Foqus ML Task — Contrastive MRI Embedding Pipeline

Mohsen Soltanpour — September 2025

Abstract:

This report presents the design and implementation of a complete machine learning pipeline for contrastive representation learning on synthetic multi-coil MRI data. The work follows the Foqus ML engineering task specification and is structured into four stages: dataset generation, model architecture, training, and repository cleanup.

In Task 1, I developed modular preprocessing transformations — normalization, equispaced undersampling in k-space, light augmentations, and tensor conversion — to produce realistic, model-ready MRI inputs. In Task 2, I implemented a compact convolutional encoder (MRIEmbeddingModel) that maps complex MRI inputs to fixed-size, L2-normalized embeddings suitable for contrastive objectives. Task 3 involved training this encoder using triplet-margin loss on the RandomPhantomTripletDataset, with diagnostics including positive/negative distance statistics, violation rates, and Recall@1 retrieval accuracy. Finally, Task 4 focused on productionizing the repository: adding packaging metadata, dependency management, pre-commit hooks, CI integration, CLI entrypoints, and a complete runbook for reproducibility.

Task 1 — Dataset Generation

Objective

The first task required implementing preprocessing transformations for simulated phantom MRI data to prepare it for contrastive learning. Specifically, the transformations needed to handle multi-coil, complex-valued MRI data, simulate undersampling in k-space, apply light augmentations, and ensure the data format was compatible with PyTorch convolutional models.

The goal was to design modular, production-ready components that could be reused within a broader ML pipeline, while maintaining clarity, correctness, and efficiency.

Implementation & Design Rationale

1. Normalization (`Normalize`)

- **Problem:** MRI data is inherently complex-valued, and raw coil outputs vary widely in intensity. Without normalization, models may overfit to magnitude ranges instead of learning meaningful representations.
- **Approach:**
    - Computed the **magnitude image**:

$$Magnitude = \sqrt{real^2 + img^2}$$

    - Derived **per-sample mean and standard deviation** over all coils and pixels.
    - Normalized both **real** and **imag** channels using these magnitude statistics.
- **Extras:**
    - Added an **epsilon guard** to prevent division by near-zero values.
    - Optional **clipping** to a configurable range for outlier robustness.
- **Why:** This ensures data consistency across batches, stabilizing training while preserving both phase and amplitude information.

---

2. Equispaced Undersampling (`EquispacedUndersample`)

- **Problem:** Accelerated MRI acquisition undersamples k-space to reduce scan times. The task required simulating undersampling strategies (e.g., 4x with 8% center fraction).
- **Approach:**
    - Applied a **2D FFT** (provided helper) to move data into k-space.
    - Constructed a 1D undersampling mask along the frequency-encoding direction:
        - Keep every *nth* line (acceleration factor).
        - Always fully sample a central low-frequency band (center fraction).
    - Applied the mask and returned to image space with inverse FFT.
- **Why:** This reproduces realistic undersampling artifacts and ensures the model learns to extract meaningful representations under clinically relevant acceleration settings.

---

3. Augmentation (`Augmentation`)

- **Problem:** Contrastive learning benefits from diverse but semantically consistent views of the same sample. MRI-specific augmentations (like noise) were less emphasized here, so geometric variations were applied.
- **Approach:**
    - Random **small rotations** (±10°).
    - Random **pixel translations** (±2px).
    - Packed real/imag channels together to ensure augmentations are applied consistently.
    - Used PyTorch's `affine_grid` + `grid_sample` for differentiable transforms.

- **Why:** Introduces small spatial variability without destroying anatomical structure, improving embedding robustness.

---

4. Tensor Conversion (`ToCompatibleTensor`)

- **Problem:** Standard CNNs expect `(channels, H, W)` format, not `(C, H, W, 2)` with complex-valued data.
- **Approach:**
    - Flattened the real and imaginary parts into separate channels.
    - Final format: `(C*2, H, W)`.
- **Trade-off:**
    - Pros: Preserves both real and imaginary signals explicitly, enabling downstream models to learn from phase as well as magnitude.
    - Cons: Doubles channel dimensionality, but this is manageable and preferable for a foundation feature extractor.

---

5. Helper Functions (`fft`, `ifft`)

- Designed reusable wrappers around PyTorch's FFT/IFFT operators:
    - Handled both real/imag tensor formats and native complex types.
    - Applied **fftshift** / **ifftshift** to center frequencies.
- Ensured modularity and compatibility across tasks.

