# HTF-MPR: A Heterogeneous TensorFlow Mapper Targeting Performance using Genetic Algorithms and Gradient Boosting Regressors

Ahmad Albaqsami, Maryam S. Hosseini, Nader Bagherzadeh
Department of Electrical Engineering and Computer Science, University of California, Irvine, CA, USA
Email: {aalbaqsa,mseyyedh,nader}@uci.edu

***Abstract*—TensorFlow [1] is a library developed by Google to implement Artificial Neural Networks using computational dataflow graphs. The neural network has many iterations during training. A distributed, parallel environment is ideal to speed-up learning. Parallelism requires proper mapping of devices to TensorFlow operations. We developed HTF-MPR framework for that reason. HTF-MPR utilizes a genetic algorithm approach to search for the best mapping that outperforms the default Tensorflow mapper. By using Gradient Boosting Regressors to create the fitness predictive model, the search space is expanded which increases the chances of finding a solution mapping. Our results on well-known neural network benchmarks, such as ALEXNET, MNIST softmax classifier, and VGG-16, show an overall speedup in the training stage by 1.18, 3.33, and 1.13, respectively.**

## I. Introduction

Machine Learning (ML) algorithms [2] have found a large number of applications in computer vision, data tracking, recommender systems, search engines and Artificial Intelligence (AI) in games. One notable advancement in ML algorithms is the use of Artificial Neural Networks (ANNs) [3]. ANNs are constructs that mimic how the brain works. In their most basic form, they consist of synapses and neurons, where the synapses are the weights and neurons are the functions (see Fig 1). Companies invest in improving and utilizing ANNs for different tasks [4], leading to many applications applied to ANNs, creating deep and complex ANN architectures, finding techniques and accelerating the training and the inference of ANNs [5]–[7]. A number of softwares libraries have been developed to ease the construction of ANNs for end-users. One such library is TensorFlow [8]; a computational graph and numerical models library developed by Google. The application programming interface (API) makes it possible for data scientists to work with large models and many data samples in a distributed system without prior knowledge of the hardware architecture.

The current state-of-the-art ANNs consist of hundreds of thousands of parameters, and require large data sets to train. The number of layers, features (inputs) and interconnections, result in a large number of parameters that require training, which prolongs the training process.

Training in ANNs are iterative [9]; in each iteration, the process would require a feed-forward step through the ANN, and a back-propagation that flows backwards. With each iteration, a set of data-samples (batch, or mini-batch) are fed to the ANN. This modifies the parameters (weights and biases) which reduces a given loss-function.

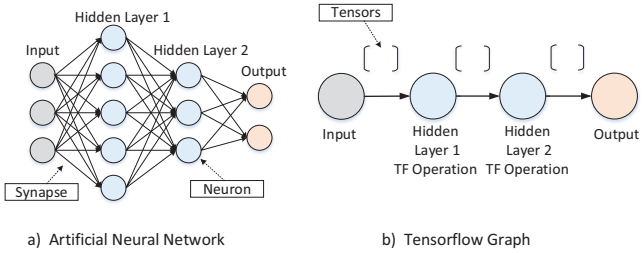


Fig. 1. Artificial Neural Network and its TensorFlow depiction

In TensorFlow, parameters, functions, and inputs are represented by computational graphs [1]. Computational graphs consist of edges and vertices in a Directed Acyclic Graph (DAG). Edges carry multi-dimensional arrays known as tensors, and vertices are the functions, known as operations, applied to tensors. A simple translation from ANN to a TensorFlow computational graph is shown in Fig 1b.

Speedup of these computational graphs is of importance. One such approach is to reduce the number of parameters in a ANN [10]. In [10], the authors compressed the ANN by pruning the number of neurons and synapses, which reduces the number of computations. However, this would slightly change the accuracy of the prediction model [10]. In regards to TensorFlow, pruning would require a lot of invasive changes such as changing tensors or using sparse tensors. We intend to keep the prediction accuracy of the ANN intact.

