School of Informatics

INFR11212

Computer Vision coursework
Image segmentation in pets classes

s2677266 s2748897

April 2025

Professor: Laura Sevilla-Lara

1 Introduction

Image segmentation is a fundamental task in Computer Vision (CV) that involves partitioning an image into semantically meaningful regions. This consists in assigning labels to individual pixels, distinguishing between entire visualised objects, or combining both objectives (i.e., semantic, instance, and panoptic segmentation, respectively). Specifically, semantic segmentation performs granular-level classification with a set of object categories for all the relevant image's pixels [1], aiming to simplify image representation under a structured informational schema [2]. This process is foundational in many CV pipelines, enriching data for higher-level tasks like object recognition, scene reconstruction, and medical analysis [3].

Advancements in **Machine Learning** (ML) have greatly improved segmentation performance; being able to learn hierarchical representations directly from data with no handcrafted features defined, Deep Learning (DL) models in particular leverage large-scale datasets and computational power to achieve superior accuracy and generalization [4]. This predominance clearly arises from the ability of Convolutional Neural Networks (CNNs) and transformer-based methods to capture complex spatial dependencies and semantic relationships within images [5]. Additionally, a successful strategy for Neural Network (NN) classificators in this field is the use of pre-trained feature-extraction architectures, which help to leverage representational knowledge from the relevant dataset, or even from vast collections of images depicting objects beyond task's focus. By providing robust embeddings — eventually tailored by fine-tuning — the segmentation performance is thus bolstered even when the available labelled data is scarce, the necessary training time is reduced, and generalization enhanced [6].

A leading architecture is the **UNet**, a fully convolutional network originally designed for biomedical segmentation. It employs an encoder-decoder structure: the contracting path captures spatial hierarchies through convolution and pooling, while the expanding one upsamples predictions and merge features via skip connections, preserving detail for pixel-level classification [7].

Similarly, **AutoEncoder** (AE) architectures consist of an encoder, that compresses input into a latent space, and a decoder; however, in this case, the de-contracting path tries to reconstruct the original input directly, instead of predicting any sort of classification already. AEs are unsupervised learning framework for feature extraction objectives: by pre-training such an architecture to minimize the input-recreation loss over a specific dataset, the embeddings flowing through the imposed bottleneck can work as wise summaries in the desired dimensionality of the instances [8, 9]. Variational or Denoising AEs produce robust representations too, modelling latent spaces as distributions for data augmentation and anomaly detection, or learning clean reconstruction from deliberately corrupted input to increase robustness to perturbations [10, 11].

Contrastive Language-Image Pretraining (**CLIP**) learns joint visual-text representations from large-scale image-text pairs, aligning images with descriptions rather than relying solely on pixel-level labels [12]. CLIP-based features generalize well across datasets, enabling zero- or few-shot segmentation and improving boundary detection through semantically rich embeddings [13].

Working in parallel with other techniques, **prompt-based segmentation** introduces minimal user input (e.g., points, boxes or scribbles) to guide the model in identifying specific regions [14]. Improving efficiency and adaptability thanks to user-provided cues which dynamically refine the output, this approach is widely adopted in strategies like the Segment Anything Model [15].

1.1 Objective

The aim of the present paper is to experiment disparate image segmentation techniques on a properly pre-processed and augmented images dataset (Sec. 2), exploring both traditional and modern DL approaches to this task. Specifically, the importance of pre-training and feature-

extraction strategies is investigated, comparing the performance of a UNet-based end-to-end NN (Sec. 3.1.1) to the one of Autoencoder-based (Sec. 3.1.2) and CLIP-bolstered (Sec. 3.1.3) architectures. The latter well-performing model is also experimented in a prompt-based version (Sec. 3.1.4), enriching the input format through a systematic annotation of the images with an informative rationale (Sec. 2.3). Evaluating their test performance and robustness to input perturbations (Sec. 3.3), these methods' outcomes are then comparatively discussed (Sec. 4).

2 Data

A processed version of The Oxford-IIIT Pet Dataset [16] is given, already split into training and test subsets: a total of 3680 instances are for learning purpose, while other 3710 stay unseen for subsequent evaluation. Of the former set, a 20% is kept for validation purposes, while the remaining part is directly used for models optimization. Each image depicts a dog or a cat — rarely both, but with one of them only annotated — and comes associated with a ground-truth masks, indicating the pixels pertaining to each category (Appx. A.1.2). In particular, 4 classes are differentiated with these labels: cat, dog, animal's outline, and background. A total of 37 breeds appear, each with about 100 instances both in the *Train/Val* and *Test* set, making the latter collection fair for evaluation of models based on the corresponding training set (Appx. A.1.1). Nevertheless, a strong class imbalance is noticeable, as the number of cat breeds is just 12 with respect to 25 dog ones; given the almost equal images population for each breed uniform across the two sets, both have less than $\frac{1}{3}$ of cat instances (Tab. 1).

	Train/Val	**Test**	**Breeds**
Cats	1188	1212	12
Dogs	2492	2498	25
Total	3680	3710	37

Table 1: Datasets' classes counts and number of breeds per species

2.1 Resizing

The images and respective masks have different sizes, but in the *Train/Val* set most of their dimensions are near to 500x500 pixels, with a distribution that highly suggests this as a sensible resizing shape (Fig. 1), entailing minimal information loss [17]. For this reason, all the instances — ground-truth labels included — are initially converted to the aforementioned dimensions, which is useful to operate the subsequent preprocessing steps onto standardized tensors.

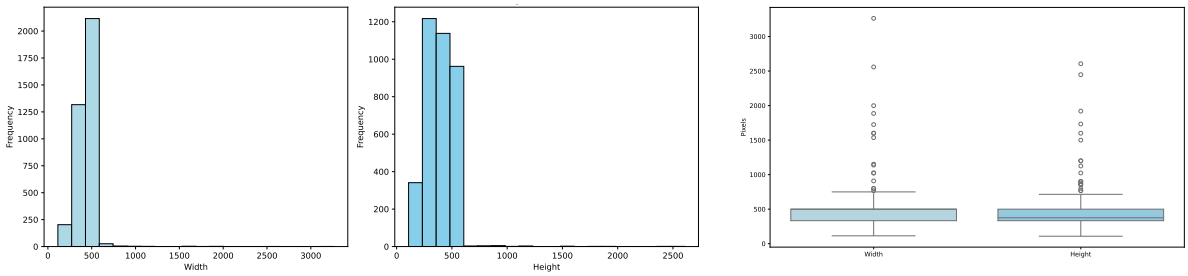


Figure 1: Images dimensions distribution in the training and validation set

2.2 Augmentation

Before performing conventional data augmentation, it is essential to address the issue of class imbalance: a significant asymmetry in the training set may cause the optimization process to prioritize the dominant classes, thereby neglecting the minimization of loss associated with predictions of the minority class. In this context, the considerably lower number of cat images in the *Train/Val* set is likely to result in inferior segmentation performance for cat pixels. This is because the model's gradient descent process would be disproportionately influenced by the more prevalent dog images, leading it to better capture the patterns associated with dogs. To mitigate this issue, two primary strategies can be employed. The first involves assigning class-specific weights during loss computation (e.g., assigning a higher weight to cat pixels, potentially

nearly double that of dog pixels) to compensate for the imbalance. The second approach is to rebalance the dataset by augmenting the minority class through derived images [18]. The latter method is adopted, to avoid the need for assigning weights to non-animal classes as well, which could inadvertently interfere with the model's autonomous learning of those categories.

Moreover, due to cats' natural behaviour, cat images frequently show the animal in particular postures and with its snout oriented at various angles relative to the camera and often in agile, upward-paws poses. This characteristic suggests that rotating all original training images by 180° could yield additional informative inputs that align with the natural variability of feline posture. Such an augmentation not only preserves the physiological plausibility of the visual representation — particularly for cats — but also contributes to learning direction-invariant anatomical features. While simple vertical flipping may suffice for augmenting dog images, this rotation-based augmentation may provide more substantial benefits for the model's understanding of cat morphology. Therefore, the resized *Train/Val* images and their corresponding masks are included in the training set. In addition, cat-labeled instances are also added in their 180°-rotated versions. This preliminary augmentation results in a total of 2,376 cat images, effectively **resolving the class imbalance** without introducing any artificial noise.

Importantly, this was considered the only significant imbalance in the dataset. It is assumed that variations in breed within each species have a negligible impact on the overall performance, since the classification task operates at the species level, where physical traits remain consistently distinct and largely independent of breed. Although the segmentation task is based on pixel-level information, no further disequilibriums were identified: visual inspection suggests that, on average, cats and dogs occupy a similar amount of space within their respective images. Regarding other labels, the background class is expected to occupy a larger portion of pixels overall. However, this can be beneficial to Neural Networks, as the visual patterns in background regions are highly varied — spanning natural and indoor scenes depending on where each photo was taken, thus helping the model generalize better. In contrast, the *outline* class contains significantly fewer pixels than the others, but it is excluded from evaluation (Sec. 3.3) and retained in the target masks, being expected to aid the model to segment into well-defined regions. Finally, loss functions were used which account for per-image class imbalance (Sec. 3.1).

Following this setup, the dataset undergoes full **augmentation** procedures to further expand the number of training instances and enhance the model's generalization capabilities. For each image, a new version is generated by probabilistically applying a series of transformations. The corresponding mask is augmented using geometric transformations only, ensuring that the label intensity and structure remain intact.

Geometric transformations on image and mask (application probability of 50%)

- Horizontal Flip: mirrors left-to-right
- Vertical Flip: mirrors top-to-bottom
- Random 90° Rotation: rotates by a random multiple of 90°
- Affine Transformations: applies random scaling, translation, rotation, and shearing
- Elastic Transform: distorts using random elastic deformations
- Grid Distortion: applies a random grid deformation
- Optical Distortion: introduces random lens-like warping effects

Intensity transformations on image only (application probability of 30%)

- Gaussian Blur: applies a Gaussian blur with random kernel size between 3×3 and 7×7
- Motion Blur: simulates motion blur with random kernel size up to 5 pixels
- Random Brightness/Contrast: adjusts brightness and contrast randomly within $\pm 20\%$
- Hue-Saturation-Value Shift: randomly modifies the hue, saturation, and brightness values
- Gaussian Noise: adds Gaussian noise with random variance in the range $[10, 50]$
- Multiplicative Noise: multiplies pixel values by a random factor in the range $[0.9, 1.1]$

2.3 Annotation

Finally, as a modification to the original database, for both the training and test images, one pixel pertaining to the centre of the depicted animal's nose is individuated and its coordinate over the 500x500 representation stored. After a trial with AI-based automatic annotation, producing poor-quality results, the above additional labelling procedure is carried manually, using a script to iteratively visualize one of the resized pictures and saving the position of a mouse click on it. For the really few cases with no nose displayed due to the animal's posture, the spacially nose-nearest visible point is indicated instead. The aim is preparing an additional informationally valid input to train a sensible prompt-enriched model (Sec. 3.1.4). In fact, the chosen pixel is deemed able to individuate the snout area, which is dense of species-characteristic visual features (e.g., muzzle, eyes and ears shape). Pointing this spot to the model would thus not only hint to the model the position of the animal, but also help recognizing felines from canids, through suggesting specifically relevant area [19]. To feed this information layer to the NNs, a one-dimensional heatmap is produced for each resized instance — to be conceived as the images' fourth channel — and eventually augmented geometrically like the respective mask in the *Train/Val* case. This greyscale 500x500 representation is built through a symmetric Gaussian kernel centred in the annotated pixel, represented by maximum brightness, which shades down to black with a sigma-parameter tuned to averagely allow the light area to extend over all the anatomies of interest (Fig. 2).

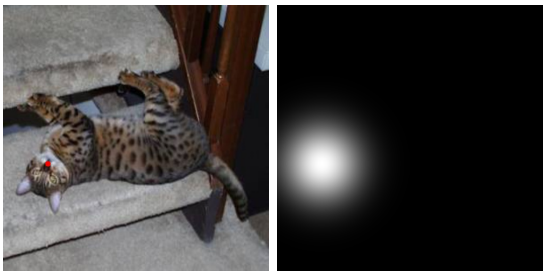


Figure 2: Annotated image and its heatmap

3 Experiments

With the presented background domain understanding (Sec. 1), this paper's specific segmentation task is addressed: the objective is to classify each pixel of the input colour (i.e., JPG-given in 3 channels) images into one of 4 labels, predicting the respective single-channel PNG-represented masks accurately, trough efficient ML-models. The classes are (i) *Background*, (ii) *Cat*, (iii) *Dog*, and (iv) *Outline* of the animal. The original instances used for training are the augmented 500x500 ones, outcome of the described pre-processing (Sec. 2). Anyway, all the models use an input and output dimension of 256x256, due to computational limitations; nevertheless, the test procedures (Sec. 3.3) will be conducted resampling back to the original mask size.

3.1 Models Architectures and Training

For the segmentation task at issue, it is crucial to choose loss functions that not only maximize pixel accuracy but also handle class imbalances and overlapping regions effectively. To achieve this, the following loss types were experimented for NNs' gradient-descent optimization:

Mean Square Error MSE measures the average squared difference between predicted and target values: $\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$, where x_i is the input pixel and $\hat{x}_i$ the reconstructed pixel. It is used solely to pre-train the unsupervised AutoEncoder (Sec. 3.1.2) for learning compressed representations without labels [8].

Cross-entropy Loss Cross Entropy Loss focuses on pixel-wise classification by comparing predicted probabilities with ground truth labels, penalizing incorrect predictions. While effective for accuracy, it struggles with class imbalance, often biasing predictions toward the dominant background class. The loss is given by $\mathcal{L}_{CE} = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$, where C is the number of classes, y_c is the ground truth probability, and $\hat{y}_c$ is the predicted probability for class c.

Dice Loss Dice Loss maximizes overlap between predicted and ground truth masks, making it robust against class imbalance. It prioritizes shape and boundary accuracy, ensuring well-defined object contours [20]. The loss is defined as $\mathcal{L}_{Dice} = 1 - \frac{2\sum p_i g_i}{\sum p_i + \sum g_i}$, where p_i and g_i are the predicted and ground truth values for pixel i, respectively.

