Computer Vision Mini Project Report

Abstract

In this report, we address a semantic segmentation task involving three classes: cats, dogs, and background. We develop and compare models based on encoder–decoder architectures, including U-Net, autoencoder-based variants, and CLIP-feature-enhanced designs. The best-performing model (U-Net) achieves an Intersection over Union (IoU) of 0.7992. Its robustness is evaluated under multiple perturbations. Additionally, the U-Net structure is adapted to support a prompt-based segmentation interface using points or boxes.

1. Introduction

This report addresses the task of semantic segmentation on the Oxford-IIIT Pet Dataset, focusing on classifying pixels into cats, dogs, and background. We implement and compare three segmentation models, U-Net, autoencoder-based, and CLIP-enhanced architectures, to identify the most effective approach in terms of accuracy, robustness, and adaptability.

To tackle dataset challenges such as class imbalance and limited diversity, we apply targeted preprocessing and augmentation. We also extend the system with a prompt-based segmentation interface, enabling user-guided refinement through points or boxes. The report presents the methodology, experiments, results, and an interactive user interface, providing a comprehensive evaluation of the strengths and limitations of each approach.

2. Data Preprocessing and Augmentations

2.1. Data Introduction

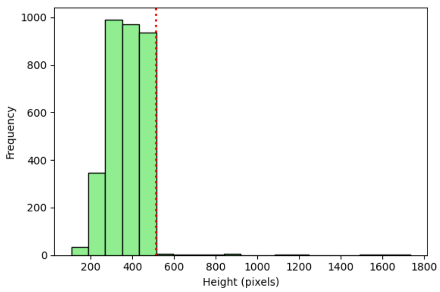
Overall review. This project utilizes a filtered subset of the Oxford-IIIT Pet Dataset. The dataset consists of two primary subsets: Train&Val (3,673 images) and Test (3,694 images). Each image is accompanied by a corresponding RGB segmentation mask. The segmentation task defines four pixel classes, each with a specific RGB colour mapping: **black** (0, 0, 0) represents the background, **white** (255, 255, 255) denotes uncertain regions, **red** (128, 0, 0) corresponds to cats, and **green** (0, 128, 0) indicates dogs.

Uncertain regions, frequently located around cat boundaries and including elements such as clothing, accessories, or watermark overlays, are regarded as noise and hence treated as background during preprocessing. The RGB segmentation masks, initially unsuitable for loss calculations due to their 0–255 value range, are converted into discrete class maps, encoding the background (black and white regions) as class 0, cats as class 1, and dogs as class 2 (detailed in *preprocessing.py*, Appendix C.2). To prevent information leakage, the Test set remains entirely isolated from training and validation processes. The Train&Val subset is further divided into training and validation subsets in a 9:1 ratio (Appendix C.4), with validation solely used for model selection.

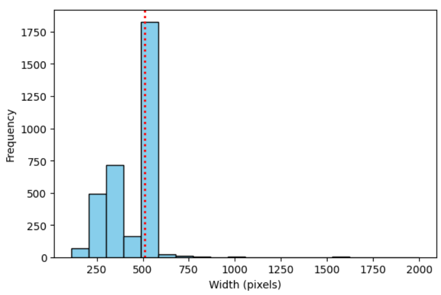
Reshape. Due to varying dimensions across images, batch operations and training efficiency are compromised. Consequently, the first preprocessing step standardizes all images and masks to a uniform resolution of (512×512). Figure 1 demonstrates that only 1.27% images exceed this dimension, either width or height bigger than 512. It implies minimal information loss from resizing. The (512×512) resolution was specifically chosen to:

1. Ensure compatibility with standard convolutional neural network architectures (e.g., U-Net), which commonly use square-shaped inputs.
2. Facilitate valid feature map sizes through the downsampling and upsampling stages typical in encoder-decoder segmentation models, by adopting dimensions divisible by 16.

Resizing is performed using bilinear interpolation for images to maintain visual consistency, while nearest-neighbor interpolation is used for masks to preserve discrete class labels accurately. Despite these preparations, the dataset remains limited in both size and class balance — with a significant skew toward dog images and fewer variations in appearance or lighting. These issues motivate the need for targeted data augmentation strategies, detailed in the following subsection.

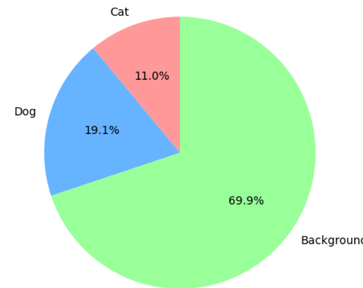


(a) Histogram of image heights in the raw training set. The red dotted line indicates the 512-pixel reshape target.

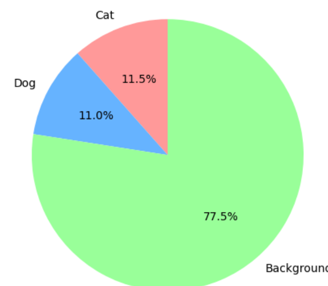


(b) Histogram of image widths in the raw training set. The red dotted line indicates the 512-pixel reshape target.

Figure 1. Distributions of image dimensions in the raw training dataset. A threshold of 512 pixels is highlighted to evaluate how many images exceed standard input size constraints.

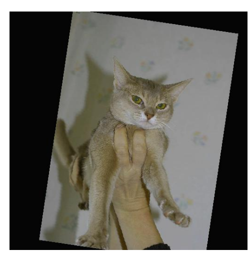


(a) Pixel class distribution of raw training dataset. Image-wise, there are 1055 cat images and 2250 dog images

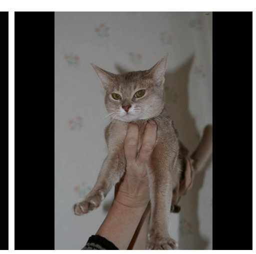


(b) Pixel class distribution of augmented dataset. Image-wise, there are 8440 cat images and 9000 dog images

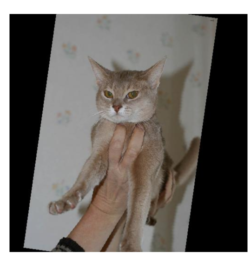
Figure 2. Pixel-wise class distributions of raw and augmented training datasets.



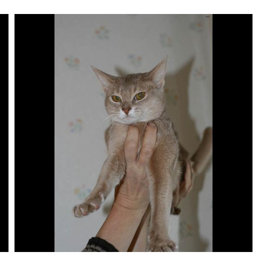
(a) Jittering



(b) Brightness adjustment



(c) Rotation + translation



(d) Horizontal flip

Figure 3. Examples of data augmentations applied to the same image. Note that multiple augmentation techniques are often applied together; the subcaptions indicate the most representative transformation in each case.

2.2. Data Augmentation

To address the dataset limitations discussed in Section 2.1, this augmentation pipeline serves two primary goals: 1) mitigating class imbalance; and 2) expanding the training dataset to improve model accuracy, generalisation, and robustness.

Class Imbalance Handling. The original dataset exhibited significant imbalance (Figure 2a), with dogs dominating both image and pixel counts. To counter this, we applied heavier augmentation to cat images—doubling their count relative to dogs—resulting in a more balanced dataset (Figure 2b).

Augmentation Strategy. To simulate real-world variability, we applied spatial and photometric augmentations that preserved mask alignment. These include: 1) horizontal flips, 2) rotations, translations, and scaling, 3) brightness and contrast adjustments, 4) color jittering, and 5) sparse Gaussian blur. They simulate (examples seen in Figure 3) real-world variations and common image defects that a model may encounter in deployment to ensure the generalization and robustness of trained model.

These transformations were applied with moderate probability, ensuring that each augmented sample retained core semantic structure while introducing meaningful variation. All operations were carefully configured to avoid over-distortion or label corruption, particularly important in pixel-wise segmentation tasks where precise spatial correspondence between image and mask must be preserved.

Although the image labels are converted to class maps with values of 0, 1, and 2 (Section 2.1), the input colour images initially have pixel values in the range of 0 to 255. This wide range is not ideal for training deep neural networks, as it may lead to unstable gradients and slower convergence. Therefore, all input images are normalized to the range [0,1], improving numerical stability and facilitating more effective learning.

Outcome. Through this augmentation pipeline, the number of training images increased from 3,305 to 17,440. The class distribution became more balanced, and the diversity of training examples was significantly enhanced. This not only reduces the risk of overfitting but also equips the model to perform better in real-world conditions, especially on underrepresented cat images and in visually noisy or inconsistent scenes.

3. Segmentation networks design and implementation

3.1. Model Size and Selection

For an effective and fair comparison of segmentation performance across various architectures (see Section 4), all models were constrained to a similar parameter count of approximately 31×10^6. A model size around 31 million parameters is commonly recognized in the literature as a balanced choice, effectively striking a trade-off between computational efficiency, memory usage, and segmentation accuracy, making it suitable for practical deployment on typical GPU hardware (He et al., 2016; Ronneberger et al., 2015). Specifically, this scale provides sufficient representational capacity without excessive computational demands, thus facilitating thorough experimentation within resource-limited settings typical for academic and applied research.

To achieve consistency in model comparisons, the parameter sizes of the U-Net, autoencoder-based, and CLIP-based segmentation models were explicitly matched. This uniformity ensures that observed performance differences primarily reflect architectural effectiveness rather than capacity disparities. The U-Net architecture, which requires training all parameters, establishes a baseline parameter count. In contrast, autoencoder and CLIP-based models incorporate pre-trained, frozen encoders, thereby significantly reducing the number of parameters that require training while maintaining comparable total model sizes.

We specifically selected ResNet-50 (RN50) as the visual backbone for the CLIP-based model due to several critical advantages. Compared to Vision Transformer (ViT) architectures like ViT-B/32, ResNet-50 employs smaller convolutional kernels (3×3), offering enhanced spatial precision essential for pixel-wise segmentation tasks. Moreover, RN50's visual encoder consists of approximately 23.53×10^6 parameters, enabling adequate capacity for decoder components within the fixed parameter budget, thus aligning with the controlled experiment criteria established. Detailed parameter distributions for each model variant are summarized in Table 1.

Model	Config Size (MB)	Params #
U-Net	118.42	31,043,651
Autoencoder	119.59	31,350,275
CLIP	116.89	30,642,915

Table 1. Comparison of models based on configuration size and parameter count

3.2. Model Architecture

3.2.1 U-Net-based End-to-End Neural Network

The implemented U-Net segmentation model follows the established encoder-decoder architecture proposed by Ronneberger et al. [6]. Specifically adapted for RGB image segmentation, the network accommodates three input channels corresponding to preprocessed colour images (see Section 2.2), predicting pixel-level class maps during training. The encoder compresses input images into latent features with 1024 channels, using convolutional blocks followed by max-pooling operations (Figure 4). Each convolutional block contains two convolutional layers, batch normalization (BN), and rectified linear unit (ReLU) activation functions. BN stabilizes the learning process by normalizing inputs to each layer, which reduces internal covariate shift and accelerates convergence [2]. The ReLU activation function is used due to its computational efficiency, reduced risk of vanishing gradients, and effectiveness in promoting sparsity in activations, which aids generalization [3]. The decoder upsamples the latent features symmetrically using transpose convolutional layers. Skip connections between corresponding encoder and decoder blocks (grey arrows in Figure 4) mitigate gradient vanishing and enable the model to leverage both deep and shallow features effectively [6]. The final segmentation output is produced through a convolutional layer mapping to the segmentation class channels (Appendix D.1).

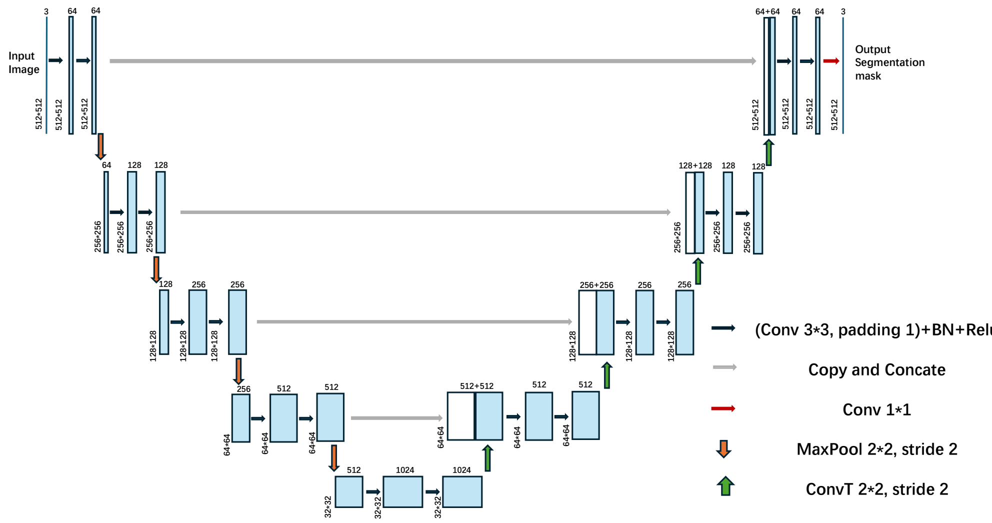


Figure 4. U-Net architecture used for baseline segmentation.

3.2.2 Autoencoder Pre-training Segmentation

The autoencoder-based pre-training model capitalizes on unsupervised feature extraction by initially training the network to reconstruct input images, approximating the identity function [4]. This unsupervised learning phase employs the mean squared error (MSE) loss, effectively encouraging the encoder to capture robust latent representations of the original data. The encoder sequentially reduces spatial dimensions using convolutional layers with strides, combined with BN and ReLU activations for stability and non-linearity. The decoder symmetrically reconstructs images using transpose convolutional layers (top row, Figure 5). Post pretraining, the encoder's weights are frozen, serving as a feature extractor in a supervised segmentation network. A dedicated segmentation decoder, consisting of transpose convolutional layers, directly translates pretrained latent representations into segmentation masks (Appendix D.2). This approach aligns with previous research demonstrating that unsupervised pre-training significantly improves downstream supervised segmentation performance by providing effective initialization and richer feature representations [1, 7].

3.2.3 CLIP-based Segmentation

The CLIP-based segmentation model integrates pretrained visual representations from CLIP (Contrastive Language-Image Pre-training) [5], exploiting task-agnostic, semantically rich features extracted from its visual backbone. The weights of CLIP's visual encoder are maintained frozen to preserve general feature representations beneficial for downstream tasks. These features are then processed through a lightweight decoder designed to progressively recover the original image resolution using transpose convolutions and bilinear interpolation, thereby enhancing the spatial precision of the segmentation maps (Figure 6).

This architectural choice is supported by prior research indicating that leveraging pre-trained, general-purpose representations (such as those from CLIP) significantly improves semantic segmentation tasks by incorporating broad contextual and semantic insights [5, 9]. The use of transpose convolutions and bilinear interpolation strategically balances computational efficiency and output quality, ensuring accurate segmentation while maintaining manageable computational demands (Appendix D.3).

4. Experiments, Evaluation and Comparison

4.1. Experimental Settings

Data. We use the filtered dataset described in Sections 2.1 and 2.2, which contains RGB images of cats and dogs with pixel-wise segmentation masks. The original TrainVal set is split into training and validation subsets in a 9:1 ratio, detailed in Appendix C.4. All images and masks are resized to 512×512, and the input images are normalized to the range $[0, 1]$. During training, extensive augmentations, including flipping, rotation, scaling, brightness and contrast adjustments, and Gaussian blurring, are applied. To mitigate class imbalance, the number of augmented cat images is doubled, resulting in a final training set of 17,440 images.

Models. The candidate segmentation models include

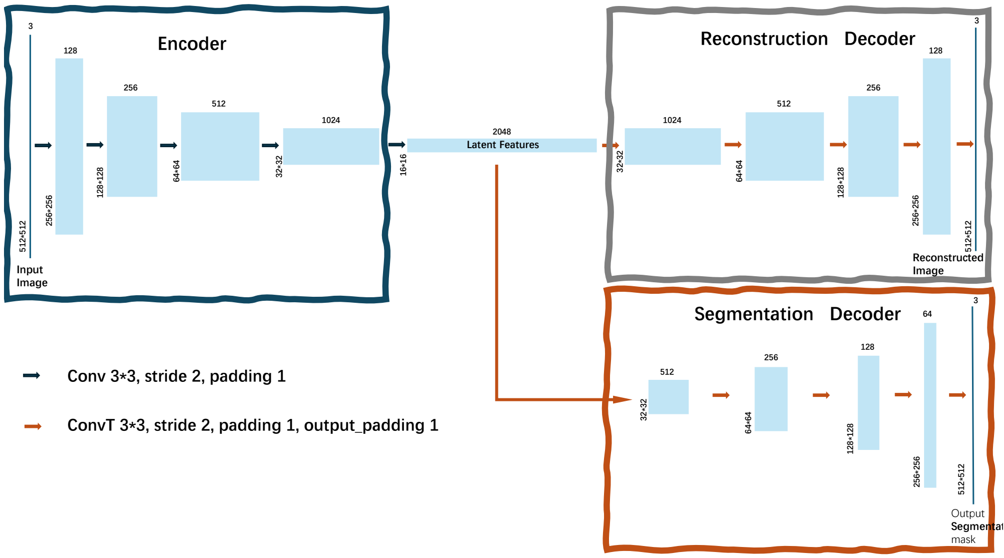


Figure 5. Autoencoder-based segmentation model with a frozen encoder.

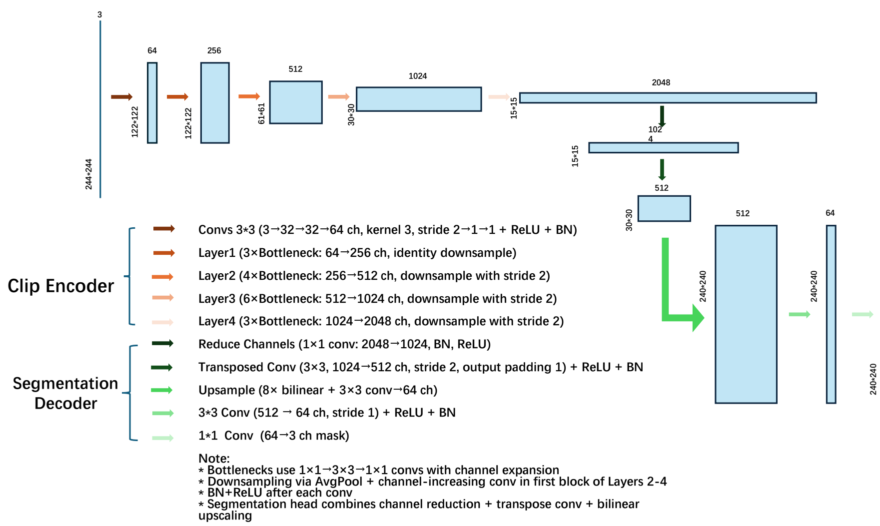


Figure 6. CLIP-enhanced model incorporating prompt-based feature guidance.

U-Net, Autoencoder, and CLIP-based architectures, as described in Section 3. For a fair comparison, the total number of parameters—including both trainable and frozen components—is controlled to be approximately 31 million for all models, as detailed in Section 3.1.

Hyperparameters. For fair comparison, we fix the learning rate to 1×10^{-4}, batch size to 4, epoch to 30 and use the Adam optimizer with a fixed random seed of 42. Each model checkpoint's selection is based on cross-validation using the mean IoU, where historical checkpoints are evaluated to select the one with the best validation performance. All training and validation processes are done on one GPU-4070s.

