GPU Programming and distributed deep learning with PyTorch

Mahesh C

2023

Why Parallelism

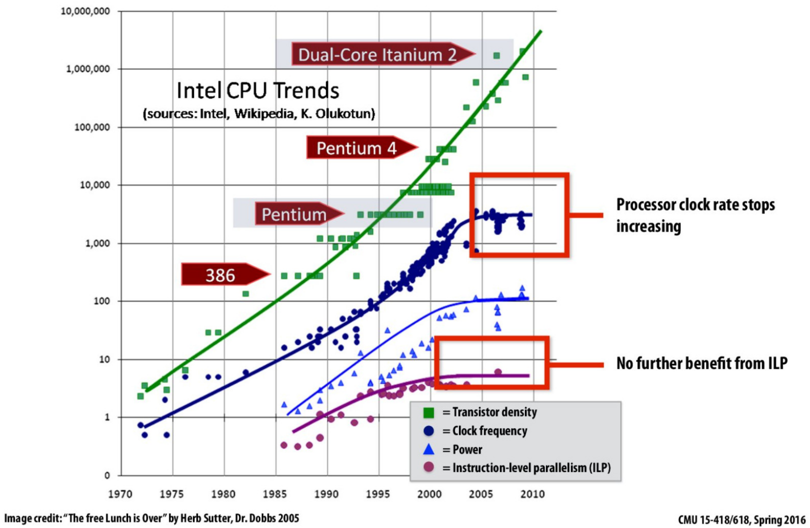


Figure: Intel CPUs

CPU vs GPU

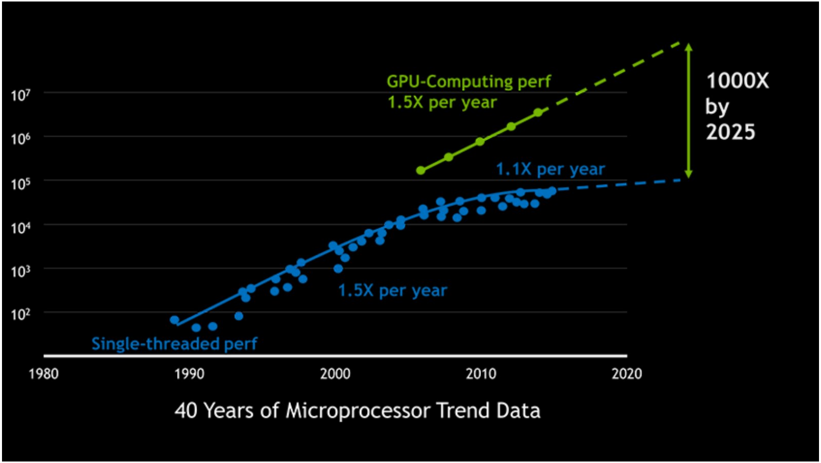


Figure: Performance Comparison , credits : GTC

Why PyTorch

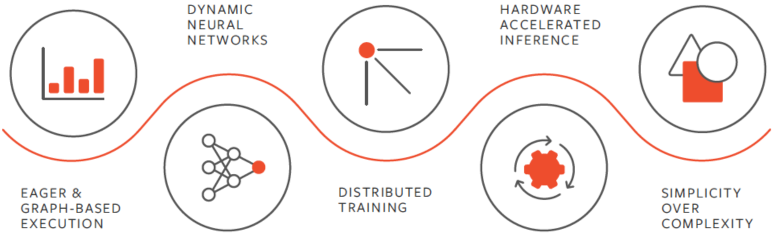


Figure: Credits : pytorch.org

Preparing the environment - Native

```
fisat@debian:~/class$ python3 -m venv env
fisat@debian:~/class$ source env/bin/activate
(env) fisat@debian:~/class$ pip install pytorch lightning torchvision
Collecting pytorch
  Downloading pytorch-1.0.2.tar.gz (689 bytes)
Collecting lightning
```

