Project Report

Kimia Honari

***Abstract*—Multi-layer perceptrons (MLPs) and Radial Basis Function Networks (RBFNs) are the most popular and important classifier in the neural network. In this project, we compared the performance of these two architecture as a classifier in Convolutional Neural Network (CNN), which is the best model for Image classification. The CNN-MLP and CNN-RBFN models have been evaluated with 10-fold stratified cross-validation techniques on MNIST, Fashion-MNIST, and CIFAR-10 dataset. The MLPs classifier could achieve a better performance than RBFNs in CNN model due to the large size of features that we have.**

I. Introduction

Classification is one of the most important and useful techniques in machine learning and has a variety of applications in different areas such as science, technology, etc. Neural networks are one of the models that could achieve a good performance on classification, so developing and using proper ANN to get small error rates on the dataset is a major concern. Multi-layer Perceptrons (MLPs) and Radial basis function networks (RBFNs) are two major neural networks that have been applied successfully for classification tasks. Two networks are similar in shape, but their training mechanism and classification process are very different. MLPs could have multiple hidden layers and use the back-propagation technique to train the connection weights, which could take so long because it uses the greedy search algorithm, and it may trap in local optima. On the other hand, RBFNs just have three layers, including the input layer, hidden layer, and output layer, and use the clustering technique and RBF function to classify the dataset. RBFNs have achieved good performance as well as the MLPs in many types of research. Besides, in some researches like [9], RBFNs could get better accuracy and sensitivity.

Convolutional Neural Network (CNN) is a very popular architecture in image processing, and MLPs is usually used as a classifier for this kind of network. In this project, we want to compare these two networks as a classifier and explore the possibility of achieving better performance with RBFNs. For our experiment, we used a simple dataset like MNIST, and two more complex dataset like Fashion-MNIST, and CIFAR-10. We used our two proposed model in project 1, and one time evaluated and parametrized the model with MLPs classifier (CNN-MLP), and the other time with RBFNs classifier (CNN-RBFN).

Our two models in project 1 used MLPs as a classifier, so we just replicated the parametrization for this project. For training the RBFNs classifiers, we used k-means clustering to compute the RBF in the hidden layer, and Recursive Least Square(RLS) optimization for training the weights for output neurons. As the method of training for the Convolutional layer is back-propagation, and it is different from the RBFNs training method, we used transfer learning techniques to parametrize our model. After we were training the CNN-MLP model, we re-run the training and testing data one time on our model and extract the output of the last convolutional layer for each data. Then we pass the output that we extracted along with the label of classes to the RBFN classifier for training. We evaluated both CNN-MLP and CNN-RBFN with 10-fold stratified cross-validation and compared two classifiers based on the geometric mean accuracy, which could give us a better view of the performance of the model in each class. The CNN-RBFN could achieve accuracy near CNN-MLP in a small and less complicated data set, but CNN-MLP performed better in a complicated and larger data set. Models have been implemented in python using Keras and TensorFlowlibrary, and the networks have been trained on NVIDIA GeForce RTX 2080 Ti.

The remain sections organized as follows. In the next section, we will explain CNN architecture, MLPs, and the Back-propagation technique. In Section 3, we will elaborate on RBFN and its training mechanisms. Section 4 consists of information on the transfer learning method in machine learning. In Section 5, we will describe the three datasets that we used in our experiment. In Section 6, we will elaborate on our models and how we chose the number of classification in RBFNs. Then we will explain our methodology more in Section 7, and Finally, we provide our results and related work in the last two sections. The code of our implementation and some of the results are provided in the Appendix.

II. Convolutional Neural Network (CNN)

One of the popular techniques in image recognition is the Convolutional neural network (CNN) introduced by LeCun et al. [1] in 1998. This method overcomes the weakness of traditional pattern recognition techniques like a fully-connected network. One of the most superior feature of CNN over traditional techniques is a unique design and techniques that help us to take advantage of the N-dimensional local structure surrounding every pixel. Besides, this design needs fewer parameters than the fully-connected layer, which requires a lot of parameters to connect each layer to the next other. In this section, we will elaborate on the architecture of CNN in the following.

A. *Convolutional Layers*

The first layers of CNN typically consist of a series of convolutional layers following by pooling layers. These layers help to take advantage of the two-dimensional structure of the input data by extracting elementary visual features such as oriented edges at first layers and then combined by the following layers to detect more complicated features such as objects. In each Convolutional layer, a filter or kernel slides over the input image. The filter slides over all the location, and multiplies the values in the filter with the original pixel values of different areas, and then sum up and represent that area with a single number. The output will be an array of numbers, which we call an activation map or feature map. Each filter responsible for extracting different features, and the weight of the filters will be learned by the training process. This way shares the weighs among a lot of pixels and reduce the time of training process, error, and loss. If the kernel size is n*n, the output of the Convolutional layer before applying the activation function is computed as below:

$$V_{ij}^{l} = \sum_{a=1}^{n-1}\sum_{b=1}^{n-1} w_{ab} X_{(i+a)(i+b)}^{l-1} \tag{1}$$

After each convolutional layer, we may have local or global pooling layers to eliminating irrelevant features for identifying the pattern. Two popular techniques in the pooling layers are average and max sub-sampling. Local pooling kernel size is typically 2 x 2, but global pooling runs on all the whole neurons of the feature map. There is no activation function apply and use in the pooling layer. Equation (2) shows the output of trainable average sub-sampling with kernel size n*m, and Equation (3) shows the output of max-pooling layer:

$$y_{ij}^{l} = Average(X_{(i+n)(j+m)}^{l}).w_{ij} + b_{ij} \tag{2}$$

$$y_{ij}^{l} = Max_n Max_b (X_{(i+n)(j+m)}^{l}) \tag{3}$$

After combining a series of convolution and pooling layers, we attached a fully connected layer to the end of the network as a classifier that helps to combine the extracted feature and classify it to different groups. In the following section, we will elaborate more on two different classifiers, such as the Multi-layer perceptron (MLP) and the Radial basis function network (RBFN).

B. *Multi-layer Perceptron (MLP)*

Figure 1 shows a Multi-layer perceptron architecture. In the Multi-layer perceptron, we can have more than one linear layer (combinations of neurons). For example, the simple example of the three-layer network consists of an input layer as the first layer, one hidden layer as the middle layer, and output layer. We feed our input data into the input layer and take the output from the output layer. We can increase the number of the hidden layer as much as we want, to make the model more complicated according to our task. As shown in Figure 1, each neuron or perceptron receives some inputs and uses some weights to implement an impact level of each input. Then the overall value passes to an activation function. Activation functions describe the input-output relations in a non-linear or linear way, which gives the model the power to be more flexible in describing arbitrary relations. In the architecture of MLP as a classifier, the number of neurons in the output layer is equal to the number of our classes and usually use softmax as an activation function. The Softmax function gives outputs as a vector in range of 0 and 1 that represents the probability distributions of a list of potential outcomes. The activation function of hidden layers depends on the architecture and the weights trained by the back-propagation technique.

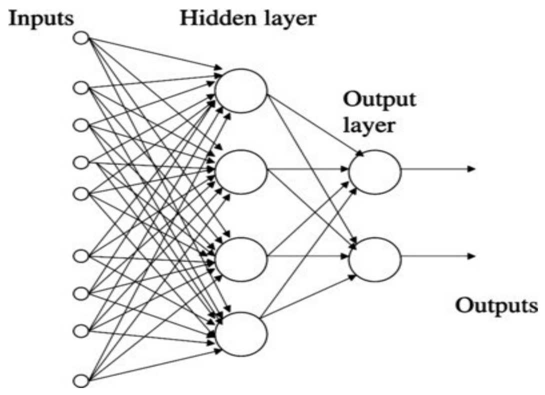


Fig. 1. Architecture of Multi-layer perceptron.

C. *Back-propagation Technique*

Back-propagation is an algorithm used to train a neural network. In this section, we summarized the procedure of this algorithm that tries to minimize the error energy over the training sample by adjusting the network's weights and biases. The error of energy or cost function is defined as follows:

$$\xi(n) = \frac{1}{2}\sum_{j\in C}(d_j(n) - y_j(n))^2, \; C = output\ neurons \tag{4}$$

The output of each neuron layer j at iteration defines by the following equation:

$$v_j(n) = \sum_{i=0}^{m} w_{ji} y_i, \; m = \#input\ neurons \tag{5}$$

$$y_j(n) = \phi_j(v_j(n)), \; \phi_j = activation\ function \tag{6}$$

For the second layer, the $y_i(n)$ is the value of input data without applying any activation function on it.

To apply a correction $\triangle w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$. we need to take partial derivative of $\frac{d\xi(n)}{dw_{ji}(n)}$. According to the chain rule of calculus, we may express this gradient as:

$$\frac{d\xi(n)}{dw_{ji}(n)} = \frac{d\xi(n)}{de_j(n)}\frac{de_j(n)}{dy_j(n)}\frac{dy_j(n)}{dv_j(n)}\frac{dv_j(n)}{dw_{ji}(n)} \tag{7}$$

If we take all the above derivatives, we can define the above equation as follows:

For output neuron:

$$\frac{d\xi(n)}{dw_{ji}(n)} = -\gamma e_j(n)\phi_j^{'}(v_j(n))y_i(n) \tag{8}$$

For hidden neuron:

$$\frac{d\xi(n)}{dw_{ji}(n)} = -\gamma e_j(n)\phi_j^{'}(v_j(n))\sum_k (e_k(n)\phi_k^{'}(v_k(n)w_{kj}(n)) \tag{9}$$

The k refers to all existence neurons in the next layer, and γ is the learning-rate parameter of the back-propagation algorithm. Equation 5 and 6 considered as a forward pass in back-propagation, and the rest of the equations are considered as a backward pass.

III. Radial Basis Function Network (RBFN)

A Radial Basis Function Network (RBFN) is a type of neural network, which was proposed by Broomhead and Lowe in 1988 [2]. The RBF network in most of the problems has equivalent capabilities as the MLP model, but in some cases, it gets more accurate results, with much faster training speed. The RBF network consists of a three-layer feedforward neural network, in which each layer is fully connected to the one following. Figure 2 illustrates the network architecture of RBFN. In the following, we will elaborate on the detailed architecture of RBFN.

1) The **input Layer** or the source node receives the m-dimensional input vector of x.
2) The **hidden layer** neuron called RBF nodes, which use a radial basis function (RBF) as an activation function. RBF is a non-linear transformation of the input and the output layer. Equation 10 shows the RBF function determined by one vector parameter called ”center” and one scalar called ”width”:

$$\phi_i(x) = \phi||x-c_i|| = exp(\frac{-1}{2\sigma^2}||x-c_i||^2),\ i = 1, ..., N \tag{10}$$

   The c_i defines the center, and σ represents the width of the radial basis function, and the vector x is the signal (pattern) applied to the input layer. Unlike a multi-layer perceptron, the links between the input layer and hidden units are direct connections with no weights to train.
3) The **output layer** consists of output neurons, which computes as follows:

$$y_i = \sum_{k=1}^{J} w_k i\phi||x-c_i||,\ i = 1, ..., J = \#output\ neuron \tag{11}$$

There are different methods for training the Radial-Basis Function Networks and defining centers and width like K-means clustering. One of the differences between the clustering algorithm and RBFN is that the RBF models the data using smooth transitioning circular shapes instead of sharp cut-off circles. This feature helps to address the ambiguity about the class of the point that belongs to none of the clusters if it is far enough away from all the centroid. The width parameter controls how fast the Gaussian function will decay, or in other words, defines the smoothness of the transition between two clusters. In this project, we used the K-means algorithm to cluster the input data and defines the centers. Based on the interpolation theory, the value of k, which defines the number of hidden layer neurons, is the same as the size of the training set due to the existence of noisy data. Unfortunately, the use of interpolation based on noisy data could lead to misleading results and could be of wasteful computational resources, particularly when dealing with large training samples. Hence we fin-tuned the value of the K and make the size of the hidden layer a fraction of the size of the training sample, and For defining the width parameter, we used the below function : (Dmax is the largest distance between the two centers)

$$\sigma = \frac{Dmax}{\sqrt{2K}} \tag{12}$$

At last, we need to optimize the value of weights between the hidden and output layer. For this purpose, we chose the Recursive Least Square optimization function that will be discussed along with K-means clustering in the next two subsections.

