Neural network knowledge distillation in tensor networks

Dereck Piché

April 16, 2023

Abstract

1 Introduction

2 Knowledge Distillation

Knowledge Distillation is a machine learning practice which involves taking a trained model and using its parameters to train another one. The already trained model is referred to as the "teacher", and the model in which his "knowledge" is to be distilled is referred to as the "student". While this is a relatively novel technique, there are already several distinct approaches introduced by researchers. In this project, we used two of these approaches. Our inspiration for the distillation methodology was found in a a 2021 survey which resumed the emerging practice [1].

2.1 Response-Based Knowledge Distillation

The first approach we used was *Response-Based Knowledge Distillation*. The idea of this approach is to stimulate the teacher and the student with some input and to try to minimise a certain loss function with respect to the teacher and student outputs. For classification tasks, it is recommended to use the Kullback-Leibler divergence loss as our loss function.

$$KL(P,Q) = \frac{1}{n}\sum_{i}^{n} Q\frac{log_e(Q)}{log_e(P)} \tag{1}$$

Here, Q is the distribution we are aiming for and P is the one we have.

The reason is quite simple. Since it's a classification task, it is expected that the teacher and student will, given an input, return a probability distribution for each class. As is currently common, we used the softmax function on the logits of the student and the teacher in order to obtain probability distributions corresponding to the classes.

$$softmax(v_i) = \frac{e^{v_i}}{\sum_j e^{v_j}} \tag{2}$$

Now, it would be logical to use a function aimed at measuring the difference between two probability distributions as our loss function. This is exactly what the Kullback-Leibler divergence is for.

2.2 Layer-based approach

The survey on Knowledge Distillation coined the term *Layer-based approach*. Deep learning models work in layers of feature maps. It is hypothesized that these different layers represent different layers of abstraction in the internal representation of their input. If we simply use the Response-Based approach, it could prove difficult for gradient descent otpimization to find these usefull representation without a bit of help. This layer-based approach, as you might have guessed, aims to do precisely that. If our student model has some form of composition, we can train the parts separately. Thus, if an intermediate layer of the teacher l were particularly useful, we could train a certain part of the student on the layer l before proceding with the Response-Based approach.

3 Tensor Networks

Tensor Networks are mathematical objects which were created by physicists in order to help the modelisation of quantum phenomena. In 2017, researchers form TODO add the brilliant idea of using these networks as machine learning models. In order to make this report as self-contained as possible, we decided to provide a short explanation of what they are. Some confusion can be avoided by mentionning that the expression *Tensor Networks* refers, in the scientific community, to both a series of methods and a notational system.

3.1 Tensors

Before explaining these two, we shall disambiguate the meaning of "tensor". In this report (and very often in the context of machine learning), we use the word "tensor" to refer to the mathematical arrays of arbitrary indices. Here, the number of indices is called the "order" of the tensor, meaning that vectors are simply tensors of order 1 and matrices tensors of order 2.

3.2 Contraction

At the heart of tensor networks is the *contraction* operation. Tensor Networks are used to compute a larger network by "contracting" several smaller tensors over chosen indices. A "contraction" is simply an operation where we sum over indices. For example, the contraction of A_{ijk} and B_{ijk} on index j will produce the tensor $C_{ik} = \sum_j A_{ijk}B_{ijk}$. Evidently, the two indices present in a contraction must be of the same size.

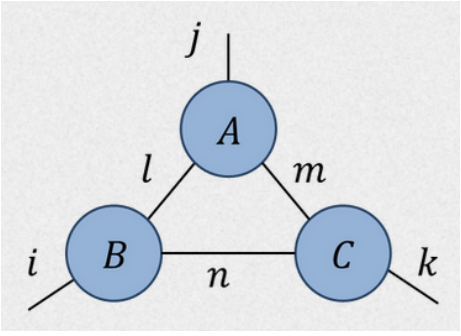
3.3 Tensor Networks as a graphical system of notation

The principal motivation behind the creation of Tensor Networks to compute or approximate large tensors *contracting* several smaller tensors over chosen indices. This is all well and good until our conventional summation notation begins overclocking our primal brains. So, in order to make the manipulation of these network of tensors, mathematicians created a notational system. Tensors are represented as nodes where each vertex connected to the node represents one of the tensor's indices. If a vertex connects two nodes, it means that the indices shall be contracted in order to produce the post-contraction tensor. It should be evident that by the definitions above, no node (tensor) is completely isolated in a tensor network, as it would be completely purposeless. The shape of the post-contraction tensor can be easily visually identified, since it is found in the unconnected vertices. A simple tensor network can be found in figure 1.

