CV Coursework on Image Segmentation

Anonymous Authors *
University of Edinburgh

Abstract

This project investigates several deep learning approaches for semantic image segmentation using the Oxford-IIIT Pet Dataset. We implement and compare UNet-based models, a self-supervised Autoencoder, CLIP feature-driven segmentation, and an interactive prompt-based model. Our experiments evaluate the impact of data augmentation, loss functions, class weighting, and pretraining strategies. Results show that CLIP-based models significantly outperform others by utilizing strong visual representations learned from contrastive pretraining. Data augmentation improves resilience to input perturbations, while prompt-guided segmentation enables effective user interaction. Our findings highlight the value of combining pre-trained features with tailored model design for accurate and robust segmentation.

1. Introduction

Semantic image segmentation is a fundamental task in computer vision that involves classifying each pixel in an image into predefined categories. It plays a crucial role in applications ranging from medical imaging to autonomous navigation and human-computer interaction.

In this work, we address the challenge of segmenting cats and dogs using the Oxford-IIIT Pet Dataset. For this purpose, we explore multiple neural network architectures: a baseline UNet trained from scratch, an Autoencoder whose encoder is reused for downstream segmentation, a hybrid approach combining frozen CLIP visual embeddings with custom decoders, and a prompt-guided interactive segmentation approach. We aim to evaluate these methods in terms of accuracy and robustness to perturbations through multiple metrics.

2. Background

Semantic image segmentation assigns a class label (e.g., 'cat', 'dog', 'background') to every pixel in an image. Deep learning is a primary tool for this, particularly Convolutional Neural Networks (CNNs). CNNs use convolution operations with learnable filters to automatically detect spatial patterns like edges and textures, making them effective for image data.

The U-Net architecture, designed for segmentation, is frequently used. It employs an encoder to learn contextually rich embeddings and a decoder to produce the final segmentation map. U-Net's key innovation is skip connections, which pass feature maps directly from the encoder to corresponding decoder layers. This allows the reuse of fine grained spatial details, potentially lost during downsampling, combining them with high level semantic context for more accurate segmentation [4].

To further enhance architectures like U-Net, especially in settings with limited labeled data, transfer learning has become an increasingly valuable approach. In this paradigm, models pre-trained on large datasets are reused for related tasks, eliminating the need to train from scratch. Pre-trained networks can serve as feature extractors or be fine-tuned for specific applications, transferring knowledge from high-resource settings to low-resource ones. In computer vision, large vision-language models such as CLIP (Contrastive Language Image Pre-training) [3], provide powerful visual representations learned by aligning images with text, making them well-suited for transfer learning. CLIP's robust image encoder serves as an effective feature extractor, often boosting performance on downstream tasks like segmentation, especially with limited data.

Training deep networks involves iteratively adjusting weights to minimize a loss function that quantifies prediction error against the ground truth; common examples are Mean Squared Error (MSE) and Cross Entropy (CE). Optimization algorithms use gradients, indicating how loss changes with weights, to systematically update weights and reduce error, enabling the network to learn.

3. Methodology

3.1. Hardware and Software Setup

Experiments were conducted using both Kaggle Kernels (with NVIDIA Tesla P100 GPUs) and Google Colab environments (with NVIDIA A100 GPUs and 40GB of VRAM). Development was primarily done on macOS

*B269422, B270876

systems running Python 3.10.13. The implementation was based on `PyTorch`, with additional libraries including `torchvision` for model components, `NumPy` for numerical operations, `Pillow` for image handling, `Hugging Face Transformers` to access pre-trained CLIP models, and `Matplotlib` graph creation.

3.2. Dataset Preprocessing and Data augmentation

We utilized the Oxford-IIIT Pet Dataset [2], which provides cat and dog images paired with pixel level segmentation masks. A 80/20 training-validation split was performed, stratified by breed to ensure similar class representation in both sets. All image-label pairs in the training set were resized to 256x256 pixels, with zero padding to maintain the original aspect ratio. To resize segmentation masks, we consistently use nearest-neighbour interpolation, while bilinear interpolation is applied to the images. In contrast, the validation and test sets are left unaltered.

To increase volume of training data and improve model robustness, data augmentation techniques were applied using the `imgaug` library, expanding the training set to 6353 pairs. The specific augmentations and their parameters are detailed in Table 1 (with visual examples in Figures 9, 10 in the Appendix). These enhancements were strategically applied to balance the initial 2:1 dog-to-cat ratio in the training data towards a 1:1 ratio.

Finally, for the prompt-based segmentation model, specialized data was generated from the augmented training set. For each image-label pair, two image-heatmap-label triplets were created. The heatmaps simulate point prompts, with the two points per image deliberately placed on different semantic classes (e.g., one on background, one on the animal).

For the prompt-segmentation model, an additional preprocessing step is required: heatmap creation. For each image in the training data, two random points are selected such that they belong to different classes. Corresponding heatmaps are then generated for these points, resulting in two distinct label maps per image. The same procedure is applied to the validation and test sets.

3.3. Training and Evaluation Setup

Data Handling: We implemented two custom PyTorch Dataset classes. The first, for standard segmentation, reads image-label pairs, applying optional augmentations and scaling image pixel values from [0, 255] to [0, 1]. The second, for prompt-based segmentation, reads image-heatmap-label triplets, similarly scaling image and heatmap values. Data is batched using standard DataLoaders. A custom `collate_fn` handles variable image sizes in validation/test sets by batching samples into lists.

Training Loop: Each epoch iterates through the training DataLoader. The images and labels can optionally be resized with zero padding to meet the model's requirements. The loop executes a forward pass, computes the loss between the prediction and label, and performs a backward pass to calculate gradients. Gradient accumulation can be adopted, by summing the gradients over several steps before an optimizer update, effectively simulating larger batch sizes. All the provided models are trained for 100 epochs.

Evaluation Loop: After each training epoch, the model is evaluated on the validation set. Input images are resized (bilinear interpolation, with padding) to the model's expected size, storing original size metadata. The model generates predictions (logits), which are then reverted to their original dimensions using the stored metadata (removing padding, applying bilinear interpolation). These original-sized predictions are compared against the original ground-truth labels.

Metrics: A `MetricsHistory` class accumulates True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN) per class over the validation set. At the end of an epoch, it computes mean IoU, mean Dice score, and pixel accuracy based on these counts, assessing performance at the original image scale.
Losses: Several loss functions were employed:

- **MSE:** Used for reconstruction. It measures the average squared distance between predicted pixel value, $\hat{y}$, and true pixel value, y, across the image.

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i \in \text{pixels}} (y_i - \hat{y}_i)^2$$

- **CE:** Used for multi-class segmentation. For a single pixel i with true class c, the model yields logits, $\hat{y}$. Minimizing the CE loss corresponds to maximizing the predicted probability $P(c|i)$ obtained after applying softmax to these logits.

$$\mathcal{L}_{CE} = -\frac{1}{N} \sum_{i \in \text{pixels}} \log(\text{softmax}(\hat{y}))$$

- **CE+Soft Dice:** To balance pixel-wise accuracy and region overlap, especially under class imbalance, CE loss was combined with Soft Dice loss. Soft Dice loss computes the Dice coefficient directly on predicted probabilities ensuring differentiability. For a single class c: $\mathcal{L}_{Dice,c} = -\frac{2\sum_{i \in \text{pixels}} P(c|i) y_{ic} + \epsilon}{\sum_{i \in \text{pixels}} P(c|i) + \sum_{i \in \text{pixels}} y_{ic} + \epsilon}$ where y_{ic} is 1 if pixel i belongs to class c and 0 otherwise, and ϵ is a small smoothing constant. This is averaged over the selected classes and added to $\mathcal{L}_{CE}$.

Table 1. Reasoning and Implementation Details of augmentations.

Augmentation	Implementation Details
Rotation	Decrease dependence on animal position. Rotate images and their labels by a random angle between 45 and 315 degrees, applying zero padding. Some parts of the image, including parts of the object, may be cut. Then resize with zero padding.
Random Masking	Increase robustness by simulating partial occlusions and forcing the model to learn more context-aware representations. Increase ability to segment animals partially covered by objects. Randomly mask (drop) pixels (set to black), on both the image and label. Then resize with zero padding.
Cropping	Learn smaller animal features and parts and provide unpadded training images as input to the model. Extract a square from a random position in the image, where the side of the square is $\frac{2}{3}$ of the shorter side of the original image. Then resize with zero padding.
Resizing+Cropping	Learn smaller animal features and parts and provide unpadded training images as input to the model. Scaling the shorter side to the target size 512 while preserving aspect ratio, followed by center cropping. No additional resize is needed.
Random color jitter	Decrease reliance on color and increase robustness to noise. Randomly adjust each pixel's color by adding noise sampled from laplace distributions elementwise to the images. Then resize with zero padding.
Grayscale	Decrease reliance on color, learn from texture and shape. Convert images to grayscale.
Blurring	Increase robustness to image quality degradation. Learn global shape and structure. Blur an image by computing means over neighbourhoods.
Contrast Decrease	Decrease reliance on irrelevant fine details and focus on semantically meaningful structures, by making important features stand out under subdued visual conditions. Adjust contrast by scaling each pixel (v) to 127 + `alpha`*(v-127).
Merging two images	Concatenate two images side by side, then resize with zero padding. Approximately one-third of the augmented pairs consist of two cats, another third pair a cat with a dog, and the remaining third consist of two dogs. Increase ability to distinguish cats from dogs in the same picture.

$$\mathcal{L}_{CE+Dice} = \mathcal{L}_{CE} + \mathcal{L}_{Dice}$$

- **Class Weighting:** To counteract class imbalance, we experimented with weighted versions of these losses. Instead of a simple average, a weighted average can assign higher importance to less frequent classes. We tested two schemes:
  - **Full Weight:** Class weights calculated as inversely proportional to the number of pixels belonging to each class in the training set.
  - **Min Weight:** Same as "Full Weight", but the weight for the boundary class (which is ignored during evaluation) was manually set to the minimum weight assigned among the other classes.

Optimizer: We used the AdamW optimizer [1]. AdamW modifies the Adam optimizer by implementing decoupled weight decay. Instead of adding the L2 regularization term to the gradient itself before the adaptive moment updates (as in standard Adam), AdamW applies the weight decay directly to the weights after the gradient-based update step. This distinction often leads to improved generalization performance and more effective regularization compared to standard Adam with L2 regularization.

3.4. UNet-based end-to-end segmentation neural network

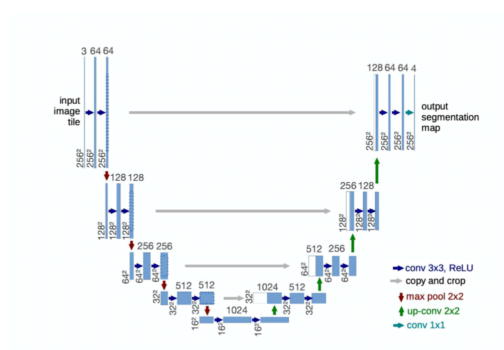


Figure 1. Our U-Net architecture. Each blue box represents a multi-channel feature map, with the number of channels indicated above the box and the spatial dimensions shown at its lower left corner. White boxes denote copied feature maps. Arrows indicate operations in the network. Figure adapted from "U-Net: Convolutional Networks for Biomedical Image Segmentation" [4], using our channels and sizes.

Our U-Net implementation (Figure 1) features a standard encoder-decoder structure with skip connections. The encoder captures context through five blocks. Each block applies two sequential 3x3 convolutions, each followed by Batch Normalization (BN) and ReLU, using padding to maintain spatial dimensions. The first convolution in each block doubles the number of input feature channels, while the second maintains this channel count. A 2x2 max pooling layer (stride 2) then halves the spatial resolution. This process culminates in a 16x16 bottleneck layer with 1024 channels, holding the most abstract image representation.

The decoder symmetrically reconstructs the segmentation map via four blocks. Each block begins with a 2x2 transposed convolution (stride 2), which doubles spatial resolution and halves the number of input channels. Crucially, features from the corresponding encoder level are concatenated via skip connections. Consistent padding and stride choices ensure spatial alignment for this concatenation, merging high-resolution encoder details with upsampled decoder context. Following concatenation, two sequential 3x3 convolutions (Padding+BN+ReLU) process the merged features; the first of these convolutions halves the channel count (relative to the concatenated feature map), while the second maintains it.

After the final decoder block restores the original 256x256 resolution, a concluding 1x1 convolution maps the 64 feature channels to 4 output channels representing the scores for background, cat, dog, and boundary classes.

For training, we explored several loss functions: standard Cross Entropy, Weighted Cross Entropy (addressing class imbalance), and combinations of CE with Soft Dice Loss (standard or weighted). We employed the AdamW optimizer with a learning rate of 0.001 and weight decay of 0.01. The target effective batch size was 64, achieved using gradient accumulation across multiple steps when necessary due to hardware constraints.

3.5. Autoencoder pre-training for segmentation

As an unsupervised representation learning task, we first train an autoencoder on raw, unlabeled images for image reconstruction. Once trained, the encoder is reused as a frozen feature extractor for the downstream semantic segmentation task with labeled masks. The goal is to learn meaningful latent features and evaluate their effectiveness in transfer learning for segmentation. Training and evaluation were performed using the dataset of augmented 256×256 RGB images.

Figure 2 illustrates the models used in the two stages of our complete pipeline:

Stage 1 – Reconstruction Autoencoder (AE):

The reconstruction network follows a symmetric encoder-decoder design similar to U-Net, encouraging the bottleneck to learn latent embeddings from which the entire image can be reconstructed. For reconstruction, we intentionally exclude skip connections. Allowing them would give the model direct access to low-level features, reducing the need for meaningful compression, and potentially leading to less effective latent representations.

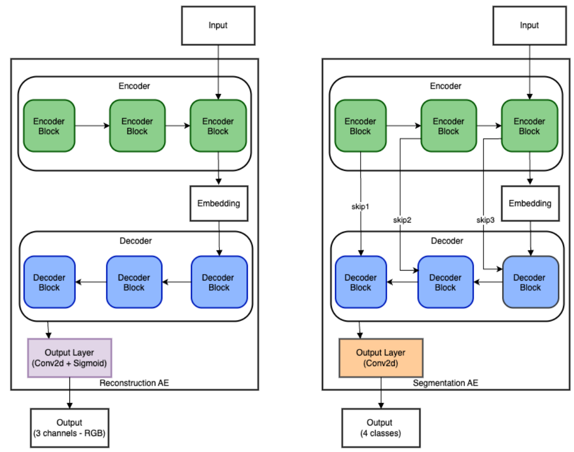


Figure 2. Architecture of Reconstruction AE and Segmentation AE models

Encoder:
The encoder consists of three sequential blocks. Each block contains two convolutional (Conv) layers (kernel size 3, padding 1), each followed by BN and ReLU. A `MaxPool2d` layer at the end of each block halves the spatial resolution.
Starting from an input of shape $(N, 3, 256, 256)$, Block 1 increases channels to 64 and reduces spatial size to 128×128. The following 2 encoder blocks double the channels and halve the image size again. This leads to a final embedding (bottleneck) of shape $(N, 256, 32, 32)$.

Decoder:
The decoder mirrors the encoder with three upsampling blocks. Each decoder block (`DecoderBlockNoSkips`) uses transposed convolution (ConvTranspose2d) with kernel size = 2 and stride = 2 to double height and width. In the first two blocks, channel depth is halved: 256→128, then 128→64; and in the third block, it is maintained at 64. Each upsampled feature map is refined using two convolutional layers (kernel size 3, padding 1) + BN + ReLU. Finally, a separate Conv2D layer maps from 64 channels back to RGB space (3 channels), followed by Sigmoid activation for normalized output: $(N, 64, 256, 256) \rightarrow (N, 3, 256, 256)$.

Reconstruction Training Details:
Reconstruction training was performed using batch size 64, Adam optimizer with learning rate of 0.001 and MSE loss (to promote pixel-wise accuracy). As an additional

experiment, reconstruction was also trained using skip connections. To enable this, the decoder architecture used was identical to that of the segmentation model, employing the same `DecoderBlockWithSkips` structure described in the following section.

Stage 2 – Segmentation Network Using Pre-trained Encoder:

The segmentation model reuses the encoder from the reconstruction AE but adds skip connections in a U-Net-style architecture. These connections merge high-resolution encoder features with upsampled decoder outputs, helping to recover spatial details lost during downsampling. This is essential for semantic segmentation, which relies on precise pixel-level localization, especially along object boundaries.

Encoder:
We reuse the encoder from the reconstruction model by wrapping it in a reusable module (`SegmentationEncoder`) that loads the pre-trained weights from the reconstruction encoder and freezes them during segmentation training.

Decoder with Skip Connections:
Each decoder block (`DecoderBlockWithSkips`) performs upsampling via transposed convolution (kernel size=2, stride=2), doubling spatial dimensions and reducing channels, followed by concatenation with the corresponding skip connection from the encoder and refinement through two stacked Conv–BN–ReLU layers. For example, in the first block: $(N, 256, 32, 32) \xrightarrow{upsample} (N, 128, 64, 64)$; $Concat((N, 128, 64, 64), (N, 256, 64, 64)_{\mathtt{skip_3}}) \rightarrow$
$\rightarrow \texttt{Conv-BN-ReLU} \times 2 \rightarrow (N, 128, 64, 64)$

After 3 decoder blocks, a $1{\times}1$ convolution maps the 64 channels to the 4 classes: background (#0), cat (#1), dog (#2), and boundary (#3). Output shape becomes $(N, 4, 256, 256)$, representing class logits per pixel.

Segmentation Training Details:
Segmentation training used a batch size of 64, the AdamW optimizer, with learning rate of 0.001 and weight decay of 0.01, and weighted CE+Soft Dice loss. Both the *Full Weight* and *Min Weight* class weighting schemes, as defined previously, were compared.

3.6. CLIP features for segmentation

The model's architecture is depicted in Figure 3.We utilized features from the pre-trained openai/clip-vit-base-patch16 CLIP model, accessed via Hugging Face's transformers. This Vision Transformer (ViT) was chosen for its strong representations and accessible hidden states. The model processes the input image into 16x16 patches; this patch size, combined with the ViT architecture, results in

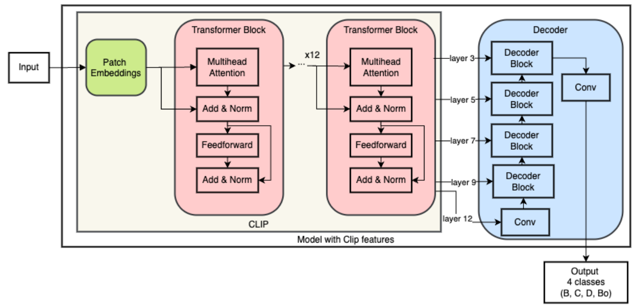


Figure 3. Architecture of Model using CLIP features.

final hidden state outputs (excluding the CLS token) that correspond to 49 patch embeddings (shape 49, 768). These can be reshaped into a 7x7 spatial grid (768, 7, 7) serving as the bottleneck feature map. Crucially, starting from this 7x7 bottleneck dimension allows a UNet-style decoder with four 2x upsampling stages, as described below, to reconstruct a feature map matching the CLIP encoder's input resolution (224x224).

Consistent with CLIP requirements, input images were resized to 224x224. The CLIP image encoder's weights remained frozen, serving purely as a feature extractor. Its final hidden state output (excluding the CLS token) was processed as described above into the (768, 7, 7) spatial grid for the decoder.

This decoder was inspired by the UNet's expansive path. An initial 1x1 convolution adapted the (768, 7, 7) bottleneck features to 1024 channels. The primary decoder configuration included four up-sampling blocks. Each block performed a 2x2 transposed convolution, followed by concatenation with features extracted from intermediate layers [3, 5, 7, 9] of the frozen CLIP encoder. These intermediate CLIP features (CLS token removed, reshaped, and resized) served as skip connections, aiming to incorporate hierarchical information. A standard double convolution block (3x3 Conv + BN + ReLU, repeated twice) processed the concatenated features. An alternative decoder variant omitted these skip connections, adjusting internal filter counts accordingly. Both decoder versions concluded with a 1x1 convolution projecting features to the 4 required output classes (background, cat, dog, boundary). The overall architecture is depicted in Figure 3.

We employed the training strategy proven effective for the UNet model. Specifically, the weighted version of the combined Cross-Entropy and Dice loss (CE+Dice) was used. Optimization was performed using AdamW with a learning rate of 0.001 and weight decay of 0.01. The model was trained for 100 epochs with a target batch size of 64, employing gradient accumulation as needed due to hardware constraints.

3.7. Prompt-based segmentation

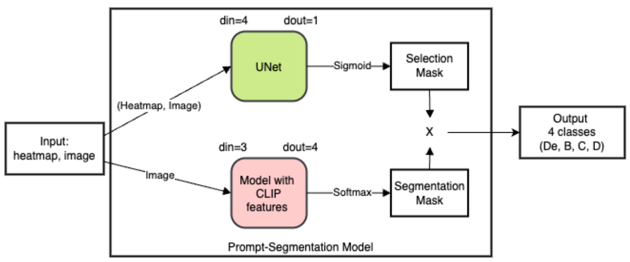


Figure 4. Architecture of Prompt-based Model. Class De - 'deactivated' pixels, B - background/boundary, C - cat, D - dog.

For prompt-based segmentation, we generated triplets of (image, heatmap, label) from the augmented dataset, as stated in 3.2. The heatmap represents the prompt, constructed as a discretized Gaussian (σ=3) centered on a selected pixel. Augmented images created by merging distinct originals were excluded, as generating accurate ground truth masks for prompts in these complex merged scenarios would require an already perfected segmenter.

The task objective is to segment only the class indicated by the prompt point, treating all other pixels as 'deactivated'. To achieve this, target labels were modified: original background and boundary classes were merged into a single 'background/boundary' class, and a new 'deactivated' class was introduced. Prompts were placed on either the merged background/boundary, cat, or dog regions. While the effective ground truth mask for a given sample highlights only the single selected class against the 'deactivated' background, the model was trained to output scores for all potential semantic classes alongside the 'deactivated' state to retain context. The final label values used for training were: 0 ('deactivated'), 1 ('background/boundary'), 2 ('cat'), and 3 ('dog'). This four-value representation was chosen over a simpler binary output to leverage the detailed cat/dog distinctions for potentially improving shape identification and segmentation accuracy.

We adapted the best-performing non-interactive model, the CLIP feature-based architecture, for this task. The prompt model takes both an image and its corresponding heatmap as input. The pre-existing CLIP-based segmentation component processes the image, outputting initial class segmentation probabilities via Softmax. In parallel, the image is concatenated with the heatmap (4 input channels: R,G,B,Heatmap) and fed into a new 'Selection Network'. This network uses the same UNet decoder architecture but takes 4 input channels and produces a single output channel representing pixel selection probability (via Sigmoid). The architecture of the model is represented in Figure 4.

The final output is computed by element-wise multiplying the class probabilities from the segmentation component with the selection probability from the Selection Network (broadcast across class channels). This yields a tensor representing joint probabilities $P(\text{class}, \text{selected}|\text{pixel}, \text{prompt})$ for the active classes (background/boundary, cat, dog). The probability for the 'deactivated' class is implicitly one minus the sum of the probability vector. For loss computation, we construct a 4-value probability vector per pixel: $(P(\text{deactivated}), P(\text{bg/boundary}, \text{selected}), P(\text{cat}, \text{selected}), P(\text{dog}, \text{selected}))$.

