End-to-end deep representation learning for time series clustering: a comparative study

Baptiste Lafabregue, Jonathan Weber, Pierre Gançarski, Germain Forestier

Noname manuscript No.
(will be inserted by the editor)

End-to-end deep representation learning for time series clustering: a comparative study

Baptiste Lafabregue · Jonathan Weber · Pierre Gançarski · Germain Forestier

Received: DD Month YEAR / Accepted: DD Month YEAR

Abstract Time series are ubiquitous in data mining applications. Similar to other types of data, annotations can be challenging to acquire, thus preventing from training Time Series Classification (TSC) models. In this context, clustering methods can be an appropriate alternative as they create homogeneous groups allowing a better analysis of the data structure. Time series clustering has been investigated for many years and multiple approaches have already been proposed. Following the advent of deep learning in computer vision, researchers recently started to study the use of deep clustering to cluster time series data. The existing approaches mostly rely on representation learning (imported from computer vision), which consists of learning a representation of the data and performing the clustering task using this new representation. The goal of this paper is to provide a careful study and an experimental comparison of the existing literature on time series representation learning for deep clustering. In this paper, we went beyond the sole comparison of existing approaches and proposed to decompose deep clustering methods into three main components: (1) network architecture, (2) pretext loss, and (3) clustering loss. We evaluated all combinations of these components (totaling 300 different models) with the objective to study their relative influence on the

B. Lafabregue
IRIMAS, Université de Haute Alsace, Mulhouse, France
ICube, Université de Strasbourg, Strasbourg, France
E-mail: baptiste.lafabregue@uha.fr

J. Weber · G. Forestier
IRIMAS, Université de Haute Alsace, Mulhouse, France
E-mail: firstname.lastname@uha.fr

P. Gançarski
ICube, Université de Strasbourg, Strasbourg, France
E-mail: gancarski@unistra.fr

clustering performance. We also experimentally compared the most efficient combinations we identified with existing non-deep clustering methods. Experiments were performed using the largest repository of time series datasets (the UCR/UEA archive) composed of 128 univariate and 30 multivariate datasets. Finally, we proposed an extension of the Class Activation Maps (CAM) method to the unsupervised case which allows to identify patterns providing highlights on how the network clustered the time series.

Keywords Clustering · Deep Learning · Time series

Acknowledgements The authors would like to thank the creators and providers of the datasets: Hoang Anh Dau, Anthony Bagnall, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Eamonn Keogh, and Mustafa Baydogan. The authors would also like to thank the Mésocentre of Strasbourg for providing access to the GPU cluster. The authors would also like to thank Hassan Fawaz that gave free access to his code on time series processing and cd-diagrams. This work was supported by the ANR TIMES project (grant ANR-17-CE23-0015) of the French Agence Nationale de la Recherche.

General notations used in this paper

- X: the dataset to cluster
- x: an element of X
- x_i: the i^{th} element of X
- N: the number of elements in X
- k: the number of expected clusters
- $|.|$: the cardinatlity of the set
- $f()$: the encoder non-linear function
- $g()$: the decoder non-linear function
- Z: projection of X in the latent space, equals to $f(X)$
- z: an element of Z
- z_i: the i^{th} element of Z

1 Introduction

Automated acquisition systems and the growing storage capacity have made time series data available in a wide range of domains. It includes a wide range of applications such as finance stock prices, electrocardiogram measurements, blood pressure in health, satellite images, earthquake in earth observation, or even social media (Dau et al., 2019). Similar to other types of data, annotations can be challenging to acquire, thus preventing from training Time Series Classification (TSC) models (Dempster et al., 2020; Fawaz et al., 2019; Kotsiantis et al., 2007; Wang et al., 2017).

In this context, clustering can be seen as an alternative to partition time series into homogeneous groups allowing a better analysis of the structure of the data (Saxena et al., 2017). The specificity of the time dimension makes the use of traditional clustering methods challenging. Indeed, each time step cannot be seen as an independent feature. Two time series can represent similar objects but the time signal can be delayed, stretched, or subject to noise. This may result in high differences in the Euclidean space, even though time series denote a similar signal. Thus, clustering methods dedicated to time series have been proposed in the literature (Aghabozorgi et al., 2015; Liao, 2005; Rani and Sikka, 2012).

Most of the existing methods consist in applying a standard clustering method but uses either a specific dissimilarity measure or a time series representation (Aghabozorgi et al., 2015). The dissimilarity measures are tailored to take the specificity of the time dimension into account (i.e. stretches or shifts). Representation learning methods seek to remove the time dimension while keeping neighbors' structure or to rectify the comparison by aligning time series. Some approaches define a stochastic model of the time series, e.g. Hidden Markov Model (Panuccio et al., 2002), or split time series into characteristic segments, e.g. Symbolic Aggregate ApproXimation (Lin et al., 2007), or shapelets, e.g. Unsupervised Salient Subsequence Learning (Zhang et al., 2018). Others aim to apply a transformation operation, e.g. Discrete wavelet transform (Chan and Fu, 1999), or realign time series, e.g. Dynamic Time Warping (Sakoe and Chiba, 1978).

Meanwhile, in computer vision, advances in Deep Neural Networks (DNNs) allowed to make significant progress in clustering domain (Caron et al., 2018; Ghasedi Dizaji et al., 2017; Guo et al., 2017a; Xie et al., 2016; Yang et al., 2019). In addition to their high performance, DNNs consist in an end-to-end system that does not require extra pre-processing steps, resulting in saving time on designing complex frameworks. We will refer to them as deep clustering methods. In parallel, DNNs have proved to have the capacity to achieve competitive performance in time series supervised classification (Dempster et al., 2020; Fawaz et al., 2019). However, to the best of our knowledge, no previous work has been done to largely adapt state-of-the-art deep clustering methods from computer vision to the specificities of the time series domain.
The main contributions of this paper are:

– We establish a review of both existing types of neural network architecture for time series and existing end-to-end clustering methods based on DNNs.
– We detail how these clustering methods can be adapted to process time series.
– We evaluate all combinations on two standard time series benchmarks, the UCR (128 univariate data sets) (Dau et al., 2019) and UEA archives (30 multivariate data sets) (Bagnall et al., 2018).
– We provide insight on the advantages and limitations of these methods for time series.

The rest of this paper is organized as follows. In Sec. 2, we present the major components used in deep learning for clustering and for time series. In Sec. 3, we present the different approaches selected in this study and how we evaluate them. In Sec. 5, we present the results obtained from this evaluation and propose tools to give more insight to the reader on the deep clustering utility for data mining in Sec. 6. Then, we summarize and discuss the main observations that we made in this study in Sec. 7 and conclude in Sec. 8.

2 Background

2.1 Clustering, Deep learning and Time series

Let X be a set of N objects :

$$X = \{x_1, ..., x_N\} \tag{1}$$

and $d(x_i, x_j)$ a measure of dissimilarity between the objects x_i and x_j.

The clustering task can be defined as separating X into a partition $C = \{c_1, ..., c_k\}$ of K clusters, that both maximize the similarity between objects of the same cluster and maximize the dissimilarity between objects of different clusters.

The use of deep learning methods in clustering usually consists in learning a new representation of the data and performing clustering on this new representation instead of the raw data. This representation is obtained from encoding the data with a deep neural network (DNNs), that is called an encoder. An encoder is a non-linear mapping $f_\Theta : X \to Z$, where Θ are the learnable parameters of the encoder. Z is the representation of X as learned by the DNN. The new space created by the DNN is called the latent space, in opposition to the original data space. Thus, the task is now to partition the set Z defined as:

$$Z = \{z_1, ..., z_N\} = \{f_\Theta(x_1), ..., f_\Theta(x_N)\} \tag{2}$$

in respect to the dissimilarity measure $d_z(z_i, z_j)$. Most of the methods in the literature use the Euclidean distance for d_z and K-Means as partitioning method (Bo et al., 2020; Guo et al., 2017a; Jiang et al., 2016; Ma et al., 2019; Xie et al., 2016; Yang et al., 2019). Therefore, the objective is to find the mapping function f_Θ that allows to obtain a relevant partition C for time series data. In the following sections, $f_\Theta()$ may be referred as $f()$.

Time series is a specific kind of data, where each object can be seen as a sequence of time steps. Therefore a time series of length T can be noted as:

$$x_i = [x_{i,1}, x_{i,2}, ..., x_{i,T}] \tag{3}$$

where $x_i \in \mathbb{R}^{d \times T}$, d being the number of features for each time step. Time series can be univariate, $d = 1$ or multivariate, $d > 1$. In this paper, we will refer to time series as multivariate and consider univariate time series as a specific case of multivariate ones where the number of features is equal to one.

In the following sections, we present the main methods proposed in the literature for deep clustering. We present the different types of layers that can be used for time series and then the different types of losses used to learn the model parameters. But first, in the next section, we will introduce the challenges of clustering time series.

2.2 Time dimension and clustering

Usually, classes of a time series dataset represent a view of a phenomenon in a given period of time (e.g. person's electrocardiogram, coordinate evolution of a specific hand gesture). This phenomenon may happen multiple times, on a shifted time-lapse, or/and on a distorted time lapse. It may be quick (e.g. a sudden share price increase) or spread over a long period (e.g. the daily highway traffic). Clustering methods are used in this context to create groups that contain time series representing the same phenomena.

Classically, the data is studied in the Euclidean space. Hence, the distance between two objects is computed as:

$$\begin{aligned} d_{eucl}(x_i, x_j) &= d_{eucl}\left(\begin{bmatrix} x_{i,1} \\ \dots \\ x_{i,T} \end{bmatrix}, \begin{bmatrix} x_{j,1} \\ \dots \\ x_{j,T} \end{bmatrix}\right) \\ &= \sqrt{(x_{i,1} - x_{j,1})^2 + \dots + (x_{i,T} - x_{j,T})^2} \end{aligned} \tag{4}$$

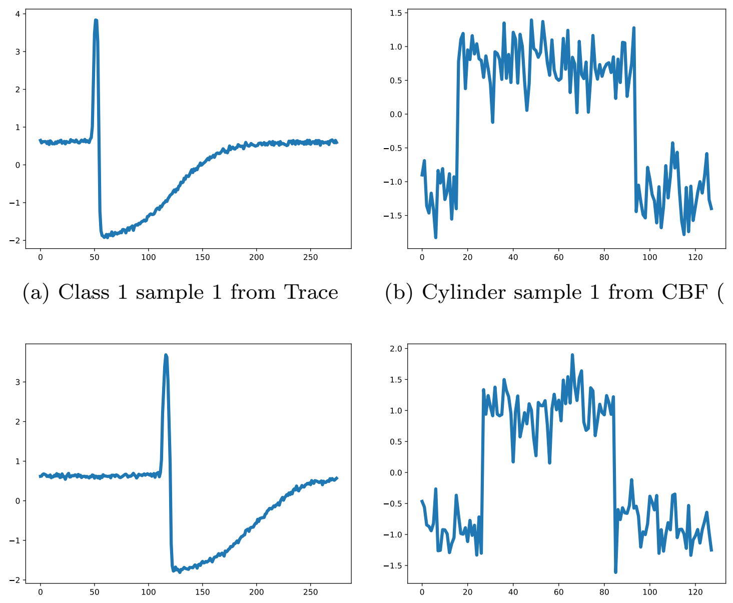
Therefore, each time step is seen as an independent feature. The use of this distance as similarity causes different problems with time series due to the nature of this dimension. Indeed, a simple shift or stretch of the time series may result in a high distance in the Euclidean space. Examples are shown in Fig. 1 where CBF is a synthetic dataset designed to discriminate between three shapes, Cylinder, Bell, and Funnel, and Trace is a synthetic dataset designed to simulate instrumentation failures in a nuclear power plant.

Clustering method are expected to take into account this possibility and be able to recognize shifted or stretched patterns. In the literature, each method uses a different strategy to tackle this issue. For the Symbolic Aggregate ApproXimation method (Lin et al., 2007), this is achieved by dividing the time series into segments and computing and replacing this segment with a computed approximation, reducing the time series dimension. For the Dynamic Time Warping metric, it consists in computing a *warped* path to align time series and computing the distance in the resulting *warped* dimension.

For DNNs, this issue can be solved by the non-linear transformation learned during the training. A specific focus will be given in Sec. 6.3 to evaluate the DNNs' capacity to handle stretching and shifting.

2.3 Encoder architecture

DNNs are structured as a set of layers that follow each other. In this paper, the term **architecture** only refers to the set of layers used and their hyperparameters. In a DNN, layers are divided into three types in the following order: the input layer that corresponds to the actual input, followed by hidden layers, and then the output layer. For encoders, this last layer is usually called the embedding layer, l_z. Each layer is a non-linear function, and the network, with



(a) Class 1 sample 1 from Trace (b) Cylinder sample 1 from CBF (

(c) Class 1 sample 2 from Trace (d) Cylinder sample 2 from CBF

Fig. 1 Examples of time series belonging to the same class with a shift (Figs. 1a and 1c) and a stretch (Figs. 1b and 1d)

L hidden layers, can be noted as:

$$f_{\Theta}(x) = l_z(\theta_z, l_L(\theta_L, l_{L-1}(\theta_{L-1}, ...(l_1(\theta_1(x)))))) \tag{5}$$

where θ_k are the learning parameters of the layer l_k. Multiple types of DNNs layers have been used in the literature to handle time series. They can be divided into three families: Fully connected layers, Convolutional layers, and Recurrent layers.

2.3.1 FCNN: Fully Connected Neural Network Layers

This is the simplest form of layers for DNNs (Rosenblatt, 1958). In this type of layer, every neuron from the layer is connected to every neuron from the previous layer. The layer's result can be expressed for the i^{th} layers, l_i, by the following equation:

$$\begin{gathered} out_{l_0} = x, \\ \forall 0 < i < L, out_{l_i} = a_{l_i}(W_{l_i} out_{l_{i-1}} + b), \end{gathered} \tag{6}$$

with $W_{l_i} \in \mathbb{R}^{out_m \times m}$ being the set of weights, where out_m is the dimension of $out_{l_{i-1}}$ vector and m the number of neurons in layer l_i, b the bias term and a_{l_i} an activation function (e.g. relu, tanh or linear).

FCNN layer can be used for any kind of data, regardless of the input number of dimensions or size, as long as the input has a fixed size. Note that the input can be padded to reach the fixed input size. This makes them easy to use in various domain, such as image (Guo et al., 2017a; Xie et al., 2016), modelization (Sun et al., 2013), or time series (Fawaz et al., 2019). However, this makes it also difficult for them to capture specific dimension's relations such as space or time. Indeed, for a time series, each time step has its own weight and the link to its *time neighbor* is lost during the computation of the layer in Eq. 6.

It has to be noted that FCNN layers are in general used for the embedding layer l_z to get rid of the specific dimensionality of the data, the number of time steps T, and the number of features d in our case. Thus, we obtain a representation in the form of a vector of fixed size given by the user. It allows to use a partitioning method agnostic to data specific structure and uniform the processing of this data.

2.3.2 CNN: 1D Convolutional Neural Network Layers

Convolutional layers allowed huge progress in DNNs performance, especially in computer vision (Krizhevsky et al., 2012; LeCun et al., 1998). Contrary to FCNN layers that treat all time steps independently, they aim to take advantage of hierarchical patterns in the data by learning small and simple patterns in the first layers and assemble them when going towards the last layers. For images, 2D-convolutions are used to capture spatial patterns. Identically, 1D-convolutions are used to capture temporal patterns. They have already demonstrated their good performance in supervised tasks (Fawaz et al., 2019; Wang et al., 2017).

For 1D-convolutions each layer consists in applying m filters of kernel size k to the input sequence out_{l_i-1}, with $out_{l_0} = x$. For each time step t we compute:

$$out_{l_i,t} = a_{l_i}(F.[out_{l_{i-1},t-\lfloor\frac{k}{2}\rfloor}, out_{l_{i-1},t-\lfloor\frac{k}{2}\rfloor+1}, ..., out_{l_{i-1},t+\lfloor\frac{k+1}{2}\rfloor}]) \qquad (7)$$

where $out_{l_i,t}$ is in $\mathbb{R}^m$ and $F \in \mathbb{R}^{m\times k}$ is a matrix composed of the stacked m filters. Then we obtain the following output sequence:

$$out_{l_i} = (out_{l_i,1+\lfloor\frac{k}{2}\rfloor}, out_{l_i,2+\lfloor\frac{k}{2}\rfloor}, ..., out_{l_i,T-\lfloor\frac{k}{2}\rfloor}) \qquad (8)$$

where $out_{l_i} \in \mathbb{R}^{m\times T-(k-1)}$. For 2D-convolutions, different values of stride and padding are often used. Stride is a parameter that controls the step made by the filter when sliding through the sequence. The padding defines how the border of a sample is handled by the filter, by padding the sequence with a certain number of zeros. This results, for a stride of value s, and a padding value of p into:

$$out_{l_i} = (out_{l_i,1+\delta+0\times s}, out_{l_i,1+\delta+1\times s}, ..., out_{l_i,\lceil\frac{T-\delta}{s}\rceil}) \qquad (9)$$

where $\delta = \lfloor \frac{k}{2} \rfloor - p$.

A *half* padding (padding of half the kernel size) is often used, otherwise, if the kernel size, k, is larger than 1, the convolution would crop away the outputted sequence's border. Even though these parameters are often modified in image processing, a stride of 1 and a *half* padding is generally used for time series (Fawaz et al., 2019; Xiao and Cho, 2016; Wang et al., 2017). In consequence, the output of each layer conserves its time dimension size.

