Unsupervised Learning with Matrix Product States

Michele Guadagnini, Alessandro Lambertini

(Dated: May 3, 2021)

In this project we study and implement two different unsupervised machine learning models in the tensor network framework. In particular, exploiting the properties of the Matrix Product State structure we develop a generative model that is able to learn features and correlations from arrays of binary data. Therefore, we are able to utilize the learned probability distribution to generate new samples. We apply this model to the MNIST dataset of handwritten digits. Moreover, we apply the MPS tensor network to the task of clustering, implementing a quantum-inspired version of the k-means algorithm and applying it to the HIGGS dataset, which is a collection of simulated events often used to evaluate the performance of ML-algorithms in high energy physics.

I. INTRODUCTION

The success of Tensor Networks (TN) formalism in tackling quantum many body problems inspired the implementation of a wide range of architectures and algorithm useful to describe different physical situations and phenomena. In particular, Matrix product state (MPS), that are a particular one-dimensional architecture of TN, allows us to properly describe the ground state of one-dimensional gapped local Hamiltonian. In practice, optimization schemes for MPS such as density-matrix renormalization group (DMRG) have been successful even for some quantum systems in higher dimension.

More recently MPS and other TN structures have been exploited to successfully solve some supervised learning (SL) problems as showed in [6] and [7].

Less explored is the possibility to apply matrix product state and DMRG approach to unsupervised learning (UL) tasks. In particular, generative models, which are essentially methods to build probability distributions from unlabeled data in order to be able to generate new sample from that distributions, share similarities with the quantum many body problem modelled by TN and, in our case, by MPS. In fact, the role played by the wavefunction in quantum physics is played by the target probability distribution in the generative modelling task. According to Born's interpretation, in both cases probabilities are given by the squared norm of these functions that live in huge Hilbert spaces. Exploiting the interpretation of probability given in quantum physics is a fundamentally different *ansatz* with respect to the statistical physics one. Moreover, with a TN such as MPS also the expressibility of the models is leveraged, both the relatively low number of parameters with respect to a classical NN and the quantum physical interpretation of these structure help us in the interpretation process.

Once the connection between TN and their ability to efficiently describe elements that live in high dimensional spaces is understood, one can think to play a little with structures and try, for example, to evaluate distances.

Being able to represent data in high dimensional spaces as MPS through different features maps and the ability to evaluate distances between them, makes the MPS structure feasible to tackle another unsupervised learning task related to the classification problem: clustering.

A standard algorithm for clustering is the K-means algorithm, which divides the data into k different classes by minimizing the variance of the data in each class. The most popular K-means clustering algorithm is Lloyd's algorithm. It starts from k random centroids, and then iteratively divides the data into k classes according to their distances with the centroids. At each iteration, the centers within each cluster are recomputed until convergence.

In this project we produce two Python implementations of unsupervised machine learning algorithms. The first one is a generative model built upon the work presented in [1].

The second is the code implementation of the quantum inspired k-means algorithm described in [2].

In section II we provide a brief theoretical introduction based on the work [3].

II. THEORY

A. tensor network: Matrix product state

The basic objects on which our algorithms acts are tensors. Tensors are ways of organizing and representing information. In general, tensors are lists of numbers represented in a N-dimensional space. If N is equal to one, then our data are organized into a familiar array. If N is equal to two, then we are dealing with matrices, and so on and so forth. The number N is called the *rank* of a tensor, and according to it we can write a tensor as an object with N indeces:

$$T_{\alpha_1\alpha_2\ldots\alpha_N} \tag{1}$$

Exploiting the Penrose graphical notation for tensors we can think this object as in Fig 1:

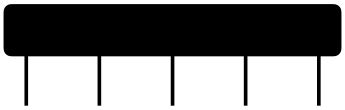

FIG. 1. Graphical representation of a 5-rank tensor.

It is worth to define in a general way an operation that can be carried out between objects of these type, the contraction of one index:

$$A_{ijk}B_{thi} = \sum_i A_{ijk}B_{thi} \tag{2}$$

where on the left the contraction is implicit, according to the Einstein notation and on the right is made explicit.
A tensor network is a set of tensors connected together through the contraction of, some or all, their indeces.
In principle, tensors can describe arbitrary complex systems together with the correlations among the elements that form them. This is possible at the cost of handle huge objects that live in very high dimensional spaces. What is remarkable is that the corner of the Hilbert space described well by algorithms of the DMRG type can be parametrized well, and this parametrization is given by *Matrix Product State.* MPS is a particular kind of tensor network that factorize an arbitrary large tensor in a one dimensional network of 3-rank tensors:

$$T_{\vec{\sigma}} = \sum_{\vec{a}} A^{\sigma_1}_{a_1} A^{\sigma_2}_{a_1 a_2} \dots A^{\sigma_N - 1}_{a_{N-2} a_{N-1}} A^{\sigma_N}_{a_{N-1}} \tag{3}$$

where $\vec{\sigma}$ has N entries and a_i are auxiliary indices.

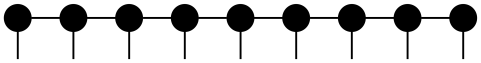

FIG. 2. Graphical representation of a 9-site MPS with open boundary condition.

