Parallel Machine Learning and Artificial Intelligence

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Northeastern University
College of Engineering

Content

- Performance from optimization and GPU mixed precision
  - Optimizer
  - GPU mixed precision

Optimizer

- The role of the optimizer:
  - The optimizer updates the parameters of the network model according to the gradient of the backward propagation of the model to reduce the calculated value of the loss function, thereby *improving the performance of training*, especially for complex models with a large number of parameters.
- For the optimizer to work, two main things are needed:
  - the parameters of the models
  - the parameters of the optimizer

Optimizing with an optimizer

- PyTorch provides most common optimization algorithms encapsulated into "optimizer classes".
- torch.optim is a package implementing various optimization algorithms in PyTorch.
- Below are the most commonly used optimizers. Each of them has its specific parameters that you can check on the Pytorch Doc.

```
parameters = [x]          # This should be the list of model parameters

optimizer = optim.SGD(parameters, lr=0.01, momentum=0.9)
optimizer = optim.Adam(parameters, lr=0.01)
optimizer = optim.Adadelta(parameters, lr=0.01)
optimizer = optim.Adagrad(parameters, lr=0.01)
optimizer = optim.RMSprop(parameters, lr=0.01)
optimizer = optim.LBFGS(parameters, lr=0.01)

# and there is more………
```

Optimizing "by hand"

- Minimize the function f "by hand" using the gradient descent algorithm.
- As a reminder, the update step of the algorithm is:

$$x_{i+1} = x_i - \lambda \nabla_x f(x_i)$$

- Note:
  - The gradient information $\nabla_x f(x_i)$ will be stored in x.grad once we run the *backward* function.
  - The gradient is accumulated by default, so we need to clear x.grad after each iteration.
  - We need to use with torch.no_grad(): context for the update step since we want to change x in place but don't want autograd to track this change.

Using an optimizer

- You will need 2 new functions:
  - optimizer.zero_grad() : This function sets the gradient of the parameters (x here) to 0 (otherwise it will get accumulated)
  - optimizer.step() : This function applies an update step

! Instead of model.zero_grad()

Using a learning rate scheduler

- In addition to an optimizer, a learning rate scheduler can be used to adjust the learning rate during training by reducing it according to a pre-defined schedule.
- Below are some of the schedulers available in PyTorch.

```
optim.lr_scheduler.LambdaLR
optim.lr_scheduler.ExponentialLR
optim.lr_scheduler.MultiStepLR
optim.lr_scheduler.StepLR

# and some more ...
```

Main Takeaway:

```
optimizer.zero_grad()                    # optimizer gradient cleared to 0
loss = loss_fn(outputs, y_train)         # calculate loss
loss.backward()                          # loss backpropagation
optimizer.step()                         # update parameters according to the
                                         loss reverse gradient optimizer

scheduler.step()
```

Epoch, Batch Size, Iteration

- Gradient Descent:
  - It is an *iterative* optimization algorithm used in machine learning to find the best results (minima of a curve).
  - Learning rate, Cost function/Loss function
- Epoch
  - One "Epoch" is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.
  - Why we use more than one Epoch?
  - What is the right numbers of epochs?
- Batch
  - Since one epoch is too big to feed to the computer at once we divide it in several smaller batches.
- Batch Size
  - Total number of training examples present in a single batch.

  <u>Note:</u> Batch size and number of batches are two different things.
- Iterations
  - Iterations is the number of batches needed to complete one epoch.

  <u>Note:</u> The number of batches is equal to number of iterations for one epoch.

Installation

On cluster:

- $ module load cuda/10.0 # <= Don't forget this!
  - Current pytorch binaries only provides py3.7_cuda10.2_cudnn7.6.5_0
  - On Discovery, only cuda 10.0 provides cudnn7.6.5
- $ source activate py37
  - (py37) $ conda create -n py37 -y python=3.7 anaconda
  - (py37)$ conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
- $ srun -p gpu --gres=gpu:p100:1 --pty /bin/bash

Discovery GPU Type: https://rc-docs.northeastern.edu/en/latest/using-discovery/workingwithgpu.html

