Spotlight: Optimizing Device Placement for Training Deep Neural Networks

Yuanxiang Gao, Li Chen, Baochun Li
University of Toronto

And some other work on finding a device placement for training DNN

Device placement problem

- *Model parallelism* in a heterogeneous environment of both CPU and GPU devices
- How each operation in a neural network should be matched to each of these CPU and GPU devices
- The objective is to find a placement of operations to devices, so that the **time required to train a neural network can be minimized**
- Balance the computational load while minimizing communication cost
- Especially useful for large models, such as NMT

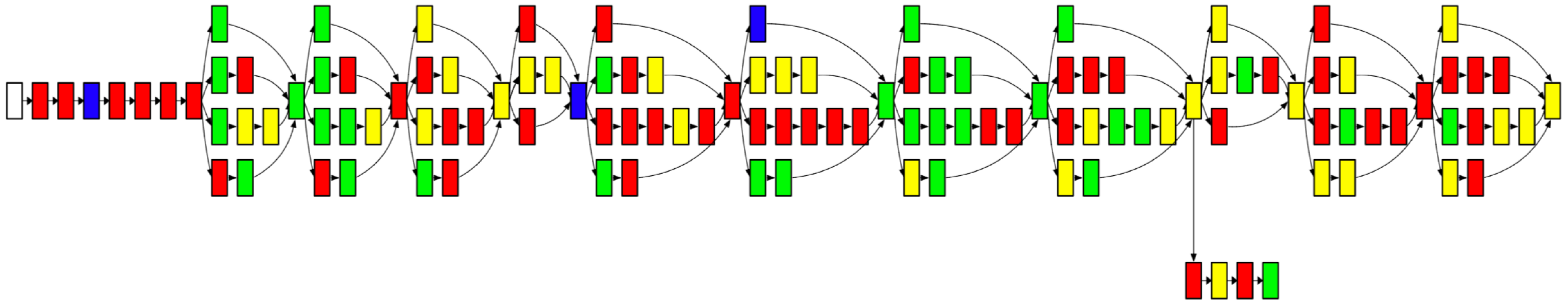


Figure 5. RL-based placement of Inception-V3. Devices are denoted by colors, where the transparent color represents an operation on a CPU and each other unique color represents a different GPU. RL-based placement achieves the improvement of 19.7% in running time compared to expert-designed placement.

Model	#operations	#groups
RNNLM	8943	188
NMT	22097 to 80k	280
Inception-V3	31180	83

Table 1. Model statistics.

Related work

- Naturally a graph partitioning problem (heuristics)
- Mirhoseini, A., *et al*. Device Placement Optimization with Reinforcement Learning. ICML 2017.
- Mirhoseini, A., *et al*. A Hierarchical Model for Device Placement. In ICLR 2018.
- Gao, Y., *et al*. Spotlight: Optimizing Device Placement for Training Deep Neural Networks. ICML 2018.
- Gao, Y., *et al*. Post: Device Placement with Cross-Entropy Minimization and Proximal Policy Optimization. NIPS 2018.

Motivations

- Some huge model cannot be placed on one GPU (OOM), requiring model parallelism
- Google's RL-based method (ICML 2017) is too expensive (27 hours and 160 workers), requiring lower overhead of training
- The standard policy gradient method is known to be inefficient, as it performs one gradient update for each data sample (Shulman et al., 2017).
- Expert-designed placements are not always the best, and takes time

Google’s method (ICML 2017)

- Using reinforcement learning based on policy gradient method (REINFORCE)
- Sample inefficient: 27 hours and 160 workers to find assignments on 4 GPUs

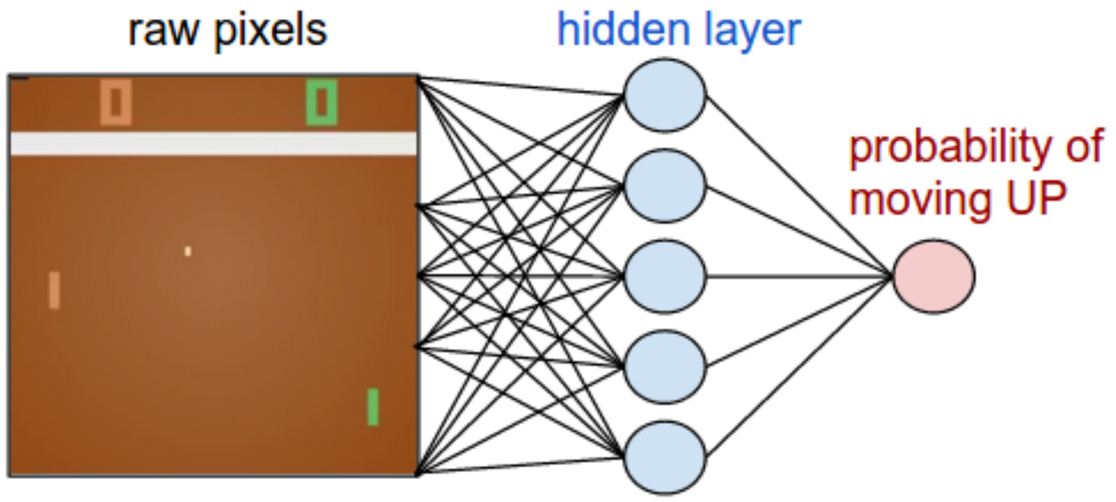
Spotlight: using Proximal Policy Optimization (PPO) to solve device placement

- Developed by OpenAI, DeepMind
- Better sample efficiency
- default RL algorithm at OpenAI because of its ease of use and good performance

Shulman, J., et al. Trust region policy optimization. In ICML, 2015.
https://blog.openai.com/openai-baselines-ppo/

Policy Gradient Method (Sutton, McAllester, Singh, and Mansour, 2000)