Before computing explicitly the matrices A_i we introduce an important operation in the context of tensor network which is *gauging.* It is nothing more but exploiting the fact that inserting an identity matrix, so a unitary operator, in the chain of an MPS, and more in general in a connected TN, doesn't affect the system in any way:

$$A_i B_i = A_i \mathbb{I}_{ij} B_j = A_i U^{\dagger}_{ik} U_{kj} B_j = \tilde{A}_k \tilde{B}_k \tag{4}$$

In this way we have been able to modify the entries of our tensors, so we can exploit some useful properties, without affect the structure of our network and the system that it describes. To compute the set of matrices A_i we start by grouping all the indices of our tensor in two indices forming a matrix with dimensions $(d \times d^{N-1})$. Therefore, we apply SVD decomposition to the resulting matrix:

$$\begin{aligned} T_{\sigma_1 \dots \sigma_N} &= T_{\sigma_1,(\sigma_2 \dots \sigma_N)} \\ &= \sum_{a_1}^{D_1} U_{\sigma_1 a_1} S_{a_1 a_1} V^{\dagger}_{a_1,(\sigma_2 \dots \sigma_N)} \\ &= \sum_{a_1}^{D_1} U_{\sigma_1 a_1} T_{a_1 \sigma_2 \dots \sigma_N} \end{aligned} \tag{5}$$

where the matrices S and V have been multiplied and then reshaped back to the original tensor but with the first index as an auxiliary index with dimension $D_1 \leq d$. We now decompose the matrix U into a collection of d row vectors $A^{\sigma_1}_{a_1}$ with entries $A^{\sigma_1}_{a_1} = U_{\sigma_1 a_1}$. By iteratively repeating this process on the resulting tensor T we finally arrive to the result expressed in eq.(3). Now we have successfully factorized our tensor, but we have not gained anything in terms of dimensionality or computing efficiency, in fact our MPS size scales exponentially in N as the tensor from we have started. However, if in the SVD decomposition the last r singular values are zero, then we can throw them away without losing information about the system. Therefore, even if all the singular values are different from zero,we can approximate the systems behaviour locally by keeping only the heavier part of the spectrum, i.e. the D largest singular values. And it can be shown that:

$$\|T - T_{trunc}\|_2^2 \leq 2 \sum_{i=1}^{N} \epsilon_i(D_i) \tag{6}$$

The number D called *bond dimension* of the MPS, controls the precision of the approximation process. Moreover, at each SVD iteration, the relation $U^\dagger U = \mathbb{I}$ holds, and the replacement of U by a set of $A^{\vec{\sigma}}$ entails the following relationship:

$$\sum_{\sigma_i} A^{\sigma_i \dagger} A^{\sigma_i} = \mathbb{I}. \tag{7}$$

Matrices that obey this condition we will refer to as left-normalized, matrix product states that consist only of left-normalized matrices, a part of the right-most site, we say that is in *left-canonical* form.

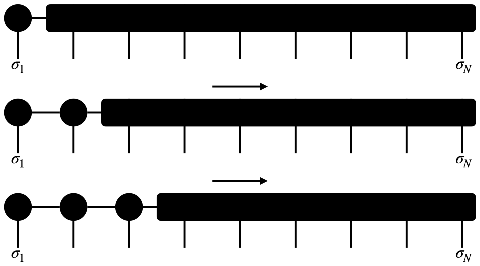

FIG. 3. Graphical representation of the construction of an MPS starting from the first site on the left.

Clearly, there is nothing special in beginning the factorization from the first site on the left, we can proceed in the inverse order beginning from the last site on the right obtaining:

$$\begin{aligned} T_{\sigma_1\ldots\sigma_N} &= T_{(\sigma_1\ldots\sigma_{N-1}),\sigma_N)} \\ &= \sum_{a_1}^{D_1} U_{(\sigma_1\ldots\sigma_{N-1})a_{N-1}} S_{a_{N-1}a_{N-1}} V^{\dagger}_{a_{N-1},(\sigma_N)} \\ &= \sum_{a_1}^{D_1} T_{(\sigma_1\ldots\sigma_{N-2}),(\sigma_{N-1},a_{N-1})} B_{\sigma_L a_{L-1}} \\ &\vdots \\ &= \sum_{\vec{a}} B^{\sigma_1}_{a_1} B^{\sigma_2}_{a_1 a_2} B^{\sigma_N-1}_{a_{N-2}a_{N-1}} B^{\sigma_N}_{a_{N-1}} \end{aligned} \tag{8}$$

Here we have contracted the matrices U and S and reshaped $V^\dagger$ in B iteratively. The B matrices can be shown to have the same matrix dimension bounds as the A matrices and also, from $V^\dagger V = \mathbb{I}$, to obey:

$$\sum_{\sigma_i} B^{\sigma_i \dagger} B^{\sigma_i} = \mathbb{I}. \tag{9}$$

such that we refer to them as right-normalized matrices. An MPS entirely built from such matrices, except the left-most site, we say that is in *right-canonical* form.

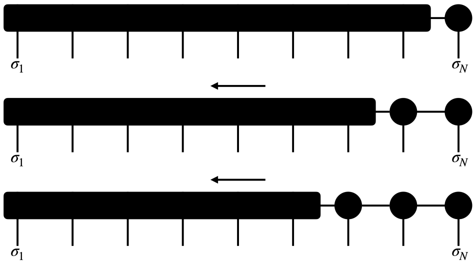

FIG. 4. Graphical representation of the construction of an MPS starting from the last site on the right.

Of course, there is nothing special in the first and last site of the MPS, so we can start the factorization process from the left and reach a certain site $n \ll N$ and then proceed from the right, starting from the site N and go back until the site n is reached, obtaining:

$$T_{\sigma_1 \ldots \sigma_N} = A^{\sigma_1} \ldots A^{\sigma_l} S B^{\sigma_{l+1}} \ldots B^{\sigma_N}, \tag{10}$$

An MPS built in this way we say that is in *mixed-canonical* form.
These are the three most important ways of writing an MPS but, as said before, they are not the only ones because of the gauge degree of freedom that allows to modify our MPS trough any unitary operator. Finally, we want to highlight that for some applications the SVD decomposition is an overkill. In fact, whenever one do not need the singular values then there exists a numerically cheaper decomposition techniques named QR.