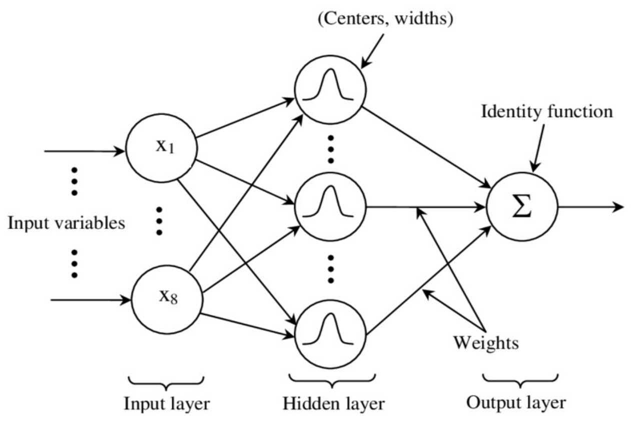


Fig. 2. Architecture of Radial Basis Function Network.

A. K-means Clustering

Clustering is one of the most common exploratory data analysis techniques used to get an intuition about the structure of the data. Clustering defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar, while data points in different clusters are very different. In other words, it tries to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as euclidean-based distance or correlation-based distance. The decision of which similarity measure to use depends on the subject of the application.

K-means clustering, proposed by MacQueen in 1967 [3], is an iterative algorithm used to partition a data set into k predefined groups automatically. The way the K-means algorithm works is as follows:

1) Selecting the number of cluster K.
2) Randomly selecting K points for the centroids.
3) Compute the sum of the squared distance between data points and all centroids.
4) Assign each data point to the closest cluster.
5) Update the centroids of each clusters by the average value of the all data points that belong to that cluster.

The objective function that we need to minimize is as follows: (μ_j is a current center of cluster j)

$$J = \sum_{j=1}^{K} \sum_{C(i)=j} ||x_i - \mu_j||^2 \tag{13}$$

The random initialization of centroids at the start of the algorithm may lead to different clusters. In some cases, it may stick in a local optimum and not converge to a global optimum. Therefore, it's better to run the algorithm using different initialization of centroids and pick the best result. We used this algorithm to define two parameters of center and width in the RBF function.

B. Recursive Least Square Optimizer

The recursive least-squares (RLS) algorithm is a popular and practical optimization algorithm used extensively in signal processing, communications, and control. This algorithm was discovered by Johann Carl Friedrich Gauss, who is a German mathematician, in 1821. The idea behind RLS filters is to minimize the weighted least squares error by appropriately selecting the filter of coefficient, which is in our problem is the value of weights between the hidden and output layer. The error implicitly depends on the filter coefficients through the estimate is defined as follow:

$$estimated\ value = y(n) = \phi(n)^T w(n) \tag{14}$$

$$error = \alpha(n) = d(n) - y(n),\ d(n) = desiredvalue \tag{15}$$

The summary of this algorithm is as follows:
Given the training sample $\{\phi(n), d(n)\}$ do the following computations:

Parameters:

- p=filter order
- λ =regularizing parameter (is a small positive constant)

Initialization:

- w(n)=0
- $P(0) = \lambda I$

for n = 1; 2; :::; N:

$$\alpha(n) = d(n) - \phi(n)^T w(n) \tag{16}$$

$$g(n) = P(n-1)\phi(n)\{1 + \phi_n^T P(n-1)\phi(n)\}^{-1} \tag{17}$$

$$P(n) = P(n-1) - g(n)\phi(n)^T P(n-1) \tag{18}$$

$$w(n) = w(n-1) + \alpha(n) g(n) \tag{19}$$

The cost function based on the regularization parameter compute as follows:

$$\xi_a v(w) = \frac{1}{2}\sum_{i=1}^{N}(d(i) - y(i))^2 + \frac{1}{2}\lambda||w||^2 \tag{20}$$

We implemented and used this optimization algorithm to estimate the value of weights in the output layer of RBFN.

IV. Transfer Learning

Transfer learning is a machine learning method where a model's parameters or output that developed for a specific task is reused as the starting point for a model on a second task. This method first time was proposed by Lorien Pratt In 1993 [6]. It is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones. The key motivation for using this technique is that most models that solve complex problems need a whole lot of data and getting vast amounts of labeled data for supervised models. Gathering and labeling this amount of data for each problem can be tough. Therefore, transfer learning tries to see how to leverage and utilize knowledge from pre-trained models and use it to solve new problems. Figure 3 compares the traditional learning method with transfer learning.

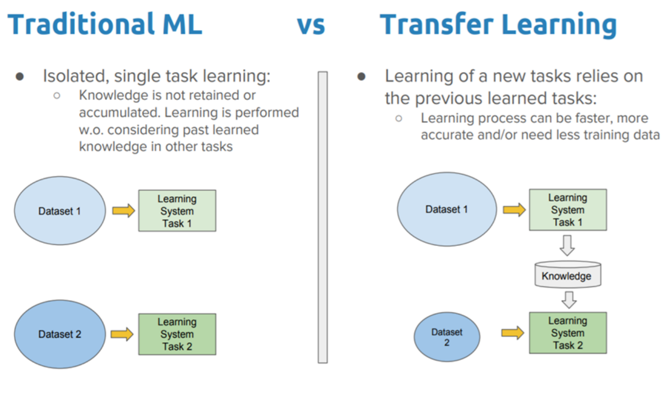

Traditional ML
vs
Transfer Learning
Isolated, single task learning:
Knowledge is not retained or accumulated. Learning is performed w.o. considering past learned knowledge in other tasks
Dataset 1
Learning System Task 1
Dataset 2
Learning System Task 2
Learning of a new tasks relies on the previous learned tasks:
Learning process can be faster, more accurate and/or need less training data
Dataset 1
Learning System Task 1
Knowledge
Dataset 2
Learning System Task 2

Fig. 3. Traditional Learning Vs Transfer Learningg. [4]

Deep learning systems and models have layered architectures that learn different features at different layers. For example, in CNN models after series of convolutional layers, it usually connects to a fully connected layer to get the final output. This layered architecture allows us to utilize a pre-trained network without its final layer as a fixed feature extractor for other tasks. Figure 4 shows this method very clear.

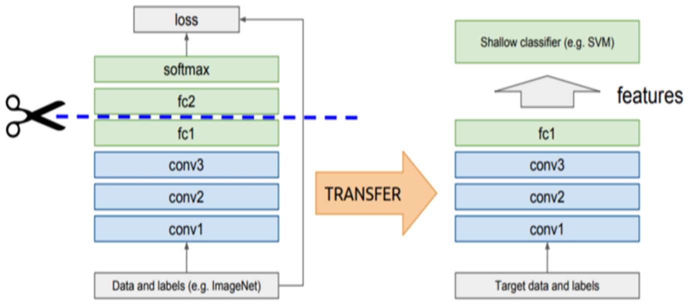


Fig. 4. Transfer Learning in Deep learning. [4]

In this project, we want to compare MLP and RBFN as a classifier. Because the RBFN uses K-means and RLS for optimization, we could not train it along with the Convolutional layers. Therefore, we used transfer learning and used the output of the last Convolutional layer as an input of our RBFN model.

V. Dataset Description

For evaluating our model, we used the MNIST-Digit dataset, and two harder classification problem, called Fashion-MNIST, and CIFAR-10 dataset. We will describe each of these datasets and the data preprocessing in the following subsections.

A. MNIST-Digits Dataset

MNIST consists of handwritten digits images in 28*28 pixels with 256 grey levels. The digits are between 0 to 9 (10 classes). It has a training set of 60,000 examples and a test set of 10,000 samples. This dataset is a subset of a larger dataset collected in the National Institute of Standards and Technology(NIST). The samples belong to 250 different people, half of them are from the Census Bureau, and the rest of them are high school students. More details could be found in [5].

B. Fashion-MNIST Dataset

The Fashion-MNIST dataset created in 2017 by Xiao H. et al. [7]. It has the same image size, data format, and structure as MNIST, and it uses as a direct drop-in replacement for the original MNIST dataset because the MNIST dataset is simple, overused, and can not represent modern computer vision tasks. The dataset contains images shot by professional photographers from different aspects of fashion products such as clothes, dress, etc. The detailed of the database could be found in https://github.com/zalandoresearch/fashion-mnist.

C. CIFAR-10 Dataset

CIFAR-10 [8] is an RGB image dataset, consisting of 60,000 images of size 32×32 in ten classes, with 6000 images per class. There are 50,000 training images and 10,000 testing images in the dataset. The ten mutually exclusive classes are airplane, automobile, bird, cat deer, dog, frog, horse, ship, and truck. Figure 5 shows some samples from the CIFAR-10 dataset.

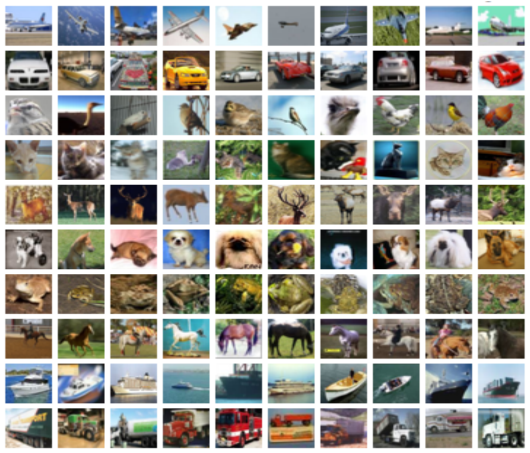


Fig. 5. Samples of CIFAR-10 dataset. [17]

D. Data preprocessing

We normalized the value of pixel value in input data between 0 and 1 by dividing each pixel by 255. Normalization helps to prevent processing large integer values and speed up the training process. After that, we applied the centering approach and subtracted the mean value from each pixel. This technique helps every pixel value will be centered close to 0.5. Finally, for output data, we use the to_categorical function in Keras to convert a class vector (integers) to a binary class matrix. Because our purpose is to compare MLP with RBFN, we did not use augmentation in our evaluation.

VI. CNN models

In project 1, we proposed two updated versions of Lenet-5 architecture proposed in [1], called updated version 1, and updated version 2. Both our models used MLP for classification. To compare the performance of RBFN and MLP, we used our previous model and replaced the MLP with the RBFN classifier. Due to the different training methods that we used for RBFN and convolutional layers, we used transfer learning techniques to train RBFN. We first trained our two modes with MLP and stored all of the output of the convolutional layer for each sample and then trained the RBFN on the output of the last convolutional layer. In the following subsection, we will elaborate on the architecture of our two models with MLP and RBFN and explain how we tuned the hyper-parameter K for the RBFN classifier.

A. Updated Version 1

In this version, we have just updated some parts of LeNet-5, such as activation function and pooling layer. Based on some studies like Scherer et al. [15], max-pooling is more powerful than the sub-sampling operations, so we replaced the pooling layer of Lenet-5 with the max-pooling layer. We also replaced the tanh activation function with ReLU due to the less computation, and good performance that it has. In the following, we elaborate more on the architecture of the model.

