Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training

Geoffrey X. Yu, University of Toronto/Vector Institute; Yubo Gao, University of Toronto; Pavel Golikov and Gennady Pekhimenko, University of Toronto/Vector Institute

https://www.usenix.org/conference/atc21/presentation/yu
Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training

Geoffrey X. Yu  
University of Toronto  
Vector Institute

Yubo Gao  
University of Toronto

Pavel Golikov  
University of Toronto  
Vector Institute

Gennady Pekhimenko  
University of Toronto  
Vector Institute

Abstract

Deep learning researchers and practitioners usually leverage GPUs to help train their deep neural networks (DNNs) faster. However, choosing which GPU to use is challenging both because (i) there are many options, and (ii) users grapple with competing concerns: maximizing compute performance while minimizing costs. In this work, we present a new practical technique to help users make informed and cost-efficient GPU selections: make performance predictions with the help of a GPU that the user already has. Our technique exploits the observation that, because DNN training consists of repetitive compute steps, predicting the execution time of a single iteration is usually enough to characterize the performance of an entire training process. We make predictions by scaling the execution time of each operation in a training iteration from one GPU to another using either (i) wave scaling, a technique based on a GPU’s execution model, or (ii) pre-trained multilayer perceptrons. We implement our technique into a Python library called Habitat and find that it makes accurate iteration execution time predictions (with an average error of 11.8%) on ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN across six different GPU architectures. Habitat supports PyTorch, is easy to use, and is open source.

1 Introduction

Over the past decade, deep neural networks (DNNs) have seen incredible success across many machine learning tasks [26, 37, 39, 50, 93, 96, 99]—leading them to become widely used throughout academia and industry. However, despite their popularity, DNNs are not always straightforward to use in practice because they can be extremely computationally-expensive to train [23, 53, 95, 109]. This is why, over the past few years, there has been a significant and ongoing effort to bring hardware acceleration to DNN training [10, 16, 35, 36, 45, 78, 80].

As a result of this effort, today there is a vast array of hardware options for deep learning users to choose from for training. These options range from desktop and server-class GPUs (e.g., 2080Ti [70] and A100 [78]) all the way to specialized accelerators such as the TPU [45], AWS Trainium [10], Gaudi [36], IPU [35], and Cerebras WSE [16]. Having all these options offers flexibility to users, but at the same time can also lead to a paradox of choice: which hardware option should a researcher or practitioner use to train their DNNs?

A natural way to start answering this question is to first consider CUDA-enabled GPUs. This is because they (i) are commonly used in deep learning; (ii) are supported by all major deep learning software frameworks (PyTorch [86], TensorFlow [1], and MXNet [19]); (iii) have mature tooling support (e.g., CUPTI [76]); and (iv) are readily available for rent and purchase. In particular, when considering GPUs, we find that there are many situations where a deep learning user needs to choose a specific GPU to use for training:

• Choosing between different hardware tiers. In both academia and industry, deep learning users often have access to several tiers of hardware: (i) a workstation with a GPU used for development (e.g., 2080Ti), (ii) a private GPU cluster that is shared within their organization (e.g., RTX6000 [84]), and (iii) GPUs that they can rent in the cloud (e.g., V100 [66]). Each tier offers a different cost, availability, and performance trade-off. For example, a private cluster might be “free” (in monetary cost) to use, but jobs may be queued because the cluster is also shared among other users. In contrast, cloud GPUs can be rented on-demand for exclusive use.

• Deciding on which GPU to rent or purchase. Cloud providers make many different GPUs available for rent (e.g., P100 [62], V100, T4 [71], and A100 [78]), each with different performance at different prices. Similarly, a wide variety of GPUs are available for purchase (e.g., 2080Ti, 3090 [82]) both individually and as a part of pre-built workstations [52]. These GPUs can vary up to 6× in price [98] and 6× in peak performance [79].

• Determining how to schedule a job in a heterogeneous
GPU cluster. A compute cluster (e.g., operated by a cloud provider [8, 32, 58]) may have multiple types of GPUs available that can handle a training workload. Deciding which GPU to use for a job will typically depend on the job’s priority and performance on the GPU being considered [18, 61].

- Selecting alternative GPUs. When a desired GPU is unavailable (e.g., due to capacity constraints in the cloud), a user may want to select a different GPU with a comparable cost-normalized performance. For example, when training ResNet-50 [37] on Google Cloud [31], we find that both the P100 and V100 have similar cost-normalized throughputs (differing by just 0.8%). If the V100 were to be unavailable, a user may decide to use the P100 instead since the total training cost would be similar.

What makes these situations interesting is that there is not necessarily a single “correct” choice. Users make GPU selections based on whether the performance benefits of the chosen configuration are worth the cost to train their DNNs. But making these selections in an informed way is not easy, as performance depends on many factors simultaneously: (i) the DNN being considered, (ii) the GPU being used, and (iii) the underlying software libraries used during training (e.g., cuDNN [74], cuBLAS [77]).

To do this performance analysis today, the common wisdom is to either (i) directly measure the computational performance (e.g., throughput) by actually running the training job on the GPU, or (ii) consult existing benchmarks (e.g., MLPerf [53]) published by the community to get a “ballpark estimate.” While convenient, these approaches also have their own limitations. Making measurements requires users to already have access to the GPUs they are considering; this may not be the case if a user is deciding whether or not to buy or rent that GPU in the first place. Secondly, benchmarks are usually only available for a subset of GPUs (e.g., the V100 and T4) and only for common “benchmark” models (e.g., ResNet-50 [37] and the Transformer [99]). They are not as helpful if you need an accurate estimate of the performance of a custom DNN on a specific GPU (a common scenario when doing deep learning research).

In this work, we make the case for a third complementary approach: making performance predictions. Although predicting the performance of general compute workloads can be prohibitively difficult due to the large number of possible program phases, we observe that DNN training workloads are special because they contain repetitive computation. DNN training consists of repetitions of the same (relatively short) training iteration, which means that the performance of an entire training process can be characterized by just a few training iterations.

