arXiv:1706.04972v1 [cs.LG] 13 Jun 2017

Device Placement Optimization with Reinforcement Learning

Azalia Mirhoseini [*1,2] **Hieu Pham** [*1,2] **Quoc V. Le** [1] **Benoit Steiner** [1] **Rasmus Larsen** [1] **Yuefeng Zhou** [1]
Naveen Kumar [3] **Mohammad Norouzi** [1] **Samy Bengio** [1] **Jeff Dean** [1]

Abstract

The past few years have witnessed a growth in size and computational requirements for training and inference with neural networks. Currently, a common approach to address these requirements is to use a heterogeneous distributed environment with a mixture of hardware devices such as CPUs and GPUs. Importantly, the decision of placing parts of the neural models on devices is often made by human experts based on simple heuristics and intuitions. In this paper, we propose a method which learns to optimize device placement for TensorFlow computational graphs. Key to our method is the use of a sequence-to-sequence model to predict which subsets of operations in a TensorFlow graph should run on which of the available devices. The execution time of the predicted placements is then used as the reward signal to optimize the parameters of the sequence-to-sequence model. Our main result is that on Inception-V3 for ImageNet classification, and on RNN LSTM, for language modeling and neural machine translation, our model finds non-trivial device placements that outperform hand-crafted heuristics and traditional algorithmic methods.

1. Introduction

Over the past few years, neural networks have proven to be a general and effective tool for many practical problems, such as image classification (Krizhevsky et al., 2012; Szegedy et al., 2015; He et al., 2016), speech recognition (Hinton et al., 2012; Graves & Jaitly, 2014; Hannun et al., 2014; Chan et al., 2015), machine translation (Sutskever et al., 2014; Cho et al., 2014; Bahdanau et al., 2015; Wu et al., 2016) and speech synthesis (Oord et al., 2016; Arik et al., 2017; Wang et al., 2017). Together with their success is the growth in size and computational requirements of training and inference. Currently, a typical approach to address these requirements is to use a heterogeneous distributed environment with a mixture of many CPUs and GPUs. In this environment, it is a common practice for a machine learning practitioner to specify the device placement for certain operations in the neural network. For example, in a neural translation network, each layer, including all LSTM layers, the attention layer, and the softmax layer, is computed by a GPU (Sutskever et al., 2014; Wu et al., 2016).

Although such decisions can be made by machine learning practitioners, they can be challenging, especially when the network has many branches (Szegedy et al., 2016), or when the minibatches get larger. Existing algorithmic solvers (Pellegrini, 2009; Karypis & Kumar, 1995b), on the other hand, are not flexible enough to work with a dynamic environment with many interferences.

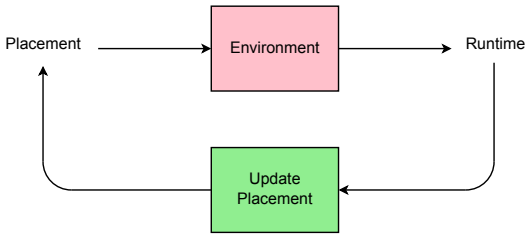

Figure 1. An overview of the RL based device placement model.

In this paper, we propose a method which learns to optimize device placement for training and inference with neural networks. The method, illustrated in Figure 1, takes into account information of the environment by performing series of experiments to understand which parts of the model should be placed on which device, and how to arrange the computations so that the communication is optimized. Key to our method is the use of a sequence-to-sequence model to read input information about the operations as well as the dependencies between them, and then propose a placement for each operation. Each proposal is executed in the hardware environment to measure the execution time. The execution time is then used as a reward signal to train the

[*]Equal contribution [1]Google Brain [2]Members of the Google Brain Residency Program (g.co/brainresidency) [3]Google. Correspondence to: Azalia Mirhoseini <azalia@google.com>, Hieu Pham <hyhieu@google.com>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

recurrent model so that it gives better proposals over time.

Our main result is that our method finds non-trivial placements on multiple devices for Inception-V3 (Szegedy et al., 2016), Recurrent Neural Language Model (Zaremba et al., 2014; Jozefowicz et al., 2016) and Neural Machine Translation (Sutskever et al., 2014; Wu et al., 2016). Single-step measurements show that Scotch (Pellegrini, 2009) yields disappointing results on all three benchmarks, suggesting that their graph-based heuristics are not flexible enough for them. Our method can find non-trivial placements that are up to 3.5 times faster. When applied to train the three models in real time, the placements found by our method are up to 20% faster than human experts' placements.

2. Related Work

Our work is closely related to the idea of using neural networks and reinforcement learning for combinatorial optimization (Vinyals et al., 2015; Bello et al., 2016). The space of possible placements for a computational graph is discrete, and we model the placements using a sequence-to-sequence approach, trained with policy gradients. However, experiments in early work were only concerned with toy datasets, whereas this work applies the framework to a large-scale practical application with noisy rewards.

Reinforcement learning has also been applied to optimize system performance. For example, Mao et al. (2016) propose to train a resource management algorithm with policy gradients. However, they optimize the expected value of a hand-crafted reward, unlike this work, where we optimize directly for the running time of the configurations, hence relieving the need to design intermediate cost models.