The first layer is the convolutional layer with 6 feature maps that receives the input shape of the 32*32*3 color image form CIFAR-10 dataset, and the input shape of 28*28 from MNIST and Fashion-MNIST dataset. The kernel size is 5*5, and the stride value is (1,1). After that, we applied a max-pooling layer with a kernel size 2*2 and a stride value of (2,2).

Layer third is another convolutional layer with 16 feature maps, kernel size 5*5, and stride value of (1,1), which following by a max-pooling layer as same as the second layer.

The fifth layer is a convolutional layer with 120 feature maps, kernel size 5*5, and stride value of (1,1). Finally, we flattened the convolutional layer and used the same MLP classifier, one dense layer with a size of 84, followed by a softmax layer.

Due to the different input shapes of CIFAR-10, the output size of the last convolutional layer is different from MNIST and Fashion-MNIST. For CIFAR-10 is an array (None,480), and for the other two, it is (None,120) as we used the same MLP architecture with the size of 84 for the hidden layer, and a softmax layer as output. We used the same RBFN to have a fair comparison. In other words, in both MLP and RBFN, the input size is different for CIFAR-10, but the hidden and output layer is the same.The detail of architecture and layer size for three datasets is available in Appendix B.

1) RBFN model: For determining the best value of K for RBFN model that is suitable for updated version 1, we repeated the training process on all training data of three datasets for the range of 20 to 200 classification. Figure 6 shows the geometric mean accuracy for the test datasets, as you can see by increasing the value of K, the accuracy goes up and down, but it increases with a low slope. Therefore, we pick the value of 90 for K, because it is computationally efficient and achieves relatively good accuracy.

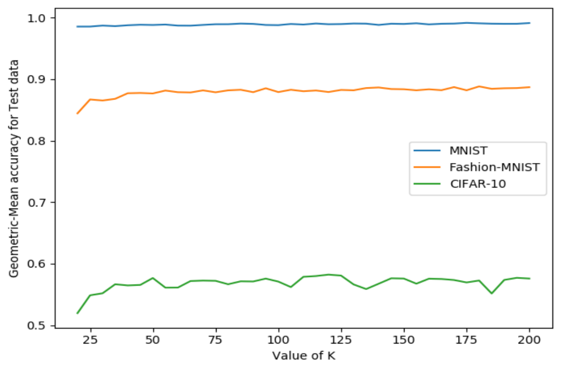


Fig. 6. Geometric mean accuracy for test data of three dataset in different range of K.(updated version 1)

B. Updated Version 2

In this model, additional to using max-pooling layer and ReLU activation function like updated version 1, we changed the kernel size and the number of feature maps to get better performance. Based on some experiments in Kaggle [16], we altered the kernel size and number of feature maps. We removed the third convolutional layer and changed the first and second convolutional layer kernel size to 3, and the number of feature maps for fist one to 32, and the second one to 64. For the MLP classifier, we changed the hidden layer size from 84 to 256, with the same output softmax layer. Like the updated version 1, the output size of the last convolutional layer for CIFAR-10 is different from MNIST and Fashion-MNIST dataset. The detail of architecture and layer size for three datasets is available in Appendix C .

1) RBFN model: The same method, like updated version 1, has been done to tuning the hyper-parameter K. We repeated the training process for all three datasets in the range of 10 to 400 classification. Figure 7 shows the geometric mean accuracy of test datasets; as you can see after 200 classifications, the accuracy decrease for the MNIST dataset, and do not improve a lot for other two datasets. After analyzing the accuracy for the value of K, we chose the 120 classifications because it is computationally efficient and achieves a good accuracy in all three datasets.

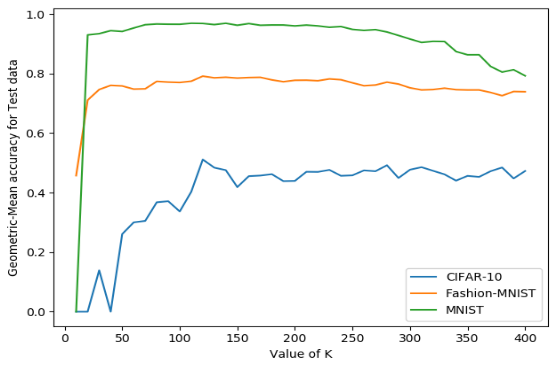


Fig. 7. Geometric mean accuracy for test data of three dataset in different range of K. (updated version 2)

VII. METHODOLOGY

For comparing MLP with RBFN, we evaluated CNN-MLP and CNN-RBFN models with 10-fold stratified cross-validation and then one time trained the models on the whole training dataset and validated it with the testing samples. We used the class of k-fold cross validation that we implemented for project 1. In the class, we randomly split the data of each class to 10 fold, and then in 10 iterations, we train the model with 9 folds and validate the model with the remaining fold. For validation, we calculated the geometric mean of the accuracy of each class, which we will clarify more in the following subsection. For training the CNN-MLP, we replicated and used the code that we explained in project 1, but for training CNN-RBFN we used transfer learning which we will explain in the following subsection.

A. Transfer Learning for CNN-RBFN

For training RBFN, it needs to use the transfer learning technique because the method of training CNN and RBFN is not the same. So, we first splited the data into 10 fold and saved them in the .numpy file. Then trained the CNN-MLP with the data, and saved the best model. After that, we load the saved model, run the model one time with the related saved training, and saved validation data, and extracted the output of the last convolutional layer for all training and validation data. Finally, we passed the output of the last convolutional layer as an input to the RBFN model along with the original label of classes. In the training process of RBFN, we first classified the input data to k classification and then computed the value of the hidden layer with radial basis function. At last, we optimized the weight of the output layer with RLS optimization algorithm, and apply softmax activation function on the output.

B. Geometric Mean Accuracy

If the classes do not have balanced samples, overall accuracy is a poor measurement for evaluating the performance. The geometric mean accuracy is another method for assessing the performance that is sensitive to the model accuracy in each class. For calculating the Geometric mean, we extract the confusion matrix after predicting the model with data. Then calculate the accuracy of each class by Equation 21, and computer the geometric mean by Equation 22. The implementation of geometric mean is available in Appendix D.

$$acc_i = \frac{Correctly\ classified\ instances\ from\ class\ i}{total\ number\ of\ instances\ in\ class\ i} \tag{21}$$

$$Geometric\ mean = (acc_1 * acc_2 * ... * acc_{10})^{\frac{-1}{10}} \tag{22}$$

VIII. Result and Conclusions

All the results and accuracy for both versions are available in Appendix H. The output of the last convolutional layer for updated version 1 is 120 for MNIST, and Fashion-MNIST dataset, and 480 for CIFAR-10 dataset. The RBFN could achieve accuracy near MLP in the MNIST dataset, but by increasing the number of input data for RBFN in CIFAR-10 and getting the dataset more complex, the RBFN did not perform as well as MLP. This can also be shown in updated version model 2, in which the output size of the convolutional layer is 1600 for MNIST, and Fashion-MNIST, and 2304 for CIFAR-10 datasets. Besides, due to the high dimension of input data for classification, RBFN is so sensitive to the number of K, and the k-means algorithm may easily trap in local optima in the large dataset. Furthermore, when we were increasing the K, the geometric mean accuracy goes up and down. That is may because the accuracy of some classes gots better by increasing the K, but some other accuracy decreased when the number of classification increased.

For this kind of classification that we have, we prefer to use MLP rather than the RBFN, and base on the previous research, it seems that RBFN has a better performance on a dataset that has low features.

IX. Related Work

RBFN is a popular feed-forward network, and some papers like [9] [11] [12] showed better performance for RBFNs or as well as MLPs. For example, Ture et al. [9] compared different classification techniques for predicting essential hypertension. The dataset that they used for comparison have information of 649 patient, and nine features used for classification. RBFN model could get better accuracy and had more sensitivity than the MLP classifier. There are also some papers that tried to proposed evolutionary search algorithms to overcome the local optima problem in RBFN like [13] [14]. The other factor that dependant on the behavior of any neural network is the training dataset. In [10], H sug compared the performance of MLP and RBFN respect to training data size and found out MLPs have better performance for larger data sets, and RBFNs have better performance for smaller data sets.

References

[1] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[2] Broomhead, D. S., Lowe, D. (1988). Radial basis functions, multi-variable functional interpolation and adaptive networks (No. RSRE-MEMO-4148). Royal Signals and Radar Establishment Malvern (United Kingdom).

[3] MacQueen, J. (1967, June). Some methods for classification and analysis of multivariate observations. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability (Vol. 1, No. 14, pp. 281-297).

[4] https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a

[5] "Dataset, MNIST," [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[6] Pratt, L. Y. (1993). Discriminability-based transfer between neural networks. In Advances in neural information processing systems (pp. 204-211).

[7] Xiao, H., Rasul, K., Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

[8] https://www.cs.toronto.edu/ kriz/cifar.html

[9] Ture, M., Kurt, I., Kurum, A. T., Ozdamar, K. (2005). Comparing classification techniques for predicting essential hypertension. Expert Systems with Applications, 29(3), 583-588.

[10] Sug, H. (2009, August). Performance Comparison of RBF networks and MLPs for Classification. In Proceedings of the 9th WSEAS International Conference on applied Informatics and Communications (AIC'09) (pp. 450-454).

[11] Qu, M., Shih, F. Y., Jing, J., Wang, H. (2003). Automatic solar flare detection using MLP, RBF, and SVM. Solar Physics, 217(1), 157-172.

[12] Marinai, S., Gori, M., Soda, G. (2005). Artificial neural networks for document analysis and recognition. IEEE Transactions on pattern analysis and machine intelligence, 27(1), 23-35.

[13] Esposito, A., Marinaro, M., Oricchio, D., Scarpetta, S. (2000). Approximation of continuous and discontinuous mappings by a growing neural RBF-based algorithm. Neural Networks, 13(6), 651-665.

[14] Buchtala, O., Klimek, M., Sick, B. (2005). Evolutionary optimization of radial basis function classifiers for data mining applications. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 35(5), 928-947.

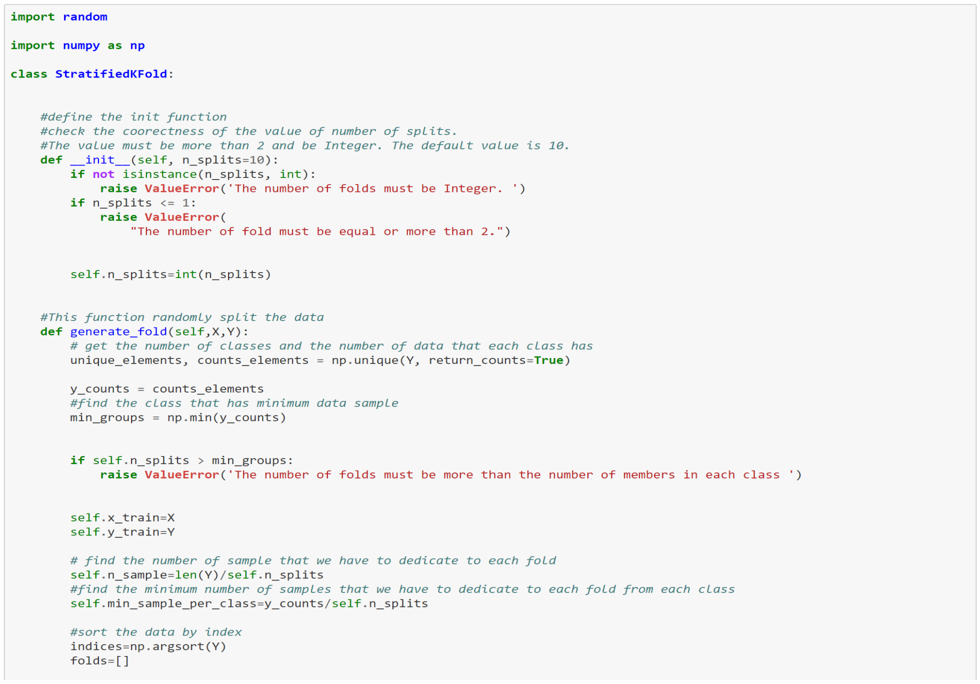
[15] Scherer, D., Müller, A., Behnke, S. (2010, September). Evaluation of pooling operations in convolutional architectures for object recognition. In International conference on artificial neural networks (pp. 92-101). Springer, Berlin, Heidelberg.

