Exploration with Multi-Scale Tensor Networks

Tayssir Doghri [1] **Adel Nabli** [2] **Bhairav Mehta** [2]

Abstract

Multi-scale tensor networks offer an interesting approach to unsupervised feature learning. Originally inspired from their physics applications, multi-scale tensor networks and *coarse graining* were shown to be an effective way to do feature learning on simple datasets. This report focuses on examining the method in detail, and extends some of the original hypotheses to multi-task learning. Overall, we find that in memory-constrained settings, the method's expensive procedure does not provide a significant enough benefit in our classification and multi-task learning experiments to be justified over traditional deep networks. However, as with all new methods, further research and iterations are required for progress, which we support by providing an efficient Python and Pytorch implementation of the original method. Along with a thorough review and ablation studies of the method's design decisions, we implement some augmentations inspired by state-of-the-art deep learning training schemes.

1. Introduction

Machine learning aims to give computers some knowledge from the data we provide. This acquired knowledge is then used to make new decisions without any human intervention. Thanks to advanced computational and storage capabilities, machine learning has become a promising technology applied to various domains. However, two main challenges need to be solved: how to represent the data and how to learn from it. Being able to represent and manipulate high dimensional data, tensor decomposition techniques are proving to be a powerful tool for machine learning.

Recently, Stoudenmire (2018) used tensor networks along with coarse graining, a method from physics, to compress data originally represented in a very high dimensional space. Then, using this reduced description of the data, the author performs learning tasks such as classification. While the results are shown on simple datasets with only reasonable performance, the proposed algorithm shows promising results and opens the door to alternate training methods and network architectures. In particular, with tensor networks, the cost of training each of the models discussed in the paper is linear in both training set size and input dimension.

Inspired by kernel learning, the author maps inputs $\mathbf{x} \in \mathbb{R}^N$ into a space of dimension $\mathbb{R}^{2^N}$ by using a feature map of the form:

$$\Phi^{s_1 s_2 \dots s_N}(\mathbf{x}) = \phi^{s_1}(x_1)\phi^{s_2}(x_2)...\phi^{s_N}(x_N) \qquad (1)$$

where each $\phi^{s_i}(x_i)$ with $i = 1, .., N$ defines a local feature map. Then, by finding an isometry layer $\mathcal{U}$ in an unsupervised manner, the feature space is compressed. The author explores a model of the form shown in (2):

$$f(\mathbf{x}) = \mathbf{W} \cdot \Phi(\mathbf{x}) \qquad (2)$$

where Φ is the feature map defined in (1) and $\mathbf{W}$ represent the weights to be optimized.

1.1. Contributions

Using the method described in Stoudenmire (2018) as a starting point, we provide (1) an efficient, multiprocess implementation of the original algorithm[1] (originally written in C++). In addition, we (2) report results on a thorough set of ablations and enhancements regarding design decisions in the original paper, studying the effects of approximation error thresholds, dataset sizes, and feature maps on resulting performance. Lastly, we (3) perform a new experiments, testing the method's viability for learning transferable, shared representations.

2. Multi-Scale Tensor Networks

We have $\{\mathbf{x_1}, ..., \mathbf{x_{N_T}}\}$ a training dataset of N_T vectors $\in \mathbb{R}^N$. The task is, given any new input vector $\mathbf{x}$, to output

[1]Institut National de la Recherche Scientifique [2]Universite de Montreal. Correspondence to: Tayssir Doghri <tayssir.doghri@gmail.com>.

Project report for IFT6760A. Authors listed in random order.