Loss functions. To effectively supervise this 3-class segmentation task, we evaluated multiple loss functions, including cross-entropy, Dice Loss, and IoU Loss. Ultimately, a composite of Focal Loss and Dice Loss was selected to jointly address class imbalance and region-level segmentation accuracy.Given predicted logits $\mathbf{z}_i \in \mathbb{R}^C$ at each pixel i, the *Focal Loss* is defined below and coded in Appendix E.3

$$\mathcal{L}_{\text{Focal}} = -\frac{1}{N}\sum_{i=1}^{N} \alpha_{y_i} \cdot (1 - \hat{p}_{i,y_i})^{\gamma} \cdot \log(\hat{p}_{i,y_i}) \quad (1)$$

where:

- N is the total number of pixels,
- $y_i \in \{0, 1, 2\}$ is the ith ground truth class index,
- $\hat{p}_{i,c} = \frac{\exp(z_{i,c})}{\sum_{k=1}^{C} \exp(z_{i,k})}$ is the softmax probability for class c at pixel i,
- $\hat{p}_{i,y_i}$ is the probability assigned to the true class,
- γ is the focusing parameter,
- α_{y_i} is an optional weighting factor for class y_i (can be scalar or class-dependent).

This formulation focuses on training on misclassified pixels by down-weighting the loss contribution of well-classified examples. The loss is averaged over all pixels.

The *Dice Loss* is defined as:

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_i y_i + \epsilon}{\sum_i p_i + \sum_i y_i + \epsilon} \tag{2}$$

where p_i and y_i are the predicted and ground truth values respectively, and ϵ is a small constant to avoid division by zero.

The **combined loss function** is:

$$\mathcal{L} = (1 - \alpha) \cdot \mathcal{L}_{\text{Focal}} + \alpha \cdot \mathcal{L}_{\text{Dice}} \tag{3}$$

where $\alpha \in [0, 1]$ is a tunable parameter that controls the balance between the two components.

Note: Intersection-over-Union (IoU) was used as the validation metric to align with the baseline evaluation protocol.

Training. For the U-Net model, we apply full end-to-end training, where all parameters are learned from scratch with combined loss function as defined in Equation 3.

For the autoencoder-based model, training is divided into two phases. During the pre-training phase, the encoder is trained as a reconstruction network using the mean squared error (MSE) loss:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N} \|x_i - \hat{x}_i\|^2 \tag{4}$$

where x_i and $\hat{x}_i$ denote the original and reconstructed images, respectively. After pre-training, the encoder weights are frozen, and a segmentation decoder is trained on top using the same segmentation loss as the U-Net model (Equation 3).

In the CLIP-based model, the visual encoder (CLIP-RN50) remains frozen throughout training, while a lightweight decoder is trained to map high-level CLIP features to segmentation masks with the same segmentation loss function in Equation 3.

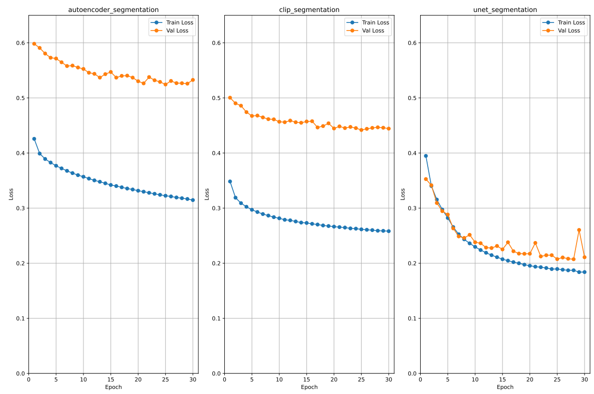


Figure 7. Curves for training loss(3) and validation loss (2). Plots generated by codes in Appendix E.2.

Validation and Model Selection. During training, we apply cross-validation with a fixed train–validation split ratio of 9:1. To select the best-performing checkpoints, we use the validation loss defined as the pure IoU loss, computed as $(1 - \text{IoU})$:

$$\mathcal{L}_{\text{val}} = 1 - \text{IoU} \tag{5}$$

This choice aligns with the final evaluation metric, ensuring consistency when comparing the three models.

After training, the model achieving the highest validation IoU score is selected as the final model for each architecture. The IoU is calculated as:

$$\text{IoU} = \frac{TP}{TP + FP + FN} \tag{6}$$

where TP, FP, and FN represent the number of true positive, false positive, and false negative pixels, respectively.

4.2. Quantitative Results and Analysis

Table 2 summarizes the segmentation performance of all models on the held-out validation set. The mean Intersection-over-Union (IoU) is reported, along with per-class IoU scores for background, cats, and dogs.

Model	Background IoU	Cats IoU	Dogs IoU	Mean IoU
U-Net	**0.9315**	**0.7234**	**0.7429**	**0.7992**
CLIP-based	0.8775	0.4864	0.5394	0.6344
Autoencoder	0.8003	0.2849	0.2171	0.4341

Table 2. Segmentation performance comparison on the validation set (per-class IoU and mean IoU).

Training Dynamics. Figure 7 shows the training and validation loss curves for the UNet, CLIP-based, and autoencoder segmentation models over 30 epochs. Among the three, UNet demonstrates the most efficient and effective training dynamics. It begins with a higher initial training loss (0.40) compared to CLIP-based (0.35) and autoencoder (0.42), but exhibits a steep and consistent decline in both training and validation losses early in training. This rapid convergence indicates that the UNet architecture is well-suited to the segmentation task, likely due to its strong spatial inductive bias. In contrast, the CLIP-based model improves more slowly and shows a larger gap between training and validation losses, suggesting a tendency to overfit and limited alignment with pixel-level segmentation tasks. The autoencoder performs the worst, with the highest initial loss and the smallest reduction over time, indicating weak generalization and poor suitability for this task.

Overall, the training curves further support the quantitative results in Table 2, showing that U-Net not only achieves the best final IoU but also exhibits the most stable and effective training behavior.

U-Net. The U-Net model achieves the highest overall performance with a mean IoU of 0.7992. It significantly outperforms the other models across all classes, particularly in segmenting cats (0.7234) and dogs (0.7429). This result demonstrates the benefit of end-to-end training and its encoder–decoder architecture with skip connections, effectively preserving spatial details crucial for pixel-level segmentation. As shown in Figure 4, U-Net produces segmentations with well-defined edges and clear separation between foreground and background. However, a notable failure case is its misclassification of the chair beneath the cat as part of the dog, which suggests a limitation in distinguishing object context and inter-class relationships, especially when objects are physically close or visually similar. This may point to U-Net's reliance on local texture and shape cues rather than broader semantic understanding.

CLIP-Featured. The CLIP-based model achieves a mean IoU of 0.6344, inferor from the performence of the Unet. The model performs reasonably well on cats and dogs (0.4864 and 0.5394, respectively), highlighting the value of using pre-trained visual features. However, performance still lags behind U-Net, likely due to CLIP's relatively coarse patch-wise representation and limited spatial granularity. These characteristics make it more difficult to capture object boundaries precisely. As shown in Figure 6, the CLIP-based model identifies most of the cat's region but struggles with accurately delineating edges. This imprecision at boundaries may lead to merging or loss of fine structural information, particularly around thin or small regions like tails or legs.

Autoencoder. The autoencoder-based model shows the weakest performance, with a mean IoU of only 0.4341, and particularly poor results on cats (0.2849) and dogs (0.2171). These numbers reflect the model's limited ability to capture task-specific features necessary for semantic segmentation.. The encoder, pre-trained for image reconstruction, likely focuses on global appearance and texture rather than precise object delineation, which hinders its utility for dense labeling. As shown in Figure 5, reconstructions from the autoencoder appear blurred, lacking critical spatial details, and resulting in fragmented, blocky segmentation outputs. Such fragmentation can be directly attributed to the loss of spatial resolution in the latent representations, compounded by the absence of architectural features like skip connections, essential for preserving high-frequency spatial information. Consequently, segmentation masks produced by this model are inconsistent, patchy, and inadequate for accurately segmenting smaller or complex-shaped regions.

5. Robustness Exploration

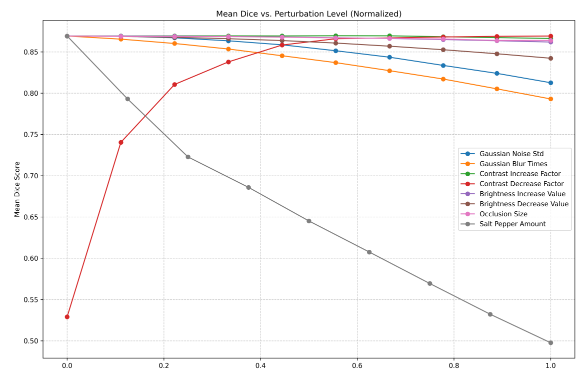


Figure 9. Mean Dice score across all perturbations

We further explore the robustness of the best-performing model (selected based on the validation dataset in Section 4) by applying eight types of perturbations to the test set:

1. **Gaussian pixel noise**: Zero-mean Gaussian noise with varying standard deviations is added independently to each pixel channel. The noise is sampled from a normal distribution $\mathcal{N}(0, \sigma^2)$ and clipped to remain within the valid image range $[0, 255]$.

2. **Gaussian blurring**: A fixed 3×3 Gaussian kernel is convolved multiple times with the image. The number of convolution iterations determines the blurring strength.

3. **Image contrast increase**: Pixel intensities are multiplied by a factor $\alpha > 1$, effectively stretching the dynamic range. The result is clipped to $[0, 255]$ to remain a valid image.

4. **Image contrast decrease**: Similar to the increase operation, but with a factor $0 < \alpha < 1$, reducing

the dynamic range and making the image appear more muted.

5. **Image brightness increase**: A fixed value is added to all pixel channels, uniformly brightening the image. Clipping ensures that pixel values remain within $[0, 255]$.

6. **Image brightness decrease**: A fixed value is subtracted from all pixel channels, uniformly darkening the image while also applying clipping.

7. **Image occlusion**: A square patch of specified size is randomly placed within the image and replaced with black pixels $(0, 0, 0)$, simulating partial obstruction.

8. **Salt-and-pepper noise**: A proportion of pixels is randomly replaced with either black $(0, 0, 0)$ or white $(255, 255, 255)$, simulating impulse noise.

These perturbations are applied at multiple severity levels to generate a range of distorted test datasets. The performance degradation curves under these perturbations are presented in Figures 9 and 10.

The model demonstrates strong robustness to brightness increase, contrast enhancement, and occlusion, showing only a slight Dice score drop (0.02 from 0.875) across perturbation levels.

In contrast, brightness reduction, Gaussian blur, and Gaussian noise cause exponential performance degradation, lowering the Dice score from 0.87 to 0.80. Contrast decrease and salt-and-pepper noise lead to the most severe drops: the former declines exponentially from 0.87 to 0.53, while the latter shows a linear decline to 0.50.

To explore worst-case scenarios, we assess one large object (dog) and one small object (cat) from the perturbed test set under the highest severity level (Figure 11).

For the dog, predictions remain close to the ground truth under benign perturbations (brightness/contrast increase, occlusion), indicating robustness—likely due to the large spatial context aiding segmentation even when corrupted.

In contrast, the cat example—representing smaller, low-contrast regions—suffers more. Brightness/contrast increase causes misclassification (e.g., the white cat's black tail blends with the background), and occlusion hides critical regions. This suggests the model heavily relies on local texture and contrast.

With brightness reduction, noise, and blur, the dog's predictions lose boundary precision, showing the importance of edge information. Interestingly, brightness reduction improves contrast for the cat, slightly aiding performance.

Gaussian blur severely impacts both examples by erasing fine details, leading to boundary erosion and shape distortion. Contrast decrease and salt-and-pepper noise result in sparse, fragmented predictions, highlighting the model's reliance on intensity differences to delineate objects.

6. Prompt-based segmentation

Based on the best model architecture UNet as we discussed in the previous section, we develop a prompt-based segmentation model and build a UI for users to interact with their uploaded images.

6.1. Model Architecture

A prompt-based segmentation approach was implemented using a custom-designed point prompt encoder, which encodes spatial prompts (heatmaps) into feature embeddings. This encoder incorporates convolutional layers to transform prompt heatmaps into spatial representations, which are then concatenated with image features extracted from a simple convolutional backbone. The combined feature maps are processed through a decoder comprising convolutional layers to generate the segmentation masks. This approach allows for guided segmentation, leveraging external prompt information to improve segmentation accuracy.

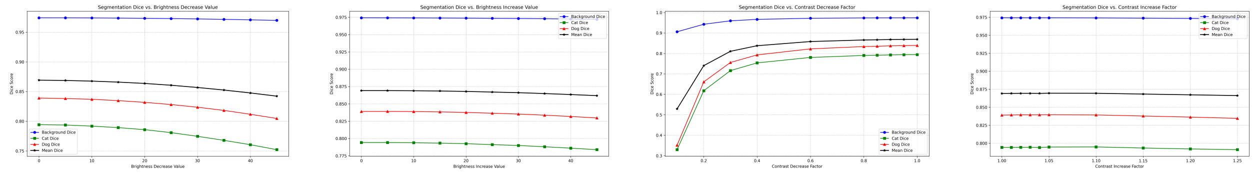
6.2. Training

Data with Prompt. The data preprocessing involved generating spatial prompts (heatmaps) based on annotated points or bounding boxes provided in prompt files. During preprocessing, points or bounding boxes were randomly sampled from mask labels, balancing between foreground and background classes. These sampled points were then converted into Gaussian heatmaps with a specified sigma value . When bounding boxes were used, the heatmap consisted of a rectangular region corresponding to the box area. The final processed dataset thus consisted of RGB images, corresponding segmentation masks, and spatial prompt heatmaps to guide the segmentation model.

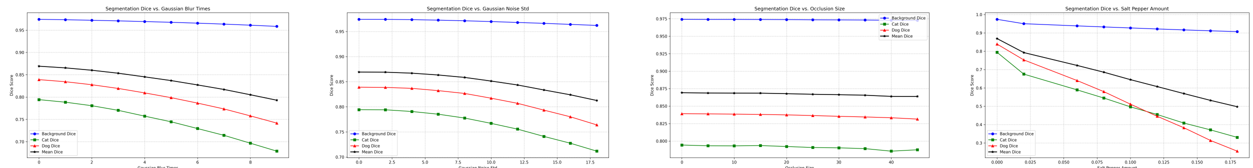
Loss Function. To train the model, we used the binary cross-entropy (BCE) loss, which is suitable for binary segmentation tasks. It compares the predicted probability $\hat{y}_i$ for each pixel with the ground truth label y_i, penalizing incorrect predictions. The loss is defined as:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{7}$$

where N is the total number of pixels, $y_i \in \{0, 1\}$ is the ground truth label, and $\hat{y}_i \in [0, 1]$ is the predicted probability of the foreground class.

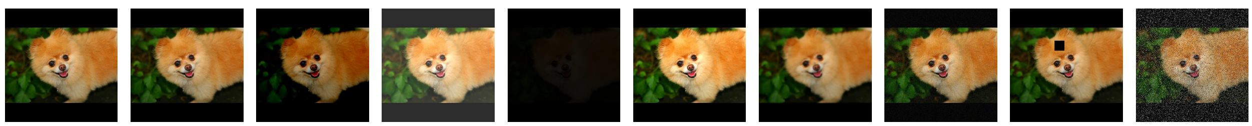


(a) Brightness Decrease (b) Brightness Increase (c) Contrast Decrease (d) Contrast Increase

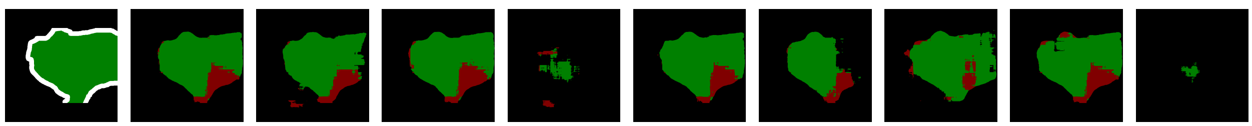


(e) Gaussian Blur (f) Gaussian Noise (g) Occlusion (h) Salt & Pepper Noise

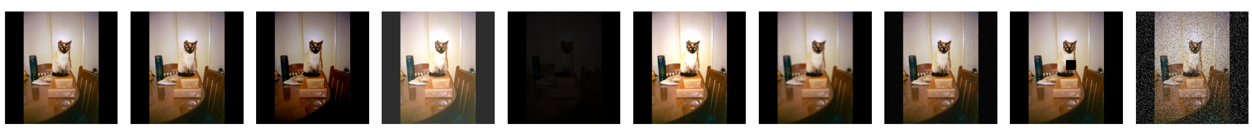
Figure 10. Dice scores under individual perturbation types



Original Original Brightness - Brightness + Contrast - Contrast + Blur Noise Occlusion salt&pepper



GT Original Brightness - Brightness + Contrast - Contrast + Blur Noise Occlusion salt&pepper



Original Original Brightness - Brightness + Contrast - Contrast + Blur Noise Occlusion salt&pepper

GT Original Brightness - Brightness + Contrast - Contrast + Blur Noise Occlusion salt&pepper

Figure 11. Visual comparison of model robustness under different perturbations. Each column corresponds to a perturbation type at its highest severity level: **Brightness:** pixel value shift (±45), **Contrast:** multiplicative scaling (×0.10 / ×1.25), **Gaussian Blur:** repeated convolution (×9), **Gaussian Noise:** standard deviation σ = 18, **Occlusion:** 45×45 black square, **Salt & Pepper Noise:** density = 0.18, Rows 1 & 2: dog input and prediction; Rows 3 & 4: cat input and prediction; Ground truths are shown in the first column of Rows 2 & 4.

Training Settings. The training was conducted using a custom PyTorch training loop specifically designed for prompt-based segmentation. The model input includes two components: RGB images, and generated prompt heatmaps. The output of the model is the predicted segmentation mask. Training was performed using early stopping based on validation loss, with parameters optimized via Adam or a similar optimizer, leveraging GPU acceleration for efficient computation.

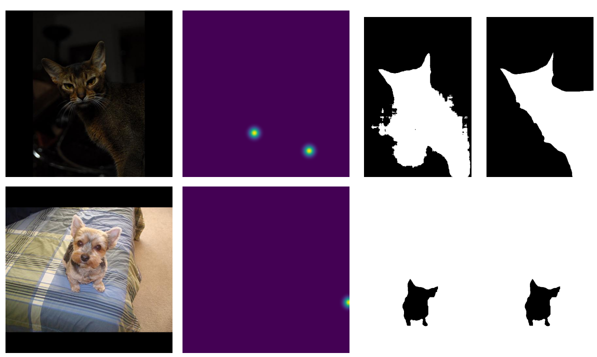
6.3. Test

We achieve a mean IoU of 0.8780 on the test set using randomly sampled point and box prompts. The model demonstrates strong generalization on unseen samples. Figure 12 shows visual results from the test set, including input images, heatmaps of prompts, predicted masks, and ground truth masks. The model effectively segments foreground and background regions using either point or box prompts, showing especially strong performance on background separation. However, the

Table 3. Mean IoU / Dice under different perturbation types at various unified levels

Level	Gaussian Noise	Gaussian Blur	Contrast+	Contrast–	Brightness+	Brightness–	Occlusion	Salt & Pepper
1	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692	0.7771/0.8692
2	0.7768/0.8690	0.7716/0.8654	0.7772/0.8693	0.7765/0.8688	0.7771/0.8692	0.7764/0.8687	0.7764/0.8687	0.6731/0.7930
3	0.7739/0.8670	0.7640/0.8601	0.7773/0.8694	0.7754/0.8680	0.7768/0.8690	0.7748/0.8676	0.7762/0.8686	0.6325/0.7596
4	0.7686/0.8634	0.7545/0.8535	0.7773/0.8694	0.7738/0.8669	0.7762/0.8686	0.7724/0.8660	0.7761/0.8685	0.5909/0.7227
5	0.7617/0.8585	0.7431/0.8453	0.7772/0.8693	0.7723/0.8659	0.7753/0.8680	0.7692/0.8637	0.7750/0.8678	0.5523/0.6859
6	0.7515/0.8514	0.7317/0.8370	0.7776/0.8695	0.7613/0.8583	0.7740/0.8671	0.7649/0.8607	0.7737/0.8669	0.5128/0.6452
7	0.7405/0.8435	0.7186/0.8272	0.7774/0.8694	0.7325/0.8378	0.7727/0.8662	0.7595/0.8569	0.7729/0.8664	0.4787/0.6075
8	0.7270/0.8336	0.7052/0.8171	0.7758/0.8683	0.6959/0.8105	0.7708/0.8649	0.7535/0.8526	0.7718/0.8656	0.4465/0.5694
9	0.7140/0.8240	0.6899/0.8052	0.7742/0.8673	0.6106/0.7404	0.7687/0.8635	0.7467/0.8477	0.7694/0.8639	0.4171/0.5322
10	0.6992/0.8126	0.6745/0.7930	0.7726/0.8662	0.4128/0.5289	0.7664/0.8619	0.7392/0.8423	0.7692/0.8638	0.3915/0.4976

model still exhibits instability under box prompts: edge predictions tend to be less accurate, and minor background regions are occasionally misclassified as foreground. This might be caused by the coarse nature of box prompts, which provide less precise spatial guidance compared to point annotations.