Figure: Commands

Preparing the environment - Docker

dockerfile

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install -y python3 python3-dev libgdal-dev
RUN apt-get install -y python3-pip
RUN pip install lightning torchvision
RUN apt-get install -y wget
CMD ["/usr/bin/python3"]
```

Figure: Dockerfile

$ commands

```
mkdir src
docker build --tag gpuclass .
docker run --gpus all  --rm -it --entrypoint bash -v `pwd`/src:/src gpuclass
```

Figure: Dockerfile

PyTorch- Basics

```
import torch
# Create tensors
tensor_a = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
tensor_b = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)
# Basic tensor operations
tensor_sum = tensor_a + tensor_b
tensor_diff = tensor_a - tensor_b
tensor_product = tensor_a * tensor_b
tensor_division = tensor_a / tensor_b
# Print the results
print("Tensor A:", tensor_a)
print("Tensor B:", tensor_b)
print("Sum:", tensor_sum)
print("Difference:", tensor_diff)
print("Product:", tensor_product)
print("Division:", tensor_division)
# Calculate the gradient
tensor_sum.backward(torch.ones_like(tensor_sum))
# Print the gradients
print("Gradient of Tensor A:", tensor_a.grad)
print("Gradient of Tensor B:", tensor_b.grad)
```

Figure: Basics

PyTorch- Basics - Simple ANN

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Step 1: Define the dataset
# Let's create a simple toy dataset with two features and binary labels.
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# Step 2: Define the neural network architecture
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(2, 4)  # Input layer with 2 features and 4 neurons in the hidden layer.
        self.fc2 = nn.Linear(4, 1)  # Output layer with 1 neuron for binary classification.

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))  # Apply sigmoid activation to the hidden layer.
        x = torch.sigmoid(self.fc2(x))  # Apply sigmoid activation to the output layer.
        return x

# Step 3: Create an instance of the model
model = SimpleNN()
```

Figure: ANN

```python
# Step 4: Define loss function and optimizer
criterion = nn.BCELoss()  # Binary Cross Entropy Loss for binary classification.
optimizer = optim.SGD(model.parameters(), lr=0.1)  # Stochastic Gradient Descent optimizer.
# Step 5: Training loop
num_epochs = 10000
for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X)
    loss = criterion(outputs, y)
    # Backpropagation and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 1000 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
# Step 6: Test the model
with torch.no_grad():
    test_inputs = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)
    predictions = model(test_inputs)
    predictions = (predictions > 0.5).float()  # Convert to binary (0 or 1) predictions
    print("Predictions:")
    print(predictions)
```

Figure: ANN

GPU training - Simple ANN

```
# Step 3: Create an instance of the model and move to GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = SimpleNN().to(device)

# Step 4: Move the data to the GPU
X = X.to(device)
y = y.to(device)
```

Figure: ANN

Multi GPU training - Distributed Data parallel (DDP)

