PyTorch

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the difference between torch.Tensor and torch.nn.Parameter. When would you use each?

torch.Tensor is the fundamental data structure in PyTorch for storing multi-dimensional arrays, while **torch.nn.Parameter** is a specialized tensor that is automatically registered as a trainable parameter when assigned as a module attribute.

Key Differences:

- **Parameter Registration:** Parameters are automatically added to module.parameters() and included in optimizer updates
- **Gradient Tracking:** Parameters have requires_grad=True by default
- **State Dict:** Parameters are saved/loaded automatically with model checkpoints

Example Usage:

```
class CustomLayer(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(out_features, in_features))
        self.buffer = torch.zeros(out_features)  # Not trainable
        self.register_buffer('running_mean', torch.zeros(out_features))

    def forward(self, x):
        return F.linear(x, self.weight) + self.running_mean
```

Use Parameter for trainable weights, and regular Tensors (or registered buffers) for non-trainable state like batch norm statistics.

2. How does PyTorch's autograd system work internally? Explain the computation graph and backward pass mechanism.

PyTorch's autograd implements reverse-mode automatic differentiation using a dynamic computation graph (DAG) built during the forward pass.

Internal Mechanism:

- **Graph Construction:** Each tensor operation creates a Function node storing the operation and inputs
- **grad_fn:** Output tensors store references to their creating function via the grad_fn attribute
- **Backward Pass:** Calling .backward() traverses the graph in reverse topological order, applying the chain rule
- **Gradient Accumulation:** Gradients are accumulated in tensor.grad for leaf nodes

Example:

```
x = torch.tensor([2.0], requires_grad=True)
y = x ** 2
z = y * 3

print(z.grad_fn)  # MulBackward
print(y.grad_fn)  # PowBackward

z.backward()
```

```
print(x.grad)  # tensor([12.]) = dz/dx = 6x
```

Key Insight: The graph is dynamic and rebuilt on each forward pass, enabling flexible control flow. Use torch.no_grad() or detach() to prevent graph construction when gradients aren't needed.

3. What are the differences between view(), reshape(), and contiguous() in PyTorch? When does each fail or require memory copying?

view(), reshape(), and contiguous() are tensor manipulation methods with different memory layout requirements and guarantees.

view():

- Returns a tensor sharing the same underlying data
- **Requires:** The tensor must be contiguous in memory
- **Fails:** If the tensor is not contiguous (e.g., after transpose)
- Zero-copy operation

reshape():

- Returns a view if possible, otherwise copies data
- More flexible than view(), works on non-contiguous tensors
- May or may not share memory

contiguous():

- Returns a contiguous copy if needed, otherwise returns self
- Ensures memory layout matches the logical tensor order

Example:

```
x = torch.randn(3, 4)
y = x.transpose(0, 1)  # Non-contiguous

# y.view(12)  # RuntimeError: not contiguous
z = y.reshape(12)  # Works, may copy
w = y.contiguous().view(12)  # Explicit copy then view
```

Best Practice: Use reshape() for flexibility, view() when you need guaranteed memory sharing, and contiguous() when interfacing with operations requiring contiguous memory.

4. Explain torch.nn.Module's forward hooks and backward hooks. Provide a practical use case for each.

Forward and backward hooks allow you to intercept and modify tensors during the forward and backward passes without modifying the module's code.

Forward Hooks:

- **register_forward_hook(hook_fn):** Called after forward() completes
- Signature: hook_fn(module, input, output)
- **Use Cases:** Feature extraction, activation visualization, debugging

Backward Hooks:

- **register_full_backward_hook(hook_fn):** Called during backward pass
- Signature: hook_fn(module, grad_input, grad_output)
- **Use Cases:** Gradient clipping, gradient analysis, custom gradient modifications

Practical Example - Feature Extraction:

```
features = {}

def get_features(name):
    def hook(model, input, output):
        features[name] = output.detach()
```

```
    return hook

model.layer3.register_forward_hook(get_features('layer3'))
output = model(x)
intermediate = features['layer3']  # Access layer3 activations
```

Gradient Monitoring Example:

```
def grad_hook(module, grad_in, grad_out):
    print(f'Gradient norm: {grad_out[0].norm().item()}')

model.conv1.register_full_backward_hook(grad_hook)
```

5. How do you implement a custom autograd Function in PyTorch? What are the requirements for forward and backward methods?

Custom autograd Functions enable you to define operations with custom forward and backward logic, essential for non-differentiable operations or performance optimization.

Requirements:

- Inherit from **torch.autograd.Function**
- Implement **@staticmethod forward(ctx, *args)** and **@staticmethod backward(ctx, *grad_outputs)**
- Use **ctx.save_for_backward()** to store tensors needed for gradient computation
- Return gradients for all inputs that require_grad=True in backward

Example - Custom ReLU with Gradient Clipping:

```
class ClippedReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, threshold):
        ctx.save_for_backward(input)
        ctx.threshold = threshold
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        grad_input = grad_input.clamp(-ctx.threshold, ctx.threshold)
        return grad_input, None

clipped_relu = ClippedReLU.apply
```

Critical: Return None for non-tensor arguments and tensors that don't require gradients. Test with torch.autograd.gradcheck() to verify correctness.

6. What is the difference between DataParallel and DistributedDataParallel? When should you use each for multi-GPU training?

DataParallel (DP) and **DistributedDataParallel (DDP)** are PyTorch's multi-GPU training strategies with significant architectural differences.

