Speed up ML

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

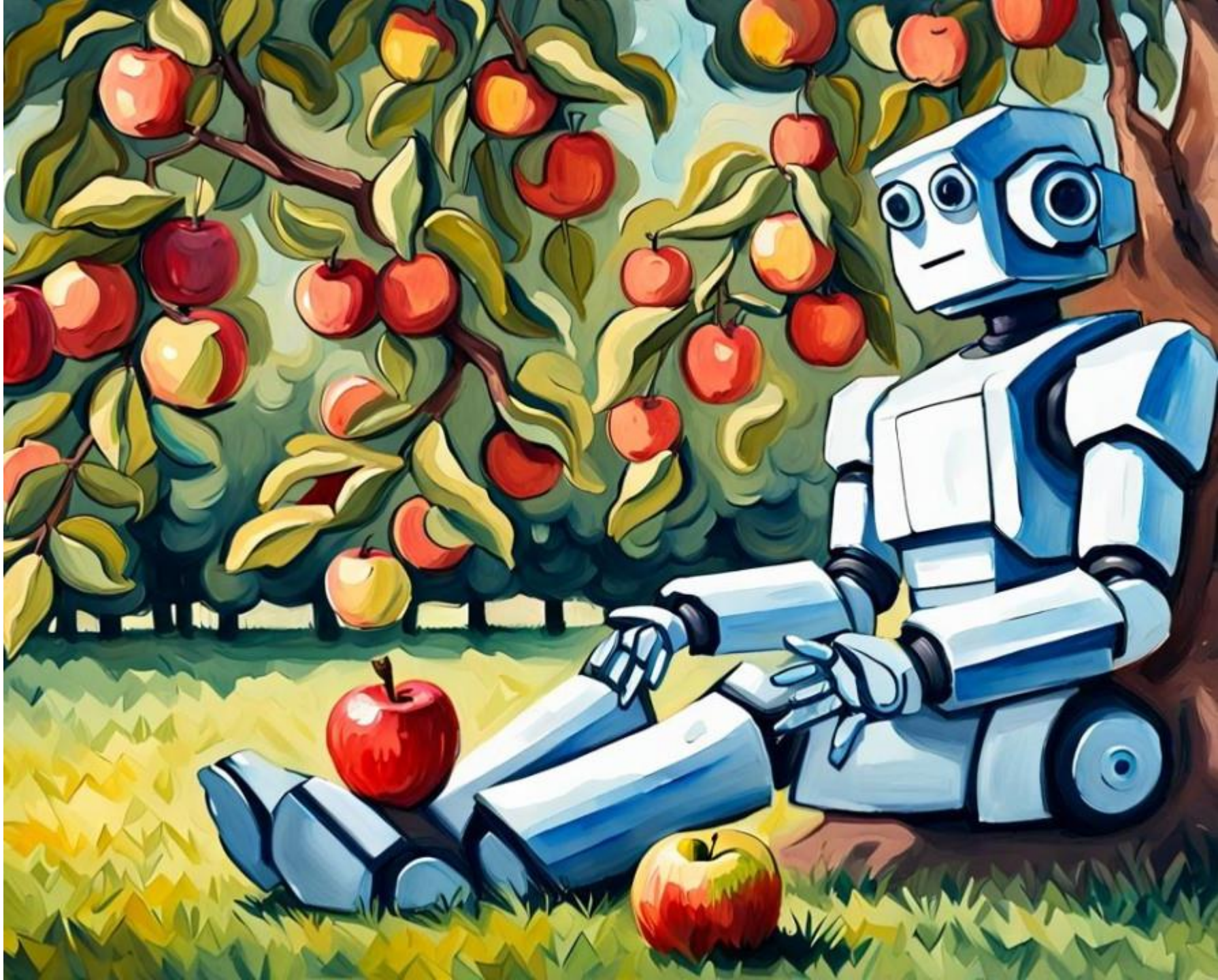
bottlenecks

solutions

solutions

…

Bottleneck



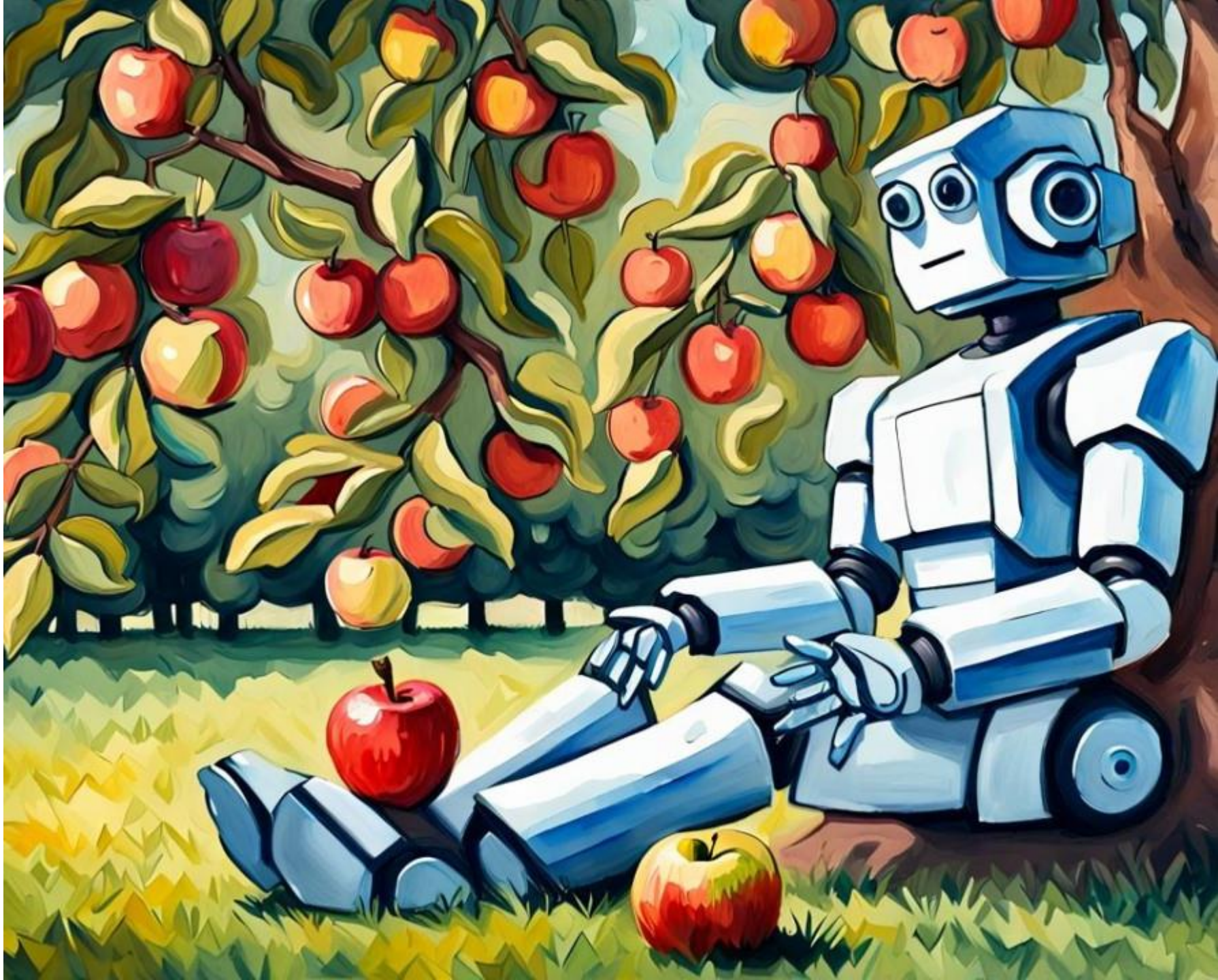
Bottlenecks for ML speed

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

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.
4. Data Preprocessing
   - **CPU Preprocessing**: Preprocessing data using CPUs can be a major bottleneck. Solutions like offloading preprocessing to GPUs are being explored.

Computational Power and Hardware Limitations



GPU and CPU Bottlenecks

Root Cause:

- GPUs are designed for parallel processing, handling thousands of tasks simultaneously. However, their efficiency is limited by how fast data can be transferred from CPUs and storage devices.
- CPUs are responsible for coordinating tasks and preprocessing data before it reaches the GPU. If CPUs are slow, they create a bottleneck in the ML pipeline.

Effects:

- The GPU often remains idle, waiting for data instead of processing it.
- Training time increases due to slow data transfer between CPU, memory, and GPU.

Solutions:

- Use high-bandwidth memory (HBM) in GPUs to improve data transfer speeds.
- Implement optimized CPU-GPU communication using techniques like NVIDIA's GPUDirect RDMA.
- Use specialized AI accelerators such as TPUs (Tensor Processing Units) for better parallel processing.

Memory Bottlenecks

Root Cause:

- The speed at which memory can supply data to the processor is much slower than the speed of computation.
- Complex deep learning models require high memory bandwidth, and slow memory access reduces GPU efficiency.