Another approach is to allocate resources to operations efficiently in the TensorFlow computational graph, i.e., splitting up the ANN so that different processors may work in parallel. In TensorFlow, the graph is constructed at design-time and then run [8]. After constructed and runs for the first time, no modification is allowed to the structure of the computational graph. A work around of such limitation to reconstruct the computational graph. Currently, TensorFlow carries out the mapping in a simplified way as specified in *device_factory.cc* [11]. Luckily, the TensorFlow API allows the programmer to override the default mapping in a per operation manner. Note that scheduling is taken care of by the TensorFlow engine and the API has no access to manipulate the scheduling.

978-3-9819263-0-9/DATE18/©2018 EDAA

With the above approach, there are two ways to get a better (faster runtime) mapping. One is to use static-list-based mapping algorithms such as Heterogeneous Earliest Finish Time (HEFT); a fast Heuristic greedy approach with a well proven record. To utilize HEFT, three pieces of information are required; the operation dependencies (represented by the DAG), the execution time of said operations on every device, and the communication cost between each device given each operation. Unfortunately, the later two can not be obtained; both are a limitation to the API, while communication cost would require a very large number of mappings to be tested.

With the absence of the above mentioned pieces of information, another way would be to use a meta-heuristic approach. In HTF-MPR, A genetic-algorithms-based [12] approach is used. Genetic algorithms (GA) have been used in many combinatorial problems and have provided good solutions in heterogeneous computing mapping problems [13].

In GA-based approaches, the following steps are taken: 1.*initial population* of mappings are generated. 2.The *fitness* of each mapping is obtained. 3. The breeding of new mappings according to crossovers.

steps *2* and *3* are repeated for a prescribed number of times, until a solution is found (or stop due to time constraints).

To be able to generate many mappings and obtain their fitness, while bypassing the overhead of actually running the TensorFlow computational graph of said benchmark, a predictive model of the fitness is to be used. We use an ML ensemble algorithm, called *Gradient Boosting Regressors* (GBR) [14], to construct the mapping-to-fitness predictive model. The initial population of mappings and their actual fitness are used to construct the said predictive model, and the accuracy of the model is tested using Kendall tau rank distance as a metric. The metric takes into affect the pair-wise agreement between two lists (in this case between the order of the actual fitness of the mappings versus the order of the predicted fitness of the mappings). We used k-fold cross validation method [15] to validate the accuracy of the Kendall tau rank distance of the mappings.

The rest of the paper is organized as follows. Section II covers the background. In Section III we describe in detail of the HTF-MPR Framework and the system model. In Section IV, we evaluate our approach with comparison to the default TensorFlow device mapper. Finally, the paper is concluded in V.

## II. Background

### A. TensorFlow

TensorFlow is a library for constructing machine learning algorithms [8]. One of the upsides of TensorFlow is th ease of use and seamless integration into heterogeneous systems. Models in TensorFlow are described with Directed Graphs, where the input/output to/from each TensorFlow operation is zero or more tensors(see Fig 2). The following is an example of a simple code snippet that describes a TensorFlow dataflow graph in Python (taken from [16] with slight modification): Code 1 is represented in Fig 2 (without the device mapping).

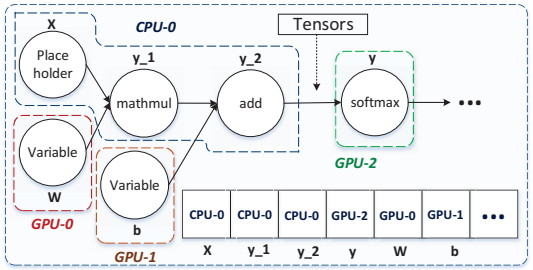


Fig. 2. Example of a model in TensorFlow, and of device-operation mapping

Code 1. Model in TensorFlow

```
import tensorflow as tf
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y_1 = tf.matmul(x, W)
y_2 = tf.add(y_1,b)
y = tf.nn.softmax(y_2)
...
```

