Accelerating Deep Learning Workloads through Efficient Multi-Model Execution

Deepak Narayanan*, **Keshav Santhanam***, **Amar Phanishayee**†, **Matei Zaharia***
* Stanford University † Microsoft Research

Abstract

Deep neural networks (DNNs) with millions of parameters are increasingly being applied across a variety of domains. GPUs - the compute platform of choice for DNNs - have become progressively more powerful to keep pace with this growing computational demand. However, many multi-model workloads are unable to leverage the available computational capacity of modern GPUs. For example, model search applications use smaller models to automatically design model architectures for a given task, and low-latency model serving applications operate in small minibatch regimes. We show that the natural baseline of simply launching GPU operations from different models in parallel fails to provide substantial speedups due to data transfer, memory-bound kernels, and the overhead of kernel launches for short-duration kernels. We propose HiveMind, a system designed specifically to optimize multi-model deep learning workloads. HiveMind optimizes a "model batch" by performing cross-model operator fusion, and sharing I/O across models. HiveMind then uses a parallel runtime to efficiently execute this fused graph. Preliminary results show HiveMind can accelerate simple hyperparameter tuning and multi-model inference workloads by up to 10× on NVIDIA P100 and V100 GPUs compared to sequential model execution.

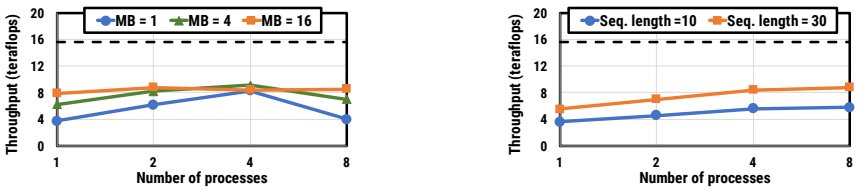
1 Introduction

Over the last five years, deep learning has facilitated groundbreaking results for many machine learning applications, including computer vision [6, 10], ranking [16], and language modeling [7]. Training and inference of deep neural networks (DNNs) for these and other tasks is extremely computationally expensive and requires powerful accelerators such as GPUs.

There are wide classes of applications, however, that are *not* able to efficiently use these accelerators. For example, inference is highly latency-sensitive in most production settings, and needs to be performed when only a few inputs are available (small minibatch sizes). Moreover, techniques like model compression and knowledge distillation [5, 9] have been developed to generate less-computationally intensive models that can mimic the predictions of more expensive models. In addition to workloads that operate over small minibatch sizes, other applications make use of smaller models. For example, automatic *model search* (or "AutoML") has become popular in recent years, where an algorithm automatically searches for a model architecture suitable for a specific task instead of requiring human experts to design the architecture by hand [14, 21, 22]. Since model search workloads can consume enormous amounts of resources (22,000 GPU-hours [21]), approaches to find models for large tasks first train smaller models on a simpler version of the task (e.g., 32×32 instead of 230×230 images [22]) using architectures that can then be scaled up. These smaller models are unable to fully utilize the latest GPUs.

One natural approach to improve resource utilization for these workloads is to run multiple models concurrently on the GPU. In practice, we find that this approach does not significantly improve utilization. Figure 1 shows the effect on throughput (in floating point operations per second) of concurrently executing multi-model computations for two workloads using NVIDIA's Multi-Process Service: inference on a ResNet-50 model for image classification, and training on a RNN model for

Preprint. Work in progress.



(a) ResNet-50 inference. **(b)** RNN for Language Modeling, training.

Figure 1: Throughput vs. number of processes for an image classification inference workload and a Language Model training workload on a V100 GPU. The dotted line in both graphs represents the advertised peak device throughput of the V100 GPU. MB in Figure 1a stands for "Minibatch Size".

language modeling. We focus on three important takeaways: a) Performing computation on a single model severely underutilizes the GPU, b) Parallelizing computation improves utilization, but still fails to reach peak device throughput, and c) Increasing the number of processes *does not* necessarily increase the observed throughput.

Figure 1 shows us that concurrent model computation is *insufficient* to reach peak device throughput. Data suggests that this is the case for three main reasons. First, small-model and small-minibatch workloads contain GPU kernels with low computational intensity (number of floating point operations performed per byte loaded from memory) – this means that the GPU's compute units are often stalled on memory reads. Second, data transfer and input preprocessing on the CPU (e.g. "data augmentation" operations such as rotating or slightly perturbing input images) can be expensive, preventing the GPU from being supplied with inputs fast enough (this shows up as periods of idle activity for the GPU). Third, the overhead of launching GPU kernels is often significant (up to 26.7% for low minibatch size inference of ResNet-18).