However, an alternative padding technique is sometimes used for time series to take into account the specificity of the time dimension, called *causal* padding. Instead of padding on both sides of the input sequence, a padding of $k-1$ is added at the beginning of the sequence. In consequence, the time step t of the outputted sequence is computed/*predicted*, only based on time step t or prior to it from the input sequence. Note that in this case the padding is only added at the beginning, the time dimension size being also conserved after each CNN layer.

Causal padding is, in general, combined with another parameter called dilation (Yu and Koltun, 2015). Dilated convolution works like stride, but the stride effect is applied on the kernel instead of the input sequence. Thus, Eq. 7 can be rewritten with a dilation factor d as:

$$out_{l_i,t} = a_{l_i}(F.[out_{l_{i-1},t-\lfloor\frac{k\times d}{2}\rfloor+0\times d}, out_{l_{i-1},t-\lfloor\frac{k\times d}{2}\rfloor+1\times d}, ..., out_{l_{i-1},t+\lfloor\frac{k\times d+1}{2}\rfloor}]) \tag{10}$$

It can be noted that a dilation factor of 1 corresponds to the vanilla convolution. Usually, the factor d is set in an exponential manner, the dilation factor getting multiple by 2 at each layer. Also, the first layer uses a dilation factor of 1 to preserve the dependency to the previous time step.

2.3.3 RNN: Recurrent Neural Network Layers

Recurrent layers have been proposed specifically to take into account the time dimension (Hochreiter and Schmidhuber, 1997; Hopfield, 1982). This type of layer allowed huge progress in speech recognition (Sak et al., 2014) and language translation (Sutskever et al., 2014). Contrary to other layer types, the input sequence is fed to the layer time step by time step to update the hidden state of the layer. This state can be seen as the memory of the previous steps. The layer itself consists of applying recursive function g that takes as input the current step x_t and the previous hidden state h_t and output the new hidden state:

$$h_t = g(x_t, h_{t-1}) \tag{11}$$

In general, h_0 vector is filled with zero values. Originally, the recursive function was defined as follow:

$$h_t = \tanh(Wx_t + Uh_{t-1} + b) \tag{12}$$

where h_t is a vector of size u, also called the number of units, $W \in \mathbb{R}^{u\times d}$ and $U \in \mathbb{R}^{u\times u}$ being the weights and $b \in \mathbb{R}^u$ the bias vector of the layer that are learned during training. However, this type of recursive function, also called a cell, often leads to a vanishing gradient (Bengio et al., 1994), making the training difficult. Other types of cells have been proposed in the literature, Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU) cells.

An LSTM unit (Gers et al., 2000; Hochreiter and Schmidhuber, 1997) consists of four sub-unit, usually called gates, that controls the information to update the hidden state and the output: an input gate, an output gate, a forget gate, and a candidate memory gate. Each gate is respectively computed as:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ \tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \end{aligned} \tag{13}$$

where σ is the sigmoid function.
From these gates is computed the memory cell:

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \tag{14}$$

where $\circ$ denotes the element-wise product. And the hidden state is updated as follow:

$$h_t = o_t \circ \tanh(c_t) \tag{15}$$

A GRU unit (Cho et al., 2014) is similar to LSTM but works with fewer parameters. The performance of GRU cells has shown to be similar to LSTM, even though it seems more limited than LSTM on some tasks (Weiss et al., 2018). It consists of three gates: an update gate, a reset gate, and a candidate gate, respectively computed as:

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\ \hat{h}_t &= \tanh(W_h x_t + U_h (r_t \circ h_{t-1}) + b_h) \end{aligned} \tag{16}$$

The hidden state is then updated as follow:

$$h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \hat{h}_t \tag{17}$$

For all these types of units, the hidden state, after all the sequence is fed, h_T, usually constitutes the learned representation of the sequence.

Some further options are often used within RNNs. The most common one is the use of bidirectional layers. It consists in training in parallel two identical RNN layers, but one processes each time step in the time order (1 to T), and the other one processes them backward (T to 1). Similarly to CNN layers, the use of dilation has been proposed (Chang et al., 2017), even though it has seen fewer applications. The declination for RNN layers is simpler than for CNN. It can be resumed, for a dilation factor d, to the following equation:

$$h_t = cell(x_t, h_{t-d}), \tag{18}$$

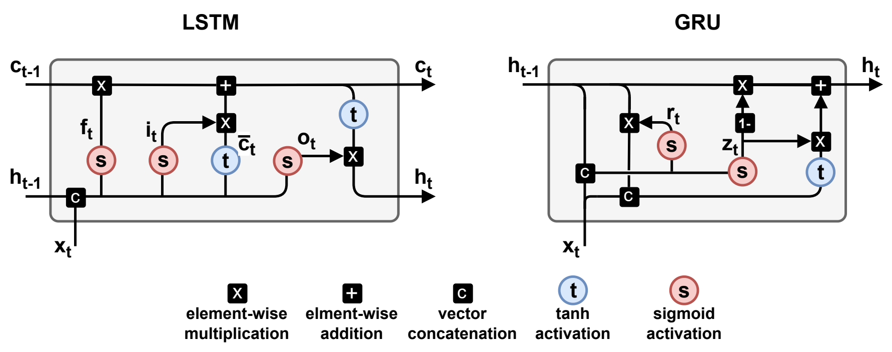


Fig. 2 LSTM and GRU cells with input x_t and the computation of the new hidden state h_t, and for LSTM the additional computation of the new cell state c_t

where *cell*() is the computation of the RNN unit hidden state. It is a skip connection to compute the hidden state at time t with respect to the d^{th} previous hidden state instead of the direct previous state. It is also used with exponentially increasing dilation. Finally, it can be pointed out that RNNs were originally proposed as a unique layer. However, they can be stacked (Pascanu et al., 2013). As RNN layers need to be fed a sequence, each additional RNN layer takes the sequence of previous RNN layer hidden states as input.

2.3.4 Attention mechanism

An attention mechanism, more specifically *self-attention* in our case, allows to quantify the interdependence within the input elements in order to focus the network's attention only on the elements that are important for the training task. Attention mechanisms often come as a component or as the basic structure for new DNN models, with the Transformers (Vaswani et al., 2017). They have attracted a lot of interest recently due to their very good performances, surpassing classical convolutions in some cases, but also to their much lower computation times (Vaswani et al., 2017). Attention mechanisms were first used in Natural Language Processing (NLP) (Bahdanau et al., 2014; Vaswani et al., 2017) then in computer vision (Guan et al., 2018; Jaderberg et al., 2015; Woo et al., 2018). Some methods have been proposed for unsupervised learning, but mainly on specific cases, like graph clustering (Wang et al., 2019) or the use of spatial attention (Souza and Zanchettin, 2019). However, a few works have also been conducted for time series clustering.

One of the main proposition is the DeTSEC method (Deep Time Series Embedding Clustering). This method, proposed by Ienco and Pensa (2019), is based on the use of an autoencoder composed of a bidirectional GRU layer for the encoder and the decoder. However, an attention mechanism is placed at the output of the *forward* layer and the *backward* layer of the bidirectional layer. The attention layer h^{att} is computed as follows:

$$\begin{aligned} v_a &= \tanh(H.W_a + b_a) \\ \lambda &= SoftMax(v_a \circ u_a) \\ h^{att} &= \sum_{j=1}^{T} \lambda_j.h_{t_j} \end{aligned} \tag{19}$$

where *circ* denotes the element-wise product, $H \in \mathbb{R}^{T,l}$ is a matrix constructed by vertically stacking the set of hidden states h_{t_j} learned at different T time steps by the GRU layer, with l the size of the hidden state of the layer. The matrix $W_a \in \mathbb{R}^{l,l}$ and the vectors $b_a, u_a \in \mathbb{R}^{l}$ are parameters learned by the network. The latent representation is then computed for an input x using the outputs of the*forward* (h^{att}_{forw}) attention mechanism layer and the *backward* one (h^{att}_{back}), as:

$$\begin{aligned} f(x) &= gate(h^{att}_{back}) \circ h^{att}_{back} + gate(h^{att}_{forw}) \circ h^{att}_{forw} \\ gate(o) &= \sigma(W_g.o + b_g) \end{aligned} \tag{20}$$

where σ is the sigmoid function and W_g and b_g are parameters learned during the model's training. The *gate* function adds an additional level of decision in order to better discriminate the information returned by the *forward* and *backward* layers.

We can also mention the method proposed by Jiao et al. (2020). This is a general method that can handle several tasks, such as anomaly detection and clustering. It is based on a general model composed of several modules, which can be activated or not according to certain hyperparameters, the hyperparameters being fixed by Bayesian optimization (Shahriari et al., 2015). However, they do not provide precise information on the implementation used for the attention mechanism.

2.4 Training Encoder's parameters for meaningful features

The objective is to train the DNNs to learn a representation that will favor the data clustering into relevant groups. As labels are unknown in unsupervised learning, we need to optimize our DNNs on a side objective. To do so, we use a self-supervised objective, where the data provides the supervision, to obtain meaningful features. This objective is referred to as a ”pretext” or ”proxy” task (Doersch et al., 2015; Larsson et al., 2017; Xu et al., 2019). As the term proxy loss is often used to refer to a loss much easier to optimize computationally than the standard loss function, we will use the term **pretext loss** to refer to them in the rest of this paper.

2.4.1 Autoencoders (AEs)

AEs were first proposed as a dimensionality reduction method (Kramer, 1991; Rumelhart et al., 1986) or as a pre-training method (Ballard, 1987), but they also show to give useful representation for clustering purpose (Becker, 1991). They are the first use of DNNs for clustering and remain the base of most of them.

AEs consist of two parts, an encoder, f, and a decoder, g. The encoder is a non-linear mapping $f : X \rightarrow Z$ that project the data into a latent space (as described in Sec. 2.1), and the decoder a non-linear mapping $g : Z \rightarrow X$ that project latent space variables into the data space. In consequence, an object x_i can be passed to the encoder to obtain its representation z_i, then z_i can be passed through the decoder to obtain a new object $\hat{x_i}$, called the x_i reconstruction. An AE is evaluated by its capacity to reconstruct faithfully x_i into $\hat{x_i}$. Thus, we expect the encoder to be able to retain the important features from the data space into the latent space to allow a good reconstruction. To train the AE weights, we simply minimize the mean square error, also called the reconstruction loss:

$$L_r = \frac{1}{n}\sum_{i=1}^{n}\|x_i - g(f(x_i))\|^2 \tag{21}$$

The decoder is, in general, constructed as a mirror of the encoder (Kramer, 1991; Guo et al., 2017a; Xie et al., 2016) at the exception of the embedding layer l_z.

2.4.2 Regularized autoencoders

The first alternatives to AEs consist of regularized forms of the AE:

– Denoising Autoencoder (DAE) (Vincent et al., 2008): Instead of feeding the original input (x_i) to the AE, a partially corrupted version ($\tilde{x_i}$) is used. The DAE is then trained to reconstruct the original data x_i :

$$L_{dae} = \frac{1}{n}\sum_{i=1}^{n}\|x_i - g(f(\tilde{x_i}))\|^2 \tag{22}$$

  Therefore the objective is to clean the corrupted input, resulting in an embedding robust to small variations. The corruption, or noise, is generated randomly for each object and at each iteration. Different methods exist to generate the corrupted input. Originally, masking noise (a fraction of the input is set to 0) was used, but other types, like isotopic Gaussian noise or salt-and-pepper noise (a fraction of the input is set to min or max value), can be used. Note that the corruption is only applied during the training phase. No corruption is performed when the representation is computed for the clustering task.

A derived form of DAE, Stacked Denoising Autoencoders (SDAE), was also proposed and showed good results (Vincent et al., 2008; Xie et al., 2016). The concept is similar to DAEs but it differs by including a pre-training phase done one layer at a time. For each step i from 1 to L, the AEs is composed with only the i first encoder layers and the i last decoder layers. Then, only the i^th encoder layer and i^th from the end decoder layer are trained for the denoising task. This pre-training phase is followed by a fine-tuning phase similar to the basic DAE.

– Sparse Autoencoder (SAE) (Makhzani and Frey, 2013): SAEs differ from AEs by only allowing a small number of neurons to be active at once in the embedding layer. For a k-sparsity, it simply consists in selecting the k largest hidden units outputted by the encoder and set the others to zero before passing them to the decoder. Other versions express the sparsity penalty term directly in the loss function by, for example, take advantage of the Kullback-Leiber divergence (Zeng et al., 2018).
– Contractive Autoencoder (CAE) (Rifai et al., 2011): Whereas DAEs are designed to increase the robustness of reconstruction to small modifications in the input, CAEs are designed to increase the robustness of the representation itself. The regularization term corresponds to the Frobenius norm of the Jacobian matrix of the encoder activation, J_f, with respect to the input x_i. We obtain the following loss function:

$$L_{cae} = L_r + \lambda \sum_{i=1}^{n} \|J_f(x_i)\|_F^2 \tag{23}$$

where λ is a hyper-parameter that controls the strength of the regularization. This regularization allows CAEs to ignore variations present in the data (e.g. translation or rotation for images) but also more small and rare variations (present in specific examples), while the reconstruction loss only ensures that the reconstruction is faithful (Rifai et al., 2011).

2.4.3 Generative methods

Contrary to AEs that rely on the reconstruction task, other methods rely on generating realistic data to train the encoder:

– Variational Autoencoder (VAE) (Kingma and Welling, 2013): Based on AEs architecture (an encoder and a decoder), VAEs differ significantly on their training phase. VAEs want to exploit the capacity of the decoder to generate data. In AE, the decoder is only used to reconstruct a previously encoded input, but we could think about take a random point in the latent space and decode it to obtain new content. However, this supposes that the latent space is regular enough. VAEs aim to introduce such regularization by assuming that the data follows a distribution, in practice an isotopic Gaussian distribution.
The VAEs' concept is to pass an input set x to the encoder and map it to a Gaussian distribution $q(z_g|x)$. We sample z_g from $q(z_g|x)$ and pass z_g

through the decoder to obtain the distribution $p(x|z_g)$. To train the network we use the following loss (Kingma and Welling, 2013), called evidence lower bound (ELBO):

$$L_{vae} = \sum_{i=1}^{n} - \mathop{\mathbb{E}}_{z_g \sim q(z_g|x_i)} [\log p(x_i|z_g)] + KL(q(z_g|x_i) \| p(z_g)) \tag{24}$$

The first term is the reconstruction loss to ensure good reconstruction from the decoder. The second, is a regularization term that aims to make converge the expected distribution $p(z_g)$ to the observed one generated by the encoder. $p(z_g)$ is constructed as a standard Normal distribution with mean zero and variance one.
Nonetheless, VAE's learned representation is not the most fitted to clustering. Some works proposed some slight modifications to that extends. Jiang et al. (2016) proposed a new version, called Variational Deep Embedding (*VADE*), that, instead of trying to learn one distribution, learn as many distributions as expected clusters. This result, for Eq. 24, in replacing the single distribution $q(z_g|x)$ by a set of distribution $q(z_g, c|x)$ for $c \in 1, .., K$ for K clusters, and the expected distribution $p(z_g)$ by $p(z_g, c)$. The set of distributions is initialized from a Gaussian Mixture Models on the pre-trained latent space with another loss (e.g. AE, or VAE). Li et al. (2018) proposed a method based on VAE to learn a multi-facet clustering structure of the latent space instead of a single partition.

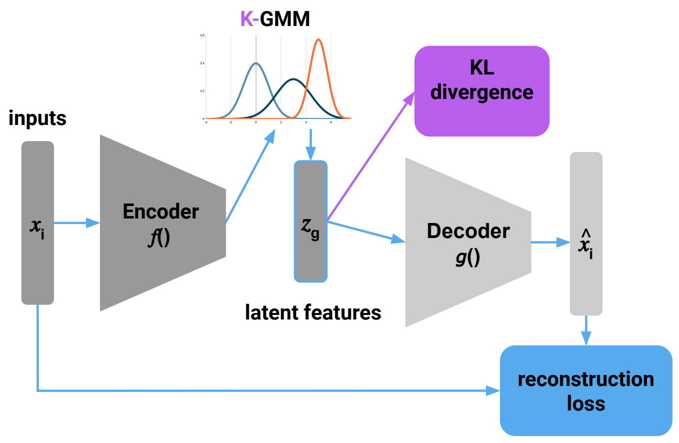


Fig. 3 *VADE* method with K clusters: the encoded representations are used to generate K Gaussian distributions. Samples are generated for each distribution. The DNN is trained to reconstruct them and fit the generated distribution with the original one.