[16] https://www.kaggle.com/c/digit-recognizer/discussion/61480

[17] https://www.kaggle.com/c/cifar-10/

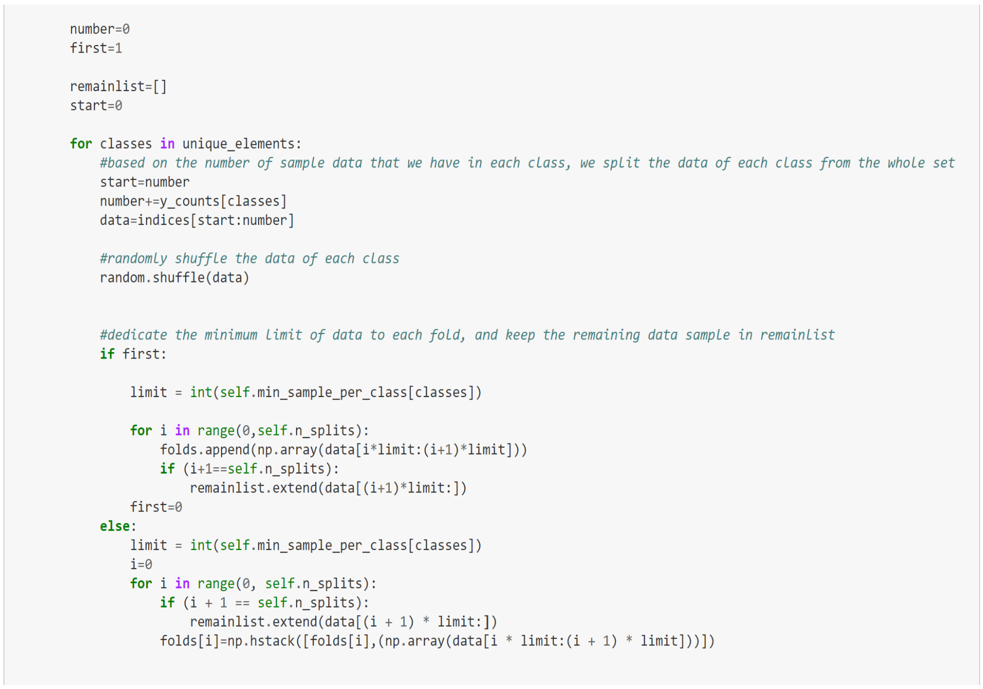
Appendix A
StratifiedKFold class

The three following figure 8,9,10 shows the code of StratifiedKFold class that we used to split the data into 10 folds and apply cross-validation on it (code was implemented in project 1).



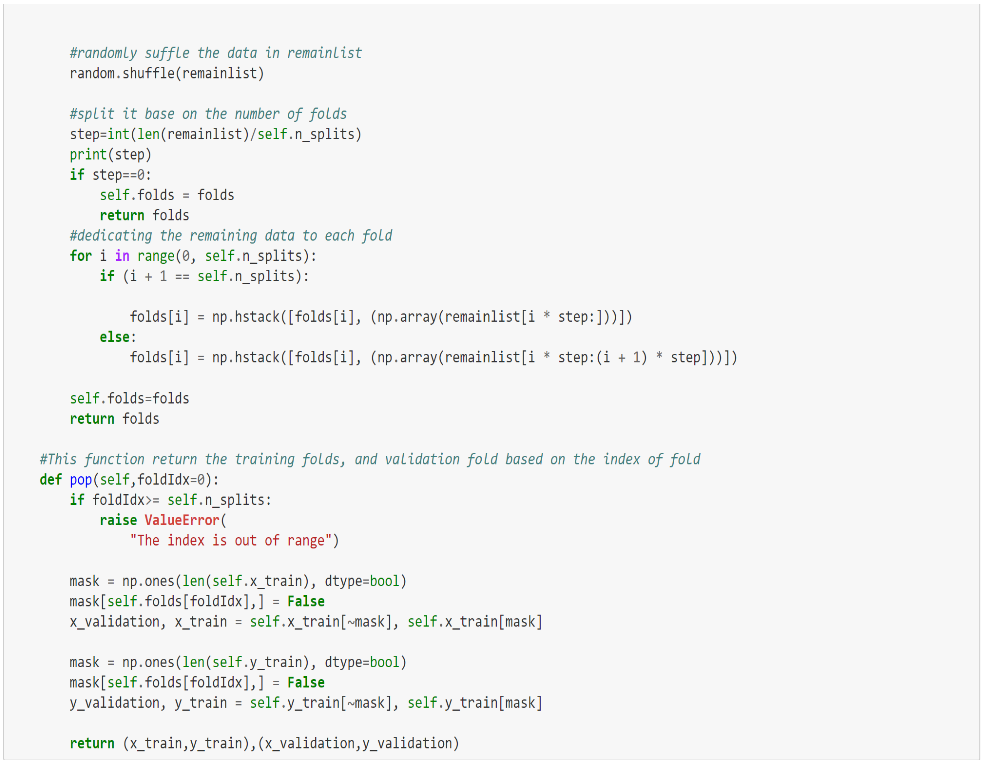
```
import random

import numpy as np

class StratifiedKFold:


    #define the init function
    #check the coorectness of the value of number of splits.
    #The value must be more than 2 and be Integer. The default value is 10.
    def __init__(self, n_splits=10):
        if not isinstance(n_splits, int):
            raise ValueError('The number of folds must be Integer. ')
        if n_splits <= 1:
            raise ValueError(
                "The number of fold must be equal or more than 2.")


        self.n_splits=int(n_splits)


    #This function randomly split the data
    def generate_fold(self,X,Y):
        # get the number of classes and the number of data that each class has
        unique_elements, counts_elements = np.unique(Y, return_counts=True)

        y_counts = counts_elements
        #find the class that has minimum data sample
        min_groups = np.min(y_counts)


        if self.n_splits > min_groups:
            raise ValueError('The number of folds must be more than the number of members in each class ')


        self.x_train=X
        self.y_train=Y

        # find the number of sample that we have to dedicate to each fold
        self.n_sample=len(Y)/self.n_splits
        #find the minimum number of samples that we have to dedicate to each fold from each class
        self.min_sample_per_class=y_counts/self.n_splits

        #sort the data by index
        indices=np.argsort(Y)
        folds=[]
```

Fig. 8. StratifiedKFold class part 1.



```
        number=0
        first=1

        remainlist=[]
        start=0

        for classes in unique_elements:
            #based on the number of sample data that we have in each class, we split the data of each class from the whole set
            start=number
            number+=y_counts[classes]
            data=indices[start:number]

            #randomly shuffle the data of each class
            random.shuffle(data)


            #dedicate the minimum limit of data to each fold, and keep the remaining data sample in remainlist
            if first:

                limit = int(self.min_sample_per_class[classes])

                for i in range(0,self.n_splits):
                    folds.append(np.array(data[i*limit:(i+1)*limit]))
                    if (i+1==self.n_splits):
                        remainlist.extend(data[(i+1)*limit:])
                first=0
            else:
                limit = int(self.min_sample_per_class[classes])
                i=0
                for i in range(0, self.n_splits):
                    if (i + 1 == self.n_splits):
                        remainlist.extend(data[(i + 1) * limit:])
                    folds[i]=np.hstack([folds[i],(np.array(data[i * limit:(i + 1) * limit]))])
```

Fig. 9. StratifiedKFold class part 2.

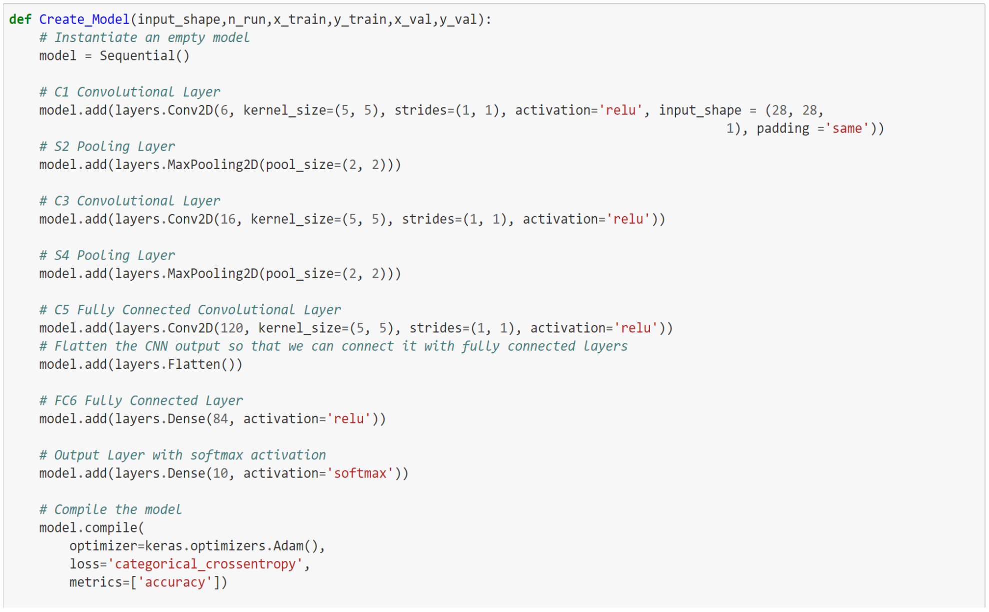


```
        #randomly suffle the data in remainlist
        random.shuffle(remainlist)

        #split it base on the number of folds
        step=int(len(remainlist)/self.n_splits)
        print(step)
        if step==0:
            self.folds = folds
            return folds
        #dedicating the remaining data to each fold
        for i in range(0, self.n_splits):
            if (i + 1 == self.n_splits):

                folds[i] = np.hstack([folds[i], (np.array(remainlist[i * step:]))])
            else:
                folds[i] = np.hstack([folds[i], (np.array(remainlist[i * step:(i + 1) * step]))])

        self.folds=folds
        return folds

    #This function return the training folds, and validation fold based on the index of fold
    def pop(self,foldIdx=0):
        if foldIdx>= self.n_splits:
            raise ValueError(
                "The index is out of range")

        mask = np.ones(len(self.x_train), dtype=bool)
        mask[self.folds[foldIdx],] = False
        x_validation, x_train = self.x_train[~mask], self.x_train[mask]

        mask = np.ones(len(self.y_train), dtype=bool)
        mask[self.folds[foldIdx],] = False
        y_validation, y_train = self.y_train[~mask], self.y_train[mask]

        return (x_train,y_train),(x_validation,y_validation)
```

Fig. 10. StratifiedKFold class part 3

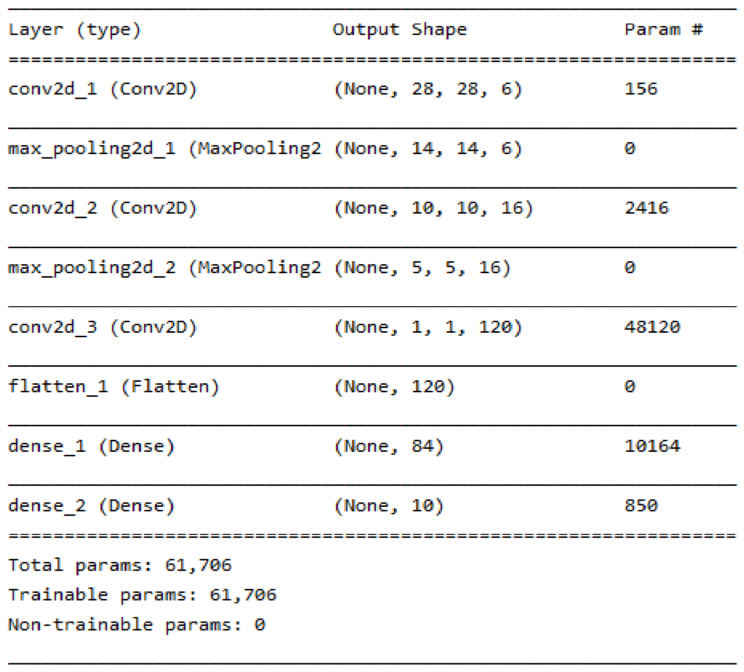
APPENDIX B
UPDATED VERSION 1

Figure 11 shows the implementation of Updated version 1 in python using Keras and Tensorflow library. Figure 12 and 13 show the details and number of trainable parameters for the updated version1 in each dataset.



