EFFICIENT HYPERPARAMETER OPTIMISATION THROUGH TENSOR COMPLETION

Anonymous authors
Paper under double-blind review

ABSTRACT

Hyperparameter optimisation is a prerequisite for state-of-the-art performance in machine learning, with current strategies including Bayesian optimisation, hyperband, and evolutionary methods. While such methods have been shown to improve performance, none of these is designed to explicitly take advantage of the underlying data structure. To this end, we introduce a completely different approach for hyperparameter optimisation, based on low-rank tensor completion. This is achieved by first forming a multi-dimensional tensor which comprises performance scores for different combinations of hyperparameters. Based on the realistic underlying assumption that the so-formed tensor has a low-rank structure, reliable estimates of the unobserved validation scores of combinations of hyperparameters can be obtained through tensor completion, from knowing only a fraction of the elements in the tensor. Through extensive experimentation on various datasets and learning models, the proposed method is shown to exhibit competitive or superior performance to state-of-the-art hyperparameter optimisation strategies. Distinctive advantages of the proposed method include its ability to simultaneously handle any hyperparameter type (e.g., kind of optimiser, number of neurons, number of layer, etc.), its relative simplicity compared to competing methods, as well as the ability to suggest multiple optimal combinations of hyperparameters.

1 INTRODUCTION

Machine learning (ML) applications have been steadily growing in number and scope over recent years, especially in Computer Vision and Natural Language Processing. This growth is mainly attributed to the ability of large deep learning (DL) models to learn very complex functions. Consequently, the performance of such models (but even of simpler models that perform less computationally heavy tasks) is critically dependent on fine-tuning of their internal hyperparameters. Given the importance of hyperparameter optimisation and the ever increasing complexity of training modern ML models, efficient tuning of hyperparameters has become an area of utmost importance, not only in obtaining state-of-the-art performance, but also in alleviating the complexity of the training process. It is therefore not surprising that a large effort of the machine learning community has been focused on developing efficient hyperparameter optimisation methods. Commonly used methods include grid search, random search, methods based on Bayesian optimisation, multi-fidelity optimisation and evolutionary strategies. For a comprehensive review of currently available hyperparameter optimisation methods we refer the reader to Yu & Zhu (2020).

Despite success, current approaches are either rather heuristic or depend on strong underlying assumptions. To this end, we introduce a radically different approach based on exploiting the low-rank structure of the hyperparameter space. For example, we expect that a given optimiser with its learning rate set to $1e-3$ will most likely yield similar validation loss performance to a learning rate of $9.9e-4$. Extending this argument to multiple dimensions, so as to reflect multiple hyperparameters, we aim to identify highly promising subspaces in the vast space of hyperparameter combinations by evaluating only a small fraction of these combinations. More specifically, the proposed method models the set of all possible hyperparameter combinations as a multidimensional tensor. Each entry in this tensor corresponds to a score indicating relative suitability of a particular hyperparameter combination with respect to others (e.g., validation loss). By evaluating a subset of these combinations, we construct an incomplete tensor with only a fraction of its elements known. Assuming

the complete tensor is of low rank, the unknown entries can be estimated using low-rank tensor completion techniques. This makes it possible to predict the relative performance of different hyperparameter configurations without the need for their explicit evaluation. The best hyperparameter configurations can then be found by searching the so completed tensor.

To take full advantage of the tensor completion framework, we propose a sequential tensor completion algorithm which narrows down the hyperparameter search space based on identified promising subspaces from previous tensor completion cycles. Furthermore, for each cycle, we employ the Cross method (Zhang (2019)), a tensor sampling scheme which ensures that as few hyperparameter evaluations as possible are required for accurate tensor completion. We show that such an approach results in a speed of optimisation that is highly competitive with, and often surpassing, other state-of-the-art hyperparameter optimisation techniques. Comprehensive numerical results and extensive experimentation illustrate the potential of the proposed framework as a competitive and intuitive, yet physically meaningful, alternative option for efficient hyperparameter optimisation.

The rest of the paper is organised as follows. Section 2 discusses related work. After briefly discussing key tensor preliminary concepts in Section 3, we present a validation of the assumed low-rank property of the hyperparameter tensor in 4 and the proposed tensor completion algorithm for hyperparameter optimisation in Section 5. Next, comprehensive experimental results are presented in Section 6, followed by the Conclusion in Section 7.

2 Related work

Deng & Xiao (2022) provide a meta-learning approach based on low-rank tensor completion which allows the optimal hyperparameter configuration in a new machine learning problem to be predicted based on the optimal configurations found in other related problems. Our work is crucially different since it does not require the results of related problems or any other prior knowledge to optimise a given machine learning problem. A plethora of hyperparameter optimisation techniques exist that also do not require prior knowledge of the problem e.g. random search or Bayesian optimisation. However, to the best of our knowledge, there is no such hyperparameter optimisation method based on low-rank tensor completion. Furthermore, we have been unable to find any work hypothesising that the performance distribution of various hyperparameter combinations has an underlying low-rank structure, even though this is a natural assumption for any physically meaningful data structure.

3 Tensor Preliminaries

Low-rank tensor completion is based on the premise that elements in a low rank tensor exhibit certain interrelationships. This makes it possible to infer values of unknown elements based on the elements whose values are known. Research abounds with algorithms for tensor completion e.g., Bengua et al. (2017), Song et al. (2018), Liu et al. (2014) or Acar et al. (2011); however, most of these algorithms assume there is no prior knowledge of which elements of the true tensor are known. In the strategy proposed by this paper, one is able to "choose" which elements of the tensor are known by choosing which hyperparameter combinations to evaluate. Since each hyperparameter evaluation is time-consuming, it is important to be achieve accurate tensor completion with as few known elements as possible; this capability is provided by the Cross technique for efficient low rank tensor completion from Zhang (2019).