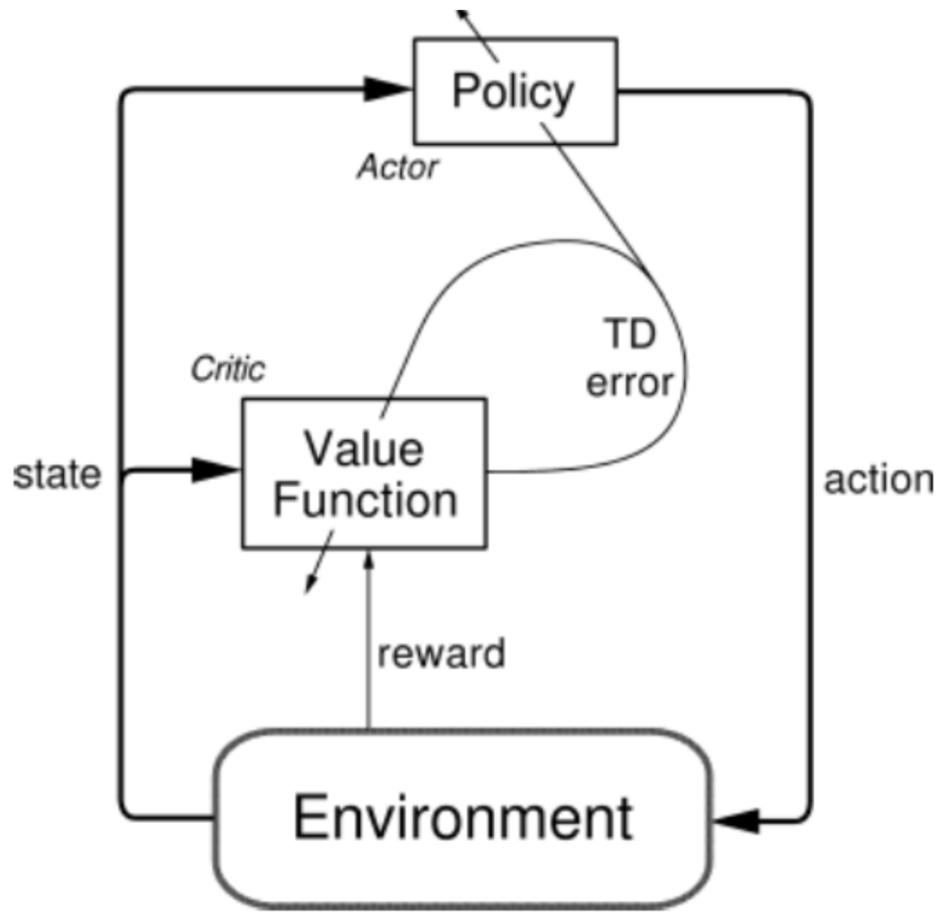
- Policy Network



- Policy Gradient

$$
\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}
$$

From Tian Han's slides



```
Input: policy $\pi(a|s,\boldsymbol{\theta})$, $\hat{v}(s,\boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s,\boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for true do
    generate an episode $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    for each step $t$ of episode $0, \cdots, T-1$ do
        $G_t \leftarrow$ return from step $t$
        $\delta \leftarrow G_t - \hat{v}(s_t,\boldsymbol{w})$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta\delta\nabla_{\boldsymbol{w}}\hat{v}(s_t,\boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t\boldsymbol{\delta}\nabla_{\boldsymbol{\theta}}log\pi(a_t|s_t,\boldsymbol{\theta})$
    end
end
```

REINFORCE: Actor-Critic Method

From Tian Han's slides

Modeled as a MDP

- multi-stage Markov decision process (MDP)
- At each **stage**, select the next operation and sample a probability distribution to obtain a placement recommendation on one of the available devices.
- Transit to the next **stage** with a new placement **state** where the previous operation has been placed.
- Training time of DNN with the final placement state is the reward. With this reward, we update the set of probability distributions and repeat the placements again.

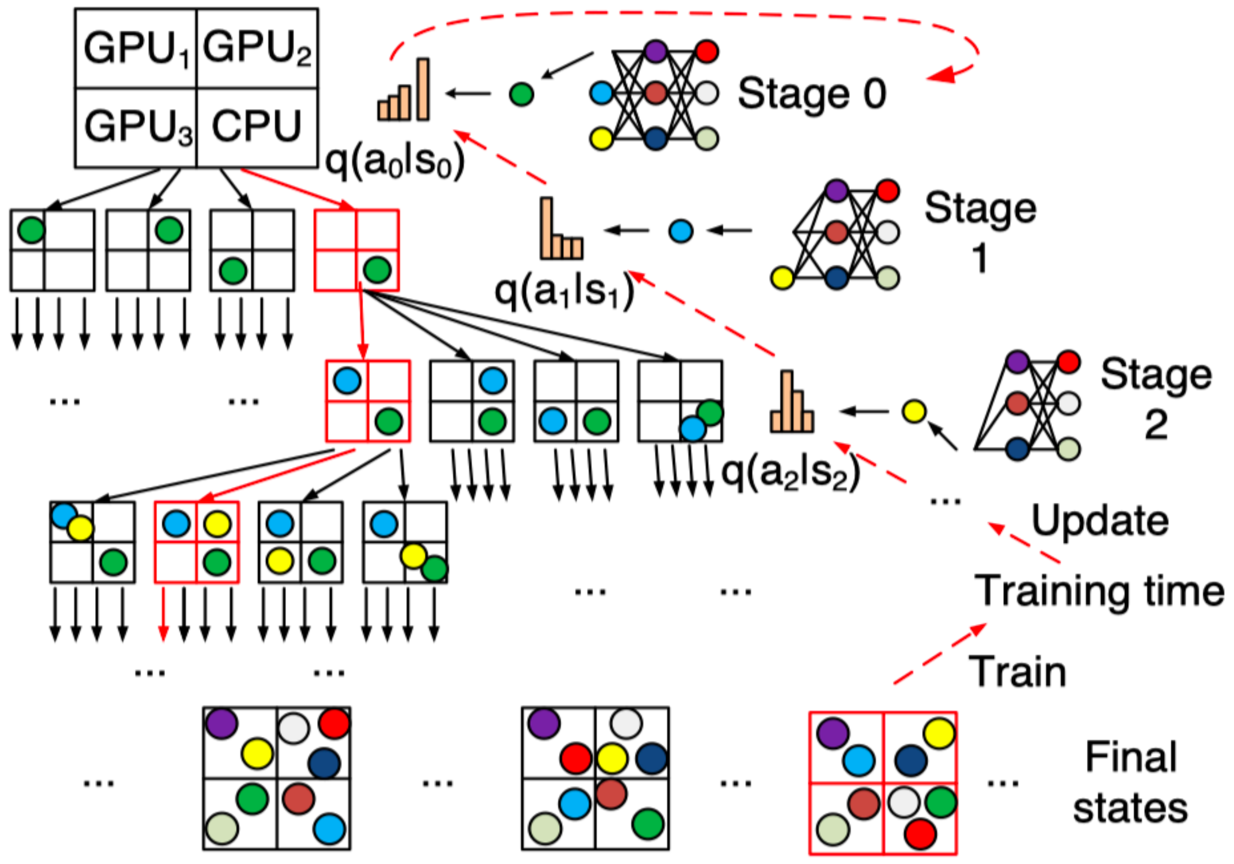


Figure 2. The state tree of a device placement MDP.