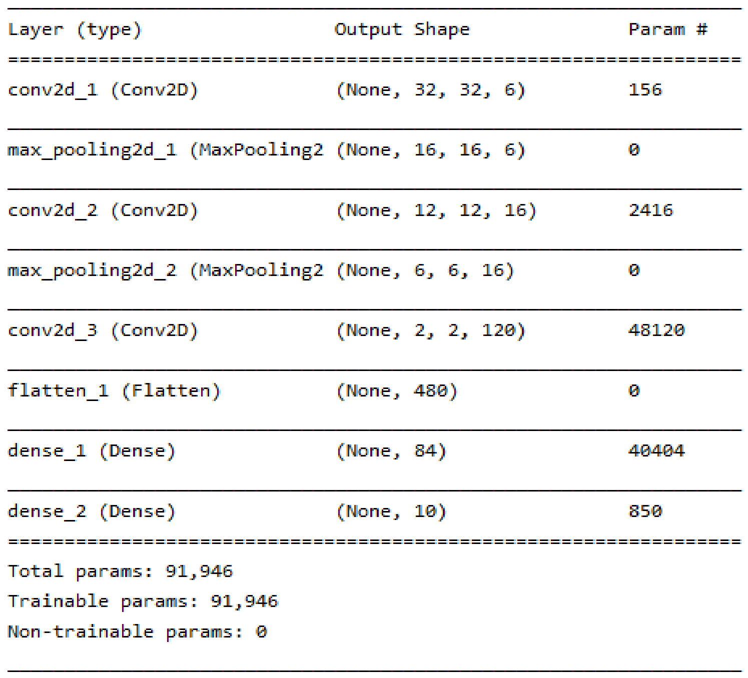
```
def Create_Model(input_shape,n_run,x_train,y_train,x_val,y_val):
    # Instantiate an empty model
    model = Sequential()

    # C1 Convolutional Layer
    model.add(layers.Conv2D(6, kernel_size=(5, 5), strides=(1, 1), activation='relu', input_shape = (28, 28,
                                                                                               1), padding ='same'))
    # S2 Pooling Layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    # C3 Convolutional Layer
    model.add(layers.Conv2D(16, kernel_size=(5, 5), strides=(1, 1), activation='relu'))

    # S4 Pooling Layer
    model.add(layers.MaxPooling2D(pool_size=(2, 2)))

    # C5 Fully Connected Convolutional Layer
    model.add(layers.Conv2D(120, kernel_size=(5, 5), strides=(1, 1), activation='relu'))
    # Flatten the CNN output so that we can connect it with fully connected layers
    model.add(layers.Flatten())

    # FC6 Fully Connected Layer
    model.add(layers.Dense(84, activation='relu'))

    # Output Layer with softmax activation
    model.add(layers.Dense(10, activation='softmax'))

    # Compile the model
    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

Fig. 11. Implementation of updated version 1.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 6)         156
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 6)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 16)        2416
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 1, 1, 120)         48120
_________________________________________________________________
flatten_1 (Flatten)          (None, 120)               0
_________________________________________________________________
dense_1 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
_________________________________________________________________
```

Fig. 12. Architecture of updated version 1 for MNIST and Fashion-MNIST.The input shape is 28*28*1.

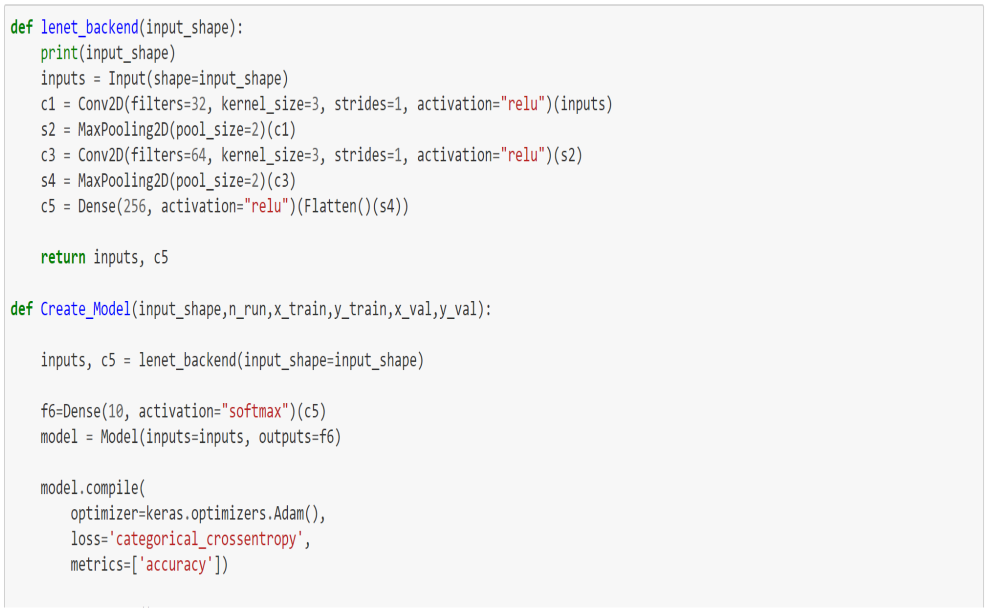


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 6)         156
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 6)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 16)        2416
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 16)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 2, 2, 120)         48120
_________________________________________________________________
flatten_1 (Flatten)          (None, 480)               0
_________________________________________________________________
dense_1 (Dense)              (None, 84)                40404
_________________________________________________________________
dense_2 (Dense)              (None, 10)                850
=================================================================
Total params: 91,946
Trainable params: 91,946
Non-trainable params: 0
_________________________________________________________________
```

Fig. 13. Architecture of updated version 1 for CIFAR-10.The input shape is 32*32*3.

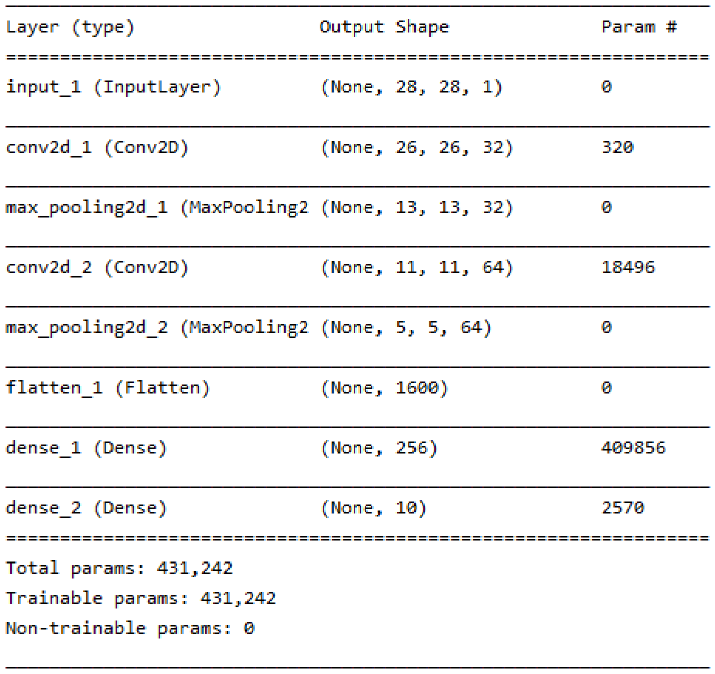
Appendix C
Updated Version 2

Figure 14 shows the implementation of Updated version 2 in python using Keras and Tensorflow library. Figure 15 and 16 show the details and number of trainable parameters for the updated version2 in each dataset.



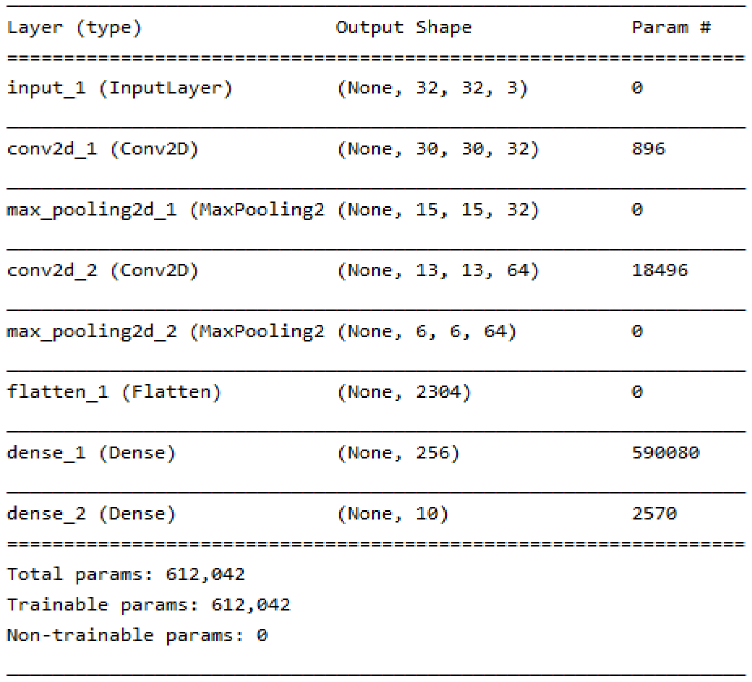
```
def lenet_backend(input_shape):
    print(input_shape)
    inputs = Input(shape=input_shape)
    c1 = Conv2D(filters=32, kernel_size=3, strides=1, activation="relu")(inputs)
    s2 = MaxPooling2D(pool_size=2)(c1)
    c3 = Conv2D(filters=64, kernel_size=3, strides=1, activation="relu")(s2)
    s4 = MaxPooling2D(pool_size=2)(c3)
    c5 = Dense(256, activation="relu")(Flatten()(s4))

    return inputs, c5

def Create_Model(input_shape,n_run,x_train,y_train,x_val,y_val):

    inputs, c5 = lenet_backend(input_shape=input_shape)

    f6=Dense(10, activation="softmax")(c5)
    model = Model(inputs=inputs, outputs=f6)

    model.compile(
        optimizer=keras.optimizers.Adam(),
        loss='categorical_crossentropy',
        metrics=['accuracy'])
```

Fig. 14. Implementation of updated version 1.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 5, 5, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               409856
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 431,242
Trainable params: 431,242
Non-trainable params: 0
_________________________________________________________________
```

