Interactive Distributed Deep Learning with Jupyter Notebooks

Steve Farrell[1], Aaron Vose[2], Oliver Evans[3], Matthew Henderson[3], Shreyas Cholia[3], Fernando Pérez[3], Wahid Bhimji[1(✉)], Shane Canon[1], Rollin Thomas[1], and Prabhat[1]

[1] NERSC, Berkeley, CA, USA
{sfarrell,wbhimji}@lbl.gov
[2] Cray Inc., Seattle, WA, USA
avose@cray.com
[3] Lawrence Berkeley National Laboratory, Berkeley, CA, USA
scholia@lbl.gov

Abstract. Deep learning researchers are increasingly using Jupyter notebooks to implement interactive, reproducible workflows with embedded visualization, steering and documentation. Such solutions are typically deployed on small-scale (e.g. single server) computing systems. However, as the sizes and complexities of datasets and associated neural network models increase, high-performance distributed systems become important for training and evaluating models in a feasible amount of time. In this paper we describe our vision for Jupyter notebook solutions to deploy deep learning workloads onto high-performance computing systems. We demonstrate the effectiveness of notebooks for distributed training and hyper-parameter optimization of deep neural networks with efficient, scalable backends.

Keywords: Jupyter · Deep learning · Distributed training
Hyperparameter optimization · High-performance computing
Genetic algorithms

1 Introduction

Deep learning (DL) [14], the sub-field of machine learning which uses multi-layer neural networks (NNs) to solve complex tasks with data, has gained a great deal of popularity in recent years in part due to the availability of large datasets and increasingly powerful computing resources. Meanwhile, Jupyter [13] notebooks enable code, graphical results, and rich documentation to be combined into an interactive computational narrative, and have become the de facto standard for data science collaboration, development and pedagogy. Notebook-based DL workflows are widely used but typically deployed on small-scale computing systems (e.g. single server), but as the sizes and complexities of datasets and associated neural network models increase, distributed computing systems become

R. Yokota et al. (Eds.): ISC 2018 Workshops, LNCS 11203, pp. 678–687, 2018.
https://doi.org/10.1007/978-3-030-02465-9_49

essential for generating the best results in a reasonable time. In particular, models that are slow to train can benefit from distributed data-parallel training, and model-selection tasks can be accelerated with distributed hyper-parameter optimization (HPO). The development of scalable notebook-based DL workflows for high performance computing (HPC) systems will therefore be highly valuable to the DL and science communities and will enable new kinds of human-in-the-loop interactivity and monitoring in DL-based research.

In this paper we demonstrate several distributed notebook-based deep learning workflows on HPC systems, interfacing to the Cori supercomputer at NERSC. In Sect. 2, we describe our architecture to effectively use these HPC resources via Jupyter, taking into account the hardware and policy restrictions that are common with other large HPC machines. We then present our notebook-based approaches, including scaling results for distributed training in Sect. 3 and for distributed HPO in Sect. 4.

We take advantage of and build on features available in the Jupyter and Python ecosystem including JupyterHub, IPyWidgets, IPython "magic" commands, Dask, and IPyParallel which provide us with tools to interact with the backend tasks directly through a visual interface. This allows us to close the interactivity loop so that operating real-time, entirely within the notebook, we can allocate nodes on the Cori supercomputer; configure, monitor, tune and steer deep learning training runs; and perform further analysis on the outputs of models. As well as human interaction we couple this infrastructure with a powerful automated hyper-parameter framework incorporating genetic algorithms and population-based training as described in Sect. 4.3. Example notebooks and recipes for running these at NERSC are provided at the links in Sect. 6.

We demonstrate these methods with a science use-case and dataset from High Energy Physics which applies convolutional neural networks (CNNs) to classify images formed from Large Hadron Collider events, and is described in detail in [8]. We find that we can improve on state-of-the-art results by using distributed HPC resources to improve model parameters.

