Multi-GPU examples

Data Parallelism is when we split the mini-batch of samples into multiple smaller mini-batches and run the computation for each of the smaller mini-batches in parallel.

Data Parallelism is implemented using `torch.nn.DataParallel`. One can wrap a Module in `DataParallel` and it will be parallelized over multiple GPUs in the batch dimension.

DataParallel

```
import torch
import torch.nn as nn


class DataParallelModel(nn.Module):

    def __init__(self):
        super().__init__()
        self.block1 = nn.Linear(10, 20)

        # wrap block2 in DataParallel
        self.block2 = nn.Linear(20, 20)
        self.block2 = nn.DataParallel(self.block2)

        self.block3 = nn.Linear(20, 20)

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        return x
```

The code does not need to be changed in CPU-mode.

The documentation for DataParallel can be found here.

Primitives on which DataParallel is implemented upon:

In general, pytorch's *nn.parallel* primitives can be used independently. We have implemented simple MPI-like primitives:

- replicate: replicate a Module on multiple devices
- scatter: distribute the input in the first-dimension
- gather: gather and concatenate the input in the first-dimension
- parallel_apply: apply a set of already-distributed inputs to a set of already-distributed models.

To give a better clarity, here function `data_parallel` composed using these collectives

```
def data_parallel(module, input, device_ids, output_device=None):
    if not device_ids:
        return module(input)

    if output_device is None:
        output_device = device_ids[0]

    replicas = nn.parallel.replicate(module, device_ids)
    inputs = nn.parallel.scatter(input, device_ids)
    replicas = replicas[:len(inputs)]
    outputs = nn.parallel.parallel_apply(replicas, inputs)
    return nn.parallel.gather(outputs, output_device)
```

Part of the model on CPU and part on the GPU

Let's look at a small example of implementing a network where part of it is on the CPU and part on the GPU

```
device = torch.device("cuda:0")

class DistributedModel(nn.Module):

    def __init__(self):
        super().__init__(
            embedding=nn.Embedding(1000, 10),
            rnn=nn.Linear(10, 10).to(device),
        )

    def forward(self, x):
        # Compute embedding on CPU
        x = self.embedding(x)

        # Transfer to GPU
        x = x.to(device)

        # Compute RNN on GPU
        x = self.rnn(x)
        return x
```

This was a small introduction to PyTorch for former Torch users. There's a lot more to learn.

Look at our more comprehensive introductory tutorial which introduces the `optim` package, data loaders etc.: Deep Learning with PyTorch: A 60 Minute Blitz.

Also look at

- Train neural nets to play video games
- Train a state-of-the-art ResNet network on imagenet
- Train an face generator using Generative Adversarial Networks
- Train a word-level language model using Recurrent LSTM networks
- More examples
- More tutorials
- Discuss PyTorch on the Forums
- Chat with other users on Slack

Total running time of the script: (0 minutes 0.000 seconds)

Download Python source code: parallelism_tutorial.py

Download Jupyter notebook: parallelism_tutorial.ipynb

Gallery generated by Sphinx-Gallery