2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)

Deep Learning Evolutionary Optimization for Regression of Rotorcraft Vibrational Spectra

Daniel Martinez[1], Wesley Brewer[2], Gregory Behm[2], Andrew Strelzoff[1], Andrew Wilson[3], Daniel Wade[3]

***Abstract*—A method for Deep Neural Network (DNN) hyperparameter search using evolutionary optimization is proposed for nonlinear high-dimensional multivariate regression problems. Deep networks often lead to extensive hyperparameter searches which can become an ambiguous process due to network complexity. Therefore, we propose a user-friendly method that integrates Dakota optimization library, TensorFlow, and Galaxy HPC workflow management tool to deploy massively parallel function evaluations in a Genetic Algorithm (GA). Deep Learning Evolutionary Optimization (DLEO) is the current GA implementation being presented. Compared with random generated and hand-tuned models, DLEO proved to be significantly faster and better searching for optimal architecture hyperparameter configurations. Implementing DLEO allowed us to find models with higher validation accuracies at lower computational costs in less than 72 hours, as compared with weeks of random and manual search. Moreover, DLEO parallel coordinate plots provided valuable insights about network architecture designs and their regression capabilities.**

Keywords—hyperparameter tuning, high performance computing (HPC), genetic algorithms, neural architecture search (NAS)

I. INTRODUCTION

Deep learning is revolutionizing machine learning and data science by implementing multi-layer processing networks that outperform traditional statistical models. Classical methods such as linear or logistic regression do not work well with high dimensional data such as speech and images, which has led researchers to explore deep, multi-layer architectures [1]. More specifically, deep architectures are capable of learning representations of data with multiple levels of abstraction and are well suited for mapping high dimensional data [2, 3].

State of the art models using deep implementations of fully connected and convolutional networks have significantly improved accuracy in many regression-based methods [4]. Recent success of highly accurate models has motivated research on deep architectures and configurations for reproducibility purposes. However, the inability of any one network to generalize for other datasets presents a great challenge for users to overcome [5]. It has been observed that, due to the stochastic nature in the optimization, small changes in architecture network can produce noticeably different results [6].

Many users explore DNN architecture hyperparameters randomly by searching in a trial-and-error process to determine optimal network configurations. Bergstra and Bengio [7] demonstrated that random searching is more effective than grid searches and with enough sampling, sub-optimal architectures can be found. However, pushing the model optimization further to higher accuracies requires user interaction and manual tuning. We believe a random search is effective and appropriate as a starting point, however it does not provide enough insight about the architecture design or paths to produce better models.

Optimization algorithms, such as Evolutionary algorithms and Bayesian search, intend to provide a systematic methodology of searching the hyperparameter space by iteratively generating and evaluating models. Many of these optimization algorithms are capable of mapping hyperparameter configuration and an objective function with the purpose of generating optimized models. This paper will discuss the implementation and performance of a Genetic Algorithm designed for DNN hyperparameter optimization in the application of multivariate regression for spectrum reconstruction. DLEO is the presented method developed for the DoD and Army machine learning model generation workflows, so data scientist can accelerate their design process. DLEO is based on three major software packages: (1) Galaxy Simulation Builder, (2) Dakota optimization toolkit, and (3) TensorFlow deep learning framework, which will be further discussed.

Our objectives in this research are three-fold: (1) develop an HPC-based, user-friendly methodology of systematically computing optimal hyperparameters, (2) use the method to find candidate networks that achieve a desired accuracy, (3) use the method to develop principles for designing optimal architectures and correlations with data dimensionality and network size.

The United States Army Aviation Engineering Directorate (AED) is currently employing machine learning-based diagnostics to improve the capability of Health and Usage Monitoring Systems (HUMS) installed in rotorcraft [8]. HUMS compute Condition Indicators (CI) which are features designed to quantify rotorcraft health state from flight operational data collected from on-board sensors. Computing

[1] U.S. Army Engineer Research and Development Center (ERDC)
[2] HPCMP PETTT/Engility Corporation
[3] U.S. Army Aviation and Missile Research Development and Engineering Center (AMRDEC)

978-1-7281-0180-4/18/$31.00 ©2018 IEEE
DOI 10.1109/MLHPC.2018.000-3

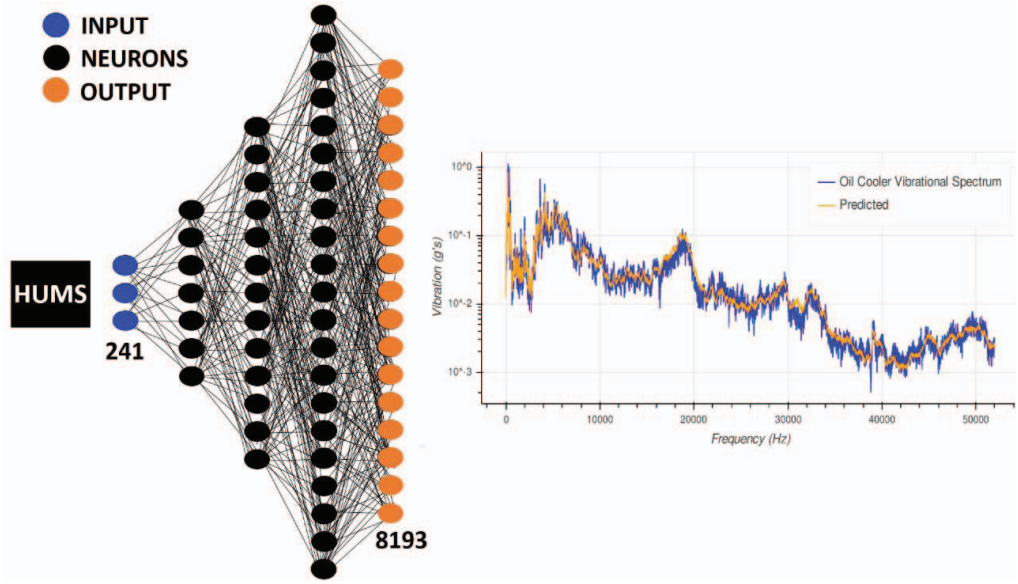

Fig. 1 Rotorcraft Oil Cooler Vibrational Spectrum Reconstruction.

