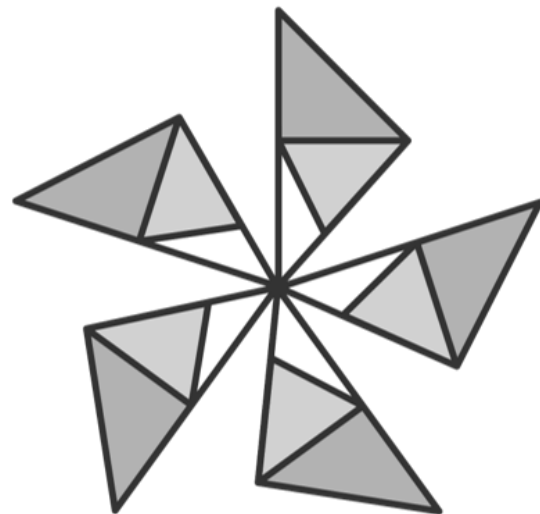
UPDATE
(March 2021)
Topic: Training



ONNX RUNTIME

- **Peng Wang**| AI Frameworks @ Microsoft

Common problems impacting ML productivity

Inference latency is too high to put into production

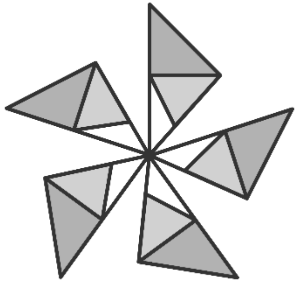
Training in Python but need to deploy into a C#/C++/Java app

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

high-performance engine for machine learning models

Flexible

Supports full ONNX-ML spec (v1.2-1.7)

Supports CPU, GPU, VPU

C#, C, C++, Java, JS and Python APIs

Cross Platform

Works on
-Mac, Windows, Linux
-x86, x64, ARM

Also built-in to Windows 10 natively (WinML)

Extensible

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

Training (preview)

Distributed training acceleration on multi-node GPU

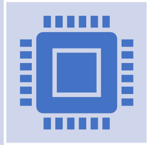
Large scale Transformer models

Mobile (preview)

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

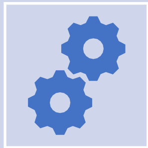
github.com/microsoft/onnxruntime

Training Design Principles

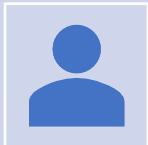


Generic Framework for Training DNNs

Extensible with new kernels, optimization algorithms, etc.

