GETTING MORE DL TRAINING WITH TENSOR CORES AND AMP
AGENDA

• What is Automatic Mixed Precision
• AMP Technical Details
• Getting started on AMP
• Performance Guide
• Additional Resources
BACKGROUND: TENSOR CORES

Hardware support for accelerated 16-bit FP math

- 125 TFlops in FP16 vs 15.7 TFlops in FP32 (8x speed-up)
- Inherently mixed precision: 32bit accumulation
- Available in Volta and Turing architecture GPUs
- Optimized 4x4 dot operation (GEMM)

Memory Savings
- Half Storage Requirements (larger batch size)
- Half the memory traffic by reducing size of gradient/activation tensors
WHAT IS AUTOMATIC MIXED PRECISION
MAXIMIZING MODEL PERFORMANCE

FP16 is fast and memory-efficient.

FP32

1x compute throughput
1x memory throughput
1x memory storage

FP16 with Tensor Cores

8X compute throughput
2X memory throughput
1/2X memory storage
MIXED PRECISION TRAINING

Motivation

• Balance a pure tradeoff of speed and accuracy:
  • Reduced precision (16-bit floating point) for speed or scale
  • Full precision (32-bit floating point) to maintain task-specific accuracy

• Under the constraints:
  • Maximize use of reduced precision while matching accuracy of full precision training
  • No changes to hyperparameters

This easy integration enables TensorFlow developers to literally flip a switch in their AI model and get up to 3X speedup with mixed precision training while maintaining model accuracy.

Rajat Monga, Engineering Director, TensorFlow
AUTOMATIC MIXED PRECISION IN TENSORFLOW

Upto 3X Speedup

TensorFlow Medium Post: Automatic Mixed Precision in TensorFlow for Faster AI Training on NVIDIA GPUs

All models can be found at:

All performance collected on 1xV100-16GB, except bert-squadqa on 1xV100-32GB.

Speedup is the ratio of time to train for a fixed number of epochs in single-precision and Automatic Mixed Precision. Number of epochs for each model was matching the literature or common practice (it was also confirmed that both training sessions achieved the same model accuracy).

Batch sizes: rn50 (v1.5): 128 for FP32, 256 for AMP+XLA; ssd-rn50-fpn-640: 8 for FP32, 16 for AMP+XLA; NCF: 1M for FP32 and AMP+XLA; bert-squadqa: 4 for FP32, 10 for AMP+XLA; GNMT: 128 for FP32, 192 for AMP.
MIXED PRECISION TRAINING
With Tensor Cores

- 8GPU training of ResNet-50 (ImageNet classification) on DGX-1
  - NVIDIA mxnet-18.08-py3 container
- Total time to run full training schedule in mixed precision is well under four hours
  - 2.9x speedup over FP32 training
  - Equal validation accuracies
  - No hyperparameters changed
    - Minibatch = 256 per GPU
**MIXED PRECISION IS GENERAL PURPOSE**
Models trained to match FP32 results (same hyperparameters)

<table>
<thead>
<tr>
<th>Image Classification</th>
<th>Detection / Segmentation</th>
<th>Generative Models (Images)</th>
<th>Language Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>DeepLab</td>
<td>DLSS</td>
<td>BERT</td>
</tr>
<tr>
<td>DenseNet</td>
<td>Faster R-CNN</td>
<td>Partial Image Inpainting</td>
<td>BigLSTM</td>
</tr>
<tr>
<td>Inception</td>
<td>Mask R-CNN</td>
<td>Progress GAN</td>
<td>8k mLSTM (NVIDIA)</td>
</tr>
<tr>
<td>MobileNet</td>
<td>Multibox SSD</td>
<td>Pix2Pix</td>
<td></td>
</tr>
<tr>
<td>NASNet</td>
<td>NVIDIA Automotive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNet</td>
<td>RetinaNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResNeXt</td>
<td>UNET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VGG</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCeption</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Recommendation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeepRecommender</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NCF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Speech</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deep Speech 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tacotron</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WaveNet</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>WaveGlow</td>
<td></td>
</tr>
</tbody>
</table>

**Speech**

**Translation**

FairSeq (convolution)