We leverage this observation to build a new technique that predicts a DNN’s training iteration execution time for a given batch size and GPU using both runtime information and hardware characteristics. We make predictions in two steps: (i) we measure the execution time of a training iteration on an existing GPU, and then (ii) we scale the measured execution times of each individual operation onto a different GPU using either wave scaling or pre-trained multilayer perceptrons (MLPs) [29]. Wave scaling is a technique that applies scaling factors to the GPU kernels in an operation, based on a mix of the ratios between the two GPUs’ memory bandwidth and compute units. We use MLPs for certain operations (e.g., convolution) where the kernels used differ between the two GPUs; we describe this phenomenon and the MLPs in more detail in Sections 3.2 and 3.4. We believe that using an existing GPU to make operation execution time predictions for a different GPU is reasonable because deep learning users often already have a local GPU that they use for development.

We implement our technique into a Python library that we call Habitat, and evaluate its prediction accuracy on five DNNs that have applications in image classification, machine translation, and image generation: (i) ResNet-50, (ii) Inception v3 [97] (iii) the Transformer, (iv) GNMT [102], and (v) DCGAN [89]. We use Habitat to make iteration execution time predictions across six different GPUs and find that it makes accurate predictions with an average error of 11.8%. Additionally, we present two case studies to show how Habitat can be used to help users make accurate cost-efficient GPU selections according to their needs (Section 5.3).

We designed Habitat to be easy and practical to use (see Listing 1). Habitat currently supports PyTorch [86] and is open source: github.com/geoffxy/habitat [105, 106].

In summary, this work makes the following contributions:

- Wave scaling: a new technique that scales the execution time of a kernel measured on one GPU to a different GPU by using scaled ratios between the (i) number of compute units on each GPU, and (ii) their memory bandwidths.
- The implementation and evaluation of Habitat: a new library that uses wave scaling along with pre-trained MLPs to predict the execution time of DNN training iterations on different GPUs.
2 Why Predict the Computational Training Performance of DNNs on Different GPUs?

This work presents a new practical technique for predicting the execution time of a DNN training iteration on different GPUs, with the goal of helping deep learning users make informed cost-efficient GPU selections. However, a common first question is to ask why we need to make these performance predictions in the first place. Could other performance comparison approaches (e.g., simple heuristics or measurements) be used instead? In this section, after providing some background about DNN training, we outline the problems with these alternative approaches to further motivate the need for practical performance predictions.

2.1 Background on DNN Training

DNNs, at their heart, are mathematical functions that produce predictions given an input and a set of learned parameters, also known as weights \[29\]. They are built by combining together a series of different layers, each of which may contain weights. The layers map to mathematical operations. For example, a fully connected layer is implemented using matrix multiplication \[29\]. To produce predictions, a DNN takes a tensor (an n-dimensional array) as input and applies the operations associated with each layer in sequence.

**Training.** A DNN learns its weights in an iterative process called training. Each training iteration operates on a batch of labelled inputs and consists of a forward pass, backward pass (using backpropagation \[90\]), and weight update. The forward and backward passes compute gradients for the weights, which are then used by an optimization algorithm (e.g., stochastic gradient descent \[12\] or Adam \[49\]) to update the weights so that the DNN produces better predictions. These steps are repeated until the DNN makes acceptably accurate predictions.

**Computational performance.** Although conceptually simple, prior work has shown that DNN training can be an extremely time-consuming process \[23,53,95,109\]. There are two primary factors that influence the time it takes a DNN to reach an acceptable accuracy during training \[59\]: (i) statistical efficiency, and (ii) hardware efficiency. Statistical efficiency governs the number of training iterations (i.e., weight updates) required to reach a target test accuracy whereas hardware efficiency governs how quickly a training iteration runs. In this work, we focus on helping deep learning users make informed cost-efficient hardware configuration selections to improve their DNN’s hardware efficiency. As a result, we compare the performance of different GPUs when training a DNN using the time it takes a training iteration to run. This metric equivalently captures the training throughput for that particular DNN.

2.2 Why Not Measure Performance Directly?

Perhaps the most straightforward approach to compare the performance of different GPUs is to just measure the iteration execution time (and hence, throughput) on each GPU when training a given DNN. However, this approach also has a straightforward downside: it requires the user to actually have access to the GPU(s) being considered in the first place. If a user is looking to buy or rent a cost-efficient GPU, they would ideally want to know its performance on their DNNs before spending money to get access to the GPU.

2.3 Why Not Apply Heuristics?

Another approach is to use heuristics based on the hardware specifications published by the manufacturer. For example, one could use the ratio between the peak floating point operations per second (FLOPS) of two GPUs or the ratio between the number of CUDA cores on each GPU. The problem with this approach is that these heuristics do not always work. Heuristics often assume that a DNN training workload can be repeated, and then use this measurement to predict the iteration execution time on each GPU. However, this approach is unreliable in general \[109\].

To show an example of when simple heuristics do not work well, we use a GPU’s peak FLOPS to make iteration execution time predictions. We measure the execution time of a DCGAN training iteration on the T4 and then use this measurement to predict the iteration execution time on different GPUs by multiplying by the ratio between the devices’ peak FLOPS. Figure 1 shows the measured and predicted execution times on each GPU, along with the prediction error as a percentage. The main takeaway from this figure is that using simple heuristics can lead to high prediction errors; the highest prediction error in this experiment is 64.9%, and all the prediction errors are at least 42.5%. In contrast, Habitat can make these exact same predictions with an average error of 10.2% (maximum 21.8%).

---

\[3\] We use a batch size of 128 LSUN \[104\] synthetic inputs. See Section 5.1 for details about our methodology.
2.4 Why Not Use Benchmarks?

A third potential approach is to consult published benchmarking results [23, 53, 81, 109]. However, the problem with relying on benchmarking results is that they are limited to a set of “common” DNNs (e.g., ResNet-50 or BERT [26]) and are usually only available for a small selection of GPUs (e.g., the T4, V100, and A100). Moreover, benchmarking results also vary widely among different models and GPUs [53, 81, 109]. Therefore if no results exist for the GPU(s) a user is considering, or if a user is working with a new DNN architecture, there will be no benchmark results for them to consult.

2.5 Why Not Always Use the “Best” GPU?

Finally, a fourth approach is to always use the most “powerful” GPU available with the assumption that GPUs are already priced based on their performance. Why make performance predictions when the cost-efficiency of popular GPUs should be the same? However, this assumption is a misconception. Prior work has already shown examples where the performance benefits of different GPUs changes depending on the model [18, 61, 109]. In this work, we also show additional examples in our case studies (Section 5.3) where (i) cost-efficiency leads to selecting a different GPU, and (ii) where the V100 does not offer significant performance benefits over a common desktop-class GPU (the 2080Ti).