DataParallel:

- Single-process, multi-threaded approach
- Replicates model on each GPU, scatters input, gathers output on GPU 0
- GPU 0 becomes a bottleneck (gradient aggregation)
- Simpler to use but slower and less efficient
- **Use when:** Quick prototyping, single-node only

DistributedDataParallel:

- Multi-process approach (one process per GPU)

- All-reduce gradients across GPUs using NCCL backend
- Better GPU utilization, scales to multi-node
- Requires process group initialization
- **Use when:** Production training, multi-node, or >2 GPUs

DDP Setup Example:

```
import torch.distributed as dist

dist.init_process_group(backend='nccl')
local_rank = int(os.environ['LOCAL_RANK'])
torch.cuda.set_device(local_rank)

model = MyModel().to(local_rank)
model = DDP(model, device_ids=[local_rank])

# Launch: torchrun --nproc_per_node=4 train.py
```

Performance: DDP is typically 2-3x faster than DP due to better parallelization and reduced communication overhead.

7. Explain PyTorch's memory management and the common causes of CUDA out of memory errors. How do you debug and optimize memory usage?

PyTorch memory management uses a caching allocator for CUDA tensors to minimize expensive cudaMalloc/cudaFree calls.

Common OOM Causes:

- **Large batch sizes:** Memory scales linearly with batch size
- **Gradient accumulation:** Computation graphs retained until backward()
- **Hidden references:** Tensors kept in lists or global variables
- **Large intermediate activations:** Especially in deep networks

Debugging Techniques:

```
# Monitor memory usage
print(torch.cuda.memory_allocated() / 1024**3)  # GB
print(torch.cuda.memory_reserved() / 1024**3)
print(torch.cuda.memory_summary())

# Find memory leaks
import gc
gc.collect()
torch.cuda.empty_cache()

# Profile memory
from torch.profiler import profile, ProfilerActivity
with profile(activities=[ProfilerActivity.CUDA]) as prof:
    output = model(input)
```

Optimization Strategies:

- **Gradient checkpointing:** Trade compute for memory by recomputing activations
- **Mixed precision (AMP):** Use float16 to halve memory
- **Gradient accumulation:** Simulate larger batches with smaller steps
- **Del unused tensors:** Explicitly delete large intermediates
- **torch.no_grad():** Disable autograd for inference

8. What is gradient checkpointing and how does it work in PyTorch? When should you use it?

Gradient checkpointing is a memory optimization technique that trades computation for memory by selectively recomputing intermediate activations during the backward pass instead of storing them.

How It Works:

- During forward pass: Only checkpoint activations are saved, intermediates are discarded
- During backward pass: Forward pass is recomputed for segments between checkpoints
- **Memory savings:** O(n) reduced to O(√n) for n layers with optimal checkpointing
- **Compute cost:** Increases by ~20-30% due to recomputation

PyTorch Implementation:

```
from torch.utils.checkpoint import checkpoint

class CheckpointedModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList([Layer() for _ in range(100)])

    def forward(self, x):
        for layer in self.layers:
            x = checkpoint(layer, x, use_reentrant=False)
        return x
```

When to Use:

- **Very deep networks:** Transformers, ResNets with 100+ layers
- **Limited GPU memory:** When you can't fit the full model otherwise
- **Large batch sizes:** When activation memory dominates

Best Practice: Use use_reentrant=False for better compatibility with autograd. Checkpoint every few layers rather than every single layer for optimal compute/memory tradeoff.

9. How do you implement mixed precision training in PyTorch? Explain the role of GradScaler and autocast.

Mixed precision training uses float16 (half precision) for forward and backward passes while maintaining float32 master weights, reducing memory usage and increasing throughput on modern GPUs.

Key Components:

- **torch.cuda.amp.autocast:** Context manager that automatically casts operations to float16
- **GradScaler:** Scales loss to prevent gradient underflow in float16
- **Dynamic loss scaling:** Adjusts scale factor to maximize float16 range utilization

Implementation:

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for data, target in dataloader:
    optimizer.zero_grad()

    with autocast():
        output = model(data)
        loss = criterion(output, target)

    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

How GradScaler Works:

- **Scale:** Multiplies loss by scale factor before backward()
- **Unscale:** Divides gradients before optimizer.step()
- **Check:** Skips update if inf/NaN gradients detected
- **Update:** Adjusts scale factor based on recent overflow history

Benefits: 2-3x speedup on Tensor Cores (V100/A100), ~50% memory reduction. Some operations like softmax remain in float32 for numerical stability.

10. Explain the difference between torch.jit.script and torch.jit.trace. When would you use each for model optimization?

TorchScript provides two methods to convert PyTorch models to an optimized intermediate representation for production deployment: tracing and scripting.

torch.jit.trace:

- Records operations executed during a sample forward pass
- **Pros:** Simple, preserves exact computation for the traced path
- **Cons:** Cannot capture control flow (if/loops), only records one execution path
- **Use when:** Model has fixed architecture, no dynamic control flow

torch.jit.script:

- Analyzes Python code and compiles to TorchScript directly
- **Pros:** Preserves control flow, supports dynamic behavior
- **Cons:** Limited Python subset support, may require code modifications
- **Use when:** Model has conditionals, loops, or dynamic shapes

Examples:

```
# Tracing
model = MyModel()
example_input = torch.randn(1, 3, 224, 224)
traced = torch.jit.trace(model, example_input)

# Scripting
@torch.jit.script
def dynamic_model(x, threshold: float):
    if x.sum() > threshold:
        return x * 2
    return x
```

Hybrid Approach: Use torch.jit.script for modules with control flow and trace the outer model for best results.

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. How do you implement an LRU Cache using PyTorch tensors for storing numerical data with O(1) operations?

LRU Cache Implementation

While PyTorch tensors aren't ideal for LRU cache, you can combine Python's **OrderedDict** with tensors for data storage:

```
from collections import OrderedDict
import torch

class LRUCache:
    def __init__(self, capacity):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key):
        if key not in self.cache:
            return None
        self.cache.move_to_end(key)
        return self.cache[key]

    def put(self, key, tensor):
        if key in self.cache:
            self.cache.move_to_end(key)
        self.cache[key] = tensor
        if len(self.cache) > self.capacity:
            self.cache.popitem(last=False)
```

Time Complexity: O(1) for both get and put operations using OrderedDict's hash table and doubly-linked list.

