TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

By Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, Alex Aiken

Samuel Stark
22/11/2021
University of Cambridge
• Deep Neural Networks can be expressed as a computational graph
• A fresh DNN may not be very performant
• DNNs can be optimized by substituting subgraphs for equivalent, faster ones

Example substitution chain on NasNet-A[1, Fig 7]
TASO Concept

- Previous work used *manual* substitutions

- 155 substitutions = 53KLoC in TensorFlow
  - Especially bad when new operators are created

- Substitutions are not verified, may be buggy

- The graph and the data layouts are optimized separately

Previous DNN optimization flow[1, Fig 1]
• TASO *automatically* generates substitutions

• 743 substitutions = 1KLoC in TASO

• Substitutions are formally proven to be correct

• The graph and data layouts are optimized *together*

TASO optimization flow[1, Fig 1]
Graph Substitution Generator

Goal: Find Equivalent Subgraphs

1. **Enumerate potential graphs**
   - Depth-first search, excluding duplicated computation

2. **Compute Fingerprint for each graph**
   - Hash outputs for constant integer input

3. **Test matching-Fingerprint pairs with more data**
   - Check with floating-point input, $\epsilon = 10^{-5}$
Graph Substitution Generator

Goal: Find Equivalent Subgraphs

1. Enumerate potential graphs
   - Depth-first search, excluding duplicated computation

2. Compute Fingerprint for each graph
   - Hash outputs for constant integer input

   \[ \text{hash}_{sym}(\{\text{hash}_{tensor}(t_i) \mid i \in \text{Outputs}\}) \]

3. Test matching-Fingerprint pairs with more data
   - Check with floating-point input, \( \epsilon = 10^{-5} \)
Goal: Find Equivalent Subgraphs

1. Enumerate potential graphs
   - Depth-first search, excluding duplicated computation

2. Compute Fingerprint for each graph
   - Hash outputs for constant integer input

3. Test matching-Fingerprint pairs with more data
   - Check with floating-point input, $\epsilon = 10^{-5}$
Graph Substitution Generator

Goal: Find Equivalent Subgraphs

Operations that produce zeroes need more special handling:

- **relu** often returns 0 for -ve units
  - Use a different non-linear function

- **enlarge** literally pads with 0
  - Only allow **enlarge** on inputs, not intermediate values
Goal: Remove Redundant/Overly Specific Substitutions

<table>
<thead>
<tr>
<th>Pruning Techniques</th>
<th>Remaining Substitutions</th>
<th>Reduction v.s. Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>28744</td>
<td>1×</td>
</tr>
<tr>
<td>Input tensor renaming</td>
<td>17346</td>
<td>1.7×</td>
</tr>
<tr>
<td>Common subgraph</td>
<td>743</td>
<td>39×</td>
</tr>
</tbody>
</table>

Table 3 from [1]
Goal: Remove Redundant/Overly Specific Substitutions

1. Remove substitutions that are identical other than input names

Figs 2a, 4a, b, c from [1]
Goal: Remove Redundant/Overly Specific Substitutions

2. Remove substitutions with common subgraphs

Figs 2a, 4a, b, c from [1]
Removing Redundancies

Goal: Remove Redundant/Overly Specific Substitutions

2. Remove substitutions with common subgraphs

Figs 2a, 4a, b, c from [1]
Graph Substitution Verifier

Goal: Formally Prove Substitutions are Equivalent

- Define a set of logical properties for each operator
  - 43 operators total

- Verify the operator properties hold
  - Use an SMT solver to verify the properties hold for a Python version

- Use properties to prove substitutions are equivalent
  - Use a theorem solver (Z3)

\[
\forall x. \text{transpose}(\text{transpose}(x)) = x \\
\forall x, y. \text{transpose}(\text{ewadd}(x, y)) = \text{ewadd}(\text{transpose}(x), \text{transpose}(y)) \\
\forall x, y. \text{transpose}(\text{ewmul}(x, y)) = \text{ewmul}(\text{transpose}(x), \text{transpose}(y)) \\
\forall x, w. \text{smul}(\text{transpose}(x), w) = \text{transpose}(\text{smul}(x, w))
\]

transpose is its own inverse operator commutativity
operator commutativity
operator commutativity