Example Pipeline & Results

- Applied transforms in sequence:
  Undersample → Augmentation → Normalize → ToCompatibleTensor
    - *Undersample first:* ensures augmentation operates on realistic aliased images.
    - *Normalize after augmentation:* ensures consistent intensity distributions.
    - *Conversion last:* provides model-ready input format.
- Visualization confirmed that:
    - Low-frequency aliasing was correctly simulated.
    - Augmented images retained structural integrity.
    - Normalization produced stable value ranges across samples.

Dataset Preview Script (`preview_task1_dataset.py`)

To verify correctness and document results, I implemented a lightweight script that instantiates the dataset and saves representative preview images.

Purpose

- Compose the transform pipeline with configurable parameters (acceleration, center fraction, augmentation strengths).
- Instantiate `RandomPhantomDataset` with these transforms.
- Save preview images to disk for inspection and reporting.

Key Features

- **Reproducibility:** Supports setting a fixed random seed.
- **CLI Configurable:** Command-line arguments for number of samples, output directory, acceleration factor, center fraction, max rotation, max shift, and seed.

- **Visualization Strategy:**
    - Converts `(channels, H, W)` tensors back into `(coils, 2, H, W)`.
    - Reconstructs magnitude per coil.
    - Computes a **coil-wise max projection** to display as a single grayscale image.
- **Headless Saving:** Configured `matplotlib` to `Agg` backend for compatibility with servers/CI.
- **Output:** Generates PNGs like `task1_sample_0.png`, stored in `report/figs`.

Rationale

This script separates **dataset validation** from the training loop:

- Lightweight and independent of training.
- Ensures that transforms produce valid, visually interpretable images.
- Provides ready-to-embed figures for the final report.

Deliverables

- **Code:** `transforms.py` with modular, unit-testable implementations.
- **Dataset Instance:** Previewed `RandomPhantomDataset` with transforms applied.
- **Writeup (this report):** Documented choices, reasoning, and validation images.

Key Takeaways

This step emphasized **data engineering discipline**:

- Designed transformations as reusable PyTorch `nn.Module`s for easy composition in training pipelines.
- Followed MRI physics principles (complex-valued data, k-space undersampling).
- Balanced simplicity with extensibility — ensuring that future extensions (e.g., coil compression, noise simulation) can be integrated seamlessly.

Ultimately, this created a robust and production-aligned dataset preparation pipeline suitable for pretraining MRI foundation models with contrastive learning.

Task 2 — Model Architecture

Objective

The second task required designing a model that takes **multi-coil complex MRI images** (real and imaginary channels stacked from Task 1) and produces a **fixed-size embedding vector** suitable for contrastive learning. The model should be compact, efficient, robust to small batch sizes, and production-friendly while maintaining extensibility for future datasets.

Final Architecture

Input / Output

- **Input:**
  $x \in \mathbb{R}^{B \times (2 \cdot N\_coils) \times H \times W}$
  where $2 \cdot N\_coils$ channels come from stacking real and imaginary parts.

- **Output:**
  $z \in \mathbb{R}^{B \times D}$ (default $D = 256$)
  embeddings are **L2-normalized** by default for stable distance computations.

Layer Stack

1. **Stem**
    - `LazyConv2d(out=32, k3, s2, p1) → GroupNorm → GELU`
    - Learns low-level features and immediately downsamples to reduce compute.
    - `LazyConv2d` adapts to any coil configuration without code changes.
2. **Backbone (×4 stages)**
    - Each stage:
      `Conv2d(k3, s2, p1) → GroupNorm → GELU → (optional Dropout)`

- Default widths: `(32, 64, 128, 256)`
- Progressive stride-2 downsamples reduce spatial resolution `/32` overall. Example: `256×256 → 8×8`.

3. **Head**
   - `AdaptiveAvgPool2d(1) → Linear(256→D)`
   - Optional **MLP projection head** (`D → 2D → D`) for richer embeddings.
   - Optional **L2 normalization** (enabled by default).

Implementation Details

- **Normalization:** `GroupNorm` chosen over `BatchNorm` for stability with small batch sizes (common in MRI).
- **Activation:** `GELU` for smoother gradients than ReLU.
- **Initialization:** Kaiming Normal for convs, Xavier Uniform for linears.
- **API:** `forward(..., normalize=None)` and `embed(no-grad)` for clean usage in both training and inference.

