Evolving a Deep Neural Network Training Time Estimator

Frédéric Pinel[1(✉)], Jian-xiong Yin[2], Christian Hundt[2], Emmanuel Kieffer[1], Sébastien Varrette[1], Pascal Bouvry[1], and Simon See[2]

[1] University of Luxembourg, Luxembourg City, Luxembourg
{frederic.pinel,emmanuel.kieffer,sebastien.varrette,pascal.bouvry}@uni.lu
[2] NVIDIA AI Tech Centre, Santa Clara, USA
{jianxiongy,chundt,ssee}@nvidia.com

Abstract. We present a procedure for the design of a Deep Neural Network (DNN) that estimates the execution time for training a deep neural network per batch on GPU accelerators. The estimator is destined to be embedded in the scheduler of a shared GPU infrastructure, capable of providing estimated training times for a wide range of network architectures, when the user submits a training job. To this end, a very short and simple representation for a given DNN is chosen. In order to compensate for the limited degree of description of the basic network representation, a novel co-evolutionary approach is taken to fit the estimator. The training set for the estimator, i.e. DNNs, is evolved by an evolutionary algorithm that optimizes the accuracy of the estimator. In the process, the genetic algorithm evolves DNNs, generates Python-Keras programs and projects them onto the simple representation. The genetic operators are dynamic, they change with the estimator's accuracy in order to balance accuracy with generalization. Results show that despite the low degree of information in the representation and the simple initial design for the predictor, co-evolving the training set performs better than near random generated population of DNNs.

Keywords: Deep Learning · Genetic algorithm

1 Introduction

Deep Learning [16] related computation has become a fast growing workload in High Performance Computing (HPC) facilities and cloud data centers DTT/B to the rapid advancement and proliferation of Deep Learning technology. It allows for scalable and fully automatic learning of robust features from a broad range of multimedia data, e.g., image, video, audio, and text. The highly regular structure of commonly used primitives in Deep Learning is amenable to massively parallel architectures such as CUDA-enabled GPUs especially when processing huge amounts of data. Nevertheless, the ever growing amount of recorded data and associated operations needed to train modern DNNs outpaces the compute

B. Dorronsoro et al. (Eds.): OLA 2020, CCIS 1173, pp. 13–24, 2020.
https://doi.org/10.1007/978-3-030-41913-4_2

capabilities of mid-sized data centers usually found in academia. As an example, a state-of-the-art transformer network such as GPT-2 [20] exhibits 774 million trainable parameters in comparison to the 62 million parameters of AlexNet [15] from 2012. In the last decade it has empirically been observed [3] that the quality and amount of data might have a significantly higher impact on the model quality than the specific choice of classifiers. Recent research [11] suggests that the same is applicable to Deep Learning – empirical improvement of generalization properties is correlated to increasing amounts of training data. As a result, Deep Learning is widely adopted by a broad range of scientists in a diversity of disciplines not necessarily related to computer science.

This demand can be addressed by efficient scheduling of Deep Learning tasks to fully saturate available compute resources. However, existing job schedulers in large scale compute cluster resource management system or large scale batch processing framework such as MESOS [13] in a Tensorflow Cluster [2], YARN [22] in MXNet cluster [5], SLURM [24] in general, scientific High Performance Computing tends to statically allocate compute resource based on user resource quota or requested quantity. (1) Using dynamic resource allocation, (2) recommending optimal job execution time to users from scheduler perspective are two natural ideas for improvements.

In the case of static resource allocation, the resource allocation is done one-time off when the resource for the job is initialized with the best matching resource, and it might prevent the job from getting accelerated from later released more suitable compute resource unless manually reconfigured by cluster operation team or job submitter.

Deep Learning training time highly depends on DNN model architecture and other factors such as training environment and setup, and the training finish time still highly depends on human empirical observation, hence is challenging for average job submitter to estimate job execution time without special knowledge on the targeting system. If a recommended DNN training job time could be provided, job submitter will be able to better manage not only their job monitoring cycle but also model development turn-around-time, hence save the compute resource from being occupied in the long tail of DNN training.

In this paper, we present a DNN training time per batch estimator, which aims to address the common requirements of DNN execution time estimation which could potentially pave the path forward towards an intelligent Deep Learning task scheduler. In our work, we empirically assume batch size as the major hyperparameter factor, and accelerator throughput as the major environmental factor, for execution time.