2 System Architecture

In order to achieve the aims of this project, we extended the Jupyter infrastructure at NERSC [1] to be able to execute distributed training on the Cori supercomputer [2]: a Cray XC40 with 2388 Intel Haswell and 9688 Intel Xeon Phi Knight's Landing compute nodes. The overall architecture is shown in Fig. 1. The components of this are described in more detail below.

The JupyterHub installation at NERSC [1] provides a multi-user gateway through which individual users can authenticate and spawn a private notebook server on the Cori login nodes to launch notebooks. Each notebook is associated with a Jupyter kernel, a wrapper around the Python process that executes user code. Currently these kernels run on the login node itself, a configuration that is widely used at NERSC, sufficient for small workloads and allows access to the Cori file-systems and for submitting scripts to the Cori batch-queues.

However, here we further use the HPC batch queue system to allocate dedicated compute nodes for executing long-running and resource-demanding tasks such as DL training from the notebook. To this end, we use two distributed execution frameworks, IPyParallel [3] and Dask Distributed [11,16], which facilitate the use of remote compute resources from the notebook. In both frameworks, a central Controller handles scheduling and communication with a number of execution Engines that are instantiated by the user. In our execution model (Fig. 1), the Controller and Engines run from the compute nodes after a batch allocation has been provided. To minimize boilerplate, we have created an IPython "magic" command[1] (an IPython-specific command beginning with `%`) which allocates HPC resources and deploys the IPyParallel cluster with a one-line command from the notebook.

As a further extension, the ability to individually stop and start tasks is supported through Kale [4], an extension to the Jupyter ecosystem for interactive HPC. Kale wraps each task with a worker process that provides full control of the task as well as resource usage monitoring for the task and participating compute node.

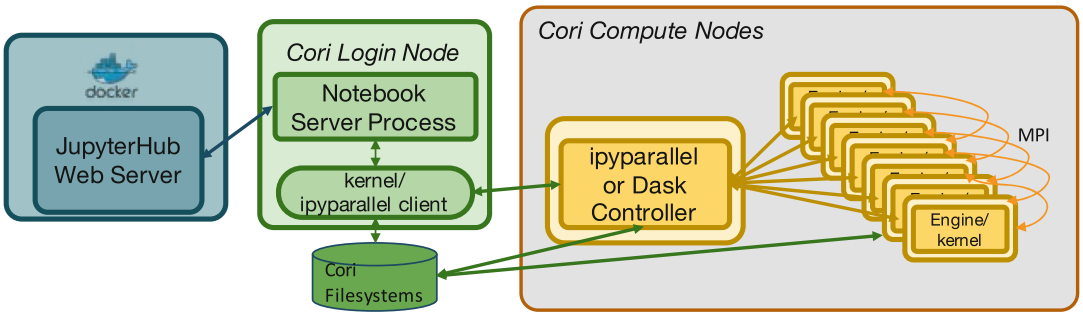


Fig. 1. Diagram of the distributed workload system. A user launches a notebook process via JupyterHub. They obtain an allocation of compute nodes on the batch system and launch the cluster processes (e.g. IPyParallel controller and worker engines) via a SLURM script or our `%ipcluster` magic within the Jupyter notebook. They then connect to the cluster via the notebook; launch the NN training tasks to the batch system and continue to monitor and interacts with those tasks. The notebook to cluster coordination occurs in communication with the controller node, however large-scale distributed training can avoid this bottleneck via the native MPI interface of, for example horovod (Sect. 3).

In order to create an interactive feedback loop to visualize results that the user can interact with, we make use of built-in Jupyter widgets (iPyWidgets [5]) along with third party Jupyter widgets qgrid [6] (Quantopian) and bqplot [7] (Bloomberg). Model results presented by the widgets are communicated asynchronously through the parallel controller (Dask or iPyParallel) in the Jupyter

[1] https://github.com/sparticlesteve/cori-intml-examples/blob/master/ipcluster_magics.py.