– Generative Adversarial Network (GAN) (Goodfellow et al., 2014): GAN is the other major generative type of DNNs. GANs are composed of two elements, the generator G_{Θ_G} and the discriminator D_{Θ_D}, that are both DNNs. The generator, $G : X \rightarrow Z$, generates data from a latent space, similarly to the AE's decoder. The discriminator, $D : X \to \mathbb{R}$, generates a real value from the data space that can be seen as the probability of the data to be real. The two networks are trained in a two-player game. The two adversaries are trained through the following min-max objective:

$$\min_{\Theta_G} \max_{\Theta_D} \mathop{\mathbb{E}}_{x \sim \mathbb{P}_x^r} q(D(x)) + \mathop{\mathbb{E}}_{z \sim \mathbb{P}_z} q(1 - D(G(z))) \tag{25}$$

where $\mathbb{P}_x^r$ is the distribution of real data samples, $\mathbb{P}_z$ is the prior noise distribution on the latent space, and $q(x)$ is the quality function. For vanilla GAN, $q(x) = \log x$, and for Wasserstein GAN $q(x) = x$. We expect, at the end of the training, the convergence of $\mathbb{P}_z$ towards $\mathbb{P}_x^r$.
However, one can notice that GANs do not include any encoder part in their framework. Therefore, they cannot be used as-is for clustering purposes as no representation can be extracted from our data. Few works have been proposed in the literature to include an encoder. In Lipton and Tripathi (2017), they tested to back-propagate the data through G, but the obtained representation was not suitable for clustering purpose (Mukherjee et al., 2019). Ghasedi et al. (2019) added a third network, a clusterer $E : X \to Z$, and modified the discriminator to not only discriminate if the data space example is real or not but if the joint distribution of samples (E(x), x) and (z, G(z)) is coming from either the generator or the clusterer. This result in the following objective function:

$$\min_{\Theta_G, \Theta_E} \max_{\Theta_D} \mathop{\mathbb{E}}_{x \sim \mathbb{P}_x^r} q(D(C(x), x)) + \mathop{\mathbb{E}}_{z \sim \mathbb{P}_z} q(1 - D(z, G(z))) \tag{26}$$

Mukherjee et al. (2019) also included an encoder, E. But in this case, they enforce the K last features of the latent space to be a one-hot vector of the expected cluster. Therefore the latent feature is composed of $z = concat(z_n, z_c)$, where z_n is sampled from a normal distribution and z_c is a one-hot vector of a cluster selected at random. Then, the networks are trained with the following objective function:

$$\begin{aligned}\min_{\Theta_G, \Theta_E} \max_{\Theta_D} \mathbb{E}_{x \sim \mathbb{P}_x^r} q(D(x)) + \mathop{\mathbb{E}}_{z \sim \mathbb{P}_z} q(1 - D(G(z))) + \\ \beta_n \mathop{\mathbb{E}}_{z \sim \mathbb{P}_z} \|z_n - E(G(z_n))\|_2^2 + \beta_c \mathop{\mathbb{E}}_{z \sim \mathbb{P}_z} \mathcal{H}(z_c, E(G(z_c))\end{aligned} \tag{27}$$

where $\mathcal{H}()$ is the cross-entropy loss. The first two equation elements are the vanilla GAN, the third ensure the reconstruction quality of the re-encoded features, and the last ensures that the generated one-hot encoded part is preserved by the encoding. β_n and β_c are weights to leverage between continuous and discrete characteristics of the latent space.

2.4.4 Time series triplet loss

Another loss has been proposed to train an encoder specific to time series, called triplet loss (Franceschi et al., 2019). This loss has the advantage to require an architecture that only includes an encoder. Removing the decoder

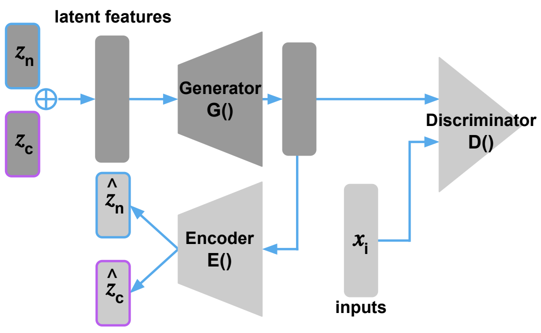


Fig. 4 *ClusterGAN* method: the latent feature is separated into two components, one from a Gaussian distribution, z_n, and the other to one-hot encoded the clustering assignment, z_c. An encoder is added and trained to both preserve z_n and z_c encoding.

has the advantage of computational gain as we do not need to train its parameters, but also to remove the problem of designing the decoder architecture. This loss is based on a former work (Schroff et al., 2015) that proposed a loss to obtain similar representation for similar objects while pushing apart representation of dissimilar objects. However, it supposes a supervised knowledge on objects' similarity. The authors in Franceschi et al. (2019) proposed to solve this problem by using a time-based sampling strategy.

They do the assumption that if we pick a subseries at random, x^{ref}, from a time series x_i, than we can except, with a good probability, two things. In one hand, x^{ref} representation will be close to any of its own subseries x^{pos} (positive example), x^{ref} can be seen as the *context* of x^{pos}. On the other hand, x^{ref} representation will be distant from any subserie x^{neg} randomly taken in another time series x_j, with $j \neq i$. They also introduced another parameter $K_{triplet}$ that sets the number of negative samples to use for each training object to improve stability. The loss is computed with the following equation:

$$L_{triplet} = -\log(\sigma(f(x^{ref})^T f(x^{pos})) - \sum_{l=1}^{K_{triplet}} \log(\sigma(-f(x^{ref})^T f(x_l^{neg})), \quad (28)$$

where σ is the sigmoid function. The first term trains the DNN to minimize the dissimilarity between x^{ref} and x^{pos} representations, whereas the second term train the DNN to maximize the dissimilarity between x^{ref} and the $K_{triplet}$ x^{neg} samples representations.

2.5 Training Encoder's parameters for clustering task

In the previous section, we have described different objective functions to obtain meaningful features from the latent space. However, most of these representations are not necessarily suitable for clustering as they well describe the input time series but do not output a discrete latent space. In this case, discriminate each cluster can be difficult.

To this end, multiple methods proposed to add, either in parallel or as post-processing, a complementary loss to obtain a more separable latent space. We will refer to them as **clustering loss**.

2.5.1 Deep Embedded Clustering (DEC)

Proposed in Xie et al. (2016), it is one of the most referred clustering loss, and it has been the subject of many adaptations (Bo et al., 2020; Guo et al., 2017a; Ma et al., 2019; Yang et al., 2019). The idea is to learn clusters as we train the encoder's parameters. The method is divided into two steps. First, the encoder's parameters are initialized through an AE architecture. Then, the decoder is detached and the encoder's parameters are optimized by computing an auxiliary target distribution and minimizing the Kullback–Leibler (KL) divergence to it.

After the first phase, we obtain an initial estimate of the final latent space. An initial K-Means clustering is performed on the encoded features that outputs as set of k centroids $\{\mu_j\}_{j=1}^k$. In the second step, we compute a new distribution of the latent space Q, from Z that uses the Student's t-distribution as a kernel to measure the similarity between embedded point z_i and centroid μ_j:

$$q_{ij} = \frac{(1 + ||z_i - \mu_j||^2/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_{j=1}^k (1 + ||z_i - \mu_j||^2/\alpha)^{-\frac{\alpha+1}{2}}}, \tag{29}$$

where α is the degree of freedom of the Student's t- distribution. The authors set this value to $\alpha = 1$ as it cannot be directly cross-validated (Xie et al., 2016). The obtained value $q_{i,j}$ can be seen as the degree of belief that the object x_i belongs to the cluster j. Therefore, for each value x_i we obtain a soft assignment vector q_i. The function that computes q_{ij} from z_i is called the clustering layer. The objective is now to make this soft assignment q_i *harder* and, as explained by the authors, that have the following properties: (1) strengthen predictions (i.e., improve cluster purity), (2) put more emphasis on data points assigned with high confidence, and (3) normalize loss contribution of each centroid to prevent large clusters from distorting the latent feature space. To this extends they used the distribution P defined as:

$$p_{ij} = \frac{q_{ij}^2 / \sum_{i=1}^N q_{ij}}{\sum_{j=1}^k (q_{ij}^2 / \sum_{i=1}^N q_{ij})} \tag{30}$$

The encoder is then trained with the KL divergence:

$$L_c = KL(P|Q) = \sum_{i=1}^N \sum_{j=1}^k p_{ij} \log \frac{p_{ij}}{q_{ij}} \tag{31}$$

As the authors wanted a generic method they used a simple architecture with only fully connected layers.

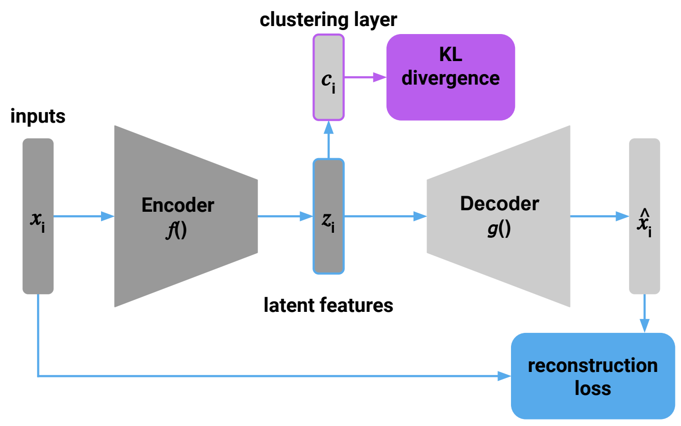


Fig. 5 *IDEC* method: The DNN is trained to both reconstruct the data and obtain a more densely distributed representation with KL divergence.

As it was one of the first clustering method with deep learning proposed to give significantly good results, many works have extended this clustering loss.

The first one, Improved Deep Embedded Clustering (*IDEC*) (Guo et al., 2017a), proposed a simple modification that consists in keeping the decoder and the reconstruction loss in the second phase. The idea is to keep the feature's informativeness acquired from the first phase. It results in the new loss:

$$L_{IDEC} = (1 - \gamma)L_c + \gamma L_r \tag{32}$$

One can notice that *DEC* is a particular application of *IDEC* with $\gamma = 0$. However, this modification seems not relevant for all data sets (Guo et al., 2017a). It can also be made mention of the *IDEC* version that uses convolutional AE (Guo et al., 2017b) instead of fully connected layers. This last adaptation already shows the importance of AE architecture.

More sophisticated variations were also proposed. For example, the method Structural Deep Clustering Network (*SDCN*) (Bo et al., 2020), that incorporate a Graph Convolutional Network (GCN) (Kipf and Welling, 2016), trained in parallel of the *IDEC* encoder with the same number of layers. A GCN layer allows keeping the graph relation of the data. The authors use it to keep neighborhood relations from the data space in the latent space. A KNN-graph is created from the train set, where an edge is added for each sample with its k_{knn} nearest neighbors. Each encoder is regularised by a GCN layer to keep this graph structure.

2.5.2 Other noticeable clustering loss

Many other methods have been proposed in the literature, even though all of them could not be cited, here are some of the main ones:

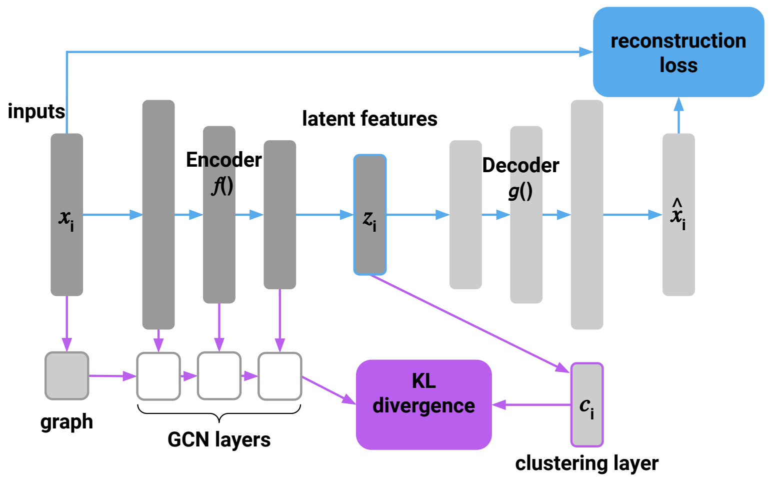


Fig. 6 *SDCN* method: It adds GCN layers to IDEC framework to preserve local structure from the data space to the latent space.

– *DeepCluster* (Caron et al., 2018): The authors propose a simple training solution that consists in alternating between two phases. First, it clusters the train set in the latent space with K-Means to obtain an assignment for each object. Then, the encoder parameters are optimized to predict these pseudo labels. However, the authors only used this method as a pre-training method to initialize the DNNs weight for supervised classification.
– Deep Embedded Regularized Clustering (*DEPICT*) (Ghasedi Dizaji et al., 2017): The *DEPICT* framework is similar to *IDEC* work but made further modifications to *DEC* method. First, it uses a clustering loss similar to *DEC* and also incorporate a clustering layer composed of a Dense layer of size k, followed, this time, by a softmax layer. In consequence, the distribution Q is defined as:

$$q_{ij} = Q(y_i = j|z_i, \Theta_{soft}) = \frac{exp(\theta^T_{soft,k} z_i)}{\sum_{j=1}^{k} exp(\theta^T_{soft,k} z_i)}, \tag{33}$$

where $\Theta_{soft} = [\theta_{soft,1}, .., \theta_{soft,k}]$ are the weights of the clustering layer. P is computed similarly to *DEC* by Eq. 30. Then, they use the KL divergence between P and Q to train the DNN, but also add a regularisation term to avoid degenerated solution with only a few big clusters:

$$\begin{aligned} L &= KL(Q|P) + KL(f|u) \\ &= [\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} p_{ij}\log\frac{p_{ij}}{q_{ij}}] + [\frac{1}{N}\sum_{j=1}^{k} f_j \log\frac{f_j}{u_j}] \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{k} p_{ij}\log\frac{p_{ij}}{q_{ij}} + p_{ij}\log\frac{f_j}{u_j}, \end{aligned} \tag{34}$$

where $f_j = \frac{1}{N}\sum_{i=1}^{N} q_{ij}$ is the empirical cluster distribution, i.e. the frequency of the clusters. And u_j is a uniform distribution. This implies the strong assumption that clusters are equally represented in the training set. Then, they show that this regularization can be approximated by computing the standard cross-entropy between Q and P

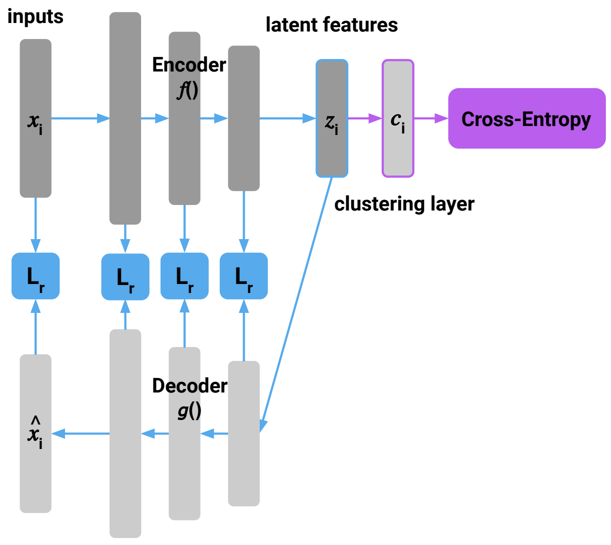


Fig. 7 *DEPICT* method: The DNN is trained to both reconstruct the data at each layer depth and obtain better confidence into predicting clustering pseudo labels with cross-Entropy.

Second, they also changed the AE, to use a Denoising version with masking noise on each layer (i.e. with dropout layers). After a pre-training phase, the noisy AE is jointly trained with the clustering loss as in *IDEC* method. Moreover, they extend the classical reconstruction loss to be computed as the sum of the reconstruction at each depth of the autoencoder:

$$L_{multi\_rec} = \frac{1}{n}\sum_{i=1}^{n}\sum_{l=0}^{L-1}\frac{1}{|z_i^l|}(z_i^l - \hat{z}_i^l)^2 \tag{35}$$

where z_i^l is the output of the l^{th} layer of the encoder (the input for $l = 0$), $|z_i^l|$ its output size, and $\hat{z}_i^l$ the output the l^{th} layer of the decoder from the end. This loss insure that the reconstruction is kept at all the stages of the autoencoder.

– Joint Unsupervised Learning (JULE) (Yang et al., 2019): This approach is highly different from others as it does not use a separated clustering loss. This is still included in this section because it is often referred to. This method is an agglomerative clustering approach, at each step two clusters are merged until reached the desired number of clusters. This merging is

done with respect to an affinity matrix computed in the latent space. This choice is mitigated by considering the local structure of the data, to favor the merging of clusters that are, at the same time, close to each other and far from other neighbors.
The encoder f is updated every p steps by the following loss:

$$L_{JULE} = -\frac{1}{K_c - 1} \sum_{i,j,k} (\gamma \mathcal{A}(f(x_i), f(x_j)) - \mathcal{A}(f(x_i), f(x_k))), \qquad (36)$$

where γ is a weight, $\mathcal{A}()$ is the affinity measure between two objects. x_i and x_j are from the same cluster, while x_k is from the K_c closest neighbouring clusters. $\mathcal{A}()$ is equal to the weight of the affinity matrix W from vertex x_i to x_j defined by:

$$W(i,j) = \begin{cases} exp(-\frac{\|f(x_i)-f(x_j)\|_2^2}{\delta}), & \text{if } x_i \in \mathcal{N}_i^{K_s} \\ 0, & \text{otherwise} \end{cases} \qquad (37)$$

where $\mathcal{N}_i^{K_s}$ is the set of K_s x_i's nearest neighbors and δ the mean squared error between x_i and its neighborhood $\mathcal{N}_i^{K_s}$.