Summary. Straightforward approaches that users might consider to make GPU selections all have their own downsides. In particular, existing approaches either require access to the GPUs themselves or are only applicable for common DNNs and GPUs. Therefore there is a need for a complementary approach: making performance predictions—something that we explore in this work.

3 Habitat

Our approach to performance predictions is powered by three key observations. In this section, after describing these observations, we outline the key ideas behind Habitat.

3.1 Key Observations

**Observation 1: Repetitive computation.** While training a DNN to an acceptable accuracy can take on the order of hours to days [23, 53, 109], a single training iteration takes on the order of hundreds of milliseconds. This observation improves the predictability of DNN training as we can characterize the performance of an entire DNN training session using the performance of a single iteration.

**Observation 2: Common building blocks among DNNs.** Although DNNs can consist of hundreds of operations, they are built using a relatively small set of unique operations. For example, convolutional neural networks typically comprise convolutional, pooling, fully connected, and batch normalization [42] layers. This observation reduces the problem of predicting the performance of an arbitrary DNN’s training iteration to developing prediction mechanisms for a small set of operations.

**Observation 3: Runtime information available.** When developing DNNs, users often have a GPU available for use in their workstations. These GPUs are used for development purposes and are not necessarily chosen for the highest performance (e.g., 1080Ti [64], TITAN Xp [68]). However, they can be used to provide valuable runtime information about the GPU kernels that are used to implement a given DNN. In Section 3.3, we describe how we can leverage this runtime information to predict the performance of the GPU kernels on different GPUs (e.g., from a desktop-class GPU such as the 2080Ti [70] to a server-class GPU such as the V100 [66, 67]).

3.2 Habitat Overview

Habitat records information at runtime about a DNN training iteration for a specific batch size on a given GPU (Observation 3) and then uses that information to predict the training iteration execution time on a different GPU (for the same batch size). Predicting the iteration execution time is enough (Observation 1) to compute metrics about the entire training process on different GPUs. These predicted metrics, such as the training throughput and cost-normalized throughput, are then used by end-users (e.g., deep learning researchers) to make informed hardware selections.

To actually make these predictions for a different GPU, Habitat predicts the new execution time of each individual operation in a training iteration. Habitat then adds these predicted times together to arrive at an execution time prediction for the entire iteration. For an individual operation, Habitat makes predictions using either (i) wave scaling (Section 3.3), or (ii) pre-trained MLPs (Section 3.4).

The reason why Habitat uses two techniques together is that wave scaling assumes that the same GPU kernels are used to implement a given DNN operation on each GPU. However, some DNN operations are implemented using different GPU kernels on different GPUs (e.g., convolutions, recurrent layers). This is done for performance reasons as these operations are typically implemented using proprietary kernel libraries that leverage GPU architecture-specific kernels (e.g., cuDNN [21], cuBLAS [77]). We refer to these operations as kernel-varying, and scale their execution times to different GPUs using pre-trained MLPs. Habitat uses wave scaling for the rest of the operations, which we call kernel-alike.

3.3 Wave Scaling

Wave scaling works by scaling the execution times of the kernels used to implement a kernel-alike DNN operation. The
computation performed by a GPU kernel is partitioned into groups of threads called thread blocks [28], which typically execute in concurrent groups, resulting in waves of execution. The key idea behind wave scaling is to compute the number of thread block waves in a kernel and scale the wave execution time using ratios between the origin and destination GPUs.

We describe wave scaling formally in Equation 1. Let \( T_i \) represent the execution time of the kernel on GPU \( i \), \( B \) the number of thread blocks in the kernel, \( W_i \) the number of thread blocks in a wave on GPU \( i \), \( D_i \) the memory bandwidth on GPU \( i \), and \( C_i \) the clock frequency on GPU \( i \). Here we let \( i \in \{ o, d \} \) to represent the origin and destination GPUs. By measuring \( T_o \) (Observation 3), wave scaling predicts \( T_d \) using

\[
T_d = \left[ \frac{B}{W_d} \right] \left( \frac{D_o}{D_d} \right) \left( \frac{W_o}{W_d} \right)^\gamma \left( \frac{C_o}{C_d} \right)^{1-\gamma} \left[ \frac{B}{W_o} \right]^{-1} T_o
\]

where \( \gamma \in [0, 1] \) represents the “memory bandwidth boundedness” of the kernel. Habitat selects \( \gamma \) by measuring the kernel’s arithmetic intensity and then leveraging the roofline model [101] (see Section 4.2).

As shown in Equation 1, wave scaling uses the ratios between the GPUs’ (i) memory bandwidths, (ii) clock frequencies, and (iii) the size of a wave on each GPU. The intuition behind factors (i) and (iii) is that a higher relative memory bandwidth allows more memory requests to be served in parallel whereas having more thread blocks in a wave results in more memory requests being made. Thus, everything else held constant, waves in memory bandwidth bound kernels (i.e., large \( \gamma \)) should see speedups on GPUs with more memory bandwidth. The intuition behind factor (ii) is that higher clock frequencies may benefit waves in compute bound kernels (i.e., small \( \gamma \)).

For large \( \frac{B}{W_i} \) (i.e., when there are a large number of waves) we get that \( \frac{B}{W_i} \approx B/W_i \). In this case, Equation 1 simplifies to

\[
T_d = \left( \frac{D_o}{D_d} \right) \gamma \left( \frac{W_o}{W_d} \right)^{1-\gamma} \left( \frac{C_o}{C_d} \right)^{1-\gamma} T_o
\]

Habitat uses Equation 2 to predict kernel execution times because we find that in practice, most kernels are composed of many thread blocks.

Habitat computes \( W_i \) for each kernel and GPU using the thread block occupancy calculator that is provided as part of the CUDA Toolkit [80]. We obtain \( C_i \) from each GPU’s specifications, and we obtain \( D_i \) by measuring the achieved bandwidth on each GPU ahead of time. Note that we make these measurements once and then distribute them in a configuration file with Habitat.