Graph partitioning is an intensively studied subject in computer science. Early work such as Kernighan & Lin (1970); Kirkpatrick et al. (1983); Fiduccia & Mattheyses (1988); Johnson et al. (1989) employ several iterative refinement procedures that start from a partition and continue to explore similar partitions to improve. Alternative methods such as Hagen & Kahng (1992); Karypis & Kumar (1995b) perform spectral analyses on matrix representations of graphs to partition them. Despite their extensive literature, graph partitioning algorithms remain heuristics for computational graphs. The reason is that in order to apply these algorithms, one has to construct cost models for the graphs of concern. Since such models are expensive to even estimate and in virtually all cases, are not accurate, graph partitioning algorithms applied on them can lead to unsatisfying results, as we show in Section 4 of this paper.

A well-known graph partitioning algorithm with an open source software library is the Scotch optimizer (Pellegrini, 2009), which we use as a baseline in our experiments. The Scotch mapper attempts to balance the computational load of a collection of tasks among a set of connected processing nodes, while reducing the cost of communication by keeping intensively communicating tasks on nearby nodes. Scotch relies on a collection of graph partitioning techniques such as k-way Fiduccia-Mattheyses (Fiduccia & Mattheyses, 1988), multilevel method (Barnard & Simon, 1994; Hendrickson & Leland, 1993; Karypis & Kumar, 1995a), band method (Chevalier & Pellegrini, 2006), diffusion method (Pellegrini, 2007), and dual recursive bipartitioning mapping (Pellegrini & Roman, 1996)).

Scotch models the problem with 2 graphs. The first graph is called the target architecture graph, whose vertices represent hardware resources such as CPUs or GPUs and whose edges represent the communication paths available between them, such as a PCIe bus or a network link. The second graph is called the source graph, which models the computation to be mapped onto the target architecture graph. In the case of TensorFlow (Abadi et al., 2016), the computations of programs are modeled as a graph whose vertices represent operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. Scotch users have to choose how and when given partitioning should be applied to graphs. However, in our experiment, we rely on the software's default strategies implemented in Scotch, which have already been extensively tuned.

3. Method

Consider a TensorFlow computational graph $\mathcal{G}$, which consists of M operations $\{o_1, o_2, ..., o_M\}$, and a list of D available devices. A placement $\mathcal{P} = \{p_1, p_2, ..., p_M\}$ is an assignment of an operation $o_i \in \mathcal{G}$ to a device p_i, where $p_i \in \{1, ..., D\}$. Let $r(\mathcal{P})$ denote the time that it takes to perform a complete execution of $\mathcal{G}$ under the placement $\mathcal{P}$. The goal of *device placement optimization* is to find $\mathcal{P}$ such that the execution time $r(\mathcal{P})$ is minimized.

3.1. Training with Policy Gradients

While we seek to minimize the execution time $r(\mathcal{P})$, direct optimization of $r(\mathcal{P})$ results in two major issues. First, in the beginning of the training process, due to the bad placements sampled, the measurements of $r(\mathcal{P})$ can be noisy, leading to inappropriate learning signals. Second, as the RL model gradually converges, the placements that are sampled become more similar to each other, leading to small differences between the corresponding running times, which results in less distinguishable training signals. We empirically find that the square root of running time, $R(\mathcal{P}) = \sqrt{r(\mathcal{P})}$, makes the learning process more robust. Accordingly, we propose to train a stochastic pol-

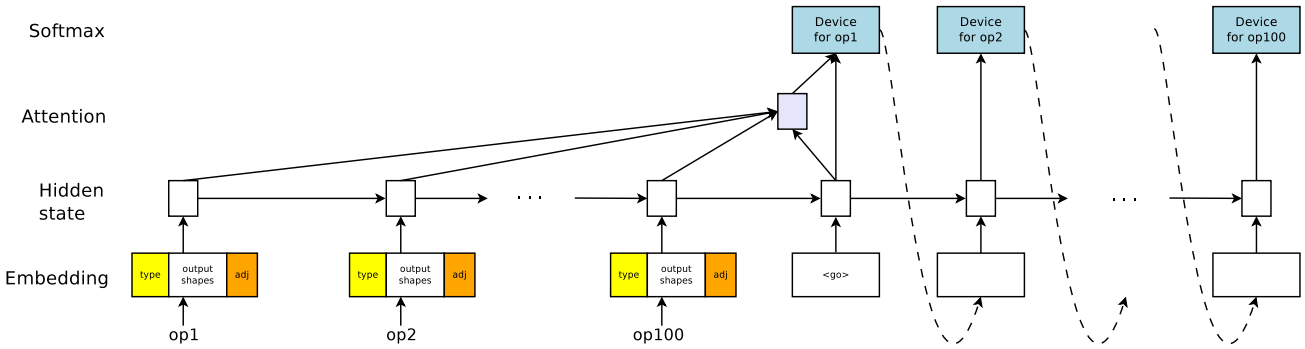

Figure 2. Device placement model architecture.

icy $\pi(\mathcal{P}|\mathcal{G};\theta)$ to minimize the objective

$$J(\theta) = \mathbf{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}\left[R\left(\mathcal{P}\right)|\mathcal{G}\right] \quad (1)$$