Input Image Prompt Heatmap Predicted Mask Ground Truth

Figure 12. Qualitative results on the test set using point and box prompts.

6.4. UI Implementation

The UI is built with PyQt5, providing an interactive interface for prompt-based image segmentation. Users can load images, annotate with multiple points and boxes to guide segmentation using mouse, and adjust the threshold via a slider. The interface includes a central canvas for visualization and a control panel with buttons for loading images, running segmentation, clearing prompts, and saving masks. The system overlays segmentation results on the image and allows toggling the view. It supports combining multiple prompts for refined segmentation in a single session, offering a smooth and userfriendly experience. The GIFs in the supplemental materials show the end-to-end user workflow, including multi-points, boxes prompts, model inference, and visual feedback.

7. Limitation and Conclusion

In this project, we addressed the task of pixel-wise semantic segmentation for cats, dogs, and background using a filtered version of the Oxford-IIIT Pet Dataset. We developed and compared three architectures under a unified parameter budget: a classic U-Net, an autoencoder-based model with frozen encoders, and a CLIP-enhanced segmentation model leveraging pretrained vision-language representations.

Among the three, the U-Net achieved the best performance, reaching a mean Intersection over Union (mIoU) of 0.7992. Its end-to-end training capability and skip connections allowed effective fusion of spatial details from early layers with deep semantic understanding, resulting in accurate segmentation even around complex object boundaries such as fur and ears.

In contrast, the CLIP-based model underperformed, highlighting several limitations. Although CLIP's ResNet-50 encoder provides semantically rich features, it was originally trained for image-text alignment rather than dense prediction. Consequently, the extracted features lack the fine spatial granularity required for accurate segmentation. This challenge is exacerbated by the use of a *shallow decoder*, which lacks sufficient capacity to refine and upsample coarse features back to pixel-level precision. As a result, segmentation boundaries tend to be blurry or misaligned. Prior works such as MaskCLIP [10] and SegCLIP [8] also report similar issues and address them through deeper or multi-scale decoders.

The autoencoder-based model with frozen encoders faced similar issues: limited adaptability and a lack of spatial refinement reduced its effectiveness for detailed segmentation.

We applied targeted data augmentation to improve generalisation, address class imbalance, and expand the dataset to over 17,000 training samples. Robustness testing showed that U-Net maintained strong performance under most perturbations, although all models exhibited degradation under conditions like blur and low contrast.

Finally, we extended the U-Net with a prompt-based interface, supporting interactive segmentation via user-provided points or bounding boxes. This extension demonstrates the model's potential for real-world applications in interactive editing and annotation tools.

Future work may explore enhanced decoders for CLIP-based models, fine-tuning strategies for pre-trained encoders, multi-scale architectures like FPN or DeepLab, and domain adaptation for cross-species generalization.

References

[1] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? In *Journal of Machine Learning Research*, volume 11, pages 625–660, 2010. 4

[2] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 448–456, 2015. 3

[3] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010. 3

[4] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011. 4

[5] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021. 4

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015. 3

[7] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. In *Journal of Machine Learning Research*, volume 11, pages 3371–3408, 2010. 4

[8] Yuchao Wang, Peihao Xu, Zhe Zhang, Xiang Xu, Yifan Wang, Yuhui He, and Xiaojuan Qi. Segclip: Leveraging clip for semantic segmentation. *arXiv preprint arXiv:2209.11888*, 2022. 10

[9] Bowen Zhou, Junjie He, Xiang Yao, Philipp Krahenbuhl, Vladlen Koltun, and Alexei A. Efros. Image segmentation using text and image prompts. *arXiv preprint arXiv:2112.10003*, 2022. 4

[10] Bowen Zhou, Chen Wang, Xinlong Wang, Dongdong Yu, Jifeng Dai, Yuning Lu, and Lu Yuan. Maskclip: Masked language-image pre-training for open-world semantic segmentation. *arXiv preprint arXiv:2208.08984*, 2022. 10

Appendix

A. Team Member Responsibilities

This appendix outlines the specific responsibilities and contributions of each team member. For tasks completed collaboratively, the individual portions are clearly specified.

Task	**Team Member A**	**Team Member B**
Data Preprocessing & Augmentation	Designed and implemented core preprocessing pipeline; generated balanced training datasets through augmentation	Developed preprocessing routines for prompt-based segmentation, including heatmap generation and spatial prompt encoding
Segmentation Model Development	Implemented and trained the autoencoder-based and CLIP-featured segmentation models	Implemented and trained the U-Net baseline and extended it for prompt-based segmentation
Evaluation & Analysis	Produced training plots and quantitative results for autoencoder and CLIP models	Conducted comparative evaluation and robustness analysis; visualized failure cases and strengths
Report Writing	Contributed to Sections 1–2 (Data preparation, Model design)	Contributed to Sections 3–4 (Evaluation, Robustness, Prompt-based extension)

Table 4. Breakdown of responsibilities by team member.

B. Code Structure Overview & Training Entry

B.1. Directory Tree

This appendix presents the structure and organization of the project source code. It outlines directories and scripts, highlighting their roles in dataset preprocessing, model architecture definitions, training logic, and inference routines. A central training script `main.py` provides a unified interface for training various segmentation models, including U-Net, AutoEncoder-based, CLIP-based, and prompt-enhanced variants. The modular design ensures clarity, flexibility, and ease of experimentation throughout the development pipeline.

```
src/
|-- data/
|   |-- __init__.py
|   |-- PetDataset.py                 % Dataset definitions for all segmentation
|                                       and prompt models
|   |-- preprocess_data_with_prompt.py  % Preprocess prompt-based datasets,
|                                         taking CLI args
|   |-- preprocess_data.py            % Preprocess segmentation datasets,
|                                       taking CLI args
|   |-- preprocessing.py              % General preprocessing functions
|                                       (class mapping, reshape, augmentor)
|   |-- train_val_split.py            % Dataset splitting utility
|-- models/
|   |-- unet_segmentation.py          % U-Net model definition
|   |-- autoencoder_segmentation.py   % AutoEncoder & Segmentation Head
|   |-- clip_segmentation.py          % CLIP-based segmentation model
|   |-- prompt_segmentation.py        % Prompt-enhanced U-Net
|-- utils/
|   |-- inference.py
|   |-- loss_functions.py             % Loss definitions (Focal, Dice, IoU)
```

```
|    |-- restore_image_size.py        % Restore predicted mask to original size
|    |-- training_plot.py             % Plot training/validation curves
|    |-- training.py                  % Training loops (standard & prompt)
|-- compute_IoU.py                    % Script to compute evaluation metric
|-- main.py                           % Entry point for training with CLI args
|-- run_inference.py                  % Perform model inference
```

B.2. Training Entry Script: `main.py`

The `main.py` script allows training four types of models via a unified CLI:

- **Mode 0:** U-Net-based segmentation
- **Mode 1:** AutoEncoder with optional pretraining
- **Mode 2:** CLIP-based segmentation (ResNet-50 backbone)
- **Mode 3:** Prompt-based segmentation using (x, y, class) inputs

The script handles argument parsing, dataset loading and splitting, model instantiation, training, and saving.
main.py

```python
#!/usr/bin/env python

import torch
import argparse
import clip
import torch.optim as optim
from pathlib import Path
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader
from data.preprocessing import *
from data.PetDataset import PetDataset, PetDatasetWithPrompt
# import unet related functions
from models.unet_segmentation import UNet
# import autoencoder related functions
from models.autoencoder_segmentation import (AutoEncoder, AutoEncoderSegmentation,
                                             pretrain_autoencoder)
# import CLIP related functions
from models.clip_segmentation import CLIPSegmentationModel
# import prompt based model
from models.prompt_segmentation import PromptSegmentation
from utils.loss_functions import CombinedFocalDiceLoss, BinaryFocalLoss, IouLoss
from utils.training_plot import training_plot
from utils.training import training, prompt_training

# Define Arguments
parser = argparse.ArgumentParser(description="Train segmentation model")
parser.add_argument("--img_dir", type=str, default="Dataset/Processed/color", help="Directory for training images")
parser.add_argument("--msk_dir", type=str, default="Dataset/Processed/label", help="Directory for training masks")
parser.add_argument("--pnt_dir", type=str, default="Dataset/ProcessedWithPrompt/color/points",
    help="Directory for training prompt points")
parser.add_argument("--mode", type=int, choices=[0, 1, 2, 3], required=True, help="0 for AutoEncoder, 1 for CLIP-based segmentation")
parser.add_argument("--pretrain", type=int, choices=[0, 1], default=0, help="1 to pretrain autoencoder")
parser.add_argument("--epochs", type=int, required=True, help="Number of training epochs")
parser.add_argument("--batch_size", type=int, default=4, help="Batch size for training")
parser.add_argument("--lr", type=float, default=1e-4, help="Learning rate")
parser.add_argument("--save_dir", type=str, default="params/", help="Path to save the model")
parser.add_argument("--patience", type=int, default=5, help="Patience for early stopping")
args = parser.parse_args()

# Define Device
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f'Training run on: "{device}"')
print(torch.cuda.get_device_name(0)) if torch.cuda.is_available() else print('no cuda devices')
# Ensure save directory exists
Path(args.save_dir).mkdir(parents=True, exist_ok=True)

# Load Dataset Paths
img_paths = sorted(Path(args.img_dir).glob("*.*"))
msk_paths = sorted(Path(args.msk_dir).glob("*.*"))
pnt_paths = sorted(Path(args.pnt_dir).glob("*.*"))

if args.mode == 3:
    train_img_paths, val_img_paths, \
    train_msk_paths, val_msk_paths, \
    train_pnt_paths, val_pnt_paths  = train_test_split(img_paths, msk_paths, pnt_paths, test_size=0.2, random_state=42)
else:
    train_img_paths, val_img_paths, \
    train_msk_paths, val_msk_paths  = train_test_split(img_paths, msk_paths, test_size=0.2, random_state=42)

# Select Model

###############################
############ U-Net ############
###############################
if args.mode == 0:
    print("Training Unet-Based Segmentation Model ...")
    # define the model
    model = UNet()

    # define training data split
    train_dataset = PetDataset(
                img_paths=train_img_paths,
                msk_paths=train_msk_paths,
                transform=standard_transform)
    val_dataset = PetDataset(
                    img_paths=val_img_paths,
                    msk_paths=val_msk_paths,
                    transform=standard_transform)

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False, num_workers=4)

    # define save paths
    train_plot = "unet_segmentation.png"
    training_plot_save_name = Path(args.save_dir) / train_plot
    save_name = "unet_segmentation.pth"

    # Define Loss Functions and Optimizer
    criterion= CombinedFocalDiceLoss(alpha=0.5, gamma=2.0)

    optimizer = optim.Adam(model.parameters(), lr=args.lr)

    # Train Model
    history = training(
        model=model,
        train_loader=train_loader,
        val_loader=val_loader,
        train_criterion=criterion,
        val_criterion=criterion,
        optimizer=optimizer,
        num_epochs=args.epochs,
        device=device,
        save_dir=args.save_dir,
        save_name=save_name,
```

```
        patience=args.patience
    )

    training_plot(history,save_path=training_plot_save_name)

###############################
######### Autoencoder #########
###############################
elif args.mode == 1:
    # define training data split
    train_dataset = PetDataset(
                img_paths=train_img_paths,
                msk_paths=train_msk_paths,
                transform=standard_transform)
    val_dataset = PetDataset(
                    img_paths=val_img_paths,
                    msk_paths=val_msk_paths,
                    transform=standard_transform)

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False, num_workers=4)
    # define config file name
    pretrain_model_save_name = "train_autoencoder_pretrain_unified_size.pth"
    model_save_name = "train_autoencoder_segmentation_unified_size.pth"
    pretrain_plot = "train_autoencoder_pretrain_unified_size.png"
    seg_train_plot = "train_autoencoder_segmentation_unified_size.png"
    if args.pretrain == 1: # the pretrain mode
        print('Autoencoder Pretrain ...')
        autoencoder = AutoEncoder()
        training_plot_save_name = Path(args.save_dir) / pretrain_plot
        history = pretrain_autoencoder(autoencoder, train_loader, val_loader, num_epochs=args.epochs,
                                       save_dir=args.save_dir, save_name=pretrain_model_save_name,
                                       device=device, patience=args.patience)
        training_plot(history,save_path=training_plot_save_name)
        print("Autoencoder pretrained and saved.")
    else: # the segmentation head train mode (encode freezed)
        print('Training Autoencoder-Based Segmentation Model ...')
        autoencoder = AutoEncoder()
        pretrain_config_path = Path(args.save_dir) / pretrain_model_save_name
        pretrain_state_dict = torch.load(pretrain_config_path, map_location=device)
        autoencoder.load_state_dict(pretrain_state_dict)
        training_plot_save_name = Path(args.save_dir) / seg_train_plot
        model = AutoEncoderSegmentation(autoencoder.encoder, num_classes=3)
        train_criterion = CombinedFocalDiceLoss()
        val_criterion = IouLoss()
        # filter(function,iterable) and lambda arguments: expression freeze the pretraining
        #  and train the decoder only
        optimizer = optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr)
        history = training(
            model, train_loader, val_loader,
            train_criterion=train_criterion, val_criterion=val_criterion,
            optimizer=optimizer,
            num_epochs=args.epochs, device=device,
            save_dir=args.save_dir,
            save_name=model_save_name,
            patience=args.patience
        )
        training_plot(history,save_path=training_plot_save_name)
        print("Autoencoder pretrained and saved.")

###############################
############ ClIP #############
###############################
elif args.mode == 2:
```

```python
    print("Training CLIP-Based Segmentation Model ...")
    clip_model, _ = clip.load("RN50", device=device)
    model = CLIPSegmentationModel(clip_model, num_classes=3)
    train_dataset = PetDataset(
                img_paths=train_img_paths,
                msk_paths=train_msk_paths,
                resize_fn=resize_with_padding,
                resize_target_size=224,
                transform=clip_transform)
    val_dataset = PetDataset(
                    img_paths=val_img_paths,
                    msk_paths=val_msk_paths,
                    resize_fn=resize_with_padding,
                    resize_target_size=224,
                    transform=clip_transform)

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4)
    val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False, num_workers=4)

    # define save paths
    train_plot = "train_clip_segmentation_unified_size.png"
    training_plot_save_name = Path(args.save_dir) / train_plot
    save_name = "train_clip_segmentation_unified_size.pth"

    # Define Loss Functions and Optimizer
    train_criterion = CombinedFocalDiceLoss()
    val_criterion = IouLoss()

    optimizer = optim.Adam(model.parameters(), lr=args.lr)

    # Train Model
    history = training(
        model=model,
        train_loader=train_loader,
        val_loader=val_loader,
        train_criterion=train_criterion,
        val_criterion=val_criterion,
        optimizer=optimizer,
        num_epochs=args.epochs,
        device=device,
        save_dir=args.save_dir,
        save_name=save_name,
        patience=args.patience
    )

    training_plot(history,save_path=training_plot_save_name)

elif args.mode == 3:
    print('Training Prompt-based Segmentation model ...')
    # define model
    model = PromptSegmentation(unet_in_channels=3,
                               prompt_dim=1024,
                               unet_init_features=64)
    train_dataset = PetDatasetWithPrompt(
                img_paths=train_img_paths,
                msk_paths=train_msk_paths,
                pnt_paths= train_pnt_paths,
                transform=standard_transform)
    val_dataset = PetDatasetWithPrompt(
                    img_paths=val_img_paths,
                    msk_paths=val_msk_paths,
                    pnt_paths = val_pnt_paths,
                    transform=standard_transform)

    train_loader = DataLoader(train_dataset, batch_size=args.batch_size, shuffle=True, num_workers=4)
```

```
    val_loader = DataLoader(val_dataset, batch_size=args.batch_size, shuffle=False, num_workers
        =4)

    # define save paths
    train_plot = "prompt_unet_segmentation.png"
    training_plot_save_name = Path(args.save_dir) / train_plot
    save_name = "prompt_unet_segmentation.pth"

    # Define Loss Functions and Optimizer
    criterion= BinaryFocalLoss()

    optimizer = optim.Adam(model.parameters(), lr=args.lr)

    # Train Model
    history = prompt_training(
        model=model,
        train_loader=train_loader,
        val_loader=val_loader,
        train_criterion=criterion,
        val_criterion=criterion,
        optimizer=optimizer,
        num_epochs=args.epochs,
        device=device,
        save_dir=args.save_dir,
        save_name=save_name,
        patience=args.patience
    )

    training_plot(history,save_path=training_plot_save_name)

else:
    raise ValueError("Invalid␣mode.␣Use␣0␣for␣U-Net,␣1␣for␣autoencoder-based␣segmentation,␣2␣for␣
        CLIP-based␣segmentation␣or␣3␣for␣prompt␣segmentation.")
```

C. Data Preprocessing

This appendix provides the implementation details of the data preprocessing pipeline used in the segmentation project. It includes dataset loaders, preprocessing utilities, and command-line preprocessing scripts for both standard and prompt-based segmentation tasks. These components handle colour-class conversions, input resizing, data augmentation, and train-validation splits, ensuring consistent and reproducible input preparation across all training modes.

C.1. Torch Dataset Loader *PetDataset.py*

This file defines the dataset loaders for both standard segmentation and prompt-based segmentation tasks.

