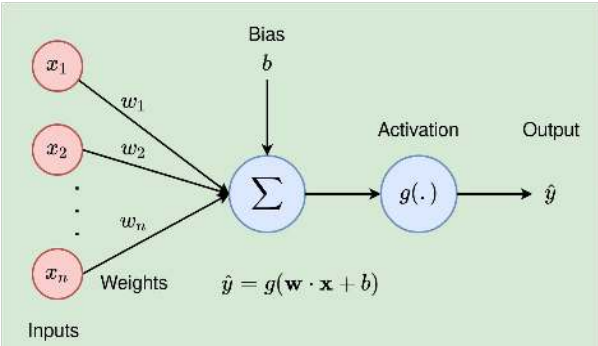
Deep Learning Model Optimization with Pytorch

Anatomy of a Neural Net

- Input Layer: The entry point for data into the neural network, where each neuron represents a feature of the input dataset.
- Weights: Parameters that are learned during the training process. They amplify or dampen the input signal, influencing the network's output.
- Optimizer: Algorithms that adjust the weights of connections within the network to minimize the loss function. Examples include SGD, Adam, and RMSprop.
- Activation Function: Non-linear functions that determine the output of a neural processing unit, ReLU, Sigmoid, and Tanh, adding non-linearity to enable the network to learn complex patterns.
- Loss Function: A metric that measures the difference between the network's prediction and the actual target values, i.e. MSE for regression and Cross-Entropy for classification tasks.
- Output Layer: Produces the model's predictions, tailored to the specific type of problem being solved, such as classification or regression.



Structuring a Training Loop

- Reproducibility: Set deterministic behaviors with fixed seeds
- Dataset & DataLoaders with multiprocessing optimize batch wise data handling
- Optimizer Choice: Choose appropriate optimizer (e.g., Adam) and loss function (e.g., CrossEntropyLoss)
- Training and Validation Execution: Separate training and validation loops, enable/disable gradient computations as needed.
- Performance Monitoring: Implement real-time monitoring and logging with TensorBoard to track metrics like loss and accuracy.
- Checkpointing and Recovery: Regularly save model states and enable easy recovery with checkpoints for handling long training sessions.

Seed it all

```
import torch
import numpy as np
import random
def seed_everything(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
seed_everything()
```

Use DataLoaders!

```
from torch.utils.data import DataLoader
train_loader = DataLoader(dataset=train_dataset,
batch_size=64, shuffle=True, num_workers=4)
```

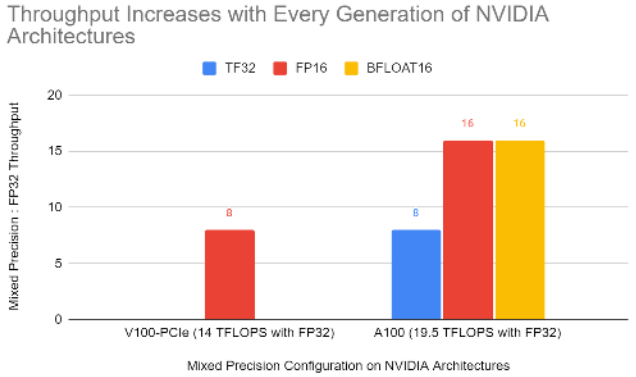
Example Training Loop

```
for epoch in range(num_epochs):
    model.train()  # Set model to train mode
    train_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device),
labels.to(device)
        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
    train_loss /= len(train_loader)
    # Validation loop
    model.eval()  # Set model to val mode
    val_loss = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device),
labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
        val_loss /= len(val_loader)
    # Checkpointing the model
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(),
'best_model.pth')
```