GNMT (RNN)

Transformer (self-attention)
## MIXED PRECISION SPEEDUPS

Not limited to image classification

<table>
<thead>
<tr>
<th>Model</th>
<th>Category</th>
<th>FP32 -&gt; M.P. Speedup</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResNet-50 v1.5</td>
<td>Image Recognition</td>
<td>3.5x</td>
<td>Iso-batch size</td>
</tr>
<tr>
<td>FairSeq Transformer</td>
<td>Translation</td>
<td>2.9x</td>
<td>Iso-batch size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.9x</td>
<td>2x lr + larger batch</td>
</tr>
<tr>
<td>BERT</td>
<td>SQuAD (fine-tuning)</td>
<td>1.9x</td>
<td>Iso-batch size</td>
</tr>
<tr>
<td>Deep Speech 2</td>
<td>Speech recognition</td>
<td>4.5x</td>
<td>Larger batch</td>
</tr>
<tr>
<td>Tacotron 2 + WaveGlow</td>
<td>Speech synthesis</td>
<td>1.2x</td>
<td>Iso-batch size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.9x</td>
<td></td>
</tr>
<tr>
<td>Nvidia Sentiment</td>
<td>Language modeling</td>
<td>4.0x</td>
<td>Larger batch</td>
</tr>
<tr>
<td>NCF</td>
<td>Recommender</td>
<td>1.8x</td>
<td>Iso-batch size</td>
</tr>
</tbody>
</table>

trained to **SAME ACCURACY** as FP32 model

**No hyperparameter changes**, except as noted
TECHNICAL DETAILS OF MIXED PRECISION
MIXED PRECISION TRAINING

Three-part methodology

- **Model conversion:**
  - Switch everything to run on FP16 values
  - Insert casts to FP32 for loss functions and normalization/pointwise ops that need full precision

- **Master weights:**
  - Keep FP32 model parameters, update at each iteration
  - Use an FP16-casted copy for both forward pass and backpropagation

- **Loss scaling:**
  - Scale the loss value, un-scale the gradients (in FP32!)
  - Check gradients at each iteration for overflow - adjust loss scale and skip update, if needed
MIXED PRECISION TRAINING
Assign each operation its optimal precision & performance

Before Mixed Precision

- **GEMMs**, **Convolutions** can use Tensor Cores
- Most **pointwise ops** (e.g. add, multiply): 1/2X memory storage for intermediates, 2X memory throughput

- **Weight** updates benefit from precision
- **Loss** functions (often reductions) benefit from precision and range
- **Softmax**, norms, some other ops benefit from precision and range
MIXED PRECISION TRAINING
Assign each operation its optimal precision & performance

- GEMMs, Convolutions can use Tensor Cores
- Most pointwise ops (e.g. add, multiply): 1/2X memory storage for intermediates, 2X memory throughput
- Weight updates benefit from precision
- Loss functions (often reductions) benefit from precision and range
- Softmax, norms, some other ops benefit from precision and range
AUTOMATIC MIXED PRECISION

Concepts

• Allows to implement the three part methodology **automatically**:

  • The framework software can transform existing model code to run with mixed precision fully automatically

  • No new code required should result in no new bugs

• Two components:

  • **Automated casting**: operation-level logic to insert casts between FP32 and FP16, transparent to the user

  • **Automatic loss scaling**: wrapper class for the optimizer object the can scale the loss, keep track of the loss scale, and skip updates as necessary
AUTOMATIC CASTING

Operation Classification

- Divide the universe of operations into three kinds
  - Whitelist: ops for which FP16 can use Tensor Cores (MatMul, Conv2d)
  - Blacklist: ops for which FP32 is required for accuracy (Exp, Sum, Softmax)
  - Everything else: ops that can run in FP16, but only worthwhile if input is already FP16 (ReLU, pointwise Add, MaxPool)

- Lists are framework-specific, since each framework has its own abstractions
  - PyTorch: https://github.com/NVIDIA/apex/tree/master/apex/amp/lists
**KEEP FP32 MASTER WEIGHTS**

- At each iteration of training, perform a *weight update* of the form $w_{t+1} = w_t - \alpha \nabla_t$
  - $w_t$'s are weights; $\nabla_t$'s are gradients; $\alpha$ is the learning rate
- As a rule, gradients are smaller than weights, and learning rate is less than one
- Consequence: weight update *can be* a no-op, since you can’t get to next representable value
- Conservative solution: keep a high-precision copy of weights so small updates accumulate across iterations
LOSS SCALING

Precision of Weights & Gradients

Gradients Vanishing

Loss Scale!!