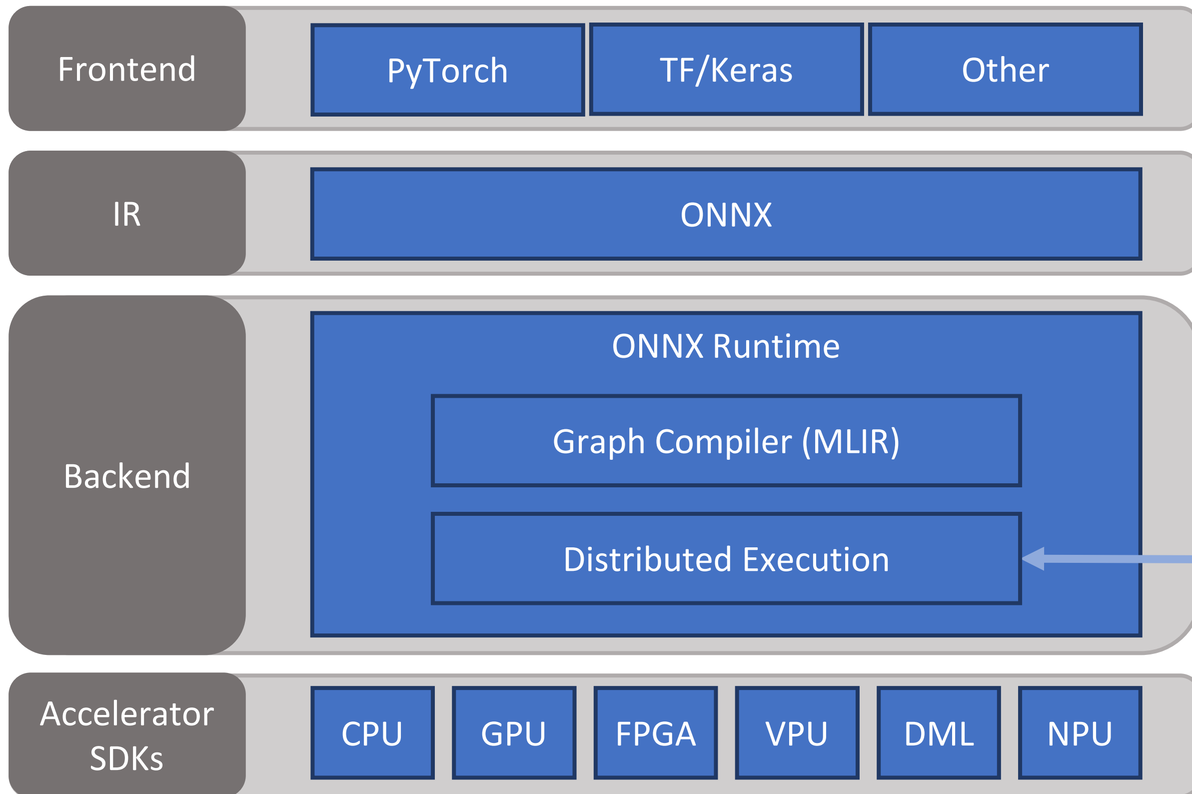


Current Implementation optimizes for Transformer-Based models.

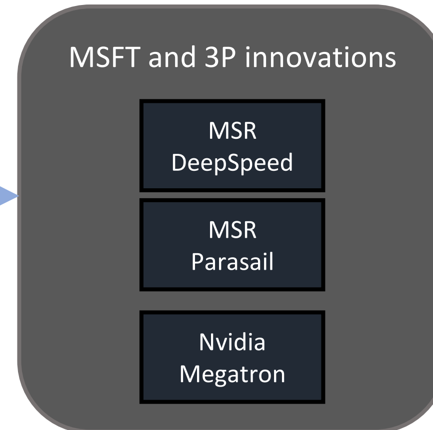


Adding model support based on customer demand

ONNX Runtime Training (Public Preview)

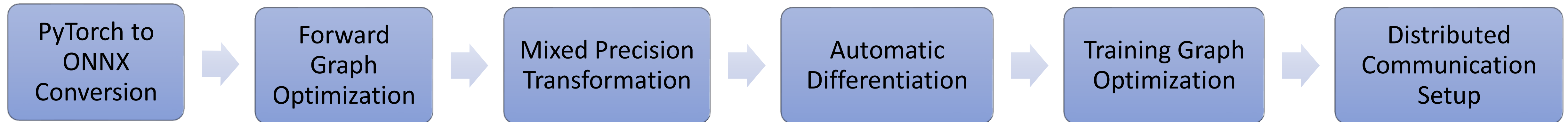


- Seamless integration with existing training frameworks for accelerated training and fine tuning of large transformer models
- Incorporates latest algorithms and techniques such as DeepSpeed/ZeRO and Parasail/Adasum
- Integrates with GPU for distributed training

