Università degli Studi di Padova, corso di Laurea Magistrale in Physics of Data, secondo anno

Information Theory: Supervised Learning With Quantum-Inspired Tensor Networks

Nicola Dainese, Stefano Mancone, Francesco Vidaich

(Dated: 29 February 2020)

In this work we implemented a Tensor Network architecture in Python in order to solve supervised learning tasks.

I. INTRODUCTION

One of the most common problems in Machine Learning is the classification of the handwritten digits of the MNIST dataset. The dataset is composed by 60000 grayscale images of 28x28 pixels and by their corresponding labels. This task is usually accomplished using Neural Networks or models derived from them, but here we propose a different kind of method: Tensor Networks.

These networks can be very useful in non-linear kernel learning, where the input $\boldsymbol{x}$ is mapped into a higher dimensional space via a feature map $\boldsymbol{\Phi}(\boldsymbol{x})$ and then classified by the set of weights W^l in the following way: $f^l(\boldsymbol{x}) = W^l \cdot \boldsymbol{\Phi}(\boldsymbol{x})$.

Since the feature vector $\boldsymbol{\Phi}$ and the weight vector W^l can be exponentially large, it is necessary to approximate W^l with a tensor network (l is the index corresponding to the label).

The type of Tensor Network that we are going to use in this work is the Matrix Product State (MPS), which is best suited for describing one-dimensional systems but has been shown to be powerful enough to be applied to higher-dimensional systems as well. Although there are some generalizations of MPS better suited for higher dimensions, here we will test the generalisation capabilities of its basic decomposition, where an order-N tensor is approximated by a contracted chain of N lower-order tensors. More in concrete the MPS tensor network approximates the weight tensor W as follows:

$$W^l_{s_1,\dots,s_N} = \sum_{\alpha_1=1}^{M} \cdots \sum_{\alpha_{N-1}=1}^{M} A^{\alpha_1 l}_{s_1} A^{\alpha_1\alpha_2}_{s_2} \dots A^{\alpha_{N-1}}_{s_N}$$

for a linear chain of matrices or

$$W^l_{s_1,\dots,s_N} = \sum_{\alpha_0=1}^{M} \cdots \sum_{\alpha_{N-1}=1}^{M} A^{\alpha_0\alpha_1 l}_{s_1} A^{\alpha_1\alpha_2}_{s_2} \dots A^{\alpha_{N-1}\alpha_0}_{s_N}$$

if we consider a closed chain (ring). In both cases the label index l has been placed in the first tensor A, but through Singular Value Decomposition (SVD) that index can be transferred to any other tensor (each SVD transformation can transfer the index to the left or to the right neighbour of the tensor currently owning that index). For instance if A_{s_1} has the label index and we want to transfer it to the right we can do it as follows:

$$B^{l\alpha_0\alpha_2}_{s_1,s_2} \equiv \sum_{\alpha_1=1}^{M} A^{\alpha_0\alpha_1 l}_{s_1} A^{\alpha_1\alpha_2}_{s_2}$$

Then we group the indices as $\alpha_0, s_1 \to \boldsymbol{i}$ and $\alpha_2, s_2, l \to \boldsymbol{j}$ and perform SVD on $B_{\boldsymbol{i},\boldsymbol{j}}$ obtaining $B = USV^T$. Now assuming that the diagonal of S is ordered, we keep only the first M entries of S (that becomes a squared $M \times M$ matrix now), then we cut U in order to keep only the first M columns and V^T so to keep only the first M rows. After that we get rid of S multiplying it to the left or to the right (actually we take its square root and multiply it once on the right and once to the left because it's more stable numerically) and split the indices back, e.g. for the left tensor $A^{\alpha_1}_{\boldsymbol{i}} = U \cdot \sqrt{S}$ and $\boldsymbol{i} \to \alpha_0, s_1$ and for the right one $A^{\alpha_1}_{\boldsymbol{j}} = \sqrt{S} \cdot V^T$ and $\boldsymbol{j} \to \alpha_2, l, s_2$.