The graph is then described but not yet constructed. To create and run the graph the following is done (see Code 2):

Code 2. Run model in Session

```
...
sess = tf.Session()
sess.run(y,feeddict=...)
...
```

In Code 2, a $Session$ is created and the operation for which an output is desired is given as the input argument to the session object's run method (in this case $y$). A single run would provide the actual *makespan* of the graph. The makespan is the time it takes to complete one *iteration* (i.e run) of the graph. TensorFlow would take care of the rest; the **Distributed Master** would take in the graph and evaluate the nodes and distributes the tasks. The **worker services** would take in requests from the master and schedule the execution of the tasks.

### B. Task Mapping

TensorFlow uses dataflow graphs for the computation, in addition the design of most ANNs is done in a DAG approach. In this paper, we assume that the dataflow graphs are DAG, and that no changes to the mapping may occur during runtime. Note that **edges** carry **tensors**, and **vertices** are the **operations**. During design-time, the API user has the option of assigning a **device** (CPU-0, GPU-0, GPU-1, etc) to each TensorFlow operation rather than TensorFlow default mapping. No changes may occur while the TensorFlow graph is running. The structure of a single mapping $M_i$ is shown in the array in Fig 2. Note that $M_i \in \boldsymbol{M}_P$ where $\boldsymbol{M}_P$ is the set of all mappings in a particular population, and $\boldsymbol{M}_P \subset \boldsymbol{M}_U$ where $\boldsymbol{M}_U$ is the set of all possible mappings within a given DAG, also described as the universal set of mappings.

### C. Fitness and Makespan

In order to evaluate the mapping's performance a metric is required. In this case the *makespan* is the metric of choice.

The makespan is the total time it takes for a single run of the TensorFlow graph, i.e. a single session run. The fitness is inversely proportional to the makespan; The higher the fitness the shorter the makespan. Our target is to find a mapping that results in a smaller makespan than the one provided by TensorFlow's default mapping. Note that the makespan is dependent on several factors; operation-device mapping,communication costs, and scheduling of operations.

Given the restrictions on the availability of communication cost and the scheduling, the makespan value can only be found by a session run.

### D. Meta-Heuristic Approach and The prediction Model

A session run is costly when considering the search space of finding a solution. An alternative would be to use a predictive model to obtain the makespan (fitness), or rather, the relative rank of a mapping in relation to other mappings. Such a model would then be used in a meta-heuristic approach to navigate the search space and find the best possible mapping. The choice of mappings used to train the fitness predictive model and used for the initial search, in the meta-heuristic approach, is of importance [17]. In addition, the *features* that are used for the training and the ML algorithm will determine the success of the predictive model. We have found that the simplest feature selection (i.e the tasks) and the values (i.e the devices) is sufficient. Feature extraction was used but did not result in a better prediction model. We thus reverted to simple features. In terms of the initial population of mappings (also used in the training) a restriction was made which will be elaborated in III-E.

## III. HTF-MPR

### A. System Model

Our framework targets TensorFlow dataflow models, which are numerical computations that use dataflow graphs [1] [8]. The Graph $T = (V, E)$ consists of vertices $V = \{\tau_i, \tau_{i+1}, ...\}$ where $\tau_x$ is a TensorFlow operation and $E = \{(\tau_i, \tau_j), (\tau_m, \tau_n), ..\}$ where $(\tau_i, \tau_j)$ is a directed edge from operation $\tau_i$ to $\tau_j$ . The directed edges in a TensorFlow graph carry tensors.

The list of devices is $D = \{d_x, d_y, ..\}$, where $d_x$ is a device (CPU-0, GPU-0, GPU-1 etc). Each mapping $M_i$ is a unique mapping of operations-to-devices, where $M_i = \{(\tau_0, d_x), (\tau_1, d_x), (\tau_2, d_y)..\}$. Every $M_i$ has a runtime (makespan) of $t_i$ and a fitness $f_i = 1/t_i$.

