Parallel ML

Maciej Marchwiany, PhD

Plan

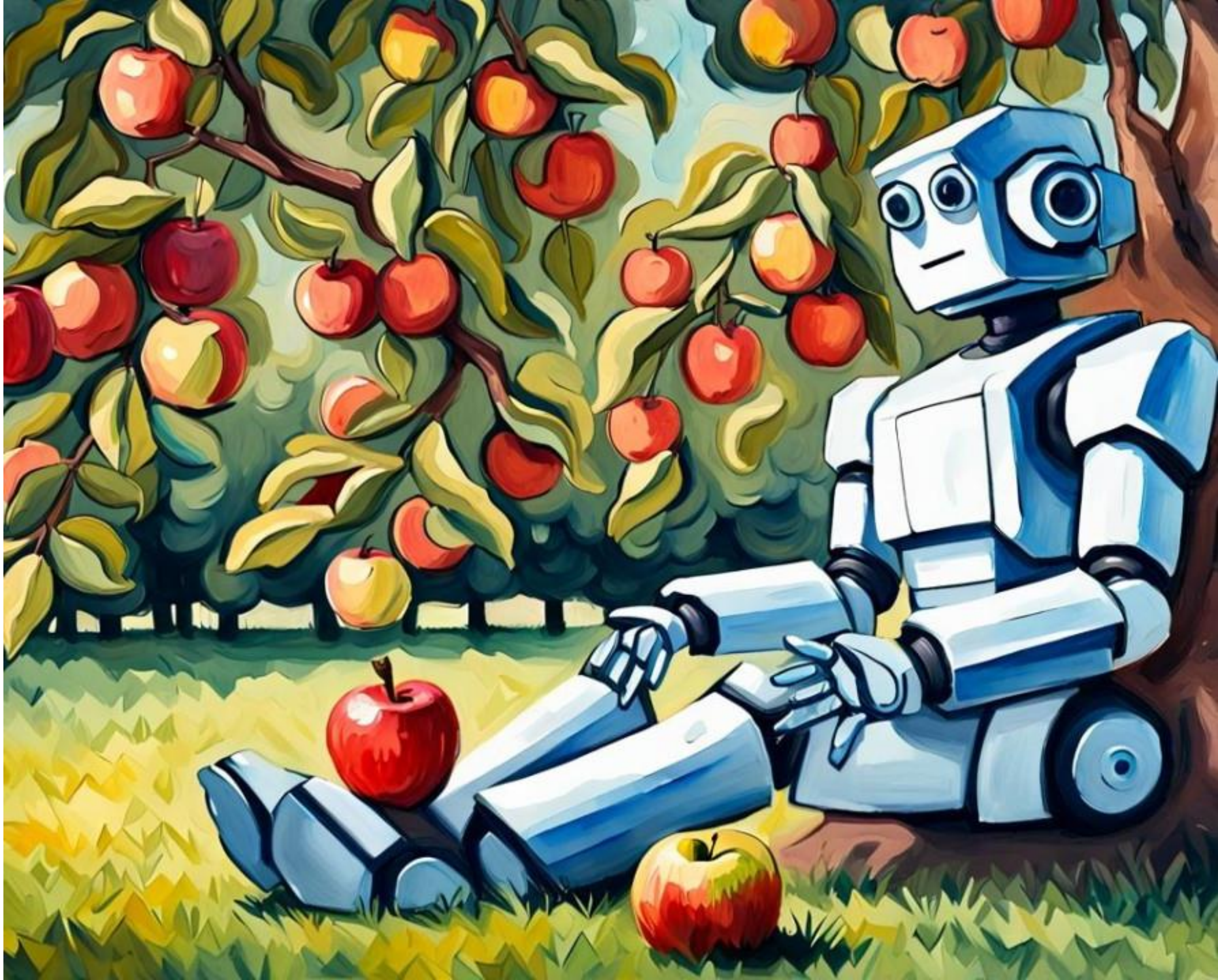
Scikit learn

Data parallelization

Model parallelization

Horovod

Scikit learn



Parallel algorithms

1. **Ensemble Methods**
   1. Random Forest - Uses parallelism for training multiple decision trees.
      1. `RandomForestClassifier`
      2. `RandomForestRegressor`
   2. Gradient Boosting - Parallelizes only over data samples, not trees.
      1. `GradientBoostingClassifier`
      2. `GradientBoostingRegressor`
   3. Histogram-Based Gradient Boosting - Uses parallel histogram-based training.
      1. `HistGradientBoostingClassifier`
      2. `HistGradientBoostingRegressor`
   4. Extra Trees - Uses parallelism for training multiple decision trees
      1. `ExtraTreesClassifier`
      2. `ExtraTreesRegressor`

