Neural Network Homework (1)

SOM, SLFN

Instructed by Dr.Mansoori

Reza Tahmasebi

saeed77t@gmail.com, Stu ID: 40160957

Table of Contents

1 Read And Preprocess Data

1.1 Define the root folder of the dataset and two empty lists to store the image data and labels.

This section of the code sets the variable '*root_folder'* to the directory path where the MNIST dataset is stored on disk. The empty '*lists'* data and labels will be used later to store the image data and corresponding labels, respectively.

1.2 Loop over each of the 10 folders in the root folder and extract image data and labels.

This section of the code loops over each of the 10 folders in the '*root_folder'*, where each folder contains images of one digit (0-9). For each folder, the code gets the label corresponding to the folder name (i.e., the digit) and loops over the images in the folder.

1.3 Read each image using OpenCV and convert it to grayscale.

For each image, the code reads the image using OpenCV and converts it to grayscale using the cv2.imread() and cv2.IMREAD_GRAYSCALE functions, respectively. The resulting grayscale image is then added to the data list, and the corresponding label is added to the labels list.

1.4 Convert the lists of image data and labels to NumPy arrays.

After all the images have been read and labeled, the data and label lists are converted to NumPy arrays using the '*np. array()*' function. The resulting data and label arrays are used in the next steps to split the data into training and testing sets.

1.5 Split the data into training and testing sets.

This section of the code uses the '*train_test_split()*' function from scikit-learn to split the image data and labels into training and testing sets separately for each digit class (0-9). The split is done so that 90% of each class is used for training and 10% is used for testing.

1.6 Concatenate the training and testing data and labels for each class into separate NumPy arrays.

After splitting the data into training and testing sets for each class, the resulting arrays are concatenated into larger arrays for both the training and testing data and labels. This is done using the '*np. concatenate()*' function.

1.7 Create two pandas data frames to store the training and testing data and labels.

This section of the code creates two pandas data frames, '*train_df*' and *'test_df'*, to store the training and testing data and labels, respectively. Each data frame has one column for the flattened image data and another column for the corresponding label.

1.8 Shuffle the rows of the training and testing data frames randomly.

Finally, the code shuffles the rows of both the *train_df* and *test_df* data frames randomly using the *sample()* function with *frac=1*. This random shuffling is usually done to avoid any bias in the order of the data during training.

2 SOM Clustering

2.1 Winner-Takes-All approach

2.1.1 Load and prepare the data

The first step of the code is to load the training and test data and their respective labels. The training and test data are flattened and normalized by dividing each pixel value by 255.

2.1.2 Define the SOM parameters

The Self-Organizing Map (SOM) algorithm is used to create a map with one neuron for each class. The parameters for the SOM are defined, including the input length (number of features), number of classes, size of the SOM, neighborhood radius, and learning rate.

2.1.3 Implement the winner-takes-all approach

The winner-takes-all approach is used to classify the data into 10 clusters. The SOM is initialized with the PCA weights of the training data, and it is trained on the training data for 300,000 iterations. The number of images of each class that are placed in each cluster is determined, and the label for each cluster is determined by finding the class that has the highest count in the cluster. The final class labels are determined by assigning the class with the highest count to each cluster.

2.1.4 Compute train and test accuracies

The train and test accuracies are computed using the final class labels. For each training and test sample, the closest neuron is found, and its label is assigned to the sample. The accuracy is computed by comparing the predicted labels with the actual labels.

2.1.5 Compute the Davies-Bouldin Index

The Davies-Bouldin Index (DBI) is computed for the SOM, which is a measure of the clustering quality. The Euclidean distances between each training sample and each neuron in the SOM are computed, and each training sample is assigned to the neuron it is closest to. The DBI is then computed using the assignments.

2.1.6 Print the cluster labels

Finally, the cluster labels are printed, which are the class labels assigned to each cluster using the winner-takes-all approach.

2.1.7 Results

The reported results indicate the performance of the Winner-Takes-All (WTA) approach on a dataset using a Self-Organizing Map (SOM) with 10 clusters.

- The SOM is trained using 300,000 iterations with a neighborhood radius of 1 and a learning rate of 0.5.
- The quantization error is reported as 5.3166, which measures the average distance between each training sample and its closest neuron in the SOM.
- The training accuracy is reported as 4.51%, indicating that only a small fraction of the training samples are correctly classified by the SOM.
- The test accuracy is reported as 5.15%, which is even lower than the training accuracy. This suggests that the model is overfitting the training data and not generalizing well to new, unseen data.
- The Davies-Bouldin index (DBI) is computed as 2.9089, which measures the quality of the clustering by comparing the average distance between each neuron and its closest neighbors to the

average distance between each neuron and the centroid of its assigned cluster. The higher the DBI, the worse the clustering quality.

