PhD Research Proposal

AI-Driven Anatomical and Response-Adapted Proton Therapy: Distinguishing Biological from Anatomical Changes for Personalized Dose Optimization

Applicant Name
RAPTORplus Marie-Sklodowska-Curie-Action EU Doctoral Network
Supervisor: Professor Stine Sofia Korreman

Aarhus University & Aarhus University Hospital
Danish Centre for Particle Therapy

January 15, 2026

Contents

Abstract

Adaptive radiotherapy has revolutionized cancer treatment by accounting for interfractional anatomical variations. However, current approaches primarily focus on dose restoration to compensate for anatomical changes, overlooking the fact that many image changes during treatment reflect genuine biological responses—tumor regression or progression, and early normal-tissue effects—which may require dose level adaptation rather than mere dose restoration. This PhD project aims to develop novel AI-based methods to distinguish between anatomical and biological components of daily image changes during proton therapy, and subsequently implement corresponding dose optimization strategies.

The research will progress through four interconnected methodological tasks: (1) synthetic image generation to produce anatomically and biologically plausible training datasets; (2) AI-based response characterization using multimodal features including population anatomy models, quantitative image biomarkers, radiomics, and accumulated dose; (3) dose optimization strategies that execute appropriate dose restoration or adaptation based on the identified change type; and (4) in-silico integration implementing a proof-of-concept pipeline for clinical evaluation.

This work will contribute to the RAPTORplus project's vision of "Right-time Adaptive Particle Therapy Of canceR" by enabling treatment personalization through both anatomical and biological adaptation, ultimately improving patient outcomes through more precise and individualized radiation therapy.

Keywords: Adaptive Proton Therapy, Artificial Intelligence, Deep Learning, Radiomics, Dose Optimization, Biological Response, Medical Image Analysis

1 Introduction

1.1 Background and Motivation

Proton therapy represents a major advancement in radiation oncology, offering superior dose conformality compared to conventional photon therapy due to the unique physical properties of charged particles. The Bragg peak phenomenon allows precise dose deposition at the tumor site while sparing surrounding healthy tissues [1]. However, this precision comes with increased sensitivity to anatomical variations—even minor changes in patient anatomy can lead to significant dose perturbations due to the finite range of proton beams.

Adaptive radiotherapy (ART) has emerged as a solution to account for interfractional variations, utilizing daily imaging to modify treatment plans. Current ART implementations primarily focus on *anatomical adaptation*, where the goal is to restore the planned dose distribution when anatomical changes occur. However, this approach overlooks a critical aspect: many image changes during treatment reflect *biological responses*—tumor regression or progression, changes in tissue density due to treatment effects, and early normal-tissue reactions—which may necessitate genuine dose level adaptation rather than simple dose restoration.

The distinction between anatomical and biological changes is crucial for optimizing treatment outcomes:

- **Anatomical changes** (e.g., patient positioning variations, organ filling states) require dose restoration to maintain the original treatment plan.
- **Biological changes** (e.g., tumor shrinkage, treatment response, early toxicity) may require dose escalation, de-escalation, or redistributed to maximize tumor control while minimizing toxicity.

Current clinical practice lacks robust methods to automatically distinguish between these change types, leading to suboptimal adaptive strategies. This PhD project addresses this critical gap by developing AI-based methods to characterize image changes and implement appropriate dose optimization strategies.

1.2 Research Problem Statement

The central research problem is: *How can we automatically distinguish between anatomical and biological components of daily image changes during proton therapy, and how should dose optimization strategies differ based on this characterization?*

This problem encompasses several technical challenges:

1. **Limited training data:** Biological response data with ground truth labels is scarce, necessitating synthetic image generation methods.
2. **Multimodal integration:** Effectively combining anatomical imaging, radiomics, dose accumulation, and population models for response characterization.
3. **Uncertainty quantification:** Distinguishing genuine biological changes from image noise and registration uncertainties.
4. **Clinical translation:** Integrating AI-based methods into clinical treatment planning systems with acceptable computational efficiency.

1.3 Research Objectives

The overarching goal of this PhD project is to develop and validate AI-driven methods for distinguishing anatomical from biological image changes in proton therapy and implementing corresponding adaptive dose optimization strategies. Specific objectives include:

1. **Objective 1:** Develop and validate synthetic image generation methods that produce anatomically and biologically plausible training datasets incorporating realistic tumor response patterns and normal tissue changes.

2. **Objective 2:** Build AI models that distinguish anatomy-driven from biology-driven image changes using multimodal features including population anatomy models, quantitative image biomarkers, radiomics features, segmentation, accumulated dose distributions, and uncertainty measures.

3. **Objective 3:** Design dose optimization algorithms that execute appropriate dose restoration, dose adaptation, or combined strategies based on the identified type of change.

4. **Objective 4:** Implement a proof-of-concept pipeline integrating response categorization and adaptive dose planning within a clinical treatment planning system and evaluate its performance using retrospective patient data.

1.4 Significance and Expected Impact

This research will contribute to the field of adaptive radiotherapy in several ways:

- **Scientific contribution:** First systematic approach to distinguish anatomical from biological changes in adaptive radiotherapy using AI.

- **Clinical impact:** Improved treatment outcomes through personalized dose adaptation strategies.

- **Efficiency gains:** Automated response characterization reducing manual planning burden.

- **RAPTORplus consortium:** Direct contribution to the EU doctoral network's mission of enabling right-time adaptive particle therapy.

2 Literature Review

2.1 Adaptive Proton Therapy: Current State

Adaptive proton therapy has evolved significantly over the past decade. Online adaptive proton therapy (OAPT) represents the state-of-the-art, where treatment plans are modified while the patient is on the treatment couch based on daily imaging [2]. Recent developments include PET-integrated systems that target biological changes rather than purely anatomical variations, advancing the concept of biology-driven adaptation [3].

Key challenges:

- Computational efficiency for real-time adaptation
- Uncertainty in dose calculation due to anatomical variations
- Integration of biological response information
- Clinical workflow disruption

2.2 AI in Radiation Oncology

Artificial intelligence has demonstrated transformative potential across all aspects of the radiotherapy workflow [4]:

1. **Segmentation:** Deep learning-based auto-segmentation achieves near-expert accuracy with dice similarity coefficients > 0.90 for most organs [5].
2. **Dose prediction:** CNNs can predict dose distributions from contours, achieving mean absolute errors $< 2\%$ of prescription dose [6].
3. **Image synthesis:** GANs and diffusion models generate synthetic CT from CBCT or MRI for dose calculation [7].
4. **Outcome prediction:** Radiomics and deep learning predict treatment response and toxicity [8].

2.3 Distinguishing Anatomical vs. Biological Changes

This represents an emerging research area with limited prior work:

Anatomical change prediction: Deep learning methods can predict anatomical changes with DSC > 0.94 for tumors in nasopharyngeal carcinoma [9], but these focus on geometry rather than biology.

Biological response modeling: Traditional approaches use empirical tumor control probability (TCP) and normal tissue complication probability (NTCP) models [10]. Recent work incorporates radiogenomics—linking imaging features to molecular biomarkers—to predict radioresistance [11].

Research gap: No existing methods systematically distinguish between anatomical and biological image changes for adaptive therapy decision-making.

2.4 Synthetic Medical Image Generation

Generating realistic synthetic training data is crucial for medical AI applications where labeled data is scarce:

- **GANs (Generative Adversarial Networks):** Widely used for medical image synthesis, including CT-to-MRI translation, dose prediction, and image denoising [12].
- **Diffusion Models:** Recently emerged as superior alternatives, producing higher quality and more diverse synthetic medical images [13].
- **Deformable Image Registration (DIR):** Combined with biomechanical models to generate plausible anatomical variations [14].

2.5 Radiomics and Image Biomarkers

Radiomics extracts quantitative features from medical images that may reflect underlying tumor biology:

- **Texture features:** Gray-level co-occurrence matrix (GLCM), gray-level run length matrix (GLRLM), and grey-level size zone matrix (GLSZM) features capture tissue heterogeneity [15].
- **Delta-radiomics:** Temporal changes in radiomic features during treatment predict response better than pre-treatment features alone [16].
- **Radiogenomics:** Radiomic features correlate with genomic biomarkers, potentially serving as non-invasive surrogates for molecular profiling [17].

2.6 Dose Optimization in Adaptive Radiotherapy

Dose optimization strategies vary based on the type of adaptation required:

- **Dose restoration:** Re-optimization to achieve original dose objectives on updated anatomy using identical constraints [18].
- **Dose escalation:** Increasing tumor dose when response is suboptimal or when normal tissue sparing allows [19].
- **Dose de-escalation:** Reducing dose to minimize toxicity when early response is favorable [20].
- **AI-guided optimization:** Using reinforcement learning for treatment planning decisions [21].

3 Methodology

This section details the methodological approach for each of the four main research tasks. The methodology follows a structured progression from data generation to clinical integration.

3.1 Task 1: Synthetic Image Generation

3.1.1 Objectives

Develop and validate methods to produce anatomically and biologically plausible synthetic training images that capture both geometric variations and biological response patterns.

3.1.2 Approach

We will implement a multi-method synthetic data generation pipeline combining three complementary approaches:

Method 1: Deformation-Based Anatomical Variation Generate anatomical variations using learned deformation fields from population data.

Mathematical Framework:

Let I_{ref} be a reference CT image. We model anatomical variations as:

$$I_{\text{anat}}(x) = I_{\text{ref}}(\phi_{\text{anat}}(x))$$

where $\phi_{\text{anat}} : \Omega \to \Omega$ is a deformation field sampled from a learned statistical model:

$$\phi_{\text{anat}} = \phi_{\text{identity}} + \sum_{i=1}^{K} w_i \phi_i$$

where $\{\phi_i\}_{i=1}^{K}$ are principal deformation modes from PCA on population registration data, and $w_i \sim \mathcal{N}(0, \lambda_i)$ are weights sampled from a Gaussian distribution with variance equal to the eigenvalues λ_i.

Implementation:

```python
import numpy as np
import torch
import torch.nn as nn
from scipy.ndimage import map_coordinates
from skimage.transform import resize
import SimpleITK as sitk

class DeformableAnatomicalGenerator:
    """
    Generate anatomical variations using learned deformation fields
    from population data using PCA-based statistical shape models.
    """

    def __init__(self, reference_image, n_components=10):
        """
        Args:
            reference_image: Reference CT image (numpy array)
            n_components: Number of principal components to retain
```

```
        """
        self.reference_image = reference_image
        self.n_components = n_components
        self.deformation_modes = None
        self.eigenvalues = None

    def fit_population_model(self, image_list, registration_method='demons'):
        """
        Learn principal deformation modes from population data.

        Args:
            image_list: List of CT images from different patients
            registration_method: Registration algorithm ('demons', 'bspline', etc.)
        """
        deformation_fields = []

        # Register all images to reference
        for img in image_list:
            dvf = self._register_images(
                self.reference_image,
                img,
                method=registration_method
            )
            deformation_fields.append(dvf)

        # Stack deformation fields: shape (N_patients, 3, D, H, W)
        dvf_matrix = np.stack(deformation_fields, axis=0)

        # Flatten spatial dimensions for PCA
        n_patients, n_dims, *spatial_dims = dvf_matrix.shape
        dvf_flat = dvf_matrix.reshape(n_patients, -1)

        # Perform PCA
        mean_dvf = np.mean(dvf_flat, axis=0)
        centered_dvf = dvf_flat - mean_dvf

        # Compute covariance matrix and eigendecomposition
        cov_matrix = (centered_dvf.T @ centered_dvf) / (n_patients - 1)
        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)

        # Sort by descending eigenvalues
        idx = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[idx]
        eigenvectors = eigenvectors[:, idx]

        # Keep top k components
        self.eigenvalues = eigenvalues[:self.n_components]
        self.deformation_modes = eigenvectors[:, :self.n_components]

        # Reshape modes back to spatial dimensions
        self.deformation_modes = self.deformation_modes.T.reshape(
            self.n_components, n_dims, *spatial_dims
        )

        self.mean_dvf = mean_dvf.reshape(n_dims, *spatial_dims)

```

```
        print(f"Variance explained: {np.sum(self.eigenvalues) / np.sum(eigenvalues):.2%}")

    def _register_images(self, fixed_img, moving_img, method='demons'):
        """
        Register moving image to fixed image using specified method.
        Returns deformation vector field.
        """
        fixed = sitk.GetImageFromArray(fixed_img.astype(np.float32))
        moving = sitk.GetImageFromArray(moving_img.astype(np.float32))

        if method == 'demons':
            demons = sitk.DemonsRegistrationFilter()
            demons.SetNumberOfIterations(50)
            demons.SetStandardDeviations(1.0)

            displacementField = demons.Execute(fixed, moving)

        elif method == 'bspline':
            # B-spline registration
            registration = sitk.ImageRegistrationMethod()
            registration.SetMetricAsMattesMutualInformation(numberOfHistogramBins=50)
            registration.SetOptimizerAsGradientDescentLineSearch(
                learningRate=1.0,
                numberOfIterations=100
            )

            transform = sitk.BSplineTransformInitializer(
                fixed,
                transformDomainMeshSize=[8]*3
            )

            registration.SetInitialTransform(transform)
            final_transform = registration.Execute(fixed, moving)
            displacementField = sitk.TransformToDisplacementField(
                final_transform,
                sitk.sitkVectorFloat64,
                fixed.GetSize()
            )

        # Convert to numpy array
        dvf = sitk.GetArrayFromImage(displacementField)
        # Rearrange dimensions: (D, H, W, 3) -> (3, D, H, W)
        dvf = np.transpose(dvf, (3, 0, 1, 2))

        return dvf

    def generate_sample(self, variation_scale=1.0):
        """
        Generate a synthetic image with anatomical variation.

        Args:
            variation_scale: Scale factor for variation magnitude

        Returns:
            Synthetic image with anatomical variation
        """
```

```
        if self.deformation_modes is None:
            raise ValueError("Must fit population model first")

        # Sample weights from Gaussian distribution
        weights = np.random.randn(self.n_components) * np.sqrt(self.
    eigenvalues)
        weights *= variation_scale

        # Construct deformation field
        dvf = self.mean_dvf.copy()
        for i in range(self.n_components):
            dvf += weights[i] * self.deformation_modes[i]

        # Apply deformation to reference image
        synthetic_image = self._apply_deformation(self.reference_image,
    dvf)

        return synthetic_image, dvf

    def _apply_deformation(self, image, dvf):
        """
        Apply deformation vector field to image.

        Args:
            image: Original image (D, H, W)
            dvf: Deformation vector field (3, D, H, W)

        Returns:
            Deformed image
        """
        # Create coordinate grid
        dims = image.shape
        coords = np.meshgrid(
            np.arange(dims[0]),
            np.arange(dims[1]),
            np.arange(dims[2]),
            indexing='ij'
        )
        coords = np.stack(coords, axis=0)  # (3, D, H, W)

        # Add deformation
        deformed_coords = coords + dvf

        # Interpolate
        deformed_image = map_coordinates(
            image,
            [deformed_coords[0].ravel(),
             deformed_coords[1].ravel(),
             deformed_coords[2].ravel()],
            order=3,
            mode='nearest'
        ).reshape(dims)

        return deformed_image

# Example usage
if __name__ == "__main__":
    # Load reference image and population
```

```
    reference_image = np.load("reference_ct.npy")
    population_images = [np.load(f"patient_{i}_ct.npy") for i in range(50)]

    # Create generator
    generator = DeformableAnatomicalGenerator(
        reference_image,
        n_components=15
    )

    # Fit population model
    generator.fit_population_model(population_images)

    # Generate synthetic samples
    synthetic_image, dvf = generator.generate_sample(variation_scale=1.0)

    print(f"Generated synthetic image with shape: {synthetic_image.shape}")
```

Listing 1: Deformation-Based Anatomical Variation Generation

Method 2: Diffusion-Based Biological Response Generation Generate tumor response patterns using conditional diffusion models.