### B. HTF-MPR Overview

The objective of the HTF-MPR is to create a modified Python file that contains the *tf.with(..)* directive. This directive would allow the user, via API, to allocate a device to a particular operation or set of operations. In the HTF-MPR case, the Python file would have the sub-optimal device allocated to each operation. An example of an output, a modified file, of HTF-MPR:

Code 3. Addition of tf.device

```
import tensorflow as tf
with tf.device('/cpu:0'):
      x = tf.placeholder(tf.float32, [None,
          ↪ 784])
with tf.device('/gpu:0'):
      W = tf.Variable(tf.zeros([784, 10]))
with tf.device('/gpu:1'):
      b = tf.Variable(tf.zeros([10]))
with tf.device('/cpu:0'):
      y_1 = tf.matmul(x, W)
with tf.device('/cpu:0'):
      y_2 = tf.add(y_1,b)
with tf.device('/gpu:2'):
      y = tf.nn.softmax(y_2)
...
```

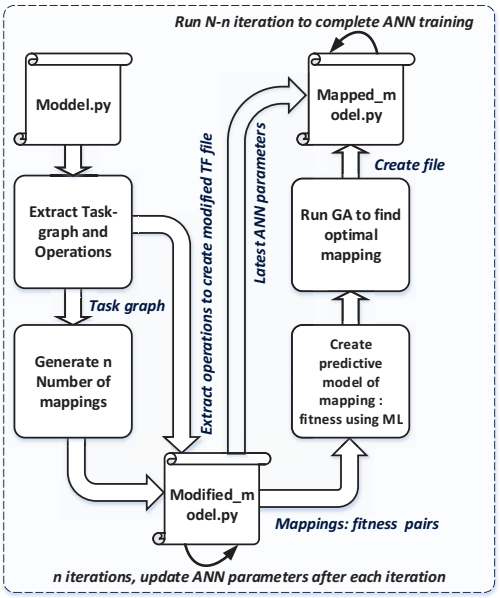


Fig. 3. HTF-MPR workflow.

Code 3 is reflected in Fig 2. An overview of HTF-MPR is shown in Fig 3. First, the TensorFlow operations from a $Model.py$ file are identified. Then, to account for all the hidden operations that are created by TensorFlow when the Task-graph is constructed, we create and run a *Session*. The directed graph is then pruned to remove hidden operations that cannot be assigned via API. This dependency graph structure will determine the initial mappings that will be generated according to certain criteria which will be discussed in section III-E. From the initial population of mappings, a predictive model of the mappings-to-fitness is created using the modified TensorFlow file. This predictive model is then used in the GA to search the for a better mappings. Finally once a mapping has been found a mapped model file is created where the file will be run for the remaining number of iterations while starting with the updated ANN parameters that were saved during the initial mappings sessions run stage.

### C. Extract Operations

First step in HTF-MPR framework is to identify the essential TensorFlow operations. Each operation is assigned a name that coincides with its variable name.

Once those operations have been identified, a single *Session* is run in order to construct the graph. Note that TensorFlow creates other operations not specified in the Python model file.

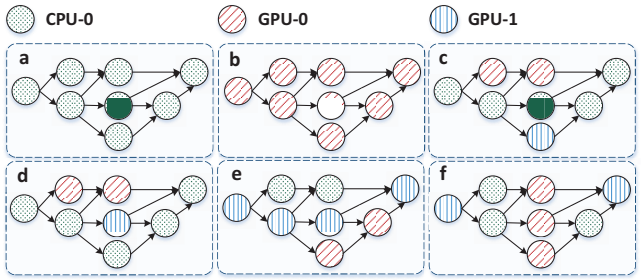


Fig. 4. Examples of some initial mappings; **a** and **b** are homogeneous (single device), **c**,and **d** are longest paths, **e** is non-longest path, and **f** is color-mapped.