Notations

R - the training time

$\overline{R}$ - the average training time of all the previous trials

$$r(s_n) = \begin{cases} 0, & n < N \\ \overline{R} - R, & n = N \end{cases}$$

performance of the policy

$\eta(\pi) = E_{\{a_0, a_1, \cdots, a_{N-1}\} \sim \pi}[r(s_N)]$

state-action value function of a placement policy π

$Q_\pi(s_n, a_n) = E_{\{a_{n+1}, \cdots, a_{N-1}\} \sim \pi}[r(s_N)]$

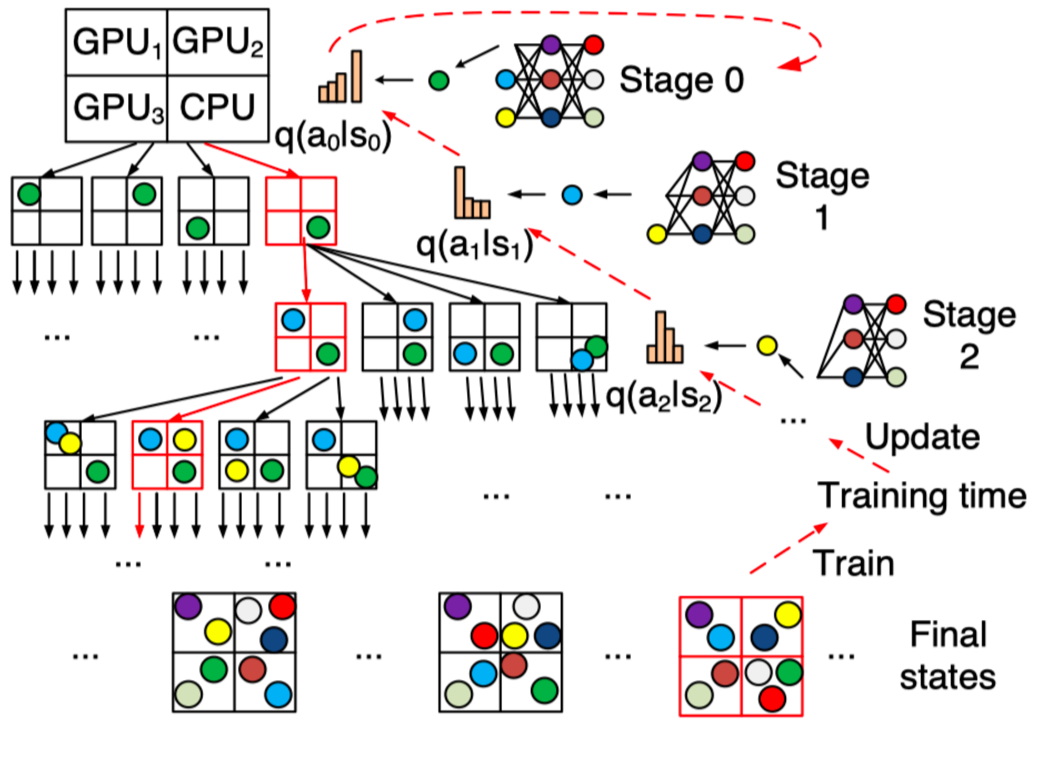


Figure 2. The state tree of a device placement MDP.

Expected reward value obtained at the final state s_N , starting from a particular state s_n and assignment a_n , following the set of assignments {a_n+1 , ..., a_N−1 } generated from π

How to update the old policy π to π'?

The performance of π can be naturally represented with such state-action values Q.

$$Q_\pi(s_n, a_n) = \sum_{a_i}[\prod_{i=n+1}^{N-1} q(a_i|s_i)r(s_N)].$$

Combining $\eta(\pi)$ and $Q_\pi(s_n, a_n)_n$, performance and Q-value can be connected as: $\eta(\pi) = \sum_{a_j}[\prod_{j=0} q(a_j | s_j)Q_\pi(s_n, a_n)]$

Updating policy π' to maxmize: $\eta(\pi') = E_{\{a_0,a_1,\cdots,a_{n-1}\}\sim\pi'}[\sum_{a_n} q'(a_n | s_n)Q_{\pi'}(s_n, a_n)]$

Challenging to maxmize because π' is decision to be made

Approximation:

$$\begin{aligned} F_\pi(\pi') = \eta(\pi) &= E_{\{a_0,a_1,\cdots,a_{n-1}\}\sim\pi}[\sum_{a_n} q'(a_n | s_n)Q_\pi(s_n, a_n)] \\ &= E_{a[0:n-1]\sim\pi}[\sum_{a_n} q(a_n | s_n)\{\frac{q'(a_n | s_n)}{q(a_n | s_n)}Q_\pi(s_n, a_n)\}] \\ &= E_{a[0:n]\sim\pi}[\frac{q'(a_n | s_n)}{q(a_n | s_n)}Q_\pi(s_n, a_n)] \end{aligned}$$

How to evaluate the distance between approximation and original policies?

Performance lower bound

Intuitively, $D_{KL}^{max}(\pi||\pi')$ represents the distance between the two policies.

Theorem 1: The expected performance of a new policy is bounded from below as follows:

$$\eta(\pi') \geq F_{\pi}(\pi') - \epsilon_1 - 2\epsilon_2 n D_{KL}^{\max}(\pi||\pi').$$

Constants can be treated as hyperparameters

Spotlight: Maximizing the performance lower bound in each update on the placement policy

$$\eta(\pi') \geq E_{a_{[o:n]}\sim\pi}[\frac{q'(a_n|s_n)}{q(a_n|s_n)}Q_{\pi}(s_n, a_n)] - \beta D_{KL}^{max}(\pi||\pi')$$

maximal KL divergence between the old policy and the new policy over all the states as penalty