Combined Loss Combining both losses balances pixel-wise accuracy with shape preservation: $\mathcal{L}_{Combined} = \alpha\mathcal{L}_{CE} + \beta\mathcal{L}_{Dice}$, where α and β control their contributions. This hybrid approach ensures robust segmentation by mitigating class imbalance, preserving object boundaries [21].

These loss functions guide model training, evaluation, and comparison across architectures.

3.1.1 UNet-based end-to-end Neural Network

The implementation of U-Net is structured for image segmentation tasks, adhering to the classic encoder-decoder architecture to effectively capture both local and global context.

The encoder component comprises four convolutional blocks, each containing two convolutional layers followed by batch normalization and ReLU activation. Additionally, dropout with a rate of 0.2 is incorporated within each block to enhance robustness and mitigate overfitting. Max pooling layers with stride of 2 are employed to reduce dimensions and capture coarse features.

To facilitate efficient learning and ensure stable gradient propagation, batch normalization and ReLU activations are utilized throughout the convolutional blocks. This combination accelerates convergence and enhances the model's ability to learn complex patterns.

The decoder is designed to restore the spatial dimensions of the encoded features through transposed convolutions for upsampling. Each upsampling step is followed by a convolutional block analogous to those in the encoder. To preserve critical spatial details that may be lost during downsampling, skip connections are introduced, directly concatenating feature maps from the encoder to the corresponding decoder layers. This integration of contracting and expansive paths aids in maintaining high-frequency details essential for accurate segmentation.

The final output layer employs a 1×1 convolution to map the learned features to the desired number of output classes, ensuring precise segmentation. By incorporating dropout, batch normalization, and skip connections, the U-Net model achieves a balance between generalization and fine-grained segmentation performance (Appx. A.2.1).

3.1.2 Autoencoder-based Decoder

Pre-Training: Training the Encoder - The autoencoder learns compact image representations by encoding and reconstructing inputs without labels (Appx. A.1.6). It consists of two components: an encoder that compresses the image and a decoder that reconstructs it.

- **Encoder**: A series of convolutional layers (64, 128, 256, 128 features) with max pooling reduce spatial resolution, while LeakyReLU activations capture low- and mid-level features (e.g., edges, textures, shapes). The output is a latent vector encoding image information.
- **Decoder**: Transposed convolutions upsample the latent vector to reconstruct the image. LeakyReLU activations follow each layer, with a final sigmoid output in [0,1]. Reconstruction loss is computed using Mean Squared Error (MSE), encouraging the network to capture meaningful features for downstream tasks like segmentation.

Fine-Tuning: Fixed Encoder and Segmentation Decoder - After pre-training, the encoder is paired with a segmentation decoder for pixel classification.

- **Fixed Encoder**: Its weights remain unchanged to preserve learned representations and prevent overfitting. Acting as feature extractor, it outputs compact image embeddings.
- **Segmentation Decoder**: Built with transposed convolutions and residual blocks, it upsamples latent features and maintains gradient flow via skip connections. The decoder

maps features to per-pixel class predictions, producing segmentation maps with categories like background, cat, dog, or outline.

Some experiments which will be discussed in later sections unfreeze this encoder and thus allow it to learn for some experiments as well (Appx. A.2.2).

3.1.3 CLIP-bolstered Decoder

The implementation of CLIP features for image segmentation leverages the pre-trained ViT-B-32 model to extract high-level semantic representations. These CLIP features are utilized alongside convolutional neural network (CNN)-based spatial features to construct a robust segmentation pipeline. The CLIP features are extracted directly from the input image using the model's image encoder, normalized for consistency, and stored individually in a designated directory.

In the decoder architecture, the extracted 512-dimensional CLIP features are projected to 256 dimensions through a fully connected layer. Simultaneously, the input image is processed by a lightweight CNN encoder comprising two convolutional layers, ReLU activations, and max pooling operations to capture spatial details. The projected CLIP features are then concatenated with the CNN output along channel dimension, integrating semantic and spatial information.

The fused features are subsequently passed through a decoder network composed of transposed convolutions and upsampling layers, which reconstruct the segmentation map at the desired resolution. This combination of CLIP's rich semantic understanding with the spatial precision of CNN-based features enables the model to generate context-aware accurate masks (Appx. A.2.3).

3.1.4 Prompt-enriched CLIP-bolstered Decoder

The proposed prompt-enriched CLIP-based decoder extends the previous model by incorporating additional spatial context through a location-specific heatmap. The primary objective of this architecture is to improve segmentation accuracy by directing the model's attention toward specific anatomical features, specifically the nose of the animal. To achieve this, a heatmap corresponding to the nose location is generated and appended as a fourth channel to the original RGB image, resulting in a unified 4-channel input. These heatmaps were manually annotated to ensure the accuracy and reliability of the training data, as automated nose detection algorithms were found to be insufficiently precise.

As in the previous model, CLIP features are extracted using the pre-trained ViT-B-32 model and normalized to maintain consistency. These features are then concatenated with the augmented image, forming a fused input that integrates CLIP's high-level semantic information with the spatial specificity provided by the heatmap. This combined representation is subsequently processed by the decoder network.

The decoder architecture retains the structure of the prior model, with modifications to the encoder to accommodate the 4-channel input. The network begins by projecting the 512-dimensional CLIP features to 256 dimensions using a fully connected layer. Concurrently, the augmented image (RGB + heatmap) is processed through a CNN encoder consisting of two convolutional layers with ReLU activations and max pooling operations. The projected CLIP features are reshaped and expanded to match the spatial dimensions of the CNN output, after which the two feature sets are concatenated along the channel axis (Appx. A.2.4).

The fused representation is then passed through the decoder module, which consists of transposed convolution layers and upsampling operations to reconstruct the segmentation map. By incorporating location-based heatmaps alongside CLIP's semantic features, the model effectively integrates prompt-based training that aligns more closely with object-specific anatomical structures. This targeted fusion of contextual and semantic information enhances precision.

3.2 User Interface

The best-performing model was integrated into an interactive User Interface (UI), enabling direct user interaction and multiple forms of input (Appx. A.1.4 + demo attached).

For part (a), users are prompted to upload an image and select a point of interest by clicking on it. This interaction generates a heatmap, which is then processed by the prompt-based CLIP model described earlier. The segmentation result is subsequently displayed, allowing users to refine the segmentation process by providing explicit input.

For part (b), additional functionality was implemented to support multiple forms of prompts, including scribbling and bounding box inputs. Users can either draw directly on the uploaded image to guide the model toward specific shapes or define a bounding box around the region of interest. These features provide enhanced flexibility, enabling users to influence the segmentation outcome based on their intended focus.

By integrating point-based selection, freehand annotations, and bounding box inputs, the interface offers an intuitive and interactive experience, significantly improving the model's segmentation capabilities.

3.3 Models Evaluation

The best-performing epochs are chosen by examining the training and validation loss curves (Appx. A.1.5), to ensure accuracy and generalization of the models.

We evaluate our models on three metrics: Intersection over Union (IoU) Score, Dice Score, and Pixel Accuracy.

Intersection over Union (IoU) Score: measures the overlap between the predicted and ground truth segmentation masks, calculated as the ratio of the intersection to the union: $\text{IoU} = \frac{|A \cap B|}{|A \cup B|}$, where A is the predicted segmentation mask and B is the ground truth mask for a given class [22].

Dice Score: evaluates the similarity between the predicted and ground truth segmentation masks. It is defined as $\text{Dice} = \frac{2 \cdot |A \cap B|}{|A| + |B|}$, where A and B are the predicted and ground truth masks, respectively [23].

Pixel Accuracy: metric computes the ratio of correctly predicted pixels to the total number of valid pixels: $\text{Pixel Accuracy} = \frac{\text{Number of correct pixels}}{\text{Total number of valid pixels}}$ [24]

Note that the outline class, was used in the labels while training the models. However, it was omitted for the calculation of the above metrics.

Another point to note is that the segmentation output from the models was resized to match the Ground Truth label size of the image to ensure that there was no loss of information when calculating these scores.

3.3.1 Robustness Exploration

Subsequently, a robustness exploration is run on CLIP and its prompt-enriched counterpart, aiming at uncovering their performance's dependence on images' quality. Incremental levels of the following perturbation types were applied, tuning the specified parameter; the mean Dice score on *Test* set was used as metric, re-calculated at each gradual input worsening, comparing the predicted and gorund-truth masks directly at models' output size to ignore resizing effects.
Perturbations applied on test images (or heatmaps, in the last case)

a) Gaussian Noise: adds Gaussian-distributed noise to each pixel with increasing variance
b) Gaussian Blurring: applies a 3x3 mask iteratively to approximate Gaussian blur
c) Contrast Increase: multiplies pixel values by an increasing factor > 1, clamping at 255

d) Contrast Decrease: multiplies pixel values by a decreasing factor < 1
e) Brightness Increase: adds an incremental offset to each pixel, clamping at 255
f) Brightness Decrease: subtracts an incremental offset from each pixel, clamping at 0
g) Square Occlusion: replaces a random square region of incremental size with black pixels
h) Salt & Pepper Noise: randomly sets pixels to black or white with increasing probability
i) Imprecise Prompt: shifts the heatmap in random direction by increasing number of pixels

4 Results

Model	**Loss function**	**Epoch**	**Per-class IoU**			**Mean IoU**	**Mean Dice**	**Pixel Acc.**(%)
			bkg	**cat**	**dog**			
UNet	Cross-entropy	39	.844	.488	.544	.6255	.5764	86.48
	Dice	45	.876	.498	.628	.6680	.6680	87.78
	C.-E.+Dice	22	.861	.485	.558	.6347	.5829	86.00
Frozen AutoEncoder -based	Cross-entropy	92	.821	.374	.480	.5582	.5233	81.73
	Dice	77	.800	.329	.421	.5168	.5098	76.79
	C.-E.+Dice	97	.821	.369	.485	.5581	.5230	81.06
Fine-tuned AutoEncoder -based	Cross-entropy	54	.849	.458	.557	.6213	.5631	85.41
	Dice	96	.837	.408	.503	.5825	.5438	81.46
	C.-E.+Dice	44	.857	.456	.573	.6287	.5636	85.24
CLIP-bolstered	Cross-entropy	100	.776	.654	.630	.6868	.7880	85.72
	Dice	61	.735	.598	.579	.6374	.7548	78.14
	C.-E.+Dice	14	.774	.649	.628	.6835	.7883	85.65
Prompt-enriched CLIP-b.	Cross-entropy	31	.796	.677	.657	.7098	.8108	87.92
	Dice	87	.744	.627	.610	.6608	.7766	80.56
	C.-E.+Dice	15	.782	.673	.649	.7013	.8032	85.04

Table 2: Segmentation Model Performance Metrics

By considering the all test results obtained (Tab. 2), we notice that the end-to-end approach gained already satisfactory results on the task at issue. In fact, based on properly pre-processed data and leveraging a sufficiently deep architecture, the **UNet** model reaches an IoU score over 0.6, with the background label predicted more accurately than animal ones, and an almost balanced performance on cat and dog classes.

A similar overall performance is noticeable for the fine-tuned **AutoEncoder-based** segmenter, which though present a less negligible gap between the animal classes' IoUs of about 0.1 for any loss function used. In general, this strategy did not outperform UNet models, probably due to its inherently high reliance on the input images' variability balance. In other words, the reconstructive AE might have suffered the cat instances, despite balanced in number trough the preliminary 180°-rotated duplication (Sec. 2.2), were expressing less widely their species-relevant features with respect to dog ones, as the original dataset actually consist by less than $\frac{1}{3}$ of cat images. Ultimately causing the subsequent segmentation model to be fed biased embeddings, this impeded learning is well exemplified by the frozen counterpart of the same model: when the AutoEncoder layers are not fine-tuned for classification, the leveraged latent representations can be not even partially corrected at training time to offset the specific task's labelling asymmetry, resulting in consistently lower scores.

On the other hand, the **CLIP-bolstered** model gains a coherently spread set of per-class IoU, with cats being discriminated even slightly better than dogs; a balance mirrored also by unparalleled Dice scores, despite standard pixel accuracies. Being pre-trained for computer vision purposes on a huge and inclusive set of instances, this feature-encoder manages indeed to provide highly valuable embeddings, which leveraged together with the dataset at issue allow to totally overcome its disequilibrium problem and reach the best performance. Preserving this quality, the mean IoU and Dice grow even more in the prompt-enriched case, demonstrating that the heatmap strategy works in hinting the animal-characteristic regions, making the learning quicker for the cross-entropy training too.

The CLIP-bolstered best-performing models undergo a **robustness exploration** too (Fig, 3), which unveils an higher sensibility for Salt & Pepper Noise in particular, while square occlusion seems to have relatively little influence on the segmentation output. Especially in the latter case, but consistently across all perturbation types, the prompt-aided method shows significantly better performances, with interestingly slower Dice scores reductions deriving from Gaussian Blurring and Brightness Decrease, as also in the case of Gaussian and S&P Noise applications from certain levels on. Furthermore, the heatmap-based model also demonstrated good robustness for small shifts, meaning that segmentation is comprehensibly impaired just when the prompt is clearly imprecise (i.e., indicating outside of the animal at all, rather than simply far from its muzzle).