The network was trained using a combined Negative Log Likelihood (NLL) and Dice Loss applied to this final 4-value probability vector. NLL loss (similar to Cross Entropy, minimizing the negative log probability of the correct class configuration) is suitable for the probabilistic output, while Dice loss optimizes segmentation overlap modulated by the selection. We tested two strategies for the incorporated CLIP component: keeping its weights frozen versus allowing fine-tuning to adapt specifically to the prompt-guided task. Recognizing that correct selection is as critical as classification, equal class weights were applied across background/boundary, cat, and dog classes during loss calculation. Optimization used AdamW (learning rate 0.001, weight decay 0.01) for 100 epochs with an effective batch size of 64 (using gradient accumulation).

The user interface shown in Figure 5 is a locally hosted interactive web application designed to showcase the prompt-based model, allowing users to upload an image, select a point prompt, and view the resulting segmentation mask.

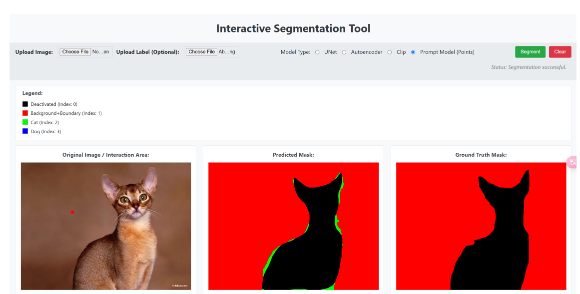


Figure 5. Reconstruction AE and Segmentation AE

4. Evaluation and Discussion of Results

We compared UNet input resolutions of 256x256 and 512x512 to balance accuracy and efficiency. While 512x512 showed marginal gains in Dice/IoU (Figure 8a), it significantly increased computational cost (approx. 4x slower, higher memory). This minor gain is likely because the segmentation task, focused on large coherent regions (animals, background), doesn't heavily rely on fine-grained details. The spatial continuity of classes, combined with interpolation during evaluation, means 256x256 provides sufficient detail. Due to this unfavorable cost-benefit trade-off, we standardized on the efficient 256x256 resolution for subsequent work.

We compared standard Cross Entropy (CE) loss with a combined CE + Soft Dice loss. As shown in Figure 8b, the combined loss yielded slightly superior and more stable metrics (Dice, IoU, Accuracy). We attribute this to CE optimizing pixel-level accuracy while Dice directly encourages better spatial overlap and region consistency, key for segmentation. Dice's inherent robustness to class imbalance might also contribute to stability. Offering these benefits at no added computational cost, we adopted the combined CE + Soft Dice loss for subsequent work.

To address class imbalance from dominant background pixels, we applied class weighting (Subsection 3.3) to the CE+Soft Dice loss. We tested 'Full Weight' (inversely proportional to pixel counts) and 'Min Weight' (assigning the minimum weight to the boundary class). Figure 8c shows 'Full Weight' performed noticeably better, indicating that assigning higher weights to boundary pixels during training led to improved segmentation performance.

This suggests that accurately learning boundary pixels, though ignored in final evaluation, is crucial. Boundaries define the transition between foreground objects and the background. By precisely learning these boundary regions, the model can better delineate the enclosed objects (cats/dogs) and distinguish them from their surroundings, leading to sharper segmentations and higher IoU/Dice scores. De-emphasizing boundaries ('Min Weight') likely permits less precise edge learning, hindering overall segmentation.

Finally, to evaluate the impact of data augmentation on the UNet model under optimal settings (256×256 input, CE+Soft Dice loss, 'Full Weight' class weights), we compared its performance with and without augmentation. As shown in Figure 8d, improvements in standard validation metrics (Dice, IoU, Accuracy) were marginal. This unexpected result may be due to the validation set containing clean images, unlike the perturbed ones seen during training. We hypothesize that augmentation primarily enhances model robustness to distortions rather than significantly improving performance on unaltered validation data, a topic explored further in the following subsection.

We evaluated using an Autoencoder (AE) for self-supervised pre-training of the segmentation encoder. Two AE designs were compared during pre-training for image reconstruction: one standard AE ('AE without Skips') and one with internal skip connections linking encoder and decoder layers ('AE with Skips').

While the 'AE with Skips' achieved significantly better reconstruction (lower validation loss, Figure 8e), its utility for the downstream segmentation task proved inferior. After freezing the pre-trained encoders and training segmentation decoders, the encoder pre-trained without skip connections yielded markedly better segmentation performance (Dice, IoU, Accuracy) than the one pre-trained with skips (Figure 8f).

This inverse relationship suggests the skip connections in the AE, although improving reconstruction fidelity, likely acted as information shortcuts. By allowing the decoder to directly access low-level features from the encoder, these connections reduced the burden on the encoder to learn a compressed, semantically rich representation in the bottleneck. In contrast, training without skip connections forced the encoder to rely solely on its bottleneck for reconstruction, resulting in more robust and transferable features that were better suited for segmentation, despite a modest decrease in reconstruction quality.

For CLIP-based models leveraging a pre-trained encoder, we first examined the impact of adding skip connections between frozen CLIP layers and the custom UNet-style decoder. As shown in Figure 7, the variant with skip connections (using features from CLIP layers [3, 5, 7, 9]) consistently outperformed the version without them across all metrics (Dice, IoU, Accuracy).

This highlights the value of multi-level feature fusion in segmentation, as combining high-level semantic features with intermediate spatial details, similar to standard UNet architecture, proves beneficial. Beyond this architectural similarity, they also help overcome a key weakness of vision transformers. Because CLIP encoders work with fixed-size image patches, discarding precise pixel-wise alignment early in the encoding process, and rely on global attention, they tend to lose fine spatial detail. By bringing in features from earlier layers, where more local structure is preserved, the decoder can recover important details needed for accurate pixel-level predictions.

In the same experiment (Figure 7), we also re-evaluated the impact of class weighting schemes ('Full Weight' vs. 'Min Weight'). Unlike with UNet, where 'Full Weight' clearly performed better, the CLIP-based model showed negligible performance differences between the two (e.g., solid vs. dashed blue/red lines), suggesting that CLIP's rich pre-trained features may already distinguish classes effectively, reducing sensitivity to loss weighting.

Finally, we assessed the effect of data augmentation on the best-performing CLIP-based model configuration (with skip connections and 'Full Weight'). As shown in Figure 8g, augmentation yielded only minor improvements in standard validation metrics, consistent with UNet results, supporting the hypothesis that its primary benefit lies in improving robustness to perturbations rather than boosting performance on clean validation data.

Since the prompt-based model builds on the CLIP-based segmentation architecture, we evaluated how its training

Table 2. Comparison of Model Performance Metrics on Test Set

Model	Accuracy	Dice	IoU
UNET (Aug)	0.9462	0.8661	0.7687
UNET (No Aug)	0.9444	0.8632	0.7643
CLIP (Aug)	**0.9732**	**0.9442**	**0.8946**
CLIP (No Aug)	0.9723	0.9414	0.8897
Autoencoder	0.8712	0.6804	0.5382
Prompt	0.8321	0.7088	0.5497

state affects final performance. Specifically, we compared keeping the CLIP component frozen (using weights from its initial non-interactive training) versus fine-tuning it alongside the newly introduced Selection Network.

As shown in Figure 8h, fine-tuning led to consistently better performance across all metrics. While the frozen CLIP component was already effective, allowing it to adapt within the context of prompt-based training improved both segmentation accuracy and coordination with the Selection Network. This highlights that joint optimization enhances synergy between components, making fine-tuning essential for optimal interactive segmentation. Fine-tuning adapts CLIP's global semantic representations toward the spatially localized task of prompt-guided segmentation, where precise alignment between image features and user-specified points is critical. Therefore, the best performing Prompt-based model utilized fine-tuning for the integrated CLIP component.

After identifying optimal configurations, we now summarize their final performance on the held out test set. Table 2 reports key metrics, Accuracy, Dice score, and IoU, for the best-performing variant of each model: UNet (with/without augmentation), CLIP-based model (with/without augmentation), Autoencoder and Prompt-based model.

The performance comparison presented in Table 2 clearly indicates the superiority of the CLIP-based model, which significantly outperformed all other approaches across accuracy, Dice, and IoU metrics, achieving a notably high mIoU of approximately 0.89. This strong performance is attributed to the powerful, generalizable image representations learned by the CLIP encoder during its extensive pre-training on a massive and diverse dataset, coupled with its Vision Transformer architecture's ability to effectively capture global context.

In comparison, the standard UNet model performed respectably, validating its architecture with skip connections for this task and surpassing the Autoencoder approach. However, lacking large-scale pre-training, it could not reach the segmentation quality of the CLIP-based model.

Consistent with earlier experiments, data augmentation provided only marginal benefits for both UNet and CLIP models when evaluated on this clean test set.

The Autoencoder pre-trained model yielded significantly weaker results, likely because its features were learned solely through reconstruction on the comparatively small Oxford-IIIT dataset using a less complex architecture, resulting in less robust representations than CLIP.

Finally, the Prompt-based model's metrics appear lower but must be interpreted in the context of its distinct task, which combines selection based on a heatmap prompt with segmentation. Evaluated on a derived dataset focusing only on the prompted class, its scores reflect performance on this more complex, interactive objective and are not directly comparable to the full semantic segmentation models, yet demonstrate its effectiveness in that specific setting.

As evidenced above, the transfer learning approach yielded significantly different results when using the pre-trained CLIP encoder versus the reconstruction autoencoder (AE) for image segmentation, achieving the highest and lowest IoU scores, respectively. A major factor is the scale and diversity of CLIP's training dataset, compared to the smaller dataset used for training the AE. Additionally, the pretraining tasks differ in relevance: while the AE is trained for image reconstruction, capturing low-level structural features, CLIP is trained for image classification, a task more closely related to classification. This enables CLIP to learn high-level semantic features that are more transferable to segmentation tasks. For example, by learning to classify an image as a "cat," CLIP develops features useful not only for identification, helping distinguish between the cat and dog classes) but also for delineation in segmentation. This greater task alignment contributes to why the transfer learning was more effective for the CLIP-based model than the AE.

4.1. Robustness exploration

To evaluate generalization and the impact of data augmentation on resilience, we subjected the CLIP (Aug/No Aug) and UNet (Aug/No Aug) models to eight perturbation types: Gaussian pixel noise, Gaussian blurring, contrast increase/decrease, brightness increase/decrease, occlusion, and salt and pepper noise, each applied at ten increasing severity levels. Performance (Mean Dice) against perturbation level is plotted in Figure 6.

The results consistently demonstrate the value of augmentation: across most perturbations, the augmented models (solid lines) maintained higher performance or degraded less rapidly than their non-augmented counterparts (dashed lines), strongly supporting our hypothesis that augmentation enhances robustness.

Model stability varied by perturbation. Gaussian pixel noise, contrast increase, brightness changes, and occlusion generally caused only minor or gradual declines, with augmented models showing a slight advantage. However, per-

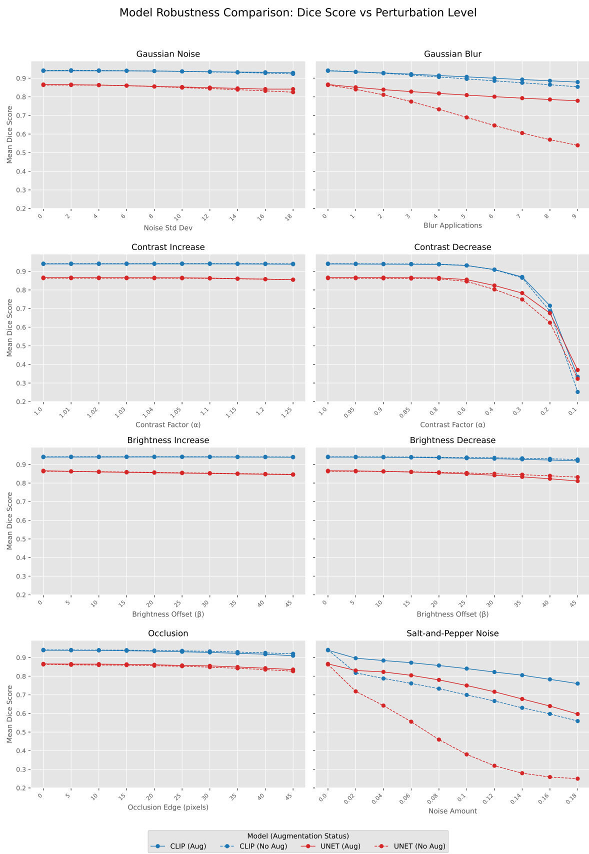


Figure 6. Comparison of CLIP-based model and UNet, with and without augmentation, under increasing perturbation levels.

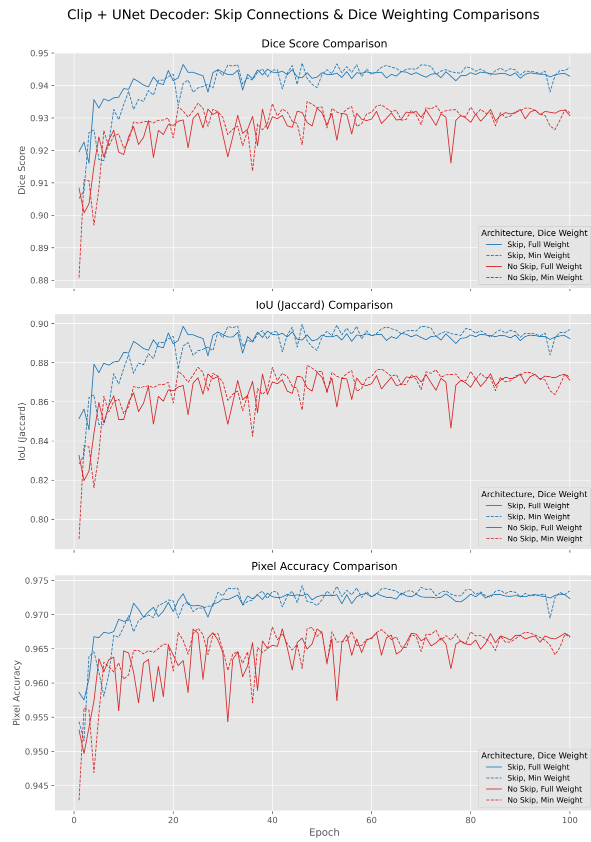


Figure 7. Comparison of the CLIP-based model performance with vs. without skip connections, using CE+Soft Dice loss with simple class weights vs. reduced boundary weight.

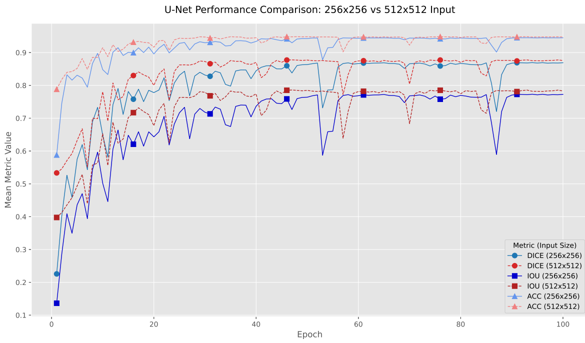
formance dropped significantly under Gaussian blur and salt and pepper noise, particularly for non-augmented models (especially UNet). Notably, under these specific strong perturbations, the augmented UNet occasionally surpassed the non-augmented CLIP model, highlighting how targeted augmentation can provide substantial resilience against certain degradations, sometimes outweighing general pre-training benefits.

Universally, contrast decrease posed the greatest challenge. All models experienced sharp performance plunges as contrast approached zero. This underscores the fundamental reliance of segmentation algorithms on detecting discernible edges, which are defined by differences in pixel intensity (i.e., contrast). Severely reducing contrast effectively diminishes or erases this critical edge information, crippling the models' ability to accurately delineate object boundaries, regardless of the training strategy employed.

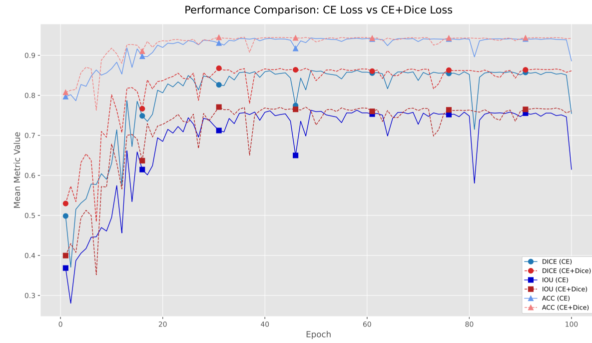
In summary, these experiments quantitatively confirm that data augmentation significantly boosts model robustness, especially against spatial noise like blur and salt and pepper. While CLIP's pre-training provides superior performance on clean data, augmentation adds a crucial layer of resilience. Severe contrast reduction remains a fundamental challenge for these segmentation approaches.

5. Conclusion

This work explored deep learning approaches for image segmentation, showing that models using pre-trained CLIP encoders outperform others by transferring semantically rich representations learned via large-scale vision-language classification. Skip connections in transformer-based decoders recovered spatial detail, and retaining boundary contributions via class weighting enhances boundary precision. While the UNet remained competitive, its reliance on lower-level features made it less adaptable than CLIP-based models. Fine-tuning pre-trained components significantly improved prompt-guided segmentation, where spatial reasoning is vital, and augmentation significantly enhanced robustness to noise and distributional shifts. Overall, our results show the value of combining pre-trained visual features with well-designed architectures and training for robust and accurate segmentation.

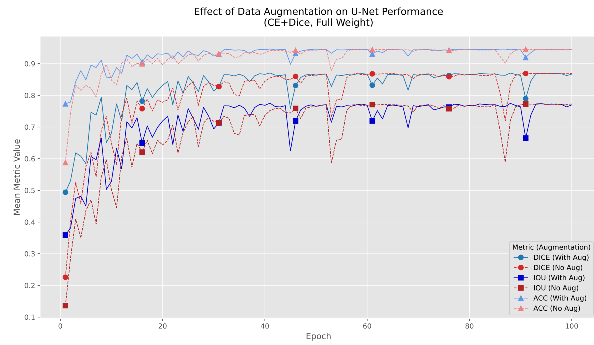


(a) Comparison of UNet performance metrics when trained on 256×256 vs. 512×512 input images.

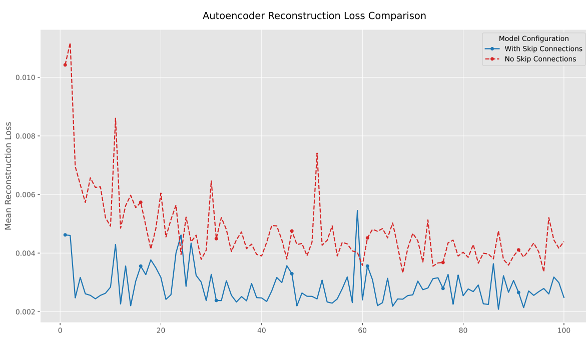


(b) Comparison of UNet performance using CE loss vs. CE + Soft Dice loss.

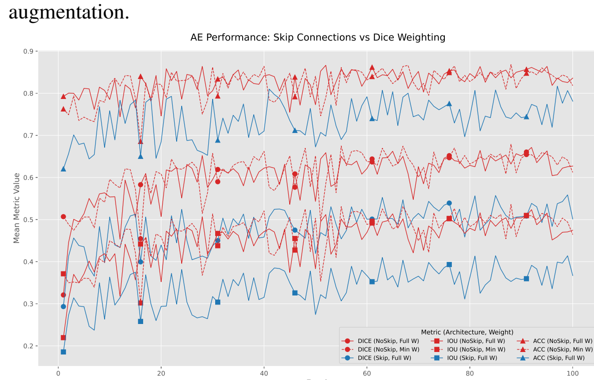
(c) Comparison of UNet performance with CE+Soft Dice loss using simple class weights vs. reduced boundary weight.



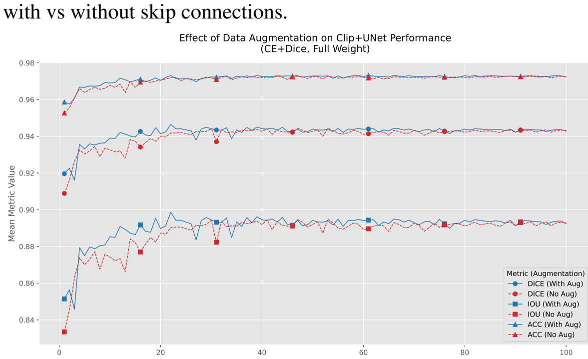
(d) Comparison between the UNet performance with vs. without data augmentation.



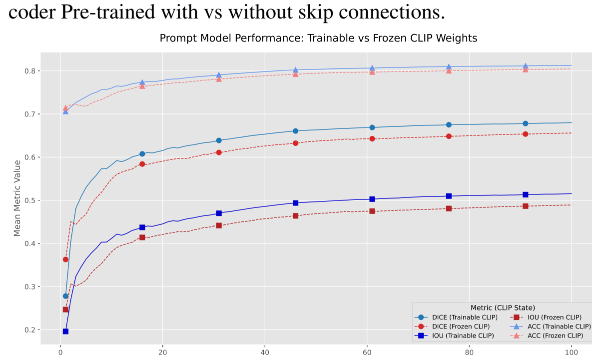
(e) Comparison of AE Reconstruction performance (validation loss) with vs without skip connections.



(f) Comparison of AE Segmentation performance when using an Encoder Pre-trained with vs without skip connections.



(g) CLIP-based model performance with vs. without data augmentation.



(h) Prompt-based model performance with frozen vs. fine-tuned CLIP weights.

Figure 8. Performance Experiments for UNet (rows 1-2), AE (3rd row), CLIP-based (bottom left) and Prompt-based (bottom right) models.

6. Appendix

Division of Responsibilities

The work was divided equally between both group members, with each taking responsibility for specific components and contributing jointly to others. Both students collaborated on writing the report and worked together on data augmentation techniques and the development of the UNet model. Specifically, Student B269422 focused on implementing and refining the encoder part of the UNet architecture, while Student B270876 concentrated on optimizing the decoder section.

Student B269422 was solely responsible for designing and implementing the Autoencoder module, as well as developing the user interface (UI) for interacting with the models. On the other hand, Student B270876 took charge of building and training two alternative models: one utilizing CLIP features and another based on prompt-based learning strategies.

Both students independently ran evaluation experiments to assess model performance. They also explored robustness under varying conditions in parallel by dividing up datasets and scenarios to test against. The analysis of these evaluations was later compiled collaboratively into the final report.

Augmentation Images

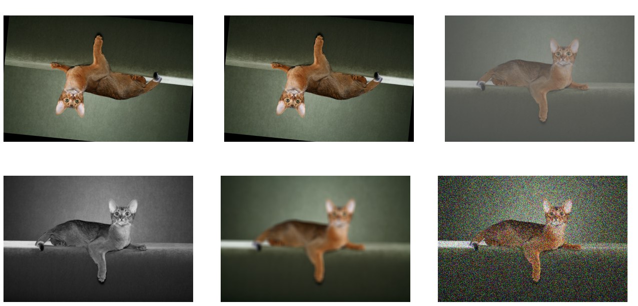


Figure 9. Examples of image augmentations used in training, including rotation, cropping, contrast decrease (top row), and grayscale, blur, and noise/color jitter (bottom row). These are not yet resized to 256×256.

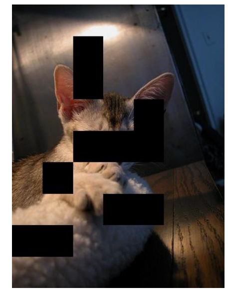


Figure 10. Example of image augmented with random masking, not yet resized to 256×256.