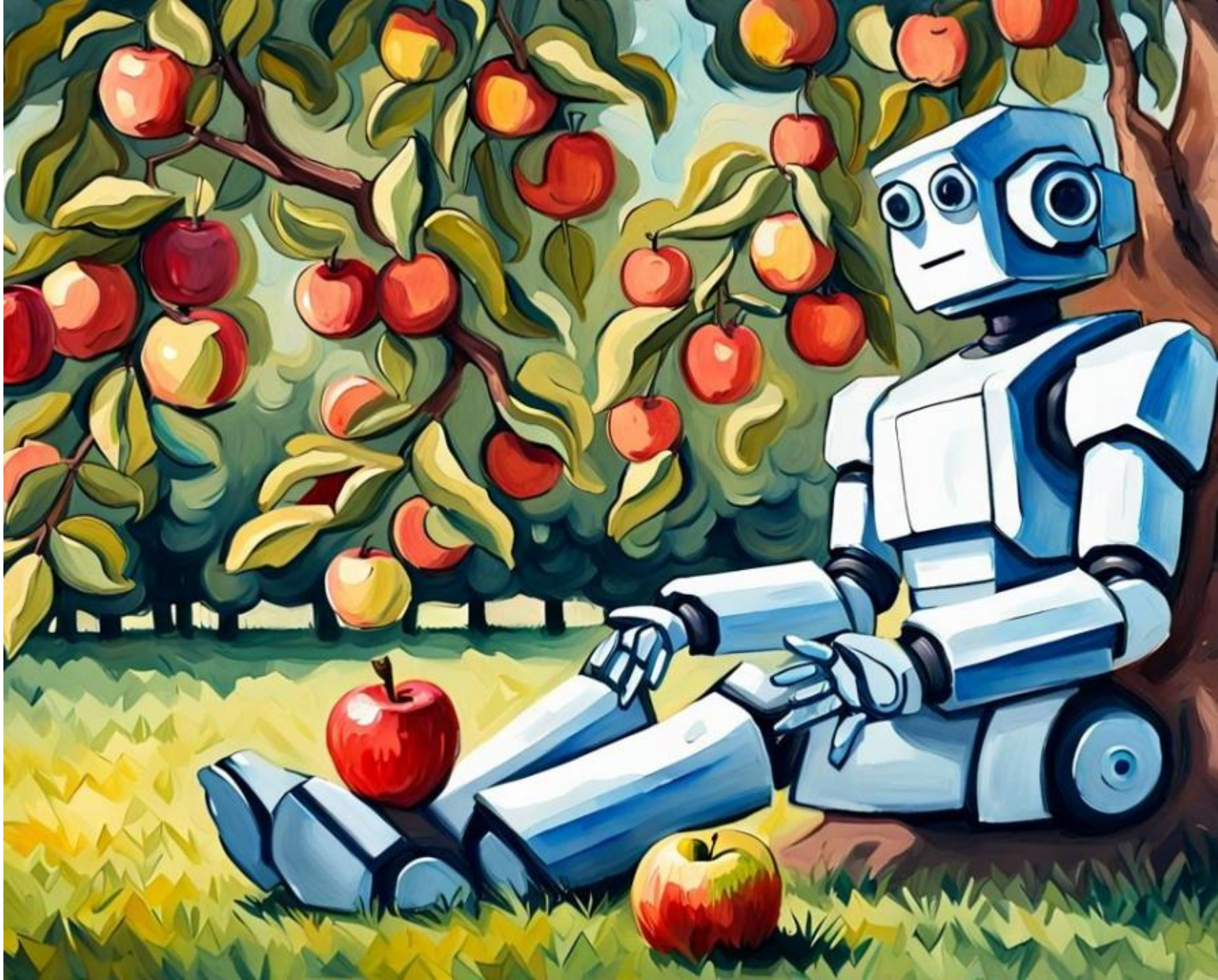
Effects:

- Increased latency in training and inference.
- Frequent memory swapping between RAM and disk storage, further slowing down processing.

Solutions:

- Use GPUs with higher VRAM capacity.
- Optimize memory management by batching data efficiently.
- Leverage memory-efficient frameworks such as TensorRT to reduce memory footprint.

Data Ingestion and Storage



I/O Bottlenecks

Root Cause:

- Data is stored on traditional hard drives (HDDs) or network-attached storage (NAS), which have limited read/write speeds.
- Even SSDs, while faster than HDDs, still have lower bandwidth than what GPUs require.

Effects:

- Training times increase due to the GPU waiting for data.
- Reduced throughput in real-time inference applications.

Solutions:

- Use NVMe SSDs or RAM-based storage to speed up data access.
- Implement memory-mapped files to accelerate dataset loading.
- Use data pipelines like TensorFlow's tf.data to prefetch and stream data efficiently.

Storage Limitations

Root Cause:

- Large datasets exceed system memory (DRAM) capacity.
- Systems must constantly move data between storage and RAM, creating bottlenecks.