Moreover, estimating a training time allows to assess the cost of training (as in the pay-per-use model of cloud computing). This cost estimate is useful per se, but also influences the design process as it controls the number of neural architectures to explore, hyperparameter tuning and data requirements, all of which contribute to the accuracy of the model [14].

The proposed DNN training time per batch estimator (abbreviated as DTT/B from here onwards) can be used by data center and HPC task scheduler,

which would complement its estimation with additional information such as data volume, allocated resources (i.e. GPUs) and their characteristics. For this purpose the DTT/B estimator's role is to provide a time estimation for any given DNN architecture, with a standard batch size (32), on a single GPU accelerator. The approach followed for the design of the DTT/B is our contribution, and different DTT/B can be designed under the same approach. More specifically, our contributions are:

- A DNN that predicts the training time of a submitted Keras [6]-TensorFlow [1] model for a batch.
- A simple, succinct representation for DNNs, that is easily extracted from the model definition (such as source code).
- A novel co-evolutionary [12] like procedure to train the DTT/B DNN. The training data necessary to fit the DTT/B (i.e. different DNNs for which the DTT/B predicts runtimes) is grown incrementally over successive generations of a genetic algorithm [9]. The new DNNs are generated according to their predicted runtime: the training data for the DTT/B evolves with the accuracy of the DTT/B. Also, the DNNs evolved are converted to executable Python-Keras DNN programs.

The description of the DTT/B DNN, the simple representation and the co-evolutionary data generation process are presented in Sect. 3.

2 Related Work

Paleo [19] is an analytical model for runtime, fitted with measured runtimes on specific architectures. The results show that accurate estimates for different components (networking, computation) can be produced from simple models (linear). Paleo's approach relies on detailed representation of an architecture (FLOP for an algorithm). The analytical models are fitted from few training data, and evaluated on one unseen example, its generalization is therefore uncertain. Moreover the hyperparameter space and data dependency are not dependent variables. Our approach is similar to Paleo's in that different models are used for different factors (networking, computation, memory), yet, a higher-level description of an architecture is used (Tensorflow code). Data and hyperparameters are also explicitely included. Generalization is a key objective, and is accordingly reported in our results.

NeuralPower [4] is a predictor for energy consumption of a Convolutional Neural Network (CNN) inference, and relies on runtime prediction. The scope of our paper is the prediction of the training time, not the inference time, of any DNN, not only CNN. Compared with [19], NeuralPower differs in the choice of the model class to be fitted (polynomial), and improves the model fitting with cross-validation. It is similar in that the same lower-level features are used (FLOP, memory access count). Also, it is based on a few CNN architectures. The differences with our approach are therefore similar to those mentioned in

the review of [19]. In addition, NeuralPower considers only CNNs, and their prediction runtimes, not training times.

Approaches similar (from this paper's perspective) to Paleo and NeuralPower are presented in [18,21,23]. The runtime prediction model is composed of several analytical sub-models. Each model is fitted with measurements obtained from a selection of well-known CNNs. The accuracy of the predictions are evaluated on a limited number of CNNs (typically three). As with the above results, the models rely on detailed features of the algorithms (for example: FLOP, clock cycles), and the hardware. In addition, the target platform in [23] is Xeon PHI, and the prediction model's generalization is aimed at the number of threads. Also, the runtime predicted in [8,21] is for CNN inference, because their objective is to tailor a CNN inference model for the specific user needs.

Our approach does not fit an analytical model of detailed information on the algorithms used. Also, our scope is not restricted to CNNs.

In [14], the predictor is trained from existing architectures (restricted to Fully Connected Networks -FCN- and CNN) and their respective data sets. The model estimates the runtime per type of layer, under different hyperparameters and GPU configuration. Unseen architecture runtimes are said to be extrapolated from these individual layers (but not composition rule is provided). This estimator design leads to a large input space, that is sampled to train the estimator. Also, they propose a complete runtime estimator, whereas this paper focuses on a part of a larger estimator. They report results for a variety of GPUs. The predictor we present is also a DNN, but in contrast, our proposed batch estimator aims to make predictions from features derived from known architectures, where those features will also be available in future or unseen architectures (not just individual layers). The estimator's training data is generated with a genetic algorithm throughout the estimator's training. Our estimator also aims to support any, unseen, data set (data records and hyperparameters).