To explain this technique, it is necessary to explain two important prerequisites: Tucker decomposition and Tucker rank. Any N-dimensional tensor $\mathcal{X}$ can be expressed as a Tucker decomposition, consisting of an N-dimensional core tensor $\mathcal{G}$ and N factor matrices $A^{(n)}$, one for each dimension. The relationship between the tensor $\mathcal{X}$ and its Tucker decomposition is

$$\mathcal{X} = \mathcal{G} \times_1 A^{(1)} \times_2 A^{(2)} \ ... \ \times_N A^{(N)} \tag{1}$$

The operation $\times_n$ denotes the mode-n product. Consider a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 ... \times I_N}$ and matrix $A \in \mathbb{R}^{J \times I_n}$, such that $1 \leq n \leq N$. The n-mode product of $\mathcal{X}$ and A is a tensor whose elements are

$$(\mathcal{X} \times_n A)_{i_1,i_2...i_{n-1},j,i_{n+1}...i_n} = \sum_{i_n=1}^{I_n} x_{i_1,i_2...i_{n-1},i_n,i_{n+1}...i_n} \cdot a_{j,i_n} \tag{2}$$

where x and a are respectively elements of $\mathcal{X}$ and A with indices given by their subscripts. Based on the Tucker decomposition, the Tucker rank of a tensor $\mathcal{X}$ is a list containing the smallest possible dimensions of $\mathcal{G}$ in a Tucker decomposition that accurately reconstructs $\mathcal{X}$. Tucker rank is the definition of tensor rank used throughout this paper. For more details, refer to Cichocki et al. (2015).

Let the tensor $\mathcal{X}$ to be completed have dimensions $I_1 \times I_2...I_N$. The Cross technique requires the Tucker rank of the complete tensor to be assumed beforehand - let this assumed rank be $[r_1, r_2...r_N]$. It provides a heuristic to accurately estimate the complete tensor by sampling $r_1 \times r_2...r_N + r_1 \times (I_1 - r_1) + r_2 \times (I_2 - r_2)...r_N \times (I_N - r_N)$ elements - the authors prove this is the minimum number of elements required for accurate tensor completion at a given assumed rank. The heuristic samples a body tensor of dimensions $r_1 \times r_2...r_N$, starting from $\mathcal{X}_{0,0...0}$, and r_n arm vectors for each n^{th} tensor dimension that stretch along the entire dimension and must intersect with the body. The authors of Zhang (2019) provide 2 algorithms to generate a complete tensor estimate from these samples; throughout this paper we use "Noisy Tensor Completion with Cross Measurements", which accounts for the presence of "noise" that prevents the tensor from being perfectly low-rank.

4 Investigation of the Low Rank Property of Hyperparameter Tensors

Before introducing the proposed hyperparameter optimisation algorithm, it is necessary to provide validation to the assumption that a tensor constructed from validation losses of hyperparameter combinations, when constructed in a particular manner, is of low (Tucker) rank. This section aims to illustrate this through experiments on widely used machine learning setups.

4.1 Mapping the Hyperparameter search space to the tensor

The first step is to discuss how we construct the tensor $\mathcal{T}$ based on the sets of possible values i.e., search spaces for the hyperparameters. Firstly, if there are N hyperparameters, the tensor $\mathcal{T}$ would be N-dimensional. Each hyperparameter H corresponds to one dimension of $\mathcal{T}$ - there is no constraint on which dimension this can be. In experiments, we consider three kinds of hyperparameters based on their search spaces: *categorical*: a set of category-based values; *uniform integer*: a uniformly spaced sequence of integers and *uniform real* a uniformly spaced sequence of real numbers.

The search space for a *categorical* hyperparameter, H_{cat}, is specified as a list: consider the activation function in neural networks as an example: ["*ReLU*", "*tanh*", "*sigmoid*"]. If H_{cat} corresponds to the l^{th} dimension of $\mathcal{T}$ then, based on the example provided, elements of $\mathcal{T}$ with index 0 for the l^{th} dimension correspond to all hyperparameter combinations with H_{cat} ='*ReLU*'. Similarly, indices 1 and 2 correspond to '*tanh*' and '*sigmoid*', respectively. The search space for a *uniform integer* or *real* hyperparameter is specified through three values: start s, resolution interval r, and end e. For *uniform integer* hyperparameters, r is an integer, while for *uniform real* r is real. The search space is then $\{s, s + r, s + 2r, ..., min(e, \lfloor (e - s)/r \rfloor \times r)\}$. Let H_{uni} be a uniform integer or real hyperparameter on the m^{th} dimension of $\mathcal{T}$. The elements of $\mathcal{T}$ whose index for the m^{th} dimension is i correspond to all hyperparameter combinations having $H_{uni} = s + i \times r$.

There is thus a mapping between hyperparameter combinations and tensor indices, generated based on the set of search spaces for the hyperparameters. Each tensor element represents the validation loss of the hyperparameter combination corresponding to that index. We hypothesize that a tensor constructed in this way is (at least approximately) of low rank.

4.2 Setup for Tensor Completion Experiments

Experiments were performed with the following machine learning problems: support-vector machine (SVM) with polynomial kernel in binary classification of the iris data set from Fisher (1936) (abbreviated **SVM-P-iris**); K-NN regression on the diabetes data set from Efron et al. (2004) (**KNN-R-diab**), and random forest binary classification of the wine data set (**RF-wine**). The wine data set

of size 178 was modified for binary classification by retaining only samples from classes 0 and 1 to give a data set of size 130.

For **SVM-P-iris**, 4 hyperparameters were used to construct the (4-dimensional) tensor: the SVM regularisation hyperparameter, and the degree, constant term and scaling factor of the polynomial kernel. For **KNN-R-diab**, 3 hyperparameters were optimised: the number of neighbours K of the K-NN algorithm, the order of Minkowski norm (used to calculate distances between points) and the weighting heuristic (*uniform* or *distance-based*) for values of the K neighbours when making a prediction at any data point. For **RF-wine**, 5 hyperparameters were optimised: the number of decision trees in the forest; maximum depth of any tree; minimum number of data samples needed to split a tree node; number of data features to consider when splitting a tree node, and a Boolean categorical hyperparameter determining whether a subset or the entire training data is used to construct each tree. The search spaces for all these hyperparameters are provided in Appendix A.1. For each of these problems p, a tensor $\mathcal{T}_p$ was generated based on the mapping discussed in part 4.1. $\mathcal{T}_p$ was then sampled and estimated through tensor completion based on the Cross heuristic. In all problems, validation loss was calculated using cross-validation with 5 folds predefined for each data set. The loss metric used to calculate validation loss varied across problems: *hinge* loss for **SVM-P-iris**, *logcosh* loss for **KNN-R-diab** and *Kullback-Leibler divergence* for **RF-wine**.