CIs on-board during data acquisition is necessary to reduce data storage and transfer overhead. Unfortunately, most rotorcraft do *not* store the spectra used to compute standard HUMS on-board CIs. However, a training dataset that saves *both* spectra and CIs was provided to us for machine learning research.

Currently, to receive a life extension for any rotorcraft sub-system, vibrational spectra must be submitted to AED for post-processing. Otherwise, systems are maintained under fixed scheduled programs rather than robust continuous monitoring programs known as condition-based maintenance (CBM). We developed models capable of mapping existing CIs to helicopter oil cooler raw vibrational spectra using deep neural network regression. Fig. 1 shows an overall diagram of the deep neural network model to infer frequency spectra of the oil cooler. The model is first trained on data containing *both* CIs and raw frequency spectra. Given unseen CIs, the trained model is then able to infer the frequency spectrum of the oil cooler, and thereby infer the quality of the internal bearing faults [9].

Utilizing High-Performance Computing (HPC) was instrumental in the process of developing a neural network for building a condition-based maintenance program for a number of reasons:

1. **Big data.** The entire dataset was on the order of terabytes in size, which required parallel file systems with high bandwidth (e.g. 400 GB/s for Cray XC50).
2. **Training speed**. Data-based parallelism techniques such as Uber's Horovod (https://github.com/uber/horovod) can be utilized for the purpose of speeding up training.
3. **Model size requirements**. When a neural network model becomes too large to fit on a single GPU, model-based parallelism techniques may be used to parallelize a neural network model across multiple GPUs.
4. **Hyperparameter search**. Using hyperparameter search techniques, such as Genetic Algorithms (GA), generally require evaluating many candidate models in parallel. In GA, this corresponds to the population size of the problem.
5. **Ensemble averaging**. It is common to have multiple models for inference, and create an ensemble of the results. This may require evaluating a number of models in parallel.

To this end, we utilized both Cray XC50 and IBM Power 8 supercomputers to train our models. We primarily used the HPCs for massive, embarrassingly-parallel hyperparameter searches. We present a background of previous work, then our methodology, followed by results and discussion.

Fig. 1. Rotorcraft Subsystem's Vibrational Spectrum Reconstruction. Deep neural networks are capable of performing advanced nonlinear regression with high accuracy.

II. Background

A. *Predictive and Condition-based Maintenance*

There are a number of papers that address using machine and deep learning techniques for predictive and condition-based maintenance [10 - 16]. This area of intelligent predictive maintenance is becoming of paramount importance as the dawn of Industry 4.0 arises [17, 18].

Wade and Wilson [19] used machine learning to compute diagnostic classifiers for airworthiness. In a related but different paper, Wilson and Wade [20] developed a linear regression model to estimate a vibrational frequency spectrum from approximately 500 conditional indicators. Wade et al. [20] extended this effort to unsupervised learning for both engine output transmissions and turboshaft engines.

Our work extends the work of Wade et al. [8, 19, 20] by extending the machine learning model to a deep learning model, which is able to reconstruct the vibrational spectra on an oil cooler with better accuracy and precision than previous models. Specifically, in our paper we focus on hyperparameter tuning to design an optimal neural network architecture.

B. *Deep Regression*

While there has been little research published on the area of deep regression, Lathuilière et al. [6] provide the most comprehensive analysis to date. In their case, they typically used a deep convolutional neural network and added one or two fully connected layers to the end of the network in order to do regression. They investigate a number of hyperparameters such as optimizer, batch size, dropout ratio on well-known architectures such as VGG and ResNet-50.

Our work differs in that we are primarily interested in how to design and optimize network architectures from scratch. In this scenario, parameters such as *number of layers* and *units per layer* (or filter and kernel size for CNNs) are the most relevant, while parameters such as optimizer and batch size are not relevant to the architecture.

C. *Hyperparameter Tuning Methods*

Designing any neural network architecture requires some level of hyperparameter tuning. Choosing which hyperparameters to tune depend on the type of network. For a network of fully connected layers (FCL)[1], hyperparameters may include number of layers, number of units per layer, and dropout rate. For convolutional neural networks (CNN), hyperparameters may include the number of layers, the number of filters and kernel size. For Long-Short Term Memory (LSTM) networks, parameters may include the number of layers, and the number of LSTM units per layer. There are additional hyperparameters that are not related to network architecture but nevertheless affect the results, such as batch size or learning rate.

Young et al. [5] proposed a method called Multi-node Evolutionary Neural Networks for Deep Learning (MENNDL) which uses Genetic Algorithms to optimize the kernel size and the number of filters in convolutional neural networks using a Caffe-based deep learning framework.

Our work is similar in nature to MENNDL, but uses a different framework, Galaxy-Dakota-TensorFlow, and also is primarily focused on fully connected layers, as opposed to convolutional neural networks (which we also studied). Fig. 2 shows both fully connected and convolutional neural network architectures that were used in this study.

D. *Neuroevolution and Neural Architecture Search*

In the past couple years, there have been a number of papers published on methods to automatically generate neural networks.