References

[1] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. 3

[2] Omkar M. Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2012. 2

[3] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021. 1

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015. 1, 3

Code

UNet (unet/unet.py)

```python
import torch
from torch import nn

class DoubleConvReLU(nn.Module):
    """
    Applies two convolutional layers with ReLU activation and batch normalization.
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after applying the double convolution with ReLU.
    """
    def __init__(self, din, dout):
        super().__init__()
        self.doubleConvReLU = nn.Sequential(
            nn.Conv2d(din, dout, kernel_size=3, padding=1),
            nn.BatchNorm2d(dout),
            nn.ReLU(),
            nn.Conv2d(dout, dout, kernel_size=3, padding=1),
            nn.BatchNorm2d(dout),
            nn.ReLU(),
        )

    def forward(self, x):
        return self.doubleConvReLU(x)


class Down(nn.Module):
    """
    Downscales the input using max pooling and applies double convolution.
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after downscaling and double convolution.
    """
    def __init__(self, din, dout):
        super().__init__()
        self.maxpool_doubleConv = nn.Sequential(
            nn.MaxPool2d(kernel_size=2, stride=2),
            DoubleConvReLU(din, dout)
        )

    def forward(self, x):
        return self.maxpool_doubleConv(x)

class Up(nn.Module):
    """
    Upscales the input using transposed convolution, concatenates with a skip connection, and applies double convolut
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after upscaling, concatenation, and double convolution.
    """
    def __init__(self, din, dout):
        super().__init__()

        self.upsample = nn.ConvTranspose2d(din, dout, kernel_size=2, stride=2)
        self.doubleConv = DoubleConvReLU(din, dout)

    def forward(self, x1, x2):
        x = torch.cat([x1, self.upsample(x2)], dim=1)
        return self.doubleConv(x)
```

```python
class unet(nn.Module):
    """
    Implements the U-Net architecture.
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after passing through the U-Net.
    """
    def __init__(self, din, dout):
        super().__init__()
        self.scale = 1

        self.down1 = DoubleConvReLU(din, self.scale * 64)
        self.down2 = Down(self.scale * 64, self.scale * 128)
        self.down3 = Down(self.scale * 128, self.scale * 256)
        self.down4 = Down(self.scale * 256, self.scale * 512)
        self.down5 = Down(self.scale * 512, self.scale * 1024)

        self.up1 = Up(self.scale * 1024, self.scale * 512)
        self.up2 = Up(self.scale * 512, self.scale * 256)
        self.up3 = Up(self.scale * 256, self.scale * 128)
        self.up4 = Up(self.scale * 128, self.scale * 64)

        self.output = nn.Conv2d(self.scale * 64, dout, kernel_size=1)

    def forward(self, x):
        x1 = self.down1(x)
        x2 = self.down2(x1)
        x3 = self.down3(x2)
        x4 = self.down4(x3)
        x5 = self.down5(x4)

        x = self.up1(x4, x5)
        x = self.up2(x3, x)
        x = self.up3(x2, x)
        x = self.up4(x1, x)

        return self.output(x)
```

Autoencoder (autoencoder/autoencoder.py)

```python
import torch.nn as nn
import numpy as np
import torch


class EncoderBlock(nn.Module):
    """
    A block for the encoder, consisting of two convolutional layers, batch normalization, ReLU activations, and max po
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
    Returns:
        tuple: A tuple containing the pooled output and the skip connection features.
    """
    def __init__(self, din, dout):
        super().__init__()
        self.conv1 = nn.Conv2d(din, dout, kernel_size=3, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(dout)
        self.relu1 = nn.ReLU()
        self.conv2 = nn.Conv2d(dout, dout, kernel_size=3, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(dout)
        self.relu2 = nn.ReLU(inplace=True)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

```

```python
    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        skip_connection = self.relu2(x) # Features before pooling
        pooled_output = self.pool(skip_connection)
        return pooled_output, skip_connection

class Encoder(nn.Module):
    """
    The encoder module composed of multiple encoder blocks.
    Args:
        din (int): Number of input channels.
        base_channels (int): Base number of channels to be used for the encoder blocks.
    Returns:
        tuple: A tuple containing the bottleneck features and skip connections from each encoder block.
    """
    def __init__(self, din, base_channels):
        super().__init__()
        self.encoderPart1 = EncoderBlock(din, base_channels)
        self.encoderPart2 = EncoderBlock(base_channels, base_channels*2)
        self.encoderPart3 = EncoderBlock(base_channels*2, base_channels*4)

    def forward(self, x):
        x1_pooled, skip1 = self.encoderPart1(x)
        x2_pooled, skip2 = self.encoderPart2(x1_pooled)
        bottleneck, skip3 = self.encoderPart3(x2_pooled)
        return bottleneck, skip3, skip2, skip1


class DecoderBlockWithSkips(nn.Module):
    """
    A decoder block that uses skip connections.
    Args:
        din_up (int): Number of input channels from the previous upsampled layer.
        din_skip (int): Number of input channels from the skip connection.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after upsampling, concatenation, and convolution.
    """
    def __init__(self, din_up, din_skip, dout):
        super().__init__()
        self.up = nn.ConvTranspose2d(din_up, dout, kernel_size=2, stride=2)
        conv_input_channels = dout + din_skip # Input to convs includes skip features
        self.convs = nn.Sequential(
            nn.Conv2d(conv_input_channels, dout, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(dout),
            nn.ReLU(inplace=True),
            nn.Conv2d(dout, dout, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(dout),
            nn.ReLU(inplace=True)
        )

    def forward(self, x, skip_features):
        x_upsampled = self.up(x)
        if skip_features.shape[2:] != x_upsampled.shape[2:]:
            diffY = skip_features.size()[2] - x_upsampled.size()[2]
            diffX = skip_features.size()[3] - x_upsampled.size()[3]
            if diffY < 0 or diffX < 0:
                raise ValueError("Upsampled larger than skip")
            skip_features = skip_features[:, :, diffY // 2 : diffY // 2 + x_upsampled.size()[2],
                                          diffX // 2 : diffX // 2 + x_upsampled.size()[3]]


        x_concat = torch.cat([x_upsampled, skip_features], dim=1)
```

```python
        output = self.convs(x_concat)
        return output


class DecoderWithSkips(nn.Module):
    """
    The decoder module with skip connections.
    Args:
        base_channels (int): Base number of channels.
    Returns:
        torch.Tensor: Output tensor after decoding.
    """
    def __init__(self, base_channels):
        super().__init__()
        self.decoderBlock1 = DecoderBlockWithSkips(din_up=base_channels*4, din_skip=base_channels*4, dout=base_channel
        self.decoderBlock2 = DecoderBlockWithSkips(din_up=base_channels*2, din_skip=base_channels*2, dout=base_channel
        self.decoderBlock3 = DecoderBlockWithSkips(din_up=base_channels, din_skip=base_channels, dout=base_channels)

    def forward(self, bottleneck, skip3, skip2, skip1):
        d1 = self.decoderBlock1(bottleneck, skip3)
        d2 = self.decoderBlock2(d1, skip2)
        d3 = self.decoderBlock3(d2, skip1)
        return d3 # Output feature map (B, base_channels, H, W)


class DecoderBlockNoSkips(nn.Module):
    """
    A decoder block without skip connections.
    Args:
        din_up (int): Number of input channels from the previous upsampled layer.
        dout (int): Number of output channels.
    Returns:
        torch.Tensor: Output tensor after upsampling and convolution.
    """
    def __init__(self, din_up, dout):
        super().__init__()
        # Upsample and change channels
        self.up = nn.ConvTranspose2d(din_up, dout, kernel_size=2, stride=2)
        # Convolutions only process the upsampled features
        # Input channels to convs is just 'dout' (output channels of self.up)
        self.convs = nn.Sequential(
            nn.Conv2d(dout, dout, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(dout),
            nn.ReLU(inplace=True),
            nn.Conv2d(dout, dout, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(dout),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        # Only takes features from the previous layer 'x'
        x_upsampled = self.up(x)
        # No concatenation
        output = self.convs(x_upsampled)
        return output


class DecoderNoSkips(nn.Module):
    """
    The decoder module without skip connections.
    Args:
        base_channels (int): Base number of channels.
    Returns:
        torch.Tensor: Output tensor after decoding.
    """
    def __init__(self, base_channels):
        super().__init__()
```

```python
        self.decoderBlock1 = DecoderBlockNoSkips(din_up=base_channels*4, dout=base_channels*2) # 256 -> 128
        self.decoderBlock2 = DecoderBlockNoSkips(din_up=base_channels*2, dout=base_channels)   # 128 -> 64
        self.decoderBlock3 = DecoderBlockNoSkips(din_up=base_channels, dout=base_channels)       # 64 -> 64

    def forward(self, bottleneck):
        # Only takes the bottleneck as input
        d1 = self.decoderBlock1(bottleneck)
        d2 = self.decoderBlock2(d1)
        d3 = self.decoderBlock3(d2)
        return d3 # Output feature map (B, base_channels, H, W)


class ReconstructionAutoencoder(nn.Module):
    """
    A reconstruction autoencoder model.
    Args:
        din (int): Number of input channels.
        dout (int): Number of output channels.
        base_channels (int): Base number of channels.
    Returns:
        torch.Tensor: Reconstructed output tensor.
    """
    def __init__(self, din, dout=3, base_channels=64):
        super().__init__()
        self.encoder = Encoder(din, base_channels)
        # No skips in Reconstruction AE
        self.decoder = DecoderNoSkips(base_channels)

        # Final layer maps DecoderSimple output (base_channels=64) to reconstruction
        self.decoderOut = nn.Sequential(
            nn.Conv2d(base_channels, dout, kernel_size=3, padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        # Encode, getting bottleneck and skip connections
        bottleneck, skip3_ignored, skip2_ignored, skip1_ignored = self.encoder(x)
        # Decode using ONLY the bottleneck with the simple decoder
        decoded_features = self.decoder(bottleneck)
        # Apply final layer
        reconstructed = self.decoderOut(decoded_features)
        return reconstructed


class SegmentationEncoder(nn.Module):
    """
    An encoder module for segmentation tasks, with optional pre-trained weights and freezing.
    Args:
        din (int): Number of input channels.
        base_channels (int): Base number of channels.
        pretrained_encoder_path (str, optional): Path to the pre-trained encoder weights. Defaults to None.
        freeze_encoder (bool, optional): Whether to freeze the encoder parameters. Defaults to True.
    Returns:
        tuple: Bottleneck and skip connections from the encoder.
    """
    def __init__(self, din, base_channels, pretrained_encoder_path=None, freeze_encoder=True):
        super().__init__()
        self.encoder = Encoder(din, base_channels)

        if pretrained_encoder_path:
            try:
                full_state_dict = torch.load(pretrained_encoder_path, weights_only=False, map_location=lambda storage,
                # Handle potential checkpoint structure variations
                if "model_state_dict" in full_state_dict:
                    model_state_dict = full_state_dict["model_state_dict"]
                elif "state_dict" in full_state_dict:
                    model_state_dict = full_state_dict["state_dict"]
```

```
                else:
                    model_state_dict = full_state_dict # Assume it's the state dict directly

                encoder_state_dict = {}
                has_encoder_prefix = any(k.startswith('encoder.') for k in model_state_dict.keys())

                for key, value in model_state_dict.items():
                    if has_encoder_prefix:
                         if key.startswith('encoder.'):
                             new_key = key[len('encoder.'):]
                             encoder_state_dict[new_key] = value

                if not encoder_state_dict:
                     print("Warning: Could not extract encoder state dict. Checkpoint might be empty or incompatible.'
                else:
                    load_result = self.encoder.load_state_dict(encoder_state_dict, strict=True) # Use strict=False fo
                    print(f"Loaded encoder weights. Load result:")
                    if load_result.missing_keys:
                        print("  Missing keys:", load_result.missing_keys)
                    if load_result.unexpected_keys:
                        print("  Unexpected keys:", load_result.unexpected_keys)
                    if not load_result.missing_keys and not load_result.unexpected_keys:
                        print("  All keys matched successfully.")

            except FileNotFoundError:
                print(f"Warning: Pre-trained encoder file not found: {pretrained_encoder_path}. Using random weights.'
            except Exception as e:
                print(f"Warning: Error loading weights: {e}. Check compatibility. Using random weights.")

        if freeze_encoder:
            if not pretrained_encoder_path:
                print("Warning: Freezing encoder, but no pre-trained weights were loaded.")
            for param in self.encoder.parameters():
                param.requires_grad = False
            print("Encoder parameters frozen.")
        else:
            print("Encoder parameters are trainable.")


    def forward(self, x):
        # Returns bottleneck, skip3, skip2, skip1
        return self.encoder(x)



class SegmentationAutoencoder(nn.Module):
    """
    A segmentation autoencoder model.
    Args:
        din (int): Number of input channels.
        base_channels (int): Base number of channels.
        num_classes (int): Number of output classes.
        pretrained_encoder_path (str, optional): Path to the pre-trained encoder weights. Defaults to None.
        freeze_encoder (bool, optional): Whether to freeze the encoder parameters. Defaults to True.
    Returns:
        torch.Tensor: Segmentation logits.
    """
    def __init__(self, din, base_channels=64, num_classes=4, pretrained_encoder_path=None, freeze_encoder=True):
        super().__init__()
        self.num_classes = num_classes

        # Initialize the Encoder (via wrapper for loading/freezing)
        self.encoder = SegmentationEncoder(din, base_channels, pretrained_encoder_path=pretrained_encoder_path, freeze

        # Use the Decoder WITH Skips for Segmentation
        self.decoder = DecoderWithSkips(base_channels)
```

```python
        # Final convolution maps DecoderWithSkips output (base_channels=64) to class scores
        self.finalConv = nn.Conv2d(base_channels, num_classes, kernel_size=1)

    def forward(self, x):
        # 1. Encoder gets bottleneck and skips
        bottleneck, skip3, skip2, skip1 = self.encoder(x)

        # 2. Decoder uses bottleneck AND skips
        decoder_output = self.decoder(bottleneck, skip3, skip2, skip1)

        # 3. Final 1x1 convolution for class logits
        segmentation_logits = self.finalConv(decoder_output)

        return segmentation_logits
```

Clip (clip/clipunet.py)

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from transformers import CLIPVisionModel, CLIPVisionConfig


class ClipViTEncoder(nn.Module):
    """
    Encodes an image using CLIP's Vision Transformer.

    Args:
        model_name (str): The name of the pre-trained CLIP model to use.
        freeze_encoder (bool): Whether to freeze the CLIP encoder's parameters.
        skip_indices (list): The indices of the hidden states to use for skip connections.

    Returns:
        bottleneck_features: The bottleneck features from the encoder.
        skip_features_list: A list of skip features from the encoder.
    """
    def __init__(self, model_name="openai/clip-vit-base-patch16", freeze_encoder=True, skip_indices=[3, 5, 7, 9]):
        super().__init__()

        self.skip_indices = sorted(skip_indices)

        self.config = CLIPVisionConfig.from_pretrained(model_name)
        self.clip_vit = CLIPVisionModel.from_pretrained(model_name)

        if freeze_encoder:
            for param in self.clip_vit.parameters():
                param.requires_grad = False

        self.grid_size = self.config.image_size // self.config.patch_size
        self.hidden_dim = self.config.hidden_size

    def forward(self, x):
        if x.shape[2] != self.config.image_size or x.shape[3] != self.config.image_size:
             print(
                 f"Input image size ({x.shape[2]}x{x.shape[3]}) doesn't match "
                 f"CLIP expected size ({self.config.image_size}x{self.config.image_size}). "
                 f"Behavior may be unexpected. Consider resizing input."
             )

        outputs = self.clip_vit(pixel_values=x, output_hidden_states=True)
        all_hidden_states = outputs.hidden_states
        last_hidden_state = outputs.last_hidden_state

        # (N, 49, 768) -> (N, 7, 7, 768) -> (N, 768, 7, 7) = (N, C, H, W)
        patch_embeddings = last_hidden_state[:, 1:, :] # Remove CLS
        bottleneck_features = patch_embeddings \
```

```python
                                .reshape(x.shape[0], self.grid_size, self.grid_size, self.hidden_dim) \
                                .permute(0, 3, 1, 2).contiguous()

        skip_features_list = []
        for i in self.skip_indices:
            hidden_state = all_hidden_states[i]
            patch_embeddings = hidden_state[:, 1:, :] # Remove CLS

            # (N, 49, 768) -> (N, 7, 7, 768) -> (N, 768, 7, 7) = (N, C, H, W)
            reshaped_features = patch_embeddings \
                                    .reshape(x.shape[0], self.grid_size, self.grid_size, self.hidden_dim) \
                                    .permute(0, 3, 1, 2).contiguous()

            skip_features_list.append(reshaped_features)

        return bottleneck_features, skip_features_list


class DecoderBlock(nn.Module):
    """
    A single decoder block for the UNet.

    Args:
        in_channels (int): The number of input channels.
        in_channels_skip (int): The number of channels from the skip connection.
        out_channels (int): The number of output channels.

    Returns:
        x (torch.Tensor): The output tensor after the decoder block.
    """
    def __init__(self, in_channels, in_channels_skip, out_channels):
        super().__init__()

        self.upsample = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
        self.skip_conv = nn.Conv2d(in_channels_skip, in_channels // 2, kernel_size=1) # 768 channels to in_channels//2

        self.conv_block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x, skip):
        x = self.upsample(x)
        skip = self.skip_conv(skip)

        if skip.shape[2:] != x.shape[2:]:
              skip= F.interpolate(skip, size=x.shape[2:], mode='bilinear', align_corners=False)

        x = torch.cat([x, skip], dim=1)

        x = self.conv_block(x)
        return x


class UNetDecoder(nn.Module):
    """
    The UNet decoder.

    Args:
        encoder_hidden_dim (int): The hidden dimension of the encoder.
        decoder_channels (list): A list of the number of channels for each decoder block.

    Returns:
```

```python
        x (torch.Tensor): The output tensor after the decoder.
    """
    def __init__(self, encoder_hidden_dim, decoder_channels):
        super().__init__()

        self.init_conv = nn.Conv2d(encoder_hidden_dim, decoder_channels[0], kernel_size=1)

        self.decoder_blocks = nn.ModuleList()
        in_channels = decoder_channels[0]
        for i in range(len(decoder_channels)-1):
            out_ch = decoder_channels[i+1]

            block = DecoderBlock(
                in_channels=in_channels,
                in_channels_skip=encoder_hidden_dim,
                out_channels=out_ch
            )
            self.decoder_blocks.append(block)

            in_channels = out_ch


    def forward(self, x, skips):
        x = self.init_conv(x) # 768 -> 1024 channels
        for block, skip in zip(self.decoder_blocks, reversed(skips)):
            x = block(x, skip)

        return x


class ClipUNet(nn.Module):
    """
    The main ClipUNet model.

    Args:
        num_classes (int): The number of output classes.
        decoder_channels (list): A list of the number of channels for each decoder block.
        freeze_encoder (bool): Whether to freeze the CLIP encoder's parameters.
        model_name (str): The name of the pre-trained CLIP model to use.
        skip_indices (list): The indices of the hidden states to use for skip connections.

    Returns:
        output (torch.Tensor): The output tensor.
    """
    def __init__(self,
                 num_classes=4,
                 decoder_channels=[1024, 512, 256, 128, 64],
                 freeze_encoder=True,
                 model_name="openai/clip-vit-base-patch16",
                 skip_indices=[3, 5, 7, 9]
                 ):
        super().__init__()

        self.encoder = ClipViTEncoder(
            model_name=model_name,
            freeze_encoder=freeze_encoder,
            skip_indices=skip_indices
        )

        self.decoder = UNetDecoder(
            encoder_hidden_dim=self.encoder.hidden_dim,
            decoder_channels=decoder_channels
        )

        self.output_layer = nn.Conv2d(decoder_channels[-1], num_classes, kernel_size=1)


```

```
    def forward(self, x):
        x, skips = self.encoder(x)
        decoder_output = self.decoder(x, skips)
        output = self.output_layer(decoder_output)
        return output

```

Clip without skip connections (clip/clipunet$_{n}oskips.py$)

```
import torch.nn as nn
from transformers import CLIPVisionModel, CLIPVisionConfig

PRETRAINED_MODEL_NAME = "openai/clip-vit-base-patch16"

class ClipViTEncoderNoSkips(nn.Module):
    """
    This class implements a CLIP ViT encoder without skip connections.

    Args:
        model_name (str, optional): The name of the pre-trained CLIP model to use. Defaults to "openai/clip-vit-base-p
        freeze_encoder (bool, optional): Whether to freeze the encoder weights. Defaults to True.

    Returns:
        torch.Tensor: The bottleneck features of the input image.
    """
    def __init__(self, model_name="openai/clip-vit-base-patch16", freeze_encoder=True): # Removed bottleneck_index
        super().__init__()

        self.config = CLIPVisionConfig.from_pretrained(model_name)
        self.clip_vit = CLIPVisionModel.from_pretrained(model_name)

        if freeze_encoder:
            # Freeze the encoder parameters if specified
            for param in self.clip_vit.parameters():
                param.requires_grad = False

        self.grid_size = self.config.image_size // self.config.patch_size
        self.hidden_dim = self.config.hidden_size

    def forward(self, x):
        if x.shape[2] != self.config.image_size or x.shape[3] != self.config.image_size:
             # Warn the user if the input image size doesn't match the expected size
             print(
                 f"Input image size ({x.shape[2]}x{x.shape[3]}) doesn't match "
                 f"CLIP expected size ({self.config.image_size}x{self.config.image_size}). "
                 f"Behavior may be unexpected. Consider resizing input."
             )

        outputs = self.clip_vit(pixel_values=x)
        last_hidden_state = outputs.last_hidden_state

        patch_embeddings = last_hidden_state[:, 1:, :] # Remove CLS token
        bottleneck_features = patch_embeddings \
                              .reshape(x.shape[0], self.grid_size, self.grid_size, self.hidden_dim) \
                              .permute(0, 3, 1, 2).contiguous()

        return bottleneck_features


class DecoderBlockNoSkip(nn.Module):
    """
    This class implements a decoder block without skip connections.

    Args:
        in_channels (int): Number of input channels.
        out_channels (int): Number of output channels.
```

```python

    Returns:
        torch.Tensor: Output tensor after the decoder block.
    """
    def __init__(self, in_channels, out_channels):
        super().__init__()

        # Upsample the input feature map
        self.upsample = nn.ConvTranspose2d(in_channels, in_channels, kernel_size=2, stride=2) # Maybe reduce channels

        # Convolutional block
        self.conv_block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1, bias=False), # Adjusted input channels here
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        x = self.upsample(x)
        x = self.conv_block(x)
        return x


class UNetDecoderNoSkips(nn.Module):
    """
    This class implements a UNet decoder without skip connections.

    Args:
        encoder_hidden_dim (int): The hidden dimension of the encoder.
        decoder_channels (list of int): The number of channels for each decoder block.