[1]Found at https://github.com/bhairavmehta95/learning-relevant-tensor-networks

a scalar $f(\mathbf{x})$ using the inner product defined in (2). In order to do that, we first use the non-linear feature map Φ defined in (1) to represent the input data in a richer space. As we fix the feature map that we will use, what we want to learn is a convenient $\mathbf{W}$ for the task. Due to the representer theorem, we can limit our search to $\mathbf{W}$ being a linear combinations of the $\Phi(\mathbf{x_j})^\top$ with $j \in \{1, .., N_T\}$. Then, what remains to be learned are the weights of this linear combination. As the $\Phi(\mathbf{x_j})$ are not necessarily linearly independent, we can reduce the number of weights to learn by only considering a linear combination of the transposed vectors of a basis of the space spanned by them. For convenience, we will use the vectors from an orthonormal basis of this space. One way of finding those vectors is by performing an SVD on the matrix $\mathbf{\Phi} \in \mathbb{R}^{2^N \times N_T}$ (the concatenation of the feature maps of the training set) and to take the columns of $\mathbf{U}$ (the matrix diagonalizing $\rho = \mathbf{\Phi}\mathbf{\Phi}^\top \in \mathbb{R}^{2^N \times 2^N}$) as the basis vectors. But performing an SVD on such a big matrix is usually intractable in practice, so we have to make some approximations in order to compute $\mathbf{U}$.

In this section, first we will present the approximations we make for this computation, and then we will go back to our original task which was finding the weights of the linear combination that make $\mathbf{W}$.

2.1. Unsupervised Coarse Graining

The idea for approximating $\mathbf{U}$ is to take advantage of the way we built Φ in (1) and use an analogy with physics. Indeed, we can see each training example $\mathbf{x_j}$ as a collection of N particles x_i^j (some recent works demonstrate the validity of this analogy when the $\mathbf{x_j}$ are images, see Levine et al. (2018)). Then, we can further the analogy by saying that each local feature map $\phi^{s_i}(x_i^j) \in \mathbb{R}^2$ defined in (1) represent the normalized wave function of the particle x_i^j. Thus, $\Phi(\mathbf{x_j})$ has the structure of a product state of a system of N particles. As the product state of a system grows exponentially with its size, physicist have found clever ways called "coarse graining" to compress them using tensor networks. The method presented in Stoudenmire (2018) use a particular type of tensor network: a tensor tree network.

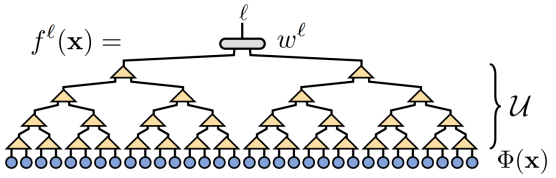

Figure 1. Tree tensor approximating $\mathbf{U}$

The intuition behind coarse graining is the following: we first define an ordering in our system and partition it in neighborhoods. Then, we find a mapping that summarize each of these small scale systems. Then, we consider the set of summaries as our new system, and iterate the procedure until we have a system of only two elements. What we will keep in memory is the tree of mappings that hierarchically summarized our coarse grained systems. This tree will constitute our approximation of $\mathbf{U}$ as Figure 1 show. Let's formalize the procedure.

2.1.1. Goal

We want to approximate $\mathbf{U}$ that diagonalizes the covariance matrix ρ defined in Figure 2 and which we can write as:

$$\rho = \frac{1}{N_T}\mathbf{\Phi}\mathbf{\Phi}^\top \tag{3}$$

$$= \frac{1}{N_T}\sum_{j=1}^{N_T}\Phi(\mathbf{x_j})\Phi(\mathbf{x_j})^\top \tag{4}$$

$$= \frac{1}{N_T}\sum_{j=1}^{N_T}\left(\phi^{s_1}(x_1^j)\phi^{s_1}(x_1^j)^\top\right) \otimes ... \otimes \left(\phi^{s_N}(x_N^j)\phi^{s_N}(x_N^j)^\top\right) \tag{5}$$

with $\otimes$ the kronecker product. We can note that an ordering had to be specified in order to assign indices to the x_i^j.

Note: To simplify the notations we will use $\phi_j^{s_i} := \phi^{s_i}(x_i^j)$.

$$\rho = \Phi\Phi^\dagger = \frac{1}{N_T}$$

Figure 2. The covariance matrix ρ.