Parallel algorithms

1. Nearest Neighbors
    1. K-Neighbors - Uses parallelism for distance computations.
        1. `KNeighborsClassifier`
        2. `KNeighborsRegressor`

Parallel algorithms

1. **Linear Models**
   1. Ridge Regression with cross-validation - Uses threads parallelism.
      1. `RidgeCV`

Parallel algorithms

1. **Clustering**
   1. K-means - Uses threads parallelizm.
      1. `KMeans`
      2. `MiniBatchKMeans`

Parallel algorithms

1. **Dimensionality Reduction**
   1. TruncatedSVD - parallel SVD decomposition.
      1. `TruncatedSVD`

Parallel algorithms

1. **Cross-Validation & Hyperparameter Tuning**
   1. Cross-Validation.
      1. `cross_val_score`
   2. Grid Search Cross-Validation.
      1. `GridSearchCV`
   3. Randomized Search Cross-Validation.
      1. `RandomizedSearchCV`

Parallel algorithms

Algorithm	Parallel Support?	Notes
RandomForestClassifier/Regressor	✅ Full	Uses multiple trees
ExtraTreesClassifier/Regressor	✅ Full	Faster than Random Forest
HistGradientBoostingClassifier/Regressor	✅ Full	More efficient than GBM
KNeighborsClassifier/Regressor	✅ Full	Parallelizes distance computations
GradientBoostingClassifier/Regressor	🚫 Limited	Only data parallelism
LinearSVC / LinearSVR	🚫 Limited	Multi-threaded, no n_jobs
SVC / SVR	❌ No	No parallelism
DBSCAN / AgglomerativeClustering	❌ No	Single-threaded
KMeans	✅ Full	Uses OpenMP for multi-threading
PCA	❌ No	No parallelism
TruncatedSVD	✅ Full	n_jobs for parallel SVD
GridSearchCV / RandomizedSearchCV	✅ Full	Parallel hyperparameter tuning
cross_val_score	✅ Full	Parallel cross-validation

Parall
algori

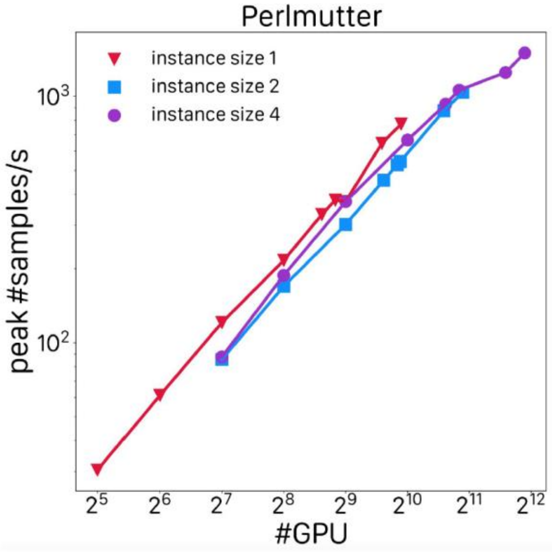
```
import numpy as np.
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score


# Generate a synthetic dataset
X, y = make_classification(n_samples=10000, n_features=20, random_state=42)


# Initialize a Random Forest Classifier with parallelism
rf = RandomForestClassifier(n_estimators=100, n_jobs=-1, random_state=42)


# Perform 5-fold cross-validation in paralel
scores = cross_val_score(rf, X, y, cv=5, n_jobs=-1)


# Print results
print(f"Cross-validation scores: {scores}")
print(f"Mean accuracy: {np.mean(scores):.4f}")
```

Why parallel training?