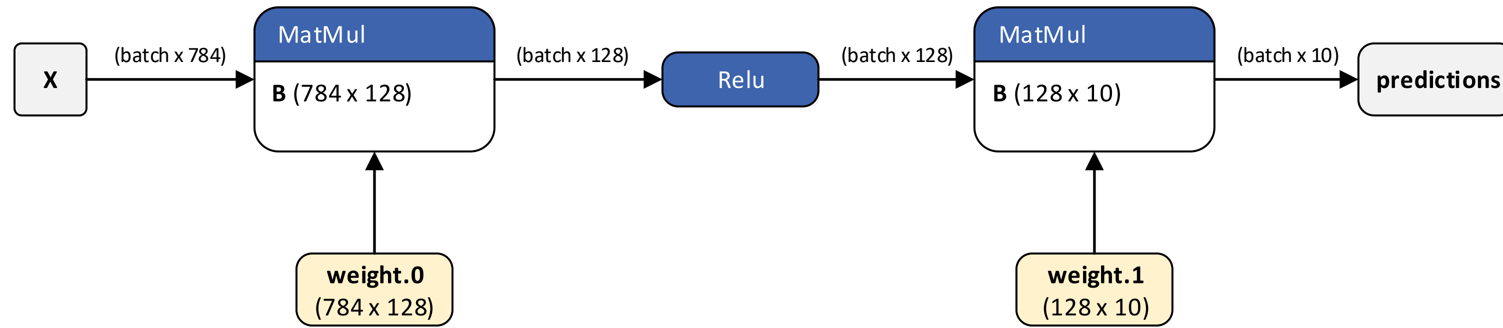


Augmenting ONNX graphs for training

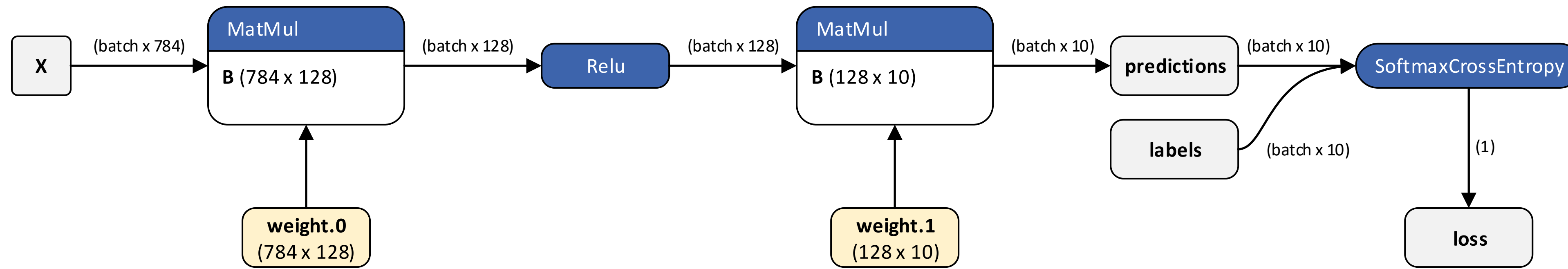
- ORT Training takes an inference ("forward") graph as input
- Training-specific functionality implemented as graph transformations



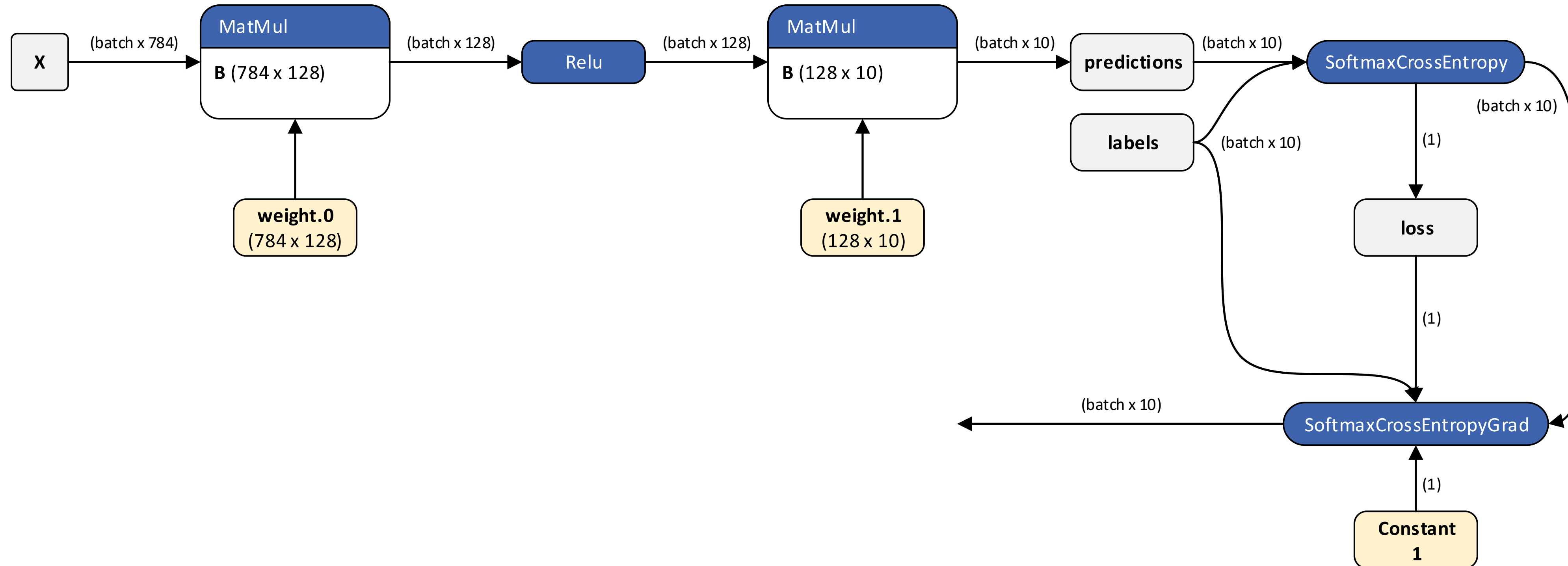
Forward graph (inference graph)