Data Types

Reduced vs Mixed Precision

- Reducing Memory Usage: Lower precision data types decrease the memory footprint, allowing for larger models or bigger batch sizes within the same GPU constraints, i.e. float16 vs float32.
- Accelerating Computations: Modern GPUs and hardware accelerators are optimized for lower precision, speeding up training and inference operations.
- Enabling Mixed Precision Training: Using a combination of lower and higher precision data types balances speed and accuracy, supported by automatic mixed precision (AMP) features in frameworks like PyTorch.
- Improving Throughput: Smaller data types reduce data transfer volumes, enhancing the overall data throughput and efficiency.
- Increasing Energy Efficiency: Lower precision operations are more power efficient (consideration for large scale models/efficiency at inference)



Automated Mixed Precision

```
import torch
# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()
...
#training loop
```

Parallelization

DataParallel

- Parallelize Across Multiple GPUs: DataParallel in PyTorch allows for the distribution of input data across multiple GPUs, enabling parallel processing that significantly speeds up training times by utilizing the combined computational resources.
- Automatic Splitting of Data: DataParallel automatically splits a batch of input data, sends each split to a different GPU, and then aggregates the results on the primary GPU. This simplifies the implementation of multi-GPU training without needing detailed manual configuration.
- Ease of Use: DataParallel is straightforward to implement; wrap the model with torch.nn.DataParallel(model). This wrapper handles the distribution and collection of data, making it user-friendly for those new to parallel processing.
- Limitations on Scale: While effective for small to medium-sized projects, DataParallel might not be as efficient for very large models or extremely high volumes of data due to its limitations in scaling and potential bottlenecks in data communication between GPUs.
- Best Suited for Single-Node Training: DataParallel is primarily designed for single-node, multi-GPU training.

DataParallel Example

```
import torch
# Wrap the model with DataParallel
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()}
GPUs!")
    model = torch.nn.DataParallel(model)
# Move the model to GPU if available
model.to(device=torch.device('cuda' if
torch.cuda.is_available() else 'cpu'))
```

DistributedDataParallel

- Designed for Multi-Node Scalability: DistributedDataParallel (DDP) in PyTorch is optimized for both multi-GPU and multi-node training, making it highly scalable and efficient for large-scale machine learning projects that require substantial computational resources.
- Efficient Communication: DDP minimizes communication overhead by synchronizing gradients only once at the end of the backward pass and leveraging efficient communication strategies such as ring-reduce to distribute gradients across multiple nodes and GPUs.
- Single Program, Multiple Data (SPMD) Model: DDP operates under the SPMD model, where the same program is run on each node, but each instance processes a different segment of the data. This model simplifies programming and deployment across complex distributed environments.
- Robust and Fault-Tolerant: DDP includes features to handle failures and restarts, making it more robust for training over large distributed systems where node or network failures might occur, thus ensuring that the training process can recover and continue.
- High Performance and Customizability: DDP provides better performance than DataParallel by reducing bottlenecks associated with GPU utilization and network communication. It also supports customization of gradient reduction and offers hooks for users to plug in different algorithms for gradient aggregation.

DDP Snippet

```
import torch
from torch.nn.parallel import DistributedDataParallel as DDP
rank = int(os.getenv('SLURM_PROCID'))
world_size = int(os.getenv('SLURM_NPROCS'))
local_rank = int(os.getenv('SLURM_LOCALID'))
model = models.resnet18(pretrained=True).to(local_rank)
model = DDP(model, device_ids=[local_rank])
```

Fully Sharded Data Parallel

- Memory Efficiency Enhancement: FullyShardedDataParallel (FSDP) in PyTorch is specifically designed to drastically reduce memory usage by sharding the model parameters, optimizer states, and gradients across all available GPUs. This allows training of much larger models than typically possible with traditional data parallel methods.
- Dynamic Parameter Sharding: Unlike standard sharding techniques that statically partition model parameters, FSDP implements dynamic sharding. This means that parameters are only loaded onto GPU memory when needed during the forward or backward pass, and are otherwise offloaded, further optimizing memory usage.
- Automatic Model Rebuilding: FSDP automatically reconstructs the full model state when necessary, such as for checkpointing or evaluation, handling the complexities of parameter gathering and scattering transparently to the user.
- Scalable Across Multiple Nodes: FSDP is designed to scale efficiently not only across multiple GPUs within a single node but also across multiple nodes, making it suitable for extremely large-scale distributed training scenarios.
- Integration with Mixed Precision Training: FSDP seamlessly integrates with mixed precision training techniques, allowing users to benefit from both sharded parameter management and the computational speedups of using lower precision (e.g., float16) calculations.