III. MPS APPLICATIONS

A. Generative modeling with MPS

The goal of a generative task is to model the joint probability distribution of a dataset and then use it to generate new samples consistent with it. As proposed in [1], we try to model the distribution of data into a quantum state using a MPS with open boundary condition. We use the dataset of handwritten digits from MNIST [11], which is very commonly used in machine learning applications. The distribution to learn can be expressed with the following formula:

$$P(\Phi_i) = \frac{|\langle \Psi | \Phi_i \rangle|^2}{Z} \tag{11}$$

where Ψ is the state represented by the MPS, Φ_i is a data instance and Z is the normalization factor defined as $Z = |\langle \Psi | \Psi \rangle|^2$. Note that, thanks to the mixed canonical form, Z is very easy and cheap to compute, as it involves only the contraction of the orthogonality center tensor with itself.
As usual in machine learning, the training of the model is done through the minimization of a cost function that evaluate the *distance* between the distribution represented by the model and the one resulting from data. In this work, as suggested in [1], we use a *Negative Log-Likelihood* (NLL) function that is defined as:

$$L = -\frac{1}{N} \sum_i \ln P(\Phi_i) = \ln |\langle \Psi | \Psi \rangle|^2 - \frac{\sum_i \ln |\langle \Psi | \Phi_i \rangle|^2}{N} \tag{12}$$

where N is the size of the dataset used for training. Note that, as done in [1], we have transformed the input images from gray-scaled to binary (i.e. black and white). This makes easier the contraction of the input images with the tensors. Also, to save computational time and resources, we rescaled them from 28 × 28 to 20 × 20 resolution. In order to map the images to the MPS we reshaped them into one-dimensional arrays.
To minimize the cost function over the MPS we need, starting from randomly initialized tensors, to sweep back and forth through the MPS chain, updating them iteratively. We denote every complete sweep from left to right and then back to left as a training *epoch*. In this work we adopt a procedure that involves a two site optimization. This method allows us to adjust dynamically the bond dimension between the two tensors; it is used in [1] and is well described also in [6], although for a supervised learning task. The procedure can be summarized in three main steps:

- the first step is to merge two adjacent tensors into one of rank-4, one of them being the orthogonality center of the MPS;
- perform the optimization of the merged tensor based on input data (in machine learning context this is usually referred as *gradient descent*);
- after the update, it is necessary to restore the two rank-3 tensors from the optimized merged one. This can be done by applying a *SVD* decomposition to the matrix resulting from the reshaping of the merged tensor.

To adjust the weights of the merged tensor we need to compute the gradient of the cost function with respect to the merged tensor itself, $M^{k,k+1}$. The gradient can be expressed with the formula:

$$\frac{\partial L}{\partial M^{k,k+1}} = \frac{Z'}{Z} - \frac{2}{N}\sum_i \frac{\Lambda_i}{|\langle\Psi|\Phi_i\rangle|} \tag{13}$$

where Z' is the derivative of the normalization factor and $\Lambda_i = \frac{\partial}{\partial M^{k,k+1}}|\langle\Psi|\Phi_i\rangle|$. Thanks to the mixed canonical form, the computation of the derivative Z' can be extremely simplified to: $Z' = 2M^{k,k+1}$.
Once computed the gradient, a gradient descent technique must be chosen: we explored the usage of two in particular: *mini-batch Stochastic Gradient Descent*, simple method where the gradient is computed on a batch of the training data whose size becomes a model hyper-parameter; *Adam*, which is able to apply the momentum on the gradient descent and adaptively adjust the learning rate [4].
Once the merged tensor has been updated according to the gradient and the hyper-parameters, we need to split it back into two rank-3 tensors by apply *SVD* decomposition.
The *SVD* step has another important role in the procedure: it is responsible of the trade-off between the model complexity, and so its expressive power, and its training computational cost. Indeed, the computation of the singular values gives us the possibility to apply a truncation on them based on a maximum truncation error, that becomes a hyper-parameter of the model. This way we are keeping only the most relevant correlations between the sites of the MPS, while neglecting all the others, that potentially would mine the generalization capability of the model and increase the needed resources in terms of memory and CPU time. Also, with this method it becomes convenient to start from a MPS with small bond dimension (for example 2) and increase it dynamically only when and where it is needed, which results in a consistent speed up in the first training epochs. It is important to note that with this procedure it is also easy to keep the MPS into mixed-canonical form; indeed, we need only to multiply the diagonal matrix of the (eventually truncated) singular values with one of the two unitary matrices, depending on where we want to be the new orthogonality center for the next training step (i.e. if we are sweeping to the right or to the left).
The possibility to directly interpret the MPS as a probability distribution allows to easily realize an algorithm to generate new samples from the learned distribution, exploiting also the canonical form of the MPS. Indeed, starting from one end of the MPS, which has to be also the orthogonality center of the chain, it is possible to iteratively sample one bit according to the conditional probability of all the previously sampled bits.