2. Implement a function to find all pairs in a PyTorch tensor that sum to a target value. What's the optimal time complexity?

Two Sum Problem with PyTorch

Use a hash set approach for **O(n)** time complexity:

```
import torch

def find_pairs(tensor, target):
    seen = set()
    pairs = []
    for num in tensor.tolist():
        complement = target - num
        if complement in seen:
            pairs.append((complement, num))
        seen.add(num)
    return pairs
```

Key Points:

- Convert tensor to list for iteration
- Hash set lookup is O(1)
- Overall complexity: O(n) time, O(n) space
- For GPU tensors, transfer to CPU first to avoid overhead

3. How would you implement a sliding window maximum for a 1D PyTorch tensor efficiently?

Sliding Window Maximum

Use **torch.nn.functional.max_pool1d** for vectorized O(n) solution:

```
import torch
import torch.nn.functional as F

def sliding_window_max(tensor, window_size):
    tensor = tensor.unsqueeze(0).unsqueeze(0)
    result = F.max_pool1d(tensor, kernel_size=window_size,
                          stride=1, padding=0)
    return result.squeeze()
```

Alternative: For custom logic, use deque-based monotonic queue approach for O(n) time. PyTorch's max_pool1d is highly optimized and GPU-accelerated.

Time Complexity: O(n) with max pooling vs O(nk) naive approach.

4. Explain how to implement a priority queue for tensor operations. What data structure would you use?

Priority Queue for Tensors

Use Python's **heapq** module with tensor metadata:

```
import heapq
import torch

class TensorPriorityQueue:
    def __init__(self):
        self.heap = []
        self.counter = 0

    def push(self, priority, tensor):
        heapq.heappush(self.heap, (priority, self.counter, tensor))
        self.counter += 1

    def pop(self):
        return heapq.heappop(self.heap)[2]
```

Complexity:

- Push: O(log n)
- Pop: O(log n)
- Peek: O(1)

The counter ensures stable sorting when priorities are equal. Store tensor references, not copies, to avoid memory overhead.

5. How do you efficiently implement a circular buffer using PyTorch tensors for streaming data?

Circular Buffer with Tensors

Pre-allocate a fixed-size tensor and use **modulo indexing**:

```
import torch

class CircularBuffer:
    def __init__(self, capacity, shape):
        self.buffer = torch.zeros((capacity, *shape))
        self.capacity = capacity
        self.index = 0
        self.size = 0

    def append(self, tensor):
```

```
        self.buffer[self.index] = tensor
        self.index = (self.index + 1) % self.capacity
        self.size = min(self.size + 1, self.capacity)
```

Benefits:

- O(1) append operation
- No memory reallocation
- GPU-friendly for streaming applications
- Cache-efficient for sequential access

6. Implement a function to find the k-th largest element in a PyTorch tensor. Compare different approaches.

K-th Largest Element

PyTorch provides **torch.kthvalue** for O(n) average case:

```
import torch

def kth_largest(tensor, k):
    # Method 1: Using kthvalue (quickselect-based)
    return torch.kthvalue(tensor, tensor.numel() - k + 1).values

# Method 2: Using topk (heap-based)
def kth_largest_topk(tensor, k):
    return torch.topk(tensor, k).values[-1]
```

Comparison:

- **kthvalue:** O(n) average, $O(n^2)$ worst case (quickselect)
- **topk:** O(n log k) using min-heap
- **sort:** O(n log n), use only if you need sorted output
- For small k, topk is preferred; for k ≈ n/2, kthvalue is better

7. How would you implement a trie (prefix tree) for storing and searching tensor embeddings by string keys?

Trie for Tensor Embeddings

Implement a **TrieNode** structure with tensor storage:

```
import torch

class TrieNode:
    def __init__(self):
        self.children = {}
        self.embedding = None
        self.is_end = False

class EmbeddingTrie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word, embedding):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.embedding = embedding
        node.is_end = True
```

Use Cases: Vocabulary lookups, prefix matching for embeddings, autocomplete with semantic search. **Time Complexity:** O(m) for insert/search where m is word length.

8. Explain how to implement a disjoint set (Union-Find) data structure for graph operations on PyTorch tensors.

Union-Find with Path Compression

Implement using tensors for **GPU-accelerated graph algorithms**:

```
import torch

class UnionFind:
    def __init__(self, n, device='cpu'):
        self.parent = torch.arange(n, device=device)
        self.rank = torch.zeros(n, device=device)

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if self.rank[px] < self.rank[py]:
            self.parent[px] = py
        elif self.rank[px] > self.rank[py]:
            self.parent[py] = px
        else:
            self.parent[py] = px
            self.rank[px] += 1
```

Complexity: $O(\alpha(n))$ amortized per operation where α is inverse Ackermann function, nearly $O(1)$.

9. How do you implement a rolling hash for efficient substring matching in tensor-based text processing?

Rolling Hash Implementation

Use **Rabin-Karp algorithm** with PyTorch for vectorized operations:

```
import torch

def rolling_hash(text_tensor, pattern_len, base=256, mod=10**9+7):
    n = text_tensor.size(0)
    powers = torch.pow(base, torch.arange(pattern_len-1, -1, -1))
    powers = powers % mod

    hashes = torch.zeros(n - pattern_len + 1)
    for i in range(n - pattern_len + 1):
        window = text_tensor[i:i+pattern_len]
        hashes[i] = torch.sum(window * powers) % mod
    return hashes
```

Applications:

- Efficient pattern matching in sequences
- Duplicate detection in token streams
- Time Complexity: $O(n)$ for preprocessing, $O(1)$ for hash updates

10. Implement a segment tree for range queries on PyTorch tensors. What operations can be optimized?

Segment Tree for Range Queries

Build a **binary tree structure** using tensor indexing:

```
import torch

class SegmentTree:
    def __init__(self, arr):
        n = arr.size(0)
        self.n = n
        self.tree = torch.zeros(4 * n)
        self.build(arr, 0, 0, n - 1)
```

```
def build(self, arr, node, start, end):
    if start == end:
        self.tree[node] = arr[start]
    else:
        mid = (start + end) // 2
        self.build(arr, 2*node+1, start, mid)
        self.build(arr, 2*node+2, mid+1, end)
        self.tree[node] = self.tree[2*node+1] + self.tree[2*node+2]
```

Optimized Operations:

- Range sum/min/max: O(log n)
- Point update: O(log n)
- Ideal for batch statistics in training pipelines

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. How would you design a distributed training system for large-scale PyTorch models across multiple GPU clusters?