II. THEORY

A. Input encoding

Each digit contained in the MNIST dataset is a grayscale image represented by N pixels, whose values range from 0.0 to 1.0. Since Tensor Networks are mainly used in quantum mechanics, we want to represent each image as a product state between the states represented by its pixels. This means that, if we can define a feature map $\phi^{s_i}(x_i)$ for a generic pixel, their joint state can be computed as

$$\boldsymbol{\Phi}(\boldsymbol{x}) = \phi^{s_1}(x_1) \otimes \phi^{s_2}(x_2) \otimes \dots \otimes \phi^{s_N}(x_N)$$

where the indices s_i range between 1 and d, which indicates the dimension of the space describing each pixel. A good choice for this local feature map is the following

$$\phi^{s_i}(x_i) = \left[\cos\left(\frac{\pi}{2}x_i\right), \sin\left(\frac{\pi}{2}x_i\right)\right]$$

ϕ^{s_i} represents the normalized wavefunction of a single qubit, where the superposition of a "white" state ($x_i = 1$) and a "black" state ($x_i = 0$) corresponds to a gray pixel.

B. Learning rule

We now discuss how a learning rule can be derived for a classification task with a tensor network. In order to do so, we fix the l index in the generic position j, then we aggregate the tensor A_j with the tensor A_{j+1} obtaining the tensor $B^{\alpha_{j-1} l \alpha_{j+1}}_{jj+1}$. Our aim is to find an update rule that has to be applied to the weights of the tensor B. Firstly we contract all the tensors with the inputs and between themselves, with the exception of the ones that are used to compute to B. By doing so we remain with the left part of the network, the j and $j+1$ inputs tensors and the right part of the network. We call the tensor product of these quantities $\tilde{\Phi}$. See Figure **??** for a graphical explanation. Since the derivative of a tensor network w.r.t. a particular tensor is just the network without that tensor, if we perform the derivative of the cost function C w.r.t. the B tensor

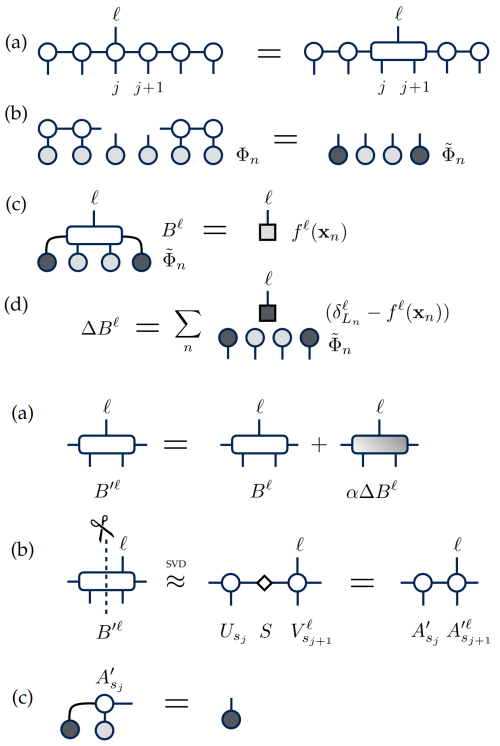

FIG. 1: Diagram representation of the update rule derivation. Source: Stoudenmire, Schwab, 2017.

we obtain

$$\begin{aligned}\Delta B^l = -\frac{\partial C}{\partial B^l} &= \sum_{i=1}^{m}\sum_{l'}(y_i^{l'} - f^{l'}(x_i))\frac{\partial f^{l'}(x_i)}{\partial B^l} \\ &= \sum_{i=1}^{m}(y_i^l - f^l(x_i))\tilde{\Phi}_n\end{aligned} \tag{1}$$

where the "-" already accounts the fact that we want to minimize the cost function.

The newfound gradient has the same dimensions of the tensor B and can be used to update it. ΔB is then multiplied by a learning rate parameter that regulates the update and then added to the tensor B. The last operation that must be performed in the update step is to retrieve the two A tensors separated. In order to do this we perform a SVD transform that yields us

$$B^{\alpha_{j-1} l \alpha_{j+1}}_{j\,j+1} = \sum_{j'j} U^{\alpha_{j-1}}_{j\,\alpha'_j} S^{\alpha'_j}_{\alpha_j} V^{\alpha_j l \alpha_{j+1}}_{j+1}$$

In this passage the authors of the paper used the U tensor as the new A_j tensor and the matrix multiplication between S and V as the new A_{j+1} tensors. We found this solution badly performing in terms of stability, because all of the eigenvalues magnitude is transferred through the network to the next A_{j+1} tensor. We thus decided to further decompose the diagonal matrix S in two diagonal matrices $\sqrt{S}$, whose elements are the square roots of the eigenvalues. Then one of these new matrices is multiplied to the left while the other to the right, obtaining the two new A tensors.