2.5.3 Deep clustering methods for time series

Some deep clustering methods have been also proposed in the specific context of time series:

- Deep Temporal Clustering (*DTC*) (Madiraju et al., 2018): This method is organized as *IDEC* framework. However, they changed the architecture of the encoder by replacing the encoder with a 1D-Convolution layer followed by a MaxPooling layer and two stacked bi-LSTM cells. The decoder consists of a simple upsampling followed by a 1D-Convolution layer to reconstruct the data. Moreover, it is not the state of the last bi-LSTM cell that is used for representation but the reconstructed sequence. Therefore they keep the time dimension in the embedding. Based on this, they modified the computation of *DEC* distribution Q from Eq. 29 by:

$$q_{ij} = \frac{(1 + sim(z_i, \mu_j)/\alpha)^{-\frac{\alpha+1}{2}}}{\sum_{j=1}^{k} (1 + sim(z_i, \mu_j)/\alpha)^{-\frac{\alpha+1}{2}}}, \qquad (38)$$

  where $sim(x_i, x_j)$ is a measure of similarity between objects x_i and x_j. They then used different similarity measures than the Euclidean, the Complexity Invariant Distance giving the best results.
- Deep Temporal Clustering Representation (*DTCR*) Ma et al. (2019): Similarly to *DTC*, the authors used three stacked bi-directional RNN network as the encoder. However, they also add exponential dilatation. The decoder is a single RNN layer, its hidden state is initialized with the concatenation of the final hidden state of encoder's layers. For the training objective, the authors proposed a new loss composed of three parts:

- A classical reconstruction loss L_r.
- A real/fake loss $L_{classif}$: the encoder is trained to discriminate if the input is real or fake, fake samples being generated by randomly shuffling 20 % of the time steps. To this extent, a 2-dim soft-max layer is attached to the encoder and trained to predict a 2-dim one-hot vector indicating real or fake with the categorical cross-entropy.
- A K-Means loss $L_{K-Means}$: this loss is based on the spectral relaxation for k-means clustering proposed in Zha et al. (2002). It consists in minimizing the following function:

$$\min_F Trace(H^T H) - Trace(F^T H^T H F), s.t. F^T F = I, \tag{39}$$

  where $H \in \mathbb{R}^{m \times N}$ is the data matrix, with m the latent space dimensions. And $F \in \mathbb{R}^{N \times k}$ is the cluster indicator matrix. F is fixed when the DNNs are trained but it is updated every 10 iterations by computing the k-truncated singular value decomposition of H.

We obtain the combined loss:

$$L_{DTCR} = L_r + L_{classif} + \lambda L_{K-Means}, \tag{40}$$

where λ is a regularization coefficient. One can notice that this loss is not specific to time series and could be used for other data types.

3 Evaluated methods and implementations

All of the methods presented in Sec. 2.5 have their specific framework, with their own **architecture** (i.e. types and number of layers, optimizer, etc.) their own **clustering loss** and often their own **pretext loss**. Thus, the comparison is often difficult to do, as we cannot be sure which parts or combinations of elements in the method are explaining the performance difference. Moreover, comparing a generic method that can handle various types of data types to a tailored method to a specific set of data may be seen as unfair, especially when the generic method can be easily adapted.

This study aims to cover the different elements available to cluster time series in deep learning and highlight the influence of each element on the clustering performance. However, given the high number of approaches proposed in the literature, we only studied a selection of methods. In this section, we first explain how we decomposed the analysis of the clustering methods, then we give more details on the selected elements.

All the methods were implemented in `TensorFlow 2` based on the existing code when available. A reference to the authors' code is added in this case. The code used for the study is available online[1].

[1] https://github.com/blafabregue/TimeSeriesDeepClustering

Table 1 Summarize of loss and architecture compatibility. For each pretext and clustering loss, we list the compatible architecture types. *: indicates that it is with the exclusion of the Dilated-RNN architecture

		Clustering loss						
		DEPICT	SDCN	DTCR	DEC	IDEC	ClusterGAN	VADE
Pretext loss	Multi_rec	✓FCNN ✓CNN ✗RNN	✓FCNN ✓CNN ✗RNN	✓FCNN ✓CNN ✗RNN	✓FCNN ✓CNN ✗RNN	✓FCNN ✓CNN ✗RNN	✗All	✗All
	Reconstr.	✓All	✓FCNN ✓CNN ✗RNN	✓All	✓All	✓All	✗All	✗All
	Triplet	✓All	✓FCNN ✓CNN ✗RNN	✓All	✓All	✓All	✗All	✗All
	VAE	✓All	✓FCNN ✓CNN ✗RNN	✓All	✓All	✓All	✗All	✓All*
	GAN	✗All	✗All	✗All	✗All	✗All	✓All*	✗All

3.1 The deep clustering method's decomposition

End-to-end deep clustering methods are usually decomposed into two phases. A pre-training phase, where the network is trained to retain meaningful features with a pretext loss. And a clustering phase, where the network is trained to output features suitable for the clustering task. Therefore, DNNs for clustering are often the combination of three elements: an architecture (i.e. the set of layers in this paper), a clustering loss, and a pretext loss.

For example, the *DEC* method presented in Xie et al. (2016) can be decomposed as follow:

1. A FCNN auto-encoder as architecture of the DNN.
2. The reconstruction loss is used as pretext loss in the pre-training phase.
3. The clustering loss based on the KL divergence is used as clustering loss in the clustering phase.

The list of all combinations used is summarized in Table 1.

3.2 Architectures

As explained in Sec. 2.3, three DNNs architecture families can be used for time series, Fully Connected Neural Networks (FCNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN). However, various configurations can be made for each type depending on a lot of hyperparameters, like the number of layers, the size of each layer (number of neurons in FCNN, number of filters in CNN, and number of cells in RNN), the addition of specific layers (e.g. pooling layers in CNN, use of bidirectional layers in RNN). Hence, we decided to follow the configurations used in other articles for our experiments as detailed below. By default, we fixed the size of the embedding layer to $D_{ls} = 320$, the effect of this parameter will be investigated in Sec. 5.2.2.

FCNN: for this type, we used the configuration proposed in *DEC* and *IDEC* papers (Xie et al., 2016; Guo et al., 2017a). The encoder is composed of three FCNN layers of dimensions d-500-500-2000-D_{ls}, where d is the data-space dimension. The decoder is constructed as a mirror of the encoder architecture excluding the embedding layer.

CNN: for convolutional networks, we use three configurations:

– *ResNet*: This architecture was proposed in Wang et al. (2017). Residual DNNs use skip connections to jump over some layers (i.e. residual blocks). The use of skip connections is mainly motivated to avoid the problem of vanishing gradient when the number of layers increases (He et al., 2016).
  The encoder's implementation used is composed of three residual blocks followed by a global average pooling layer and the embedding layer as an FCNN layer. Each residual block is first composed of three convolutional layers with a fixed filter size of 64. The filter's length is set to 8, 5, and 3 respectively for the first, second, and third convolution. A ReLU activation function, preceded by a batch normalization operation, is then added at the end of the block. Finally, the FCNN embedding layer is added at the end of size D_{ls}.
– Simple-CNN (*SCNN*): It is a simplified version of the *ResNet* architecture composed of only one residual block, without the skip connection. This choice is motivated to evaluate if it is justified to use the deep *ResNet* architecture.
– Dilated-CNN (*DCNN*): This architecture was proposed in Franceschi et al. (2019). It uses two particular hyperparameters, causal padding, and exponential dilated convolutions.
  The encoder is composed of a set of dilated causal convolutional layers followed by the embedding layer as an FCNN layer. The number of filters is fixed to 40 for all layers with a filter length of 3. In the original version, the number of layers was arbitrarily fixed to 10 with an exponential factor for the dilation rate of 2. However, we modified this parameter by computing the number of layers and dilation rate with the function described in algorithm 1. It gives better results and it is faster to compute. Finally, an

Algorithm 1 Compute layers' dilation size

```
 1: procedure (Time series length $ts\_length$)
 2:     $last\_dilation = 1$
 3:     $dilation\_list = []$
 4:     if $ts\_length > 50$ then
 5:         $rate = 2$
 6:     else
 7:         $rate = 4$
 8:     while $last\_dilation < ts\_length/2$ do
 9:         $last\_dilation* = rate$
10:         $dilation\_list+ = last\_dilation$
11:     return dilation_list
```

FCNN embedding layer of size D_{ls} is added at the end. This architecture is based on the authors' code available online [2]

For these three architectures, the decoder is constructed as the mirror of the encoder architecture excluding the embedding layer.

RNN: for recurrent networks, we use three configurations:

- Deep Temporal Clustering (*DTC*): This architecture was proposed in Madiraju et al. (2018). The encoder is first composed of a convolutional layer followed by a max-pooling layer of size 10 to reduce the number of time steps, especially for long series. Then, the output is fed to two stacked bidirectional LSTM (Bi-LSTM) layers of size 50. For the latent space, the output of the Bi-LSTM is retrieved as the hidden state sequence. This is motivated to keep the time dimension in the latent space. Therefore the size of the latent space is not fixed as it depends on the input's dimensions. The decoder is composed of a single deconvolutional layer (an upsampling layer followed by a convolutional layer) with kernel size 10 and a number of filters equal to the number of features in the inputted time series.
- Bidirectional LSTM (*BLSTM*): It is a simple architecture composed of two stacked Bi-LSTM layers. The first layer has a fixed size of 50 and the second one of $\lfloor D_{ls}/2 \rfloor$. For the latent space, we use the final hidden state of the last Bi-LSTM layer. The decoder is constructed as the mirror of the encoder architecture.
- Bidirectional GRU (*BGRU*): This architecture is identical to the *BLSTM* architecture but use GRU cells instead of LSTM cells.
- Dilated-RNN (*DRNN*): This architecture was proposed in Ma et al. (2019). It is composed of three stacked bidirectional dilated RNN encoder, with respectively a dilation rate of 1, 4, and 16. It uses the GRU cells in the model's layers. In the article, the number of units of each layer is either 100-50-50 or 50-30-30. However, as no indication is given on how to make this choice, we fixed it to 100-50-50 for all datasets as it results in better result on average. The final hidden state of the last layer (of size $50 \times 2 = 100$) is used as the latent space.
  The decoder is composed of a single RNN layer with GRU units of size $(100+50+50)\times 2 = 400$. Its initial state is initialized with the concatenation of all encoder layers' final hidden states. Then, the decoder iteratively predicts the reconstructed sequence from the output at $t-1$, where the output at time $t = 0$ is a zero vector. This architecture is based on the authors code available online [3]
- Bidirectional GRU with attention mechanism (*Attention*): This architecture is based on the one presented in Ienco and Pensa (2019). It is composed of a Bi-GRU layer a size 64 if there are less than 250 sample in the train set, and 512 otherwise. This layer followed by a temporal attention mechanism on both the forward and backward layer of size of D_{ls}. However,

[2] https://github.com/White-Link/UnsupervisedScalableRepresentationLearningTimeSeries
[3] https://github.com/qianlima-lab/DTCR

the decoder differs from the one used in Ienco and Pensa (2019) as they use a double reconstruction loss (one for the forward layer and one for the backward layer). To make the architecture closer to other methods we used a simple Bi-GRU layer of size $\lfloor D_{ls}/2 \rfloor$. This architecture is based on the authors code available online [4].

3.3 Pretext losses

For the pretext loss, we use four losses:

- Reconstruction loss (*rec*): It is the classical autoencoder's reconstruction loss that consists in computing the mean squared error between the inputted sequence and its reconstruction (see Eq. 21).
- *DEPICT* reconstruction loss (*multi_rec*): This pretext loss is the one used in Ghasedi Dizaji et al. (2017). It extended the reconstruction loss by computing the mean square error between each encoder's layer and its corresponding decoder's layer (see Eq. 35). One can notice that this loss requires that the decoder is constructed as the mirror to the encoder, it excludes all RNN architecture in our case.
- VAE loss (*vae*): It is the classical VAE loss that balances between reconstruction and the normalization of the latent space distribution (see Eq. 24):
- Triplet loss (*triplet*): This is the loss proposed in Franceschi et al. (2019) that aims to obtain similar representation between a time subseries and its neighborhood (see Eq. 28). For this loss, we used four values of $K_{triplet}$, 1, 2, 5, and 10. We also computed the result with the combined version (the concatenation of representation on the four different $K_{triplet}$ values). It is based on the authors code available online [5]

We also used the GAN pretext loss, but as GAN does not involve an encoder it can only be used with DNNs designed for the clustering purpose. Therefore it is only used in combination with the *ClusterGAN* clustering loss (see next section).

3.4 Clustering losses

Many losses have been proposed to simplify the clustering task in the latent space, especially for images (Caron et al., 2018; Ghasedi Dizaji et al., 2017; Guo et al., 2017b; Xie et al., 2016; Yang et al., 2019). We selected a subset of them by covering the different types of approaches. We favored the one where the code was available to ensure that the validity of our implementation. The selected losses are presented bellow, but were already further explained in Sec. 2.1:

[4] https://gitlab.irstea.fr/dino.ienco/detsec

[5] https://github.com/White-Link/UnsupervisedScalableRepresentationLearningTimeSeries

- *DEC* (Xie et al., 2016) and *IDEC* (Guo et al., 2017a): The *DEC* loss use the KL divergence to improve assignment confidence of an object to its cluster. The *IDEC* loss extended the latter by keeping the pretext loss (the reconstruction loss in the article) in the clustering phase (see Eq. 32). Following the default parameters, we use a γ value set to 0.1. It is based on the authors code available online [6]
- *DEPICT* (Ghasedi Dizaji et al., 2017): This loss is similar to *IDEC* loss. However, they use the standard cross-entropy to train the DNN parameters. We also use a γ value of 0.1.
- *SDCN* (Bo et al., 2020): This loss is also based on *IDEC* loss. They use Graph Convolutional Networks to regularize the training to keep the local structure of the data based on a KNN graph. We used a number of neighbors equals to 3 given the small size of some datasets. We also changed the KNN algorithm to use DTW instead of Euclidean distance, as DTW give better results on this benchmark (Dau et al., 2019). It is based on the authors code available online [7]
- *VADE* (Jiang et al., 2016): This loss is based on ELBO loss, and therefore is only compatible with VAEs. They extend the ELBO loss by learning one distribution by expected clusters instead of one. It is based on the authors code available online [8]
- *ClusterGAN* (Ghasedi et al., 2019): This loss is based on GAN loss, and therefore is only compatible with it. They add an encoder that is trained to reproduce the generated latent feature. It is based on the authors code available online [9]
- *DTCR* (Ma et al., 2019): This loss is the only proposed for time series, even though it is generic. It combines three components, the pretext loss, a K-Means loss (see Eq. 39), and real/fake loss. We use a λ value of 0.5. It is based on the authors code available online [10]
- None: We also evaluate the result obtained without using any clustering loss.

For all configurations, we use the Adam optimizer with a learning rate of 0.001 (Bo et al., 2020; Franceschi et al., 2019; Madiraju et al., 2018; Mukherjee et al., 2019) at the exception of the *DRNN* architecture where we use the SGD optimizer with exponential decay, a learning rate of 0.1, and a decay rate of 0.1. The batch size used is set to 10 at the exception of the *SDCN* and *DTCR* losses that require a batch size equals to the train set size.

[6] https://github.com/XifengGuo/IDEC
[7] https://github.com/bdy9527/SDCN
[8] https://github.com/slim1017/VaDE
[9] https://github.com/sudiptodip15/ClusterGAN
[10] https://github.com/qianlima-lab/DTCR

4 Evaluation setup

4.1 UCR and UEA archives

To validate our result we used two benchmarks, the UCR univariate archive (Dau et al., 2019) and the UEA multivariate archive (Bagnall et al., 2018). Even though the archives were designed to evaluate supervised classification methods, no benchmark is available yet to evaluate specifically clustering methods. Moreover, these archives are already often used in the time series clustering context (Ma et al., 2019; Madiraju et al., 2018; Paparrizos and Gravano, 2015)

We used the extended version of the UCR archive with 128 datasets and the UEA archive with 30 datasets, both available online[11]. They contain a large number of datasets from different domains. The datasets are grouped into different categories with the main ones being Image outlines, Sensor Readings, Motion Capture, Spectrographs, ECG, Electric Devices, Audio, and Simulated Data. All datasets are split into train and test sets. The number of sequences in train sets goes from 3000 to 12 sequences and from 20000 to 15 in test sets. Some datasets have different time series lengths, however, we use the equaled length versions provided in the archive. In this version, all series are zero-padded at the end. The length varies from 3000 to 2 time steps but with a median length of 218. Also, for each dataset, a reference data is provided. The number of classes per dataset varies from 2 to 60, with a median at 4 (49 of them have only 2 classes). For multivariate datasets, the number of features varies between 2 and 1345. Previously, all UCR datasets were z-normalized but some are now provided without any preprocessing. To simplify the evaluation we decided to perform a z-normalization on all datasets.

Besides offering a variety of datasets, these two archives have been subject to numerous use (Fawaz et al., 2019; Franceschi et al., 2019; Ma et al., 2019; Madiraju et al., 2018; Xiao et al., 2020; Zhang et al., 2018). It allows a better comparison with other methods.

4.2 Evaluation protocol

All combinations are trained with 1000 batch iterations for the training phase and also 1000 for the clustering phase. After the training phase, a first DNN is obtained (the clustering loss *None*). This DNN is then used as initialization for all other clustering losses, with the exception of *ClusterGAN* and *VADE* that are trained from scratch.

The DNNS are trained (training phase + clustering phase) on the train set, and the clustering is performed on the test set, following the choice made in Ma et al. (2019) and Xiao et al. (2020) for time series deep clustering. This protocol insure that the latent space learned can be generalized.

[11] https://timeseriesclassification.com/index.php

To cluster the test set, most of the clustering loss models directly provides a clustering assignment. For the others, the clustering is performed with a K-Means on the test set projected in the latent space (i.e. the encoded features).

To evaluate the clustering performance, we use the Normalized Mutual Information (NMI), as it is the most common metric used for deep clustering (Ghasedi Dizaji et al., 2017; Guo et al., 2017b; Ma et al., 2019; Xie et al., 2016; Yang et al., 2019; Zhang et al., 2018) and as it also takes in consideration the expected distribution contrary to measure such as the Rand Index or the Clustering Accuracy.
The NMI is computed between a partition of M groups $A = \{A_1, \dots, A_M\}$ and a partition of M' groups $B = \{B_1, \dots, B_{M'}\}$ by the following formula:

$$NMI = \frac{\sum_{i=1}^{M} \sum_{j=1}^{M'} N_{ij} log \frac{N.N_{ij}}{|A_i||B_j|}}{\sqrt{(\sum_{i=1}^{M} |A_i| log \frac{|A_i|}{N})(\sum_{j=1}^{M'} |B_j| log \frac{|B_j|}{N})}}, \tag{41}$$

where $N_{ij} = |G_i \bigcap A_j|$. The value varies between 0 and 1, where the distributions are identical when the value is equal to 1.