    Returns:
        torch.Tensor: The output tensor after passing through the decoder.
    """
    def __init__(self, encoder_hidden_dim, decoder_channels):
        super().__init__()

        # Initial convolution layer
        self.init_conv = nn.Conv2d(encoder_hidden_dim, decoder_channels[0], kernel_size=1)

        # Decoder blocks
        self.decoder_blocks = nn.ModuleList()
        in_channels = decoder_channels[0]
        for i in range(len(decoder_channels)-1):
            out_ch = decoder_channels[i+1]

            block = DecoderBlockNoSkip(
                in_channels=in_channels,
                out_channels=out_ch,
            )
            self.decoder_blocks.append(block)
            in_channels = out_ch

    def forward(self, x):
        x = self.init_conv(x)
        for block in self.decoder_blocks:
            x = block(x)
        return x


class ClipUNetNoSkips(nn.Module):
    """
    This class implements a CLIP UNet without skip connections.

```

```
    Args:
        num_classes (int, optional): The number of output classes. Defaults to 4.
        decoder_channels (list of int, optional): The number of channels for each decoder block. Defaults to [1024, 51
        freeze_encoder (bool, optional): Whether to freeze the encoder weights. Defaults to True.
        model_name (str, optional): The name of the pre-trained CLIP model to use. Defaults to "openai/clip-vit-base-p

    Returns:
        torch.Tensor: The output tensor after passing through the UNet.
    """
    def __init__(self,
                 num_classes=4,
                 decoder_channels=[1024, 512, 256, 128, 64],
                 freeze_encoder=True,
                 model_name="openai/clip-vit-base-patch16"
                 ):
        super().__init__()

        # Encoder
        self.encoder = ClipViTEncoderNoSkips(
            model_name=model_name,
            freeze_encoder=freeze_encoder
        )

        # Decoder
        self.decoder = UNetDecoderNoSkips(
            encoder_hidden_dim=self.encoder.hidden_dim,
            decoder_channels=decoder_channels
        )
        # Output layer
        self.output_layer = nn.Conv2d(decoder_channels[-1], num_classes, kernel_size=1)

    def forward(self, x):
        x = self.encoder(x)
        decoder_output = self.decoder(x)
        output = self.output_layer(decoder_output)
        return output
```

Prompt based model (prompt$_{b}ased/prompt.py$)

```
from clip.clipunet import ClipUNet
from unet.unet import unet
import torch
from torch import nn

class PromptModel(nn.Module):
    """
    Initializes the PromptModel.
    Args:
        path (str, optional): Path to the checkpoint file. Defaults to None.
    """
    def __init__(self, path=None):
        super().__init__()

        self.clip = ClipUNet()
        self.mask = unet(4, 1)
        self.softmax = nn.Softmax(dim=1)
        self.sigmoid = nn.Sigmoid()

        if path is not None:
            try:
                # Load the checkpoint.
                checkpoint = torch.load(path, weights_only=False, map_location=lambda storage, loc: storage) # Load to
                self.clip.load_state_dict(checkpoint["model_state_dict"])
            except Exception as e:
                print(f"Error loading checkpoint: {str(e)[:200]}")
                raise
```

```python

        # Freeze the clip parameters.
        for param in self.clip.parameters():
            param.requires_grad = False

    def forward(self, x, heatmap):
        # Pass input through clip model.
        clip_logit = self.clip(x)
        # Apply softmax to clip logits.
        clip_prob = self.softmax(clip_logit)

        # Concatenate input and heatmap, pass through mask model.
        mask_logit = self.mask(torch.concat([x, heatmap], dim=1))
        # Apply sigmoid to mask logits.
        mask_prob = self.sigmoid(mask_logit)

        # Initialize tensor to store final probabilities.
        final_probs = torch.empty_like(clip_prob)
        # Calculate selected probabilities.
        selected_prob = mask_prob * clip_prob

        # Assign probabilities to the final tensor.
        # Assign the selected probabilities for background, cat, and dog.
        final_probs[:, 1:4, :, :] = selected_prob[:, 0:3, :, :]
        # Assign the mask probability for deactivated class.
        final_probs[:, 0:1, :, :] = 1.0 - mask_prob
        # Merge boundary class with background
        final_probs[:, 1:2, :, :] += selected_prob[:, 3:4, :, :]

        return final_probs
```

Datasets (utils/dataset.py)

```python
import os
import matplotlib.pyplot as plt
from torch.utils.data import Dataset
from torchvision.io import decode_image

class dataset(Dataset):
    """
    Initializes the dataset class.
    Args:
        img_dir (str): Directory containing the images.
        label_dir (str): Directory containing the labels.
        transform (callable, optional): Optional transform to be applied on a sample. Defaults to None.
        target_transform (callable, optional): Optional transform to be applied on a target. Defaults to None.
    """
    def __init__(self, img_dir, label_dir, transform=None, target_transform=None):
        # Initialize the image and label directories
        self.img_dir = img_dir
        self.label_dir = label_dir
        # Get the names of the images, without the extension, and sort them
        self.img_names = sorted([os.path.splitext(filename)[0] for filename in os.listdir(img_dir)])
        # Calculate the length of the dataset
        self.len = len(self.img_names)
        # Initialize the transforms
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        """
        Loads and returns an image label pair from the dataset.
        Args:
```

```python
            idx (int): Index of the item to retrieve.
        Returns:
            tuple: A tuple containing the image and its corresponding label.
        """
        # Load the image and normalize it
        img = decode_image(os.path.join(self.img_dir, self.img_names[idx] + ".jpg")).float() / 255
        # Load the label
        label = decode_image(os.path.join(self.label_dir, self.img_names[idx] + ".png"))

        # Apply transformations to the image if specified
        if self.transform:
            img = self.transform(img)

        # Apply transformations to the label if specified
        if self.target_transform:
            label = self.target_transform(label)

        return img, label

class promptDataset(Dataset):
    """
    Initializes the promptDataset class.
    Args:
        img_dir (str): Directory containing the images.
        heatmap_dir (str): Directory containing the heatmaps.
        label_dir (str): Directory containing the labels.
        transform (callable, optional): Optional transform to be applied on a sample. Defaults to None.
        target_transform (callable, optional): Optional transform to be applied on a target. Defaults to None.
    """
    def __init__(self, img_dir, heatmap_dir, label_dir, transform=None, target_transform=None):
        # Initialize the image, heatmap, and label directories
        self.img_dir = img_dir
        self.heatmap_dir = heatmap_dir
        self.label_dir = label_dir
        # Get the names of the images, without the extension, and sort them
        self.img_names = sorted([os.path.splitext(filename)[0] for filename in os.listdir(img_dir)])
        # Calculate the length of the dataset
        self.len = len(self.img_names)
        # Initialize the transforms
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        """
        Loads and returns an image, heatmap, label triplet from the dataset.
        Args:
            idx (int): Index of the item to retrieve.
        Returns:
            tuple: A tuple containing the image, heatmap, and its corresponding label.
        """
        # Load the image and normalize it
        img = decode_image(os.path.join(self.img_dir, self.img_names[idx] + ".jpg")).float()/255
        # Load the heatmap and normalize it
        heatmap = decode_image(os.path.join(self.heatmap_dir, self.img_names[idx] + ".png"))/255
        # Load the label
        label = decode_image(os.path.join(self.label_dir, self.img_names[idx] + ".png"))

        # Apply transformations to the image if specified
        if self.transform:
            img = self.transform(img)

        # Apply transformations to the label if specified
        if self.target_transform:
            label = self.target_transform(label)
```

```python


        return img, heatmap, label


def display_img_label(data, idx):
    """
    Displays an image and its corresponding label using matplotlib.
    Args:
        data (torch.utils.data.Dataset): The dataset containing images and labels.
        idx (int): The index of the image and label to display.
    """
    # Get the image and label from the dataset
    img, label = data[idx]
    # Create a figure with two subplots
    figure = plt.figure(figsize=(10, 20))
    # Add the first subplot for the image
    figure.add_subplot(1, 2, 1)
    # Display the image, permuting the dimensions to match matplotlib's expected format
    plt.imshow(img.permute(1, 2, 0))

    # Add the second subplot for the label
    figure.add_subplot(1, 2, 2)
    # Display the label as a grayscale image, permuting the dimensions
    plt.imshow(label.permute(1, 2, 0), cmap='grey')

    # Show the plot
    plt.show()


class target_remap(object):
    """
    Remaps boundary class (255) to 3
    """
    def __call__(self, img):
        # Remap pixel value 255 to 3
        img[img == 255] = 3
        return img


def diff_size_collate(batch):
    """
    Collates a batch of data with potentially different image sizes.
    Args:
        batch (list): A list of tuples, where each tuple contains an image and its label.
    Returns:
        tuple: A tuple containing two lists: a list of images and a list of labels.
    """
    # Extract images from the batch
    imgs = [item[0] for item in batch]
    # Extract labels from the batch
    labels = [item[1] for item in batch]
    return imgs, labels
```

MetricsHistory (utils/MetricsHistory.py)

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

class MetricsHistory:
    """
    Accumulates TP, FP, FN, TN over an epoch for multi-class segmentation
    and computes Dice, IoU, and Accuracy metrics.
    """
    def __init__(self, num_classes: int, ignore_index: int = None, device: str = 'cpu'):
        """
```

```
        Initializes the MetricsHistory object.
        Args:
            num_classes (int): Number of classes including background.
            ignore_index (int, optional): Index of the class to ignore during metric calculation. Defaults to None.
            device (str): Device to perform initial calculations, results accumulated on CPU.
        """
        self.num_classes = num_classes
        self.ignore_index = ignore_index

        # Initialize tensors to store the sums of TP, FP, FN, and TN.
        self.total_tp = torch.zeros(num_classes, dtype=torch.float64, device='cpu')
        self.total_fp = torch.zeros(num_classes, dtype=torch.float64, device='cpu')
        self.total_fn = torch.zeros(num_classes, dtype=torch.float64, device='cpu')
        self.total_tn = torch.zeros(num_classes, dtype=torch.float64, device='cpu')

        # History lists for epoch metrics
        self.epoch_mean_dice_history = []
        self.epoch_mean_iou_history = []
        self.epoch_mean_acc_history = []

        self.epoch_per_class_dice_history = []
        self.epoch_per_class_iou_history = []
        self.epoch_per_class_acc_history = []

        self.last_per_class_iou = None
        self.last_per_class_dice = None
        self.last_per_class_acc = None

        # Metric mask (calculated once)
        self.mask = torch.ones(num_classes, dtype=torch.bool)
        # Set mask element to False if ignore_index is specified.
        if self.ignore_index is not None and 0 <= self.ignore_index < self.num_classes:
            self.mask[self.ignore_index] = False

    def reset(self):
        """
        Resets the accumulated TP, FP, FN, TN counts.
        """
        # Resets the accumulated statistics to zero at the start of each epoch.
        self.total_tp.zero_()
        self.total_fp.zero_()
        self.total_fn.zero_()
        self.total_tn.zero_()

    def accumulate(self, pred: torch.Tensor, label: torch.Tensor):
        """
        Accumulates statistics for a single prediction-label pair.

        Args:
            pred (torch.Tensor): Predicted logits or probabilities (C, H, W). Should be on self.device or moved.
            label (torch.Tensor): Ground truth label map (H, W), LongTensor. Should be on self.device or moved.
        """

        # Get hard predictions
        pred_hard = torch.argmax(pred.squeeze(0), dim=0) # (H, W)

        # One-hot encode
        label_onehot = F.one_hot(label.squeeze(0), num_classes=self.num_classes).permute(2, 0, 1).bool() # (C, H, W)
        pred_onehot = F.one_hot(pred_hard, num_classes=self.num_classes).permute(2, 0, 1).bool() # (C, H, W)

        # Calculate TP, FP, FN, TN per class
        tp = (pred_onehot & label_onehot).sum(dim=(1, 2))
        fp = (pred_onehot & ~label_onehot).sum(dim=(1, 2))
        fn = (~pred_onehot & label_onehot).sum(dim=(1, 2))
        tn = (~pred_onehot & ~label_onehot).sum(dim=(1, 2))

        # tp = (pred_onehot & label_onehot).sum(dim=(1, 2))
```

```python
        # fp = pred_onehot.sum(dim=(1, 2)) - tp
        # fn = label_onehot.sum(dim=(1, 2)) - tp
        # tn = label.numel() - fn - fp - tp

        # Accumulate on CPU with float64
        self.total_tp += tp.cpu().to(torch.float64)
        self.total_fp += fp.cpu().to(torch.float64)
        self.total_fn += fn.cpu().to(torch.float64)
        self.total_tn += tn.cpu().to(torch.float64) # Accumulate TN if needed for accuracy


    def compute_epoch_metrics(self, epsilon: float = 1e-6):
        """
        Computes the macro-averaged metrics for the accumulated epoch statistics,
        appends them to the history lists, and returns the computed mean metrics.

        Args:
            epsilon (float): Small value to avoid division by zero.

        Returns:
            tuple: (mean_dice, mean_iou, mean_acc) for the current epoch.
        """

        tp = self.total_tp
        fp = self.total_fp
        fn = self.total_fn
        tn = self.total_tn

        per_class_iou = tp / (tp + fp + fn)
        per_class_dice = (2 * tp) / (2 * tp + fp + fn)
        per_class_acc = (tp + tn) / (tp + tn + fp + fn)

        # Compute the mean IoU, Dice, and accuracy, considering the mask.
        mean_iou = per_class_iou[self.mask].mean().item()
        mean_dice = per_class_dice[self.mask].mean().item()
        mean_acc = per_class_acc[self.mask].mean().item()

        # Append to history
        self.epoch_mean_iou_history.append(mean_iou)
        self.epoch_mean_dice_history.append(mean_dice)
        self.epoch_mean_acc_history.append(mean_acc)

        self.epoch_per_class_iou_history.append(per_class_iou.numpy())
        self.epoch_per_class_dice_history.append(per_class_dice.numpy())
        self.epoch_per_class_acc_history.append(per_class_acc.numpy())

        self.last_per_class_iou = per_class_iou
        self.last_per_class_dice = per_class_dice
        self.last_per_class_acc = per_class_acc

        return mean_dice, mean_iou, mean_acc

    def to(self, device):
        """
        Moves the internal tensors to the specified device.
        Args:
            device (str): The device to move the tensors to (e.g., 'cuda', 'cpu').
        """
        self.total_tp = self.total_tp.to(device)
        self.total_fp = self.total_fp.to(device)
        self.total_fn = self.total_fn.to(device)
        self.total_tn = self.total_tn.to(device)

        self.mask = self.mask.to(device)

        if self.last_per_class_iou is not None:
            self.last_per_class_iou = self.last_per_class_iou.to(device)
```

```python

        if self.last_per_class_dice is not None:
            self.last_per_class_dice = self.last_per_class_dice.to(device)

        if self.last_per_class_acc is not None:
            self.last_per_class_acc = self.last_per_class_acc.to(device)

    def get_ignore_index(self):
        return self.ignore_index

    def get_num_classes(self):
        return self.num_classes

    def get_mean_dice_history(self):
        return self.epoch_mean_dice_history

    def get_mean_iou_history(self):
        return self.epoch_mean_iou_history

    def get_mean_acc_history(self):
        return self.epoch_mean_acc_history

    def get_class_dice_history(self):
        return self.epoch_per_class_dice_history

    def get_class_iou_history(self):
        return self.epoch_per_class_iou_history

    def get_class_acc_history(self):
        return self.epoch_per_class_acc_history

    def get_last_per_class_dice(self):
        return self.last_per_class_dice

    def get_last_per_class_iou(self):
        return self.last_per_class_iou

    def get_last_per_class_acc(self):
        return self.last_per_class_acc
```

Training Pipeline (utils/training.py)

```python
import os
import numpy as np
import torch
from torch import nn
from torch.utils.data import DataLoader
from tqdm.notebook import tqdm
from utils.utils import process_batch_forward, process_batch_reverse
from utils.MetricsHistory import MetricsHistory
from torchvision.transforms import InterpolationMode

if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

def train_loop(dataloader, model, loss_fn, optimizer, accumulation_steps, device, scheduler=None, target_size = None):
    """
    Performs one epoch of training.
    Args:
        dataloader: DataLoader for training data.
        model: The neural network model.
        loss_fn: The loss function.
```

```python
        optimizer: The optimizer.
        accumulation_steps: Number of steps to accumulate gradients before updating.
        device: The device to train on (CPU or GPU).
        scheduler: Learning rate scheduler (optional).
        target_size: Target size for image resizing (optional).
    """
    model.train()
    total_loss = 0.0
    processed_batches = 0

    optimizer.zero_grad()

    pbar = tqdm(enumerate(dataloader), total=len(dataloader), desc="Training")
    for batch_idx, (X, y) in pbar:

        if target_size is not None:
            # Resize images and labels if target_size is specified
            X, _ = process_batch_forward(X, target_size=target_size)
            y, _ = process_batch_forward(y, target_size=target_size, interpolation=InterpolationMode.NEAREST)

        X, y = X.to(device), y.to(device).long()
        pred = model(X)
        loss = loss_fn(pred, y.squeeze(1))

        scaled_loss = loss / accumulation_steps
        scaled_loss.backward()

        if (batch_idx + 1) % accumulation_steps == 0 or (batch_idx + 1) == len(dataloader):
            optimizer.step()
            if scheduler:
                scheduler.step()
            optimizer.zero_grad()

            total_loss += loss.item()
            processed_batches += 1
            pbar.set_postfix({'loss': loss.item(), 'lr': optimizer.param_groups[0]['lr']})

    avg_loss = total_loss / processed_batches if processed_batches > 0 else 0
    print(f"Training Avg loss (per effective batch): {avg_loss:>8f}")
    return avg_loss


def eval_loop(dataloader, model, loss_fn, device, target_size, agg):
    """
    Evaluation loop calculating loss, aggregated Dice, aggregated Acc, and aggregated IoU.

    Args:
        dataloader: yields batches of (list[Tensor(C,H,W)], list[Tensor(H,W)])
        model: the neural network model (on device)
        loss_fn: the combined loss function (e.g., DiceCELoss) used for training
        device: the torch device (cuda or cpu)
        target_size: the size the model expects for input
    """
    model.eval()
    num_images_processed = 0
    total_loss = 0.0
    num_classes = 4
    agg.reset()

    with torch.no_grad():
        for X, y in tqdm(dataloader, desc="Eval"):

            X, meta_list = process_batch_forward(X, target_size=target_size)
            X = X.to(device)
            preds = model(X) # Logits [N, C, target, target]

            preds = process_batch_reverse(preds, meta_list, interpolation='bilinear')
```

```
            for pred, label in zip(preds, y):
                pred = pred.to(device) # (C,H,W)
                label = label.to(device).long() # (H,W)

                loss = loss_fn(pred.unsqueeze(0), label.unsqueeze(0).squeeze(1)) # Add batch dimension
                total_loss += loss.item()
                agg.accumulate(pred, label)

                num_images_processed += 1

    avg_loss = total_loss / num_images_processed

    mean_dice, mean_iou, mean_acc = agg.compute_epoch_metrics()
    per_class_iou = agg.get_last_per_class_iou()
    ignore_index = agg.get_ignore_index()

    print(f"\n--- Evaluation Complete ---")
    print(f"  Images Processed: {num_images_processed}")
    print(f"  Average Loss (Original Size): {avg_loss:>8f}")
    print(f"  Ignored Class : {ignore_index}")
    print(f"  Macro Avg Acc score: {mean_acc:>8f}")
    print(f"  Macro Avg Dice Score: {mean_dice:>8f}")
    print(f"  Mean IoU (mIoU): {mean_iou:>8f}")
    print(f"  --- Per-Class IoU ---")
    for c in range(num_classes):
        print(f"    Class {c}: {per_class_iou[c].item():>8f}")
    print("-" * 25)

    return avg_loss, mean_dice, mean_iou

def trainReconstruction(dataloader, model, loss_fn, optimizer, accumulation_steps):
    """
    Trains a reconstruction model.
    Args:
        dataloader: DataLoader for the training data.
        model: The reconstruction model.
        loss_fn: The loss function.
        optimizer: The optimizer.
        accumulation_steps: Number of steps to accumulate gradients.
    """
    losses = []
    model.train()
    for batch_idx, (X, _) in enumerate(tqdm(dataloader, total=len(dataloader), desc="Training")):
        X = X.to(device)
        # Compute prediction
        pred = model(X)

        # Compute loss
        loss = loss_fn(pred, X)
        losses.append(loss.item())
        scaled_loss = loss / accumulation_steps

        scaled_loss.backward()

        if (batch_idx + 1) % accumulation_steps == 0 or (batch_idx + 1) == len(dataloader):
            optimizer.step()
            optimizer.zero_grad()

    return np.mean(losses)

def train_loop_prompt(dataloader, model, loss_fn, optimizer, accumulation_steps, device, scheduler=None, target_size =
    """
    Performs one epoch of training for a prompt-based model.
    Args:
        dataloader: DataLoader for training data.
        model: The prompt-based model.
```

```
        loss_fn: The loss function.
        optimizer: The optimizer.
        accumulation_steps: Number of steps to accumulate gradients.
        device: The device to train on (CPU or GPU).
        scheduler: Learning rate scheduler (optional).
        target_size: Target size for image resizing (optional).
    """
    model.train()
    total_loss = 0.0
    processed_batches = 0

    optimizer.zero_grad()

    pbar = tqdm(enumerate(dataloader), total=len(dataloader), desc="Training")
    for batch_idx, (X, p, y) in pbar:

        if target_size is not None:
            X, _ = process_batch_forward(X, target_size=target_size)
            p, _ = process_batch_forward(p, target_size=target_size)
            y, _ = process_batch_forward(y, target_size=target_size, interpolation=InterpolationMode.NEAREST)

        X, p, y = X.to(device), p.to(device), y.to(device).long()
        pred = model(X, p)
        loss = loss_fn(pred, y.squeeze(1))

        scaled_loss = loss / accumulation_steps
        scaled_loss.backward()

        if (batch_idx + 1) % accumulation_steps == 0 or (batch_idx + 1) == len(dataloader):
            optimizer.step()
            if scheduler:
                scheduler.step()
            optimizer.zero_grad()

            total_loss += loss.item()
            processed_batches += 1
            pbar.set_postfix({'loss': loss.item(), 'lr': optimizer.param_groups[0]['lr']})

    avg_loss = total_loss / processed_batches if processed_batches > 0 else 0
    print(f"Training Avg loss (per effective batch): {avg_loss:>8f}")
    return avg_loss


def evalReconstruction(dataloader, model, loss_fn, target_size, interpolation = 'bilinear'):
    """
    Evaluates a reconstruction model.
    Args:
        dataloader: DataLoader for the evaluation data.
        model: The reconstruction model.
        loss_fn: The loss function.
        target_size: Target size for image resizing.
        interpolation: Interpolation mode for resizing.
    """
    model.eval()
    num_batches = len(dataloader)
    total_loss = 0.0
    losses = []
    with torch.no_grad():
        for batch, (original_X, _) in enumerate(tqdm(dataloader, total=len(dataloader), desc="Evaluation")):
            resized_X, meta_list = process_batch_forward(original_X, target_size=target_size)   # resize X for networ
            resized_X = resized_X.to(device)

            # Compute prediction
            pred = model(resized_X)

            pred = process_batch_reverse(pred, meta_list, interpolation=interpolation)