Why This Model Was Chosen

1. **Contrastive-ready:** L2-normalized embeddings make cosine and Euclidean distances stable and bounded, simplifying triplet loss optimization.
2. **Small-batch robustness:** GroupNorm avoids noisy batch statistics when training with limited GPU memory.
3. **Efficiency:** Aggressive stride-2 downsampling keeps compute and memory low while retaining sufficient representation capacity.
4. **Flexibility:** `LazyConv2d` adapts to arbitrary coil counts automatically.

5. **Production-friendly:** Few moving parts, easy to export/deploy, modular code structure, and extensibility via optional MLP head or wider backbones.

Alternatives Considered

Model	Advantages	Why Not Used Here
ResNet (e.g., ResNet-18)	Strong baseline, residuals ease training, widely adopted	Heavier than needed for synthetic phantom data; BatchNorm unstable with small batches
UNet Encoder	Excellent for medical imaging, strong multi-scale features	Overkill for global embeddings; higher memory/computation cost
Vision Transformers (ViT / Swin)	Capture long-range dependencies; scalable	Data-hungry; unstable with small batches; more complex to train
3D CNNs	Capture volumetric context across slices	Very high memory/compute cost; unnecessary for 2D phantom task

I deliberately prioritized a **lean, stable CNN** that is easy to train with small synthetic datasets, preserves both magnitude and phase, and is directly usable in Task 3. Heavier models like ResNet or ViTs could be explored later if scaling to real MRI datasets.

Embedding Space Design

- **Unit-norm embeddings ($||z||_2 = 1$)** → cosine similarity = dot product; distances are bounded.
- **Embedding dimension (D = 256)** → large enough to capture coil variability, compact enough for efficiency.
- **Optional MLP head** → left disabled by default to avoid overfitting on small datasets, but available for scaling up.

Practical Considerations

- **Throughput & Memory:** With `/32` downsampling and GAP, the model is lightweight and fast on modest GPUs.
- **Extensibility:** Can swap in ResNet or transformer blocks, or enable the MLP head, without breaking the pipeline.
- **Future Experiments:** Ablations could test different widths, enabling the projection head, or experimenting with LayerNorm in the head.

Deliverables

- **Code:** `model.py` containing the modular `MRIEmbeddingModel`.
- **Documentation:** Rich docstrings outlining input/output, rationale, and configuration options.
- **Flexibility:** Supports custom embedding sizes, dropout, normalization toggle, and projection head.

Key Takeaways

- Built a **compact CNN encoder** tailored for complex MRI inputs.
- Prioritized **simplicity, robustness, and production-readiness** over unnecessary complexity.
- Demonstrated awareness of alternative architectures and deliberately selected the most pragmatic solution for this task.
- Produced embeddings that are **contrastive-learning ready** and directly usable in Task 3.

Task 3 — Training

Objective

Train the encoder from Task 2 to produce **contrastive embeddings** that pull together two transformed views of the **same** phantom (positive pair) and push away a **different** phantom (negative), using a **triplet-margin loss** on L2-normalized embeddings.

Dataset & Transforms

Triplet construction

I used `RandomPhantomTripletDataset`, which returns `(same1, same2, diff)` where `same1` and `same2` originate from the **same base phantom** with different transform lists, and `diff` comes from a **different** base phantom.

Two transform lists (as required)

- **List A (view 1):** `EquispacedUndersample(accel=4, center=0.08) → Augmentation → Normalize → ToCompatibleTensor`
- **List B (view 2):** `EquispacedUndersample(accel=8, center=0.04) → Augmentation → Normalize → ToCompatibleTensor`

Validation uses the **same** undersampling + normalization stacks but **without augmentation**. I also set:

- `offset = len(train_dataset)` to avoid overlap with training, and
- `deterministic = True` for reproducible validation.

Training Loop & Loss

Encoder

`MRIEmbeddingModel(embed_dim=256, normalize=True)` from Task 2. L2-normalization ensures distances are well-behaved.

Triplet margin loss

For embeddings $e_1 = f(x_{same1})$, $e_2 = f(x_{same2})$, $e_d = f(x_{diff})$ (all L2-normalized):

$$L = max\left(0, \left\|e_1 - e_2\right\|_1 - \left\|e_1 - e_d\right\|_2 + m\right) m = 0.2$$

This minimizes positive distance and enforces a margin to negatives. With unit-norm embeddings, Euclidean and cosine distances are consistent and stable.