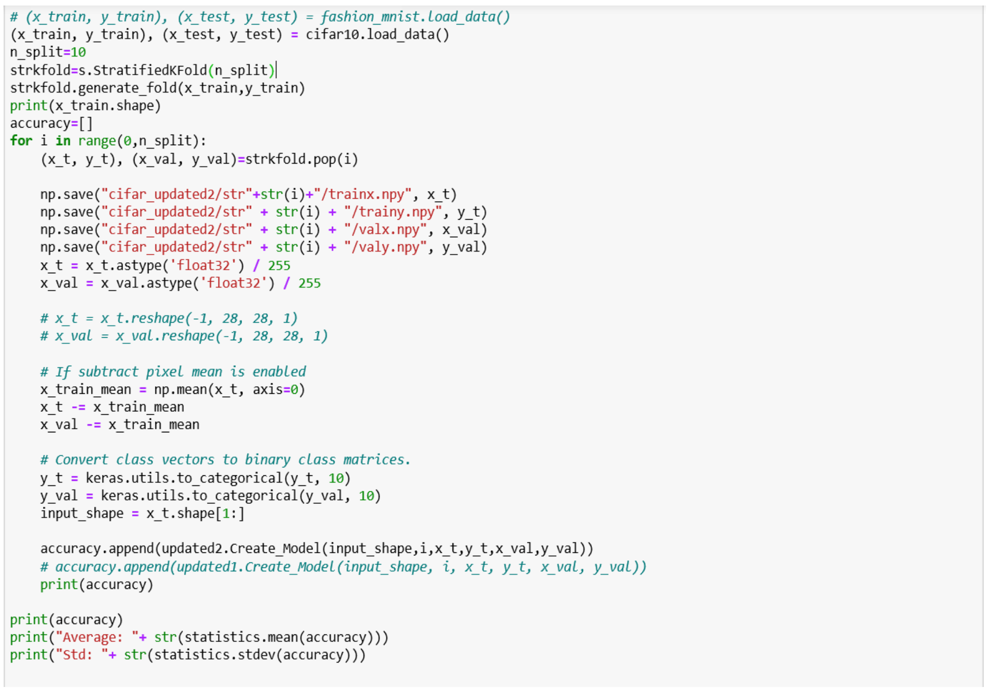
Fig. 15. Architecture of updated version 1 for MNIST and Fashion-MNIST.The input shape is 28*28*1.



```
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 30, 30, 32)        896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               590080
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 612,042
Trainable params: 612,042
Non-trainable params: 0
_________________________________________________________________
```

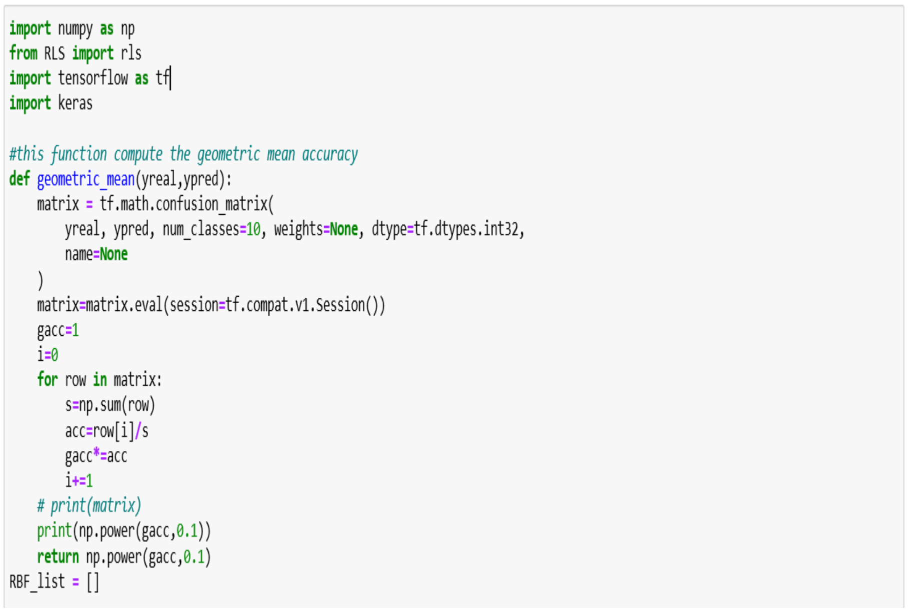
Fig. 16. Architecture of updated version 1 for CIFAR-10.The input shape is 32*32*3.

Appendix D
Data Preprocessing, saving model, adn Geometric Mean accuracy



```
# (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
n_split=10
strkfold=s.StratifiedKFold(n_split)
strkfold.generate_fold(x_train,y_train)
print(x_train.shape)
accuracy=[]
for i in range(0,n_split):
    (x_t, y_t), (x_val, y_val)=strkfold.pop(i)

    np.save("cifar_updated2/str"+str(i)+"/trainx.npy", x_t)
    np.save("cifar_updated2/str" + str(i) + "/trainy.npy", y_t)
    np.save("cifar_updated2/str" + str(i) + "/valx.npy", x_val)
    np.save("cifar_updated2/str" + str(i) + "/valy.npy", y_val)
    x_t = x_t.astype('float32') / 255
    x_val = x_val.astype('float32') / 255

    # x_t = x_t.reshape(-1, 28, 28, 1)
    # x_val = x_val.reshape(-1, 28, 28, 1)

    # If subtract pixel mean is enabled
    x_train_mean = np.mean(x_t, axis=0)
    x_t -= x_train_mean
    x_val -= x_train_mean

    # Convert class vectors to binary class matrices.
    y_t = keras.utils.to_categorical(y_t, 10)
    y_val = keras.utils.to_categorical(y_val, 10)
    input_shape = x_t.shape[1:]

    accuracy.append(updated2.Create_Model(input_shape,i,x_t,y_t,x_val,y_val))
    # accuracy.append(updated1.Create_Model(input_shape, i, x_t, y_t, x_val, y_val))
    print(accuracy)

print(accuracy)
print("Average: "+ str(statistics.mean(accuracy)))
print("Std: "+ str(statistics.stdev(accuracy)))
```

Fig. 17. Code that we used to evaluate each model with stratified and save the data.

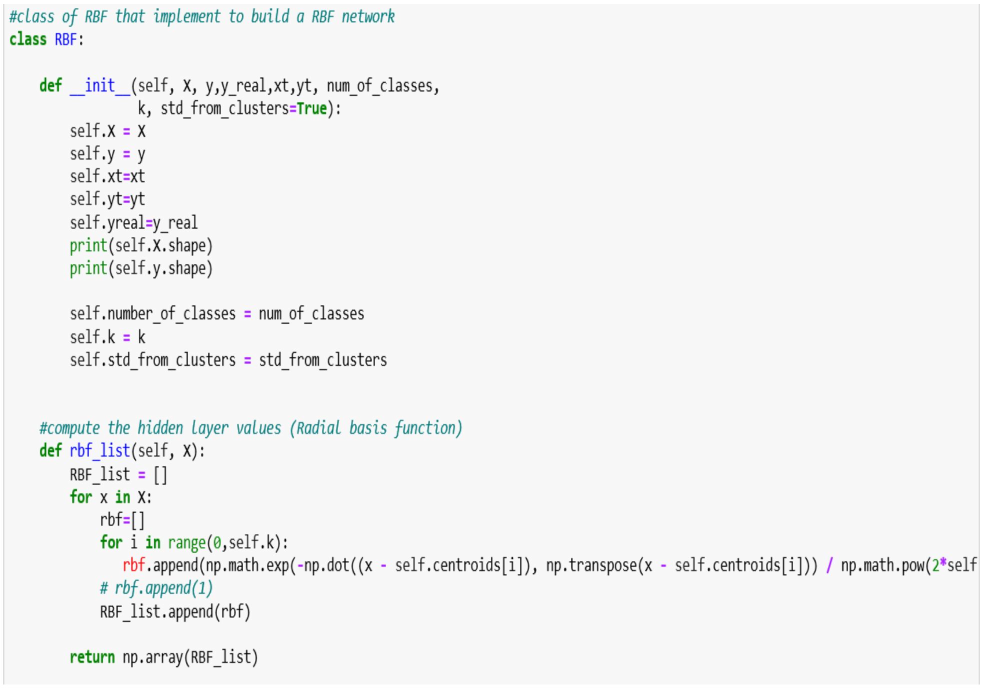


```
import numpy as np
from RLS import rls
import tensorflow as tf
import keras

#this function compute the geometric mean accuracy
def geometric_mean(yreal,ypred):
    matrix = tf.math.confusion_matrix(
        yreal, ypred, num_classes=10, weights=None, dtype=tf.dtypes.int32,
        name=None
    )
    matrix=matrix.eval(session=tf.compat.v1.Session())
    gacc=1
    i=0
    for row in matrix:
        s=np.sum(row)
        acc=row[i]/s
        gacc*=acc
        i+=1
    # print(matrix)
    print(np.power(gacc,0.1))
    return np.power(gacc,0.1)
RBF_list = []
```

Fig. 18. Computation of geometric mean accuracy.

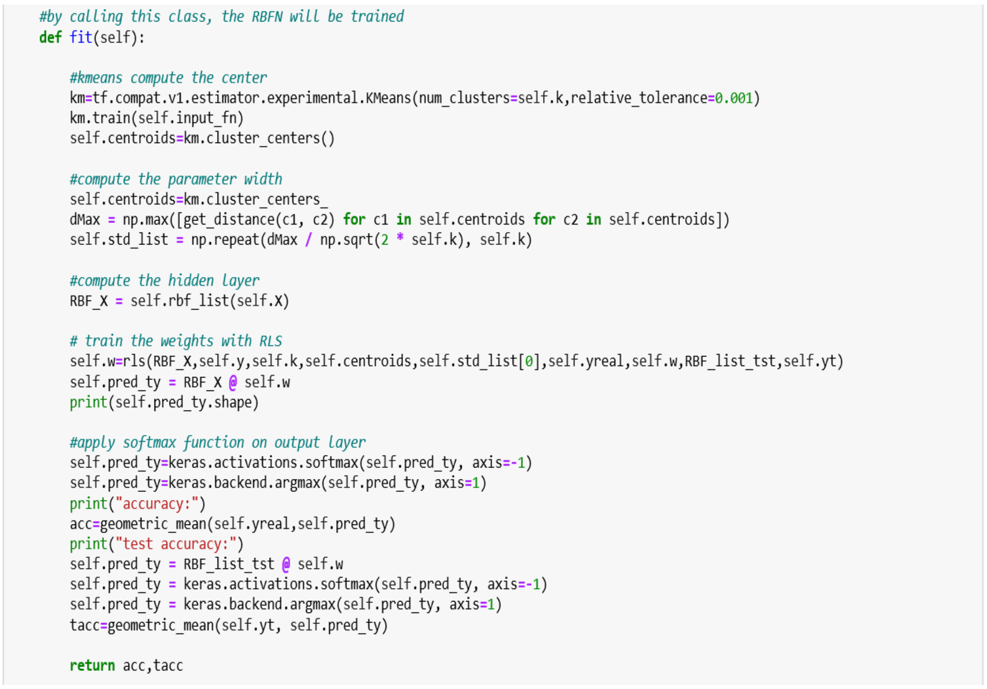
APPENDIX E
RBFN CLASS

Figure 19 and 20 show the class of RBF that we implement to build and train RBF network.



```python
#class of RBF that implement to build a RBF network
class RBF:

    def __init__(self, X, y,y_real,xt,yt, num_of_classes,
                 k, std_from_clusters=True):
        self.X = X
        self.y = y
        self.xt=xt
        self.yt=yt
        self.yreal=y_real
        print(self.X.shape)
        print(self.y.shape)

        self.number_of_classes = num_of_classes
        self.k = k
        self.std_from_clusters = std_from_clusters


    #compute the hidden layer values (Radial basis function)
    def rbf_list(self, X):
        RBF_list = []
        for x in X:
            rbf=[]
            for i in range(0,self.k):
               rbf.append(np.math.exp(-np.dot((x - self.centroids[i]), np.transpose(x - self.centroids[i])) / np.math.pow(2*self
            # rbf.append(1)
            RBF_list.append(rbf)

        return np.array(RBF_list)
```

