Automated Mixed-Precision for TensorFlow Training

Reed Wanderman-Milne (Google) and Nathan Luehr (NVIDIA)  
March 20, 2019
Mixed Precision Training Background

What is Mixed Precision?

Using a mix of float32 and float16 precisions

float16 is much faster on accelerators

Model parameters and some layers need float32 for numerical stability

Loss scaling needed to shift gradient computation into half representable range

Mixed precision improves performance by 1.5-3x on Volta GPUs
Mixed Precision Training Background

Mixed Precision in TensorFlow

tf.keras API

- Keras is the recommended API for training and inference in TensorFlow 2.0
- Allows direct control of layer types
- API not complete yet, but actively being worked on

Automatic Mixed Precision Graph Optimizer

- Single precision graph is converted to mixed precision at runtime
- Does not require tf.keras and will work with your existing TensorFlow 1.x models
Outline

Mixed Precision in tf.keras
  Model Construction
  Automatic Loss Scaling

Automatic Mixed Precision Graph Optimizer
  Graph conversion
  Automatic Loss Scaling

Results
Mixed Precision in tf.keras
tf.keras API

- Will just need one line:
  
  ```python
  tf.keras.mixed_precision.experimental.set_policy("default_mixed")
  ```

  ```python
  model = tf.keras.layers.Sequential()
  model.add(tf.keras.layers.Dense(32, activation="relu"))
  model.add(tf.keras.layers.Dense(32, activation="softmax"))
  model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
  ```

- TensorFlow will automatically choose what to do in each dtype
tf.keras Example
Model before mixed precision

Input layer

Dense layer 1
- MatMul
- Relu
- Var

Dense layer 2
- MatMul
- Softmax
- Var
tf.keras Example
Model after mixed precision
Passthrough Layers

For many layers, TensorFlow will infer the dtype from the input types

Cast + float16 execution may be slower than float32 execution.
If no float16 cast is needed, leave the layer in float16

```
x = tf.keras.layers.Input(() , dtype='float32')
y = tf.keras.layers.Add([x , x])  # float32
z = tf.cast(y , 'float16')
w = tf.keras.layers.Add([z , z])  # float16
```

If a layer is fed inputs of different types, it will upcast the lower precision inputs
Passthrough Layers

Example

In practice, our casting decisions tend to provide near optimal performance without reducing accuracy.

```python
x = tf.keras.layers.Input(()
)x = tf.keras.layers.Dense(10)(x)  # Dense chooses float16
y = tf.keras.layers.Dense(10)(x)
# Add does not choose, so will infer float16 from inputs
z = tf.keras.layers.Add([x, y])
```

Add is done in float16, which is likely the right choice.

Note, if the second line was removed, Add would be done in float32 due type promotion. This can be suboptimal, but we err on side of caution.
How to Override TensorFlow’s Decisions

Option 1: Pass an explicit dtype

tf.keras.mixed_precision.experimental.set_policy("default_mixed")

model = tf.keras.layers.Sequential()
model.add(tf.keras.layers.Dense(32, activation="relu",
                               dtype="float32"))
model.add(tf.keras.layers.Dense(32, activation="softmax"))
model.compile(optimizer="rmsprop",
               loss="categorical_crossentropy",
               metrics=["accuracy"])


How to Override TensorFlow’s Decisions

Option 2: Set the policy

```python
model = tf.keras.layers.Sequential()

model.add(tf.keras.layers.Dense(32, activation="softmax"))

model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])
```
User Defined Layers

- If you write a layer, you can adjust the casting behaviour
  - Just need to override the ‘cast_inputs’ method of a layer
- For example, to define a layer that is done in float16 when mixed precision is enabled

```python
def cast_inputs(self, inputs):
    return self._mixed_precision_policy.cast_to_lowest(inputs)
```

- Variables will be created in float32 and automatically cast to float16 as needed
class CustomBiasLayer(tf.keras.layers.Layer):
    def build(self, _):
        self.v = self.add_weight('v', ())
        self.built = True

    def call(self, inputs)
        return inputs + self.v

    def cast_inputs(self, inputs):
        # Casts to float16, the policy's lowest-precision dtype
        return self._mixed_precision_policy.cast_to_lowest(inputs)
Automatic Loss Scaling

- tf.keras API will automatically enable dynamic loss scaling
  - Loss scale will be doubled every 2000 steps
  - Loss scale will half if any NaNs or Infs are found in the gradients
- Can optionally customize loss scaling behavior:

```
# Dynamic loss scaling, tripling the loss scale every 1000 steps
params = tf.keras.mixed_precision.DynamicLossScaleParameters(
    incr_every_n_steps=1000, loss_scale_multiplier=3)
policy = tf.keras.mixed_precision.Policy("default_mixed", loss_scale=params)
tf.keras.mixed_precision.experimental.set_policy(policy)
```

```
# Fixed loss scale of 128
policy = tf.keras.mixed_precision.Policy("default_mixed", loss_scale=128)
tf.keras.mixed_precision.experimental.set_policy(policy)
```
tf.keras API Roadmap