- The cluster labels are reported as [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.], which indicates that each cluster is assigned to a distinct class label.

2.1.8 Possible ways to improve the method include:

Increasing the number of clusters in the SOM: By increasing the number of clusters in the SOM, we can create a more fine-grained representation of the data, which may improve the model's accuracy and clustering quality.

Using a different neighborhood function: The WTA approach uses a Gaussian neighborhood function to update the weights of the neurons during training. Experimenting with different neighborhood functions, such as a bubble or a triangle, may lead to better performance.

Using a different learning rate schedule: The learning rate determines the amount by which the weights of the neurons are updated during training. Using a different learning rate schedule, such as decreasing the learning rate over time, may help the model converge to a better solution.

Adding regularization: Regularization techniques, such as weight decay or dropout, can help prevent overfitting and improve the model's generalization performance.

Using a different type of SOM: There are different types of SOMs, such as Growing Neural Gas (GNG) or Growing Hierarchical SOM (GHSOM), that may be more suitable for certain types of data and tasks. Experimenting with different SOM types may lead to better performance.

2.2 On-Center, Off-Surround approach

The first step in this code is to load the training and test data along with their respective labels. This is done by assigning the *TrainData* and *TestData* to the *training_data* and *test_data* variables, and *TrainLabels* and *TestLabels* to the *training_labels* and *test_labels* variables, respectively.

Title: Building a Self-Organizing Map (SOM)

The code then builds a SOM with 30x30 neurons using the MiniSom package. The SOM is initialized with the dimensions of the input data, a sigma of 1.0, a learning rate of 0.5, a neighborhood function of "bubble", and a rectangular topology.

Title: Training the SOM

The SOM is trained using the training_data with a loop that iterates 1000 times. The training data is randomized and the train_random method is called with a num_iteration of 1 for each iteration. The tqdm package is used to show a progress bar during the training process.

2.2.1 Getting cluster labels for training data

After training the SOM, the cluster labels for each training data point are obtained by finding the winner neuron for each data point and assigning it to the corresponding cluster. For each cluster, the indices of the training data points belonging to the cluster are found, and their corresponding labels are extracted. The most frequent label among the points in the cluster is then assigned as the label for the entire cluster.

2.2.2 Getting the number of images of each class in each cluster

The number of images of each class in each cluster is then obtained. For each class, the cluster class counts are obtained for each of the 30x30 clusters by iterating through all the training data points and checking if they belong to the cluster. The corresponding label is extracted and counted. The counts are then stored in a list of lists with the outer list containing the counts for each class and the inner lists containing the counts for each cluster.

2.2.3 Assigning a label to each cluster based on the class with the highest count

The label for each cluster is then assigned by choosing the class with the highest count among the images in the cluster.

2.2.4 Computing Davies-Bouldin Index on the clustering result

The Davies-Bouldin index is computed on the clustering result to evaluate the quality of the clustering. The *davies_bouldin_score* method from the *sklearn* package is used for this purpose.

2.2.5 Getting test data accuracy

The test data accuracy is obtained by finding the cluster for each test data point and assigning the label of the class with the highest count among the images in the cluster as the label for the test data point. The accuracy is then calculated by comparing the predicted labels with the actual test labels.

2.2.6 Getting training data accuracy

The training data accuracy is obtained similarly by comparing the predicted labels with the actual training labels.

2.2.7 Methods

This work has been done for 30×30 and 20×20 neurons with neighborhood topology.

2.2.8 Results

The purpose of this code is to build a Self-Organizing Map (SOM) for image clustering and classification. The code utilizes the *MiniSom* package to build SOM with 30x30 and 20x20 neurons. The SOM is trained with a dataset of training images and then used to classify both training and test images.

The results of the code are presented below:
For SOM with 30x30 neurons:

- Davies-Bouldin index: 4.0326
- Test data accuracy: 90.7 %5
- Training data accuracy: 88.18 %

For SOM with 20x20 neurons:

- Davies-Bouldin index: 4.04131
- Test data accuracy: 85%
- Training data accuracy: 84.5%

The Davies-Bouldin index is a measure of the clustering quality, where lower values indicate better clustering. The results show that the SOM with 30x30 neurons has a slightly better clustering quality compared to the SOM with 20x20 neurons, with a Davies-Bouldin index of 4.03 and 4.04 respectively.

The accuracy of the classification on both training and test datasets is also presented. The results show that the SOM with 30x30 neurons performs better on both training and test datasets, with accuracy values of 0.881 and 0.907 respectively. The SOM with 20x20 neurons has lower accuracy values on both training and test datasets, with accuracy values of 0.845 and 0.855 respectively.