Kernel. We poll the kernel for these results through a background thread, updating the table in real-time as they come in, and rendering plots for the currently selected model.

3 Distributed Training

Slow training of NNs due to large datasets and complex models can inhibit DL research productivity. Distributed training techniques can speed up model learning by exploiting parallelism at the data and model levels. For example, in synchronous data-parallel training, batches of data are distributed across worker nodes along with copies of the model. Each worker processes its local batch of data and computes gradients. The gradients are then accumulated across workers to compute a global optimization step and all the models are updated synchronously. Tools like Uber's Horovod [17] and the Cray PE ML Plugin [15] provide the synchronization mechanism via MPI communication for efficiently scaling to large numbers of workers.

To implement distributed training in notebooks, we use the IPyParallel cluster with MPI-enabled worker engines. Once the cluster client is connected in the notebook, a user can mark individual cells for parallel execution using the `%%px` IPython magic command. Results from the model can be retrieved in the main notebook process for further analysis.

We used the Cori system to demonstrate the scalability of this approach with notebooks. We implemented a Horovod and Keras [9] distributed training implementation of the LHC CNN use-case and scaled up to 180 worker nodes both using the Jupyter notebook with IPyParallel infrastructure as well as a pure "batch" submission from python scripts. The training throughput, plotted in Fig. 2, shows that the notebook infrastructure introduces no noticeable overhead relative to distributed training with batch scripts.

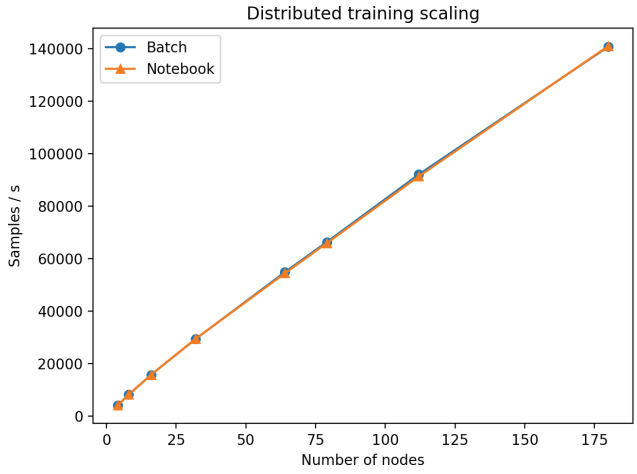


Fig. 2. Scaling of distributed training with and without the notebook infrastructure. No overhead from IPyParallel is observed up to 180 nodes on Cori.

4 Distributed Hyper-parameter Optimization

Hyper-parameter optimization (HPO) helps solve the problem of model selection for non-trained parameters such as hidden layer size, dropout probability and learning rate. HPO algorithms generally involve training models at various points in this parameter space and evaluating a metric such as model accuracy on a validation dataset. Depending on the choice of algorithm, many such tasks may be run in parallel on a distributed system.

We have developed several examples for distributed HPO with Jupyter notebooks. The first (Sect. 4.1) uses random search with independent tasks training in parallel on IPyParallel engines with load balancing. The second (Sect. 4.2) extends this with live, interactive widgets. The third (Sect. 4.3) is an advanced HPO example using population-based training via Dask scheduling.

4.1 Random Search HPO Notebook

We implemented a random search of hundreds of configurations of convolution and fully-connected layers, learning rate and dropout for the LHC CNN use case using Keras [9] in a Jupyter notebook and ran this using a 100 node allocation on the Cori system which was able to complete all experiments in under an hour.