Loss Distribution

Percentage of all activation gradient values

log$_2$(magnitude)

FP16 Representable range

FP16 denoms

Become zero in FP16

Weights

Activations

Weight Grads

Activation Grads
AUTOMATIC LOSS SCALING

- All frameworks implement some kind of optimizer wrapper
  - Internally, tracks the current loss scale and history of overflows
  - Provides the loss scale when needed for backpropagation
  - Overrides optimizer step to instead checking overflow
    - No overflow, then increase grads and pass to wrapped optimizer step
    - overflow, then decrease loss scale and don’t called wrapped optimizer step
AUTOMATIC LOSS SCALE

- Internally, tracks the current loss scale and history of overflows
- Provides the loss scale when needed for backpropagation
  - If an Inf or a NaN is present in the gradient, decrease the scale
    *And* skip the update, including optimizer state
  - If no Inf or NaN has occurred for some time, increase the scale
MIXED PRECISION MAINTAINS ACCURACY

Benefit From Higher Throughput Without Compromise

Mixed Precision - Same hyperparameters and learning rate schedule as FP32 ILSVRC12 classification top-1 accuracy.
(Sharan Narang, Paulius Micikevicius et al., "Mixed Precision Training", ICLR 2018)
GETTING STARTED AUTO MIXED PRECISION
**AUTOMATIC MIXED PRECISION**

Easy to Use, Greater Performance and Boost in Productivity

Insert ~ two lines of code to introduce Automatic Mixed-Precision and get upto 3X speedup

AMP uses a graph optimization technique to determine FP16 and FP32 operations

Support for TensorFlow, PyTorch and MXNet

Unleash the next generation AI performance and get faster to the market!

TENSORFLOW AMP
A simple method

- Designed to work with existing float32 models, with minimal changes
- Support since NGC TensorFlow Container 19.03
- 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.

```
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

```
export TF_ENABLE_AUTO_MIXED_PRECISION_GRAPH_REWRITE=1
python training_script.py
```
Since NGC TensorFlow Container 19.07 (TF 1.14+)

- Supports an explicit optimizer wrapper to perform loss scaling
- Enables auto casting / loss scaling and mixed precision graph optimizer

```python
import tensorflow as tf

opt = tf.train.GradientDescentOptimizer(0.5)

opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
from apex import amp
N, D_in, D_out = 64, 1024, 512
x = torch.randn(N, D_in, device="cuda")
y = torch.randn(N, D_out, device="cuda")

model = torch.nn.Linear(D_in, D_out).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")

for _ in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)
    optimizer.zero_grad()
    with amp.scale_loss(loss, optimizer) as scaled_loss:
        scaled_loss.backward()
    optimizer.step()

* Use apex.parallel.DistributedDataParallel for multi-GPU training
PYTORCH OPTIMIZATION LEVEL

**OPT_LEVEL=“O0”**

FP32 training.
Your incoming model should be FP32 already, so this is likely a no-op. O0 can be useful to establish an accuracy baseline.

**O1**

Mixed Precision
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. Because cats occur in functions, model weights remain FP32.

**O2**

“Almost FP16” Mixed Precision.
FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. Model weights, except batchnorm weights, are cast to FP16.

**O3**

FP16 training.
O3 can be useful to establish the “speed of light” for your model. If your model uses batch normalization, and keep_batchnorm_fp32=True, which enables cudnn batchnorm.
## PyTorch Optimization Level

### OPT_LEVEL="O0"

**FP32 training.**
Your incoming model should be FP32 already, so this is likely a no-op. **O0** can be useful to establish an accuracy baseline.