[7,17] collect training times for well-known DNNs. This is related to our work because it records measured runtimes of known DNNs, yet fundamentally different as it does perform any prediction.

GAN [10] could be applied to the generation of DNN architectures for training the DTT/B, but it pursues a different objective from the evolutionary approach we present. GAN would generate DNNs in-the-style-of a given model, while we also need to generate different DNNs to cover any future architecture.

3 Proposed Approach

3.1 Overview

As mentioned in Sect. 1, the objective of the DTT/B is to predict the training time per batch of a given DNN, from a model representation that is easily extracted from the DNN definition or source code. As the DTT/B is to be embedded in a scheduler of a shared infrastructure of GPUs, the simplicity of the representation is more important than its accuracy, because only a simple

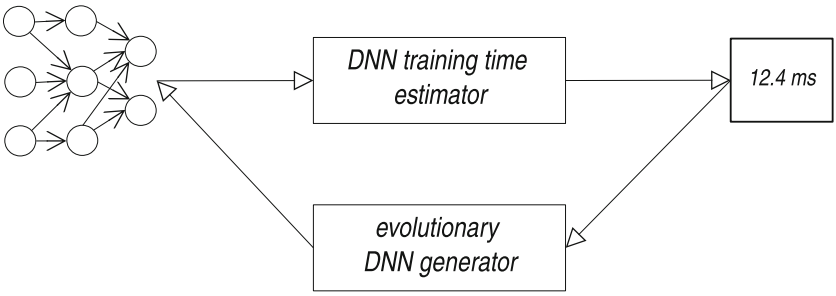

Fig. 1. Overview of the DTT/B evolutionary design.

representation will allow the DTT/B to be actually deployed, whereas estimation errors can be accounted for.

The approach presented consists of a very simple representation of the DNN, the DTT/B modeled as a DNN, and a co-evolutionary process that generates appropriate training data for the DTT/B. Figure 1 illustrates that the DTT/B is evolved through its training set. DTT/B accepts as input DNN representations, predicts runtimes that then serve to evolve the next training data set, such that each cycle -or generation- improves the DTT/B's accuracy.

3.2 DTT/B Model

The DTT/B is modeled as a DNN, and defined by the code listing below. The DTT/B is a simple sequential DNN, because the key element in the DTT/B's design is not the DNN design, but the training data set used for its fitting [11]. Of course, the DTT/B's architecture can be further refined to improve the prediction results. The notable feature of the DTT/B DNN is that it solves a regression problem: predicting a runtime.

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(
    32, input_shape=(32,), activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(
    64, input_shape=(32,),activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(
    64, input_shape=(32,), activation='relu'))
model.add(tf.keras.layers.Dropout(0.2))
model.add(tf.keras.layers.Dense(
    1, activation='linear', kernel_initializer='zeros'))
model.compile(loss='mean_squared_error', optimizer='rmsprop')
model.fit(x_train, y_train, batch_size=16, epochs=500)
```

3.3 DTT/B Features

The requirement for the DTT/B is to provide runtime estimates from a readily extracted representation of a given DNN. Our approach is use a simple representation, available both to the designer of the DNN and to the scheduler of the shared computing platform.

We propose to represent a complete DNN as the *sequence of each layer's number of trainable parameters* (i.e. without any layer type information). As an example, the DNN representation of the DTT/B DNN defined above is `[1056, 0, 2112, 0, 4160, 0, 65]` as can be seen from the output of the `summary()` function of Keras applied to the DTT/B model.