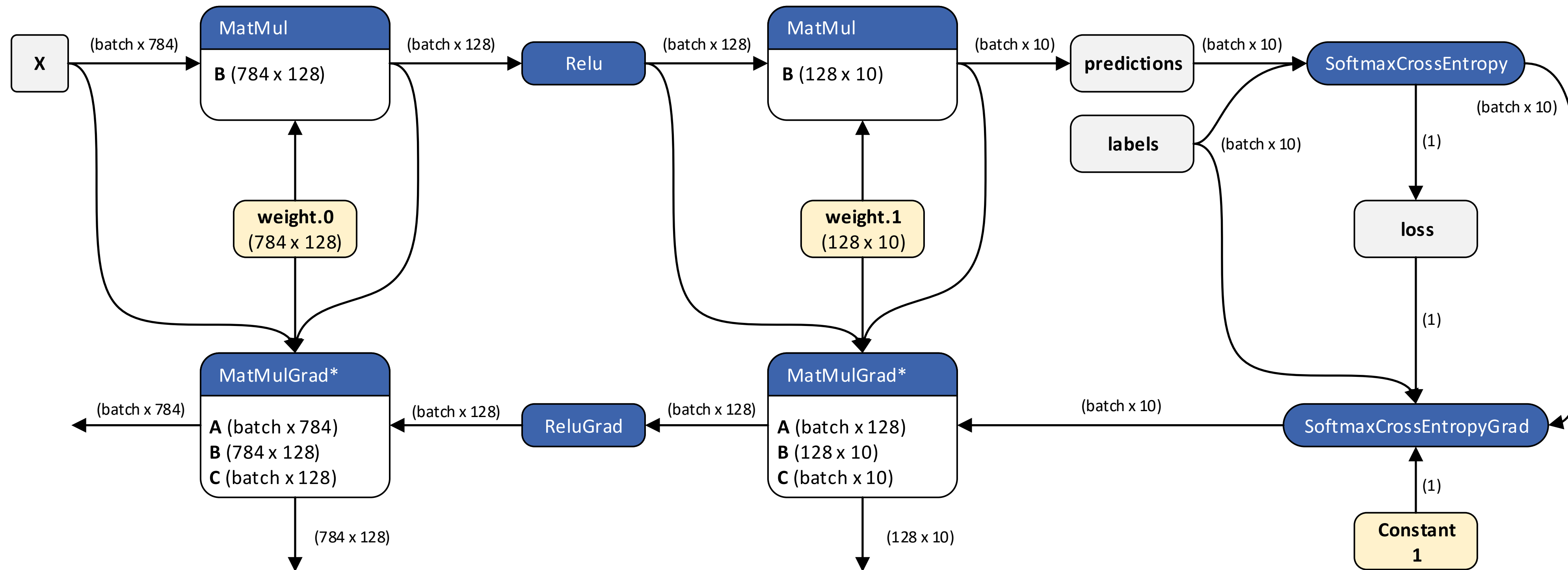
Loss function (user-supplied)