```

```
            for p, label in zip(pred, original_X):
                # Move individual prediction and label to the device
                p = p.to(device).unsqueeze(0)  # Add batch dimension
                label = label.to(device).unsqueeze(0)  # Add batch dimension and ensure type is long

                if label.shape[1] == 4 and label.ndim == 4:
                    label = label[:, :3, :, :] # RGBA to RGB

                loss = loss_fn(p, label.squeeze(1))
                total_loss += loss.item()
                # Loss list
                losses.append(loss.item())

    return total_loss / num_batches, np.mean(losses)


def eval_loop_prompt(dataloader, model, loss_fn, device, target_size, agg):
    """
    Evaluation loop calculating loss, aggregated Dice, and aggregated IoU for a prompt-based model.

    Args:
        dataloader: yields batches of (list[Tensor(C,H,W)], list[Tensor(H,W)])
        model: the neural network model (on device)
        loss_fn: the combined loss function (e.g., DiceCELoss) used for training
        device: the torch device (cuda or cpu)
        target_size: the size the model expects for input
    """
    model.eval()
    num_images_processed = 0
    total_loss = 0.0
    num_classes = 4

    with torch.no_grad():
        for X, p, y in tqdm(dataloader, desc="Eval"):

            X, meta_list = process_batch_forward(X, target_size=target_size)
            p, _ = process_batch_forward(p, target_size=target_size)
            X, p = X.to(device), p.to(device)
            preds = model(X, p) # Logits [N, C, target, target]

            preds = process_batch_reverse(preds, meta_list, interpolation='bilinear')

            for pred, label in zip(preds, y):
                pred = pred.to(device) # (C,H,W)
                label = label.to(device).long() # (H,W)

                loss = loss_fn(pred.unsqueeze(0), label.unsqueeze(0).squeeze(1)) # Add batch dimension
                total_loss += loss.item()
                agg.accumulate(pred, label)

                num_images_processed += 1

    avg_loss = total_loss / num_images_processed

    mean_dice, mean_iou, mean_acc = agg.compute_epoch_metrics()
    per_class_iou = agg.get_last_per_class_iou()
    ignore_index = agg.get_ignore_index()

    print(f"\n--- Evaluation Complete ---")
    print(f"  Images Processed: {num_images_processed}")
    print(f"  Average Loss (Original Size): {avg_loss:>8f}")
    print(f"  Ignored Class : {ignore_index}")
    print(f"  Macro Avg Acc score: {mean_acc:>8f}")
    print(f"  Macro Avg Dice Score: {mean_dice:>8f}")
    print(f"  Mean IoU (mIoU): {mean_iou:>8f}")
    print(f"  --- Per-Class IoU ---")
    for c in range(num_classes):
```

```python
        print(f"    Class {c}: {per_class_iou[c].item():>8f}")
    print("-" * 25)

    return avg_loss, mean_dice, mean_iou


def start_prompt(
        model_save_dir: str,
        model_save_name: str,
        model: nn.Module,
        optimizer: torch.optim,
        train_dataloader: DataLoader,
        val_dataloader: DataLoader,
        accumulation_steps: int,
        device: torch.device,
        train_loss_fn: nn.Module,
        val_loss_fn: nn.Module,
        target_size: int,
        scheduler: torch.optim.lr_scheduler = None,
        agg: MetricsHistory = None,
        load: bool = True,
        save: bool = True,
        num_classes: int = 4,
        ignore_index: int = 3,
        epochs: int = 100,
):
    """
    Starts the training pipeline for a prompt-based model.
    Args:
        model_save_dir: Directory to save the model.
        model_save_name: Name of the model file.
        model: The prompt-based model.
        optimizer: The optimizer.
        train_dataloader: DataLoader for the training data.
        val_dataloader: DataLoader for the validation data.
        accumulation_steps: Number of steps to accumulate gradients.
        device: The device to train on (CPU or GPU).
        train_loss_fn: The loss function for training.
        val_loss_fn: The loss function for validation.
        target_size: Target size for image resizing.
        scheduler: Learning rate scheduler (optional).
        agg: MetricsHistory object (optional).
        load: Whether to load a checkpoint (default: True).
        save: Whether to save the model (default: True).
        num_classes: Number of classes (default: 4).
        ignore_index: Index to ignore in the loss calculation (default: 3).
        epochs: Number of training epochs (default: 100).
    """
    start_epoch = 0
    best_dev_dice = -np.inf
    best_dev_miou = -np.inf
    best_dev_loss = np.inf

    os.makedirs(model_save_dir, exist_ok=True)
    os.makedirs(f"{model_save_dir}/metrics", exist_ok=True)
    if load and os.path.isfile(f"{model_save_dir}/{model_save_name}"):
        print(f"Loading checkpoint from: {model_save_dir}/{model_save_name}")

        checkpoint = torch.load(f"{model_save_dir}/{model_save_name}", map_location=device)

        model.load_state_dict(checkpoint["model_state_dict"])
        print(" -> Model state loaded.")

        # Load optimizer state
        try:
            optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
            print(" -> Optimizer state loaded.")
```

```
        except Exception as e:
            print(f" -> Warning: Could not load optimizer state: {e}. Optimizer will start from scratch.")

        # Load scheduler state
        try:
            scheduler.load_state_dict(checkpoint["scheduler_state_dict"])
            print(" -> Scheduler state loaded.")
        except Exception as e:
            print(f" -> Warning: Could not load scheduler state: {e}. Scheduler will start from scratch.")

        # Load Metrics History
        try:
            agg = checkpoint.get("history")
            agg.to(device)
            print(" -> Metrics History loaded.")
        except Exception as e:
            print(f" -> No metric history saved")
            agg = MetricsHistory(num_classes, ignore_index)

        # Load training metadata
        start_epoch = checkpoint.get("epoch", 0)
        best_dev_dice = checkpoint.get("best_dev_dice", -np.inf)
        best_dev_miou = checkpoint.get("best_dev_miou", -np.inf)
        best_dev_loss = checkpoint.get("best_dev_loss", np.inf)

        print(f" -> Resuming training from epoch {start_epoch + 1}")
        print(f" -> Loaded best metrics: Dice={best_dev_dice:.6f}, mIoU={best_dev_miou:.6f}, Loss={best_dev_loss:.6f}'
        loaded_notes = checkpoint.get("notes", "N/A")
        print(f" -> Notes from checkpoint: {loaded_notes}")

    else:
        print(f"Checkpoint file not found at {model_save_dir}/{model_save_name}. Starting training from scratch.")

    # --- Training and Evaluation Loop ---
    print("\nStarting Training...")
    for t in range(start_epoch, epochs):
        print(f"Epoch {t+1}\n-------------------------------")

        train_loss = train_loop_prompt(train_dataloader, model, train_loss_fn, optimizer, accumulation_steps, device,
        val_loss, val_dice, val_miou = eval_loop_prompt(val_dataloader, model, val_loss_fn, device, target_size, agg)

        if save:
            metrics = {
                "epoch": t + 1,
                "history": agg
            }
            torch.save(metrics, f"{model_save_dir}/metrics/{model_save_name}")

        if val_miou > best_dev_miou:
            best_dev_dice = val_dice
            best_dev_miou = val_miou
            best_dev_loss = val_loss

            if save and scheduler:
                print(f"Validation IoU score improved ({best_dev_miou:.6f}). Saving model...")

                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                    "optimizer_state_dict": optimizer.state_dict(),
                    "scheduler_state_dict": scheduler.state_dict(),
                    "best_dev_dice": best_dev_dice,
                    "best_dev_miou": best_dev_miou,
                    "best_dev_loss": best_dev_loss,
                    "history": agg,
                    "notes": f"Model saved based on best Micro Dice. Ignored index for metric: {ignore_index}"
                }
```

```python
                torch.save(checkpoint, f"{model_save_dir}/{model_save_name}")
            elif save:
                print(f"Validation IoU score improved ({best_dev_miou:.6f}). Saving model...")

                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                    "optimizer_state_dict": optimizer.state_dict(),
                    "best_dev_dice": best_dev_dice,
                    "best_dev_miou": best_dev_miou,
                    "best_dev_loss": best_dev_loss,
                    "history": agg,
                    "notes": f"Model saved based on best Micro Dice. Ignored index for metric: {ignore_index}"
                }
                torch.save(checkpoint, f"{model_save_dir}/{model_save_name}")
        else:
            print(f"Validation IoU score did not improve from {best_dev_miou:.6f}")


    print("\n--- Training Finished! ---")
    print(f"Best validation IoU score achieved: {best_dev_miou:.6f}")
    print(f"Corresponding validation dice: {best_dev_dice:.6f}")
    print(f"Corresponding validation loss: {best_dev_loss:.6f}")
    print(f"Best model saved to: {os.path.join(model_save_dir, model_save_name)}")


def start(
        model_save_dir: str,
        model_save_name: str,
        model: nn.Module,
        optimizer: torch.optim,
        train_dataloader: DataLoader,
        val_dataloader: DataLoader,
        accumulation_steps: int,
        device: torch.device,
        train_loss_fn: nn.Module,
        val_loss_fn: nn.Module,
        target_size: int,
        scheduler: torch.optim.lr_scheduler = None,
        agg: MetricsHistory = None,
        load: bool = True,
        save: bool = True,
        num_classes: int = 4,
        ignore_index: int = 3,
        epochs: int = 100,
):
    """
    Starts the training pipeline.
    Args:
        model_save_dir: Directory to save the model.
        model_save_name: Name of the model file.
        model: The model.
        optimizer: The optimizer.
        train_dataloader: DataLoader for the training data.
        val_dataloader: DataLoader for the validation data.
        accumulation_steps: Number of steps to accumulate gradients.
        device: The device to train on (CPU or GPU).
        train_loss_fn: The loss function for training.
        val_loss_fn: The loss function for validation.
        target_size: Target size for image resizing.
        scheduler: Learning rate scheduler (optional).
        agg: MetricsHistory object (optional).
        load: Whether to load a checkpoint (default: True).
        save: Whether to save the model (default: True).
        num_classes: Number of classes (default: 4).
        ignore_index: Index to ignore in the loss calculation (default: 3).
        epochs: Number of training epochs (default: 100).
```

```
    """
    start_epoch = 0
    best_dev_dice = -np.inf
    best_dev_miou = -np.inf
    best_dev_loss = np.inf

    os.makedirs(model_save_dir, exist_ok=True)
    os.makedirs(f"{model_save_dir}/metrics", exist_ok=True)
    if load and os.path.isfile(f"{model_save_dir}/{model_save_name}"):
        print(f"Loading checkpoint from: {model_save_dir}/{model_save_name}")

        # Load checkpoint
        checkpoint = torch.load(f"{model_save_dir}/{model_save_name}", map_location=device, weights_only=True)

        # Load model state
        model.load_state_dict(checkpoint["model_state_dict"])
        print(" -> Model state loaded.")

        # Load optimizer state
        try:
            optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
            print(" -> Optimizer state loaded.")
        except Exception as e:
            print(f" -> Warning: Could not load optimizer state: {e}. Optimizer will start from scratch.")

        # Load scheduler state
        try:
            scheduler.load_state_dict(checkpoint["scheduler_state_dict"])
            print(" -> Scheduler state loaded.")
        except Exception as e:
            print(f" -> Warning: Could not load scheduler state: {e}. Scheduler will start from scratch.")

        # Load Metrics History
        try:
            agg = checkpoint.get("history")
            agg.to(device)
            print(" -> Metrics History loaded.")
        except Exception as e:
            print(f" -> No metric history saved")
            agg = MetricsHistory(num_classes, ignore_index)

        # Load training metadata
        start_epoch = checkpoint.get("epoch", 0)
        best_dev_dice = checkpoint.get("best_dev_dice", -np.inf)
        best_dev_miou = checkpoint.get("best_dev_miou", -np.inf)
        best_dev_loss = checkpoint.get("best_dev_loss", np.inf)

        print(f" -> Resuming training from epoch {start_epoch + 1}")
        print(f" -> Loaded best metrics: Dice={best_dev_dice:.6f}, mIoU={best_dev_miou:.6f}, Loss={best_dev_loss:.6f}"
        loaded_notes = checkpoint.get("notes", "N/A")
        print(f" -> Notes from checkpoint: {loaded_notes}")

    else:
        print(f"Checkpoint file not found at {model_save_dir}/{model_save_name}. Starting training from scratch.")

    # --- Training and Evaluation Loop ---
    print("\nStarting Training...")
    for t in range(start_epoch, epochs):
        print(f"Epoch {t+1}\n-------------------------------")

        train_loss = train_loop(train_dataloader, model, train_loss_fn, optimizer, accumulation_steps, device, schedul
        val_loss, val_dice, val_miou = eval_loop(val_dataloader, model, val_loss_fn, device, target_size, agg)

        if save:
            metrics = {
                "epoch": t + 1,
                "history": agg
```

```
            }
            torch.save(metrics, f"{model_save_dir}/metrics/{model_save_name}")

        if val_miou > best_dev_miou:
            best_dev_dice = val_dice
            best_dev_miou = val_miou
            best_dev_loss = val_loss

            if save and scheduler:
                print(f"Validation IoU score improved ({best_dev_miou:.6f}). Saving model...")

                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                    "optimizer_state_dict": optimizer.state_dict(),
                    "scheduler_state_dict": scheduler.state_dict(),
                    "best_dev_dice": best_dev_dice,
                    "best_dev_miou": best_dev_miou,
                    "best_dev_loss": best_dev_loss,
                    # "history": agg,
                    "notes": f"Model saved based on best Micro Dice. Ignored index for metric: {ignore_index}"
                }
                torch.save(checkpoint, f"{model_save_dir}/{model_save_name}")
                # Save model weights only
                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                }
                torch.save(checkpoint, f"{model_save_dir}/MO_{model_save_name}")
            elif save:
                print(f"Validation IoU score improved ({best_dev_miou:.6f}). Saving model...")

                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                    "optimizer_state_dict": optimizer.state_dict(),
                    "best_dev_dice": best_dev_dice,
                    "best_dev_miou": best_dev_miou,
                    "best_dev_loss": best_dev_loss,
                    # "history": agg,
                    "notes": f"Model saved based on best Micro Dice. Ignored index for metric: {ignore_index}"
                }
                torch.save(checkpoint, f"{model_save_dir}/{model_save_name}")
                # Save model weights only
                checkpoint = {
                    "epoch": t + 1,
                    "model_state_dict": model.state_dict(),
                }
                torch.save(checkpoint, f"{model_save_dir}/MO_{model_save_name}")
        else:
            print(f"Validation IoU score did not improve from {best_dev_miou:.6f}")


    print("\n--- Training Finished! ---")
    print(f"Best validation IoU score achieved: {best_dev_miou:.6f}")
    print(f"Corresponding validation dice: {best_dev_dice:.6f}")
    print(f"Corresponding validation loss: {best_dev_loss:.6f}")
    print(f"Best model saved to: {os.path.join(model_save_dir, model_save_name)}")
```

Utils (utils/utils.py)

```
import torch
import matplotlib.pyplot as plt
import numpy as np
import torchvision.transforms.functional
import torch.nn.functional as F
```

```python
from torchvision.transforms import InterpolationMode
from torchvision.transforms import functional as TF
from tqdm import tqdm
from PIL import Image
from typing import Optional, List


def resize_with_padding(image, target_size=512, interpolation=InterpolationMode.BILINEAR):
    """
    Resize a single image (Tensor of shape (C, H, W)) so that the longer side
    equals target_size, preserving aspect ratio; add black padding as needed.
    Args:
        image (Tensor): Input image tensor of shape (C, H, W).
        target_size (int): The target size for the longer side of the image.
        interpolation (InterpolationMode): The interpolation method to use.
    Returns:
        tuple: A tuple containing the resized and padded image, plus a metadata dictionary.
    """
    _, orig_h, orig_w = image.shape
    scale = min(target_size / orig_w, target_size / orig_h)
    new_w = int(round(orig_w * scale))
    new_h = int(round(orig_h * scale))

    # Resize the image
    image_resized = TF.resize(image, size=(new_h, new_w), interpolation=interpolation)

    # Compute padding on each side
    pad_w = target_size - new_w
    pad_h = target_size - new_h
    pad_left = pad_w // 2
    pad_right = pad_w - pad_left
    pad_top = pad_h // 2
    pad_bottom = pad_h - pad_top

    # Pad the image (padding order: left, top, right, bottom)
    image_padded = TF.pad(image_resized, padding=(pad_left, pad_top, pad_right, pad_bottom), fill=0)

    meta = {
        "original_size": (orig_h, orig_w),
        "new_size": (new_h, new_w),
        "pad": (pad_left, pad_top, pad_right, pad_bottom),
        "scale": scale
    }
    return image_padded, meta

def reverse_resize_and_padding(image, meta, interpolation="bilinear"):
    """
    Remove the padding from image (Tensor of shape (C, target_size, target_size))
    using metadata and then resize the cropped image back to the original size.
    Args:
        image (Tensor): The input image tensor.
        meta (dict): The metadata dictionary containing information about the original and new sizes and padding.
        interpolation (str): The interpolation method to use ("bilinear" or "nearest").
    Returns:
        Tensor: The image with padding removed and resized to the original size.
    """
    pad_left, pad_top, pad_right, pad_bottom = meta["pad"]
    new_h, new_w = meta["new_size"]

    # Crop out the padding: from pad_top to pad_top+new_h and pad_left to pad_left+new_w.
    image_cropped = image[..., pad_top: pad_top + new_h, pad_left: pad_left + new_w]

    # Resize the cropped image back to the original size.
    orig_h, orig_w = meta["original_size"]
    # F.interpolate expects a 4D tensor.
    image_original = F.interpolate(image_cropped.unsqueeze(0),
                                   size=(orig_h, orig_w),
```

```python
                                    mode=interpolation,
                                    align_corners=False if interpolation != "nearest" else None)
    return image_original.squeeze(0)

def process_batch_forward(batch_images, target_size=512, interpolation=InterpolationMode.BILINEAR):
    """
    Process a batch (Tensor of shape (N, C, H, W)) by resizing each image to target_size
    with aspect ratio preserved (adding black padding).
    Args:
        batch_images (Tensor): A batch of images (N, C, H, W).
        target_size (int): The target size for the longer side of the image.
        interpolation (InterpolationMode): The interpolation method to use.
    Returns:
        tuple: A tuple containing the processed batch and a list of meta dictionaries.
    """
    resized_batch = []
    meta_list = []
    for image in batch_images:
        # If the image has 4 channels, slice to keep only the first 3 (RGB).
        if image.ndim == 3 and image.shape[0] == 4:
            image = image[:3, ...]
        image_resized, meta = resize_with_padding(image, target_size, interpolation)
        resized_batch.append(image_resized)
        meta_list.append(meta)
    return torch.stack(resized_batch), meta_list

def process_batch_reverse(batch_outputs, meta_list, interpolation="bilinear"):
    """
    Given a batch of network outputs of shape (N, C, target_size, target_size) and the
    corresponding meta info, reverse the transform for each one to obtain predictions at their
    original sizes.
    Args:
        batch_outputs (Tensor): A batch of network outputs (N, C, target_size, target_size).
        meta_list (list): A list of meta dictionaries, one for each image in the batch.
        interpolation (str): The interpolation method to use ("bilinear" or "nearest").
    Returns:
        list: A list of tensors, each with the original size.
    """
    original_outputs = []
    for output, meta in zip(batch_outputs, meta_list):
        restored = reverse_resize_and_padding(output, meta, interpolation=interpolation)
        original_outputs.append(restored)
    return original_outputs

def calculate_class_weights(
    label_source,
    num_classes: int,
    ignore_index: Optional[int] = None,
    source_type: str = 'files',
    unimportant_class_indices: Optional[List[int]] = None, # Indices to down-weight
    target_unimportant_weight: float = 1.0, # Target weight for unimportant classes
    normalize_target_sum: float = -1.0 # Normalize weights sum (-1 means num_classes)
) -> torch.Tensor:
    """
    Calculates class weights based on inverse frequency, then adjusts weights
    for specified unimportant classes and re-normalizes.
    Args:
        label_source: Source of labels, can be a list of file paths or a dataset.
        num_classes (int): The total number of classes.
        ignore_index (Optional[int]): Index to ignore in the labels.
        source_type (str): Type of source, 'files' or 'dataset'.
        unimportant_class_indices (Optional[List[int]]): Indices of unimportant classes.
        target_unimportant_weight (float): Target weight for unimportant classes.
        normalize_target_sum (float): Normalize weights sum.  If -1, normalize to num_classes.
    Returns:
        torch.Tensor: A tensor of class weights.
    """
```

```python

    counts = torch.zeros(num_classes, dtype=torch.float64)
    total_valid_pixels = 0

    iterator = None
    num_labels = 0
    if source_type == 'files':
        iterator = label_source
        num_labels = len(label_source)
    elif source_type == 'dataset':
        iterator = range(len(label_source))
        num_labels = len(label_source)
    else:
        raise ValueError("source_type must be either 'files' or 'dataset'")

    print(f"Processing {num_labels} labels...")
    pbar = tqdm(iterator, total=num_labels)
    for idx_or_path in pbar:
        label_data = None
        if source_type == 'files':
            path = idx_or_path; img = Image.open(path)
            label_data = torch.from_numpy(np.array(img))
        elif source_type == 'dataset':
                _, label_data = label_source[idx_or_path];
                label_data = torch.tensor(label_data) if not isinstance(label_data, torch.Tensor) else label_data

        label_long = label_data.long().view(-1)

        if ignore_index is not None:
            valid_mask = (label_long != ignore_index)
            label_valid = label_long[valid_mask]
        else:
            label_valid = label_long
        label_valid = torch.clamp(label_valid, 0, num_classes - 1)

        if label_valid.numel() > 0:
                counts += torch.bincount(label_valid, minlength=num_classes).double()
                total_valid_pixels += label_valid.numel()

    print("\nFinished counting.")
    print(f"Raw pixel counts per class: {counts.long().tolist()}")
    print(f"Total valid pixels counted: {total_valid_pixels}")

    frequencies = counts / total_valid_pixels
    epsilon = 1e-6
    inverse_frequencies = 1.0 / (frequencies + epsilon)

    weights = inverse_frequencies

    if unimportant_class_indices:
        for idx in unimportant_class_indices:
            weights[idx] = min(weights)

    target_sum = normalize_target_sum if normalize_target_sum > 0 else float(num_classes)
    final_weights = weights / weights.sum() * target_sum

    print(f"Calculated Final Class Weights: {final_weights.tolist()}")

    return final_weights.float()


def convert_rgb_label_to_classes(label_array_rgb):
    """
    Converts a 3-channel RGB label map to a 1-channel class map.

    Mapping:
    [0, 0, 0] (Black)      -> 0 (Background)
```

```
    [128, 0, 0] (Red)       -> 1 (Cat)
    [0, 128, 0] (Green)     -> 2 (Dog)
    [255, 255, 255] (White) -> 0 (Background) - Assuming white is also background
    Other                  -> 255 (Ignore)

    Args:
        label_array_rgb (np.ndarray): A HxWx3 NumPy array (uint8).

    Returns:
        np.ndarray: A HxW NumPy array (uint8) with class indices.
    """
    # Input validation
    if label_array_rgb.ndim != 3 or label_array_rgb.shape[2] != 3:
        raise ValueError(
            "Input label must be 3-channel RGB (HxWx3), "
            f"but got shape {label_array_rgb.shape}"
        )

    h, w, _ = label_array_rgb.shape
    # Initialize with ignore value (255)
    label_map_1channel = np.full((h, w), 255, dtype=np.uint8)

    # Define colors as tuples for comparison
    black = (0, 0, 0)
    red = (128, 0, 0)
    green = (0, 128, 0)
    white = (255, 255, 255)

    # Create boolean masks for each color
    # Comparing tuples is faster for multi-channel exact matches typically
    mask_black = np.all(label_array_rgb == black, axis=2)
    mask_red = np.all(label_array_rgb == red, axis=2)
    mask_green = np.all(label_array_rgb == green, axis=2)
    mask_white = np.all(label_array_rgb == white, axis=2)

    # Apply mapping (order can matter if masks could overlap, but shouldn't here)
    # Map backgrounds first
    label_map_1channel[mask_black] = 0
    label_map_1channel[mask_white] = 0 # Map white to background class 0
    # Map foreground classes
    label_map_1channel[mask_red] = 1   # Cat
    label_map_1channel[mask_green] = 2 # Dog
    # Any remaining pixels stay 255

    return label_map_1channel
```

Losses (utils/weighted$_{loss.py}$)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from typing import Optional, Callable

class WeightedMemoryEfficientDiceLoss(nn.Module):
    """
    Calculates a memory-efficient Soft Dice loss, optionally with class weights.
    Args:
        apply_softmax (bool): Whether to apply softmax to the input logits. Defaults to True.
        ignore_index (int, optional): Index to ignore in the loss calculation. Defaults to None.
        class_weights (torch.Tensor, optional): Weights for each class. Defaults to None.
        smooth (float): Smoothing factor to prevent division by zero. Defaults to 1e-5.
    """
    def __init__(self,
                 apply_softmax: bool = True,
                 ignore_index: Optional[int] = None,
                 class_weights: Optional[torch.Tensor] = None,
```