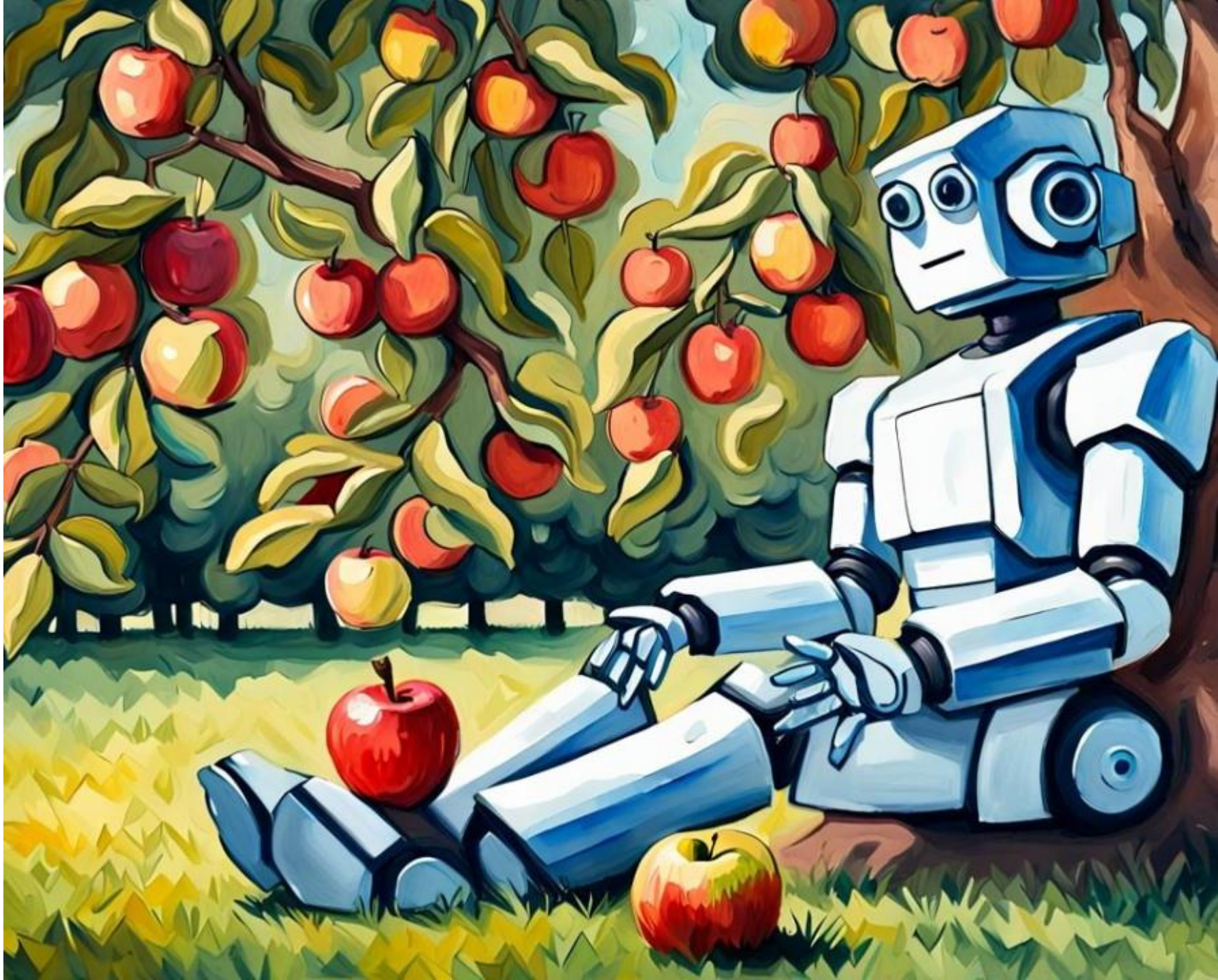
Effects:

- Increased latency when training large models.
- System crashes due to insufficient memory.

Solutions:

- Use distributed storage systems such as HDFS or AWS S3.
- Implement data compression techniques to reduce storage load.
- Use techniques like "lazy loading" where data is fetched only when needed.

Software Optimization and Integration



Software Frameworks

Root Cause:

- Many ML frameworks (TensorFlow, PyTorch) are not always optimized for specific hardware architectures.
- Poor software-hardware compatibility can lead to inefficient computation.

Effects:

- Suboptimal performance in model training and inference.
- Higher energy consumption and increased training costs.

Solutions:

- Use hardware-optimized libraries such as cuDNN (for NVIDIA GPUs) or Intel MKL-DNN (for CPUs).
- Optimize computational graphs using compilers like XLA (for TensorFlow) or TorchScript (for PyTorch).

Deployment Challenges

Root Cause:

- ML models are often developed in research environments and need to be deployed in production systems.
- Differences in infrastructure (development vs. production) lead to compatibility issues.

