Device Placement Optimization using Reinforcement Learning

By Mirhoseini et al.

Shyam Tailor
21/11/18
The Problem

- Neural Networks are getting bigger and require greater resources for training and inference.
- Want to schedule in a **heterogeneous distributed environment**.
  - CPUs and GPUs in the paper.
  - All benchmarks run on a single machine.

- Traditionally: **use heuristics**
  - Previous automated approaches e.g. Scotch [3] do not work too well.

Figure from TensorFlow website.
This Paper’s Approach

- Use Reinforcement Learning to create the placements.
- Run placements in the real environment and measure their execution time as a reward signal.
- Use the evaluated reward signals to improve placement policy.
Revision: Policy Gradients

- We have parameterised policies $\pi_\theta$, where $\theta$ is the parameter.
- We want to pick a policy $\pi^*$ that maximises our reward $R(\tau)$.

- With policy gradients, we have an objective $J(\theta)$.

$$J(\theta) = E_{\tau \sim \pi_\theta(\cdot)}[R(\tau)]$$

- Use gradient descent to optimise $J(\theta)$ to find $\pi^*$.
  - Details out of scope but can be done using Monte Carlo Sampling.
$R(\mathcal{P}) = \textit{Square root}$ of total time for forward pass, backward pass, and parameter update.

- Sometimes placements just don’t run — have a large constant representing a failed placement.
- Square root to make training more robust.
- **Variance reduction**: take ten runs and discard the first.
Use an attentional sequence-to-sequence model which knows about devices that can be used for placements.

- **Input**: sequence of operations in the computation graph.
- **Output**: sequence of placements for the input operations.
Cutting Down the Search Space

- **Problem**: the computation graph can be very big.
- **Solution**: try to fuse portions of the graph as a pre-processing step where possible.

- Co-locate operations when it makes sense to.
  - e.g. if an operation’s output only goes to one other operation, keep them together.
  - Can be architecture specific too e.g. keeping LSTM cells together or keeping convolution / pool layers together.

- On evaluated networks, fused graph is around 1% the size of the original.
- To avoid bottleneck, distribute parameters to controllers.
- Controllers take samples, and instruct workers to run them.
Experiments involved 3 popular network architectures:

1. Recurrent Neural Network Language Model [5, 2].
2. Neural Machine Translation with Attention Mechanism [1].
3. Inception-V3 [4].

*Single* machine used to run experiments.
- Either 2 or 4 GPUs per machine for experiment purposes.
1. Run entire network on the CPU.
2. Run entire network on a *single* GPU.
3. Use Scotch to create a placement over the CPU and GPU.
   - Also run experiment without allowing the CPU.
4. Expert-designed placements from the literature.
Evaluation: How Fast are the RL Placements?

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

- Took between 12-27 hours to find placements.
Analysis: Why are the Placements Chosen Faster?

- The RL placements generally do a better job of distributing computation load and minimizing copying costs.
- **This is tricky** — and it’s different for different architectures!
  - Inception — it’s hard to exploit model parallelism due to dependencies restricting parallelism so try to *minimise copying*
  - NMT — the opposite applies, so balance computation load.
- It looks like RL can optimise around the tradeoff between computation and copying.
- The policy is learnt with nothing except the computation graph and the number of available devices.
Opinion: Positives

- This method shows promise, as it learns simple baselines automatically, and can exceed human performance where more advanced setup is required.
  - At least on the networks they tested it on.
- The technique was applied to different architectures, and positive results were obtained for each one.
- The technique should be generalisable to other system optimisation problems, in principle.
Opinion: Flaws in Evaluation

- Policy gradients are *stochastic* — so why haven’t multiple runs been reported?
- Is there a large variance between solutions found?
- Does the algorithm sometimes fail to converge to anything useful?
Opinion: Improvement — Post-Processing

- Is there low hanging fruit missed by the RL optimisation?
- The authors never attempt to interpret the placements beyond superficial comments about computation and copying.
Opinion: Improvement — Transfer Learning

- Each time the algorithm is run, it is learning about balancing copying and computation \textit{from scratch}.
- These concepts are not inherently unique to each network though — the precise tradeoffs may change, but the general concepts remain.