4.3 Tensor Completion Results

Table 1 represents the result of tensor completion using an assumed Tucker rank of 1 for each dimension i.e., $r_1 = r_2 ... r_N = 1$. Table 2 represents the best result obtained when using any Tucker rank different from the one used in Table 1. The columns represent metrics devised to compare the predicted tensor from tensor completion $\hat{\mathcal{T}}_p$ with the true tensor $\mathcal{T}_p$. NND stands for "normalised norm difference", and is given by:

$$NND = \frac{||\hat{\mathcal{T}}_p - \mathcal{T}_p||}{||\mathcal{T}_p||} \tag{3}$$

where $||\mathcal{X}||$ is the norm of a tensor $||\mathcal{X}||$, given by the square root of the sum of squares of all its elements. A lower NND indicates better accuracy, although $NND = 1$ is still poor: equivalent to all elements of $\hat{\mathcal{T}}_p$ being 0. CE10% is the percentage of tensor indices of the top 10% elements in $\hat{\mathcal{T}}_p$, by lowest validation loss, in common with the top 10% of $\mathcal{T}_p$. This metric provides an indication of whether tensor completion can identify the best hyperparameter combinations. Note that the number of elements in T_p is the same in tables 1 and 2; we found it convenient to display in table 1.

Metric	**SVM-P-iris**	**KNN-R-diab**	**RF-wine**
Elements in $\mathcal{T}_p$	111,600	20,000	4,500
Elements sampled	92	200	27
NND	0.09	0.09	0.27
CE10%	7.5%	14.6%	2%

Table 1: Tensor completion results using assumed Tucker rank of $[1, 1...1]$.

Metric	**SVM-P-iris**	**KNN-R-diab**	**RF-wine**
Tucker rank	[2,2,2,2]	[2,2,4]	[2,2,2,1,3]
Elements sampled	190	596	65
NND range	[0.97, 1.37]	[0.14, 1.00]	[0.92, 1.55]
NND mean	1.07	0.47	1.16
CE10% range	[0, 46.2]%	[0, 85.1]%	[0, 8]%
CE10% mean	19.9%	68%	1.8%

Table 2: Tensor completion results using best assumed Tucker rank $\neq [1, 1...1]$.

Due to the inherent randomness of the Cross heuristic, the set of elements of $\mathcal{T}_p$ sampled when the Tucker rank is not $[1, 1...1]$ varies over trials - hence the range and mean of each metric over 10 trials is provided in Table 2. As seen in table 1, tensor completion with the Cross technique consistently provides a degree of accuracy in approximation (NND < 1) when the Tucker rank is $[1, 1...1]$. This is in spite of the fact that across all problems, the proportion of tensor elements being sampled is always $< 1\%$. However, the CE10% value is always $< 15\%$, indicating that, while some of the best validation loss elements are identified, many are not. When higher rank values are used as in Table 2, the NND worsens to the extent that NND ≈ 1 but the CE10% performance improves. These results suggest that a Tucker rank of 1 for each dimension is able to capture general trends in the variation of validation loss values throughout the tensor - this explains why it can consistently produce low NND values. This can only be true if the validation loss values can be roughly described by a low-rank structure of rank $[1, 1...1]$. However, this Tucker rank is less effective at capturing specific local variations in the values, which is why its CE10% performance is lower. On the other hand, using a higher Tucker rank may at times be able to capture local variations accurately - hence the higher CE10% - it tends to overfit these variations i.e., emphasise the variations over the underlying low rank structure. This results in more variable performance and poorer NND. For higher Tucker ranks to be able to capture the (approximate) rank $[1, 1...1]$ structure underlying the validation loss values, a larger number of samples of $\mathcal{T}_p$ would be required.

For hyperparameter optimisation, we decided that while in some cases, a higher rank may prove more effective at identifying the best hyperparameter combinations, using a Tucker rank of $[1, 1...1]$ is the best default choice as it can capture the structure of different $\mathcal{T}_p$, while ensuring the best time performance by evaluating the lowest number of hyperparameter combinations.

5 Tensor Completion for Hyperparameter optimisation

In this section, we present a technique for hyperparameter optimisation based on tensor construction and completion as performed in Section 4. Throughout this paper, we refer to this technique as "Hyperparameter optimisation through tensor completion", abbreviated HOTC. From the results of Section 4, it is apparent that low-rank tensor completion can approximate the global distribution of validation losses over hyperparameter combinations, but may miss local variations. Accounting for these variations is, however, crucial to obtaining the optimal hyperparameter combination. Therefore, we decided a suitable approach would be to use tensor completion to predict the general region of the search space most likely to hold the optimal combination(s), and then focus the optimisation on this region for another round of tensor completion. This focusing of the optimisation can be repeatedly applied until the region being searched is small enough for an exhaustive i.e., grid search.

5.1 The Proposed Algorithm

Algorithm 1 describes HOTC. Note that the search space for each hyperparameter H is as defined in part 4.1. In each tensor completion cycle, the function $generate\_tensor\_cross\_components$ first evaluates different hyperparameter combinations i.e., samples the true validation loss tensor according to the Cross sampling scheme. It returns the sampled elements in the form of Cross measurements $\mathbf{B}, \mathcal{J}$ and $\mathcal{A}$ which are respectively the body tensor, array of joint matricisations and array of arm matricisations - see Zhang (2019) for more information. The function $generate\_complete\_tensor$ then applies the noisy tensor completion algorithm from Zhang (2019) to estimate the complete tensor. In this, $\mathbf{B}, \mathcal{J}, \mathcal{A}$ are used to generate the core and factor matrices of a Tucker decomposition of the rank $\mathbf{T}$ estimate of the true tensor $\mathcal{T}$. An overall illustration of how true tensor samples can form a Tucker decomposition of the estimate is illustrated in figure 1.