III. CODE DEVELOPMENT

The program is organized in two core classes with their methods and some function that are used for the production of the results. First, we have the Tensor class which implements the main features that a tensor object must have and the operation that are needed to use it properly. We have for example the transposition of the axes and the contraction operation, the latter implemented as a function external to the class. We also have provided the possibility to give names to the axes in order to perform the contraction operation just by specifying the interested axes names. We believe that this improves the code readability and its development. Another interesting operation that we implemented is the sum of two different tensor objects.

The next core class, which is the Network class, provides some options to initialize the MPS tensor network. At least the characteristics of the input data and of the labels must be provided, the starting internal bond dimension of the matrices can be given as input as well and also other parameters that define the architecture can be set from the initialization.

The main methods needed for the network are implemented in this class. We have a forward function that takes an array with the right shape as input and returns the network prevision. This function is used also in many other methods of the class, so is especially relevant. In particular, the output is computed differently depending on the position of the index l: first, every tensor is contracted with its corresponding input, then if the index is on the first tensor, we contract iteratively every tensor starting from the last and storing all of them in the *r_cum_contraction* list; if the index is on the last one, the same process is performed starting from the first tensor (in this case we store every intermediate result in *l_cum_contraction*). If the l index is in any other position, the forward function should not be called and an error is raised.

We then have the train function that, provided a training and validation loaders and other training parameters, performs the optimization steps in the proper order. Here is crucial the call to the sweep step function. This step consists in the calculation of the ΔB gradient, the update step (both handled by the *update_B()* method) and finally the SVD decomposition of the tensor B, in order to return to the two A tensors configuration. This can be tricky when considering that the function is used for sweeps that go though the network in both directions, so the 2D tensor passed as argument to the *tensor_svd()* method has to by built aggregating the axes in different ways. The other functions present in the class are side functions in support of the previously described methods.

One problem that we faced while implementing the training was that during the sweep procedure for the optimization the index l is moved from the first tensor of the chain to the last one, but in the paper there is no indication about how they move it back to the initial position. Therefore we implemented two different methods to solve this problem: the first one was to consider a circular configuration, where the last tensor is

directly bounded to the first one, thus realizing a periodic boundary condition. This allowed us to train the model continuously from the last tensor straight to the first one through SVD. The second method was to consider a linear architecture where, once the training procedure arrives to the last tensor, it then continues in the inverse order all the way to the start of the chain. This method has the drawback that tensors on the sides of the chain are trained twice very close in sequence, while those in the middle are more equally spaced. Another solution would have been to bring back the l index without actually performing the optimization steps, but just by doing the SVD decompositions; unfortunately we estimated that doing so for the whole network chain implied a loss of information of approximately the 40%, because the reconstruction error committed by a single SVD transform is amplified exponentially by repeating it N times. So we decided that this solution was not acceptable. We also started thinking about implementing a 2D lattice in order to reflect the actual disposition of the images pixels but we didn't go further than conceptualization.

In the end, the linear architecture was our final choice, but it was changed to perform a training sweep also when moving the l index back to the first tensor (using a different batch than the one used during the previous sweep). The network can be trained using one of the two provided scripts (*training_diagonals.py* and *training_binary_MNIST.py*), which will also return the evolution of the Accuracy and of the Mean Absolute Error of the network during the training. The trained networks can be tested with the corresponding test scripts.

IV. CRITICAL ISSUES

A. Weights initialization

A major problem that we had to solve during the coding part was that of initializing the weights of the network. Calling W the whole tensor network, A one of the matrices of the MPS approximation, and N the order of the tensor W, we can compute the magnitude of each element of W considering the entries of the matrices A as identical and independently distributed (i.i.d.) random variables, for example in [0,1]. This is done in the following way

$$\begin{aligned} W^l_{s_1,\dots,s_N} &= \sum_{\alpha_0=1}^{M} \cdots \sum_{\alpha_{N-1}=1}^{M} A^{\alpha_0\alpha_1 l}_{s_1} A^{\alpha_1\alpha_2}_{s_2} \dots A^{\alpha_{N-1}\alpha_0}_{s_{N-1}} \\ &\approx \sum_{\alpha_0=1}^{M} \cdots \sum_{\alpha_{N-1}=1}^{M} A^N \\ &\approx A^N \prod_{i=0}^{N-1} \sum_{\alpha_i=1}^{M} 1 \\ &\approx A^N \prod_{i=0}^{N-1} M \\ &\approx A^N M^N \end{aligned} \tag{2}$$