● Basic functionality (available in nightly builds)
  ○ Variables created in float32 and automatically cast to required dtype
  ○ User must cast model inputs to float16 and outputs to float32
  ○ User must explicitly wrap optimizer to enable loss scaling

● In upcoming months, the final API will require just one line
  ○ tf.keras.mixed_precision.experimental.set_policy("default_mixed")
  ○ Will have public RFC in tensorflow/community GitHub repo -- feel free to comment
  ○ Final API may be slightly different than what was described here
Automatic Mixed Precision Graph Optimizer
x = tf.placeholder(tf.float32, shape=(1024, 1024))
w = tf.get_variable('w', shape=(1024, 1024))
z = tf.add(x, tf.matmul(x, w))
Transformed Graphs

VariableV2
  FP32

Identity
  FP32

Cast
  FP32 to FP16

MatMul
  FP16

Add
  FP16

Placeholder
  FP32

Cast
  FP32 to FP16
Enabling AMP Graph Pass
Preview Feature in NGC 19.03 TensorFlow Container

Designed to work with existing float32 models with minimal changes.

If your training script uses a `tf.train.Optimizer` to compute and apply gradients
Both Loss Scaling and mixed precision graph conversion can be enabled with a single env var.

```bash
export TF_ENABLE_AUTO_MIXED_PRECISION=1
python training_script.py
```

If your model does not use a `tf.train.Optimizer`, then
You must add loss scaling manually to your model
Then enable the grappler pass as follows

```bash
export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
python training_script.py
```
Enabling AMP Graph Pass

Coming Soon ...

Preview implementation

- Does not work with Distribution Strategies
- Provides a single hard-coded loss scaling implementation

A more complete and flexible implementation is being upstreamed now.

```
opt = tf.train.GradientDescentOptimizer(0.001)
opt = tf.mixed_precision.experimental.mixed_precision_optimizer(opt, 1000.)
```

This enables both loss scaling and mixed precision graph optimizer.
Choosing What to Cast

Guiding Principles

1. Use float16 as much as possible, particularly for ops that can run on Tensor Cores

2. Use float32 where needed to maintain full accuracy (e.g., master weights and loss functions)

3. Minimize “cast thrashing” between float16 and float32
Choosing What to Cast
Categorize Ops into 3+1 Categories

**Always Cast:** Ops highly accelerated by float16. These always justify performance costs of casting inputs. Examples: MatMul and Conv2d.

**Maybe Cast:** Ops available for float16 execution but not accelerated sufficiently to justify casting overhead on their own. Examples: Add and Relu.

**Never Cast:** Ops requiring float32 evaluation in order to maintain numerical stability. Examples: Exp and SoftmaxCrossEntropyWithLogits.

**Everything Else:** Ops lacking float16 implementations or operating on non-floating point inputs.
Graph Coloring Example

Step 1: Initialize Op Colors
Graph Coloring Example
Step 2: Propagate ‘Never’ Tags Forward

[Diagram with nodes and edges representing the operations and data flow]
Graph Coloring Example

Step 3: Paint ‘Maybe’ Ops Bounded by ‘Always’
Graph Coloring Exmple
Step 4: Find boundaries of ‘always’ sections
Graph Coloring Example

Step 5: Insert casts (with reuse)
AMP LARGELY DEPLOYED INTO ALIBABA PAI PLATFORM

AMP + Automatic Loss Scaling Optimizer

- No laborious FP32/FP16 casting work anymore
- Already supporting diversified internal workloads: NLP/CNN/BERT/Sparse Embedding/…
- 1.3~3X time-to-accuracy speed-up
- Collaboratively work with NVIDIA to push into TF community
AMP LARGELY DEPLOYED INTO ALIBABA PAI PLATFORM

More than 10,000 training jobs have benefited from AMP over last half a year.

<table>
<thead>
<tr>
<th>Model Class</th>
<th>Details</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>Data Mining/Content Mining</td>
<td>2.4-2.9X</td>
</tr>
<tr>
<td>CNN</td>
<td>Image Understanding/Image Recognition/Video Recommendation</td>
<td>1.4-1.5X</td>
</tr>
<tr>
<td>Transformer</td>
<td>NLP</td>
<td>1.5X</td>
</tr>
<tr>
<td>BERT</td>
<td>NLP/Knowledge Graph</td>
<td>1.5-1.6X</td>
</tr>
</tbody>
</table>
ResNet50 v1.5
Training on a single V100

![Graph showing performance comparison between Float32 BS=128 and Enable AMP. The graph indicates a 2.1x performance increase with AMP enabled.]
ResNet50 v1.5
Training on a single V100

- Float32 BS=128: 34
- Enable AMP: 2.1x
- Enable XLA: 2.8x
ResNet50 v1.5
Training on a single V100

Images / Second

- Float32 BS=128
- Enable AMP: 2.1x
- Enable XLA: 2.8x
- Boost BS to 256: 3.3x
Additional Results
V100 Training Speedups

AMP Resources

NVIDIA NGC 19.03 TensorFlow Container
https://ngc.nvidia.com/catalog/containers/nvidia:tensorflow

Example Models

AMP Graph Optimizer PR
https://github.com/tensorflow/tensorflow/pull/26342