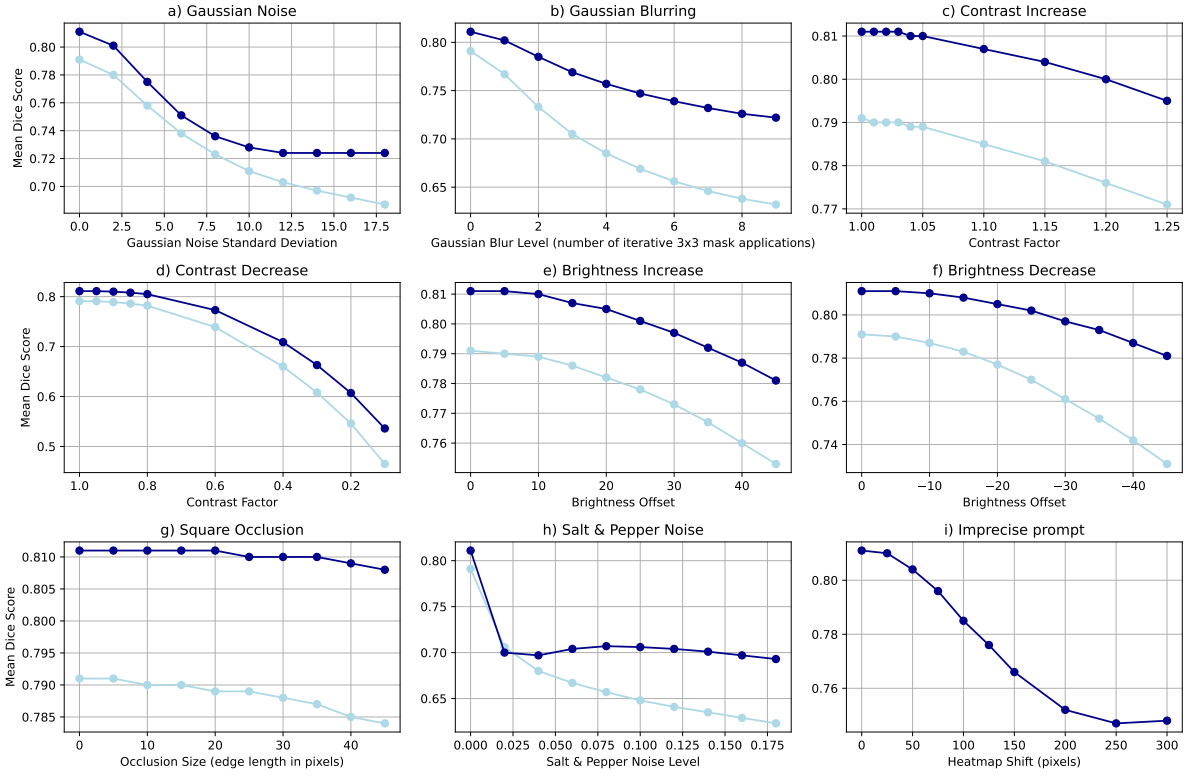


Figure 3: Comparative robustness exploration

5 Conclusion

The experimental objective was reached, individuating ML models which leverage widely pre-trained feature-extractors as better performing on limited datasets, fostering generalization and overcoming label asymmetries in the available instances. The effects of the latter problem appear more difficult to avoid by training embedders on the same dataset, as biased latent rep-

resentations would lead to poorer segmentation; in this case, even a plain end-to-end approach might reach better results.

One of the key challenges addressed was the class imbalance and label asymmetry in the dataset, which contained four distinct pixel-level labels: background, cat, dog, and outline. Models that relied on feature extractors pre-trained on large, diverse datasets (such as CLIP) were more robust to this imbalance, compared to those trained entirely end-to-end on the same small dataset. In contrast, models without such priors tended to develop biased latent representations, leading to poorer segmentation performance. Interestingly, the U-Net model successfully handled the presence of both a cat and a dog in the same image (Appx. A.1.3), accurately segmenting each. This shows the strength of neural networks in segmentation tasks.

The inclusion of the "outline" class, which serves as a boundary between animals and background (or between different animals), proved beneficial: though often predicted slightly thicker than annotated, it helped guide the model in cleanly separating adjacent regions, improving segmentation results.

Overall, Dice loss does not significantly help to reach better accuracy in the long run, probably because it focuses on counterbalancing the image-wise *background*'s higher pixel presence, while this class presents more complex and varied features to learn. However, in co-presence with cross-entropy, Dice ensures faster learning, making the architectures reach comparably satisfactory performances in a lower number of epochs.

In summary, architectures that integrate pre-trained vision-language models such as CLIP, paired with hybrid loss functions, were found to be the most effective for semantic segmentation under low-data conditions. Furthermore, the performance was enhanced by a prompt-based model that extended CLIP with annotated guidance, allowing the network to focus on salient image regions. This approach led to noticeable improvements in segmentation outputs.

5.1 Future work

To solve the class imbalance, having proper computational resources, label weights might be tuned trough a grid search which can precisely offset this disequilibrium at loss computation time. These coefficients could in fact be trial-and-error experimented starting from values inversely proportional to the overall pixel-presence in the dataset; however, they might be tuned to find the best solution also considering the variability of features in each label. For instance, although the background has far more pixels in this case, it is represented by many different types of visual elements compared to the animals' areas, making its patterns harder to learn. A larger number of pixel-instances can aid learning up to a point, so the optimal ratio of background loss-weight to that of an animal class isn't simply expected to follow the inverse of their total pixel count ratio.

Additionally, experimenting with different combinations and relative weightings of Dice loss and cross-entropy loss presents another avenue for optimization. Such hybrid loss functions could allow better control over pixel-level precision and region-level coherence, particularly in boundary and minority-class regions. Furthermore, exploring transformer-based segmentation architectures, such as Segmenter or Mask2Former, may yield performance gains, especially when fine-tuned on tailored augmentations. Lastly, expanding the dataset through synthetic data generation, including techniques like GAN-based image synthesis or diffusion models, could alleviate data scarcity and improve generalization.

A Appendices

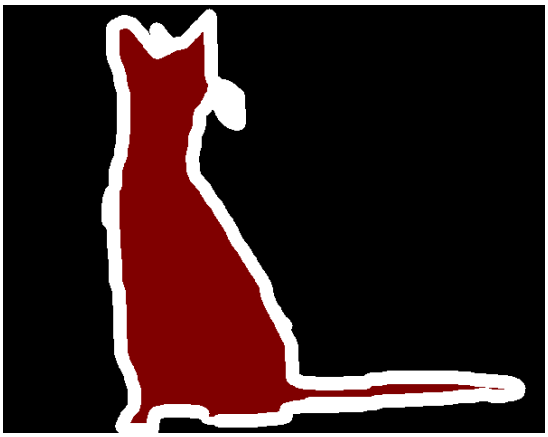
A.1 Additional material

A.1.1 Counts of cats and dogs images by breed in the training/validation and test datasets

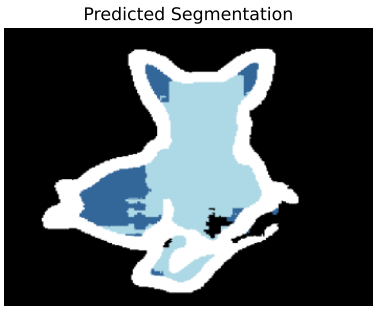
Species counts	Train/Val set	Test set
Cats		
Abyssinian	100	100
Bengal	100	100
Birman	100	100
Bombay	96	104
British Shorthair	100	100
Egyptian Mau	93	107
Maine Coon	100	100
Persian	100	100
Ragdoll	100	100
Russian Blue	100	100
Siamese	99	101
Sphynx	100	100
Total cats	1188	1212
Dogs		
American Bulldog	100	100
American Pit Bull Terrier	100	100
Basset Hound	100	100
Beagle	100	100
Boxer	100	100
Chihuahua	100	100
English Cocker Spaniel	96	104
English Setter	100	100
German Shorthaired	100	100
Great Pyrenees	100	100
Havanese	100	100
Japanese Chin	100	100
Keeshond	100	100
Leonberger	100	100
Miniature Pinscher	100	100
Newfoundland	96	104
Pomeranian	100	100
Pug	100	100
Saint Bernard	100	100
Samoyed	100	100
Scottish Terrier	100	99
Shiba Inu	100	100
Staffordshire Bull Terrier	100	91
Wheaten Terrier	100	100
Yorkshire Terrier	100	100
Total dogs	2492	2498

A.1.2 Examples of cat- and dog-labelled instances

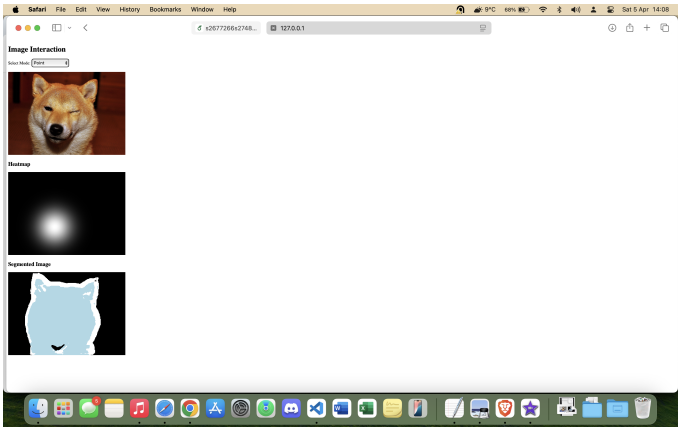


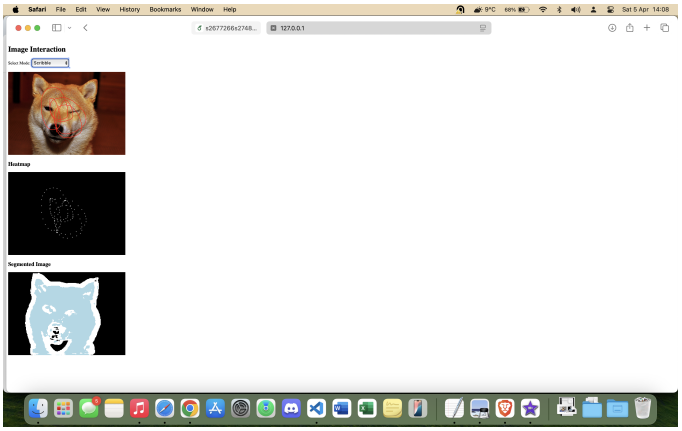



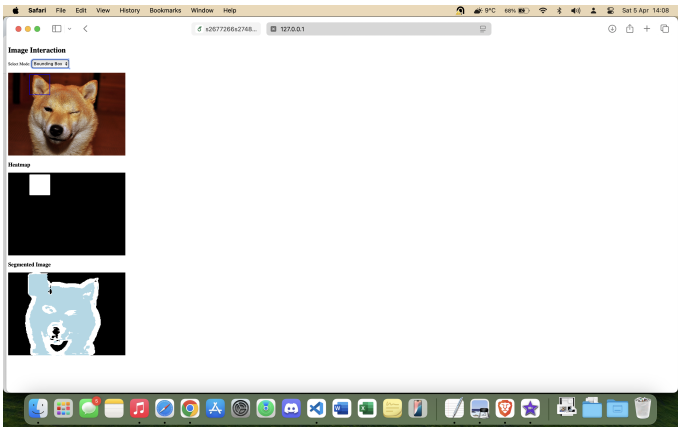
A.1.3 UNet segmentation of an example test image containing both animals



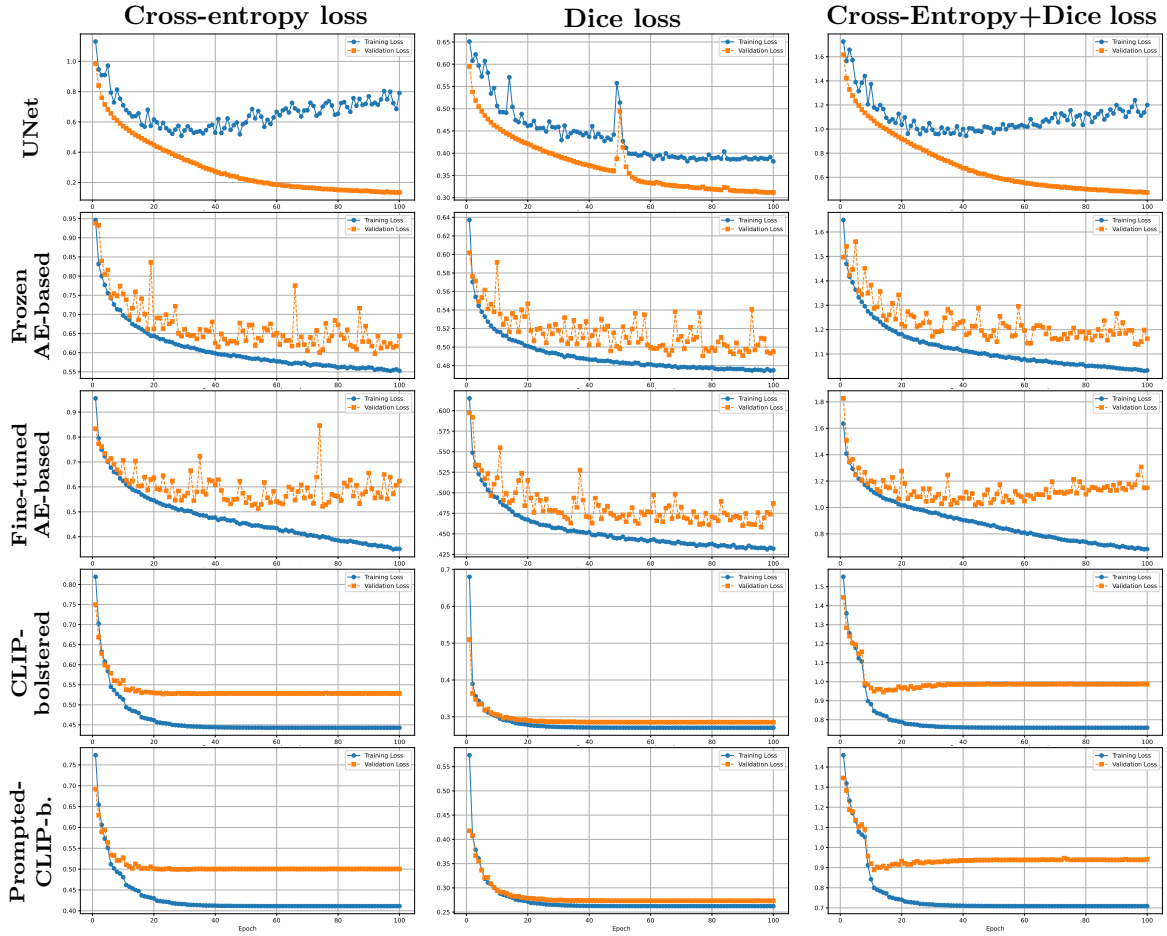
A.1.4 UI screenshots: point- scribble- and box-based examples