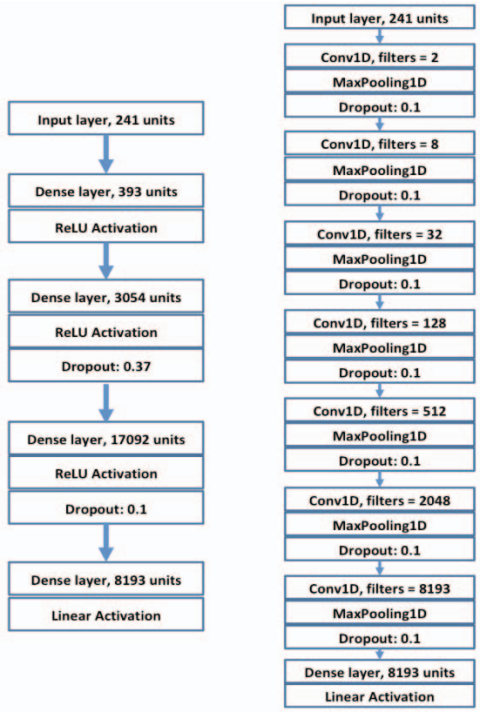

Fig. 2. Fully-connected layers (FCL) architecture on left vs. Convolutional Neural Network (CNN) architecture on right.

The Google Brain team has researched using both Genetic Algorithms and Reinforcement Learning to automatically generate deep neural network architectures. In their GA research, Real et al. [21] were able to demonstrate they could generate fully convolutional neural networks from scratch which could achieve 94.6% accuracy, which bested the majority of hand-generated models. Moreover, Zoph and Le [22] use the technique of reinforcement learning to generate recurrent neural networks (RNN) that rival human-generated networks. This latter approach has become the basis of Google's Cloud AutoML service (https://cloud.google.com/automl).

Furthermore, Such et al. [23] from Uber AI Labs recently showed that Genetic Algorithms work well at scale for reinforcement learning problems.

III. METHODS

A. *Software Stack*

As shown in Fig. 3, the software stack that was used for this project primarily consists of:

Galaxy[2] is an HPC simulation builder and management platform developed by Stellar Science (stellarscience.com)

[1] We prefer to use FCL instead of "fully-connected network" (FCN) due to the ambiguity of FCN, which also can mean Fully-Convolutional Network.

[2] Note: there is another open source Galaxy workflow management tool that is used primarily in the Life Sciences, which we mention to prevent confusion. The Galaxy that we use is distinctly different than the open source software with the same name, and is only available through the DoD HPCMP.

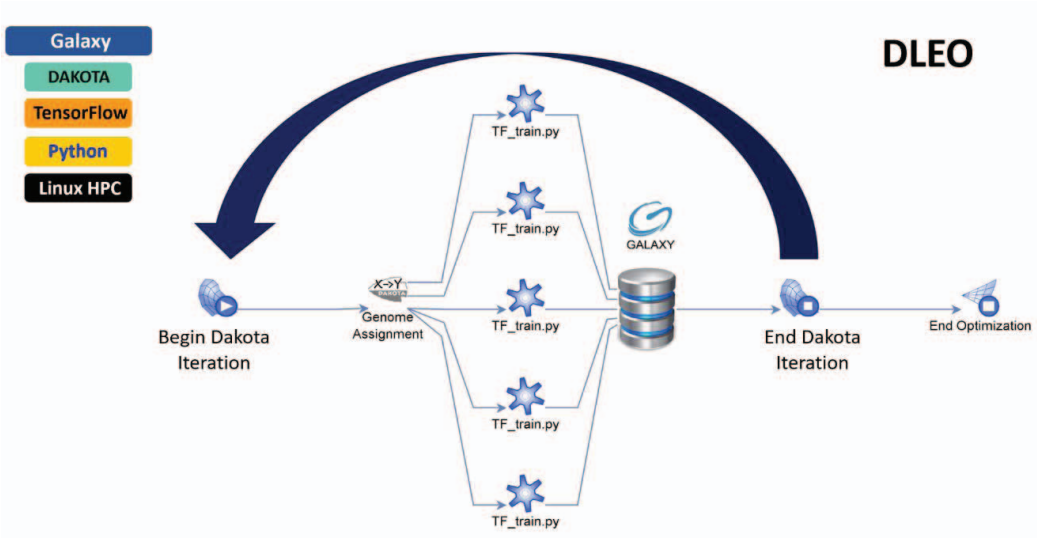

Fig. 3. Deep Learning Evolutionary Optimization (DLEO) workflow. Designed for DoD machine learning applications, DLEO is used to design optimal DNN models while generating valuable information in the process.

specifically for the DoD. Galaxy was used to setup a workflow for the hyperparameter optimization by interfacing directly

with Dakota, and also automates generating all necessary directory and files including the creation and submission of PBS job scripts.

Dakota (Design Analysis Kit for Optimization and Terascale Applications) is an open source software developed by Sandia National Laboratory (dakota.sandia.gov) for optimization and uncertainty quantification [24]. Dakota was used in this project to perform the Genetic Algorithm optimization. Dakota includes a number of other optimization algorithms that may be used as well, such as Bayesian search.

TensorFlow (tensorflow.org) is an open source deep learning framework by Google that runs both on CPUs, GPUs, and TPUs [25]. Our models also utilize Keras (keras.io), a high-level API which provides a relatively easy way to specify neural network architectures. Our version of TensorFlow is 1.8.0, which we use within the Anaconda Data Science platform (anaconda.com) with Python 3.6.5 on SLES 12.

B. *Implementation*

We implemented the software stack on a Cray XC50 platform. The XC50 has 32 GPU compute nodes, each containing one Intel Xeon E5-2699v4 Broadwell processor and one NVIDIA Tesla P100 GPU with 16GB of memory. Function evaluations, i.e. training and evaluating DNN inference accuracy, are scheduled as parallel asynchronous jobs for each generation. However, to define the next generation of candidates the algorithm has to synchronize, which is basically waiting until all candidates in the current generation are evaluated in order to assess fitness and perform genetic operators. As shown in Fig. 3, the genetic algorithm is accessed using Dakota and managed by the Galaxy Server. The algorithm runs on a master node in the Galaxy Server and communicates with slave nodes to keep track of population fitness and genome designs. As shown in Fig. 4, Galaxy has a web interface for simulation monitoring and post-processing that is useful for managing massively parallel optimization workflows.