Each DNN combination is trained 5 times. We use the NMI score mean of these 5 runs for the evaluation. These processes were run on a cluster of more than 60 GPU composed of GTX 1080Ti, Tesla P100, K20, K40, and K80. Furthermore, it should also be mentioned that some combinations have some convergence problems that led to vanishing or exploding gradients. These combinations are essentially leading to poor results. However, some combinations were excluded from the evaluation when too many datasets without any results were obtained. A limit of 10 datasets without results for the univariate archive and 3 for the multivariate archive was set. It only happened for combinations with *FCNN* architecture and *triplet* pretext losses on the multivariate archive (a note is added in the reported result). All the detailed results with NMI but also Adjusted Rand Index (ARI) and Clustering Accuracy measure are also reported on our git repository [12].

Finally, we want to evaluate the overall performance of each combination and evaluate if the difference with other combinations is significant or not. For the comparison, we use the average win/loss rank. Following the recommendation in Dau et al. (2019), we use the pairwise Wilcoxon signed-rank tests (Wilcoxon, 1992) and form cliques using the Holm correction (Holm, 1979) to determine the critical difference between each combination with a significant level $\alpha = 0.05$. To visualize these comparisons we use a critical difference diagram proposed by Demšar (2006), where a thick horizontal line shows the clique computed previously. This method of comparison has some limitations as the rank may not reflect the overall robustness of a method, especially when the number of datasets and compared methods is high. However, in our experiments, the results gave information that was correlated to the observed individual results. Note also that both the ranking and the Wilcoxon test handle missing values without skewing the outcome.

[12] https://github.com/blafabregue/TimeSeriesDeepClustering

5 Results

In this section, we present the results obtained by running all valid combinations. We first present the evaluation between all combinations in Sec. 5.1, then we show the effect of other parameters on performances in Sec. 5.2. Finally, we compare the best combinations to standard non-deep approaches in Sec. 5.3.

5.1 Cross-comparison on deep combinations

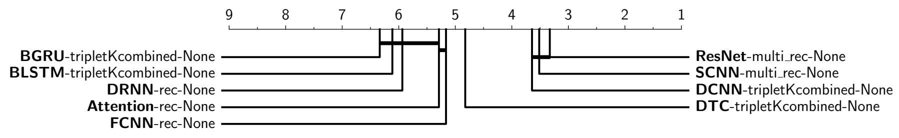
In order to better compare the performance of each choice, we compare each element (architecture, pretext loss, and clustering loss) separately. For each element, we pick the best combination that includes it (e.g. the best combination that use *FCNN* architecture) according to the average win/loss rank on the train set. Then, we compare the best candidates for the architecture, the pretext loss, and the clustering loss separately on the test set.

The results for the univariate archive are reported in Figs. 8 and 9 for the multivariate. The best average NMI score is obtained by the *ResNet-mulit_rec-None* combination with an average NMI score of 0.349 for the univariate archive and by the *DCNN-mulit_rec-SDCN* for the multivariate with 0.356 (this last value is approximated because two datasets could not be clustered with this combination).

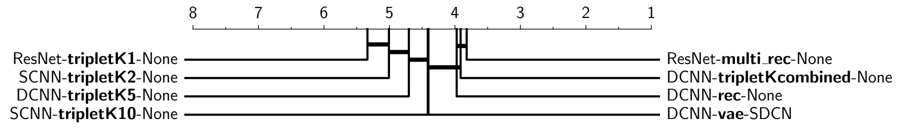
The first thing that comes out of these results is that no method outperforms others with a critical difference, especially if we take into account both archives. However, the following observations can be made:

- On architectures: when looking at the univariate results, the CNN based architectures outperform all other types. On the multivariate datasets, the *DRNN* architecture gives also good results, but the *DCNN* and *SCNN* are also performing well.
- On pretext losses: the reconstruction based loss (*rec* and *multi_rec*) and the *triplet* loss combined are the one that give the best results on univariate archive. However, the *triplet* loss seem to be not consistent on multivariate archive. Moreover, if we report the average NMI over all univariate datasets, we obtain for the *DCNN* architecture respectively 0.328, 0.329 and 0.339 for the *triplet_combined*, *multi_rec* and *rec* losses. But the difference greatly increase with 0.137, 0.343 and 0.339 for the multivariate ones.
- On clustering losses: surprisingly, no addition of a clustering loss results in a gain in performance at the exception of the *SDCN* loss for the multivariate archive. Therefore, the obtained results tend to not justify the additional computational time and complexity required by the use of a clustering loss.

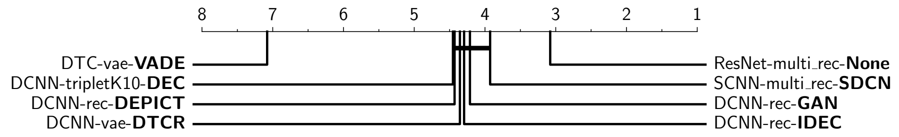
It can also be pointed out that every element, taken separately, achieves to obtain the best NMI score on at least one dataset. For example, the *BGRU* architecture obtains the best score on the univariate CBF dataset (with 0.71



(a) Best for each architecture



(b) Best for each pretext loss



(c) Best for each clustering loss

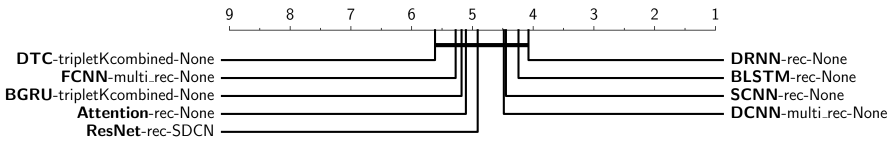
Fig. 8 Results for univariate time series with NMI measure

average NMI) even so this architecture obtains the lower win/loss rate. Hence, it can be interesting to look for relations between some element types and their performance on datasets.

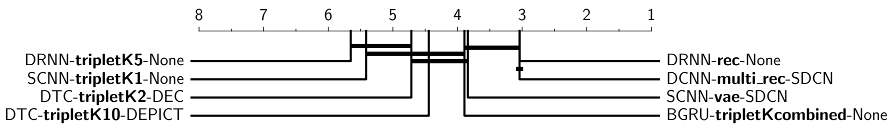
However, if each combination achieves some good results on some dataset, the global comparison of individual combinations shows that the best combination clearly outperforms the others on most of the datasets. For example, we have plot the pairwise NMI score comparison of different combination in Fig. 10. These three plots show that, even if one combination performs significantly better than the other, the performance differences for each dataset only spread on one side of the identity line (the low performing combination performs poorly compared to the best preforming one). Therefore, it seems that there are no “specialisations”, i.e. best performing combinations get better results on all datasets, not only on a subset of them. Note that we have only plot pairwise comparison for the univariate archive, as the limited number of datasets makes it less significant.

Furthermore, we conducted a more in depth analysis that did not lead to any correlation, more details are reported in Sec. 6.3.

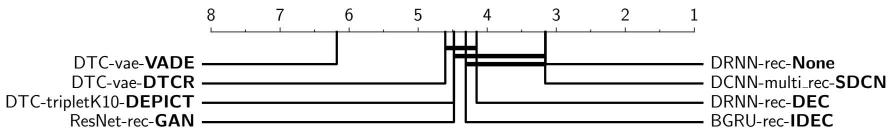
It should also be mentioned, that clustering is often considered to be an ill-posed problem (Jain, 2010). Multiple partitioning for each dataset may be considered relevant. This issue is particularly illustrated in the univariate archive, where some datasets have identical data but with different labeling. For example, the datasets GunPointAgeSpan, GunPointMaleVersusFemale,



(a) Best for each architecture



(b) Best for each pretext loss*



(c) Best for each clustering loss

Fig. 9 Results for multivariate time series with NMI measure (*: the *FCNN* combined with *triplet* based losses were excluded because they result in too many error of computation).

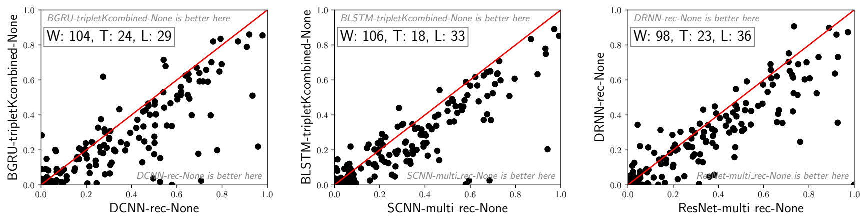


Fig. 10 Pairwise NMI score comparison of different deep clustering combination on both univariate and multivariate archives with win, loss, and tie scores. From left to right : *BGRU-tripletKcombined-None* vs *DCNN-tripletKcombined-None*, *BLSTM-tripletKcombined-None* vs *SCNN-multi_rec-None*, *DRNN-rec-None* vs *ResNet-multi rec-None*

and GunPointOldVersusYoung, that record actors' motions, refer to the same records but are respectively aiming to discriminate between the gesture type, the gender of the actor, and the older versus recent record. Identically, the DodgerLoopGame and DodgerLoopWeekend use the same data on traffic concentration but aim to discriminate between a day with a game at a stadium and no game for the former, and between weekday and weekend for the latter. Hence, no unsupervised method can perform well on a dataset without having poor results on the other(s). Moreover, the observed standard deviation over all methods is non-negligible. The NMI standard deviation goes from 0.010 to 0.170 with a median at 0.060. This can be observed with all combinations. This may suggest that all methods tend to fall in local minima. More insight will be given in Sec. 6.1 where we discuss the NMI evolution through the training process.

Finally, another point of comparison is the number of failure encountered by the different combinations. This led to exclude the *FCNN-triplet* combinations for the multivariate comparison on clustering losses. Overall, any architecture and the pretext loss are robust and lead to almost no failures (with 1% failure rate for the univariate archive, and 6% for the mutivariate one), at the exception of the *FCNN-triplet* combinations on both archives (with 35% failure rate for the univariate archive, and 82% for the mutivariate one). For the use of clustering loss, most of the clustering losses give robust results, with almost no difference with the pre-trained model. However *SDCN* (with 12% failure rate for the univariate archive, and 24% for the mutivariate one) and *DTCR* (with 5% failure rate for the univariate archive, and 18% for the mutivariate one) methods lead to more failures. This is explained by the batch size that is fixed to the size of the dataset, which leads to memory errors. Detailed statistics are available on our github repository [13].

5.2 Other parameters influence

Even though the previous results cover different variations of DNNs, other parameters are used in the literature when training DNNS for clustering. It is standard in supervised classification to tune the choices of these parameters, with grid search for example, especially for the hyperparameters (e.g. the size and number of layers, the optimizer, the learning rate, etc). Unfortunately, it is impossible to conduct such optimization for each dataset in an unsupervised context as no train set can be used.

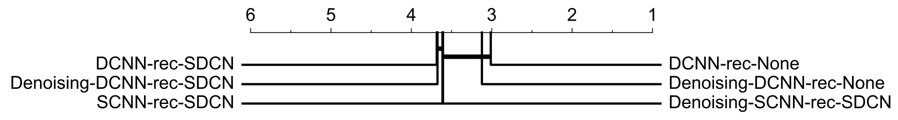
In this section, given the high number of compared methods, we have decided to report the effect of a small selection of these parameters. We have selected parameters or additional processing that are often used in clustering methods.

However, note that multiple other parameters have shown to greatly influence the performance of deep clustering methods (e.g. type of optimizer, size of layers, etc.).

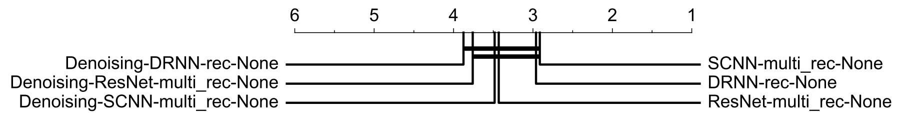
5.2.1 Denoising

Denoising autoencoders (DAE) are often used as pretext loss (Ghasedi Dizaji et al., 2017; Guo et al., 2017b; Xie et al., 2016) (see Sec. 2.4.2 for more details). We used masking noise as it is the most used in the selected methods (Ghasedi Dizaji et al., 2017; Guo et al., 2017a; Xie et al., 2016). The

[13] in https://github.com/blafabregue/TimeSeriesDeepClustering/blob/main/paper_results/ folder



(a) Best of denoising combinations on univariate archive



(b) Best of denoising combinations on multivariate archive

Fig. 11 Comparison of performance for DNNs trained with denoising and without with NMI measure

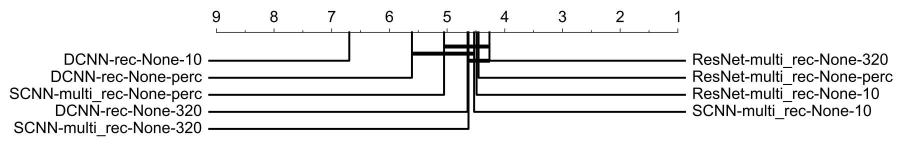
masking noise is generated with dropout layers with a dropout rate of 20%. Note that this method is designed for AE frameworks and therefore will only be applied with *rec* and *multi_rec* losses. We have reported the three top-ranked combinations with denoising and compared them to their version without it in Fig. 11.

The reported results show that the denoising either degrades the results or has a really small effect on the performance. When looking at individual results on each dataset it can also be noticed that the improvements tend to be very small. But at the opposite it may also lead to important performance degradation. This may be explained by the effect of the noise on time series, where a high variation of the signal may more easily create confusion between classes. Moreover, for 1D-convolution DNNs the information on the neighborhood may be more limited than 2D-convolutions due to the filters' size and their 1D nature. Indeed we only have 2 immediate neighbors in 1D for 8 in 2D. This may alter the DNNs' capacity to discriminate between noise and real signal.

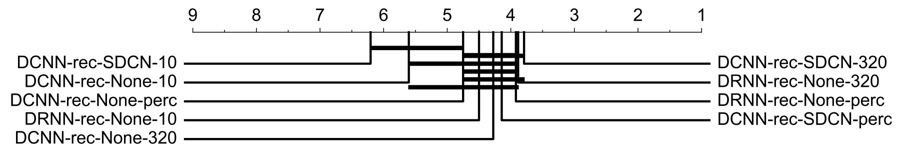
5.2.2 Size of the latent space

It is often recommended to have a latent space with a significantly smaller number of dimensions than the one in the original data space. This aims to force the DNN to retains only meaningful features for the reconstruction or the selected pretext task. Therefore, we want to test the effect on time series data.

Given the computation time required to launch all combinations we only launched three different options of clustering size. The latent space's number of features is fixed to either 10, 320, or 10% of the time series length (noted *perc*). Thus, we can compare the effect of a small latent space, a large latent space, or one adapted to the dataset. The results are displayed in Fig. 12.



(a) Best of latent dimension size on univariate archive



(b) Best of latent dimension size on multivariate archive

Fig. 12 Comparison of DNNs' performance trained with different size of latent dimension, 10, 320 or perc (10% of time series length) with NMI measure

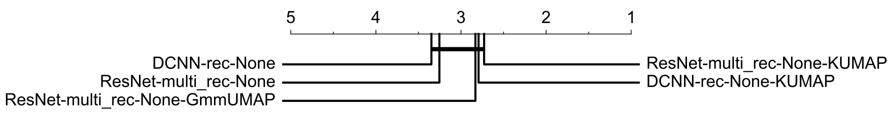
The choice of a large latent dimension seems more relevant, especially for the *DCNN-rec-None* architecture for univariate and *DCNN-rec-SDCN* multivariate. For the latter, it manages to obtain the best average ranking for the 320 features version and the worst with its 10 version with a significant difference. The difference between the 10 and 10% features versions seems more relative as the 10% version mostly tends to have a latent dimension size around 20. Hence, this version is more related to a small latent dimension size.

However, these general observations hide a highly variable behavior among datasets. In the case of the *DCNN-rec-None* architecture and the Chinatown dataset, the 10 features version obtains the best score with 0.69 against 0.53 for size 320. This can also be observed for other datasets like GesturePebbleZ2 or Trace, with respectively an NMI gain of 0.13 and 0.10. Hence, even if a large latent dimension size seems a good choice by default, it may strongly minimize the DNN performance in some cases.

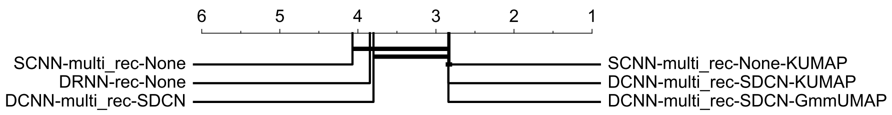
This limited comparison already highlights that hyperparameters modification may result in major modification of the DNNs' capacity to extract features. In a non-supervised context, this may strongly mitigate the application of such methods. It is even more problematic if we take into consideration the number of possible hyperparameters combinations for DNNs (e.g. number of layers, size and number of filters for CNN, learning rate options, ...).

5.2.3 Dimension reduction

This section is motivated by the work presented in McConville et al. (2019). The authors propose to use a dimension reduction method on the latent space before applying the clustering method. The reduction dimension methods proposed are Isomap (Tenenbaum et al., 2000), t-SNE (Maaten and Hinton, 2008), Linear Embedding (LLE) (Roweis and Saul, 2000) and UMAP (McInnes et al.,



(a) Best of reduction dimension with comparison to only K-Means on univariate archive



(b) Best of reduction dimension with comparison to only K-Means on multivariate archive

Fig. 13 Comparison clustering performance performed either directly on the learnt representation or after a dimension reduction (UMAP) with different clustering methods (GMM an K-Means) with NMI measure.