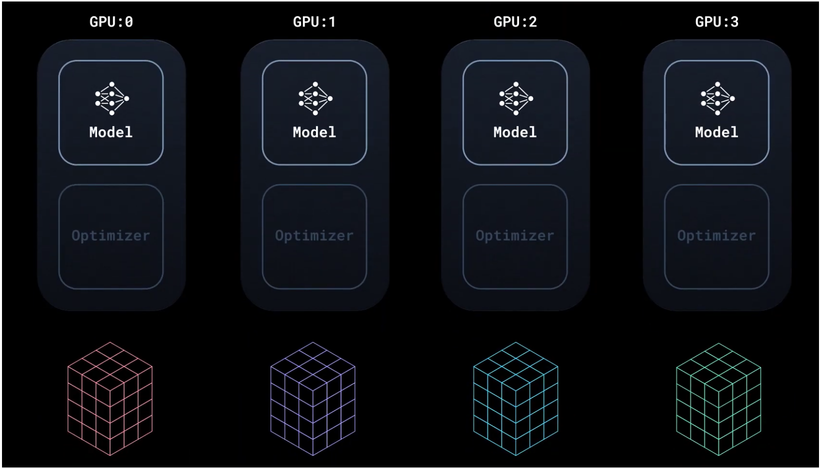


Figure: Credit: pytorch.org

Multi GPU training - Gradient Aggregation

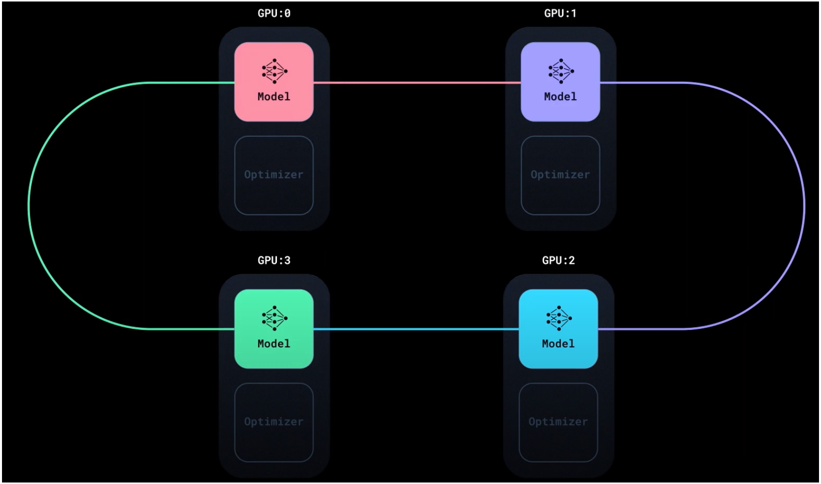


Figure: Credit: pytorch.org

Application of Chain Rule

Consider the function $w = x \cdot y + 2$.

1. Derivative of w with respect to x:

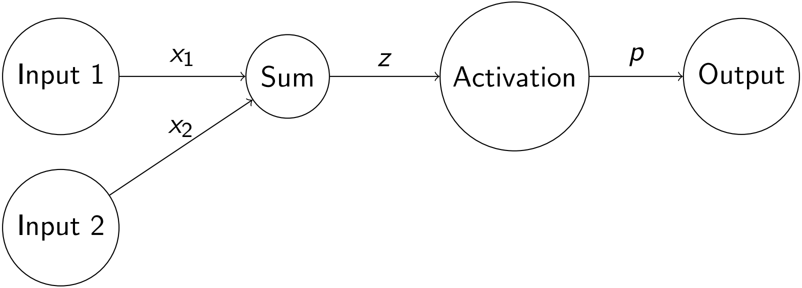
$$\frac{dw}{dx} = \frac{d}{dx}(x \cdot y) + \frac{d}{dx}(2) = y$$

2. Derivative of w with respect to y:

$$\frac{dw}{dy} = \frac{d}{dy}(x \cdot y) + \frac{d}{dy}(2) = x$$

These derivatives tell us how w changes concerning changes in x and y, respectively.

Single Perceptron (Two Inputs)



- **Input 1** (x_1): First input feature.
- **Input 2** (x_2): Second input feature.
- **Sum** (z): Weighted sum of inputs.
- **Activation**: Applies a non-linear function.
- **Output** (p): Final prediction.

Key Terms in Neural Network Training

Let's start by clarifying some key terms in neural network training:

- z: Represents the weighted sum of inputs to a neuron or layer before applying an activation function.
- θ: Denotes the vector of all network parameters, including weights and biases.
- Weighted Sum (z):

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n$$

- Activation Function: Applies non-linearity to z to produce the neuron's output.

$$\text{Output} = \text{Activation}(z)$$

- Training involves adjusting weights (θ) to minimize a loss function, optimizing network performance.

These terms play a crucial role in understanding the training process of neural networks.

Chain Rule for BCE Loss

The Binary Cross-Entropy (BCE) loss measures the dissimilarity between predicted probabilities and target binary values. It is defined as:

$$\text{BCELoss} = -\left(y \cdot \log(p) + (1-y) \cdot \log(1-p)\right)$$

To compute gradients, we apply the chain rule.