The single objective genetic algorithm is based on John Eddy Genetic Algorithms (JEGA) library [26] that is provided in Dakota. The FCL architecture to be optimized consists of 241 inputs, three hidden layers and an 8,193 outputs layer. The hyperparameters to be optimized are: (1) the *number of neurons* in the three hidden layers and (2) the *dropout rates* in the three hidden layers, a total of six design variables. Design constraints for the number of units per layer is based on available memory since fully connected networks utilize a great amount of memory. We have developed a methodology to calculate the memory requirements in order to constrain hyperparameter bound values in Dakota input files and ensure the simulation does not die because an out of memory (OOM) problem. The lower and upper bounds for number of neurons lie between 241 and 18,000, while dropout values are allowed to vary between 0 and 1.

Datasets consist of over 100,000 samples of CIs paired with oil cooler vibrational spectra, features and labels respectively. The machine learning models are designed to perform deep regression taking a vector of 241 features (CIs) as input to reconstruct a vibrational spectrum with 8,193 frequency bins. Implementing an automated method for hyperparameter search was crucial to develop optimal deep neural networks architectures for specific HUMS configurations in rotorcraft. Fig. 2 shows both the FCL and CNN basic architectures that were used in this study. For this study, we primarily focus on FCL architectures, but hope to continue further research on CNN architectures in the future.

Supervised training is performed using TensorFlow implementation of stochastic gradient descent Adam optimizer with default values for learning rate (0.001) and momentum. Training is configured to run for 100 epochs and then evaluating on 8,000 validation samples.

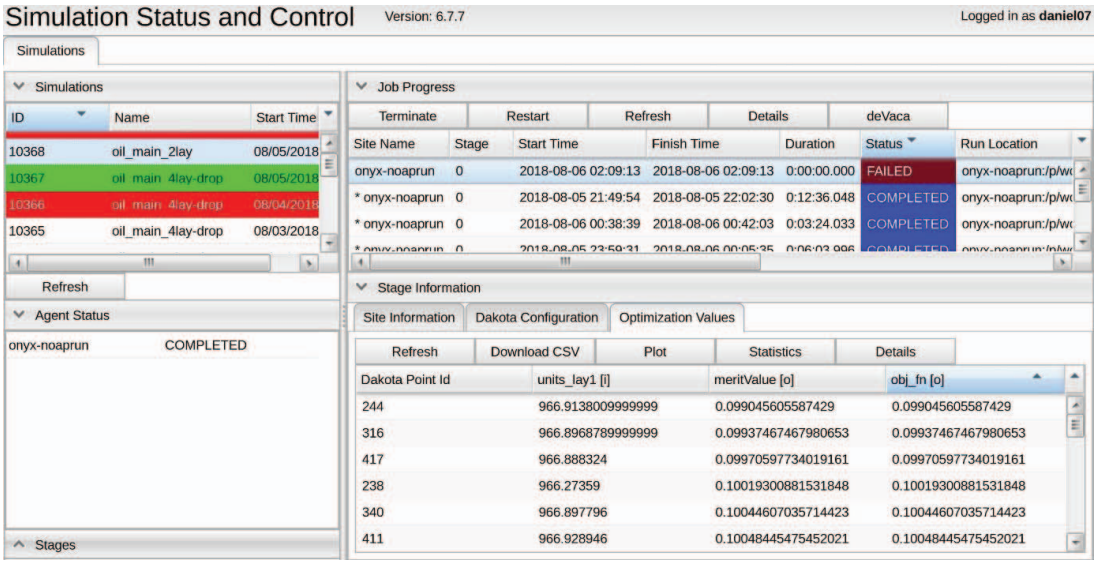

Fig 4. Galaxy Simulation Status and Control Interface. The interface allows to monitor and visualize results of previous and currently running simulations (top left corner). Post processing capabilities such as filtering by objective function (obj_fn [o]), plotting and statistics are convenient for further data analysis.

The choice of 100 epochs represents a happy medium between transitory results at the beginning of training, and overfitting by running too many epochs. For the loss function, we used the mean squared error (MSE) between predicted spectrum and true spectrum.

The performance of evolutionary algorithms is greatly influenced by the choice of selection, crossover and mutation parameters. The optimal configuration found for the DLEO genetic algorithm is presented in Table 1. It is important to understand these parameters in a particular context in order to achieve convergence and find desired global optima. For example, for this particular regression-based application we have observed thousands of hyperparameter combinations that yield high validation accuracies in a four-layer architecture, however we want to focus on the very best candidates and even see if there is one combination that significantly outperforms others. Through training multiple architectures, it is possible to observe patterns and establish correlations between architecture and performance.

C. Understanding GPU Memory Allocation Constraints

Because Genetic Algorithms generate random initial deep neural networks, in many cases we experienced the GA generating networks that exhausted the memory of the GPU. To address this issue, we wanted to develop a predictive tool which could provide a constraint on the size of network that GA could create. Therefore, we performed a study by randomly generating models to create an understanding and constraint for memory requirements. Currently the constraint is limited by our GPUs, which are NVIDIA P100s with 16GB of memory.

The memory allocation within TensorFlow is nontrivial. For example, these computational graphs request 695 GPU allocations using the best-fit with coalescing (BFC) algorithm. The BFC algorithm is based on Doug Lea's Malloc algorithm also known as *dlmalloc* [27].

The memory allocation in TensorFlow can be recorded in a trace allowing statistics to be generated on the model's memory requirements. We profiled 9,793 randomly generated models, where each layer can have up to 500,000 neurons and below 600 million trainable parameters. The idea is to reduce the hyperparameter space to only test models that can fit in the GPU's available memory.

