Alma Mater Studiorum – University of Bologna

Discriminately Boosted Clustering

Artificial Intelligence in Industry (2021-2022)

Author : Zarmina Ursino - zarmina.ursino@studio.unibo.it

Source code available : https://github.com/Zarmina97/DBC_Ai_industry

Summary

Abstract

Deep clustering is a new research direction that combines deep learning and clustering. It performs feature representation and cluster assignments simultaneously, and its clustering performance is significantly superior to traditional clustering algorithms. We observe that existing deep clustering algorithms either do not well take advantage of convolutional neural networks or do not considerably preserve the local structure of data generating distribution in the learned feature space. To address this issue, I propose a deep convolutional embedded clustering algorithm in this paper. Specifically, I develop a convolutional autoencoders structure to learn embedded features in an end-to-end way.

Introduction

The dataset was provided to perform “Discriminately Boosting Clustering” (DBC) . The MARCONI100 computing system installed at Cineca in early 2020 is the largest supercomputer available in Academic sector in Italy and in Europe today. It is powered by IBM Power9 processors and NVIDIA Volta V100 GPUs, employing dual-rail Mellanox EDR InfiniBand as the system network. The data was collected with a tool called Examon and the dataset is composed of several folders, a folder for each selected node.

Data exploration

The information monitored on Marconi100's nodes is varied, ranging from the load of the different cores, to the temperature of the room where the nodes are located, the speed of the fans, details on memory accesses in writing/reading, etc.The sampling rate of the data at the source varies between 5 and 10 seconds.

However, in the data set the data are aggregated in 15-minutes intervals; in particular, the mean value ("avg: <metric_name>") and variance ("var: <metric_name>") are computed over each 15-minute interval. In the CSVs, each row corresponds to a different timestamp (first column on the left), therefore separated by intervals of 5 minutes.

The column called "New Label" column indicates the presence or absence of a failure on the node. After loading the data, I drop the ‘label’ and ‘timestamp’ columns from the dataset. I check for any missing data to treat the missing values. There are no missing values in the dataset.

I split the dataset into X and y for further model training and evaluation of the model. Here we can observe that there are two values in y: ‘0’ means normal state of the node, ‘2’ means anomalous state. So, I map all ‘2’ values with ‘1’ value for our convenience in evaluation part.

We can see that class 0 has 97,52% of entries and class 1 has only 2.48% as It’s possible to see in Figure 1. There is lot of imbalance in the dataset. K-means is sensitive to the scale of feature values because it uses Euclidean distance as similarity metrics. For this reason, I scale these features using Minmax scaler.

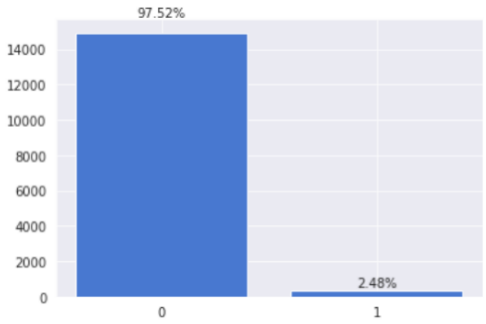

Figure 1

Convolutional Autoencoders

Data preparation

The input array passed to the CNN should be a 4D array, with a shape of *(batch_size, height, width, depth)*, where the first dimension represents the batch size of the image and the other three dimensions represent dimensions of the image which are height, width, and depth. To do this I used the NumPy function *expand_dims()*, to expand the shape. Afterwards I splitted X and target y into *X_train*, *X_test*, *y_train* and *y_test* by applying the sklearn method '*train_test_split()*', with a size of the test set of 20%.

Moreover, since there is no batch size value in the input_shape argument, we could go with any batch size while fitting the data; thus, I set input array as *Input(shape=(460, 1, 1))*

Autoencoder Model

The framework contains two parts. It's possible to see in See Figure 2 a glance of the overall framework and in Figure 3 the Algorithm applied.