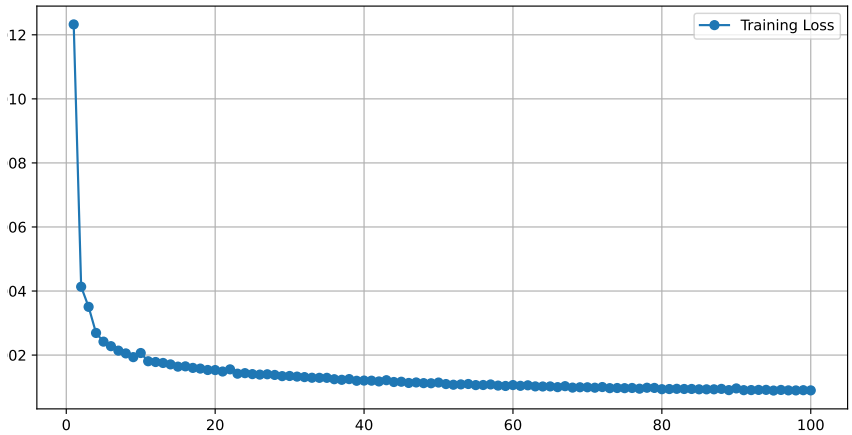




A.1.5 Models learning: *Train* and *Val* loss curves by architecture and loss type



A.1.6 Autoencoder pre-training: MSE reconstruction loss across epochs



A.1.7 Prompt-enriched CLIP's robustness: Dice scores for significant perturbation levels

Transformation	(Parameter, Score)
a) Gaussian Noise	(0, 0.811), (4, 0.775), (8, 0.736), (12, 0.724), (18, 0.724)
b) Gaussian Blurring	(0, 0.811), (3, 0.769), (6, 0.739), (9, 0.722)
c) Contrast Increase	(1.00, 0.811), (1.02, 0.811), (1.05, 0.810), (1.15, 0.804), (1.25, 0.795)
d) Contrast Decrease	(1.00, 0.811), (0.90, 0.810), (0.80, 0.805), (0.40, 0.709), (0.10, 0.536)
e) Brightness Increase	(0, 0.811), (10, 0.810), (25, 0.801), (40, 0.787), (45, 0.781)
f) Brightness Decrease	(0, 0.811), (-10, 0.810), (-25, 0.802), (-40, 0.787), (-45, 0.781)
g) Square Occlusion	(0, 0.811), (15, 0.811), (30, 0.810), (45, 0.808)
(h) Salt & Pepper Noise	(0.00, 0.811), (0.04, 0.697), (0.10, 0.706), (0.18, 0.693)
(i) Imprecise Prompt	(0, 0.811), (50, 0.804), (100, 0.785), (200, 0.752), (300, 0.748)

A.2 Model Architectures

A.2.1 U-Net

```
UNet(
  - Encoder:
    - enc1: ConvBlock(3 -> 64)
    - pool1: MaxPool2d(2)
    - enc2: ConvBlock(64 -> 128)
    - pool2: MaxPool2d(2)
    - enc3: ConvBlock(128 -> 256)
    - pool3: MaxPool2d(2)
    - enc4: ConvBlock(256 -> 512)

  - Decoder:
    - up5: ConvTranspose2d(512 -> 256)
    - dec5: ConvBlock(512 -> 256)
    - up6: ConvTranspose2d(256 -> 128)
    - dec6: ConvBlock(256 -> 128)
    - up7: ConvTranspose2d(128 -> 64)
    - dec7: ConvBlock(128 -> 64)

  - Output:
    - final_conv: Conv2d(64 -> num_classes, kernel_size=1)
)

ConvBlock(in -> out) consists of:
  - Conv2d(in, out, kernel=3, padding=1)
  - BatchNorm2d(out)
  - ReLU
  - Dropout(0.2)
  - Conv2d(out, out, kernel=3, padding=1)
  - BatchNorm2d(out)
  - ReLU
  - Dropout(0.2)
```

A.2.2 Autoencoder

Pre-Training of Encoder Architecture:

```
Autoencoder(
  - Encoder:
    - Conv2d(3 -> 64, kernel=3, stride=1, padding=1)
    - LeakyReLU(0.2)
    - MaxPool2d(kernel=2, stride=2)
    - Conv2d(64 -> 128, kernel=3, stride=1, padding=1)
    - LeakyReLU(0.2)
    - MaxPool2d(kernel=2, stride=2)
    - Conv2d(128 -> 256, kernel=3, stride=1, padding=1)
    - LeakyReLU(0.2)
```

```
    - MaxPool2d(kernel=2, stride=2)
    - Conv2d(256 -> 128, kernel=3, stride=1, padding=1)
    - LeakyReLU(0.2)

  - Decoder:
    - ConvTranspose2d(128 -> 256, kernel=3, stride=2, padding=1, output_padding=1)
    - LeakyReLU(0.2)
    - ConvTranspose2d(256 -> 128, kernel=3, stride=2, padding=1, output_padding=1)
    - LeakyReLU(0.2)
    - ConvTranspose2d(128 -> 64, kernel=3, stride=2, padding=1, output_padding=1)
    - LeakyReLU(0.2)
    - Conv2d(64 -> 64, kernel=3, stride=1, padding=1)
    - LeakyReLU(0.2)
    - Conv2d(64 -> 3, kernel=3, stride=1, padding=1)
    - Sigmoid
)
```

Fine-Tuning with Frozen Encoder Architecture:

```
SegmentationDecoder(
  - Encoder: <frozen pre-trained encoder, not trained during decoder optimization>

  - Decoder:
    - ConvTranspose2d(128 -> 128, kernel=3, stride=2, padding=1, output_padding=1)
    - ResidualBlock(128 -> 128)
    - ConvTranspose2d(128 -> 64, kernel=3, stride=2, padding=1, output_padding=1)
    - ResidualBlock(64 -> 64)
    - ConvTranspose2d(64 -> 32, kernel=3, stride=2, padding=1, output_padding=1)
    - ResidualBlock(32 -> 32)
    - Conv2d(32 -> 32, kernel=3, padding=1)
    - BatchNorm2d(32)
    - ReLU
    - Conv2d(32 -> num_classes, kernel=1)
)
Each Residual Block consists of the following:
ResidualBlock(in_channels -> out_channels):
  - Conv2d(in_channels -> out_channels, kernel=3, padding=1)
  - BatchNorm2d(out_channels)
  - ReLU
  - Conv2d(out_channels -> out_channels, kernel=3, padding=1)
  - BatchNorm2d(out_channels)
  - Skip Connection (input + output)
  - ReLU

(Note: To allow the encoder to continue training,
set the requires_grad to True in the encoder part.)
```

A.2.3 Clip-Bolstered

```
SegmentationDecoder(
  - CLIP Projection:
    - clip_fc: Linear(clip_feature_dim -> 256)

  - CNN Encoder:
    - Conv2d(3 -> 64, kernel=3, stride=1, padding=1)
    - ReLU
    - MaxPool2d(kernel=2)
    - Conv2d(64 -> 128, kernel=3, stride=1, padding=1)
    - ReLU

  - Decoder:
    - ConvTranspose2d(384 (128 + 256) -> 64, kernel=3, stride=1, padding=1)
    - ReLU
    - Upsample(to 256x256, mode='bilinear', align_corners=True)
    - ConvTranspose2d(64 -> num_classes, kernel=3, stride=1, padding=1)
)
```

A.2.4 Prompt-Enriched-Clip-Bolstered

```
SegmentationDecoder(
  - CLIP Projection:
    - clip_fc: Linear(clip_feature_dim -> 256)

  - CNN Encoder (input: RGB + heatmap, 4 channels):
    - Conv2d(4 -> 64, kernel=3, stride=1, padding=1)
    - ReLU
    - MaxPool2d(kernel=2)
    - Conv2d(64 -> 128, kernel=3, stride=1, padding=1)
    - ReLU

  - Decoder:
    - ConvTranspose2d(384 -> 64, kernel=3, stride=1, padding=1)
    - ReLU
    - Upsample(to 256x256, mode='bilinear', align\_corners=True)
    - ConvTranspose2d(64 -> num_classes, kernel=3, stride=1, padding=1)
)
```

A.3 Code

The entire code consisting of Data Preprocess, Model Architectures, Model Training, Model Testing, Robustness Exploration and UI

A.3.1 data_preprocessing.py: Code to resize and augment the images

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Pre-Processing
# Date: 21.03.2025


"""0. Preliminary code"""

# Setting code-behaviour varaibles
doResize = False     # if True, the input images' resizing is run, otherwise the saved outcomes are used
doAugment = False    # if True, the resized images' augmentaion is run, otherwise the saved outcomes are used
genVISUALS = False   # set to False in order to avoid time-consuming visualizations' genaration (images are instead
      displayed as pre-saved in 'Output/Visuals' folder)
dpi = 500            # dpi for .pdf-saved images visualization (with genVISUALS = False)

# Importing useful libraries
import os
from pathlib import Path
from pdf2image import convert_from_path # (also install poppler-utils)
from IPython.display import display
from collections import Counter
from tabulate import tabulate
import numpy as np
from PIL import Image
import seaborn as sns
import cv2
import albumentations as A
import matplotlib.pyplot as plt

# Loading input data
input_folder_trainval = 'Data/Input/TrainVal'
input_trainval = [f for f in os.listdir(input_folder_trainval+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_trainval_labels = [f for f in os.listdir(input_folder_trainval+'/label') if f.lower().endswith(('.png'))]

input_folder_test = 'Data/Input/Test'
input_test = [f for f in os.listdir(input_folder_test+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_test_labels = [f for f in os.listdir(input_folder_test+'/label') if f.lower().endswith(('.png'))]

output_folder = 'Data/Output'
output_folder_resized = os.path.join(output_folder, 'Resized')
output_folder_augmented = os.path.join(output_folder, 'Augmented')

output_folder_resized_color = os.path.join(output_folder_resized, 'color')
output_folder_resized_label = os.path.join(output_folder_resized, 'label')
output_folder_augmented_color = os.path.join(output_folder_augmented, 'color')
output_folder_augmented_label = os.path.join(output_folder_augmented, 'label')
output_folder_resized_heatmaps = os.path.join(output_folder_resized, 'heatmaps')
output_folder_augmented_heatmaps = os.path.join(output_folder_augmented, 'heatmaps')


"""1. Dataset exploration"""

# Gauging image sizes
if genVISUALS or doResize:
  widths = []
  heights = []
  for img_file in input_trainval:
      with Image.open(os.path.join(input_folder_trainval+'/color', img_file)) as img:
          width, height = img.size
          widths.append(width)
          heights.append(height)

# Visualizing histogram distribution of dimensions
if genVISUALS or doResize:
  plt.figure(figsize=(12, 5))
  plt.subplot(1, 2, 1)
  plt.hist(widths, bins=20, color='lightblue', edgecolor='black')
  plt.xlabel("Width")
  plt.ylabel("Frequency")
  plt.title("Widths distribution")
  plt.subplot(1, 2, 2)
  plt.hist(heights, bins=20, color='skyblue', edgecolor='black')
  plt.xlabel("Height")
  plt.ylabel("Frequency")
  plt.title("Heights distribution")
  plt.tight_layout()
  plt.savefig(Path('Data/Output/Visuals/dimensions_histogram.pdf'))
  plt.show()
else:
  for image in convert_from_path(Path('Data/Output/Visuals/dimensions_histogram.pdf'), dpi=dpi):
    display(image)

# Visualizing boxplot distribution of dimensions
if genVISUALS or doResize:
```

```
  plt.figure(figsize=(10, 6))
  data = {"Width": widths, "Height": heights}
  sns.boxplot(data=data, palette=['lightblue', 'skyblue'])
  plt.title("Boxplot of Widths and Heights")
  plt.ylabel("Pixels")
  plt.tight_layout()
  plt.savefig(Path('Data/Output/Visuals/dimensions_boxplot.pdf'))
  plt.show()
else:
  for image in convert_from_path(Path('Data/Output/Visuals/dimensions_boxplot.pdf'), dpi=dpi):
    display(image)

# Printing stats
print(f"Input set size: {len(input_trainval)}\n")
if doResize:
  min_width, min_height = np.min(widths), np.min(heights)
  median_width, median_height = np.median(widths), np.median(heights)
  mean_width, mean_height = np.mean(widths), np.mean(heights)
  mode_width, mode_height = max(set(widths), key=widths.count), max(set(heights), key=heights.count)
  q3_width, q3_height = np.percentile(widths, 75), np.percentile(heights, 75)
  iqr_width = np.percentile(widths, 75) - np.percentile(widths, 25)
  iqr_height = np.percentile(heights, 75) - np.percentile(heights, 25)
  outlier_count_width = np.sum(widths > (q3_width + 1.5 * iqr_width))
  outlier_count_height = np.sum(heights > (q3_height + 1.5 * iqr_height))
  print(f"Min Size: {min_width}x{min_height}")
  print(f"Median Size: {median_width}x{median_height}")
  print(f"Mean Size: {mean_width:.2f}x{mean_height:.2f}")
  print(f"Mode Size: {mode_width}x{mode_height}")
  print(f"Q3 Size: {q3_width}x{q3_height}")
  print(f"Outliers in width: {outlier_count_width}")
  print(f"Outliers in height: {outlier_count_height}")
else:
  print("Q3 width and height values (both 500 pixels) were chosen for resizing.")