In our work, $\pi(\mathcal{P}|\mathcal{G};\theta)$ is defined by an attentional sequence-to-sequence model, which we will describe in Section 3.2. We learn the network parameters using Adam (Kingma & Ba, 2014) optimizer based on policy gradients computed via the REINFORCE equation (Williams, 1992),

$$\nabla_\theta J(\theta) = \mathbf{E}_{\mathcal{P}\sim\pi(\mathcal{P}|\mathcal{G};\theta)}\left[R\left(\mathcal{P}\right)\cdot\nabla_\theta \log p\left(\mathcal{P}|\mathcal{G};\theta\right)\right] \quad (2)$$

We estimate $\nabla_\theta J(\theta)$ by drawing K placement samples using $\mathcal{P}_i \sim \pi(\cdot|\mathcal{G};\theta)$. We reduce the variance of policy gradients by using a baseline term B, leading to

$$\nabla_\theta J(\theta) \approx \frac{1}{K}\sum_{i=1}^{K}\left(R\left(\mathcal{P}_i\right) - B\right)\cdot\nabla_\theta \log p\left(\mathcal{P}_i|\mathcal{G};\theta\right) \quad (3)$$

We find that a simple moving average baseline B works well in our experiments. In practice, on computational graphs with large memory footprints, some placements can fail to execute, e.g., putting all of the operations of a huge LSTM on a single GPU will exceed the device's memory limit. For such cases, we set the square root of running time $R(\mathcal{P})$ to a large constant, which we call the failing signal. We specify the failing signal manually depending on the input graph. We observe that throughout our training process, some placements sporadically and unexpectedly fail, perhaps due to factors such as the state of the machine (we train our model on a shared cluster). This phenomenon is particularly undesirable towards the end of the training process, since a large difference between $R(\mathcal{P}_i)$ and the baseline B leads to a large update of the parameters, which potentially perturbs parameters θ out of a good minimum. We thus hard-code the training process so that after $5,000$ steps, one performs a parameter update with a sampled placement $\mathcal{P}$ only if the placement executes. In our experiments, we also find that initializing the baseline B with the failing signal results in more exploration.

3.2. Architecture Details

We use a sequence-to-sequence model (Sutskever et al., 2014) with LSTM (Hochreiter & Schmidhuber, 1997) and a content-based attention mechanism (Bahdanau et al., 2015) to predict the placements. Figure 2 shows the overall architecture of our model, which can be divided into two parts: encoder RNN and decoder RNN.

The input to the encoder RNN is the sequence of operations of the input graph. We embed the operations by concatenating their information. Specifically, for each input graph $\mathcal{G}$, we first collect the types of its operations. An operation's type describes the underlying computation, such as `MatMul` or `conv2d`. For each type, we store a tunable embedding vector. We then record the size of each operation's list of output tensors and concatenate them into a fixed-size zero-padded list called the output shape. We also take the one-hot encoding vector that represents the operations that are direct inputs and outputs to each operation. Finally, the embedding of each operation is the concatenation of its type, its output shape, and its one-hot encoded adjacency information.

The decoder is an attentional LSTM (Bahdanau et al., 2015) with a fixed number of time steps that is equal to the number of operations in a graph $\mathcal{G}$. At each step, the decoder outputs the device for the operation at the same encoder time step. Each device has its own tunable embedding, which is then fed as input to the next decoder time step.

3.3. Co-locating Operations

A key challenge when applying our method to TensorFlow computational graphs is that these graphs generally have thousands of operations (see Table 1). Modeling such a large number of operations with sequence-to-sequence models is difficult due to vanishing and exploding gradient issues (Pascanu et al., 2013) and large memory footprints. We propose to reduce the number of objects to place on dif-

ferent devices by manually forcing several operations to be located on the same device. In practice, this is implemented by the `colocate_with` feature of TensorFlow.

We use several heuristics to create co-location groups. First, we rely on TensorFlow's default co-location groups, such as co-locating each operation's outputs with its gradients. We further apply a simple heuristic to merge more operations into co-location groups. Specifically, if the output of an operation X is consumed *only* by another operation Y, then operations X and Y are co-located. Many initialization operations in TensorFlow can be grouped in this way. In our experiments, we apply this heuristic recursively, and after each iteration, we treat the co-location groups as operations until there are not any further groups that can be merged. For certain models, we apply specific rules to construct co-location groups. For example, with ConvNets, we can treat several convolutions and pooling layers as a co-location group, and with RNN models, we treat each LSTM cell as a group.

3.4. Distributed Training

We speed up the training process of our model using asynchronous distributed training, as shown in Figure 3. Our framework consists of several controllers, each of which execute the current policy defined by the attentional sequence-to-sequence model as described in Section 3.2. All of the controllers interact with a single shared parameter server. We note that the parameter server holds only the controllers' parameters, and not the input graph's parameters, because keeping the input graph's parameters on the parameter server can potentially create a latency bottleneck to transfer these parameters. Each controller in our framework interacts with K workers, where K is the number of Monte Carlo samples in Equation 3.