Architecture Overview

A distributed PyTorch training system requires careful consideration of **data parallelism**, **model parallelism**, and **pipeline parallelism**.

Key Components

- **Orchestration Layer:** Use Kubernetes with GPU operators for resource management and scheduling
- **Communication Backend:** NCCL for GPU-to-GPU communication, Gloo for CPU operations
- **Storage:** Distributed file system (HDFS, S3) with data sharding and prefetching
- **Training Coordinator:** Master-worker architecture using torch.distributed

Implementation Strategy

```
import torch.distributed as dist

dist.init_process_group(
    backend='nccl',
    init_method='env://'
)
model = DistributedDataParallel(
    model.cuda(local_rank),
    device_ids=[local_rank]
)
```

Considerations

- **Gradient Synchronization:** All-reduce operations after backward pass
- **Fault Tolerance:** Checkpointing with elastic training (torch.distributed.elastic)
- **Load Balancing:** Dynamic batch sizing based on GPU memory
- **Monitoring:** TensorBoard integration with distributed metrics aggregation

2. Design a real-time model serving infrastructure for PyTorch models that handles 100K+ requests per second with sub-100ms latency.

System Architecture

A high-throughput, low-latency serving system requires **horizontal scaling**, **request batching**, and **efficient model optimization**.

Core Components

- **Load Balancer:** NGINX or HAProxy with health checks and sticky sessions
- **Model Servers:** TorchServe or custom FastAPI servers with async workers
- **Model Optimization:** TorchScript compilation, quantization, ONNX Runtime
- **Caching Layer:** Redis for frequent predictions and feature caching
- **Message Queue:** Kafka for async processing of non-critical requests

Optimization Techniques

```
model = torch.jit.script(model)
```

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

Scaling Strategy

- **Dynamic Batching:** Accumulate requests for 10-50ms, batch inference
- **GPU Utilization:** Multiple model replicas per GPU with CUDA streams
- **Auto-scaling:** Kubernetes HPA based on request queue depth
- **Monitoring:** Prometheus + Grafana for latency percentiles and throughput

3. How would you architect a PyTorch-based recommendation system that handles both batch and real-time inference for millions of users?

Hybrid Architecture

The system needs to balance **batch processing** for model updates with **real-time serving** for user requests.

System Components

- **Offline Training Pipeline:** Spark/Ray for data processing, PyTorch for model training
- **Feature Store:** Feast or custom solution with online/offline stores
- **Batch Inference:** Nightly jobs generating candidate recommendations
- **Real-time Ranker:** Lightweight PyTorch model for personalization
- **Serving Layer:** Two-tier approach (candidate retrieval + ranking)

Data Flow

```
class RecommenderModel(nn.Module):
    def __init__(self, n_users, n_items):
        super().__init__()
        self.user_emb = nn.Embedding(n_users, 128)
        self.item_emb = nn.Embedding(n_items, 128)

    def forward(self, user_ids, item_ids):
        return (self.user_emb(user_ids) *
                self.item_emb(item_ids)).sum(1)
```

Scalability Considerations

- **Candidate Generation:** ANN search using FAISS for top-K retrieval
- **Model Updates:** Blue-green deployment with A/B testing framework
- **Caching:** Multi-level cache (user embeddings, top recommendations)
- **Feedback Loop:** Real-time feature updates via streaming pipeline

4. Design a fault-tolerant PyTorch training pipeline that can recover from node failures without losing significant progress.

Fault Tolerance Strategy

Implement **elastic training** with automatic checkpointing and state recovery mechanisms.

Key Components

- **Checkpoint Manager:** Periodic state snapshots to distributed storage
- **Elastic Agent:** torch.distributed.elastic for dynamic worker management
- **State Synchronization:** Consistent checkpoint versioning across workers
- **Health Monitoring:** Heartbeat mechanism with automatic node replacement

Implementation

```
from torch.distributed.elastic.multiprocessing.errors import record
```

```
@record
def train():
    checkpoint = load_checkpoint()
    model.load_state_dict(checkpoint['model'])
    optimizer.load_state_dict(checkpoint['opt'])

    for epoch in range(start_epoch, max_epochs):
        train_epoch()
        save_checkpoint(epoch)
```

Recovery Mechanisms

- **Checkpoint Strategy:** Save every N iterations + best model based on validation
- **Metadata Tracking:** Store epoch, global step, RNG state, optimizer state
- **Graceful Degradation:** Continue training with fewer workers if nodes fail
- **Rollback:** Automatic revert to last stable checkpoint on corruption
- **Monitoring:** Alert on repeated failures, track recovery time metrics

5. How would you design a multi-tenant PyTorch model training platform that isolates resources and ensures fair scheduling?

Platform Architecture

A multi-tenant platform requires **resource isolation**, **quota management**, and **fair scheduling** across teams.