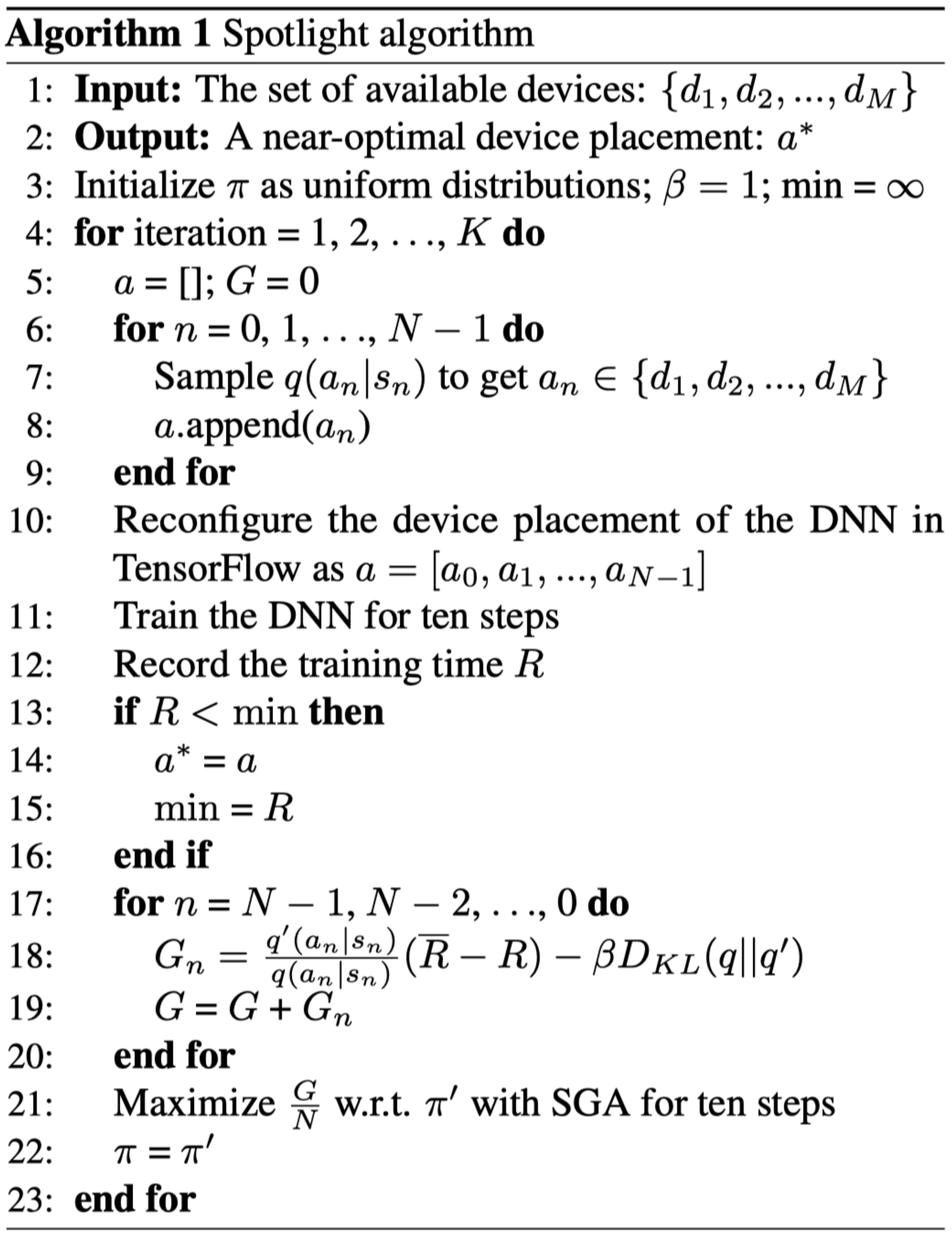
Practical optimization objective

$$\max_{\pi'} \frac{1}{N} \sum_{n=0, a_n \sim \pi}^{N-1} [\frac{q'(a_n | s_n)}{q(a_n | s_n)}(\overline{R} - R)] - \beta D_{KL}(\pi || \pi')$$

Max divergence can be replaced with divergence without impacting the final performance (Shulman et al., 2015)

The policy π is initialized with uniformly random distributions, and the hyperparameter β is set as the typical value of 1 (Shulman et al., 2017).

$\overline{R} - R$ is an estimate of $Q_\pi(s_n, a_n)$

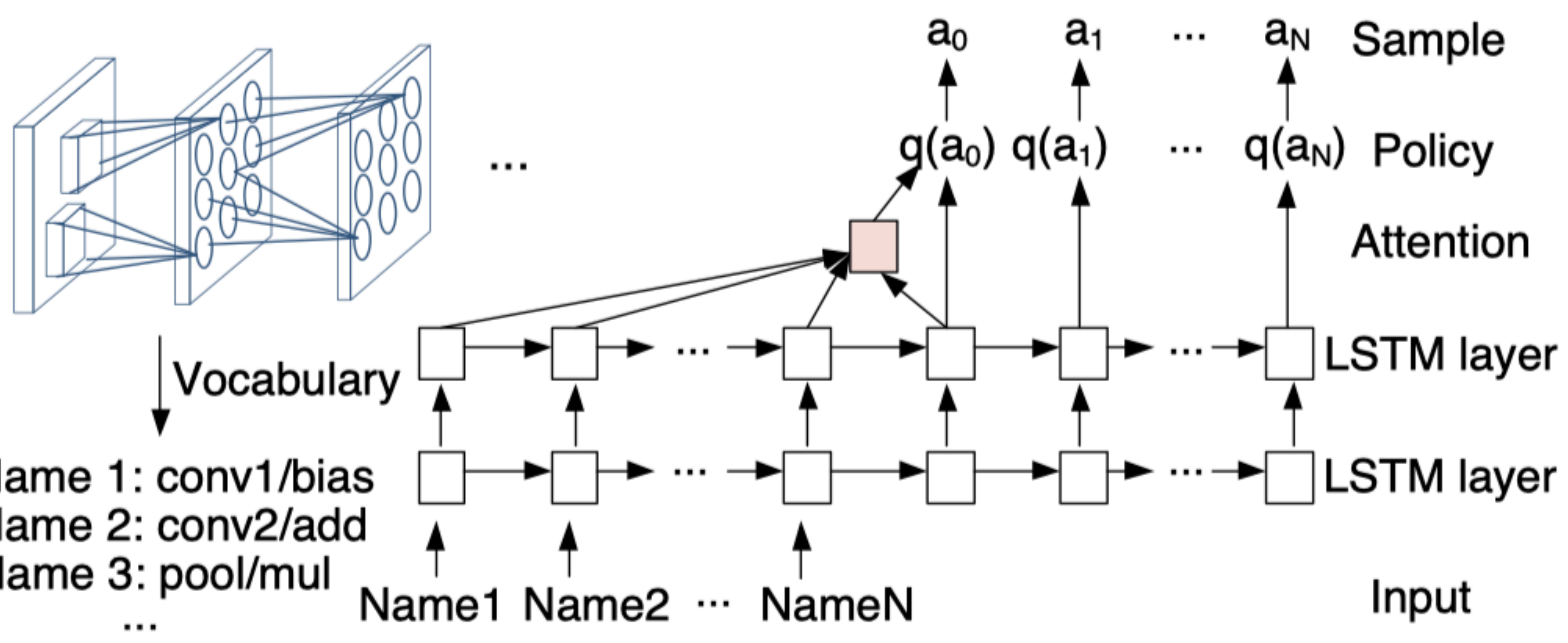


Algorithm 1 Spotlight algorithm

```
1:  Input: The set of available devices: ${d_1, d_2, ..., d_M}$
2:  Output: A near-optimal device placement: $a^*$
3:  Initialize $\pi$ as uniform distributions; $\beta = 1$; min = $\infty$
4:  for iteration = 1, 2, . . ., $K$ do
5:      $a$ = []; $G$ = 0
6:      for $n$ = 0, 1, . . ., $N - 1$ do
7:          Sample $q(a_n|s_n)$ to get $a_n \in \{d_1, d_2, ..., d_M\}$
8:          $a$.append($a_n$)
9:      end for
10:     Reconfigure the device placement of the DNN in
        TensorFlow as $a = [a_0, a_1, ..., a_{N-1}]$
11:     Train the DNN for ten steps
12:     Record the training time $R$
13:     if $R <$ min then
14:         $a^* = a$
15:         min = $R$
16:     end if
17:     for $n = N - 1, N - 2, . . ., 0$ do
18:         $G_n = \frac{q'(a_n|s_n)}{q(a_n|s_n)}(\overline{R} - R) - \beta D_{KL}(q||q')$
19:         $G = G + G_n$
20:     end for
21:     Maximize $\frac{G}{N}$ w.r.t. $\pi'$ with SGA for ten steps
22:     $\pi = \pi'$
23: end for
```