We identify three opportunities to overcome GPU under-utilization. First, many multi-model workloads like model search, hyperparameter tuning, and model ensembles use the same inputs for multiple models – in these cases, preprocessing pipelines can be shared to amortize away the preprocessing overhead. Second, model search workloads like Efficient Neural Architecture Search (ENAS) [14] as well as model ensembles used for inference often feature models with shared weights – we can exploit this to concatenate inputs, increasing the computational intensity of operations like convolutions and reducing the kernel launch overhead. Finally, ensembles of fine-tuned models can share the first k layers, allowing for inputs to be shared while concatenating weights.

We exploit these opportunities in HiveMind, the first system designed to optimize the hardware utilization of *multi-model* training and inference while keeping application semantics *the same*. Current deep learning frameworks, including TensorFlow [1] and PyTorch [13], are unable to perform such kernel fusion automatically as they are optimized for the single-model use case. A recently proposed GPU cluster scheduler called Gandiva [20] supports the packing of applications onto a single GPU when the cluster is overloaded. However, Gandiva does not perform *cross-model optimization* to increase the utilization of the co-located group of models, which we call a "model batch".

Our proposed design for HiveMind features two main components: a compiler and a runtime. Given a batch of models to execute at once, HiveMind's compiler optimizes data transfer, input preprocessing and computation across the models, helping increase GPU utilization without changing application semantics. HiveMind's runtime then transforms the optimized model batch into an execution DAG, and executes this DAG on the GPU while trying to extract as much concurrency as possible.

Our initial evaluation shows us that HiveMind's compiler and runtime produce up to a $10\times$ speedup compared to running the same computation sequentially.

2 Target Workloads

In this section, we briefly describe HiveMind's target workloads.

Neural Architecture Search. Manually determining the optimal DNN architecture for a particular task is difficult. To address this challenge, a number of recent papers [2, 8, 14, 21, 22] have proposed automatic methods to find architectures that perform well for a particular task.

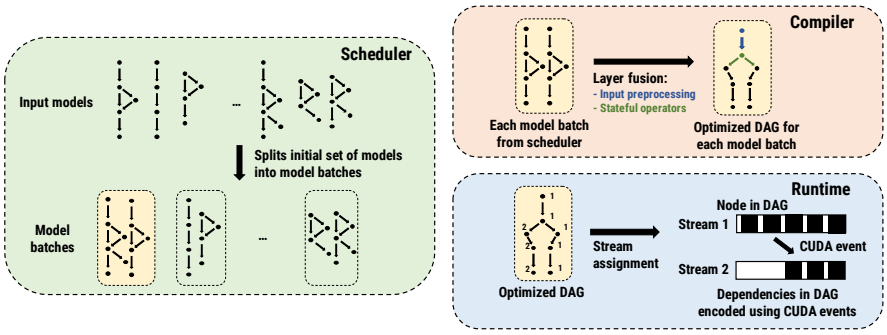


Figure 2: HiveMind's proposed architecture. Input models are first grouped into model batches using HiveMind's *scheduler*; each model batch is then optimized using HiveMind's *compiler*; the optimized DAG is then executed on GPUs using HiveMind's *runtime*, which makes use of the CUDA Stream and Event APIs.

Most neural architecture search (NAS) methods use a controller to propose architectures. Each of these candidate architectures is then trained for some number of iterations; the resulting validation accuracy is used by the controller as signal. In practice, the controller samples thousands of architectures before it can generate an architecture that outperforms human-crafted state-of-the-art solutions.

Model Cascades. Recent work has looked at using shallow architectures to speed up inference over video. NoScope [9] searches over a *pre-defined* set of architectures to find a model cascade that performs well for a certain video and query. The optimal model cascade is highly video- and query-specific – as a result, model search needs to be performed for each input video and query.

Hyperparameter Tuning. Tuning hyperparameters for a particular model architecture is a crucial step in the DNN training process. These hyperparameters include the learning rate (which controls the size of the gradient step taken every iteration) and layer-specific parameters such as dropout probability [17] and momentum [18]. Hyperparameter search involves training multiple copies of the model with different hyperparameter values.

Model Ensembles. Ensembles of models have helped produce state-of-the-art performance on a range of tasks, such as ImageNet [4]. A common type of ensemble is the Random Initialization Ensemble (RIE) which uses multiple copies of the same architecture trained with different random weight initializations. Other work [11] has looked at using ensembles consisting of fine-tuned versions of the same architecture – architectures with the same weight parameters for the first k layers.