# Counting dogs' and cats' species instances
species_counts = Counter(filename.rsplit('_', 1)[0] for filename in input_trainval)
test_species_counts = Counter(filename.rsplit('_', 1)[0] for filename in input_test)
cat_species = ["abyssinian", "bengal", "birman", "bombay", "british_shorthair", "egyptian_mau", "maine_coon", "persian
    ", "ragdoll", "russian_blue", "siamese", "sphynx"]
cat_counts = {species: count for species, count in species_counts.items() if species.lower() in cat_species}
dog_counts = {species: count for species, count in species_counts.items() if species.lower() not in cat_species}
total_cats, total_dogs = sum(cat_counts.values()), sum(dog_counts.values())
test_cat_counts = {species: count for species, count in test_species_counts.items() if species.lower() in cat_species}
test_dog_counts = {species: count for species, count in test_species_counts.items() if species.lower() not in
    cat_species}
test_total_cats, test_total_dogs = sum(test_cat_counts.values()), sum(test_dog_counts.values())

# Displaying breeds' distribution
table_data = []
for species in sorted(cat_counts.keys()):
    table_data.append([species, cat_counts[species], test_cat_counts[species]])
table_data.append(["TOT CATS", total_cats, test_total_cats])
table_data.append(["-" * 20, "-" * 10, "-" * 10])
for species in sorted(dog_counts.keys()):
    table_data.append([species, dog_counts[species], test_dog_counts.get(species, 0)])
table_data.append(["TOT DOGS", total_dogs, test_total_dogs])
print(tabulate(table_data, headers=["Species", "Train/Val Count", "Test Count"], tablefmt="grid"))


"""2. Images' resizing"""

# Resizing images and labels (to Q3 width, Q3 height)
if doResize:
  imgResize = (int(q3_width), int(q3_height))
  widthsNP = np.array(widths)
  heightsNP = np.array(heights)
  i = 0
  for img_file in input_trainval:
      with Image.open(os.path.join(input_folder_trainval+'/color', img_file)) as img:
          img_resized = img.resize(imgResize, Image.Resampling.LANCZOS)
          if img_resized.mode == "RGBA":
            img_resized = img_resized.convert("RGB")
          img_resized.save(os.path.join(output_folder_resized_color, img_file), format="JPEG")
          i += 1
  print(f"{i} images resized to {int(q3_width)}x{int(q3_height)} and saved in {output_folder_resized_color}.")

  imgResize = (int(q3_width), int(q3_height))
  widthsNP = np.array(widths)
  heightsNP = np.array(heights)
  i = 0
  for img_file in input_trainval_labels:
      with Image.open(os.path.join(input_folder_trainval+'/label', img_file)) as img:
          img_resized = img.resize(imgResize, Image.Resampling.LANCZOS)
          if img_resized.mode == "RGBA":
            img_resized = img_resized.convert("RGB")
          img_resized.save(os.path.join(output_folder_resized_label, img_file), format="PNG")
          i += 1
  print(f"{i} labels resized to {int(q3_width)}x{int(q3_height)} and saved in {output_folder_resized_label}.")

else:
  print("Using previously resized images and labels (500x500).")
  imgResize = (500, 500)


"""3. Augmenting dataset"""
```

```
176
177 # Defining augmentations
178 augmentation = A.Compose([
179
180     # Geometric transformations (applied to all)
181     A.HorizontalFlip(p=0.5),
182     A.VerticalFlip(p=0.5),
183     A.RandomRotate90(p=0.5),
184     A.Affine(scale=(0.9, 1.1), translate_percent=(0.1, 0.1), rotate=(-15, 15), shear=(-10, 10), p=0.5),
185     A.ElasticTransform(alpha=1, sigma=50, alpha_affine=50, p=0.5),
186     A.GridDistortion(num_steps=5, distort_limit=0.3, interpolation=0, p=0.5), # NN-interpolation
187     A.OpticalDistortion(distort_limit=0.3, shift_limit=0.3, interpolation=0, p=0.5), # Nearest Neighbor
188
189     # Intensity augmentations (applied to images only)
190     A.GaussianBlur(blur_limit=(3, 7), p=0.3),
191     A.MotionBlur(blur_limit=5, p=0.3),
192     A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2, p=0.3),
193     A.HueSaturationValue(hue_shift_limit=20, sat_shift_limit=30, val_shift_limit=20, p=0.3),
194     A.GaussNoise(var_limit=(10.0, 50.0), p=0.3),
195     A.MultiplicativeNoise(multiplier=(0.9, 1.1), p=0.3)
196 ], additional_targets={'mask': 'mask', 'heatmap': 'mask'})
197
198 def is_cat(filename, cat_breeds = cat_species):
199   filename_lower = filename.lower()
200   return any(breed in filename_lower for breed in cat_breeds)
201
202 if doAugment:
203     i = 0
204     print('Augmenting␣the␣data...')
205     for img_file in os.listdir(output_folder_resized_color):
206         img_path = os.path.join(output_folder_resized_color, img_file)
207         mask_filename = os.path.splitext(img_file)[0] + ".png"
208         mask_path = os.path.join(output_folder_resized_label, mask_filename)
209         heatmap_filename = os.path.splitext(img_file)[0] + "_heatmap.png"
210         heatmap_path = os.path.join(output_folder_resized_heatmaps, heatmap_filename)
211
212         # Loading image, mask(grayscale) and heatmap
213         img = cv2.imread(img_path)
214         mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
215         heatmap = cv2.imread(heatmap_path, cv2.IMREAD_GRAYSCALE)
216
217         if img is None or mask is None or heatmap is None:
218             print(f"Skipping␣{img_file}␣as␣corresponding␣image␣or␣mask␣is␣missing!")
219             continue
220
221         img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
222
223         # Saving original image, mask and heatmap
224         Image.fromarray(img).save(os.path.join(output_folder_augmented_color, f'orig_{img_file}'), "JPEG")
225         Image.fromarray(mask).save(os.path.join(output_folder_augmented_label, f'orig_{mask_filename}'), "PNG")
226         Image.fromarray(heatmap).save(os.path.join(output_folder_augmented_heatmaps, f'orig_{heatmap_filename}'), "PNG
                ")
227
228         # Applying augmentation
229         augmented = augmentation(image=img, mask=mask, heatmap=heatmap)
230         aug_img, aug_mask, aug_heatmap = augmented['image'], augmented['mask'], augmented['heatmap']
231         aug_mask = np.round(aug_mask).astype(np.uint8)
232         aug_heatmap = np.round(aug_heatmap).astype(np.uint8)
233
234         # Saving augmented image and mask
235         Image.fromarray(aug_img).save(os.path.join(output_folder_augmented_color, f'aug_{i}_{img_file}'), "JPEG")
236         Image.fromarray(aug_mask).save(os.path.join(output_folder_augmented_label, f'aug_{i}_{mask_filename}'), "PNG")
237         Image.fromarray(aug_heatmap).save(os.path.join(output_folder_augmented_heatmaps, f'aug_{i}_{heatmap_filename}'
                ), "PNG")
238
239         i += 1
240
241         # If image is a cat, rotate 180 degrees and augment further
242         if is_cat(img_file):
243             rot_img = cv2.rotate(img, cv2.ROTATE_180)
244             rot_mask = cv2.rotate(mask, cv2.ROTATE_180)
245             rot_heatmap = cv2.rotate(heatmap, cv2.ROTATE_180)
246
247             Image.fromarray(rot_img).save(os.path.join(output_folder_augmented_color, f'rot_{img_file}'), "JPEG")
248             Image.fromarray(rot_mask).save(os.path.join(output_folder_augmented_label, f'rot_{mask_filename}'), "PNG")
249             Image.fromarray(rot_heatmap).save(os.path.join(output_folder_augmented_heatmaps, f'rot_{heatmap_filename}'
                    ), "PNG")
250
251             rotated_augmented = augmentation(image=rot_img, mask=rot_mask, heatmap=rot_heatmap)
252             rot_aug_img, rot_aug_mask, rot_aug_heatmap = rotated_augmented['image'], rotated_augmented['mask'],
                    rotated_augmented['heatmap']
253             rot_aug_mask = np.round(rot_aug_mask).astype(np.uint8)
254             rot_aug_heatmap = np.round(rot_aug_heatmap).astype(np.uint8)
255
256             Image.fromarray(rot_aug_img).save(os.path.join(output_folder_augmented_color, f'rot_aug_{i}_{img_file}'),
                    "JPEG")
257             Image.fromarray(rot_aug_mask).save(os.path.join(output_folder_augmented_label, f'rot_aug_{i}_{
                    mask_filename}'), "PNG")
258             Image.fromarray(rot_aug_heatmap).save(os.path.join(output_folder_augmented_heatmaps, f'rot_aug_{i}_{
                    heatmap_filename}'), "PNG")
259
260             i += 1
261
262     print(f"{i}␣images␣and␣masks␣augmented,␣including␣originals,␣output␣saved␣in␣{output_folder_augmented_color}␣&␣{
            output_folder_augmented_label}")
```

```
263
264 else:
265     print("Using previously augmented data.")
```

A.3.2 U-Net.py: Unet Architecture and Training

```
1  # Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
2  # Subject: Computer Vision Coursework. U-Net
3  # Date: 21.03.2025
4
5
6  # Global Constants
7  genVISUALS = False  # set to False in order to avoid time-consuming visualizations' genaration (images are instead
       displayed as pre-saved in 'Output/Visuals' folder)
8  dpi = 500           # dpi for .pdf-saved images visualization (with genVISUALS = False)
9  rePREPROC = False   # if True, the input images' resizing and augmentation are run, otherwise the saved outcomes are
       used
10 random_seed = 42
11
12
13 # IMPORTS
14 import os
15 from pathlib import Path
16 from pdf2image import convert_from_path  # (also install !apt-get install poppler-utils)
17 from IPython.display import display
18 import numpy as np
19 from PIL import Image
20 import seaborn as sns
21 import cv2
22 import tensorflow as tf
23 import albumentations as A
24 from sklearn.model_selection import train_test_split
25 import matplotlib.pyplot as plt
26 import torch
27 import torch.nn as nn
28 import torch.optim as optim
29 import torch.nn.functional as F
30 from torch.utils.data import DataLoader, Dataset
31 from torchvision import transforms
32 from torch.optim.lr_scheduler import StepLR
33 from tqdm import tqdm
34 import pickle as pkl
35
36
37 # Loading input data
38 input_folder_trainval = 'Data/Input/TrainVal'
39 input_trainval = [f for f in os.listdir(input_folder_trainval+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
40 input_trainval_labels = [f for f in os.listdir(input_folder_trainval+'/label') if f.lower().endswith(('.png'))]
41
42 input_folder_test = 'Data/Input/Test'
43 input_test = [f for f in os.listdir(input_folder_test+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
44 input_test_labels = [f for f in os.listdir(input_folder_test+'/label') if f.lower().endswith(('.png'))]
45
46 output_folder = 'Data/Output'
47 output_folder_resized = os.path.join(output_folder, 'Resized')
48 output_folder_augmented = os.path.join(output_folder, 'Augmented')
49
50 output_folder_resized_color = os.path.join(output_folder_resized, 'color')
51 output_folder_resized_label = os.path.join(output_folder_resized, 'label')
52 output_folder_augmented_color = os.path.join(output_folder_augmented, 'color')
53 output_folder_augmented_label = os.path.join(output_folder_augmented, 'label')
54
55 """### 1. Dataset preprocessing and augmentation
56
57 The images are resized to the dimensions (H<sub>min</sub>, W<sub>min</sub>), thus to take the Q3 height and width size
       over all the images in the dataset; the instances will be processed in this format, then the output resized back
       to the original dimensions...
58
59 Furthermore...(data augmentation)
60 """
61
62 # Printing stats
63 print(f"Input set size: {len(input_trainval)}\n")
64 print("Q3 width and height values (both 500 pixels) were chosen for resizing.")
65
66 """#### a) Resizing"""
67
68 # Resizing images (to Q3 width, Q3 height)
69 print("Using previously resized images and labels (500x500).")
70 imgResize = (256, 256)
71
72 """#### b) Augmenting dataset"""
73 print("Using previously augmented data.")
74
75
76 """#### c) Preparing datasets"""
77
78 # Preparing train, valid and test sets
79 validSize = 0.2
80 batchSize = 16
81 imgChannels = 3 ## this was 3
```

```
inputSize = (imgResize[0], imgResize[1], imgChannels)

import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

# Define preprocessed size
preprocessedSize = imgResize  # Desired size for input images and masks
inputSize = (128,128,3)

trainval_images = [os.path.join(output_folder_augmented_color, f) for f in os.listdir(output_folder_augmented_color)
     if f.endswith('.jpg')]
trainval_masks = [os.path.join(output_folder_augmented_label, f) for f in os.listdir(output_folder_augmented_label) if
      f.endswith('.png')]

train_images, val_images, train_masks, val_masks = train_test_split(trainval_images, trainval_masks, test_size=
     validSize, random_state=random_seed)

test_images = [os.path.join(input_folder_test, 'color', f) for f in os.listdir(os.path.join(input_folder_test, 'color'
     )) if f.endswith('.jpg')]
test_masks = [os.path.join(input_folder_test, 'label', f) for f in os.listdir(os.path.join(input_folder_test, 'label')
     ) if f.endswith('.png')]


print(f"Train set size: {len(train_images)} ({(1-validSize)*100}%)")
print(f"Valid set size: {len(val_images)} ({(validSize)*100}%)")
print(f" Test set size: {len(test_images)}")
print(f"\nInput dimension: {preprocessedSize +(imgChannels,)}")  # Adjusted for the preprocessed size
print(f"Batches' size: {batchSize}")

classesNum = 4 # number of output classes
epochsNum = 100 # number of training epochs
batchSize = 16


# Define the UNet architecture
class UNet(nn.Module):
    def __init__(self, num_classes=classesNum):
        super(UNet, self).__init__()

        # Encoder
        self.enc1 = self.conv_block(3, 64)
        self.pool1 = nn.MaxPool2d(2)

        self.enc2 = self.conv_block(64, 128)
        self.pool2 = nn.MaxPool2d(2)

        self.enc3 = self.conv_block(128, 256)
        self.pool3 = nn.MaxPool2d(2)

        self.enc4 = self.conv_block(256, 512)