2.1.2. Local covariance matrices

We partition our set of local feature maps by forming couples and define local covariance matrices $\rho_{k\ k+1}$ as showed in (a) of Figure 3. Let's take the example of ρ_{12}:

$$\rho_{12} = \frac{1}{N_T}\sum_{j=1}^{N_T}\left(\left(\phi_j^{s_1}\phi_j^{s_1\top}\right) \otimes \left(\phi_j^{s_2}\phi_j^{s_2\top}\right)\right) \times Tr(\phi_j^{s_3}\phi_j^{s_3\top}) \times \\ ... \times Tr(\phi_j^{s_N}\phi_j^{s_N\top}) \tag{6}$$

2.1.3. Local truncated U

Once the local covariance matrices are defined, we can summarize the local information by only retaining the most

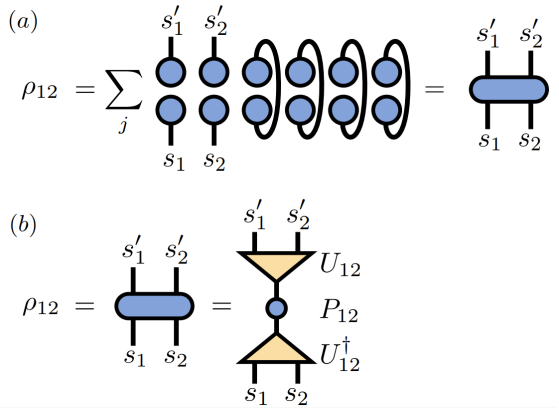

Figure 3. Local covariance matrices and local isometries

informative eigenvectors in the eigenvalue decomposition of $\rho_{k\ k+1}$. Example with ρ_{12}:

$$\rho_{12} = \underbrace{\mathbf{U}_{12}}_{\in\mathbb{R}^{4\times 4}} \underbrace{\mathbf{P}_{12}}_{\in\mathbb{R}^{4\times 4}} \underbrace{\mathbf{U}_{12}^\top}_{\in\mathbb{R}^{4\times 4}} \tag{7}$$

$$\rho_{12} \simeq \underbrace{\mathbf{U}^*_{12}}_{\in\mathbb{R}^{4\times r_{12}}} \underbrace{\mathbf{P}^*_{12}}_{\in\mathbb{R}^{r_{12}\times r_{12}}} \underbrace{\mathbf{U}^{*\top}_{12}}_{\in\mathbb{R}^{r_{12}\times 4}} \tag{8}$$

with r_{12} defined as the number of eigenvalues to keep so that the truncation error is less than ϵ. If we suppose that the eigenvalues are ordered in **decreasing order** in $\mathbf{P}_{12}$, then the truncation error E is:

$$E = \frac{\sum_{i=r_{12}}^{4} p_{ii}}{Tr(\rho_{12})} < \epsilon \tag{9}$$

2.1.4. New local feature maps

We use the set of local truncated $\mathbf{U}$ to create a new set of local feature maps:

$$\Phi_j^{s_1\ new} = \mathbf{U}_{12}^{*\top} vec\big(\phi_j^{s_1}\phi_j^{s_2\top}\big) \in \mathbb{R}^{r_{12}} \tag{10}$$

with $vec\big(\phi_j^{s_1}\phi_j^{s_2\top}\big)$ the vectorization of the outer product between the two vectors. Then, we normalize the new local feature maps.