The hyperparameter combination $\mathbf{h}$ with the lowest validation loss $\mathcal{T}$ is then found by the function $find\_best\_combination$, which converts the tensor index to its corresponding hyperparameter combination. The function $narrow\_search\_spaces$ then generates a new version of $\mathbf{S}$ with smaller search spaces centred around the values in the hyperparameter combination, $\mathbf{h}$. The search space for each hyperparameter is narrowed around its corresponding value in $\mathbf{h}$; assume this value to be $h_{H_{uni}}$ for a *uniform integer/real* hyperparameter H_{uni}. Let the start, resolution interval, and end (see part 4.1 for definitions) of the original search space before narrowing for H_{uni} be s, r and e respectively. The range of values searched in this space is $G_i = \lfloor (e - s)/r \rfloor \times r$; the range of the narrowed space G_{i+1} would be at most $\lfloor G_i/2r \rfloor \times r + 1$ with $h_{H_{uni}}$ at the centre of the new sequence of

Algorithm 1 The HOTC Algorithm

Input

C	Number of tensor completion cycles
M	Maximum number of hyperparameter combinations for grid search
r_{min}	Minimum search space resolution interval for real-valued hyperparameters
$\mathbf{T}$	Tucker rank assumed for the completed tensor
f	Function to generate validation loss for a hyperparameter combination
$\mathbf{S}$	Map of each hyperparameter to its search space

$\mathbf{h} \leftarrow \phi$
for $cycle\_num$ in $1, 2...C$ **do**
  $\mathbf{B}, \mathcal{J}, \mathcal{A} \leftarrow generate\_tensor\_cross\_components(f, \mathbf{S}, \mathbf{T})$
  $\mathcal{T} \leftarrow generate\_complete\_tensor(\mathbf{B}, \mathcal{J}, \mathcal{A})$
  $\mathbf{h} \leftarrow find\_best\_combination(\mathcal{T}, \mathbf{S})$
  if number of elements in $\mathcal{T} \leq M$ **then**
    $\mathbf{h} \leftarrow grid\_search(f, \mathbf{S})$
    Exit loop
  end if
  if $cycle\_num < C$ **then**
    $\mathbf{S} \leftarrow narrow\_search\_spaces(\mathbf{S}, \mathbf{h}, r_{min})$
  end if
end for
Output

$\mathbf{h}$	Selected hyperparameter combination

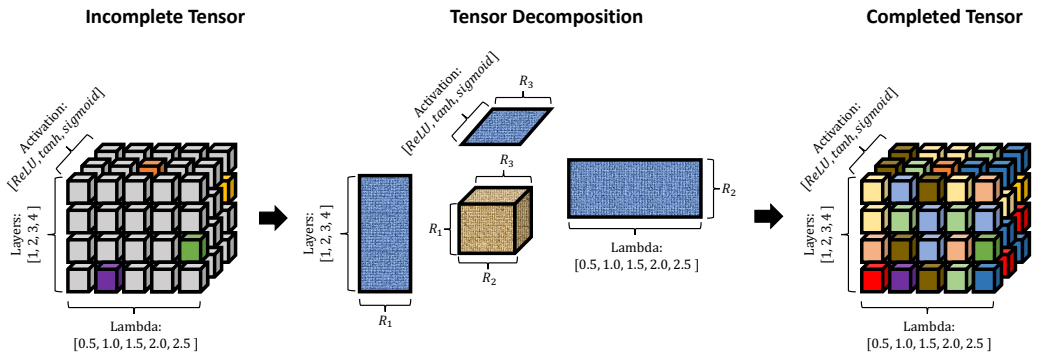

Figure 1: Illustration of a tensor completion approach, based on Tucker decomposition, for hyperparameter optimisation. Missing values of the incomplete tensor are designated in grey.

values. The new start and end of the search space hence become $\max\left(h_{H_{uni}} - \lfloor (G_i/4r) \rfloor \times r,\ s\right)$ and $\min\left(h_{H_{uni}} + \lfloor (G_i/4r) \rfloor \times r,\ e\right)$ respectively. The new resolution interval is $\max(\lfloor r/2 \rfloor, r_{min})$ when H_{uni} is *uniform real*, or $\max(\lfloor r/2 \rfloor, 1)$ for *uniform integer* hyperparameters.

Some *categorical* hyperparameters may take numerical category values, where closer numbers indicate a closer relationship. To narrow the search space of such a hyperparameter, H_{cat}, the list of numerical values representing the search space is first sorted in an ascending order. The narrowed search space is then the sequence of values from the a^{th} to the b^{th} element (both inclusive) of the sorted list, where $a = \max\left(i - round\_to\_integer(L_i/4),\ 1\right)$ and $b = \min\left(i + round\_to\_integer(L_i/4),\ L\right)$. Here, the length of the original list is L_i and $h_{H_{cat}}$ is the i^{th} element in the sorted list. The size L_{i+1} of the new search space list is at most $\lceil L_i/2 \rceil + 1$, with $h_{H_{cat}}$ roughly in the middle. The search spaces of categorical hyperparameters taking unrelated non-numerical values e.g.,['*ReLU*', '*tanh*', '*sigmoid*'], are not narrowed down, and left as they are.

Once the search spaces have been sufficiently narrowed that the tensor estimate $\mathcal{T}$ has $< M$ elements, $grid\_search$ performs an exhaustive grid search by evaluating every hyperparameter com-

bination in $\mathbf{S}$ to find the lowest validation loss. Note that M can be set to 0, in which case the predictions would be purely from tensor completion cycles without any grid search.