PetDataset.py

```
import torch
import random
import numpy as np
from PIL import Image
from torch.utils.data import Dataset
from data.preprocessing import color2class


class PetDataset(Dataset):
    """
    A PyTorch Dataset for loading and preprocessing image-mask pairs for segmentation.

    Args:
        image_paths (list of str): List of file paths to the images.
        mask_paths (list of str): List of file paths to the corresponding masks.
        resize_fn (callable, optional): A function to resize the image & mask.
        resize_target_size (int): the target size of input images to the model. (assuming it’s a
            square image)
        augment_fn (callable, optional): A function to apply data augmentation to the image.
```

```python
        transform (callable, optional): Function to apply final normalization (e.g., CLIP
            transform).

    Returns:
        Tuple[torch.Tensor, torch.Tensor, Tuple[int, int], str]:
            - image: A tensor of shape (C, H, W) representing the normalized image.
            - mask: A tensor of shape (H, W), with values {0,1} or {0,2} (background vs cat or
                background vs dog).
            - initial_img_size: Tuple (H, W) representing the original image size.
            - img_path: str name of the color image
    """
    def __init__(self, img_paths, msk_paths, resize_fn=None, resize_target_size=None, transform=
        None):
        self.img_paths = img_paths
        self.msk_paths = msk_paths
        self.resize_fn = resize_fn  # Function to resize images and masks differently
        self.resize_target_size = resize_target_size
        self.transform = transform
        # make sure # of images is equal to # of masks
        assert len(self.img_paths) == len(self.msk_paths), "Mismatch between images and masks."

    def __len__(self):
        return len(self.img_paths)

    def __getitem__(self, idx):
        # Load and convert image to RGB
        img_path = str(self.img_paths[idx])
        img = Image.open(img_path).convert("RGB")
        img = np.array(img)
        initial_img_size = tuple(img.shape[:2])

        # Load and process mask
        msk_path = str(self.msk_paths[idx])
        msk = Image.open(msk_path).convert("RGB")
        msk = np.array(msk)  # Convert to NumPy array

        # Convert color mask to class labels
        msk = color2class(msk)  # Convert mask from RGB to class labels

        # Resize if a resizing function is provided
        if self.resize_fn:
            img = self.resize_fn(img, target_size=self.resize_target_size, is_mask=False)
            msk = self.resize_fn(msk, target_size=self.resize_target_size, is_mask=True)

        # Apply transform only to the image (not the mask)
        if self.transform:
            img = self.transform(Image.fromarray(img)) # only apply the normalization to img

        # Convert mask to tensor
        msk = torch.tensor(msk, dtype=torch.long)

        return img, msk, initial_img_size, img_path

class PetDatasetWithPrompt(Dataset):
    def __init__(self, img_paths, msk_paths, pnt_paths, resize_fn=None,
                 resize_target_size=None, transform=None, load_multiple_points=False,
                     use_box_prompt=True):
        self.img_paths = img_paths
        self.msk_paths = msk_paths
        self.pnt_paths = pnt_paths
        self.resize_fn = resize_fn
        self.resize_target_size = resize_target_size
        self.transform = transform
        self.load_multiple_points = load_multiple_points
        self.use_box_prompt = use_box_prompt

        assert len(self.img_paths) == len(self.msk_paths) == len(self.pnt_paths), \
            "Mismatch between images, masks, and point files."
```

```python

    def __len__(self):
        return len(self.img_paths)

    def load_points_from_file(self, point_path):
        with open(point_path, 'r') as f:
            line = f.readline().strip()

        if not line:
            raise ValueError(f"Empty prompt file: {point_path}")

        try:
            parts = list(map(int, line.strip().split(',')))
        except Exception as e:
            raise ValueError(f"Failed to parse prompt file: {point_path} with line: '{line}' ({e
                })")

        if len(parts) == 5:
            # Box format: x1, y1, x2, y2, cls
            x1, y1, x2, y2, cls = parts
            return [], True, (x1, y1, x2, y2), cls
        elif len(parts) == 3:
            # Point format: x, y, cls
            x, y, cls = parts
            return [(x, y, cls)], False, None, -1
        else:
            raise ValueError(f"Invalid prompt format in: {point_path} (got {len(parts)} values)")



    def generate_gaussian_heatmap(self, H, W, x, y, sigma=10):
        xs = np.arange(W)
        ys = np.arange(H)
        xs, ys = np.meshgrid(xs, ys)
        g = np.exp(-((xs - x) ** 2 + (ys - y) ** 2) / (2 * sigma ** 2))
        return g.astype(np.float32)

    def __getitem__(self, idx):
        img_path = str(self.img_paths[idx])
        img = Image.open(img_path).convert("RGB")
        img = np.array(img)
        img_h, img_w = img.shape[:2]
        initial_img_size = (img_h, img_w)

        msk_path = str(self.msk_paths[idx])
        msk = Image.open(msk_path).convert("RGB")
        msk = np.array(msk)
        msk = color2class(msk)


        if self.resize_fn:
            img = self.resize_fn(img, target_size=self.resize_target_size, is_mask=False)
            msk = self.resize_fn(msk, target_size=self.resize_target_size, is_mask=True)

        point_path = str(self.pnt_paths[idx])
        points, is_box, box_coords, box_class = self.load_points_from_file(point_path)


        H, W = msk.shape[:2]

        if is_box and self.use_box_prompt:
            x1, y1, x2, y2 = box_coords
            heatmap = np.zeros((H, W), dtype=np.float32)
            heatmap[y1:y2+1, x1:x2+1] = 1.0
            point_class = box_class
            # Use center of box as representative point
            x = (x1 + x2) // 2
            y = (y1 + y2) // 2
```

```
        else:
            if points ==[]:
                print(is_box, box_coords, box_class)
                raise ValueError(f"Unknown prompt format in: {point_path} with points []")

            x, y, cls = points[0]
            heatmap = self.generate_gaussian_heatmap(H, W, x, y, sigma=10)
            point_class = cls

        if self.transform:
            img = self.transform(Image.fromarray(img))

        output = {
            'image': img,
            'gt_mask': torch.tensor(msk, dtype=torch.long).unsqueeze(0),
            'prompt_heatmap': torch.tensor(heatmap, dtype=torch.float32).unsqueeze(0),
            'prompt_point': torch.tensor([x, y], dtype=torch.long),
            'initial_img_size': initial_img_size,
            'img_path': img_path
        }

        if point_class != -1:
            output['point_class'] = torch.tensor(point_class, dtype=torch.long)
            if point_class not in [0,1,2]:
                print(f"invalid point_class in: {point_path}")
                if point_class==255:
                    output['point_class'] = torch.tensor(0, dtype=torch.long)
                    print("change to 0")

        return output
```

C.2. Preprocessing Utilities *preprocessing.py*

This file contains core utility functions for data preprocessing, including:

1. Mapping functions between color-encoded masks and class arrays.
2. A padding-based image resizing function for uniform input dimensions.
3. Image normalization transforms.
4. Data augmentation using the `albumentations` library.
5. A function to generate point-wise masks for the prompt-based model.

preprocessing.py

```
# All preprocessing utilities (image/mask transformations & augmentations)

import numpy as np
import cv2 as cv
import torchvision.transforms as transforms
from pathlib import Path
from PIL import Image
import torch
from tqdm import tqdm
import albumentations as A

# ----------------------------
# Color Mapping Utilities
# ----------------------------

def color2class(img: np.ndarray) -> np.ndarray:
    """
    Convert a label RGB image to pixel-wise class labels.

    Parameters:
```

```
        img (np.ndarray): Input image as a NumPy array (expected shape: (H, W, 3)).

    Returns:
        np.ndarray: Output array pixel-wise class labels (expected shape: (H, W)).
    """

    color_to_class = {
        (0, 0, 0): 0,          # Black -> Class 0
        (255, 255, 255): 0,    # White -> Class 0
        (128, 0, 0): 1,        # Dark Red -> Class 1
        (0, 128, 0): 2         # Green -> Class 2
    }
    h,w,_ = img.shape
    class_map = np.zeros((h,w), dtype=np.uint8)
    img_reshaped = img.reshape(-1,3)
    class_map_reshaped = np.zeros(img_reshaped.shape[0], dtype=np.uint8)

    # Assign class labels
    for color, class_id in color_to_class.items():
        mask = np.all(img_reshaped == color, axis=1)
        class_map_reshaped[mask] = class_id

    # Reshape back to original dimensions
    class_map = class_map_reshaped.reshape(h, w)

    return class_map

def class2color(class_map: np.ndarray) -> np.ndarray:
    """
    Convert a pixel-wise class label map to an RGB image representation.

    Parameters:
        class_map (np.ndarray): 2D numpy array of shape (H, W) containing class labels.

    Returns:
        np.ndarray: Output RGB image as a NumPy array of shape (H, W, 3).
    """

    # Define the mapping from class labels to RGB colors
    class_to_color = {
        0: (0, 0, 0),          # Class 0 -> Black (background)
        1: (128, 0, 0),        # Class 1 -> Dark Red (cat)
        2: (0, 128, 0)         # Class 2 -> Green (dog)
    }

    # Get image dimensions
    h, w = class_map.shape

    # Initialize an empty RGB image
    color_image = np.zeros((h, w, 3), dtype=np.uint8)

    # Assign colors based on class labels
    for class_id, color in class_to_color.items():
        mask = (class_map == class_id)
        color_image[mask] = color  # Assign the corresponding RGB color

    return color_image

# ----------------------------
# Resizing Utilities
# ----------------------------

def resize_with_padding(img: np.ndarray, target_size, fill=0, is_mask=False) -> np.ndarray:
    """
    Resize an image while maintaining its aspect ratio and pad it to a square.

    Args:
        img (np.ndarray): Input image (H, W, C) or (H, W) if grayscale/mask.
```

```
        target_size (int, optional): The target width and height (default: 224).
        fill (int or tuple, optional): Padding color, either an int (grayscale) or
            (R, G, B) tuple for color images. Default is black (0).
        is_mask (bool, optional): If True, uses NEAREST interpolation for masks
            to avoid artifacts. Default is False (for normal images).

    Returns:
        np.ndarray: The resized and padded image/mask with dimensions (target_size, target_size).
    """

    # Get current dimensions
    h, w = img.shape[:2]

    # Compute scale factor to fit the longest side
    scale = target_size / max(w, h)
    new_w, new_h = int(w * scale), int(h * scale)  # New dimensions

    # Select interpolation method (NEAREST for masks, BICUBIC for images)
    interpolation = cv.INTER_NEAREST if is_mask else cv.INTER_CUBIC

    # Resize image
    img_resized = cv.resize(img, (new_w, new_h), interpolation=interpolation)

    # Create a blank canvas with padding
    if len(img.shape) == 3:  # RGB image
        padded_img = np.full((target_size, target_size, 3), fill, dtype=img.dtype)
    else:  # Grayscale/mask
        padded_img = np.full((target_size, target_size), fill, dtype=img.dtype)

    # Compute padding offsets to center the image
    paste_x = (target_size - new_w) // 2
    paste_y = (target_size - new_h) // 2

    # Place the resized image onto the padded canvas
    padded_img[paste_y:paste_y + new_h, paste_x:paste_x + new_w] = img_resized

    return padded_img

clip_transform = transforms.Compose([
                    transforms.ToTensor(),  # Converts image to [0, 1] range
                    transforms.Normalize(mean=[0.48145466, 0.4578275, 0.40821073],
                    std=[0.26862954, 0.26130258, 0.27577711])  # Converts to ~[-1, 1]
                    ])

standard_transform = transforms.Compose([
    transforms.ToTensor()
])

# ---------------------------
# Data Augmentation Utilities
# --------------------------

def augmentor(image: np.ndarray, mask: np.ndarray) -> dict:
    """
    Apply Albumentations-based augmentations to both image and mask.

    Args:
        image (np.ndarray): The input image (H, W, C).
        mask (np.ndarray): The segmentation mask (H, W).

    Returns:
        dict: Dictionary containing:
            - 'image' (np.ndarray): The augmented image (H, W, C).
            - 'mask' (np.ndarray): The augmented mask (H, W).
    """
    transform = A.Compose([
        A.HorizontalFlip(p=0.5),  # Flip 50% of the time
        A.RandomBrightnessContrast(p=0.2),  # Adjust brightness & contrast
```

```
        A.Affine(
            scale=(0.9, 1.1),
            translate_percent=(-0.0625, 0.0625),
            rotate=(-15, 15),
            interpolation=0,  # cv2.INTER_NEAREST (Ensures mask values stay discrete)
            p=0.5
        ),
        A.GaussianBlur(blur_limit=(3, 5), p=0.2),  # Blur occasionally
        A.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1, p=0.3)  # Color jitter
    ])

    # Apply transformations
    augmented = transform(image=image, mask=mask)

    return augmented

###########################
# Create Prompt-wise Mask #
###########################
def create_point_wise_mask(point_classes, gt_masks):
    """
    Create a point-wise mask based on the prompt point and its class.

    Args:
        prompt_point (torch.Tensor): Tensor of shape (B, 2) with x,y coordinates
        point_class (torch.Tensor): Tensor of shape (B,) with class indices
        gt_mask (torch.Tensor): Ground truth mask of shape (B, 1, H, W)

    Returns:
        torch.Tensor: Binary mask showing regions that should belong to the same class
                     as the prompt point
    """
    B, _, H, W = gt_masks.shape
    point_wise_masks = []

    for b in range(B):
        cls = point_classes[b]

        # Create binary mask where 1s represent pixels of the same class as the prompt point
        point_wise_mask = (gt_masks[b, 0] == cls).float()

        point_wise_masks.append(point_wise_mask)

    return torch.stack(point_wise_masks).unsqueeze(1).long()  # (B, 1, H, W)
```

C.3. Preprocessing Runners *preprocess_data.py* and *preprocess_data_with_prompt.py*

These are command-line preprocessing scripts that allow configurable options such as input/output directories, augmentation iterations, image resizing, and prompt point sampling for the prompt-based model.

preprocess_data.py

```
# src/preprocess_dataset.py

from preprocessing import (
    resize_with_padding,
    color2class,
    augmentor
)
from pathlib import Path
from PIL import Image
import numpy as np
from tqdm import tqdm
import torch
import argparse

def prepare_final_dataset(raw_img_dir, raw_msk_dir,
```

```python
                           processed_img_dir, processed_msk_dir,
                           base_augmentations, extra_augmentations,
                           target_size):
    """
    Runs augmentation and resizing, creating the final training dataset.
    """

    # Step 0: Initialize the paths of raw images and raw masks into lists
    raw_img_paths = sorted(Path(raw_img_dir).glob("*.*"))
    print(len(raw_img_paths))
    raw_msk_paths = sorted(Path(raw_msk_dir).glob("*.*"))

    # create the save directories for processing images and masks
    processed_img_dir.mkdir(parents=True, exist_ok=True)
    processed_msk_dir.mkdir(parents=True, exist_ok=True)

    print("Starting dataset preparation...")
    for img_path, msk_path in tqdm(zip(raw_img_paths, raw_msk_paths),
                                    total=len(raw_msk_paths),
                                    desc="Final resizing & augmentation",
                                    unit= 'image'):
        # Extract the names of raw image and mask
        img_name = img_path.stem
        msk_name = msk_path.stem
        # Load image and mask
        img = np.array(Image.open(img_path).convert("RGB"))
        msk = np.array(Image.open(msk_path).convert("RGB"))

        # Step 1: Resize with padding
        img_resized = resize_with_padding(img, target_size=target_size, is_mask=False)
        msk_resized = resize_with_padding(msk, target_size=target_size, is_mask=True)

        msk_class = color2class(msk_resized)
        unique_classes = torch.unique(torch.tensor(msk_class)).tolist()
        num_augmentations = extra_augmentations if 1 in unique_classes else base_augmentations
        for i in range(num_augmentations):
            augmented = augmentor(img_resized, msk_resized)
            aug_image, aug_mask = augmented['image'], augmented['mask']

            Image.fromarray(aug_image.astype(np.uint8)).save(
                Path(processed_img_dir) / f"aug_{img_name}_{i+1}.jpg")

            Image.fromarray(aug_mask.astype(np.uint8)).save(
                Path(processed_msk_dir) / f"aug_{msk_name}_{i+1}.png")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Prepare the final dataset for segmentation training.")
    parser.add_argument('--img_dir', default="./Dataset/Train/color",
                        help="Directory containing raw images")
    parser.add_argument('--msk_dir', default="./Dataset/Train/label",
                        help="Directory containing raw masks")
    parser.add_argument('--processed_img_dir', default="./Dataset/TrainProcessed/color",
                        help="Directory to save processed images and points")
    parser.add_argument('--processed_msk_dir', default="./Dataset/TrainProcessed/label",
                        help="Directory to save processed masks")
    parser.add_argument('--base_augmentations', default=4,
                        help="Number of base augmentations")
    parser.add_argument('--extra_augmentations', default=8,
                        help="Number of extra augmentations")
    parser.add_argument('--target_size', default=512,
                        help="Target size for resizing images and masks")
    args = parser.parse_args()
    prepare_final_dataset(
        raw_img_dir=Path(args.img_dir),
        raw_msk_dir=Path(args.msk_dir),
        processed_img_dir=Path(args.processed_img_dir),
        processed_msk_dir=Path(args.processed_msk_dir),
```

```
        base_augmentations=args.base_augmentations,
        extra_augmentations=args.extra_augmentations,
        target_size=args.target_size
    )
```

preprocess_data_with_prompt.py

```
#!/usr/bin/env python

import numpy as np
from pathlib import Path
from PIL import Image
import torch
import argparse
from tqdm.contrib.concurrent import process_map  # tqdm multiprocess support

from data.preprocessing import (
    resize_with_padding,
    color2class,
    augmentor
)

def sample_prompt_type_and_count(max_points=1, box_prob=0.5):
    """
    Returns (prompt_type, count)
    - prompt_type: "points" or "box"
    - count: number of points (1 for "points", 2 for "box")
    """
    if np.random.rand() < box_prob:
        return "box", 2
    return "points", 1




def sample_point_on_mask(mask, num_points, bg_ratio=0.5):
    mask = color2class(mask)
    unique_classes = np.unique(mask)

    # ensure the background class (0) is included
    class_weights = []
    class_pixel_lists = []

    for cls in unique_classes:
        class_pixels = np.argwhere(mask == cls)
        if len(class_pixels) == 0:
            continue
        class_pixel_lists.append((cls, class_pixels))

        weight = 1.0 if cls != 0 else bg_ratio
        class_weights.append(weight * len(class_pixels))

    if len(class_pixel_lists) == 0:
        return []

    # normalize class weights
    class_weights = np.array(class_weights, dtype=np.float32)
    class_weights /= class_weights.sum()

    sampled_points = []
    for _ in range(num_points):
        selected_class_idx = np.random.choice(len(class_pixel_lists), p=class_weights)
        selected_class, class_pixels = class_pixel_lists[selected_class_idx]
        point_idx = np.random.randint(0, len(class_pixels))
        y, x = class_pixels[point_idx]
        sampled_points.append((x, y, selected_class))
    return sampled_points

```

```python


def process_one_image(args_tuple):
    (img_path, msk_path, processed_img_dir, processed_msk_dir, processed_point_dir,
     base_augmentations, extra_augmentations, target_size, points_per_image, mode, box_prob) =
         args_tuple

    img_name = img_path.stem
    msk_name = msk_path.stem

    img = np.array(Image.open(img_path).convert("RGB"))
    msk = np.array(Image.open(msk_path).convert("RGB"))

    img_resized = resize_with_padding(img, target_size=target_size, is_mask=False)
    msk_resized = resize_with_padding(msk, target_size=target_size, is_mask=True)

    if mode in [1, 2, 3]:  # test modes: no augmentation
        Image.fromarray(img_resized.astype(np.uint8)).save(processed_img_dir / f"{img_name}.jpg")
        Image.fromarray(msk_resized.astype(np.uint8)).save(processed_msk_dir / f"{msk_name}.png")

        prompt_type, num = sample_prompt_type_and_count(points_per_image, box_prob=box_prob)
        # Here we sample points and each point is a tuple (x, y, cls)
        sampled_points = sample_point_on_mask(msk_resized, num_points=num, bg_ratio=0.2)

        points_file_path = processed_point_dir / f"{img_name}_points.txt"
        with open(points_file_path, 'w') as f:
            if prompt_type == "box":
                # Compute box using first two sampled points
                x_coords = [pt[0] for pt in sampled_points[:2]]
                y_coords = [pt[1] for pt in sampled_points[:2]]
                x1, x2 = sorted(x_coords)
                y1, y2 = sorted(y_coords)
                # Convert box region to class labels before processing
                box_mask_class = color2class(msk_resized[y1:y2+1, x1:x2+1])
                if box_mask_class.size == 0:
                    box_cls = -1
                else:
                    cls_counts = np.bincount(box_mask_class.flatten())
                    box_cls = np.argmax(cls_counts) if len(cls_counts) > 0 else -1

                with open(points_file_path, 'w') as f:
                    f.write(f"{x1},{y1},{x2},{y2},{box_cls}\n")
            else:
                # For single point prompt
                x, y, cls = sampled_points[0]
                f.write(f"{x},{y},{cls}\n")
        return

    # mode == 0: train with augmentation
    msk_class = color2class(msk_resized)
    unique_classes = torch.unique(torch.tensor(msk_class)).tolist()
    num_augmentations = extra_augmentations if 1 in unique_classes else base_augmentations

    for i in range(num_augmentations):
        augmented = augmentor(img_resized, msk_resized)
        aug_image, aug_mask = augmented['image'], augmented['mask']

        prompt_type, num = sample_prompt_type_and_count(points_per_image, box_prob=box_prob)
        sampled_points = sample_point_on_mask(aug_mask, num_points=num)

        aug_img_path = processed_img_dir / f"aug_{img_name}_{i+1}.jpg"
        aug_msk_path = processed_msk_dir / f"aug_{msk_name}_{i+1}.png"
        points_file_path = processed_point_dir / f"aug_{img_name}_{i+1}_points.txt"

        Image.fromarray(aug_image.astype(np.uint8)).save(aug_img_path)
        Image.fromarray(aug_mask.astype(np.uint8)).save(aug_msk_path)