2.1.5. Repeat

We repeat the 3 last steps until there remains only 2 top local feature maps of respective dimension t_1 and t_2. Then, we can define the reduced representation of the input which is the application of the tree tensor to the feature mapped data:

$$\tilde{\mathbf{\Phi}}^{t_1 t_2}(\mathbf{x}) = \sum_{s_1 \dots s_N} \boldsymbol{\mathcal{U}}^{t_1 t_2}_{s_1 \dots s_N} \mathbf{\Phi}(\mathbf{x}) \tag{11}$$

2.2. Supervised Optimization

Once we trained in an unsupervised way our approximation of $\mathbf{U}$ as a tree tensor, we can train our top tensor for the end task defined in (2). We showed at the beginning of this section that $\mathbf{W}$ is in fact a linear combination of the rows of $\mathbf{U}^\top$ so we can introduce $\mathbf{w}^\top = [w_1, ..., w_r]$ the row vector containing the weights so that $\mathbf{W} = \mathbf{w}^\top \mathbf{U}^\top$ and r being the dimension of the space spanned by the $\Phi(\mathbf{x_j})$. In the end, we thus have $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{U}^\top \Phi(\mathbf{x})$. But thanks to the coarse graining procedure, we can approximate $\mathbf{U}^\top \Phi(\mathbf{x})$ using the reduced representation defined in (11) which is found by applying the tree tensor $\boldsymbol{\mathcal{U}}$ on $\Phi(x)$ and outputs an order two tensor $\tilde{\mathbf{\Phi}}^{t_1 t_2}(\mathbf{x})$ which we can vectorize in a vector $\tilde{\Phi}(\mathbf{x})$ of dimension $t_1 \times t_2$. Therefore, in the end we have $f(\mathbf{x}) = \mathbf{w}^\top \tilde{\Phi}(\mathbf{x})$, so we can identify $r = t_1 \times t_2$ and train $\mathbf{w}$ as a linear regression task.

In the multitask setting, we train a collection of $\mathbf{w}_l^\top$ vectors, one for each label l (Stoudenmire (2018) found it sufficient to use the same tree network $\boldsymbol{\mathcal{U}}$ for each label). So in practice, we could just train a matrix which is the concatenation of the row vectors $\mathbf{w}_l^\top$.

3. Experiments

For all experiments, we run the two phases of the algorithm as described in the original work. To begin, we construct a tree tensor using the method detailed in Section 2.1. We then run a supervised, logistic regression algorithm on the final weight tensor. For all experiments with the exception of those detailed in Section 4.4, we use the MNIST image database (Lecun, 1998), which consists of 60,000 train images and 10,000 test images of 10 classes. At the beginning of each section, we note how many images we use to construct the tree tensor (for discussion on why this number is different than the training set size, please see Section 3.1) and what truncation cutoff we are using. For the supervised learning phase, we use the full training and test sets, and report accuracy on both. We also report the final dimensions of the top tensor, which is a pseudo-metric for expressivity of the representation (larger dimensionality, more expressivity).

3.1. Effect of Batch Size

The original proposal requires using the entire training dataset for the unsupervised construction of the tree tensor. Depending on implementation and compute constraints, this can either be overly expensive (keeping the entire transformed dataset in memory, and performing costly outer product operations on various elements), or incredibly slow (if reading from disk).

In some of the later layers of the tree tensor, some of the

coarse-grained feature vectors, $\phi(x)$ can become of moderate dimensionality (150 - 200 elements). While seemingly harmless, as the algorithm requires an explicit representation of outer products (shown below, in order to decompose the resulting matrix), the method requires consistently storing large matrices in memory:

$$SVD(\phi(x) \otimes \phi(x) \otimes \phi(y) \otimes \phi(y))$$

where $\phi(x)$ and $\phi(y)$ are arbitrary, coarse-grained vectors. We show the correlation between memory usage and batch size in Table 1.

In an attempt to save memory (since the outer product issue above cannot be avoided), we only use a subset of the training data to store in memory. In Table 2 and Table 3, we ablate the subset size on algorithm performance.

BATCH SIZE	TRUNCATION ϵ	PEAK RAM USAGE
250	$1e-3$	8GB
500	$1e-3$	13GB
1000	$1e-3$	26GB
1000	$5e-4$	32GB
2000	$1e-3$	31.5GB

Table 1. Compute requirements ablated across batch sizes and truncation epsilons. All experiments run on multi-core, 32GB RAM machine.