### O1

**Mixed Precision**
Patches Torch functions to internally carry out Tensor Core-friendly ops in FP16, and ops that benefit from additional precision in FP32. Also uses dynamic loss scaling. Because cats occur in functions, model weights remain FP32.

### O2

**“Almost FP16” Mixed Precision.**
FP16 model and data with FP32 batchnorm, FP32 master weights, and dynamic loss scaling. **Model weights, except batchnorm weights, are cast to FP16.**

### O3

**FP16 training.**
**O3** can be useful to establish the “speed of light” for your model. If your model uses batch normalisation, and keep_batchnorm_fp32=True, which enables cudnn bachnorm.
net = get_network()
trainer = mx.gluon.Trainer(...)

for data in dataloader:
    with autograd.record(True):
        out = net(data)
        l = loss(out, label)

        autograd.backward(scaled_loss)
        trainer.step()
From mxnet.contrib import amp
amp.init()
net = get_network()
trainer = mx.gluon.Trainer(…)
amp.init_trainer(trainer)
for data in dataloader:
    with autograd.record(True):
        out = net(data)
l = loss(out, label)
    with amp.loss_scale(loss, trainer) as scaled_loss:
        autograd.backward(scaled_loss)
trainer.step()
TENSOR CORES
PERFORMANCE GUIDE
GETTING MORE FROM TENSOR CORES

- **Matrix Multiplication**
  - All the dimensions (M, N, K) should be multiples of 8
- **Recommended to be a multiple of 8**
  - Input size, output size, batch size
  - Linear layer dimensions
  - Convolution layer channel counts (NCHW format)
  - Pad the sequence length For sequence problems
- **Ensure good Tensor Cores GEMM efficiency**
  - Choose the above dimensions as multiples of 64/128/256
- **Finally, Double** the batch size
AM I USING TENSOR CORES?"

- cuBLAS and cuDNN are optimized for Tensor Cores, coverage always increasing
- Run with nvprof and look for “s[some digits]” in kernel name
  - Eg: volta_fp16_s884gemm_fp16_128x128_ldg8_f2f_nn
BERT FP32 BENCHMARK
HuggingFace’s pretrained BERT

API List

~60% GEMM

~460 ms
BERT FP16 BENCHMARK

HuggingFace’s pretrained BERT

~222 ms

2.1x Speed up
NVIDIA NGC MODEL SCRIPTS
Tensor Core Optimized Deep Learning Examples

16 Available today!
- Tensor Core optimized for greater performance
- Test drive automatic mixed precision
- Actively updated by NVIDIA
- State-of-the-art accuracy using Tensor Cores
- Serves as a reference implementation
- Exposes hyperparameters and source code for further adjustment

Accessible via:
- NVIDIA NGC https://ngc.nvidia.com/catalog/model-scripts
- GitHub https://www.github.com/NVIDIA/deeplearningexamples
- NVIDIA NGC Framework containers https://ngc.nvidia.com/catalog/containers
NVIDIA NGC MODEL SCRIPTS

Tensor Core Examples Built for Multiple Use Cases and Frameworks

A dedicated hub to download Tensor Core Optimized Deep Learning Examples on NGC

MODEL SCRIPTS FOR VARIOUS APPLICATIONS

https://developer.nvidia.com/deep-learning-examples

**Computer Vision**
- SSD PyTorch
- SSD TensorFlow
- UNET-Industrial TensorFlow
- UNET-Medical TensorFlow
- ResNet-50 v1.5 MXNet
- ResNet-50 PyTorch
- ResNet-50 TensorFlow
- Mask R-CNN PyTorch

**Speech & NLP**
- GNMT v2 TensorFlow
- GNMT v2 PyTorch
- Transformer PyTorch
- **BERT** (Pre-training and Q&A) TensorFlow

**Recommender Systems**
- NCF PyTorch
- NCF TensorFlow

**Text to Speech**
- Tacotron2 and WaveGlow PyTorch
## ENABLING AUTOMATIC MIXED PRECISION

Add Just A Few Lines of Code, Get Upto 3X Speedup

### TensorFlow

- `os.environ['TF_ENABLE_AUTO_MIXED_PRECISION'] = '1'`
- OR
- `export TF_ENABLE_AUTO_MIXED_PRECISION=1`