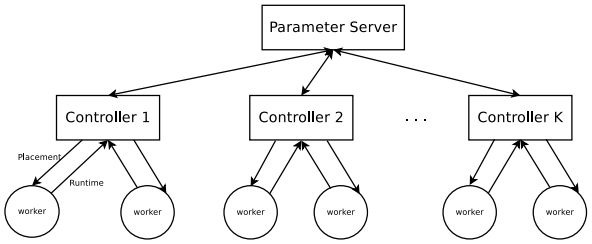

Figure 3. Distributed and asynchronous parameter update and reward evaluation.

The training process has two alternating phases. In the first phase, each worker receives a signal that indicates that it should wait for placements from its controller, while each controller receives a signal that indicates it should sample K placements. Each sampled placement comes with a probability. Each controller then independently sends the placements to their workers, one placement per worker, and sends a signal to indicate a phase change.

In the second phase, each worker executes the placement it receives and measures the running time. To reduce the variance in these measurements, each placement is executed for 10 steps and the average running time of the steps but the first one is recorded. We observe that in TensorFlow, the first step can take longer to execute compared to the following steps, and hence we treat the first layer as an outlier. Each controller waits for all of its workers to finish executing their assigned placements and returning their running times. When all of the running times are received, the controller uses the running times to scale the corresponding gradients to asynchronously update the controller parameters that reside in the parameter server.

In our experiments, we use up to 20 controllers, each with either 4 or 8 workers. Under this setting, it takes between 12 to 27 hours to find the best placement for the models in our experiments. Using more workers per controller yields more accurate estimates of the policy gradient as in Equation 3, but comes at the expense of possibly having to put more workers in idle states. We also note that due to the discrepancies between machines, it is more stable to let each controller have its own baseline.

4. Experiments

In the following experiments, we apply our proposed method to assign computations to devices on three important neural networks in the deep learning literature: Recurrent Neural Language Model (RNNLM) (Zaremba et al., 2014; Jozefowicz et al., 2016), Attentional Neural Machine Translation (Bahdanau et al., 2015), and Inception-V3 (Szegedy et al., 2016). We compare the RL placements against strong existing baselines described in Section 4.2.

4.1. Experiment Setup

Benchmarks. We evaluate our approach on three established deep learning models:

- Recurrent Neural Network Language Model (RNNLM) with multiple LSTM layers (Zaremba et al., 2014; Jozefowicz et al., 2016). The grid structure of this model introduces tremendous potential for parallel executions because each LSTM cell can start as soon as its input and previous states are available.
- Neural Machine Translation with attention mechanism (NMT) (Bahdanau et al., 2015; Wu et al., 2016). While the architecture of this model is similar to that of RNNLM, its large number of hidden states due to the source and target sentences necessitates model

parallelism. Both Sutskever et al. (2014) and Wu et al. (2016) propose to place each LSTM layer, the attention layer, and the softmax layer, each on a separate device. While the authors observe significant improvements at training time, their choices are not optimal. In fact, we show in our experiments that a trained policy can find significantly better placements.

- Inception-V3 (Szegedy et al., 2016) is a widely-used architecture for image recognition and visual feature extraction (Khetan & Oh, 2016; Esteva et al., 2016). The Inception network has multiple blocks. Each block has several branches of convolutional and pooling layers, which are then concatenated to make the inputs for the next block. While these branches can be executed in parallel, the network's depth restricts such potential since the later blocks have to wait for the previous ones.

Model details. For Inception-V3, each step is executed on a batch of images, each of size $299 \times 299 \times 3$, which is the widely-used setting for the ImageNet Challenge (Szegedy et al., 2015). For RNNLM and NMT, we use the model with 2 LSTM layers, with sizes of 2048 and 1024, respectively. We set the number of unrolling steps for RNNLM, as well as the maximum length for the source and target sentences of NMT, to 40. Each pass on RNNLM and NMT consists of a minibatch of 64 sequences.

Co-location groups. We pre-process the TensorFlow computational graphs of the three aforementioned models to manually create their co-location groups. More precisely; for RNNLM and NMT, we treat each LSTM cell, each embedding lookup, each attention step and each softmax prediction step as a group; for Inception-V3, we treat each branch as a group. Table 1 shows the grouping statistics of these models.

Model	#operations	#groups
RNNLM	8943	188
NMT	22097	280
Inception-V3	31180	83

Table 1. Model statistics.

Metrics. We implement training operations for RNNLM and NMT using Adam (Kingma & Ba, 2014), and for Inception-V3 using RMSProp (Tieleman & Hinton, 2012). We evaluate a placement by the total time it takes to perform one forward pass, one backward pass and one parameter update. To reduce measurement variance, we average the running times over several trials. Additionally, we train each model from scratch using the placements found by our method and compare the training time to that of the strongest baseline placement.

Devices. In our experiments, the available devices are 1 Intel Haswell 2300 CPU, which has 18 cores, and either 2 or 4 Nvidia Tesla K80 GPUs. We allow 50 GB of RAM for all models and settings.

4.2. Baselines

Single-CPU. This placement executes the whole neural network on a single CPU. Processing some large models on GPUs is infeasible due to memory limits, leaving Single-CPU the only choice despite being slow.

Single-GPU. This placement executes the whole neural network on a single CPU. If an operation lacks GPU implemention, it will be placed on CPU.