From this we can already see that there is an exponential dependence both on the bond dimension M and the expected value of the random entries of A. We can be even more precise and by looking at the magnitude of the output f (prediction) of the network while considering i.i.d. input components $\phi(x_{s_i})$

$$\begin{aligned} f^l &= \sum_{s_1=1}^{D} \cdots \sum_{s_{N-1}=1}^{D} W^l_{s_1,\dots,s_N} \phi(x_{s_1}) \dots \phi(x_{s_N}) \\ &\approx (AM)^N \sum_{s_1=1}^{D} \cdots \sum_{s_{N-1}=1}^{D} \phi(x_{s_1}) \dots \phi(x_{s_N}) \\ &\approx (AM\phi(x))^N \sum_{s_1=1}^{D} \cdots \sum_{s_{N-1}=1}^{D} 1 \\ &\approx (AM\phi(x)D)^N \end{aligned} \tag{3}$$

Since the output is a linear combination of the inputs (after a proper non-linear transformation ϕ), we still have an exponential dependence on the entries of A; this is of course a receipt for numerical instability unless a careful weight initialization is done.

In general we can try to rescale the weights A in such a way that the outputs are between 0 and 1. This is a reasonable starting point, since the f^l can be interpreted both as the probabilities that the label of a given input is l (cross-entropy loss function) or directly as the value that the label l should have, in case of a mean square error (MSE) loss function. Notice that

1. The expectation value of A is 0.5 if drawn uniformly at random in [0,1];
2. D and M are known, since they have to be specified when the network object is created;
3. If $\phi(x) = cos(\frac{\pi}{2}x)$ (or $\sin(\frac{\pi}{2}x)$) with $x \in [0,1]$, then the average of $\phi(x)$ is approximately 0.64 .

Hence in the case of completely random inputs, we expect

$$f^l(A) \approx (AM\phi(x)D)^N$$

$$f^l(A/C) \approx \left(\frac{A}{C} M\phi(x)D\right)^N$$

Imposing that $f^l(A/C)$ is on average 1 we can obtain an expression for the scaling factor C:

$$C = 0.5 \cdot 0.64 \cdot M \cdot D$$

The problem with that is that with real inputs (like the ones from the MNIST dataset) the value statistics is very different from the random one and this can result in outputs of many orders of magnitude greater or smaller than the ones we would obtain from random inputs. To take also this factor into account, during initialization we introduced the possibility of passing as argument an array of samples that are used to perform a fine calibration of the tensor's weights. Basically we

use the forward method to compute the actual outputs, than we take the maximum, denoted as K, among the predictions (both among the samples and the labels) and rescale the weights A dividing by $K^{\frac{1}{N}}$.

B. Activation and loss functions

In this section we briefly discuss how to implement activation and loss functions for a tensor network and how that changes the update rule. In the original paper the output of the network is directly confronted with the true labels of the inputs and the mean square error is chosen as a loss

$$L_{MSE}(\{x_i, y_i\}_{i=1,\dots,m}) = \frac{1}{2m}\sum_{i=1}^{m}\sum_{l=1}^{D}(f^l(x_i) - y_i^l)^2$$

where $y_i^l = \delta y_i, l$. So the update of the tensor B (for a single sample (x_i, y_i)) is

$$\frac{\partial L_{MSE}}{\partial B^l} = \frac{\partial L_{MSE}}{\partial f^l}\frac{\partial f^l}{\partial B^l}$$

The interesting thing here is that both the activation and the loss function act just on the output of the network regardless of how that output depends on the weights of the network. This means that we can compute analytically the partial derivative of the loss function on the output of the network as

$$\frac{\partial L}{\partial f^l} = \sum_{l'=1}^{D}\frac{\partial L}{\partial a^{l'}}\frac{\partial a^{l'}}{\partial f^l}$$

where a^l is a generic activation function. The activation functions that we considered were

1. Linear activation (identity function), as in the paper;
2. Sigmoid function

   $$\sigma(s) = \frac{1}{1+e^{-s}}$$

   with output in [0,1] that can be interpreted as the confidence that the network has in classifying a digit with a certain label.
3. Softmax function

   $$SMax(\{f^l\}_{l=1,\dots,D}) = \frac{e^{f^l}}{\sum_{l'=1}^{D} e^{f^{l'}}}$$

   whose output can be interpreted as the probability of a sample being of a certain class.