### D. Extract Task-Graph

Once operations are extracted, and one of the operations (the *final* operation in the DAG) had been run on the *Session*, all the hidden operations are pruned because they cannot be mapped. Hidden operations are mapped automatically by TensorFlow in accordance to whatever the *Master* operation is mapped to.

### E. Initial Mappings Generation

The initial mappings will serve two purposes; Training the fitness predictive model and initial population in the GA stage.

The types of generated mappings are;

- **Homogeneous mapping**: single device for all operations. The number of mappings will be $N_D$.
- **Longest Path mapping**: single device on the longest path. #mappings=$N_D$!.
- **Random Homogeneous mapping**: single device to a non-longest single path.
- **Color-mapping**:whenever possible, no two connected operations should be mapped to the same device. #mappings=$N_D$!.

An illustrative example of the mappings types is shown in Fig 4. The initial mapping types provide variety, useful in training the fitness predictive model, and are good initial starting points for the GA. One of the mappings is the default mapping of heterogeneous (GPU support) TensorFlow (all GPU-0). The other mapping is the default mapping of for non-GPU supported TensorFlow (all CPU-0).

### F. Prediction Model for Fitness

We use a fitness predictive model to obtain the proper ranking as compared to other mappings, rather than obtaining a highly accurate makespan (and therefore fitness). To evaluate the accuracy of the model (which is done off-line, and is used as a justification for our choice of GBR) we use the Kendall tau rank distance as a metric and the k-fold cross validation method [15] .

*1) Kendall tau rank distance:* The Kendall tau rank distance is calculated by obtaining the actual fitness $f_i$ and the predicted fitness $\hat{f}_i$ of $M_i$ for all initial $\boldsymbol{M}_P$. if the size of $\boldsymbol{M}_P$ is $n$, then there are $n(n-1)/2$ ranking comparisons. The Kendall number between $f_i$ and $f_j$ is:

$$k(f_i, \hat{f}_i, f_j, \hat{f}_j) = \begin{cases} 1, & \text{if } f_i < f_j \text{ and} \hat{f}_i > \hat{f}_j. \\ 1, & \text{if } f_i > f_j \text{ and} \hat{f}_i < \hat{f}_j. \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

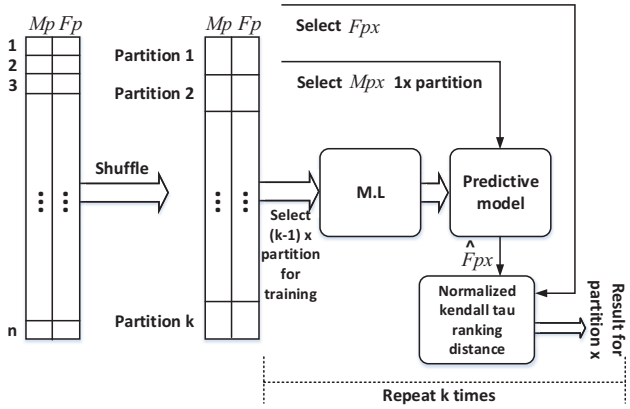


Fig. 5. Cross-Validation using k-fold.

where $i \neq j$. The normalized Kendall tau ranking distance;

$$K_{norm}(F_P, \hat{F}_P) = \sum_i \sum_j \frac{2 \cdot k(f_i, \hat{f}_i, f_j, \hat{f}_j)}{n(n-1)} \quad (2)$$

Where $F_P$ and $\hat{F}_P$ are the actual and predicted fitnesses of the mappings in the population, respectively.

*2) K-fold Cross Validation:* The mappings $\boldsymbol{M}_P$ and actual fitnesses $F_P$ are split into a number of partitions (i.e. K-partitions). The samples are randomly partitioned, where *cross-validation* is done $K$ times. Each partition is set as the validation set and the others are used in the training (see Fig 5).

