UPDATE (March 2021)
Topic: Training

- Peng Wang | AI Frameworks @ Microsoft
Common problems impacting ML productivity

- Inference latency is too high to put into production
- Training in Python but need to deploy into a C#/C++/Java app
- Model needs to run on edge/IoT devices
- Same model needs to run on different hardware and operating systems
- Need to support running models created from several different frameworks
- Training very large models takes too long
ONNX RUNTIME

Supports full ONNX-ML spec (v1.2-1.7)
Supports CPU, GPU, VPU
C#, C, C++, Java, JS and Python APIs

Flexible

Works on
-Mac, Windows, Linux
-x86, x64, ARM
Also built-in to Windows 10 natively (WinML)

Cross Platform

Extensible “execution provider” architecture to plug-in custom operators, optimizers, and hardware accelerators

Extensible

Distributed training acceleration on multi-node GPU
Large scale Transformer models

Training (preview)

Model-specific package
Reduced size
Android, iOS, Linux
X86, ARM

Mobile (preview)

github.com/microsoft/onnxruntime
Training Design Principles

- Generic Framework for Training DNNs
  - Extensible with new kernels, optimization algorithms, etc.

- Current Implementation optimizes for Transformer-Based models.

- Adding model support based on customer demand.
ONNX Runtime Training (Public Preview)

- Seamless integration with existing training frameworks for accelerated training and fine tuning of large transformer models
- Incorporates latest algorithms and techniques such as DeepSpeed/ZeRO and Parasail/Adasum
- Integrates with GPU for distributed training
Augmenting ONNX graphs for training

- ORT Training takes an inference (“forward”) graph as input
- Training-specific functionality implemented as graph transformations
Forward graph (inference graph)
Loss function (user-supplied)
Backward graph (loss function gradient)
Backward graph (compute gradients)
Backward graph (use existing operators)
Optimizer (Adam/Lamb)
Training Acceleration

Transformer models
# Usage of ORT Training at Microsoft

<table>
<thead>
<tr>
<th>Team</th>
<th>Scenario / Model</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office services</td>
<td>Pre-training TuringNLR</td>
<td>From 4 days to ~2 days (1.4x higher throughput)</td>
</tr>
<tr>
<td>Bing Ads</td>
<td>Pre-training RoBERTa-XL as base model</td>
<td>From 8 days to 4.5 days (1.4x higher throughput)</td>
</tr>
<tr>
<td>Office apps</td>
<td>Fine-tuning GPT-2 for word prediction</td>
<td>Now able to train; stock PyTorch could not train with data parallelism</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>Pre-training GPT-2 Medium for IntelliSense</td>
<td>From 8 days to 6.5 days (1.19x higher throughput)</td>
</tr>
</tbody>
</table>

### Accelerated training with ONNX Runtime

- **Office 365 pre-training 400M+ Model:**
  - Using PyTorch + ONNX Runtime: 2.2 days
  - Using PyTorch: 4 days

- **Bing Ads pre-training 500M+ Model:**
  - Using PyTorch + ONNX Runtime: 4.5 days
  - Using PyTorch: 8 days

- **Visual Studio fine-tuning 300M+ Model:**
  - Using PyTorch + ONNX Runtime: 6.5 days
  - Using PyTorch: 8 days
Nvidia A100

- FP16 and TF32 supported, BF16 is in progress
- Scales up to 512 A100 GPUs

### BERT-L Pretraining (ORT vs. Nvidia PT)

<table>
<thead>
<tr>
<th>Phase</th>
<th>GPUs</th>
<th>Batch size / GPU</th>
<th>Accumulation steps</th>
<th>Throughput (seq/sec)</th>
<th>Throughput Speedup (ORT vs PT)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>PT</td>
<td>ORT</td>
<td>PT</td>
</tr>
<tr>
<td>Phase 1</td>
<td>1</td>
<td>65536</td>
<td>1024</td>
<td>512</td>
<td>415</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>16384</td>
<td>256</td>
<td>128</td>
<td>1618</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8192</td>
<td>128</td>
<td>64</td>
<td>3231</td>
</tr>
<tr>
<td>Phase 2</td>
<td>1</td>
<td>32768</td>
<td>2048</td>
<td>1024</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8192</td>
<td>512</td>
<td>256</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4096</td>
<td>256</td>
<td>128</td>
<td>620</td>
</tr>
</tbody>
</table>