We then coupled with these functions three kinds of loss functions

1. Mean Square Error:

   $$L_{MSE}(x, y) = \frac{1}{D}\sum_{l=1}^{D}(f^l(x) - y^l)^2$$

2. Cross-entropy:

   $$L_{CE} = -\sum_{l} y^l log(f^l(x))$$

   this is 0 only if the correct labels have been assigned probability 1, greater than 0 otherwise. Does not consider explicitly the errors made in classifying the wrong labels.
3. Full cross-entropy:

   $$L_{FCE} = -log(f^l(x)) - \sum_{l'\neq l} log(1 - f^{l'}(x))$$

   where l is the right label (y) for x. This is a custom loss function that we introduced to generalize the more used cross-entropy.

With respect to MSE, the other two loss functions seem to be smoother and in particular the cross-entropy is usually the choice made in classification problems in Machine Learning. The other difference, that from a numerical stability point of view may as well be regarded as a drawback, is that they expect probabilities as inputs. This is completely fine if the output of the network is filtered with a softmax activation; it works also in the case of the sigmoid function (in this case the full cross-entropy should be better since we focus on all predictions and not just the ones for the right labels), but cannot be used together with a linear activation, since the network could learn to maximize all outputs (well above 1) in order to minimize the loss.

C. Gradient clipping and weight decay

Since we found the gradient ΔB to be very unstable (in the sense that its magnitude was oscillating a lot), we introduced two techniques to keep the tensor B and its gradient under control. The gradient clipping is simply the rescaling of all the components of the gradient if the module of the gradient (or the absolute sum of its components) is greater than a certain threshold. More in concrete we choose as threshold the absolute sum of all the entries of B and forced all gradients ΔB to have their absolute sum of elements smaller or equal to the threshold. The weight decay instead is used to constrain the weights of the network to remain "small"; usually this is done by adding a term $\gamma \sum_i (W_i)^2$ to the loss function, where the greater γ is, the more the optimization update will focus on reducing the weights magnitude instead of increasing the accuracy of predictions. For instance if the optimization was switched off and only the weight decay term remained, the update would be

$$W_i' \longleftarrow W_i - 2\gamma W_i$$

The weight decay is always applied before gradient clipping.

This method works independently from the optimization of B, reducing the weights using the same factor. In order to make an update of B that takes in account both the loss value

and the weight decay, we added the L2 regularization to the loss function as follows

$$\text{Loss} = L(\boldsymbol{x}) + \gamma ||W||^2$$

where $L(\boldsymbol{x})$ is the Loss value when considering the input $\boldsymbol{x}$ and $||W||^2$ is the L2 norm of all weights of the network. Since the update of the network is done iteratively through the gradient of the loss function with respect to the tensor B (as explained in section II B), we will compute the derivative of the regularization term with respect to the same tensor in a similar way. First, knowing that $||W||^2 = \langle W|W\rangle$, we can visualize this term using the tensor representation (assume that the l index is on tensor j)

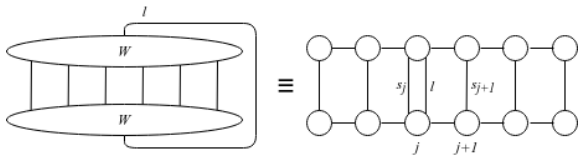

FIG. 2: Tensor representation of $||W||^2$, the L2 norm of the network

Now, considering B obtained contracting the tensors A_j and A_{j+1}, we can compute the derivate of $||W||^2$ by removing B from the network and contracting the rest of if. However, the term $||W||^2$ contains two instances of B, so the derivation has to follow the logic steps explained in Figure 3.

