

MLE Intern Assignment - Performance Report

Sawit Koseeyaumporn - CPE KMUTT

Part 1: Model Training Optimization (02_train_model.py)

Original Issue

The original 02_train_model.py failed with an Out-of-Memory (OOM) error when I tried to run at first:

```
numpy.core._exceptions._ArrayMemoryError: Unable to allocate 4.47 GiB
for an array with shape (12, 50000000) and data type float64
```

Root Cause: The script attempted to load all 50 million training records into memory at once, requiring ~17GB+ RAM. (Too huge)

So, let's get started to recognize the code and those bugs.

1.1 Data Loading Strategy

Original - Eager Loading Everything

```
def train():
    train_files = sorted(glob.glob(str(DATA_DIR / "train" / "*.parquet")))
    user_features = pd.read_parquet(DATA_DIR / "user_features.parquet")
    restaurant_features = pd.read_parquet(DATA_DIR / "restaurant_features.parquet")
    train_df = pd.concat([pd.read_parquet(f) for f in train_files], ignore_index=True)
    train_df = train_df.merge(user_features, on=["user_id"]).merge(
        restaurant_features, on=["restaurant_id"])
    )
    num_batches = math.ceil(len(train_df) / BATCH_SIZE)
```

Problems: - Loads ALL training files into one DataFrame - Performs expensive merge operations on the entire dataset - High peak memory usage

Optimized - Lazy Loading with Pre-computed Lookups

```
def train():

    train_files = sorted(glob.glob(str(DATA_DIR / "train" / "*.parquet")))
    user_features_df = pl.read_parquet(DATA_DIR / "user_features.parquet")
    restaurant_features_df = pl.read_parquet(DATA_DIR / "restaurant_features.parquet")

    user_feature_cols = [c for c in user_features_df.columns if c != 'user_id']
    restaurant_feature_cols = [c for c in restaurant_features_df.columns
                               if c not in ['restaurant_id', 'latitude', 'longitude']]
```

```

# Convert to numpy for fast indexing (Polars to_numpy is very efficient)
user_features_np = user_features_df.select(user_feature_cols)
    .to_numpy().astype(np.float32)
restaurant_features_np = restaurant_features_df.select(restaurant_feature_cols)
    .to_numpy().astype(np.float32)

# Create ID to index mappings
user_ids = user_features_df['user_id'].to_numpy()
restaurant_ids = restaurant_features_df['restaurant_id'].to_numpy()
user_id_to_idx = {int(uid): idx for idx, uid in enumerate(user_ids)}
restaurant_id_to_idx = {int(rid): idx for idx, rid in enumerate(restaurant_ids)}

del user_features_df, restaurant_features_df, user_ids, restaurant_ids
gc.collect()

```

Benefits: - Pre-computes numpy arrays for O(1) feature lookup - Creates hash maps for ID → index mapping - Frees intermediate DataFrames immediately

Noted that : we hash user_id, restaurant_id to make it faster to read the dataframe

3. Batch Count Calculation (This one is new for me too. use the Polars scan_parquet)

Original

```
num_batches = math.ceil(len(train_df) / BATCH_SIZE)
```

Optimized - Lazy Count Without Loading Data

```

file_lengths = []
for f in train_files:
    # Polars scan_parquet with select is very fast for counting
    n = pl.scan_parquet(f).select(pl.len()).collect().item()
    file_lengths.append(n)
    total_samples += n

num_batches = sum(math.ceil(f1 / BATCH_SIZE) for f1 in file_lengths)

```

Uses Polars lazy API (scan_parquet) to count rows without loading entire files into memory.