Optimization & scheduling

- **Optimizer:** `AdamW(lr=3e-4, weight_decay=1e-4)` — stable with good generalization.
- **Scheduler:** `ReduceLROnPlateau` on **val loss** (factor 0.5, patience 2, floor 1e-6) to adapt learning rate without manual tuning.
- **Early stop:** patience 5 on best validation loss; best checkpoint persisted.

Reproducibility & loaders

- Global seeding for `torch` and `numpy`.
- Optional `worker_init_fn` for DataLoader to reduce worker RNG drift.
- Pinned memory, `drop_last=True` for train to keep batch size consistent.

Diagnostics, Validation & Curves

Per epoch, I logged:

- **Loss (train/val)** — primary optimization signal.
- **Distance stats:**

    - $d_{pos} = \|e_1 - e_2\|_2$ — should **decrease**
    - $d_{neg} = \|e_1 - e_2\|_2$ — should **increase**
- **Violation rate (%):** fraction with $d_{pos} + m > d_{neg}$ — lower is better.
- **Recall@1 (val):** Treat `same1` as queries, `same2` as gallery (no shuffle). For each query, does the top-1 match its paired view?

Outputs:

- **PNG** loss curves saved to `report/figs/task3_curves.png`.
- **CSV** with loss history and final summary metrics to `report/figs/task3_curves.csv`.
- **Best checkpoint** to `report/figs/task3_best.pt`.

Run Command (example)

```
PYTHONPATH=. python train.py \
  --epochs 20 \
  --train-length 800 --val-length 200 \
  --batch-size 16 --num-workers 0 \
  --image-size 128 --n-coils 8 \
  --lr 3e-4 --margin 0.2 \
  --out-dir report/figs
```

Results Expectations (synthetic setup)

Given the simplicity of synthetic phantoms and the strong signal from undersampling differences, I targeted:

- **Low val loss**, **low violation rate**, **high Recall@1**

- Typical ranges I observed/expect:
    - $d_{pos} <= 0.3$, $d_{neg} >= 1.1$
    - **viol% ≤ 1%**, **Recall@1 ≥ 99%**

(Exact numbers depend on seed, batch size, and image size; the trend and separation matter most.)

Engineering Notes

- **Augment only in train**; val is deterministic to isolate representation quality from augmentation noise.
- **Metric granularity:** distances and violation % give immediate insight if the margin is too small/large or embeddings collapse.
- **Checkpoints & artifacts:** all artifacts land under a single `--out-dir` for easy packaging with the report.

Potential Improvements

If I had more time or a larger (real) dataset, I would consider:

- **InfoNCE / NT-Xent** with temperature for richer negatives (batch-all).
- **Hard / semi-hard triplet mining** within a batch to speed separation.
- **Margin scheduling** or cosine distance directly (equivalent under L2-norm).
- **AMP + grad clipping** for faster, stabler training.
- **K-space mask diversity** (variable density, Poisson-disc) to improve invariance.
- Additional retrieval metrics (**Recall@K**), clustering (**NMI / ARI**), and **t-SNE/UMAP** projections of embeddings for sanity checks.

Deliverables

- `train.py` — complete training/validation loop with metrics, scheduling, early stop, and artifact saving.
- **Artifacts** — loss curves (PNG), logs (CSV), and best model checkpoint (PT).

Task 4 — Repository Cleanup & Packaging

Objective

The final task focused on transforming the prototype code from Tasks 1–3 into a **production-ready, maintainable, and reproducible repository**. Beyond working models, ML engineers must ensure code is easy to install, test, and extend by others. This meant adding packaging metadata, dependency management, pre-commit tooling, an entrypoint, tests, CI integration, and a complete runbook.

Implementation

1. Packaging (`pyproject.toml` + `requirements.txt`)

- Defined project metadata (`name`, `version`, `description`, `authors`) in `pyproject.toml`.
- Centralized runtime dependencies (Torch, NumPy, matplotlib, tqdm, PyYAML, einops).
- Added optional `dev` dependencies (pytest, coverage, ruff, black, pre-commit, mypy).
- Configured ruff, black, and pytest directly in `pyproject.toml`.
- Provided `requirements.txt` for lightweight installs (pinned Torch + NumPy, flexible versions for utilities).

Why: Guarantees reproducibility, single-source dependency management, and makes the project installable as a proper Python package.

