THE CYPRUS INSTITUTE

RESEARCH•TECHNOLOGY•INNOVATION

Efficient Scaling of Machine Learning Models: Distributed Training

Dr Charalambos Chrysostomou
Associate Research Scientist
The Cyprus Institute

Why Use Distributed Training?

- Distributed Training involves training machine learning models across several computing units simultaneously.
  - **Scalability**: Train larger models efficiently as datasets grow in size.
  - **Speed**: Parallel processing significantly cuts down training time.
  - **Innovation**: Enables the development of more complex and accurate models, pushing the boundaries of AI research and applications.

Distributed Training Strategies

- **Data Parallelism:** Split data across multiple processors to train a single model.
- **Model Parallelism:** Divide a model across different processors when it’s too large for one.
- **Hybrid Approaches:** Combine methods to optimize training efficiency and model performance.

Key Challenges in Distributed Training

- **Communication Overhead:** Coordinating between devices can introduce delays.
- **Complexity:** Requires careful planning and execution to split models or data efficiently.
- **Resource Management:** Effective distribution of tasks to avoid bottlenecks and maximize resource utilization.

Introduction to Data Parallel in PyTorch

- **Data Parallel (DP)** allows you to distribute your data across multiple GPUs to parallelize model training.
- PyTorch simplifies this process with its DataParallel module, enabling efficient training on multiple GPUs with minimal code changes.

Preparing for Data Parallel Training

Ensure PyTorch is installed and you have access to multiple GPUs.

Import necessary libraries:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Verify multiple GPUs are available:

```
print("Available GPUs:", torch.cuda.device_count())
```

Implementing Data Parallel with PyTorch

Wrap your model with **torch.nn.DataParallel** to enable automatic data parallelism.

```
model = MyModel()  # Your custom model
if torch.cuda.device_count() > 1:
  print("Let's use", torch.cuda.device_count(), "GPUs!")
  model = nn.DataParallel(model)
model.to('cuda')
```

Training a Model in Data Parallel Mode

The training process remains similar to a standard training loop. DataParallel automatically divides the data and aggregates the gradients.

Example training loop snippet:

```
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss()
for epoch in range(total_epochs):
    for inputs, labels in dataloader:
        inputs, labels = inputs.to('cuda'), labels.to('cuda')
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

Best Practices for Data Parallel Training

- **Even Data Distribution**: Ensure your dataset is evenly distributed to prevent bottlenecks.
- **Batch Size Consideration**: Increase your batch size to fully utilize the parallel processing power. Remember, the effective batch size is multiplied by the number of GPUs.
- **Resource Monitoring**: Keep an eye on memory usage and adjust model and batch sizes as necessary to optimize training.

Model Parallelism

- Model parallelism involves splitting a neural network's components across multiple GPUs.
- It's ideal for models whose size exceeds the memory capacity of a single GPU.
- We'll explore a detailed example of model parallelism in PyTorch, focusing on a neural network that's divided across two GPUs.

How to Split a Model Across GPUs

- The key is identifying model segments that can independently operate on different GPUs.
- For a neural network with multiple layers, consider grouping these layers into blocks.
- Each block can be assigned to a different GPU.
- Example: Split a model into two blocks, Block1 on GPU 0 and Block2 on GPU 1.

PyTorch Model Parallelism Example

Consider a neural network with two major blocks, designed to run on separate GPUs.

```python
import torch
import torch.nn as nn

class TwoBlockNN(nn.Module):
    def __init__(self):
        super(TwoBlockNN, self).__init__()
        # First block of layers on GPU 0
        self.block1 = nn.Sequential(
            nn.Linear(10, 20),
            nn.ReLU(),
            nn.Linear(20, 50)
        ).to('cuda:0')

        # Second block of layers on GPU 1
        self.block2 = nn.Sequential(
            nn.ReLU(),
            nn.Linear(50, 1)
        ).to('cuda:1')

    def forward(self, x):
        # Move input to GPU 0 and pass through the first block
        x = self.block1(x.to('cuda:0'))
        # Move intermediate output to GPU 1 and pass through the second block
        x = self.block2(x.to('cuda:1'))
        return x
```

Running the Split Model on Input Data

With the model defined, we can pass input data through it, utilizing both GPUs.

```
# Initialize the model
model = TwoBlockNN()

# Create input tensor
input_tensor = torch.randn(64, 10)  # Example input

# Forward pass through the model
output = model(input_tensor)
print("Output shape:", output.shape)
```

Optimizing Model Parallelism

- **Communication Overhead:** Keep data movement between GPUs to a minimum.
- **Load Balancing:** Aim for an even distribution of computational load across GPUs.
- **Debugging:** Model parallelism introduces complexity, making debugging more challenging. Start with a small, simple model to test your setup.
- **Efficiency:** Consider the efficiency of parallel execution versus the overhead of data transfer between GPUs.

Leveraging DeepSpeed's ZeRO in PyTorch for Large Model Training

ZeRO (Zero Redundancy Optimizer) optimizes memory usage in distributed training, enabling larger model training.

Integrated with PyTorch through DeepSpeed, it reduces memory footprint per device.

Configuring ZeRO

Define your model as usual in PyTorch.

Configure ZeRO via DeepSpeed's JSON configuration:

```
{
  "train_batch_size": 32,
  "train_micro_batch_size_per_gpu": 8,
  "optimizer": {
    "type": "Adam",
    "params": {
      "lr": 0.001
    }
  },
  "fp16": {
    "enabled": true
  },
  "zero_optimization": {
    "stage": 2
  }}
```

Starting Training with ZeRO

Wrap your model and optimizer with DeepSpeed's engine.