Mathematical Framework:

Diffusion models work by gradually adding noise to data, then learning to reverse this process:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\bar{\alpha}_t}x_0, (1 - \bar{\alpha}_t)I)$$

The reverse process is modeled by a neural network ϵ_θ that predicts the noise:

$$p_\theta(x_{t-1}|x_t, c) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t, c), \Sigma_\theta(x_t, t, c))$$

where c represents conditioning information (baseline image, dose, time point).

Implementation:

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

class SinusoidalPositionEmbeddings(nn.Module):
    """Positional encoding for timestep in diffusion model."""

    def __init__(self, dim):
        super().__init__()
        self.dim = dim

    def forward(self, time):
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
```

```
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings


class ConditionalUNet3D(nn.Module):
    """
    3D U-Net for conditional diffusion model.
    Conditions on baseline CT, accumulated dose, and treatment time.
    """

    def __init__(self, in_channels=1, out_channels=1,
                 time_emb_dim=256, condition_channels=2):
        super().__init__()

        self.time_emb_dim = time_emb_dim

        # Time embedding
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(time_emb_dim),
            nn.Linear(time_emb_dim, time_emb_dim * 4),
            nn.GELU(),
            nn.Linear(time_emb_dim * 4, time_emb_dim),
        )

        # Encoder (downsampling)
        self.enc1 = self._make_layer(in_channels + condition_channels, 64, time_emb_dim)
        self.pool1 = nn.MaxPool3d(2)

        self.enc2 = self._make_layer(64, 128, time_emb_dim)
        self.pool2 = nn.MaxPool3d(2)

        self.enc3 = self._make_layer(128, 256, time_emb_dim)
        self.pool3 = nn.MaxPool3d(2)

        self.enc4 = self._make_layer(256, 512, time_emb_dim)
        self.pool4 = nn.MaxPool3d(2)

        # Bottleneck
        self.bottleneck = self._make_layer(512, 1024, time_emb_dim)

        # Decoder (upsampling)
        self.upconv4 = nn.ConvTranspose3d(1024, 512, 2, stride=2)
        self.dec4 = self._make_layer(1024, 512, time_emb_dim)

        self.upconv3 = nn.ConvTranspose3d(512, 256, 2, stride=2)
        self.dec3 = self._make_layer(512, 256, time_emb_dim)

        self.upconv2 = nn.ConvTranspose3d(256, 128, 2, stride=2)
        self.dec2 = self._make_layer(256, 128, time_emb_dim)

        self.upconv1 = nn.ConvTranspose3d(128, 64, 2, stride=2)
        self.dec1 = self._make_layer(128, 64, time_emb_dim)

        # Output
        self.out = nn.Conv3d(64, out_channels, 1)

```

```
    def _make_layer(self, in_ch, out_ch, time_emb_dim):
        """Create a residual block with time embedding."""
        return ResidualBlock3D(in_ch, out_ch, time_emb_dim)

    def forward(self, x, t, condition):
        """
        Args:
            x: Noisy image at timestep t, shape (B, 1, D, H, W)
            t: Timestep, shape (B,)
            condition: Conditioning information (baseline CT + dose),
                       shape (B, condition_channels, D, H, W)
        """
        # Time embedding
        t_emb = self.time_mlp(t)

        # Concatenate input with condition
        x = torch.cat([x, condition], dim=1)

        # Encoder
        e1 = self.enc1(x, t_emb)
        e2 = self.enc2(self.pool1(e1), t_emb)
        e3 = self.enc3(self.pool2(e2), t_emb)
        e4 = self.enc4(self.pool3(e3), t_emb)

        # Bottleneck
        b = self.bottleneck(self.pool4(e4), t_emb)

        # Decoder with skip connections
        d4 = self.dec4(torch.cat([self.upconv4(b), e4], dim=1), t_emb)
        d3 = self.dec3(torch.cat([self.upconv3(d4), e3], dim=1), t_emb)
        d2 = self.dec2(torch.cat([self.upconv2(d3), e2], dim=1), t_emb)
        d1 = self.dec1(torch.cat([self.upconv1(d2), e1], dim=1), t_emb)

        # Output
        return self.out(d1)


class ResidualBlock3D(nn.Module):
    """Residual block with time embedding for 3D U-Net."""

    def __init__(self, in_channels, out_channels, time_emb_dim):
        super().__init__()

        self.time_mlp = nn.Sequential(
            nn.SiLU(),
            nn.Linear(time_emb_dim, out_channels)
        )

        self.conv1 = nn.Conv3d(in_channels, out_channels, 3, padding=1)
        self.conv2 = nn.Conv3d(out_channels, out_channels, 3, padding=1)

        self.norm1 = nn.GroupNorm(8, out_channels)
        self.norm2 = nn.GroupNorm(8, out_channels)

        self.act = nn.SiLU()

        # Residual connection
        if in_channels != out_channels:
```

```
            self.residual = nn.Conv3d(in_channels, out_channels, 1)
        else:
            self.residual = nn.Identity()

    def forward(self, x, t_emb):
        residual = self.residual(x)

        # First convolution
        h = self.conv1(x)
        h = self.norm1(h)

        # Add time embedding
        time_emb = self.time_mlp(t_emb)
        h = h + time_emb[..., None, None, None]

        h = self.act(h)

        # Second convolution
        h = self.conv2(h)
        h = self.norm2(h)

        # Residual connection
        return self.act(h + residual)


class ConditionalDDPM:
    """
    Conditional Denoising Diffusion Probabilistic Model for
    generating biological response images.
    """

    def __init__(self, model, timesteps=1000, beta_start=1e-4, beta_end
    =0.02):
        self.model = model
        self.timesteps = timesteps

        # Define beta schedule (linear)
        self.betas = torch.linspace(beta_start, beta_end, timesteps)
        self.alphas = 1. - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1], (1,
    0), value=1.0)

        # Calculations for diffusion q(x_t | x_{t-1})
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1. - self.
    alphas_cumprod)

        # Calculations for posterior q(x_{t-1} | x_t, x_0)
        self.posterior_variance = (
            self.betas * (1. - self.alphas_cumprod_prev) / (1. - self.
    alphas_cumprod)
        )

    def q_sample(self, x_start, t, noise=None):
        """
        Forward diffusion process: add noise to x_start.

```

```
        Args:
            x_start: Original image
            t: Timestep
            noise: Noise to add (if None, sample from N(0,1))
        """
        if noise is None:
            noise = torch.randn_like(x_start)

        sqrt_alphas_cumprod_t = self._extract(self.sqrt_alphas_cumprod,
    t, x_start.shape)
        sqrt_one_minus_alphas_cumprod_t = self._extract(
            self.sqrt_one_minus_alphas_cumprod, t, x_start.shape
        )

        return sqrt_alphas_cumprod_t * x_start +
    sqrt_one_minus_alphas_cumprod_t * noise

    def p_sample(self, x, t, condition):
        """
        Reverse diffusion process: denoise x at timestep t.

        Args:
            x: Noisy image at timestep t
            t: Current timestep
            condition: Conditioning information
        """
        # Predict noise
        pred_noise = self.model(x, t, condition)

        # Get parameters
        betas_t = self._extract(self.betas, t, x.shape)
        sqrt_one_minus_alphas_cumprod_t = self._extract(
            self.sqrt_one_minus_alphas_cumprod, t, x.shape
        )
        sqrt_recip_alphas_t = self._extract(torch.sqrt(1.0 / self.alphas
    ), t, x.shape)

        # Predict x_0
        model_mean = sqrt_recip_alphas_t * (
            x - betas_t * pred_noise / sqrt_one_minus_alphas_cumprod_t
        )

        if t[0] == 0:
            return model_mean
        else:
            posterior_variance_t = self._extract(self.posterior_variance
    , t, x.shape)
            noise = torch.randn_like(x)
            return model_mean + torch.sqrt(posterior_variance_t) * noise

    def sample(self, shape, condition, device='cuda'):
        """
        Generate synthetic image by sampling from the model.

        Args:
            shape: Shape of image to generate
            condition: Conditioning information (baseline CT + dose)
            device: Device to run on
```

```
        """
        # Start from pure noise
        x = torch.randn(shape, device=device)

        # Iteratively denoise
        for t in reversed(range(self.timesteps)):
            t_batch = torch.full((shape[0],), t, device=device, dtype=
    torch.long)
            x = self.p_sample(x, t_batch, condition)

        return x

    def training_loss(self, x_start, condition):
        """
        Calculate training loss (simple MSE loss on noise prediction).

        Args:
            x_start: Ground truth image (e.g., follow-up CT)
            condition: Conditioning information
        """
        batch_size = x_start.shape[0]
        device = x_start.device

        # Sample random timesteps
        t = torch.randint(0, self.timesteps, (batch_size,), device=
    device).long()

        # Sample noise
        noise = torch.randn_like(x_start)

        # Add noise to x_start
        x_noisy = self.q_sample(x_start, t, noise)

        # Predict noise
        pred_noise = self.model(x_noisy, t, condition)

        # Calculate loss
        loss = F.mse_loss(pred_noise, noise)

        return loss

    @staticmethod
    def _extract(a, t, x_shape):
        """Extract coefficients at specified timesteps."""
        batch_size = t.shape[0]
        out = a.gather(-1, t)
        return out.reshape(batch_size, *((1,) * (len(x_shape) - 1)))


# Training function
def train_diffusion_model(model, ddpm, train_loader, epochs=100, lr=1e
    -4):
    """
    Train conditional diffusion model for biological response generation
    .

    Args:
        model: ConditionalUNet3D model
```

```
        ddpm: ConditionalDDPM instance
        train_loader: DataLoader with (baseline, followup, dose) pairs
        epochs: Number of training epochs
        lr: Learning rate
    """
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
    device = next(model.parameters()).device

    for epoch in range(epochs):
        model.train()
        total_loss = 0

        for batch_idx, (baseline, followup, dose) in enumerate(
    train_loader):
            baseline = baseline.to(device)
            followup = followup.to(device)
            dose = dose.to(device)

            # Prepare conditioning (baseline CT + dose)
            condition = torch.cat([baseline, dose], dim=1)

            # Calculate loss
            loss = ddpm.training_loss(followup, condition)

            # Optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

    return model


# Example usage
if __name__ == "__main__":
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'
    )

    # Initialize model
    model = ConditionalUNet3D(
        in_channels=1,
        out_channels=1,
        condition_channels=2  # baseline CT + dose
    ).to(device)

    # Initialize DDPM
    ddpm = ConditionalDDPM(model, timesteps=1000)

    # Generate synthetic follow-up image
    batch_size = 1
    image_shape = (batch_size, 1, 128, 128, 128)

    # Create dummy conditioning data
    baseline_ct = torch.randn(batch_size, 1, 128, 128, 128, device=
```

```
    device)
    dose = torch.randn(batch_size, 1, 128, 128, 128, device=device)
    condition = torch.cat([baseline_ct, dose], dim=1)

    # Sample synthetic image
    synthetic_followup = ddpm.sample(image_shape, condition, device)

    print(f"Generated synthetic follow-up CT with shape: {
    synthetic_followup.shape}")
```

Listing 2: Conditional Diffusion Model for Biological Response

Method 3: GAN-Based Image-to-Image Translation Alternative approach using conditional GANs for paired image synthesis.