NVIDIA Container 19.07+ and TF 1.14+, explicit optimizer wrapper available:

```
opt = tf.train.experimental.enable_mixed_precision_graph_rewrite(opt)
```

### PyTorch

```
model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()
```

### MXNet

```
amp.init()
amp.init_trainer(trainer)
with amp.scale_loss(loss, trainer) as scaled_loss:
    autograd.backward(scaled_loss)
```

GTC SESSION RECORDINGS 2019

Recommended on-demand-gtc.gputechconf.com Talks

- Overview
  - (E8494) Mixed precision training with Deep Neural Networks
  - (S91022) Text-to-speech: Overview of latest research using Tacotron and Waveglow

- PyTorch
  - (S9998) Automatic Mixed Precision in PyTorch
  - (S9832) Taking advantage of mixed precision to accelerate training in PyTorch

- TensorFlow
  - (S91029) Automatic mixed precision tools for TensorFlow Training

- MXNet
  - (S91003) MXNet Computer Vision and Natural Language Processing Models Accelerated with NVIDIA Tensor Cores
TAKEAWAY

- Getting **3x math performance** with Tensor Cores
- Reduced memory usage → larger batch size
- Achieves the **same accuracy** of FP32 training
- Just need a 2-3 lines of codes
- **16 samples** in NGC, nvidia/DeepLearningExamples in CV, NLP, Speech, and Recommendation
- Increasing providing samples, we do first for you

GETTING MORE IN TRAINING

Deep learning training acceleration

- Deep Learning Research of NAVER Clova for AI-Enhanced Business
  - Mixed Precision’s contribution to LarVa (Language Representations by Clova) research
  - Track 1, Session 2 (13:50 - 14:30), 하정우 리더 (CLOVA AI Research 리더)

- GPU를 활용한 Image Augmentation 가속화 방안 - DALI
  - Track 1, Session 4 (15:40 - 16:20), 한재근 과장 (NVIDIA Solutions Architect)

- GPU Profiling 기법을 통한 Deep Learning 성능 최적화 기법 소개
  - Track 3, Session 5 (16:30 - 17:10), 홍광수 과장 (NVIDIA Solutions Architect)
SUPPLEMENTS: TENSOR CORES API
TENSOR CORE
Mixed Precision Matrix Math
4x4 matrices

\[
D = \begin{pmatrix}
A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\
A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\
A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\
A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3}
\end{pmatrix}
\begin{pmatrix}
B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\
B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\
B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\
B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3}
\end{pmatrix}
+ \begin{pmatrix}
C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\
C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\
C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\
C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3}
\end{pmatrix}
\]

\[
D = AB + C
\]
TENSOR SYNCHRONIZATION
Full Warp 16x16 Matrix Math

Warp-synchronizing operation
Composed Matrix Multiply and Accumulate for 16x16 matrices
Result distributed across warp
CUDA TENSOR CORE PROGRAMMING
16x16x16 Warp Matrix Multiply and Accumulate (WMMA)

\[ D = AB + C \]
WARP MATRIX API

Overview

Introduced in Volta as an abstraction layer

Provides CUDA C++ API that:

- Defines fragment abstraction (array of values)
- Load Matrix A/B from SMEM to Registers
- Perform the *MMA operation
- Store Accumulators from Registers to SMEM
- More information in Programming Guide

\[ D = AB + C \]
API operations now include 8-bit integer

- Turing (sm_75) only
- Signed & unsigned 8-bit input
- 32-bit integer accumulator
- Match input/output dimensions with `half`
- Experimental Sub-Byte Operations (4-bit, 1-bit)
SUPPLEMENTS: MIXED PRECISION PERFORMANCE
**IMAGE CLASSIFICATION: MXNet ResNet-50 v1.5**

https://ngc.nvidia.com/catalog/model-scripts/nvidia:resnet_50_v1_5_for_mxnet

<table>
<thead>
<tr>
<th>DGX-1V 8GPU 16G</th>
<th>MXNet ResNet FP32</th>
<th>MXNet ResNet Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train [Hours]</td>
<td>11.1</td>
<td>3.3</td>
</tr>
<tr>
<td>Train Accuracy Top 1%</td>
<td>76.67%</td>
<td>76.49%</td>
</tr>
<tr>
<td>Perf.</td>
<td>2,957 Img/sec</td>
<td>10,263 Img/sec</td>
</tr>
</tbody>
</table>

