Simple techniques to improve training performance

Implement by changing a few lines of code
ENABLE ASYNC DATA LOADING & AUGMENTATION

- PyTorch DataLoader supports asynchronous data loading / augmentation
  - Default settings: num_workers=0, pin_memory=False
- Use num_workers > 0 to enable asynchronous data processing
- Use pin_memory=True

Example: PyTorch MNIST example: DataLoader with default {'num_workers': 1, 'pin_memory': True}.

<table>
<thead>
<tr>
<th>Setting for the training DataLoader</th>
<th>Time for one training epoch</th>
</tr>
</thead>
<tbody>
<tr>
<td>{'num_workers': 0, 'pin_memory': False}</td>
<td>8.2 s</td>
</tr>
<tr>
<td>{'num_workers': 1, 'pin_memory': False}</td>
<td>6.75 s</td>
</tr>
<tr>
<td>{'num_workers': 1, 'pin_memory': True}</td>
<td>6.7 s</td>
</tr>
<tr>
<td>{'num_workers': 2, 'pin_memory': True}</td>
<td>4.2 s</td>
</tr>
<tr>
<td>{'num_workers': 4, 'pin_memory': False}</td>
<td>4.5 s</td>
</tr>
<tr>
<td>{'num_workers': 4, 'pin_memory': True}</td>
<td>4.1 s</td>
</tr>
<tr>
<td>{'num_workers': 8, 'pin_memory': True}</td>
<td>4.5 s</td>
</tr>
</tbody>
</table>

PyTorch 1.6, NVIDIA Quadro RTX 8000
DISABLE BIAS FOR CONVOLUTIONS DIRECTLY FOLLOWED BY A BATCH NORM

Also applicable to Conv1d, Conv3d if BatchNorm normalizes on the same dimension as convolution's bias.
EFFICIENTLY SET GRADIENTS TO ZERO

model.zero_grad()

# or

optimizer.zero_grad()

for param in model.parameters():
    param.grad = None

# or (in PyT >= 1.7)

model.zero_grad(set_to_none=True)

- executes memset for every parameter in the model
- backward pass updates gradients with "+=" operator (read + write)

- doesn't execute memset for every parameter
- memory is zeroed-out by the allocator in a more efficient way
- backward pass updates gradients with "=" operator (write)
DISABLE GRADIENT CALCULATION FOR INFERENCE

# torch.no_grad() as a context manager:
with torch.no_grad():
    output = model(input)

# torch.no_grad() as a function decorator:
@torch.no_grad()
def validation(model, input):
    output = model(input)
    return output
**FUSE POINTWISE OPERATIONS**

- **PyTorch JIT** can fuse pointwise operations into a single CUDA kernel.
- Unfused pointwise operations are memory-bound, for each unfused op PyTorch:
  - launches a separate kernel
  - loads data from global memory
  - performs computation
  - stores results back into global memory

Example:

```python
@torch.jit.script
def fused_gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

<table>
<thead>
<tr>
<th>Function name</th>
<th>Number of CUDA kernels launched</th>
<th>Execution time [us] (input vector with 1M elements)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gelu(x)</td>
<td>5</td>
<td>65</td>
</tr>
<tr>
<td>fused_gelu(x)</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

PyTorch 1.6, NVIDIA Quadro RTX 8000
GPU SPECIFIC OPTIMIZATIONS
USE MIXED PRECISION AND AMP

- Set sizes to multiples of 8
  - See Deep Learning Performance Documentation for more details and guidelines specific to layer type
  - Use explicit padding when necessary (e.g. vocabulary size in NLP)

- Enable AMP
  - Introduction to Mixed Precision Training and AMP: video, slides
  - Native PyTorch AMP is available starting from PyTorch 1.6: documentation, examples, tutorial
ENABLE cuDNN AUTOTUNER

For convolutional neural networks, enable cuDNN autotuner by setting:

```python
torch.backends.cudnn.benchmark = True
```

- cuDNN supports many algorithms to compute convolution
- autotuner runs a short benchmark and selects algorithm with the best performance

Example:

`nn.Conv2d` with 64 3x3 filters applied to an input with batch size = 32, channels = width = height = 64.

<table>
<thead>
<tr>
<th>Setting</th>
<th>cudnn.benchmark = False (the default)</th>
<th>cudnn.benchmark = True</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward propagation (FP32) [us]</td>
<td>1430</td>
<td>840</td>
<td>1.70</td>
</tr>
<tr>
<td>Forward + backward propagation (FP32) [us]</td>
<td>2870</td>
<td>2260</td>
<td>1.27</td>
</tr>
</tbody>
</table>

PyTorch 1.6, NVIDIA Quadro RTX 8000
CREATE TENSORS DIRECTLY ON TARGET DEVICE

```
torch.rand(size).cuda()
torch.rand(size, device=torch.device('cuda'),
)
```

Also applicable to:

```
torch.empty(), torch.zeros(), torch.full(), torch.ones(),
torch.eye(), torch.randint(), torch.randn()
```

and similar functions.
AVOID CPU-GPU SYNC

- Operations which require synchronization:
  - `print(cuda_tensor)`
  - `cuda_tensor.item()`
  - **memory copies:** `tensor.cuda()`, `cuda_tensor.cpu()` and `tensor.to(device)` calls
  - `cuda_tensor.nonzero()`
  - python control flow which depends on operations on CUDA tensors e.g.
    ```python
    if (cuda_tensor != 0).all()
    ```
DISTRIBUTED OPTIMIZATIONS
USE EFFICIENT MULTI-GPU BACKEND

**DataParallel**
- 1 CPU core drives multiple GPUs
- 1 python process drives multiple GPUs (GIL)
- only up to a single node

**DistributedDataParallel**
- 1 CPU core for each GPU
- 1 python process for each GPU
- single-node and multi-node (same API)
- efficient implementation:
  - automatic bucketing for grad all-reduce
  - all-reduce overlapped with backward pass
- multi-process programing
LOAD-BALANCE WORKLOAD ON MULTIPLE GPUs

Gradient all-reduce after backward pass is a synchronization point in a multi-GPU setting
LOAD-BALANCE WORKLOAD ON MULTIPLE GPUs

Gradient all-reduce after backward pass is a synchronization point in a multi-GPU setting.
SUMMARY

- **General optimizations:**
  - Use asynchronous data loading
  - Disable bias for convolutions directly followed by batch norm
  - Efficiently set gradients to zero
  - Disable gradient calculation for validation/inference
  - Fuse pointwise operations with PyTorch JIT

- **GPU specific optimizations:**
  - Use mixed precision and AMP
  - Enable cuDNN autotuner
  - Create tensors directly on a GPU
  - Avoid CPU-GPU sync

- **Distributed optimizations**
  - Use DistributedDataParallel
  - Load-balance workload on all GPUs
ADDITIONAL RESOURCES

- PyTorch Tutorial: Performance Tuning Guide
  - Check for more optimizations
- NVIDIA Deep Learning Performance Documentation
- Introduction to Mixed Precision Training and AMP: video, slides
- Using Nsight Systems to profile GPU workload (PyTorch Dev forum)