```
import torch
import torch.nn as nn

class Generator3D(nn.Module):
    """
    3D U-Net Generator for conditional GAN.
    Generates follow-up CT from baseline CT and dose distribution.
    """

    def __init__(self, in_channels=2, out_channels=1, features=64):
        super().__init__()

        # Encoder
        self.enc1 = self._block(in_channels, features)
        self.enc2 = self._block(features, features * 2)
        self.enc3 = self._block(features * 2, features * 4)
        self.enc4 = self._block(features * 4, features * 8)

        self.pool = nn.MaxPool3d(kernel_size=2, stride=2)

        # Bottleneck
        self.bottleneck = self._block(features * 8, features * 16)

        # Decoder
        self.upconv4 = nn.ConvTranspose3d(features * 16, features * 8,
    2, 2)
        self.dec4 = self._block(features * 16, features * 8)

        self.upconv3 = nn.ConvTranspose3d(features * 8, features * 4, 2,
     2)
        self.dec3 = self._block(features * 8, features * 4)

        self.upconv2 = nn.ConvTranspose3d(features * 4, features * 2, 2,
     2)
        self.dec2 = self._block(features * 4, features * 2)

        self.upconv1 = nn.ConvTranspose3d(features * 2, features, 2, 2)
        self.dec1 = self._block(features * 2, features)

        self.out = nn.Conv3d(features, out_channels, kernel_size=1)

    def _block(self, in_channels, out_channels):
        return nn.Sequential(
```

```
            nn.Conv3d(in_channels, out_channels, 3, padding=1),
            nn.InstanceNorm3d(out_channels),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv3d(out_channels, out_channels, 3, padding=1),
            nn.InstanceNorm3d(out_channels),
            nn.LeakyReLU(0.2, inplace=True),
        )

    def forward(self, x):
        """
        Args:
            x: Concatenated baseline CT and dose, shape (B, 2, D, H, W)
        """
        # Encoder with skip connections
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))
        e4 = self.enc4(self.pool(e3))

        # Bottleneck
        b = self.bottleneck(self.pool(e4))

        # Decoder
        d4 = self.dec4(torch.cat([self.upconv4(b), e4], dim=1))
        d3 = self.dec3(torch.cat([self.upconv3(d4), e3], dim=1))
        d2 = self.dec2(torch.cat([self.upconv2(d3), e2], dim=1))
        d1 = self.dec1(torch.cat([self.upconv1(d2), e1], dim=1))

        return torch.tanh(self.out(d1))


class Discriminator3D(nn.Module):
    """
    3D PatchGAN Discriminator.
    Classifies whether image pairs are real or generated.
    """

    def __init__(self, in_channels=3, features=[64, 128, 256, 512]):
        super().__init__()

        layers = []
        in_ch = in_channels

        for idx, feature in enumerate(features):
            layers.append(
                nn.Conv3d(
                    in_ch,
                    feature,
                    kernel_size=4,
                    stride=2,
                    padding=1,
                    bias=False if idx > 0 else True
                )
            )
            if idx > 0:
                layers.append(nn.InstanceNorm3d(feature))
            layers.append(nn.LeakyReLU(0.2, inplace=True))
            in_ch = feature
```

```
        # Final layer
        layers.append(
            nn.Conv3d(in_ch, 1, kernel_size=4, stride=1, padding=1)
        )

        self.model = nn.Sequential(*layers)

    def forward(self, x, y):
        """
        Args:
            x: Condition (baseline CT + dose), shape (B, 2, D, H, W)
            y: Target/generated follow-up CT, shape (B, 1, D, H, W)
        """
        # Concatenate condition and target
        input_pair = torch.cat([x, y], dim=1)
        return self.model(input_pair)


def train_gan(generator, discriminator, train_loader,
              epochs=100, lr_g=2e-4, lr_d=2e-4, lambda_l1=100):
    """
    Train conditional GAN for response image generation.

    Args:
        generator: Generator network
        discriminator: Discriminator network
        train_loader: DataLoader with (baseline, dose, followup) triplets
        epochs: Number of training epochs
        lr_g: Generator learning rate
        lr_d: Discriminator learning rate
        lambda_l1: Weight for L1 loss
    """
    device = next(generator.parameters()).device

    # Optimizers
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr_g, betas=(0.5, 0.999))
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr_d, betas=(0.5, 0.999))

    # Loss functions
    criterion_gan = nn.BCEWithLogitsLoss()
    criterion_l1 = nn.L1Loss()

    for epoch in range(epochs):
        generator.train()
        discriminator.train()

        g_loss_total = 0
        d_loss_total = 0

        for batch_idx, (baseline, dose, followup_real) in enumerate(train_loader):
            baseline = baseline.to(device)
            dose = dose.to(device)
            followup_real = followup_real.to(device)
```

```
            # Prepare conditioning
            condition = torch.cat([baseline, dose], dim=1)

            # ==================
            # Train Discriminator
            # ==================
            opt_d.zero_grad()

            # Generate fake images
            followup_fake = generator(condition)

            # Real and fake predictions
            pred_real = discriminator(condition, followup_real)
            pred_fake = discriminator(condition, followup_fake.detach())

            # Labels
            real_labels = torch.ones_like(pred_real)
            fake_labels = torch.zeros_like(pred_fake)

            # Discriminator loss
            loss_d_real = criterion_gan(pred_real, real_labels)
            loss_d_fake = criterion_gan(pred_fake, fake_labels)
            loss_d = (loss_d_real + loss_d_fake) * 0.5

            loss_d.backward()
            opt_d.step()

            # ==================
            # Train Generator
            # ==================
            opt_g.zero_grad()

            # Generate fake images
            followup_fake = generator(condition)

            # Fool discriminator
            pred_fake = discriminator(condition, followup_fake)
            loss_g_gan = criterion_gan(pred_fake, real_labels)

            # L1 loss for image similarity
            loss_g_l1 = criterion_l1(followup_fake, followup_real)

            # Total generator loss
            loss_g = loss_g_gan + lambda_l1 * loss_g_l1

            loss_g.backward()
            opt_g.step()

            g_loss_total += loss_g.item()
            d_loss_total += loss_d.item()

        avg_g_loss = g_loss_total / len(train_loader)
        avg_d_loss = d_loss_total / len(train_loader)

        print(f"Epoch {epoch+1}/{epochs}, "
              f"G_loss: {avg_g_loss:.4f}, D_loss: {avg_d_loss:.4f}")
```

```
    return generator, discriminator


# Example usage
if __name__ == "__main__":
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Initialize networks
    generator = Generator3D(in_channels=2, out_channels=1).to(device)
    discriminator = Discriminator3D(in_channels=3).to(device)

    print("Generator parameters:", sum(p.numel() for p in generator.parameters()))
    print("Discriminator parameters:", sum(p.numel() for p in discriminator.parameters()))
```

Listing 3: Conditional GAN for Response Image Generation

3.1.3 Validation Strategy

Validate synthetic images using:

1. **Visual Turing Test:** Expert radiation oncologists assess realism.
2. **Quantitative metrics:**
   - Structural Similarity Index (SSIM)
   - Peak Signal-to-Noise Ratio (PSNR)
   - Fréchet Inception Distance (FID) adapted for medical images
3. **Physical plausibility:** Verify HU value distributions, anatomical constraints
4. **Downstream task performance:** Train response classifier on synthetic data, test on real data

3.2 Task 2: AI-Based Response Characterization

3.2.1 Objectives

Build models that distinguish anatomy-driven from biology-driven image changes using multimodal features.

3.2.2 Feature Engineering

We will extract and integrate multiple feature modalities:

```
import numpy as np
import SimpleITK as sitk
from radiomics import featureextractor
import pandas as pd

class RadiomicFeatureExtractor:
    """
```

```
    Extract radiomic features from CT images and regions of interest.
    Focuses on features sensitive to biological changes (texture, intensity).
    """

    def __init__(self, settings=None):
        if settings is None:
            # Default PyRadiomics settings
            settings = {
                'binWidth': 25,
                'interpolator': 'sitkBSpline',
                'resampledPixelSpacing': [1, 1, 1],
                'normalize': True,
                'normalizeScale': 100,
            }

        self.extractor = featureextractor.RadiomicsFeatureExtractor(**settings)

        # Enable feature classes
        self.extractor.enableImageTypeByName('Original')
        self.extractor.enableImageTypeByName('Wavelet')
        self.extractor.enableFeatureClassByName('firstorder')
        self.extractor.enableFeatureClassByName('glcm')
        self.extractor.enableFeatureClassByName('glrlm')
        self.extractor.enableFeatureClassByName('glszm')
        self.extractor.enableFeatureClassByName('gldm')
        self.extractor.enableFeatureClassByName('ngtdm')

    def extract_features(self, image, mask):
        """
        Extract radiomic features from image within mask region.

        Args:
            image: SimpleITK image or numpy array
            mask: SimpleITK mask or numpy array (binary)

        Returns:
            Dictionary of feature values
        """
        # Convert to SimpleITK if necessary
        if isinstance(image, np.ndarray):
            image = sitk.GetImageFromArray(image.astype(np.float32))
        if isinstance(mask, np.ndarray):
            mask = sitk.GetImageFromArray(mask.astype(np.uint8))

        # Extract features
        features = self.extractor.execute(image, mask)

        # Filter to only feature values (remove diagnostics)
        feature_dict = {
            key: val for key, val in features.items()
            if not key.startswith('diagnostics_')
        }

        return feature_dict

```

```
    def extract_delta_features(self, image_baseline, image_followup, mask):
        """
        Extract delta-radiomic features (temporal changes).

        Args:
            image_baseline: Baseline CT image
            image_followup: Follow-up CT image
            mask: ROI mask

        Returns:
            Dictionary of delta features (absolute and relative changes)
        """
        # Extract features from both timepoints
        features_baseline = self.extract_features(image_baseline, mask)
        features_followup = self.extract_features(image_followup, mask)

        # Calculate delta features
        delta_features = {}

        for key in features_baseline.keys():
            if key in features_followup:
                baseline_val = features_baseline[key]
                followup_val = features_followup[key]

                # Absolute change
                delta_features[f'delta_abs_{key}'] = followup_val - baseline_val

                # Relative change (%)
                if abs(baseline_val) > 1e-6:
                    delta_features[f'delta_rel_{key}'] = (
                        (followup_val - baseline_val) / baseline_val * 100
                    )

        return delta_features

    def extract_multiregion_features(self, image, tumor_mask,
                                     organ_masks_dict):
        """
        Extract features from multiple regions (tumor + organs at risk).

        Args:
            image: CT image
            tumor_mask: Tumor segmentation mask
            organ_masks_dict: Dictionary of organ masks {organ_name: mask}

        Returns:
            DataFrame with features for all regions
        """
        all_features = {}

        # Tumor features
        tumor_features = self.extract_features(image, tumor_mask)
        for key, val in tumor_features.items():
            all_features[f'tumor_{key}'] = val
```

```
        # Organ features
        for organ_name, organ_mask in organ_masks_dict.items():
            organ_features = self.extract_features(image, organ_mask)
            for key, val in organ_features.items():
                all_features[f'{organ_name}_{key}'] = val

        return pd.Series(all_features)


class BiologicalResponseFeatures:
    """
    Extract features specifically designed to capture biological response.
    Includes volume changes, density changes, and spatial pattern changes.
    """

    @staticmethod
    def volume_change(mask_baseline, mask_followup, voxel_spacing):
        """
        Calculate volume change between baseline and follow-up.

        Args:
            mask_baseline: Baseline segmentation mask
            mask_followup: Follow-up segmentation mask
            voxel_spacing: Tuple of (x, y, z) voxel spacing in mm

        Returns:
            Dictionary with absolute and relative volume changes
        """
        voxel_volume = np.prod(voxel_spacing)  # mm^3

        vol_baseline = np.sum(mask_baseline) * voxel_volume / 1000  # cm^3
        vol_followup = np.sum(mask_followup) * voxel_volume / 1000  # cm^3

        abs_change = vol_followup - vol_baseline
        rel_change = (abs_change / vol_baseline * 100) if vol_baseline > 0 else 0

        return {
            'volume_baseline_cm3': vol_baseline,
            'volume_followup_cm3': vol_followup,
            'volume_change_abs_cm3': abs_change,
            'volume_change_rel_percent': rel_change
        }

    @staticmethod
    def density_change(image_baseline, image_followup, mask):
        """
        Calculate changes in tissue density (HU values).

        Args:
            image_baseline: Baseline CT image
            image_followup: Follow-up CT image
            mask: ROI mask
```

```
        Returns:
            Dictionary with mean, median, and std of HU changes
        """
        roi_baseline = image_baseline[mask > 0]
        roi_followup = image_followup[mask > 0]

        mean_change = np.mean(roi_followup) - np.mean(roi_baseline)
        median_change = np.median(roi_followup) - np.median(roi_baseline)
        std_change = np.std(roi_followup) - np.std(roi_baseline)

        return {
            'density_mean_change_HU': mean_change,
            'density_median_change_HU': median_change,
            'density_std_change_HU': std_change,
            'density_mean_baseline_HU': np.mean(roi_baseline),
            'density_mean_followup_HU': np.mean(roi_followup),
        }

    @staticmethod
    def heterogeneity_change(image_baseline, image_followup, mask):
        """
        Calculate changes in tumor heterogeneity.

        Args:
            image_baseline: Baseline CT image
            image_followup: Follow-up CT image
            mask: ROI mask

        Returns:
            Dictionary with heterogeneity metrics
        """
        roi_baseline = image_baseline[mask > 0]
        roi_followup = image_followup[mask > 0]

        # Coefficient of variation
        cv_baseline = np.std(roi_baseline) / np.mean(roi_baseline) if np.mean(roi_baseline) != 0 else 0
        cv_followup = np.std(roi_followup) / np.mean(roi_followup) if np.mean(roi_followup) != 0 else 0

        # Entropy
        def calculate_entropy(data, bins=50):
            hist, _ = np.histogram(data, bins=bins, density=True)
            hist = hist[hist > 0]
            return -np.sum(hist * np.log2(hist))

        entropy_baseline = calculate_entropy(roi_baseline)
        entropy_followup = calculate_entropy(roi_followup)

        return {
            'heterogeneity_cv_baseline': cv_baseline,
            'heterogeneity_cv_followup': cv_followup,
            'heterogeneity_cv_change': cv_followup - cv_baseline,
            'heterogeneity_entropy_baseline': entropy_baseline,
            'heterogeneity_entropy_followup': entropy_followup,
```

```
            'heterogeneity_entropy_change': entropy_followup - entropy_baseline,
        }

    @staticmethod
    def spatial_concordance(mask_baseline, mask_followup):
        """
        Calculate spatial overlap metrics to distinguish rigid motion from shape change.