TABLE 1. GENETIC ALGORITHM PARAMETERS

Parameter	**Value**
No. Design Variables	6
Fitness Type	Objective Function (Validation Loss)
Replacement Type	Elitist
Convergence Type	Best fitness tracker
Percent Change	0.01
Population Size	100
Number of Generations	20
Crossover Type	Multi Point Binary
Crossover Rate	0.6
Mutation Type	Bit Random
Mutation Rate	0.06

The GPU memory allocated is approximately equal to the number of trainable parameters:

$$M \cong \ w \sum_{i=0}^{N-1} L_i L_{i+1} + L_{i+1} \pm u$$

where M is the total memory allocated in bytes that is required to complete one epoch of training on 100,000 samples with a batch size of 1,024, L_i is the number of units in the ith layer, w is number of bytes per unit (float32, float64, etc.), and u is the uncertainty.

In Fig. 5a, the uncertainty increases proportionally with the total number of parameters; therefore, in order to build models with the most trainable parameters there needed to be a way to predict OOM models prior to running. We started by training a fully-connected regression model in order to predict model memory requirements; however, in Fig. 5b only 710 out of 3,487 (20.4%) OOM models were correctly identified. Hence, a fully-connected binary classifier was needed to increase the OOM accuracy. This allowed DLEO to select models with over 556 million parameters and avoid OOM models that can occur as early as 400 million parameters shown in Fig. 5b and Fig. 5c. As shown in Fig. 5d the binary classifier performs significantly better than the regression model with an area under the ROC curve of 99.85%.

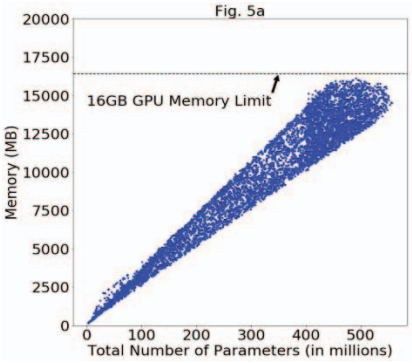

Fig. 5a. GPU memory allocated from TensorFlow's trace metadata in megabytes (MB) as a function of total number of parameters. The 6,306 blue points are the models that were successfully able to complete one epoch of training on one NVIDIA P100 GPU with 16GB of memory.

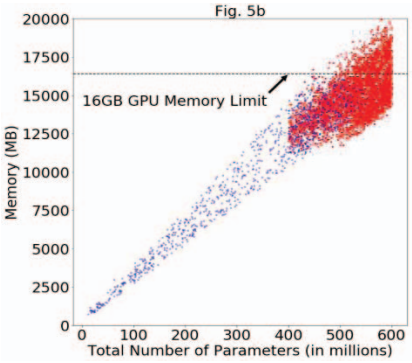

Fig. 5b. Shows a regression model that predicts TensorFlow's GPU memory allocation for 946 unseen trace data from Fig. 5a (blue), and 3,487 OOM models (red) where no trace data can be captured. Only 710 (20.4%) of the OOM models were correctly identified to allocate more than the 16GB limit.

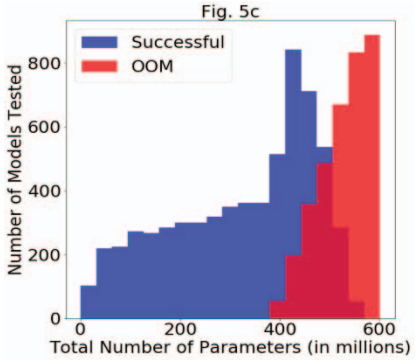

Fig. 5c. Shows 9,793 models containing the 6,306 models from Fig. 5a that can run in blue and the 3,487 OOM models from Fig. 5b in red as a function of the total number of parameters.

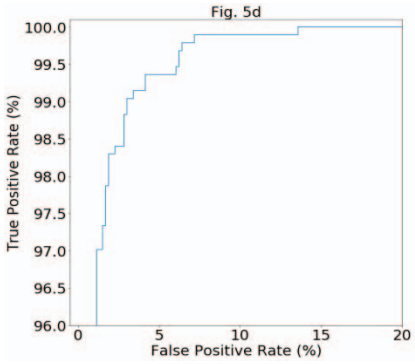

Fig. 5d. Shows a ROC curve for a binary classifier model that predicts TensorFlow's GPU OOM models specific to a NVIDIA P100 GPU with 16GB of memory. It allows for a conservative threshold to be set allowing all OOM models to be detected.

IV. Results & Discussions

The most important step for a robust genetic algorithm optimization is to initialize the first population with random genomes, and with a well determined sampling size. The size of the initial population is crucial since a few candidates will not be able to map the entire hyperparameter design space. It is necessary to initialize a significant number of candidates so that the genetic diversity is appropriate to map the wide hyperparameter space found in deep networks. If the initialization is appropriate, the algorithm will probably find a global minimum in a few iterations, otherwise one of many local minima will probably be identified as the optimal solution.

After the initial randomly generated population is evaluated, the algorithm will use genetic operators to create the following generations. Fitness is defined by the DNN validation accuracy after training for 100 epochs with a fixed batch size of 1,024 samples. The algorithm is configured to track elitist fitness in the population in order to perform crossover between them and create new individuals with potentially higher probability of great fitness. Crossover is used to stochastically combine genetic information of two candidates. The multi-point binary crossover option performs a bit-switching crossover at N randomly chosen crossover points in the binary encoded genome of two designs. The crossover probability rate establishes that not all newly generated candidates are to be

created by genetic recombination of two elitist parents. For this case, the crossover probability p_c is set to 60% since it has been observed that higher crossover tends to exploit an identified local optimum, significantly driving solutions towards it. Algorithm convergence is determined by the improvement of the best fitness value in a generation compared to the previous one. We configured the algorithm to stop if there is less than 1% difference between the best fitness value in the current generation compared to the previous one. However, this convergence criteria could be ignored so the algorithm runs until the specified number of generations.