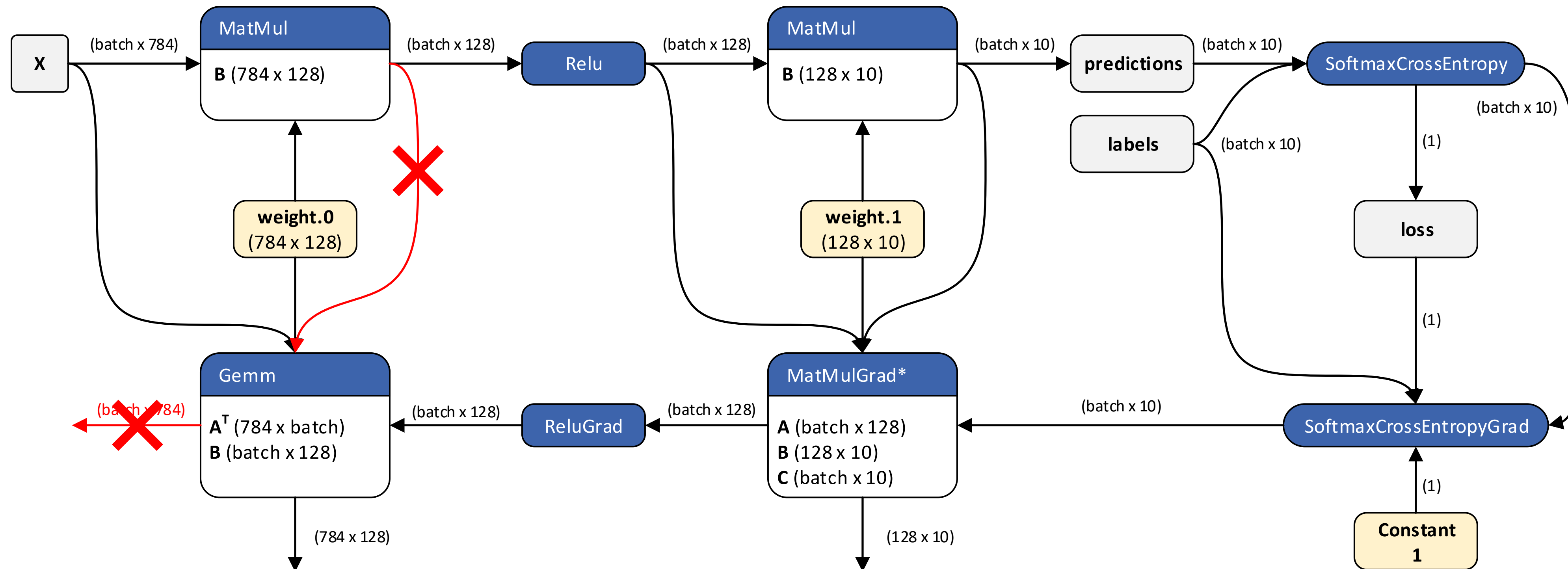
Backward graph (loss function gradient)