Scotch. We estimate the computational costs of each operation as well as the amount of data that flows along each edge of the neural network model, and feed them to the Scotch static mapper (Pellegrini, 2009). We also annotate the architecture graph (see Section 2) with compute and communication capacities of the underlying devices.

MinCut. We use the same Scotch optimizer, but eliminate the CPU from the list of available devices fed to the optimizer. Similar to the single-GPU placement, if an operation has no GPU implementation, it runs on the CPU.

Expert-designed. For RNNLM and NMT, we put each LSTM layer on a device. For NMT, we also put the attention mechanism and the softmax layer on the same device with the highest LSTM layer, and we put the embedding layer on the same device with the first LSTM layer. For Inception-V3, the common practice for the batch size of 32 is to put the entire model on a single GPU. There is no implementation of Inception-V3 with batch 32 using more than 1 GPU. To create an intuitive baseline on multiple GPUs, we heuristically partition the model into contiguous parts that have roughly the same number of layers. We compare against this approach in Section 4.3. The common practice for Inception-V3 with the larger batch size of 128 is to apply data parallelism using 4 GPUs. Each GPU runs a replica of the model and processes a batch of size 32 (Szegedy et al., 2016). We compare against this approach in Section 4.4.

4.3. Single-Step Runtime Efficiency

In Table 2, we present the per-step running times of the placements found by our method and by the baselines. We observe that our model is either on par with or better than

Tasks	Single-CPU	Single-GPU	#GPUs	Scotch	MinCut	Expert	RL-based	Speedup
RNNLM (batch 64)	6.89	**1.57**	2	13.43	11.94	3.81	**1.57**	0.0%
			4	11.52	10.44	4.46	**1.57**	0.0%
NMT (batch 64)	10.72	OOM	2	14.19	11.54	4.99	**4.04**	23.5%
			4	11.23	11.78	4.73	**3.92**	20.6%
Inception-V3 (batch 32)	26.21	**4.60**	2	25.24	22.88	11.22	**4.60**	0.0%
			4	23.41	24.52	10.65	**3.85**	19.0%

Table 2. Running times (in seconds) of placements found by RL-based method and the baselines (lower is better). For each model, the first row shows the results with 1 CPU and 2 GPUs; the second row shows the results with 1 CPU and 4 GPUs. Last column shows improvements in running time achieved by RL-based placement over fastest baseline. To reduce variance, running times less than 10 seconds are measured 15 times and the averages are recorded. OOM is Out Of Memory.

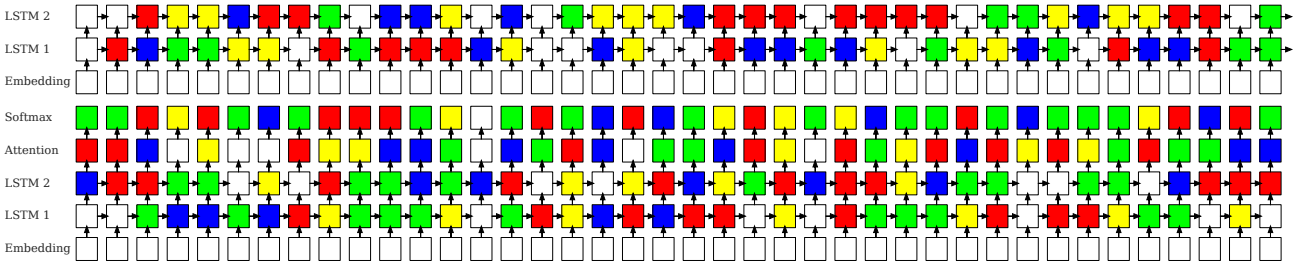

Figure 4. RL-based placement of Neural MT graph. Top: encoder, Bottom: decoder. Devices are denoted by colors, where the transparent color represents an operation on a CPU and each other unique color represents a different GPU. This placement achieves an improvement of 19.3% in running time compared to the fine-tuned expert-designed placement.

other methods of placements. Despite being given no information other than the running times of the placements and the number of available devices, our model learns subtle tradeoffs between performance gain by parallelism and the costs induced by inter-device communications.

RNNLM. Our method detects that it is possible to fit the whole RNNLM graph into one GPU, and decides to do so to save the inter-device communication latencies. The resulting placement is more than twice faster than the best published human-designed baseline.

Neural MT. Our method finds a non-trivial placement (see Figure 4) that leads to a speedup of up to 20.6% for 4 GPUs. Our method also learns to put the less computational expensive operations, such as embedding lookups, on the CPU. We suspect that whilst being the slowest device, the CPU can handle these lookup operations (which are less computationally expensive than other operations) to reduce the load for other GPUs.

Inception-V3. For Inception-V3 with the batch size of 32, RL-based placer learns that when there are only 2 GPUs available, the degree of freedom for model parallelism is limited. It thus places all the operations on a single GPU (although it could use 2 GPUs). However, when 4 GPUs are available, the RL-based placer finds an efficient way to use all of the GPUs, reducing the model's per-step running time from 4.60 seconds to 3.85 seconds. This result is significant, as neither of our baselines could find a placement better than assigning all the operations to a single GPU.