---

2. Pre-commit Hooks

- Configured `.pre-commit-config.yaml` to enforce:
    - **Ruff** linting and auto-fix (py-upgrade, import sorting, code smells).
    - **Black** auto-formatting.
    - **Basic hygiene checks** (trailing whitespace, end-of-file newline, YAML/JSON validity).

Why: Enforces style and correctness at commit-time, reducing CI failures and messy diffs.

---

3. CLI Entrypoint

- Registered `foqus-train` in `pyproject.toml` to run training directly from the CLI.
- Added `cli.py` wrapper to delegate to `train.py` (`main/run/train`) with fallbacks.

Why: Improves developer experience — no need to remember script paths, and the repo can be executed as a proper Python module.

---

4. Repository Runbook (Embedded)

Below is the **full runbook** I wrote to ensure anyone can set up, test, and run the project reproducibly.

Foqus ML Task Runbook

This document provides a clean, reproducible setup and run guide for the repository, including:

- Environment setup (venv)
- Editable install
- Formatting/linting checks
- CLI smoke test
- Unit tests
- CI integration and troubleshooting

Step 0: Clone

```
git clone https://github.com/msoltanpour/foqus-ml-task.git
cd foqus-ml-task
```

Step 1: Cleanup (optional)

```
find . -name "__pycache__" -type d -prune -exec rm -rf {} +
```

Step 2: Create Virtualenv

```
python3 -m venv .venv
source .venv/bin/activate
python -V
python -m pip install -U pip
```

Step 3: Editable Install

```
pip install -e ".[dev]"
```

This installs runtime deps (torch, numpy, matplotlib, tqdm, pyyaml, einops) and dev tools (pytest, ruff, black, pre-commit, etc.).

Step 4: Pre-commit (optional)

```
pre-commit install
pre-commit run -a
```

Run repeatedly until all files pass without changes.

Step 5: CLI Smoke Test

```
foqus-train --epochs 1 --batch-size 8 --num-workers 0 --device cpu
--out-dir tmp_run
ls -lah tmp_run
```

Expected artifacts in `tmp_run/`:

- `task3_curves.png` (loss plot)
- `task3_curves.csv` (loss history)
- `task3_best.pt` (best checkpoint)

Step 6: Run Unit Tests

```
pytest -q
```

All tests should pass (PyTorch Lazy modules may show harmless warnings).

Step 7: Full Training Example

```
foqus-train \
  --epochs 20 \
  --train-length 800 --val-length 200 \
  --batch-size 16 --num-workers 0 \
  --image-size 128 --n-coils 8 \
  --lr 3e-4 --margin 0.2 \
  --out-dir report/figs/exp1
```

Outputs will be saved under `report/figs/exp1/`.

Step 8: CI on GitHub

- Workflow file: `.github/workflows/ci.yml`
- Runs: Ruff, Black (check), pytest on Python 3.11
- Badge included in README

Appendix A: Full Reset

```
deactivate
rm -rf .venv
find . -name "__pycache__" -type d -prune -exec rm -rf {} +
python3.11 -m venv .venv
source .venv/bin/activate
pip install -U pip
pip install -e ".[dev]"
foqus-train --epochs 1 --batch-size 8 --num-workers 0 --device cpu
--out-dir tmp_run
pytest -q
```

Appendix B: Troubleshooting

- `ModuleNotFoundError`: ensure you ran `pip install -e ".[dev]"`.
- Editable install fails: check `[tool.setuptools.packages.find]` in `pyproject.toml`.
- Ruff lint warnings: run `pre-commit run -a`.
- CUDA not found: use `--device cpu` or install a CUDA-enabled Torch.

Why This Matters

This task demonstrates engineering discipline: the repo is not just functional, but **installable, reproducible, and CI-validated**. A new contributor can clone, set up, lint, test, and train in <15 minutes. This is what turns research code into production-grade ML engineering.

Further Options

- **Dockerfile** for containerized reproducibility.
- **Makefile / tox / nox** for consistent task automation.
- **Coverage reports** in CI.
- **Mypy strict typing** in CI + pre-commit.
- **Docs site (Sphinx/MkDocs)** auto-built from README + runbook.
- **GPU-enabled CI** for Torch integration testing.

Deliverables

- `pyproject.toml` + `requirements.txt` — packaging & dependencies.
- `.pre-commit-config.yaml` — enforced linting/formatting/hygiene.
- **CLI Entrypoint (`foqus-train`)** — easy execution.
- **Runbook (above)** — complete reproducibility checklist.
- **CI workflow + tests** — quality enforced at every push.