**Data set**

ImageNet

Source: https://github.com/NVIDIA/DeepLearningExamples/tree/master/MxNet/Classification/RN50v1.5

GPU: 1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)
# SPEECH SYNTHESIS: Tacotron 2 And WaveGlow v1.0


## Tacotron 2 and WaveGlow for PyTorch

<table>
<thead>
<tr>
<th>Publisher</th>
<th>NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Tac To Speech</td>
</tr>
<tr>
<td>Model Format</td>
<td>PyTorch,PTH</td>
</tr>
<tr>
<td>Precision</td>
<td>FP32,FP16,FP22</td>
</tr>
<tr>
<td>Version</td>
<td>5</td>
</tr>
<tr>
<td>Last Modified</td>
<td>March 18, 2019</td>
</tr>
<tr>
<td>Training Framework</td>
<td>PyTorch</td>
</tr>
</tbody>
</table>

**Description**

PyTorch scripts for defining, training and using Tacotron 2 and WaveGlow model optimized for Tensor Cores. The Tacotron 2 and WaveGlow model form a text-to-speech system that enables user to synthesize a natural sounding speech from raw transcripts.

<table>
<thead>
<tr>
<th>DGX-1V 16G</th>
<th>Tacotron 2 FP32</th>
<th>Tacotron 2 Mixed Precision</th>
<th>WaveGlow FP32</th>
<th>WaveGlow Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train [Hours]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44 @ 1500 epochs</td>
<td>33.14 @ 1500 epochs</td>
<td>109.96 @ 1000 epochs</td>
<td>54.83 @ 1000 epochs</td>
<td></td>
</tr>
</tbody>
</table>

| Train Accuracy Loss (@1000 Epochs) | | | |
| 0.3629 | 0.3645 | -6.1087 | -6.0258 |

| Perf. | | | |
| 10,843 tokens/sec | 12,742 tokens/sec | 257,687(*) samples/sec | 500,375(*) samples/sec |

| Data set | | |
| LJ Speech Dataset | | |

(*) With sampling rate equal to 22050, one second of audio is generated from 22050 samples


GPU: 1xV100-16GB | DGX-1V | Batch Size: 208 (FP16), 96 (FP16)
**LANGUAGE MODELING: BERT for TensorFlow**

https://ngc.nvidia.com/catalog/model-scripts/nvidia:bert_for_tensorflow

<table>
<thead>
<tr>
<th>GPU Configuration</th>
<th>TF BERT FP32</th>
<th>TF BERT Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-1V 8GPU 32G</td>
<td>0.77 (BSxGPU = 4)</td>
<td>0.51 (BSxGPU = 4)</td>
</tr>
<tr>
<td>Time to Train [Hours]</td>
<td>90.83</td>
<td>90.99</td>
</tr>
<tr>
<td>Train F1 (mean)</td>
<td>66.65</td>
<td>129.16</td>
</tr>
<tr>
<td>Perf. (BSxGPU = 4)</td>
<td>sentences/sec</td>
<td>sentences/sec</td>
</tr>
<tr>
<td>Data set</td>
<td>SQuaD (fine-tuning)</td>
<td></td>
</tr>
</tbody>
</table>

**NGC 19.03 TensorFlow container**


GPU:8xV100-32GB | DGX-1 | Batch size per GPU: 4
## OBJECT DETECTION: TensorFlow SSD

[Link](https://ngc.nvidia.com/catalog/model-scripts/nvidia:ssd_for_tensorflow)