```

```python
        with open(points_file_path, 'w') as f:
            if prompt_type == "box":
                # Compute box using first two sampled points
                x_coords = [pt[0] for pt in sampled_points[:2]]
                y_coords = [pt[1] for pt in sampled_points[:2]]
                x1, x2 = sorted(x_coords)
                y1, y2 = sorted(y_coords)
                # Extract the mask region to compute the mode class
                box_mask_class = color2class(aug_mask[y1:y2+1, x1:x2+1])
                if box_mask_class.size == 0:
                    box_cls = -1
                else:
                    box_cls_counts = np.bincount(box_mask_class.flatten())
                    if len(box_cls_counts) == 0:
                        box_cls = -1
                    else:
                        box_cls = np.argmax(box_cls_counts)
                f.write(f"{x1},{y1},{x2},{y2},{box_cls}\n")
            else:
                # For single point prompt
                x, y, cls = sampled_points[0]
                f.write(f"{x},{y},{cls}\n")




def prepare_final_dataset(raw_img_dir, raw_msk_dir,
                          processed_img_dir, processed_msk_dir,
                          base_augmentations, extra_augmentations,
                          target_size, points_per_image=5, mode=0, num_workers=4,box_prob=0.2):
    raw_img_paths = sorted(Path(raw_img_dir).glob("*.*"))
    raw_msk_paths = sorted(Path(raw_msk_dir).glob("*.*"))

    processed_img_dir = Path(processed_img_dir)
    processed_msk_dir = Path(processed_msk_dir)
    processed_point_dir = processed_img_dir / "points"
    processed_img_dir.mkdir(parents=True, exist_ok=True)
    processed_msk_dir.mkdir(parents=True, exist_ok=True)
    processed_point_dir.mkdir(parents=True, exist_ok=True)

    print(f"Preparing {len(raw_img_paths)} image-mask pairs using {num_workers} workers...")

    args_list = [
        (img_path, msk_path, processed_img_dir, processed_msk_dir, processed_point_dir,
         base_augmentations, extra_augmentations, target_size, points_per_image, mode, box_prob)
        for img_path, msk_path in zip(raw_img_paths, raw_msk_paths)
    ]


    process_map(process_one_image, args_list, max_workers=num_workers, chunksize=1, desc="Processing")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--mode', type=int, default=0, choices=[0, 1, 2, 3],
        help="0 = full preprocessing with augmentation, 1 = prompt point only, 2 = test set (no augmentation, with class), 3 = test set (no augmentation, no class)")
    parser.add_argument('--img_dir', default="./Dataset/TrainVal/color",
        help="Directory containing raw images")
    parser.add_argument('--msk_dir', default="./Dataset/TrainVal/label",
        help="Directory containing raw masks")
    parser.add_argument('--processed_img_dir', default="./Dataset/ProcessedWithPrompt_test/color",
        help="Directory to save processed images and points")
    parser.add_argument('--processed_msk_dir', default="./Dataset/ProcessedWithPrompt_test/label",
        help="Directory to save processed masks")
    parser.add_argument('--points_per_image', type=int, default=1,
```

```
        help="Number␣of␣prompt␣points␣to␣generate")
    parser.add_argument('--num_workers', type=int, default=8,
        help="Number␣of␣parallel␣workers␣for␣processing")
    parser.add_argument('--box_prob', type=float, default=0.5,
        help="Probability␣of␣sampling␣a␣box␣instead␣of␣points")

    args = parser.parse_args()

    prepare_final_dataset(
        raw_img_dir=Path(args.img_dir),
        raw_msk_dir=Path(args.msk_dir),
        processed_img_dir=Path(args.processed_img_dir),
        processed_msk_dir=Path(args.processed_msk_dir),
        base_augmentations=4,
        extra_augmentations=8,
        target_size=512,
        points_per_image=args.points_per_image,
        mode=args.mode,
        num_workers=args.num_workers,
        box_prob=args.box_prob
    )
```

C.4. Train/Validation Split Utility *train_val_split.py*

This script partitions the dataset into training and validation subsets. The test set remains completely unseen to ensure fair evaluation, and the best model is selected based on performance on the validation set.

train_val_split.py

```
from pathlib import Path

source_color_dir = Path("Dataset/TrainVal/color/")
source_lable_dir = Path("Dataset/TrainVal/label/")

import random
from pathlib import Path
import shutil

# Set seed for reproducibility
random.seed(42)

# Source directories
source_color_dir = Path("Dataset/TrainVal/color/")
source_label_dir = Path("Dataset/TrainVal/label/")

# Destination directories
train_color_dir = Path("Dataset/Train/color/")
train_label_dir = Path("Dataset/Train/label/")
val_color_dir = Path("Dataset/Val/color/")
val_label_dir = Path("Dataset/Val/label/")

# Create destination folders
for path in [train_color_dir, train_label_dir, val_color_dir, val_label_dir]:
    path.mkdir(parents=True, exist_ok=True)

# Get list of color image files (assuming same names for labels)
color_files = sorted(source_color_dir.glob("*"))
total = len(color_files)
split_idx = int(total * 0.9)

# Shuffle and split
random.shuffle(color_files)
train_files = color_files[:split_idx]
val_files = color_files[split_idx:]

# Helper to copy files
def copy_files(file_list, color_dest, label_dest):
    for color_file in file_list:
```

```
        label_filename = color_file.stem + ".png"
        label_file = source_label_dir / label_filename
        shutil.copy2(color_file, color_dest / color_file.name)
        shutil.copy2(label_file, label_dest / label_file.name)

# Copy files to train and val directories
copy_files(train_files, train_color_dir, train_label_dir)
copy_files(val_files, val_color_dir, val_label_dir)

print(f"Total images: {total}")
print(f"Training images: {len(train_files)}")
print(f"Validation images: {len(val_files)}")
```

D. Model Structures

This appendix provides the implementation details of the four segmentation models compared in this report. These model definitions serve as supporting material for Section 3, where the design rationale and high-level architecture are discussed.

D.1. U-Net Model

unet_segmentation.py

```
import torch
import torch.nn as nn

class UNet(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, init_features=64):
        super(UNet, self).__init__()

        features = init_features
        # Encoder blocks
        self.encoder1 = self._conv_block(in_channels, features)
        self.encoder2 = self._conv_block(features, features * 2)
        self.encoder3 = self._conv_block(features * 2, features * 4)
        self.encoder4 = self._conv_block(features * 4, features * 8)

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

        # Decoder upsampling layers (now ConvTranspose2d)
        self.up4 = nn.ConvTranspose2d(features * 16, features * 8, kernel_size=2, stride=2)
        self.decoder4 = self._conv_block(features * 16, features * 8)

        self.up3 = nn.ConvTranspose2d(features * 8, features * 4, kernel_size=2, stride=2)
        self.decoder3 = self._conv_block(features * 8, features * 4)

        self.up2 = nn.ConvTranspose2d(features * 4, features * 2, kernel_size=2, stride=2)
        self.decoder2 = self._conv_block(features * 4, features * 2)

        self.up1 = nn.ConvTranspose2d(features * 2, features, kernel_size=2, stride=2)
        self.decoder1 = self._conv_block(features * 2, features)

        # Final output layer
        self.final_layer = nn.Conv2d(features, out_channels, kernel_size=1)

        # Max pooling
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def _conv_block(self, in_channels, out_channels):
        return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
```

```python
        )

    def forward(self, x):
        # Encoder
        enc1 = self.encoder1(x)  # (B, features, 512, 512)
        enc2 = self.encoder2(self.pool(enc1))  # (B, features*2, 256, 256)
        enc3 = self.encoder3(self.pool(enc2))  # (B, features*4, 128, 128)
        enc4 = self.encoder4(self.pool(enc3))  # (B, features*8, 64, 64)

        # Bottleneck
        bottleneck = self.bottleneck(self.pool(enc4))  # (B, features*16, 32, 32)

        # Decoder with skip connections
        dec4 = self.up4(bottleneck)  # (B, features*8, 64, 64)
        dec4 = torch.cat((dec4, enc4), dim=1)  # (B, features*16, 64, 64)
        dec4 = self.decoder4(dec4)

        dec3 = self.up3(dec4)  # (B, features*4, 128, 128)
        dec3 = torch.cat((dec3, enc3), dim=1)  # (B, features*8, 128, 128)
        dec3 = self.decoder3(dec3)

        dec2 = self.up2(dec3)  # (B, features*2, 256, 256)
        dec2 = torch.cat((dec2, enc2), dim=1)  # (B, features*4, 256, 256)
        dec2 = self.decoder2(dec2)

        dec1 = self.up1(dec2)  # (B, features, 512, 512)
        dec1 = torch.cat((dec1, enc1), dim=1)  # (B, features*2, 512, 512)
        dec1 = self.decoder1(dec1)

        # Final output layer
        return self.final_layer(dec1)
```

D.2. Autoencoder-based Model

autoencoder_segmentation.py

```python
import torch
import torch.nn as nn
import torch.optim as optim
from pathlib import Path
from tqdm import tqdm

################################
## Autoencoder Architecture ##
################################

class AutoEncoder(nn.Module):
    """
    Autoencoder architecture for image reconstruction.
    The encoder compresses the input image into a latent representation,
    and the decoder reconstructs the image from this representation.
    Args:
        init_features (int): Number of initial features for the encoder.
    """
    def __init__(self, init_features=128):
        super(AutoEncoder, self).__init__()

        f = init_features

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(3, f, kernel_size=3, stride=2, padding=1), # 3*512*512 > 64*256*256
            nn.BatchNorm2d(f),
            nn.ReLU(),
            nn.Conv2d(f, f * 2, kernel_size=3, stride=2, padding=1), # 64*256*256 > 128*128*128
            nn.BatchNorm2d(f * 2),
            nn.ReLU(),
```

```python
            nn.Conv2d(f * 2, f * 4, kernel_size=3, stride=2, padding=1), # 128*128*128 > 256*64*64
            nn.BatchNorm2d(f * 4),
            nn.ReLU(),
            nn.Conv2d(f * 4, f * 8, kernel_size=3, stride=2, padding=1), # 256*64*64 > 512*32*32
            nn.BatchNorm2d(f * 8),
            nn.ReLU(),
            nn.Conv2d(f * 8, f * 16, kernel_size=3, stride=2, padding=1), #512*32*32 > 1024*16*16
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(f * 16, f * 8, kernel_size=3, stride=2, padding=1, output_padding=1),  # 1024*16*16 > 512*32*32
            nn.BatchNorm2d(f * 8),
            nn.ReLU(),
            nn.ConvTranspose2d(f * 8, f * 4, kernel_size=3, stride=2, padding=1, output_padding=1),  # 512*32*32 > 256*64*64
            nn.BatchNorm2d(f * 4),
            nn.ReLU(),
            nn.ConvTranspose2d(f * 4, f * 2, kernel_size=3, stride=2, padding=1, output_padding=1),  # 256*64*64 > 128*128*128
            nn.BatchNorm2d(f * 2),
            nn.ReLU(),
            nn.ConvTranspose2d(f * 2, f, kernel_size=3, stride=2, padding=1, output_padding=1),      # 128*128*128 > 64*256*256
            nn.BatchNorm2d(f),
            nn.ReLU(),
            nn.ConvTranspose2d(f, 3, kernel_size=3, stride=2, padding=1, output_padding=1),              # 64*256*256 > 3*512*512
            nn.Sigmoid()
        )

    def forward(self, x):
        latent = self.encoder(x)
        reconstructed = self.decoder(latent)
        return reconstructed

##############################
## Segmentation Decoder Head ##
##############################

class SegmentationDecoder(nn.Module):
    '''
    Segmentation Decoder for the Autoencoder.
    Args:
        input_channel (int): Number of input channels from the encoder.
        output_channel (int): Number of output channels for segmentation.
    '''
    def __init__(self, input_channel, output_channel):
        super().__init__()
        self.seg_decoder = nn.Sequential(
            nn.ConvTranspose2d(input_channel, 512, 3, stride=2, padding=1, output_padding=1),  # 1024*16*16 > 512*32*32
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.ConvTranspose2d(512, 256, 3, stride=2, padding=1, output_padding=1),   # 512*32*32 > 256*64*64
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.ConvTranspose2d(256, 128, 3, stride=2, padding=1, output_padding=1),   # 256*64*64 > 128*128*128
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.ConvTranspose2d(128, 64, 3, stride=2, padding=1, output_padding=1),    # 128*128*128 > 64*256*256
            nn.BatchNorm2d(64),
            nn.ReLU(),
```

```python
            nn.ConvTranspose2d(64, output_channel, 3, stride=2, padding=1, output_padding=1),  # 64*256*256 > 3*512*512
        )

    def forward(self, features):
        return self.seg_decoder(features)

###########################################
## Final Autoencoder-based Segmentation ##
###########################################

class AutoEncoderSegmentation(nn.Module):
    '''
    Autoencoder-based Segmentation Model.
    Args:
        encoder (nn.Module): Encoder part of the autoencoder.
        num_classes (int): Number of classes for segmentation.
    Returns:
        out (torch.Tensor): Segmentation output.
    '''
    def __init__(self, encoder, num_classes=3):
        super().__init__()
        # Frozen encoder from pre-trained autoencoder
        self.encoder = encoder
        for param in self.encoder.parameters():
            param.requires_grad = False
        # Segmentation Decoder
        self.seg_decoder = SegmentationDecoder(input_channel=self.encoder[-1].out_channels, output_channel=num_classes)

    def forward(self, x):
        with torch.no_grad():
            features = self.encoder(x)  # Extract latent features
        out = self.seg_decoder(features)
        return out

########################################
## Autoencoder Pretraining Function  ##
########################################

def pretrain_autoencoder(autoencoder, train_loader, val_loader,
                         save_dir, save_name,
                         num_epochs, device, patience):
    '''
    Pretrain the autoencoder on the training dataset.
    Args:
        autoencoder (nn.Module): Autoencoder model.
        train_loader (DataLoader): DataLoader for training data.
        val_loader (DataLoader): DataLoader for validation data.
        save_dir (str): Directory to save the pretrained model.
        save_name (str): Name of the saved model file.
        num_epochs (int): Number of epochs for pretraining.
        device (str): Device to use ("cuda" or "cpu").
        patience (int): Number of epochs with no improvement after which training will be stopped.
    Returns:
        history (dict): Dictionary containing training and validation losses.
    '''
    autoencoder.to(device)
    criterion = nn.MSELoss()
    optimizer = optim.Adam(autoencoder.parameters(), lr=1e-3)
    best_val_loss = float('inf')
    best_model_state = None
    epochs_no_improve = 0
    save_path = Path(save_dir) / save_name

    # Initialize history dict to store losses
    history = {"train_loss": [], "val_loss": []}
```

```


    for epoch in range(num_epochs):
        # Training Phase
        autoencoder.train()
        running_loss = 0.0
        for images, _, _, _ in tqdm(train_loader, desc=f"Pretrain Epoch {epoch+1}/{num_epochs}"):
            images = images.to(device)
            optimizer.zero_grad()
            reconstructed = autoencoder(images)
            loss = criterion(reconstructed, images)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        avg_train_loss = running_loss / len(train_loader)

        # Validation Phase
        autoencoder.eval()
        val_running_loss = 0.0
        with torch.no_grad():
            for images, _, _, _ in tqdm(val_loader, desc=f"Pretrain Epoch {epoch+1}/{num_epochs} [Val]",unit='image'):
                images = images.to(device)
                reconstructed = autoencoder(images)
                val_loss = criterion(reconstructed, images)
                val_running_loss += val_loss.item()

        avg_val_loss = val_running_loss / len(val_loader)

        # Store the losses for plotting
        history["train_loss"].append(avg_train_loss)
        history["val_loss"].append(avg_val_loss)
        # Print Losses
        print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {avg_train_loss:.6f} | Val Loss: {avg_val_loss:.6f}")

        # Early Stopping Check (on validation loss)
        if avg_val_loss < best_val_loss:
            best_val_loss = avg_val_loss
            best_model_state = autoencoder.state_dict()
            epochs_no_improve = 0
        else:
            epochs_no_improve += 1
            print(f"No improvement for {epochs_no_improve} epoch(s)")

        if epochs_no_improve >= patience:
            print(f"Early stopping triggered at epoch {epoch+1}")
            print(f"Minimum Val MSE Loss: {best_val_loss:.6f}")
            break
    # save the model_state with the best performance
    torch.save(best_model_state, save_path)
    print(f"Best autoencoder model saved at {save_path}")

    return history
```

D.3. CLIP-featured Model

clip_segmentation.py

```
# The related blocks of CLIP segmentation models are store all-in-one here

import torch
import torch.nn as nn
######################
# Feature Extraction #
######################

```

```python
class CLIPFeatureExtractor(nn.Module):
    def __init__(self, clip_model):
        super().__init__()
        visual = clip_model.visual
        self.conv1 = visual.conv1
        self.bn1 = visual.bn1
        self.relu1 = visual.relu1
        self.conv2 = visual.conv2
        self.bn2 = visual.bn2
        self.relu2 = visual.relu2
        self.conv3 = visual.conv3
        self.bn3 = visual.bn3
        self.relu3 = visual.relu3

        self.layer1 = visual.layer1
        self.layer2 = visual.layer2
        self.layer3 = visual.layer3
        self.layer4 = visual.layer4

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.relu2(x)
        x = self.conv3(x)
        x = self.bn3(x)
        x = self.relu3(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)
        return x

#######################
## Segmentation Head ##
#######################

class CLIPSegmentationHead(nn.Module):
    """
    Updated segmentation head for ModifiedResNet in CLIP.
    Adjusted for 2048 input channels and 14x14 input spatial size.
    """
    def __init__(self, input_channels, output_channels):
        super().__init__()

        # Reduce channels first
        self.reduce_channels = nn.Sequential(
            nn.Conv2d(input_channels, 1024, kernel_size=1),  # Reduce channels 2048 > 1024
            nn.BatchNorm2d(1024),
            nn.ReLU()
        )

        # Upsampling layers
        self.upsample1 = nn.Sequential(
            nn.ConvTranspose2d(1024, 512, kernel_size=3, stride=2, padding=1, output_padding=1),  # 14x14 > 28x28
            nn.BatchNorm2d(512),
            nn.ReLU()
        )

        self.upsample2 = nn.Sequential(
            nn.Upsample(scale_factor=8, mode='bilinear', align_corners=True),  # 112x112 > 224x224
            nn.Conv2d(512, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
```

```
        )

        # Final segmentation layer
        self.final_conv = nn.Conv2d(64, output_channels, kernel_size=1)  # Output segmentation map

    def forward(self, x):
        x = self.reduce_channels(x)  # Reduce channels: (B, 20248, 14, 14) > (B, 1024, 14, 14)
        x1 = self.upsample1(x)  # 14x14 > 28x28
        x2 = self.upsample2(x1) # 112x112 > 224x224
        out = self.final_conv(x2)

        return out

#####################################
### Final CLIP Segmentation model ###
#####################################

class CLIPSegmentationModel(nn.Module):
    """
    Lightweight CLIP-based segmentation model using only the necessary vision layers from RN50.
    """
    def __init__(self, clip_model, num_classes):
        super().__init__()

        # Replace full CLIP model with a custom extractor that only includes needed layers
        self.feature_extractor = CLIPFeatureExtractor(clip_model)  # defined above
        for param in self.feature_extractor.parameters():
            param.requires_grad = False  # Freeze backbone

        self.seg_head = CLIPSegmentationHead(input_channels=2048, output_channels=num_classes)

    def forward(self, x):
        with torch.no_grad():
            x = self.feature_extractor(x)
        x = self.seg_head(x)
        return x
```

