University of Thessaly

Neuro-Fuzzy Computing

ECE447

3rd Problem Set

Alexandra Gianni ID: 3382

Nikos Stylianou ID: 2917

February 26, 2024

Problem 1

Problem 2

We are asked to write a Python program that implements steepest descent algorithm for the 1-S^1-1 RBF network. The input function that we want to approximate is

$$g(p) = 1 + \sin(p\pi/8), \quad \text{for } p \in [-4, 4]$$

We select 30 data randomly from that interval and all parameters are initialized as small numbers using `numpy.random.randn` function. It returns a number at the exact specification as needed.

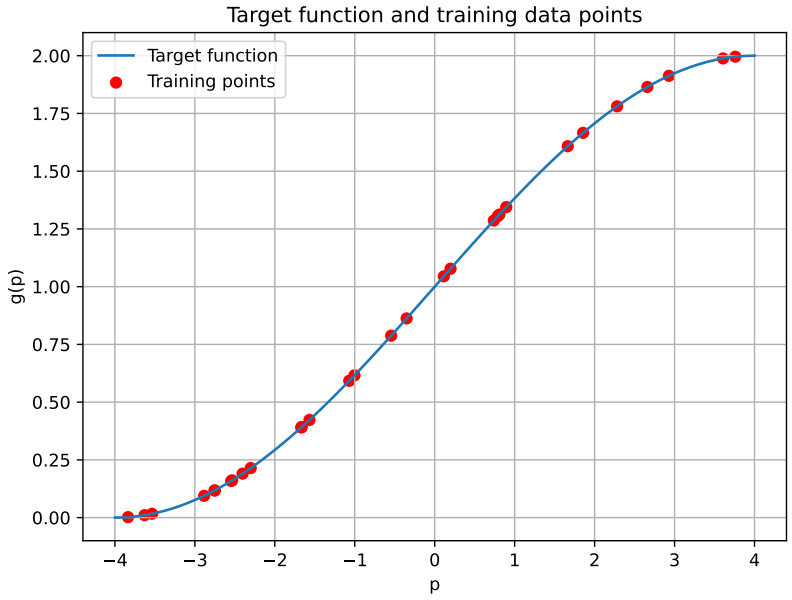
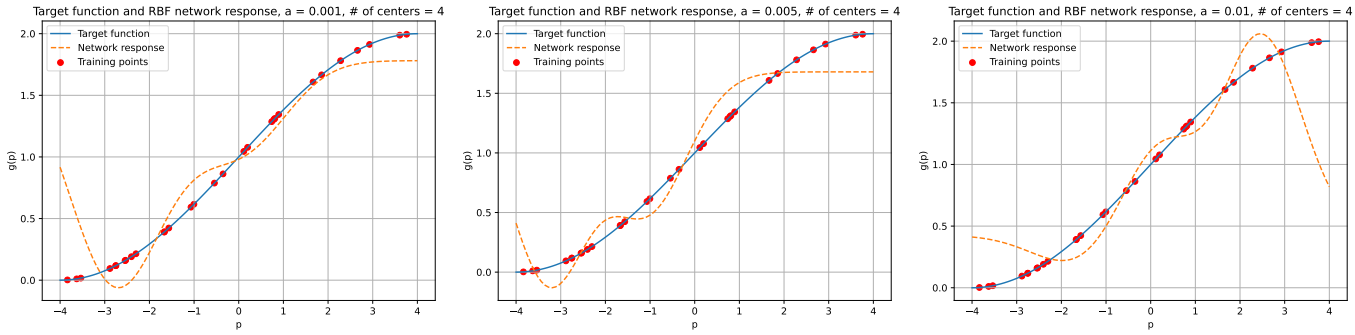