### 3.4 MLP Predictors

To handle kernel-varying operations, Habitat uses pre-trained MLPs to make execution time predictions. We treat this prediction task as a regression problem: given a series of input features about the operation and a target GPU (described below), predict the operation’s execution time on that target GPU. We learn an MLP for each kernel-varying operation that Habitat currently supports: (i) convolutions (2-dimensional), (ii) LSTMs [38], (iii) batched matrix multiplies, and (iv) linear layers (matrix multiply with an optional bias term). As we show in Section 5, relatively few DNN operations are kernel-varying and so training separate MLPs for each of these operations is a feasible approach. Furthermore, these MLPs can be used for many different DNNs as these operations are common “building blocks” used in DNNs (Observation 2).

**Input features.** Each operation-specific MLP takes as input: (i) layer dimensions (e.g., the number of input and output channels in a convolution); (ii) the memory capacity and bandwidth on the target GPU; (iii) the number of streaming multiprocessors (SMs) on the target GPU; and (iv) the peak FLOPS of the target GPU, specified by the manufacturer.

**Model architecture.** Each MLP comprises an input layer, eight hidden layers, and an output layer that produces a single real number—the predicted execution time (this includes the forward and backward pass) for the MLP’s associated operation. We use ReLU activation functions in each layer and we use 1024 units in each hidden layer. We outline the details behind our datasets and how these MLPs are trained in Section 4.3.

### 4 Implementation Details

Habitat is built to work with PyTorch [86]. However, the ideas behind Habitat are general and straightforward to implement in other frameworks as well. Habitat performs its analysis using a DNN’s computation graph, which is also available in other frameworks (e.g., TensorFlow [1] and MXNet [19]).

### 4.1 Extracting Runtime Metadata

Habitat extracts runtime metadata in a training iteration by “monkey patching” PyTorch operations with special wrappers. These wrappers allow Habitat to intercept and keep track of all the operations that run in one training iteration, as they are executed. As shown in Listing 1, users explicitly indicate to Habitat when to start and stop tracking the operations in a DNN by calling `track()`.

**Execution time.** To measure the execution time of each operation, Habitat re-runs each operation independently with

---

\(^4\)The clock’s impact on execution time depends on other factors too (e.g., the GPU’s instruction set architecture). Wave scaling aims to be a simple and understandable model and therefore does not model these complex effects.

\(^5\)This is, in part, because implementing performant architecture-specific kernels for each kernel-varying operation takes significant engineering effort.
the same inputs as recorded when the operation was intercepted. Habitat also measures the execution time associated with the operation’s backward pass, if applicable. The reason why Habitat makes measurements by re-running the individual operations is that the operations could be very short (in execution time). Thus, Habitat needs to run them multiple times to make accurate measurements. Habitat uses CUDA events [73] to make these timing measurements.

Kernel metadata and metrics. Habitat uses CUPTI [76] to record execution times and metrics (see Section 4.2) for the kernels used to implement each operation in the DNN. This information is used by wave scaling.

4.2 Selecting Gamma (γ)

Recall from Section 3.3 that wave scaling scales its ratios using γ, a factor that represents the “memory bandwidth boundedness” of a kernel. In this section, we describe in more detail how Habitat automatically selects γ for each kernel.

Roofline model. Habitat uses the roofline model [101] to estimate a kernel’s memory boundedness. Figure 2 shows an example roofline model. The roofline model introduces the notion of a kernel’s arithmetic intensity: the number of floating point operations it performs per byte of data read or written to memory (represented by x in Figure 2).

A key idea behind the roofline model is that it models a kernel’s peak performance as the minimum of either the hardware’s peak performance (P) or the hardware’s memory bandwidth times the kernel’s arithmetic intensity (D · x) [101]. This minimum is shown by the solid line in Figure 2. The arithmetic intensity where these two limits meet is called the “ridge point” (R), where R = P/D. The model considers a kernel with an arithmetic intensity of x to be memory bandwidth bound if x < R and compute bound otherwise. For example, in Figure 2, a kernel with an arithmetic intensity of x1 would be considered memory bandwidth bound whereas a kernel with an intensity of x2 would be considered compute bound.

Wave scaling leverages the observation that a kernel’s arithmetic intensity is fixed across GPUs (i.e., arithmetic intensity only depends on the kernel’s code). R changes across GPUs because P and D vary among GPUs, but can be computed using a GPU’s performance specifications. Therefore, if Habitat computes a kernel’s arithmetic intensity, it can use the arithmetic intensity’s distance from the destination GPU’s ridge point to estimate the kernel’s memory bandwidth boundedness (on the destination GPU).

Selecting γ. When profiling each kernel, Habitat gathers metrics that allow it to empirically calculate the kernel’s arithmetic intensity (floating point efficiency, number of bytes read and written to DRAM). If we let x be the kernel’s measured arithmetic intensity and R = P/D for the destination GPU (using the notation presented above), Habitat sets γ using

$$
γ = \begin{cases} 
-0.5/R \times x + 1 & \text{if } x < R \\
0.5R/x & \text{otherwise}
\end{cases}
$$

This equation means that γ decreases linearly from 1 to 0.5 as x increases toward R. After passing R, γ approaches 0 as x approaches infinity.

Practical optimizations. In practice, gathering metrics on GPUs is a slow process because the kernels need to be replayed multiple times to capture all the needed performance counters. To address this challenge, we make two optimizations: (i) we cache measured metrics, keyed by the kernel’s name and its launch configuration (number of thread blocks and block size); and (ii) we only measure metrics for operations that contribute significantly to the training iteration’s execution time (e.g., with execution times at or above the 99.5th percentile). Consequently, when metrics are unavailable for a particular kernel, we set γ = 1. We believe that this is a reasonable approximation because kernel-alike operations tend to be very simple (e.g., element-wise operations) and are therefore usually memory bandwidth bound.

4.3 MLPs: Data and Training

In this section, we describe the details behind Habitat’s MLPs: how we (i) collect training data, (ii) preprocess the data, and (iii) train the MLPs.

4.3.1 Data Collection

We gather training data by measuring the forward and backward pass execution times of kernel-varying operations at randomly sampled input configurations. An input configuration is a setting of an operation’s parameters (e.g., batch size and number of channels in a convolution). We use predefined ranges for each operation’s parameters, as described in more detail below, and ignore any configurations that result in running out of memory. We make these measurements for all six of the GPUs listed in Section 5.1. We use the same seed when sampling on different GPUs to ensure we have measurements for the same random input configurations across all the GPUs.
We multiply by six because we make measurements on six different GPUs.