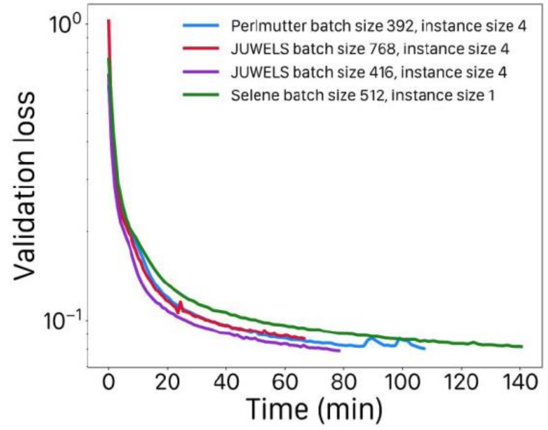
FourCastNet: Large-compute scaling

Kurth et al. 2022
arXiv:2208.05419
PASC 2023 Plenary Video

Scaled to e.g. 3808 GPUs on Perlmutter with model parallel on 4-gpus

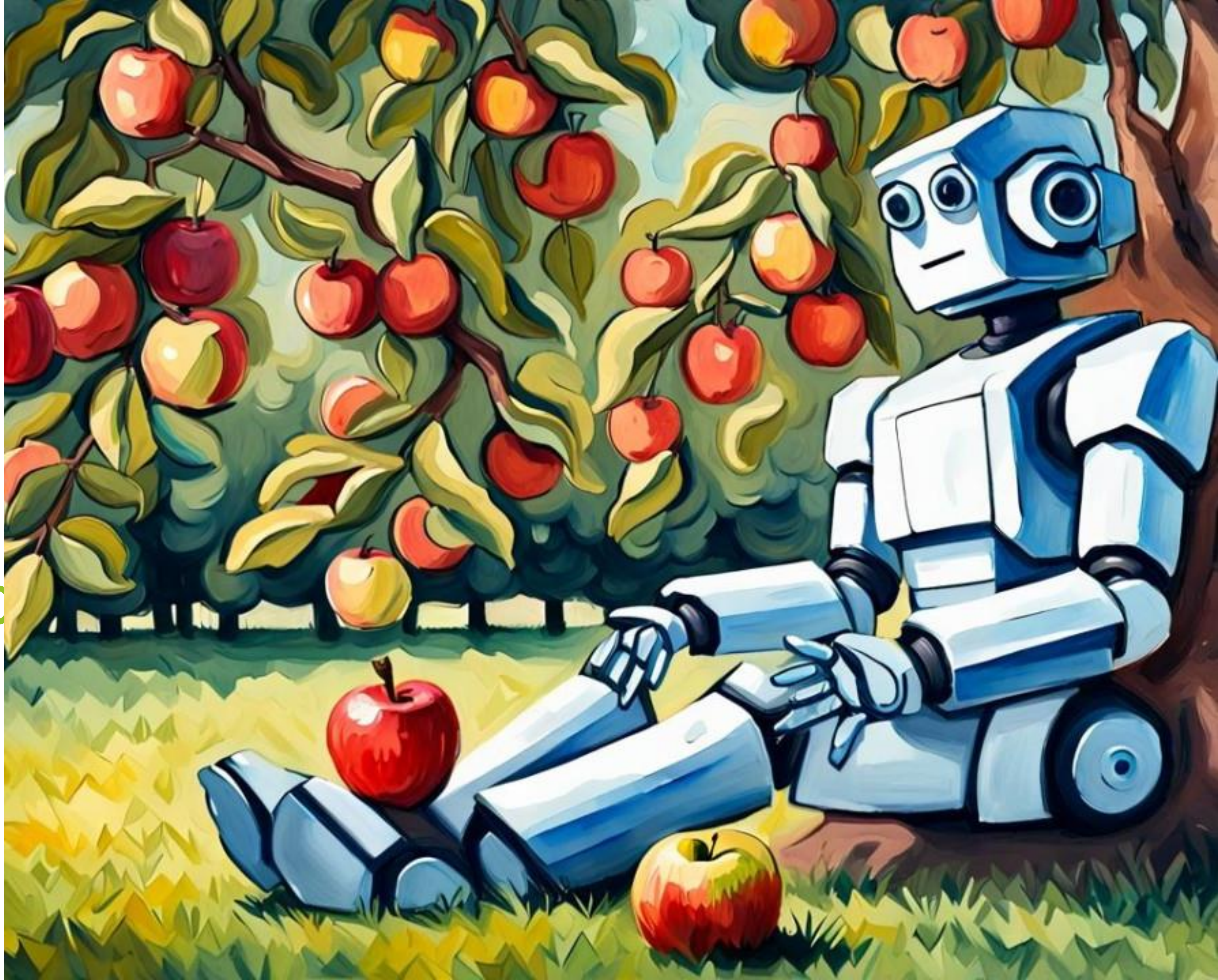


Train large models on ~1hr timescales
compared to 40 hrs on 32 nodes or >~45days on a single GPU:



Model and weights made available to community at
https://github.com/NVlabs/FourCastNet

Data parallelizatio



Data parallelization

Data parallelization is a technique where large datasets are **split across multiple processors or machines**, allowing computations (such as model training or inference) to happen in parallel. This helps speed up training and processing times, especially when dealing with **big data**.

