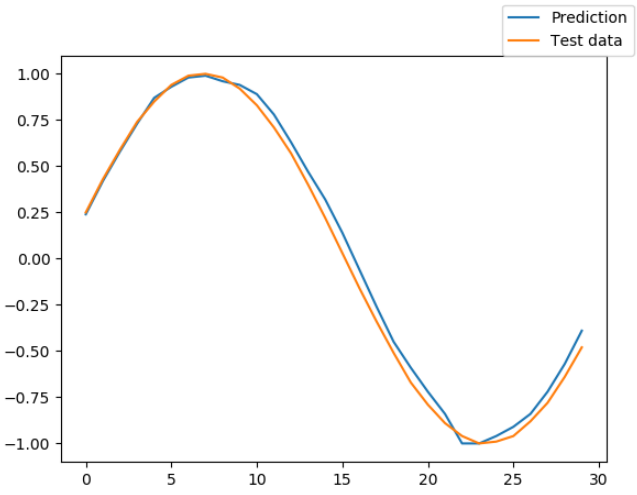

Figure 4 Prediction when generalization factor is 5

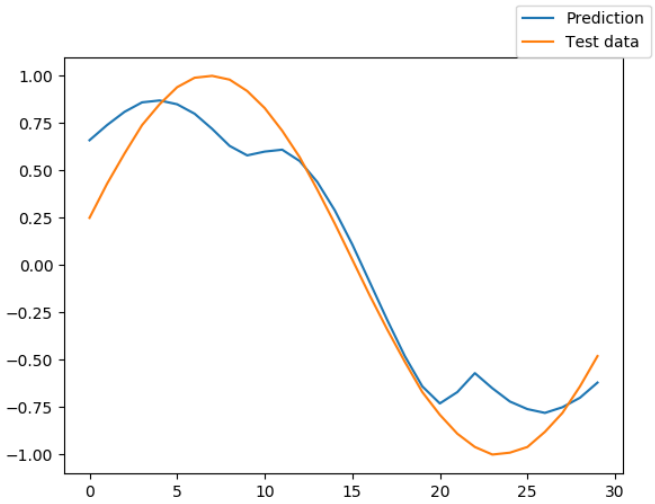

Figure 5 Prediction when generalization factor is 23

Figure 4 and 5 above show the prediction of f(x) = sin x by discrete CMAC when the value of generalization factor is 5 (Best Performing) and 23 (Worst Performing) respectively.

2. Program a Continuous CMAC by allowing partial cell overlap and modifying the weight update rule accordingly. Use only 35 weights for your CMAC, and sample your function at 100 evenly spaced points. Use 70 for training and 30 for testing. Report the accuracy of your CMAC network using only the 30 test points. Compare the output of the Discrete CMAC with that of the Continuous CMAC.

Most of the code to program a continuous CMAC is same as the one used to program a discreet CMAC, except that in a continuous CMAC, partial cell overlap is allowed. This also changes the weight update equations a little bit as now each weight has a contributing factor ranging from 0-1, depending on how much that weight is contributing to the output. Note that the contributing factor for each weight was equal to 1 in discreet CMAC.

The results for continuous CMAC are shown below:

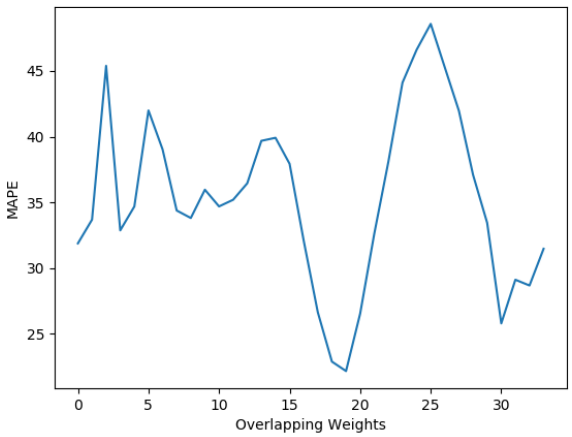

Figure 6 Mean Absolute Percentage Error - Continuous CMAC

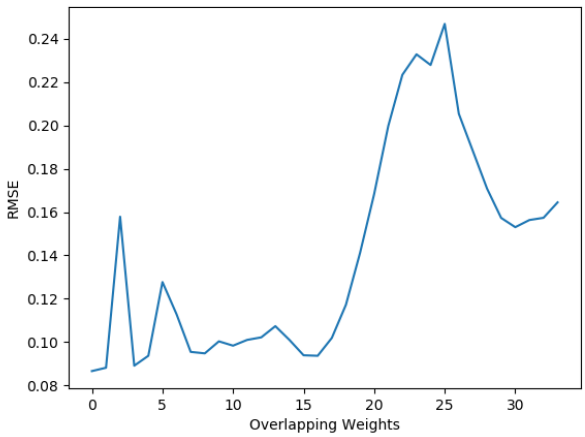

Figure 7 Root Mean Square Error - Continuous CMAC

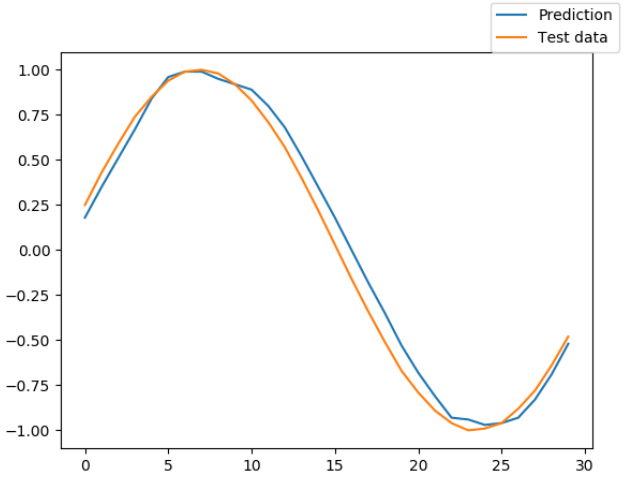

Figure 8 Prediction when generalization factor is 19

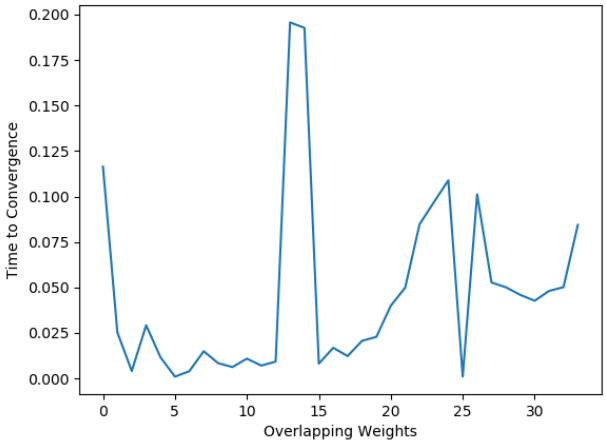

Figure 9 Time to convergence - Continuous CMAC

The results of continuous CMAC are shown in figures 6-9. The CMAC was found to have the best and worst performance when generalization factor is 19 and 25 respectively. Also, it can be seen in figure 8 that the time to convergence varies haphazardly.

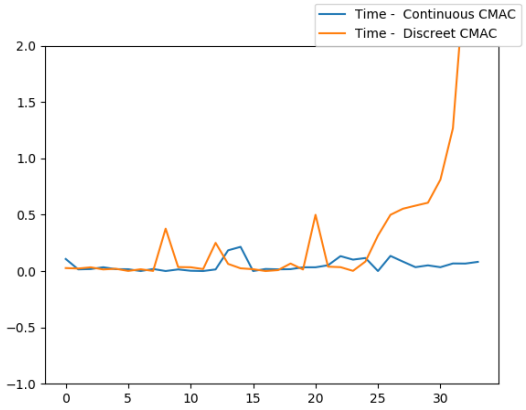

Figure 10 Comparison of Time to Convergence

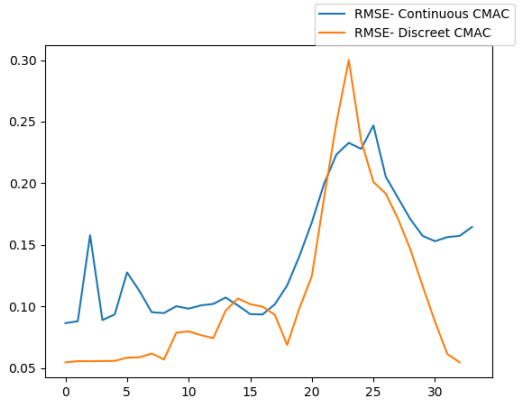

Figure 11 Comparison of RMSE Error

Figure 10 and 11 show the comparison of performance between the continuous and Discrete CMAC. On this dataset, in general we can see that the discrete CMAC has a better performance but the time to convergence is faster in continuous CMAC. It has to be noted that the results shown are for one set of values of the hyperparameters (learning rate, error to convergence etc). Changing the hyper parameters may give us different results.

3. **Discuss how you might use recurrent connections to train a CMAC to output a desired trajectory without using time as an input (e.g., state only). You may earn up to 5 extra homework points if you implement your idea and show that it works.**

Recurrent Neural Networks (RNN) are an interesting addition to basic neural networks that can process a sequence with any arbitrary length in contrast to basic NN which only take an input of fixed size. An RNN can be described as any network whose neurons send feedback signals to each other. Recurrent Neural Networks memorize the past and its decisions are impacted by what they have learned earlier. Recurrent connections have demonstrated tremendous success in predicting trajectories. To implement a recurrent connection to train a CMAC to output a desired trajectory, we must make our function dependent on its past output values. Our network should combine the input vector with a state vector with a fixed but learned function to produce a new state vector.