        Args:
            mask_baseline: Baseline segmentation mask
            mask_followup: Follow-up segmentation mask (already registered)

        Returns:
            Dictionary with overlap metrics
        """
        intersection = np.sum((mask_baseline > 0) & (mask_followup > 0))
        union = np.sum((mask_baseline > 0) | (mask_followup > 0))

        dice = 2 * intersection / (np.sum(mask_baseline > 0) + np.sum(mask_followup > 0))
        jaccard = intersection / union if union > 0 else 0

        # Hausdorff distance (simplified - use full implementation for production)
        # This would require distance transforms

        return {
            'spatial_dice': dice,
            'spatial_jaccard': jaccard,
            'spatial_overlap_percent': intersection / np.sum(mask_baseline > 0) * 100
        }


# Example usage
if __name__ == "__main__":
    # Load images
    baseline_ct = np.load("baseline_ct.npy")
    followup_ct = np.load("followup_ct.npy")
    tumor_mask = np.load("tumor_mask.npy")

    # Extract radiomic features
    radiomics_extractor = RadiomicFeatureExtractor()

    # Delta-radiomic features
    delta_features = radiomics_extractor.extract_delta_features(
        baseline_ct, followup_ct, tumor_mask
    )

    # Biological response features
    bio_features = {}
    bio_features.update(BiologicalResponseFeatures.volume_change(
        tumor_mask, tumor_mask, voxel_spacing=(1.0, 1.0, 3.0)
    ))
    bio_features.update(BiologicalResponseFeatures.density_change(
```

```
        baseline_ct, followup_ct, tumor_mask
    ))
    bio_features.update(BiologicalResponseFeatures.heterogeneity_change(
        baseline_ct, followup_ct, tumor_mask
    ))

    print(f"Extracted {len(delta_features)} delta-radiomic features")
    print(f"Extracted {len(bio_features)} biological response features")
```

Listing 4: Radiomic Feature Extraction

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

class MultimodalResponseClassifier(nn.Module):
    """
    Deep learning model to classify image changes as anatomical vs biological.
    Integrates multiple input modalities:
    - Baseline CT
    - Follow-up CT
    - Dose distribution
    - Deformation vector field
    - Radiomic features
    """

    def __init__(self, num_radiomics=100, dropout=0.3):
        super().__init__()

        # 3D CNN for image feature extraction
        self.image_encoder = Image3DEncoder(in_channels=4)  # baseline, followup, dose, DVF

        # MLP for radiomic features
        self.radiomics_encoder = nn.Sequential(
            nn.Linear(num_radiomics, 256),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(dropout),
        )

        # Fusion and classification
        self.fusion = nn.Sequential(
            nn.Linear(512 + 128, 256),  # 512 from image encoder, 128 from radiomics
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(128, 3)  # 3 classes: anatomical, biological, mixed
        )
```

```
        # Uncertainty estimation (evidential deep learning)
        self.uncertainty_head = nn.Linear(128, 3)

    def forward(self, baseline, followup, dose, dvf, radiomics):
        """
        Args:
            baseline: Baseline CT, shape (B, 1, D, H, W)
            followup: Follow-up CT, shape (B, 1, D, H, W)
            dose: Accumulated dose, shape (B, 1, D, H, W)
            dvf: Deformation vector field magnitude, shape (B, 1, D, H, W)
            radiomics: Radiomic features, shape (B, num_radiomics)

        Returns:
            logits: Classification logits, shape (B, 3)
            uncertainty: Uncertainty estimate, shape (B, 3)
        """
        # Concatenate imaging inputs
        image_input = torch.cat([baseline, followup, dose, dvf], dim=1)

        # Extract image features
        image_features = self.image_encoder(image_input)

        # Extract radiomic features
        radio_features = self.radiomics_encoder(radiomics)

        # Fuse features
        combined = torch.cat([image_features, radio_features], dim=1)

        # Classification
        logits = self.fusion(combined)

        # Uncertainty estimation
        uncertainty = F.softplus(self.uncertainty_head(combined))

        return logits, uncertainty


class Image3DEncoder(nn.Module):
    """
    3D CNN encoder for extracting features from multi-channel 3D images.
    """

    def __init__(self, in_channels=4):
        super().__init__()

        self.conv1 = self._conv_block(in_channels, 32)
        self.conv2 = self._conv_block(32, 64)
        self.conv3 = self._conv_block(64, 128)
        self.conv4 = self._conv_block(128, 256)
        self.conv5 = self._conv_block(256, 512)

        self.pool = nn.MaxPool3d(2)
        self.adaptive_pool = nn.AdaptiveAvgPool3d(1)

    def _conv_block(self, in_ch, out_ch):
        return nn.Sequential(
```

```
            nn.Conv3d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm3d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv3d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm3d(out_ch),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(self.pool(x1))
        x3 = self.conv3(self.pool(x2))
        x4 = self.conv4(self.pool(x3))
        x5 = self.conv5(self.pool(x4))

        # Global average pooling
        x_pooled = self.adaptive_pool(x5)
        x_flat = x_pooled.view(x_pooled.size(0), -1)

        return x_flat


class EvidentialLoss(nn.Module):
    """
    Evidential deep learning loss for uncertainty-aware classification.
    Based on: Sensoy et al., "Evidential Deep Learning to Quantify
    Classification Uncertainty"
    """

    def __init__(self, num_classes=3, lambda_reg=0.01):
        super().__init__()
        self.num_classes = num_classes
        self.lambda_reg = lambda_reg

    def forward(self, evidence, target):
        """
        Args:
            evidence: Evidence values from model, shape (B, num_classes)
            target: Ground truth labels, shape (B,)

        Returns:
            loss: Total loss (classification + uncertainty
    regularization)
        """
        # Convert to Dirichlet parameters
        alpha = evidence + 1
        S = torch.sum(alpha, dim=1, keepdim=True)

        # Expected probability
        prob = alpha / S

        # One-hot encode targets
        target_one_hot = F.one_hot(target, self.num_classes).float()

        # Classification loss (cross-entropy with Dirichlet)
        A = torch.sum(target_one_hot * (torch.digamma(S) - torch.digamma
    (alpha)), dim=1)

```

```
        # KL divergence regularization
        alpha_tilde = target_one_hot + (1 - target_one_hot) * alpha
        S_tilde = torch.sum(alpha_tilde, dim=1, keepdim=True)

        kl_div = torch.lgamma(S_tilde) - torch.sum(torch.lgamma(alpha_tilde), dim=1) + \
                 torch.sum((alpha_tilde - 1) * (torch.digamma(alpha_tilde) - torch.digamma(S_tilde)), dim=1)

        loss = torch.mean(A + self.lambda_reg * kl_div)

        return loss


def train_response_classifier(model, train_loader, val_loader,
                              epochs=100, lr=1e-4):
    """
    Train multimodal response classifier with uncertainty quantification.

    Args:
        model: MultimodalResponseClassifier model
        train_loader: Training data loader
        val_loader: Validation data loader
        epochs: Number of training epochs
        lr: Learning rate
    """
    device = next(model.parameters()).device
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=1e-5)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, epochs)
    criterion = EvidentialLoss(num_classes=3)

    best_val_acc = 0.0

    for epoch in range(epochs):
        # Training
        model.train()
        train_loss = 0
        train_correct = 0
        train_total = 0

        for batch in train_loader:
            baseline = batch['baseline'].to(device)
            followup = batch['followup'].to(device)
            dose = batch['dose'].to(device)
            dvf = batch['dvf'].to(device)
            radiomics = batch['radiomics'].to(device)
            labels = batch['label'].to(device)

            # Forward pass
            logits, uncertainty = model(baseline, followup, dose, dvf, radiomics)

            # Loss
            loss = criterion(uncertainty, labels)

```

```
            # Backward pass
            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()

            # Metrics
            train_loss += loss.item()
            predictions = torch.argmax(logits, dim=1)
            train_correct += (predictions == labels).sum().item()
            train_total += labels.size(0)

        # Validation
        model.eval()
        val_loss = 0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for batch in val_loader:
                baseline = batch['baseline'].to(device)
                followup = batch['followup'].to(device)
                dose = batch['dose'].to(device)
                dvf = batch['dvf'].to(device)
                radiomics = batch['radiomics'].to(device)
                labels = batch['label'].to(device)

                logits, uncertainty = model(baseline, followup, dose,
    dvf, radiomics)
                loss = criterion(uncertainty, labels)

                val_loss += loss.item()
                predictions = torch.argmax(logits, dim=1)
                val_correct += (predictions == labels).sum().item()
                val_total += labels.size(0)

        # Calculate metrics
        train_acc = train_correct / train_total
        val_acc = val_correct / val_total

        print(f"Epoch {epoch+1}/{epochs}, "
              f"Train Loss: {train_loss/len(train_loader):.4f}, "
              f"Train Acc: {train_acc:.4f}, "
              f"Val Loss: {val_loss/len(val_loader):.4f}, "
              f"Val Acc: {val_acc:.4f}")

        # Save best model
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), 'best_response_classifier.pth
    ')

        scheduler.step()

    print(f"Best validation accuracy: {best_val_acc:.4f}")

    return model
```

```
262
263 # Example usage
264 if __name__ == "__main__":
265     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu'
    )
266
267     # Initialize model
268     model = MultimodalResponseClassifier(num_radiomics=100).to(device)
269
270     print(f"Model parameters: {sum(p.numel() for p in model.parameters()
    ):,}")
271
272     # Test forward pass
273     batch_size = 2
274     baseline = torch.randn(batch_size, 1, 64, 64, 64, device=device)
275     followup = torch.randn(batch_size, 1, 64, 64, 64, device=device)
276     dose = torch.randn(batch_size, 1, 64, 64, 64, device=device)
277     dvf = torch.randn(batch_size, 1, 64, 64, 64, device=device)
278     radiomics = torch.randn(batch_size, 100, device=device)
279
280     logits, uncertainty = model(baseline, followup, dose, dvf, radiomics
    )
281
282     print(f"Logits shape: {logits.shape}")
283     print(f"Uncertainty shape: {uncertainty.shape}")
```

Listing 5: Deep Learning Feature Extractor for Response Classification

3.2.3 Population Anatomy Modeling

```
1 import torch
2 import torch.nn as nn
3 from sklearn.decomposition import PCA
4 import numpy as np
5
6 class PopulationAnatomyModel:
7     """
8     Statistical model of expected anatomical variations in a population.
9     Used to distinguish expected vs unexpected (biological) changes.
10    """
11
12    def __init__(self, n_components=20):
13        """
14        Args:
15            n_components: Number of principal components to retain
16        """
17        self.n_components = n_components
18        self.pca = None
19        self.mean_anatomy = None
20        self.anatomy_std = None
21
22    def fit(self, anatomy_features_list):
23        """
24        Fit population model from training data.
25
26        Args:
27            anatomy_features_list: List of anatomy feature vectors
```

```
                                    (e.g., organ volumes, positions,
    shapes)
        """
        # Stack features
        X = np.stack(anatomy_features_list, axis=0)

        # Store statistics
        self.mean_anatomy = np.mean(X, axis=0)
        self.anatomy_std = np.std(X, axis=0)

        # Fit PCA
        self.pca = PCA(n_components=self.n_components)
        self.pca.fit(X)

        print(f"Explained variance ratio: {np.sum(self.pca.
    explained_variance_ratio_):.2%}")

    def calculate_anatomical_likelihood(self, anatomy_features):
        """
        Calculate likelihood of observed anatomy under population model.
        Low likelihood suggests biological change rather than typical
    variation.

        Args:
            anatomy_features: Feature vector for patient anatomy

        Returns:
            Log-likelihood under population model
        """
        # Project to PC space
        z = self.pca.transform(anatomy_features.reshape(1, -1))

        # Reconstruct
        reconstruction = self.pca.inverse_transform(z)

        # Reconstruction error
        reconstruction_error = np.linalg.norm(anatomy_features -
    reconstruction.ravel())

        # Mahalanobis distance in PC space
        eigenvalues = self.pca.explained_variance_
        mahal_dist = np.sum((z ** 2) / eigenvalues)

        # Log-likelihood (assuming Gaussian)
        log_likelihood = -0.5 * mahal_dist - 0.5 * np.sum(np.log(
    eigenvalues))

        return {
            'log_likelihood': log_likelihood,
            'reconstruction_error': reconstruction_error,
            'mahalanobis_distance': mahal_dist,
            'is_outlier': mahal_dist > 3 * self.n_components  # Chi-
    squared threshold
        }

    def generate_typical_variation(self):
        """
        Generate typical anatomical variation from population model.
```

```
        """
        # Sample from standard normal in PC space
        z = np.random.randn(self.n_components) * np.sqrt(self.pca.
    explained_variance_)

        # Transform back to feature space
        anatomy = self.pca.inverse_transform(z.reshape(1, -1))

        return anatomy.ravel()


class GeometricFeatureExtractor:
    """
    Extract geometric features from segmentation masks for population
    modeling.
    """

    @staticmethod
    def extract_features(mask, organ_name):
        """
        Extract geometric features from segmentation mask.

        Args:
            mask: Binary segmentation mask
            organ_name: Name of organ

        Returns:
            Dictionary of geometric features
        """
        features = {}

        # Volume
        features[f'{organ_name}_volume'] = np.sum(mask)

        # Centroid
        coords = np.argwhere(mask > 0)
        if len(coords) > 0:
            centroid = np.mean(coords, axis=0)
            features[f'{organ_name}_centroid_x'] = centroid[0]
            features[f'{organ_name}_centroid_y'] = centroid[1]
            features[f'{organ_name}_centroid_z'] = centroid[2]

            # Bounding box
            bbox_min = np.min(coords, axis=0)
            bbox_max = np.max(coords, axis=0)
            bbox_size = bbox_max - bbox_min

            features[f'{organ_name}_bbox_x'] = bbox_size[0]
            features[f'{organ_name}_bbox_y'] = bbox_size[1]
            features[f'{organ_name}_bbox_z'] = bbox_size[2]

            # Principal axes (using PCA)
            coords_centered = coords - centroid
            cov = np.cov(coords_centered.T)
            eigenvalues, eigenvectors = np.linalg.eigh(cov)

            # Sort by eigenvalue
            idx = eigenvalues.argsort()[::-1]
```

```
            eigenvalues = eigenvalues[idx]

            features[f'{organ_name}_axis1_length'] = np.sqrt(eigenvalues
    [0])
            features[f'{organ_name}_axis2_length'] = np.sqrt(eigenvalues
    [1])
            features[f'{organ_name}_axis3_length'] = np.sqrt(eigenvalues
    [2])

            # Shape metrics
            features[f'{organ_name}_compactness'] = (
                features[f'{organ_name}_volume'] /
                (features[f'{organ_name}_bbox_x'] *
                 features[f'{organ_name}_bbox_y'] *
                 features[f'{organ_name}_bbox_z'])
            )

        return features

    @staticmethod
    def extract_multiregion_features(masks_dict):
        """
        Extract features from multiple regions.

        Args:
            masks_dict: Dictionary of {organ_name: mask}

        Returns:
            Combined feature dictionary
        """
        all_features = {}

        for organ_name, mask in masks_dict.items():
            organ_features = GeometricFeatureExtractor.extract_features(
                mask, organ_name
            )
            all_features.update(organ_features)

        # Add inter-organ relationships
        if len(masks_dict) > 1:
            organ_names = list(masks_dict.keys())
            for i in range(len(organ_names)):
                for j in range(i+1, len(organ_names)):
                    organ1 = organ_names[i]
                    organ2 = organ_names[j]

                    # Distance between centroids
                    if f'{organ1}_centroid_x' in all_features and \
                       f'{organ2}_centroid_x' in all_features:
                        dist = np.sqrt(
                            (all_features[f'{organ1}_centroid_x'] -
                             all_features[f'{organ2}_centroid_x'])**2 +
                            (all_features[f'{organ1}_centroid_y'] -
                             all_features[f'{organ2}_centroid_y'])**2 +
                            (all_features[f'{organ1}_centroid_z'] -
                             all_features[f'{organ2}_centroid_z'])**2
                        )
                        all_features[f'distance_{organ1}_{organ2}'] =
```

```
    dist

        return all_features


# Example usage
if __name__ == "__main__":
    # Simulate population data
    n_patients = 100
    n_features = 50

    # Generate synthetic anatomy features
    population_features = [
        np.random.randn(n_features) for _ in range(n_patients)
    ]

    # Fit population model
    pop_model = PopulationAnatomyModel(n_components=15)
    pop_model.fit(population_features)

    # Test new patient
    new_patient_features = np.random.randn(n_features) * 2  # Larger
    variation

    likelihood_metrics = pop_model.calculate_anatomical_likelihood(
    new_patient_features)

    print("Anatomical likelihood metrics:")
    for key, val in likelihood_metrics.items():
        print(f"  {key}: {val}")

    if likelihood_metrics['is_outlier']:
        print("-> Suggests biological change rather than typical
    anatomical variation")
    else:
        print("-> Consistent with typical anatomical variation")
```

Listing 6: Population Anatomy Model for Expected Variations

3.3 Task 3: Dose Optimization Strategies

Deep Learning Feature Extraction

3.3.1 Objectives

Design algorithms that execute appropriate dose restoration, dose adaptation, or combined strategies based on identified change type.