In order to maintain genetic diversity, it is necessary to randomly mutate encoded hyperparameters according to a user-specified mutation rate, p_m. Mutation prevents the solution from getting stuck in local minima traps [28]. High mutation rates tend to slow the algorithm's convergence to single values, while low rates tend to limit the searching process. Therefore, we have found a good trade-off in choosing a 6% mutation probability rate. In initial tests where DLEO was being configured, we had set a mutation rate higher than 10%, however after a few generations solutions did not seem to converge to single architecture designs but rather kept exploring the space.

The genetic algorithm was challenged by the high incidences of local minima when optimizing deep networks. Therefore, fine tuning between mutation, crossover and the initial population size might be of interest to achieve a specific goal. If computational resources and time are available, increasing the initial population size should be the best option to improve the algorithm's performance in terms of finding a unique global optimum. Many randomly generated architectures while optimizing deep networks seem to perform similarly and surprisingly well for most hyperparameter configurations. However, there was always clear paths towards a very few outperforming configuration designs. Compared to shallow scenarios with a few trainable parameters, these optimal deep networks allow for significantly better global optima to exist in the stochastic gradient descent optimization problem performed by the TensorFlow engine.

Monitoring and visualizing the genetic algorithm process allowed us to identify design patterns and their influence over the regression accuracy. For example, optimal architecture designs seem to have a divergent shape towards the output size. Considering that we are building a network that extracts information from 241 inputs to produce 8193 outputs, it makes sense to have a divergent architecture shape. However, choosing the appropriate number of neurons in each layer, as well as the dropout rate, is a non-trivial process specific to each dataset. In our case, the first hidden layer performed better only with a few neurons above the input size (241) and below 1000 units. However, there are a couple of optimal designs that use a higher neuron count in the first layer combined with a dropout rate to compensate and maintain accuracy. For example, a DLEO job to find the optimal number of units in one hidden layer of a two-layer network (one hidden layer and one output layer) returned a value 966 neurons (as shown in Fig. 6). This job was initialized with a population size of 40 and configured to run for 20 generations, however it found the optimal value in the very first generations. This example case successfully converged to a very close range of values and then the best value out of them was returned (966).

The optimization process is visualized using a parallel coordinate plot that allowed us to observe the design progress and genome exploration as line paths. Fig. 7 shows a parallel coordinate plot with filtered visualizations for a DLEO job configured with parameters given in Table 1. To the far left, iteration axis shows how many candidates have been evaluated, and to the far right their fitness or objective function value. Because this plot can get messy, the selected iteration colormap in the top image is useful to observe the performance of the last generations in terms of convergence. As it can be observed, the last generations (red) all lead to very low objective function values (validation loss) which indicates signs of convergence. The plot in the middle shows a filtered visualization with a selected range of candidates to observe architecture design possibilities within a selected objective function tolerance. Moreover, the bottom plot shows the top 5 designs for model selection and further studies. From Fig. 7 we can see that the optimal design range for the first hidden layer lies between 241 to 1,200 neuron units.

However, when considering the dropout rate, this range is significantly reduced. In addition, the model seems to be less sensitive to the number of units in layer 2 with a range from about 1,700 to 14,000 units. For the last layer, the optimal number of neurons seem to be slightly higher to the output size (8193). Dropout tendencies are towards lower values of dropout in the first layer 0.1~0.25, higher dropout in the second and middle layer 0.5~0.6, and again lower dropout for the third layer between 0.1~0.3. The fact that the GA computed optimal dropout of 0.5 in the middle-hidden layer is interesting in that the developers of dropout, Srivastava et al. [29], suggest that 0.5 is an optimal value to use in the hidden layers, with exception to earlier and latter layers. Fig. 9 provides further information about the influence of each hyperparameter over the objective function via Pearson's correlation coefficient, which defines linear correlations between variables. As can be observed, the number of units in the first hidden layer has the highest correlation with the objective function.

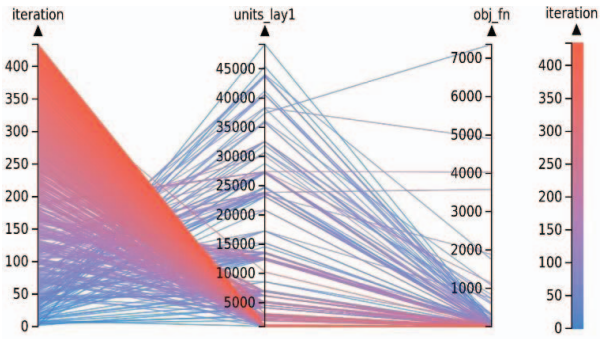

Fig. 6. DLEO implementation on a fully-connected network of two layers. In this simple case, only one hyperparameter was being optimized.