Table 2 from [1]
Goal: Find Optimal Graph with Substitutions

- Cost-Based Backtracking Search
  - Based on MetaFlow[2]

1. Pop graph off of priority queue
2. Try applying substitutions
3. Check costs of results
4. Push results onto queue
5. Repeat until queue is empty

- Hyperparameter $\alpha$ tunes backtracking
  - 1 = No backtracking
  - 1.05 chosen for evaluation

```
for substitution $s \in S$ do
  // LAYOUT($G, s$) returns possible layouts applying $s$ on $G$.
  for layout $l \in LAYOUT(G, s)$ do
    // APPLY($G, s, l$) applies $s$ on $G$ with layout $l$.
    $G' = APPLY(G, s, l)$
    if $G'$ is valid then
      if Cost($G'$) < Cost($G_{opt}$) then
        $G_{opt} = G'$
      if Cost($G'$) < $\alpha \times$ Cost($G_{opt}$) then
        $P$.enqueue($G'$)
```

Algorithm 1 from [1], based on [2]
TASO Cost Function

- TASO improves the cost function to include data layout
- \(\text{Cost}(\text{Operator, Layout})\) measured on-device
- Data Layout = Column-Major or Row-Major
- Consider each permutation of data layouts
- \(\text{Cost}(G) = \sum \text{Cost}(o_i, l_i)\)
TASO Cost Function

- TASO improves the cost function to include data layout

- **Cost(Operator, Layout)** measured on-device

- Data Layout = Column-Major or Row-Major

- Consider each permutation of data layouts

- **Cost(G) = \( \sum \text{Cost}(o_i, l_i) \)**

Fig 5 from [1]
TASO Cost Function

- TASO improves the cost function to include data layout

- \( \text{Cost}(\text{Operator}, \text{Layout}) \) measured on-device

- Data Layout = Column-Major or Row-Major

- Consider each permutation of data layouts

- \( \text{Cost}(G) = \sum \text{Cost}(o_i, l_i) \)

Fig 12 from [1]
• TASO evaluates cost from real-world performance

• This allows it to find optimal strategies which might be device-specific

• But this might prevent it from mapping to distributed computing

Fig 9 from [1]
Evaluation - Overall Optimization

- Consistently better performance than alternatives
  - Although they don’t specify alternative optimization configs

- Only 27/743 optimizations actually used...

Fig 7 from [1]
• Consistently better performance than alternatives
  • Although they don’t specify alternative optimization configs
  • Only 27/743 optimizations actually used...

Fig 10 from [1]
Where are they now?

- Repo has 480+ GitHub stars!

- Repo is basically dead.
  - Only bugfixes since 2019

- Paper has 42 citations, was directly followed up by first author
  - Pet (next presentation!) relaxes the need for completely equivalent transformations, and then strengthens it again.

- TensorFlow has stuck with Grappler[3]
  - Applies generic optimizations
  - e.g. constant folding
  - Similar to how compilers work
Summary

Pros

• Formal verification of substitutions
• Optimizing Layout + Graph together is very cool
• Low ratio of code/optimizations
• Produces good results!

Cons

• Lots of redundancy in generated substitutions
  • Only 27 end up used at all!
• Substitutions limited to size=4
• Doesn’t evaluate time taken to optimize
• Cost model = Sum, no parallelization
Summary

Pros

- Formal verification of substitutions
- Optimizing Layout + Graph together is very cool
- Low ratio of code/optimizations
- Produces good results!

Cons

- Lots of redundancy in generated substitutions
  - Only 27 end up used at all!
- Substitutions limited to size=4
- Doesn’t evaluate time taken to optimize
- Cost model = Sum, no parallelization

Questions?
References