We also conduct a simple extension of our experiments, by increasing the batch sizes of RNNLM and NMT to 256, and their LSTM sizes to 4,096 and 2,048, respectively. This makes the models' memory footprints so large that even one layer of them cannot be fitted into any single device, hence ruling out the human-designed placement. Nevertheless, after several steps of finding placements that fail to run, our approach manages to find a way to successfully place input models on devices The running times of the placements found for large RNNLM and NMT are 33.46 and 35.84 seconds, respectively.

4.4. End-to-End Runtime Efficiency

We now investigate whether the RL-based placements can speedup not only the single-step running time but also the entire training process.

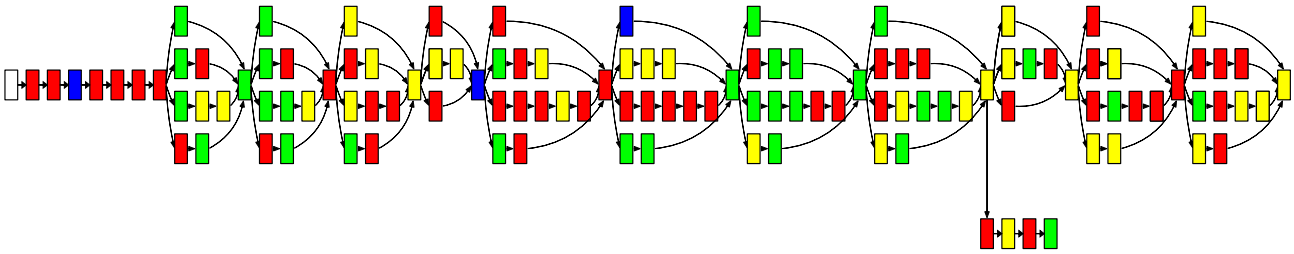

Figure 5. RL-based placement of Inception-V3. Devices are denoted by colors, where the transparent color represents an operation on a CPU and each other unique color represents a different GPU. RL-based placement achieves the improvement of 19.7% in running time compared to expert-designed placement.

Neural MT. We train our Neural MT model on the WMT14 English-German dataset.[1] For these experiments, we pre-process the dataset into word pieces (Wu et al., 2016) such that the vocabularies of both languages consist of 32,000 word pieces. In order to match our model's settings, we consider only the translation pairs where no sentence has more than 40 word pieces. We train each model for 200,000 steps and record their train perplexities. Each training machine has 4 Nvidia Tesla K80 GPUs and 1 Intel Haswell 2300 CPU. Since there are inevitable noises in the computer systems when measuring the running times, we train each model 4 times independently and average their per-step running times and perplexities.

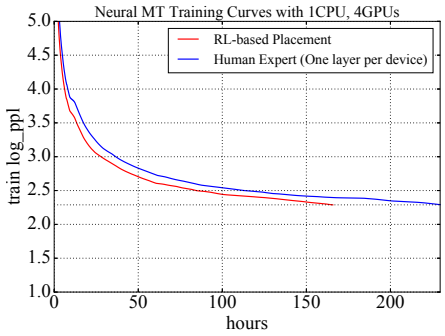

Figure 6. Training curves of NMT model using RL-based placement and expert-designed placement. The per-step running time as well as the perplexities are averaged over 4 runs.

The RL-based placement runs faster than the expert-designed placement, as shown in the training curves in Figure 6. Quantitatively, the expert-designed placement, which puts each layer (LSTM, attention and softmax) on a different GPU, takes 229.57 hours; meanwhile the RL-based placement (see Figure 4) takes 165.73 hours, giving 27.8% speed up of total training time. We note that the measured speedup rate (and the running times) of these models appear different than reported in Table 2 because measuring them in our RL method has several overheads.

[1] http://www.statmt.org/wmt14/

Inception-V3. We train Inception-V3 on the ImageNet dataset (Russakovsky et al., 2015) until the model reaches the accuracy of 72% on the validation set. In practice, more often, inception models are trained with data parallelism rather than model parallelism. We thus compare the placements found by our algorithm (see Figure 5) against two such baselines.

The first baseline, called Asynchronous Tower, puts one replica of the Inception-V3 network on each GPU. These replicas share the data reading operations, which are assigned to the CPU. Each replica independently performs forward and backward passes to compute the model's gradients with respect to a minibatch of 32 images and then updates the parameters asynchronously. The second baseline, called Synchronous Tower, is the same as Asynchronous Tower, except that it waits for the gradients of all copies before making an update. All settings use the learning rate of 0.045 and are trained using RMSProp.

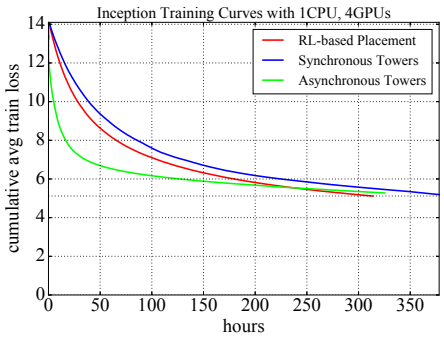

Figure 7. Training curves of Inception-V3 model using RL-based placement and two expert-designed placements: synchronous towers and asynchronous towers. The per-step running time as well as the perplexities are averaged over 4 runs.