1. *Code development and implementation*

Instead of implementing from scratch all the tools we needed to set up this model, which would have been very difficult and time consuming, we decided to build our class called *GenerativeMPS* on top of another class of the *tensornetwork* library [8]. In particular, we made our class inheriting from *FiniteMPS* class; in this way we inherited some useful functions for our purposes:

- *random*: it allows to initialize an MPS with random numbers already in canonical form with the desired physical and bond dimensions and with the desired orthogonality center position.
- *position*: it allows to easily shift the center position using QR decomposition.
- *SVD*: it performs SVD decomposition and it can also perform truncation of the singular values based on a max allowed error and a max allowed bond dimension.

The library also support different backends for performing the computations. In this work we only used the default one, which is based on the Python module *NumPy*, but, in principle it should be possible to switch to a different backend (like *TensorFlow* or *PyTorch*) with little modifications of the code. Doing this would open the possibility to use GPU computation, that we would expect to give a great speed up on the training, as stated in [1].

Below in Listing 1 we report the function *bond train* that we used to perform all the two-site optimizations during training. It is easy to see the steps described above: the function *merge bond* is used to create the merged tensor; then the gradient descent is performed, either with *Adam* optimization or with simple mini-batch SGD; finally *rebuild bond* takes care of updating the MPS tensors by mean of SVD decomposition.

```python
    def _bondtrain(self, going_right):
        """Training on the current bond
        - going_right: if the direction of the sweeping is 'right'
        """
        assert self.merged_tensor is None

        # merging two adiacent tensors
        self.merge_bond()

        # moments for Adam optimization procedure (used only if self.use_Adam = True)
        mt = 0.
        vt = 0.

        for tt in range(1, self.descent_steps+1):
            if self.debug_: print("descent_step: ", tt)

            # computing batch gradient
            grad = self._gradient_cumulants()

            # if True, the update is recomputed using Adam procedure
            if self.use_Adam:
                grad, mt, vt = self._AdamOptimizer(grad, mt, vt, tt)

            # updating and normalizing the merged_tensor
            self.merged_tensor -= self.lr * grad
            self.merged_tensor /= self.backend.norm(self.merged_tensor)

        # restoring the two tensors of the MPS from the merged tensor
        self.rebuild_bond(going_right)

        # updating the cumulants cache using the new computed tensors
        self.update_cumulants(going_right)

        return
```

Listing 1. Implementation of the function that perform an optimization step

In our implementation of the sampling algorithm we start the iterations from the first tensor on the left as we have the MPS in right-canonical form. The code we used to generate samples is reported below in Listing 2.

```python
    def generate_sample(self):
        """ It generate a new sample from the learned probability distribution. """

        if self.center_position != 0:
            print("Right-canonicalizing the MPS...")
            self.position(0)

        pixels = np.empty((self.__len__(),), dtype=np.int8)

        prev_psi = np.ones((1,))
        for idx in range(self.__len__()):
            if self.debug_: print(prev_psi.shape, " ^ ", self.tensors[idx][:,1,:].shape)
            psi = np.dot(prev_psi, self.tensors[idx][:,1,:])

            # conditional probability P(K|K-1) = P(K)/P(K-1)
            prob = ( np.linalg.norm(psi)/np.linalg.norm(prev_psi) )**2
            if prob > np.random.rand():
                pixels[idx] = 1
                prev_psi = psi
            else:
                pixels[idx] = 0
                prev_psi = np.dot(prev_psi, self.tensors[idx][:,0,:])

        return pixels
```

Listing 2. Implementation of the function that generates new sample from the learned probability distribution

The procedure starts from computing the conditional probability of the first bit assuming an arbitrary value as previous bit. Then a random number in the interval $[0,1]$ is generated and compared with the conditional probability computed before: if the probability is greater than the generated number, then the current bit is set to 1, otherwise it is set to 0; also the conditional probability to be used in the next step is updated accordingly. Repeating this procedure for every pixel we end up with a sequence of bits that we can reshape into an image.

Before the illustration of the results, we want to highlight an implementation detail that allows us to save a lot of tensor contractions, again empowered by the canonical form of the MPS. Indeed, we have build up a cache for the tensors contractions between the MPS and the training data at left and right of the orthogonality center. In the code this cache is indicated as *cumulants*. Every time we perform a gradient descent on a merged tensor and we split it again in two rank-3 tensors, we also recompute the cache for the old center site using only the cache one site left the current one (or right, it depends on the direction of sweeping) and the new computed tensor. This is clearly more efficient than recomputing all the contractions with data at any training step just because one tensor has been updated, especially when dealing with long chains of tensors.

2. Application and results

As stated above, we have used the MNIST dataset of handwritten digits [11]. In particular, we trained the model with the first 1000 images of the dataset and used another 200 of them as a test set to evaluate the NLL on unseen data and so the generalization capability of the model. We used a tool from *sklearn* library to rescale the images to a lower resolution of 20×20 in order to decrease the MPS size and save some computational resources and time. Something similar has been done also in [1] and [6]. A lower resolution allows us to have a shorter MPS, since we need to map each pixel to a tensor of the chain. We also transformed gray-scale pixels to black and white using a 0.5 threshold on the pixel value. This simplify the contractions between data and the tensors, as we need only to use the pixel value to select the correct position of the tensor physical dimension.

We decide to train the model with two different sets of hyper-parameters in order to see how the model behaves with different values of the bond-dimensions. To train our models we set the hyper-parameters as follows:

	Max bond dim.	**Learning rate**	**Max error**	**Descent steps**	**Batchsize**
Model 1	400	0.0001	0.0001	20	100
Model 2	600	0.0001	0.001	20	100

TABLE I. Hyper-parameters used to train the two models described in this section.