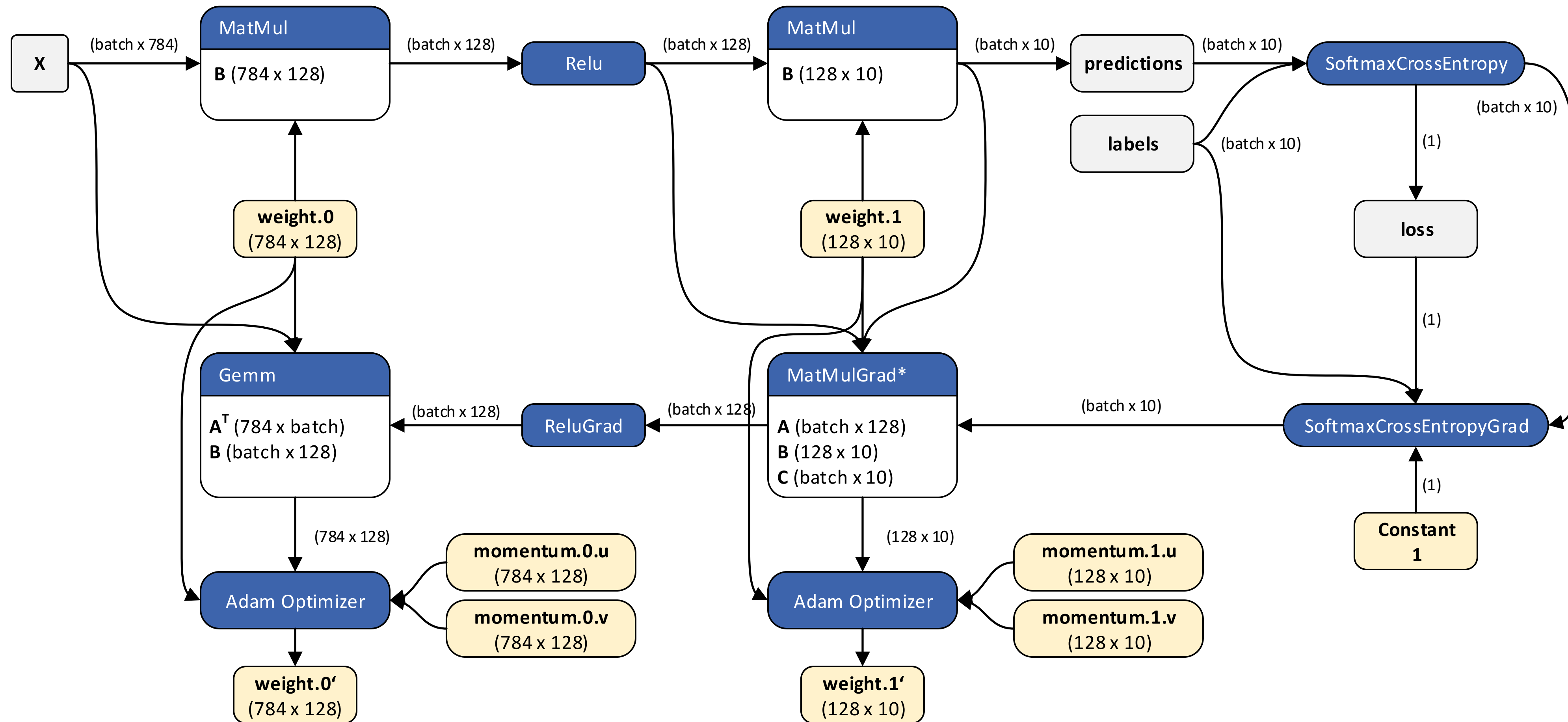
Backward graph (compute gradients)



Backward graph (use existing operators)



Optimizer (Adam/Lamb)