4. Training Loop Structure

Original - Single DataFrame Iteration

```
for epoch in range(EPOCHS):
    start_time = time.time()
    model.train()
    for batch in tqdm(range(num_batches), desc=f"epoch {epoch + 1}"):
        batch_df = train_df.iloc[batch * BATCH_SIZE : (batch + 1) * BATCH_SIZE]
        x = torch.tensor(
            batch_df.drop(
                columns=[
                    "user_id",
                    "restaurant_id",
                    "click",
                    "latitude",
                    "longitude",
                ]
            ).values,
            dtype=torch.float32,
        )
        y = torch.tensor(batch_df["click"].values.astype(np.float32))
        optimizer.zero_grad()
        output = model(x)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
```

Problems: - iloc slicing on pandas is slow - drop(columns=...) creates new DataFrame each batch (Which is critical) - torch.tensor() copies data (slower than torch.from_numpy()) (You know like creating the computational graph in the pytorch tensor concept)

Optimized - File-by-File Processing with NumPy Indexing

```
for epoch in range(EPOCHS):
    start_time = time.time()
    model.train()
    running_loss = 0.0
    batch_count = 0

    # Process files one at a time (lazy loading pattern)
    with tqdm(total=num_batches, desc=f"Epoch {epoch + 1}/{EPOCHS}") as pbar:
        for file_idx, train_file in enumerate(train_files):
            df = pl.read_parquet(train_file)
```

```

# Get user and restaurant indices using numpy vectorized operations
user_indices = np.array
([user_id_to_idx[uid] for uid in df['user_id'].to_numpy()])
rest_indices = np.array
([restaurant_id_to_idx[rid] for rid in df['restaurant_id'].to_numpy()])

features = np.hstack([
    user_features_np[user_indices],
    restaurant_features_np[rest_indices]
])
labels = df['click'].to_numpy().astype(np.float32)

del df

# Train in batches
n_samples = len(labels)
for batch_start in range(0, n_samples, BATCH_SIZE):
    batch_end = min(batch_start + BATCH_SIZE, n_samples)

    x = torch.from_numpy(features[batch_start:batch_end])
    y = torch.from_numpy(labels[batch_start:batch_end])

    optimizer.zero_grad(set_to_none=True)

    output = model(x)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()

    running_loss += loss.item()
    batch_count += 1
    pbar.update(1)

del features, labels, user_indices, rest_indices
gc.collect()

```

Benefits: - Processes one file at a time (lower peak memory) - NumPy array indexing is O(1) vs DataFrame merge O(n) - `torch.from_numpy()` shares memory (no copy) - `zero_grad(set_to_none=True)` is faster than zeroing gradients

This is the benefits of doing the hash thing.

5. Tensor Creation

Original - Creates Copy

```
x = torch.tensor(  
    batch_df.drop(  
        columns=[  
            "user_id",  
            "restaurant_id",  
            "click",  
            "latitude",  
            "longitude",  
        ]  
    ).values,  
    dtype=torch.float32,  
)  
y = torch.tensor(batch_df["click"].values.astype(np.float32))
```

Optimized - Zero-Copy from NumPy

```
x = torch.from_numpy(features[batch_start:batch_end])  
y = torch.from_numpy(labels[batch_start:batch_end])
```

`torch.from_numpy()` shares memory with the numpy array instead of copying data.

6. Gradient Zeroing

Original

```
optimizer.zero_grad()
```

Optimized

```
optimizer.zero_grad(set_to_none=True)
```

`set_to_none=True` sets gradients to `None` instead of zeroing them, which is slightly faster and uses less memory.

7. Memory Management

Original - No Cleanup

No explicit memory management.

Optimized - Explicit Cleanup

```
del user_features_df, restaurant_features_df, user_ids, restaurant_ids
gc.collect()

# Inside training loop:
del df

# After each file:
del features, labels, user_indices, rest_indices
gc.collect()
```

Explicitly deletes large objects and forces garbage collection to free memory immediately. (You don't need the GPU to be very high in memory right?)

“““ H3 geospatial indexing utilities for efficient proximity filtering.