In both the training we have used the Adam optimizer cited above. In Figure 5 we report the plots for the train and test loss for the two models. We can see that the lighter model tends to converge more rapidly, and this is probably due to the smaller maximum error allowed during the SVD decomposition. However, in general we see a huge difference between the values of the training and test loss in both cases. This happen because we took a small size (1000) for the training dataset, and we do so for computational limits of our machines.

In Figure 6 we show some results of the **Model 2** after 13 epochs of training. On the left it is reported the heat-map of the bond dimensions of the MPS. We can notice that on top and bottom of the picture, where our training images are always black, we have lower bond dimensions, while at the center, where the important features of the images show up, we can see that the model reached the maximum allowed dimension. Also, we would expect a decreasing dimension going from the center to the sides. This does not happen probably because the algorithm search correlations between the pixels of consecutive lines of the image.

Finally, in Figure 7 we show the results of the generative process during different moments of the training. It can be noticed that at the beginning the generate image is very noisy, although the model seems to have learned that the borders of the images are always black. After only 4 epochs, the generated images have reached a quite good quality that remains almost unchanged in all the following epochs. This is consistent with the plot of the losses reported in the left part of Figure 5.

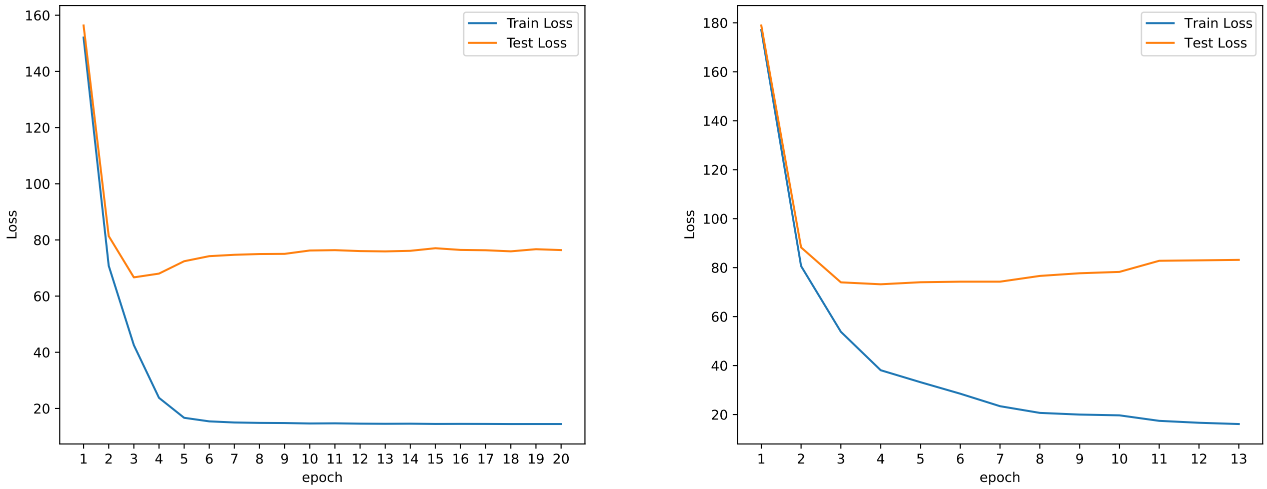

FIG. 5. Left: values of the training and test loss during the training phase with a maximum allowed bond dimension of 400. Right: same plot but for the model with a maximum allowed bond dimension of 600.

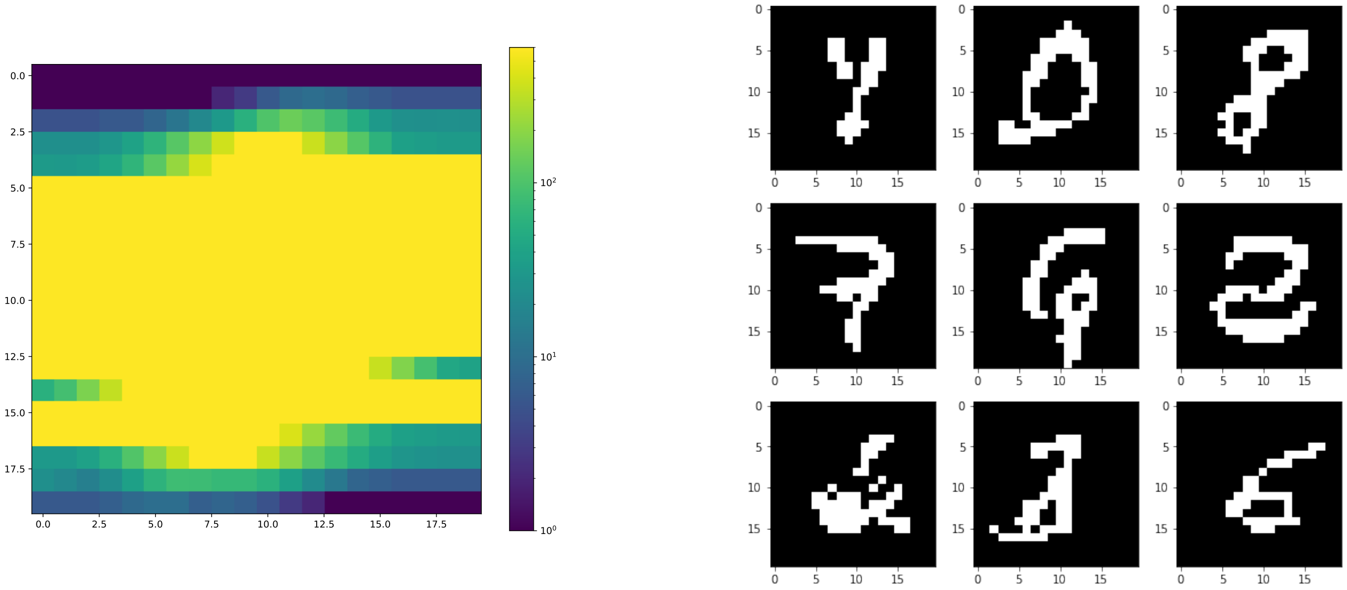

FIG. 6. Left: Heat-map of the final bond dimensions of the trained MPS with maximum allowed bond dimension of 600. Right: a set of samples generated by the model.