3.3.2 Optimization Framework

```
import numpy as np
import scipy.optimize as opt
from dataclasses import dataclass
from typing import List, Dict, Tuple
import torch
```

```python

@dataclass
class OptimizationObjective:
    """Define optimization objective for dose planning."""
    structure_name: str
    objective_type: str  # 'min_dose', 'max_dose', 'mean_dose', 'dvh_constraint'
    dose_value: float  # Gy
    volume_fraction: float = None  # For DVH constraints
    priority: int = 1
    weight: float = 1.0


class AdaptiveDoseOptimizer:
    """
    Adaptive dose optimization engine that selects strategy based on
    response categorization (anatomical vs biological change).
    """

    def __init__(self, planning_ct, structures_dict, beam_model):
        """
        Args:
            planning_ct: Planning CT image
            structures_dict: Dictionary of structure masks
            beam_model: Proton beam dose calculation model
        """
        self.planning_ct = planning_ct
        self.structures = structures_dict
        self.beam_model = beam_model
        self.original_objectives = []

    def set_objectives(self, objectives: List[OptimizationObjective]):
        """Set dose optimization objectives."""
        self.original_objectives = objectives

    def optimize_dose_restoration(self, updated_ct, deformation_field):
        """
        Dose restoration for anatomical changes.
        Goal: Restore original dose distribution on updated anatomy.

        Args:
            updated_ct: Updated CT image for the day
            deformation_field: Deformation from planning to updated anatomy

        Returns:
            Optimized spot weights
        """
        print("Performing dose restoration optimization...")

        # Warp original objectives to updated anatomy
        warped_objectives = self._warp_objectives(
            self.original_objectives,
            deformation_field
        )

        # Standard dose optimization with warped objectives
        spot_weights = self._optimize_spot_weights(
```

```
            updated_ct,
            warped_objectives,
            strategy='restoration'
        )

        return spot_weights

    def optimize_dose_adaptation(self, updated_ct, response_info):
        """
        Dose adaptation for biological changes.
        Goal: Adapt dose levels based on biological response.

        Args:
            updated_ct: Updated CT image
            response_info: Dictionary with response characterization:
                - 'tumor_shrinkage': float (%)
                - 'tumor_density_change': float (HU)
                - 'predicted_response': str ('good', 'poor', 'intermediate')
                - 'risk_score': float (0-1)

        Returns:
            Optimized spot weights with adapted dose levels
        """
        print("Performing dose adaptation optimization...")

        # Modify objectives based on response
        adapted_objectives = self._adapt_objectives(
            self.original_objectives,
            response_info
        )

        # Optimize with adapted objectives
        spot_weights = self._optimize_spot_weights(
            updated_ct,
            adapted_objectives,
            strategy='adaptation'
        )

        return spot_weights

    def optimize_combined(self, updated_ct, deformation_field, response_info):
        """
        Combined optimization for mixed anatomical and biological changes.

        Args:
            updated_ct: Updated CT image
            deformation_field: Anatomical deformation
            response_info: Biological response information

        Returns:
            Optimized spot weights
        """
        print("Performing combined optimization...")

        # Warp objectives for anatomical component
```

```
        warped_objectives = self._warp_objectives(
            self.original_objectives,
            deformation_field
        )

        # Adapt objectives for biological component
        final_objectives = self._adapt_objectives(
            warped_objectives,
            response_info
        )

        # Optimize
        spot_weights = self._optimize_spot_weights(
            updated_ct,
            final_objectives,
            strategy='combined'
        )

        return spot_weights

    def _warp_objectives(self, objectives, deformation_field):
        """Warp dose objectives according to deformation."""
        warped_objectives = []

        for obj in objectives:
            # For structure objectives, warp the structure mask
            if obj.structure_name in self.structures:
                original_mask = self.structures[obj.structure_name]
                warped_mask = self._apply_deformation(original_mask,
    deformation_field)

                # Create new objective with warped mask
                new_obj = OptimizationObjective(
                    structure_name=obj.structure_name,
                    objective_type=obj.objective_type,
                    dose_value=obj.dose_value,  # Keep same dose level
                    volume_fraction=obj.volume_fraction,
                    priority=obj.priority,
                    weight=obj.weight
                )
                warped_objectives.append(new_obj)

        return warped_objectives

    def _adapt_objectives(self, objectives, response_info):
        """Adapt dose objectives based on biological response."""
        adapted_objectives = []

        # Response-based dose modification rules
        if response_info['predicted_response'] == 'poor':
            # Poor response -> consider dose escalation
            dose_modifier = 1.1  # 10% escalation
            print(f"  Applying dose escalation: {dose_modifier}x")

        elif response_info['predicted_response'] == 'good':
            # Good response -> may allow de-escalation for toxicity
    reduction
            dose_modifier = 0.95  # 5% de-escalation
```

```
            print(f"  Applying dose de-escalation: {dose_modifier}x")

        else:
            # Intermediate response -> maintain current dose
            dose_modifier = 1.0
            print(f"  Maintaining current dose levels")

        # Apply modifications
        for obj in objectives:
            if 'tumor' in obj.structure_name.lower() or 'gtv' in obj.
    structure_name.lower():
                # Modify tumor objectives
                new_obj = OptimizationObjective(
                    structure_name=obj.structure_name,
                    objective_type=obj.objective_type,
                    dose_value=obj.dose_value * dose_modifier,
                    volume_fraction=obj.volume_fraction,
                    priority=obj.priority,
                    weight=obj.weight
                )
                adapted_objectives.append(new_obj)
            else:
                # Keep OAR objectives unchanged (or tighten if dose
    escalation)
                if dose_modifier > 1.0:
                    # If escalating tumor dose, maintain strict OAR
    constraints
                    weight_modifier = 1.2
                else:
                    weight_modifier = 1.0

                new_obj = OptimizationObjective(
                    structure_name=obj.structure_name,
                    objective_type=obj.objective_type,
                    dose_value=obj.dose_value,
                    volume_fraction=obj.volume_fraction,
                    priority=obj.priority,
                    weight=obj.weight * weight_modifier
                )
                adapted_objectives.append(new_obj)

        return adapted_objectives

     def _optimize_spot_weights(self, ct_image, objectives, strategy='
    restoration'):
        """
        Core optimization routine to find optimal spot weights.

        Args:
            ct_image: CT image for dose calculation
            objectives: List of optimization objectives
            strategy: Optimization strategy identifier

        Returns:
            Optimized spot weights
        """
        # Get dose influence matrix from beam model
        D = self.beam_model.get_dose_influence_matrix(ct_image)
```

```
        n_spots = D.shape[1]
        n_voxels = D.shape[0]

        # Initialize spot weights
        w0 = np.ones(n_spots) * 0.01

        # Define objective function
        def objective_function(w):
            """Total objective function to minimize."""
            dose = D @ w
            total_cost = 0

            for obj in objectives:
                structure_mask = self.structures[obj.structure_name]
                structure_dose = dose[structure_mask.ravel() > 0]

                if obj.objective_type == 'min_dose':
                    # Penalize dose below target
                    underdose = np.maximum(0, obj.dose_value -
structure_dose)
                    cost = obj.weight * np.sum(underdose ** 2)

                elif obj.objective_type == 'max_dose':
                    # Penalize dose above limit
                    overdose = np.maximum(0, structure_dose - obj.
dose_value)
                    cost = obj.weight * np.sum(overdose ** 2)

                elif obj.objective_type == 'mean_dose':
                    # Penalize deviation from target mean
                    mean_dose = np.mean(structure_dose)
                    cost = obj.weight * (mean_dose - obj.dose_value) **
2

                elif obj.objective_type == 'dvh_constraint':
                    # DVH constraint: V_dose < volume_fraction
                    n_voxels_above = np.sum(structure_dose > obj.
dose_value)
                    fraction_above = n_voxels_above / len(structure_dose
)
                    violation = max(0, fraction_above - obj.
volume_fraction)
                    cost = obj.weight * violation ** 2 * 1e6  # Large
penalty

                total_cost += cost

            # Regularization: prefer smooth spot weight distributions
            regularization = 1e-4 * np.sum(np.diff(w) ** 2)
            total_cost += regularization

            return total_cost

        # Constraints: non-negative weights
        bounds = [(0, None) for _ in range(n_spots)]

        # Optimize
        result = opt.minimize(
```

```
            objective_function,
            w0,
            method='L-BFGS-B',
            bounds=bounds,
            options={'maxiter': 500, 'disp': False}
        )

        optimal_weights = result.x

        print(f"  Optimization converged: {result.success}")
        print(f"  Final objective value: {result.fun:.2f}")
        print(f"  Active spots: {np.sum(optimal_weights > 1e-6)}/{
    n_spots}")

        return optimal_weights

    def _apply_deformation(self, image, deformation_field):
        """Apply deformation field to image."""
        from scipy.ndimage import map_coordinates

        dims = image.shape
        coords = np.meshgrid(
            np.arange(dims[0]),
            np.arange(dims[1]),
            np.arange(dims[2]),
            indexing='ij'
        )
        coords = np.stack(coords, axis=0)

        deformed_coords = coords + deformation_field

        warped_image = map_coordinates(
            image,
            [deformed_coords[0].ravel(),
             deformed_coords[1].ravel(),
             deformed_coords[2].ravel()],
            order=1,
            mode='nearest'
        ).reshape(dims)

        return warped_image


class ProtonBeamModel:
    """
    Simplified proton beam dose calculation model.
    In practice, would interface with treatment planning system.
    """

    def __init__(self, spot_positions, spot_energies):
        """
        Args:
            spot_positions: Array of spot positions shape (N, 3)
            spot_energies: Array of spot energies shape (N,)
        """
        self.spot_positions = spot_positions
        self.spot_energies = spot_energies
        self.n_spots = len(spot_positions)
```

```
    def get_dose_influence_matrix(self, ct_image):
        """
        Calculate dose influence matrix D where D[i,j] is the dose
        deposited in voxel i by spot j with unit weight.

        Args:
            ct_image: CT image for dose calculation

        Returns:
            Dose influence matrix, shape (n_voxels, n_spots)
        """
        dims = ct_image.shape
        n_voxels = np.prod(dims)

        # Simplified: use Gaussian spots with range determined by energy
        # In practice, use Monte Carlo or analytical pencil beam algorithm

        D = np.zeros((n_voxels, self.n_spots))

        # Create voxel coordinates
        voxel_coords = np.stack(
            np.meshgrid(
                np.arange(dims[0]),
                np.arange(dims[1]),
                np.arange(dims[2]),
                indexing='ij'
            ),
            axis=-1
        ).reshape(-1, 3)

        for j, (pos, energy) in enumerate(zip(self.spot_positions, self.spot_energies)):
            # Simplified Bragg peak model
            range_cm = 0.03 * energy  # Rough estimate
            sigma_lateral = 0.5  # cm
            sigma_range = 0.2  # cm

            # Distance from spot
            lateral_dist = np.linalg.norm(voxel_coords[:, :2] - pos[:2], axis=1)
            depth_diff = voxel_coords[:, 2] - (pos[2] + range_cm)

            # Gaussian lateral, Bragg peak longitudinal
            dose = (
                np.exp(-lateral_dist**2 / (2 * sigma_lateral**2)) *
                np.exp(-depth_diff**2 / (2 * sigma_range**2)) *
                (1 + depth_diff / sigma_range)  # Asymmetric peak
            )

            # Normalize
            dose = dose / np.max(dose) if np.max(dose) > 0 else dose

            D[:, j] = dose

        return D
```

```python

# Example usage
if __name__ == "__main__":
    # Simulate planning CT and structures
    dims = (100, 100, 80)
    planning_ct = np.random.randn(*dims) * 50 + 1000  # HU values

    # Structures
    structures = {
        'tumor': np.zeros(dims),
        'spinal_cord': np.zeros(dims),
        'lung': np.zeros(dims)
    }

    # Simple geometric structures
    structures['tumor'][40:60, 40:60, 30:50] = 1
    structures['spinal_cord'][45:55, 10:20, :] = 1
    structures['lung'][20:80, 20:80, 10:70] = 1

    # Beam model
    n_spots = 500
    spot_positions = np.random.rand(n_spots, 3) * np.array([100, 100, 50])
    spot_energies = np.random.rand(n_spots) * 200 + 50  # MeV
    beam_model = ProtonBeamModel(spot_positions, spot_energies)

    # Initialize optimizer
    optimizer = AdaptiveDoseOptimizer(planning_ct, structures, beam_model)

    # Set objectives
    objectives = [
        OptimizationObjective('tumor', 'min_dose', 60.0, priority=1, weight=10.0),
        OptimizationObjective('spinal_cord', 'max_dose', 45.0, priority=1, weight=20.0),
        OptimizationObjective('lung', 'mean_dose', 20.0, priority=2, weight=5.0),
    ]
    optimizer.set_objectives(objectives)

    # Scenario 1: Anatomical change (dose restoration)
    print("\n=== Scenario 1: Anatomical Change ===")
    updated_ct_anat = planning_ct.copy()
    deformation = np.random.randn(3, *dims) * 2  # Small deformation

    weights_restoration = optimizer.optimize_dose_restoration(
        updated_ct_anat,
        deformation
    )

    # Scenario 2: Biological change (dose adaptation)
    print("\n=== Scenario 2: Biological Change ===")
    updated_ct_bio = planning_ct.copy()
    response_info = {
        'tumor_shrinkage': 25.0,
        'tumor_density_change': -15.0,
        'predicted_response': 'poor',
```

```
        'risk_score': 0.75
    }

    weights_adaptation = optimizer.optimize_dose_adaptation(
        updated_ct_bio,
        response_info
    )

    # Scenario 3: Combined change
    print("\n=== Scenario 3: Combined Change ===")
    weights_combined = optimizer.optimize_combined(
        updated_ct_anat,
        deformation,
        response_info
    )
```

Listing 7: Adaptive Dose Optimization Framework

3.3.3 Reinforcement Learning for Sequential Adaptation

For handling sequential decisions across multiple fractions:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import deque
import random

class AdaptiveRTEnvironment:
    """
    Reinforcement learning environment for adaptive radiotherapy.
    State: current anatomy, accumulated dose, response biomarkers
    Action: dose adaptation strategy (restore, escalate, de-escalate,
   maintain)
    Reward: predicted tumor control - normal tissue toxicity
    """

    def __init__(self, patient_model, total_fractions=30):
        self.patient_model = patient_model
        self.total_fractions = total_fractions
        self.current_fraction = 0
        self.accumulated_dose = None
        self.reset()

    def reset(self):
        """Reset environment to initial state."""
        self.current_fraction = 0
        self.accumulated_dose = np.zeros(self.patient_model.
   anatomy_shape)
        initial_state = self._get_state()
        return initial_state

    def step(self, action):
        """
        Take action and advance one fraction.

