PyTorch Coding Challenges

Interview Questions and Answers

Core Concepts

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

1. Implement a custom autograd function in PyTorch that computes the logarithm of the sum of exponentials.

Here's an implementation of LogSumExp with custom gradients:

```
class LogSumExp(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        max_val = torch.max(x, dim=1, keepdim=True)[0]
        exp_x = torch.exp(x - max_val)
        sum_exp = torch.sum(exp_x, dim=1, keepdim=True)
        ctx.save_for_backward(exp_x, sum_exp)
        return max_val + torch.log(sum_exp)
```

2. Write a custom DataLoader that handles variable-length sequences with padding.

Implementation of a custom collate function for variable-length sequences:

```
def collate_fn(batch):
    sequences = [item[0] for item in batch]
    lengths = torch.tensor([len(seq) for seq in sequences])
    padded = pad_sequence(sequences, batch_first=True)
    labels = torch.tensor([item[1] for item in batch])
    return padded, lengths, labels
```

3. Implement a custom loss function for contrastive learning with temperature scaling.

Here's a contrastive loss implementation:

```
def contrastive_loss(features, labels, temperature=0.07):
    similarity = F.cosine_similarity(features.unsqueeze(1), features.unsqueeze(0), dim=2)
    mask = (labels.unsqueeze(1) == labels.unsqueeze(0)).float()
    logits = similarity / temperature
    return -torch.log(torch.exp(logits.diag()) / torch.exp(logits).sum(dim=1))
```

4. Create a custom Layer that implements Squeeze-and-Excitation attention mechanism.

Implementation of SE-Net block:

```
class SEBlock(nn.Module):
    def __init__(self, channels, reduction=16):
        super().__init__()
        self.squeeze = nn.AdaptiveAvgPool2d(1)
        self.excite = nn.Sequential(
            nn.Linear(channels, channels // reduction),
            nn.ReLU(),
            nn.Linear(channels // reduction, channels),
            nn.Sigmoid())
```

5. Implement gradient clipping with custom backward hook in PyTorch.

Example of implementing gradient clipping using hooks:

```
def clip_gradient_hook(module):
    for p in module.parameters():
        if p.grad is not None:
            p.grad.data.clamp_(-1, 1)
```

```
model.register_backward_hook(lambda m, grad_in, grad_out: clip_gradient_hook(m))
```

6. Write a custom Optimizer that implements AdaBelief algorithm.

Implementation of AdaBelief optimizer:

```
class AdaBelief(Optimizer):
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8):
        defaults = dict(lr=lr, betas=betas, eps=eps)
        super().__init__(params, defaults)
        self.belief = defaultdict(float)
        for group in self.param_groups:
            for p in group['params']:
                self.belief[p] = torch.zeros_like(p.data)
```

7. Implement a custom Scheduler that combines cosine annealing with warm restarts.

Implementation of Cosine Annealing with Warm Restarts:

```
class CosineAnnealingWarmRestarts(torch.optim.lr_scheduler._LRScheduler):
    def __init__(self, optimizer, T_0, T_mult=1, eta_min=0):
        self.T_0 = T_0
        self.T_mult = T_mult
        self.eta_min = eta_min
        super().__init__(optimizer)
        self.T_cur = 0
```

8. Create a custom Module that implements Multi-Head Self-Attention mechanism.

Implementation of Multi-Head Self-Attention:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.d_k = d_model // num_heads
        self.num_heads = num_heads
        self.query = nn.Linear(d_model, d_model)
        self.key = nn.Linear(d_model, d_model)
        self.value = nn.Linear(d_model, d_model)
```

9. Implement a custom activation function with learnable parameters.

Implementation of PReLU with learnable parameters:

```
class LearnableReLU(nn.Module):
    def __init__(self, num_parameters=1, init=0.25):
        super().__init__()
        self.alpha = nn.Parameter(torch.ones(num_parameters) * init)

    def forward(self, x):
        return torch.max(0, x) + self.alpha * torch.min(0, x)
```

10. Write a custom Dataset class for handling paired image-to-image translation data.

Implementation of paired image translation dataset:

```
class PairedImageDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_pairs = [(os.path.join(root_dir, 'A', f),
                             os.path.join(root_dir, 'B', f))
                            for f in os.listdir(os.path.join(root_dir, 'A'))]
```

Data Structures and Algorithms

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

1. How would you implement a custom memory-efficient stack for storing PyTorch tensors?

Implementation Approach:

A memory-efficient tensor stack can be implemented using a list-based approach with automatic memory management:

```
class TensorStack:
    def __init__(self):
        self.items = []
        self.max_size = 1000

    def push(self, tensor):
        if len(self.items) >= self.max_size:
            self.items[0].detach_()  # Free unused gradients
        self.items.append(tensor)
```

Key considerations:

- Use tensor.detach_() to remove gradient history
- Implement size limits to prevent memory overflow
- Consider using torch.cuda.empty_cache() for GPU memory

2. Explain how to implement an LRU cache for PyTorch model weights with a given capacity.

Solution:

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

    def get(self, key):
        if key in self.cache:
            self.cache.move_to_end(key)
            return self.cache[key]
        return None
```

Time Complexity:

- Get operation: O(1)
- Put operation: O(1)
- Space complexity: O(capacity)

The OrderedDict maintains insertion order and allows efficient key reordering.

3. How would you implement a sliding window operation on a PyTorch tensor efficiently?

Efficient Implementation:

```
def sliding_window(tensor, window_size, stride=1):
    shape = tensor.shape[:-1] + (tensor.shape[-1] - window_size + 1, window_size)
    strides = tensor.stride() + (tensor.stride()[-1],)
    return torch.as_strided(tensor, shape, strides)
```

Key Points:

- Uses as_strided for memory efficiency
- Avoids creating unnecessary copies
- Time complexity: O(1) for creation, O(n) for iteration
- Memory complexity: O(1) additional memory

4. Implement a function to find pairs of tensors that sum to a target value using a hash table approach.

Efficient Solution:

```
def find_tensor_pairs(tensor, target):
    seen = {}
    for i, val in enumerate(tensor):
        complement = target - val
        if complement in seen:
            return (seen[complement], i)
        seen[val.item()] = i
```

Complexity Analysis:

- Time complexity: O(n)
- Space complexity: O(n)
- Handles both CPU and GPU tensors
- Uses dictionary for O(1) lookups

5. How would you implement a custom priority queue for batch processing in PyTorch?

Implementation:

```
class TensorPriorityQueue:
    def __init__(self):
        self.queue = []
        heapq.heapify(self.queue)

    def push(self, priority, tensor):
        heapq.heappush(self.queue, (-priority, tensor.clone()))
```

Features:

- Uses heapq for efficient priority management
- Clones tensors to prevent reference issues
- O(log n) push and pop operations
- Suitable for dynamic batch prioritization

6. Implement an efficient method to find the k nearest neighbors for a given tensor using a heap.

Solution:

```
def k_nearest_neighbors(tensor, query, k):
    heap = []
    for i, point in enumerate(tensor):
        dist = torch.norm(point - query)
        heapq.heappush(heap, (-dist, i))
        if len(heap) > k:
            heapq.heappop(heap)
```

Analysis:

- Time complexity: O(n log k)
- Space complexity: O(k)
- Uses min-heap for efficient selection
- Handles both Euclidean and custom distance metrics

7. How would you implement a tensor cache with least-frequently-used (LFU) eviction policy?

Implementation:

```
class LFUTensorCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.freq = defaultdict(set)
        self.counts = defaultdict(int)
```

Key Components:

- Uses defaultdict for frequency tracking
- Maintains O(1) access time
- Efficiently handles frequency updates
- Automatically evicts least frequently used items