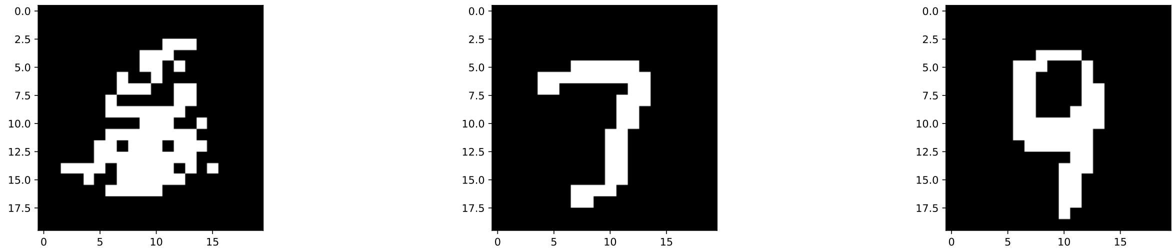

FIG. 7. From the left: evolution of generated images at epochs 2,4,14. Model with maximum allowed bond dimension 400.

B. K-means clustering with MPS

The classical k-means algorithm is a clustering method based on the minimization of the variance of the data inside each cluster. The algorithm works as follow:

- Firstly, we randomly choose k random arrays $\mathcal{C}_i$ as initial centroids of our clusters;
- we then compute the distance of each sample of our dataset from all the centroids and assign each of them to the cluster identified by the closest centroid:

$$d_{ni} = \left\| \vec{x_n} - \vec{\mathcal{C}_i} \right\|^2 \tag{14}$$

- Finally, we update the positions of the centroids finding the point that minimize the variance of the data inside each cluster independently:

$$f(\mathcal{C}_i) = \sum_{n_i=1}^{N_i} d_{n_i i} \qquad \longrightarrow \qquad \mathcal{C}_i = \underset{n_i}{\operatorname{argmin}} f(\mathcal{C}_i) \tag{15}$$

Repeating the second and third steps iteratively the algorithm quickly converge to a local minimum.
The algorithm described above is usually very fast in converging to a local minimum, but sometimes, due to a bad random initialization, it can take very long to converge or it could even converge to a minimum that is far away from the global optimum. In order to improve the initialization process several techniques have been developed. The most effective one and the one that is implemented in our algorithm is *k-means++*. The idea behind this initialization procedure is to take as initial centroids some points from the dataset that lie as far apart from each others as possible. We implemented a slightly different version of this procedure in order to avoid the sampling from a probability distribution. It works as follow:

- we begin by randomly choosing the first centroid from the data.
- The second point is then selected as the farthest point, that belong to a randomly chosen batch of the dataset, from the first centroid.
- All the remaining centroids are then sampled iteratively from different random batches as the points that maximise the distance from the nearest centroid between the already defined ones.

The quantum inspired K-means algorithm, as presented in [2], is very similar to the classic algorithm, as it is still based on distances between data points and centroids. But, in this framework, the data are represented by a list of quantum states parameterized into a separable MPS by mean of a feature map. The centroids instead are chosen to be fixed bond-dimension's MPS initialized at random. The distance between a data instance $|X\rangle$ and a centroid $|\mathcal{C}_i\rangle$ is defined as:

$$\| |X\rangle - |\mathcal{C}_i\rangle \|_2^2 = \langle X|X\rangle - \langle X|\mathcal{C}_i\rangle - \langle \mathcal{C}_i|X\rangle + \langle \mathcal{C}_i|\mathcal{C}_i\rangle \tag{16}$$

In our case, since we work only with real numbers, the formula above reduces to:

$$\| |X\rangle - |\mathcal{C}_i\rangle \|_2^2 \approx 1 - \langle X|\mathcal{C}_i\rangle \tag{17}$$

where we have neglected a factor of 2.
Given this definition of the distance, we can define the cost function for the i-th centroid $\mathcal{C}_i$:

$$f(\mathcal{C}_i) = \sum_{n_i=1}^{N_i} (1 - \langle X_{n_i}|\mathcal{C}_i\rangle) \tag{18}$$

The last step we need to complete the algorithm is to define how to update the centroids from the assigned data instances. This can be achieved by minimizing the cost function of the equation 18. The resulting formula for the recomputation of the centroid, as stated in [2], is:

$$\mathcal{C}_i = \sum_{n_i}^{N_i} A_{a_k} X_{n_i}^{\sigma_k} B_{a_{k+1}} \tag{19}$$

where A_{a_k} and $B_{a_{k+1}}$ represent the left and right contraction of the centroid tensors until the site k, where k is the center of our mixed canonical form.