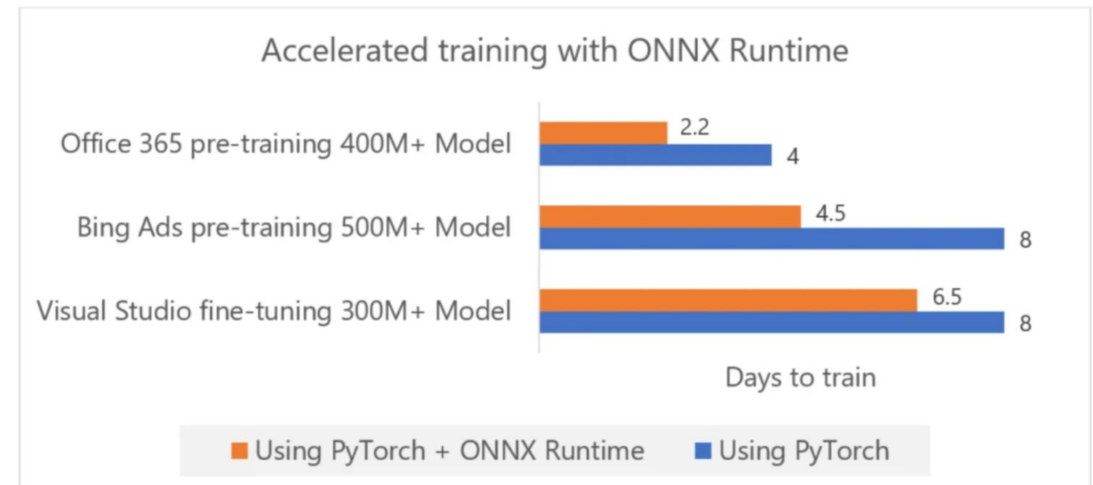


Training Acceleration

Transformer models

Usage of ORT Training at Microsoft

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



Nvidia A100

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

BERT-L Pretraining (ORT vs. Nvidia PT)

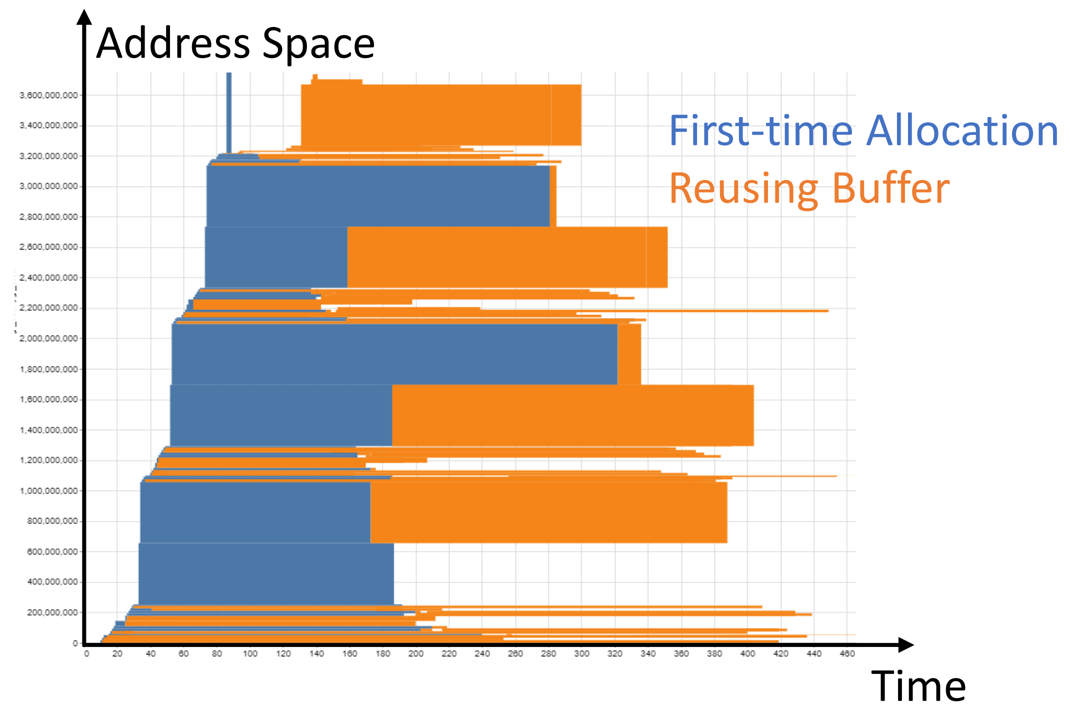
	GPUs	Batch size / GPU	Accumulation steps		Throughput (seq/sec)		Throughput Speedup (ORT vs PT)
			PT	ORT	PT	ORT	
Phase 1	1	65536	1024	512	415	509.4	**22.7%**
	4	16384	256	128	1618	2024.3	**25.1%**
	8	8192	128	64	3231	4058.5	**25.6%**
Phase 2	1	32768	2048	1024	78	96.2	**23.3%**
	4	8192	512	256	308	382	**24.0%**
	8	4096	256	128	620	766.9	**23.7%**