performance objective at each stage

Policy architecture



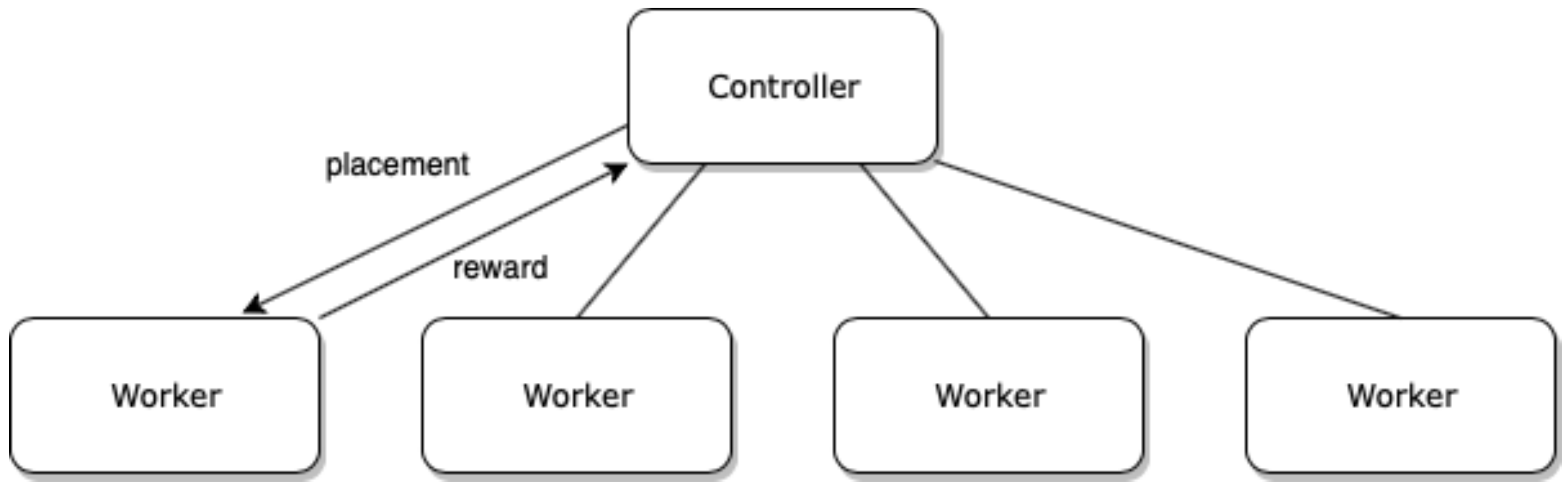
Generate a set of probability distributions $\pi = \{q(a_0, s_0), \cdots, q(a_N, s_N)\}$

Figure 3. Using a sequence-to-sequence recurrent neural network to represent the placement policy.

The policy π in Spotlight is represented by a two-layer sequence-to-sequence recurrent neural network (RNN) with long short term memory (LSTM) cells and a content-based attentional mechanism.

Experiment setup

- 5 4-GPU machines
- Benchmark: CIFAR-10 image classification (CNN)
- Distributed training:

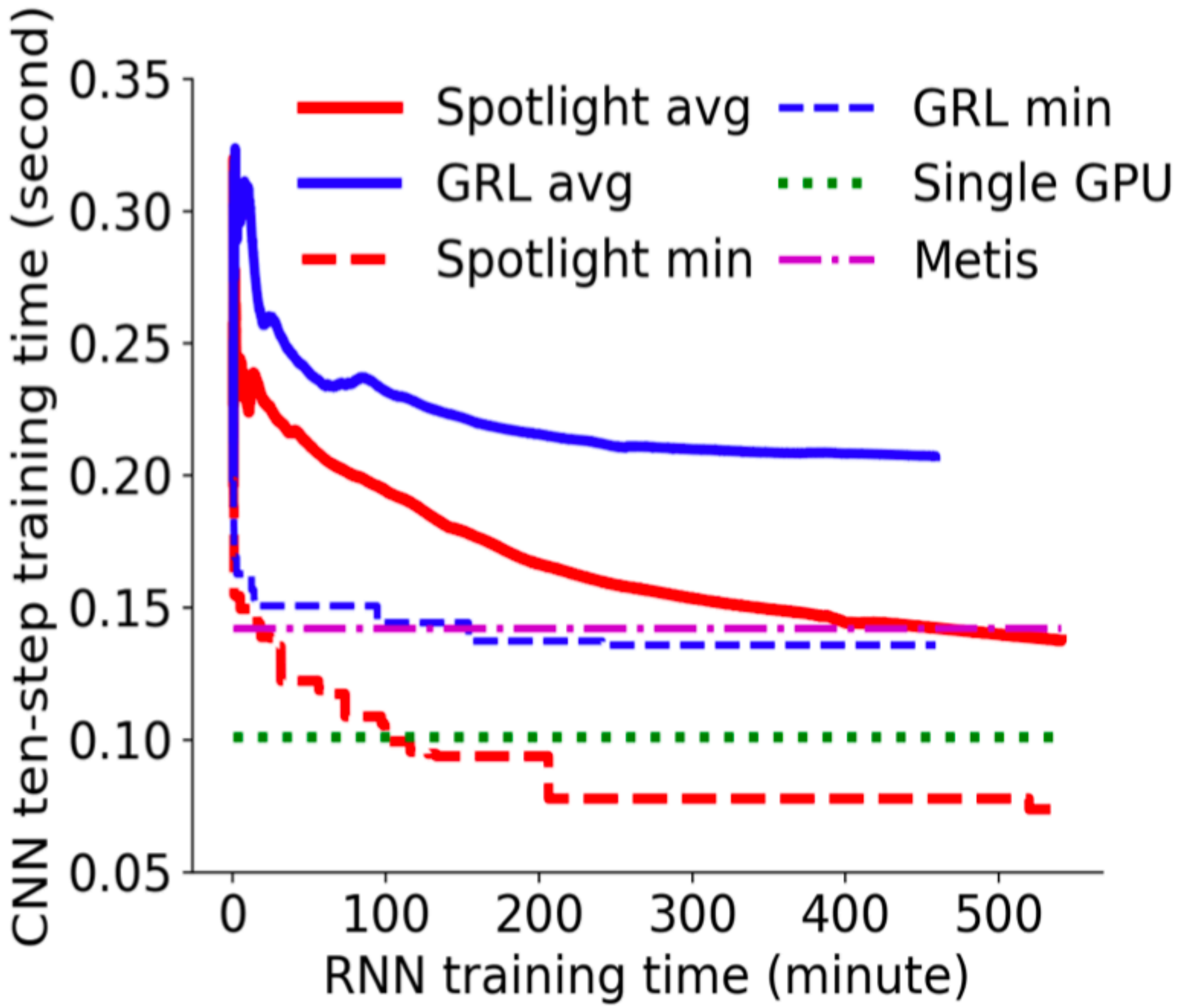


Co-location group

For CNN of CIFAR-10, **926** operations in total, hard to read into RNN model

Group all the operations that share the same two-level prefix into a super operation, **86** in total, better than default co-location group in TensorFlow (`tensorflow/core/grappler/utils/colocation.cc`) used in GRL2017

‘conv1/biases/', 'input_ops/input_ops/', ‘loss/cross_entropy/‘, …



20000 sampled placements were evaluated over five distributed machines, within a period of 500 minutes.