1. Code development and implementation

For the implementation of the *k-means* algorithm we decide to write two classes, one that will be used by the other:

- The class *Centroid* that contains the definition of a centroid as an MPS. It inherit from the class *FiniteMPS* of the *tensornetwork* library and exploit its tools to:
  - define a random MPS as the object of the class, with a certain number of sites and a fixed bond dimension;
  - initialize and update the cache of tensors contraction that is useful during the training sweeps, with the functions *init_AB, update_AB*;
  - computing the distance between a centroid and a data instance, with the function *distance*;
  - perform the minimizations steps and the computation of the loss through the functions *minimize* and *LossFunction*;
  - perform the sweeping back and forth in order to recompute the tensors of the centroid with the function *sweep*.
- The class *KMeansMPS* instead defines as its object the algorithm with its hyper-parameters, uses the class *Centroid* to create the centroids of the clusters and take care of:
  - import the dataset and pre-process it in order to make it suitable for the clustering process through a feature map. This is done with the function *import_data*;
  - initialize all the centroids of our clusters through the *kmeans++* initialization process with the function *init_centroids*;
  - perform the training loop and assign the labels to unseen data through the functions *train* and *assign_labels*;
  - compute the total loss by summing up the losses of all the centroids.

We have implemented two different feature maps to pre-process our data. Before applying the feature map, we need to rescale the input data as reported below in order to ensure they belong to the interval $[0, 1]$.
The *__trig_feature_map* maps each entry of our data:

$$x_i \quad \longrightarrow \quad \tilde{x}_i = \frac{x_i - \min\limits_n(x_i^n)}{\max\limits_n(x_i^n)} \quad \longrightarrow \quad \begin{pmatrix} \cos\frac{\pi}{2}\tilde{x}_i \\ \sin\frac{\pi}{2}\tilde{x}_i \end{pmatrix} \tag{20}$$

The *__linear_feature_map* istead process our data as:

$$x_i \quad \longrightarrow \quad \tilde{x}_i = \frac{x_i - \min\limits_n(x_i^n)}{\max\limits_n(x_i^n)} \quad \longrightarrow \quad \begin{pmatrix} \tilde{x}_i \\ 1 - \tilde{x}_i \end{pmatrix} \tag{21}$$

In the following we present the code of the two fundamental functions *distance* and *minimize*:

```python
    def distance(self, Xn):
        """
        Calculate distance between the centroid and data instance. The distance is calculated
        as: 1 - the overlap resulting from the full contraction of the MPS with the datum.

        Xn : row containing vectors of feature map for one data instance.
        """

        PP = self.backend.einsum("i,jik -> jk"  , Xn[-1,:], self.tensors[-1])
        for idx in range(self.__len__()-2, -1, -1):
            # auxiliary index contraction
            BB = self.backend.einsum("lij,jk -> lik", self.tensors[idx], PP)

            # physical index contraction
            PP = self.backend.einsum("i,jik -> jk"  , Xn[idx,:], BB)

        overlap = PP.reshape(1)

        return 1. - overlap
```

Listing 3. Implementation of the function that computes the distance between a data instance and its centroid

```
    def _minimize(self, data):
        """
        It performs the optimization of the tensor based on the assigned data.
        Note that the updated tensor is the current orthogonality center of theMPS.
        """
        kk = self.center_position

        #j = b_l   #k = b_l+1    #i = sigma_l   #n=n
        ### Note: data shape [#data row, feature col, mapped feature]
        self.tensors[kk]  = self.backend.einsum("nj,ni,kn -> jik", self.AB[kk],
                                                                   data[:,kk,:],
                                                                   self.AB[kk+1])

        return
```

Listing 4. Implementation of the function that perform the minimization step

2. *Application and results*

a. Random Points In order to do some simple tests on our algorithm, we generated a dataset made of clusters of points normally distributed around their center. We applied our algorithm to the dataset, setting the bond dimension to 1 by now as the dataset complexity is very low. We compare its results with the classic K-means algorithm using the implementation of the *sklearn* library. From the plots in Figure 8, it is possible to see that our algorithm is able to separate the clusters as the classic K-means does.

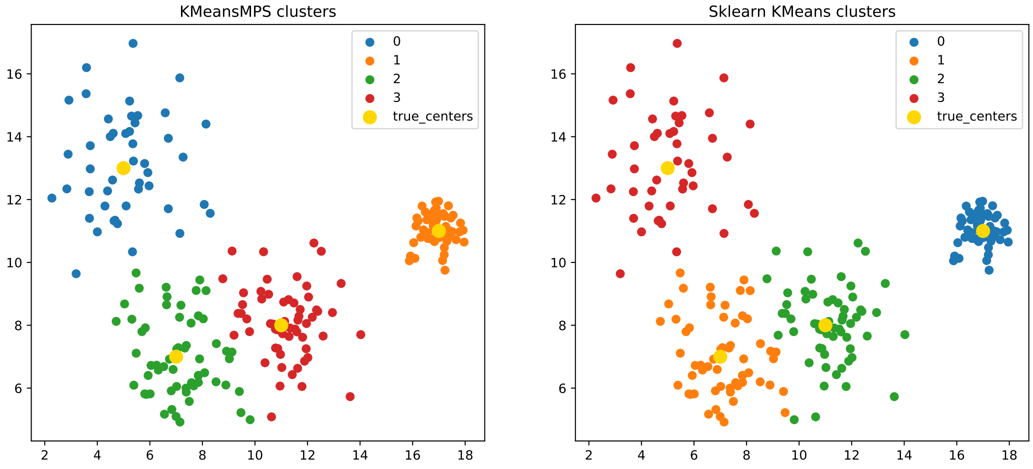

FIG. 8. Comparison of clustering of our algorithm and the K-Means implementation of *sklearn*.

b. Breast-Cancer dataset After this test we are ready to apply it to some different and more complex datasets. The first one we test has been used also by the authors of [2] for the same purpose and it is available at [9]. It is a dataset containing breast cancer information with 9 attributes that should be used to discriminate between a *benign* or *malignant* cancer. We split the 683 instances of the dataset in train set (550 instances) and test set (the remaining 133). We decided to run our algorithm 10 times with 4 different bond dimension of the centroids: $[1, 2, 8, 32]$. From these runs we keep the best obtained accuracies both for the train and the test set for every bond dimension. We report our results in Table II, also comparing them with the results from the *sklearn* K-means implementation.