1. Calculate $\nabla_p \text{BCELoss}$:

$$\frac{\partial \text{BCELoss}}{\partial p} = -\left(\frac{y}{p} - \frac{1-y}{1-p}\right)$$

2. Calculate $\nabla_z \text{BCELoss}$:

$$\frac{\partial \text{BCELoss}}{\partial z} = \nabla_p \text{BCELoss} \cdot \frac{\partial p}{\partial z}$$

Weight Update Process

In training a neural network, we adjust the weights to minimize the loss. Here's the weight update process:

1. Calculate gradients:

$$\nabla_\theta \text{BCELoss} = \nabla_z \text{BCELoss} \cdot \nabla_\theta z$$

2. Update weights using an optimizer:

$$\theta_{\text{new}} = \theta_{\text{old}} - \text{Learning Rate} \times \nabla_\theta \text{BCELoss}$$

3. Learning Rate: - Controls step size during weight updates. - Smaller rates lead to slower convergence, larger rates may cause overshooting.
4. Iterative Process: - Repeat gradient calculation and weight updates for multiple epochs.
5. Convergence: - Training stops when loss reaches a threshold or after a fixed number of epochs.

Quantization

Quantization involves reducing the precision (number of bits) used to represent model weights. Typically, this is done to reduce the model's memory footprint and potentially speed up inference. There are two types of quantization.

1. post-training quantization
2. quantization-aware training

post-training quantization

```python
import torch
import torchvision.models as models
import torch.quantization as quantization
import time
from torchsummary import summary

# Load a pre-trained model (e.g., ResNet18)
model = models.resnet18(pretrained=True)

# Create a dummy input with the same shape as expected by the model
dummy_input = torch.randn(1, 3, 224, 224)  # Batch size 1, 3 channels, 224x224 image

# Prepare the model for quantization
quantized_model = quantization.quantize_dynamic(
    model,  # Original pre-trained model
    {torch.nn.Conv2d, torch.nn.Linear},  # Specify which layers to quantize
    dtype=torch.qint8  # Specify the quantization data type (int8)
)

# Set the model to evaluation mode
quantized_model.eval()
```

Figure: post training

QAT

```python
import torch.quantization as quantization
# Enable quantization-aware training (QAT)
model.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')
quantized_model = quantization.prepare_qat(model)
```

Figure: QAT

Introduction to Pruning

Pruning is a technique for reducing the size of a neural network by removing less important connections or neurons. It involves identifying and eliminating parameters (weights or neurons) that contribute less to the model's overall performance.

Unstructured random

prune at random 30% of the connections in the parameter named weight in the conv1 layer.

```python
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square conv kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5x5 image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, int(x.nelement() / x.shape[0]))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = LeNet().to(device=device)

module = model.conv1
prune.random_unstructured(module, name="weight", amount=0.3)
```

Figure: Pruning

Unstructured L1

prune the 3 smallest entries in the bias by L1 norm

```
prune.l1_unstructured(module, name="bias", amount=3)
```

Figure: Pruning

structured pruning along the 0th axis of the tensor (the 0th axis corresponds to the output channels of the convolutional layer and has dimensionality 6 for conv1), based on the channel's L2 norm

```
prune.ln_structured(module, name="weight", amount=0.5, n=2, dim=0)
```

Figure: Pruning

Global pruning

lowest 20% of connections across the whole model, instead of removing the lowest 20

```
parameters_to_prune = (
    (model.conv1, 'weight'),
    (model.conv2, 'weight'),
    (model.fc1, 'weight'),
    (model.fc2, 'weight'),
    (model.fc3, 'weight'),
)

prune.global_unstructured(
    parameters_to_prune,
    pruning_method=prune.L1Unstructured,
    amount=0.2,
)
```

Figure: Pruning

Mixed-precision training

Mixed-precision training is a technique that leverages both single-precision (float32) and reduced-precision (e.g., float16) data types to accelerate deep learning model training while conserving memory.

```
# Enable mixed-precision training with Apex
model, optimizer = amp.initialize(model, optimizer, opt_level="O2")  # "O2" enables mixed-precision training
```

Figure: APEX

```
with amp.scale_loss(loss, optimizer) as scaled_loss:
    scaled_loss.backward()  # Backpropagate using mixed precision
optimizer.step()
```

Figure: APEX