3.4 Tensor Network Methods

The term *Tensor Network Methods* is used to refer to, you guessed it, the methods. There are several architectures of Tensor Networks that are frequently used, such as the *Matrix Product State (MPS)*, the *Tenso*. There are many ways of transforming a Tensor into a contraction of smallers tensors. With time, some particular ways became popular for their properties. Some are faster to compute, some are clearer, some are easier to train in a machine learning context. One of these recurring architectures, the Matrix Product State, is at the core of the research done by the author. Not only is there multiple valid ways of setting up the network for contractions, there are also different ways of performing the contraction. In Tensor Networks, the order of contraction affects the computationnal complexity of the whole process! All of these kinds of concepts are referred to by the expression *Tensor Network Methods*.

Figure 1: Simple tensor network illustrating the notation.



4 Creating the Υ_T function

In this report, we used the MPS tensor network to approximate a particular parameterizable function. **Troughout this report, the parameterizable function we are approximating with the MPS will be denoted by Υ_T.** We found it clearer to explain the Υ_T function we are approximating before introducing the MPS tensor network, as we can explain it trough its use. Now, enough talking (or rather, writing?), let's create the Υ_T function!

4.1 Generating $\Phi(X)$ from the Kronecker Product

Consider an input vector $X \in R^d$. Let $\phi : R \mapsto R^{d_\phi}$ be a function which creates a feature map vector $\phi(\lambda)$ from its input λ. We shall form the vector $\Phi(X)$ by taking the Kronecker product of the feature map $\phi(x_i)$ of every element of the input vector X.

$$\Phi(X) = \phi(x_1) \otimes \phi(x_2) \otimes (\dots)\phi(x_d) \tag{3}$$

Here, the symbol $\otimes$ represents the Kronecker Product. The Kronecker Product is an extremely generalisable operation that can be applied to a pair of tensors of arbitrary order. In this project, we use the Kronecker Product exclusively on vectors. Here is a simple example that illustrates what the kronecker product does for simple vectors of integers.

$$\begin{bmatrix}9\\4\end{bmatrix} \otimes \begin{bmatrix}6\\8\end{bmatrix} \otimes \begin{bmatrix}7\\3\end{bmatrix} = \begin{bmatrix}54\\72\\24\\32\end{bmatrix} \otimes \begin{bmatrix}7\\3\end{bmatrix} = \begin{bmatrix}378\\162\\504\\216\\168\\72\\224\\96\end{bmatrix}$$

The element i of the first operand vector becomes itself multiplied by the second operand vector. Here's a slightly more abstract example:

$$\begin{bmatrix}v_1\\v_2\end{bmatrix} \otimes \begin{bmatrix}u_1\\u_2\end{bmatrix} = \begin{bmatrix}u_1v_1\\u_2v_1\\u_1v_2\\u_2v_2\end{bmatrix}$$

Thus, if we perform a Kronecker Product on n vectors, it will produce a vector such that every of it's element is a multiplication of n elements, each one from a different vector. We can thus interpret $\Phi(X)$ as being a feature space vector which *captures the multiplicative interaction of the elements of the $\phi(x_i)$ feature maps.* Here's a visual and intuitive way to think about the Kronecker Product applied to a set of vectors. Suppose we have the set of vectors and that we draw a square an element of each one

$$\begin{bmatrix} \boxed{9} \\ 4 \end{bmatrix}, \begin{bmatrix} 6 \\ \boxed{8} \\ 5 \end{bmatrix}, \begin{bmatrix} \boxed{7} \\ 3 \end{bmatrix}$$

Then the product of these elements, $9 \times 8 \times 7 = 504$, will find iteself in the vector $v = \begin{bmatrix} 9 \\ 4 \end{bmatrix} \otimes \begin{bmatrix} 6 \\ 8 \\ 5 \end{bmatrix} \otimes \begin{bmatrix} 7 \\ 3 \end{bmatrix}$. This is true for every combination of the numbers chosen from each vector. This is was what was meant when we said that the resulting vector captures the *multiplicative interaction of the elements*.