BATCH SIZE	TRAIN ACC.	TEST ACC.	FINAL DIM.
250	10.9%	9.2%	(3, 2)
500	13.3%	11.0%	(8, 3)
1000	20.1%	15.4%	(18, 4)
2000	34.1%	24.2%	(22, 6)

Table 2. Classification accuracies for various batch sizes methods on MNIST with $\epsilon = 1e-3$.

BATCH SIZE	TRAIN ACC.	TEST ACC.	FINAL DIM.
250	15.4%	9.2%	(4, 2)
500	18.2%	11.0%	(7, 5)
1000	26.2%	22.4%	(26, 3)
2000[2]	-	-	-

Table 3. Classification accuracies for various batch sizes methods on MNIST with $\epsilon = 5e-4$.

We find that not only do we not match the reported numbers in the original work, but also see severe drops in performance as batch size decreases. Deep learning methods, even modest-sized networks, beat the numbers reported by this method. One can imagine that memory constraints would increase as bigger images and datasets are tackled (i.e ImageNet, CIFAR-10, CIFAR-100), our methods cannot afford to operate on the *entire* training set, similar to full-batch gradients hindering machine learning before the adoption of stochastic gradient descent.

[2]Unable to train; Memory Error; see Section 5

3.2. Effect of Approximation Accuracy

After diagonalizing the reduced covariance matrix as shown in Equation 8, we only keep the eigenvectors corresponding to the largest eigenvalues depending on a certain truncation cutoff ϵ. As we can see in Table 4, this parameter ϵ has a huge impact on the performance of the model. With smaller ϵ, we retain more capacity, which empirically leads to better performance.

ϵ	TRAIN ACC.	TEST ACC.	FINAL DIM.
$1e-2$	9.6%	7.5%	(3, 3)
$5e-3$	17.0%	16.8%	(7, 8)
$1e-3$	20.1%	15.4%	(18, 4)
$5e-4$	26.2%	22.4%	(26, 3)

Table 4. Classification accuracies for various truncation cutoff ϵ on MNIST with batch size $= 1000$.

3.3. Effect of Image Traversal

To construct the tree tensor, we require an image discretization. While in images the discretization is natural (pixels), it is less clear what order to do the traversal in. In the original paper. the author uses row-by-row traversal (Figure 4, first). Here, we examine the effects of traversal choice on optimization and performance by comparing four methods (shown in Figure 4, left to right):

Figure 4. Traversal choices visualized. Ordering is Red-Green-Blue, lightest-to-darkest.

- Row-by-Row: Traverse all pixels in first row, move down.
- Column-by-Column: Traverse all pixels in first column, move right.
- Spiral: Start from center of image, move outwards in clockwise manner.
- Block: Inspired by ConvNet filters, a 2x2 block of image flattened out.

Using a batch size of 1000 and truncation cutoff $\epsilon = 1e-3$, we present the results when varying the traversal order in Table 5.

TRAVERSAL	TRAIN ACC.	TEST ACC.	FINAL DIM.
ROWS	20.1%	15.4%	(18, 4)
COLUMNS	13.3%	9.8%	(5, 3)
SPIRAL	8.7%	4.9%	(2, 4)
BLOCK	22.6%	18.1%	(17, 5)

Table 5. Classification accuracies for various traversal methods on MNIST with batch size $= 1000$ and $\epsilon = 1e-3$.

A concern that we raise here regards the disparity in training and testing accuracies between *arbitrary* choices of traversal. Since the method does not prescribe an optimal traversal, and since the traversals do not need to fulfill specific requirements (i.e filters in convolutional neural networks are learned in order to be translationally invariant), the drop in performance between traversal methods may be due to data-specific features. Data dependent performance can be hard to debug, but this phenomenon may need to be tested on harder datasets to be claimed as true.