Core Components

- **Resource Manager:** Kubernetes with GPU quotas and namespace isolation
- **Job Scheduler:** Custom scheduler or Kubeflow with priority queues
- **Storage Isolation:** Per-tenant S3 buckets or NFS mounts with access controls
- **Monitoring:** Per-tenant dashboards with resource usage tracking
- **API Gateway:** Authentication, rate limiting, and request routing

Scheduling Policy

```
class FairShareScheduler:
    def schedule(self, jobs, resources):
        shares = self.calculate_shares(jobs)
        for job in sorted(jobs, key=lambda j:
                         j.priority * shares[j.tenant]):
            if self.can_allocate(job, resources):
                self.allocate(job)
                resources -= job.required
```

Design Considerations

- **Resource Quotas:** GPU hours, storage, memory limits per tenant
- **Priority Tiers:** Production > development > experimental workloads
- **Preemption:** Lower priority jobs can be paused for urgent requests
- **Cost Tracking:** Chargeback system based on actual resource consumption
- **Security:** Network policies, secrets management, model encryption

6. Design a PyTorch-based computer vision pipeline for real-time video processing that handles multiple concurrent video streams.

Pipeline Architecture

Real-time video processing requires **stream multiplexing**, **frame batching**, and **GPU optimization**.

System Components

- **Ingestion Layer:** FFmpeg for video decoding, frame extraction
- **Frame Buffer:** Circular buffer with configurable size per stream
- **Batch Processor:** Aggregate frames from multiple streams for GPU efficiency
- **Model Server:** Optimized PyTorch model with TorchScript/TensorRT

- **Output Handler:** WebRTC or RTMP for streaming results

Processing Pipeline

```
class VideoProcessor:
    def __init__(self, model, batch_size=32):
        self.model = torch.jit.script(model).cuda()
        self.buffer = FrameBuffer(batch_size)

    async def process_stream(self, stream_id):
        while frame := await self.get_frame(stream_id):
            self.buffer.add(frame, stream_id)
            if self.buffer.ready():
                await self.batch_inference()
```

Optimization Strategies

- **Frame Skipping:** Process every Nth frame based on motion detection
- **Multi-resolution:** Different models for different stream priorities
- **GPU Sharing:** CUDA streams for concurrent model execution
- **Adaptive Batching:** Dynamic batch size based on latency requirements
- **Caching:** Temporal caching for static scene portions

7. How would you design a PyTorch model versioning and experiment tracking system for a large ML team?

System Design

A comprehensive MLOps platform needs **version control**, **reproducibility**, and **collaboration features**.

Core Components

- **Model Registry:** Centralized repository with metadata (MLflow, custom DB)
- **Experiment Tracker:** Log hyperparameters, metrics, artifacts
- **Artifact Store:** S3/GCS for model weights, datasets, checkpoints
- **Lineage Tracking:** Data provenance and model ancestry graphs
- **Collaboration Tools:** Model comparison, annotation, approval workflows

Version Control Schema

```
class ModelVersion:
    def __init__(self, name, version):
        self.id = f"{name}:v{version}"
        self.metadata = {
            'architecture': model.__class__.__name__,
            'params': count_parameters(model),
            'training_config': config,
            'metrics': validation_metrics,
            'git_commit': get_git_hash()
        }
```

Key Features

- **Reproducibility:** Store code snapshot, dependencies, random seeds
- **Model Promotion:** Staging → production workflow with approval gates
- **A/B Testing:** Traffic splitting between model versions
- **Rollback:** Quick revert to previous versions on performance degradation
- **Governance:** Access controls, audit logs, compliance tracking

8. Design a PyTorch-based NLP system for processing and analyzing millions of documents daily with near real-time updates.

System Architecture

A scalable NLP pipeline requires **distributed processing**, **incremental updates**, and **efficient indexing**.

Pipeline Stages

- **Ingestion:** Kafka for document streaming with partitioning by source
- **Preprocessing:** Distributed workers for tokenization, cleaning
- **Model Inference:** Batch processing with PyTorch on GPU clusters
- **Embedding Store:** Vector database (Pinecone, Milvus) for similarity search
- **Index Update:** Incremental updates to Elasticsearch for search

Processing Implementation

```
class DocumentProcessor:
    def __init__(self, model, batch_size=64):
        self.model = model.cuda()
        self.tokenizer = AutoTokenizer.from_pretrained('bert-base')

    def process_batch(self, docs):
        inputs = self.tokenizer(docs, padding=True, return_tensors='pt')
        with torch.no_grad():
            embeddings = self.model(**inputs.to('cuda'))
```

Scalability Considerations

- **Horizontal Scaling:** Stateless workers with auto-scaling based on queue depth
- **Model Optimization:** Distillation, quantization for faster inference
- **Caching:** Document embeddings cached with TTL for frequent access
- **Incremental Processing:** Only process new/updated documents
- **Monitoring:** Track processing lag, throughput, error rates per document type

9. How would you architect a PyTorch-based anomaly detection system that learns from streaming data and adapts to concept drift?

Adaptive Learning Architecture

The system must balance **online learning**, **drift detection**, and **model stability**.

System Components

- **Stream Processor:** Apache Flink/Kafka Streams for real-time data ingestion
- **Feature Extractor:** Sliding window aggregations and temporal features
- **Anomaly Detector:** Autoencoder or isolation forest in PyTorch
- **Drift Detector:** Statistical tests (KS test, PSI) on feature distributions
- **Model Updater:** Incremental training with replay buffer

Online Learning Implementation

```
class OnlineAnomalyDetector:
    def __init__(self, model, buffer_size=10000):
        self.model = model
        self.buffer = ReplayBuffer(buffer_size)
        self.optimizer = torch.optim.Adam(model.parameters())

    def update(self, batch):
        loss = self.model.compute_loss(batch)
        loss.backward()
        self.optimizer.step()
```

Drift Handling Strategy

- **Detection:** Monitor reconstruction error distribution shifts
- **Adaptation:** Trigger retraining when drift score exceeds threshold
- **Ensemble:** Maintain multiple models trained on different time windows
- **Validation:** Shadow mode for new models before production deployment
- **Feedback Loop:** Human-in-the-loop for labeling edge cases

10. Design a PyTorch model compression and optimization pipeline that reduces model size by 10x while maintaining 95%+ accuracy.