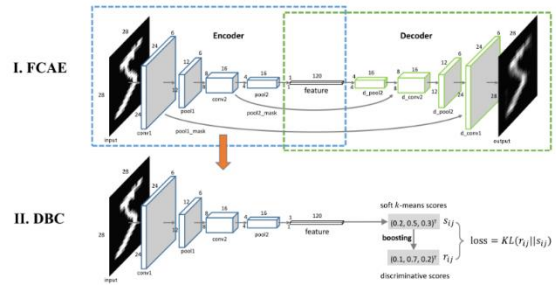

Figure 2

Algorithm 1 Discriminatively Boosted Clustering (DBC)

Require: X, T, B, m_b, α, k

Ensure: θ and μ

//**Stage I: Train a FCAE and clustering with its features**

1: Train a deep fully convolutional auto-encoder

$$x_i \xrightarrow{\theta_e} z_i(\text{features}) \xrightarrow{\theta_d} \hat{x}_i \qquad \text{(M1)}$$

with the Euclidian loss

$$(\theta_e^*, \theta_d^*) = \arg\min_{\theta_e,\, \theta_d} \sum_{i=1}^{m} ||x_i - \hat{x}_i||_2^2 = \sum_{i=1}^{m} ||x_i - g_{\theta_d}(f_{\theta_e}(x_i))||_2^2$$

by using the traditional error back-propagation algorithm.

2: Extract features: $Z \leftarrow f_{\theta_e^*}(X)$

3: Clustering with the features: $\mu_z \leftarrow k$-means centers

Figure 3

First, I implement a **fully convolutional** auto-encoder (FCAE) which is composed of convolution-type layers (convolution and de-convolution layers) and pool-type layers (pooling and un-pooling layers). I adopt convolution layers along with max-pooling layers to make a fully convolutional encoder (FCE). Since the down-sampling operations in the FCE reduce the size of the output feature maps, I use an unpooling layer to recover the feature maps. As a result, the unpooling layers along with de-convolution layers are adopted to make a fully convolutional decoder (FCD)

The overall architecture is **symmetric** around the feature layer. In practice, it is suggested to design layers of an odd number. Otherwise, it will be ambiguous to define the feature layer.

The depth of the whole network grows in log-magnitude as the input size increases. This could make the network very deep if the original image has a very large width or height. To overcome this problem, I adopt the batch **normalization** (BN) strategy for reducing the internal covariate shift and speeding up the training. The BN operation is performed after each convolutional layer and each deconvolutional layer except for the last output layer. This avoids a tedious and time-consuming layer-wise pretraining stage adopted in the traditional stacked (convolutional) autoencoders. To the best of our knowledge, this is the first attempt to learn a deep auto-encoder in an end-to-end manner.

The optimizer used is *'adam*' and the type of loss is the the Mean Squared Error ('mse').