        Args:
```

```
            action: Integer action code
                0: Dose restoration (anatomical)
                1: Dose escalation (biological - poor response)
                2: Dose de-escalation (biological - good response)
                3: Maintain current plan

        Returns:
            next_state: New state after action
            reward: Immediate reward
            done: Whether treatment is complete
            info: Additional information
        """
        # Simulate treatment delivery
        fraction_dose = self.patient_model.get_fraction_dose(action)
        self.accumulated_dose += fraction_dose

        # Update patient state (anatomy + biology)
        self.patient_model.update_state(fraction_dose, self.
    current_fraction)

        # Calculate reward
        reward = self._calculate_reward(action)

        # Advance fraction
        self.current_fraction += 1
        done = self.current_fraction >= self.total_fractions

        # Get next state
        next_state = self._get_state()

        info = {
            'fraction': self.current_fraction,
            'tumor_control_prob': self.patient_model.get_tcp(),
            'ntcp': self.patient_model.get_ntcp(),
        }

        return next_state, reward, done, info

    def _get_state(self):
        """
        Get current state representation.

        Returns:
            State vector combining anatomy, dose, and biomarkers
        """
        state = {
            'anatomy': self.patient_model.get_current_anatomy(),
            'accumulated_dose': self.accumulated_dose,
            'fraction_number': self.current_fraction / self.
    total_fractions,
            'tumor_volume': self.patient_model.get_tumor_volume(),
            'biomarkers': self.patient_model.get_biomarkers(),
        }

        # Flatten to vector
        state_vector = np.concatenate([
            state['anatomy'].ravel(),
            state['accumulated_dose'].ravel(),
```

```
            [state['fraction_number']],
            [state['tumor_volume']],
            state['biomarkers']
        ])

        return state_vector

    def _calculate_reward(self, action):
        """
        Calculate reward based on predicted outcomes.

        Reward = TCP -  1 *NTCP -  2 *ActionCost
        """
        tcp = self.patient_model.get_tcp()
        ntcp = self.patient_model.get_ntcp()

        # Action costs (replanning burden)
        action_costs = [0.1, 0.2, 0.15, 0.0]  # restore, escalate, de-escalate, maintain
        action_cost = action_costs[action]

        # Combined reward
        reward = tcp - 2.0 * ntcp - 0.1 * action_cost

        return reward


class DQNAgent:
    """Deep Q-Network agent for adaptive RT decision making."""

    def __init__(self, state_dim, action_dim, hidden_dim=256):
        self.state_dim = state_dim
        self.action_dim = action_dim

        # Q-networks
        self.q_network = self._build_network(hidden_dim)
        self.target_network = self._build_network(hidden_dim)
        self.target_network.load_state_dict(self.q_network.state_dict())

        # Training parameters
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=1e-4)
        self.memory = deque(maxlen=10000)
        self.batch_size = 64
        self.gamma = 0.99
        self.epsilon = 1.0
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01
        self.update_target_every = 100
        self.steps = 0

    def _build_network(self, hidden_dim):
        """Build Q-network."""
        return nn.Sequential(
            nn.Linear(self.state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
```

```
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, self.action_dim)
        )

    def select_action(self, state, training=True):
        """Select action using epsilon-greedy policy."""
        if training and random.random() < self.epsilon:
            return random.randint(0, self.action_dim - 1)
        else:
            with torch.no_grad():
                state_tensor = torch.FloatTensor(state).unsqueeze(0)
                q_values = self.q_network(state_tensor)
                return q_values.argmax().item()

    def store_transition(self, state, action, reward, next_state, done):
        """Store transition in replay buffer."""
        self.memory.append((state, action, reward, next_state, done))

    def train(self):
        """Train Q-network on a batch from replay buffer."""
        if len(self.memory) < self.batch_size:
            return

        # Sample batch
        batch = random.sample(self.memory, self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.FloatTensor(np.array(states))
        actions = torch.LongTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(np.array(next_states))
        dones = torch.FloatTensor(dones)

        # Current Q values
        current_q = self.q_network(states).gather(1, actions.unsqueeze
(1))

        # Target Q values
        with torch.no_grad():
            next_q = self.target_network(next_states).max(1)[0]
            target_q = rewards + self.gamma * next_q * (1 - dones)

        # Loss
        loss = nn.MSELoss()(current_q.squeeze(), target_q)

        # Optimize
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update target network
        self.steps += 1
        if self.steps % self.update_target_every == 0:
            self.target_network.load_state_dict(self.q_network.
state_dict())

        # Decay epsilon
```

```
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

        return loss.item()


def train_rl_agent(env, agent, num_episodes=1000):
    """
    Train RL agent for adaptive RT decision making.

    Args:
        env: AdaptiveRTEnvironment
        agent: DQNAgent
        num_episodes: Number of training episodes (patients)
    """
    rewards_history = []

    for episode in range(num_episodes):
        state = env.reset()
        episode_reward = 0
        done = False

        while not done:
            # Select and perform action
            action = agent.select_action(state, training=True)
            next_state, reward, done, info = env.step(action)

            # Store transition
            agent.store_transition(state, action, reward, next_state,
    done)

            # Train
            loss = agent.train()

            episode_reward += reward
            state = next_state

        rewards_history.append(episode_reward)

        if episode % 10 == 0:
            avg_reward = np.mean(rewards_history[-10:])
            print(f"Episode {episode}, Avg Reward: {avg_reward:.3f}, "
                  f"Epsilon: {agent.epsilon:.3f}, "
                  f"TCP: {info['tumor_control_prob']:.3f}, "
                  f"NTCP: {info['ntcp']:.3f}")

    return agent, rewards_history


# Placeholder patient model (would be replaced with real biological
    model)
class PatientModel:
    """Simplified patient model for RL environment."""

    def __init__(self):
        self.anatomy_shape = (50, 50, 40)
        self.tumor_volume_initial = 100.0
        self.tumor_volume = self.tumor_volume_initial
```

```
        self.alpha_beta_ratio = 10.0

    def get_current_anatomy(self):
        return np.random.randn(*self.anatomy_shape) * 0.1

    def get_tumor_volume(self):
        return self.tumor_volume

    def get_biomarkers(self):
        return np.random.randn(10)

    def get_fraction_dose(self, action):
        base_dose = 2.0  # Gy per fraction
        if action == 1:  # Escalation
            dose_level = base_dose * 1.1
        elif action == 2:  # De-escalation
            dose_level = base_dose * 0.9
        else:
            dose_level = base_dose

        dose_map = np.random.rand(*self.anatomy_shape) * dose_level
        return dose_map

    def update_state(self, dose, fraction):
        # Simulate tumor regression
        regression_rate = 0.02
        self.tumor_volume *= (1 - regression_rate)

    def get_tcp(self):
        # Linear-quadratic model
        total_dose = 60.0  # Will be calculated from accumulated dose
        sf = np.exp(-0.3 * total_dose - 0.03 * total_dose**2 / self.
    alpha_beta_ratio)
        tcp = 1 - np.exp(-np.log(self.tumor_volume / 1e6) * (1 - sf))
        return max(0, min(1, tcp))

    def get_ntcp(self):
        return 0.1  # Simplified


# Example usage
if __name__ == "__main__":
    # Initialize environment and agent
    patient_model = PatientModel()
    env = AdaptiveRTEnvironment(patient_model)

    state_dim = len(env.reset())
    action_dim = 4  # restore, escalate, de-escalate, maintain

    agent = DQNAgent(state_dim, action_dim)

    # Train agent
    print("Training RL agent for adaptive RT...")
    trained_agent, rewards = train_rl_agent(env, agent, num_episodes
    =100)

```

```
    print(f"\nTraining completed. Final average reward: {np.mean(rewards[-10:]):.3f}")
```

Listing 8: RL-Based Sequential Adaptation Strategy

3.4 Task 4: In-Silico Integration

3.4.1 Objectives

Implement proof-of-concept pipeline integrating all components and evaluate within clinical treatment planning system.

3.4.2 Complete Pipeline Implementation

```
import numpy as np
import torch
from dataclasses import dataclass
from typing import Dict, List, Tuple
import logging
from pathlib import Path
import json

# Setup logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


@dataclass
class PatientData:
    """Container for patient data."""
    patient_id: str
    baseline_ct: np.ndarray
    baseline_structures: Dict[str, np.ndarray]
    daily_ct: np.ndarray = None
    accumulated_dose: np.ndarray = None
    fraction_number: int = 0


@dataclass
class ResponseClassification:
    """Container for response classification results."""
    primary_type: str  # 'anatomical', 'biological', 'mixed'
    confidence: float
    anatomical_score: float
    biological_score: float
    uncertainty: float
    features: Dict


class IntegratedAdaptivePipeline:
    """
    Complete integrated pipeline for AI-driven adaptive proton therapy.
    Combines all four tasks into a cohesive clinical workflow.
    """

    def __init__(self, models_dir='./models'):
```

```
        """
        Initialize pipeline with trained models.

        Args:
            models_dir: Directory containing trained model weights
        """
        self.models_dir = Path(models_dir)

        logger.info("Loading pipeline components...")

        # Load models for each task
        self.synthetic_generator = self._load_generator()
        self.response_classifier = self._load_classifier()
        self.dose_optimizer = None  # Initialized per patient
        self.feature_extractor = RadiomicFeatureExtractor()

        logger.info("Pipeline initialized successfully")

    def _load_generator(self):
        """Load synthetic image generation model."""
        # In practice, load from saved weights
        from Task1 import ConditionalDDPM, ConditionalUNet3D

        model = ConditionalUNet3D()
        ddpm = ConditionalDDPM(model)

        # Load weights if available
        model_path = self.models_dir / 'generator.pth'
        if model_path.exists():
            model.load_state_dict(torch.load(model_path))
            logger.info(f"Loaded generator from {model_path}")

        return ddpm

    def _load_classifier(self):
        """Load response classification model."""
        from Task2 import MultimodalResponseClassifier

        model = MultimodalResponseClassifier()

        model_path = self.models_dir / 'classifier.pth'
        if model_path.exists():
            model.load_state_dict(torch.load(model_path))
            model.eval()
            logger.info(f"Loaded classifier from {model_path}")

        return model

    def process_daily_adaptation(self, patient_data: PatientData) ->
Dict:
        """
        Main pipeline for daily adaptive decision-making.

        Args:
            patient_data: PatientData object with baseline and daily
imaging

        Returns:
```

```
            Dictionary with adaptation decision and optimized plan
        """
        logger.info(f"Processing patient {patient_data.patient_id}, "
                    f"fraction {patient_data.fraction_number}")

        results = {}

        # Step 1: Image registration and change detection
        logger.info("Step 1: Image registration")
        registration_results = self._register_images(
            patient_data.baseline_ct,
            patient_data.daily_ct
        )
        results['registration'] = registration_results

        # Step 2: Feature extraction
        logger.info("Step 2: Feature extraction")
        features = self._extract_multimodal_features(
            patient_data,
            registration_results
        )
        results['features'] = features

        # Step 3: Response classification
        logger.info("Step 3: Response classification")
        classification = self._classify_response(
            patient_data,
            features,
            registration_results
        )
        results['classification'] = classification

        # Step 4: Adaptive strategy selection
        logger.info("Step 4: Strategy selection and dose optimization")
        adaptation_decision = self._select_adaptation_strategy(
    classification)
        results['adaptation_decision'] = adaptation_decision

        # Step 5: Dose optimization
        optimized_plan = self._optimize_dose(
            patient_data,
            registration_results,
            adaptation_decision
        )
        results['optimized_plan'] = optimized_plan

        # Step 6: Quality assurance
        logger.info("Step 5: Quality assurance")
        qa_results = self._perform_qa(
            patient_data,
            optimized_plan,
            classification
        )
        results['qa'] = qa_results

        # Step 7: Generate report
        report = self._generate_report(results)
        results['report'] = report
```

```
        logger.info(f"Pipeline completed: {adaptation_decision['strategy']}")

        return results

    def _register_images(self, baseline_ct, daily_ct):
        """
        Perform deformable image registration.
        """
        import SimpleITK as sitk

        # Convert to SimpleITK
        fixed = sitk.GetImageFromArray(baseline_ct.astype(np.float32))
        moving = sitk.GetImageFromArray(daily_ct.astype(np.float32))

        # Registration
        demons = sitk.DemonsRegistrationFilter()
        demons.SetNumberOfIterations(50)
        demons.SetStandardDeviations(1.0)

        displacementField = demons.Execute(fixed, moving)

        # Convert to numpy
        dvf = sitk.GetArrayFromImage(displacementField)
        dvf = np.transpose(dvf, (3, 0, 1, 2))

        # Calculate deformation magnitude
        dvf_magnitude = np.linalg.norm(dvf, axis=0)

        # Warp baseline structures to daily anatomy
        warped_ct = sitk.GetArrayFromImage(
            sitk.Resample(moving, fixed, sitk.Transform(),
                          sitk.sitkLinear, 0.0, moving.GetPixelID())
        )

        return {
            'dvf': dvf,
            'dvf_magnitude': dvf_magnitude,
            'warped_ct': warped_ct,
            'mean_deformation_mm': np.mean(dvf_magnitude),
            'max_deformation_mm': np.max(dvf_magnitude)
        }

    def _extract_multimodal_features(self, patient_data, registration_results):
        """
        Extract comprehensive multimodal features.
        """
        features = {}