Figure 7 shows the training curves of the three settings for Inception-V3. As can be seen from the figure, the end-to-end training result confirms that the RL-based placement indeed speedups the training process by 19.7% com-

pared to the Synchronous Tower. While Asynchronous Tower gives a better per-step time, synchronous approaches lead to faster convergence. The training curve of the RL-based placement, being slower at first, eventually crosses the training curve of Asynchronous Tower.

4.5. Analysis of Found Placements

In order to understand the rationale behind the RL-based placements, we analyze their profiling information and compare them against those of expert-designed placements.

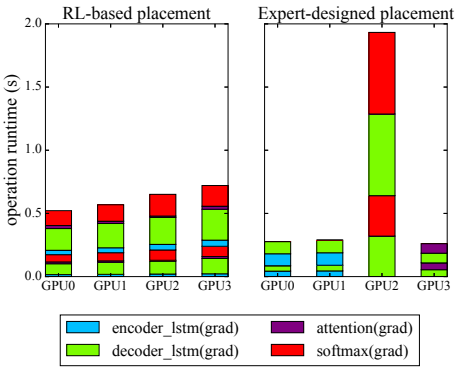

Figure 8. Computational load profiling of NMT model for RL-based and expert-designed placements. Smaller blocks of each color correspond to feedforward path and same-color upper blocks correspond to backpropagation. RL-based placement performs a more balanced computational load assignment than the expert-designed placement.

Neural MT. We first compare the per-device computational loads by RL-based placement and expert-designed placement for the NMT model. Figure 8 shows such performance profiling. RL-based placement balances the workload significantly better than does the expert-designed placement. Interestingly, if we do not take into account the time for back-propagation, then expert-designed placement makes sense because the workload is more balanced (whilst still less balanced than ours). The imbalance is much more significant when back-propagation time is considered.

Inception-V3. On Inception-V3, however, the RL-based placement does not seek to balance the computations between GPUs, as illustrated in Figure 9-top. We suspect this is because Inception-V3 has more dependencies than NMT, allowing less room for model parallelism across GPUs. The reduction in running time of the RL-based placement comes from the less time it spends copying data between devices, as shown in Figure 9-bottom. In particular, the models parameters are on the same device as the operations that use them, unlike in Synchronous Towers, where

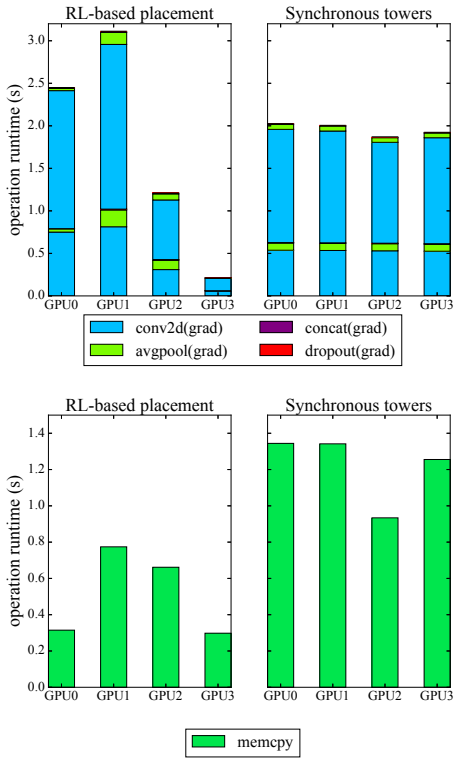

Figure 9. Computational load and memory copy profiling of Inception-V3 for RL-based and Synchronous Towers placements. Top figure: Operation runtime for GPUs. Smaller blocks of each color correspond to feedforward path and same-color upper blocks correspond to backpropagation. RL-based placement produces less balanced computational load than Synchronous Towers. Bottom figure: Memory copy time. All memory copy activities in Synchronous Towers are between a GPU and a CPU, which are in general slower than GPU to GPU copies that take place in the RL-based placement.

all towers have to wait for all parameters have to be updated and sent to them. On the contrary, that use them to reduce the communication cost, leading to overall reduction in computing time.

5. Conclusion

In this paper, we present an adaptive method to optimize device placements for neural networks. Key to our approach is the use of a sequence-to-sequence model to propose device placements given the operations in a neural network. The model is trained to optimize the execution time of the neural network. Besides the execution time, the number of available devices is the only other information about the hardware configuration that we feed to our model.

Our results demonstrate that the proposed approach learns the properties of the environment including the complex

tradeoff between computation and communication in hardware. On a range of tasks including image classification, language modeling, and machine translation, our method surpasses placements carefully designed by human experts and highly optimized algorithmic solvers.

Acknowledgements

We thank Martin Abadi, Stephan Gouws, and the Google Brain team for their help with the project.

References

Abadi, Martn, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek G., Steiner, Benoit, Tucker, Paul, Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. Tensorflow: A system for large-scale machine learning. *arXiv preprint arXiv:1605.08695*, 2016.