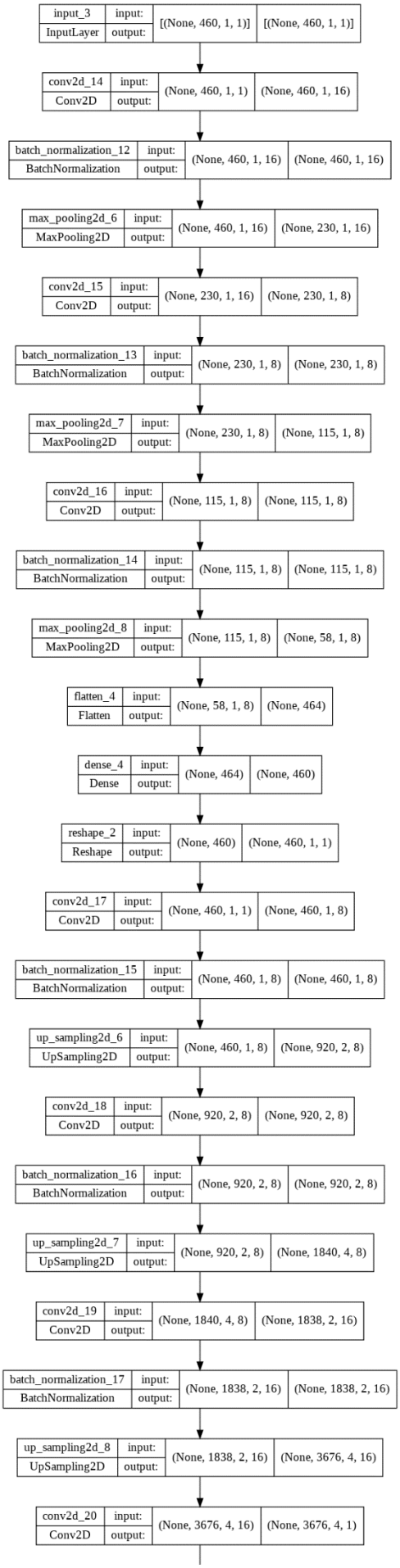
input_3 | InputLayer | input: | output: | [(None, 460, 1, 1)] | [(None, 460, 1, 1)]
conv2d_14 | Conv2D | input: | output: | (None, 460, 1, 1) | (None, 460, 1, 16)
batch_normalization_12 | BatchNormalization | input: | output: | (None, 460, 1, 16) | (None, 460, 1, 16)
max_pooling2d_6 | MaxPooling2D | input: | output: | (None, 460, 1, 16) | (None, 230, 1, 16)
conv2d_15 | Conv2D | input: | output: | (None, 230, 1, 16) | (None, 230, 1, 8)
batch_normalization_13 | BatchNormalization | input: | output: | (None, 230, 1, 8) | (None, 230, 1, 8)
max_pooling2d_7 | MaxPooling2D | input: | output: | (None, 230, 1, 8) | (None, 115, 1, 8)
conv2d_16 | Conv2D | input: | output: | (None, 115, 1, 8) | (None, 115, 1, 8)
batch_normalization_14 | BatchNormalization | input: | output: | (None, 115, 1, 8) | (None, 115, 1, 8)
max_pooling2d_8 | MaxPooling2D | input: | output: | (None, 115, 1, 8) | (None, 58, 1, 8)
flatten_4 | Flatten | input: | output: | (None, 58, 1, 8) | (None, 464)
dense_4 | Dense | input: | output: | (None, 464) | (None, 460)
reshape_2 | Reshape | input: | output: | (None, 460) | (None, 460, 1, 1)
conv2d_17 | Conv2D | input: | output: | (None, 460, 1, 1) | (None, 460, 1, 8)
batch_normalization_15 | BatchNormalization | input: | output: | (None, 460, 1, 8) | (None, 460, 1, 8)
up_sampling2d_6 | UpSampling2D | input: | output: | (None, 460, 1, 8) | (None, 920, 2, 8)
conv2d_18 | Conv2D | input: | output: | (None, 920, 2, 8) | (None, 920, 2, 8)
batch_normalization_16 | BatchNormalization | input: | output: | (None, 920, 2, 8) | (None, 920, 2, 8)
up_sampling2d_7 | UpSampling2D | input: | output: | (None, 920, 2, 8) | (None, 1840, 4, 8)
conv2d_19 | Conv2D | input: | output: | (None, 1840, 4, 8) | (None, 1838, 2, 16)
batch_normalization_17 | BatchNormalization | input: | output: | (None, 1838, 2, 16) | (None, 1838, 2, 16)
up_sampling2d_8 | UpSampling2D | input: | output: | (None, 1838, 2, 16) | (None, 3676, 4, 16)
conv2d_20 | Conv2D | input: | output: | (None, 3676, 4, 16) | (None, 3676, 4, 1)

Figure 4

DBC

In the second stage, I propose a discriminatively boosted clustering (DBC) framework based on the learned FCAE and an additional soft k-means model. I train the DBC model in a self-paced learning procedure, where deep representations of raw images and cluster assignments are jointly learned. This overcomes the separation issue of the traditional clustering methods that use features directly learned from auto-encoders.