Given the training set $\{((M_1, f_1), (M_2, f_2), ..., (M_n, f_n)\}$ where $M_i$ is the mapping and $f_i$ is the actual fitness, a predictive model $\mathbf{F}(M)$ is to be found which minimizes the loss function $\mathbf{L}(f, \mathbf{F}(M))$. Note $\mathbf{F}(M_1) = \hat{f}_1$. The loss function in use is the least square, i.e. $\mathbf{L}(f_i, \mathbf{F}(M_i)) = (f_i - \mathbf{F}(M_i))^2$ . After investigating several machine learning algorithms and using different feature extraction methods, we settled on the Gradient Boosting Regression (GBR) [14] algorithm. GBR consists of weak learners, in the form of decision trees, that are added together (ensemble) to make a stronger prediction model. This is done by iterative means. At each iteration, a weak learner is introduced that compensates for the shortcomings of the previous iteration's weak learner. The overall prediction model is updated by the gradient descent method. The residual, also known as the negative gradient $g(M_i)$ is calculated as:

$$-g(M_i) = \frac{\delta L(f_i, F(M_i))}{\delta F(M_i)} \quad (3)$$

At each iteration, $-g(M_i)$ is calculated and a regression tree $h_j$ is fit to the $-g(M_i)$ updating the overall predictive model:

$$F(M) \leftarrow F(M) + h \quad (4)$$

### G. Search via Genetic Algorithm

When searching, the following factors are taken into consideration:The search space, the method search, the crossover operation, and the fitness function.

Selection of mappings is proportional to the fitness; the higher the fitness the higher the probability of selecting the mapping for breeding.

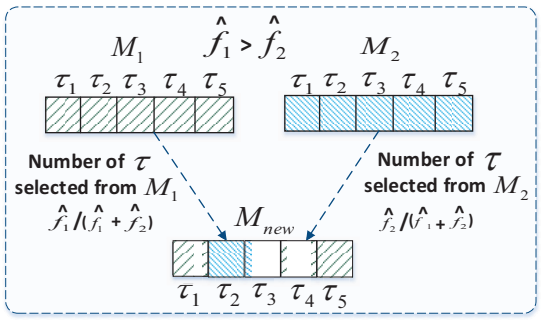


Fig. 6. Breeding using a stochastic method. Newly generated mapping takes more from the fitter mapping parent

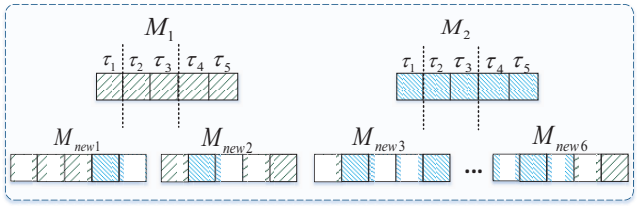


Fig. 7. Breeding using Crossover points. In this case 2 crossover points resulting in 6 new mappings

Once two mappings are selected for breeding, crossover takes place. We implemented two types of breeding. The first looks at each individual $\tau$ mapped and stochastically decides which parents $\tau$ is chosen(see Fig 6). The second uses crossover points where the number (between 1 and 3) and position of the crossover points are determined randomly within each mappings pair. The number of crossover points determines the number of new mappings generated from the parent pair (i.e. $(N_C+1)^2-2$ where $N_C$ is the number of crossover points). See Fig 7. both breeding methods are used so that one provides randomness during exploration (Fig 6) while the other provides improved performance (Fig 7).

The top $x$ mappings (corresponding to the top $\hat{f}$) are then run on TensorFlow to get the actual fitness $f$. This is done to compensate for the error of the fitness predictive model. Once the top mapping corresponding to the highest $f$ is found, the ANN training continues with said optimal mapping.

## IV. Experimental Results

### A. Experimental setup

To test our framework, we used a system that consists of a multi-core CPU (Intel(R) Core(TM) i7-7700 CPU 3.60Ghz), and 2x GPUs (Nvidia GeForce GTX 1050 Ti). The TensorFlow version used is 1.1 with GPU capability (using Nvidia CUDA 8.0 and cuDNN v5), and Python version is 2.7.12. For constructing the predictive model of the makespan, we used the Python-based library scikit-learn version 0.19.0. The following benchmarks were tested: ALEXNET [18] and VGG-16 [19], are convolutional neural network used to classify images from ImageNet [20]. MNIST softmax classifier [16], a very simple image classifier used for characters.