Run Errors on GPU

- If you run on old GPU architecture with low versions CUDA Driver which the kernel is mismatched with cudnn7.6, the error will occur:

  RuntimeError: CUDA error: no kernel image is available for execution on the device

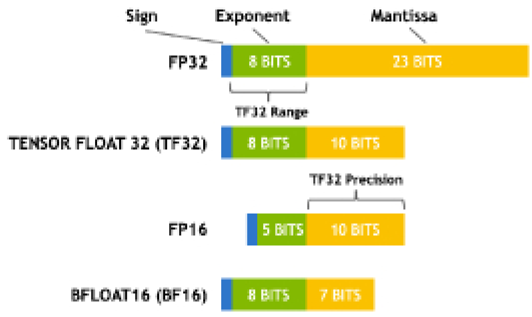
- But *torch.cuda.is_available()* will be *True* ==> Be careful!

Mixed-Precision Training of Deep Neural Networks

- DNN complexity has been increasing, which in turn has increased the computational resources required to train these networks.
- Mixed-precision training lowers the required resources by using lower-precision arithmetic and achieve high performance.
  - Decrease the required amount of memory. Lowering the required memory enables training of larger models or training with larger minibatches.
  - Shorten the training or inference time. Half-precision halves the number of bytes accessed, thus reducing the time spent in memory-limited layers.

Mixed Precision Training

- NVIDIA GPUs have special hardware units, known as tensor cores (TC), that accelerate FP16 matmul operations.

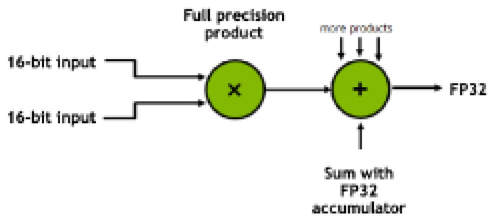


Benefits:

✔ Up to 8-16x speedup for math-bound operations with TC vs FP32

✔ Up to 2x speedup for memory-bound operations vs FP32

✔ FP16 reduces storage requirements for activation and weight tensors



How GPU Support AMP

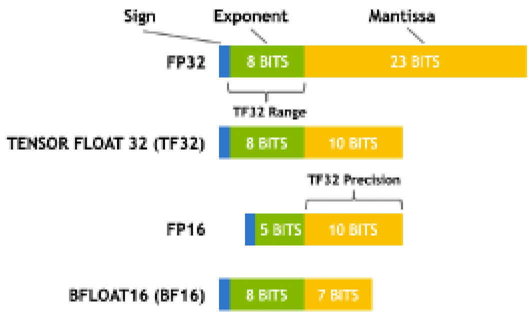
- Mixed precision primarily benefits Tensor Core-enabled architectures (Volta, Turing, Ampere).
- On earlier architectures (Kepler, Maxwell, Pascal), you may observe a modest speedup.
- Run nvidia-smi to display your GPU's architecture.

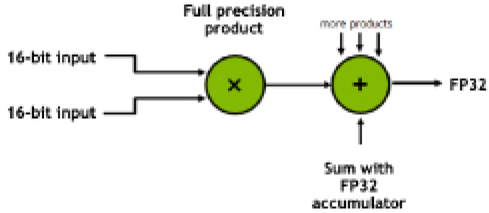
Discovery GPU Type: https://rc-docs.northeastern.edu/en/latest/using-discovery/workingwithgpu.html

Mixed Precision Training

Challenges:

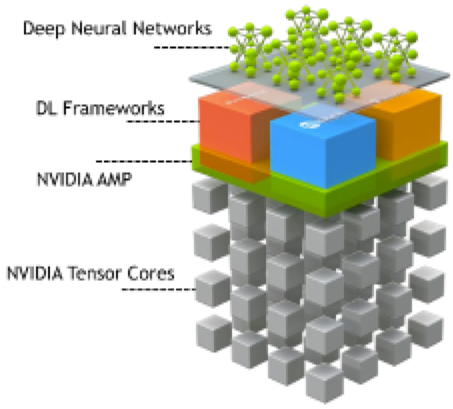
- Not all operations are suitable for FP16
  - Long running sums with small values
  - Operations where |f(x)| >> |x|
  - Loss functions
- Small gradients can be lost to zero in lower precision, impacting training
  - Need to scale loss to bring gradients into representable range