```
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 32)                1056
dropout (Dropout)            (None, 32)                0
dense_1 (Dense)              (None, 64)                2112
dropout_1 (Dropout)          (None, 64)                0
dense_2 (Dense)              (None, 64)                4160
dropout_2 (Dropout)          (None, 64)                0
dense_3 (Dense)              (None, 1)                 65
=================================================================
Total params: 7,393
Trainable params: 7,393
Non-trainable params: 0
```

3.4 Co-evolving the DTT/B Training Set

The training data for the DTT/B DNN are short sequences of each layer's number of trainable parameters. In order to generate this DTT/B training data, DNNs must first be generated. Our objective is to accurately predict training

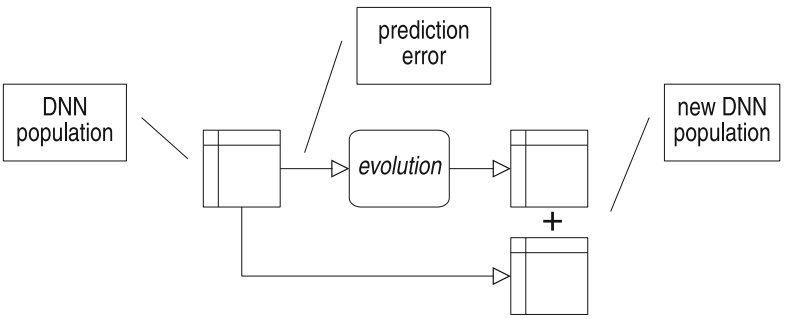

Fig. 2. Evolutionary data generation.

runtimes for DNNs similar to well-known architectures and to generalize to different DNNs that may be submitted in the future.

Our approach to meet this objective is to grow the DTT/B training data (DNNs). From an initial set of a few well-known DNNs, an evolutionary process generates additional DNNs in a co-evolutionary fashion: the population of DNNs evolves with the accuracy of the DTT/B. The intent is to add similar DNNs to the population, until the accuracy of the DTT/B is satisfactory, then to introduce different DNNs, and loop. From each DNN in the population, the simple proposed representation is extracted as input to the DTT/B.

As presented in Fig. 2, at each generation in the evolutionary process, each DNN in the population is evaluated. The evaluation consists of (1) generating executable model (a Python-Keras program), (2) executing the program and recording the observed runtime, (3) training the DTT/B with the extracted representation, (4) predicting the runtime on a test set (unseen data). This evaluation results in a runtime prediction error for the DNN. For each DNN evaluated, a new DNN is produced according to following rule:

- if the prediction error ratio is greater than 25%, then a *similar* child DNN is generated,
- if the error ratio is greater than 10%, then a *slightly different* child DNN is generated,
- if the error ratio is less than 10%, then a *rather different* child DNN is generated.

The exact meaning for similar, slightly different and rather different is defined below. The generated or child DNNs (according to the rule above) are added to the DTT/B training set (the child DNN does not replace its parent DNN). Therefore, at each generation, the population doubles.

Table 1. Elementary operations on DNN layers.

Operator name	Function performed	Domain (supported layers)
Mutation	Randomly changes several layer parameters: units, filters, kernel size, stride, use bias, regularizer function, activation function, dropout rate	Dense, Conv1D, Conv2D, LSTM, ConvLSTM2D, Dropout, Activation, BatchNormalization
Addition	Duplicates the previous layer and mutates	Dense, Conv1D, Conv2D, Conv3D, LSTM, ConvLSTM2D, Dropout, Flatten, Activation, BatchNormalization
Removal	Deletes a layer	Layers previously added

A child DNN in the population is generated by combining three elementary layer operations: mutation, removal and addition, summarized in Table 1.

The operators are valid only on sequential DNN architectures. The elementary layer operations are combined to generate a child DNN:

- A *similar* child DNN is the result of a single layer mutation.
- A *slightly different* child DNN is the result of a layer removal, mutation and addition.
- A *rather different* child DNN is the result of two slightly different changes.

The design of the layer operators aims to introduce changes that modify the chosen representation (sequence of layer variables), but also to make changes that do not, in order to test our representation with counter-examples. The layer addition and removal functions are chosen such as to ensure that almost all generated architectures produce valid DNNs.

4 Results

4.1 Experimental Setup

The initial population of DNNs consists of six well-known architectures: MNIST MLP, MNIST CNN, Reuters MLP, Conv LSTM, Addition RNN, IMDB CNN, as provided as examples by the Keras framework. The unit of work predicted is the batch training time, for a batch size of 32. The evolutionary process lasts 8 generations, leading to a maximum of 1536 DNNs.

The evolutionary algorithm operates on a JSON representation of a DNN [1]. The JSON representation is transformed into a Python program that calls the Keras framework. The Python program is then executed to record the expected batch runtime (the label). The generated python program includes instrumentation code to record the batch runtime. DNNs from the previous generation are carried over to the next generation without modification. The GPUs used for the measured and predicted batch training time are NVIDIA Tesla V100 SXM2 32GiB of memory. The evaluation of the DNN population is performed with 4-fold cross-validation, such that each DNN receives a prediction while not present in the training of the DTT/B.