```
model_engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    optimizer=optimizer,
    config="path/to/deepspeed_config.json"
)
```

Implementing a ZeRO-Optimized Training Loop

Example training loop snippet:

```
for data, labels in dataloader:
    data, labels = data.to(model_engine.device), labels.to(model_engine.device)
    optimizer.zero_grad()
    outputs = model_engine(data)
    loss = criterion(outputs, labels)
    model_engine.backward(loss)
    model_engine.step()
```

Leveraging Ray for Scalable Distributed Training

Ray is an advanced framework for distributed computing, designed for scalability and ease of use in machine learning projects. It seamlessly integrates with PyTorch, offering efficient solutions for handling large datasets and complex computations across multiple CPUs or GPUs.

- **Simplicity**: Intuitive APIs for easy adaptation of existing code.
- **Scalability**: Effective scaling from a single machine to large clusters.
- **PyTorch Integration**: Facilitates distributed training and hyperparameter tuning.
- **Comprehensive Tools**: Includes Ray Tune and RaySGD for enhanced machine learning capabilities.

Setting Up Ray with PyTorch

To leverage Ray for distributed machine learning, start by setting up your environment. This setup ensures your projects can efficiently utilize Ray's capabilities alongside PyTorch for distributed training.

Start Ray to manage resources. Adjust the number of CPUs and GPUs based on your setup.

```
import ray
ray.init(num_cpus=4, num_gpus=2)
```

- **Resource Allocation:** Tailor the initialization parameters (num_cpus, num_gpus) to match your available computational resources.
- **Environment Compatibility:** Check compatibility between Ray and PyTorch versions, and update if necessary.
- **Cluster Setup:** For larger scale projects, consider configuring a Ray cluster. This setup allows distributed training over multiple machines, further enhancing scalability and processing power.

Ray Datasets for Efficient Preprocessing

Ray Datasets simplify the management and preprocessing of large datasets across distributed systems, integrating seamlessly with machine learning workflows.

- **Parallel Processing:** Transform data across multiple nodes, reducing preparation time.
- **PyTorch Integration:** Convert Ray Datasets to DataLoader objects for PyTorch models.

Load Data: Convert datasets to Ray Datasets for distributed handling.

```
from ray.data import from_pandas
ray_dataset = from_pandas(your_dataframe, parallelism=10)
```

Preprocess Data: Apply functions in parallel to preprocess data efficiently.

```
processed_dataset = ray_dataset.map(preprocess_function)
```

Prepare for Training: Transform to PyTorch DataLoader for model training.

```
from ray.data.torch import TorchDataset
dataloader = DataLoader(TorchDataset(processed_dataset), batch_size=32)
```

Simplify Distributed Training with Ray

Ray with PyTorch streamlines distributed training, optimizing resource use and simplifying model synchronization and data distribution.

- **Ray SGD**: Simplifies distributed training setup in PyTorch.
- **Ray Tune:** Enables scalable hyperparameter tuning for optimal model performance.

Ray SGD for Training:

- Define a training function with your model, data loaders, and training logic.

```
from ray.util.sgd import TorchTrainer

trainer = TorchTrainer(
    train_func=lambda config: your_training_routine(),
    num_workers=4, use_gpu=True
)
trainer.train()
```

Simplify Distributed Training with Ray

Optimize with Ray Tune:

- Automatically tune hyperparameters for best results.

```
from ray import tune
tune.run(
    lambda config: your_training_routine(config),
    config={"lr": tune.loguniform(1e-4, 1e-1)}
)
```

Advantages:

- Streamlined Process: Focus on model logic, letting Ray handle the complexity.
- Resource Optimization: Efficiently leverages CPUs and GPUs for faster training.
- Rapid Hyperparameter Tuning: Quickly finds the best settings, improving model accuracy.

Streamline Training and Enhance Performance with RAY

Ray provides essential tools for monitoring and fine-tuning distributed training, ensuring optimal efficiency and resource use.

Optimization Tips:

- Dynamic Resource Adjustment:
  - Tailor the number of workers and GPU allocation to current needs for best efficiency.
- Efficient Data Handling:
  - Use Ray's capabilities to reduce data loading times and balance workloads.

```
# Dynamically allocate workers for optimal performance
trainer = TorchTrainer(
    train_func=lambda config: your_training_routine(),
    num_workers=adjust_workers_based_on_load(),
    use_gpu=True
)
```

- Improved Efficiency: Ensure resources are fully utilized without overloading.
- Faster Training: Identify and eliminate bottlenecks for quicker model development.

Thank You!

Thank you for attending!

We hope this workshop on Distributed Training with PyTorch has empowered your ML projects.

Keep exploring and happy coding!

Part 2: Hands on Tutorial

https://t.ly/NkQjq