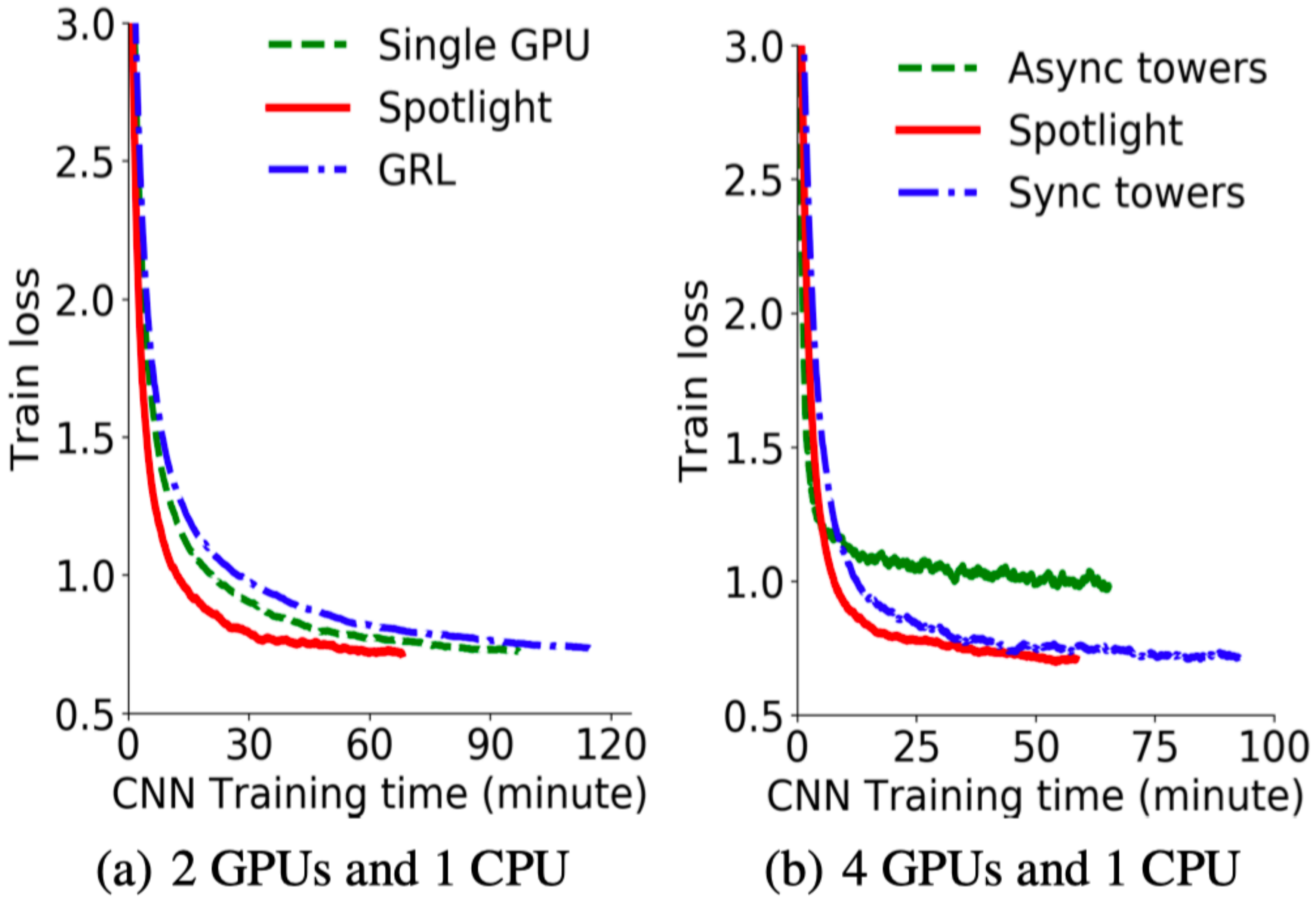
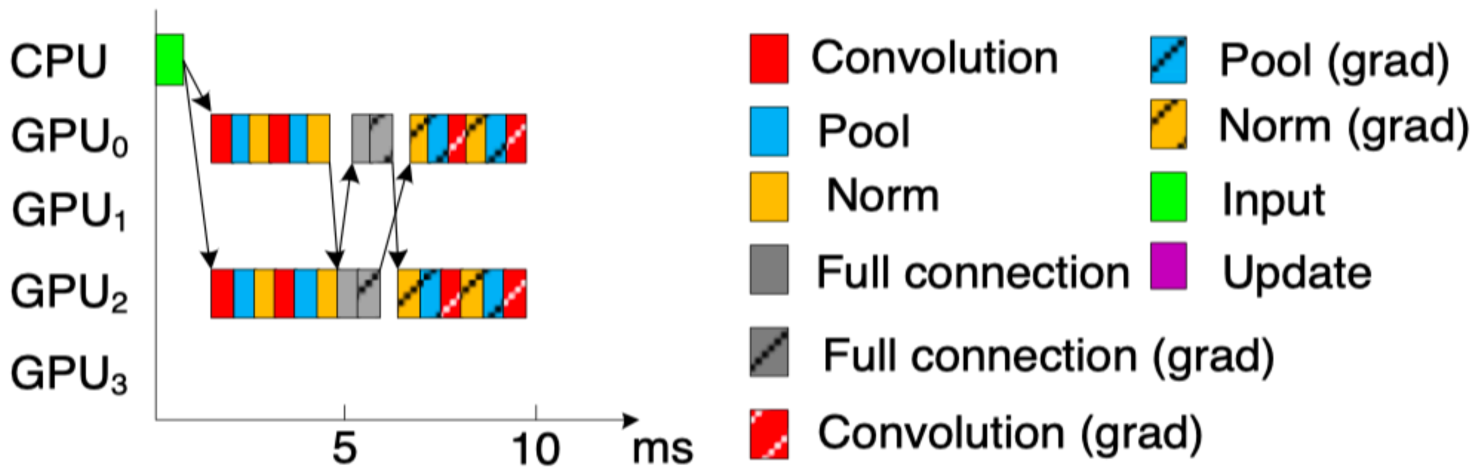


Figure 5. Performance in CIFAR-10 training.