4.2 Results

Table 2 summarizes the results of the evolved DTT/B through its training set. The objective is to evaluate the suitability of a simple representation (sequence of layer parameters), and the co-evolution process to train the DTT/B. The DTT/B's design is at this moment secondary and can later be changed.

In order to evaluate the co-evolutionary design of the DTT/B, the evolved training set is compared to a *more random population generation*, fourth column in Table 2. The more random generation process consists in applying the *rather different* (Sect. 3.4) changes to each DNN, indistinctly of the prediction error.

[1] https://gitlab.uni.lu/src/ola2020.

Table 2. DTT/B accuracy results.

Layer distance	**#DNNs**	**Wilcoxon sign-rank**	**Evolved training set** (median error)	**More random training set** (median error)
0	566	W=20344.500 p=0.000	**39.8%**	56.8%
1	548	W=44483.500, p=0.000	**38.9%**	40.0%
2	383	W=28469.000, p=0.000	41.5%	**39.2%**
3	237	W=13682.000, **p=0.774**	*43.2%*	*37.2%*
4	274	W=15059.500, p=0.004	52.6%	**43.7%**
5	283	W=16162.000, p=0.006	52.5%	**44.5%**
6	205	W=8048.000, p=0.003	59.8%	**49.8%**
7	119	W=3357.500, **p=0.573**	*59.1%*	*56.1%*
8	40	W=327.500, **p=0.267**	*69.2%*	*67.5%*

While the changes are more random, they apply the same elementary operations as the evolutionary process. Also, the evolutionary process applies the same changes, albeit less often.

The results shown are obtained from the 6 different models, three evolved DTT/B, and three more random. From each model's final training set (at generation 8), 10% of the DNNs are sampled and set aside for the comparison. The sampled are not used in the training of the 6 models. Thus, the test DNNs come from the final training sets of the different models. Although the evolutionary operators introduce diversity, sampling from both evolved and more random models helps measure the generalization of the estimator, which would otherwise be evaluated on DNNs issued from the same generator.

In addition, the 10% sampling is performed separately across different categories: each category corresponds to a distance between the DNN and its original template (one of the 6 initial models from Keras). The distance is expressed as the difference in number of layers (without distinguishing type). For example, Keras's MNIST MLP contains 5 layers (including 2 without trainable parameters), after 8 generations if a generated DNN contains 6 layers, the distance is 1. Column 1 of Table 2 indicates the distance to the original model. Five samplings are performed, and for each sample, each of the 6 models is tested three times. Each test DNN is therefore evaluated 9 times for the evolved DTT/Bs and also 9 times for the more random DTT/Bs.

The results indicate that when the number of layers of the evolved DTT/B training set is close to the original DNN, the evolved population yields a more accurate DTT/B (distances 0 and 1). It is important to remind that even if a layer distance is small, the evolved DNN will be significantly different from the original template, in the number of parameters (visible in the simple representation chosen) and in other properties of the layer. The evolved population and more random population is equivalent (the statistical test show the results come from the same distribution) when the layer distance is 3, 7 and 8. This means that although the evolved population initially targeted DNNs similar (in number of layers) to their original model, the resulting DTT/B still performs well on DNN with more layers. Nevertheless, because the difference in population generation between the evolutionary and more random process are small, the accuracy difference is not very different.

Overall, the current design for the DTT/B yields prediction errors of 39 to 50%. This is more than previous analytical prediction models, yet the representation of the DNNs to predict is much simpler, and the scope of DNNs is greater. More elaborate DTT/B designs can be proposed, this is considered future work, as we focused on the DNN representation and DNN generation.

5 Conclusion and Future Work

