National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Software Development for Algorithmic Problems

Image Similarity Search and Clustering in Reduced Space

Project implemented by :

Nikolaos Galanis - sdi1700019
Sofoklis Strompolas - sdi1700153

Contents

Abstract

The goal of this project is to create several executables in order to test in various ways specific images for their similarity. We are going to implement:

- An image auto-encoder model, with bottleneck architecture, that is then used in order to convert images into a lower space representation.
- A comparison of the similarity search between the images in the original and the reduced space.
- The Earth Mover's Distance metric, and its comparison to the Manhattan distance, to conclude which one performs the best while searching for similar images.
- Perform clustering on the original, the reduced space, and the classifier of the previous project, in order to determine which method provides best classifying results.

The auto-encoder model consists of two different types of layers: the **encoding** and the **decoding** layers. The program created is a handy interface in order for the user to insert different values of several hyper-parameters and see the behavior of each model, with ultimate goal to chose one model that best handles the dataset given.

The classification model aims to classifying images in a category. To do so, it uses a pre-trained auto-encoder model, by taking advantage of its encoding layers, which are then connected to a fully connected layer, and then to an output one, aiming to the best possible classification of the images. Once again, multiple models can be trained, in order for the user to decide the best for the training set needs, and then, the best one will be used in order to predict the test dataset.

Directories Structure and Files

```
/
├── Autoencoder ..... Autoencoder Implementation
│   ├── classifier_py ..... Used to classify images for q4
│   ├── decoder_py ..... Used by the autoencoder
│   ├── encoder_py ..... Used by the autoencoder
│   └── reduce_py ..... Autoencoder that produces reduced dim images, q1
├── misc ..... Miscellaneous files
│   ├── original_space ..... Datasets in the original dimension (784)
│   ├── reduced_space ..... Datasets in the reduced dimension (10)
│   ├── classes_for_clustering ..... Input file for q4, containing the clusters
│   └── classes_for_clustering_small . Cluster classes for the smaller datasets
├── Models ..... Models used by q1 and q2
├── NearestNeighbour_EMD ..... EMD metric files
│   ├── brute_force_py ..... Py implementation of BF class from pr1
│   ├── EMD_py ..... Implementation of the EMD metric
│   ├── parse_flat_py ..... Parser
│   └── run_knn_py ..... Main function of q3
└── Similarity_Search ..... Clustering, LSH files
```

Note: The structure of the Similarity Search Directory is the same as in the first project.

1 Autoencoder and Dimension Reducer (q1)

1.1 Running

In order to run the Autoencoder model and create the reduced space datasets, you should navigate to the directory Autoencoder, and run the file *autoencoder.py*, as following:

```
python3 reduce.py -d <inp dataset> -q <inp queryset>
-od <out dataset> -oq <out queryset>
```

1.2 Implementation

The autoencoder receives a file containing the MNIST dataset, and tries to apply auto-encoding for the images of the dataset, based on the hyperparameters given by the user. The program will train the models and plot the loss function and the accuracy results. The user also has an option to save the model for later use.

The autoencoder consists of an encoder and a decoder, which are located on the *encoder.py* and *decoder.py* file respectively.

The encoder contains a number of convolution blocks, and each block has a convolution layer followed by a a batch normalization layer. A Max-pooling layer is also used after the first and the second convolution blocks. Additionally we added a dropout layer too, to prevent over-fitting. We finally reduce the neurons by providing a fully connected layer consisted of 10 neurons, which represents the latent dimension.

The decoder also contains a number of convolution blocks that contain a convolution later followed by a batch normalization layer. An Up-sampling layer is used after the last two convolution blocks and a final layer reconstructing back the input in a single channel.

First of all the program parses the arguments given by the user, and reads the appropriate train_set file. In order to train the model, the training data must be split into a training set and a validation set. Then, we define the input shape (the shape of the input neurons), and we create an empty list that aims on storing data about the models.

After this process, the program asks the user about the hyper-parameters in order to train a model.