D.4. Prompt-based Model

prompt_segmentation.py

```
import torch
import torch.nn as nn
from models.unet_segmentation import UNet

class UNetEncoder(nn.Module):
    def __init__(self, in_channels, init_features):
        super().__init__()
        f = init_features
        self.encoder1 = self._conv_block(in_channels, f)
        self.encoder2 = self._conv_block(f, f * 2)
        self.encoder3 = self._conv_block(f * 2, f * 4)
        self.encoder4 = self._conv_block(f * 4, f * 8)
        self.bottleneck = self._conv_block(f * 8, f * 16)
        self.bottleneck_channels = f * 16
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

    def _conv_block(self, in_c, out_c):
        return nn.Sequential(
            nn.Conv2d(in_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
        )

```

```python
    def forward(self, x):
        enc1 = self.encoder1(x)
        enc2 = self.encoder2(self.pool(enc1))
        enc3 = self.encoder3(self.pool(enc2))
        enc4 = self.encoder4(self.pool(enc3))
        bottleneck = self.bottleneck(self.pool(enc4))
        return [enc1, enc2, enc3, enc4], bottleneck

class UNetDecoder(nn.Module):
    def __init__(self, bottleneck_channels, out_channels, init_features):
        super().__init__()
        f = init_features

        self.up4 = nn.ConvTranspose2d(bottleneck_channels, f * 8, kernel_size=2, stride=2)
        self.dec4 = self._conv_block(f * 8 + f * 8, f * 8)

        self.up3 = nn.ConvTranspose2d(f * 8, f * 4, kernel_size=2, stride=2)
        self.dec3 = self._conv_block(f * 4 + f * 4, f * 4)

        self.up2 = nn.ConvTranspose2d(f * 4, f * 2, kernel_size=2, stride=2)
        self.dec2 = self._conv_block(f * 2 + f * 2, f * 2)

        self.up1 = nn.ConvTranspose2d(f * 2, f, kernel_size=2, stride=2)
        self.dec1 = self._conv_block(f + f, f)

        self.final = nn.Conv2d(f, out_channels, kernel_size=1)

    def _conv_block(self, in_c, out_c):
        return nn.Sequential(
            nn.Conv2d(in_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_c, out_c, 3, padding=1),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
        )

    def forward(self, enc_feats, bottleneck):
        enc1, enc2, enc3, enc4 = enc_feats

        d4 = self.up4(bottleneck)
        d4 = self.dec4(torch.cat([d4, enc4], dim=1))

        d3 = self.up3(d4)
        d3 = self.dec3(torch.cat([d3, enc3], dim=1))

        d2 = self.up2(d3)
        d2 = self.dec2(torch.cat([d2, enc2], dim=1))

        d1 = self.up1(d2)
        d1 = self.dec1(torch.cat([d1, enc1], dim=1))

        return self.final(d1)


class PointPromptEncoder(nn.Module):
    """
    Encode prompt points into a spatial representation using pre-generated heatmaps
    """
    def __init__(self, input_channels=1, output_dim=1024, num_classes=3):
        """
        Args:
            input_channels (int): Number of input channels for the prompt heatmap.
            output_dim (int): Dimension of the final output representation.
            num_classes (int): Number of classes for point classification.
        """
        super().__init__()
        #  define the intermediate hidden dimension
```

```
        self.hidden_dim = output_dim // 2
        # Point class embedding (optional)
        self.class_embedding = nn.Embedding(num_embeddings=num_classes, embedding_dim=output_dim) # 3 classes (0=background, 1=cat, 2=dog)

        # Spatial encoding mechanism
        self.spatial_encoder = nn.Sequential(
            nn.Conv2d(input_channels, self.hidden_dim, kernel_size=3, padding=1, stride=4),
            nn.ReLU(),
            nn.Conv2d(self.hidden_dim, output_dim, kernel_size=3, padding=1, stride=4)
        )

    def forward(self, prompt_heatmap, point_class=None):
        """
        Args:
            prompt_heatmap (torch.Tensor): Pre-generated heatmap from dataset (B, 1, H, W)
            point_class (torch.Tensor, optional): Class labels for the prompt points (B,)

        Returns:
            torch.Tensor: Encoded prompt embedding (B, output_dim, H, W)
        """
        B = prompt_heatmap.shape[0]

        # Spatially encode heatmap
        spatial_prompt = self.spatial_encoder(prompt_heatmap)

        # Incorporate class information if provided
        if point_class is not None:
            class_embed = self.class_embedding(point_class)  # (B, output_dim)
            class_embed = class_embed.view(B, -1, 1, 1)  # (B, output_dim, 1, 1)

            # Broadcast class embedding across spatial dimensions so the final output is globally class-aware
            class_embed = class_embed.expand(-1, -1, spatial_prompt.size(2), spatial_prompt.size(3))

            # Combine class information with spatial prompt
            return spatial_prompt + class_embed

        return spatial_prompt

class PromptSegmentation(nn.Module):
    def __init__(self, unet_in_channels=3, prompt_dim=1024, unet_init_features=64):
        super().__init__()

        # bottleneck features = init_features * 16 -> (64 * 16 = 1024)
        self.encoder = UNetEncoder(in_channels=unet_in_channels, init_features=unet_init_features)
        # prompt_encoder output channels = 64 * 16 = 1024
        self.prompt_encoder = PointPromptEncoder(input_channels=1, output_dim=prompt_dim, num_classes=3)  # 3 classes (0=background, 1=cat, 2=dog)
        # decoder expects (B, 1024 32, 32) from encoder concentated (B, 1024, 32, 32) from prompt_encoder
        self.decoder = UNetDecoder(bottleneck_channels=self.encoder.bottleneck_channels + prompt_dim,
                                   out_channels=1, init_features=unet_init_features)


    def forward(self, image, prompt_heatmap, point_class=None):
        # Step 1: Encode image -> get encoder features and bottleneck
        enc_features, image_bottleneck = self.encoder(image)

        # Step 2: Encode prompt heatmap
        prompt_features = self.prompt_encoder(prompt_heatmap, point_class)  # (B, prompt_dim, H/16, W/16)

        # Step 3: Concatenate prompt with image bottleneck
        combined_bottleneck = torch.cat([image_bottleneck, prompt_features], dim=1)
```

```

        # Step 4: Decode
        mask_logits = self.decoder(enc_features, combined_bottleneck)

        return mask_logits
```

E. Training Utilities

This appendix presents the core components related to model training. It includes modular training functions tailored to the nuances of each model type, visualization tools for tracking training progress, and customized loss functions for robust segmentation performance.

E.1. Training Functions *training.py*

training.py
Provides general training loops, including support for both standard and prompt-based segmentation models. Implements early stopping, optimizer setup, and performance evaluation during training.

```
import os
from pathlib import Path
from tqdm import tqdm
import torch
from data.preprocessing import create_point_wise_mask

#########################
# Segmentaiton Training #
#########################
def training(
    model, train_loader, val_loader,
    train_criterion, val_criterion,  # Use different loss functions
    optimizer, num_epochs=10, device="cuda",
    save_dir="../params", save_name=None,
    patience=5):
    # assgin model
    model.to(device).float()

    # print model-related info
    print("    ␣Model␣Structure:\n", model)

    # Count total parameters and trainable parameters
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)

    print(f"    ␣Total␣Parameters:␣{total_params:,}")
    print(f"    ␣Trainable␣Parameters:␣{trainable_params:,}")

    # Estimate model size (in MB)
    param_size_MB = total_params * 4 / (1024 ** 2)  # float32 => 4 bytes
    print(f"    ␣Estimated␣Parameter␣Size:␣{param_size_MB:.2f}␣MB")

    # training utils
    history = {"train_loss": [], "val_loss": []}

    os.makedirs(save_dir, exist_ok=True)  # Create checkpoint directory
    save_path = Path(save_dir) / save_name

    # Initialize Early Stopping Variables
    best_val_loss = float("inf")  # Set to a large value
    best_model_config = None  # To store the best model parameters
    epochs_no_improve = 0  # Counter for non-improving epochs

    for epoch in range(num_epochs):
        #      Training Phase
        model.train()
        running_loss = 0.0

```

```
        for images, masks, _,_ in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Train]", unit=' Batches'):
            images, masks = images.to(device), masks.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = train_criterion(outputs, masks)

            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        history["train_loss"].append(train_loss)

        # Validation Phase
        model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for images, masks, _,_ in tqdm(val_loader, desc=f"Epoch {epoch+1}/{num_epochs} [Val]", unit=' Batches'):
                images, masks = images.to(device), masks.to(device)

                outputs = model(images)
                loss = val_criterion(outputs, masks)

                val_loss += loss.item()

        val_loss /= len(val_loader)
        history["val_loss"].append(val_loss)
        print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {train_loss:.6f} | Val Loss: {val_loss:.6f}")

        # Early Stopping Logic
        if val_loss < best_val_loss:
            best_val_loss = val_loss  # Update best loss
            best_epoch = epoch + 1
            epochs_no_improve = 0  # Reset counter
            best_model_config = model.state_dict()
        else:
            epochs_no_improve += 1  # Increment counter
            print(f"No improvement for {epochs_no_improve} epochs.")

        if epochs_no_improve >= patience:
            # Save model
            torch.save(best_model_config, save_path)  # Save best model
            print(f"Early stopping triggered after {epoch+1} epochs!")
            print(f"Best val loss at epoch {best_epoch}: {best_val_loss:.8}")
            print(f"  Best  model config saved at: {save_path}")
            return history

    torch.save(best_model_config, save_path)  # save the model if the early stop is not triggered
    print(f"  Model  config saved at: {save_path} in epoch {best_epoch}")
    return history

def prompt_training(
    model, train_loader, val_loader,
    train_criterion, val_criterion,
    optimizer, num_epochs, device="cuda",
    save_dir="../params", save_name= None,
    patience=5):
    model.to(device).float()

    ############################
    # Print model-related info #
    ############################
```

```
    print("    ␣Model␣Structure:\n", model)

    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    param_size_MB = total_params * 4 / (1024 ** 2)  # float32 => 4 bytes

    print(f"    ␣Total␣Parameters:␣{total_params:,}")
    print(f"    ␣Trainable␣Parameters:␣{trainable_params:,}")
    print(f"    ␣Estimated␣Parameter␣Size:␣{param_size_MB:.2f}␣MB")

    ##################
    # Training Setup #
    ##################
    history = {"train_loss": [], "val_loss": []}
    os.makedirs(save_dir, exist_ok=True)
    save_path = Path(save_dir) / save_name

    best_val_loss = float("inf")
    best_model_config = None
    epochs_no_improve = 0

    for epoch in range(num_epochs):
        # Training Phase
        model.train()
        running_loss = 0.0

        for batch in tqdm(train_loader, desc=f"Epoch␣{epoch+1}/{num_epochs}␣[Train]", unit='␣Batches'):
            images = batch['image'].to(device)               # (B, 3, H, W)
            gt_masks = batch['gt_mask'].to(device)           # (B, 1, H, W)
            prompt_heatmaps = batch['prompt_heatmap'].to(device) # (B, 1, H, W)
            point_classes = batch['point_class'].to(device)      # (B,)

            # Create target masks based on point class
            target_masks = create_point_wise_mask(
                point_classes,
                gt_masks
            )

            optimizer.zero_grad()

            # Forward pass
            outputs = model(
                image=images,
                prompt_heatmap=prompt_heatmaps,
                point_class=None
            )

            # Calculate loss
            loss = train_criterion(outputs, target_masks)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        history["train_loss"].append(train_loss)

        # Validation Phase
        model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for batch in tqdm(val_loader, desc=f"Epoch␣{epoch+1}/{num_epochs}␣[Val]", unit='␣Batches'):
                images = batch['image'].to(device)               # (B, 3, H, W)
                gt_masks = batch['gt_mask'].to(device)           # (B, 1, H, W)
                prompt_heatmaps = batch['prompt_heatmap'].to(device) # (B, 1, H, W)
```

```
                point_classes = batch['point_class'].to(device)      # (B,)

                    # Create target masks based on point class
                target_masks = create_point_wise_mask(
                    point_classes,
                    gt_masks
                )

                # Forward pass
                outputs = model(
                    image=images,
                    prompt_heatmap=prompt_heatmaps,
                    point_class=None
                )
                loss = val_criterion(outputs, target_masks)

                val_loss += loss.item()

        val_loss /= len(val_loader)
        history["val_loss"].append(val_loss)
        print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {train_loss:.6f} | Val Loss: {val_loss
            :.6f}")

        # Early Stopping Logic
        if val_loss < best_val_loss:
            best_val_loss = val_loss  # Update best loss
            best_epoch = epoch + 1
            epochs_no_improve = 0  # Reset counter
            best_model_config = model.state_dict() # record the best model checkpoint so far
            # Ensure the directory exists
            tmp_save_dir = Path(save_dir) / "tmp_prompt_checkpoint"
            tmp_save_dir.mkdir(parents=True, exist_ok=True)

            # Construct save path
            tmp_save_name = f'best_prompt_checkpoint_epoch_{epoch+1}.pth'
            tmp_save_path = tmp_save_dir / tmp_save_name

            torch.save(best_model_config, tmp_save_path)
        else:
            epochs_no_improve += 1  # Increment counter
            print(f"No improvement for {epochs_no_improve} epochs.")

        if epochs_no_improve >= patience:
            # Save model
            torch.save(best_model_config, save_path)  # Save best model
            print(f"Early stopping triggered after {epoch+1} epochs!")
            print(f"Best val loss at epoch {best_epoch}: {best_val_loss:.8}")
            print(f"  Best  model config saved at: {save_path}")
            return history

    torch.save(best_model_config, save_path)  # save the model if the early stop is not triggered
    print(f"  Model  config saved at: {save_path} (early stop not triggered)")
    return history
```

E.2. Training Visualization *training_plot.py*

Generates loss and metric plots across training epochs for both training and validation sets, aiding in model performance analysis and debugging. ***training_plot.py***

```
import matplotlib.pyplot as plt

def training_plot(history, save_path=None):
    """
    Plots the training and validation loss curves from the history dict.

    Args:
```

```python
        history (dict): Contains 'train_loss' and 'val_loss' lists.
        save_path (str, optional): If provided, saves the plot to this path.
        title (str): Title of the plot.
    """
    plt.figure(figsize=(8, 6))
    plt.plot(history['train_loss'], label='Train Loss', marker='o', ms=4)
    plt.plot(history['val_loss'], label='Validation Loss', marker='o', ms=4)
    plt.xlabel('Epoch')
    plt.legend()
    plt.grid(True)

    if save_path:
        plt.savefig(save_path, dpi=300)
        print(f"Loss plot saved to {save_path}")
    else:
        plt.show()

    plt.close()
```

E.3. Loss Functions *loss_functions.py*

Defines custom loss functions including Focal Loss, Dice Loss, IoU Loss, and combined variants. These are tailored to handle class imbalance and improve segmentation accuracy. ***loss_functions.py***

```python
import torch.nn as nn
import torch
import torch.nn.functional as F
######################
### Loss Functions ###
######################

class DiceLoss(nn.Module):
    '''
    calculate the (1 - Dice) loss
    Args:
        ouputs(torch.Tensor): model outputs, shape (B, 3, 512, 512)
        masks(torch.Tensor): ground truth, shape (B, 512, 512)
    Returns:
        (1 - mean_dice_over_classes) Loss
    '''
    def __init__(self, smooth=1e-6):
        super(DiceLoss, self).__init__()
        self.smooth = smooth

    def forward(self, outputs, masks):
        # Convert logits to probabilities using softmax
        outputs = F.softmax(outputs, dim=1)

        # Convert masks to one-hot encoding
        masks_one_hot = F.one_hot(masks.long(), num_classes=outputs.shape[1]).permute(0, 3, 1, 2)
            .float()

        # Compute Dice Score
        intersection = torch.sum(outputs * masks_one_hot, dim=(2, 3))
        union = torch.sum(outputs, dim=(2, 3)) + torch.sum(masks_one_hot, dim=(2, 3))

        dice_score = (2. * intersection + self.smooth) / (union + self.smooth)
        return 1 - dice_score.mean()

class IouLoss(nn.Module):
    '''
    calculate the (1 - IoU) loss
    Args:
        ouputs(torch.Tensor): model outputs, shape (B, 3, 512, 512)
        masks(torch.Tensor): ground truth, shape (B, 512, 512)
    Returns:
```

```
        (1 - mean_iou_over_classes) Loss
    '''
    def __init__(self, smooth=1e-6):
        super().__init__()
        self.smooth = smooth
    def forward(self, outputs, masks):
        # Convert logits to probabilities using softmax
        outputs = F.softmax(outputs, dim=1)

        # Convert masks to one-hot encoding
        masks_one_hot = F.one_hot(masks.long(), num_classes=outputs.shape[1]).permute(0, 3, 1, 2).float()

        # Compute IoU Score
        intersection = torch.sum(outputs * masks_one_hot, dim=(2, 3))
        union = torch.sum(outputs + masks_one_hot, dim=(2, 3)) - intersection

        iou_score = (intersection + self.smooth) / (union + self.smooth)
        return 1 - iou_score.mean()

    def forward(self, outputs, masks):
        # Convert logits to probabilities using softmax
        outputs = F.softmax(outputs, dim=1)

        # Convert masks to one-hot encoding
        masks_one_hot = F.one_hot(masks.long(), num_classes=outputs.shape[1]).permute(0, 3, 1, 2).float()

        # Compute Dice Score
        intersection = torch.sum(outputs * masks_one_hot, dim=(2, 3))
        union = torch.sum(outputs, dim=(2, 3)) + torch.sum(masks_one_hot, dim=(2, 3))

        dice_score = (2. * intersection + self.smooth) / (union + self.smooth)
        return 1 - dice_score.mean()

class FocalLoss(nn.Module):
    """
    Focal Loss for class imbalance.
    Args:
        gamma (float): Focusing parameter (>1 focuses more on hard samples).
        alpha (tensor, optional): Class weighting (for additional balance).
        reduction (str): 'mean' or 'sum'.
    Returns:
        torch.Tensor: Focal loss.
    """
    def __init__(self, gamma=2.0, alpha=None, reduction='mean'):
        """
        Args:
            gamma (float): Focusing parameter (>1 focuses more on hard samples).
            alpha (tensor, optional): Class weighting (for additional balance).
            reduction (str): 'mean' or 'sum'.
        """
        super().__init__()
        self.gamma = gamma
        self.alpha = alpha
        self.reduction = reduction

    def forward(self, inputs, targets):
        """
        Args:
            inputs (torch.Tensor): Model logits (B, C, H, W).
            targets (torch.Tensor): Ground truth (B, H, W).

        Returns:
            torch.Tensor: Focal loss.
        """
        log_probs = F.log_softmax(inputs, dim=1)  # Convert logits to log-probs
        probs = torch.exp(log_probs)              # Convert to probabilities
```

```
        # Gather log-probabilities of correct class
        target_log_probs = log_probs.gather(1, targets.unsqueeze(1))  # (B, 1, H, W)
        target_probs = probs.gather(1, targets.unsqueeze(1))

        # Compute Focal Loss term
        focal_weight = (1 - target_probs) ** self.gamma  # Focus on hard examples

        if self.alpha is not None:
            alpha_factor = self.alpha[targets] if isinstance(self.alpha, torch.Tensor) else self.alpha
            focal_weight = focal_weight * alpha_factor

        loss = -focal_weight * target_log_probs  # Apply weighting

        if self.reduction == 'mean':
            return loss.mean()
        elif self.reduction == 'sum':
            return loss.sum()
        else:
            return loss

class CombinedFocalDiceLoss(nn.Module):
    """
    Combined Loss = (1 - alpha)*FocalLoss + alpha*DiceLoss
    Args:
            alpha (float): Balance between Focal and Dice Loss.
            gamma (float): Focal Loss focusing parameter.
            focal_alpha (tensor, optional): Class weighting for Focal Loss.
            smooth (float): Smoothing factor for Dice Loss.
    Returns:
            torch.Tensor: Combined loss.
    """
    def __init__(self, alpha=0.5, gamma=2.0, focal_alpha=None, smooth=1e-6):
        super().__init__()
        self.alpha = alpha
        self.focal_loss = FocalLoss(gamma=gamma, alpha=focal_alpha)
        self.dice_loss = DiceLoss(smooth=smooth)

    def forward(self, outputs, masks):
        fl = self.focal_loss(outputs, masks)
        dl = self.dice_loss(outputs, masks)
        return (1 - self.alpha) * fl + self.alpha * dl

# Binary Loss Function
class BinaryFocalLoss(nn.Module):
    '''
    Focal Loss for binary classification in prompt-based training.
    Args:
        alpha (float): Balance between positive and negative classes.
        gamma (float): Focusing parameter (>1 focuses more on hard samples).
        reduction (str): 'mean' or 'sum'.
    Returns:
        torch.Tensor: Focal loss.
    '''
    def __init__(self, alpha=0.25, gamma=2.0, reduction='mean'):
        super().__init__()
        self.gamma = gamma
        self.alpha = alpha
        self.reduction = reduction

    def forward(self, logits, targets):
        # Apply sigmoid to get probabilities
        probs = torch.sigmoid(logits)
        probs = probs.clamp(min=1e-7, max=1 - 1e-7)  # avoid log(0)

        # Focal loss formula
        pt = probs * targets + (1 - probs) * (1 - targets)
```

```
        alpha_t = self.alpha * targets + (1 - self.alpha) * (1 - targets)
        focal_loss = -alpha_t * (1 - pt) ** self.gamma * torch.log(pt)

        if self.reduction == 'mean':
            return focal_loss.mean()
        elif self.reduction == 'sum':
            return focal_loss.sum()
        else:
            return focal_loss
```

F. Inference & Evaluation

This appendix shows the pipeline of inference and evaluation and related utilities.