```
                 smooth: float = 1e-5):
        super().__init__()
        self.apply_softmax = apply_softmax
        self.ignore_index = ignore_index
        self.smooth = smooth

        if class_weights is not None:
            self.class_weights = class_weights
        else:
            self.class_weights = None


    def forward(self, x, y):

        num_classes = x.shape[1]
        shp_y = y.shape
        if self.apply_softmax:
            probs = F.softmax(x, dim=1)
        else:
            probs = x

        with torch.no_grad():
            # Handle potential shape mismatches between predictions and targets
            if len(shp_y) != len(probs.shape):
                if len(shp_y) == len(probs.shape) - 1 and len(shp_y) >= 2 and shp_y == probs.shape[2:]:
                     y = y.unsqueeze(1)
                elif len(shp_y) == len(probs.shape) and shp_y[1] == 1: pass # ok
                else: raise ValueError(f"Shape mismatch: probs {probs.shape}, y {shp_y}")
            y_long = y.long()

            mask = None
            if probs.shape == y.shape:
                 y_onehot = y.float()
                 if mask is not None:
                      y_indices_for_mask = torch.argmax(y_onehot, dim=1, keepdim=True)
                      mask = (y_indices_for_mask != self.ignore_index)
                      y_onehot = y_onehot * mask
            else:
                y_onehot = torch.zeros_like(probs, device=probs.device)
                y_onehot.scatter_(1, y_long, 1)
                if mask is not None: y_onehot = y_onehot * mask

            sum_gt = y_onehot.sum(dim=(2, 3))

        if mask is not None:
             probs = probs * mask

        intersect_persample = (probs * y_onehot).sum(dim=(2, 3))
        sum_pred_persample = probs.sum(dim=(2, 3))
        sum_gt_persample = sum_gt

        # Aggregate across batch
        intersect = intersect_persample.sum(0)
        sum_pred = sum_pred_persample.sum(0)
        sum_gt = sum_gt_persample.sum(0)

        # Dice
        denominator = sum_pred + sum_gt
        dc = (2. * intersect + self.smooth) / (torch.clip(denominator + self.smooth, 1e-8))

        valid_classes_mask = torch.ones_like(dc, dtype=torch.bool)
        if self.ignore_index is not None and 0 <= self.ignore_index < num_classes:
            valid_classes_mask[self.ignore_index] = False

        dc_final = torch.tensor(0.0, device=dc.device)

        dc_valid = dc[valid_classes_mask]
```

```python

        if self.class_weights is not None:
            # Use weighted average for valid classes
            weights = self.class_weights.to(dc_valid.device)
            weights_valid = weights[valid_classes_mask]

            weighted_sum = (dc_valid * weights_valid).sum()
            weight_sum = weights_valid.sum()
            dc_final = weighted_sum / weight_sum.clamp(min=1e-8)
        else:
            dc_final = dc_valid.mean()

        return -dc_final # Return negative Dice score as loss



class WeightedDiceCELoss(nn.Module):
    """
    Combines WeightedMemoryEfficientDiceLoss and Cross Entropy Loss with class_weights support.
    Args:
        dice_weight (float): Weight for the Dice loss. Defaults to 1.0.
        ce_weight (float): Weight for the Cross Entropy loss. Defaults to 1.0.
        ignore_index (int, optional): Index to ignore in the loss calculation. Defaults to None.
        class_weights (torch.Tensor, optional): Weights for each class. Defaults to None.
        smooth_dice (float): Smoothing factor for Dice loss. Defaults to 1e-5.
        ce_kwargs (dict): Keyword arguments for CrossEntropyLoss. Defaults to {}.
    """
    def __init__(self,
                 dice_weight: float = 1.0,
                 ce_weight: float = 1.0,
                 ignore_index: Optional[int] = None,
                 class_weights: Optional[torch.Tensor] = None,
                 smooth_dice: float = 1e-5,
                 ce_kwargs={}):
        super().__init__()
        self.dice_weight = dice_weight
        self.ce_weight = ce_weight
        self.ignore_index = ignore_index

        self.dice = WeightedMemoryEfficientDiceLoss(
            apply_softmax=True,
            ignore_index=ignore_index,
            class_weights=class_weights,
            smooth=smooth_dice
        )

        ce_final_kwargs = ce_kwargs.copy()
        if ignore_index is not None:
            ce_final_kwargs['ignore_index'] = ignore_index
        if class_weights is not None:
            ce_final_kwargs['weight'] = class_weights

        self.cross_entropy = nn.CrossEntropyLoss(**ce_final_kwargs)

    def forward(self, outputs, targets):
        if targets.ndim == 3:
            targets_dice = targets.unsqueeze(1).long() # Assuming [N, H, W]
        elif targets.ndim == 4 and targets.shape[1] == 1:
            targets_dice = targets.long()
        else:
            if targets.ndim == outputs.ndim and targets.shape[1] != 1:
                raise ValueError(f"Target shape {targets.shape} has multiple channels but expected class indices [N, H
            else:
                raise ValueError(f"Unsupported target shape {targets.shape} for CE. Expected [N, H, W] or [N, 1, H, W]

        dice_loss = self.dice(outputs, targets_dice)

```

```python
        if targets.ndim == 4 and targets.shape[1] == 1:
            targets_ce = targets.squeeze(1).long()
        elif targets.ndim == 3:
            targets_ce = targets.long()
        else:
            if targets.ndim == outputs.ndim and targets.shape[1] != 1:
                raise ValueError(f"Target shape {targets.shape} has multiple channels but expected class indices [N, H
            else:
                raise ValueError(f"Unsupported target shape {targets.shape} for CE. Expected [N, H, W] or [N, 1, H, W]

        ce_loss = self.cross_entropy(outputs, targets_ce)

        combined_loss = (self.dice_weight * dice_loss) + (self.ce_weight * ce_loss)
        return combined_loss



class WeightedMemoryEfficientDiceLossPrompt(nn.Module):
    """
    Calculates a memory-efficient Dice loss, optionally with class weights and a non-linearity.
    Args:
        dice_nonlin (Callable, optional): Non-linearity function to apply to the predictions. Defaults to None.
        apply_softmax (bool): Whether to apply softmax to the input logits. Defaults to True.
        ignore_index (int, optional): Index to ignore in the loss calculation. Defaults to None.
        class_weights (torch.Tensor, optional): Weights for each class. Defaults to None.
        smooth (float): Smoothing factor to prevent division by zero. Defaults to 1e-5.
    """
    def __init__(self,
                 dice_nonlin: Callable = None,
                 apply_softmax: bool = True,
                 ignore_index: Optional[int] = None,
                 class_weights: Optional[torch.Tensor] = None, # New parameter
                 smooth: float = 1e-5):
        super().__init__()
        self.apply_softmax = apply_softmax
        self.ignore_index = ignore_index
        self.smooth = smooth
        self.dice_nonlin = dice_nonlin

        # Store class weights, ensuring they are a Tensor if provided
        if class_weights is not None:
            assert isinstance(class_weights, torch.Tensor), "class_weights must be a torch.Tensor"
            self.class_weights = class_weights
        else:
            self.class_weights = None


    def forward(self, x, y):
        num_classes = x.shape[1]
        shp_y = y.shape

        if self.apply_softmax:
            probs = F.softmax(x, dim=1)
        else:
            probs = x

        if self.dice_nonlin is not None:
            probs = self.dice_nonlin(probs)

        with torch.no_grad():
            if len(shp_y) != len(probs.shape):
                if len(shp_y) == len(probs.shape) - 1 and len(shp_y) >= 2 and shp_y == probs.shape[2:]:
                     y = y.unsqueeze(1)
                elif len(shp_y) == len(probs.shape) and shp_y[1] == 1: pass # ok
                else: raise ValueError(f"Shape mismatch: probs {probs.shape}, y {shp_y}")
            y_long = y.long()

```

```
            mask = None
            if probs.shape == y.shape:
                 y_onehot = y.float()
                 if mask is not None:
                      y_indices_for_mask = torch.argmax(y_onehot, dim=1, keepdim=True)
                      mask = (y_indices_for_mask != self.ignore_index)
                      y_onehot = y_onehot * mask
            else:
                y_onehot = torch.zeros_like(probs, device=probs.device)
                y_onehot.scatter_(1, y_long, 1)
                if mask is not None: y_onehot = y_onehot * mask

            sum_gt = y_onehot.sum(dim=(2, 3))

        if mask is not None:
             probs = probs * mask

        intersect_persample = (probs * y_onehot).sum(dim=(2, 3))
        sum_pred_persample = probs.sum(dim=(2, 3))
        sum_gt_persample = sum_gt

        # Aggregate across batch
        intersect = intersect_persample.sum(0)
        sum_pred = sum_pred_persample.sum(0)
        sum_gt = sum_gt_persample.sum(0)

        # Dice
        denominator = sum_pred + sum_gt
        dc = (2. * intersect + self.smooth) / (torch.clip(denominator + self.smooth, 1e-8))

        valid_classes_mask = torch.ones_like(dc, dtype=torch.bool)
        if self.ignore_index is not None and 0 <= self.ignore_index < num_classes:
            valid_classes_mask[self.ignore_index] = False

        dc_final = torch.tensor(0.0, device=dc.device)
        dc_valid = dc[valid_classes_mask]
        if self.class_weights is not None:
            weights = self.class_weights.to(dc_valid.device)
            weights_valid = weights[valid_classes_mask]
            weighted_sum = (dc_valid * weights_valid).sum()
            weight_sum = weights_valid.sum()
            dc_final = weighted_sum / weight_sum.clamp(min=1e-8)
        else:
            dc_final = dc_valid.mean()

        return -dc_final # Return negative Dice score as loss


class WeightedDiceNLLLoss(nn.Module):
    """
    Combines WeightedMemoryEfficientDiceLossPrompt and Cross Entropy Loss with class_weights support.
    Args:
        dice_weight (float): Weight for the Dice loss. Defaults to 1.0.
        nll_weight (float): Weight for the NLL loss. Defaults to 1.0.
        ignore_index (int, optional): Index to ignore in the loss calculation. Defaults to None.
        class_weights (torch.Tensor, optional): Weights for each class. Defaults to None.
        smooth_dice (float): Smoothing factor for Dice loss. Defaults to 1e-5.
        apply_softmax (bool): Whether to apply softmax to the input logits. Defaults to True.
        dice_nonlin (Callable, optional): Non-linearity function to apply to the predictions for Dice loss. Defaults t
        nll_nonlin (Callable, optional): Non-linearity function to apply to the predictions for NLL loss. Defaults to
        nll_kwargs (dict): Keyword arguments for NLLLoss. Defaults to {}.
    """
    def __init__(self,
                 dice_weight: float = 1.0,
                 nll_weight: float = 1.0,
                 ignore_index: Optional[int] = None,
                 class_weights: Optional[torch.Tensor] = None,
```

```
                 smooth_dice: float = 1e-5,
                 apply_softmax: bool = True,
                 dice_nonlin: Callable = None,
                 nll_nonlin: Callable = None,
                 nll_kwargs={}):
        super().__init__()
        self.dice_weight = dice_weight
        self.nll_weight = nll_weight
        self.ignore_index = ignore_index
        self.dice_nonlin = dice_nonlin
        self.nll_nonlin = nll_nonlin

        self.dice = WeightedMemoryEfficientDiceLossPrompt(
            apply_softmax=apply_softmax,
            ignore_index=ignore_index,
            class_weights=class_weights,
            smooth=smooth_dice
        )

        nll_final_kwargs = nll_kwargs.copy()
        if ignore_index is not None:
            nll_final_kwargs['ignore_index'] = ignore_index
        if class_weights is not None:
            nll_final_kwargs['weight'] = class_weights

        self.nll = nn.NLLLoss(**nll_final_kwargs)

    def forward(self, outputs, targets):
        if targets.ndim == 3:
            targets_dice = targets.unsqueeze(1).long()
        elif targets.ndim == 4 and targets.shape[1] == 1:
            targets_dice = targets.long()
        else:
            if targets.ndim == outputs.ndim and targets.shape[1] != 1:
                raise ValueError(f"Target shape {targets.shape} has multiple channels but expected class indices [N, H
            else:
                raise ValueError(f"Unsupported target shape {targets.shape} for CE. Expected [N, H, W] or [N, 1, H, W]


        dice_loss = self.dice(outputs, targets_dice)

        if targets.ndim == 4 and targets.shape[1] == 1:
             targets_nll = targets.squeeze(1).long()
        elif targets.ndim == 3:
             targets_nll = targets.long()
        else:
            if targets.ndim == outputs.ndim and targets.shape[1] != 1:
                raise ValueError(f"Target shape {targets.shape} has multiple channels but expected class indices [N, H
            else:
                raise ValueError(f"Unsupported target shape {targets.shape} for CE. Expected [N, H, W] or [N, 1, H, W]

        if self.nll_nonlin is not None:
            outputs = self.nll_nonlin(outputs)
        nll_loss = self.nll(outputs, targets_nll)

        combined_loss = (self.dice_weight * dice_loss) + (self.nll_weight * nll_loss)
        return combined_loss
```

Augmentations (utils/augmentation.ipynb)

Resize

```
import imgaug.augmenters as iaa
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

# Padding augmenter
```

```
pad_aug = iaa.PadToAspectRatio(
    1.0, # Keep aspect ratio
    position="center",
    pad_mode="constant",
    pad_cval=0
)

# Resizers for Image and Labels
resize_img = iaa.Resize(256, interpolation="cubic")
resize_mask = iaa.Resize(256, interpolation="nearest")

# Function to resize images
def image_resizer(images, random_state, parents, hooks):
    """
    Resizes the images using the resize_img augmenter.
    Args:
        images (list): A list of images to resize.
        random_state: Random state for augmentation.
        parents: Parents for augmentation.
        hooks: Hooks for augmentation.
    Returns:
        list: A list of resized images.
    """
    return [resize_img.augment_image(img) for img in images]

# Function to resize masks
def label_resizer(segmaps, random_state, parents, hooks):
    """
    Resizes the segmentation masks using the resize_mask augmenter.
    Args:
        segmaps (list): A list of SegmentationMapsOnImage objects to resize.
        random_state: Random state for augmentation.
        parents: Parents for augmentation.
        hooks: Hooks for augmentation.
    Returns:
        list: A list of resized SegmentationMapsOnImage objects.
    """
    new_segmaps = []
    for segmap in segmaps:
        new_arr = resize_mask.augment_image(segmap.arr) # Resize the mask array.
        new_segmaps.append(SegmentationMapsOnImage(new_arr, shape=new_arr.shape)) # Create a new SegmentationMapsOnIma
    return new_segmaps

# Resizing augmenter
resize_aug = iaa.Sequential([
    pad_aug, # Apply padding to maintain aspect ratio.
    iaa.Lambda( # Apply lambda to resize images and masks.
        func_images=image_resizer, # Function to resize images.
        func_segmentation_maps=label_resizer, # Function to resize segmentation masks.
    )
])
```

Rotation

```
import imgaug.augmenters as iaa
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

# Applies rotation augmentation to both images and segmentation maps
rotation_aug = iaa.Sequential([
    iaa.Affine(
        rotate=(45, 315), # Rotation angle
        fit_output=True, # Maintain full rotated image
        mode="constant",
        cval=0,
        backend="cv2"
```

```
    ),
    resize_aug
])
```

Random Cropping

```
import imgaug.augmenters as iaa
from imgaug.imgaug import SegmentationMapsOnImage

class CenterSquareCropAugmenter(iaa.Augmenter):
    """
    CenterSquareCropAugmenter crops images to a square shape, centered.
    Args:
        name (str, optional): Name of the augmenter. Defaults to None.
        deterministic (bool, optional): Whether the augmentation is deterministic. Defaults to False.
        random_state (None, optional): Random state. Defaults to None.
    """
    def __init__(self, name=None, deterministic=False, random_state=None):
        super(CenterSquareCropAugmenter, self).__init__(
            name=name, deterministic=deterministic, random_state=random_state)
        self.cropper = iaa.CropToAspectRatio(1.0, position="center") # Square cropper

    def _augment_images(self, images, random_state, parents, hooks):
        """
        Applies the center square crop to a list of images.
        Args:
            images (list of numpy.ndarray): List of images.
            random_state (numpy.random.RandomState): Random state.
            parents (imgaug.parameters.StochasticParameter): Parents.
            hooks (imgaug.hook.HooksImages): Hooks.
        Returns:
            list of numpy.ndarray: List of cropped images.
        """
        return [self.cropper.augment_image(img) for img in images]

    def _augment_segmentation_maps(self, segmaps, random_state, parents, hooks):
        """
        Applies the center square crop to segmentation maps.
        Args:
            segmaps (list of imgaug.imgaug.SegmentationMapsOnImage): List of segmentation maps.
            random_state (numpy.random.RandomState): Random state.
            parents (imgaug.parameters.StochasticParameter): Parents.
            hooks (imgaug.hook.HooksImages): Hooks.
        Returns:
            list of imgaug.imgaug.SegmentationMapsOnImage: List of cropped segmentation maps.
        """
        out = []
        for segmap in segmaps:
            cropped_arr = self.cropper.augment_image(segmap.arr)
            out.append(SegmentationMapsOnImage(cropped_arr, shape=cropped_arr.shape))
        return out

    def get_parameters(self):
        return []


class RandomSquareCropAugmenter(iaa.Augmenter):
    """
    RandomSquareCropAugmenter crops images to a square shape, randomly.
    Args:
        crop_factor (float, optional): Ratio of the smallest edge to use for the square. Defaults to 2/3.
        name (str, optional): Name of the augmenter. Defaults to None.
        deterministic (bool, optional): Whether the augmentation is deterministic. Defaults to False.
        random_state (None, optional): Random state. Defaults to None.
    """
```

```
    def __init__(self, crop_factor=2/3, name=None, deterministic=False, random_state=None):
        """
        crop_factor: Ratio of the smallest edge to use for the square.
        """
        super(RandomSquareCropAugmenter, self).__init__(
            name=name, deterministic=deterministic, random_state=random_state)
        self.crop_factor = crop_factor

    def _augment_images(self, images, random_state, parents, hooks):
        """
        Applies the random square crop to a list of images.
        Args:
            images (list of numpy.ndarray): List of images.
            random_state (numpy.random.RandomState): Random state.
            parents (imgaug.parameters.StochasticParameter): Parents.
            hooks (imgaug.hook.HooksImages): Hooks.
        Returns:
            list of numpy.ndarray: List of cropped images.
        """
        out_images = []
        for img in images:
            H, W = img.shape[:2]
            min_side = min(H, W)
            crop_size = int(min_side * self.crop_factor)
            max_x = W - crop_size
            max_y = H - crop_size
            x1 = random_state.randint(0, max_x + 1)
            y1 = random_state.randint(0, max_y + 1)
            cropped_img = img[y1: y1 + crop_size, x1: x1 + crop_size]
            out_images.append(cropped_img)
        return out_images

    def _augment_segmentation_maps(self, segmaps, random_state, parents, hooks):
        """
        Applies the random square crop to segmentation maps.
        Args:
            segmaps (list of imgaug.imgaug.SegmentationMapsOnImage): List of segmentation maps.
            random_state (numpy.random.RandomState): Random state.
            parents (imgaug.parameters.StochasticParameter): Parents.
            hooks (imgaug.hook.HooksImages): Hooks.
        Returns:
            list of imgaug.imgaug.SegmentationMapsOnImage: List of cropped segmentation maps.
        """
        out_segmaps = []
        for segmap in segmaps:
            arr = segmap.arr
            H, W = arr.shape[:2]
            min_side = min(H, W)
            crop_size = int(min_side * self.crop_factor)
            max_x = W - crop_size
            max_y = H - crop_size
            x1 = random_state.randint(0, max_x + 1)
            y1 = random_state.randint(0, max_y + 1)
            cropped_arr = arr[y1: y1 + crop_size, x1: x1 + crop_size]
            out_segmaps.append(SegmentationMapsOnImage(cropped_arr, shape=cropped_arr.shape))
        return out_segmaps

    def get_parameters(self):
        return [self.crop_factor]

# Center crop augmenter.
center_crop_aug = iaa.Sequential([CenterSquareCropAugmenter(), resize_aug])

# Random square crop augmenter.
random_crop_aug = iaa.Sequential([RandomSquareCropAugmenter(), resize_aug])
```

Random Masking

```python
import numpy as np
import imgaug.augmenters as iaa
from imgaug.augmentables.segmaps import SegmentationMapsOnImage

# Define augmentation for masking images.
mask_im_aug = iaa.Sequential([
    iaa.CoarseDropout(p=0.15, size_percent=(1/50), random_state=2)
])

# Define augmentation for masking labels.
mask_label_aug = iaa.Sequential([
    iaa.CoarseDropout(p=0.15, size_percent=(1/50), random_state=2)
])

def random_masking_labels(segmaps, random_state, parents, hooks):
    """
    Applies random masking to segmentation maps.
    Args:
        segmaps: Input segmentation maps.
        random_state: Random state for augmentation.
        parents: Parent objects.
        hooks: Hooks for augmentation.
    Returns:
        List of augmented segmentation maps.
    """
    new_segmaps = []
    for segmap in segmaps:
        # Convert segmentation map array to uint8 for augmentation.
        segmap_arr_uint8 = segmap.arr.astype(np.uint8)
        # Apply mask_label_aug to the segmentation map.
        new_arr = mask_label_aug.augment_image(segmap_arr_uint8)
        # Create a new SegmentationMapsOnImage object with the augmented array.
        new_segmaps.append(SegmentationMapsOnImage(new_arr, shape=new_arr.shape))
    return new_segmaps

# Random Masking augmentation
masking_aug = iaa.Sequential([
    iaa.Lambda(
        func_images=lambda images, rs, parents, hooks: [mask_im_aug.augment_image(img) for img in images],
        func_segmentation_maps=random_masking_labels
    ),
    resize_aug # Apply resizing after masking
])
```

Grayscale

```python
# Grayscale augmentation
grayscale = iaa.Grayscale(alpha=1.0, from_colorspace="RGB")
grayscale_aug = iaa.Sequential([
    grayscale,
    resize_aug
])
```

Laplace Noise

```python
# Laplace Noise Augmentation
laplace = iaa.AdditiveLaplaceNoise(scale=(0.1*255, 0.3*255), per_channel=True)
laplace_aug = iaa.Sequential([
    laplace,
```

```
    resize_aug
])
```

Blur

```
# Blur Augmentation
blur = iaa.AverageBlur(k=(12))
blur_aug = iaa.Sequential([
    blur,
    resize_aug
])
```

Contrast

```
# Contrast Augmentation
contrast = iaa.LinearContrast((0.2, 0.6))
contrast_aug = iaa.Sequential([
    contrast,
    resize_aug
])
```

Merge

```
from PIL import Image
import math
import os
import numpy as np
from utils import convert_rgb_label_to_classes


def combine_images_preserve_aspect_ratio(image1_path, image2_path, output_path=None, is_label=False):
    """
    Combines two images, preserving aspect ratio, centers on 256x256, then optionally converts to a 1-channel class ma
    If is_label is True, converts the final RGB image to a 1-channel class map
    using convert_rgb_label_to_classes before saving/returning.

    Args:
        image1_path (str): Path to the first image.
        image2_path (str): Path to the second image.
        output_path (str, optional): Path to save the final image. Defaults to None.
        is_label (bool, optional): If True, apply label conversion. Defaults to False.

    Returns:
        PIL.Image.Image: The final combined image (RGB or L mode).

    Raises:
        FileNotFoundError, ValueError, IOError, RuntimeError as before.
    """
    TARGET_DIMENSION = 256
    RESAMPLE_METHOD = Image.Resampling.NEAREST

    def load_image(path):
        """
        Loads an image from the given path and converts it to RGB mode.
        Handles RGBA, LA, and P modes by converting them to RGB to avoid issues.

        Args:
            path (str): The path to the image file.