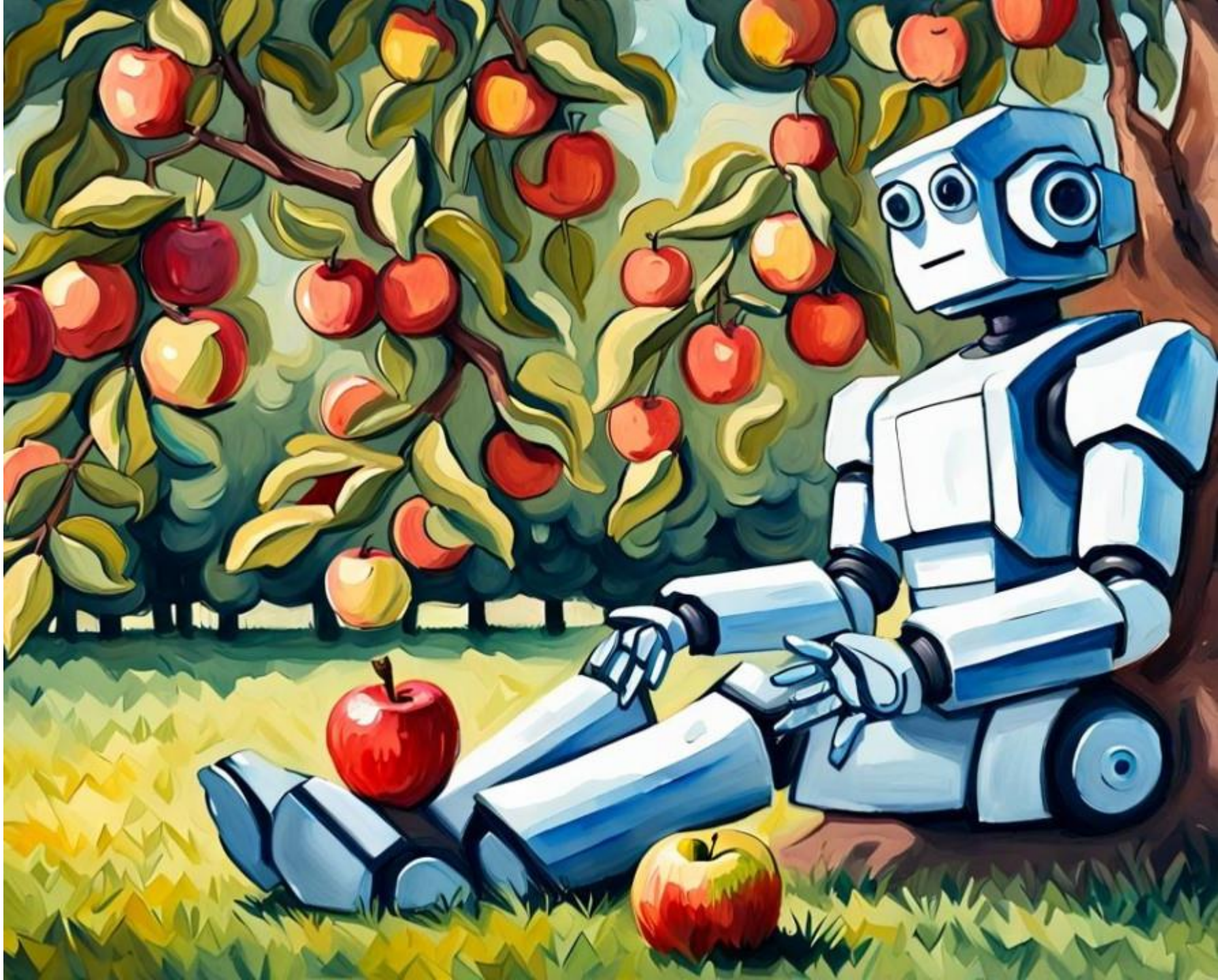
Effects:

- Models require extensive reengineering before deployment.
- Increased time-to-market for AI applications.

Solutions:

- Use containerization (Docker, Kubernetes) to maintain consistency between development and production.
- Implement Model-as-a-Service (MaaS) frameworks to streamline deployment.

Data Preprocessing



CPU Preprocessing

Root Cause:

- Data preprocessing is often handled by CPUs, which are slower than GPUs.
- Tasks such as normalization, augmentation, and feature extraction can be computationally expensive.