Time Complexity: O(1) for all operations

8. Implement an efficient method to find duplicate tensors in a large collection using hashing.

Solution:

```
def find_duplicate_tensors(tensors):
    seen = {}
    for i, tensor in enumerate(tensors):
        hash_key = hash(tensor.cpu().numpy().tobytes())
        if hash_key in seen:
            return (seen[hash_key], i)
        seen[hash_key] = i
```

Optimization Techniques:

- Uses numpy array bytes for hashing
- Handles both CPU and GPU tensors
- O(n) time complexity
- Memory-efficient for large collections

9. How would you implement a circular buffer for storing the last N tensor gradients?

Implementation:

```
class GradientBuffer:
    def __init__(self, size):
        self.size = size
        self.buffer = [None] * size
        self.current = 0
    def add(self, gradient):
        self.buffer[self.current] = gradient.detach().clone()
        self.current = (self.current + 1) % self.size
```

Features:

- O(1) insertion time
- Constant memory usage
- Automatic gradient history management
- Thread-safe implementation

10. Implement an efficient method to perform tensor deduplication while preserving order.

Solution:

```
def deduplicate_tensors(tensors):
    seen = set()
    result = []
    for tensor in tensors:
        key = tensor.cpu().numpy().tobytes()
        if key not in seen:
```

```
        seen.add(key)
        result.append(tensor)
```

Complexity Analysis:

- Time complexity: O(n)
- Space complexity: O(n)
- Preserves tensor order
- Handles both CPU and GPU tensors efficiently

System Design

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

1. Design a PyTorch-based distributed model training system that can handle millions of parameters across multiple GPUs. What architecture and components would you use?

Key Components:

- **DistributedDataParallel (DDP)** for model parallelism across GPUs
- **Parameter Server Architecture** for gradient synchronization
- **Data Sharding** for efficient data distribution

Implementation Approach:

```
def setup_distributed_training(model):
    model = DistributedDataParallel(model)
    sampler = DistributedSampler(dataset)
    optimizer = torch.optim.Adam(model.parameters())
    return model, sampler, optimizer
```

Considerations:

- Use NCCL backend for GPU-GPU communication
- Implement gradient accumulation for memory efficiency
- Add checkpoint mechanism for fault tolerance

2. How would you design a real-time PyTorch model serving system capable of handling 10,000 requests per second?

Architecture Components:

- **Model Server**: TorchServe with custom handlers
- **Load Balancer**: NGINX for request distribution
- **Caching Layer**: Redis for frequent predictions

```
class ModelServer:
    def __init__(self):
        self.model = torch.jit.load('model.pt')
        self.cache = Redis()
        self.batch_size = 32

    async def predict(self, inputs):
```

Optimizations:

- Model quantization for faster inference
- Batch prediction for throughput
- Dynamic batching based on load

3. Design a PyTorch-based automated model retraining pipeline that handles data drift and model degradation.

System Components:

- **Data Validation**: Great Expectations for data quality
- **Drift Detection**: Statistical tests on feature distributions
- **Training Pipeline**: Airflow DAGs for orchestration

```
def detect_drift(current_data, reference_data):
    drift_score = KSTest(current_data, reference_data)
    return drift_score > THRESHOLD
```

Pipeline Steps:

- Continuous data validation
- Automated drift detection
- A/B testing for new models
- Gradual model rollout

4. How would you implement a PyTorch model versioning and experiment tracking system?

Core Components:

- **Model Registry**: MLflow for versioning
- **Artifact Storage**: S3/Azure Blob
- **Metadata Store**: PostgreSQL

```
class ExperimentTracker:
    def log_model(self, model, metrics):
        mlflow.pytorch.log_model(model, 'models')
        mlflow.log_metrics(metrics)
        return mlflow.active_run().info.run_id
```

Features:

- Automatic metric logging
- Hyperparameter tracking
- Model lineage tracking
- Experiment comparison