Compression Pipeline

A comprehensive approach combines **pruning**, **quantization**, **knowledge distillation**, and **architecture search**.

Multi-Stage Optimization

- **Stage 1 - Pruning:** Structured/unstructured pruning to remove redundant weights
- **Stage 2 - Quantization:** INT8 quantization-aware training
- **Stage 3 - Distillation:** Train smaller student model from teacher
- **Stage 4 - Compilation:** TorchScript optimization and operator fusion

Implementation Example

```
import torch.nn.utils.prune as prune

# Prune 40% of weights
prune.l1_unstructured(model.conv1, 'weight', amount=0.4)

# Quantization-aware training
model.qconfig = torch.quantization.get_default_qat_qconfig('fbgemm')
model_prepared = torch.quantization.prepare_qat(model)
model_quantized = torch.quantization.convert(model_prepared)
```

Validation Strategy

- **Accuracy Monitoring:** Track metrics at each compression stage
- **Latency Benchmarking:** Measure inference time on target hardware
- **Sensitivity Analysis:** Identify which layers tolerate more compression
- **Hardware-Aware:** Optimize for specific deployment targets (mobile, edge)
- **Iterative Refinement:** Fine-tune after each compression stage

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. How do you implement a custom autograd function in PyTorch with proper backward pass?

Custom Autograd Function

To create a custom autograd function, subclass **torch.autograd.Function** and implement **forward()** and **backward()** methods:

```
class MyReLU(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

Key points:

- Use **ctx.save_for_backward()** to store tensors needed for gradient computation
- The backward method receives gradient w.r.t. output and returns gradient w.r.t. input
- Number of outputs in backward must match number of inputs in forward

2. Debug this PyTorch code that causes a CUDA out of memory error during training:

Common CUDA OOM Issues

Several issues can cause OOM errors:

```
# Problem: Accumulating gradients in graph
for data, target in loader:
    output = model(data)
    loss = criterion(output, target)
    # Missing: loss = loss.item() or loss.detach()
    total_loss += loss  # Keeps computation graph!

# Solution:
total_loss += loss.item()  # Extract scalar value
```

Debugging strategies:

- Use **torch.cuda.empty_cache()** to free unused cached memory
- Reduce batch size or use gradient accumulation
- Use **torch.cuda.memory_summary()** to track allocations
- Detach tensors when storing for logging: **loss.detach().cpu()**
- Enable **torch.backends.cudnn.benchmark = False** for deterministic memory usage

3. How do you profile memory usage and identify memory leaks in PyTorch models?

Memory Profiling Techniques

PyTorch provides several tools for memory profiling:

```
import torch.profiler as profiler
```

```
with profiler.profile(
    activities=[profiler.ProfilerActivity.CPU,
               profiler.ProfilerActivity.CUDA],
    profile_memory=True,
    record_shapes=True
) as prof:
    model(input_tensor)

print(prof.key_averages().table(
    sort_by="cuda_memory_usage", row_limit=10))
```

Additional tools:

- **torch.cuda.memory_allocated()** and **torch.cuda.max_memory_allocated()** for current/peak usage
- **torch.cuda.memory_snapshot()** for detailed allocation history
- Use **torch.no_grad()** context for inference to prevent gradient tracking
- Check for circular references in custom classes holding tensors

4. Implement gradient clipping in PyTorch to prevent exploding gradients. What are the different methods?

Gradient Clipping Methods

PyTorch provides two main gradient clipping approaches:

```
# Method 1: Clip by norm
torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm=1.0)

# Method 2: Clip by value
torch.nn.utils.clip_grad_value_(
    model.parameters(), clip_value=0.5)

# Usage in training loop:
optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
```

Key differences:

- **clip_grad_norm_**: Scales gradients so total norm doesn't exceed threshold (preserves direction)
- **clip_grad_value_**: Clips each gradient element independently to [-clip_value, clip_value]
- clip_grad_norm_ is generally preferred for RNNs and transformers

5. How do you implement mixed precision training in PyTorch? What are the benefits and pitfalls?

Mixed Precision Training

Use **torch.cuda.amp** for automatic mixed precision:

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for data, target in loader:
    optimizer.zero_grad()
    with autocast():
        output = model(data)
        loss = criterion(output, target)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
```

Benefits:

- Reduces memory usage by ~50% (FP16 vs FP32)
- Faster computation on modern GPUs (Tensor Cores)
- GradScaler prevents underflow by scaling loss

Pitfalls:

- Some operations don't benefit from FP16 (e.g., reductions, normalizations)
- May require loss scaling tuning for stability
- Not all operations support FP16

6. Explain and fix this code that causes incorrect gradients due to in-place operations:

In-place Operations and Autograd

In-place operations can corrupt gradients in PyTorch:

```
# Problem: In-place modification
class BrokenModel(nn.Module):
    def forward(self, x):
        x = self.layer1(x)
        x += 1  # In-place operation!
        x = self.layer2(x)
        return x

# Solution: Use out-of-place operations
class FixedModel(nn.Module):
    def forward(self, x):
        x = self.layer1(x)
        x = x + 1  # Creates new tensor
        x = self.layer2(x)
        return x
```

Why it matters:

- In-place ops modify tensors that may be needed for backward pass
- Causes RuntimeError: one of the variables needed for gradient computation has been modified by an inplace operation
- Use **tensor.clone()** before in-place ops if necessary
- Methods ending with underscore (_) are typically in-place