3 HiveMind Architecture

Our proposed design for Hivemind is comprised of two main components: a compiler, and a runtime. The compiler composes models in a model batch into a single computation DAG, fuses preprocessing pipelines, and performs layer fusion across models where possible. The runtime then executes this optimized DAG on the GPU, trying to run non-dependent computations concurrently.

Currently, we assume that HiveMind is given as input models grouped into model batches that are amenable to co-optimization and co-execution. This problem is challenging in general, as can be seen by the fact that throughput can decrease as the number of parallel processes increases in Figure 1. Designing such a scheduler that *automatically* performs this grouping is future work (in practice, we have been manually grouping models into batches). The scheduler would accept as input the full set of models that need to be executed – for a hyperparameter tuning workload, this might be copies of the relevant model with different hyperparameter values; for the neural architecture search workload, this might be the different child architectures proposed by the controller; for the model ensemble inference workload, this might be the constituent models in the ensemble. HiveMind's scheduler would then split this set of input models into "model batches" that can be co-optimized and executed on the GPU at once. HiveMind's full proposed architecture is shown in Figure 2.

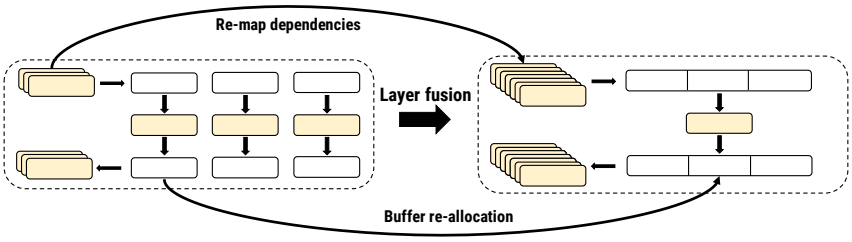


Figure 3: HiveMind's layer fusion optimization for layers with shared weights ensures that 1) the new dependencies introduced through layer fusion are respected, and 2) the outputs of the layers immediately preceding the layers to be fused are allocated in contiguous memory.

Algorithm 1 Algorithm to fuse nodes in DAG

```
1: procedure FuseNodes(G, nodes)                                   ▷ Fuses nodes in graph G
2:     prevNodes = GetPredecessors(G, nodes)
3:     nextNodes = GetSuccessors(G, nodes)
4:     newInputBuffer = ReassignInputBuffers(prevNodes)
5:     newOutputBuffer = ReassignOutputBuffers(nextNodes)
6:     newNode = CreateNode(G, nodes, newInputBuffer, newOutputBuffer)   ▷ Create a new node in
   the graph G of the same type as nodes
7:     AssignInAndOutEdges(newNode, prevNodes, nextNodes)
8:     ReplaceNodes(G, nodes, newNode)
```

3.1 Compiler

HiveMind represents DNNs internally as computation DAGs, similar to other modern deep learning computation frameworks such as TensorFlow [1] and PyTorch [13]. Each node in the computation DAG represents a layer (convolution, fully-connected layer, activation) of one of the models in the model batch; each edge represents a data dependency.

HiveMind does not run each model's computation DAG in isolation. Instead, it constructs a combined DAG that contains each constituent model's computation graph; that is, each HiveMind-generated DAG will initially contain one distinct sub-graph for each individual model in the model batch. The compiler then performs several cross-model optimizations to produce a fused DAG. Currently, these optimizations include 1) sharing input preprocessing operations across models, and 2) fusing kernels to amortize away kernel launch overhead and increase computational intensity. We describe each of these optimizations below. All optimizations preserve application semantics *exactly*.

Input I/O and Preprocessing. Our experiments show that for some models, CPU preprocessing (e.g. data augmentation, image decoding) can actually dominate the total execution time of a model. It is possible to pre-compute this step for training workloads, but for large datasets this can require terabytes of storage and is often impractical. Instead, we mitigate the overhead of preprocessing by having multiple models share the same input data pipeline. HiveMind performs this optimization automatically so that the total time spent processing input data is amortized over the batched models.

Cross-model layer fusion. Layer fusion is applicable in three settings: 1) when stateful operators in different models share the same underlying weights (and where gradients for the shared weights across different models can be safely aggregated), 2) when stateful operators share the same inputs and have same output shapes, and 3) when non-stateful operators operate on inputs of the same shape. For 1), HiveMind depends on annotations provided by the user to specify which layers share the same underlying weights. HiveMind can then search the graph for all instances of these relationships to determine the sets of nodes to be fused.