We can see that the performances of our algorithm are comparable to the one of the classic K-means in terms of accuracy. We tested both the feature maps in our implementation without getting any significant difference. Also, in contrast with what is found in [2], we did not see any improvement in the accuracy when using a higher bond dimension for the centroids.

c. HIGGS dataset One of the main goals of our work was to apply an MPS based unsupervised algorithm to High Energy Physics (HEP) data. We decided to test our algorithm on a dataset of MonteCarlo simulations of Higgs

	K-MeansMPS				**sklearn**
	D=1	D=2	D=8	D=32	**K-Means**
train acc.	0.973	0.973	0.973	0.973	0.958
test acc.	0.947	0.947	0.947	0.947	0.954

TABLE II. Table containing the accuracies obtained by clustering the breast cancer dataset. The results of our algorithm are obtained using the *linear feature map*. We compare them with the ones obtained from *sklearn* K-Means implementation. D denotes the bond dimension of the centroids.

bosons productions. The dataset is available at [10]. It contains 21 low level features and other 7 high level features computed from the low level ones. We have used only the first 5000 data instances, dividing them into training set (80%) and test set (20%). The obtained classification accuracies on the dataset are reported in Table III. As done in [5], we decided to test our algorithm on the high and low level features separately and also on the full dataset. As seen for the *Breast-Cancer dataset* the accuracies are comparable with respect to the *sklearn* implementation. Also, the algorithm seems to be unresponsive to a bond dimension change. Finally, we notice that using the linear feature map slightly improves the accuracies. In any case, the obtained accuracies are not much above the random assignment. This suggests that a distance based method for clustering is not effective for the classification of this kinf of data.

	K-MeansMPS, trig. feature map			**K-MeansMPS**, linear feature map			**sklearn**
	D=1	D=8	D=32	D=1	D=8	D=32	**K-Means**
7 high level features							
train acc.	0.592	0.591	0.593	0.603	0.601	0.601	0.556
test acc.	0.58	0.58	0.581	0.59	0.593	0.593	0.549
21 low level features							
train acc.	0.542	0.554	0.529	0.554	0.525	0.554	0.515
test acc.	0.509	0.536	0.528	0.535	0.535	0.534	0.503
all 28 features							
train acc.	0.529	0.554	0.542	0.537	0.533	0.554	0.558
test acc.	0.532	0.534	0.529	0.528	0.532	0.534	0.55

TABLE III. Table containing the accuracies obtained by clustering the HIGGS dataset. We report the results with both the implemented feature maps. As done before, we compare them with the ones obtained from *sklearn* K-Means implementation. Note that we did 10 trials on the 7 high level features, and 5 trials on the other two datasets in order to save computational time. D denotes the bond dimension of the centroids.

IV. CONCLUSIONS

In this work we have presented two unsupervised learning algorithms based on Matrix Product States. The results of our work are satisfying as a starting point. However, several improvements and researches are possible on the work that have been done. In particular, with computational resources different from the ones at our disposal, and also with a GPU implementation of the two algorithms, one can think to explore more and better the properties of the models here presented.

V. REFERENCES

[1] Han, Zhao-Yu and Wang, Jun and Fan, Heng and Wang, Lei and Zhang, Pan *Unsupervised Generative Modeling Using Matrix Product States*, `http://dx.doi.org/10.1103/PhysRevX.8.031012`

[2] Xiao Shi and Yun Shang and Chu Guo, *Quantum inspired K-means algorithm using matrix product states*, `https://arxiv.org/abs/2006.06164`

[3] Schollwöck, Ulrich, *The density-matrix renormalization group in the age of matrix product states*, `http://dx.doi.org/10.1016/j.aop.2010.09.012`

[4] Diederik P. Kingma and Jimmy Ba, *Adam: A Method for Stochastic Optimization*, `https://arxiv.org/abs/1412.6980`

[5] P. Baldi, P. Sadowski, and D. Whiteson, *Searching for Exotic Particles in High-Energy Physics with Deep Learning*, `https://arxiv.org/pdf/1402.4735.pdf`

[6] E. Miles Stoudenmire and David J. Schwab, *Supervised Learning With Quantum-Inspired Tensor Networks*, `https://arxiv.org/abs/1605.05775`

[7] Marco Trenti, Lorenzo Sestini, Alessio Gianelle, Davide Zuliani, Timo Felser, Donatella Lucchesi and Simone Montangero, *Quantum-inspired Machine Learning on high-energy physics data*, `https://arxiv.org/abs/2004.13747`

[8] TensorNetwork, *TensorNetwork: A Library for Physics and Machine Learning*, `https://arxiv.org/pdf/1905.01330.pdf`

[9] Breast Cancer Wisconsin (Original) Data Set, `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original)`

[10] HIGGS Data Set, `https://archive.ics.uci.edu/ml/datasets/HIGGS`

[11] Yann LeCun, Courant Institute, NYU; Corinna Cortes, Google Labs, New York; Christopher J.C. Burges, Microsoft Research, Redmond, *THE MNIST DATABASE of handwritten digits*, `http://yann.lecun.com/exdb/mnist/`