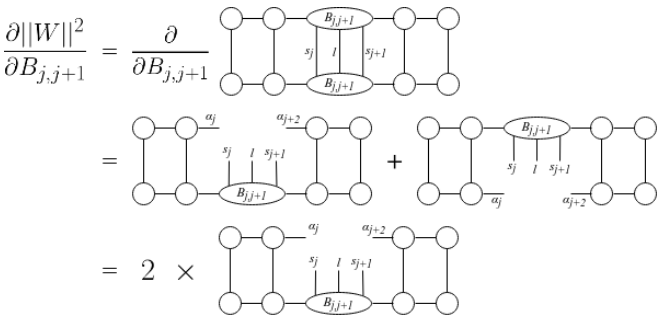

FIG. 3: Steps leading to computing the derivate of $||W||^2$ with respect to $B_{j,j+1}$

It is possible to see that the result has the same shape of B and ΔB. The input samples do not impact this process, so we could not use the cumulative contractions used to update efficiently ΔB. In order to contract the entire network, we started by contracting all tensors A_j with theirselves, resulting in tensors with two "right" indexes and two "left" indexes. To avoid this problem, we changed the axes names of one tensor to "R_2" and "L_2". Then, starting from both extremes, all resulting tensors were contracted to obtain a left and a right term. Now, contracting only one tensor B will return the (halved) derivate of $||W||^2$, which can be further contracted again with the same B to obtain the L2 regularization term added to the loss function.

This second method to implement a form of weight decay is clearly more advanced than the one described at the beginning of this section, but this does not mean that it is better in every aspect. The main advantage of using the L2 regularization is that the weight decay is added to ΔB, so the update process will both try to minimize the error and reduce the magnitude of the weights of the whole network (instead of only the ones that define the current B). Unfortunately this method has some drawbacks: the magnitude of $||W||^2$ scales with the size of the network, which is equal to the size of the input, so different datasets need very different values of γ. Moreover, this process is more computationally extensive than the first one, which can also represent another problem for large networks.

V. RESULTS

The tests of the code were performed mainly on two datasets. As stated in the introduction our goal was to obtain satisfying results with the MNIST dataset, goal that we couldn't't achieve. In order to test the functioning of the various functions and to debug the code we created a second dataset which is just a $n \times n$ matrix with a diagonal of ones. The size of the matrix is a parameter. This is useful in order to understand how the network behaves under different dataset sizes. There is also the possibility to add noise to the generated dataset through a noise parameter.

A. Time performance

First of all we did some tests on the synthetic dataset of the two kinds of diagonals in order to see how our algorithm was scaling with respect to the number of samples $n\_samples$ in the training set, the number of pixels N in an image and the bond dimension M between matrices of the MPS. As can be seen in figures 4 and 5, the training time increases linearly both in the number of samples of the training dataset and on the number of pixels in the input images, whereas increases super-linearly with the bond dimension M (see Figure 6). This checks were useful to see that the code was functioning as expected from a performance perspective and also to have a good idea of how to increase training speed for faster development, i.e. first take a small M (e.g. 10), then if possible reduce images size and finally reduce the number of samples.

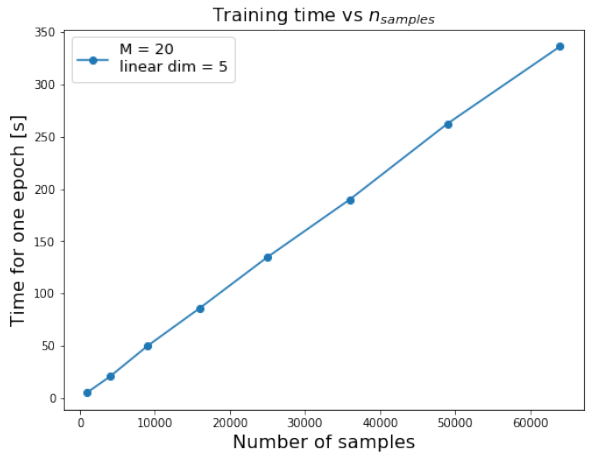

FIG. 4: Linear dependence of training time on the number of samples.