**2D convolutions.** For convolutions, we vary the (i) batch size (1 – 64), (ii) number of input features (3 – 2048) and output channels (16 – 2048), (iii) kernel size (1 – 11), (iv) padding (0 – 3), (v) stride (1 – 4), (vi) image size (1 – 256), and (vii) whether or not there is a bias weight. We only sample configurations with square images and kernel sizes. During sampling, we ignore any configurations that result in invalid arguments (e.g., a kernel size larger than the image). We selected these parameter ranges by surveying the convolutional neural networks included in PyTorch’s torchvision package [24].

**LSTMs.** For LSTMs, we vary the (i) batch size (1 – 128), (ii) number of input features (1 – 1280), (iii) number of hidden features (1 – 1280), (iv) sequence length (1 – 64), (v) number of stacked layers (1 – 6), (vi) whether or not the LSTM is bidirectional, and (vii) whether or not there is a bias weight.

**Batched matrix multiply (bmm).** For a batched matrix multiply \(A \times B\) where \(A \in \mathbb{R}^{n \times l \times m}\) and \(B \in \mathbb{R}^{n \times m \times r}\), we vary the (i) batch size \(n\) (1 – 128), and (ii) the \(l, m,\) and \(r\) dimensions (1 – 1024).

**Linear layers.** For linear layers, we vary the (i) batch size (1 – 3500), (ii) input features (1 – 32768), (iii) output features (1 – 32768), and (iv) whether or not there is a bias weight.

### 4.3.2 Data Preprocessing

After collecting data on the GPUs, we build one dataset per operation by (i) adding the forward and backward execution times to arrive at a single execution time for each operation instance on a particular GPU, and (ii) attaching additional GPU hardware features to each of these data points. We attach the GPU’s (i) memory capacity and bandwidth; (ii) number of streaming multiprocessors (SMs); and (iii) peak FLOPS, as specified by the GPU manufacturer.

We present the characteristics of the final datasets in Table 1. We add four to the number of features to account for the four GPU features (described above) that we add to each data point. Similarly, in the dataset size column we show the total number of unique operation configurations that we sample. We multiply by six because we make measurements on six different GPUs.

### 4.3.3 Training

We implement our MLPs using PyTorch. We train each MLP for 80 epochs using the Adam optimizer [49] with a learning rate of \(5 \times 10^{-4}\), weight decay of \(10^{-4}\), and a batch size of 512 samples. We reduce the learning rate to \(10^{-4}\) after 40 epochs. We use the mean absolute percentage error as our loss function:

\[
L = \frac{1}{n} \sum_{i=1}^{n} \left| \text{predicted}_i - \text{measured}_i \right|
\]

We assign 80% of our data samples to the training set and the rest to our test set. None of the configurations that we test on in Section 5 appear in our training sets. We normalize the inputs by subtracting the mean and dividing by the standard deviation of the input features in our training set.

## 5 Evaluation

Habitat is meant to be used by deep learning researchers and practitioners to predict the potential compute performance of a given GPU so that they can make informed cost-efficient choices when selecting GPUs for training. Consequently, in our evaluation our goals are to determine (i) how accurately Habitat can predict the training iteration execution time on GPUs with different architectures, and (ii) whether Habitat can correctly predict the relative cost-efficiency of different GPUs when used to train a given model. Overall, we find that Habitat makes iteration execution time predictions across pairs of six different GPUs with an average error of 11.8% on ResNet-50 [37], Inception v3 [97], the Transformer [99], GNMT [102], and DCGAN [89].

### 5.1 Methodology

**Hardware.** In our experiments, we use the GPUs listed in Table 2. For the P4000, 2070, and 2080Ti we use machines whose configurations are listed in Table 3. For the T4 and V100, we use g4dn.xlarge and p3.2xlarge instances on AWS respectively [7]. For the P100, we use Google Cloud’s V100 [66] Volta [67] 8 GB HBM2 80 $2.48/hr

Google Cloud pricing in us-central1, as of June 2021.

### Table 1: A summary of the datasets used for our MLPs.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Features</th>
<th>Dataset Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Convolution</td>
<td>7 + 4</td>
<td>91,138 × 6</td>
</tr>
<tr>
<td>LSTM</td>
<td>7 + 4</td>
<td>124,176 × 6</td>
</tr>
<tr>
<td>Batched Matrix Multiply</td>
<td>4 + 4</td>
<td>131,022 × 6</td>
</tr>
<tr>
<td>Linear Layer</td>
<td>4 + 4</td>
<td>155,596 × 6</td>
</tr>
</tbody>
</table>

### Table 2: The GPUs we use in our evaluation.

<table>
<thead>
<tr>
<th>GPU</th>
<th>Generation</th>
<th>Mem.</th>
<th>Mem. Type</th>
<th>SMs</th>
<th>Rental Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4000 [65]</td>
<td>Pascal [63]</td>
<td>8 GB</td>
<td>GDDR5 [56]</td>
<td>14</td>
<td>$1.46/hr</td>
</tr>
<tr>
<td>P100 [62]</td>
<td></td>
<td>16 GB</td>
<td>HBM2 [4]</td>
<td>56</td>
<td>$1.46/hr</td>
</tr>
<tr>
<td>V100 [66]</td>
<td></td>
<td>16 GB</td>
<td>HBM2</td>
<td>80</td>
<td>$2.48/hr</td>
</tr>
<tr>
<td>2070 [69]</td>
<td></td>
<td>8 GB</td>
<td>GDDR6 [57]</td>
<td>36</td>
<td>$1.46/hr</td>
</tr>
<tr>
<td>2080Ti [70]</td>
<td></td>
<td>11 GB</td>
<td>GDDR6</td>
<td>68</td>
<td>$1.46/hr</td>
</tr>
<tr>
<td>T4 [71]</td>
<td></td>
<td>16 GB</td>
<td>GDDR6</td>
<td>40</td>
<td>$2.48/hr</td>
</tr>
</tbody>
</table>

We assign 80% of our data samples to the training set and the rest to our test set. None of the configurations that we test on in Section 5 appear in our training sets. We normalize the inputs by subtracting the mean and dividing by the standard deviation of the input features in our training set.
Table 3: The machines we use in our evaluation.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Freq.</th>
<th>Cores</th>
<th>Main Mem.</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon E5-2680 v4</td>
<td>2.4 GHz</td>
<td>14</td>
<td>128 GB</td>
<td>P4000</td>
</tr>
<tr>
<td>Ryzen TR 1950X</td>
<td>3.4 GHz</td>
<td>16</td>
<td>16 GB</td>
<td>2070</td>
</tr>
<tr>
<td>EPYC 7371</td>
<td>3.1 GHz</td>
<td>16</td>
<td>128 GB</td>
<td>2080Ti</td>
</tr>
</tbody>
</table>

Table 4: The DNNs and training configurations we use.