        # Geometric features
        tumor_mask = patient_data.baseline_structures.get('tumor')
        if tumor_mask is not None:
            geom_features = GeometricFeatureExtractor.extract_features(
                tumor_mask, 'tumor'
            )
            features['geometric'] = geom_features
```

```
212
213         # Radiomic features (baseline)
214         baseline_radiomics = self.feature_extractor.extract_features(
215             patient_data.baseline_ct,
216             tumor_mask
217         )
218         features['radiomics_baseline'] = baseline_radiomics
219
220         # Delta-radiomic features
221         if patient_data.daily_ct is not None:
222             delta_radiomics = self.feature_extractor.
    extract_delta_features(
223                 patient_data.baseline_ct,
224                 patient_data.daily_ct,
225                 tumor_mask
226             )
227             features['radiomics_delta'] = delta_radiomics
228
229         # Biological response features
230         if patient_data.daily_ct is not None:
231             bio_features = {}
232             bio_features.update(BiologicalResponseFeatures.volume_change
    (
233                 tumor_mask, tumor_mask, (1.0, 1.0, 3.0)
234             ))
235             bio_features.update(BiologicalResponseFeatures.
    density_change(
236                 patient_data.baseline_ct,
237                 patient_data.daily_ct,
238                 tumor_mask
239             ))
240             features['biological'] = bio_features
241
242         # Deformation features
243         features['deformation'] = {
244             'mean_dvf': registration_results['mean_deformation_mm'],
245             'max_dvf': registration_results['max_deformation_mm'],
246             'dvf_std': np.std(registration_results['dvf_magnitude']),
247         }
248
249         return features
250
251     def _classify_response(self, patient_data, features,
    registration_results):
252         """
253         Classify change type using trained model.
254         """
255         device = torch.device('cuda' if torch.cuda.is_available() else '
    cpu')
256
257         # Prepare inputs for classifier
258         baseline_tensor = torch.FloatTensor(
259             patient_data.baseline_ct
260         ).unsqueeze(0).unsqueeze(0).to(device)
261
262         daily_tensor = torch.FloatTensor(
263             patient_data.daily_ct
264         ).unsqueeze(0).unsqueeze(0).to(device)
```

```
        dose_tensor = torch.FloatTensor(
            patient_data.accumulated_dose
        ).unsqueeze(0).unsqueeze(0).to(device)

        dvf_tensor = torch.FloatTensor(
            registration_results['dvf_magnitude']
        ).unsqueeze(0).unsqueeze(0).to(device)

        # Combine radiomic features
        radiomics_list = []
        for key in sorted(features['radiomics_baseline'].keys()):
            if isinstance(features['radiomics_baseline'][key], (int, float)):
                radiomics_list.append(features['radiomics_baseline'][key])
        radiomics_tensor = torch.FloatTensor(radiomics_list).unsqueeze(0).to(device)

        # Classify
        with torch.no_grad():
            logits, uncertainty = self.response_classifier(
                baseline_tensor,
                daily_tensor,
                dose_tensor,
                dvf_tensor,
                radiomics_tensor
            )

            probs = torch.softmax(logits, dim=1)
            pred_class = torch.argmax(probs, dim=1).item()
            confidence = probs[0, pred_class].item()

        class_names = ['anatomical', 'biological', 'mixed']

        classification = ResponseClassification(
            primary_type=class_names[pred_class],
            confidence=confidence,
            anatomical_score=probs[0, 0].item(),
            biological_score=probs[0, 1].item(),
            uncertainty=uncertainty[0].mean().item(),
            features=features
        )

        return classification

    def _select_adaptation_strategy(self, classification):
        """
        Select appropriate adaptation strategy based on classification.
        """
        strategy = {}

        if classification.primary_type == 'anatomical':
            strategy['strategy'] = 'dose_restoration'
            strategy['description'] = 'Restore planned dose distribution on updated anatomy'
            strategy['requires_replanning'] = True
            strategy['replan_priority'] = 'medium'
```

```
319
320         elif classification.primary_type == 'biological':
321             if classification.biological_score > 0.7:
322                 strategy['strategy'] = 'dose_adaptation'
323                 strategy['description'] = 'Adapt dose levels based on
    biological response'
324                 strategy['requires_replanning'] = True
325                 strategy['replan_priority'] = 'high'
326             else:
327                 strategy['strategy'] = 'monitor'
328                 strategy['description'] = 'Continue monitoring, moderate
     biological change'
329                 strategy['requires_replanning'] = False
330
331         elif classification.primary_type == 'mixed':
332             strategy['strategy'] = 'combined_adaptation'
333             strategy['description'] = 'Combined anatomical restoration
    and biological adaptation'
334             strategy['requires_replanning'] = True
335             strategy['replan_priority'] = 'high'
336
337         strategy['confidence'] = classification.confidence
338         strategy['uncertainty'] = classification.uncertainty
339
340         return strategy
341
342     def _optimize_dose(self, patient_data, registration_results,
    adaptation_decision):
343         """
344         Perform dose optimization based on selected strategy.
345         """
346         # Initialize optimizer for this patient
347         from Task3 import AdaptiveDoseOptimizer, OptimizationObjective,
    ProtonBeamModel
348
349         # Dummy beam model (would be from TPS)
350         n_spots = 500
351         spot_positions = np.random.rand(n_spots, 3) * np.array(
    patient_data.baseline_ct.shape)
352         spot_energies = np.random.rand(n_spots) * 200 + 50
353         beam_model = ProtonBeamModel(spot_positions, spot_energies)
354
355         optimizer = AdaptiveDoseOptimizer(
356             patient_data.baseline_ct,
357             patient_data.baseline_structures,
358             beam_model
359         )
360
361         # Set objectives
362         objectives = [
363             OptimizationObjective('tumor', 'min_dose', 60.0, priority=1,
     weight=10.0),
364         ]
365         optimizer.set_objectives(objectives)
366
367         # Optimize based on strategy
368         if adaptation_decision['strategy'] == 'dose_restoration':
369             spot_weights = optimizer.optimize_dose_restoration(
```

```
                patient_data.daily_ct,
                registration_results['dvf']
            )

        elif adaptation_decision['strategy'] == 'dose_adaptation':
            response_info = {
                'predicted_response': 'intermediate',
                'risk_score': 0.5
            }
            spot_weights = optimizer.optimize_dose_adaptation(
                patient_data.daily_ct,
                response_info
            )

        elif adaptation_decision['strategy'] == 'combined_adaptation':
            response_info = {
                'predicted_response': 'poor',
                'risk_score': 0.7
            }
            spot_weights = optimizer.optimize_combined(
                patient_data.daily_ct,
                registration_results['dvf'],
                response_info
            )
        else:
            # No replanning needed
            spot_weights = None

        return {
            'spot_weights': spot_weights,
            'n_spots': len(spot_weights) if spot_weights is not None
    else 0,
            'active_spots': np.sum(spot_weights > 1e-6) if spot_weights
    is not None else 0
        }

     def _perform_qa(self, patient_data, optimized_plan, classification):
        """
        Perform quality assurance checks on adapted plan.
        """
        qa_results = {
            'passed': True,
            'warnings': [],
            'checks': {}
        }

        # Check 1: Confidence threshold
        if classification.confidence < 0.7:
            qa_results['warnings'].append(
                f"Low classification confidence: {classification.
    confidence:.2f}"
            )

        # Check 2: Uncertainty threshold
        if classification.uncertainty > 0.5:
            qa_results['warnings'].append(
                f"High uncertainty: {classification.uncertainty:.2f}"
            )
```

```
425
426         # Check 3: Spot weight distribution
427         if optimized_plan['spot_weights'] is not None:
428             max_weight = np.max(optimized_plan['spot_weights'])
429             if max_weight > 10.0:
430                 qa_results['warnings'].append(
431                     f"High maximum spot weight: {max_weight:.2f}"
432                 )
433
434         qa_results['checks']['classification_confidence'] =
    classification.confidence > 0.7
435         qa_results['checks']['uncertainty'] = classification.uncertainty
     < 0.5
436
437         if len(qa_results['warnings']) > 0:
438             qa_results['passed'] = False
439
440         return qa_results
441
442     def _generate_report(self, results):
443         """
444         Generate comprehensive adaptation report.
445         """
446         classification = results['classification']
447         adaptation = results['adaptation_decision']
448         qa = results['qa']
449
450         report = {
451             'timestamp': str(np.datetime64('now')),
452             'summary': {
453                 'change_type': classification.primary_type,
454                 'confidence': f"{classification.confidence:.1%}",
455                 'strategy': adaptation['strategy'],
456                 'requires_replanning': adaptation['requires_replanning'
    ],
457             },
458             'details': {
459                 'anatomical_score': f"{classification.anatomical_score
    :.1%}",
460                 'biological_score': f"{classification.biological_score
    :.1%}",
461                 'uncertainty': f"{classification.uncertainty:.3f}",
462                 'mean_deformation': f"{results['registration']['
    mean_deformation_mm']:.2f} mm",
463             },
464             'qa_status': 'PASSED' if qa['passed'] else 'REVIEW REQUIRED'
    ,
465             'qa_warnings': qa['warnings'],
466             'recommendation': adaptation['description']
467         }
468
469         return report
470
471
472 def clinical_workflow_simulation(patient_list, pipeline, output_dir='./
    results'):
473     """
474     Simulate clinical workflow for multiple patients.
```

```
475
476     Args:
477         patient_list: List of PatientData objects
478         pipeline: IntegratedAdaptivePipeline instance
479         output_dir: Directory to save results
480     """
481     output_dir = Path(output_dir)
482     output_dir.mkdir(exist_ok=True)
483
484     logger.info(f"Starting clinical workflow simulation for {len(
    patient_list)} patients")
485
486     all_results = []
487
488     for patient_data in patient_list:
489         logger.info(f"\n{'='*60}")
490         logger.info(f"Processing {patient_data.patient_id}")
491         logger.info(f"{'='*60}")
492
493         try:
494             # Run pipeline
495             results = pipeline.process_daily_adaptation(patient_data)
496
497             # Save results
498             patient_output = output_dir / patient_data.patient_id
499             patient_output.mkdir(exist_ok=True)
500
501             # Save report
502             report_path = patient_output / f'report_fraction_{
    patient_data.fraction_number}.json'
503             with open(report_path, 'w') as f:
504                 json.dump(results['report'], f, indent=2)
505
506             logger.info(f"Report saved to {report_path}")
507             logger.info(f"Result: {results['report']['summary']['
    strategy']}")
508
509             all_results.append({
510                 'patient_id': patient_data.patient_id,
511                 'fraction': patient_data.fraction_number,
512                 'strategy': results['adaptation_decision']['strategy'],
513                 'qa_passed': results['qa']['passed']
514             })
515
516         except Exception as e:
517             logger.error(f"Error processing {patient_data.patient_id}: {
    str(e)}")
518             continue
519
520     # Summary statistics
521     logger.info(f"\n{'='*60}")
522     logger.info("SUMMARY STATISTICS")
523     logger.info(f"{'='*60}")
524
525     strategies = [r['strategy'] for r in all_results]
526     for strategy in set(strategies):
527         count = strategies.count(strategy)
```

```
        logger.info(f"{strategy}: {count}/{len(strategies)} ({count/len(strategies):.1%})")

    qa_passed = sum(r['qa_passed'] for r in all_results)
    logger.info(f"QA passed: {qa_passed}/{len(all_results)} ({qa_passed/len(all_results):.1%})")

    return all_results


# Example usage
if __name__ == "__main__":
    # Initialize pipeline
    pipeline = IntegratedAdaptivePipeline(models_dir='./models')

    # Create dummy patient data
    patient_data = PatientData(
        patient_id='PT001',
        baseline_ct=np.random.randn(100, 100, 80) * 50 + 1000,
        baseline_structures={
            'tumor': np.zeros((100, 100, 80)),
        },
        daily_ct=np.random.randn(100, 100, 80) * 50 + 1000,
        accumulated_dose=np.random.rand(100, 100, 80) * 30,
        fraction_number=15
    )
    patient_data.baseline_structures['tumor'][40:60, 40:60, 30:50] = 1

    # Process single patient
    results = pipeline.process_daily_adaptation(patient_data)

    # Print report
    print("\n" + "="*60)
    print("ADAPTATION REPORT")
    print("="*60)
    print(json.dumps(results['report'], indent=2))
```

Listing 9: Complete In-Silico Pipeline Integration

3.4.3 Pipeline Validation and Evaluation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
from scipy import stats

class PipelineValidator:
    """
    Comprehensive validation framework for the integrated pipeline.
    """

    def __init__(self):
        self.results = []

    def evaluate_classification_accuracy(self, predictions, ground_truth):
```

```
        """
        Evaluate response classification accuracy.

        Args:
            predictions: List of predicted class labels
            ground_truth: List of true class labels
        """
        # Confusion matrix
        cm = confusion_matrix(ground_truth, predictions)

        # Classification report
        report = classification_report(ground_truth, predictions,
                                       target_names=['anatomical', '
    biological', 'mixed'])

        # Overall accuracy
        accuracy = np.sum(predictions == ground_truth) / len(
    ground_truth)

        return {
            'accuracy': accuracy,
            'confusion_matrix': cm,
            'classification_report': report
        }

    def evaluate_dose_quality(self, optimized_doses, target_doses,
    structures):
        """
        Evaluate quality of dose optimization.

        Args:
            optimized_doses: List of optimized dose distributions
            target_doses: List of target dose distributions
            structures: Dictionary of structure masks
        """
        metrics = {}

        for struct_name, struct_mask in structures.items():
            struct_doses_opt = [dose[struct_mask > 0] for dose in
    optimized_doses]
            struct_doses_target = [dose[struct_mask > 0] for dose in
    target_doses]

            # Metrics
            mean_differences = [
                np.mean(opt) - np.mean(target)
                for opt, target in zip(struct_doses_opt,
    struct_doses_target)
            ]

            dvh_differences = [
                self._calculate_dvh_difference(opt, target)
                for opt, target in zip(struct_doses_opt,
    struct_doses_target)
            ]

            metrics[struct_name] = {
                'mean_dose_diff_gy': np.mean(mean_differences),
```

```
                'mean_dose_diff_std': np.std(mean_differences),
                'dvh_diff_mean': np.mean(dvh_differences),
            }

        return metrics

    def _calculate_dvh_difference(self, dose1, dose2, bins=100):
        """Calculate difference between two DVH curves."""
        max_dose = max(np.max(dose1), np.max(dose2))
        dose_bins = np.linspace(0, max_dose, bins)

        dvh1 = np.array([np.sum(dose1 >= d) / len(dose1) * 100 for d in
dose_bins])
        dvh2 = np.array([np.sum(dose2 >= d) / len(dose2) * 100 for d in
dose_bins])

        # Mean absolute difference
        return np.mean(np.abs(dvh1 - dvh2))

    def evaluate_clinical_outcomes(self, adapted_plans, original_plans):
        """
        Evaluate predicted clinical outcomes.

        Args:
            adapted_plans: List of adapted treatment plans
            original_plans: List of original treatment plans
        """
        tcp_improvements = []
        ntcp_changes = []

        for adapted, original in zip(adapted_plans, original_plans):
            # Calculate TCP (simplified)
            tcp_adapted = self._calculate_tcp(adapted['tumor_dose'])
            tcp_original = self._calculate_tcp(original['tumor_dose'])
            tcp_improvements.append(tcp_adapted - tcp_original)

            # Calculate NTCP (simplified)
            ntcp_adapted = self._calculate_ntcp(adapted['oar_dose'])
            ntcp_original = self._calculate_ntcp(original['oar_dose'])
            ntcp_changes.append(ntcp_adapted - ntcp_original)

        return {
            'tcp_improvement_mean': np.mean(tcp_improvements),
            'tcp_improvement_std': np.std(tcp_improvements),
            'ntcp_change_mean': np.mean(ntcp_changes),
            'ntcp_change_std': np.std(ntcp_changes),
            'therapeutic_ratio_improvement': (
                np.mean(tcp_improvements) - np.mean(ntcp_changes)
            )
        }

    def _calculate_tcp(self, tumor_dose, alpha=0.3, beta=0.03):
        """Calculate tumor control probability (simplified LQ model)."""
        mean_dose = np.mean(tumor_dose)
        sf = np.exp(-alpha * mean_dose - beta * mean_dose**2)
        tcp = 1 - sf
        return tcp