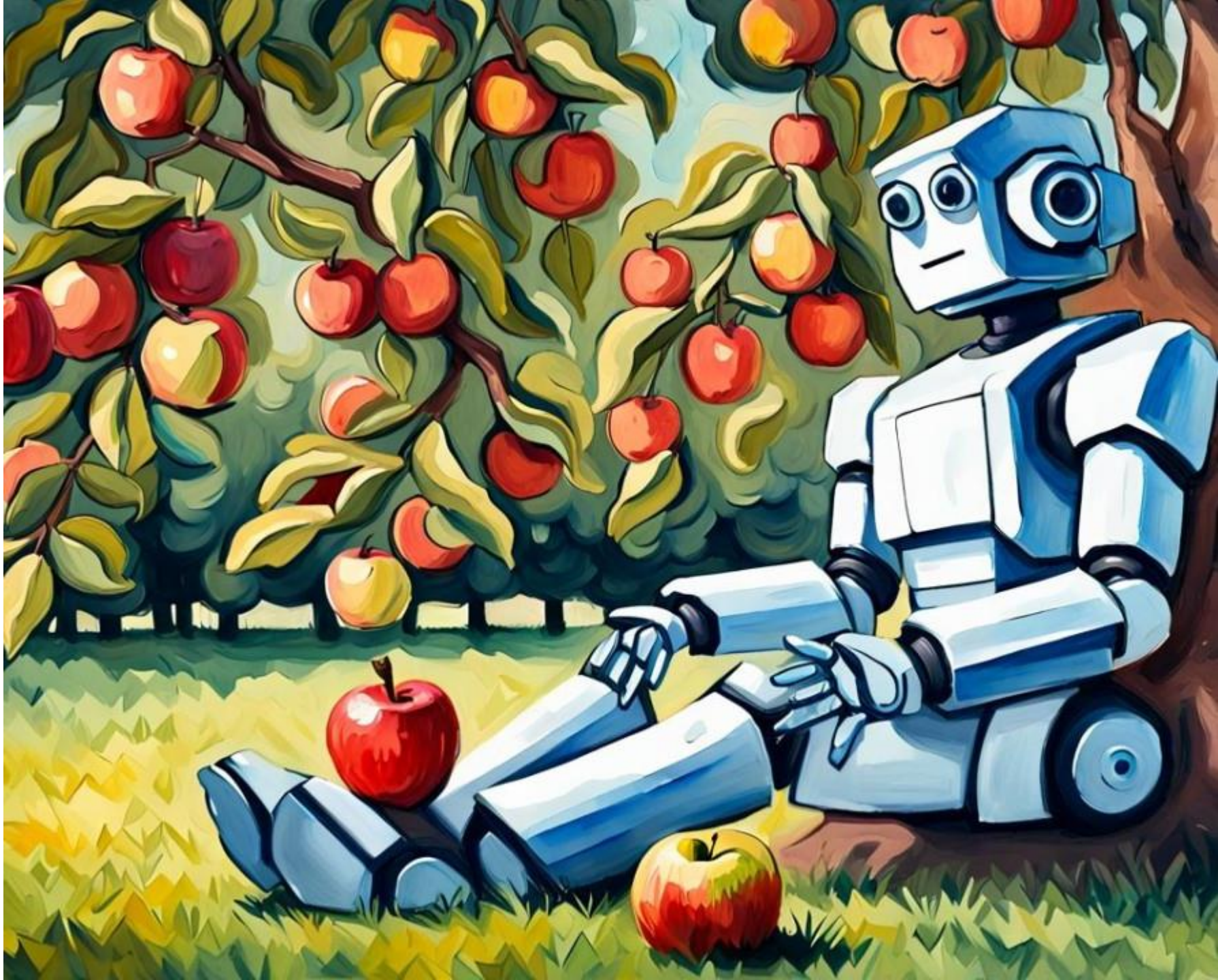
Effects:

- Slower data pipeline, delaying model training.
- Increased overall runtime of ML workflows.

Solutions:

- Offload preprocessing tasks to GPUs using frameworks like NVIDIA DALI.
- Use parallel processing techniques (e.g., multiprocessing in Python) to speed up preprocessing.

What to do



Use GPU Acceleration

- Move computations to GPU

```
device = torch.device(
      "cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
data = data.to(device)
```

- Use multiple GPUs

next lecture

- Enable Mixe r Computation

prevent gradients with small magnitudes from flushing to zero

run in mixed precision

```
scaler = torch.amp.GradScaler("cuda" ,ena
with torch.autocast(device_type=device,
                    dtype=torch.float16, enabled=use_amp):
    output = model(input)
    loss = loss_fn(output, target)
scaler.scale(loss).backward()
scaler.step(optimizer)
scaler.update()
```

Use GPU Acceleration

- Enable FP16 Mixed Precision

```
# Converts model weights to half-precision
model = model.half()
```

Tensor Processing Units

```
pip install torch_xla
```

```
import torch_xla.core.xla_model as xm
import torch_xla.debug.metrics as met

class SimpleModel(nn.Module):
…

device = xm.xla_device()
model = SimpleModel().to(device)

data = dataCPU.to(device)
labels = labelsCPU.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(10):
    model.train()
    optimizer.zero_grad()
    outputs = model(data)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    xm.mark_step()
```

cuDNN

cuDNN (CUDA Deep Neural Network library) is NVIDIA's optimized library for deep learning operations, particularly convolutions. Enabling **cuDNN benchmarking** allows PyTorch or TensorFlow to find the best convolution algorithm for your specific hardware, improving training and inference speed.

```
import torch
torch.backends.cudnn.benchmark = True
```

How It Works:

1. Runs a short test on different convolution algorithms.
2. Selects the fastest one for your specific GPU and input size.
3. Caches the best algorithm to improve performance.

JIT

```
def f1(x):
    return 2.0*x*x
```

0.084 ms

```
def f2(x):
    return 1/8 * (35 * x**4 - 30 * x*2 +3)
```

0.404 ms

JIT

```
@torch.jit.script
def f1(x):
    return 2.0*x*x
```

0.010 ms 

```
@torch.jit.script
def f2(x):
    return 1/8 * (35 * x**4 - 30 * x*2 +3)
```

0.0120 ms 

JIT

```
@torch.compile
def f1(x):
    return 2.0*x*x
```

0.015 ms 

```
@torch.compile
def f2(x):
    return 1/8 * (35 * x**4 - 30 * x*2 +3)
```

0.015 ms 

Optimize Data Loading

- Use DataLoader with Multiple Workers

```
train_loader = DataLoader(dataset,
       batch_size=64,
       num_workers=4)
```

- Enable Asynchronous Data Transfers

```
data = data.to(device, non_blocking=True)
```

- Prefetch Data

```
train_loader = DataLoader(dataset,
       batch_size=64,
       prefetch_factor=2)
```

- Use Pinned Memory

```
train_loader = DataLoader(dataset,
       batch_size=64,
       pin_memory=True)
```

Prefetch

Prefetching is a **data loading optimization** where the **next batch of data is loaded while the GPU is still processing the current batch.** This prevents the GPU from sitting idle while waiting for data to be fetched from RAM or disk.