F.1. Modular Inference Runner *run_inference.py*

run_inference.py

```
#!/usr/bin/env python

import argparse
import torch
import os
import clip
from pathlib import Path

# import models
# Import your models
from models.unet_segmentation import UNet
from models.autoencoder_segmentation import AutoEncoder, AutoEncoderSegmentation  # <-- Import your Autoencoder models
from models.clip_segmentation import CLIPSegmentationModel
from models.prompt_segmentation import PromptSegmentation
from utils.inference import inference, promptInference  # Ensure you have a separate inference function

# Define function to load the correct model
def load_model(mode, checkpoint_path, device, pretrain_path=None):
    """
    Load the segmentation model based on the selected mode.

    Args:
        mode (int): 0 for U-Net, 1 for Autoencoder, 2 for CLIP.
        checkpoint_path (str): Path to the trained model checkpoint.
        device (str): Device to use ("cuda" or "cpu").

    Returns:
        model (torch.nn.Module): The loaded model.
    """

    #################################
    ############ U-Net ##############
    #################################
    if mode == 0:
        print("Loading U-Net model... ")
        model = UNet()
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint)
        model.to(device)
        model.eval()
        return model

    #################################
    ########## Autoencoder ##########
    #################################
    elif mode == 1:
        print("Loading Autoencoder Segmentation Model...")
```

```python

        if pretrain_path is None:
            raise ValueError("      Pretrained autoencoder path must be provided in Autoencoder mode (--pretrain_path).")

        # Load pretrained autoencoder
        autoencoder = AutoEncoder()
        pretrain_checkpoint = torch.load(pretrain_path, map_location=device)
        autoencoder.load_state_dict(pretrain_checkpoint)
        autoencoder.to(device)

        # Wrap the frozen encoder with the segmentation head
        model = AutoEncoderSegmentation(encoder=autoencoder.encoder, num_classes=3)

        # Load segmentation checkpoint (contains trained decoder)
        seg_checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(seg_checkpoint)

        model.to(device)
        model.eval()
        return model

    #################################
    ############ CLIP ###############
    #################################
    elif mode == 2:
        print("Loading CLIP Segmentation model...")
        clip_model, _ = clip.load("RN50", device=device)
        model = CLIPSegmentationModel(clip_model=clip_model, num_classes=3)
        # Load checkpoint
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint)
        model.to(device)
        model.float()
        model.eval()
        return model

    ################
    # Prompt-based #
    ################
    elif mode ==3:
        print("Loading Prompt Segementation model...")
        model = PromptSegmentation()
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint)
        model.to(device)
        model.float()
        model.eval()
        return model

    else:
        raise ValueError("Invalid mode! Use 0 for U-Net, 1 for Autoencoder, 2 for CLIP or 3 for prompt model.")

# Parse command-line arguments
def parse_args():
    parser = argparse.ArgumentParser(description="Run inference for image segmentation.")
    parser.add_argument("--image_dir", type=str, required=True, help="Path to input images")
    parser.add_argument("--mask_dir", type=str, required=True, help="Path to ground truth masks")
    parser.add_argument("--point_dir", type=str, default=None, help="Path to prompt points (only for prompt model)")
    parser.add_argument("--save_dir", type=str, required=True, help="Path to save output masks")
    parser.add_argument("--checkpoint_path", type=str, required=True, help="Path to model checkpoint")
    parser.add_argument("--pretrain_path", type=str, default=None, help="Path to pretrained autoencoder (required for autoencoder mode)")
    parser.add_argument("--target_size", type=int, default=512, help="Target input size for model")
```

```python
    parser.add_argument("--device", type=str, required=True, help="Device for inference (cuda/cpu)")
    parser.add_argument("--mode", type=int, required=True, help="Model selection: 0 for U-Net, 1 for Autoencoder, 2 for CLIP, 3 for prompt model")

    return parser.parse_args()

if __name__ == "__main__":
    args = parse_args()

    # Ensure save directory exists
    os.makedirs(args.save_dir, exist_ok=True)

    # Load model
    model = load_model(args.mode, args.checkpoint_path, args.device, pretrain_path=args.pretrain_path)
    # Run inference
    if args.mode in [0,1,2]:
        inference(
            image_dir=args.image_dir,
            mask_dir=args.mask_dir,
            save_dir=args.save_dir,
            mode = args.mode,
            input_image_size=args.target_size,
            model=model,
            device=args.device
        )
    elif args.mode == 3:
        promptInference(
            image_dir=args.image_dir,
            mask_dir=args.mask_dir,
            point_dir= args.point_dir,
            gt_save_dir= Path(args.save_dir) / "gt/",
            pred_save_dir=Path(args.save_dir) / "pred/",
            threshold=0.5,
            input_image_size=args.target_size,
            model=model,
            device=args.device
        )
    else:
        raise ValueError("Invalid mode! Use 0 for U-Net, 1 for Autoencoder, 2 for CLIP or 3 for prompt model.")
```

F.2. Inference Blocks *inference.py*

inference.py

```python
# run_inference.py

import torch
from data.preprocessing import (class2color,
                                resize_with_padding,
                                clip_transform,
                                standard_transform,
                                create_point_wise_mask)
import os
from utils.restore_mask_size import restore_original_mask
from data.PetDataset import PetDataset, PetDatasetWithPrompt
from torch.utils.data import DataLoader
from pathlib import Path
from tqdm import tqdm
import numpy as np
from PIL import Image

def custom_collate_fn(batch):
    images, masks, initial_img_sizes, filenames = zip(*batch)  # Unpack batch

    # Convert images and masks to tensors (PyTorch default behavior)
```

```python
    images = torch.stack(images, dim=0)
    masks = torch.stack(masks, dim=0)

    return images, masks, list(initial_img_sizes), list(filenames)  # Keep `initial_img_sizes` as list of tuples

def inference(image_dir: str, mask_dir: str, save_dir: str, input_image_size: int,
              mode: int, model, device: str) -> np.ndarray:
    """
    Perform inference on input images and save the predicted segmentation masks.

    Args:
        image_dir (str): Directory containing input images.
        mask_dir (str): Directory containing ground truth masks (for size reference).
        save_dir (str): Directory to save the output predicted masks.
        input_image_size (int): Target size for resizing (used in CLIP mode).
        mode (int): Model mode selector:
                    0 - U-Net,
                    1 - Autoencoder,
                    2 - CLIP.
        model (torch.nn.Module): Pre-loaded segmentation model ready for inference.
        device (str): Computation device ("cuda" or "cpu").

    Returns:
        None. (Predicted color masks are saved to 'save_dir')
    """

    # Initialize paths of images (list of str)
    image_paths = sorted(Path(image_dir).glob("*.*"))
    mask_paths = sorted(Path(mask_dir).glob("*.*"))

    # assign the correct torch dataset format
    if mode == 0 or mode == 1:
        test_dataset = PetDataset(
        img_paths = image_paths,
        msk_paths = mask_paths,
        resize_fn= resize_with_padding,
        resize_target_size= 512,
        transform = standard_transform)
    elif mode == 2:
        test_dataset = PetDataset(
            img_paths = image_paths,
            msk_paths = mask_paths,
            resize_fn = resize_with_padding,
            resize_target_size = input_image_size,
            transform = clip_transform
        )

    else:
        raise ValueError("Invalid mode. Use 0 for U-Net, 1 for autoencoder-based segmentation or 2 for CLIP-based segmentation.")

    test_loader = DataLoader(test_dataset,batch_size=4, num_workers=4, collate_fn=custom_collate_fn)


    with torch.no_grad():
        total_batches = len(test_loader)  # Get total number of batches
        for images, _, initial_img_sizes, filenames in \
        tqdm(test_loader, desc=f"Processing {len(test_loader.dataset)} images in {total_batches} batches"):
            images = images.to(device)
            outputs = model(images)

            # Convert logits to class labels (B, C, H, W)    (B, H, W)
            pred_masks = torch.argmax(outputs, dim=1).cpu().numpy()
            resized_pred_masks = [restore_original_mask(pred_masks[i], initial_img_sizes[i]) for i in range(len(images))]
```

```python

            # Save predicted masks
            for i, filename in enumerate(filenames):
                pred_mask_img = class2color(resized_pred_masks[i])  # Convert class labels to color mask
                # Ensure data type is uint8 to avoid artifacts
                pred_mask_img = np.array(pred_mask_img, dtype=np.uint8)
                save_img = Image.fromarray(pred_mask_img, mode="RGB") # Convert the ndarray into RGB Image
                save_path = os.path.join(save_dir, Path(filename).stem+'.png')  # Keep original filename
                save_img.save(save_path,format='PNG')

    # print the dst directory
    print(f"Predicted masks saved to {save_dir}")

def promptInference(image_dir: str, mask_dir:str, point_dir:str,
                    gt_save_dir:str, pred_save_dir:str,
                    threshold:float, input_image_size: int,
                    model: callable, device: str) -> np.ndarray:
    '''
    Run the inference on test images with input of prompt points

    Args:
        image_dir (str): Directory containing input images
        mask_dir (str): Directory containing ground truth mask images
        point_dir (str): Directory containing point prompts
        gt_save_dir (str): Directory where ground truth masks will be saved
        pred_save_dir (str): Directory where predicted masks will be saved
        threshold (float): Threshold value for binary segmentation (sigmoid output > threshold)
        input_image_size (int): Target size for resizing images before inference
        model (callable): The segmentation model to use for inference
        device (str): Device to run inference on (e.g., 'cuda', 'cpu')

    Returns:
        np.ndarray: Processed data array containing segmentation results

    Note:
        - The function processes batches of images with corresponding masks and point prompts
        - Predicted masks are saved as PNG files in pred_save_dir
        - Ground truth masks are saved as PNG files in gt_save_dir
        - File names are preserved from the original image files
    '''
    image_paths = sorted(Path(image_dir).glob("*.*"))
    mask_paths = sorted(Path(mask_dir).glob("*.*"))
    point_paths = sorted(Path(point_dir).glob("*.*"))

    test_dataset = PetDatasetWithPrompt(
        img_paths=image_paths,
        msk_paths=mask_paths,
        pnt_paths=point_paths,
        resize_fn=resize_with_padding,
        resize_target_size=input_image_size,
        transform=standard_transform,
        load_multiple_points=True
    )

    test_loader = DataLoader(test_dataset,batch_size=4, num_workers=4,collate_fn=prompt_custom_collate_fn)


    with torch.no_grad():
        total_batches = len(test_loader)  # Get total number of batches
        for batch in tqdm(test_loader, desc=f"Processing {len(test_loader.dataset)} images in {total_batches} batches"):
            images = batch['image'].to(device)               # (B, 3, H, W)
            gt_masks = batch['gt_mask'].to(device)           # (B, 1, H, W)
            prompt_heatmaps = batch['prompt_heatmap'].to(device) # (B, 1, H, W)
```

```python
            point_classes = batch['point_class'].to(device)      # (B,)
            initial_img_sizes = batch['initial_img_size']
            filenames = batch['img_path']
            # Create target masks based on point class
            target_masks = create_point_wise_mask(
                point_classes,
                gt_masks
            )

            with torch.no_grad():
                output = model(image=images,
                               prompt_heatmap=prompt_heatmaps,
                               point_class = point_classes)

            # Convert logits to class labels (B, C, H, W)      (B, H, W)
            pred_masks = torch.sigmoid(output) > threshold
            # restore the gt and pred masks shape
            pred_masks = pred_masks.squeeze(1).cpu().numpy().astype(np.uint8)
            target_masks = target_masks.squeeze(1).cpu().numpy().astype(np.uint8)

            resized_pred_masks = [restore_original_mask(pred_masks[i], initial_img_sizes[i][:2]) for i in range(len(images))]
            resized_gt_masks = [restore_original_mask(target_masks[i], initial_img_sizes[i][:2]) for i in range(len(images))]
            # Save prompt gt binary mask
            for i, filename in enumerate(filenames):
                mask_arr = (resized_gt_masks[i]*255).astype(np.uint8)
                mask_save = Image.fromarray(mask_arr, mode='L')
                mask_save_path = os.path.join(gt_save_dir, Path(filename).stem+'.png')  # Keep original filename
                # Ensure parent directory exists
                Path(mask_save_path).parent.mkdir(parents=True, exist_ok=True)
                mask_save.save(mask_save_path,format='PNG')

            # Save predicted masks
            for i, filename in enumerate(filenames):
                pred_arr = (resized_pred_masks[i]*255).astype(np.uint8)
                pred_save = Image.fromarray(pred_arr, mode="L") # Convert the ndarray into RGB Image
                pred_save_path = os.path.join(pred_save_dir, Path(filename).stem+'.png')  # Keep original filename
                # Ensure parent directory exists
                Path(pred_save_path).parent.mkdir(parents=True, exist_ok=True)
                pred_save.save(pred_save_path,format='PNG')

def prompt_custom_collate_fn(batch):
    # batch is a list of dictionaries, one per sample
    # We'll build batched outputs manually

    images = torch.stack([sample['image'] for sample in batch], dim=0)
    gt_masks = torch.stack([sample['gt_mask'] for sample in batch], dim=0)
    prompt_heatmaps = torch.stack([sample['prompt_heatmap'] for sample in batch], dim=0)
    point_classes = torch.stack([sample['point_class'] for sample in batch], dim=0)
    # Keep `initial_img_size` as a list instead of stacking into a Tensor
    initial_img_sizes = [sample['initial_img_size'] for sample in batch]
    img_paths = [sample['img_path'] for sample in batch]

    return {
        'image': images,
        'gt_mask': gt_masks,
        'prompt_heatmap': prompt_heatmaps,
        'point_class': point_classes,
        'initial_img_size': initial_img_sizes,
        'img_path': img_paths
    }
```

F.3. Mask Size Restoration $restore_{m}ask_{s}ize.py$

It restores the predicted masks to the initial sizes for the fair calculation of IoU ***restore_mask_size.py***

```python
import cv2 as cv
import numpy as np

def restore_original_mask(mask: np.ndarray, original_size: tuple) -> np.ndarray:
    """
    Resize the class mask back to the original image size.

    Args:
        mask (np.ndarray): The resized mask of shape (target_size, target_size).
        original_size (tuple): The original image dimensions (H, W).

    Returns:
        np.ndarray: The restored mask with original dimensions (H, W).
    """
    target_size = mask.shape[0]  # Assuming square mask (224, 224)
    orig_h, orig_w = original_size

    # Compute scale factor (same used in resize_with_padding)
    scale = target_size / max(orig_w, orig_h)
    new_w, new_h = int(orig_w * scale), int(orig_h * scale)

    # Compute padding offsets
    paste_x = (target_size - new_w) // 2
    paste_y = (target_size - new_h) // 2

    # Crop the valid region (remove padding)
    cropped_mask = mask[paste_y:paste_y + new_h, paste_x:paste_x + new_w]

    # Resize back to original dimensions using NEAREST to keep discrete labels
    original_mask = cv.resize(cropped_mask, (orig_w, orig_h), interpolation=cv.INTER_NEAREST)

    return original_mask
```

F.4. IoU Computation *compute_IoU.py*

computer_IoU.py

```python
import argparse
from pathlib import Path
from PIL import Image
from tqdm import tqdm
import numpy as np
from data.preprocessing import color2class
import torch

def binarize_mask(mask_rgb):
    """
    Converts an RGB mask to binary (1: foreground, 0: background)
    Args:
        mask_rgb (np.ndarray): The RGB mask image.
    Returns:
        np.ndarray: Binary mask where 1 represents the foreground and 0 the background.
    """
    gray = np.array(Image.fromarray(mask_rgb).convert("L"))
    binary = (gray > 127).astype(np.uint8)
    return binary

def compute_dataset_iou(mode:str, gt_folder:str, pred_folder:str, output_file:str) -> None:
    '''
    Computes the Intersection over Union (IoU) for a dataset of segmentation masks.
    Args:
        mode (str): Mode of evaluation, either "3-class" or "binary".
        gt_folder (str): Path to the ground truth masks folder.
        pred_folder (str): Path to the predicted masks folder.
```

```python
        output_file (str): Path to save the IoU results.
    Returns:
        None. (results saved to <output_file>.txt)
    '''
    print(f'Calculating IoU of predicted images in {pred_folder} at mode {mode}...')

    # Define dataset paths
    gt_paths = sorted(Path(gt_folder).glob("*.*"))
    pred_paths = sorted(Path(pred_folder).glob("*.*"))
    device = 'cuda' if torch.cuda.is_available() else 'cpu'

    # Initialize accumulators
    if mode == "3-class":
        intersection_sum = {0: 0, 1: 0, 2: 0}
        union_sum = {0: 0, 1: 0, 2: 0}
    else:  # binary
        intersection_sum = {1: 0}  # foreground only
        union_sum = {1: 0}

    # Open file in write mode
    with open(output_file, "w") as file:
        for idx in tqdm(range(len(pred_paths)), desc='Test Image Loading', unit='image'):
            if mode == "3-class":
                gt_mask = Image.open(gt_paths[idx]).convert('RGB')
                gt_mask = np.array(gt_mask)
                gt_classes = color2class(gt_mask)
                gt_classes = torch.from_numpy(gt_classes).to(device).long()

                pred_mask = Image.open(pred_paths[idx]).convert('RGB')
                pred_mask = np.array(pred_mask)
                pred_classes = color2class(pred_mask)
                pred_classes = torch.from_numpy(pred_classes).to(device).long()

                # convert pred and gt images into one-hot embedding
                pred_one_hot = torch.nn.functional.one_hot(pred_classes, num_classes=3).permute(2,0,1).float()
                gt_one_hot = torch.nn.functional.one_hot(gt_classes, num_classes=3).permute(2, 0, 1).float()
                intersection = (pred_one_hot * gt_one_hot).sum(dim=(1, 2))
                union = pred_one_hot.sum(dim=(1, 2)) + gt_one_hot.sum(dim=(1, 2)) - intersection

                for cls in range(3):
                    intersection_sum[cls] += intersection[cls].item()
                    union_sum[cls] += union[cls].item()
            # For binary mode, we only care about the foreground (1) and background (0)
            elif mode == "binary":
                gt_mask = Image.open(gt_paths[idx]).convert('RGB')
                pred_mask = Image.open(pred_paths[idx]).convert('RGB')

                gt_binary = binarize_mask(np.array(gt_mask))
                pred_binary = binarize_mask(np.array(pred_mask))

                gt_tensor = torch.from_numpy(gt_binary).to(device)
                pred_tensor = torch.from_numpy(pred_binary).to(device)

                intersection = torch.logical_and(pred_tensor, gt_tensor).sum().item()
                union = torch.logical_or(pred_tensor, gt_tensor).sum().item()

                intersection_sum[1] += intersection
                union_sum[1] += union
        # initialize final iou dict
        final_iou = {}
        # Compute IoU per class
        if mode == "3-class":
            for cls in range(3):
                if union_sum[cls] > 0:
                    final_iou[cls] = intersection_sum[cls] / union_sum[cls]
                else:
```

```
                    final_iou[cls] = None  # Optional: mark as N/A if class never appears

            # Mean IoU: Only include classes that exist
            valid_ious = [iou for iou in final_iou.values() if iou is not None]
            mean_iou = sum(valid_ious) / len(valid_ious) if valid_ious else 0

            # Final results
            final_results = (f"\nFinal Dataset-level IoU Results:\n"
                            f"IoU of Background: {final_iou[0]:.4f}\n"
                            f"IoU of Cats: {final_iou[1]:.4f}\n"
                            f"IoU of Dogs: {final_iou[2]:.4f}\n"
                            f"Mean IoU: {mean_iou:.4f}\n")

        elif mode == "binary":
            if union_sum[1] > 0:
                iou = intersection_sum[1] / union_sum[1]
            else:
                iou = 0.0

            final_results = (f"\nFinal Dataset-level IoU Results (Binary):\n"
                             f"IoU of Foreground vs Background: {iou:.4f}\n")
        else:
            raise ValueError("Invalid mode. Choose '3-class' or 'binary'.")
        # Save results to file
        print(final_results.strip())
        file.write(final_results)

    print(f"\nDataset-level IoU results saved in: {output_file}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Compute Dataset-level IoU for Segmentation Masks")
    parser.add_argument('--mode', type=str, choices=["3-class", "binary"], default="3-class",
                        help='Evaluation mode: 3-class (default) or binary (SAM-style fg/bg)')
    parser.add_argument('--gt_folder', type=str, required=True, help='Path to the ground truth masks folder')
    parser.add_argument('--pred_folder', type=str, required=True, help='Path to the predicted masks folder')
    parser.add_argument('--output_file', type=str, default='iou_results.txt', help='Path to save the IoU results')

    args = parser.parse_args()
    compute_dataset_iou(args.mode, args.gt_folder, args.pred_folder, args.output_file)
```

G. UI

This appendix presents the implementation of the user interface (UI) used for prompt-based segmentation. The UI allows users to interactively provide input prompts—such as points or boxes—used to guide the segmentation model.