2018). They set the number of outputted dimensions to the number of searched clusters, K. They also propose to replace the K-Means method with either Spectral clustering or Gaussian Mixture Model (GMM) approaches. Note that in the reported results, the combination of K-Means method with the UMAP is noted as KUMAP. We have tested the different combinations and reported the three top-ranked combination with reduction dimension and compared them to their version without it in Fig. 13. We have also added the best ranked combination without dimension reduction for the multivariate archive (*DRNN-rec-None*) in the two diagrams for comparison. For the univariate archive, the *DRNN-rec-None* is already in the top three. The results were obtained based on the author's code [14].

In McConville et al. (2019), the authors observed that the UMAP reduction dimension in combination with the GMM clustering method reaches the best performance. In our case, UMAP also improves the clustering performance. For all the other dimension reduction methods we observe a degradation of the results. In our case, K-Means gave slightly better results than GMM but without a critical difference. However, it still tends to confirm the observations in McConville et al. (2019). It should also be mentioned that no critical difference is reported between results with and without reduction dimension with the Holms correction. But the Wilcoxon test reports a difference with $p < 0.02$ between clustering with and without UMAP when compared individually. On the global NMI average the *ResNet-multi_rec-None-KUMAP* combination obtain 0.399 against 0.356 without UMAP. For the multivariate, this goes from 0.417 for *DRNN-rec-None-KUMAP* to 0.348 without UMAP. Overall, the use of UMAP seems to be a consistent tool to improve clustering performance.

[14] https://github.com/rymc/n2d

5.3 Comparison to non-deep methods

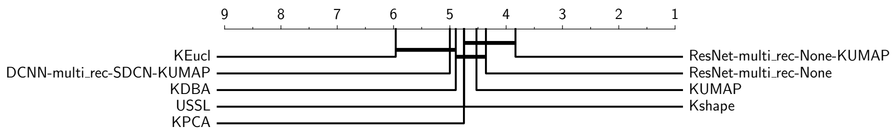
Even if this paper aims to evaluate different deep clustering against each other, we also want to position these methods among other classical clustering methods for time series. To do so, we have selected the following non-deep methods:

– *KEucl*: K-Means method with Euclidean distance and arithmetic mean to compute centroids. We use the `tslearn` implementation [15] with a maximum iteration of 200.
– *KDBA*: K-Means method with DTW (Dynamic Time Warping) (Sakoe and Chiba, 1978) measure and DBA (DTW Barycenter Averaging) Petitjean et al. (2011) to compute centroids. We used the `tslearn` implementation [15] with a maximum iteration of 200.
– *KPCA*: K-Means method with Euclidean distance and arithmetic mean to compute centroids. However, in this case, we perform a Principal Component Analysis (PCA) with a reduction to K dimensions before applying the K-Means algorithm. We used the `tslearn` implementation for the K-Means, and `sklearn` for PCA [16].
– *KUMAP*: K-Means method with Euclidean distance and arithmetic mean to compute centroids. However, in this case, we perform a UMAP with a reduction to K dimensions before applying the K-Means algorithm. We used the `tslearn` implementation for the K-Means with a maximum iteration of 200, and `umap-learn` for UMAP [17].
– *Kshape* (Paparrizos and Gravano, 2015): k-shape is a method that relies on a scalable iterative refinement procedure to extract cluster base on the cross-correlation measure. We used the author's Python implementation.[18]
– *USSL* (Zhang et al., 2018): Unsupervised Salient Subsequence Learning is a method that extracts shapes to obtain some characteristics. For this method, we use the results reported in the supplementary materials of Ma et al. (2020) on the 85 datasets UCR version.

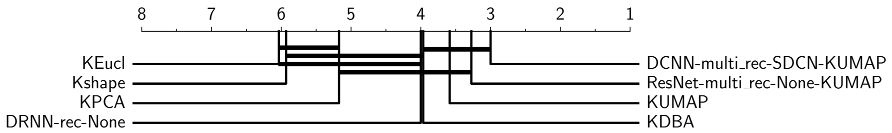
The comparison is showed in Fig. 14. For both archives, we have selected the best candidate based on NMI results on the train set with and without UMAP, note that the displayed diagram are still computed on the test set and result in similar ranking. We have also added the UMAP best candidate of each archive in the other archive's plot to evaluate the robustness of these methods on the two archives. For the *USSL* and *Kshape* methods, we did not report the multivariate results as it was not included in Ma et al. (2020) for *USSL* and as the *Kshape* authors' code does not support multivariate time series.

For both archives, the deep clustering best candidate with the use of UMAP is ranked first. It should also be noted that the *KUMAP* is in the top three ranked methods on both archives. This highlights the benefit of using UMAP before the clustering task independently of the use of a deep transformation

[15] https://tslearn.readthedocs.io/en/stable/gen_modules/clustering/tslearn.clustering.TimeSeriesKMeans.html
[16] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html
[17] https://pypi.org/project/umap-learn/#description
[18] https://github.com/johnpaparrizos/kshape



(a) Baseline comparison on univariate archive



(b) Baseline comparison on multivariate archive

Fig. 14 Results of deep clustering methods compared to non-deep methods with NMI measure

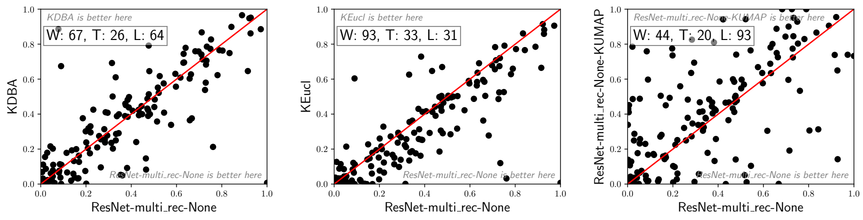


Fig. 15 Pairwise NMI score comparison of deep and non-deep clustering combinations on both univariate and multivariate archives with win, loss, and tie scores. From left to right : *KDBA* vs *ResNet-multi_rec-None*, *KEucl* vs *ResNet-multi_rec-None*, *ResNet-multi_rec-None-KUMAP* vs *ResNet-multi_rec-None*

	KEucl	*KDBA*	*KPCA*	*KUMAP*	*Kshape*	*USSL*	*Deep*	*Deep+KUMAP*
KEucl	_	6.3e-3	0*	0*	1.5e-5	0*	0*	2.6e-5
KDBA	6.3e-3	_	0.600	0.184	0.750	0*	0.317	2.8e-3
KPCA	0*	0.600	_	0.090	0.794	0*	0.118	5.6e-3
KUMAP	0*	0.184	0.090	_	0.420	0*	0.478	0.018
Kshape	1.5e-5	0.750	0.794	0.420	_	0*	0.528	0.012
USSL	0*	0*	0*	0*	0*	_	0*	0*
Deep	0*	0.317	0.118	0.478	0.528	0*	_	7.4e-3
Deep+KUMAP	2.6e-5	2.8e-3	5.6e-3	0.018	0.012	0*	7.4e-3	_

Table 2 p-values obtained on the Wilcoxon test on univariate archive. The method "Deep" correspond to the *ResNet-multi_rec-None* combination. Red indicats that the methods did not pass the Wilcoxon test with Holms correction. *: values bellow 1e-5 are rounded to 0.

or not. However, the deep latent space seems to benefit more from UMAP than the original space. Even though *KUMAP* is ranked before other non-deep methods, it remains lower than the best deep clustering candidate with a confidence $p < 0.02$ at the Wilcoxon test for the univariate archive . However, for the multivariate archive, the difference does not pass the Wilcoxon test (with $p = 0.10$). Moreover, for the multivariate archive, the deep clustering combination alone performs significantly worse than *KUMAP* method, confirming that it is the combination of UMAP and the deep embedding that allows obtaining this score on the multivariate archive. Note that the average NMI score for *KUMAP* is 0.352 for the univariate archive and 0.390 for the multivariate one.

Overall the *ResNet-multi_rec-None-KUMAP* combination obtains the best ranking if we take in consideration both archive. But the gain obtained from using DNNs remains small and difficult to generalize as it does not pass most of the Wilcoxon test with the Holms correction (see Tab. 2).

Moreover, when analyzing pairwise comparison plotted in Fig. 15, we can observe a large spread of performance indicating that each method is relevant to some datasets. For example, *KDBA* obtains a number of wins when compared to the *ResNet-multi_rec-None* combination, even the *KEucl* performs better on a fifth of the datasets. Consequently, current deep clustering methods should be seen as alternative methods and not as replacements to other non-deep methods.

5.4 Execution time

In this final evaluation's section, we have reported the average, the median, the minimal and the maximal execution times of deep clustering combinations' training compared to other non-deep methods in Fig. 3. We have only reported one combination per architecture, as it is the only one that have a major impact on the execution time. The choice of pretext loss has a really minor impact (less than a factor of 1.4 on all combinations). The execution time addition from the clustering loss is on average very small (less than 20% of the pre-training execution time), at the exception of the *SDCN* loss that results in a similar execution time (doubling the total execution time). The deep clustering methods were trained on a GPU 1080Ti (with 11.1Go RAM) and non-deep methods on an Intel Skylake (2x12 cores, with 96 Go RAM).

The reported results in Fig. 3 show that deep clustering methods take a considerably larger amount of time to execute compared to non-deep methods. The training process can go up to a full day for some combinations with only considering the pre-training (so around two days with *SDCN* clustering loss). However, the execution time can be reduced by decreasing the number of batch iterations. In general, RNN models result in longer execution time compared to CNN models and even more when compared with the FCNN model. The time series' length is the main factor that increases the execution time, as the number of batches is fixed. Finally, it can be noted that *KShape* and *KDBA* can also lead to memory usage problems when dealing with large datasets. For example, the *KDBA* method took around two days to execute on the EigenWorms multivariate dataset on a Intel Xeon E7-8891 (38 cores with 250Go of RAM).

Method	Median	Average	Minimal	Maximal	Standard deviation
Attention-rec-None	3160	6958	278	45391	8710
BGRU-rec-None	2156	3101	245	39391	4503
BLSTM-rec-None	2912	3956	251	45391	5125
DTC-rec-None	1987	2787	208	38666	4141
DRNN-rec-None	3442	9276	946	85043	85043
DCNN-rec-None	1647	4086	294	34196	5746
SCNN-rec-None	848	1927	171	15561	2552
RestNet-rec-None	1619	3430	279	26963	4303
FCNN	638	1708	139	9217	2092
KEucl	< 1	<1	< 1	20	2
KPCA	< 1	< 1	< 1	2	< 1
KUMAP	9	12	5	75	10
KDBA	39	658	1	9247*	1700
KShape	2	36	< 1	433*	75

Table 3 Execution time comparison between some selected combinations of deep clustering methods and non-deep methods in seconds on univariate and multivariate archives. *: 3 datasets could not be clustered with KDBA and KShape because of memory usage

6 Analysis of DNNs

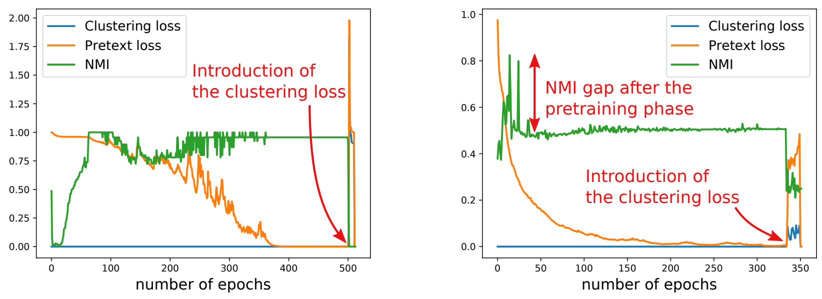
In this section, we aim to give more insight into the DNNs training and the latent space obtained through different aspects.

6.1 DNNs training and clustering task

As we do not train the DNNs to directly predict labels, the correlation between the loss and the clustering performance evolution may not be verified. In Fig. 16, we have plotted the evolution of the NMI score on the test set compared to the evolution of the pretext and clustering losses.

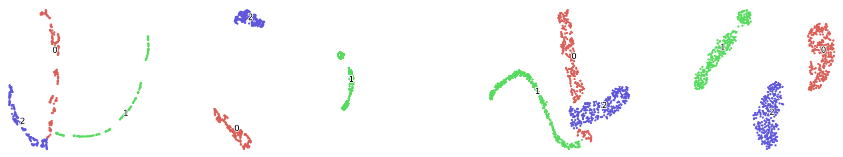
For recall, the first 1000 batches are trained without the clustering loss, for the ShapeletSim dataset this corresponds to 500 epochs and 334 for CBF. The combination of the clustering and pretext losses are also launched with 1000 batches but, for both DEC and IDEC they include a stopping criterion that fires when the clustering does not change from one iteration to the other, explaining the early stop.

On both plots, it can be observed that the NMI is not stable and may finish below its maximum value. Also, we can see that the addition of the clustering loss tends to highly disturb the latent representation. This often results in lowering the performance of the learned representation, resulting in a 0.0 score of NMI in Fig. 16a. This observation can be generalized to almost all datasets and DNN combinations. This can be confirmed by measuring the gap between maximal and final NMI at each run. For both archives this gap is significant. For the univariate archive, the average gap among all combinations ranges from 0.029 to 0.256 points of NMI score. For the two best methods, it is of 0.078 for *ResNet-mulit_rec-None* and 0.073 for *DCNN-rec-None*. For the former, this implies a drop of the NMI average score from 0.434 to 0.356. Even



(a) *ResNet-rec-IDEC* on ShapeletSim (b) *DCNN-rec-DEC* on BME

Fig. 16 Evolution of NMI and losses in the parameters training (the last epochs correspond to the addition of the clustering loss)



(a) UMAP (b) t-SNE

Fig. 17 Projection to two dimensions of CBF dataset using different methods between Raw (left) and DNN's Latent space (right) with either UMAP or t-SNE. The latent space is obtained with *DCNN-rec-None*.

though it does not seem realistic to find a way to reach this maximum score, it still shows the potential DNNs' capacity to create representation suitable for the clustering task. Moreover, we can see in Fig. 16a that if we stop the process around epoch 250, we can have a highly different result from one epoch to the other.

On some datasets, this capacity can be visually confirmed. To do so, we project the datasets into 2 dimensions. To perform the projection we use two dimension reduction methods, UMAP, and t-SNE. For each method, we have plotted the projection with the original data (without any preprocessing) and the data in the latent space learned by the DNNs. The projections are reported in Figs. 17, 18 and 19.

The representation learned on both CBF and ShapletSim datasets clearly illustrates the gain to use the DNNs on them. This is even clearer for the latter where the clusters could not be distinguished with the raw data. However, DNNs does not always lead to such good representations. For the TwoPatterns dataset, even though the data form more distinguishable groups, each class remains split into multiple groups, making the clustering task difficult. The K-Means clustering on both the original space and the latent space resulted

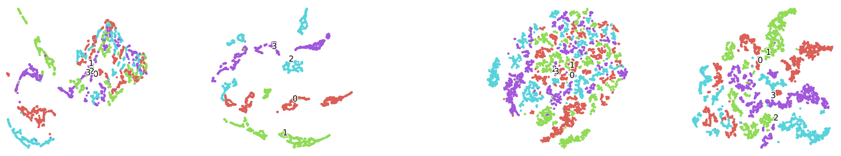

(a) UMAP (b) t-SNE

Fig. 18 Projection to two dimensions of TwoPatterns dataset between Raw (left) and DNN's Latent space (right) with either UMAP or t-SNE. The latent space is obtained with *DCNN-rec-None*.

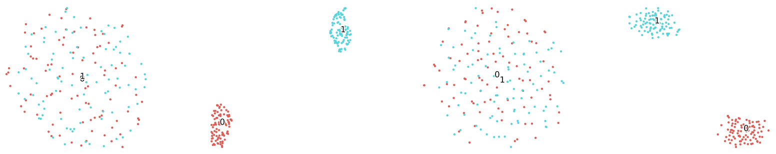

(a) UMAP (b) t-SNE

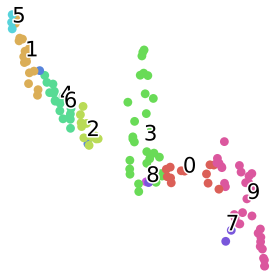
Fig. 19 Projection to two dimensions of ShapeletSim dataset between Raw (left) andDNN's Latent space (right) with either UMAP or t-SNE. The latent space is obtained with *ResNet-multi_rec-None*.

in a poor result (around 0.02 NMI score). From the plots that we analyzed, the DNNs' latent space tends to create more dense and separable groups in the data, but these groups may not necessarily match the expected partition.

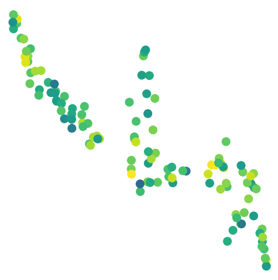
6.2 DNNs types and datasets correlations

Find correlations between an element type (e.g. the use of *RNN* DNNs) and some datasets characteristics would greatly help the user to select the best combination for its dataset.