Arik, Sercan O, Chrzanowski, Mike, Coates, Adam, Diamos, Gregory, Gibiansky, Andrew, Kang, Yongguo, Li, Xian, Miller, John, Raiman, Jonathan, Sengupta, Shubho, et al. Deep voice: Real-time neural text-to-speech. *arXiv preprint arXiv:1702.07825*, 2017.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2015.

Barnard, S. T. and Simon, H. D. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: practice and Experience, 6(2):101117*, 1994.

Bello, Irwan, Pham, Hieu, Le, Quoc V., Norouzi, Mohammad, and Bengio, Samy. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Chan, William, Jaitly, Navdeep, Le, Quoc V, and Vinyals, Oriol. Listen, attend and spell. *arXiv preprint arXiv:1508.01211*, 2015.

Chevalier, C. and Pellegrini, F. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. *EuroPar, Dresden, LNCS 4128*, pp. 243–252, September 2006.

Cho, Kyunghyun, Van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Esteva, Andre, Kuprel, Brett, Novoa, Rob, Ko, Justin, Swetter, Susan, Blau, Helen M., and Thrun, Sebastian. Dermatologist-level classification of skin cancer. *Nature*, 2016.

Fiduccia, Charles M and Mattheyses, Robert M. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*, pp. 241–247. ACM, 1988.

Graves, Alex and Jaitly, Navdeep. Towards end-to-end speech recognition with recurrent neural networks. In *International Conference on Machine Learning*, 2014.

Hagen, Lars and Kahng, Andrew B. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

Hannun, Awni, Case, Carl, Casper, Jared, Catanzaro, Bryan, Diamos, Greg, Elsen, Erich, Prenger, Ryan, Satheesh, Sanjeev, Sengupta, Shubho, Coates, Adam, et al. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

Hendrickson, B. and Leland, R. A multilevel algorithm for partitioning graphs. *Technical Report SAND931301, Sandia National Laboratories*, June 1993.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E., Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N., et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

Hochreiter, Sepp and Schmidhuber, Jurgen. Long short-term memory. *Neural Computation*, 1997.

Johnson, David S, Aragon, Cecilia R, McGeoch, Lyle A, and Schevon, Catherine. Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

Jozefowicz, Rafal, Vinyals, Oriol, Schuster, Mike, Shazeer, Noam, and Wu, Yonghui. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Karypis, G. and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *Technical Report 95-035, University of Minnesota*, June 1995a.

Karypis, George and Kumar, Vipin. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995b.

Kernighan, Brian W and Lin, Shen. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal*, 49(2):291–307, 1970.

Khetan, Ashish and Oh, Sewoong. Achieving budget-optimality with adaptive schemes in crowdsourcing. In *Advances in Neural Information Processing Systems*, pp. 4844–4852. Curran Associates, Inc., 2016.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2014.

Kirkpatrick, Scott, Vecchi, Mario P, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

Mao, Hongzi, Alizadeh, Mohammad, Menache, Ishai, and Kandula, Srikanth. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

Oord, Aaron van den, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew, and Kavukcuoglu, Koray. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2013.

Pellegrini, F. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. *EuroPar, Rennes, LNCS 4641*, pp. 191–200, August 2007.

Pellegrini, F. Distillating knowledge about scotch. 2009.

Pellegrini, F. and Roman, J. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. *Research Report, LaBRI, Universite Bordeaux I*, August 1996.

Russakovsky, Olga, Deng, Jia, Su, Hao, Krause, Jonathan, Satheesh, Sanjeev, Ma, Sean, Huang, Zhiheng, Karpathy, Andrej, Khosla, Aditya, Bernstein, Michael, Berg, Alexander C., and Fei-Fei, Li. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 2015.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014.

Szegedy, Christian, Liu, Wei, Jia, Yangqing, Sermanet, Pierre, Reed, Scott, Anguelov, Dragomir, Erhan, Dumitru, Vanhoucke, Vincent, and Rabinovich, Andrew. Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

Szegedy, Christian, Vanhoucke, Vincent, Ioffe, Sergey, Shlens, Jon, and Wojna, Zbigniew. Rethinking the inception architecture for computer vision. In *The IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

Tieleman, T. and Hinton, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Wang, Yuxuan, Skerry-Ryan, R. J., Stanton, Daisy, Wu, Yonghui, Weiss, Ron J., Jaitly, Navdeep, Yang, Zongheng, Xiao, Ying, Chen, Zhifeng, Bengio, Samy, Le, Quoc V., Agiomyrgiannakis, Yannis, Clark, Rob, and Saurous, Rif A. Tacotron: A fully end-to-end text-to-speech synthesis model. In *InterSpeech*, 2017.

Williams, Ronald. Simple statistical gradient following algorithms for connectionnist reinforcement learning. In *Machine Learning*, 1992.

Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V., Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, Klingner, Jeff, Shah, Apurva, Johnson, Melvin, Liu, Xiaobing, ukasz Kaiser, Gouws, Stephan, Kato, Yoshikiyo, Kudo, Taku, Kazawa, Hideto, Stevens, Keith, Kurian, George, Patil, Nishant, Wang, Wei, Young, Cliff, Smith, Jason, Riesa, Jason, Rudnick, Alex, Vinyals, Oriol, Corrado, Greg, Hughes, Macduff, and Dean, Jeffrey. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.