4.2 The multilinear feature map

Now, which feature map ϕ shall we pick in order to construct Φ? Well, normally, this would be up to you, but in this project we deal with the particular feature map

$$\phi_{ml}(x) = \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{4}$$

We call this particular feature map the *multilinear feature map*. Why do we call it that? Because, amazingly, when we use this feature map, the elements of $\Phi(X)$ form a basis space of the multilinear functions on the elements of the vector X. Understanding this statement without a bit of context and help is a tall order. Let's deconstruct it. First, what is a multilinear function? A multilinear function is a multivariate function that is linear for every variable if all the other variables are considered as constants. For example, $f(x_1, x_2) = x_1 x_2$ is a multilinear function because $x_1 c_2$ and $c_1 x_2$ are both linear functions. However, $f(x_1, x_2) = x_1 x_2^2$ is not a multilinear function, because the function $f(x_1, x_2) = c_1 x_2^2$ is not linear.

Let us show you a simple example to illustrate this.

$$\Phi(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \phi_{ml}(x_1) \otimes \phi_{ml}(x_2) \otimes \phi_{ml}(x_3) =$$

$$\begin{bmatrix} 1 \\ x_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ x_2 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ x_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ x_3 \\ x_2 \\ x_2 x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ x_3 \\ x_2 \\ x_2 x_3 \\ x_1 \\ x_1 x_3 \\ x_1 x_2 \\ x_1 x_2 x_3 \end{bmatrix}$$

Now, it should be clear that if we take linear combinations of the elements of the vector $\Phi(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix})$, we can express any multilinear function of the variables x_1, x_2 and x_3.

4.3 Linear Combinations

When we introduced Υ_T, we said it was a parameterizable function. However, as of yet, everything described about the function as been deterministic and fully specified. Thus, it is time to introduce the parameters. The parameters will be the coefficients of a linear map applied to the vector $\Phi(X)$. You might be asking yourself why go trough all this? It has to do with expressivity. If we take the linear combinations of X directly, we will be limited to linear functions. However, in practice, many functions are fundamentally non-linear. They can't even be approximated well by linear functions. Thus, we project the initial vector X in a new non-linear space before applying the linear map. This is why we chose the ϕ_{ml}. Multilinear functions are extremely expressive. Their limitations (as well as a way to bypass them) shall be discussed in later in this report.
We have all the requirements to finally define the Υ_T function.

$$\Upsilon_T(X) = \sum_{i_1,i_2,\ldots,i_d} T_{i_1,i_2,\ldots,i_d} \left[\phi_{ml}(x_1)\right]_{i_1} \otimes \left[\phi_{ml}(x_2)\right]_{i_2} \otimes \cdots \otimes \left[\phi_{ml}(x_d)\right]_{i_d}$$

$$\sum_{i_1,i_2,\ldots,i_d} T_{i_1,i_2,\ldots,i_d} \left(\begin{bmatrix} 1 \\ x_1 \end{bmatrix}_{i_1} \cdot \begin{bmatrix} 1 \\ x_2 \end{bmatrix}_{i_2} \cdot (\ldots) \cdot \begin{bmatrix} 1 \\ x_d \end{bmatrix}_{i_e} \right)$$

Now, since $\Upsilon_T(X) : R^d \to R^h$, and we will not constraint ourselves to $p = 1$, we will need to add an index h to T such that it becomes $T^h_{i_1,i_2,\ldots,i_d}$. At long last, we have all the requirements to finally define the Υ_t function.

$$\Upsilon_T(X) = \sum_{i_1,i_2,\ldots,i_d} T^h_{i_1,i_2,\ldots,i_d} \left(\begin{bmatrix} 1 \\ x_1 \end{bmatrix}_{i_1} \cdot \begin{bmatrix} 1 \\ x_2 \end{bmatrix}_{i_2} \cdot (\ldots) \cdot \begin{bmatrix} 1 \\ x_d \end{bmatrix}_{i_d} \right)$$

Now, have function that is theoretically trainable on data. We simply have to modify the tensor T, which represents the parameters of the model.

5 The *Matrix Product State* (MPS) Tensor Network

The problem is that as of now, the size of the tensor T will grow exponentially with the number of dimensions of the input vector X. Indeed, since our feature map ϕ_{ml} produces feature vectors of 2 dimensions, the parameter tensor T would have $h2^p$ parameters. This is where the Tensor Network Methods come into play. We are going to approximate the tensor T by using the *Matrix Product State* Tensor Network in a creative way that was proposed in the paper [3].