3.4. Effect of Feature Map

Before running the two phases of the algorithm: unsupervised coarse graining and supervised optimization, we apply first a non linear feature map Φ to represent the input data in a higher dimensional space. Though (Stoudenmire, 2018) only tested the local feature map $\phi^{s_j}(x_j) = [1, x_j]^T$ as seen in (Novikov et al., 2016), this choice is noted as arbitrary. Therefore, we investigate the influence of the choice of the local feature map to the performance of the proposed algorithm. We chose another local feature map $\phi^{s_j}(x_j) = [cos(\frac{\pi}{2}x_j), sin(\frac{\pi}{2}x_j)]^T$ from (Miles Stoudenmire & Schwab, 2016) and tested it on MNIST dataset. With a batch size of 1000 and $\epsilon = 1e-3$, we get 17.2% on the training set and 12.4% on the test set. Comparing with the feature map used in the original paper, we can see a decrease in the performance depending on the choice of the feature map.

4. Enhancements and New Experiments

In this section, we focus on augmentations and improvements to the original method. Many of these were found when trying to improve the results on small scale experiments, and through discussions with the original author, we find that these improvements can have meaningful impact on the unsupervised coarse graining procedure.

4.1. Annealing of Truncation Epsilon

With fixed batch sizes and a constant epsilon, we empirically found that later layers of the tree tensor would truncate more and more information, leading to an extremely small final weight tensor. Using a small number of parameters to fit a large dataset, we find that our models severely underfit. In an attempt to fix this issue, we experimented with linearly annealing the truncation epsilon, as to force the later layers to retain more information.

Using a batch size of 1000, we linearly anneal ϵ from $1e-3$ to $1e-4$ as the depth of the tree tensor increases, so that top layers have a smaller epsilon. With this enhancement, we found that we could increase test accuracy to 30.2%. Although empirically not a large gain, we were able to beat all fixed epsilon values shown in Table 4, while still using a small epsilon near end of training to retain more information (our tests with using a fixed epsilon of $\epsilon = 1e-4$ ran out of memory). We imagine that on large scale experiments, such a schedule would have a much greater effect, and reduce training time significantly.

4.2. Mini-batch Training

In the original paper, the entire MNIST dataset (60,000 images) was used to compute the tree tensor. Since we could keep only a truncated dataset in memory, we used a subset of the original data to perform the unsupervised coarse graining algorithm. However, related to the issue above, we found that later layers of the tree rapidly truncated information. We presume that this was due to the lack of diversity through layers; the original paper used random images to iteratively compute ρ, watching convergence of the covariance matrix in order to stop the process early. With a small amount of data, this same procedure seems to lead to a small final weight tensor.

To combat this, we increase diversity in our small dataset by sampling it randomly from the full dataset before building each layer. While we show an empirical improvement (up to 27.5% accuracy with batch size of 1000 and $\epsilon = 5e-4$), the downside of this approach is the increased expense: generally, we keep only the last layer's transformed Φ in memory, but with a new batch of data each time (sampled originally as images), we need to pass it through the entire tree tensor again, as compared to just the last layer. We imagine that more efficient algorithms will arise to solve this issue, just as deep learning libraries improved efficiency as the methods became more mainstream.

4.3. Partial Coarse Graining

Again, in an attempt to combat the small dimensionality of the final weight tensor in our experiments, we also explored a variation of the Partial Coarse Graining algorithm, detailed

in Section 6 of (Stoudenmire, 2018). Briefly, the partial algorithm does not use all the layers of the learned tree tensor, which generates a higher-dimensional weight tensor (represented in MPS form). The author is then able to learn a strong classifier on top of this higher-dimensional tensor.

In our experiment, we also generate a higher-dimensional representation of each data point by not passing it through the full tree tensor, but then take an outer product between each vector component. We flatten the result, and use it as the new, higher-dimensional representation of the data. However, we saw no empirical benefits of using this method, which points to the fact that the structure between the components of the higher-dimensional tensor seem crucial to learning.