- Give the number of convolution layers
- Give the size of each convolution filter

```
- Give the number of convolution filters per layer
- Give the number of epochs
- Give the batch size
```

The program then will try to create an autoencoder model with these hyperparameters, and upon successful creation it trains it and adds it to the models list.

The optimizer we used was RMSprop with the best arguments we could find.

Following that, the program will pop out a menu, looking like the following:

```
Experiment completed! Choose one of the following options to proceed:

        1 - Repeat the experiment with different hyperparameters
        2 - Print the plots gathered from the experiment
        3 - Save the model and exit
        4 - Load a pre-trained model
```

If the user chooses the first option, the program asks for the hyper-parameters in order to tune a new model. Those hyper-parameters are the ones mentioned above(number of convolution layers, size of each convolution filter, Give the number of convolution filters per layer, Give the number of epochs, Give the batch size). Then the model is defined using those parameters, and the training procedures begin.

If the user chooses the second option, the program:

- Plots the comparison between training and validation losses and accuracy for the last trained model, shows them and saves those plots in the appropriate directory.
- Uses the HiPlot library for high-dimension plotting in order to plot the comparison of error between all the previously trained models, and then saves those plots in the appropriate directory.

After selecting option 2, the program continues as before, and re-queries the user for a new option.

If the user chooses the third option, the program performs the reducing as following:

- Cuts all the layers after the latent, in order for the model to produce a 10-valued vector as a result, which will be then used as our new image.
- Normalizes the dataset and the queryset.
- Predicts both of the sets, thus we get lists of 10-number vectors.
- Normalizes the produced values, in order to fit in 2 bits, thus the numbers of each pixel are ranged from 0 to 65535.

- Creates 2 new binary files, that will be used to store the new datasets
- Prints the necessary data in the new files, and then each byte of each image
- Saves the files and finishes the execution.

If the user chooses the fourth option, the program asks for a pre-trained model in order to be loaded. It queries the name of the saved h5 file, as well as the hyper-parameters used to train this model, in order for the model to be successfully added to the models' list. After that we train the model for 1 epoch in order to take the loss and accuracy, he program continues as before, and re-queries the user for a new option.

1.3 Inputs

The auto-encoder takes the train and the test datasets as input, in order to reduce the dimension of their images.

The parsing of the program is based on the inputs of the MNIST dataset, and thus, all the dataset must follow the rules of MNIST.

1.4 Outputs

The program, depending on the user's choice gives a specific type of output. Those outputs are:

- The training procedure and epochs
- The plots
- The new datasets

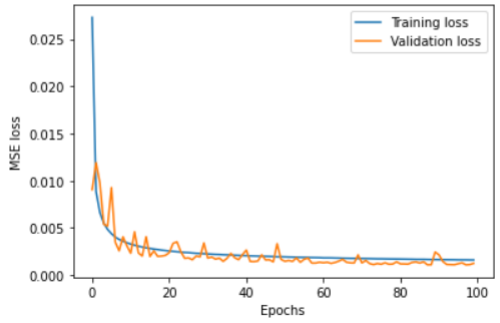
1.5 Results and plots

Having previous knowledge about the behaviour of the autoencoder, we used our best model from the second project in order to create the reduced dataset. The metrics of this model, are the following:

The best autoencoder model consists of **4 convolution layers** with **filter size 3** and **number of filters per layer 32**, **100 epochs** and **86 batch size**.