The benchmarks are run on TensorFlow without explicit mapping where the total execution time of the benchmark is

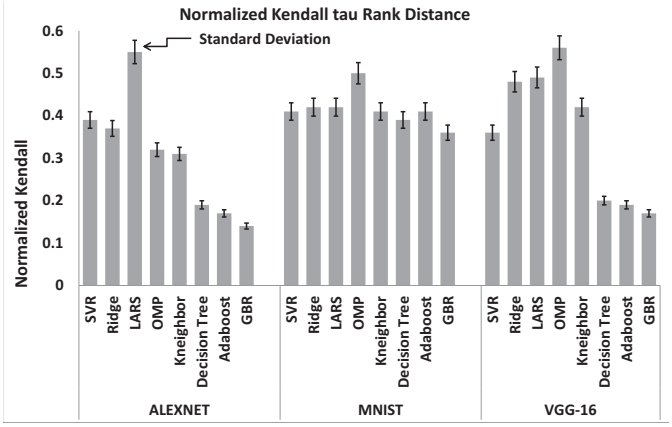


Fig. 8. Predictive Model performance using k-fold (k=5) and different ML algorithms. The chart shows the average from 5 runs and includes the standard deviation of the 5 runs. **SVR**: Support Vector Regression, **Ridge**: Ridge Regression,**LARS**: Least Angle Regression,**OMP**:Orthogonal Matching Pursuit,**Kneighbor**:Regression-based on k-nearest neighbors.

observed. The same benchmarks are then run through HTF-MPR. Bellow is a summary of the benchmarks. Note that the learning process is an iterative process. Thus, the DAG is run several times.

| Benchmark | Mapped Operations | Total Operations | Iterations |
|---|---|---|---|
| ALEXNET | 55 | 295 | 12800 |
| MNIST softmax | 10 | 99 | 60000 |
| VGG-16 | 69 | 376 | 12800 |

The majority of operations are not dealt with directly in HTF-MPR; this is because these operations are generated by TensorFlow, thus, the user may not explicitly assign a mapping via a *tf.device*. With these hidden operations, TensorFlow handles the mapping via its default mechanism, which is to assign these operation to the device of their master operation.

### B. Predictive Model Analysis Results

To validate the effectiveness of our predictive model, we used k-fold cross validation (see Fig 5) across a number of ML algorithms. The number of mappings used is different for each benchmark while the value of $k=5$ is used for all benchmarks:

| benchmark | $\mathbf{M}_p$ Size | M Size | Training Size | Testing Size |
|---|---|---|---|---|
| ALEXNET | 105 | 55 | 84 | 21 |
| MNIST softmax | 135 | 10 | 108 | 27 |
| VGG-16 | 105 | 69 | 84 | 21 |

The average results are shown in Fig 8. GBR outperforms all other ML algorithms across the board. In MNIST benchmark, GBR error is considerably higher than in the other benchmarks. So, for the GA stage, we took the top 50 $\hat{f}$ mappings to be run while in the other two benchmarks we only took the top 10 $\hat{f}$ mappings.