        Returns:
```

```
            PIL.Image.Image: The loaded image in RGB mode.
        """
        img = Image.open(path)
        if img.mode == 'RGBA':
            # Create a new RGB image with a black background and paste the image, masking the alpha channel
            background = Image.new('RGB', img.size, (0, 0, 0))
            background.paste(img, mask=img.split()[-1])
            img = background
        elif img.mode == 'LA':
            # Convert LA to RGBA, create a new RGB image with a black background, and paste the image, masking the alp
            rgba_img = img.convert('RGBA')
            background = Image.new('RGB', rgba_img.size, (0, 0, 0))
            background.paste(rgba_img, mask=rgba_img.split()[-1])
            img = background
        elif img.mode == 'P':
                # Convert P to RGBA, create a new RGB image with a black background, and paste the image, masking the
                rgba_img = img.convert('RGBA')
                background = Image.new('RGB', rgba_img.size, (0, 0, 0))
                background.paste(rgba_img, mask=rgba_img.split()[-1])
                img = background
        return img.convert('RGB')


    img1 = load_image(image1_path)
    img2 = load_image(image2_path)

    # 2. Dimensions & Orientation
    w1, h1 = img1.size
    w2, h2 = img2.size

    def get_orientation(w, h):
        """
        Determine the orientation of an image (portrait or landscape).

        Args:
            w (int): Width of the image.
            h (int): Height of the image.

        Returns:
            str: 'portrait' if the image is portrait, 'landscape' otherwise.
        """
        return 'portrait' if h > w else 'landscape'

    orientation1 = get_orientation(w1, h1)
    orientation2 = get_orientation(w2, h2)

    if orientation1 != orientation2:
        print(f" Mismatched orientations ({orientation1} vs {orientation2}) for {os.path.basename(image1_path)}, {os.p
        return None
    orientation = orientation1

    # 3. Calculate Scale
    if orientation == 'portrait':
        total_original_major_dim = w1 + w2
        if total_original_major_dim == 0:
            return None
        scale = TARGET_DIMENSION / total_original_major_dim
    else: # landscape
        total_original_major_dim = h1 + h2
        if total_original_major_dim == 0:
            return None
        scale = TARGET_DIMENSION / total_original_major_dim

    # 4. Calculate Scaled Dimensions
    scaled_w1 = max(1, math.ceil(w1 * scale))
    scaled_h1 = max(1, math.ceil(h1 * scale))
    scaled_w2 = max(1, math.ceil(w2 * scale))
```

```python
    scaled_h2 = max(1, math.ceil(h2 * scale))

    # 5. Adjust for Exact Fit
    final_w1, final_h1 = scaled_w1, scaled_h1
    final_w2, final_h2 = scaled_w2, scaled_h2
    if orientation == 'portrait':
        diff = (scaled_w1 + scaled_w2) - TARGET_DIMENSION
        if diff > 0:
            final_w1 -= diff if scaled_w1 >= scaled_w2 else 0
            final_w2 -= diff if scaled_w2 > scaled_w1 else 0
    else:
        diff = (scaled_h1 + scaled_h2) - TARGET_DIMENSION
        if diff > 0:
            final_h1 -= diff if scaled_h1 >= scaled_h2 else 0
            final_h2 -= diff if scaled_h2 > scaled_h1 else 0

    final_w1, final_h1, final_w2, final_h2 = max(1, final_w1), max(1, final_h1), max(1, final_w2), max(1, final_h2)

    # 6. Resize Images

    img1_resized = img1.resize((final_w1, final_h1), RESAMPLE_METHOD)
    img2_resized = img2.resize((final_w2, final_h2), RESAMPLE_METHOD)


    # 7. Create Combined Strip
    if orientation == 'portrait':
        combined_w, combined_h = TARGET_DIMENSION, max(final_h1, final_h2)
        combined = Image.new('RGB', (combined_w, combined_h), (0, 0, 0))
        combined.paste(img1_resized, (0, 0))
        combined.paste(img2_resized, (final_w1, 0))
    else: # landscape
        combined_w, combined_h = max(final_w1, final_w2), TARGET_DIMENSION
        combined = Image.new('RGB', (combined_w, combined_h), (0, 0, 0))
        combined.paste(img1_resized, (0, 0))
        combined.paste(img2_resized, (0, final_h1))

    # 8. Create Final Canvas & Center
    final_img = Image.new('RGB', (TARGET_DIMENSION, TARGET_DIMENSION), (0, 0, 0))
    paste_x = (TARGET_DIMENSION - combined.width) // 2
    paste_y = (TARGET_DIMENSION - combined.height) // 2
    final_img.paste(combined, (paste_x, paste_y))

    # 9. Label Conversion
    if is_label:
        # Convert final PIL Image to NumPy array
        final_img_np = np.array(final_img)
        # Apply the RGB -> Class ID conversion
        label_map_1channel = convert_rgb_label_to_classes(final_img_np)
        # Convert the 1-channel NumPy array back to a PIL Image (mode 'L')
        final_img = Image.fromarray(label_map_1channel, mode='L')


    # 10. Save if output path is provided
    if output_path:
        final_img.save(output_path)

    return final_img
```

Augmentation without merge

```python
import os
import random
import imageio
import numpy as np
from imgaug.augmentables.segmaps import SegmentationMapsOnImage
```

```python
import math

# Define a dictionary mapping augmenter names to their corresponding functions
augmenter_dict = {
    "rotation": rotation_aug,
    "center_crop": center_crop_aug,
    "random_crop": random_crop_aug,
    "masking": masking_aug,
    "grayscale": grayscale_aug,
    "laplace": laplace_aug,
    "blur": blur_aug,
    "contrast": contrast_aug
}

# Calculate the number of augmenters
num_augmenters = len(augmenter_dict) # Count based on the dictionary

# Configuration
folder_path = "Train/color" # Path to the folder containing color images
label_folder_path = "Train/label" # Path to the folder containing label images
save_color_dir = "astrain/color" # Directory to save augmented color images
save_label_dir = "astrain/label" # Directory to save augmented label images
majority_aug_factor = 1.5 # Augmentation factor to balance the dataset

# Create the output directories if they don't exist
os.makedirs(save_color_dir, exist_ok=True)
os.makedirs(save_label_dir, exist_ok=True)

# File Discovery and Classification
print("Scanning for image files...")
filenames = [
    f for f in os.listdir(folder_path)
    if f.lower().endswith(('.jpg', '.png'))
]
# This section assumes that the file names can be used to identify the species

def get_species(filename):
    """
    Extracts the species name from a filename.
    Args:
        filename (str): The name of the file.
    Returns:
        str: The species name.
    """
    base = os.path.splitext(filename)[0]
    parts = base.rsplit('_', 1)
    return parts[0] if len(parts) > 1 else base

# Define a set of cat species for classification
cat_species = {
    "Russian_Blue", "Siamese", "Sphynx", "Maine_Coon", "Abyssinian",
    "Bombay", "British_Shorthair", "Bengal", "Egyptian_Mau", "Persian",
    "Ragdoll", "Birman"
}

# Initialize lists to store cat and dog filenames
cat_files = []
dog_files = []
for fname in filenames:
    species = get_species(fname)
    name_no_ext = os.path.splitext(fname)[0]
    label_path_check = os.path.join(label_folder_path, name_no_ext + ".png")
    if species in cat_species:
        cat_files.append(name_no_ext)
    else:
        dog_files.append(name_no_ext)

```

```python
# Get the number of cat and dog files
N_cat = len(cat_files)
N_dog = len(dog_files)

print(f"Initial counts: Cats = {N_cat}, Dogs = {N_dog}")

# Copy Original Files
print("Processing originals with resize augmentation...")
all_original_files = cat_files + dog_files
processed_count = 0

# Ensure destination directories exist
os.makedirs(save_color_dir, exist_ok=True)
os.makedirs(save_label_dir, exist_ok=True)

for fname in all_original_files:
    orig_color_path = os.path.join(folder_path, fname + ".jpg")
    orig_label_path = os.path.join(label_folder_path, fname + ".png")

    # Define destination paths (using original base name)
    dest_color_path = os.path.join(save_color_dir, fname + ".jpg")
    dest_label_path = os.path.join(save_label_dir, fname + ".png")

    # Read the input image and its label
    img = imageio.v2.imread(orig_color_path)
    label = imageio.v2.imread(orig_label_path)

    # Create a segmentation map object
    segmap = SegmentationMapsOnImage(label, shape=img.shape)

    # Apply the resize augmentation to both image and label map
    resized_img, resized_segmap = resize_aug(image=img, segmentation_maps=segmap)
    resized_label = resized_segmap.get_arr()

    if resized_img.ndim == 3 and resized_img.shape[2] == 4:
        resized_img = resized_img[..., :3] # RGBA to RGB

    resized_label = convert_rgb_label_to_classes(resized_label)

    # Ensure correct data types before saving
    resized_img = resized_img.astype(np.uint8)
    resized_label = resized_label.astype(np.uint8)

    # Save the processed (resized) images
    imageio.imwrite(dest_color_path, resized_img)
    imageio.imwrite(dest_label_path, resized_label)

    processed_count += 1

print(f"Processed and saved {processed_count} original image/label pairs using resize_aug.")


# --- Calculate Augmentation Needs ---
if N_cat == N_dog:
    target_final_count = round(N_dog * majority_aug_factor)
elif N_cat < N_dog:
    target_final_count = round(N_dog * majority_aug_factor)
else: # N_dog < N_cat
    target_final_count = round(N_cat * majority_aug_factor)

total_aug_cat_needed = max(0, target_final_count - N_cat)
total_aug_dog_needed = max(0, target_final_count - N_dog)

print(f"Target final count per class: {target_final_count}")
print(f"Total augmentations needed: Cats = {total_aug_cat_needed}, Dogs = {total_aug_dog_needed}")

num_cats_per_aug = math.ceil(total_aug_cat_needed / num_augmenters)
```

```
num_dogs_per_aug = math.ceil(total_aug_dog_needed / num_augmenters)

print(f"Will select approximately {num_cats_per_aug} cats and {num_dogs_per_aug} dogs per augmenter.")

num_selected_cats = 0
num_selected_dogs = 0

# Augmentation Loop
generated_aug_count = 0
for i, (aug_name, aug_object) in enumerate(augmenter_dict.items()):
    # Randomly select files for augmentation
    selected_cats = random.choices(cat_files, k=num_cats_per_aug) if N_cat > 0 and num_cats_per_aug > 0 else []
    selected_dogs = random.choices(dog_files, k=num_dogs_per_aug) if N_dog > 0 and num_dogs_per_aug > 0 else []
    selected_files = selected_cats + selected_dogs
    num_selected_cats += len(selected_cats)
    num_selected_dogs += len(selected_dogs)

    print(f"\nUsing augmenter '{aug_name}' ({i+1}/{num_augmenters}): processing {len(selected_files)} images ({len(sel

    processed_in_batch = 0
    for fname in selected_files:
        color_path = os.path.join(folder_path, fname + ".jpg")
        label_path = os.path.join(label_folder_path, fname + ".png")

        # Read images
        img = imageio.v2.imread(color_path)
        label = imageio.v2.imread(label_path)
        segmap = SegmentationMapsOnImage(label, shape=img.shape)

        # Apply the augmentation
        augmented_img, augmented_segmap = aug_object(image=img, segmentation_maps=segmap)
        augmented_label = augmented_segmap.get_arr()

        if augmented_img.ndim == 3 and augmented_img.shape[2] == 4:
                augmented_img = augmented_img[..., :3] # RGBA to RGB

        augmented_label = convert_rgb_label_to_classes(augmented_label)

        # Type casting
        augmented_label = augmented_label.astype(np.uint8)
        augmented_img = augmented_img.astype(np.uint8)

        out_color_path = os.path.join(save_color_dir, f"{fname}_{aug_name}_{processed_in_batch}.jpg")
        out_label_path = os.path.join(save_label_dir, f"{fname}_{aug_name}_{processed_in_batch}.png")

        # Save the augmented images
        imageio.imwrite(out_color_path, augmented_img)
        imageio.imwrite(out_label_path, augmented_label)

        generated_aug_count += 1
        processed_in_batch += 1

    print(f"Augmenter '{aug_name}' finished. Processed {processed_in_batch} images.")
```

Augmentation with merge

```
import os
import random

SOURCE_COLOR_DIR = "Train/color"
SOURCE_LABEL_DIR = "Train/label"
DEST_COLOR_DIR = "astrain/color"
DEST_LABEL_DIR = "astrain/label"
NUM_COMBINATIONS_PER_TYPE = 126

```

```python
cat_species = {
    "Russian_Blue", "Siamese", "Sphynx", "Maine_Coon", "Abyssinian",
    "Bombay", "British_Shorthair", "Bengal", "Egyptian_Mau", "Persian",
    "Ragdoll", "Birman"
}

def get_species(filename):
    """
    Extracts the species name from a filename.
    Args:
        filename (str): The name of the file.
    Returns:
        str: The species name or the base filename if no species is found.
    """
    base = os.path.splitext(filename)[0]
    parts = base.rsplit('_', 1)
    return parts[0] if len(parts) > 1 else base


# 1. Create destination directories
os.makedirs(DEST_COLOR_DIR, exist_ok=True)
os.makedirs(DEST_LABEL_DIR, exist_ok=True)

# 2. Scan source directory and classify files
all_files_in_color = [
    f for f in os.listdir(SOURCE_COLOR_DIR)
    if f.lower().endswith(('.jpg', '.png')) # Assuming color can be jpg or png
]


cat_files = []
dog_files = []

for fname_ext in all_files_in_color:
    fname_no_ext = os.path.splitext(fname_ext)[0]
    label_path_check = os.path.join(SOURCE_LABEL_DIR, fname_no_ext + ".png")


    species = get_species(fname_ext)
    if species in cat_species:
        cat_files.append(fname_no_ext)
    else:
        dog_files.append(fname_no_ext)

N_cat = len(cat_files)
N_dog = len(dog_files)

print(f"Found {N_cat} cat images with labels.")
print(f"Found {N_dog} dog images with labels.")

# Function to generate combinations for a specific type
def generate_combinations(combo_type, files1_list, files2_list, num_required, output_prefix):
    """
    Generates N combinations by selecting files from lists and calling combine_images.
    Prints the source files used for each successful combination.

    Args:
        combo_type (str): Description (e.g., "1 Cat + 1 Dog")
        files1_list (list): List of base filenames for the first image.
        files2_list (list): List of base filenames for the second image.
        num_required (int): Target number of successful combinations.
        output_prefix (str): Prefix for output filenames (e.g., "cat_dog").
    """

    combinations_done = 0
    attempts = 0
    max_attempts = num_required * 10
```

```
    generated_pairs = set()
    file1_base, file2_base = None, None

    while combinations_done < num_required and attempts < max_attempts:
        attempts += 1

        if files1_list is files2_list:
            # If combining from the same list, sample 2 unique files.
            if len(files1_list) < 2:
                break
            file1_base, file2_base = random.sample(files1_list, 2)
        else:
            # If combining from different lists, sample one from each.
            if not files1_list or not files2_list:
                break
            file1_base = random.choice(files1_list)
            file2_base = random.choice(files2_list)

        pair_key = tuple(sorted((file1_base, file2_base)))
        if pair_key in generated_pairs:
            # Skip if this pair has already been generated.
            continue

        # Construct paths
        img1_color_ext = ".jpg"
        img2_color_ext = ".jpg"
        img1_label_ext = ".png"
        img2_label_ext = ".png"

        img1_color_path = os.path.join(SOURCE_COLOR_DIR, file1_base + img1_color_ext)
        img1_label_path = os.path.join(SOURCE_LABEL_DIR, file1_base + img1_label_ext)
        img2_color_path = os.path.join(SOURCE_COLOR_DIR, file2_base + img2_color_ext)
        img2_label_path = os.path.join(SOURCE_LABEL_DIR, file2_base + img2_label_ext)

        # Define output paths
        output_base_name = f"{output_prefix}_{combinations_done}"
        output_color_path = os.path.join(DEST_COLOR_DIR, output_base_name + ".jpg")
        output_label_path = os.path.join(DEST_LABEL_DIR, output_base_name + ".png")

        # Combine color images
        combined_color = combine_images_preserve_aspect_ratio(img1_color_path, img2_color_path, output_color_path)

        # Combine label images
        combined_label = combine_images_preserve_aspect_ratio(img1_label_path, img2_label_path, output_label_path, Tru

        print(f"\n  Generated: {output_base_name}.jpg/.png using [{file1_base}{img1_color_ext}, {file2_base}{img2_colo

        combinations_done += 1
        generated_pairs.add(pair_key)


# 1. 1 Cat + 1 Dog
generate_combinations(
    combo_type="1 Cat + 1 Dog",
    files1_list=cat_files,
    files2_list=dog_files,
    num_required=NUM_COMBINATIONS_PER_TYPE,
    output_prefix="cat_dog"
)

# 2. 2 Cats
generate_combinations(
    combo_type="2 Cats",
    files1_list=cat_files,
    files2_list=cat_files,
    num_required=NUM_COMBINATIONS_PER_TYPE,
```

```python
    output_prefix="cat_cat"
)

# 3. 2 Dogs
generate_combinations(
    combo_type="2 Dogs",
    files1_list=dog_files,
    files2_list=dog_files,
    num_required=NUM_COMBINATIONS_PER_TYPE,
    output_prefix="dog_dog"
)

print("\n--- Combination process finished. ---")
```

Augmentations for prompt based segmentation

```python
import numpy as np
import random
import torch
import numpy as np
import random
import os
import time
import shutil
from torchvision.io import read_image
from PIL import Image
from utils.dataset import target_remap

def create_gaussian_heatmap(size=(256, 256), sigma=3.0):
    """
    Creates a 2D heatmap array with a Gaussian spot centered at a random pixel.

    Args:
        size (tuple): The (height, width) dimensions of the heatmap array.
        sigma (float): The standard deviation (spread) of the Gaussian function.
                       Larger sigma means a wider, smoother spot.

    Returns:
        numpy.ndarray: A 2D numpy array representing the heatmap (values typically 0-1).
        tuple: The (y, x) coordinates of the chosen center pixel.
    """
    height, width = size
    if height <= 0 or width <= 0:
        raise ValueError("Size dimensions must be positive integers.")
    if sigma <= 0:
        raise ValueError("Sigma must be positive.")

    # 1. Create a black canvas (array of zeros)
    heatmap = np.zeros((height, width), dtype=np.float32) # Use float for calculations

    # 2. Pick a random center pixel
    center_y = random.randint(0, height - 1)
    center_x = random.randint(0, width - 1)
    print(f"Selected center pixel (y, x): ({center_y}, {center_x})")

    # 3. Create coordinate grids
    y_coords, x_coords = np.indices((height, width))

    # 4. Calculate the squared Euclidean distance from the center for each pixel
    dist_sq = (x_coords - center_x)**2 + (y_coords - center_y)**2

    # 5. Calculate the Gaussian function
    heatmap = np.exp(-dist_sq / (2 * sigma**2))

    return heatmap, (center_y, center_x)
```

```python


# --- Helper function for the selection process ---
def select_dominant_class(heatmap, remapped_mask):
    """
    Selects the dominant class in a mask based on heatmap scores.

    Args:
        heatmap (numpy.ndarray): The heatmap array.
        remapped_mask (numpy.ndarray): The remapped mask array.

    Returns:
        int: The selected class (0 if no class is dominant).
        dict: A dictionary of class scores.
    """
    class_scores = {}
    present_classes = np.unique(remapped_mask)
    target_classes = present_classes[present_classes > 0] # Classes 1, 2, 3

    if target_classes.size == 0: return 0, {}

    for class_val in target_classes:
        mask_pixels = (remapped_mask == class_val)
        if np.any(mask_pixels):
             score = np.sum(heatmap[mask_pixels])
             class_scores[class_val] = score
        else:
             class_scores[class_val] = 0

    if not class_scores or all(s < 1e-9 for s in class_scores.values()):
        selected_class = 0
    else:
        selected_class = max(class_scores, key=class_scores.get)

    return selected_class, class_scores


TRAIN_IMG_DIR   = "astrain/color"
TRAIN_LBL_DIR   = "astrain/label"

HEATMAP_SIGMA   = 3.0
MAX_ATTEMPTS    = 1000

PSTRAIN_BASE_DIR    = "pstrain"                # New base output directory
PSTRAIN_IMG_DIR     = os.path.join(PSTRAIN_BASE_DIR, "color") # For COPIED original images
PSTRAIN_HEATMAP_DIR = os.path.join(PSTRAIN_BASE_DIR, "point_prompt") # For heatmap IMAGES
PSTRAIN_LABEL_DIR   = os.path.join(PSTRAIN_BASE_DIR, "label") # For final label masks


start_time = time.time()

os.makedirs(PSTRAIN_IMG_DIR, exist_ok=True)
os.makedirs(PSTRAIN_HEATMAP_DIR, exist_ok=True)
os.makedirs(PSTRAIN_LABEL_DIR, exist_ok=True)
print(f"Reading original images from:   {os.path.abspath(TRAIN_IMG_DIR)}")
print(f"Reading labels from:            {os.path.abspath(TRAIN_LBL_DIR)}")
print("-" * 30)
print(f"Saving copied images to:        {os.path.abspath(PSTRAIN_IMG_DIR)}")
print(f"Saving heatmap images to:       {os.path.abspath(PSTRAIN_HEATMAP_DIR)}")
print(f"Saving final label masks to:    {os.path.abspath(PSTRAIN_LABEL_DIR)}")


all_label_files = os.listdir(TRAIN_LBL_DIR)
label_files = sorted([f for f in all_label_files if f.lower().endswith('.png') and not f.startswith('.')])