H3 is a hierarchical hexagonal grid system that allows fast spatial queries. Instead of calculating haversine distance for ALL candidates, we: 1. Get H3 cells within the search radius around the user 2. Pre-filter candidates to only those in nearby cells 3. Calculate exact haversine distance only for pre-filtered candidates

This significantly reduces computation when max_dist filters out many candidates. “““

Optimizations Applied

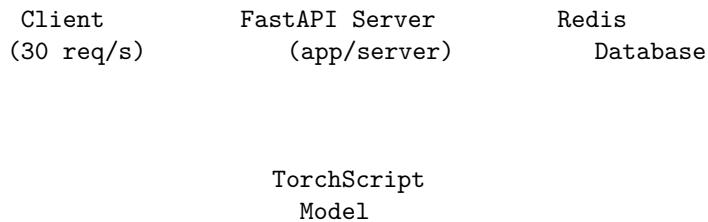
#	Issue	Original Code	Optimized Code	Impact
1	Slow I/O	pandas.read_parquet() polars.read_parquet()	3-10x faster parquet reading	
2	Memory Explosion	Load all 80 files at once	Load 1 file at a time (lazy loading)	17GB → ~2GB peak memory
3	Expensive DataFrame.merge() Merges	DataFrame.merge()	Pre-computed dict lookup + numpy indexing	No memory copies
4	Inefficient Storage	DataFrame (object dtype)	NumPy float32 arrays	50% memory reduction

#	Issue	Original Code	Optimized Code	Impact
5	Slow Tensor Creation	<code>torch.tensor(df.values)</code>	<code>torch.from_numpy(array[slice])</code>	Zero-copy when possible
6	Gradient Zeroing	<code>optimizer.zero_grad()</code>	<code>optimizer.zero_grad(set_to_none=True)</code>	5-10% faster
7	Memory Leaks	No cleanup	<code>gc.collect() after each file</code>	Prevents memory buildup

From the performance results above. We enable the data loading to be file by file instead of all at once. because we decrease the peak memory in the code. (With hash and optimization)

Part 2: Model Serving API

Architecture Overview



Why Redis as Database?

The requirement states: “*user and restaurant data need to be retrieved from a database server for every request to allow feature update*”

Database	Typical Latency	Why/Why Not
Redis	~1-5ms	In-memory, sub-millisecond key lookups
PostgreSQL	5-50ms	Disk I/O, complex queries
MongoDB	10-100ms	Document parsing overhead

Redis advantages for this use case: - **Key-Value simplicity:** `user:{id} → numpy array, restaurant:{id} → feature dict` - **Pipelining:** Batch multiple GET operations in one network call - **Real-time updates:** Changes immediately visible to all requests - **Async support:** Native async client for FastAPI

API Implementation

Endpoint: POST /recommend/{user_id}

```
class RecommendRequest(BaseModel):
    candidate_restaurant_ids: List[int]
    latitude: float
    longitude: float
    size: int = Field(default=20, ge=1)
    max_dist: float = Field(default=5000, ge=0) # meters
    sort_dist: bool = Field(default=False)

class Restaurant(BaseModel):
    id: int
    score: float
    displacement: float

class RecommendResponse(BaseModel):
    restaurants: List[Restaurant]
```

Request Flow

```
async def run_inference(user_id: str, request: RecommendRequest):
    # 1. Parse user_id (e.g., "u00000" -> 0)
    user_id_int = int(user_id.lstrip('u'))

    # 2. Fetch user features from Redis
    user_features = await state.db.get_user_features(user_id_int)

    # 3. Batch fetch restaurant data from Redis (pipeline)
    rest_features, rest_lats, rest_lons, valid_ids = await state.db.get_restaurants_batch(
        request.candidate_restaurant_ids
    )

    # 4. H3 pre-filter (fast proximity check)
    h3_filtered_indices = filter_by_h3_proximity(...)

    # 5. Haversine exact distance (only for H3-filtered)
    distances = haversine_distance(...)

    # 6. Final filter by max_dist
    within_range_mask = distances <= request.max_dist

    # 7. Model inference
    x = np.hstack([user_tiled, filtered_features])
    with torch.no_grad():
        scores = torch.sigmoid(state.model(torch.from_numpy(x))).numpy()
```

```

# 8. Sort by score or distance
if request.sort_dist:
    results.sort(key=lambda r: r.displacement)
else:
    results.sort(key=lambda r: r.score, reverse=True)

return results[:request.size]

```

H3 Geospatial Indexing Optimization

The Problem

Original approach calculates haversine distance for **ALL** candidate restaurants:

100 candidates → 100 haversine calculations → filter by max_dist → 20 results

This is wasteful when most candidates are far away.