7. How do you implement and use hooks in PyTorch for debugging activations and gradients?

PyTorch Hooks

Hooks allow inspection of intermediate values during forward/backward passes:

```
# Forward hook to inspect activations
activations = {}
def get_activation(name):
    def hook(model, input, output):
        activations[name] = output.detach()
    return hook

model.layer1.register_forward_hook(
    get_activation('layer1'))

# Backward hook for gradients
def grad_hook(grad):
    print(f"Gradient norm: {grad.norm()}")
    return grad

tensor.register_hook(grad_hook)
```

Use cases:

- Debugging vanishing/exploding gradients
- Extracting intermediate features for visualization
- Implementing gradient clipping per layer

- Remember to remove hooks with **handle.remove()** when done

8. What causes this error: 'Expected all tensors to be on the same device'? Provide a debugging strategy.

Device Mismatch Debugging

This error occurs when tensors are on different devices (CPU vs GPU, or different GPUs):

```
# Problem:
model = model.cuda()  # Model on GPU
data = torch.randn(32, 10)  # Data on CPU
output = model(data)  # Error!

# Solution 1: Move data to model's device
device = next(model.parameters()).device
data = data.to(device)

# Solution 2: Helper function
def to_device(data, device):
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)
```

Debugging strategy:

- Print tensor devices: **print(tensor.device)**
- Check model device: **next(model.parameters()).device**
- Use **torch.cuda.current_device()** to verify GPU
- Set default device: **torch.set_default_tensor_type('torch.cuda.FloatTensor')**

9. How do you implement custom exception handling for distributed training in PyTorch?

Distributed Training Exception Handling

Proper exception handling is critical in distributed settings:

```
import torch.distributed as dist

def train_with_error_handling(rank, world_size):
    try:
        dist.init_process_group(
            backend='nccl', rank=rank,
            world_size=world_size)
        # Training code here
    except Exception as e:
        print(f"Rank {rank} error: {e}")
        dist.destroy_process_group()
        raise
    finally:
        if dist.is_initialized():
            dist.destroy_process_group()
```

Best practices:

- Always call **dist.destroy_process_group()** in finally block
- Use **dist.barrier()** to synchronize before cleanup
- Log errors with rank information for debugging
- Handle NCCL timeouts with **TORCH_DISTRIBUTED_DEBUG=DETAIL**
- Implement checkpointing to recover from failures

10. Explain torch.jit.script vs torch.jit.trace. When would you use each, and what are common debugging issues?

TorchScript: Script vs Trace

torch.jit.trace records operations during execution:

```
# Tracing: records actual execution
```

traced = torch.jit.trace(model, example_input)

# Scripting: analyzes Python code
scripted = torch.jit.script(model)

Key differences:

- **trace**: Doesn't capture control flow (if/loops with data-dependent conditions)
- **script**: Analyzes source code, supports control flow but requires type annotations
- Use trace for simple models without dynamic control flow
- Use script for models with conditionals or dynamic loops

Common issues:

- Tracing with wrong input shape captures incorrect graph
- Script requires explicit type hints for function arguments
- Some Python features unsupported (e.g., list comprehensions with break)
- Debug with **print(traced.graph)** or **print(scripted.code)**

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you optimized a PyTorch model that was performing poorly in production.

Situation: Our image classification model was taking 300ms per inference in production, causing user experience issues and increased cloud costs.

Task: I needed to reduce inference time to under 100ms while maintaining accuracy within 2% of the original model.

Action: I implemented a three-step optimization:

- Applied model quantization using PyTorch's dynamic quantization, converting FP32 to INT8
- Used TorchScript to compile the model and eliminate Python overhead
- Implemented batch processing for concurrent requests and moved the model to ONNX Runtime

Result: Reduced inference time to 65ms (78% improvement), decreased memory usage by 60%, and maintained accuracy loss within 1.5%. This saved approximately $15K monthly in infrastructure costs.

2. Describe a situation where you had to debug a complex training issue in PyTorch, such as vanishing gradients or NaN losses.

Situation: During training of a deep ResNet-101 model for medical image analysis, losses suddenly became NaN after epoch 3, halting all progress.

Task: I was responsible for identifying the root cause and implementing a fix to continue training without data loss.

Action: I systematically debugged the issue:

- Added gradient clipping and registered hooks to monitor gradient magnitudes across layers
- Discovered exploding gradients in the final fully connected layers due to improper weight initialization
- Implemented gradient norm monitoring and reduced learning rate by 10x with warm-up scheduling
- Added mixed precision training with gradient scaling to stabilize training

Result: Successfully resumed training with stable loss curves, achieving convergence in 45 epochs with 94.2% validation accuracy, 3% better than the baseline.

3. Share an example of when you had to scale a PyTorch training pipeline from single GPU to distributed multi-GPU or multi-node setup.

Situation: Our NLP model training was taking 14 days on a single V100 GPU, delaying product iterations and experimentation cycles.

Task: I needed to scale the training to multiple GPUs and reduce training time to under 3 days while maintaining reproducibility.

Action: I implemented distributed training:

- Refactored code to use PyTorch's DistributedDataParallel (DDP) across 8 GPUs
- Implemented gradient accumulation to maintain effective batch size and learning rate scaling using linear scaling rule
- Added torch.distributed backend with NCCL for efficient GPU communication
- Set up synchronized batch normalization and proper random seed management for reproducibility

Result: Reduced training time from 14 days to 2.1 days (85% reduction), achieved near-linear scaling efficiency of 7.3x on 8 GPUs, and enabled the team to run 5x more experiments per week.

4. Tell me about a time when you had to make a trade-off between model complexity and inference speed in a PyTorch project.

Situation: We developed a state-of-the-art transformer model with 95% accuracy, but it required 500ms inference time, making it unsuitable for our real-time mobile application requiring <150ms latency.

Task: I needed to find the optimal balance between model performance and speed to meet product requirements.