In this paper we presented an evolutionary approach to the design of a deep neural network that predicts training time per batch. The evolutionary approach consists of co-evolving the training data with the accuracy of the predictor. This approach first exploits the initial training data, then explores new DNNs once the accuracy is satisfactory (25% error). The motivation for this approach is to validate a simple representation of a given DNN for prediction: a short sequence of the number of parameters per layer. The simple representation is motivated by the pragmatic objective of embedding this predictor in schedulers of shared GPU infrastructure. The results show that the simple representation, combined with an evolutionary design is better able to predict training times than a more random data generation process (with a 39–50% error rate). With these preliminary findings, more focus can now be placed on the accuracy of the DTT/B.

Future work will consist of:

- extending the DNN evolutionary algorithm, to support more complex DNN architectures, to add a cross-over operator that will lead to a better coverage of all possible DNNs. A possible approach is to apply programming language based evolutionary techniques, by considering the DNNs models as high-level programs.
- Refining the design of the predictor. With a more capable evolutionary DNN generator, the predictor's design could also be evolved.
- Complementing the batch training runtime estimator by taking the computing resources and data size into account.

References

1. Abadi, M., et al.: Tensorflow: Large-scale machine learning on heterogeneous distributed systems(2016). arXiv preprint arXiv:1603.04467
2. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2016, pp. 265–283. USENIX Association, Berkeley (2016), http://dl.acm.org/citation.cfm?id=3026877.3026899
3. Banko, M., Brill, E.: Scaling to very very large corpora for natural language disambiguation. In: Proceedings of the 39th Annual Meeting on Association for Computational Linguistics, ACL 2001, pp. 26–33. Association for Computational Linguistics, Stroudsburg (2001). https://doi.org/10.3115/1073012.1073017
4. Cai, E., Juan, D.C., Stamoulis, D., Marculescu, D.: Neuralpower: Predict and deploy energy-efficient convolutional neural networks (2017). arXiv preprint arXiv:1710.05420
5. Chen, T., et al.: Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems. arXiv:1512.01274 (2015)
6. Chollet, F., et al.: Keras (2015). https://keras.io/
7. Coleman, C., et al.: Dawnbench: an end-to-end deep learning benchmark and competition. Training **100**(101), 102 (2017)
8. García-Martín, E., Rodrigues, C.F., Riley, G., Grahn, H.: Estimation of energy consumption in machine learning. J. Parallel Distrib. Comput. **134**, 75–88 (2019)
9. Goldberg, D.E., Holland, J.H.: Genetic algorithms and machine learning. Mach. Learn. **3**(2), 95–99 (1988)
10. Goodfellow, I., et al.: Generative adversarial nets. In: Advances in Neural Information Processing Systems, pp. 2672–2680 (2014)
11. Hestness, J., et al.: Deep learning scaling is predictable, empirically. ArXiv arxiv:1712.00409 (2017)
12. Hillis, W.D.: Co-evolving parasites improve simulated evolution as an optimization procedure. Phys. D: Nonlinear Phenom. **42**(1–3), 228–234 (1990)
13. Hindman, B., et al.: Mesos: a platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI 2011, pp. 295–308. USENIX Association, Berkeley (2011). http://dl.acm.org/citation.cfm?id=1972457.1972488
14. Justus, D., Brennan, J., Bonner, S., McGough, A.S.: Predicting the computational cost of deep learning models. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 3873–3882. IEEE (2018)
15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25, pp. 1097–1105. Curran Associates, Inc. (2012). http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
16. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
17. MLPerf: https://mlperf.org
18. Pei, Z., Li, C., Qin, X., Chen, X., Wei, G.: Iteration time prediction for cnn in multi-gpu platform: modeling and analysis. IEEE Access **7**, 64788–64797 (2019)
19. Qi, H., Sparks, E.R., Talwalkar, A.: Paleo: a performance model for deep neural networks (2016)
20. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I.: Language models are unsupervised multitask learners (2019)

21. Song, M., Hu, Y., Chen, H., Li, T.: Towards pervasive and user satisfactory cnn across gpu microarchitectures. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 1–12. IEEE (2017)
22. Vavilapalli, V.K., et al.: Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC 2013, pp. 5:1–5:16. ACM, New York (2013). https://doi.org/10.1145/2523616.2523633, http://doi.acm.org/10.1145/2523616.2523633
23. Viebke, A., Pllana, S., Memeti, S., Kolodziej, J.: Performance modelling of deep learning on intel many integrated core architectures (2019). arXiv preprint arXiv:1906.01992
24. Yoo, A.B., Jette, M.A., Grondona, M.: SLURM: simple linux utility for resource management. In: Feitelson, D., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2003. LNCS, vol. 2862, pp. 44–60. Springer, Heidelberg (2003). https://doi.org/10.1007/10968987_3