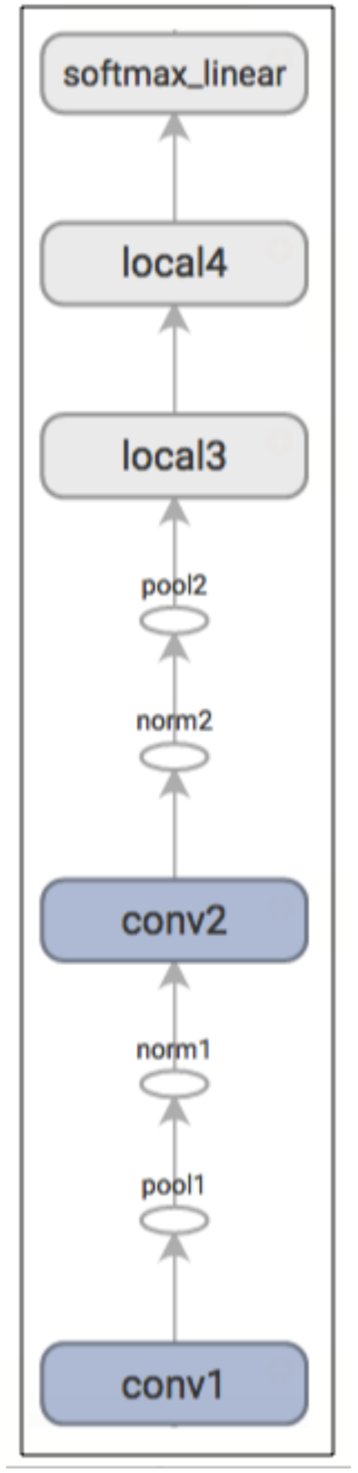
Train the model to achieve 87% accuracy.

Tricks learned by Spotlight

- Reduces inter-CPU communications with the insight that the inter-connected layers in CNNs are partially connected so that they can be split into two parts, each on a GPU, without introducing communication overhead.



(e) Spotlight



Hierarchical Learning for Device Placement

Azalia Mirhoseini*, Anna Goldie*, Hieu Pham, Benoit Steiner, Quoc V. Le, Jeff Dean

(*): Equal contribution

SUMMARY

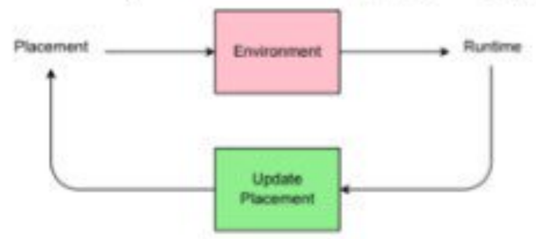
We propose a Reinforcement Learning algorithm that learns to automatically design model parallelism for TensorFlow graphs.

PROBLEM

- Given:
  - TensorFlow computational graph G with N ops
  - List of computing devices D (GPUs, CPUs, etc.)
- Find:
  - Placement P = {p_1, p_2, ..., p_N}, with $p_i \in D$
  - Minimizes the running time of G

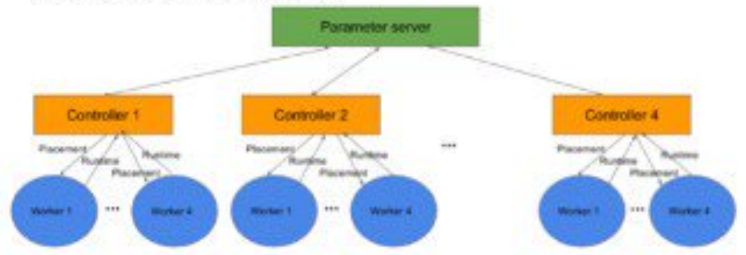
A REINFORCEMENT LEARNING APPROACH

- Using policy gradient to learn a policy π that:
  - Proposes placement and then measures runtime
  - Minimizes expected runtime $J(\theta_g, \theta_d) = \mathbf{E}_{\mathbf{P}(\mathbf{d};\theta_g,\theta_d)}[R_d]$



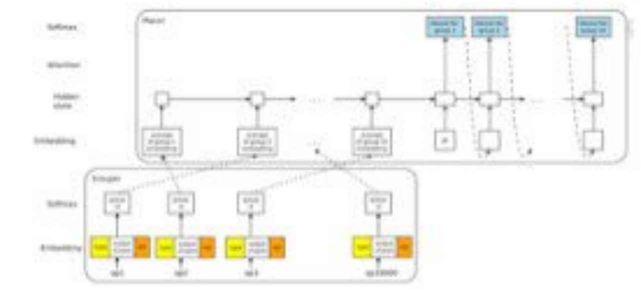
DISTRIBUTED TRAINING

- N controllers share a parameter server.
- Each controller sends placements to its children.
- Each child executes its placement.
- Each controller receives runtimes and updates the policy asynchronously.



MODEL

A two-level hierarchical network, consisting of a Grouper (which partitions the graph into groups) and a Placer (which places those groups onto devices)



TRAINING WITH REINFORCE

The goal is to minimize the expectation of runtime:

$$J(\theta_g,\theta_d) = \mathbf{E}_{\mathbf{P}(\mathbf{d};\theta_g,\theta_d)}[R_d] = \sum_{g\sim\pi_g}\sum_{d\sim\pi_d} p(g;\theta_g)p(d|g;\theta_d)R_d$$

$$\nabla_{\theta_g} J(\theta_g,\theta_d) = \sum_{g\sim\pi_g} \nabla_{\theta_g} p(g;\theta_g) \sum_{d\sim\pi_d} p(d|g;\theta_d)R_d$$

$$\approx \frac{1}{m}\sum_{g_i\sim\pi_g}^{1\le i\le m} \nabla_{\theta_g}\log p(g_i;\theta_g) . \frac{1}{k}(\sum_{d_j\sim\pi_d}^{1\le j\le k} R_{d_j})$$

$$\nabla_{\theta_d} J(\theta_g,\theta_d) = \sum_{d\sim\pi_d}\sum_{g\sim\pi_g} p(g;\theta_g)\nabla_{\theta_d} p(d|g;\theta_d)R_d$$

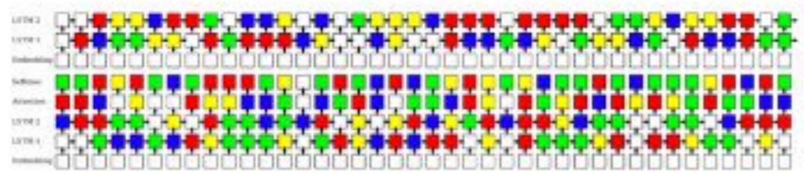
$$\approx \frac{1}{k}\sum_{d_j\sim\pi_d}^{1\le j\le k} \frac{1}{m}(\sum_{g_i\sim\pi_g}^{1\le i\le m} \nabla_{\theta_d}\log p(d_j|g_i;\theta_d)R_{d_j})$$