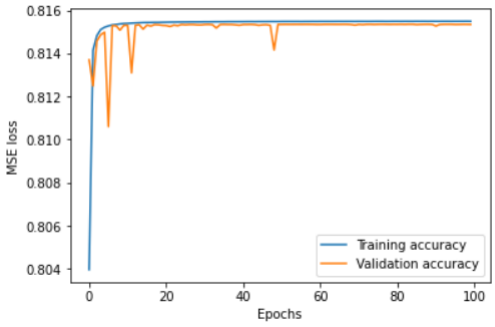
Train vs Validation Loss during the training

```
------------------TRAINED MODEL METRICS----------------------
Training/validation loss iconv_layers: 4 conv_filter_size: 3 n_conv_filters_per_layer: 32 epochs: 100 batch_size: 86
```



Train vs Validation Accuracy during the training

```
Training/validation accuracy iconv_layers: 4 conv_filter_size: 3 n_conv_filters_per_layer: 32 epochs: 100 batch_size: 86
```



2 Similarity Search with LSH and BF (q2)

2.1 Running

In order to run the Similarity Search algorithm, you should navigate to the directory Similarity Search and run the following command:

```
make run_lsh
```

By default, the datasets used are of reduced size, in order for the programs to run quickly. Shall you want to change to the original files, you should run the command:

```
make run_lsh_big
```

Note: The files (reduced and regular) should be of the same size, otherwise the program is going to fail.

Note: Our lower space dimension files are produced with the pixels as little endians, for our convenience. Shall you want to run the program, you should provide the files created by us, or others with little endianness.

2.2 Implementation

The algorithms are the same as the ones of the first assignment, with some small differences:

- **Changes in Main function**: We added the necessary class instants in order to support th lower dimension vectors.
- **Parser**: We changed the parser so when the user provides the reduced space files, 2 bytes are read for each pixel, as we were instructed.

Due to our generic implementation of the first project, when we embedded templates and generic functions, we did not make any further changes to the LSH class, as everything worked like a charm.

2.3 Inputs

The inputs are all the datasets in the reduced and the original space, that follow the rules of the MNIST binary files. There is an option for the user to run the smaller files, all of which are located in subdirs of the misc directory. The files are of the following shape:

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

2.4 Outputs

After the run of the LSH executable, an output file is generated, that includes each query and its metrics. After the program is completed, it prints to the standard output the additive results of the metrics. An example output after the execution of a query is the following:

```
Query: 457
Nearest neighbor Reduced: 2320
Nearest neighbor LSH: 62
Nearest neighbor True: 62
DistanceReduced:17048
DistanceLSH:12573
DistanceTrue:12573
```

The output file for the execution of the small dataset can be found under the directory path: Similarity Search/executables/lsh/lsh_out.
The output file for the execution of the big dataset can be found under the directory path: Similarity Search/executables/lsh/lsh_out_big.

2.5 Results and Observations

After the execution of our program with the small datasets (5000 features, 1000 queries), the results are the following:

```
tLSH: 0.00465098
tBF: 0.00770454
tRED: 0.00023193
Approximation Factor LSH: 1.01099
Approximation Factor Reduced: 1.48022
```

After the execution of our program with the regular datasets (6000 features, 10000 queries), the results are the following:

```
tLSH: 0.00208225
tBF: 0.081551
tRED: 0.00261187
Approximation Factor LSH: 1.22624
Approximation Factor Reduced: 1.26434
```

We observe the following:

- The approximation factor of the LSH is very small, and the average time very close to the brute force, because we run the query for only one nearest neighbour, which is almost every time correct. We also had a rather big w parameter, that was used in the first assignment, and is now reduced.
- The approximation factor of the reduced space brute force algorithm is very good compared to other results that we saw on the class forum.

3 Similarity Search with the EMD metric (q3)

3.1 Running

In order to run the Similarity Search with EMD, you should navigate to the directory NearestNeighbour_EMD, and then run the following command:

```
python3 search.py -d <trainingset> -q <testset> -l1 <training labels> -l2 <test labels>
```

Note1: The files should be in the original space

Note2: You should first install **pulp**

Note3: Because the datasets are big, we use 1000 features and 10 queries since the EMD takes a lot of time. We can change it again in lines 25-30.

Note4: We have results for 10 queries-1000 features and 100 queries-1000 features for 2x2 ie 4 clusters, 4x4 ie 16 clusters and 7x7 ie 49 clusters.