5. Design a PyTorch-based online learning system that can adapt to new data in real-time.

Architecture:

- **Stream Processing**: Kafka for data ingestion
- **Model Updates**: Incremental learning
- **State Management**: Redis for model state

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

Key Features:

- Sliding window training
- Adaptive learning rates
- Concept drift detection
- Performance monitoring

6. Design a PyTorch model compression pipeline for edge deployment.

Compression Techniques:

- **Quantization**: INT8/FP16 conversion
- **Pruning**: Weight sparsification
- **Knowledge Distillation**: Teacher-student transfer

```
def compress_model(model):
    quantized = torch.quantization.quantize_dynamic(
        model, {torch.nn.Linear}, dtype=torch.qint8
    )
    return quantized
```

Pipeline Steps:

- Model analysis
- Structured pruning
- Quantization-aware training
- Performance validation

7. How would you design a PyTorch-based automated hyperparameter optimization system?

Components:

- **Search Strategies**: Bayesian optimization
- **Resource Management**: Ray Tune
- **Evaluation**: Cross-validation

```
def objective(config):
    model = create_model(config)
    trainer = pl.Trainer(max_epochs=10)
    trainer.fit(model)
    return {'loss': validation_loss}
```

Features:

- Early stopping
- Parallel trial execution
- Resource-aware scheduling
- Result persistence

8. Design a PyTorch feature store system for ML feature management and serving.

Architecture:

- **Online Store**: Redis for low-latency access
- **Offline Store**: Parquet files on S3
- **Feature Registry**: Feature definitions and metadata

```
class FeatureStore:
    def get_features(self, entity_ids):
        online_features = self.redis.mget(entity_ids)
        return torch.tensor(online_features)
```

Capabilities:

- Feature versioning
- Point-in-time correctness
- Feature sharing
- Batch and online serving

9. How would you implement a PyTorch model monitoring system for production deployments?

Monitoring Components:

- **Metrics Collection**: Prometheus
- **Visualization**: Grafana dashboards
- **Alerting**: PagerDuty integration

```
def monitor_predictions(predictions, labels):
    accuracy = (predictions == labels).float().mean()
    latency = time.time() - start_time
    log_metrics({'accuracy': accuracy, 'latency': latency})
```

Metrics:

- Model performance
- Inference latency
- Resource utilization

- Data drift indicators

10. Design a PyTorch-based continuous integration/deployment (CI/CD) pipeline for ML models.

Pipeline Stages:

- **Testing**: Unit, integration, regression tests
- **Validation**: Model performance checks
- **Deployment**: Blue-green deployment

```
def validate_model(model_artifact):
    test_results = run_test_suite(model_artifact)
    if test_results['accuracy'] > threshold:
        deploy_model(model_artifact)
```

Features:

- Automated testing
- Performance regression checks
- Canary deployments
- Rollback capability

Coding and Debugging

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

1. How would you implement a custom autograd function in PyTorch?

Solution:

Here's how to implement a custom autograd function by extending torch.autograd.Function:

```
class CustomExp(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        ctx.save_for_backward(x)
        return x.exp()

    @staticmethod
    def backward(ctx, grad_output):
        x, = ctx.saved_tensors
        return grad_output * x.exp()
```

Key points:

- Implement both forward and backward passes
- Use ctx.save_for_backward() to store tensors needed in backward pass
- Register gradients properly

2. Write a function to flatten nested PyTorch tensors of arbitrary depth.

Solution:

```
def flatten_tensors(nested_tensor):
    flat_list = []
    def recurse(tensor):
        if torch.is_tensor(tensor):
            flat_list.append(tensor.view(-1))
        elif isinstance(tensor, (list, tuple)):
            for t in tensor:
                recurse(t)
    recurse(nested_tensor)
    return torch.cat(flat_list)
```

Explanation: This recursive solution handles arbitrary nesting levels and concatenates all tensors into a single flat tensor.