<table>
<thead>
<tr>
<th>Application</th>
<th>Model</th>
<th>Arch. Type</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception v3</td>
<td>[97]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformer</td>
<td>[99]</td>
<td>Attention</td>
<td>(EN-DE)</td>
</tr>
<tr>
<td>Image Gen.</td>
<td>DCGAN [89]</td>
<td>Convolution</td>
<td>LSUN [104]</td>
</tr>
</tbody>
</table>

5.2 How Accurate are Habitat’s Predictions?

To evaluate Habitat’s prediction accuracy, we use it to make training iteration execution time predictions for ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN on all six GPUs listed in Section 5.1. Recall that Habitat makes execution time predictions by scaling the execution time of a model and specific batch size measured on one GPU (the “origin” GPU) to another (the “destination” GPU). As a result, we use all 30 possible (origin, destination) pairs of these six GPUs in our evaluation.

5.2.1 End-to-End Prediction Accuracy

Figure 3 shows Habitat’s prediction errors for these aforementioned end-to-end predictions. Each subfigure shows the predictions for all five models on a specific destination GPU. We make predictions for three different batch sizes (shown on the figures) and plot both the predicted and measured iteration execution times. Since we consider all possible pairs of our six GPUs, for each destination GPU we plot the average predicted execution times among the five origin GPUs. Similarly, we show the average prediction error above each bar. From these figures, we can draw three major conclusions.

First, Habitat makes accurate end-to-end iteration execution time predictions since the average prediction error across all GPUs and models is 11.8%. The average prediction error across all ResNet-50, Inception v3, Transformer, GNMT, and DCGAN configurations are 13.4%, 9.5%, 12.6%, 11.2%, and 12.3% respectively.

Second, Habitat can predict the iteration execution time across GPU generations, which have different architectures, and across classes of GPUs. The GPUs we use span three generations (Pascal [63], Volta [67], and Turing [72]) and include desktop, professional workstation, and server-class GPUs.

Third, Habitat is general since it supports different types of DNN architectures. Habitat works with convolutional neural networks (e.g., ResNet-50, Inception v3, DCGAN), recurrent neural networks (e.g., GNMT), and other neural network architectures such as the attention-based Transformer. In particular, Habitat makes accurate predictions for ResNet, Inception, and DCGAN despite the significant differences in their architectures; ResNet has a “straight-line” computational graph, Inception has a large “fanout” in its graph, and DCGAN is a generative-adversarial model.

5.2.2 Prediction Error Breakdown

Figure 4 shows a breakdown of the prediction errors for the execution time of individual operations, which are listed on the x-axis. The operations predicted using the MLP predictors are shown on the left (conv2d, lstm, bmm, and linear). Wave scaling is used to predict the rest of the operations. Above each bar, we also show the importance of each operation as...
a percentage of the iteration execution time, averaged across all five DNNs. The prediction errors are averaged among all pairs of the six GPUs that we evaluate and among ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN. From this figure, we can draw two major conclusions.

First, MLP predictors can be used to make accurate predictions for kernel-varying operations as the average error among the conv2d, lstm, bmm, and linear operations is 18.0%. Second, wave scaling can make accurate predictions for important operations; the average error for wave scaling predictions is 29.8%. Although wave scaling’s predictions for some operations (e.g., __add__, __scatter) have high errors, these operations do not make up a significant proportion of the training iteration execution time (having an overall importance of at most 0.3%).

5.2.3 Prediction Contribution Breakdown

We also examine how wave scaling and the MLPs each contribute to making Habitat’s end-to-end predictions. In our evaluation, Habitat uses wave scaling for 95% of the unique operations; it uses MLPs for the other 5%. In contrast, when looking at execution time, Habitat uses wave scaling to predict 46% of an iteration’s execution time on average; it uses MLPs for the other 54%.

These breakdowns show that both wave scaling and the MLPs contribute non-trivially to Habitat’s predictions—each is responsible for roughly half of an iteration’s execution time. Additionally, the unique operation breakdown shows that most operations are predicted using wave scaling. This observation highlights a strength of Habitat’s hybrid approach of using both wave scaling and MLPs: most operations can be automatically predicted using wave scaling; MLPs only need to be trained for a few kernel-varying operations.

5.2.4 MLPs: How Many Layers?

In all our MLPs, we use eight hidden layers, each of size 1024. To better understand how the number of layers affects the MLPs’ prediction accuracy, we also conduct a sensitivity
In the following two case studies, we demonstrate how Habitat can make cost-efficiency predictions that empower users to make correct selections according to their needs.

5.3.1 Case Study 1: Should I Rent a Cloud GPU?

As mentioned in Section 1, one scenario a deep learning user may face is deciding whether to rent GPUs in the cloud for training or to stick with a GPU they already have locally (e.g., in their desktop). For example, suppose a user has a P4000 in their workstation and they want to decide whether to rent a P100, T4, or V100 in the cloud to train GNMT.

With Habitat, they can use their P4000 to make predictions about the computational performance of each cloud GPU to help them make this decision in an informed way. Figure 6a shows Habitat’s throughput predictions for GNMT for the P100, T4, and V100 normalized to the training throughput on the P4000. Additionally, Figure 6b shows Habitat’s predicted training throughputs normalized by each cloud GPU’s rental costs on Google Cloud as shown in Table 2. Note that (i) we make all these predictions with the P4000 as the origin device, (ii) we make our ground truth measurements on Google Cloud instances, and (iii) one can also use Habitat for a similar analysis for other cloud providers. From these results, the user can make two observations.

First, both the P100 and V100 offer training throughput speedups over the P4000 (up to 2.3× and 4.0× respectively) whereas the T4 offers marginal throughput speedups (up to 1.4×). However, second, the user would also discover that the T4 is more cost-efficient to rent when compared to the P100 and V100 as it has a higher cost-normalized throughput. Therefore, if the user wanted to optimize for maximum computational performance, they would likely choose the V100. But if they were not critically constrained by time and wanted to optimize for cost, sticking with the P4000 or renting a T4 would be a better choice.