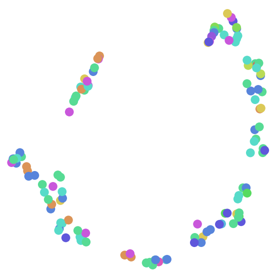
To do so, we have plotted all archives' datasets into a two-dimensional projection. Each dataset is represented by a vector composed of the performance obtained by a set of DNN combinations on this dataset (indicated by "*Data:*" in Fig. 20). We have tested different combinations (e.g. the top combinations of each architecture/pretext loss/clustering loss, all combinations, all architectures with a specific pretext loss and without clustering loss). This representation is then projected with the UMAP method along two dimensions and is colorized with another dataset property (indicated in by "*Coloring:*" in Fig. 20). With this process, we aim at identify some groups/clusters with homogeneous coloring and therefore identical behavior. We have tested the following datasets' properties: the time series length, the dataset category (e.g. ECG, Spectrography, Image, etc.), the size of the train set, the size of the test set, and the number of classes. However, the results were not conclusive. Some of the plots are reported in Fig. 20. For example, in Fig. 20b the datasets are represented by the set of performance obtained on all architectures combined with the reconstruction loss and without clustering loss, and colorized with the time series length (a lighter color meaning a longer length, and a darker one a shorter length).



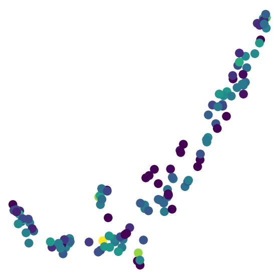
(a) *Coloring*: clustering, *Data*: all architecture with reconstruction loss and no clustering loss



(b) *Coloring*: time series length, *Data*: all architecture with reconstruction loss and no clustering loss



(c) *Coloring*: dataset category, *Data*: top 3 combination for each architecture with and without UMAP



(d) *Coloring*: Size of training set, *Data*: *DCNN* and *ResNet* with all pretext losses and no clustering loss

Fig. 20 Each plot shows univariate datasets projected into two dimensions obtain with UMAP. The value of each dataset is computed from the performance of a set of combination described in *Data* and colorized using the criterion described in *Coloring*

Overall, we can see that no trend can really be observed in the plots. For all plots, the main factor of separation corresponds to the average performance across all combinations. In Fig. 20a, we applied a clustering with K-means method on the set of datasets to try to see some similar behavior among some dataset. Most of the clusters regroup homogeneous results, with good, bad, or average performance on all selected combinations. Therefore it is difficult to find trends and to decide which combination will be the most adapted for a new dataset. Some clusters, like the number 8 and 6 group datasets where CNN architectures perform significantly better than others. However, they are totaling only 5 datasets (over 128) with no particular similarity.

The actual conclusion from this data is the difficulty to find a consistent correlation between models' performance and datasets' properties. However, it should be recalled that these two archives regroup a large variety of datasets that may have highly unrelated features/patterns types.

6.3 What patterns are learned by the DNNs ?

Even though the users desire to find relevant clusters, they often want to know how the method took its decision and also what are the discriminating patterns in the data. To achieve this, we used both the decoder to reconstruct the clusters' centroids and the Class Activation Map (CAM), introduced in Zhou et al. (2016). But first, we need to explain how we use CAM in an unsupervised framework.

6.3.1 CAM for clustering

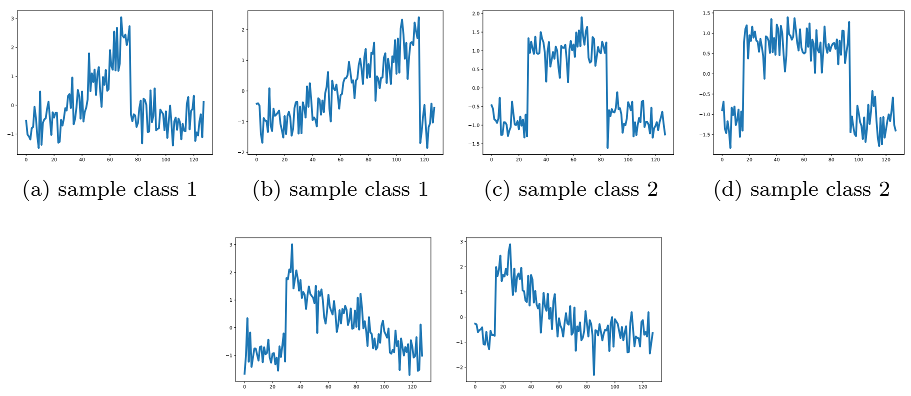
CAMs are used for supervised DNNs to highlight which part of the data is used to identify an object's (i.e. a time series in our case) class. This method relies on the presence of a Global Pooling and the classification softmax layer, but softmax layers are not used in our case. However, another version proposed in Selvaraju et al. (2017) used the gradient computation to compute the CAM.

For this method, the heatmap is computed with respect to a class, c. It uses the c^{th} value of the softmax layer, y^c, to compute the gradient value at the last convolutional layer's feature map A. The heatmap is computed for each A's weights a_k^c:

$$a_k^c = \frac{1}{D} \sum_{i=0}^{w} \sum_{j=0}^{h} \frac{\partial y^c}{\partial A_{i,j}^k} \tag{42}$$

where $D = h \times w$ is the input dimension. Note that in our case the input has only one dimension (i.e. the time dimension), but the equation is given for 2D-convolutions. Thus, we obtain the degree of activation for each layer's filter at each part of the input. The heatmap consists of the weighted sum of all the filter's output with their degree of activation:

$$L^c_{Grad_{CAM}} = ReLU(\sum_k a_k^c A^k) \tag{43}$$



(a) sample class 1 (b) sample class 1 (c) sample class 2 (d) sample class 2

(e) sample class 3 (f) sample class 3

Fig. 21 Examples of CBF dataset classes

We have modified this version, to not use the softmax layer and replaced the CAM's derivative computation with the following equation:

$$a_k^c = \frac{1}{Z}\sum_i\sum_j \frac{\partial \frac{1}{z * \|weight_{z,c}\|_1}}{\partial A_{i,j}^k}, \tag{44}$$

where $\|.\|_1$ is the normalization between 0 and 1, $. * .$ is the element wise multiplication, and $weight_{z,c}$ is computed as follow:

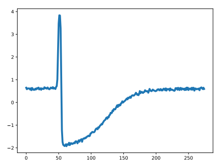
$$weight_{z,c} = \left| \frac{1}{\|z\|_z - \|\mu^c\|_z} \right|, \tag{45}$$

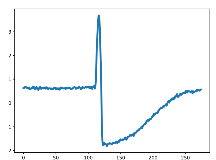
where $|.|$ is the absolute value, $\|.\|_z$ is the z-normalization, z is the latent representation and μ^c is the centroid of the time series' cluster.

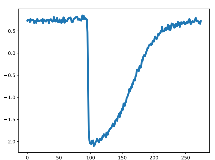
6.3.2 Does DNNs capture temporal patterns ?

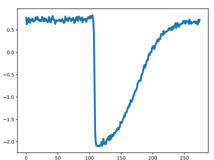
As explained in Sec. 2.2, we also want to evaluate the DNNs' capacity to take into consideration the specificity of the time dimension when clustering the data. It means recognize temporal patterns, even if they may be shifted or stretched. To illustrate that, we selected two datasets, CBF, which contains both stretched and shifted patterns, and Trace, which contains shifted patterns. They are illustrated in Figs 21 and 22. For recall, CBF is a synthetic dataset designed to discriminate between three shapes, Cylinder (class 2), Bell (class 1), and Funnel (class 3). Trace is a synthetic dataset designed to simulate instrumentation failures in a nuclear power plant, but we only have the class number.

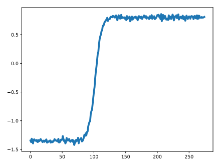
For comparison, we have plotted the centroids learned by K-Means with DTW metric and DTW Barycenter Averaging (DBA) in Fig. 23 for CBF

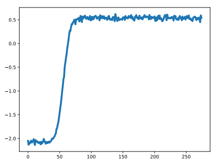

(a) sample class 1

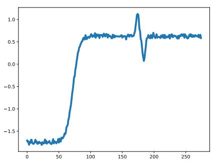

(b) sample class 1


(c) sample class 2


(d) sample class 2


(e) sample class 3


(f) sample class 3


(g) sample class 4

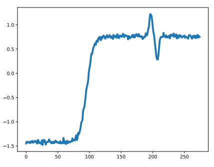
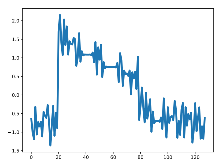
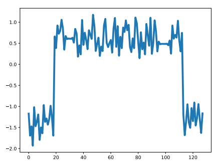

(h) sample class 4

Fig. 22 Examples of Trace dataset classes


(a) Centroid cluster 1


(b) Centroid cluster 2

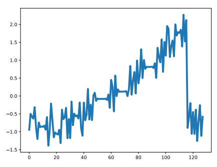
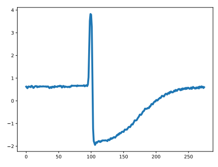
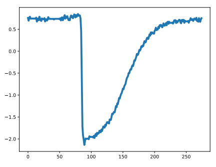
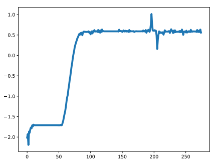

(c) Centroid cluster 3

Fig. 23 Centroids learned by K-Means with DTW and DBA on CBF dataset


(a) Centroid cluster 1


(b) Centroid cluster 2


(c) Centroid cluster 3

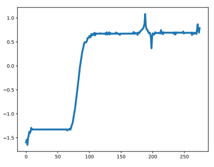
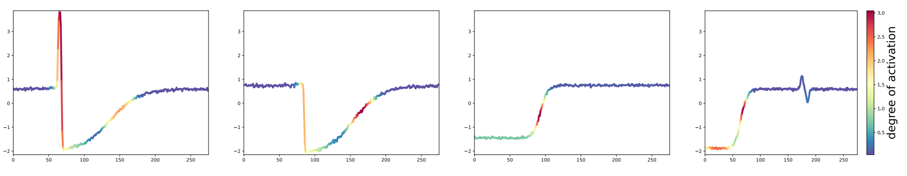
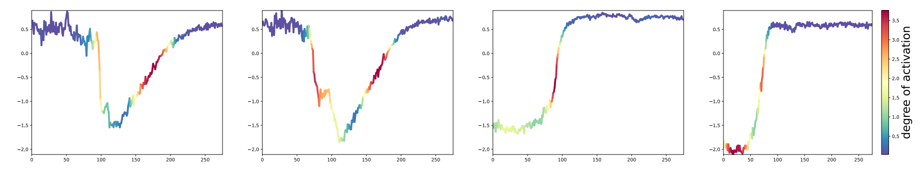

(d) Centroid cluster 4

Fig. 24 Centroids learned by K-Means with DTW and DBA on Trace dataset

dataset and in Fig. 24 for Trace dataset. DBA, and DTW are designed to be less sensitive to time distortions as they realign time series to minimize the distance between them. These two methods proved to work well on catching temporal patterns (Petitjean et al., 2011). For the CBF, we can observe that the DBA method clearly extract the three patterns. For the Trace dataset, DBA centroids also clearly identify the main patterns. However, confusion can be observed between the 3^{rd} and 4^{th} class. In the reference data, the two classes are distinguished by the presence of a final perturbation or not. The clustering actually discriminates the two clusters by the level of the first plateau, bellow 1.5 for the 3^{rd} cluster and over for the 4^{th} cluster.



(a) sample cluster 1 (b) sample cluster 2 (c) sample cluster 3 (d) sample cluster 4



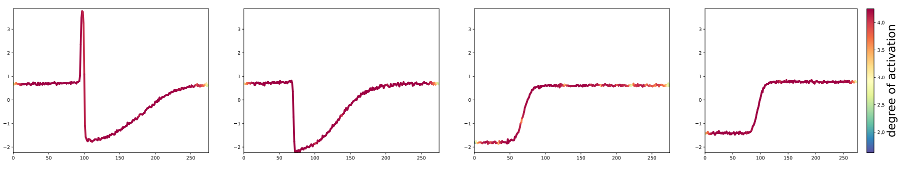
(e) Reconstruction centroid 1 (f) Reconstruction centroid 2 (g) Reconstruction centroid 3 (h) Reconstruction centroid 4

Fig. 25 Grad-CAM heatmap on Trace dataset with *ResNet-multi_rec-None* combination, where 25a, 25b, 25c and 25d are samples of each cluster and 25e, 25f and 25g are centroids reconstruction of each cluster. Red regions corresponds to high contribution and blue to almost no contribution to matching the centroid (colors are smoothed for visual clarity and better reflect filters' size)

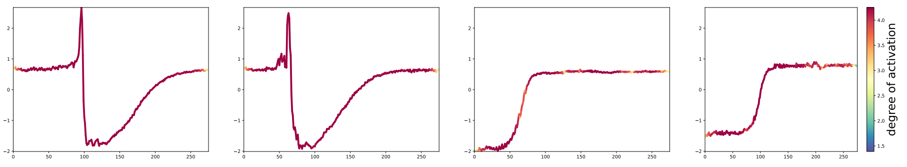
In Fig. 25 to 28 we have displayed the heatmap obtained for each cluster on respectively CBF and Trace datasets. We have also added the reconstruction of the centroids.

For the Trace dataset, the main patterns are identified by the heatmap with the exception of the final variation that discriminates between classes 3 and 4. Actually, the confusion is identical to the DTW/DBA one as it also focuses on the level of the first plateau as shown by the heatmap. Note that they obtain a similar NMI score (around 0.75). However, the reconstructed centroids for the DNNs do not render the detected patterns. This is especially the case for the class 1, even if the encoder perfectly distinguish the class 1 and 2 as the clustering obtain a perfect score on these two classes. Therefore, it is likely that the decoder fails to obtain a good reconstruction. For the decoder, it may be more optimal to not render the spike for the class 1. Indeed, when computing the mean square error, omit the spike in the reconstruction may give a better result than render it but at the wrong time step. In the first case, the spike will affect the error once. However, it is likely to affect it twice in the second case. Similarly, between the class 3 and 4, it might be more effective to focus on rendering the two plateaus correctly than render the small variation at the end.

In Fig. 26 we have also displayed the Trace Grad-CAM plots but with another combination, *DCNN-rec-None*. This combination obtains a lower NMI score (0.55). The heatmap activation is actually spread along the whole time series. When observing the clustering result carefully, it can be noticed that all the clustering is based on when the variation occurs but not its shape. The cluster 1 regroups the classes 1 and 2 when the variation happens late



(a) sample cluster 1 (b) sample cluster 2 (c) sample cluster 3 (d) sample cluster 4



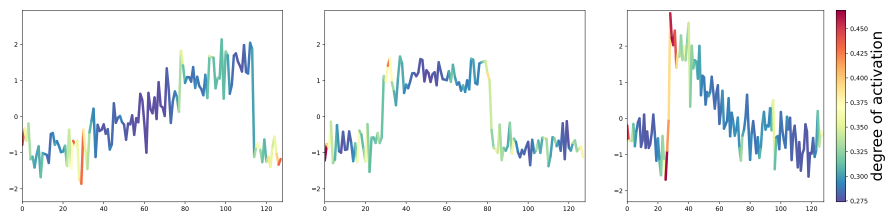
(e) Reconstruction centroid 1 (f) Reconstruction centroid 2 (g) Reconstruction centroid 3 (h) Reconstruction centroid 4

Fig. 26 Grad-CAM heatmap on Trace dataset with *DRNN-rec-None* combination, where 26a, 26b,26c and 26d are samples of each cluster and 26e, 26f, 26g and 26h are centroids reconstruction of each cluster. Red regions corresponds to high contribution and blue to almost no contribution to matching the centroid (colors are smoothed for visual clarity and better reflect filters' size)

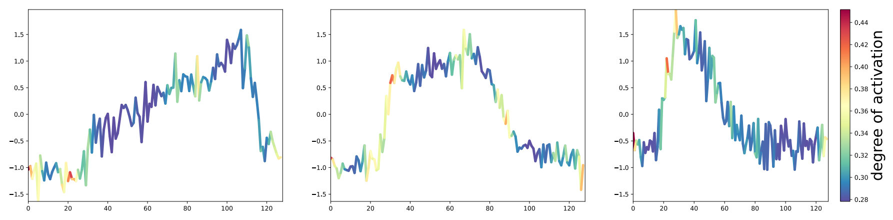
(around the 100^{th} time step) and the cluster 2 when it happens earlier (around them 60^{th} time step). This is similar for classes 3 and 4 with clusters 3 and 4. On this dataset, this combination behaves as the Euclidean distance (with a similar NMI score of 0.52) and completely misses the temporal patterns. One can notice that, in this case, rendering the pick may seem more relevant for the reconstruction, which may explain why it is visible here.

In Fig. 27, we now focus on the CBF dataset, which comprises time shifts but also time stretches and noise. Contrary to the Trace dataset, the Grad-CAM's heatmap activations are more spread along the whole series for CBF. They do not necessarily focus on the main parts but also partially on some noisy variations. On the other hand, the centroids' reconstruction is clearer, but it also shows that it can result in a partial confusion of classes 2 and 3. It results in an NMI score of only 0.35, mainly explained by the confusion of classes 2 and 3. This particular case shows that in order to obtain a good reconstruction, the encoder may need to add unnecessary information that will interfere with the clustering task.