How Data Parallelization Works

1. **Dataset Splitting** → The dataset is divided into smaller batches, each sent to a different worker (GPU, CPU, or cluster node).
2. **Model Duplication** → The **entire model** is copied onto each worker.
3. **Parallel Training** → Each worker **independently computes gradients** on its assigned mini-batch.
4. **Gradient Aggregation** → After each forward & backward pass, gradients from all workers are **averaged and synchronized**.
5. **Model Update** → The **updated parameters** are sent back to all workers, ensuring all models stay synchronized.

This technique is especially useful in **Scikit-Learn**, **Deep Learning (PyTorch, TensorFlow)**, and **distributed computing frameworks** like **Dask** or **Spark**.

Synchronization

- Synchronous updates
  - stable convergence
  - can be decentralized (allreduce)
  - computation may be blocked by communication
- Asynchronous updates
  - no waiting for gradients
  - state gradients affect convergence
  - parameter server can be a bottleneck
- Delayed-synchronous updates
  - Lagged gradients allow better comms overlap
  - stale gradients affect convergence

When to Use Data Parallelization?

- When working with **large datasets**
- When training **complex models** that require **distributed computing**
- When using **multiple CPUs or GPUs** to speed up computation

PyTorch

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

# Sample data
X = torch.randn(10000, 10)
y = torch.randint(0, 2, (10000,))

# Create a simple neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc = nn.Linear(10, 2)

    def forward(self, x):
        return self.fc(x)

# Use multiple GPUs for training
model = SimpleNN()
model = nn.DataParallel(model)  # Enables multi-GPU parallelism

# Train with DataLoader (parallelized batching)
train_loader = DataLoader(
        TensorDataset(X, y), batch_size=32,
        shuffle=True, num_workers=4)

# Training loop
optimizer = optim.Adam(model.parameters())
loss_fn = nn.CrossEntropyLoss()
```

PyTorch

```
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc = nn.Linear(10, 2)

    def forward(self, x):
        return self.fc(x)

# Use multiple GPUs for training
model = SimpleNN()
model = nn.DataParallel(model)  # Enables multi-GPU parallelism

# Train with DataLoader (parallelized batching)
train_loader = DataLoader(
         TensorDataset(X, y), batch_size=32,
         shuffle=True, num_workers=4)