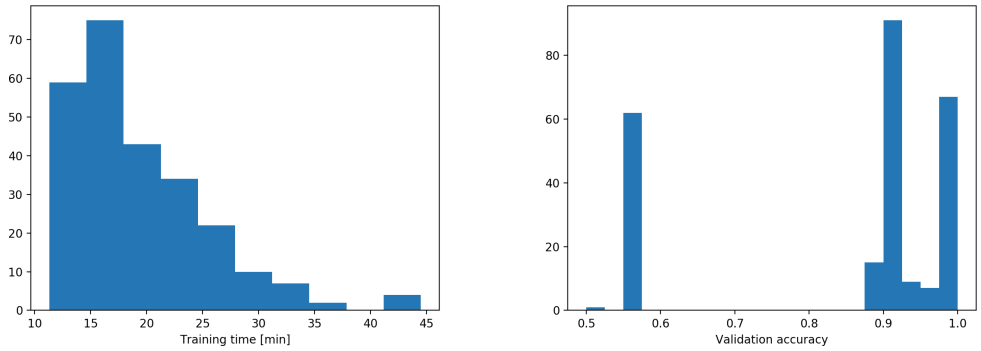


Fig. 3. Distributed hyper-parameter optimization distributions of training time [left] and best validation accuracy [right] for a collection of training tasks.

Figure 3 shows the distribution of run-times, which illustrates the value of using automatic load-balancing, and the range of model accuracies, which illustrates the need for optimizing hyper-parameters to obtain the best results. Further analysis can be conducted within the notebook to probe incoming results: for example Fig. 4 shows the loss and a ROC curve for the best performing model illustrating how performing this optimization can enable obtain similar performance results as previous analyses with a substantially smaller dataset.

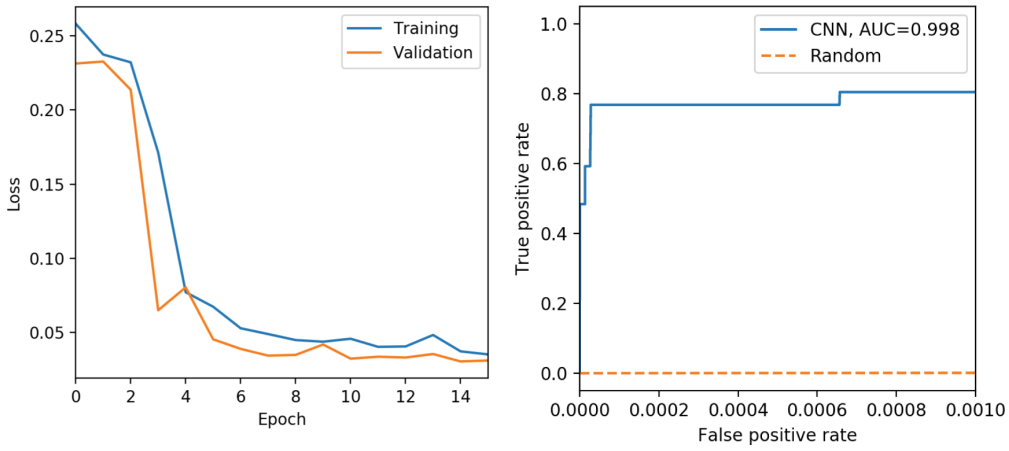


Fig. 4. Loss (left) and weighted ROC curve (right) of the best model found in hyper-parameter optimization. The achieved results are competitive with those presented in [8] while using less than 20% of the original dataset.