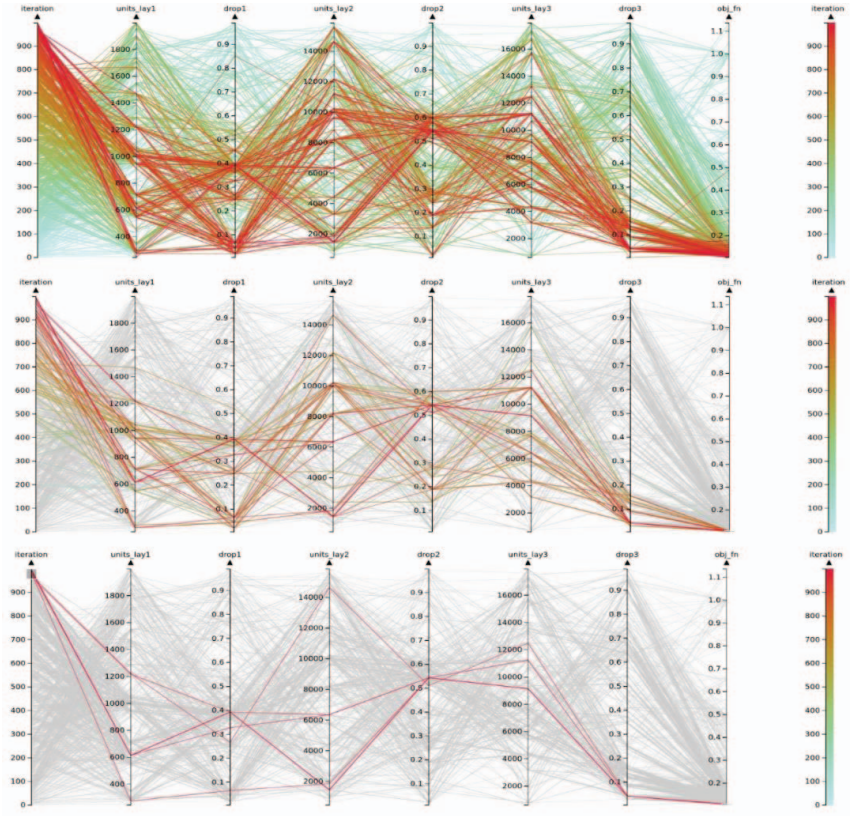

Fig. 7. DLEO parallel coordinate plot. From top to bottom, all evaluations are visualized and then filtered across the iteration and objective function axes to show the best performing DNNs. The bottom image shows the top 5 configurations.

DLEO jobs with a high number of hyperparameters in deep FCLs are significantly harder optimization problems to solve, however appropriate configurations yield impressive accuracy results. If the genetic algorithm is dealing with a big problem and does not run long enough, it will not converge to a single or few designs. However, there is a possibility that multiple architecture designs exist, making it harder for the GA to identify only a single path. To verify if there are multiple architecture designs, it is necessary to perform an isolated test by fixing all other hyperparameters. For example, we only solved for the number of neurons in the second hidden layer, as shown in Fig. 8, while holding the other parameters fixed. This is a similar test to the one presented in Fig. 6 where we only solved for the number of neurons in the first and only hidden layer. Although 20 generations might not be sufficient for the GA to converge to a single architecture, it should be enough to provide insight about the optimal search space within a closed range of values. However, that was not the case, and results for the optimal number of neurons in the second hidden layer show there are multiple paths for a target accuracy with a 0.01~0.1% tolerance.

Analyzing results of DLEO implementation for two, three and four-layer networks allowed us to understand that as we go deeper, higher order interactions between inputs can be interpreted by the network. Therefore, deep architectures can map higher order interactions between features, and by using many neurons per layer, great amount of information is generated as it is required and needed for 8,193 outputs. However, generating too much information in early stages of the network could be undesirable since we have observed it can confuse the network and degrade the regression accuracy.

V. CONCLUSIONS

In this paper, we present a methodology that successfully optimizes and designs deep neural networks architectures for nonlinear multivariate regression problems. The presented method integrates robust open source libraries and DoD HPC

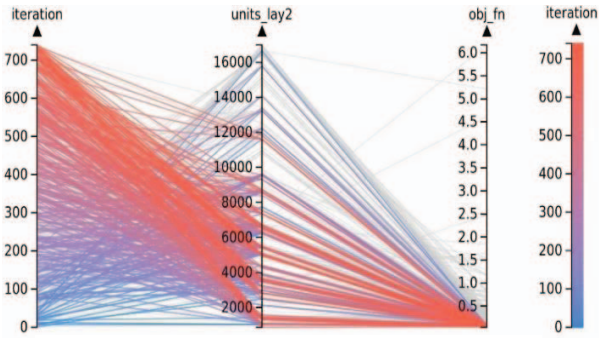

Fig. 8. Parallel coordinate plot showing lack of convergence to a single hyperparameter value.

Name	drop1	drop2	drop3	meritValue	obj_fn	units_lay1	units_lay2	units_lay3
drop1	1	-0.05273088902737…	0.02603889649004044	0.3235060346386571	0.3235060346386571	0.38535051471207654	0.11942320727909322	-0.3875629117394838
drop2	-0.05273088902737…	1	0.09405136983609952	0.3306776939810964	0.3306776939810964	0.05000268347967965	0.051273552109998…	0.09986691826686549
drop3	0.02603889649004102	0.09405136983609924	0.9999999999999999	0.31415181552333055	0.31415181552333055	0.10053206871261602	-0.01857408126440…	0.09636558879849685
meritValue	0.3235060346386571	0.33067769398109653	0.3141518155233306	1	1	0.29595427698940835	0.08839415289234574	-0.04934501125564286
obj_fn	0.3235060346386571	0.33067769398109653	0.3141518155233306	1	1	0.29595427698940835	0.08839415289234574	-0.04934501125564286
units_lay1	0.3853505147120766	0.05000268347967912	0.10053206871261568	0.2959542769894084	0.2959542769894084	1	0.07942997723650426	-0.30405296559216355
units_lay2	0.11942320727909299	0.05127355210999807	-0.01857408126439…	0.08839415289234578	0.08839415289234578	0.07942997723650459	1	-0.1292094790549945
units_lay3	-0.38756291173948…	0.09986691826686545	0.09636558879849731	-0.04934501125564…	-0.04934501125564…	-0.30405296559216…	-0.12920947905499…	1

Fig. 9. Pearson's Correlation Coefficient provide linear correlations between hyperparameters and objective function.

orchestration software. DLEO can generate optimal architecture designs in an accelerated automated workflow. By implementing this evolutionary optimization workflow, we were able to improve the validation accuracy of several models by 5-7% in less than 72 hours, compared to weeks of manually tuning models. We were also able to confirm that multiple architecture configurations can exist to achieve a target accuracy. Also, deep regression and architecture design insights were learned through the process of generating optimal designs. The resultant diverging architecture shape and its regression capabilities was well studied, in which we were able to identify optimum ratios of information generation between layers, since we saw that large ratios in early stages tend to confuse the network. Post-processing and visualization tools allowed us to study multiple architecture designs and answer fundamental questions about why deep architectures outperform shallow ones. We also present a method to constrain network architectures based on their memory requirements in order to avoid out of memory issues.