# --- Loop through all found label files ---
```

```
processed_count = 0
skipped_count = 0
error_count = 0
img_not_found_count = 0

total_files = len(label_files)
print(f"\nStarting processing for {total_files} label files...")

for i, label_filename in enumerate(label_files):
    img_name_base = os.path.splitext(label_filename)[0] # Get base name without extension
    label_filepath = os.path.join(TRAIN_LBL_DIR, label_filename)

    print(f"\nProcessing label {i+1}/{total_files}: {label_filename} (Base: {img_name_base})")

    # Find the corresponding original image file
    original_img_path = None
    original_img_ext = None
    try:
        found = False
        for img_file in os.listdir(TRAIN_IMG_DIR):
            if os.path.splitext(img_file)[0] == img_name_base:
                original_img_path = os.path.join(TRAIN_IMG_DIR, img_file)
                original_img_ext = os.path.splitext(img_file)[1] # Get extension (e.g., '.jpg')
                print(f"  Found corresponding image: {img_file}")
                found = True
                break
        if not found:
                print(f"  Skipping: Could not find corresponding image file for base name '{img_name_base}' in {TRAIN_
                img_not_found_count += 1
                skipped_count += 1
                continue
    except Exception as e:
        print(f"!!! ERROR searching for image file for {label_filename}: {e}")
        error_count += 1
        continue


    try:
        # Load the label mask file
        label_tensor_loaded = read_image(label_filepath)

        # Handle channel issues (ensure single channel)
        if label_tensor_loaded.shape[0] != 1:
            if label_tensor_loaded.shape[0] == 3:
                    label_tensor_loaded = label_tensor_loaded[0:1, :, :]
                    print(f"  Info: Label had 3 channels, took the first.")
            else:
                print(f"  Skipping: Label has unexpected shape {label_tensor_loaded.shape}, expected (1, H, W).")
                skipped_count += 1
                continue

        # Apply the first remap (255 -> 3)
        label_tensor_original = target_remap(label_tensor_loaded)

        # Process the Loaded Mask ONCE per sample
        label_squeezed = label_tensor_original.squeeze(0)
        mask_post_remap1 = label_squeezed.numpy().astype(np.uint8)
        mask_size = mask_post_remap1.shape

        # Apply the SECOND remapping (Swap 3->0, Add 1) -> Final classes 1, 2, 3
        mask_swapped = mask_post_remap1.copy()
        mask_swapped[mask_post_remap1 == 3] = 0
        remapped_mask_final = mask_swapped + 1
        final_present_classes = np.unique(remapped_mask_final)
        final_target_classes = final_present_classes[final_present_classes > 0]

        # Check if finding two distinct classes is possible
```

```python
        if len(final_target_classes) < 2:
            print(f"  Skipping: Mask only contains {len(final_target_classes)} target class(es) {final_target_classes.
            skipped_count += 1
            continue

        # Loop to find TWO distinct class selections for this sample
        selected_results = [] # List to store (selected_class, final_mask_array, heatmap_array)
        attempts = 0
        found_classes = set()

        while len(selected_results) < 2 and attempts < MAX_ATTEMPTS:
            attempts += 1
            heatmap, center_coords = create_gaussian_heatmap(size=mask_size, sigma=HEATMAP_SIGMA)
            current_selected_class, _ = select_dominant_class(heatmap, remapped_mask_final)

            if current_selected_class > 0 and current_selected_class not in found_classes:
                final_mask = np.zeros_like(remapped_mask_final, dtype=np.uint8)
                final_mask[remapped_mask_final == current_selected_class] = current_selected_class
                selected_results.append((current_selected_class, final_mask, heatmap))
                found_classes.add(current_selected_class)
                print(f"    Attempt {attempts}: Found distinct class {current_selected_class} at {center_coords}")


        if len(selected_results) == 2:
            print(f"  Successfully found two distinct classes.")
            sel_cls_1, fin_msk_1, heatmap_1 = selected_results[0]
            sel_cls_2, fin_msk_2, heatmap_2 = selected_results[1]

            # --- Define final output filenames (consistent naming) ---
            output_base_name_1 = f"{img_name_base}_1"
            output_base_name_2 = f"{img_name_base}_2"

            # Paths for triplet 1
            output_img1_path = os.path.join(PSTRAIN_IMG_DIR, f"{output_base_name_1}{original_img_ext}")
            output_heatmap1_path = os.path.join(PSTRAIN_HEATMAP_DIR, f"{output_base_name_1}.png")
            output_label1_path = os.path.join(PSTRAIN_LABEL_DIR, f"{output_base_name_1}.png")

            # Paths for triplet 2
            output_img2_path = os.path.join(PSTRAIN_IMG_DIR, f"{output_base_name_2}{original_img_ext}")
            output_heatmap2_path = os.path.join(PSTRAIN_HEATMAP_DIR, f"{output_base_name_2}.png")
            output_label2_path = os.path.join(PSTRAIN_LABEL_DIR, f"{output_base_name_2}.png")


            # Copy the original image twice
            shutil.copy2(original_img_path, output_img1_path)
            shutil.copy2(original_img_path, output_img2_path)
            print(f"    Copied original image to: {os.path.basename(output_img1_path)}")
            print(f"    Copied original image to: {os.path.basename(output_img2_path)}")

            # Save Heatmaps as PNG Images (Scaled 0-255)
            heatmap1_scaled = (heatmap_1 * 255).astype(np.uint8)
            heatmap2_scaled = (heatmap_2 * 255).astype(np.uint8)
            Image.fromarray(heatmap1_scaled, mode='L').save(output_heatmap1_path) # 'L' mode for grayscale
            Image.fromarray(heatmap2_scaled, mode='L').save(output_heatmap2_path)
            print(f"    Saved Heatmap 1: {os.path.basename(output_heatmap1_path)}")
            print(f"    Saved Heatmap 2: {os.path.basename(output_heatmap2_path)}")


            # Save Final Masks as PNG Images (already uint8)
            Image.fromarray(fin_msk_1).save(output_label1_path)
            Image.fromarray(fin_msk_2).save(output_label2_path)
            print(f"    Saved Label 1:   {os.path.basename(output_label1_path)}")
            print(f"    Saved Label 2:   {os.path.basename(output_label2_path)}")


            processed_count += 1 # Count original files that yielded 2 outputs
        else:
```

```python
            print(f"  Skipping: Failed to find two distinct classes within {MAX_ATTEMPTS} attempts.")
            skipped_count += 1

    except FileNotFoundError:
            print(f"!!! ERROR processing label file {label_filename}: File not found (unexpected).")
            error_count += 1
    except Exception as e:
        print(f"!!! ERROR processing label file {label_filename}: {e}")
        import traceback
        traceback.print_exc()
        error_count += 1
        # Continue to the next file


# --- Final Summary ---
end_time = time.time()
total_time = end_time - start_time
print("\n" + "="*40)
print("Processing Complete.")
print(f"Output base directory:                         {os.path.abspath(PSTRAIN_BASE_DIR)}")
print("-" * 40)
print(f"Total label files found:                       {total_files}")
print(f"Successfully processed (2 triplets generated): {processed_count}")
print(f"Skipped (due to various reasons):              {skipped_count}")
print(f"  - Skipped because original image not found:  {img_not_found_count}")
print(f"  - Skipped (other reasons, e.g., too few classes): {skipped_count - img_not_found_count}")
print(f"Errors during processing:                      {error_count}")
print("-" * 40)
print(f"Total files created in '{os.path.basename(PSTRAIN_IMG_DIR)}':      {processed_count * 2}")
print(f"Total files created in '{os.path.basename(PSTRAIN_HEATMAP_DIR)}':  {processed_count * 2}")
print(f"Total files created in '{os.path.basename(PSTRAIN_LABEL_DIR)}':    {processed_count * 2}")
print("-" * 40)
print(f"Total time: {total_time:.2f} seconds")
print("="*40)
```

UNet training (unet.ipynb)

```python
import torch
from torch import nn
from torch import Tensor
from torch import optim
from torch.utils.data import DataLoader
from utils.training import start
from utils.MetricsHistory import MetricsHistory
from unet.unet import unet
from utils.weighted_loss import WeightedDiceCELoss
from utils.utils import calculate_class_weights
from utils.dataset import dataset, target_remap, diff_size_collate

EVAL_IGNORE_INDEX = 3
TRAIN_IGNORE_INDEX = None
NUM_CLASSES = 4
MODEL_NAME = "tmp.pytorch"
MODEL_SAVE_DIR = "tmp"
LOAD = False
SAVE = False
EPOCHS = 100
WEIGHT_DECAY = 0.01
TARGET_SIZE = 256

# Determine the device to use (GPU if available, otherwise CPU)
if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
```

```
else:
    device = torch.device("cpu")

target_batch_size = 64
batch_size = 2

# Create datasets for training, validation, and testing
training_data = dataset("datasets/astrain/color", "datasets/astrain/label", target_transform=target_remap())
val_data = dataset("datasets/Val/color", "datasets/Val/label", target_transform=target_remap())
test_data = dataset("datasets/Test/color", "datasets/Test/label", target_transform=target_remap())

# Create data loaders for training, validation, and testing
train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True, pin_memory=True)
val_dataloader = DataLoader(val_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_colla
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_col


# Class Weights
class_weight = Tensor([0.30711034803008996, 1.5412496145750956, 1.8445296893647247, 0.30711034803008996])
class_weight = Tensor([0.2046795970925636, 1.0271954434416883, 1.2293222812780409, 1.5388026781877073])
# class_weight = [1, 1, 1, 1]
# class_weight = calculate_class_weights(training_data, 4, None, "dataset")
class_weight = class_weight.to(device)
class_weight = None

# Calculate the number of accumulation steps
accumulation_steps = target_batch_size // batch_size

# Model
model = unet(3, 4).to(device)

# Losses
train_loss_fn = WeightedDiceCELoss(ignore_index=TRAIN_IGNORE_INDEX, smooth_dice=1, class_weights=class_weight)
val_loss_fn = WeightedDiceCELoss(ignore_index=EVAL_IGNORE_INDEX, class_weights=class_weight)

train_loss_fn = nn.CrossEntropyLoss(weight=class_weight)
val_loss_fn = nn.CrossEntropyLoss(weight=class_weight, ignore_index=EVAL_IGNORE_INDEX)

train_loss_fn = nn.CrossEntropyLoss()
val_loss_fn = nn.CrossEntropyLoss(ignore_index=EVAL_IGNORE_INDEX)

# Optimizer
optimizer = optim.AdamW(model.parameters(), weight_decay=WEIGHT_DECAY)

# Scheduler
scheduler = None

# Metric History
agg = MetricsHistory(NUM_CLASSES, EVAL_IGNORE_INDEX)

# Training Pipiline
start(
    model_save_dir=MODEL_SAVE_DIR,
    model_save_name=MODEL_NAME,
    model=model,
    optimizer=optimizer,
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    accumulation_steps=accumulation_steps,
    device=device,
    train_loss_fn=train_loss_fn,
    val_loss_fn=val_loss_fn,
    scheduler=scheduler,
    agg=agg,
    load=LOAD,
    save=SAVE,
    num_classes=NUM_CLASSES,
```

```
    ignore_index=EVAL_IGNORE_INDEX,
    target_size=TARGET_SIZE
)
```

Autoencoder training (autoencoder.ipynb)

```
from pickle import TRUE
from tqdm import tqdm
import torch
import os
import numpy as np
from torch.utils.data import DataLoader
from torch import nn

from utils.dataset import *
from utils.utils import *
from utils.training import trainReconstruction, evalReconstruction
from autoencoder.autoencoder import ReconstructionAutoencoder


batch_size = 2
target_batch_size = 64
accumulation_steps = target_batch_size // batch_size

training_data = dataset("datasets/astrain/color", "datasets/astrain/label", target_transform=target_remap())
validation_data = dataset("datasets/Val/color", "datasets/Val/label", target_transform=target_remap())
test_data = dataset("datasets/Test/color", "datasets/Test/label", target_transform=target_remap())

train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True, pin_memory=True)
val_dataloader = DataLoader(validation_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_siz
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True, pin_memory=True,collate_fn=diff_size_coll


if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")


model = ReconstructionAutoencoder(din=3, dout=3).to(device)
loss_fn = nn.MSELoss()
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
MODEL_SAVE_DIR = "tmp"
MODEL_NAME = "tmp.pytorch"
start_epoch = 0
recoverCheckpoint = False
TARGET_SIZE = 256

if recoverCheckpoint and os.path.isfile(f"{MODEL_SAVE_DIR}/{MODEL_NAME}"):
    print(f"Loading checkpoint from: {MODEL_SAVE_DIR}/{MODEL_NAME}")
    # Load the checkpoint dictionary; move tensors to the correct device
    checkpoint = torch.load(f"{MODEL_SAVE_DIR}/{MODEL_NAME}", map_location=device, weights_only=False)

    # Load model state
    model.load_state_dict(checkpoint["model_state_dict"])
    print(" -> Model state loaded.")

    # Load optimizer state
    try:
        optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
        print(" -> Optimizer state loaded.")
    except Exception as e:
```

```
        print(f" -> Warning: Could not load optimizer state: {e}. Optimizer will start from scratch.")

    # Load training metadata
    start_epoch = checkpoint.get("epoch", 0) # Load last completed epoch, training continues from next one
    best_val_loss = checkpoint.get("best_val_loss", np.inf)

    print(f" -> Resuming training from epoch {start_epoch + 1}")
    loaded_notes = checkpoint.get("notes", "N/A")
    print(f" -> Notes from checkpoint: {loaded_notes}")

else:
    print(f"Checkpoint file not found at {MODEL_SAVE_DIR}/{MODEL_NAME}. Starting training from scratch.")



best_val_loss = np.inf
EPOCHS = 100
print("\nStarting Training (Autoencoder)...")
for t in range(start_epoch, EPOCHS):
    current_epoch = t + 1
    print(f"Epoch {t+1}\n-------------------------------")
    train_loss = trainReconstruction(train_dataloader, model, loss_fn, optimizer, accumulation_steps)

    wrong_val_loss, correct_val_loss = evalReconstruction(val_dataloader, model, loss_fn, target_size=TARGET_SIZE)

    # Save model based on validation val loss improvement
    if correct_val_loss < best_val_loss:
        print(f"Validation loss improved ({best_val_loss:.6f} → {correct_val_loss:.6f}). Saving model...")
        best_val_loss = correct_val_loss # Save corresponding loss
        checkpoint_path = os.path.join(MODEL_SAVE_DIR, f"{MODEL_NAME}") # Changed name
        checkpoint = {
            "epoch": t + 1,
            "model_state_dict": model.state_dict(),
            "optimizer_state_dict": optimizer.state_dict(),
            "best_val_loss": best_val_loss,
        }
        torch.save(checkpoint, checkpoint_path)

    else:
        print(f"Corresponding validation loss: {correct_val_loss:.6f} not better than {best_val_loss}")

    #print(f"Wrong Validation loss: {wrong_val_loss:.6f}")
    print(f"Train loss: {train_loss:.6f}")

    # PLot a training image reconstruction
    img, label = training_data[0]
    img = img.to(device)
    res = model(img.unsqueeze(0))
    plt.imshow(res[0].permute(1,2,0).cpu().detach().numpy())
    plt.savefig(f"drive/MyDrive/autoencoder/images/test{t}.png", format="png")
    plt.show()

print("\n--- Training Finished! ---")
print(f"Best model saved to: {os.path.join(MODEL_SAVE_DIR, f'{MODEL_NAME}')}")


import torch
from torch import Tensor
from torch import optim
from torch.utils.data import DataLoader
from utils.training import start
from utils.MetricsHistory import MetricsHistory
from utils.weighted_loss import WeightedDiceCELoss
from utils.utils import calculate_class_weights
from utils.dataset import dataset, target_remap, diff_size_collate
from autoencoder.autoencoder import SegmentationAutoencoder

```

```python
EVAL_IGNORE_INDEX = 3
TRAIN_IGNORE_INDEX = None
NUM_CLASSES = 4
MODEL_NAME = "tmp.pytorch"
MODEL_SAVE_DIR = "tmp"
LOAD = False
SAVE = False
EPOCHS = 100
WEIGHT_DECAY = 0.01
TARGET_SIZE = 256

if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

target_batch_size = 64
batch_size = 64

# With Augmentation
training_data = dataset("datasets/astrain/color", "datasets/astrain/label", target_transform=target_remap())
val_data = dataset("datasets/Val/color", "datasets/Val/label", target_transform=target_remap())
test_data = dataset("datasets/Test/color", "datasets/Test/label", target_transform=target_remap())

train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True, pin_memory=True)
val_dataloader = DataLoader(val_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_colla
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_col


# Class Weights
class_weight = Tensor([0.33265044664009075, 1.669423957743164, 1.9979255956167454, 0.0])
class_weight = Tensor([0.30711034803008996, 1.5412496145750956, 1.8445296893647247, 0.30711034803008996])
class_weight = Tensor([0.2046795970925636, 1.0271954434416883, 1.2293222812780409, 1.5388026781877073])
# class_weight = [1, 1, 1, 1]
# class_weight = calculate_class_weights(training_data, 4, None, "dataset")
class_weight = class_weight.to(device)

accumulation_steps = target_batch_size // batch_size

# Model
# pretrained_encoder_path = "/content/drive/MyDrive/autoencoder/256_with_aug_LR1e-3/checkpoint_256_with_aug_TargetSize
# model = SegmentationAutoencoder(3, 4, pretrained_encoder_path).to(device)
model = SegmentationAutoencoder(3, 4).to(device)

# Loses
train_loss_fn = WeightedDiceCELoss(ignore_index=TRAIN_IGNORE_INDEX, smooth_dice=1, class_weights=class_weight)
val_loss_fn = WeightedDiceCELoss(ignore_index=EVAL_IGNORE_INDEX, class_weights=class_weight)

# train_loss_fn = nn.CrossEntropyLoss(weight=class_weight)
# val_loss_fn = nn.CrossEntropyLoss(weight=class_weight, ignore_index=EVAL_IGNORE_INDEX)

# train_loss_fn = nn.CrossEntropyLoss()
# val_loss_fn = nn.CrossEntropyLoss(ignore_index=EVAL_IGNORE_INDEX)

# Optimizer
optimizer = optim.AdamW(model.parameters(), weight_decay=WEIGHT_DECAY)

# Scheduler
scheduler = None

# Metric History
agg = MetricsHistory(NUM_CLASSES, EVAL_IGNORE_INDEX)

# Training Pipiline
start(
```

```
    model_save_dir=MODEL_SAVE_DIR,
    model_save_name=MODEL_NAME,
    model=model,
    optimizer=optimizer,
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    accumulation_steps=accumulation_steps,
    device=device,
    train_loss_fn=train_loss_fn,
    val_loss_fn=val_loss_fn,
    scheduler=scheduler,
    agg=agg,
    load=LOAD,
    save=SAVE,
    num_classes=NUM_CLASSES,
    ignore_index=EVAL_IGNORE_INDEX,
    target_size=TARGET_SIZE
)
```

Clip training (clip.ipynb)

```
import torch
from torch import Tensor
from torch import optim
from torch.utils.data import DataLoader
from utils.training import start
from utils.MetricsHistory import MetricsHistory
from clip.clipunet import ClipUNet
from clip.clipunet_noskips import ClipUNetNoSkips
from utils.weighted_loss import WeightedDiceCELoss
from utils.utils import calculate_class_weights
from utils.dataset import dataset, target_remap, diff_size_collate

EVAL_IGNORE_INDEX = 3
TRAIN_IGNORE_INDEX = None
NUM_CLASSES = 4
MODEL_NAME = "tmp.pytorch"
MODEL_SAVE_DIR = "tmp"
LOAD = False
SAVE = False
EPOCHS = 100
WEIGHT_DECAY = 0.01
PRETRAINED_MODEL_NAME = "openai/clip-vit-base-patch16"
TARGET_SIZE = 224
SKIP_LAYER_INDICES = [3, 5, 7, 9]

if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

# Class Weights
class_weight = Tensor([0.33265044664009075, 1.669423957743164, 1.9979255956167454, 0.0])
class_weight = Tensor([0.30711034803008996, 1.5412496145750956, 1.8445296893647247, 0.30711034803008996])
class_weight = Tensor([0.2046795970925636, 1.0271954434416883, 1.2293222812780409, 1.5388026781877073])
# class_weight = [1, 1, 1, 1]
# class_weight = calculate_class_weights_v3(training_data, 4, None, "dataset")
class_weight = class_weight.to(device)

target_batch_size = 64
batch_size = 2

training_data = dataset("datasets/rstrain/color", "datasets/rstrain/label", target_transform=target_remap())
```

```
val_data = dataset("datasets/Val/color", "datasets/Val/label", target_transform=target_remap())
test_data = dataset("datasets/Test/color", "datasets/Test/label", target_transform=target_remap())

train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True, pin_memory=True)
val_dataloader = DataLoader(val_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_colla
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_col

accumulation_steps = target_batch_size // batch_size

# Model
model = ClipUNet().to(device)

# Loses
train_loss_fn = WeightedDiceCELoss(ignore_index=TRAIN_IGNORE_INDEX, smooth_dice=1, class_weights=class_weight)
val_loss_fn = WeightedDiceCELoss(ignore_index=EVAL_IGNORE_INDEX, class_weights=class_weight)

# Optimizer
optimizer = optim.AdamW(model.parameters(), weight_decay=WEIGHT_DECAY)

# Scheduler
scheduler = None

# Metric History
agg = MetricsHistory(NUM_CLASSES, EVAL_IGNORE_INDEX)

# Training Pipiline
start(
    model_save_dir=MODEL_SAVE_DIR,
    model_save_name=MODEL_NAME,
    model=model,
    optimizer=optimizer,
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    accumulation_steps=accumulation_steps,
    device=device,
    train_loss_fn=train_loss_fn,
    val_loss_fn=val_loss_fn,
    scheduler=scheduler,
    agg=agg,
    load=LOAD,
    save=SAVE,
    num_classes=NUM_CLASSES,
    ignore_index=EVAL_IGNORE_INDEX,
    target_size=TARGET_SIZE
)
```

Prompt based model training (prompt.ipynb)

```
import torch
from torch import Tensor
from torch import optim
from torch.utils.data import DataLoader
from prompt_based.prompt import PromptModel
from utils.MetricsHistory import MetricsHistory
from utils.weighted_loss import WeightedDiceNLLLoss
from utils.utils import calculate_class_weights
from utils.dataset import promptDataset, diff_size_collate
from utils.training import start_prompt

EVAL_IGNORE_INDEX = 3
TRAIN_IGNORE_INDEX = None
NUM_CLASSES = 4
MODEL_NAME = "tmp.pytorch"
MODEL_SAVE_DIR = "tmp"
LOAD = False
```

```
SAVE = False
EPOCHS = 100
WEIGHT_DECAY = 0.01
PRETRAINED_MODEL_NAME = "openai/clip-vit-base-patch16"
TARGET_SIZE = 224
SKIP_LAYER_INDICES = [3, 5, 7, 9]
CLIP_PATH="/content/drive/MyDrive/clip/runs/clip_256_ce_dice_full_weight_fix_train_eval.pytorch"

if torch.backends.mps.is_available():
    device = torch.device("mps")
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

# Class Weights
class_weight = Tensor([0.33265044664009075, 1.669423957743164, 1.9979255956167454, 0.0])
class_weight = Tensor([0.30711034803008996, 1.5412496145750956, 1.8445296893647247, 0.30711034803008996])
class_weight = Tensor([0.2046795970925636, 1.0271954434416883, 1.2293222812780409, 1.5388026781877073])
class_weight = Tensor([1, 1, 1, 1])
# class_weight = calculate_class_weights_v3(training_data, 4, None, "dataset")
class_weight = class_weight.to(device)

target_batch_size = 64
batch_size = 2

training_data = promptDataset("datasets/pstrain/color", "datasets/pstrain/point_prompt", "datasets/pstrain/label")
val_data = promptDataset("datasets/psVal/color", "datasets/psVal/point_prompt", "datasets/psVal/label")
test_data = promptDataset("datasets/psTest/color", "datasets/psTest/point_prompt", "datasets/psTest/label")

train_dataloader = DataLoader(training_data, batch_size=batch_size, shuffle=True, pin_memory=True)
val_dataloader = DataLoader(val_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_colla
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True, pin_memory=True, collate_fn=diff_size_col

accumulation_steps = target_batch_size // batch_size

# Model
# model = PromptModel(CLIP_PATH).to(device) # oad pretrained clip
model = PromptModel().to(device)

# Loses
stable_log = lambda x: torch.log(x + 1e-9)
train_loss_fn = WeightedDiceNLLLoss(ignore_index=TRAIN_IGNORE_INDEX, smooth_dice=1, class_weights=class_weight, apply_
val_loss_fn = WeightedDiceNLLLoss(ignore_index=EVAL_IGNORE_INDEX, class_weights=class_weight, apply_softmax=False, nll

# Optimizer
optimizer = optim.AdamW(model.parameters(), weight_decay=WEIGHT_DECAY)

# Scheduler
scheduler = None

# Metric History
agg = MetricsHistory(NUM_CLASSES, EVAL_IGNORE_INDEX)

# Training Pipiline
start_prompt(
    model_save_dir=MODEL_SAVE_DIR,
    model_save_name=MODEL_NAME,
    model=model,
    optimizer=optimizer,
    train_dataloader=train_dataloader,
    val_dataloader=val_dataloader,
    accumulation_steps=accumulation_steps,
    device=device,
    train_loss_fn=train_loss_fn,
    val_loss_fn=val_loss_fn,
    scheduler=scheduler,
```

```
    agg=agg,
    load=LOAD,
    save=SAVE,
    num_classes=NUM_CLASSES,
    ignore_index=EVAL_IGNORE_INDEX,
    target_size=TARGET_SIZE
)
```