5.2 Heuristics to Enhance Performance

It is important to mention heuristics to obtain the best performance from Algorithm 1, which has numerous configuration parameters. These heuristics were followed to generate the results in Section 6. Firstly, it is advisable to define large resolution intervals for the initial search spaces in $\mathbf{S}$. For example, if a uniform integer hyperparameter has a start and end of 1 and 101 respectively, a good choice for the resolution interval would be 10 or 20, so that 10 or 5 values are being searched in the completion cycle. As discussed in part 4.3, a Tucker rank of 1 for every dimension is the best default choice. With these settings, the algorithm should narrow the search space, with each tensor completion cycle, on areas likely to contain the optimal hyperparameter combinations. The resolution intervals would automatically decrease from their large values to r_{min}.

Once the initial search spaces are defined, the next step is to run the algorithm for one completion cycle, keeping the grid search limit M as zero (i.e., disabling grid search) and to observe the performance. The number of cycles, C, can be gradually increased until no improvement in performance is obtained on running the algorithm, or the maximum amount of time acceptable for optimisation is reached. At this stage, performance may be further improved by setting $M > 0$, enabling a grid search - at the cost of more execution time, this ensures that the best possible validation loss in the area of the search space being examined has been obtained.

It is possible to predict how much time the algorithm will take, if one knows the time required to evaluate one hyperparameter combination; let this be t. Consider a tensor $\mathcal{T}$ of dimensions $I_1 \times I_2 ... I_N$. To estimate $\mathcal{T}$ using the Cross technique, with assumed Tucker rank $[r_1, r_2 ... r_N]$, it is necessary to sample $samp = r_1 \times r_2 ... r_N + r1 \times (I_1 - r_1) + r_2 \times (I_2 - r_2) ... r_N \times (I_N - r_N)$ elements of $\mathcal{T}$ under the Cross scheme. In the context of algorithm 1, for the first completion cycle, I_n is the number of values in the initial search space for the n^{th} hyperparameter. Estimating the completed tensor based on the sampled elements involves N n-mode tensor multiplications that take negligible time compared evaluating $samp$ hyperparameter combinations. Thus, the first completion cycle takes roughly $samp \times t$ in time. In subsequent completion cycles, the tensor to be completed has smaller dimensions due to narrowing of the search space - thus $samp \times t$ is an upper bound for the time taken by these cycles. The maximum time taken by the completion cycles is thus $\approx C \times samp \times t$. If a grid search is performed at the end, the maximum time is $\approx (C \times samp) \times t + M$.

6 Evaluation of HOTC Against Alternative Hyperparameter Optimisation Techniques

The HOTC technique was compared with six existing hyperparameter optimisation techniques: random search (RS) (Bergstra & Bengio (2012)); Bayesian optimisation with Gaussian process (BO-GP) (Frazier (2018)); Bayesian optimisation with Tree-Parzen estimator (BO-TPE) (Feurer & Hutter (2019)); covariance matrix adaptation evolution strategy (CMA-ES) (Hansen (2016)); hyperband (HB) (Li et al. (2018)), and Bayesian optimisation hyperband (BOHB) (Falkner et al. (2018)).

The techniques were compared across five benchmark machine learning problems: K-nearest neighbours (K-NN) binary classification on the wine data set (abbreviated **KNN-C-Wine**); K-NN regression on the California Housing data set (**KNN-R-Calh**); random forest binary classification on the Forest Covertype data set (**RF-FC**); binary classification with a 3-layer convolutional neural network (CNN) on the MNIST data set (**3LC-MNIST**), and transfer-learning in the VGG16 CNN architecture in multi-class classification of the CIFAR10 data set (**VGG-CIF10**).

For each hyperparameter in each problem, the limits (maximum and minimum) of the range of values to search were kept identical for each hyperparameter optimisation technique. In **KNN-C-Wine**, **KNN-R-Calh** and **RF-FC**, each hyperparameter combination was evaluated using cross-validation loss over 5 fold pre-defined for each data set. This ensures a consistent loss value is obtained over multiple trials. For **3LC-MNIST** and **VGG-CIF10**, separate training and validation data sets, in size ratio $80:20$, were used instead, to enable faster calculation of validation loss.

6.1 IMPLEMENTATION DETAILS

In **KNN-C-Wine**, 3 hyperparameters were optimised: the number of neighbours K of the K-NN algorithm, the order of Minkowski norm (used to calculate distances between points) and the weighting heuristic (*uniform* or *distance-based*) for values of the K neighbours when making a prediction at any data point. The wine data set of size 178 was modified for binary classification to only retain samples from classes 0 and 1. The resulting set was of size 130. The fraction of misclassified validation data samples i.e., misclassification loss was the metric used to represent the validation loss. In **KNN-R-Calh**, 2 hyperparameters were optimised: the number of neighbours K and the order of Minkowski norm. Only the *uniform* wighting heuristic could be used here, as the California Housing data set had coinciding elements that resulted in infinite distance-based weights. The *logcosh* metric was used to calculate validation loss. In **RF-FC**, 5 hyperparameters were optimised: the number of decision trees in the random forest; maximum depth of any tree; minimum number of data samples needed to split a tree node; number of data features to consider when splitting a tree node, and a boolean hyperparameter determining whether a subset or the entire training data is used to construct each tree. The Forest Covertype data set, of initial size $581,012$ was too large to be processed in RAM. Hence, it was modified for binary classification by retaining the first $10,000$ samples of classes 1 and 2 each. This gave an "evenly balanced" data set of size $20,000$. The truncation also made the data set faster to train on. In **3LC-MNIST**, the CNN had 2 convolutional layers and 1 fully-connected output layer. 11 hyperparameters were optimised, the most among all the benchmark problems. These were: the numbers of neurons, activation functions, dimensions of max pooling and dimensions of stride of the first and second convolutional layers, as well as the neural network optimiser, learning rate and learning rate decay rate. To evaluate each hyperparameter combination, the CNN was trained for 5 epochs. The transfer learning problem in **VGG-CIF10** involved freezing its 16 convolutional layers and training its 3 fully connected output layers. These layers were of size 4096, 4096 and 10 (to classify 10 classes) resulting in $18,923,530$ trainable parameters. 7 hyperparameters were optimised: the optimiser, learning rate, learning rate decay, number of epochs to train the network, batch size and dropout probabilities of the fully connected layers of size 4096. It should be noted that the 32×32 CIFAR10 were resized to 224×224 to be accepted by the VGG16 network.