We note that these layer fusion optimizations are complementary to those performed by systems like Latte, TVM, and Halide [3, 12, 15, 19], which try to fuse consecutive layers in the *same* model.

The reasons why these optimizations help performance varies for the above three use cases: for 1) and 2), layer fusion increases the computational intensity of the resulting operations while keeping application semantics *exactly* the same. For 3), layer fusion helps to amortize away the overhead of launching kernels by calling a single kernel instead of multiple kernels.

We describe the algorithm used for fusing stateful operators with shared weights in Algorithm 1. The first step is ensuring that the input and output buffers of the new fused layer are contiguous, as cuDNN's GPU kernel implementations require inputs and outputs to be allocated contiguous memory. Since the input buffer of a layer is shared with the output buffer of the previous layer (to minimize memory transfers), the output buffers of the preceding layers and the input buffers of the succeeding layers need to be contiguous post-fusion, as well. HiveMind pre-allocates all of its input and output buffers during the compilation phase to make this re-mapping process easier to implement.

The second step is ensuring that the DAG representing the computation is accurate post-fusion. This requires verifying that the newly created fused layer has `prevNodes` as its list of predecessors and `nextNodes` as its list of successors. Each node in `prevNodes` and `nextNodes` also needs to update its out- and in- node lists respectively. Layer fusion also modifies some metadata associated with relevant layers, such as the dimensions of the inputs, outputs, or weights. HiveMind's runtime can now simply call a single kernel for the fused layer; for example, if the compiler fused multiple models' convolution layers, then the runtime can now call a single `cudnnConvolveForward` and `cudnnConvolveBackward` method for the forward and backward passes respectively.

The algorithm for fusing non-stateful layers with the same input and output shapes is the same. To fuse layers with the same inputs, we follow a similar procedure, but instead of concatenating the inputs, we concatenate weight tensors instead. Layer fusion provides as much as a $1.6\times$ speedup for relevant workloads, as we show in § 4.2.

3.2 Runtime

The HiveMind runtime is given as input an optimized computation DAG with arbitrary input-output dependencies. To utilize the GPU efficiently, HiveMind's runtime uses CUDA's stream abstraction to execute GPU kernels (functions executed on the GPU) in parallel as much as possible. Prior to execution, the runtime runs a stream assignment algorithm on the input graph, mapping each node in the graph to a CUDA stream. However, nodes assigned different streams might have data dependencies. HiveMind's runtime uses CUDA's event abstraction to enforce these data dependencies, thus ensuring application correctness and preventing data races.

Stream Assignment. Given a DAG, HiveMind's runtime determines a stream assignment for each node in the graph. HiveMind accomplishes this by running a Depth First Search on the graph; all nodes explored can be assigned the same stream until a backtrack is necessary. Every backtrack increases the ID of the next stream assigned by one.

Data Dependencies. DAGs in HiveMind can have nodes with multiple in-edges or out-edges, on account of layers with skip connections, or layer fusion. These layers could be executed on different streams, and thus HiveMind needs to manage these data dependencies. HiveMind uses the CUDA Event API to ensure that kernels are executed only once their inputs are available.

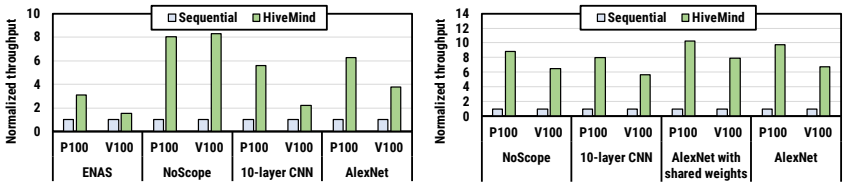
4 Evaluation

We present preliminary results for HiveMind run on end-to-end workloads, and also microbenchmarks that show the impact of individual compiler optimizations. Experiments were run on two machines: 1) a machine in a private cluster with 28 CPU cores and an NVIDIA P100 GPU (henceforth called P100), and 2) a `p3.2xlarge` instance on Amazon EC2 with 8 hyperthreads and an NVIDIA V100 GPU (henceforth called V100). Experiments were run using CUDA 9.0 and CuDNN 7.0.