How Prefetching Works in PyTorch?

1. **Parallel Data Loading**: The DataLoader uses multiple worker processes (num_workers > 0) to load batches in parallel.
2. **Prefetching**: The prefetch_factor determines how many extra batches each worker loads before they are needed.
3. **Reduced Latency**: When the GPU finishes processing a batch, the next batch is already ready in RAM, avoiding delays.

You will use significantly more CPU RAM.

Pinned Memory

Pinned memory (also called page-locked memory) is a special type of RAM (system memory) that **cannot be moved or swapped out to disk** by the operating system. This allows **faster data transfer between CPU and GPU** because:

- The GPU can **directly access** pinned memory via **DMA (Direct Memory Access)**.
- The CPU doesn't need to copy data to a temporary buffer before sending it to the GPU.

Data transfer at a Low Level

Normal Memory Transfer

1. Tensor stored in pageable memory (RAM).
2. CPU copies it into a staging buffer.
3. Data is transferred to GPU (VRAM) via PCIe.
4. GPU accesses the data for computation.

Transfer with Pinned Memory

1. Tensor is directly allocated in pinned memory (page-locked RAM)
2. GPU directly accesses it via DMA without needing a CPU copy.
3. Transfer completes faster, reducing CPU load.

Efficient Data Augmentation

- Apply on-the-fly augmentation rather than precomputing it and storing it on disk.
- Use libraries:
  - **torchvision** is a part of the PyTorch ecosystem, and its transforms module provides a set of image transformations.
  - **Albumentations** is a very powerful and flexible library for fast image augmentation. It's widely used in computer vision tasks and supports a variety of augmentation techniques.
  - **imgaug** is powerful library that offers a wide range of augmentations, and it supports both image and keypoint augmentations (for object detection).
  - **torchio** is specifically designed for medical image augmentation, supporting **3D images** (like CT/MRI scans).
  - The **augmenter** library is a lightweight, fast tool for augmenting images and comes with a simple API.
- Use parallel processing (e.g., multi-threaded data augmentation) to speed up training.

Optimize Training Configuration

- Use Larger Batch Sizes

```
train_loader = DataLoader(dataset,
        batch_size=64)
```

Batch Size	Use Case	Pros	Cons
Small (e.g., 8-32)	Faster training for smaller datasets	Faster convergence (less overfitting)	Slower per epoch, noisier gradients
Medium (e.g., 32-64)	Standard use case for general datasets	Balanced training time & performance	May still cause OOM on large models/datasets
Large (e.g., 128-256)	For very large datasets, GPUs with higher memory	Faster per epoch, better throughput	Potential for less generalization, overfitting

- **Larger Batch Size = Faster Training per Epoch**: With larger batches, the model can compute more samples in parallel, which speeds up the forward and backward passes.
- **Smaller Batch Size = More Frequent Updates**: Smaller batches provide noisier gradients, which can help escape local minima and might lead to a better final model, but training is slower since there are more updates per epoch.

Optimize Training Configuration

- Check memiry usage

```
# Current memory usage
print(torch.cuda.memory_allocated())
# Total memory reserved
print(torch.cuda.memory_reserved())
```

Optimize Training Configuration

- Check memiry usage

```
# Current memory usage
print(torch.cuda.memory_allocated())
# Total memory reserved
print(torch.cuda.memory_reserved())
```

- Different optimizers react differently to batch size:
  - **Adam** works well with smaller batches and can converge faster even with smaller batch sizes.
  - **SGD** may benefit more from larger batch sizes (but requires careful tuning of the learning rate).

Optimize Training Configuration

not just technical stuff

- Use `lr_scheduler` for Learning Rate Scheduling
  Using **learning rate warm-up** and **decay** helps stabilize training and speeds up convergence.

```
scheduler = torch.optim.lr_scheduler.OneCycleLR(
        optimizer, max_lr=0.01
        steps_per_epoch=len(dataloader),
        epochs=10)
```

Key Concepts of Learning Rate Scheduling:

- **Learning Rate Decay**:Decrease the learning rate over time to allow the model to converge slowly and avoid overshooting optimal weights.
- **Warm-up**:Gradually increase the learning rate at the beginning of training to avoid large updates that might destabilize training.
- **Cyclical Learning Rate (CLR)**:Cyclically adjust the learning rate to oscillate between a lower and higher value, helping to escape local minima.
- **Step Decay**:Reduce the learning rate by a fixed factor at regular intervals.