APPENDIX

In an effort to improve the reproducibility of these results, we are including this Appendix. In this work, we used the following libraries:

- Python==3.6.5
- h5py==2.7.1
- tensorflow==1.8.0
- Keras==2.1.6
- numpy==1.14.3
- scikit_learn==0.19.2
- PyYAML==3.13

Moreover, we have made our Dakota configuration files available via https://github.com/danielmtz107/MLHPC2018.

ACKNOWLEDGMENT

This material is based upon work supported by, or in part by, the Department of Defense High Performance Computing Modernization Program (HPCMP) under User Productivity, Technology Transfer, and Training (PETTT) contract # GS04T09DBC0017. Any opinions, finding and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the DoD HPCMP.

REFERENCES

[1] D. L. Donoho, "High-dimensional data analysis: The curses and blessings of dimensionality," *AMS math challenges lecture,* Aug. 2000.

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

[3] T. Poggio, H. Mhaskar, L. Rosasco, B. Miranda, and Q. Liao, "Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review," *International Journal of Automation and Computing*, vol. 14, no. 5, pp. 503-519, March 2017.

[4] V. Belagiannis, C. Rupprecht, G. Carneiro, and N. Navab, "Robust optimization for deep regression," In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2830-2838, Dec. 2015.

[5] S. R. Young, D. C. Rose, T. P. Karnowski, S. H. Lim, and R. M. Patton, "Optimizing deep learning hyper-parameters through an evolutionary algorithm," In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, ACM, pp. 4, Nov. 2015.

[6] S. Lathuilière, P. Mesejo, X. Alameda-Pineda, and R. Horaud, "A Comprehensive Analysis of Deep Regression," *arXiv*:1803.08450, March 2018.

[7] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281-305, Feb. 2012.

[8] D. Wade, T. Vongpaseuth, R. Lugos, J. Ayscue, A. Wilson, L. Antolick et al. "Machine Learning Algorithms for HUMS Improvement on Rotorcraft Components," In *Proceedings of the 71st Annual Forum of the American Helicopter Society*, May 2015.

[9] P. Dempsey, J. Branning, D. Wade, and N. Bolander, "Comparison of Test Stand and Helicopter Oil Cooler Bearing Condition Indicators," In *Proceedings of the American Helicopter Society 66th Annual Forum and Technology*, May 2010.

[10] R. Mobley and R. Keith, *An introduction to predictive maintenance*. Waltham, MA: Butterworth-Heinemann, 2002.

[11] B. A. Ellis, "Condition based maintenance," *The Jethro Project*, 2008.

[12] G. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi, "Machine learning for predictive maintenance: A multiple classifier approach," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 812-820, June 2015.

[13] G. Kejela, R. M. Esteves and C. Rong, "Predictive Analytics of Sensor Data Using Distributed Machine Learning Techniques," *2014 IEEE 6th International Conference on Cloud Computing Technology and Scien*ce, Singapore, pp. 626-631, Dec. 2014.

[14] S. Kunze, R. Poeschl, A. Faschingbauer and M. Eider, "Artificial neural networks based age estimation of electronic devices," 2017 International Conference on Optimization of Electrical and Electronic Equipment (OPTIM) & 2017 Intl Aegean Conference on Electrical Machines and Power Electronics (ACEMP), Brasov, pp. 827-832, July 2017.

[15] H. Zhao, J. Wang, and P. Gao, "A deep learning approach for condition-based monitoring and fault diagnosis of rod pump system," *Services Transactions on Internet of Things*, vol. 1, no. 1, June 2017.

[16] P. Jahnke, "Machine learning approaches for failure type detection and predictive maintenance," Master's Thesis, Dept. Comp. Sci., Technische Universität, Darmstadt, Germany, 2015.

[17] K. Wang, "Intelligent predictive maintenance (IPdM) system–Industry 4.0 scenario," *WIT Transactions on Engineering Sciences*, vol. 113, pp. 259-268, 2016.

[18] Z. Li, Y. Wang, and K. S. Wang, "Intelligent predictive maintenance for fault diagnosis and prognosis in machine centers: Industry 4.0 scenario," *Adv. Manuf.*, vol. 5, no. 377, Dec. 2017.

[19] D. Wade and A. Wilson, "Applying machine learning-based diagnostic functions to rotorcraft safety," *17th Australian International Aerospace*

Congress: AIAC 2017. Engineers Australia, Royal Aeronautical Society, pp. 663-670, Feb. 2017.

[20] A. Wilson and D. Wade, "Reconstructing Spectra from IVHMS Condition Indicators," Presented at *the American Helicopter Society 73rd Annual Forum & Technology Display*, Fort Worth, Texas, May 2017.

[21] E. Real, S. Moore, A. Selle, S. Saxena, Y. Suematsu, J. Tan, Q.V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv*:1703.01041, June 2017.

[22] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv*:1611.01578, Feb. 2017.

[23] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv*:1712.06567, April 2018.

[24] S. Wojtkiewicz, M. Eldred, R. Field, A. Urbina, and J. Red-Horse, "Uncertainty quantification in large computational engineering models," In *19th AIAA Applied Aerodynamics Conference*, pp. 1455. 2001, June 2001.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean et al., "TensorFlow: A System for Large-Scale Machine Learning," In *OSDI*, vol. 16, pp. 265-283, Nov. 2016.

[26] K. Hacker, J. Eddy and K. Lewis, "Efficient Global Optimization Using Hybrid Genetic Algorithms," In 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, pp. 5429-5439. Sept. 2002.

[27] D. Lea and G. Wolfram, "A memory allocator," 1996 [Online]. Available: http://g.oswego.edu/dl/html/malloc.html, [Access August 5, 2018].

[28] W. Y. Lin, W. Y. Lee, and T.P. Hong, "Adapting crossover and mutation rates in genetic algorithms," *J. Inf. Sci. Eng.,* vol. 19, no. 5, pp. 889-903, Sept. 2003.

[29] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929-1958, June 2014