4.1 End-to-end Results

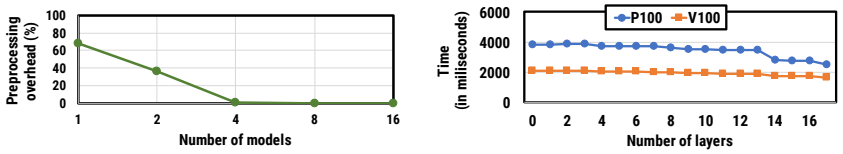
Figure 4a provides the normalized throughputs while training 8 copies of 4 models. The ENAS and NoScope results each use a model sampled from the set of candidate models described in the respective papers. HiveMind delivers speedups ranging from $3.1\times$ to $8\times$ on the P100 and $1.6\times$ to $8\times$ on the V100 compared to sequential execution.

Figure 4b provides the normalized throughputs while performing inference on 4 models. "AlexNet with shared weights" uses 8 AlexNet models with the same weights for the first 6 layers. HiveMind delivers speedups ranging from $8\times$ to $10\times$ on the P100 and $5.7\times$ to $6.7\times$ on the V100.



(a) Hyperparameter Search (training). **(b)** Model Ensemble (inference).

Figure 4: End-to-end results on multi-model training and inference workloads. **Left:** Normalized throughput for hyperparameter search with an input minibatch size of 16. **Right:** Normalized throughput for model ensembles with an input minibatch size of 16.



(a) Shared pre-processing. **(b)** Cross-model layer fusion.

Figure 5: Isolated effects of different compiler optimizations. **Left:** Overhead of preprocessing (computed as the percentage of time *waiting* on data from the preprocessing step) for ResNet-18 training using PyTorch. The number of models sharing preprocessing is varied from 1 to 16, and preprocessing is pipelined with computation. **Right:** Time (in milliseconds) for AlexNet inference as we increase the number of fused layers for 4 models.

4.2 Compiler

Input Preprocessing. Figure 5a shows the overhead of preprocessing (computed as the percentage of time waiting on data from the preprocessing step) while training a ResNet-18 model on ImageNet data (each image is 227×227 pixels) on the V100. Different numbers of models are trained while sharing the same input pipeline. Model computations are executed serially.

We highlight two key results: (a) The preprocessing overhead for a single model is high (67.8%), and (b) Sharing preprocessing among multiple models reduces this overhead drastically (0.19% when preprocessing is shared across 8 models).

Layer Fusion. We show the impact of layer fusion on the AlexNet inference workload across 4 models in Figure 5b. The number of layers fused is varied from 0 to 16; we see that in the best case, layer fusion provides a $1.6\times$ speedup on the P100 and a $1.3\times$ speedup on the V100.

5 Conclusion

Existing deep learning frameworks are optimized for running DNNs in isolation; in this work, we explore the problem of efficently running *multiple* DNNs together on a GPU. Our system, HiveMind, introduces a compiler that performs cross-model optimizations, and a runtime that extracts as much concurrency as possible from the combined model execution graph. Preliminary results show Hivemind is up to $10\times$ faster than sequential model execution for simple multi-model workloads like hyperparameter tuning and model ensembles.

Acknowledgements

We thank James Thomas for his valuable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Intel, Microsoft, NEC, Teradata, SAP, and VMware—as well as DARPA under No. FA8750-17-2-0095 (D3M), industrial gifts and support from Toyota Research Institute, Keysight Technologies, Hitachi, Northrop Grumman, NetApp, and the NSF under grants DGE-1656518 and CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] I. Bello, B. Zoph, V. Vasudevan, and Q. Le. Neural optimizer search with reinforcement learning. 2017.

[3] T. Chen, T. Moreau, Z. Jiang, and H. Shen. Tvm: An end to end ir stack for deploying deep learning workloads on hardware platforms, 2017.

[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

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

[6] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *CoRR*, abs/1404.2188, 2014.

[8] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. Xing. Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*, 2018.

[9] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[11] S. Lee, S. Purushwalkam, M. Cogswell, D. Crandall, and D. Batra. Why m heads are better than one: Training a diverse ensemble of deep networks. *arXiv preprint arXiv:1511.06314*, 2015.

[12] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[14] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Faster discovery of neural architectures by searching for paths in a large model. 2018.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[16] A. Severyn and A. Moschitti. Learning to rank short text pairs with convolutional deep neural networks. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 373–382. ACM, 2015.

[17] 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*, 15(1):1929–1958, 2014.

[18] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[19] L. Truong, R. Barik, E. Totoni, H. Liu, C. Markley, A. Fox, and T. Shpeisman. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks. *ACM SIGPLAN Notices*, 51(6):209–223, 2016.

[20] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, 2018. USENIX Association.

[21] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[22] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.