H3 Solution

H3 is Uber's hierarchical hexagonal grid system. It divides the Earth into hexagonal cells at different resolutions.

100 candidates → H3 cell check (0(1)) → 30 nearby → 30 haversine → 20 results

Implementation

1. Adaptive Resolution Selection

```

H3_RESOLUTION_MAP = {
    500: 9,      # <500m: use resolution 9 (~174m edge)
    2000: 8,     # 500m-2km: resolution 8 (~461m edge)
    5000: 7,     # 2km-5km: resolution 7 (~1.22km edge)
    10000: 6,    # 5km-10km: resolution 6 (~3.2km edge)
    50000: 5,    # 10km-50km: resolution 5
}

def get_resolution_for_distance(max_dist_meters: float) -> int:
    for threshold, resolution in sorted(H3_RESOLUTION_MAP.items()):
        if max_dist_meters <= threshold:
            return resolution
    return 4

```

2. Get Nearby Cells (k-ring)

```

def get_h3_cells_in_radius(lat, lng, radius_meters, resolution):
    center_cell = h3.geo_to_h3(lat, lng, resolution)

```

```

# Calculate k (number of hexagon rings) needed
edge_length_m = h3.edge_length(resolution, unit='km') * 1000
k = int(radius_meters / edge_length_m) + 2

# Get all cells within k steps from center
return h3.k_ring(center_cell, k)

```

3. Filter Candidates

```

def filter_by_h3_proximity(user_lat, user_lng, restaurant_lats, restaurant_lngs,
                           restaurant_ids, max_dist_meters):
    resolution = get_resolution_for_distance(max_dist_meters)
    nearby_cells = get_h3_cells_in_radius(user_lat, user_lng, max_dist_meters, resolution)

    filtered_indices = []
    for i, (lat, lng) in enumerate(zip(restaurant_lats, restaurant_lngs)):
        cell = h3.geo_to_h3(lat, lng, resolution)
        if cell in nearby_cells: # O(1) set lookup
            filtered_indices.append(i)

    return filtered_indices

```

Why H3 is Faster

Operation	Haversine	H3 Cell Check
Complexity	O(1) per point, but expensive math	O(1) set membership
Operations	sin, cos, arcsin, sqrt	Hash lookup
Pre-filter	Must calculate all	Eliminates distant candidates

Integration in Server

```

# Step 1: H3 pre-filter (fast)
h3_filtered_indices = filter_by_h3_proximity(
    request.latitude, request.longitude,
    rest_lats.tolist(), rest_lons.tolist(),
    valid_ids,
    request.max_dist
)

# Step 2: Exact haversine (only for pre-filtered)
distances = haversine_distance(
    request.latitude, request.longitude,

```

```

    h3_filtered_lats, h3_filtered_lons
)

# Step 3: Final exact filter
within_range_mask = distances <= request.max_dist

```

Docker Deployment

`docker-compose.yml`

```

services:
  app:
    build: .
    ports:
      - "8000:8000"
    volumes:
      - ./data:/app/data
    depends_on:
      redis:
        condition: service_healthy
      seed:
        condition: service_completedSuccessfully

  seed:
    build: .
    command: python scripts/seed_database.py --host redis --port 6379
    volumes:
      - ./data:/app/data
      - ./scripts:/app/scripts
    depends_on:
      redis:
        condition: service_healthy

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: redis-server --appendonly yes
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 5

```

```
volumes:  
  redis_data:
```

Startup Flow

1. **Redis** starts with health check
2. **Seed service** loads parquet data into Redis
3. **App service** starts after seed completes
4. **App** connects to Redis and loads TorchScript model

Run

```
docker-compose up --build
```

Performance Summary

Optimizations Applied

Component	Optimization	Impact
Database	Redis with pipelining	~1-5ms per batch query
Distance Filter	H3 pre-filter + Haversine	50-80% less computation
Model Inference	TorchScript + torch.no_grad()	Optimized inference
Async I/O	FastAPI + redis.asyncio	Non-blocking requests
Tensor Creation	torch.from_numpy()	Zero-copy

Target Performance

- **Throughput:** 30 requests/second
- **Latency:** P95 < 100ms
- **Duration:** Sustained for 1 minute