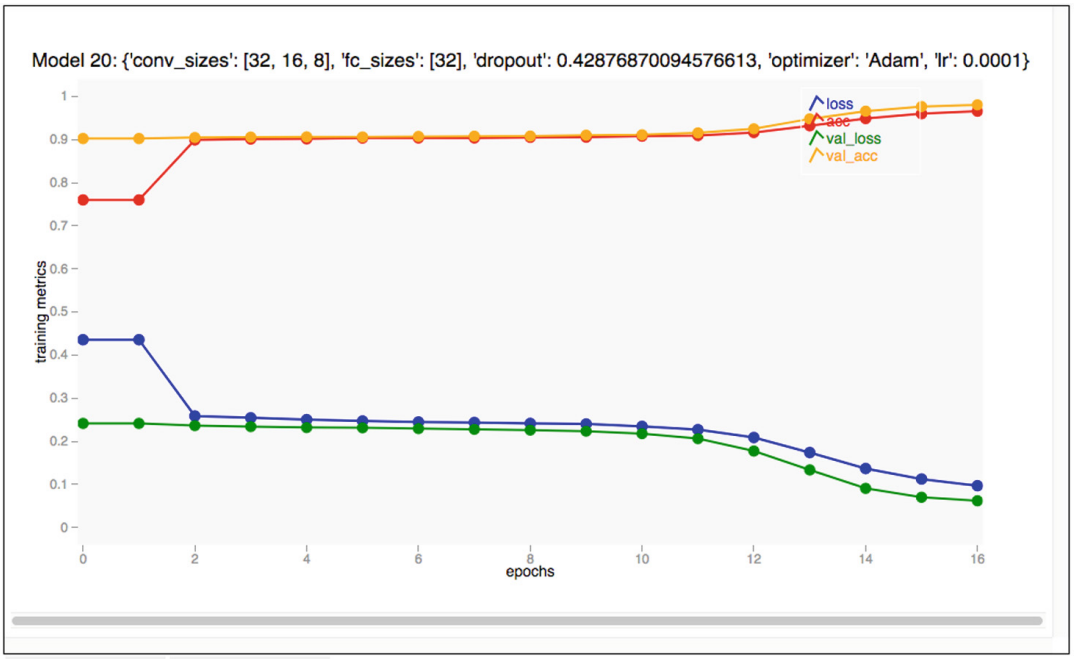
4.2 HPO with Interactive Widgets

We use Jupyter's interactive widget ecosystem as described in Sect. 2 to provide a graphical interface for controlling and monitoring training runs in human-in-the-loop hyper-parameter optimization, as shown in Fig. 5. Individual points in the hyper-parameter space can be specified in the notebook and Keras is used to construct and train the DL model with remote execution provided by IPyParallel. The qgrid widget provides a dynamic table interface for displaying the latest results and bqplot widgets provide live plotting of the full results. These tools are clickable and interactive, allowing us to switch between results or send messages back to the python kernel for further processing. This allows for dynamic task control and job steering. Based on the live results of individual training runs, the user can decide which areas of the parameter spaces are worth exploring further, and which areas are not likely to produce viable models, all before training is complete. Poorly performing runs can be prematurely canceled and replaced with runs from more hopeful regions of the hyper-parameter space.

4.3 Advanced HPO

As a more advanced example, we consider a variant of population-based training (PBT), which combines gradient-based approaches such as stochastic gradient descent (SGD) with a genetic algorithm (GA) to optimize both parameters (i.e., weights and biases) and hyper-parameters simultaneously [12]. By running a genetic algorithm synchronized with NN training such that each generation in the GA corresponds to some fixed number of SGD epochs, the GA applied to the HPs can be extended to optimize the model parameters.

At the end of each generation, each member of a population of neural networks is evaluated on a validation set to produce a fitness score based on the NN's accuracy. These fitness scores are then used to determine the probability of



index	status	epoch	conv_sizes	fc_sizes	dropout	optimizer	lr	loss	val_loss	acc	val_acc
15	Ended Traini...	15	[8, 64, 32]	[256]	0.01919	Adam	0.001	0.0067703...	0.058146...	0.99775	0.98596875
19	Ended Traini...	15	[64, 16, 64]	[256]	0.61802	Adam	0.001	0.0497609...	0.039992...	0.982625	0.98496875
21	Ended Traini...	15	[4, 4, 8]	[64]	0.13547	Adam	0.0001	0.0735504...	0.051296...	0.9740625	0.982
20	Ended Traini...	15	[32, 16, 8]	[32]	0.42877	Adam	0.0001	0.0971096...	0.062174...	0.96575	0.98021875
18	Ended Traini...	15	[8, 8, 16]	[128]	0.29008	Adam	0.01	0.0700781...	0.073561...	0.975390...	0.969625
16	Ended Traini...	15	[8, 8, 8]	[256]	0.30157	Nadam	0.01	0.1863622...	0.162577...	0.927140...	0.93790625
25	Ended Traini...	15	[32, 8, 32]	[64]	0.57433	Nadam	0.01	0.2148241...	0.181064...	0.91615625	0.93253125
13	Ended Traini...	15	[8, 64, 128]	[256]	0.42386	Nadam	0.01	0.2105947...	0.191747...	0.9185	0.92828125
24	Ended Traini...	15	[16, 8, 128]	[128]	0.59087	Nadam	0.01	0.2333927...	0.201570...	0.9076875	0.92446875