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

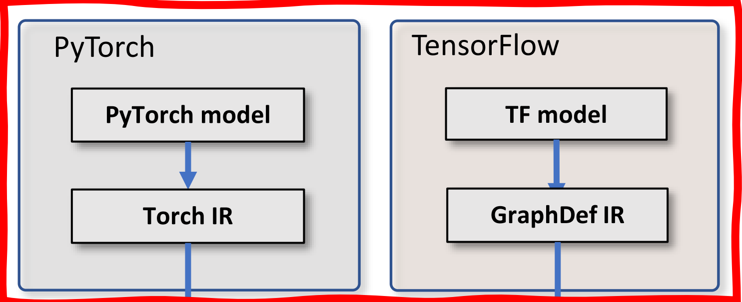
CUDA Kernel Optimizations

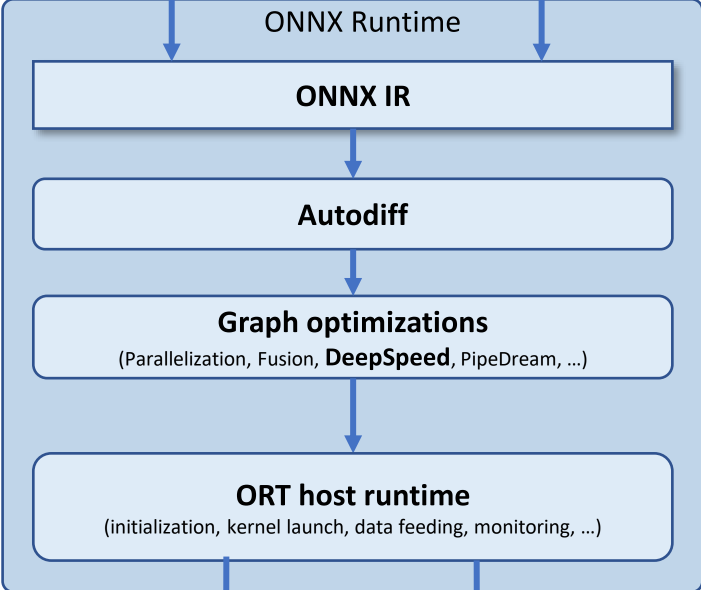
- Transformer models are sharing a “stable and small” set of operators
  - Few variations of activation and normalization functions
  - Easy to support the new models
- Kernel optimizations for BERT are transferable to other models
  - Prioritize for generally applicable and reusable kernel optimizations
  - 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

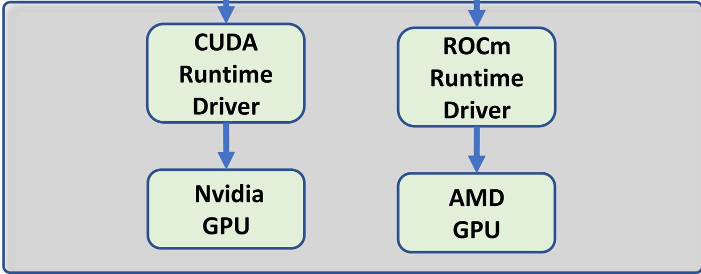
Memory Optimizations



- 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







Front-end integration

PyTorch integration: today

PyTorch

```
# 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

```
# 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 = {"inputs": [("x", ["batch", 784])],
              "outputs": [("y", ["batch", 10])]}
trainer = ORTTrainer(model, optimizer, model_desc, criterion)

# Training Loop
for data, target in data_loader:
    # forward + backward + weight update
    loss, y_pred = trainer.train_step(data, target)
```

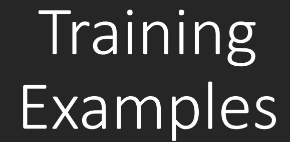
PyTorch integration: next

PyTorch

```
# 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

```
# 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)
model = ORTModule(model)
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

Example	**Description**
getting-started	Get started with ONNX Runtime with a simple PyTorch transformer model
nvidia-bert	Using ONNX Runtime Training with BERT pretraining implementation in PyTorch maintained by nvidia
huggingface-gpt2	Using ONNX Runtime Training with GPT2 finetuning for Language Modeling in PyTorch maintained by huggingface

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

Thanks
谢谢

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)