3.2 Implementation

We implemented the Earth Mover's Distance (EMD) metric, and its comparison to the Manhattan distance, to conclude which one performs the best while searching for similar images.

First we parse the parse the dataset, queryset and their labels. Since the images are a lot and the EMD complex, we use 1000 feature and 10 query images and we experimented with 1000 feature and 100 query images too.(The numbers can be changed in lines 25-30 in the parse functions)

Then we initialize the EMD class with the number of clusters we want. We experimented with 49 ,16 and 4.(can be changed in line 34 of the search.py file).

The EMD class, which is located in the EMD.py, contains the implementation of the EMD function. For the EMD function we followed step by step the notes. We used the pulp functions and solver and managed to find a solution that represents the EMD metric with Linear Programming, by Defining the variables, the constraints, objective function and results.

Afterwards we initialize the Bruteforce classes, which are located in the brute_force.py, and have the function to perform the k nearest neighbour search with brute force either with Manhattan Distance or EMD.

In the end we parse the queries, and for each one we search for the 10 nearest neighbours in all the feature images, using Manhattan metric and the EMD metric, and showing their accuracy based on the labels.

3.3 Inputs

The inputs are the train and test files , and their labels in the original space.

3.4 Outputs

The outputs of the program are of the shape:

```
Average Correct Search Results EMD: <double>

Average Correct Search Results MANHATTAN: <double>
```

3.5 Results and Observations

Our final results after experimented with different sizes for query-feature images. We run the [10 queries 1000 features] and [100 queries 1000 features]. Also we tested for 4, 16, 49 clusters.

```
2x2 (4 clusters)

    10 queries 1000 features
        Average Correct Search Results EMD:   36.0 %
        Average Correct Search Results MANHATTAN:   70.0 %

        10 minutes

    100 queries 1000 features
        Average Correct Search Results EMD:   30.5 %
        Average Correct Search Results MANHATTAN:   71.7 %

        100 minutes
```

```
4x4 (16 clusters)

    10 queries 1000 features
        Average Correct Search Results EMD:   57.0 %
        Average Correct Search Results MANHATTAN:   70.0 %

        16 minutes

    100 queries 1000 features
        Average Correct Search Results EMD:   59.7 %
        Average Correct Search Results MANHATTAN:   71.7 %

        160 minutes
```

```
7x7 (49 clusters)

    10 queries 1000 features
        Average Correct Search Results EMD:   81.9 %
        Average Correct Search Results MANHATTAN:   70.0 %
```

```
    57 minutes

100 queries 1000 features
    Average Correct Search Results EMD:  80.1 %
    Average Correct Search Results MANHATTAN:  71.7 %


    570 minutes
```

We tested with 1000 feature images, since the duration of the execution was taking too long with more.

As we can see, the more clusters we have, the more accurate the results, but the program takes significantly longer to execute. Just for the 10 queries the 7x7(49 clusters) took almost an hour in our PC, but the results were good since the average correct result was at 81.9%.

When taking 100 queries, we can observe with higher accuracy the results of the Correct Search Results EMD.

As a final observation, the EMD metric takes significantly more time to calculate, than the Manhattan metric. As for the accuracy, with little clusters the EMD does not perform so well in comparison to Manhattan, but with more clusters the calculations become more accurate(but the duration much higher).

Also as a small note, we noticed that the EMD was performing more accurately in images with less active(not 0) pixels, since the variables are less, it is easier to evaluate the correct one.

4 Clustering on Reduced and Original Spaces (q4)

4.1 Running

In order to create the clustering file from the classifier, you should navigate to the Autoencoder directory, and then run the following command:

```
python3 classifier.py -d <trainingset> -dl <training
labels> -t <testset> -tl <test labels> -model <autoencoder h5>
```

A file produced from our best classifying model is stored in misc, and is used by our makefile in the clustering program.

In order to run the Clustering algorithm, you should navigate to the directory Sim-

ilarity Search and run the following command:

```
make run_cluster_lloyds
```

By default, the datasets used are of reduced size, in order for the programs to run quickly. Shall you want to change to the original files, you should run the command:

```
make run_cluster_lloyds_big
```

Note: The files (reduced, regular, NN) should be of the same size, otherwise the program is going to fail.