FSDP Example

```
import torch
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
import torchvision.models as models
def init_distributed_mode():
    torch.cuda.set_device(int(os.environ['SLURM_LOCALID']))
    dist.init_process_group(backend='nccl')
# Load a pre-trained ResNet18 model
model = models.resnet18(pretrained=True)
init_distributed_mode()
model = FSDP(model).cuda()
```

Comparison: DP vs DDP vs FSDP

- Takeaways:
  - DataParallel for single node training
  - DistributedDataParallel for multinode training
  - FSDP for most memory efficient multinode training

Feature	DataParallel	DistributedDataParallel (DDP)	FullyShardedDataParallel (FSDP)
Scalability	Limited to single-node, multi-GPU	Scales across multiple nodes and GPUs	Designed for very large-scale, multi-node
Memory Efficiency	Lower; replicates model on each GPU	Higher; each GPU handles a part of data	Very high; shards parameters across GPUs
Communication Overhead	High; gathers outputs on a single GPU	Lower; efficient gradient synchronization	Minimal; due to dynamic sharding of parameters
Implementation Complexity	Low; simple to implement	Moderate; requires more setup	High; complex setup for dynamic sharding
Optimal Use Case	Small to medium models	Large models or high data volumes	Extremely large models requiring maximum efficiency
Fault Tolerance	Low	High; can handle node failures	High; robust against failures, can recover
Integration with Precision	Basic support	Supports mixed precision	Deep integration with mixed precision

Just in Time Compilation

- Optimizing the computational graph, which merges operations and eliminates unnecessary computations, increasing efficiency.
- Reducing execution overhead by compiling the model into a lower-level language, avoiding runtime Python interpretation.
- Enabling cross-platform deployment with platform-independent TorchScript models.
- Leveraging hardware optimizations to utilize CPU or GPU features effectively.
- Improving handling of batched operations and parallelization, which decreases processing time.
- Reducing latency, crucial for real-time applications, by pre-optimizing the model's operations.

```
import torch
import torchvision.models as models
model = models.resnet18(pretrained=True)
model.eval()
scripted_model = torch.jit.script(model)
scripted_model.save('scripted_resnet18.pt')
```

Quantization

- Increases efficiency at Inference time, adds computational overhead at train time
- Efficiency Improvement: Quantization reduces the memory footprint and computational demands of neural networks by converting model weights and activations from high precision (e.g., float32) to lower precision (e.g., int8).
- Enhanced Inference Speed: By leveraging lower precision arithmetic, quantization significantly speeds up model inference, crucial for real-time applications.
- Application in Transfer Learning: Quantization can be applied to foundation models such as ResNet18 after they are fine-tuned for specific tasks, optimizing them for deployment without extensive retraining.
- Post-Training Quantization: This method involves applying quantization after the model has been adapted and fine-tuned, offering a straightforward path to efficiency with minimal retraining effort.
- Quantization Aware Training (QAT): Incorporates quantization effects during the fine-tuning process, allowing the model to adapt to quantization-induced changes and often preserving more accuracy.
- Broad Hardware Support: Quantized models are supported by a wide range of hardware accelerators and deep learning frameworks, facilitating deployment across diverse environments, including resource-constrained edge devices.

Quantization Aware Training

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.quantization import QuantStub, DeQuantStub
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=5, padding=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, padding=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 1024)
        self.fc2 = nn.Linear(1024, 10)
        self.quant = QuantStub()  # Quantization stub for input
        self.dequant = DeQuantStub()  # De-quantization stub for output
    def forward(self, x):
        x = self.quant(x)  # Quantize input
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.dequant(x)  # De-quantize output
        return x
```

Sign up for the Next Seminar