# Training loop
optimizer = optim.Adam(model.parameters())
loss_fn = nn.CrossEntropyLoss()

for epoch in range(5):
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = loss_fn(outputs, batch_y)
        loss.backward()
        optimizer.step()
```

Limitations

For large-scale deep learning, traditional Data Parallelism (like torch.nn.DataParallel) has limitations, such as:

- Memory inefficiency (each GPU stores the full model).
- Slow communication overhead (synchronizing gradients across GPUs).

Advanced techniques solve these issues by optimizing memory usage, reducing communication overhead, and improving scalability.

FSDP

Fully Sharded Data Parallelism

FSDP **shards both model parameters and gradients across GPUs**, unlike traditional Data Parallelism, which keeps full copies.

Key Benefits

- **Lower Memory Usage** – Each GPU stores only a part of the model.
- **Scalability** – Trains LLMs like GPT-3, LLaMA on multiple GPUs.
- **Gradient Communication Efficiency** – Reduces sync time.

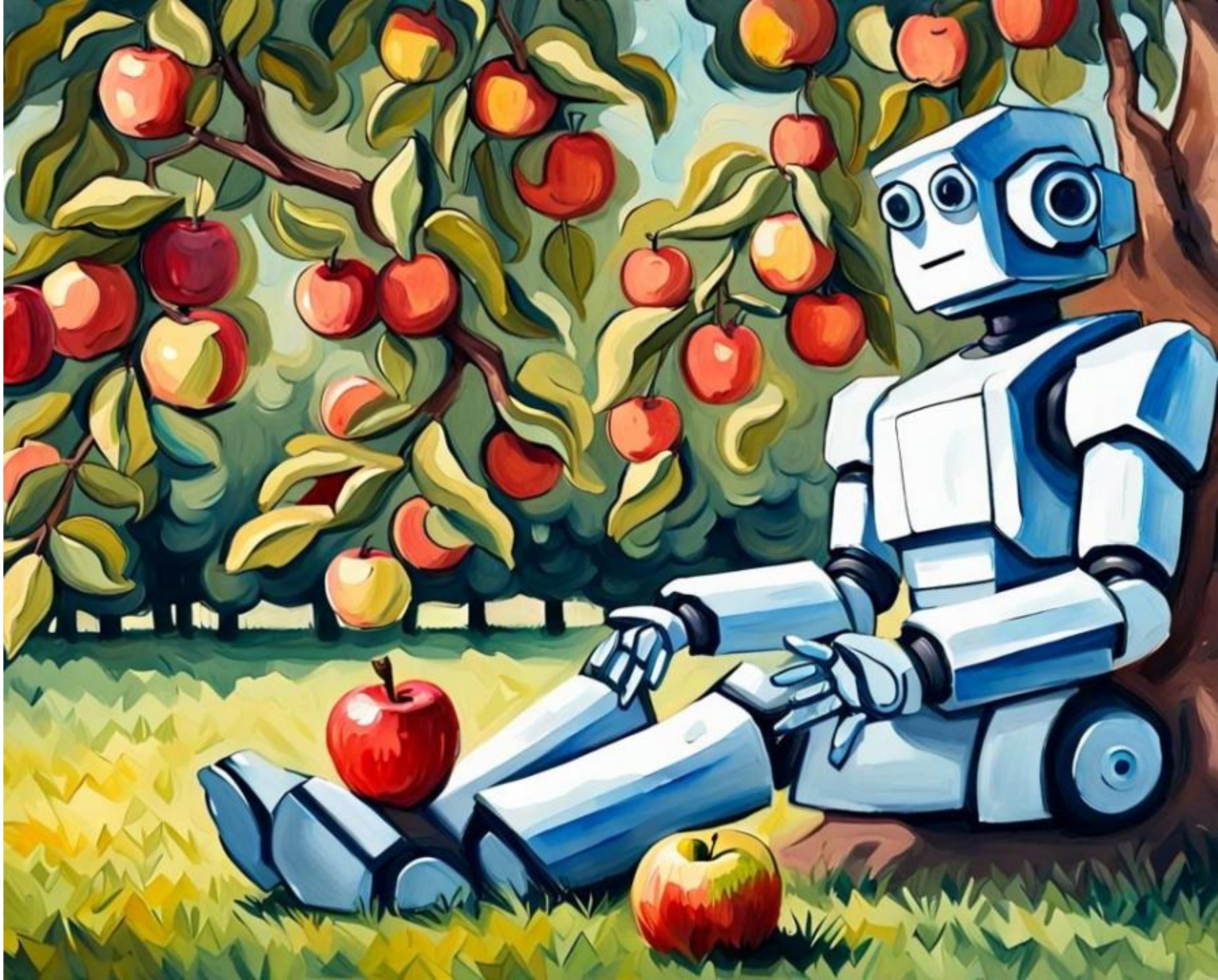
FSDP

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributed.fsdp import FullyShardedDataParallel
as FSDP
from torch.distributed.fsdp.fully_sharded_data_parallel
import CPUOffload

# Initialize distributed process group
torch.distributed.init_process_group(backend="nccl")


# Define a simple model
class Transformer(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1024, 4096)
        self.fc2 = nn.Linear(4096, 1024)

    def forward(self, x):
        return self.fc2(torch.relu(self.fc1(x)))


# Move model to FSDP (fully sharded)
device = torch.device("cuda")
model = Transformer().to(device)
```

FSDP

```python
        return self.fc2(torch.relu(self.fc1(x)))


# Move model to FSDP (fully sharded)
device = torch.device("cuda")
model = Transformer().to(device)
model = FSDP(model,
cpu_offload=CPUOffload(offload_params=True))  # Moves params
to CPU to save memory


# Define optimizer & loss
optimizer = optim.AdamW(model.parameters(), lr=3e-4)
criterion = nn.MSELoss()

# Dummy data for training
x = torch.randn(32, 1024).to(device)
target = torch.randn(32, 1024).to(device)


# Forward & backward pass
output = model(x)
loss = criterion(output, target)
loss.backward()
optimizer.step()

print(f"Training done with FSDP! Loss: {loss.item()}")
```

Model parallization



Model parallelization

Model parallelization is a technique where **different parts of a model are distributed across multiple processors, GPUs, or machines** to handle large-scale computations that a single device cannot manage efficiently.

How Model Parallelization Works

1. **Splitting the Model**: Different layers or parts of the model are assigned to different processors/GPUs.
2. **Parallel Computation**: Each processor computes only the assigned portion of the model.
3. **Communication & Synchronization**: The outputs of one stage (or layer) are transferred to the next stage, ensuring correct computations.

Model parallelization is mostly used in **deep learning**, where large models like GPT, BERT, or ResNets exceed a single device's memory capacity.