3. Implement a memory-efficient way to compute pairwise distances between points in PyTorch.

Solution:

```
def pairwise_distance(x, y):
    norm_x = (x**2).sum(1).view(-1, 1)
    norm_y = (y**2).sum(1).view(1, -1)
    dist = norm_x + norm_y - 2.0 * torch.mm(x, y.t())
    return torch.clamp(dist, min=0.0).sqrt()
```

Key features:

- Avoids explicit loops
- Uses broadcasting
- Memory efficient matrix operations
- Numerically stable with clamp

4. How would you implement a custom DataLoader that handles variable-length sequences with proper padding?

Solution:

```
class PaddedDataLoader:
    def collate_fn(self, batch):
        lengths = torch.tensor([len(x) for x in batch])
        padded = torch.nn.utils.rnn.pad_sequence(batch,
                                               batch_first=True)
        return padded, lengths
```

Usage:

- Pass this collate_fn to DataLoader
- Handles variable length sequences
- Returns padded tensor and length mask
- Enables efficient batch processing

5. Write a PyTorch function to perform gradient clipping with norm scheduling.

Solution:

```
def clip_gradients(model, max_norm, scheduler=None):
    if scheduler:
        max_norm = scheduler.get_norm()
    torch.nn.utils.clip_grad_norm_(
        model.parameters(), max_norm)
    return max_norm
```

Features:

- Supports dynamic norm adjustment
- Prevents exploding gradients
- Compatible with learning rate schedulers
- Maintains training stability

6. Implement a custom loss function that combines L1 and focal loss for object detection.

Solution:

```
def hybrid_loss(pred, target, alpha=0.25, gamma=2.0):
    bce_loss = F.binary_cross_entropy_with_logits(
        pred, target, reduction='none')
    pt = torch.exp(-bce_loss)
    focal_loss = alpha * (1-pt)**gamma * bce_loss
    return focal_loss.mean() + F.l1_loss(pred, target)
```

Benefits:

- Handles class imbalance
- Combines classification and regression loss
- Adjustable focus on hard examples

7. Create a PyTorch function to perform mixed precision training with automatic loss scaling.

Solution:

```
def mixed_precision_step(model, loss, optimizer, scaler):
    scaler.scale(loss).backward()
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    scaler.step(optimizer)
    scaler.update()
```

Key aspects:

- Uses automatic mixed precision

- Handles loss scaling automatically
- Prevents underflow/overflow
- Maintains numerical stability

8. Implement a custom scheduler that combines cosine annealing with warm restarts.

Solution:

```
class CosineWarmRestarts(torch.optim.lr_scheduler._LRScheduler):
    def __init__(self, optimizer, T_0, T_mult=1):
        self.T_0 = T_0
        self.T_mult = T_mult
        super().__init__(optimizer)

    def get_lr(self):
        return [base_lr * (1 + math.cos(math.pi * self.T_cur
                / self.T_0)) / 2 for base_lr in self.base_lrs]
```

9. Write a function to perform channel-wise feature visualization for a CNN model.

Solution:

```
def visualize_channel(model, layer_name, channel_idx):
    feature = torch.zeros(1, 3, 224, 224, requires_grad=True)
    optimizer = torch.optim.Adam([feature], lr=0.1)
    activation = {}
    def hook(name): return lambda m, i, o: activation.update({name: o})
    model.get_submodule(layer_name).register_forward_hook(hook)
    return feature.data
```

10. Implement a custom DistributedSampler for multi-GPU training with uneven batch sizes.

Solution:

```
class CustomDistributedSampler(DistributedSampler):
    def __iter__(self):
        indices = list(range(len(self.dataset)))
        if self.shuffle:
            random.shuffle(indices)
        return iter(indices[self.rank::self.num_replicas])
```

Features:

- Handles uneven dataset sizes
- Maintains data distribution
- Supports shuffling
- Efficient parallel processing