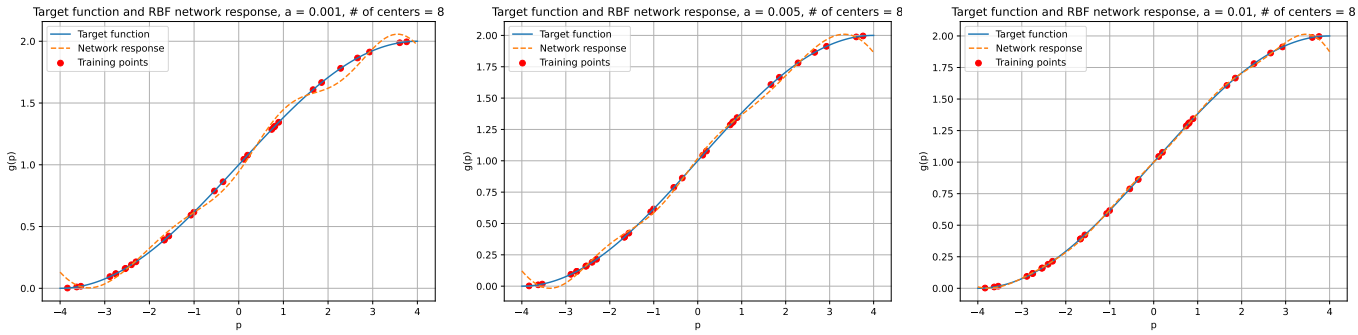
Figure 1: Input function in the specified area and the randomly assigned train data points.

For the randomly assigned data points, we added a custom seed number using in order for the results to be comparable but still use randomness. The number of centers define the total number of hidden layers in the network. So, in order to train it, thus impacting its accuracy.
Every arithmetic operation is done in matrix mode using `np.array` object type for better scalability and automation. For learning rate values α we used $[0.001, 0.005, 0.01]$ and for center the values $[4, 8, 12, 20]$ as suggested. As for the iterations number, we opted for $20 \cdot 10^3$ as we didn't want any possible bottleneck in the epoch number.
In figures 2, 3, 4 and 5 we can see the output of the neural network for each α and number of center (*hidden layers*).
Firstly, let's consider the impact of the number of centers. As we increase hat number from 4 to 8, 12, and then 20, the output dynamics undergo a notable transformation. With a smaller number of centers, the network struggles to capture intricate patterns in the data, leading to underfitting. This can be seen particularly well in figure 2, where number of centers is 4. The network fails to predict the input signal across all data points. Conversely, increasing the number of centers enables the network to model more complex relationships, reducing the underfitting and improving generalization. Again, these observations are very clear from the first change in the number of centers, from 4 to 8. The latter network has a smaller error across all data points when compared to the input signal. This increase of quality is not observed only in this increase of center numbers, but to all increases.



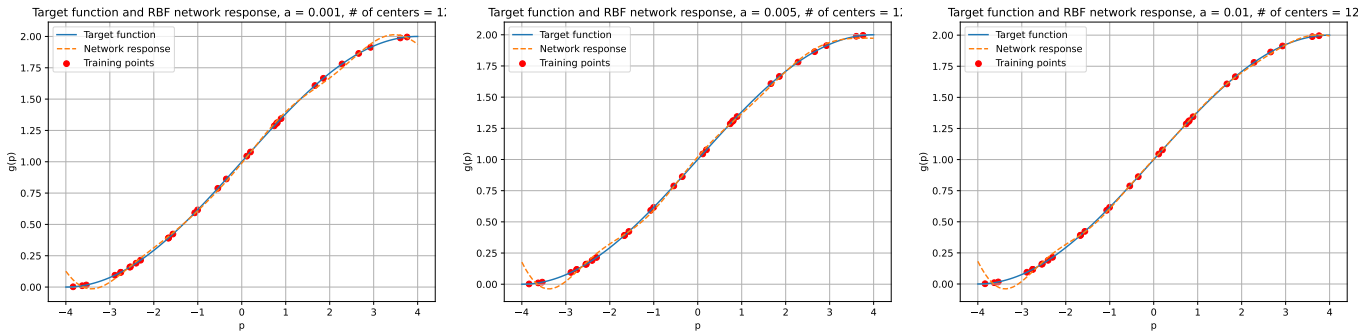
(a) $a = 0.001$ (b) $a = 0.005$ (c) $a = 0.01$

Figure 2: Input function approximation with number of centers $= 4$.