Types of Model Parallelism

- **Tensor Parallelism (TP) – Splitting Matrices Across GPUs**
  - Splits **individual tensors (weights, activations, gradients)** across multiple GPUs.
  - Common in **Transformer models (GPT-3, LLaMA, ViTs)**, where matrix multiplications are expensive.
  - **A single layer is spread across multiple GPUs**, so each device computes only part of the operation.
- **Pipeline Parallelism - Splitting Model Layers Across GPUs**
  - **Divides different layers** of the model across GPUs.
  - Each GPU **performs a stage** of forward and backward computation.
- **Expert Parallelism (Mixture of Experts, MoE)**
  - Divides the model into **multiple expert networks**, where each GPU only computes **a subset of the experts**.
  - Only a **few experts** are active per forward pass → **Huge memory savings!**
  - Used in **GPT-4, GLaM, Switch Transformer, and DeepSpeed MoE**.

Tensor Parallelism

```
from megatron import initialize_megatron, get_args
from megatron.model import GPTModel

# Initialize Megatron
initialize_megatron(extra_args_provider=None)

# Set model parallelism (split tensors across GPUs)
args = get_args()
args.tensor_model_parallel_size = 2  # Use 2 GPUs for tensor parallelism

# Define GPT model with tensor parallelism
model = GPTModel(num_layers=24, hidden_size=1024, num_attention_heads=16)

print("Model initialized with Tensor Parallelism!")
```

Pipeline Parallelism

```
import torch
import torch.nn as nn

# Define a model with different layers assigned to different GPUs
class ModelParallelNN(nn.Module):
    def __init__(self):
        super(ModelParallelNN, self).__init__()
         # First layer on GPU 0
        self.layer1 = nn.Linear(1024, 512).to('cuda:0’)
        # Second layer on GPU 1
self.layer2 = nn.Linear(512, 256).to('cuda:1’)
         # Third layer on GPU 1
        self.layer3 = nn.Linear(256, 10).to('cuda:1')

    def forward(self, x):
        x = x.to('cuda:0')  # Send input to GPU 0
        x = self.layer1(x)
        x = x.to('cuda:1')  # Move output to GPU 1
        x = self.layer2(x)
        x = self.layer3(x)
        return x

# Create model
model = ModelParallelNN()

# Generate sample input
x = torch.randn(64, 1024).to('cuda:0')  # Batch of 64 samples


# Forward pass
output = model(x)
print(output.shape)
```

Pipeline Parallelism

```
import torch
import torch.nn as nn
from torch.distributed.pipeline.sync import Pipe

# Define a simple deep model
class DeepModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.block1 = nn.Sequential(nn.Linear(1024, 2048), nn.ReLU())
        self.block2 = nn.Sequential(nn.Linear(2048, 1024), nn.ReLU())

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

# Split model into pipeline stages across 2 GPUs
model = DeepModel()
model = Pipe(model, chunks=2, balance=[1, 1], devices=[0, 1])

print("Model initialized with Pipeline Parallelism!")
```