```

```
    def _calculate_ntcp(self, oar_dose, d50=30, gamma=4):
        """Calculate normal tissue complication probability (simplified)."""
        mean_dose = np.mean(oar_dose)
        ntcp = 1 / (1 + (d50/mean_dose)**(4*gamma))
        return ntcp

    def evaluate_computational_efficiency(self, processing_times):
        """
        Evaluate computational efficiency of pipeline.

        Args:
            processing_times: Dictionary of processing times for each step
        """
        metrics = {
            'total_time_mean_s': np.mean([sum(t.values()) for t in processing_times]),
            'total_time_std_s': np.std([sum(t.values()) for t in processing_times]),
        }

        # Per-step statistics
        steps = processing_times[0].keys()
        for step in steps:
            step_times = [t[step] for t in processing_times]
            metrics[f'{step}_mean_s'] = np.mean(step_times)
            metrics[f'{step}_std_s'] = np.std(step_times)

        return metrics

    def generate_validation_report(self, all_metrics):
        """
        Generate comprehensive validation report.

        Args:
            all_metrics: Dictionary containing all evaluation metrics
        """
        report = []
        report.append("="*80)
        report.append("PIPELINE VALIDATION REPORT")
        report.append("="*80)
        report.append("")

        # Classification performance
        report.append("1. RESPONSE CLASSIFICATION PERFORMANCE")
        report.append("-" * 80)
        class_metrics = all_metrics['classification']
        report.append(f"Overall Accuracy: {class_metrics['accuracy']:.1%}")
        report.append("\nConfusion Matrix:")
        report.append(str(class_metrics['confusion_matrix']))
        report.append("\nPer-Class Metrics:")
        report.append(class_metrics['classification_report'])
        report.append("")

        # Dose optimization quality
        report.append("2. DOSE OPTIMIZATION QUALITY")
```

```
        report.append("-" * 80)
        dose_metrics = all_metrics['dose_quality']
        for struct, metrics in dose_metrics.items():
            report.append(f"\n{struct}:")
            report.append(f"  Mean dose difference: {metrics['mean_dose_diff_gy']:.2f}    {metrics['mean_dose_diff_std']:.2f} Gy")
            report.append(f"  DVH difference: {metrics['dvh_diff_mean']:.1f}%")
        report.append("")

        # Clinical outcomes
        report.append("3. PREDICTED CLINICAL OUTCOMES")
        report.append("-" * 80)
        outcome_metrics = all_metrics['clinical_outcomes']
        report.append(f"TCP improvement: {outcome_metrics['tcp_improvement_mean']:.1%}    {outcome_metrics['tcp_improvement_std']:.1%}")
        report.append(f"NTCP change: {outcome_metrics['ntcp_change_mean']:.1%}    {outcome_metrics['ntcp_change_std']:.1%}")
        report.append(f"Therapeutic ratio improvement: {outcome_metrics['therapeutic_ratio_improvement']:.1%}")
        report.append("")

        # Computational efficiency
        report.append("4. COMPUTATIONAL EFFICIENCY")
        report.append("-" * 80)
        comp_metrics = all_metrics['computational_efficiency']
        report.append(f"Total processing time: {comp_metrics['total_time_mean_s']:.1f}    {comp_metrics['total_time_std_s']:.1f} seconds")
        report.append("")

        return "\n".join(report)


# Example usage
if __name__ == "__main__":
    validator = PipelineValidator()

    # Simulate validation data
    n_patients = 100

    # Classification accuracy
    ground_truth = np.random.randint(0, 3, n_patients)
    predictions = ground_truth.copy()
    # Add some errors
    error_idx = np.random.choice(n_patients, size=10, replace=False)
    predictions[error_idx] = (predictions[error_idx] + 1) % 3

    class_metrics = validator.evaluate_classification_accuracy(predictions, ground_truth)

    # Dose quality (simplified simulation)
    structures = {'tumor': np.ones((50, 50, 40)), 'oar': np.ones((50, 50, 40))}
    optimized_doses = [np.random.rand(50, 50, 40) * 60 for _ in range(n_patients)]
    target_doses = [dose + np.random.randn(50, 50, 40) * 2 for dose in
```

```
    optimized_doses]

    dose_metrics = validator.evaluate_dose_quality(optimized_doses,
    target_doses, structures)

    # Clinical outcomes
    adapted_plans = [
        {'tumor_dose': np.random.rand(1000) * 60 + 50,
         'oar_dose': np.random.rand(1000) * 30}
        for _ in range(n_patients)
    ]
    original_plans = [
        {'tumor_dose': plan['tumor_dose'] - 5,
         'oar_dose': plan['oar_dose'] + 2}
        for plan in adapted_plans
    ]

    outcome_metrics = validator.evaluate_clinical_outcomes(adapted_plans
    , original_plans)

    # Computational efficiency
    processing_times = [
        {
            'registration': 15 + np.random.randn() * 2,
            'feature_extraction': 5 + np.random.randn(),
            'classification': 2 + np.random.randn() * 0.5,
            'optimization': 30 + np.random.randn() * 5,
            'qa': 3 + np.random.randn() * 0.5
        }
        for _ in range(n_patients)
    ]

    comp_metrics = validator.evaluate_computational_efficiency(
    processing_times)

    # Generate report
    all_metrics = {
        'classification': class_metrics,
        'dose_quality': dose_metrics,
        'clinical_outcomes': outcome_metrics,
        'computational_efficiency': comp_metrics
    }

    report = validator.generate_validation_report(all_metrics)
    print(report)
```

Listing 10: Pipeline Evaluation Metrics and Validation

4 Project Timeline

Table 1: Detailed project timeline (36 months)

Period	Task	Activities
Months 1-3	Literature Review	Comprehensive review, establish baseline knowledge
	Data Collection	Acquire anonymized patient datasets
	Setup	Development environment, computational resources
Months 4-9	Task 1	Synthetic image generation development & validation
	Training	Train diffusion models and GANs
	Validation	Expert review of synthetic images
Months 10-15	Task 2	Feature extraction pipeline implementation
	Model Training	Train multimodal response classifier
	Population Model	Build anatomical variation model
Months 16-21	Task 3	Dose optimization algorithms
	Integration	Connect with TPS
	RL Development	Train sequential adaptation agent
Months 22-27	Task 4	Complete pipeline integration
	Testing	In-silico validation on retrospective data
	Secondment 1	NTNU (Norway) - 3 months
Months 28-30	Secondment 2	Politecnico di Milano (Italy) - 3 months
Months 31-33	Analysis	Final results analysis
	Publications	Manuscript preparation
Months 34-36	Thesis Writing	PhD thesis completion
	Defense	Thesis defense preparation

5 Expected Outcomes and Impact

5.1 Scientific Contributions

1. **Novel methodology** for distinguishing anatomical from biological image changes in adaptive radiotherapy
2. **Validated AI models** for multimodal response characterization
3. **Optimization framework** for response-guided dose adaptation
4. **Clinical decision support system** for adaptive proton therapy

5.2 Clinical Impact

- Improved treatment outcomes through personalized adaptation

- Reduced workload through automated response categorization
- Better utilization of biological imaging information
- Framework for future integration of molecular biomarkers

5.3 Publications Plan

1. **Paper 1:** "Synthetic Image Generation for Adaptive Radiotherapy Training" (Months 10-12)
2. **Paper 2:** "AI-Based Distinction of Anatomical and Biological Changes in Proton Therapy" (Months 18-20)
3. **Paper 3:** "Response-Guided Dose Optimization Strategies" (Months 24-26)
4. **Paper 4:** "Clinical Validation of Integrated Adaptive Pipeline" (Months 32-34)

6 Research Environment and Training

6.1 Primary Institution

Aarhus University & Aarhus University Hospital

- State-of-the-art Danish Centre for Particle Therapy
- "AI and Big Data in Radiation Oncology" research group
- Access to clinical proton therapy data
- Integration with treatment planning systems

6.2 Secondments

Norwegian University of Science and Technology (NTNU)

- Expertise in medical image analysis
- Collaboration on deep learning methods
- Duration: 3 months

Politecnico di Milano

- Expertise in optimization algorithms
- Collaboration on dose planning methods
- Duration: 3 months

7 Ethical Considerations

All research will adhere to strict ethical guidelines:

- Use of fully anonymized retrospective patient data
- Approval from institutional review boards
- Compliance with GDPR regulations
- No patient identifiable information in publications
- Validation on synthetic/retrospective data before clinical use

8 Conclusion

This PhD project addresses a critical unmet need in adaptive radiotherapy: the systematic distinction between anatomical and biological components of image changes during treatment. By developing AI-driven methods integrating synthetic image generation, multimodal feature analysis, response classification, and adaptive dose optimization, this work will enable truly personalized "right-time" adaptive proton therapy.

The project aligns perfectly with the RAPTORplus consortium's mission and will contribute significantly to the advancement of precision radiation oncology. The combination of cutting-edge AI methods, comprehensive clinical data, and strong collaborative network positions this research for high impact in both scientific and clinical domains.

References

[1] Paganetti, H. (2012). Range uncertainties in proton therapy and the role of Monte Carlo simulations. *Physics in Medicine & Biology*, 57(11), R99.

[2] Kurz, C., et al. (2021). Medical physics challenges in clinical MR-guided radiotherapy. *Radiation Oncology*, 15, 93.

[3] Parodi, K., & Polf, J. C. (2019). In vivo range verification in particle therapy. *Medical Physics*, 45(11), e1036-e1050.

[4] Thompson, R. F., et al. (2023). Artificial intelligence in radiation oncology: A specialty-wide disruptive transformation? *Radiotherapy and Oncology*, 129(3), 421-426.

[5] Cardenas, C. E., et al. (2019). Deep learning algorithm for auto-delineation of high-risk oropharyngeal clinical target volumes with built-in dice similarity coefficient parameter optimization function. *International Journal of Radiation Oncology* Biology* Physics*, 101(2), 468-478.

[6] Nguyen, D., et al. (2019). A feasibility study for predicting optimal radiation therapy dose distributions of prostate cancer patients from patient anatomy using deep learning. *Scientific Reports*, 9(1), 1076.

[7] Kearney, V., et al. (2020). DoseNet: a volumetric dose prediction algorithm using 3D fully-convolutional neural networks. *Physics in Medicine & Biology*, 63(23), 235022.

[8] Lambin, P., et al. (2017). Radiomics: the bridge between medical imaging and personalized medicine. *Nature Reviews Clinical Oncology*, 14(12), 749-762.

[9] Liang, X., et al. (2022). Deep learning-based anatomical change prediction for adaptive radiotherapy. *Medical Physics*, 49(1), 310-324.

[10] Niemierko, A. (1999). A generalized concept of equivalent uniform dose. *Medical Physics*, 26(6), 1100.

[11] Zhou, M., et al. (2021). Radiogenomics in radiation therapy: A promising approach for personalized treatment. *Frontiers in Oncology*, 11, 624510.

[12] Yi, X., Walia, E., & Babyn, P. (2019). Generative adversarial network in medical imaging: A review. *Medical Image Analysis*, 58, 101552.

[13] Kazerouni, A., et al. (2023). Diffusion models in medical imaging: A comprehensive survey. *Medical Image Analysis*, 88, 102846.

[14] Brock, K. K., et al. (2017). Use of image registration and fusion algorithms and techniques in radiotherapy. *Medical Physics*, 44(7), e43-e76.

[15] Gillies, R. J., Kinahan, P. E., & Hricak, H. (2016). Radiomics: Images are more than pictures, they are data. *Radiology*, 278(2), 563-577.

[16] Fave, X., et al. (2017). Delta-radiomics features for the prediction of patient outcomes in non–small cell lung cancer. *Scientific Reports*, 7(1), 588.

[17] Bodalal, Z., et al. (2019). Radiogenomics: bridging imaging and genomics. *Abdominal Radiology*, 44(6), 1960-1984.

[18] Yan, D., et al. (2008). Adaptive radiation therapy. *Physics in Medicine & Biology*, 42(1), 123-132.

[19] Woodford, C., et al. (2007). Adaptive radiotherapy planning on decreasing gross tumor volumes. *International Journal of Radiation Oncology* Biology* Physics*, 69(4), 1316-1322.

[20] Gillison, M. L., et al. (2019). Radiotherapy plus cetuximab or cisplatin in human papillomavirus-positive oropharyngeal cancer. *The Lancet*, 393(10166), 40-50.

[21] Shen, C., et al. (2021). Intelligent inverse treatment planning via deep reinforcement learning. *Medical Physics*, 48(4), 2110-2124.