The search spaces for each of these hyperparameters are described in Appendix A.2. The computing hardware, software implementations of the different hyperparameter optimisation techniques, and the configurations of these techniques are described in Appendix B.

6.2 NUMERICAL RESULTS

The code for HOTC and our benchmarks is publicly available on GitHub. Figure 2 illustrates the results of the experiments. The graphs were generated for RS, BO-TPE, CMA-ES, BO-GP, HB and BOHB by sampling the best obtained validation loss at the timestamp of every trial. For HOTC, this sampling was done after each completion cycle and after the final grid search, if any. For **KNN-C-Wine**, **KNN-R-Calh** and **RF-FC**, the HOTC approach is remarkably faster than the other techniques and also obtains the lowest value of validation loss. It is not possible to run HOTC for longer than is seen on these graphs, as the completion cycles have already converged to very small region in the tensor that is trivial to exhaustively search. Note that in the diagram for **KNN-C-Wine**, the graphs of BO-GP, CMA-ES, RS and BOHB overlap each other, as do those of BO-TPE and HB.

In the diagrams for **3LC-MNIST** and **VGG-CIF10**, the ends of the completion cycles for HOTC have been indicated by numbered markers. Grid search was not enabled in either problem, as it would consume too much time. In these problems with more hyperparameters, each combination of which takes longer to evaluate (neural networks take longer to train), HOTC does not outperform the other techniques. One has to wait ≈ 1500 seconds for the first suggestion of a hyperparameter combination from HOTC in **3LC-MNIST** and ≈ 2000 seconds in **VGG-CIF10** - this is because a minimum number of tensor elements must be known in order for tensor completion to be possible. However, the effectiveness of the tensor completion can be seen as the suggested combination improves over subsequent completion cycles - in both cases, the final validation loss is comparable to the best results observed. Overall, while HOTC is not outperforming its competitors in all situations, it is still able to consistently find optimal or near-optimal combinations across machine learning paradigms.

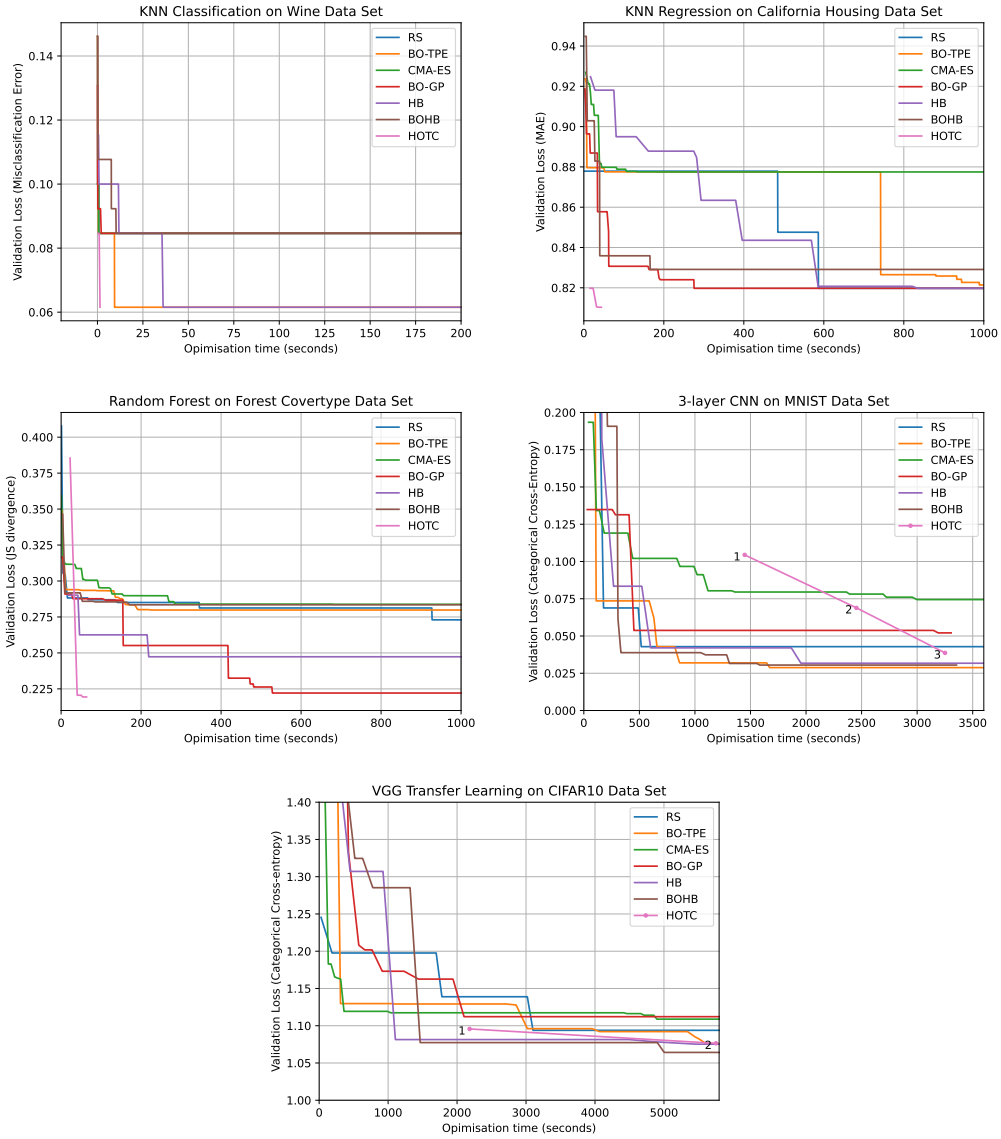

Figure 2: Graphs of validation loss against optimisation time for the different hyperparameter optimisation algorithms on the benchmark machine learning problems.