G.1. Source Code

```
import sys
import os
import numpy as np
import torch
from PIL import Image
import cv2
from pathlib import Path
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget, QVBoxLayout, QHBoxLayout,
                             QPushButton, QFileDialog, QLabel, QRadioButton, QButtonGroup,
                             QSlider, QGroupBox, QComboBox)
from PyQt5.QtGui import QPixmap, QPainter, QColor, QPen, QImage
from PyQt5.QtCore import Qt, QPoint, QRect

from models.prompt_segmentation import PromptSegmentation
from data.preprocessing import resize_with_padding
```

```python
from utils.restore_image_size import restore_original_mask

class SegmentationUI(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Interactive Segmentation Tool")
        self.setGeometry(100, 100, 1200, 800)

        # Initialize state variables
        self.original_image = None
        self.display_image = None
        self.original_size = None
        self.current_mask = None
        self.combined_mask = None
        self.prompts = []  # Store points/boxes for inference
        self.temp_box = None  # For box drawing
        self.mode = "point"  # Default mode is point placement
        self.current_class = 1  # Default class is foreground (1)
        self.target_size = 512  # Model input size
        self.threshold = 0.5  # Default threshold for segmentation

        # Load model
        self.model = self.load_model()

        # Set up UI
        self.setup_ui()

    def load_model(self):
        """Load the prompt-based segmentation model"""
        device = "cuda" if torch.cuda.is_available() else "cpu"
        print(f"Using device: {device}")

        # Load model
        model = PromptSegmentation()
        checkpoint_path = r"E:\Oxford-IIIT_Pet_Dataset_Segementation_test\params\best_prompt_checkpoint_epoch_1.pth"
        checkpoint = torch.load(checkpoint_path, map_location=device)
        model.load_state_dict(checkpoint)
        model.to(device)
        model.eval()

        return model

    def setup_ui(self):
        """Set up the user interface"""
        # Main widget and layout
        main_widget = QWidget()
        main_layout = QHBoxLayout()

        # Left side: image display
        self.canvas = QLabel()
        self.canvas.setMinimumSize(512, 512)
        self.canvas.setAlignment(Qt.AlignCenter)
        self.canvas.setStyleSheet("background-color: #f0f0f0; border: 1px solid #ccc;")
        self.canvas.mousePressEvent = self.canvas_click
        self.canvas.mouseMoveEvent = self.canvas_move
        self.canvas.mouseReleaseEvent = self.canvas_release

        # Right side: controls
        controls_layout = QVBoxLayout()

        # Image loading group
        image_group = QGroupBox("Image")
        image_layout = QVBoxLayout()
        self.load_image_btn = QPushButton("Load Image")
        self.load_image_btn.clicked.connect(self.load_image)
        image_layout.addWidget(self.load_image_btn)
        image_group.setLayout(image_layout)
```

```python
        controls_layout.addWidget(image_group)

        # Prompt tools group
        prompt_group = QGroupBox("Prompt Tools")
        prompt_layout = QVBoxLayout()

        # Prompt type selection
        prompt_type_layout = QHBoxLayout()
        self.point_radio = QRadioButton("Point")
        self.box_radio = QRadioButton("Box")
        self.point_radio.setChecked(True)
        self.point_radio.toggled.connect(lambda: self.set_mode("point"))
        self.box_radio.toggled.connect(lambda: self.set_mode("box"))
        prompt_type_layout.addWidget(self.point_radio)
        prompt_type_layout.addWidget(self.box_radio)
        prompt_layout.addLayout(prompt_type_layout)

        # # Class selection
        # class_layout = QHBoxLayout()
        # class_layout.addWidget(QLabel("Class:"))
        # self.class_combo = QComboBox()
        # self.class_combo.addItems(["Background (0)", "Foreground (1)"])
        # self.class_combo.setCurrentIndex(1)  # Default to foreground
        # self.class_combo.currentIndexChanged.connect(self.change_class)
        # class_layout.addWidget(self.class_combo)
        # prompt_layout.addLayout(class_layout)

        # Threshold slider
        threshold_layout = QHBoxLayout()
        threshold_layout.addWidget(QLabel("Threshold:"))
        self.threshold_slider = QSlider(Qt.Horizontal)
        self.threshold_slider.setMinimum(1)
        self.threshold_slider.setMaximum(99)
        self.threshold_slider.setValue(int(self.threshold * 100))
        self.threshold_slider.valueChanged.connect(self.change_threshold)
        threshold_layout.addWidget(self.threshold_slider)
        self.threshold_label = QLabel(f"{self.threshold:.2f}")
        threshold_layout.addWidget(self.threshold_label)
        prompt_layout.addLayout(threshold_layout)

        # Action buttons
        self.run_btn = QPushButton("Run Segmentation")
        self.run_btn.clicked.connect(self.run_segmentation)
        self.run_btn.setEnabled(False)
        prompt_layout.addWidget(self.run_btn)

        self.clear_prompts_btn = QPushButton("Clear Prompts")
        self.clear_prompts_btn.clicked.connect(self.clear_prompts)
        self.clear_prompts_btn.setEnabled(False)
        prompt_layout.addWidget(self.clear_prompts_btn)

        self.clear_all_btn = QPushButton("Clear All")
        self.clear_all_btn.clicked.connect(self.clear_all)
        self.clear_all_btn.setEnabled(False)
        prompt_layout.addWidget(self.clear_all_btn)

        prompt_group.setLayout(prompt_layout)
        controls_layout.addWidget(prompt_group)

        # Result group
        result_group = QGroupBox("Results")
        result_layout = QVBoxLayout()

        self.save_mask_btn = QPushButton("Save Mask")
        self.save_mask_btn.clicked.connect(self.save_mask)
        self.save_mask_btn.setEnabled(False)
        result_layout.addWidget(self.save_mask_btn)

```

```python
        self.toggle_view_btn = QPushButton("Toggle Overlay")
        self.toggle_view_btn.clicked.connect(self.toggle_overlay)
        self.toggle_view_btn.setEnabled(False)
        result_layout.addWidget(self.toggle_view_btn)

        result_group.setLayout(result_layout)
        controls_layout.addWidget(result_group)

        # Status indicator
        self.status_label = QLabel("Ready")
        controls_layout.addWidget(self.status_label)

        # Add stretcher to push everything up
        controls_layout.addStretch()

        # Arrange layouts
        main_layout.addWidget(self.canvas, 3)
        main_layout.addLayout(controls_layout, 1)

        main_widget.setLayout(main_layout)
        self.setCentralWidget(main_widget)

    def set_mode(self, mode):
        """Set the prompt mode (point or box)"""
        self.mode = mode
        self.temp_box = None
        self.update_canvas()

    def change_class(self, index):
        """Update the current class when combo box selection changes"""
        self.current_class = 0 if index == 0 else 1

    def change_threshold(self, value):
        """Update threshold value from slider"""
        self.threshold = value / 100.0
        self.threshold_label.setText(f"{self.threshold:.2f}")
        # Re-run segmentation if we have a current mask
        if self.current_mask is not None:
            self.run_segmentation()

    def load_image(self):
        """Load an image from file"""
        file_path, _ = QFileDialog.getOpenFileName(
            self, "Open Image", "", "Image Files (*.png *.jpg *.jpeg *.bmp)"
        )

        if file_path:
            # Load and store the original image
            self.original_image = np.array(Image.open(file_path).convert("RGB"))
            self.original_size = self.original_image.shape[:2]

            # Resize for display
            self.resize_and_display_image()

            # Enable UI elements
            self.run_btn.setEnabled(True)
            self.clear_all_btn.setEnabled(True)

            # Clear existing prompts and masks
            self.prompts = []
            self.current_mask = None
            self.combined_mask = None
            self.status_label.setText("Image loaded. Add prompts and run segmentation.")

    def resize_and_display_image(self):
        """Resize the original image and update display"""
        # Resize while maintaining aspect ratio
        img_resized = resize_with_padding(self.original_image, self.target_size, False)
```

```python
        self.display_image = img_resized.copy()

        # Update canvas
        self.update_canvas()

    def update_canvas(self):
        """Update the canvas with current image and overlays"""
        if self.display_image is None:
            return

        # Create a copy of the display image for drawing
        canvas_img = self.display_image.copy()

        # Draw mask overlay if available
        if self.combined_mask is not None:
            # Create a colored overlay (semi-transparent)
            overlay = np.zeros_like(canvas_img)
            overlay[self.combined_mask > 0] = [0, 255, 0]  # Green overlay

            # Blend with original image
            alpha = 0.5
            canvas_img = cv2.addWeighted(overlay, alpha, canvas_img, 1-alpha, 0)

        # Draw prompts
        for prompt in self.prompts:
            if prompt[0] == "point":
                x, y, cls = prompt[1]
                color = (0, 0, 255) if cls == 0 else (255, 0, 0)  # Red for foreground, blue for
                    background
                cv2.circle(canvas_img, (x, y), 5, color, -1)
            elif prompt[0] == "box":
                x1, y1, x2, y2, cls = prompt[1]
                color = (0, 0, 255) if cls == 0 else (255, 0, 0)
                cv2.rectangle(canvas_img, (x1, y1), (x2, y2), color, 2)

        # Draw temporary box if in progress
        if self.temp_box is not None:
            x1, y1, x2, y2 = self.temp_box
            cv2.rectangle(canvas_img, (x1, y1), (x2, y2), (255, 255, 0), 2)

        # Convert to QImage and display
        h, w, c = canvas_img.shape
        q_img = QImage(canvas_img.data, w, h, w*c, QImage.Format_RGB888)
        self.canvas.setPixmap(QPixmap.fromImage(q_img))

    def canvas_click(self, event):
        """Handle mouse click on canvas"""
        if self.display_image is None:
            return

        # Calculate image position within the canvas
        canvas_width = self.canvas.width()
        canvas_height = self.canvas.height()
        img_height, img_width = self.display_image.shape[:2]

        # Calculate offsets for centered image
        x_offset = max(0, (canvas_width - img_width) // 2)
        y_offset = max(0, (canvas_height - img_height) // 2)

        # Adjust coordinates
        x = event.x() - x_offset
        y = event.y() - y_offset

        # Check if click is within image bounds
        if 0 <= x < img_width and 0 <= y < img_height:
            if self.mode == "point":
                # Add point prompt immediately
                self.prompts.append(("point", (x, y, self.current_class)))
```

```
                self.clear_prompts_btn.setEnabled(True)
                self.update_canvas()
            elif self.mode == "box":
                # Start box drawing
                self.temp_box = [x, y, x, y]
        else:
            # Click outside the image area
            self.status_label.setText("Click within the image area")

    def canvas_move(self, event):
        """Handle mouse movement on canvas (for box drawing)"""
        if self.temp_box is None:
            return

        # Calculate image position within the canvas
        canvas_width = self.canvas.width()
        canvas_height = self.canvas.height()
        img_height, img_width = self.display_image.shape[:2]

        # Calculate offsets for centered image
        x_offset = max(0, (canvas_width - img_width) // 2)
        y_offset = max(0, (canvas_height - img_height) // 2)

        # Adjust coordinates
        x = event.x() - x_offset
        y = event.y() - y_offset

        # Constrain coordinates to image bounds
        x = max(0, min(x, img_width - 1))
        y = max(0, min(y, img_height - 1))

        self.temp_box[2], self.temp_box[3] = x, y
        self.update_canvas()

    def canvas_release(self, event):
        """Handle mouse release on canvas (for box drawing)"""
        if self.temp_box is None:
            return

        # Calculate image position within the canvas
        canvas_width = self.canvas.width()
        canvas_height = self.canvas.height()
        img_height, img_width = self.display_image.shape[:2]

        # Calculate offsets for centered image
        x_offset = max(0, (canvas_width - img_width) // 2)
        y_offset = max(0, (canvas_height - img_height) // 2)

        # Adjust coordinates
        x = event.x() - x_offset
        y = event.y() - y_offset

        # Constrain coordinates to image bounds
        x = max(0, min(x, img_width - 1))
        y = max(0, min(y, img_height - 1))

        x1, y1 = self.temp_box[0], self.temp_box[1]
        x2, y2 = x, y

        # Ensure correct ordering (x1,y1 is top-left, x2,y2 is bottom-right)
        if x1 > x2:
            x1, x2 = x2, x1
        if y1 > y2:
            y1, y2 = y2, y1

        # Only add if box has some area
        if x2 > x1 and y2 > y1:
            self.prompts.append(("box", (x1, y1, x2, y2, self.current_class)))
```

```python
            self.clear_prompts_btn.setEnabled(True)

        self.temp_box = None
        self.update_canvas()

    def generate_heatmap(self, prompt, img_shape):
        """Generate a heatmap for the given prompt"""
        h, w = img_shape
        heatmap = np.zeros((h, w), dtype=np.float32)

        if prompt[0] == "point":
            x, y, _ = prompt[1]
            # Create a Gaussian heatmap centered at the point
            sigma = 10.0  # Controls the spread of the Gaussian
            y_grid, x_grid = np.mgrid[0:h, 0:w]
            heatmap = np.exp(-((x_grid - x) ** 2 + (y_grid - y) ** 2) / (2 * sigma ** 2))
        elif prompt[0] == "box":
            x1, y1, x2, y2, _ = prompt[1]
            # Create a binary heatmap for the box region
            heatmap[y1:y2+1, x1:x2+1] = 1.0

        return heatmap

    def run_segmentation(self):
        """Run segmentation based on current prompts"""
        if not self.prompts or self.display_image is None:
            self.status_label.setText("Add at least one prompt before running segmentation")
            return

        self.status_label.setText("Running segmentation...")

        # Process each prompt
        self.combined_mask = None

        for prompt in self.prompts:
            # Generate heatmap for this prompt
            heatmap = self.generate_heatmap(prompt, (self.target_size, self.target_size))

            # Prepare inputs for the model - APPLY STANDARD TRANSFORM
            from data.preprocessing import standard_transform
            # Convert to PIL Image for transform
            pil_img = Image.fromarray(self.display_image)
            # Apply the same transform used during training/inference
            img_tensor = standard_transform(pil_img).unsqueeze(0)  # Add batch dimension

            heatmap_tensor = torch.from_numpy(heatmap).unsqueeze(0).unsqueeze(0).float()

            # Move tensors to device
            device = next(self.model.parameters()).device
            img_tensor = img_tensor.to(device)
            heatmap_tensor = heatmap_tensor.to(device)

            # Run inference
            with torch.no_grad():
                output = self.model(
                    image=img_tensor,
                    prompt_heatmap=heatmap_tensor,
                    point_class= None
                )

            # Apply threshold to get binary mask
            pred_mask = (torch.sigmoid(output) > self.threshold).squeeze().cpu().numpy().astype(
                np.uint8)

            # Union with previous masks
            if self.combined_mask is None:
                self.combined_mask = pred_mask
            else:
```

```
                self.combined_mask = np.logical_or(self.combined_mask, pred_mask).astype(np.uint8)

        # Enable save mask button
        self.save_mask_btn.setEnabled(True)
        self.toggle_view_btn.setEnabled(True)

        # Update canvas with mask overlay
        self.update_canvas()

        self.status_label.setText("Segmentation␣complete")

    def toggle_overlay(self):
        """Toggle between showing the mask overlay and original image"""
        if self.combined_mask is None:
            return

        # Toggle by temporarily removing and restoring the mask
        if hasattr(self, '_temp_mask'):
            self.combined_mask = self._temp_mask
            delattr(self, '_temp_mask')
        else:
            self._temp_mask = self.combined_mask
            self.combined_mask = None

        self.update_canvas()

    def save_mask(self):
        """Save the current mask to a file"""
        if self.combined_mask is None:
            return

        file_path, _ = QFileDialog.getSaveFileName(
            self, "Save␣Mask", "", "PNG␣Files␣(*.png)"
        )

        if file_path:
            # Resize mask back to original image size
            mask_resized = cv2.resize(
                self.combined_mask * 255,  # Scale to 0-255
                (self.original_size[1], self.original_size[0]),
                interpolation=cv2.INTER_NEAREST
            )

            # Save mask
            cv2.imwrite(file_path, mask_resized)
            self.status_label.setText(f"Mask␣saved␣to␣{file_path}")

    def clear_prompts(self):
        """Clear all prompts but keep the loaded image"""
        self.prompts = []
        self.combined_mask = None
        self.clear_prompts_btn.setEnabled(False)
        self.save_mask_btn.setEnabled(False)
        self.toggle_view_btn.setEnabled(False)
        self.update_canvas()
        self.status_label.setText("Prompts␣cleared")

    def clear_all(self):
        """Clear everything including the loaded image"""
        self.original_image = None
        self.display_image = None
        self.original_size = None
        self.prompts = []
        self.combined_mask = None
        self.temp_box = None

        # Reset UI
```

```
        self.canvas.clear()
        self.canvas.setPixmap(QPixmap())
        self.run_btn.setEnabled(False)
        self.clear_prompts_btn.setEnabled(False)
        self.clear_all_btn.setEnabled(False)
        self.save_mask_btn.setEnabled(False)
        self.toggle_view_btn.setEnabled(False)
        self.status_label.setText("Ready")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = SegmentationUI()
    window.show()
    sys.exit(app.exec_())
```