To strengthen this first observation, we have plot again the grad-CAMs on CBF but we stopped the DNN's training after only a small number of iterations (10 epochs versus 334 before) in the Fig. 28. It can be observed that the heatmap is concentrated on the discriminating part of the time series on class 1 and 2. However, the centroids reconstruction is poor but it seems normal that the decoder struggles to immediately obtain a good reconstruction. However, the encoder seems able, in this case, to discriminate between the main signal and the noise in the times series. This DNN obtains an NMI score of 0.68, far

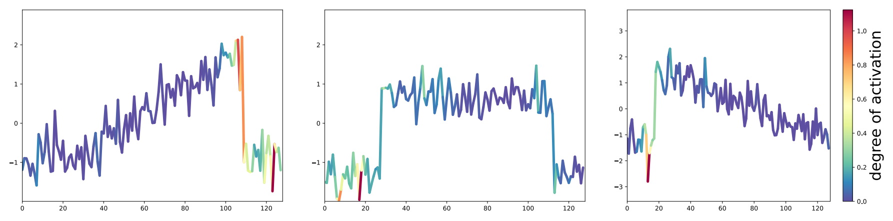


(a) sample cluster 1 (b) sample cluster 2 (c) sample cluster 3

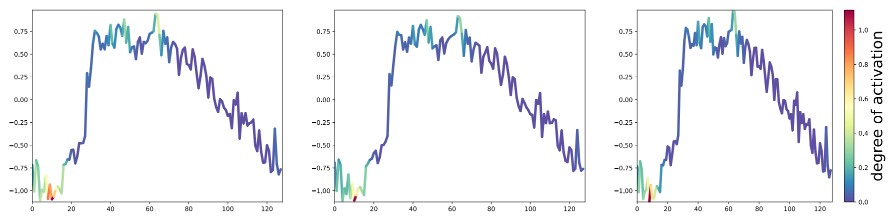


(d) Reconstruction centroid 1 (e) Reconstruction centroid 2 (f) Reconstruction centroid 3

Fig. 27 Grad-CAM heatmap on CBF dataset with *DCNN-rec-None* combination, where 27a, 27b and 27c are samples of each cluster and 27d, 27e and 27f are centroids reconstruction of each cluster. Red regions corresponds to high contribution and blue to almost no contribution to matching the centroid (colors are smoothed for visual clarity and better reflect filters' size)



(a) sample cluster 1 (b) sample cluster 2 (c) sample cluster 3



(d) Reconstruction centroid 1 (e) Reconstruction centroid 2 (f) Reconstruction centroid 3

Fig. 28 Grad-CAM heatmap on CBF dataset with *DCNN-rec-None* combination but with only 10 epochs, where 28a, 28b and 28c are samples of each cluster and 28d, 28e and 28f are centroids reconstruction of each cluster. Red regions corresponds to high contribution and blue to almost no contribution to matching the centroid (colors are smoothed for visual clarity and better reflect filters' size)

better than the previous one. In this case, the reconstruction task seems again not entirely appropriate.

7 Summary and perspectives

The main observations of this study can be listed as follow:

1. The use of deep representation leads to improvements in the clustering performance.
2. There is a high variability of the results depending on the different architecture, pretext losses, and clustering losses used.
3. The CNN based architectures seem the most appropriate for learning relevant features.
4. The use of existing clustering losses does not seem relevant for time series.
5. The classical reconstruction loss, and its variation *multi_rec*, remain the most consistent way to obtain a suitable representation for the clustering task.
6. Despite the previous observation, the reconstruction loss seems not completely fit to extract temporal patterns.
7. Multiple parameters can influence the clustering performance of DNNs' learned representation. Among them, the use of UMAP reduction dimension method results in a significant and consistent performance gain.
8. No significant correlation could be established between datasets characteristics DNNs' parameters or elements (architecture, loss type).

All the image-specific clustering frameworks proposed in the literature turned out to be less effective than a classical autoencoder on time series. These methods seem to highly disturb the DNNs which may lead to degenerated clustering, e.g. only one cluster. It clearly seems not possible to apply them as-is on time series. It would require some further adaptations or new propositions to take into account the particularities of the time dimension. Moreover, even the reconstruction loss does not seem particularly adapted to time series.

However, this put aside, the best deep clustering candidate manages to obtain good results and to rank first among all methods when combined with UMAP with a reduction to K (number of clusters) dimensions. Nevertheless, it is important to note that UMAP, when applied to original data with the K-Means method, ranks before the best deep clustering candidate (without UMAP). As UMAP search for a low dimensional projection of the data that preserve both local and global structure, it preserves indirectly the most discriminating features. Therefore it may suggest that DNNs manage to capture important features or patterns but also noisy features which may lead to poor clustering results. The UMAP helps to extract the relevant features, at least the most discriminating ones. The grad-CAM heatmap plots lead to the same conclusion as they show that DNNs have the capacity to extract temporal patterns but also that it can be partially disturbed by the training loss.

Hence, this analysis leads to the conclusion that deep clustering for time series lacks suitable pretext losses. The new losses should better take into account the specificities of the time dimension (i.e. stretched or shifted patterns) and be less sensitive to small or rare variations.

8 Conclusion

In this paper, we have conducted a large study to compare different time series clustering methods based on deep learning. We have shown that deep clustering methods can be separated into three components: the architecture (i.e. the type, number, and configuration of layers), the pretext loss, and the clustering loss. Based on this taxonomy, we have conducted a cross-comparison to evaluate the influence of each component separately. It results that the best combinations are based on a simple autoencoder architecture that uses reconstruction-based pretext losses. The more advanced frameworks, mostly proposed for image clustering, do not seem to improve the performance while processing time series.

However, even if the advances from the image domain does not translate directly to time series, deep clustering methods still appears as promising. When compared to state-of-the-art methods, the best candidates obtained competitive results. Moreover, grad-CAM and centroid reconstruction can be used to extract and identify learned patterns. However, two main limitations remain. First, the choice of the best combination and the best configuration for a new dataset is still a tedious task. A few guidelines can be retrieved from our observations, but they remain limited. Second, even though deep clustering methods have shown their ability to detect and discriminate temporal patterns, some examples show that is it not consistent over all combinations and datasets. This second limitation is at least partially explained by the use of the reconstruction loss that do not seem adapted to the time domain, even if it performs better than other losses proposed specifically for time series. Finally, it should also be mentioned that the UCR and UEA archives may not be completely fit as-is to evaluate the clustering ability of a method as we defined it. Indeed we showed that some methods that clearly fail to catch time patterns may still lead to average or good results.

To conclude, deep clustering methods gives promising results but research in time domain tailored losses is still required to significantly increase the performance of deep time series clustering.

References

Aghabozorgi S, Shirkhorshidi AS, Wah TY (2015) Time-series clustering–a decade review. Information Systems 53:16–38

Bagnall A, Dau HA, Lines J, Flynn M, Large J, Bostrom A, Southam P, Keogh E (2018) The uea multivariate time series classification archive, 2018. arXiv preprint arXiv:181100075

Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:14090473

Ballard DH (1987) Modular learning in neural networks. In: AAAI, pp 279–284

Becker S (1991) Unsupervised learning procedures for neural networks. International Journal of Neural Systems 2(01n02):17–33

Bengio Y, Simard P, Frasconi P (1994) Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks 5(2):157–166

Bo D, Wang X, Shi C, Zhu M, Lu E, Cui P (2020) Structural deep clustering network. In: Proceedings of The Web Conference 2020, pp 1400–1410

Caron M, Bojanowski P, Joulin A, Douze M (2018) Deep clustering for unsupervised learning of visual features. In: Proceedings of the European Conference on Computer Vision (ECCV), pp 132–149

Chan KP, Fu AWC (1999) Efficient time series matching by wavelets. In: Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337), IEEE, pp 126–133

Chang S, Zhang Y, Han W, Yu M, Guo X, Tan W, Cui X, Witbrock M, Hasegawa-Johnson MA, Huang TS (2017) Dilated recurrent neural networks. In: Advances in Neural Information Processing Systems, pp 77–87

Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:14061078

Dau HA, Bagnall A, Kamgar K, Yeh CCM, Zhu Y, Gharghabi S, Ratanamahatana CA, Keogh E (2019) The ucr time series archive. IEEE/CAA Journal of Automatica Sinica 6(6):1293–1305

Dempster A, Petitjean F, Webb GI (2020) Rocket: Exceptionally fast and accurate time series classification using random convolutional kernels. Data Mining and Knowledge Discovery pp 1–42

Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. Journal of Machine learning research 7(Jan):1–30

Doersch C, Gupta A, Efros AA (2015) Unsupervised visual representation learning by context prediction. In: Proceedings of the IEEE international conference on computer vision, pp 1422–1430

Fawaz HI, Forestier G, Weber J, Idoumghar L, Muller PA (2019) Deep learning for time series classification: a review. Data Mining and Knowledge Discovery 33(4):917–963

Franceschi JY, Dieuleveut A, Jaggi M (2019) Unsupervised scalable representation learning for multivariate time series. In: Advances in Neural Information Processing Systems, pp 4652–4663

Gers FA, Schmidhuber J, Cummins F (2000) Learning to forget: Continual prediction with lstm. Neural Computation 12(10):2451–2471

Ghasedi K, Wang X, Deng C, Huang H (2019) Balanced self-paced learning for generative adversarial clustering network. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 4391–4400

Ghasedi Dizaji K, Herandi A, Deng C, Cai W, Huang H (2017) Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. In: Proceedings of the IEEE international conference on computer vision, pp 5736–5745

Goodfellow I, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, Courville A, Bengio Y (2014) Generative adversarial nets. In: Advances in neural information processing systems, pp 2672–2680

Guan Q, Huang Y, Zhong Z, Zheng Z, Zheng L, Yang Y (2018) Diagnose like a radiologist: Attention guided convolutional neural network for thorax disease classification. arXiv preprint arXiv:180109927

Guo X, Gao L, Liu X, Yin J (2017a) Improved deep embedded clustering with local structure preservation. In: IJCAI, pp 1753–1759

Guo X, Liu X, Zhu E, Yin J (2017b) Deep clustering with convolutional autoencoders. In: International conference on neural information processing, Springer, pp 373–382

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural computation 9(8):1735–1780

Holm S (1979) A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics pp 65–70

Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences 79(8):2554–2558

Ienco D, Pensa RG (2019) Deep triplet-driven semi-supervised embedding clustering. In: International Conference on Discovery Science, Springer, pp 220–234

Jaderberg M, Simonyan K, Zisserman A, Kavukcuoglu K (2015) Spatial transformer networks. arXiv preprint arXiv:150602025

Jain AK (2010) Data clustering: 50 years beyond k-means. Pattern recognition letters 31(8):651–666

Jiang Z, Zheng Y, Tan H, Tang B, Zhou H (2016) Variational deep embedding: A generative approach to clustering. CoRR

Jiao Y, Yang K, Dou S, Luo P, Liu S, Song D (2020) Timeautoml: Autonomous representation learning for multivariate irregularly sampled time series. arXiv preprint arXiv:201001596

Kingma DP, Welling M (2013) Auto-encoding variational bayes. arXiv preprint arXiv:13126114

Kipf TN, Welling M (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:160902907

Kotsiantis SB, Zaharakis I, Pintelas P (2007) Supervised machine learning: A review of classification techniques. Emerging artificial intelligence applications in computer engineering 160(1):3–24

Kramer MA (1991) Nonlinear principal component analysis using autoassociative neural networks. AIChE journal 37(2):233–243

Krizhevsky A, Sutskever I, Hinton GE (2012) Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp 1097–1105

Larsson G, Maire M, Shakhnarovich G (2017) Colorization as a proxy task for visual understanding. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 6874–6883

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

Li X, Chen Z, Poon LK, Zhang NL (2018) Learning latent superstructures in variational autoencoders for deep multidimensional clustering. In: International Conference on Learning Representations

Liao TW (2005) Clustering of time series data—a survey. Pattern recognition 38(11):1857–1874

Lin J, Keogh E, Wei L, Lonardi S (2007) Experiencing sax: a novel symbolic representation of time series. Data Mining and knowledge discovery 15(2):107–144

Lipton ZC, Tripathi S (2017) Precise recovery of latent vectors from generative adversarial networks. arXiv preprint arXiv:170204782

Ma Q, Zheng J, Li S, Cottrell GW (2019) Learning representations for time series clustering. In: Advances in Neural Information Processing Systems, pp 3776–3786

Ma Q, Li S, Zhuang W, Wang J, Zeng D (2020) Self-supervised time series clustering with model-based dynamics. IEEE Transactions on Neural Networks and Learning Systems

Maaten Lvd, Hinton G (2008) Visualizing data using t-sne. Journal of machine learning research 9(Nov):2579–2605

Madiraju NS, Sadat SM, Fisher D, Karimabadi H (2018) Deep temporal clustering: Fully unsupervised learning of time-domain features. arXiv preprint arXiv:180201059

Makhzani A, Frey B (2013) K-sparse autoencoders. arXiv preprint arXiv:13125663

McConville R, Santos-Rodriguez R, Piechocki RJ, Craddock I (2019) N2d:(not too) deep clustering via clustering the local manifold of an autoencoded embedding. arXiv preprint arXiv:190805968

McInnes L, Healy J, Melville J (2018) Umap: Uniform manifold approximation and projection for dimension reduction. arXiv preprint arXiv:180203426

Mukherjee S, Asnani H, Lin E, Kannan S (2019) Clustergan: Latent space clustering in generative adversarial networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol 33, pp 4610–4617

Panuccio A, Bicego M, Murino V (2002) A hidden markov model-based approach to sequential data clustering. In: Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), Springer, pp 734–743

Paparrizos J, Gravano L (2015) k-shape: Efficient and accurate clustering of time series. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp 1855–1870

Pascanu R, Gulcehre C, Cho K, Bengio Y (2013) How to construct deep recurrent neural networks. arXiv preprint arXiv:13126026

Petitjean F, Ketterlin A, Gançarski P (2011) A global averaging method for dynamic time warping, with applications to clustering. Pattern Recognition 44(3):678–693

Rani S, Sikka G (2012) Recent techniques of clustering of time series data: a survey. International Journal of Computer Applications 52(15)

Rifai S, Vincent P, Muller X, Glorot X, Bengio Y (2011) Contractive auto-encoders: Explicit invariance during feature extraction. In: Icml

Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review 65(6):386

Roweis ST, Saul LK (2000) Nonlinear dimensionality reduction by locally linear embedding. science 290(5500):2323–2326

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning representations by back-propagating errors. nature 323(6088):533–536

Sak H, Senior A, Beaufays F (2014) Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In: Fifteenth Annual Conference of the International Speech Communication Association

Sakoe H, Chiba S (1978) Dynamic programming algorithm optimization for spoken word recognition. IEEE transactions on acoustics, speech, and signal processing 26(1):43–49

Saxena A, Prasad M, Gupta A, Bharill N, Patel OP, Tiwari A, Er MJ, Ding W, Lin CT (2017) A review of clustering techniques and developments. Neurocomputing 267:664–681

Schroff F, Kalenichenko D, Philbin J (2015) Facenet: A unified embedding for face recognition and clustering. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 815–823

Selvaraju RR, Cogswell M, Das A, Vedantam R, Parikh D, Batra D (2017) Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE international conference on computer vision, pp 618–626

Shahriari B, Swersky K, Wang Z, Adams RP, De Freitas N (2015) Taking the human out of the loop: A review of bayesian optimization. Proceedings of the IEEE 104(1):148–175

Souza TV, Zanchettin C (2019) Improving deep image clustering with spatial transformer layers. In: International Conference on Artificial Neural Networks, Springer, pp 641–654

Sun D, Wulff J, Sudderth EB, Pfister H, Black MJ (2013) A fully-connected layered model of foreground and background flow. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 2451–2458

Sutskever I, Vinyals O, Le QV (2014) Sequence to sequence learning with neural networks. In: Advances in neural information processing systems, pp 3104–3112

Tenenbaum JB, De Silva V, Langford JC (2000) A global geometric framework for nonlinear dimensionality reduction. science 290(5500):2319–2323

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. arXiv preprint arXiv:170603762

Vincent P, Larochelle H, Bengio Y, Manzagol PA (2008) Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th international conference on Machine learning, pp 1096–1103

Wang C, Pan S, Hu R, Long G, Jiang J, Zhang C (2019) Attributed graph clustering: A deep attentional embedding approach. arXiv preprint arXiv:190606532

Wang Z, Yan W, Oates T (2017) Time series classification from scratch with deep neural networks: A strong baseline. In: 2017 International joint conference on neural networks (IJCNN), IEEE, pp 1578–1585

Weiss G, Goldberg Y, Yahav E (2018) On the practical computational power of finite precision rnns for language recognition. arXiv preprint arXiv:180504908

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics, Springer, pp 196–202

Woo S, Park J, Lee JY, Kweon IS (2018) Cbam: Convolutional block attention module. In: Proceedings of the European conference on computer vision (ECCV), pp 3–19

Xiao Y, Cho K (2016) Efficient character-level document classification by combining convolution and recurrent layers. arXiv preprint arXiv:160200367

Xiao Z, Xu X, Xing H, Chen J (2020) Rtfn: Robust temporal feature network. arXiv preprint arXiv:200807707

Xie J, Girshick R, Farhadi A (2016) Unsupervised deep embedding for clustering analysis. In: International conference on machine learning, pp 478–487

Xu J, Xiao L, López AM (2019) Self-supervised domain adaptation for computer vision tasks. IEEE Access 7:156694–156706

Yang X, Deng C, Zheng F, Yan J, Liu W (2019) Deep spectral clustering using dual autoencoder network. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 4066–4075

Yu F, Koltun V (2015) Multi-scale context aggregation by dilated convolutions. arXiv preprint arXiv:151107122

Zeng N, Zhang H, Song B, Liu W, Li Y, Dobaie AM (2018) Facial expression recognition via learning deep sparse autoencoders. Neurocomputing 273:643–649

Zha H, He X, Ding C, Gu M, Simon HD (2002) Spectral relaxation for k-means clustering. In: Advances in neural information processing systems, pp 1057–1064

Zhang Q, Wu J, Zhang P, Long G, Zhang C (2018) Salient subsequence learning for time series clustering. IEEE transactions on pattern analysis and machine intelligence 41(9):2193–2207

Zhou B, Khosla A, Lapedriza A, Oliva A, Torralba A (2016) Learning deep features for discriminative localization. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 2921–2929