7 Conclusion

We have introduced the concept of tensor completion in the hyperparameter optimisation paradigm. Through sequential completion cycles that are able to identify the most promising subspaces, the proposed method has been shown to be highly competitive and often superior to a wide range of state-of-the-art and commonly used frameworks over a diverse set of benchmarks and algorithms. It is our hope that the presented intuitive yet efficient approach to hyperparameter optimisation will spur the interest of the machine learning community, and we envision tensor completion becoming a strong alternative to current approaches. Regarding future work, we aim to work on a procedure to automatically determine optimal values for the algorithm inputs, such as resolution of hyperparameter ranges or number of tensor completion cycles. Furthermore, we plan to investigate whether combining the current approach of focusing on optimal regions with random exploration of the space will improve the overall performance. Finally, an interesting direction we aim to pursue is the investigation of alternative tensorisation methods of the hyperparameter space.

REFERENCES

Evrim Acar, Daniel M. Dunlavy, Tamara G. Kolda, and Morten Mørup. Scalable tensor factorizations for incomplete data. *Chemometrics and Intelligent Laboratory Systems*, 106(1): 41–56, Mar 2011. ISSN 0169-7439. doi: 10.1016/j.chemolab.2010.08.004. URL http://dx.doi.org/10.1016/j.chemolab.2010.08.004.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework, arxiv1907.10902, 2019.

AutoML. Hpbandster :a distributed hyperband implementation on steroids, 2019. URL https://github.com/automl/HpBandSter.

Johann A. Bengua, Ho N. Phien, Hoang Duong Tuan, and Minh N. Do. Efficient tensor completion for color image and video recovery: Low-rank tensor train. *IEEE Transactions on Image Processing*, 26(5):2466–2479, May 2017. ISSN 1941-0042. doi: 10.1109/tip.2017.2672439. URL http://dx.doi.org/10.1109/TIP.2017.2672439.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(10):281–305, 2012. URL http://jmlr.org/papers/v13/bergstra12a.html.

Andrzej Cichocki, Danilo Mandic, Lieven De Lathauwer, Guoxu Zhou, Qibin Zhao, Cesar Caiafa, and Huy Anh Phan. Tensor decompositions for signal processing applications: From two-way to multiway component analysis. *IEEE Signal Processing Magazine*, 32(2):145–163, 2015. doi: 10.1109/MSP.2013.2297439.

Liping Deng and Mingqing Xiao. A new automatic hyperparameter recommendation approach under low-rank tensor completion framework. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–13, 2022. doi: 10.1109/TPAMI.2022.3195658.

Bradley Efron, Trevor Hastie, Iain Johnstone, and Robert Tibshirani. Least angle regression. *The Annals of Statistics*, 32(2), Apr 2004. doi: 10.1214/009053604000000067. URL https://doi.org/10.1214%2F009053604000000067.

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1437–1446. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/falkner18a.html.

Matthias Feurer and Frank Hutter. *Hyperparameter Optimization*, pp. 3–33. Springer International Publishing, Cham, 2019. ISBN 978-3-030-05318-5. doi: 10.1007/978-3-030-05318-5_1. URL https://doi.org/10.1007/978-3-030-05318-5_1.

R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936. doi: https://doi.org/10.1111/j.1469-1809.1936.tb02137.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x.

Peter I. Frazier. A tutorial on bayesian optimization, arxiv1807.02811, 2018.

Nikolaus Hansen. The cma evolution strategy: A tutorial, arxiv.1604.00772, 2016. URL https://arxiv.org/abs/1604.00772.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization, arxiv1603.06560, 2018.

Yuanyuan Liu, Fanhua Shang, Hong Cheng, James Cheng, and Hanghang Tong. *Factor Matrix Trace Norm Minimization for Low-Rank Tensor Completion*, pp. 866–874. 04 2014. ISBN 978-1-61197-344-0. doi: 10.1137/1.9781611973440.99.

Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014. URL https://github.com/fmfn/BayesianOptimization.

Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. Tensor completion algorithms in big data analytics, 2018.

Tong Yu and Hong Zhu. Hyper-parameter optimization: A review of algorithms and applications, 2020. URL https://arxiv.org/abs/2003.05689.

Anru Zhang. Cross: Efficient low-rank tensor completion. *The Annals of Statistics*, 47(2):936 – 964, 2019. doi: 10.1214/18-AOS1694. URL https://doi.org/10.1214/18-AOS1694.

A Hyperparameter Search Spaces

A.1 Tensor Completion Experiments

In these experiments, the hyperparameter search spaces are set in order to generate a complete tensor of validation loss values. In the tables, they are presented in the format of $(start,\ resolution\_interval,\ end)$ for *uniform integer* or *real* hyperparameters, or $[item1,\ item2,\ ...]$ for categorical hyperparameters.

SVM-P-iris Refer to table 3.

Hyperparameter	Search Space
Regularisation Hyperparameter	(0.1, 0.1, 3.0)
Polynomial Kernel Degree	(0.0, 0.1, 3.0)
Polynomial Kernel Scaling Factor	(0.1, 0.1, 3.0)
Polynomial Kernel Constant Term	(0.0, 0.1, 3.0)

Table 3: Search spaces for **SVM-P-iris**.

KNN-R-diab Refer to table 4.

Hyperparameter	Search Space
No. of Neighbours	(1, 1, 100)
Minkowski Norm Order	(1, 1, 100)
Weighting Heuristic	['*uniform*', '*distance*']

Table 4: Search spaces for **KNN-R-diab**.

RF-wine Refer to table 5.

Hyperparameter	Search Space
No. of Decision Trees	[1, 10, 20, 30, 40]
Maximum Tree Depth	[1, 5, 10, 15, 20]
Minimum Samples to Split Node	(2,1,10)
Data Features to Split Node	(1,1,10)
Subset/Entire Training Data to Construct Tree	[True, False]

Table 5: Search spaces for **RF-wine**.

A.2 Hyperparameter Optimisation Experiments

In these experiments, the limits of the search spaces for each hyperparameter were kept common for all of the tensor completion techniques. The search space resolution interval is only defined for HOTC so that it can construct a tensor representing the search spaces. For all other techniques, integer hyperparameters always have resolution 1 and the search spaces for real hyperparameters are continuous. The advantage of this is that it ensures these techniques have access to the entire set

of possible values, although the downside is that the set of combinations that can be tested becomes larger, especially with real hyperparameters.