        # Decoder
        self.up5 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.dec5 = self.conv_block(512, 256)

        self.up6 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.dec6 = self.conv_block(256, 128)

        self.up7 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec7 = self.conv_block(128, 64)

        self.final_conv = nn.Conv2d(64, num_classes, kernel_size=1) # output layer

    def conv_block(self, in_channels, out_channels):
        return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
        nn.Dropout(0.2),
        nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
        nn.BatchNorm2d(out_channels),
        nn.ReLU(inplace=True),
        nn.Dropout(0.2)
        )

    def forward(self, x):
        # Encoder
        c1 = self.enc1(x)
        p1 = self.pool1(c1)

        c2 = self.enc2(p1)
        p2 = self.pool2(c2)

        c3 = self.enc3(p2)
        p3 = self.pool3(c3)

        c4 = self.enc4(p3)

        # Decoder
        u5 = self.up5(c4)
        u5 = torch.cat([u5, c3], dim=1)
        c5 = self.dec5(u5)

        u6 = self.up6(c5)
        u6 = torch.cat([u6, c2], dim=1)
```

```
        c6 = self.dec6(u6)

        u7 = self.up7(c6)
        u7 = torch.cat([u7, c1], dim=1)
        c7 = self.dec7(u7)

        outputs = self.final_conv(c7)

        return outputs

# Define hyperparameters
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize model, optimizer, and loss function
model = UNet(classesNum).to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# criterion = nn.CrossEntropyLoss()

def dice_loss(pred, target, smooth=1e-6):
    pred = torch.softmax(pred, dim=1)
    target = torch.nn.functional.one_hot(target, num_classes=pred.shape[1]).permute(0, 3, 1, 2)
    intersection = (pred * target).sum(dim=(2, 3))
    union = pred.sum(dim=(2, 3)) + target.sum(dim=(2, 3))
    dice = (2.0 * intersection + smooth) / (union + smooth)
    return 1 - dice.mean()

def focal_loss(pred, target, alpha=0.8, gamma=2.0):
    pred = torch.softmax(pred, dim=1)
    target = torch.nn.functional.one_hot(target, num_classes=pred.shape[1]).permute(0, 3, 1, 2)
    logpt = -torch.nn.functional.cross_entropy(pred, target, reduction='none')
    pt = torch.exp(logpt)
    focal_loss = -((1 - pt) ** gamma) * logpt
    return focal_loss.mean()

# criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)

# criterion = lambda outputs, masks: 0.7 * dice_loss(outputs, masks) + 0.3 * focal_loss(outputs, masks)

# Evaluation
def evaluate(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, masks in test_loader:
            images, masks = images.to(device), masks.to(device)
            outputs = model(images)
            predictions = torch.argmax(outputs, dim=1)
            correct += (predictions == masks).sum().item()
            total += masks.numel()

    accuracy = correct / total
    print(f"Test Accuracy: {accuracy:.4f}")


class UNetDataset(Dataset):
    def __init__(self, image_files, mask_files, transform=None, target_transform=None, max_images=100):
        self.image_files = image_files  # Limit to first 100 images
        self.mask_files = mask_files  # Limit to first 100 masks
        self.transform = transform
        self.target_transform = target_transform

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

    def __getitem__(self, idx):
        img_path = self.image_files[idx]
        label_path = self.image_files[idx].replace('.jpg', '.png').replace('color', 'label')
        image = Image.open(img_path).convert('RGB')
        label = Image.open(label_path).convert('L')  # Load label as grayscale

        if self.transform:
            image = self.transform(image)

        label = label.resize(imgResize, Image.NEAREST)

        # Convert label to tensor
        label = torch.tensor(np.array(label), dtype=torch.long)  # Shape: (H, W)

        # # Map pixel values to class indices
        label = torch.where(label == 38, 1, label)
        label = torch.where(label == 75, 2, label)
        label = torch.where(label == 255, 3, label)
        label = torch.where(label == 0, 0, label)  # Ensure 0 stays as class 0

        return image, label  # No one-hot encoding

# Define transformations
image_transform = transforms.Compose([
    transforms.Resize(imgResize),
    transforms.ToTensor()
])

# Load dataset with first 100 images
```

```
train_dataset = UNetDataset(train_images, train_masks, transform=image_transform, target_transform=None, max_images=100)
train_loader = DataLoader(train_dataset, batch_size=batchSize, shuffle=True)

val_dataset = UNetDataset(val_images, val_masks, transform=image_transform, target_transform=None, max_images=100)
val_loader = DataLoader(val_dataset, batch_size=batchSize, shuffle=False)

test_dataset = UNetDataset(test_images, test_masks, transform=image_transform, target_transform=None, max_images=100)
test_loader = DataLoader(test_dataset, batch_size=batchSize, shuffle=False)

# Define the training function
def train_unet(model, train_loader, val_loader, epochs, model_save_path="/home/s2677266/CVis/Data/Output/Models_DICEONLY/", device="cuda" if torch.cuda.is_available() else "cpu"):
    model.to(device)

    # criterion = lambda outputs, masks: 0.7 * dice_loss(outputs, masks) + 0.3 * focal_loss(outputs, masks)
    # criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)
    criterion = lambda outputs, masks: dice_loss(outputs, masks)

    # criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    for epoch in range(epochs):
        model.train()
        epoch_loss = 0
        for images, masks in tqdm(train_loader, desc=f"Eppoch {epoch+1}/{epochs}"):
            images, masks = images.to(device), masks.long().squeeze(1).to(device)  # Ensure correct shape

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, masks)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

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

        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for images, masks in val_loader:
                images, masks = images.to(device), masks.long().squeeze(1).to(device)
                outputs = model(images)
                loss = criterion(outputs, masks)
                val_loss += loss.item()

        print(f"Epoch {epoch+1}/{epochs}, Validation Loss: {val_loss / len(val_loader):.4f}")

        # Save model after each epoch
        # epoch_model_path = f"{model_save_path}Better_Final_Outline_unet_model_epoch_{epoch+1}.pth"  # with 32, 128
        epoch_model_path = f"{model_save_path}UNET_DiceOnly_EPOCH_{epoch+1}.pth"
        torch.save(model.state_dict(), epoch_model_path)
        print(f"Model saved at {epoch_model_path}")

    print("Training complete!")

# Define the testing function
def test_unet(model, test_loader, device="cuda" if torch.cuda.is_available() else "cpu"):
    model.to(device)
    model.eval()
    test_loss = 0

    # criterion = lambda outputs, masks: 0.7 * dice_loss(outputs, masks) + 0.3 * focal_loss(outputs, masks)
    # criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)
    # criterion = nn.CrossEntropyLoss()
    criterion = lambda outputs, masks: dice_loss(outputs, masks)

    with torch.no_grad():
        for images, masks in test_loader:
            images, masks = images.to(device), masks.long().squeeze(1).to(device)
            outputs = model(images)
            loss = criterion(outputs, masks)
            test_loss += loss.item()

    print(f"Test Loss: {test_loss / len(test_loader):.4f}")

# Call the training function
train_unet(model, train_loader, val_loader, epochs=epochsNum)

# Evaluate on the test set
test_unet(model, test_loader)

print('Execution Completed')
```

A.3.3 Autoencoder.py: Autoencoder Architecture and Training

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Auto-Encoder
# Date: 21.03.2025

```

```
# Global Constants
genVISUALS = True  # set to False in order to avoid time-consuming visualizations' genaration (images are instead displayed as pre-saved in 'Output/Visuals' folder)
dpi = 500           # dpi for .pdf-saved images visualization (with genVISUALS = False)
rePREPROC = True   # if True, the input images' resizing and augmentation are run, otherwise the saved outcomes are used
random_seed = 42
trainEncoder = True


# IMPORTS
import os
from pathlib import Path
from pdf2image import convert_from_path  # (also install !apt-get install poppler-utils)
from IPython.display import display
import numpy as np
from PIL import Image
import seaborn as sns
import cv2
import tensorflow as tf
import albumentations as A
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
from torch.optim.lr_scheduler import StepLR
from tqdm import tqdm
import pickle as pkl
from torch.cuda.amp import autocast, GradScaler

if torch.cuda.is_available():
    print("using␣GPU")
else:
    print("using␣CPU")

# Loading input data
input_folder_trainval = 'Data/Input/TrainVal'
input_trainval = [f for f in os.listdir(input_folder_trainval+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_trainval_labels = [f for f in os.listdir(input_folder_trainval+'/label') if f.lower().endswith(('.png'))]

input_folder_test = 'Data/Input/Test'
input_test = [f for f in os.listdir(input_folder_test+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_test_labels = [f for f in os.listdir(input_folder_test+'/label') if f.lower().endswith(('.png'))]

output_folder = 'Data/Output'
output_folder_resized = os.path.join(output_folder, 'Resized')
output_folder_augmented = os.path.join(output_folder, 'Augmented')

output_folder_resized_color = os.path.join(output_folder_resized, 'color')
output_folder_resized_label = os.path.join(output_folder_resized, 'label')
output_folder_augmented_color = os.path.join(output_folder_augmented, 'color')
output_folder_augmented_label = os.path.join(output_folder_augmented, 'label')

"""### 1. Dataset preprocessing and augmentation

The images are resized to the dimensions (H<sub>min</sub>, W<sub>min</sub>), thus to take the Q3 height and width size over all the images in the dataset; the instances will be processed in this format, then the output resized back to the original dimensions...

Furthermore...(data augmentation)
"""

# Printing stats
print(f"Input␣set␣size:␣{len(input_trainval)}\n")
print("Q3␣width␣and␣height␣values␣(both␣500␣pixels)␣were␣chosen␣for␣resizing.")

"""#### a) Resizing"""

# Resizing images (to Q3 width, Q3 height)
print("Using␣previously␣resized␣images␣and␣labels␣(500x500).")
imgResize = (500, 500)

"""#### b) Augmenting dataset"""
print("Using␣previously␣augmented␣data.")


"""#### c) Preparing datasets"""

# Preparing train, valid and test sets
validSize = 0.2
batchSize = 4
imgChannels = 3 ## this was 3
inputSize = (imgResize[0], imgResize[1], imgChannels)
classesNum = 4 # number of output classes
epochsNum = 30 # number of training epochs
batchSize = 32

import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

# Define preprocessed size
```

```
preprocessedSize = (256, 256)  # Desired size for input images and masks
inputSize = (256,256,3)

trainval_images = [os.path.join(output_folder_augmented_color, f) for f in os.listdir(output_folder_augmented_color) if f.endswith('.jpg')]
trainval_masks = [os.path.join(output_folder_augmented_label, f) for f in os.listdir(output_folder_augmented_label) if f.endswith('.png')]

train_images, val_images, train_masks, val_masks = train_test_split(trainval_images, trainval_masks, test_size=validSize, random_state=random_seed)

test_images = [os.path.join(input_folder_test, 'color', f) for f in os.listdir(os.path.join(input_folder_test, 'color')) if f.endswith('.jpg')]
test_masks = [os.path.join(input_folder_test, 'label', f) for f in os.listdir(os.path.join(input_folder_test, 'label')) if f.endswith('.png')]


print(f"Train set size: {len(train_images)} ({(1-validSize)*100}%)")
print(f"Valid set size: {len(val_images)} ({(validSize)*100}%)")
print(f" Test set size: {len(test_images)}")
print(f"\nInput dimension: {preprocessedSize + (imgChannels,)}")  # Adjusted for the preprocessed size
print(f"Batches' size: {batchSize}")




#####  DEFINING THE ENCODER PART OF THE AUTO-ENCODER (PRE TRAINING) #####

scaler = GradScaler()
# Define the Autoencoder
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
        nn.LeakyReLU(0.2),
        nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
        nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
        nn.LeakyReLU(0.2),
        nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
        nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
        nn.LeakyReLU(0.2),
        nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
        nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1),
        nn.LeakyReLU(0.2),
    )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(128, 256, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),  # x2 size
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1),
            nn.Sigmoid()
        )

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


  # Below is the code to train the autoencoder

if trainEncoder == True:
    class ImageDataset(Dataset):
        def __init__(self, image_files, transform=None, max_images=100):
            self.image_files = image_files
            self.transform = transform

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

        def __getitem__(self, idx):
            img_path = self.image_files[idx]
            image = Image.open(img_path).convert('RGB')
            if self.transform:
                image = self.transform(image)
            return image

    # Transformations
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
    #     transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # Normalize to [-1,1]
    ])