4.2 Implementation

The clustering algorithms and their module are similar to the first project, with subject to some small changes in order to support the reduced space files, as well as the clustering file produced from the NN classifier.
We added some functions in the Clustering class, in order to make our module clean, and easy to use. More specifically, we have added:

- **Parser function** in order to parse the NN input file, and update the assigned centroid for each vector, because this info is being obtained from the input file.
- **Constructor for the NN clustering**. This constructor calls the parser, and then a single time the update function, in order for each centroid to be the median of all the vectors assigned to it.
- **Constructor for the reduced space clustering**. Despite the use of reduced space vectors, the metrics must be computed in the original space. Thus, if that is the case, our constructor initializes some for class fields, where we are keeping all the original space info, such as the dimension and the feature vectors. Then, the clustering algorithm proceeds normally.
- **Update for the reduced function**. This function is used before the computation of the silhouette, in order to create the regular dimension centroids, so that the comparison with the other methods is mafde possible.
- **Tweaks in the clustering algorithm**. If we are in the reduced space, after the clustering is complete, and before the user calls the metric functions, we must substitute the lower dimension info with the original ones.
- **Objective function**. A function that is used to compute the objective function results for each method.

Finally, changes have been made in the main function, that now enables the user to run all three of the different methods of clustering.

4.3 Inputs

The inputs are all the datasets in the reduced and the original space, as well as the classes created from the NN. There is an option for the user to run the smaller files, all of which are located in subdirs of the misc directory.

4.4 Outputs

The program follows the output that we were instructed.

The output of the program execution with the small files, is the following:

```
NEW SPACE
CLUSTER-1 {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
.     .     .     .     .     .     .
CLUSTER-K {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
clustering_time: <double> //in seconds
Silhouette: [s1,...,si,...,sK, stotal]
Value of Objective Function: <double>

ORIGINAL SPACE
CLUSTER-1 {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
.     .     .     .     .     .     .
CLUSTER-K {size: <int>, centroid: πίνακας με τις συντεταγμένες του centroid}
clustering_time: <double> //in seconds
Silhouette: [s1,...,si,...,sK, stotal]
Value of Objective Function: <double>

CLASSES AS CLUSTERS
Silhouette: [s1,...,si,...,sK, stotal]
Value of Objective Function: <double>
```

.

The output file for the execution of the small dataset can be found under the directory path: Similarity Search/executables/clustering/cluter_lloyds_out.

4.5 Results and Observations

While running the small datasets, we get the following results on our metric functions:

```
NEW SPACE
clustering_time: 0.243078
```

```
Silhouette: [0.0511437, -0.0384216, 0.0608401, 0.219795,
0.245333, -0.0196761, 0.135963, 0.248805, 0.283629, 0.295446
, 0.12125]
Value of Objective Function: 1.63423e+08
```

ORIGINAL SPACE

```
Silhouette: [0.40278, 0.180677, 0.448619, -0.296002, -0.184209,
0.334357, 0.101489, 0.240733, 0.448307, 0.0597763, , 0.0922905]

Value of Objective Function: 1.62802e+08
```

CLASSES AS CLUSTERS

```
Silhouette: [0.0664643, 0.334842, 0.0403352, 0.0150693, -0.0258254
, 0.0854457, 0.113966, 0.0109816, 0.0707882, 0.113254, , 0.0848895]

Value of Objective Function: 1.61299e+08
```

Besides from the basic observations, meaning the execution time of the clustering that is obviously smaller in the reduced space, we observe some interesting results when it comes down to the metrics.

The best clustering seems to be achieved from the new, reduced space, mainly due to the high accuracy of our autoencoding model. However, all the implementations are rather close when it comes down to the value of the objective function, as well as their silhouette. Noting that all the calculation were made in the original space, as we were instructed.