In conclusion, the results suggest that the SOM with 30x30 neurons is better suited for image clustering and classification than the SOM with 20x20 neurons.

So the new datasets made out of the results of the 30x30 neurons SOM to use for the SLFN classifier.

3 SLFN Classification

This classifier has been implemented in two ways, both will be explain here,

3.1 SLFN with *MLPClassifier*

The SLFNClassifier class is a wrapper around the *MLPClassifier* class from *sklearn.neural_network* module. *MLPClassifier* stands for Multi-Layer Perceptron classifier, which is a type of neural network that can be used for classification tasks. The *SLFNClassifier* class provides an easier-to-use interface for creating and training an MLP classifier.

The *__init__* method of the *SLFNClassifier* class initializes an instance of *MLPClassifier* with the hyperparameters passed as arguments. *hidden_layer_sizes* is a tuple that specifies the number of neurons in each hidden layer. activation specifies the activation function to be used in the neurons. solver specifies the optimization algorithm used to train the neural network. alpha is the regularization parameter for L2 regularization. *random_state* is a seed value used for random number generation to ensure reproducibility.

The fit method of the *SLFNClassifier* class simply calls the fit method of the *MLPClassifier* instance with the training data and labels as arguments.

The predict method of the *SLFNClassifier* class calls the prediction method of the *MLPClassifier* instance to predict the class labels of the test data.

The score method of the *SLFNClassifier* class calls the scoring method of the *MLPClassifier* instance to calculate the accuracy of the model on the test data.

The next part of the code splits the dataset into training and test sets using the *train_test_split* function from *sklearn.model_selection module*. data and label are assumed to be the feature and target matrices respectively.

An instance of the *SLFNClassifier* class is created, and the fit method is called with the training data and labels as arguments to train the model.

The *predict* method of the trained model is then used to predict the labels of the test data, and the *score* method is called to calculate the accuracy of the model on the test data.

Finally, the accuracy is printed as a percentage using the print function.

In summary, the code demonstrates the usage of the *SLFNClassifier* class to create and train a neural network model for classification tasks, and to evaluate the model's accuracy on a test set. The hyperparameters of the model can be adjusted as needed to improve its performance on different datasets.

3.2 SLFN with PyTorch

3.2.1 SLFNClassifier class and its methods

The code defines a *PyTorch* neural network class named *SLFNClassifier*, which is used for classification tasks. This class is defined by inheriting the *nn.Module* class from the *PyTorch* library. The class contains three methods: ***init***, *fit*, and *predict*.

3.2.2 *The init* method of *SLFNClassifier*

The ***init*** method of the *SLFNClassifier* class initializes the model's parameters, including the number of input, hidden, and output units. The model consists of two linear layers. The first layer transforms the input features to a hidden representation using a linear transformation followed by the *Tanh* activation function. The second layer transforms the hidden representation to the output layer using a linear transformation without any activation function.

3.2.3 *fit* method of *SLFNClassifier*

The *fit* method of the *SLFNClassifier* class trains the model using the input training data and their corresponding labels. The method converts the data and labels to *PyTorch* tensors and uses stochastic gradient descent (SGD) as the optimizer and cross-entropy loss as the loss function. The method updates the model parameters by backpropagating the error through the network and optimizing the loss function.

3.2.4 *predict* method of *SLFNClassifier*

The *predict* method of the *SLFNClassifier* class uses the trained model to make predictions on new data. The method converts the new data to *PyTorch* tensors and passes them through the trained model. The output of the model is then used to make the final predictions.

3.3 Results

In this study, we have evaluated two different methods for classification on ten different datasets. Method one is a single-layer feed-forward network (SLFN) with the *MLPClassifier*, while method two is an SLFN with *PyTorch*. The evaluation metric used for comparison was accuracy.

From the results, it is clear that method two (SLFN with *PyTorch*) performed better than method one (SLFN with *MLPClassifier*) on eight out of the ten datasets. The improvement in accuracy ranged from 0.27% to 13.23%. However, on two datasets, method one performed better than method two with an improvement in accuracy of 1.33% and 1.17%, respectively.

The datasets used in this study varied in terms of size, complexity, and type of features. Therefore, the results suggest that *PyTorch*, as a deep learning library, may be more effective than *MLPClassifier*, a traditional machine learning algorithm, in handling complex and large datasets.

Result

Dataset	**Method 1 Accuracy**	**Method 2 Accuracy**
0	81.94%	84.17%
1	80.5%	79.17%
2	83.8%	88.8%
3	83.61%	89.17%
4	79.1%	93.13%
5	79.16%	93%
6	78.8%	92.2%
7	76.9%	87.7%
8	80.83%	88.6%
9	81.6%	90%