15-442/15-642: Machine Learning Systems

Introduction to Machine Learning Systems

Spring 2025

Tianqi Chen Carnegie Mellon University



An Overview of Machine learning Systems





ML Systems



Layer 1: Automatic Differentiation



HBN

HBM

Automatically construct backward computation graph



Forward computation graph

Backward computation graph



Layer 2: Graph-Level Optimizations



Automatic Differentiation

Graph-Level Optimization

Parallelization

Kernel Generation

Memory Optimization



Recap: DNNs as Computation Graphs

 Collection of simple trainable mathematical units that work together to solve complicated tasks



Graph-Level Optimizations







Input Computation Graph

transformations

Optimized Computation Graph

Example: Fusing Convolution and Batch Normalization



W, B, R, P are constant pre-trained weights

Fusing Conv and BatchNorm



 $B_2(n,c,h,w) = B(n,c,h,w) * R(c) + P(c)$

Current Rule-based Graph Optimizations Fuse conv + relu **TensorFlow currently** Fuse conv + includes ~200 rules batch normalization (~<u>53,000</u> LOC) Fuse multi. convs **Rule-based Optimizer**

<pre>namespace tensorflow { namespace graph_transforms {</pre>
<pre>// Converts Conv2D or MatMul ops followed by column-wise Muls into equivalent // ops with the Mul baked into the convolution weights, to save computation // during inference. Status EpidBatchNurms(const GranhDef& input granh def</pre>
<pre>const TransformFuncContext& context, GraphDef* output graph def) {</pre>
GraphDef replaced_graph_def; TF_RETURN_IF_ERROR(ReplaceMatchingOpTypes(input_graph_def, // clang-format off {"Mul", // mul_node
<pre>{ Conv2D MatMul DepthwiseConv2dNative", // conv_node</pre>
<pre>{ {"*"}, // input_node {"Const"}, // weights node</pre>
} },
{"Const"}, // mul_values_node } // clang_format_on
<pre>[](const NodeMatch& match, const std::set<string>& input_nodes,</string></pre>
<pre>// Check that nodes that we use are not used somewhere else. for (const auto& node : (conv_node, weights_node, mul_values_node)) { if (output_nodes.count(node.name())) { // Return original nodes. new_nodes->insert(new_nodes->end(), new_nodes->insert(new_nodes->end), insert(new_nodes->end), insert(new</pre>
<pre>mul_values_node}; input_node; weights_node; mul_values_node});</pre>
}
<pre>Tensor weights = GetNodeTensorAttr(weights_node, "value"); Tensor mul_values = GetNodeTensorAttr(mul_values_node, "value");</pre>
<pre>// Make sure all the inputs really are vectors, with as many entries as // there are columns in the weights. int64 weights_cols; if (conv_node.op() == "Conv2D") { weights_cols = weights.shape().dim_size(3); else if (conv_node.op() == "DepthwiseConv2dNative") { weights_cols = </pre>
<pre>weights.shape().dim_size(2) * weights.shape().dim_size(3); } else { veights_sels_supers() dim_size(1); }</pre>
<pre>if ((mul values.shape().dim() != 1) </pre>
<pre>(mul_values.shape().dim_size(0) != weights_cols)) { return errors::InvalidArgument("Mul constant input to batch norm has bad shape: ", mul_values.shape().DebugString()); }</pre>
// Multiply the original weights by the scale vector.
<pre>auto weights_vector = weights.flatfloat>(); Tensor scaled_weights(DT_FLOAT, weights.shape()); auto scaled_weights_vector = scaled_weights.flatfloat>(); for (int64 row = 0; row < weights_vector.dimension(0); ++row) { scaled_weights_vector(row) = weights_vector(row) = weights_vector(row) * weights_vector(row) * weights_vector(row) * weights_vector(row) * weights_vector(row) * weights_vector(row) * weights_vector(row) * weights_vector(row) *</pre>
// Construct the new nodes.
NodeDef scaled_weights_node; scaled_weights_node.set_op("Const"); scaled_weights_node.set_name(weights_node.name()); SetNodeAttr("dtype", DT_FLOAT, &scaled_weights_node); SetNodeTensorAttr <float-("value", &scaled_weights_node);<br="" scaled_weights,="">new_nodes->push_back(scaled_weights_node);</float-("value",>
<pre>new_nodes->push_back(input_node);</pre>
NodeDef new_conv_node; new_conv_node = conv_node; new_conv_node.set_name(mui_node.name()); new_nodes->push_back(new_conv_node);
<pre>return Status::OK(); }.</pre>
<pre>{; &replaced_graph_def); *output_graph_def = replaced_graph_def; return Status::OK(); }</pre>
REGISTER_GRAPH_TRANSFORM("fold_batch_norms", FoldBatchNorms);
<pre>} // namespace graph_transforms } // namespace tensorflow</pre>