Action: I conducted a systematic analysis:

- Profiled the model using PyTorch profiler to identify bottleneck layers
- Experimented with knowledge distillation, training a smaller student model from the large teacher
- Reduced transformer layers from 12 to 6 and hidden dimensions from 768 to 512
- Applied pruning to remove 30% of less important weights and quantized to INT8

Result: Achieved 120ms inference time with 91% accuracy (4% drop), which stakeholders accepted. The solution enabled mobile deployment, reaching 2M additional users and increasing engagement by 35%.

5. Describe a situation where you identified and fixed a memory leak or out-of-memory issue in PyTorch training.

Situation: Our training pipeline was crashing with CUDA out-of-memory errors after 2-3 hours, despite starting with only 60% GPU memory utilization on 32GB V100s.

Task: I was assigned to identify the memory leak and ensure stable 24+ hour training runs.

Action: I performed detailed memory profiling:

- Used torch.cuda.memory_summary() and nvidia-smi to track memory growth over time
- Discovered that validation metrics were accumulating tensors in lists without detaching from computation graph
- Fixed by calling .detach() and .cpu() on all tensors stored for logging
- Implemented gradient checkpointing for the largest layers and reduced batch size slightly
- Added explicit torch.cuda.empty_cache() calls after validation

Result: Eliminated memory leaks completely, enabling stable 48+ hour training runs. Memory usage stabilized at 85% throughout training, and we successfully trained larger models with 2x more parameters.

6. Tell me about a challenging situation where you had to implement a custom PyTorch layer or loss function for a unique business requirement.

Situation: Our fraud detection system needed to heavily penalize false negatives (missed fraud) while tolerating more false positives, but standard loss functions didn't provide the asymmetric weighting we required.

Task: I needed to design and implement a custom loss function that aligned with business priorities and integrated seamlessly with our existing PyTorch pipeline.

Action: I developed a custom solution:

- Created a custom FocalAsymmetricLoss class inheriting from nn.Module with configurable penalty ratios
- Implemented backward pass with proper gradient computation for backpropagation
- Added hyperparameter tuning to find optimal false negative penalty multiplier (settled on 5x)
- Validated gradients using torch.autograd.gradcheck to ensure mathematical correctness

Result: Reduced false negative rate by 67% while false positive rate increased by only 15%. This prevented an estimated $2.3M in fraud losses annually, and the custom loss function was adopted across three other models.

7. Share an experience where you had to collaborate with cross-functional teams to

deploy a PyTorch model and resolve production issues.

Situation: After developing a recommendation model, the DevOps team reported that model serving was causing 30% CPU utilization spikes and occasional 5-second latencies during peak traffic.

Task: I needed to work with DevOps, backend engineers, and product managers to diagnose issues and implement a production-ready solution.

Action: I coordinated cross-functional efforts:

- Conducted joint debugging sessions with DevOps using profiling tools to identify inefficient preprocessing
- Worked with backend team to implement request batching and caching strategies
- Converted model to TorchScript and set up proper thread management with torch.set_num_threads()
- Established monitoring with Prometheus metrics for inference time, throughput, and error rates
- Created documentation and runbooks for on-call engineers

Result: Reduced CPU utilization to 12%, eliminated latency spikes, and improved mean inference time to 45ms. The collaborative approach strengthened team relationships and established best practices for future ML deployments.

8. Describe a time when you had to quickly learn and implement a new PyTorch feature or research paper to solve a critical problem.

Situation: Our object detection model was failing to identify small objects (less than 32x32 pixels), causing a 40% miss rate on critical use cases, and stakeholders needed a solution within 3 weeks.

Task: I needed to rapidly research, understand, and implement Feature Pyramid Networks (FPN) to improve small object detection.

Action: I executed an accelerated learning and implementation plan:

- Spent 2 days studying the FPN paper and existing implementations in torchvision
- Implemented custom FPN layers integrated with our existing ResNet backbone
- Created multi-scale training pipeline with proper anchor box configurations
- Conducted ablation studies to validate each component's contribution
- Optimized implementation to maintain inference time under 100ms

Result: Delivered the solution in 2.5 weeks, improving small object detection from 60% to 89% mAP. The implementation became the foundation for three subsequent projects, and I presented the approach to the broader engineering team.

9. Tell me about a situation where you had to handle imbalanced datasets or difficult training data in PyTorch.

Situation: Our medical diagnosis model was trained on a highly imbalanced dataset with 95% negative cases and 5% positive cases, resulting in a model that achieved 95% accuracy by simply predicting negative for everything.

Task: I needed to develop a training strategy that would make the model actually learn to detect positive cases with high recall.

Action: I implemented multiple techniques:

- Created a custom WeightedRandomSampler to oversample minority class during training
- Implemented focal loss to focus learning on hard examples
- Applied class weights inversely proportional to class frequencies in the loss function
- Used stratified K-fold validation to ensure proper evaluation
- Added data augmentation specifically for positive class samples using torchvision transforms

Result: Improved positive class recall from 8% to 87% while maintaining 82% precision. The model successfully identified 94% of actual medical conditions in validation, earning approval from medical advisors for clinical trials.

10. Share an example of when you mentored junior developers or conducted code reviews for PyTorch projects, and how you handled technical disagreements.

Situation: A junior engineer implemented a training pipeline using DataParallel instead of DistributedDataParallel, and when I suggested changing it during code review, they defended their choice citing simpler implementation.

Task: I needed to mentor them on best practices while respecting their perspective and maintaining a collaborative relationship.

Action: I took an educational approach:

- Scheduled a pairing session to demonstrate the performance differences between DataParallel and DDP
- Ran benchmarks together showing DDP was 2.3x faster on their specific use case
- Explained technical reasons: DataParallel's GIL limitations and single-process bottleneck
- Helped them refactor the code to DDP, explaining each step and why it matters
- Created internal documentation on distributed training best practices for the team

Result: The engineer became an advocate for DDP and later taught it to other team members. Training time improved by 58%, and they thanked me for the learning opportunity. This established a pattern of constructive technical mentorship across the team.