- Difficult to deal with these complexities as a user and can be error-prone

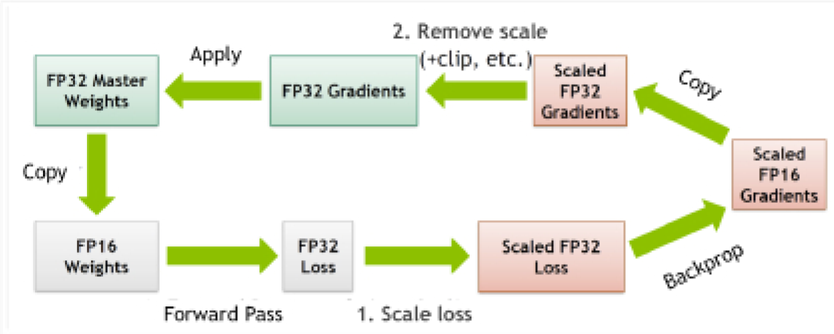
Automatic Mixed Precision (AMP)



Available for TensorFlow 1.14+ and 2.0+, PyTorch 1.6+, and MXNet 1.5+

For PyTorch, automatic Mixed Precision feature is available in the **Apex repository** on GitHub. To enable, add these two lines of code into your existing training script:

```
scaler = GradScaler()

with autocast():
    output = model(input)
    loss = loss_fn(output, target)

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()
```



Automatic Mixed Precision for Deep Learning
https://developer.nvidia.com/automatic-mixed-precision

Using AMP in PyTorch – APEX

- Apex:
  - A PyTorch Extension a Pytorch extension with NVIDIA-maintained utilities to streamline mixed precision and distributed training. See: https://pypi.org/project/apex/
- AMP: Automatic Mixed Precision
  - apex.amp
- Distributed Training
  - apex.parallel
- Fused Optimizers
  - apex.optimizers
- Install:
  - (py37)$ pip install apex

Using AMP in PyTorch

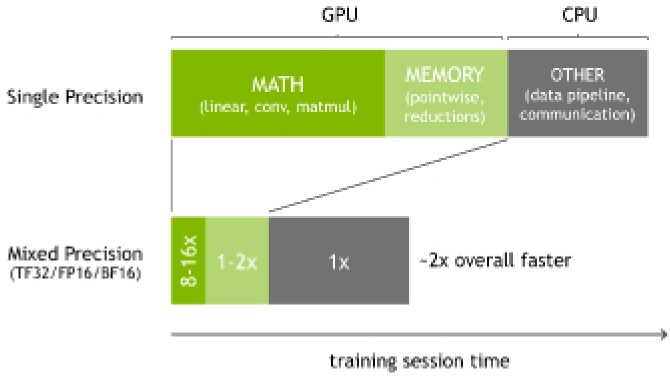
For PyTorch, Automatic Mixed Precision feature is available in the **Apex repository** on GitHub.

```
scaler = GradScaler()

with autocast():
    output = model(input)
    loss = loss_fn(output, target)

scaler.scale(loss).backward()

scaler.step(optimizer)

scaler.update()
```

```
import torch

# Create GradScaler at start of training scaler
= torch.cuda.amp.GradScaler()

for inp, target in data_loader:
    optimizer.zero_grad()

    # Casts safe operations to mixed precision
    with torch.cuda.amp.autocast():
        loss = model(inp, target)


    # Scales the loss and calls backward to
    # create scaled gradients
    scaler.scale(loss).backward()

    # Unscales gradients and calls or skips
    scaler.step(optimizer)

    # Updates the loss scale for next iteration
    scaler.update()
```


I added AMP, now what?

- Don't forget about Amhdahl's law!
  - AMP only speeds up GPU operations
- Reprofile and reevaluate bottlenecks
- Ensure efficient Tensor Core operation by:
  - Favor multiples of 8 for linear layer and convolutions
  - Avoid GEMMs with dimensions <128 which are memory bound.



- Stay safe!
- See you next class!

Next Lectures:

Quiz 2

Review Proposals

DL Data Parallelism