<table>
<thead>
<tr>
<th>GPU Configuration</th>
<th>TF SSD</th>
<th>TF SSD</th>
<th>Time to Train</th>
<th>Accuracy (map)</th>
<th>Perf. (BSxGPU = 32)</th>
<th>Data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGX-1V 8GPU 16G</td>
<td>FP32</td>
<td>Mixed Precision</td>
<td>1h 37min</td>
<td>0.268</td>
<td>569 Img/sec</td>
<td>COCO 2017</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1h 19min</td>
<td>0.269</td>
<td>752 Img/sec</td>
<td></td>
</tr>
</tbody>
</table>

### GPU Specifications
- DGX-1V 8GPU 16G
- 8xV100-16GB
- 16GB per GPU
- DGX-1V
- Batch Size: 32 (FP32, Mixed)

### Performance and Accuracy
- Time to Train: 1h 37min vs 1h 19min
- Accuracy (map): 0.268 vs 0.269
- Perf. (BSxGPU = 32): 569 Img/sec vs 752 Img/sec

### Data Set
- COCO 2017

### Source

---

**Time to Train**
- 1h 37min
- 1h 19min

**Accuracy (map)**
- 0.268
- 0.269

**Perf. (BSxGPU = 32)**
- 569 Img/sec
- 752 Img/sec

**Data set**
- COCO 2017
## TRANSLATION: PyTorch GNMT


<table>
<thead>
<tr>
<th>DGX-2V 16GPU 32G</th>
<th>PyTorch GNMT FP32</th>
<th>PyTorch GNMT Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train [min]</td>
<td>58.6</td>
<td>26.3</td>
</tr>
<tr>
<td>Train Accuracy BLEU score</td>
<td>24.16</td>
<td>24.22</td>
</tr>
<tr>
<td>Perf.</td>
<td>314,831 tokens/sec</td>
<td>738,521 tokens/sec</td>
</tr>
<tr>
<td>Data set</td>
<td>WMT16 English to German</td>
<td></td>
</tr>
</tbody>
</table>

**Source:** [https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/GNMT](https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/GNMT)

GPU:16xV100-32GB | DGX-2 | Batch size: 128 (FP32, Mixed)
# RECOMMENDER: PyTorch Neural Collaborative Filter


<table>
<thead>
<tr>
<th>DGX-1V 8GPU 16G</th>
<th>PyTorch NCF FP32</th>
<th>PyTorch NCF Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time to Accuracy [seconds]</strong></td>
<td>32.68</td>
<td>20.42</td>
</tr>
<tr>
<td><strong>Accuracy Hit Rate @10</strong></td>
<td>0.96</td>
<td>0.96</td>
</tr>
<tr>
<td><strong>Perf.</strong></td>
<td>55,004,590 smp/sec</td>
<td>99,332,230 smp/sec</td>
</tr>
</tbody>
</table>

**Data set**
- MovieLens 20M


**GPU:** 8xV100-16GB | DGX-1 | Batch size: 1,048,576
# INDUSTRIAL DEFECT DETECTION: TensorFlow U-Net


<table>
<thead>
<tr>
<th>DGX-1V 8GPU 16G</th>
<th>TF U-Net FP32</th>
<th>TF U-Net Mixed Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to Train</td>
<td>1 min 44 sec</td>
<td>1 min 36 sec</td>
</tr>
<tr>
<td>IOU (Th=0.75 Class #4)</td>
<td>0.965</td>
<td>0.960</td>
</tr>
<tr>
<td>IOU (Th=0.75 Class #9)</td>
<td>0.988</td>
<td>0.988</td>
</tr>
<tr>
<td>Perf.</td>
<td>445 Img/sec</td>
<td>491 Img/sec</td>
</tr>
</tbody>
</table>

**Data set:** DAGM 2007


**GPU:** 8xV100-16GB | **DGX-1** | **Batch size:** 16

**DAGM 2007** has 10 classes (for the competition). Each class has an independent IOU.

**UNET-Industrial for TensorFlow**

<table>
<thead>
<tr>
<th>Publisher</th>
<th>NVIDIA</th>
<th>Application</th>
<th>Segmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1</td>
<td>Last Modified</td>
<td>April 23, 2019</td>
</tr>
<tr>
<td>Training Framework</td>
<td>TensorFlow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description:**

Tensorflow scripts for defining, training, and using U-Net industrial model optimized for Tensor Cores. This model is a convolutional neural network for 2D-image segmentation turned to avoid overfitting.