Habitat makes these predictions accurately, with an average error of 10.7%. We also note that despite any prediction errors, Habitat still correctly predicts the relative ordering of these three GPUs in terms of their throughput and cost-normalized...
For example, in Figure 6b, Habitat correctly predicts that the T4 offers the best cost-normalized throughput on all three batch sizes. These predictions therefore allow users to make correct decisions based on their needs (optimizing for cost or pure performance).

5.3.2 Case Study 2: Is the V100 Always Better?

In the previous case study, Habitat correctly predicts that the V100 provides the best performance despite not being the most cost-efficient to rent. This conclusion may lead a naive user to believe that the V100 always provides better training throughput over other GPUs, given that it is the most advanced and expensive GPU available in the cloud to rent. In this case study, we show how Habitat can help a user recognize when the V100 does not offer significant performance benefits for their model.

Suppose a user wants to train DCGAN and already has a 2080Ti that they can use. They want to find out if they should use a different GPU to get better computational performance (training throughput). They can use Habitat to predict the training throughput on other GPUs. Figure 7 shows Habitat’s throughput predictions along with the measured throughput, normalized to the 2080Ti’s training throughput. Note that we use a batch size of 64 as it is the default batch size in the DCGAN reference implementation [22] and 128 because it is the size reported by the authors in their paper [89].

From this figure, the user would conclude that they should stick to using their 2080Ti as the V100 would not be worth renting. The V100 offers marginal throughput improvements over the 2080Ti (1.1 ×) while the P100, P4000, 2070, and T4 all do not offer throughput improvements at all. The reason the V100 does not offer any significant benefits over the 2080Ti despite having more computational resources (Table 2) is that DCGAN is a “computationally lighter” model compared to GNMT and so it does not really benefit from a more powerful GPU. Habitat makes these predictions accurately, with an average error of 7.7%.

Summary. These case studies show examples of situations where (i) the GPU offering the highest training throughput is not the same as the most cost-efficient GPU, and where (ii) the V100 does not offer significantly better performance when compared to a desktop-class GPU (the 2080Ti). Notably, in both case studies, Habitat correctly predicts each of these findings. As a result, deep learning researchers and practitioners can rely on Habitat to help them make correct cost-efficient GPU selections according to their needs.

6 Discussion

In this section, we discuss how Habitat can be extended to support additional (i) training setups, and (ii) deep learning frameworks. In doing so, we also highlight opportunities for future work and describe some challenges and opportunities associated with supporting additional hardware accelerators.

6.1 Additional Training Setups

Habitat is designed to make accurate cross-GPU execution time predictions for DNN training. However, users may also face situations where they need performance predictions for more complex training setups such as (i) distributed training [25], (ii) mixed precision training [55], or (iii) needing
predictions for batch sizes larger than what can fit on the origin GPU. In this subsection, we outline how Habitat could be extended to support these setups.

6.1.1 Distributed Training

Predicting the execution time of a distributed training iteration generally reduces to predicting (i) the computation time on the cluster’s GPUs, (ii) the communication time among the GPUs and/or nodes, and (iii) how the communication overlaps with the computation. For data parallel training [25], several prior works present techniques for predicting the data parallel iteration execution time given the execution time on a single GPU [87, 88, 110] (i.e., tasks (ii) and (iii)). Habitat’s computation predictions (task (i)) could be used as an input to these existing techniques.

For more complex distribution schemes such as model parallel [25] and pipeline parallel training [40, 60], Habitat could still be used for task (i), but the user would need to split up their model based on the distributed partitioning scheme before profiling it with Habitat. However, for tasks (ii) and (iii), new prediction techniques would need to be developed. This is something we leave to future work.

6.1.2 Mixed Precision Training

The Daydream paper [110] presents a technique for predicting the performance benefits of switching from single to mixed precision training on the same fixed GPU. If users want to know about the performance benefits of mixed precision training on a different GPU, they can use Daydream’s technique in conjunction with Habitat.

To show that this combined approach can work in practice, we use a P4000 to predict the execution time of a ResNet-50 mixed precision training iteration on the 2070 and 2080Ti. On the P4000, we first use Habitat to predict the single precision iteration execution time on the 2070 and 2080Ti. Then, we apply Daydream’s technique to translate these predicted single precision execution times into mixed precision execution times. We also repeat this experiment between the 2070 and 2080Ti. Overall, we find that this approach has an average error of 16.1% for predictions onto the 2070 and 2080Ti.

To distinguish between the errors introduced by Habitat versus Daydream, we also apply Daydream’s technique to the measured (i.e., ground truth) single precision iteration execution times. We find that Daydream’s technique alone has an average error of 10.7% for the 2070 and 2080Ti. Thus we believe the additional error introduced by also using Habitat is reasonable, given the extra functionality. So overall, we conclude that Habitat with Daydream should be able to effectively support mixed precision predictions on other GPUs.

6.1.3 Larger Batch Sizes

Recall that Habitat’s iteration execution time predictions are for a model and a specific batch size. This means that the origin GPU must be able to run a training iteration with the desired batch size (for Habitat’s profiling pass).

One potential approach to making predictions for batch sizes larger than what can run on the origin GPU is as follows. First, use Habitat to make iteration execution time predictions for multiple (e.g., three) different batch sizes that do fit on the origin GPU. Then, build a linear regression model on these predicted values to extrapolate to larger batch sizes. This approach is based on our prior work, where we observed an often linear relationship between the iteration execution time and batch size [107]. We leave the handling of models where only one batch size fits on the origin GPU to future work.

6.2 Additional Deep Learning Frameworks

Recall that Habitat predicts the execution time of operations using either (i) wave scaling or (ii) pre-trained MLPs, depending on whether the operation is kernel-alyke or kernel-varying. Therefore, as long as Habitat has information about a DNN’s operations and their parameters (e.g., batch size, number of channels), Habitat will be able to apply its techniques to make execution time predictions for a different GPU. Ultimately, this means that adding support for other deep learning frameworks (e.g., TensorFlow or MXNet) boils down to extracting the underlying operations that run during a training iteration and sending the operations to Habitat (i.e., extracting the computation graph). Since the other major deep learning frameworks (TensorFlow and MXNet) both already use computation graphs internally [1, 19], we believe that adding support for them would be straightforward to implement.