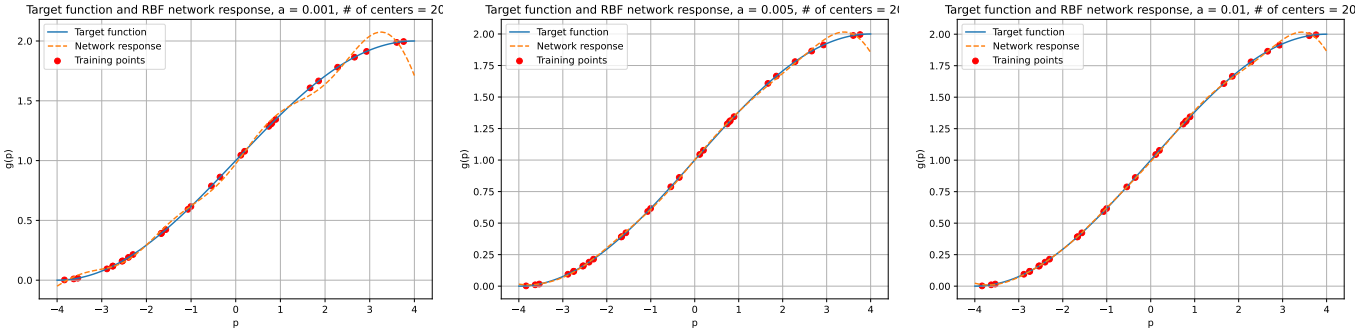
(a) $a = 0.001$ (b) $a = 0.005$ (c) $a = 0.01$

Figure 3: Input function approximation with number of centers $= 8$.



(a) $a = 0.001$ (b) $a = 0.005$ (c) $a = 0.01$

Figure 4: Input function approximation with number of centers $= 12$.



(a) $a = 0.001$ (b) $a = 0.005$ (c) $a = 0.01$

Figure 5: Input function approximation with number of centers $= 20$.

Next, let's delve into the impact of the learning rate on the RBF layer's output. A higher learning rate accelerates the convergence of the training process, enabling the model to reach a satisfactory solution faster. However, a learning rate that is too high might leads to oscillations or divergence, hindering the convergence process. From our calculations, when α is increased above 0.05, the system is diverging and by a large factor. On the other hand, a lower learning rate ensures more cautious updates to the model parameters, reducing the risk of divergence but potentially prolonging the convergence process. Thus, the output dynamics with different learning rates exhibit variations in convergence speed and possibly final performance, with an optimal learning rate striking a balance between convergence efficiency and stability.

Problem 3

LVQ (Learning Vector Quantization) is a type of artificial neural network algorithm used for supervised learning. It belongs to the category of competitive learning algorithms and is particularly effective for classification tasks.

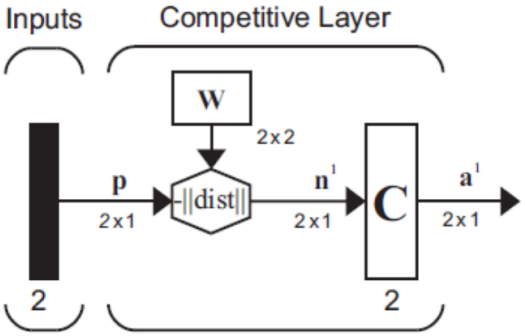


Figure 6: Given neural network.

The following $-||w_i - p||$ applies to every n_i^1. Also, we can also state that $a^1 = compet\,(n^1)$, where $compet$ is competitive learning layer.
During training, the distance between a and w is calculated using $dist = norm\,(p - w_{1,2})$ and judging by whose norm is grater, the winning neutron gets updated. Also, the presentation order of the vectors during training is: $p_1, p_2, p_3, p_2, p_3, p_1$.

All initial values are presented below:

$$p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad p_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \quad p_3 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}, \quad w_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad w_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

In order to reduce unnecessary computations, we implemented a convergence control that stops training when weights have very small differences with each other.
After letting the network converge, we have as final weights the following:
Training converged at epoch 10.

$$w_1 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}, \quad w_2 = \begin{bmatrix} 0.2 \\ 1.4 \end{bmatrix}$$

Finally, in figure 7 we can see the weights over epochs during training.

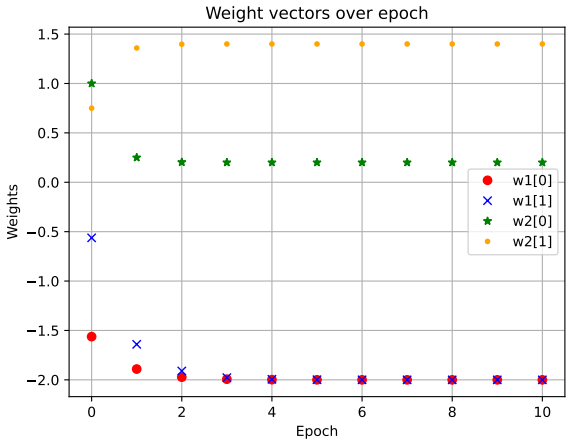


Figure 7: Weights over epoch during training.

Problem 4

Problem 5

Problem 6

Problem 7

Problem 8

Problem 9

Problem 10