**Overview**

This U-Net model is adopted from the original version of the UNet model, which is a convolutional auto-encoder for 2D image segmentation. It was first introduced by Olaf Ronneberger, Philipp Fischer, and Thomas Brox in the paper: U-Net: Convolutional Networks for Biomedical Image Segmentation

This work proposes a modified version of U-Net, called UPyNet, which performs efficiently and with very high accuracy on the industrial anomaly dataset DAGM 2007. UPyNet, like the original U-Net, is composed of two parts:

- an encoding sub-network (left side)
- a decoding sub-network (right side)

It repeatedly applies 3 downsampling blocks composed of two 2D convolutions followed by a 2D max pooling layer in the encoding sub-network. In the decoding sub-network, 3 upsampling blocks are composed of a upsample2D layer followed by a 2D convolution, a concatenation operation with the residual connection and two 2D convolutions.

PyTorch has been introduced to reduce the model capacity which was leading to a high degree of over-fitting on a small dataset like DAGM2007. The complete architecture is presented in the figure below.

---

NGC 19.03 TensorFlow container
## Matching Accuracy for FP32 and Mixed Precision

<table>
<thead>
<tr>
<th>Model Script</th>
<th>Framework</th>
<th>Data Set</th>
<th>Automatic or Manual Mixed-Precision</th>
<th>FP32 Accuracy</th>
<th>Mixed-Precision Accuracy</th>
<th>FP32 Throughput</th>
<th>Mixed-Precision Throughput</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT Q&amp;A (2)</td>
<td>TensorFlow</td>
<td>SQuaD</td>
<td>AMP</td>
<td>90.83 Top 1</td>
<td>90.99 Top 1</td>
<td>66.65 sentences/sec</td>
<td>129.16 sentences/sec</td>
<td>1.94</td>
</tr>
<tr>
<td>SSD w/RN50 (1)</td>
<td>TensorFlow</td>
<td>COCO 2017</td>
<td>AMP</td>
<td>0.268 mAP</td>
<td>0.269 mAP</td>
<td>569 images/sec</td>
<td>752 images/sec</td>
<td>1.32</td>
</tr>
<tr>
<td>GNMT (3)</td>
<td>PyTorch</td>
<td>WMT16 English to German</td>
<td>Manual</td>
<td>24.16 BLEU</td>
<td>24.22 BLEU</td>
<td>314,831 tokens/sec</td>
<td>738,521 tokens/sec</td>
<td>2.35</td>
</tr>
<tr>
<td>Neural Collaborative Filter (1)</td>
<td>PyTorch</td>
<td>MovieLens 20M</td>
<td>Manual</td>
<td>0.959 HR</td>
<td>0.960 HR</td>
<td>55,004,590 samples/sec</td>
<td>99,332,230 items/sec</td>
<td>1.81</td>
</tr>
<tr>
<td>U-Net Industrial (1)</td>
<td>TensorFlow</td>
<td>DAGM 2007</td>
<td>AMP</td>
<td>0.965-0.988</td>
<td>0.960-0.988</td>
<td>445 images/sec</td>
<td>491 images/sec</td>
<td>1.10</td>
</tr>
<tr>
<td>ResNet-50 v1.5 (1)</td>
<td>MXNet</td>
<td>ImageNet</td>
<td>Manual</td>
<td>76.67 Top 1%</td>
<td>76.49 Top 1%</td>
<td>2,957 images/sec</td>
<td>10,263 images/sec</td>
<td>3.47</td>
</tr>
<tr>
<td>Tacotron 2 / WaveGlow 1.0 (1)</td>
<td>PyTorch</td>
<td>LJ Speech Dataset</td>
<td>AMP</td>
<td>0.3629/-6.1087</td>
<td>0.3645/-6.0258</td>
<td>10,843 tok/s</td>
<td>12,742 tok/s</td>
<td>1.18/ 1.94</td>
</tr>
</tbody>
</table>

Values are measured with model running on (1) DGX-1V 8GPU 16G, (2) DGX-1V 8GPU 32G or (3) DGX-2V 16GPU 32G