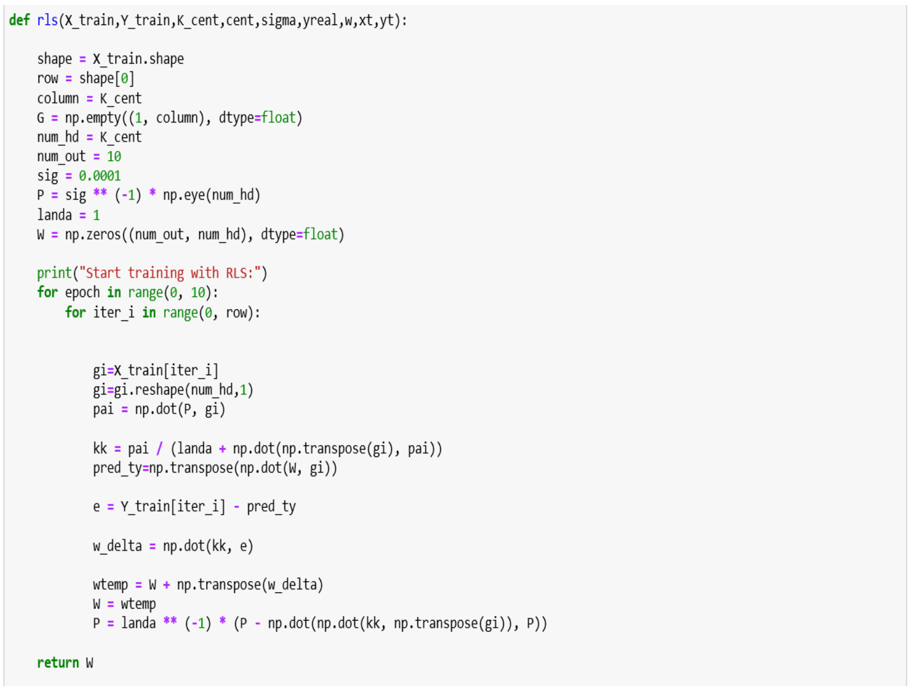
Fig. 19. RBF 1.



```python
    #by calling this class, the RBFN will be trained
    def fit(self):

        #kmeans compute the center
        km=tf.compat.v1.estimator.experimental.KMeans(num_clusters=self.k,relative_tolerance=0.001)
        km.train(self.input_fn)
        self.centroids=km.cluster_centers()

        #compute the parameter width
        self.centroids=km.cluster_centers_
        dMax = np.max([get_distance(c1, c2) for c1 in self.centroids for c2 in self.centroids])
        self.std_list = np.repeat(dMax / np.sqrt(2 * self.k), self.k)

        #compute the hidden layer
        RBF_X = self.rbf_list(self.X)

        # train the weights with RLS
        self.w=rls(RBF_X,self.y,self.k,self.centroids,self.std_list[0],self.yreal,self.w,RBF_list_tst,self.yt)
        self.pred_ty = RBF_X @ self.w
        print(self.pred_ty.shape)

        #apply softmax function on output layer
        self.pred_ty=keras.activations.softmax(self.pred_ty, axis=-1)
        self.pred_ty=keras.backend.argmax(self.pred_ty, axis=1)
        print("accuracy:")
        acc=geometric_mean(self.yreal,self.pred_ty)
        print("test accuracy:")
        self.pred_ty = RBF_list_tst @ self.w
        self.pred_ty = keras.activations.softmax(self.pred_ty, axis=-1)
        self.pred_ty = keras.backend.argmax(self.pred_ty, axis=1)
        tacc=geometric_mean(self.yt, self.pred_ty)

        return acc,tacc
```

Fig. 20. RBF 2.

Appendix F
Implementation of RLS

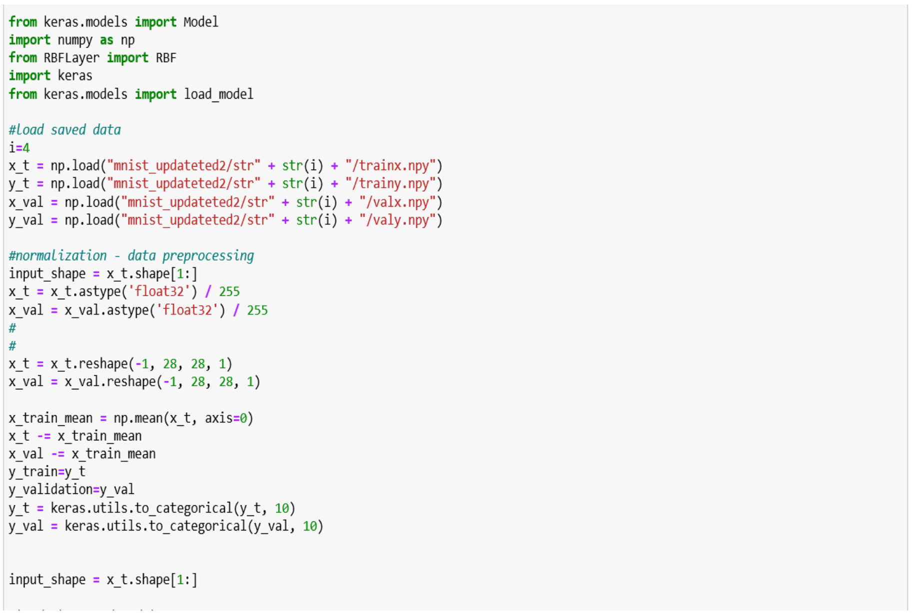


```python
def rls(X_train,Y_train,K_cent,cent,sigma,yreal,w,xt,yt):

    shape = X_train.shape
    row = shape[0]
    column = K_cent
    G = np.empty((1, column), dtype=float)
    num_hd = K_cent
    num_out = 10
    sig = 0.0001
    P = sig ** (-1) * np.eye(num_hd)
    landa = 1
    W = np.zeros((num_out, num_hd), dtype=float)

    print("Start training with RLS:")
    for epoch in range(0, 10):
        for iter_i in range(0, row):


            gi=X_train[iter_i]
            gi=gi.reshape(num_hd,1)
            pai = np.dot(P, gi)

            kk = pai / (landa + np.dot(np.transpose(gi), pai))
            pred_ty=np.transpose(np.dot(W, gi))

            e = Y_train[iter_i] - pred_ty

            w_delta = np.dot(kk, e)

            wtemp = W + np.transpose(w_delta)
            W = wtemp
            P = landa ** (-1) * (P - np.dot(np.dot(kk, np.transpose(gi)), P))

    return W
```

Fig. 21. Recursive Least Square implementationn.

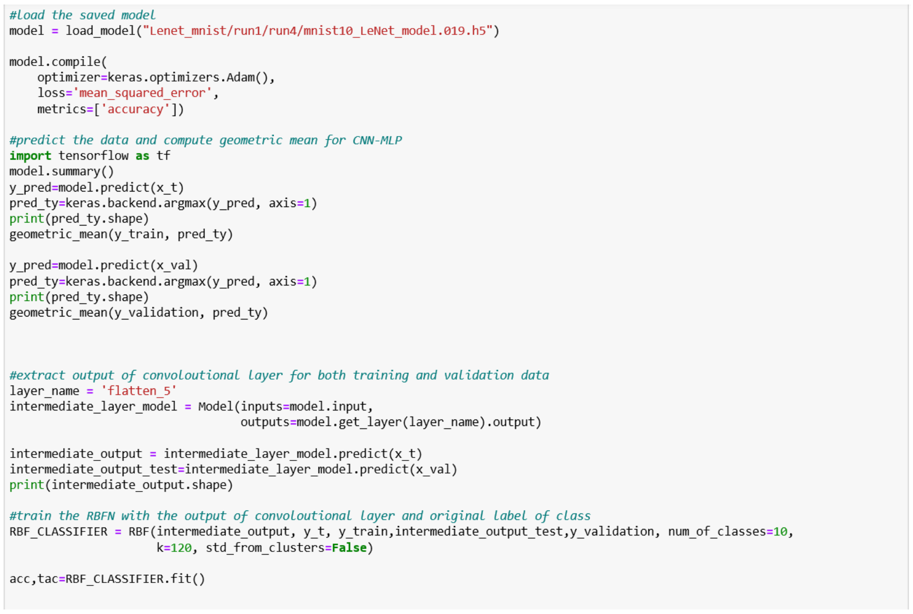
Appendix G
Transfer Learning

Figure 22 and 23 show how we load the model and data, then extract the output of convolutional layer and pass it to RBFN.



```python
from keras.models import Model
import numpy as np
from RBFLayer import RBF
import keras
from keras.models import load_model

#load saved data
i=4
x_t = np.load("mnist_updateted2/str" + str(i) + "/trainx.npy")
y_t = np.load("mnist_updateted2/str" + str(i) + "/trainy.npy")
x_val = np.load("mnist_updateted2/str" + str(i) + "/valx.npy")
y_val = np.load("mnist_updateted2/str" + str(i) + "/valy.npy")

#normalization - data preprocessing
input_shape = x_t.shape[1:]
x_t = x_t.astype('float32') / 255
x_val = x_val.astype('float32') / 255
#
#
x_t = x_t.reshape(-1, 28, 28, 1)
x_val = x_val.reshape(-1, 28, 28, 1)

x_train_mean = np.mean(x_t, axis=0)
x_t -= x_train_mean
x_val -= x_train_mean
y_train=y_t
y_validation=y_val
y_t = keras.utils.to_categorical(y_t, 10)
y_val = keras.utils.to_categorical(y_val, 10)


input_shape = x_t.shape[1:]
```

Fig. 22. Transfer Learning 1.

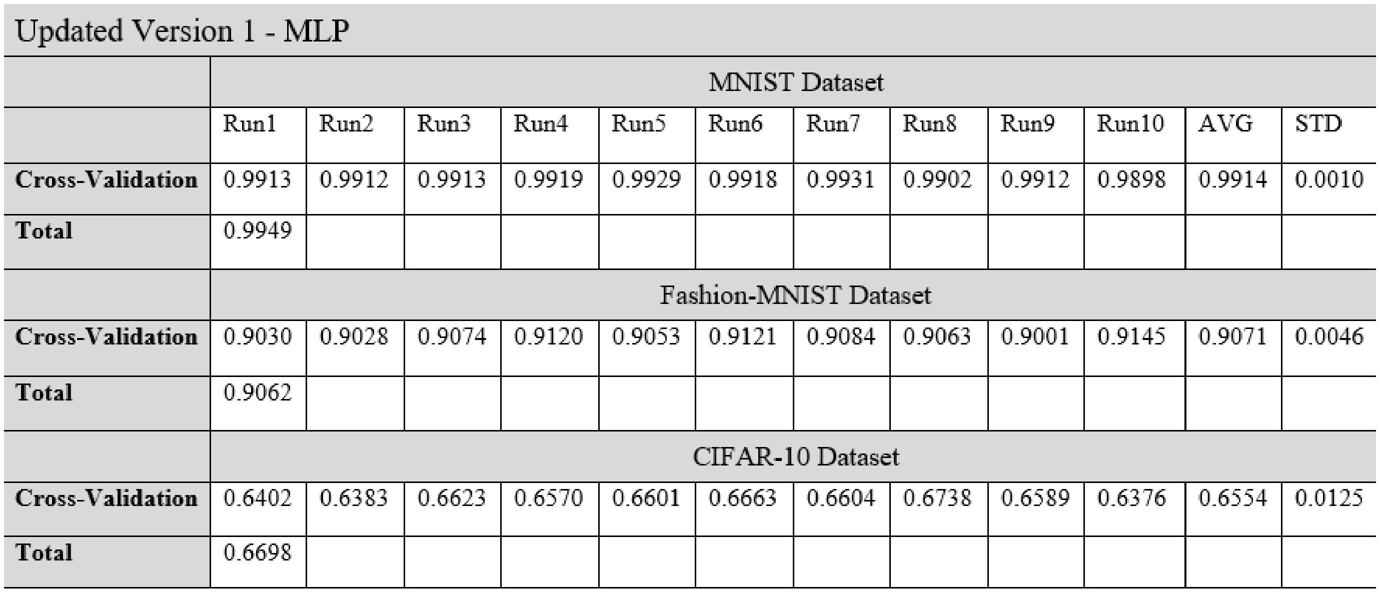


```
#load the saved model
model = load_model("Lenet_mnist/run1/run4/mnist10_LeNet_model.019.h5")

model.compile(
    optimizer=keras.optimizers.Adam(),
    loss='mean_squared_error',
    metrics=['accuracy'])

#predict the data and compute geometric mean for CNN-MLP
import tensorflow as tf
model.summary()
y_pred=model.predict(x_t)
pred_ty=keras.backend.argmax(y_pred, axis=1)
print(pred_ty.shape)
geometric_mean(y_train, pred_ty)

y_pred=model.predict(x_val)
pred_ty=keras.backend.argmax(y_pred, axis=1)
print(pred_ty.shape)
geometric_mean(y_validation, pred_ty)



#extract output of convoloutional layer for both training and validation data
layer_name = 'flatten_5'
intermediate_layer_model = Model(inputs=model.input,
                                 outputs=model.get_layer(layer_name).output)

intermediate_output = intermediate_layer_model.predict(x_t)
intermediate_output_test=intermediate_layer_model.predict(x_val)
print(intermediate_output.shape)

#train the RBFN with the output of convoloutional layer and original label of class
RBF_CLASSIFIER = RBF(intermediate_output, y_t, y_train,intermediate_output_test,y_validation, num_of_classes=10,
                     k=120, std_from_clusters=False)

acc,tac=RBF_CLASSIFIER.fit()
```