Expert Parallelism

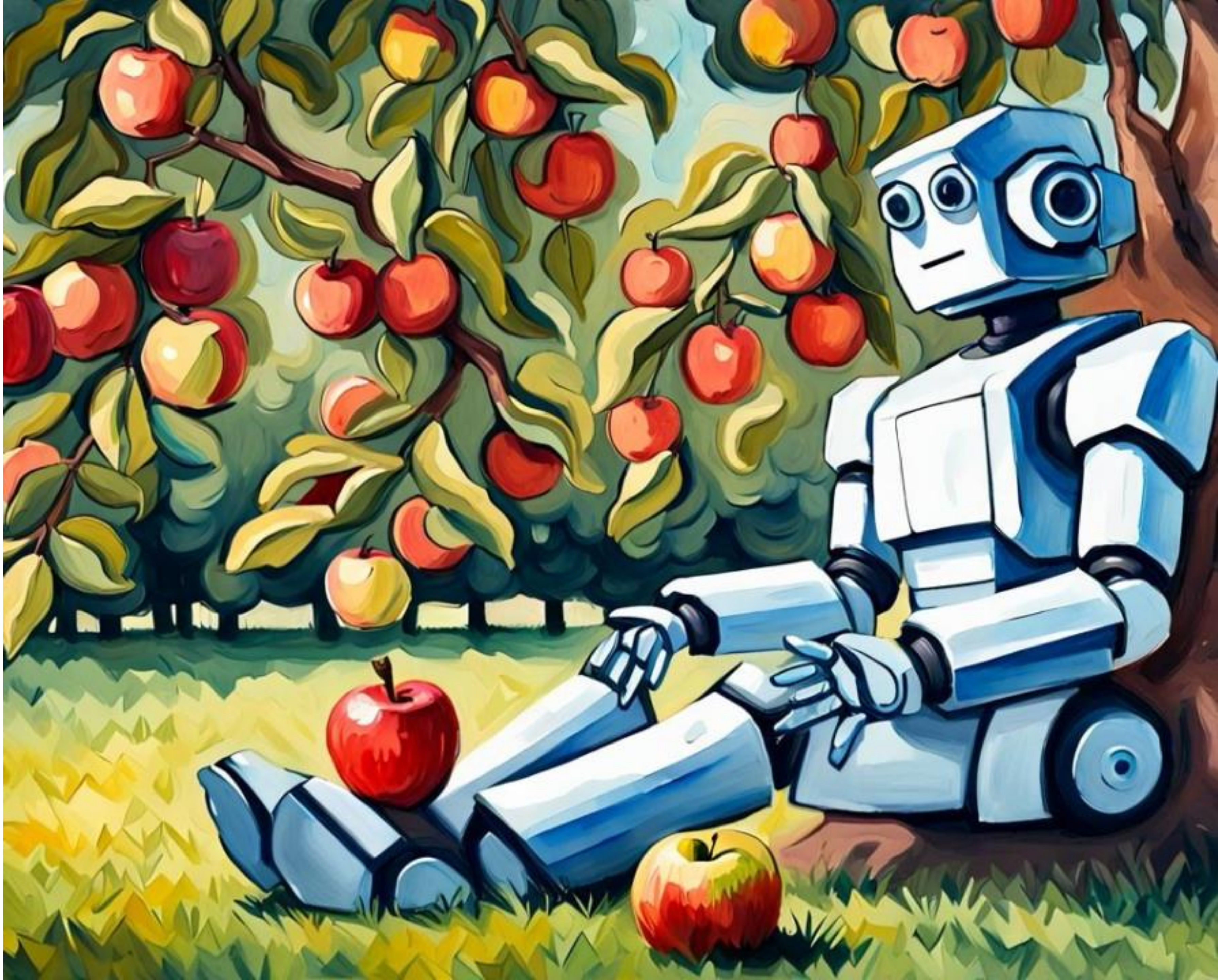
```
import torch
import torch.nn as nn


class MixtureOfExperts(nn.Module):
    def __init__(self, num_experts=4):
        super().__init__()
        self.experts = nn.ModuleList(
                [nn.Linear(1024, 1024)
                 for _ in range(num_experts)])
        self.gate = nn.Linear(1024, num_experts)  # Gating function

    def forward(self, x):
        gate_scores = torch.softmax(self.gate(x), dim=-1)
        output = sum(gate_scores[:, i].unsqueeze(1)
                 * self.experts[i](x) for i in range(len(self.experts)))
        return output


model = MixtureOfExperts()
print("MoE Model Initialized!")
```