* Both ORT and PyTorch are using mixed precision training with lamb
* PT numbers are adopted from Nvidia Deep Learning Examples Repo
CUDA Kernel Optimizations

• Transformer models are sharing a “stable and small” set of operators
  • Few variations of activation and normalization functions
  • Easy to support the new models

• Kernel optimizations for BERT are transferable to other models
  • Prioritize for generally applicable and reusable kernel optimizations
  • RoBERTa, GPT-2, and other variants of transformer models run faster with ORT out of the box

• Graph based optimization, no change in model definition
Memory Optimizations

• Optimizing tensor placement in 2D space of Memory-Time
  • Heavily reusing allocated buffer space
  • Minimizes memory fragmentations
  • Predicts peak memory consumption before running the model

• Runs BERT-L @ 2x of PyTorch’s batch size

• Enables training GPT2-Medium on 16GB V100, which PyTorch runs OOMs

• Allows fitting larger model
PyTorch integration: today

```python
# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...
    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# Training Loop
for data, target in data_loader:
    # reset gradient buffer
    optimizer.zero_grad()
    # forward
    y_pred = model(data)
    loss = criterion(output, target)
    # backward
    loss.backward()
    # weight update
    optimizer.step()
```

PyTorch + ONNX Runtime backend

```python
# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...
    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10)
criterion = torch.nn.CrossEntropyLoss()

# Describe entire computation to offload
optimizer = optim.SGDConfig(lr=1e-4)
model_desc = 
    
    # Training Loop
    for data, target in data_loader:
        # forward + backward + weight update
        loss, y_pred = trainer.train_step(data, target)
```

PyTorch

```python
# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...
    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10)
criterion = torch.nn.CrossEntropyLoss()

# Training Loop
for data, target in data_loader:
    # reset gradient buffer
    optimizer.zero_grad()
    # forward
    y_pred = model(data)
    loss = criterion(output, target)
    # backward
    loss.backward()
    # weight update
    optimizer.step()
```
PyTorch integration: next

**PyTorch**

```python
# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...

    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# Training Loop
for data, target in data_loader:
    # reset gradient buffer
    optimizer.zero_grad()

    # forward
    y_pred = model(data)
    loss = criterion(output, target)

    # backward
    loss.backward()

    # weight update
    optimizer.step()
```

**PyTorch + ONNX Runtime backend**

```python
# Model definition
class NeuralNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        ...

    def forward(self, x):
        ...

model = NeuralNet(input_size=784, hidden_size=500, num_classes=10)
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)

# Training Loop
for data, target in data_loader:
    # reset gradient buffer
    optimizer.zero_grad()

    # forward
    y_pred = model(data)
    loss = criterion(output, target)

    # backward
    loss.backward()

    # weight update
    optimizer.step()
```
## Training Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getting-started</td>
<td>Get started with ONNX Runtime with a simple PyTorch transformer model</td>
</tr>
<tr>
<td>nvidia-bert</td>
<td>Using ONNX Runtime Training with BERT pretraining implementation in PyTorch maintained by nvidia</td>
</tr>
<tr>
<td>huggingface-gpt2</td>
<td>Using ONNX Runtime Training with GPT2 finetuning for Language Modeling in PyTorch maintained by huggingface</td>
</tr>
</tbody>
</table>

- GitHub - microsoft/onnxruntime-training-examples: Examples for using ONNX Runtime for model training.
More to Read

ONNX Runtime Training Technical Deep Dive - Microsoft Tech Community
Announcing accelerated training with ONNX Runtime—train models up to 45% faster - Open Source Blog (microsoft.com)