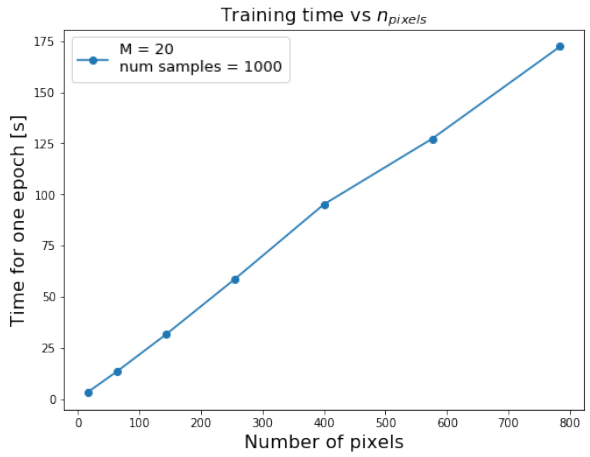

FIG. 5: Linear dependence of training time on the number of pixels of input samples.

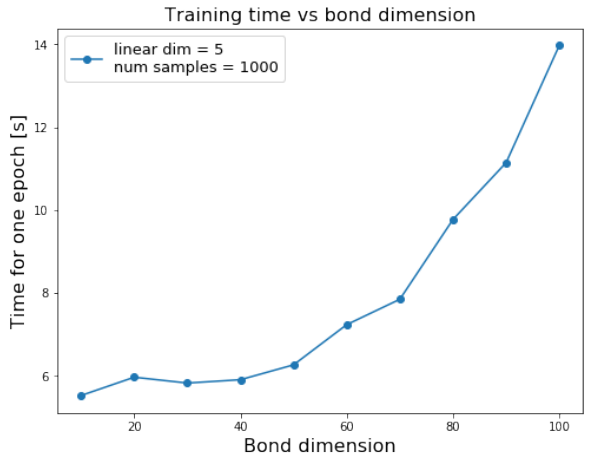

FIG. 6: Super-linear dependence of training time on the bond dimension.

We also profiled the whole training procedure in order to see where most of the time was spent during training. Tests were made in the case of 25 pixels, 5000 samples and bond dimension of 20 and we found that

- 98.9% of the total time was spent training and just 1.1% validating the results.
- 85.9% of the sweep function was spent optimizing the network and most of the remaining time was used for the first forward pass. Notice that since just a single pass is needed for optimizing the N tensors A_{s_i}, so for greater number of pixels this becomes even more efficient.
- 83.7% of each sweep step was spent contracting tensors; this ruled out any possible time leakage in the program.

B. Synthetic diagonal dataset

In this section we present a successful example of supervised learning on the dataset of synthetic diagonals. The images are 8×8 greyscale pictures with noise value of 0.7, which means that we have a diagonal of 0.3 plus a whole picture of random noise in [0,0.7] summed to the original picture. We used the linear architecture with softmax activation function and full cross-entropy loss. The results are evaluated according to two different metrics: the first one is accuracy, i.e. the fraction of correctly classified samples over the total number of samples, the second is the Mean Absolute Error (MEA), that is the mean of the absolute differences between the outputs of the network and the true label probabilities. This second metric is much more informative and reliable than accuracy, because we take into account also the confidence level used in classifying each sample, with a 51-49 % prediction yielding a MAE of 0.5 and a 100-0 % prediction yielding a null error. In both cases, assuming that the true label is 0, they would get 100% of accuracy, but is clear that the second case is a much better classifier. In figures 7 and 8 we can see the results for this dataset, that are quite good and stable, even though the curves are not completely smooth.

C. Reduced MNIST dataset

Given all the numerical instabilities discussed in the previous sections, we tried to find an intermediate problem between the synthetic diagonals dataset and the 10-classes MNIST dataset, so we defined a reduced dataset considering only classes of digits 0 and 1. We then sub-sampled the images through the mean-pooling transformation, where each image is divided in 2×2 pixel squares and then the squares are substituted by their mean. Unfortunately, even though we reduced the complexity of the problem, we weren't able to succeed in the classification task, with the network performance that was noisy and not converging, as can be seen in figures 9 and 10.

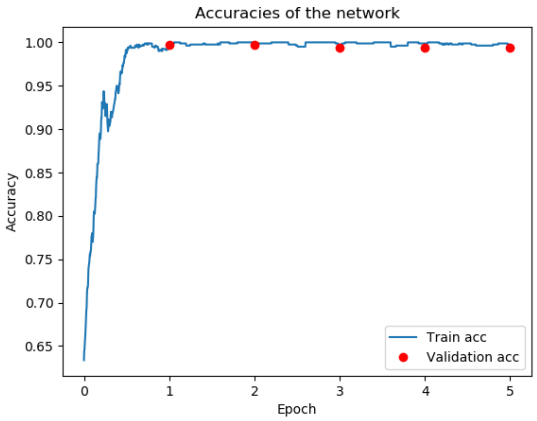

FIG. 7: Accuracy for the diagonal dataset with softmax activation function and full cross-entropy loss function.