Fig. 5. Screenshot of interactive Jupyter widget for HPO. Each row in the table showa a parameter set which has been submitted to the batch system. Loss and accuracy values in the table update live as the training progresses. Clicking on a row plots live metrics for the associated run.

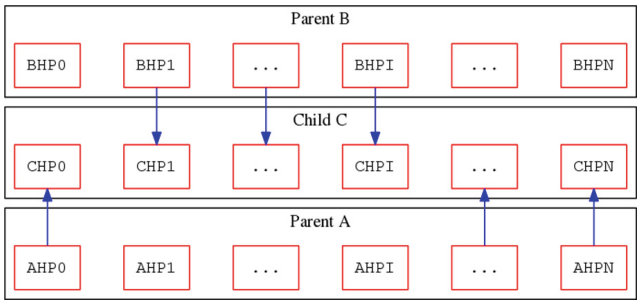


Fig. 6. Crossover combines hyperparameters from parents to create those of a child.

reproduction, where the most fit NNs are more likely to reproduce. Over multiple generations, the population of NNs evolves toward a good set of hyperparameters and parameters. The PBT variant in this work utilizes reproduction with pairs of NNs, where two sets of hyperparameters are mixed via crossover, depicted in Fig. 6, to create a child. The use of a GA with sexual reproduction and crossover provides for faster adaptation than simpler asexual reproduction [10].

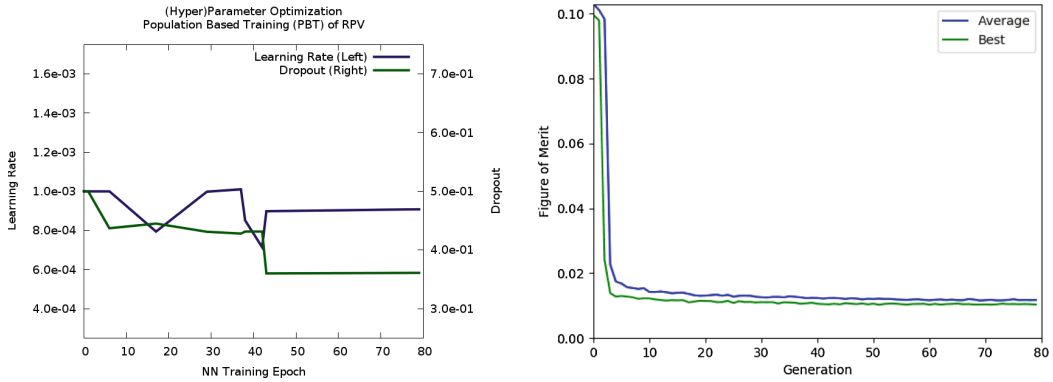


Fig. 7. Hyperparameter schedule discovered via HPO with PBT on the LHC CNN (left) as well as the average and best figure of merit in the population (right).

Figure 7 shows a hyperparameter schedule found through the application of HPO with PBT to the LHC CNN neural network (left) along with the average figure of merit and the FoM of the most-fit population member in each generation during the HPO process (right). This training schedule shows improvement can be attained through the application of a slightly decreasing learning rate as well as decreasing dropout. As can be seen in the figure of merit plot on the right (lower is better), the population of NNs quickly plateaus after only a few epochs, entering a phase which likely benefits from the different training regime introduced at that point by the discovered training schedule.