4.4. Multi-task Learning

(a) MNIST (b) Fashion MNIST

Figure 5. Datasets used across all experiments.

The original paper discussed two hypotheses regarding the approach:

- The layers of a tree tensor likely learn a hierarchy of features, similar to those seen in convolutional neural networks (Zeiler & Fergus, 2013).
- The features learned are likely transferrable to new tasks.

In unsupervised learning, the goal is to learn a *representation* of data, where the representation *should* be useful on new, but related, data. To test these claims, we train a tree tensor (batch size of 1000, $\epsilon = 5e-4$) on MNIST, and use the representation to train a supervised learning algorithm on FashionMNIST (Xiao et al., 2017). We find that the transferred representation is able to generate a test performance of 19.2%, whereas the training a FashionMNIST specific tree tensor leads to a performance of 24.1%. This close gap between the two experiments shows that likely, both hypotheses are true, although this claim needs to be more rigorously tested.

The author provides a way to induce a *prior* on the tree tensor layers, by incorporating an MPS with a mixing parameter μ (for more details, see Section 5 of Stoudenmire (2018)).

The method is shown to be helpful when the prior is trained on the same dataset, but since the goal of unsupervised learning is to learn representations, we test the multi-task learning hypothesis by mixing the tree tensors of two datasets to see what representation they can learn together. Specifically, we run unsupervised coarse graining on MNIST and Fashion-MNIST, and at each stage, we use a convex combination (using μ) of the calculated covariance matrices:

$$\rho_{ij}^{MTL} = \mu\rho_{ij}^{MNIST} + (1-\mu)\rho_{ij}^{FMNIST} \tag{12}$$

and then decompose ρ_{ij}^{MTL} to construct a single, shared tree tensor.

We use the first six classes from each individual dataset for training, and use the final four to test how well our shared representation transfers, shown in Table 6. Interestingly, the transfer does not seem to be bidirectionally beneficial.

μ	MNIST Test Acc.	FMNIST Test Acc.
0.0	14.8%	24.1%
0.25	17.1%	23.0%
0.50	18.3%	18.9%
0.75	18.1%	22.6%
1.0	22.4%	19.2%

Table 6. Effect of μ in a multi-task learning setup, with batch size $= 1000$ and $\epsilon = 5e-4$.

5. Conclusion and Notes on Compute

In this report, we provide an empirical analysis of the methods described in Stoudenmire (2018), and provide simple enhancements and ablation studies of the algorithm. While our absolute accuracies are poor, we hope to focus on the *relative* differences between experiments. We imagine that while the accuracy values may not translate to large-scale experiments, the conclusions drawn from our findings will.

However, it is important to note that this method does not work at small scales *in its present form*. The original experiments described in Stoudenmire (2018) were run on machines with 300GB of RAM (some experiments required 512GB of RAM), whereas our experiments were limited by compute (32GB of RAM). To this end, we hope that our open-source code provides a test bed for future research, and aids in solving some of the issues we have discussed in the report.

References

Lecun, Y. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*, 1998. URL https://ci.nii.ac.jp/naid/10027939599/en/.

Levine, Y., Sharir, O., Cohen, N., and Shashua, A. Bridging

many-body quantum physics and deep learning via tensor networks. 2018.

Miles Stoudenmire, E. and Schwab, D. Supervised learning with quantum-inspired tensor networks. 05 2016.

Novikov, A., Trofimov, M., and Oseledets, I. Exponential machines. *Bulletin of the Polish Academy of Sciences: Technical Sciences*, 05 2016. doi: 10.24425/bpas.2018.125926.

Stoudenmire, E. M. Learning relevant features of data with multi-scale tensor networks. *CoRR*, abs/1801.00315, 2018. URL http://arxiv.org/abs/1801.00315.

Xiao, H., Rasul, K., and Vollgraf, R. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747, 2017. URL http://arxiv.org/abs/1708.07747.

Zeiler, M. D. and Fergus, R. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013. URL http://arxiv.org/abs/1311.2901.