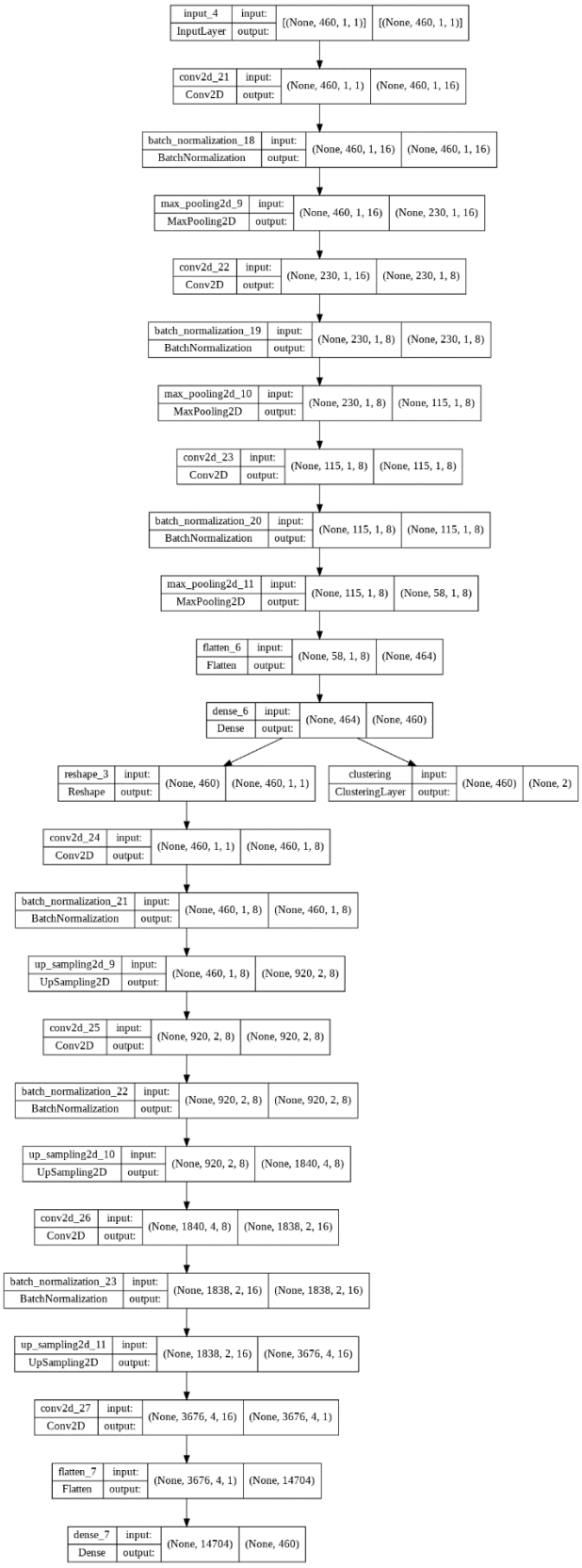

Figure 5

Training Clustering Model

I stack a clustering layer after the pre-trained encoder to form the clustering model. For the clustering layer, I initialize its weights, the cluster centers using k-means trained on feature vectors of all images.
The next step is to improve the clustering assignment and feature representation simultaneously. For this purpose, I define a centroid-based target probability distribution and minimize its KL divergence against the model clustering result. The target distribution will have the following properties:

- Strengthen predictions, i.e., improve cluster purity.
- Put more emphasis on data points assigned with high confidence.
- Prevent large clusters from distorting the hidden feature space.

The target distribution is computed by first raising q (the encoded feature vectors) to the second power and then normalizing by frequency per cluster.

It is necessary to iteratively refine the clusters by learning from the high confidence assignments with the help of the auxiliary target distribution. After a specific number of iteration, the target distribution is updated, and the clustering model will be trained to minimize the KL divergence loss between the target distribution and the clustering output. The training strategy can be seen as a form of self-training. As in self-training, we take an initial classifier and an unlabeled dataset, then label the dataset with the classifier to train on its high confidence predictions.
The loss function, KL divergence or Kullback–Leibler divergence it is a measure of behavior difference between two different distributions. I want to minimize it so that the target distribution is as close to the clustering output distribution as possible.

Results

PCA plot

To visualize the clusters, I used the Principal Component Analysis (PCA), to reduce the number of features in our data set we deployed PCA (Principal Component Analysis) which tries to find the best possible subspace. It transforms our initial features into so-called components. These components are basically new variables, derived from the original ones, and they are usually displayed in order of importance.
As you can see in Figure 6, I choose 2 components while preserving as much of the original information as possible. We incorporate the newly obtained PCA scores in the K-means algorithm. In this manner we can perform segmentation based on principal components scores instead of the original features. We add the names of the segments to the labels. To visualize our clusters on a 2D visualization we choose the two components and use them as axes with the help of matplotlib and seaborn library. Thanks to PCA we are sure that the first two components explain more variance than the others. I did the same by considering 3 components and I visualize the clusters on a 3D visualization, as you can see in Figure 7.

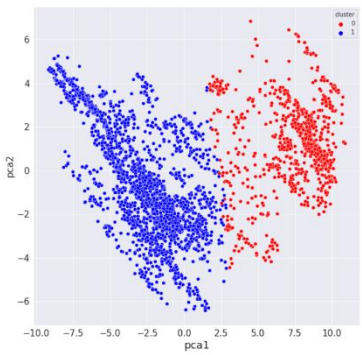

Figure 6

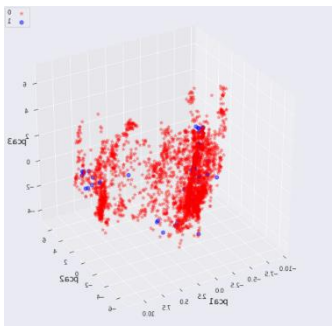

Figure 7

T-SNE plot

When dealing with CNN networks, it is extremely useful the algorithm t-SNE, which stands for "t-distributed Stochastic Neighbour Embedding". The main goal of t-SNE is to project multi-dimensional points to 2- or 3-dimensional plots so that if two points were close in the initial high-dimensional space, they stay close in the resulting projection. If the points were far from each other, they should stay far in the target low-dimensional space too. To do that, t-SNE first creates a probability distribution that captures these mutual distance relationships between the points in the initial high-dimensional space. After this, the algorithm tries to create a low-dimensional space that has similar relations between the points.