### C. Results and Discussion

Given the small overhead of running HTF-MPR, we were still able to accomplish an average speedup of **1.18** with

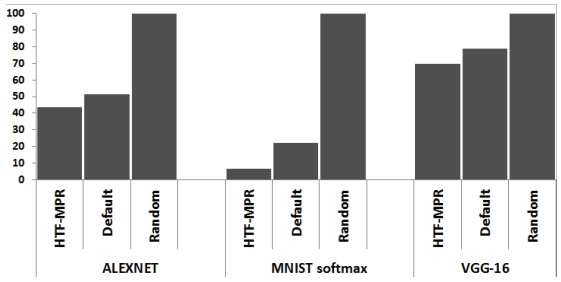


Fig. 9. Relative training time

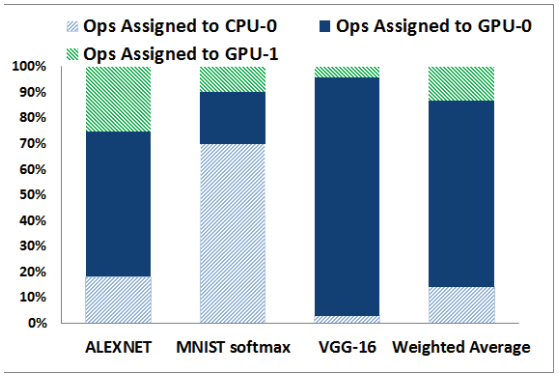


Fig. 10. Device distribution per benchmark. Weighted Average indicates all operations across all benchmarks.

ALEXNET, **3.33** with MNIST softmax classifier, and **1.13** with VGG-16. The overhead of identifying the operations, pruning the DAG, generating the initial mappings, performing ML to obtain the fitness predictive model, searching using GA, and construction of the TF graphs for the various mappings (to obtain the actual fitness) is less than **3%** for ALEXNET, **10%** for MNIST softmax, **2%** for VGG-16. The increased overhead for MNIST softmax is due to the increase in TF graph constructions (135+50+1) in addition to the GAs stage of generating mappings (5400). The choice for increasing is due to the performance of the fitness predictive model with MNIST softmax. The speed up was higher due to the fact that the search space is much smaller (size of $\mathbf{M}_U$ for MNIST softmax is $N_D^{Nv} = 3^{10}$). As a side, we used brute force to find the percentage of mappings in MNIST softmax that outperform the TF default mapping (all GPU-0). 13% of all mappings are better than the TF default. HTF-MPR did not favor GPU for every operation as can be seen by the device distribution of the mappings (see Fig 10). With such mapping, the performance was improved (see Fig 9).

## V. Conclusions

In this paper, we presented our HTF-MPR framework to optimize the mapping of devices to TensorFlow operations. The HTF-MPR uses a genetic algorithm approach to search the mappings space, utilizing a fitness prediction model to evaluate each searched mapping. The fitness prediction model is trained by using an initial population of mappings that are generated in a directed manner. Compared to the default TensorFlow mapper, our results show an overall speedup in the benchmarks, where ALEXNET, MNIST softmax classifier, and VGG-16 show a speedup of **1.18**, **3.33**, and **1.13** respectively.

## References


[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, and et. al, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2015. [Online]. Available: http://download.tensorflow.org/paper/whitepaper2015.pdf

[2] T. Mitchell, *Machine Learning*. McGraw-Hill Education, 1997.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 160–167. [Online]. Available: http://doi.acm.org/10.1145/1390156.1390177

[5] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[7] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, and et. al, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct. 1986.

[10] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[11] T. T. Authors, "tensorflow device factory," https://github.com/tensorflow/tensorflow/blob/master/tensorflow, 2017.

[12] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[13] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *J. Parallel Distrib. Comput.*, vol. 47, no. 1, pp. 8–22, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1997.1392

[14] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics and Data Analysis*, vol. 38, no. 4, pp. 367 – 378, 2002, nonlinear Methods and Data Mining. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167947301000652

[15] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143. [Online]. Available: http://dl.acm.org/citation.cfm?id=1643031.1643047

[16] T. T. Authors, "mnist classifier using softmax in tensorflow," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/mnist/, 2017.

[17] P. A. Diaz-Gomez and D. F. Hougen, "Initial population for genetic algorithms: A metric approach." in *GEM*, 2007, pp. 43–49.

[18] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, and et. al, "Imagenet large scale visual recognition challenge," *arXiv preprint arXiv:1409.0575*, 2014.