6.3 Additional Hardware Accelerators

As described in Section 1, there are also other hardware options available beyond GPUs that can be used for training (e.g., the TPU [45], AWS Trainium [10], and Gaudi [36]). Therefore, a natural opportunity for future work is to explore execution time predictions for these other hardware accelerators. We outline two challenges that arise when going beyond GPUs, as well as two examples of ways that Habitat’s guiding principles can be applied to these prediction tasks.

**Challenges.** First, specialized deep learning accelerators may have a different hardware architecture when compared to GPUs—necessitating different performance modeling techniques. For example, the TPU uses a systolic array [14, 45] whereas GPUs are general-purpose SIMT processors [85]. Second, accelerators such as the TPU rely on tensor compilers (e.g., XLA [34] or JAX [13]) to produce executable code from the high-level DNN model code written by an end-user. The compiler may apply optimizations that change the oper-
ations. These changes make the high-level operation-based analysis that Habitat performs more difficult to realize.

**Opportunities.** Despite these challenges, we believe that there are also opportunities to apply Habitat’s key idea of leveraging runtime-based information from one accelerator to predict the execution time on a different accelerator. For example, as of June 2021, Google makes two versions of the TPU available for rent (v2 and v3) [33] and has announced the v4 [51]. Execution times measured on the TPU v2 could potentially be used to make execution time predictions on the v3 and v4 and vice-versa. Similarly, assuming that the AWS Trainium also uses a systolic array, it may also be possible to leverage execution time measurements on the TPU to make execution time predictions for the Trainium and vice-versa.

7 Related Work

The key difference between Habitat and existing DNN performance modeling techniques for GPUs [46, 87, 88] is in how Habitat makes execution time predictions. Habitat takes a hybrid *runtime-based approach*: it uses information recorded at runtime on one GPU along with hardware characteristics to *scale* the measured kernel execution times onto different GPUs through either (i) wave scaling, or (ii) pre-trained MLPs. In contrast, existing techniques use analytical models [87, 88] or rely entirely on machine learning techniques [46]. The key advantage of Habitat’s hybrid scaling approach is that wave scaling works “out of the box” for all kernel-alike operations (i.e., operations implemented using the same kernels on different GPUs). Ultimately, this advantage means that new analytical or machine learning models do not have to be developed each time a new kernel-alike operation is introduced.

**DNN performance models for different hardware.** There exists prior work on performance models for DNN training on GPUs [46, 87, 88], CPUs [100], and TPUs [47]. As described above, Habitat is fundamentally different from these works because it takes a hybrid *runtime-based approach* when making execution time predictions. For example, Paleo [88] (i) makes DNN operation execution time predictions using *analytical models* based on the number of floating point operations (FLOPs) in a DNN operation, and (ii) uses heuristics to select the kernels used to implement kernel-varying operations (e.g., convolution). However, an operation’s execution time is not solely determined by its number of FLOPs, and using heuristics to select an analytical model cannot always capture kernel-varying operations correctly. This is because proprietary closed-source kernel libraries (e.g., cuDNN [21, 74], cuBLAS [77]) may select different kernel(s) to use by running benchmarks on the target GPU [44, 75].

**Performance models for compilers.** A complementary body of work on performance modeling is motivated by the needs of compilers: predicting how different implementations of some high-level functionality perform on the same hardware. These models were developed to aid in compiling high-performance (i) graphics pipelines [2], (ii) CPU code [54], and (iii) tensor operators for deep learning accelerators [20, 47]. These models have fundamentally different goals compared to Habitat, which is a technique that predicts the performance of different GPUs running the same high-level code.

**General scalability predictions.** Wave scaling is similar in spirit to ESTIMA [17], since both use the idea of making measurements on one system to make performance predictions for a different system. However, ESTIMA is a scalability predictor for CPU programs. Wave scaling instead targets GPU kernels, which run under a different execution model when compared to CPU programs.

**Repetitiveness of DNN training.** Prior work leverages the repetitiveness of DNN training computation to optimize distributed training [43, 48, 60], schedule jobs in a cluster [18, 61, 103], and to apply DNN compiler optimizations [94]. The key difference between these works and Habitat is that they apply optimizations on the *same* hardware configuration. Habitat exploits the repetitiveness of DNN training to make performance predictions on *different* hardware configurations.

**DNN benchmarking.** A body of prior work focuses on benchmarking DNN training [3, 23, 53, 109]. While these works provide DNN training performance insights, they do so only for a fixed set of DNNs and hardware configurations. In contrast, Habitat analyzes DNNs in general and provides performance predictions on different GPUs to help users make informed GPU selections.

8 Conclusion

We present Habitat: a new runtime-based library that uses wave scaling and MLPs as execution time predictors to help deep learning researchers and practitioners make informed cost-efficient decisions when selecting a GPU for DNN training. The key idea behind Habitat is to leverage information collected at runtime on one GPU to help predict the execution time of a DNN training iteration on a different GPU. We evaluate Habitat and find that it makes cross-GPU iteration execution time predictions with an overall average error of 11.8% on ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN. Finally, we present two case studies where Habitat correctly predicts that (i) optimizing for cost-efficiency would lead to selecting a different GPU for GNMT, and (ii) that the V100 does not offer significant performance benefits over a common desktop-class GPU (the 2080Ti) for DCGAN. We have also open sourced Habitat (github.com/geoffxy/habitat) to benefit both the deep learning and systems communities [105, 106].
Acknowledgments

We thank our shepherd, Marco Canini, and the anonymous reviewers for their feedback. We also thank (in alphabetical order) Moshe Gabel, James Gleeson, Anand Jayarajan, Xiaodan Tan, Alexandra Tsvetkova, Shang Wang, Qiongsi Wu, and Hongyu Zhu. We thank all members of the EcoSystem research group for the stimulating research environment they provide. This work was supported by a QEII-GSST, Vector Scholarship in Artificial Intelligence, Snap Research Scholarship, and an NSERC Canada Graduate Scholarship – Master’s (CGS M). This work was also supported in part by the NSERC Discovery grant, the Canada Foundation for Innovation JELF grant, the Connaught Fund, an Amazon Research Award, and a Facebook Faculty Award. Computing resources used in this work were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute www.vectorinstitute.ai/partners.

References


[102] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kuriyan, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado,