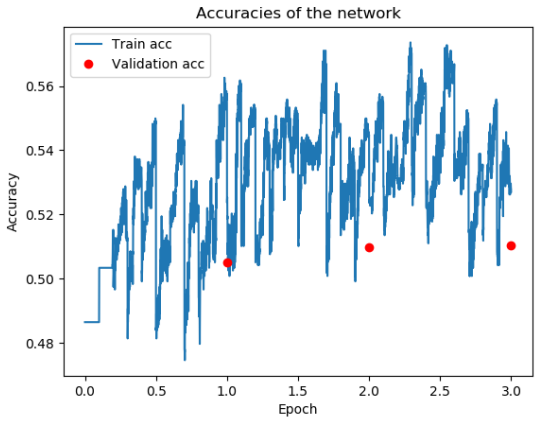

FIG. 9: Accuracy for the 0-1 classes of MNIST dataset with softmax activation function and full cross-entropy loss function.

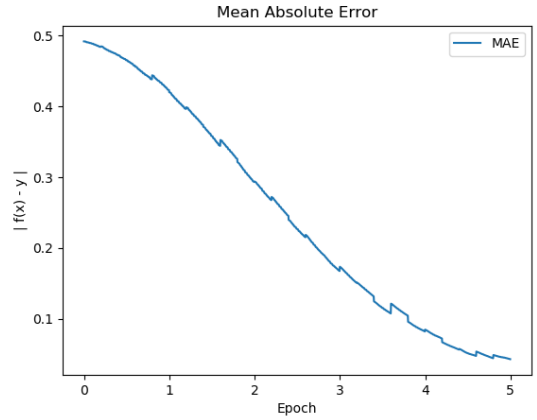

FIG. 8: Mean Absolute Error for the diagonal dataset with softmax activation function and full cross-entropy loss function.

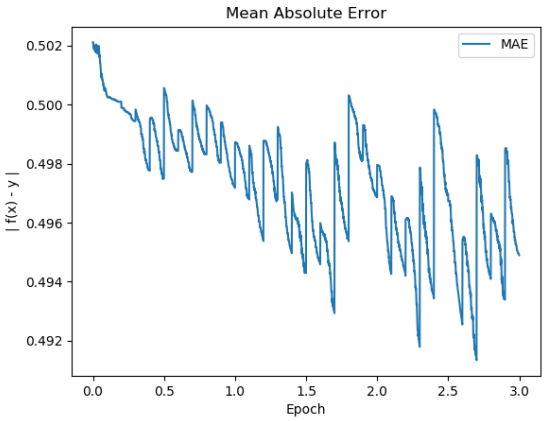

FIG. 10: Mean Absolute Error for the 0-1 classes of MNIST dataset with softmax activation function and full cross-entropy loss function.

VI. CONCLUSIONS

Even though MNIST dataset nowadays is the most basic benchmark for Machine Learning algorithms, we encountered many numerical problems that were not mentioned nor discussed in the original paper and we did not succeeded in solving all of them. This resulted in a very unstable algorithm that was not able in learning to classify handwritten digits. To sum up, the main problems that we encountered were:

1. Weights initialization - solved.
2. SVD decomposition - solved.
3. Gradient computation and update - partially solved.
4. Output f always growing - partially solved.

Also it seems that increasing the size of the batches stabilises the result (this was reported also by the authors of the paper, that actually decided to use a single batch made by all the training set), but this is not sustainable for greater datasets.

We have been partially successful in introducing new activation and loss functions, but they were unstable anyway as long as the gradient was unstable too.

Another problem that we couldn't solve with this methods was that with the increase of the input size the model dimension increases faster than expected, becoming quickly intractable. Future developments could focus on making the network more stable in regimes where it is achieving good performances, ideally having smooth and monotonically decreasing updates as the performance increases. This would allow to make use of the most advanced activation functions and losses without risking underflows or overflows while applying the exponential function (for example in the softmax). Also it is possible that the noisy behaviour of the gradient is due to some elusive bug in our code, possibly in the way the tensors are contracted.