5.1 Building the model using a single MPS

The MPS tensor network was created to approximate big tensors. It performs this by contracting multiple smaller tensors together.

$$T_{i_1,i_2,\ldots,i_d} = \sum_{b_1,b_2,(\ldots),b_d} t^{i_1}_{b_1} t^{i_2}_{b_1,b_2} t^{i_3}_{b_2 b_3} \ldots t^{i_d}_{b_{d-1}} = \text{MPS approximation of } T \quad (5)$$

In our case, the tensor we want to approximate has an extra index h. This index can be added to any tensor of the MPS. For example, we can put it in the third tensor of the chain :

$$\sum_{b_1,b_2,(\ldots),b_d} t^{i_1}_{b_1} t^{i_2}_{b_1,b_2} t^{i_3,h}_{b_2 b_3} \ldots t^{i_d}_{b_{d-1}}$$

As you can see, the original tensor T is approximated by contracting the smaller tensors over their b indices. The dimension of these indices is an hyperparameter of great importance. It is called the *bond dimension* of the MPS. The higher the bond dimension, the closer we are to T. To put it all together, we can approximate $\Upsilon_T(X)$ as such :

$$\Upsilon_T(X) \approx$$

$$\sum_{i_1,i_2,\ldots,i_d} \left(\sum_{b_1,b_2,(\ldots),b_d} t^{i_1}_{b_1} t^{i_2}_{b_1,b_2} t^{i_3,h}_{b_2 b_3} \ldots t^{i_d}_{b_{d-1}} \right) \left(\begin{bmatrix} 1 \\ x_1 \end{bmatrix}_{i_1} \cdot \begin{bmatrix} 1 \\ x_2 \end{bmatrix}_{i_2} \cdot (\ldots) \cdot \begin{bmatrix} 1 \\ x_d \end{bmatrix}_{i_d} \right)$$

In a context of machine learning, it is also important to mention that the bigger the bond dimension, the bigger the expressivity of our model. TODO Add graphic

5.2 Composition of MPS

As mentionned before, taking a linear map of tensor product of the local feature map $x \mapsto \begin{bmatrix} 1 \\ x \end{bmatrix}$ amounts to producing a multilinear function of X.

Since our MPS model can only approximate this function, it means that its expressivity will be somewhat limited. For example, it could never express the function $f(X) = x_1 x_2^2$, since it is not multilinear. It is however entirely possible that being able to express functions of this type are of great importance when it comes to modeling real world phenomena. There is a simple way to fix this problem which ties nicely with our of goals: distillating knowledge into a student MPS network. Once again, to make things clearer, we will talk about achieving this with the Υ_T function first, and talk about the MPS after. Ihe following subsection, we will prove that this composed function can express any multivariate polynomial function of degree z.

5.2.1 Proof of expressivity

Definition 5.1 (Multilinear polynomial)**.** A multilinear polynomial is a function $f : R^n \mapsto R$ of the form

$$f(X) = T \cdot \left(\begin{bmatrix} 1 \\ x_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ x_2 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} 1 \\ x_d \end{bmatrix} \right) \tag{6}$$

Where $T \in R^{p \times R^n}$. In other words, a multilinear polynomial is a polynomial that is linear if $\forall x_i$ then the polynomial is linear if we fix every variable except x_i.

Definition 5.2 (v_i-variate)**.** Function of the variables of the vector v. (Each element of v is considered as a variable).

Theorem 5.1. For every polynomial function Γ of degree z, there exists tensors T_1 and T_2 such that

$$\Gamma(X) = \left(\Upsilon_{T_2} \circ \Upsilon_{T_1}\right)(X), \forall X \in R^d$$

Proof. Let the tensor T_1 be defined such that

$$\Upsilon_{T_1}(X) = \begin{bmatrix} x_1 \\ (\dots) \\ x_1 \\ x_2 \\ (\dots) \\ x_2 \\ (\dots) \\ x_n \\ (\dots) \\ x_n \end{bmatrix} \tag{7}$$

where each of the variables are repeated z times in the vector.

Now, now can perform the vector equivalent of a change of variable by rewriting the vector $\Upsilon_{T_1}(X)$ as a new vector λ :

$$\begin{bmatrix} x_1 \\ (\dots) \\ x_1 \\ x_2 \\ (\dots) \\ x_2 \\ (\dots) \\ x_n \\ (\dots) \\ x_n \end{bmatrix} = \begin{bmatrix} \varepsilon_{1,1} \\ (\dots) \\ \varepsilon_{1,z} \\ \varepsilon_{2,1} \\ (\dots) \\ \varepsilon_{2,z} \\ (\dots) \\ \varepsilon_{d,z} \end{bmatrix} = \lambda \tag{8}$$

Let Z be the space of x_i-variate polynomial functions of degree $\leq z$. By definition, every monomial of $\zeta \in Z$ is of the form

$$x_1^{k_1} x_2^{k_2} (\dots) x_d^{k_d} \tag{9}$$

where the condition $\sum k_i \leq z$ is met.
However, for every set $\{k_1, k_2, (\dots), k_d\}$ meeting this condition, we can rewrite the monomial as

$$\left(\prod_{i_1=1}^{k_1} \varepsilon_{1,i_1} \right) \left(\prod_{i_2=1}^{k_2} \varepsilon_{2,i_2} \right) (\dots) \left(\prod_{i_d=1}^{k_d} \varepsilon_{d,i_d} \right) \tag{10}$$

by using the elements of the vector λ from equation (8).