```

```
186
187     # Load dataset with first 100 images
188     dataset = ImageDataset(image_files=train_images, transform=transform, max_images=100)
189     dataloader = DataLoader(dataset, batch_size=batchSize, shuffle=True)
190
191     # Device configuration
192     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
193     print(device)
194
195     # Initialize model, loss, and optimizer
196     autoencoder = Autoencoder().to(device)
197     criterion = nn.MSELoss()
198     optimizer = optim.Adam(autoencoder.parameters(), lr=1e-3)
199
200     def weights_init(m):
201         if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
202             nn.init.xavier_uniform_(m.weight)
203             if m.bias is not None:
204                 nn.init.zeros_(m.bias)
205
206     autoencoder.apply(weights_init)
207
208     # Training the Autoencoder
209     num_epochs = 100
210     for epoch in range(num_epochs):
211         print(f"Epoch [{epoch+1}/{num_epochs}] Running:")
212         torch.cuda.empty_cache()
213         epoch_loss = 0.0
214         for batch in dataloader:
215             batch = batch.to(device)
216             optimizer.zero_grad()
217
218             with autocast():
219                 outputs = autoencoder(batch)
220                 loss = criterion(outputs, batch)
221                 epoch_loss += loss.item()
222
223             scaler.scale(loss).backward()
224             scaler.step(optimizer)
225             scaler.update()
226
227         print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss / len(dataloader)}")
228         torch.save(autoencoder.state_dict(), f'Data/Output/Models2/updated_complex_working_autoencoder_epoch_{epoch
                +1}.pth')
229
230
231 ###### NOW THAT THE ENCODER IS PRE-TRAINED, WE CAN CREATE A GOOD SEGMENTER #######
232
233 # Residual Block
234 class ResidualBlock(nn.Module):
235     def __init__(self, in_channels, out_channels):
236         super(ResidualBlock, self).__init__()
237         self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
238         self.bn1 = nn.BatchNorm2d(out_channels)
239         self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
240         self.bn2 = nn.BatchNorm2d(out_channels)
241         self.relu = nn.ReLU(inplace=True)
242
243     def forward(self, x):
244         residual = x
245         x = self.relu(self.bn1(self.conv1(x)))
246         x = self.bn2(self.conv2(x))
247         x += residual # skip connection
248         return self.relu(x)
249
250 # Enhanced Segmentation Decoder
251 class SegmentationDecoder(nn.Module):
252     def __init__(self, encoder):
253         super(SegmentationDecoder, self).__init__()
254
255         for param in encoder.parameters():
256             param.requires_grad = False # set to False to FREEZE THE ENCODER
257
258         self.encoder = encoder
259
260         self.decoder = nn.Sequential(
261             nn.ConvTranspose2d(128, 128, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
262             ResidualBlock(128, 128),
263             nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),  # x2 size
264             ResidualBlock(64, 64),
265             nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1),   # x2 size
266             ResidualBlock(32, 32),
267             nn.Conv2d(32, 32, kernel_size=3, padding=1),
268             nn.BatchNorm2d(32),
269             nn.ReLU(inplace=True),
270             nn.Conv2d(32, classesNum, kernel_size=1)
271         )
272
273     def forward(self, x):
274         x = self.encoder(x)
275         x = self.decoder(x)
276         return x
277
278
279 # Custom Dataset for loading images and labels
```

```
class SegmentationDataset(Dataset):
    def __init__(self, image_files, label_files, transform=None):
        self.image_files = image_files
        self.label_files = label_files
        self.transform = transform

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


    def __getitem__(self, idx):
        img_path = self.image_files[idx]
        label_path = self.image_files[idx].replace('.jpg', '.png').replace('color', 'label')
        image = Image.open(img_path).convert('RGB')
        label = Image.open(label_path).convert('L')

        if self.transform:
            image = self.transform(image)

        label = label.resize((256, 256), Image.NEAREST)

        # Convert label to tensor
        label = torch.tensor(np.array(label), dtype=torch.long)

        # # Map pixel values to class indices
        label = torch.where(label == 38, 1, label)
        label = torch.where(label == 75, 2, label)
        label = torch.where(label == 255, 3, label)
        label = torch.where(label == 0, 0, label)

        return image, label

# Transformations
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
])


# Load segmentation dataset with first 100 images
segmentation_dataset = SegmentationDataset(
    image_files=train_images,
    label_files=train_masks,
    transform=transform
)

val_segmentation_dataset = SegmentationDataset(
    image_files=val_images,
    label_files=val_masks,
    transform=transform
)

segmentation_dataloader = DataLoader(segmentation_dataset, batch_size=batchSize, shuffle=True)
val_segmentation_dataloader = DataLoader(val_segmentation_dataset, batch_size=batchSize, shuffle=True)

# Load the pre-trained autoencoder
autoencoder = Autoencoder()
autoencoder.load_state_dict(torch.load('Data/Output/Models2/updated_complex_working_autoencoder_epoch_95.pth'))

print("Loaded␣Autoencoder:␣Data/Output/Models2/updated_complex_working_autoencoder_epoch_95")

print("Saving␣decoder␣at␣Data/Output/Models3/Not_Frozen_Val_Segmentation_95_Decoder_DICE_epoch_")
print("Frozen")
print("segmentation_criterion␣=␣lambda␣outputs,␣masks:␣nn.CrossEntropyLoss()(outputs,␣masks)␣+␣dice_loss(outputs,␣
     masks)")
# Initialize the segmentation model
# segmentation_model = SegmentationDecoder(autoencoder.encoder)

# Define loss and optimizer for segmentation
segmentation_model = SegmentationDecoder(autoencoder.encoder)
#class_weights = torch.tensor([1.0, 1.0, 1.0, 1.0])  # Adjust these values as needed
# segmentation_criterion = nn.CrossEntropyLoss()#(weight=class_weights)

def dice_loss(pred, target, smooth=1e-6):
    pred = torch.softmax(pred, dim=1)
    target = torch.nn.functional.one_hot(target, num_classes=pred.shape[1]).permute(0, 3, 1, 2)
    intersection = (pred * target).sum(dim=(2, 3))
    union = pred.sum(dim=(2, 3)) + target.sum(dim=(2, 3))
    dice = (2.0 * intersection + smooth) / (union + smooth)
    return 1 - dice.mean()

# segmentation_criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)
# segmentation_criterion = nn.CrossEntropyLoss()
# Example of class weights (higher weight for less frequent classes)
segmentation_optimizer = optim.Adam(segmentation_model.parameters(), lr=1e-3, weight_decay=1e-4)

segmentation_criterion = lambda outputs, masks: dice_loss(outputs, masks)

# Training the Segmentation Model

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
segmentation_model.to(device)
# scheduler = torch.optim.lr_scheduler.StepLR(segmentation_optimizer, step_size=3, gamma=0.5)
# scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(segmentation_optimizer, T_max=10, eta_min=1e-5)

```

```
scaler = torch.cuda.amp.GradScaler()

num_epochs = 100
for epoch in tqdm(range(num_epochs), desc="Training Progress"):
    print(f"Epoch [{epoch+1}/{num_epochs}] Running:")
    segmentation_model.train()
    epoch_loss = 0

    for images, labels in segmentation_dataloader:
        images, labels = images.to(device), labels.to(device)

        segmentation_optimizer.zero_grad()


        with autocast():
            outputs = segmentation_model(images)
            loss = segmentation_criterion(outputs, labels)
            epoch_loss += loss.item()
        scaler.scale(loss).backward()
        scaler.step(segmentation_optimizer)
        scaler.update()
    print(f'Epoch [{epoch+1}/{num_epochs}], Training Loss : {epoch_loss / len(segmentation_dataloader)}')

    segmentation_model.eval()
    val_loss = 0
    with torch.no_grad():
        for images, labels in val_segmentation_dataloader:
            images, labels = images.to(device), labels.to(device)
            outputs = segmentation_model(images)
            loss = segmentation_criterion(outputs, labels)
            val_loss += loss.item()

    print(f'Epoch [{epoch+1}/{num_epochs}], Validation Loss : {val_loss / len(val_segmentation_dataloader)}')
    print(f'Saved Model at Data/Output/Models_DICEONLY/Frozen_Val_Segmentation_95_Decoder_DiceOnly_epoch_{epoch+1}.pth
        ')
    torch.save(segmentation_model.state_dict(), f'Data/Output/Models_DICEONLY/
        Frozen_Val_Segmentation_95_Decoder_DiceOnly_epoch_{epoch+1}.pth')
```

A.3.4 Clip.py: Clip Architecture and Training

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Clip
# Date: 21.03.2025


"""0. Preliminary code"""

# Setting code-behaviour varaibles
extractFeatures = True  # if True, the CLIP fetures are extracted, otherwise the saved ones are used

preprocessedSize = (256, 256) # Desired size to read input images and masks in
imgChannels = 3
classesNum = 4

validSize = 0.2
epochsNum = 100
batchSize = 16
learningRate = 1e-3

random_seed = 42

# Importing useful libraries
import os
import numpy as np
from PIL import Image
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from tqdm import tqdm
import open_clip

# Loading input data
input_folder_trainval = 'Data/Input/TrainVal'
input_trainval = [f for f in os.listdir(input_folder_trainval+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_trainval_labels = [f for f in os.listdir(input_folder_trainval+'/label') if f.lower().endswith(('.png'))]

input_folder_test = 'Data/Input/Test'
input_test = [f for f in os.listdir(input_folder_test+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_test_labels = [f for f in os.listdir(input_folder_test+'/label') if f.lower().endswith(('.png'))]

output_folder = 'Data/Output'
output_folder_resized = os.path.join(output_folder, 'Resized')
output_folder_augmented = os.path.join(output_folder, 'Augmented')

output_folder_resized_color = os.path.join(output_folder_resized, 'color')
output_folder_resized_label = os.path.join(output_folder_resized, 'label')
output_folder_augmented_color = os.path.join(output_folder_augmented, 'color')
output_folder_augmented_label = os.path.join(output_folder_augmented, 'label')
```

```
52
53  # Preparing train, valid and test sets
54  os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"
55  trainval_images = [os.path.join(output_folder_augmented_color, f) for f in os.listdir(output_folder_augmented_color)
        if f.endswith('.jpg')]
56  trainval_masks = [os.path.join(output_folder_augmented_label, f) for f in os.listdir(output_folder_augmented_label) if
        f.endswith('.png')]
57  train_images, val_images, train_masks, val_masks = train_test_split(trainval_images, trainval_masks, test_size=
        validSize, random_state=random_seed)
58  test_images = [os.path.join(input_folder_test, 'color', f) for f in os.listdir(os.path.join(input_folder_test, 'color'
        )) if f.endswith('.jpg')]
59  test_masks = [os.path.join(input_folder_test, 'label', f) for f in os.listdir(os.path.join(input_folder_test, 'label')
        ) if f.endswith('.png')]
60
61
62  print(f"Train␣set␣size:␣{len(train_images)}␣({(1-validSize)*100}%)")
63  print(f"Valid␣set␣size:␣{len(val_images)}␣({(validSize)*100}%)")
64  print(f"␣Test␣set␣size:␣{len(test_images)}")
65  print(f"\nInput␣dimension:␣{preprocessedSize␣+␣(imgChannels,)}")
66  print(f"Batches'␣size:␣{batchSize}")
67
68
69  """1. CLIP features extraction"""
70
71  # Loading CLIP model
72  device = "cuda" if torch.cuda.is_available() else "cpu"
73  model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
74  model = model.to(device)
75  tokenizer = open_clip.get_tokenizer("ViT-B-32")
76
77  # Preparing features'saving directory
78  clip_features_dir = "Data/Output/Models/clipTrainFeatures"
79  os.makedirs(clip_features_dir, exist_ok=True)
80
81  if extractFeatures:
82      # Extract and save CLIP features
83      def extract_clip_features(image_path):
84          image = Image.open(image_path).convert("RGB")
85          image_tensor = preprocess_val(image).unsqueeze(0).to(device)  # Use open_clip's preprocessing
86          with torch.no_grad():
87              features = model.encode_image(image_tensor)  # Extract image features
88          features = features / features.norm(p=2, dim=-1, keepdim=True)  # Normalize
89          return features.cpu()
90
91      clip_features_trainval = {}
92      print("Extracting␣Clip␣Features")
93      for image_name in trainval_images:
94          image_path = image_name
95          clip_features = extract_clip_features(image_path)
96
97          # Save each feature tensor separately
98          feature_path = os.path.join(clip_features_dir, f"{image_name.split('/')[-1].split('.')[0]}.pt")
99          torch.save(clip_features, feature_path)
100
101         clip_features_trainval[image_name] = feature_path
102
103
104 """2. CLIP-based segmentation"""
105
106 # Loading CLIP features
107 clip_features_trainval = {
108     image_name: torch.load(os.path.join(clip_features_dir, f"{image_name.split('/')[-1].split('.')[0]}.pt"))
109     for image_name in trainval_images
110 }
111 print(f"Loaded␣{len(clip_features_trainval)}␣CLIP␣feature␣tensors.")
112
113 print("Saving␣model␣to␣this␣location:␣Data/Output/Models2/CLIP_CROSS2_epoch_")
114
115 class SegmentationDataset(Dataset):
116     def __init__(self, image_files, label_files, clip_features_dict, transform=None):
117         self.image_files = image_files
118         self.label_files = label_files
119         self.clip_features_dict = clip_features_dict
120         self.transform = transform
121
122     def __len__(self):
123         return len(self.image_files)
124
125     def __getitem__(self, idx):
126         img_path = self.image_files[idx]
127         label_path = img_path.replace('.jpg', '.png').replace('color', 'label')
128
129         # Loading image
130         image = Image.open(img_path).convert("RGB")
131         if self.transform:
132             image = self.transform(image)
133         else:
134             image = transforms.ToTensor()(image)
135
136         # Loading CLIP feature tensor
137         image_name = img_path
138         clip_feature = self.clip_features_dict[image_name]
139
140         # Loading and processing label
141         label = Image.open(label_path).convert('L')
```

```
        label = label.resize(preprocessedSize, Image.NEAREST)
        label = torch.tensor(np.array(label), dtype=torch.long)

        # Converting label classes
        label = torch.where(label == 38, 1, label)
        label = torch.where(label == 75, 2, label)
        label = torch.where(label == 255, 3, label)
        label = torch.where(label == 0, 0, label)

        return image, clip_feature, label

# Defining Transformations
transform = transforms.Compose([
    transforms.Resize(preprocessedSize, interpolation=Image.NEAREST),
    transforms.ToTensor()
])

# Loading Dataset & DataLoader
segmentation_dataset = SegmentationDataset(
    image_files=train_images,
    label_files=train_masks,
    clip_features_dict=clip_features_trainval,
    transform=None
)

segmentation_dataloader = DataLoader(segmentation_dataset, batch_size=batchSize, shuffle=True)

val_segmentation_dataset = SegmentationDataset(
    image_files=val_images,
    label_files=val_masks,
    clip_features_dict=clip_features_trainval,
    transform=None
)

val_segmentation_dataloader = DataLoader(val_segmentation_dataset, batch_size=batchSize, shuffle=True)

class SegmentationDecoder(nn.Module):
    def __init__(self, clip_feature_dim=512, num_classes=classesNum):
        super(SegmentationDecoder, self).__init__()