In the tables, search spaces are presented either in the format of $(start,\ end)$ for *real* or *integer* hyperparameters, or $[item1,\ item2\ ...]$ for *categorical* hyperparameters. Fuethermore, if the *log* column value is "True", this means the hyperparameter value varies between $start$ and end exponentially rather than uniformly. Note that while it was stated that HOTC only supports uniformly distributed hyperparameter values, it is easy to accomodate exponentially varying values by considering the exponent as a uniformly varying hyperparameter over a constant base.

KNN-C-Wine Refer to table 7.

Hyperparameter	Search Space	HOTC Inital Resolution Interval	Log
No. of Neighbours	(1, 100)	10	False
Minkowski Norm Order	(1, 100)	10	False
Weighting Heuristic	['*uniform*', '*distance*']	N/A	N/A

Table 6: Search spaces for **KNN-C-Wine**.

KNN-R-Calh Refer to table 7.

Hyperparameter	Search Space	HOTC Inital Resolution Interval	Log
No. of Neighbours	(1, 100)	10	False
Minkowski Norm Order	(1, 100)	10	False

Table 7: Search spaces for **KNN-R-Calh**.

RF-FC Refer to table 8.

Hyperparameter	Search Space	HOTC Inital Resolution Interval	Log
No. of Decision Trees	(1, 40)	5	False
Maximum Tree Depth	(1, 20)	5	False
Minimum Samples to Split Node	(2, 11)	2	False
Data Features to Split Node	(1, 11)	2	False
Subset/Entire Training Data to Construct Tree	[True, False]	N/A	N/A

Table 8: Search spaces for **RF-FC**.

3LC-MNIST Refer to table 9.

Hyperparameter	Search Space	HOTC Inital Resolution Interval	Log
No. of Layer 1 Neurons	(5, 40)	5	False
No. of layer 2 Neurons	(5, 40)	5	False
Layer 1 Activation Function	['relu', 'tanh', 'sigmoid']	N/A	N/A
Layer 2 Activation Function	['relu', 'tanh', 'sigmoid']	N/A	N/A
Layer 1 Pooling Dimension	(1, 5)	1	False
Layer 2 Pooling Dimension	(1, 5)	1	False
Layer 1 Stride Dimension	(1, 5)	1	False
Layer 2 Pooling Dimension	(1, 5)	1	False
Optimiser	['RMSProp', 'adam', 'sgd']	N/A	N/A
Learning Rate	$(10^{-6}, 10^{-2})$	1	True
Learning Rate Decay	$(10^{-8}, 10^{-4})$	1	True

Table 9: Search spaces for **3LC-MNIST**.

VGG-CIF10 Refer to table 10. Note that dropout probability is the probability of the input to a neuron being *turned off*, not turned on.

Hyperparameter	Search Space	HOTC Inital Resolution Interval	Log
Optimiser	['RMSProp', 'adam', 'sgd']	N/A	N/A
Learning Rate	$(10^{-6}, 10^{-2})$	1	True
Learning Rate Decay	$(10^{-8}, 10^{-4})$	1	True
Epochs	(1, 5)	1	False
Batch Size	$(2^4, 2^8)$	1	True
Dropout Probability of First Fully-Connected Layer	(0.0, 0.9)	0.15	False
Dropout Probability of Second Fully-Connected Layer	(0.0, 0.9)	0.15	False

Table 10: Search spaces for **VGG-CIF10**.

B Software and Hardware details

All experiments were evaluated on Google Colab: the Google Compute Engine backend consisted of two single-core Intel Xeon CPUs of clock frequency 2.2 GHz, each accommodating two threads. The total RAM available was 12.68 GB. For experiments with **3LC-MNIST** and **VGG-CIF10**, the environment was accelerated with an NVIDIA Tesla T4 GPU. While HOTC was implemented in Python by us, open-source Python libraries were used for the other techniques: *Optuna* (Akiba et al. (2019)) version 2.10 for RS, BO-TPE, CMA-ES and HB; *Bayesianoptimisation* (Nogueira (2014)) version 1.2 for BO-GP, and *hpbandster* (AutoML (2019)) version 0.7 for BOHB.

The configuration parameters of HOTC were set for each problem according to the heuristics described in part 5.2, and are listed in table 11. For RS, BO-GP, BO-TPE and CMA-ES only the number of trials were varied according to the time for which optimisation was to be performed. The time of optimisation for each problem can be seen in the graph of figure 2. For multi-fidelity techniques: HB and BOHB, the number of trials, the minimum and maximum budget values, and the step size with which the minimum budget was increased to the maximum were set to reasonable values based on the data to achieve good performance - these values are described in table 12. Apart from this, all other configurations were left with the default values from their respective library implementations.

For HB and BOHB, the multi-fidelity budget was the fraction of the training data used to train the machine learning model in all problems except **KNN-C-Wine**. Since the size of the modified wine dataset, 130, was small compared to the maximum searched value of K i.e., 100, the budget was defined as the fraction of the total number of data features (which was 13) used in training.

Problem	Assumed Tucker Rank	Maximum Number of Completion Cycles	Minimum Number of Elements Grid Search	Minimum Resolution for Real-Valued Hyperparameters
KNN-C-Wine	[1,1,1]	5	51	N/A
KNN-R-Calh	[1,1]	5	51	N/A
RF-FC	[1,1,1,1,1]	4	51	N/A
3LC-MNIST	[1] $\times 11$	3	0	0.25
VGG-CIF10	[1] $\times 7$	2	0	0.05

Table 11: Configuration parameters of HOTC in different machine learning problems.

Problem	Minimum Budget	Maximum Budget	Step Size
KNN-C-Wine	2/13	13/13	1/13
KNN-R-Calh	0.2	1.0	0.2
RF-FC	0.2	1.0	0.2
3LC-MNIST	0.2	1.0	0.2
VGG-CIF10	0.2	1.0	0.2

Table 12: Multi-fidelity budget limits used for HB and BOHB in the different machine learning problems.