RESULTS

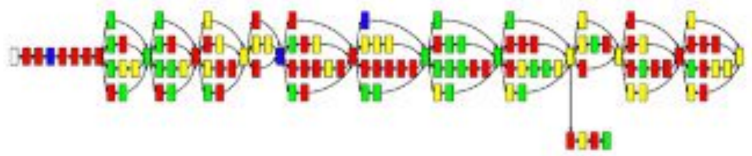
Tasks	CPU Only	GPU Only	#GPUs	Human Expert	Scotch	MinCut	Hierarchical Planner	Runtime Reduction
Inception-V3	0.61	0.15	2	0.15	0.93	0.82	**0.13**	16.3%
ResNet	-	1.18	2	1.18	6.27	2.92	**1.18**	0%
RNNLM	6.89	1.57	2	1.57	5.62	5.21	**1.57**	0%
NMT (2-layer)	6.46	OOM	2	2.13	3.21	5.34	**0.84**	60.6%
NMT (4-layer)	10.68	OOM	4	3.64	11.18	11.63	**1.69**	53.7%
NMT (8-layer)	11.52	OOM	8	**3.88**	17.85	19.01	4.07	-4.9%

EXAMPLE PLACEMENTS

- Each color is a GPU; transparent is the CPU.
- Neural Machine Translation with 2 layers

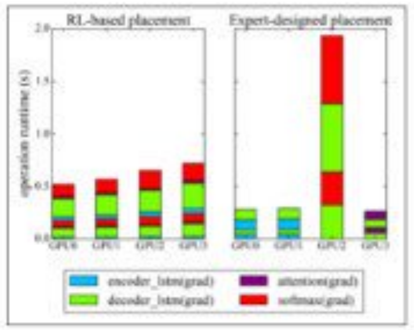


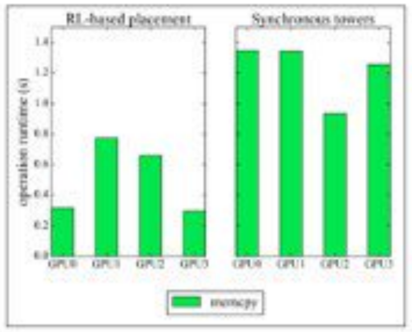
- Inception-V3



UNDERSTANDING THE PLACEMENTS

- Our method learns to optimize for different objectives for different models.
  - For RNNLM: learns that it is best to put all ops on a single GPU.
  - For NMT: learns to balance computation across devices.
  - For Inception-V3: learns to mitigate the time spent on inter-device memory copy.





On the left, we show the computational load profiling of NMT model for RL-based and expert-designed placements. Smaller blocks of each color correspond to forward pass and same-color upper blocks correspond to back-propagation. On the right, we show memory copy time profiling. All memory copy activities in Synchronous tower are between a GPU and a CPU, which are in general slower than GPU copies that take place in the RL-based placement.

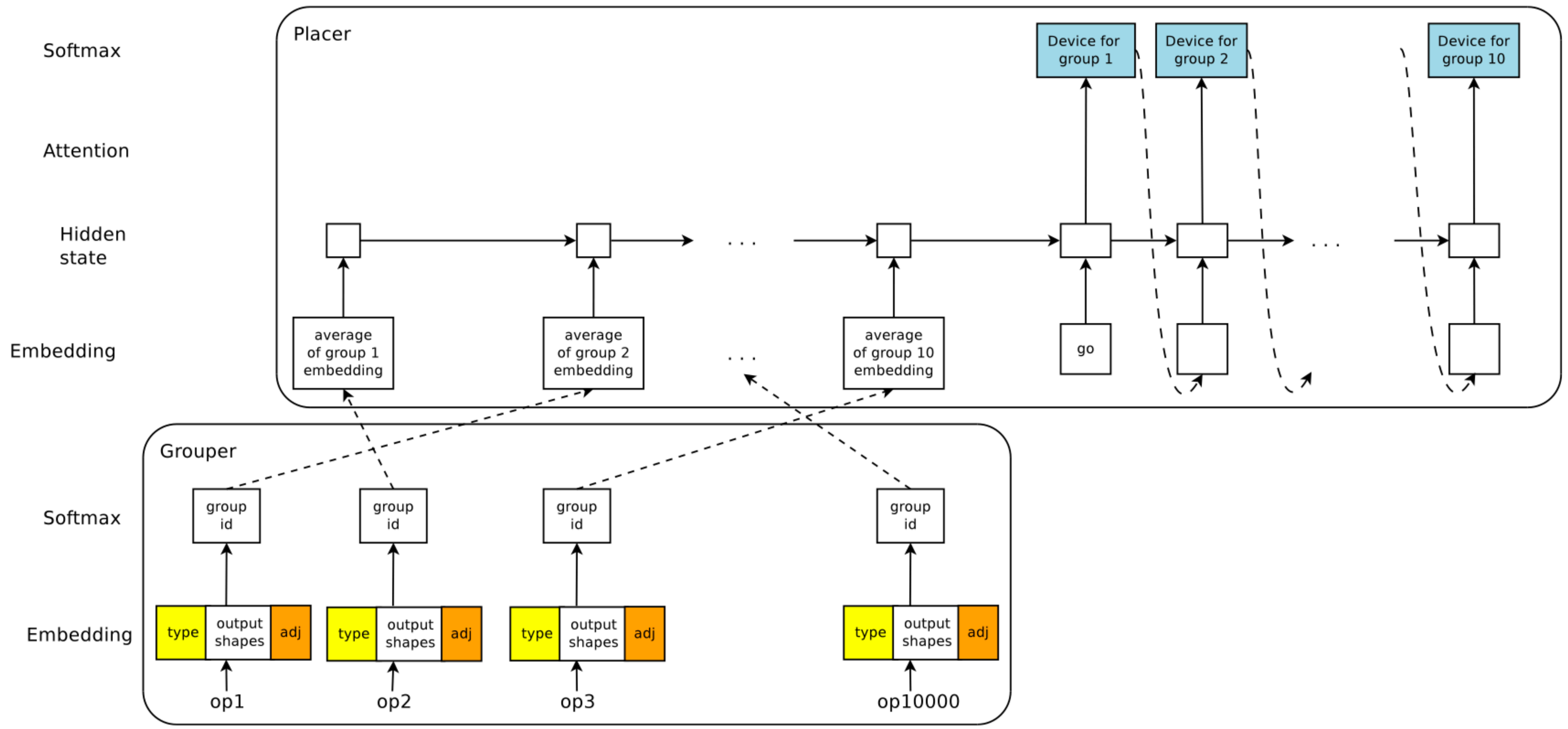


Figure 1: Hierarchical model for device placement (see text for more details).

Use neural network Grouper to do co-location.
Input: operation type information (e.g., MatMul, Conv2d, Sum, etc.); output shapes of operations; adjacency information for that operation.

The Hierarchical Planner places each step of an unrolled LSTM layer across multiple GPUs, whereas ColocRL colocates all operations in a step.

The Placer is a sequence-to-sequence model with Long Short-Term Memory and a content-based attention mechanism to predict the placements.