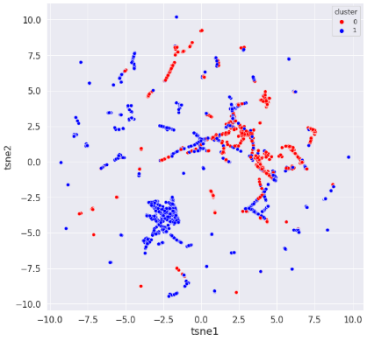

Figure 9

It's essentially an optimization problem — and the algorithm uses Adam optimizer to solve it. As a cost function, it uses *Kullback–Leibler* divergence — a commonly used measure of how different two data distributions are.

Silhouette plot

Silhouette refers to a method of interpretation and validation of consistency within clusters of data.

The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighbouring clusters. If most objects have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters.

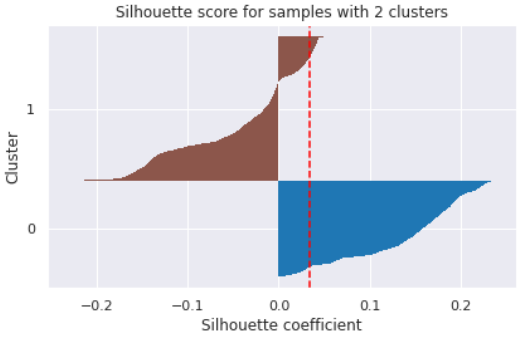

Figure 10

Evaluation metrics

Two standard metrics are used to evaluate the experiment results explained as follows:

- **Accuracy (ACC)** . Given the ground truth labels $\{c_i \mid 1 \le i \le m\}$ and the predicted assignments $\{\hat{c_i} \mid 1 \le i \le m\}$, ACC measures the average accuracy:

$$ACC(\hat{c}, c) = \max_g \frac{1}{m} \sum_{i=1}^{m} 1\{c_i = g(\hat{c_i})\}$$

  Where g ranges over all possible one-to-one mappings between the labels of the predicted clusters and the ground truth labels. This metric takes a cluster assignment from an unsupervised algorithm and a ground truth assignment and then finds the best matching between them.
  The best mapping can be efficiently computed by the Hungarian algorithm which is implemented in scikit learn library as '*linear_assignment*'.

- **Normalized mutual information (NMI)** . From the information theory point of view NMI can be interpreted as

$$NMI(\hat{c}, c) = \frac{MI(\hat{c}, c)}{max(H(\hat{c}), H(c))}$$

  Where $H(c)$ is the entropy of c and $NMI(\hat{c}, c)$ is the mutual information of $\hat{c}$ and c.

- **Silhouette Score**. I now define a silhouette (value) of one data point i

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}$$

Where $a(i) = \frac{1}{|C_I|-1}\sum_{j \in C_I, i \neq j} d(i,j)$ and $b(i) = \min_{J \neq I} \frac{1}{|C_J|}\sum_{j \in C_J} d(i,j)$

I have tried different experiments by tuning the values of the epochs, the batch size and the validation batch size: with the following:

```
e = 100 #@param [100, 500, 1000] {type:"raw"}
bs= 256#@param [64, 128, 256] {type:"raw"}
v_bs= 128 #@param [64, 128, 256] {type:"raw"}
```

Considering the hyperparameters epochs=100, the batch size = 256 and the validation size = 128, I obtained the following results.

- Accuracy = 59.102361%
- NMI = 0.002651
- Silhouette Score = 0.034447

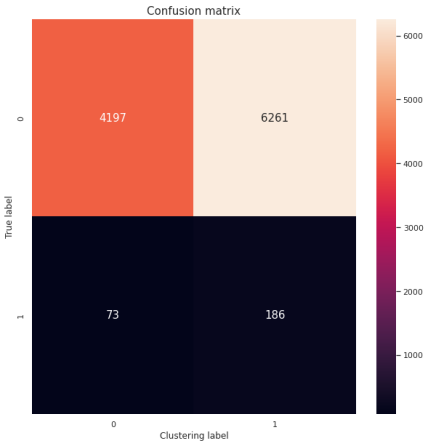

Figure 11

References

- Deep Clustering - Link
- Convolutional Autoencoders for Image Noise Reduction - Link
- How to do unsupervised clustering with Keras, Link
- Deep Clustering with Convolutional Autoencoders - Link
- t-SNE clearly explained - Link