Optimize Training Configuration

not just technical stuff

- Use the Right Optimizer
  - **AdamW** is often faster than regular **Adam** because of its weight decay implementation
  - For **large datasets** or simpler tasks, **SGD** with momentum may work better

Optimize Training Configuration

not just technical stuff

Choosing the Right Optimizer:

- **Stochastic Gradient Descent (SGD)**: Good for simple models or when you want control over learning rates, momentum, etc.
- **Adaptive Moment Estimation / Adam with Weight Decay (Adam/AdamW)**: Good default for most problems; generally faster convergence and works well with sparse data.
- **Root Mean Square Propagation (RMSprop)**: Often used for recurrent neural networks (RNNs).
- **Adaptive Gradient Algorithm (Adagrad)**: Best for sparse data (e.g., text).
- **Adadelta**: A more refined version of Adagrad with a constant learning rate.
- **Nesterov-accelerated Adaptive Moment Estimation (Nadam)**: A good choice when you want both adaptive learning rates and momentum.
- **LBFGS**: Suitable for small models or problems where you need higher accuracy and second-order methods.

Use smaller models

Model Pruning

Speed up inference

Pruning in machine learning refers to the **removal** of certain parts of the model (typically weights, neurons, or entire channels) that are **not contributing significantly** to the model's output. The goal is to:

- **Reduce the model size**: This is useful for deploying models on resource-constrained devices like mobile phones, edge devices, etc.
- **Speed up inference**: Fewer operations means faster inference times.
- **Maintain or improve generalization**: Pruning might reduce overfitting and improve generalization in some cases, although this isn't always guaranteed.

Torch ne

Speed up inference

The process of pruning with **TorchPrune** generally involves three steps:

1. **Model Setup**: You start with a trained PyTorch model that you want to prune.
2. **Pruning Strategy**: You specify the pruning method

prunes the smallest 30%

```
pruning_method =
      torch_prune.methods.PruningByMagnitude(sparsity=0.3)
```

3. **Apply Pruning**: The pruning method is applied to the model, and certain weights/neurons/channels are zeroed out.

```
pruned_model =
torch_prune.prune_model(model, pruning_method)
```

4. **Fine-Tuning**: After pruning, you often need to fine-tune the model to restore or even improve performance. This is crucial because removing parts of the model without fine-tuning can lead to a drop in performance.

```
for epoch in range(10):
   …
```

Quantization

Speed up inference

Quantization is a technique used in deep learning to reduce the memory footprint and computational requirements of models by converting high-precision floating-point numbers into lower-precision numbers (int8, float16, etc.). PyTorch provides built-in support for quantization, allowing models to run efficiently on CPUs and edge devices without significant loss of accuracy.

- Dynamic Quantization (Best for LSTMs & Transformers)

```
import torch.quantization
quantized_model = torch.quantization.quantize_dynamic(
model, {torch.nn.Linear}, dtype=torch.qint8)
```

- Static Quantization (Best for CNNs & Vision Models)

```
# Fuse layers for better quantization
model = torch.quantization.fuse_modules(model, [["conv", "relu"]])
# Define quantization configuration
model.qconfig = torch.quantization.get_default_qconfig("fbgemm")
# Prepare model for quantization
model_prepared = torch.quantization.prepare(model)
# Run a few inference steps for calibrationfor
data, _ in dataloader:
    model_prepared(data)
# Convert to a fully quantized modelquantized_
model = torch.quantization.convert(model_prepared)
```

Scikit learn

- **Use Efficient Algorithms:**
  - Choose faster algorithms like **LinearRegression**, **LogisticRegression**, or **Ridge**.
  - **Avoid** more complex models like RandomForestClassifier or **SVC** unless necessary.
  - For gradient boosting tasks, prefer **HistGradientBoostingClassifier** over GradientBoostingClassifier for **faster** training on large datasets.
- **Parallelization with joblib:**
  - Next lecture
- **Use Sparse Matrices:**
  - For sparse data, **usescipy.sparse** formats like **csr_matrix** to reduce memory usage and speed up computations.
- **Preprocessing and Feature Selection:**
  - **Use Dimensionality reduction** (e.g., PCA) to reduce the number of features.
  - **Scale or normalize** data to improve speed and **convergence** for certain algorithms.
- **Use Incremental Learning:**
  - Models like **SGDClassifier**, **LogisticRegression** (with the saga solver), and

Scikit learn

- **Scale or normalize** data to improve speed and **convergence** for certain algorithms.

- **Use Incremental Learning:**
  - Models like **SGDClassifier**, **LogisticRegression** (with the saga solver), and **MiniBatchKMeans** support online learning for faster training with large datasets.
- **Use warm_start Option:**
  - Some models support the `warm_start=True` parameter to reuse previous computations and speed up model updates.
- **Use Cython or Numba:**
  - Speed up custom operations by compiling Python code with Cython or Numba.
- **Tune Hyperparameters:**
  - Adjust hyperparameters like `max_depth`, `n_estimators`, and `learning_rate` to speed up training by balancing performance and speed.
- **Use Efficient Data Structures:**
  - Use **numpy.ndarray** or **pandas.DataFrame** for efficient data handling instead of Python lists.