Hybrid Parallelism

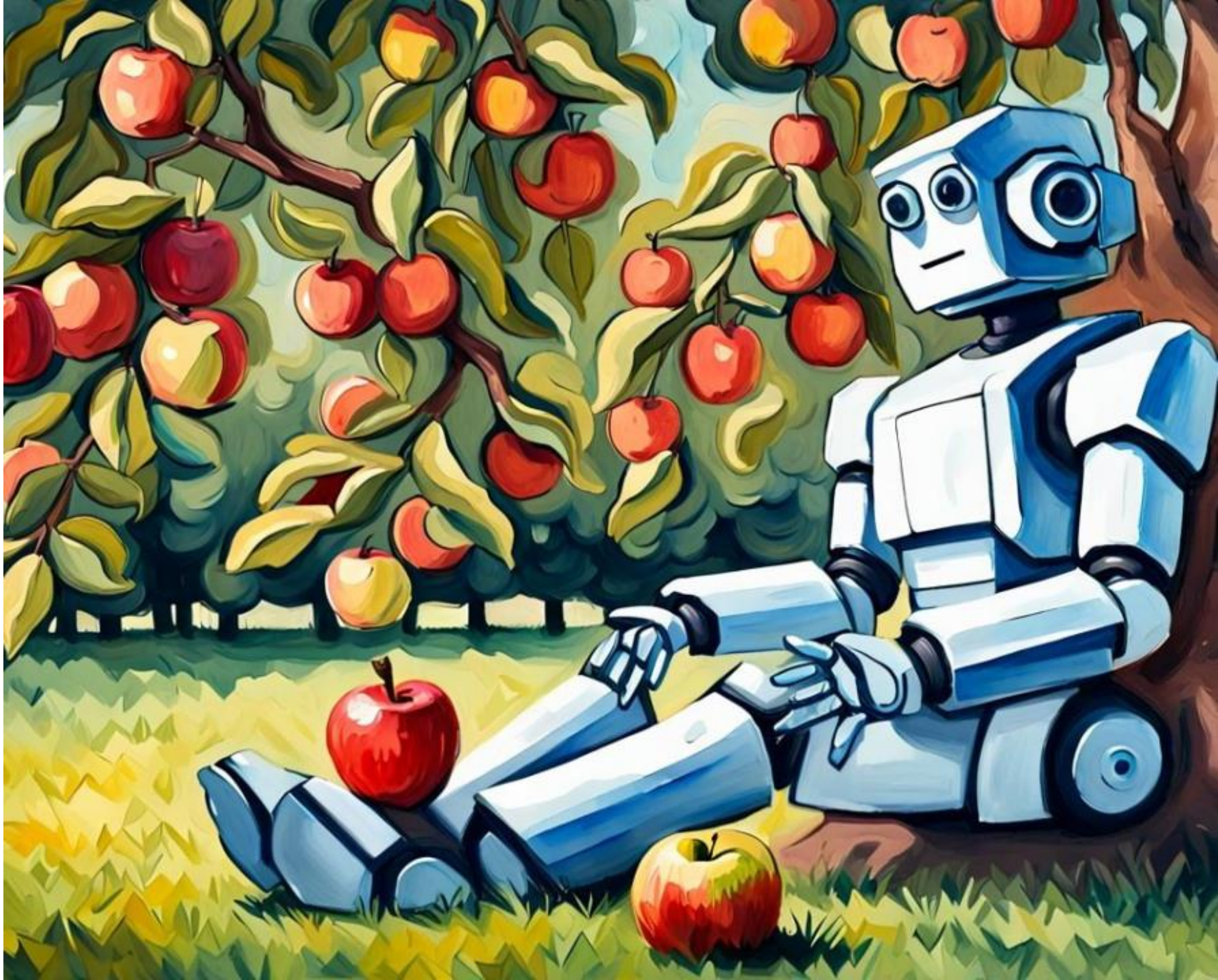
For **massive models like GPT-4, LLaMA-65B, PaLM-540B**, we **combine multiple parallelization techniques:**

- **Data Parallelism** – Distribute data across GPUs.
- **Tensor Parallelism** – Split tensors across GPUs.
- **Pipeline Parallelism** – Distribute layers across GPUs.
- **Mixture of Experts (MoE)** – Activate only a subset of experts per forward pass.

Horovod



Bottleneck



Bottlenecks for ML speed

1. Computational Power and Hardware Limitations
    - **GPU and CPU Bottlenecks**: While GPUs are powerful for ML computations, they can be underutilized due to slow data input/output (I/O) operations. CPUs are often insufficient for handling pre-processing tasks efficiently, leading to bottlenecks in data preparation.
    - **Memory Bottlenecks**: The disparity between processing speeds and memory access speeds creates significant bottlenecks. As models become more complex, memory bandwidth becomes a limiting factor.
2. Data Ingestion and Storage
    - **I/O Bottlenecks**: The speed at which data can be loaded into memory is often slower than the computation speed of GPUs. This results in GPUs waiting for data, reducing overall efficiency.
    - **Storage Limitations**: Large datasets exceed DRAM capacity, causing I/O bottlenecks during training. Solutions like massively parallel storage systems are being developed to address this.
3. Software Optimization and Integration
    - **Software Frameworks**: Efficient software frameworks are crucial for maximizing AI inference performance. However, optimizing software for specific hardware configurations remains a challenge.
    - **Deployment Challenges**: A significant bottleneck is the transition from model development to production. This often involves disconnects between data scientists and IT teams.

Bottlenecks for ML speed

- **Storage Limitations**: Large datasets exceed DRAM capacity, causing I/O bottlenecks during training. Solutions like massively parallel storage systems are being developed to address this.

3. Software Optimization and Integration
   - **Software Frameworks**: Efficient software frameworks are crucial for maximizing AI inference performance. However, optimizing software for specific hardware configurations remains a challenge.
   - **Deployment Challenges**: A significant bottleneck is the transition from model development to production. This often involves disconnects between data scientists and IT teams.
4. Data Preprocessing
   - **CPU Preprocessing**: Preprocessing data using CPUs can be a major bottleneck. Solutions like offloading preprocessing to GPUs are being explored.