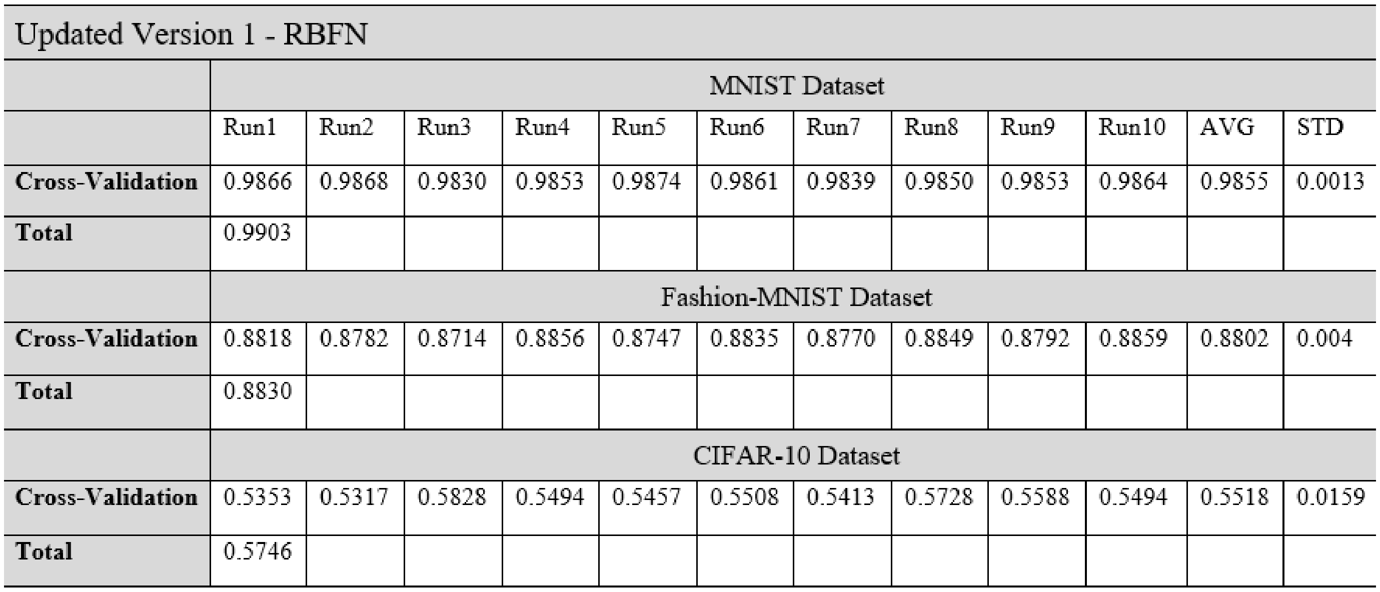
Fig. 23. Transfer Learning 2.

Appendix H
Results

Figure 24 shows the result of updated version 1 for MLP and RBFN, and Figure 25 shows the result of updated version 2 for MLP and RBFN.

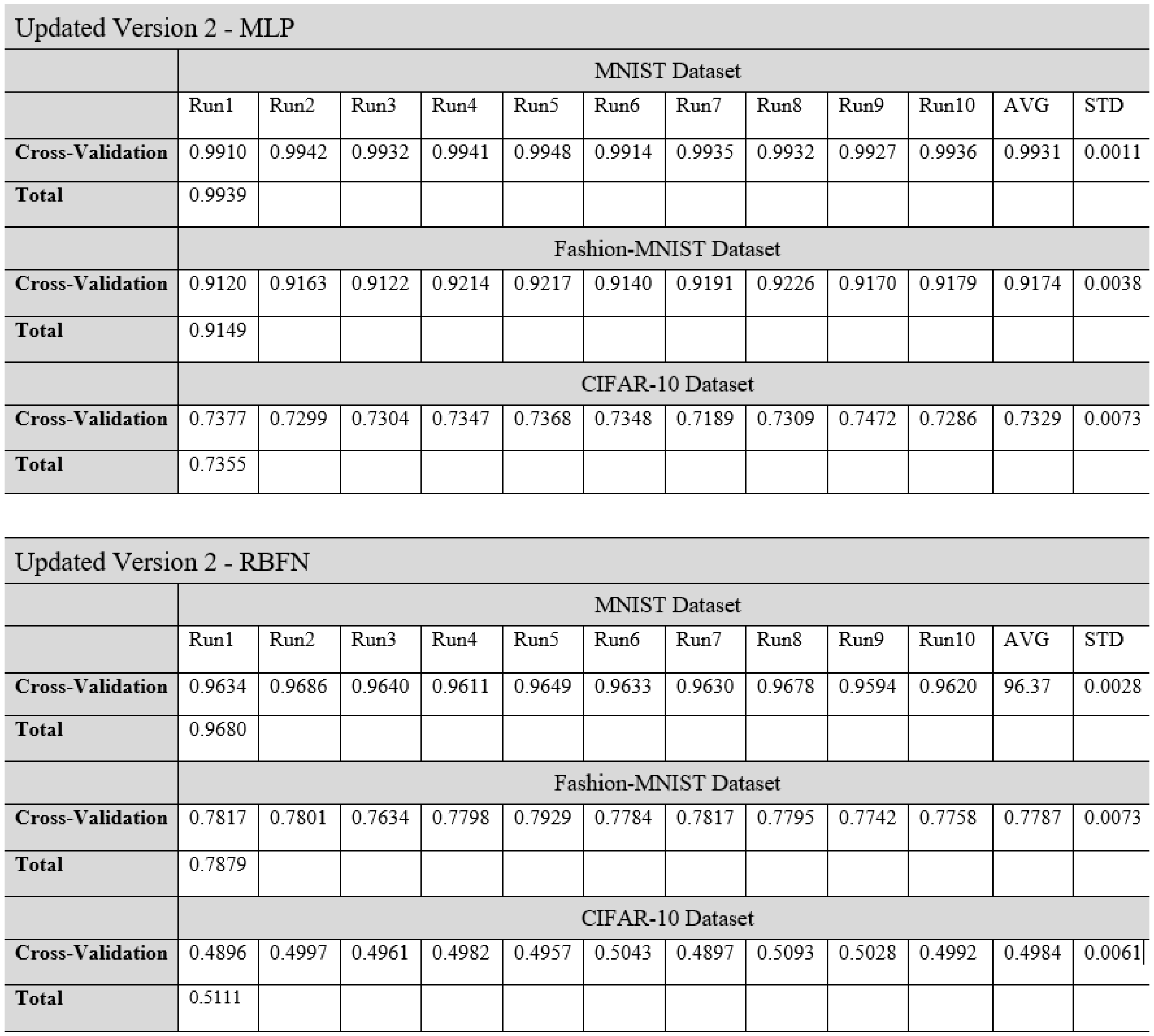


Updated Version 1 - MLP												
	MNIST Dataset											
	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	AVG	STD
Cross-Validation	0.9913	0.9912	0.9913	0.9919	0.9929	0.9918	0.9931	0.9902	0.9912	0.9898	0.9914	0.0010
Total	0.9949											
	Fashion-MNIST Dataset											
Cross-Validation	0.9030	0.9028	0.9074	0.9120	0.9053	0.9121	0.9084	0.9063	0.9001	0.9145	0.9071	0.0046
Total	0.9062											
	CIFAR-10 Dataset											
Cross-Validation	0.6402	0.6383	0.6623	0.6570	0.6601	0.6663	0.6604	0.6738	0.6589	0.6376	0.6554	0.0125
Total	0.6698											



Updated Version 1 - RBFN												
	MNIST Dataset											
	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	AVG	STD
Cross-Validation	0.9866	0.9868	0.9830	0.9853	0.9874	0.9861	0.9839	0.9850	0.9853	0.9864	0.9855	0.0013
Total	0.9903											
	Fashion-MNIST Dataset											
Cross-Validation	0.8818	0.8782	0.8714	0.8856	0.8747	0.8835	0.8770	0.8849	0.8792	0.8859	0.8802	0.004
Total	0.8830											
	CIFAR-10 Dataset											
Cross-Validation	0.5353	0.5317	0.5828	0.5494	0.5457	0.5508	0.5413	0.5728	0.5588	0.5494	0.5518	0.0159
Total	0.5746											

Fig. 24. Geometric accuracy for validation and test data (updated version 1).



Updated Version 2 - MLP

	MNIST Dataset											
	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	AVG	STD
Cross-Validation	0.9910	0.9942	0.9932	0.9941	0.9948	0.9914	0.9935	0.9932	0.9927	0.9936	0.9931	0.0011
Total	0.9939											
	Fashion-MNIST Dataset											
Cross-Validation	0.9120	0.9163	0.9122	0.9214	0.9217	0.9140	0.9191	0.9226	0.9170	0.9179	0.9174	0.0038
Total	0.9149											
	CIFAR-10 Dataset											
Cross-Validation	0.7377	0.7299	0.7304	0.7347	0.7368	0.7348	0.7189	0.7309	0.7472	0.7286	0.7329	0.0073
Total	0.7355											

Updated Version 2 - RBFN

	MNIST Dataset											
	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	AVG	STD
Cross-Validation	0.9634	0.9686	0.9640	0.9611	0.9649	0.9633	0.9630	0.9678	0.9594	0.9620	96.37	0.0028
Total	0.9680											
	Fashion-MNIST Dataset											
Cross-Validation	0.7817	0.7801	0.7634	0.7798	0.7929	0.7784	0.7817	0.7795	0.7742	0.7758	0.7787	0.0073
Total	0.7879											
	CIFAR-10 Dataset											
Cross-Validation	0.4896	0.4997	0.4961	0.4982	0.4957	0.5043	0.4897	0.5093	0.5028	0.4992	0.4984	0.0061
Total	0.5111											

Fig. 25. Geometric accuracy for validation and test data (updated version 1).