While automated approaches like PBT take a lot of the repetitive work out of HPO, they can achieve even better results when guided by humans. Specifically, the HPO process can be fine-tuned for specific NN and dataset combinations by adjusting details of the HPO search process such as the mutation rate and crossover rate of a GA. The image on the right of Fig. 7 is taken from a live plot produced in a Jupyter notebook which updates in real time during the HPO search. If the human in the loop sees that the automated HPO search is not performing well, details such as the mutation rate or search starting point can be quickly changed. Interactive and automated HPO can be combined to achieve better results than would be possible using either approach alone.

5 Conclusions

We have demonstrated that Jupyter notebooks can be a powerful interface for deploying distributed deep learning workflows on HPC systems. By framing mod-

ern DL frameworks on large computational resources in an interactive context, we achieve a human-in-the-loop system which enables rapid development of NN models through expert guidance of automated training.

We demonstrate several examples of this, all driven entirely from Jupyter notebooks, including distributed training of a single model across multiple compute nodes; distributed HPO of individual models where tasks are load-balanced across nodes; the use of widgets and plots within the notebook for steering and visualizing the runs; and more advanced hyper-parameter optimization methods. All these can be run within large interactive allocations on the Cori supercomputer, with that allocation also setup and configured from within the notebook.

Future work will seek to achieve deeper integration with Kale to achieve more detailed task and node monitoring; as well as providing more insightful visualizations; and building out the suite of tools and example notebooks for use at NERSC and for the community to build upon.

This work paves the way to understanding how human domain-expertise; automated optimization tools; and deep neural networks can be optimally combined to maximize the insight we can derive in a world of ever-growing datasets while also minimizing the time and technical knowledge necessary to achieve this on large-scale computational resources.

6 Code and Recipes

Example notebooks used in this study and recipes for running at NERSC are available at https://github.com/sparticlesteve/cori-intml-examples.

Acknowledgements. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work was in part supported by the NERSC Big Data Center; we acknowledge Cray for their funding support.

References

1. http://docs.nersc.gov/services/jupyter/
2. http://www.nersc.gov/users/computational-systems/cori/
3. https://ipyparallel.readthedocs.io/en/latest/intro.html
4. https://github.com/Jupyter-Kale/kale
5. https://ipywidgets.readthedocs.io/en/stable/
6. https://qgrid.readthedocs.io/en/latest/
7. https://bqplot.readthedocs.io/en/stable/introduction.html
8. Bhimji, W., Farrell, S.A., Kurth, T., Paganini, M., Racah, E., Prabhat: Deep neural networks for physics analysis on low-level whole-detector data at the LHC. arXiv preprint arXiv:1711.03573 (2017)
9. Chollet, F.: keras (2015). https://github.com/fchollet/keras
10. Crow, J.F.: Advantages of sexual reproduction. Genesis **15**(3), 205–213 (1994)
11. Dask Development Team: Dask: Library for dynamic task scheduling (2016). http://dask.pydata.org

12. Jaderberg, M., et al.: Population based training of neural networks. CoRR abs/1711.09846 (2017). http://arxiv.org/abs/1711.09846
13. Kluyver, T., et al.: Jupyter notebooks – a publishing format for reproducible computational workfows. In: Loizides, F., Schmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas. pp. 87 – 90. IOS Press (2016)
14. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
15. Mendygral, P., Hill, N., Kandalla, K., Davis, M., Balma, J., Schongens, M.: High performance scalable deep learning with the cray programming environments deep learning plugin. In: Proceedings of CUG (2018)
16. Rocklin, M.: Dask: parallel computation with blocked algorithms and task scheduling. In: Huff, K., Bergstra, J. (eds.) Proceedings of the 14th Python in Science Conference, pp. 130–136 (2015)
17. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. arXiv preprint arXiv:1802.05799 (2018)