However, we can clearly see that this term is a multilinear monomial of the variables in λ. This implies that $f(\lambda *)$ returns a basis-vector of x_i-variate polynomial function of degree $\leq z$.

can express any x_i-variate polynomial function of degree $\leq z$ under fixed Θ.

This process can be visualised as

$$X \in R^d \xrightarrow{f} Y_1 \in R^{2^d} \xrightarrow{g} Y_2 \in R^{zd} \xrightarrow{f} Y_3 \in R^{2^{zd}} \xrightarrow{g'} Y_4 \in R \tag{11}$$

□

A pretty obvious solution for this problem is to add another layer by feeding to ouputs of our MPS model into a second one.

Instead of looking at the expressivity of this composition directly, we will analyse the expressivity of the functions that the MPS models are trying to approximate. This will give us a good idea of how expressive this composition can be. Also, from now on, this composition will be refered to as *2-MPS*.

The 2-MPS model approximates a function of the form.

$$g' \circ f \circ g \circ f(X)$$

TODO mention the 2-MPS name. TODO mention that we won't need to let the MPS reach full expressivity. It can decide what to sacrifice to get squared terms.

5.3 Patching of MPS (experimental)

6 Methodology

The experiments done for the projet were programmed using Python and the wonderful deep learning library Pytorch, created by Meta. Unfortunately, Pytorch does not provide any tools to train and build Tensor Networks. Luckily for us, Jacob Miller, created a library [2] built on top of Pytorch that provides the tools to build and train MPS Tensor Networks.

TODO: mention custom feature map in forked code

Learning rate
We used the very standard learning rate of $1e-4$. This is the proposed learning rate in the code of [2].

Model size

Approach to the results As a matter of scientific integrity, we have chosen to show the results even if they are heavily disappointing. Not doing so can result in certain statistical biases which can be avoided.

Learning rate for neural network: 0.01 Learning rate for MPS: 1e-3 = 0.001 Nb of normal epochs: 25 Nb of gaussian epochs: 5

7 Results

Bond Dimensions	MPS	FC to MPS	CNN to MPS
10	$0.897 \pm 1.34 \times 10^{-4}$	$0.541 \pm 9.50 \times 10^{-5}$	$0.906 \pm 8.49 \times 10^{-5}$
20	$0.911 \pm 12.90 \times 10^{-4}$	$0.550 \pm 9.32 \times 10^{-5}$	$0.924 \pm 6.77 \times 10^{-5}$
40	$0.914 \pm 8.15 \times 10^{-5}$	$0.556 \pm 5.99 \times 10^{-5}$	$0.941 \pm 4.38 \times 10^{-5}$
80	$0.916 \pm 3.13 \times 10^{-4}$		$0.944 \pm 7.35 \times 10^{-5}$

Bond Dimensions	2-MPS	NN to 2-MPS
10	$0.897 \pm 1.34 \times 10^{-4}$	$0.897 \pm 1.34 \times 10^{-4}$
20	$0.897 \pm 1.34 \times 10^{-4}$	$0.897 \pm 1.34 \times 10^{-4}$
40	$0.897 \pm 1.34 \times 10^{-4}$	$0.897 \pm 1.34 \times 10^{-4}$

8 Analysis

9 Conclusion

9.1 Further Exploration

TODO talk about capturing locality in the mappings TODO talk about capturing locality in general with tensors

References

[1] Jianping Gou et al. "Knowledge Distillation: A Survey". In: *International Journal of Computer Vision* 129.6 (Mar. 2021), pp. 1789–1819. DOI: 10.1007/s11263-021-01453-z. URL: https://doi.org/10.1007%2Fs11263-021-01453-z.

[2] Jacob Miller. *TorchMPS*. https://github.com/jemisjoky/torchmps. 2019.

[3] E. Miles Stoudenmire and David J. Schwab. *Supervised Learning with Quantum-Inspired Tensor Networks*. 2017. arXiv: 1605.05775 [stat.ML].