10

Limitations of Rule-based Optimizations

Robustness

Experts' heuristics do not apply to all models/hardware



When I turned on XLA (TensorFlow's graph optimizer), the training speed is about 20% slower

🖄 stack overflow	Search
Home	Tensorflow XLA makes it slower?
PUBLIC	
Stack Overflow	I am writing a very simple tensorflow program with XLA enabled. Basically it's something like:
Tags	1 import tensorflow as tf
Users Jobs	<pre>def ChainSoftMax(x, n) tensor = tf.nn.softmax(x) for i in range(n-1): tensor = tf.nn.softmax(tensor) return tensor</pre>
Teams Q&A for work Learn More	<pre>config = tf.ConfigProto() config.graph_options.optimizer_options.global_jit_level = tf.OptimizerOptions.ON_1 input = tf.placeholder(tf.float32, [1000]) feed = np.random.rand(1000).astype('float32') with tf.Session(config=config) as sess: ref = ref = rm(choisecffMay(isput 2000) = food dist=figut; food))</pre>
	Basically the idea is to see whether XLA can fuse the chain of softmax together to avoid multiple kernel launches. With XLA on, the above program is almost 2x slower than that without XLA on a machine with a GPU card. In my gpu profile, I saw XLA produces lots of kernels named as " reduce_xxx " and " fusion_xxx " which seem to overwhelm the overall runtime. Any one know wha happened here?

With XLA, my program is almost 2x slower than without XLA

Limitations of Rule-based Optimizations

Robustness

Experts' heuristics do not apply to all models/hardware

Scalability

New operators and graph structures require more rules

TensorFlow currently uses ~4K LOC to optimize convolution

Limitations of Rule-based Optimizations

Robustness

Experts' heuristics do not apply to all models/hardware

Scalability

New operators and graph structures require more rules

Performance

Miss subtle optimizations for specific models/hardware

Motivating Example (ResNet*)



* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015



* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015



* Kaiming He. et al. Deep Residual Learning for Image Recognition, 2015

Motivating Example (ResNet*)





Infeasible to manually design graph optimizations for all cases

Automated Graph Optimizations

 Sraph Optimization Generator
 Image: Condidate Optimizations

 Mathematical Properties of ML
 Candidate Optimizations



Layer 3: Parallelizing ML Computations



Automatic Differentiation

Graph-Level Optimization

Parallelization

Kernel Generation

Memory Optimization



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

- 1. Forward propagation: apply model to a batch of input samples and run calculation through operators to produce a prediction
- 2. Backward propagation: run the model in reverse to produce error for each trainable weight
- 3. Weight update: use the loss value to update model weights



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

- 1. Forward propagation: apply model to a batch of input samples and run calculation through operators to produce a prediction
- 2. Backward propagation: run the model in reverse to produce error for each trainable weight
- 3. Weight update: use the loss value to update model weights



Recap: Stochastic Gradient Descent (SGD)

Train ML models through many iterations of 3 stages

- 1. Forward propagation: apply model to a batch of input samples and run calculation through operators to produce a prediction
- 2. Backward propagation: run the model in reverse to produce error for each trainable weight
- 3. Weight update: use the loss value to update model weights

$$w_i \coloneqq w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^n \nabla L_j(w_i)$$

How can we parallelize ML training?

$$w_i \coloneqq w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^n \nabla L_j(w_i)$$



1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Model Parallelism

• Split a model into multiple subgraphs and assign them to different devices





An Overview of Deep Learning Systems



Automatic Differentiation

Graph-Level Optimization

Parallelization

Kernel Generation

Memory Optimization



Kernel Generation: How to find performant programs for each operator?



Existing Approach: Engineer Optimized Tensor Programs

- Hardware vendors provide operator libraries manually developed by software/hardware engineers
- cuDNN, cuBLAS, cuRAND, cuSPARSE for GPUs
 - cudnnConvolutionForward() for convolution
 - cublasSgemm() for matrix multiplication

Issues:

- Cannot provide immediate support for new operators
- Increasing complexity of hardware -> hand-written kernels are suboptimal

Automated Code Generation





An Overview of Deep Learning Systems



Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

Code Optimization

Memory Optimization



GPU Memory is the Bottleneck in DNN Training

- The biggest model we can train is bounded by GPU memory
- Larger models often achieve better predictive performance
- Extremely critical for modern accelerators with limited on-chip memory





HBM

- 4 Automatic Differentiation
- 5 Optimizing Linear Algebra
- 6 GPU Programming (1)
- 7 GPU Programming (2)
- 8 ML Compilation (1)
- 9 ML Compilation (2)
- 10 Graph-level Optimization
- 11 Memory Optimizations
- 12 ML Parallelization (1)
- 13 ML Parallelization (2)