        # Project CLIP features
        self.clip_fc = nn.Linear(clip_feature_dim, 256)

        # CNN Feature Extractor
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
            nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
        )

    def forward(self, image, clip_features):
        cnn_features = self.encoder(image)
        clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
        clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
        fusion = torch.cat([cnn_features, clip_features], dim=1)
        segmentation_output = self.decoder(fusion)
        return segmentation_output


def dice_loss(pred, target, smooth=1e-6):
    pred = torch.softmax(pred, dim=1)
    target = torch.nn.functional.one_hot(target, num_classes=pred.shape[1]).permute(0, 3, 1, 2)
    intersection = (pred * target).sum(dim=(2, 3))
    union = pred.sum(dim=(2, 3)) + target.sum(dim=(2, 3))
    dice = (2.0 * intersection + smooth) / (union + smooth)
    return 1 - dice.mean()

# Initialize Model, Loss, Optimizer
segmentation_model = SegmentationDecoder().to(device)
# segmentation_criterion = nn.CrossEntropyLoss()
segmentation_criterion = lambda outputs, masks: dice_loss(outputs, masks)
# segmentation_criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)

print("Loss: segmentation_criterion = nn.CrossEntropyLoss()")

segmentation_optimizer = optim.Adam(segmentation_model.parameters(), lr=learningRate)
scheduler = torch.optim.lr_scheduler.StepLR(segmentation_optimizer, step_size=5, gamma=0.5)
scaler = torch.cuda.amp.GradScaler()

# Training model
for epoch in tqdm(range(epochsNum), desc="Training Progress"):
    segmentation_model.train()
    running_loss = 0.0

    print(f"Epoch [{epoch+1}/{epochsNum}] Running:")
```

```

    for images, clip_features, labels in segmentation_dataloader:
        images, clip_features, labels = images.to(device), clip_features.to(device), labels.to(device)

        segmentation_optimizer.zero_grad()

        # Mixed precision training
        with torch.cuda.amp.autocast():
            outputs = segmentation_model(images, clip_features)
            loss = segmentation_criterion(outputs, labels)

        # Scaling loss and backpropagating
        scaler.scale(loss).backward()
        scaler.step(segmentation_optimizer)
        scaler.update()

        running_loss += loss.item()

    segmentation_model.eval()
    val_loss = 0
    with torch.no_grad():
        for images, clip_features, labels in val_segmentation_dataloader:
            images, clip_features, labels = images.to(device), clip_features.to(device), labels.to(device)
            outputs = segmentation_model(images, clip_features)
            loss = segmentation_criterion(outputs, labels)
            val_loss += loss.item()

    print(f'Epoch [{epoch+1}/{epochsNum}], Validation Loss: {val_loss / len(val_segmentation_dataloader)}')

    scheduler.step()

    avg_loss = running_loss / len(segmentation_dataloader)

    print(f"Epoch [{epoch+1}/{epochsNum}], Training Loss: {avg_loss:.4f}")
    print(f"Saved here: Data/Output/Models_DICEONLY/CLIP_DiceOnly_epoch_{epoch+1}.pth")
    torch.save(segmentation_model.state_dict(), f'Data/Output/Models_DICEONLY/CLIP_DiceOnly_epoch_{epoch+1}.pth')

print("Training complete!")
```

A.3.5 annotation.py: Annotating the Data for Prompted-Clip Training

```
import cv2
import os
import pickle

# Define paths
image_dir = "Data/Output/Resized/color"
annotations_dir = "Data/Output/Resized/annotations"

# Ensure the annotations directory exists
os.makedirs(annotations_dir, exist_ok=True)

# Specify the range of images
start = 1
end = 3680

# Generate the output file name for coordinates
output_file = os.path.join(annotations_dir, f"annotations_{start}_to_{end}.pkl")

# Get all images in the directory and sort them
image_files = sorted([f for f in os.listdir(image_dir) if f.endswith((".jpg", ".png"))])
image_files = image_files[start - 1:end]

# Dictionary to store annotations
annotations = {}
clicked = False  # Flag to track click status
point = None  # Store clicked point

# Mouse click event callback function
def click_event(event, x, y, flags, param):
    global clicked, point
    if event == cv2.EVENT_LBUTTONDOWN:
        point = (x, y)
        clicked = True  # Set flag

# Loop through images
for image_name in image_files:
    current_image = image_name
    image_path = os.path.join(image_dir, image_name)

    img = cv2.imread(image_path)
    if img is None:
        print(f"Error loading {image_name}, skipping...")
        continue

    img = cv2.resize(img, (500, 500))
    clicked = False
    point = None

    # Display image and wait for a click
    cv2.imshow("Annotate Nose - Click on the nose", img)
```

```
    cv2.setMouseCallback("Annotate Nose - Click on the nose", click_event)

    while not clicked:
        cv2.waitKey(1)

    cv2.destroyAllWindows()

    if point:
        annotations[current_image] = point

        # Draw point on image
        cv2.circle(img, point, 5, (0, 0, 255), -1)  # Red dot

        # Save annotated image
        annotated_path = os.path.join(annotations_dir, f"annotated_{image_name}")
        cv2.imwrite(annotated_path, img)
        print(f"Saved annotated image: {annotated_path}")

# Save annotations to a .pkl file
with open(output_file, "wb") as f:
    pickle.dump(annotations, f)

print(f"Annotations saved to {output_file}")
cv2.destroyAllWindows()
```

A.3.6 buildHeatmaps.py: Building Heatmaps for Prompted-Clip Training

```
import os
import pickle
import numpy as np
import matplotlib.pyplot as plt

# Parameters
merged_annotations_file = "Data/Output/test_annotations.pkl"
output_dir = "Data/Input/Test/heatmaps/"
os.makedirs(output_dir, exist_ok=True)

def generate_gaussian_heatmap(size, center, intensity=1.0, sigma=100):
    x0, y0 = center
    x = np.arange(size[0])
    y = np.arange(size[1])
    X, Y = np.meshgrid(x, y)
    heatmap = intensity * np.exp(-((X - x0)**2 + (Y - y0)**2) / (2 * sigma**2))
    return heatmap

# Load merged annotations
with open(merged_annotations_file, "rb") as f:
    annotations = pickle.load(f)

# Generate heatmaps
for img_name, (x, y) in annotations.items():
    heatmap = generate_gaussian_heatmap((500, 500), (x, y), intensity=1.0, sigma=50)
    plt.imsave(os.path.join(output_dir, f"{img_name.split('.')[0]}_heatmap.png"), heatmap, cmap='gray')

print(f"Generated {len(annotations)} heatmaps in {output_dir}")
```

A.3.7 Prompt-Clip.py: Prompt-Clip Architecture and Training

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Clip
# Date: 21.03.2025


"""0. Preliminary code"""

# Setting code-behaviour varaibles
extractFeatures = True  # if True, the CLIP fetures are extracted, otherwise the saved ones are used

preprocessedSize = (256, 256) # Desired size to read input images and masks in
imgChannels = 3
classesNum = 4

validSize = 0.2
epochsNum = 100
batchSize = 16
learningRate = 1e-3

random_seed = 42

# Importing useful libraries
import os
import numpy as np
from PIL import Image
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
```

```
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms
from tqdm import tqdm
import open_clip

# Loading input data
input_folder_trainval = 'Data/Input/TrainVal'
input_trainval = [f for f in os.listdir(input_folder_trainval+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_trainval_labels = [f for f in os.listdir(input_folder_trainval+'/label') if f.lower().endswith(('.png'))]

input_folder_test = 'Data/Input/Test'
input_test = [f for f in os.listdir(input_folder_test+'/color') if f.lower().endswith(('.jpg', '.jpeg'))]
input_test_labels = [f for f in os.listdir(input_folder_test+'/label') if f.lower().endswith(('.png'))]

output_folder = 'Data/Output'
output_folder_resized = os.path.join(output_folder, 'Resized')
output_folder_augmented = os.path.join(output_folder, 'Augmented')

output_folder_resized_color = os.path.join(output_folder_resized, 'color')
output_folder_resized_label = os.path.join(output_folder_resized, 'label')

output_folder_augmented_color = os.path.join(output_folder_augmented, 'color')
output_folder_augmented_label = os.path.join(output_folder_augmented, 'label')

# Preparing train, valid and test sets
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"
trainval_images = [os.path.join(output_folder_augmented_color, f) for f in os.listdir(output_folder_augmented_color) if f.endswith('.jpg')]
trainval_masks = [os.path.join(output_folder_augmented_label, f) for f in os.listdir(output_folder_augmented_label) if f.endswith('.png')]
train_images, val_images, train_masks, val_masks = train_test_split(trainval_images, trainval_masks, test_size=validSize, random_state=random_seed)
test_images = [os.path.join(input_folder_test, 'color', f) for f in os.listdir(os.path.join(input_folder_test, 'color')) if f.endswith('.jpg')]
test_masks = [os.path.join(input_folder_test, 'label', f) for f in os.listdir(os.path.join(input_folder_test, 'label')) if f.endswith('.png')]


print(f"Train set size: {len(train_images)} ({(1-validSize)*100}%)")
print(f"Valid set size: {len(val_images)} ({(validSize)*100}%)")
print(f" Test set size: {len(test_images)}")
print(f"\nInput dimension: {preprocessedSize + (imgChannels,)}")
print(f"Batches' size: {batchSize}")


"""1. CLIP features extraction"""

# Loading CLIP model
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
model = model.to(device)
tokenizer = open_clip.get_tokenizer("ViT-B-32")

# Preparing features'saving directory
clip_features_dir = "Data/Output/Models/ResOnlyClipTrainFeatures3"
os.makedirs(clip_features_dir, exist_ok=True)

if extractFeatures:
    # Extract and save CLIP features
    def extract_clip_features(image_path):
        image = Image.open(image_path).convert("RGB")
        image_tensor = preprocess_val(image).unsqueeze(0).to(device)  # Use open_clip's preprocessing
        with torch.no_grad():
            features = model.encode_image(image_tensor)  # Extract image features
        features = features / features.norm(p=2, dim=-1, keepdim=True)  # Normalize
        return features.cpu()

    clip_features_trainval = {}
    print("Extracting Clip Features")
    for image_name in trainval_images:
        image_path = image_name
        clip_features = extract_clip_features(image_path)

        # Save each feature tensor separately
        feature_path = os.path.join(clip_features_dir, f"{image_name.split('/')[-1].split('.')[0]}.pt")
        torch.save(clip_features, feature_path)

        clip_features_trainval[image_name] = feature_path


"""2. CLIP-based segmentation"""

# Loading CLIP features
clip_features_trainval = {
    image_name: torch.load(os.path.join(clip_features_dir, f"{image_name.split('/')[-1].split('.')[0]}.pt"))
    for image_name in trainval_images
}
print(f"Loaded {len(clip_features_trainval)} CLIP feature tensors.")

print("Saving Model here: Data/Output/Models2/PROMPTED_CLIP_DICE2_epoch_")

class SegmentationDataset(Dataset):
    def __init__(self, image_files, label_files, clip_features_dict, transform=None):
        self.image_files = image_files
```

```
        self.label_files = label_files
        self.clip_features_dict = clip_features_dict
        self.transform = transform

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

    def __getitem__(self, idx):
        img_path = self.image_files[idx]
        label_path = img_path.replace('.jpg', '.png').replace('color', 'label')

        # Compute heatmap path: assuming heatmaps are stored in a folder "heatmaps"
        # located in the same parent folder as "color"
        base_name = os.path.splitext(os.path.basename(img_path))[0]
        color_dir = os.path.dirname(img_path)
        parent_dir = os.path.dirname(color_dir)
        heatmap_dir = os.path.join(parent_dir, "heatmaps")
        heatmap_path = os.path.join(heatmap_dir, base_name + "_heatmap.png")

        # Loading image
        image = Image.open(img_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        else:
            image = transforms.ToTensor()(image)

        # Loading heatmap and converting to 1-channel tensor
        heatmap = Image.open(heatmap_path).convert('L')
        heatmap = transforms.ToTensor()(heatmap)

        # Concatenating image and heatmap to form a 4-channel input
        image = torch.cat([image, heatmap], dim=0)

        # Loading CLIP feature tensor
        image_name = img_path
        clip_feature = self.clip_features_dict[image_name]  # Shape: (512,)

        # Loading and processing label
        label = Image.open(label_path).convert('L')
        label = label.resize(preprocessedSize, Image.NEAREST)
        label = torch.tensor(np.array(label), dtype=torch.long)

        # Converting label classes
        label = torch.where(label == 38, 1, label)
        label = torch.where(label == 75, 2, label)
        label = torch.where(label == 255, 3, label)
        label = torch.where(label == 0, 0, label)

        return image, clip_feature, label

# Defining Transformations
transform = transforms.Compose([
    transforms.Resize(preprocessedSize, interpolation=Image.NEAREST),
    transforms.ToTensor()
])

# Loading Dataset & DataLoader
segmentation_dataset = SegmentationDataset(
    image_files=train_images,
    label_files=train_masks,
    clip_features_dict=clip_features_trainval,
    transform=None
)
segmentation_dataloader = DataLoader(segmentation_dataset, batch_size=batchSize, shuffle=True)

val_segmentation_dataset = SegmentationDataset(
    image_files=val_images,
    label_files=val_masks,
    clip_features_dict=clip_features_trainval,
    transform=None
)

val_segmentation_dataloader = DataLoader(val_segmentation_dataset, batch_size=batchSize, shuffle=True)

class SegmentationDecoder(nn.Module):
    def __init__(self, clip_feature_dim=512, num_classes=classesNum):
        super(SegmentationDecoder, self).__init__()

        # Project CLIP features
        self.clip_fc = nn.Linear(clip_feature_dim, 256)

        # CNN Feature Extractor (with 4 input channels: RGB + Heatmap)
        self.encoder = nn.Sequential(
            nn.Conv2d(4, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
```

```
            nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
        )

    def forward(self, image, clip_features):
        cnn_features = self.encoder(image)
        clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
        clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
        fusion = torch.cat([cnn_features, clip_features], dim=1)
        segmentation_output = self.decoder(fusion)
        return segmentation_output


def dice_loss(pred, target, smooth=1e-6):
    pred = torch.softmax(pred, dim=1)
    target = torch.nn.functional.one_hot(target, num_classes=pred.shape[1]).permute(0, 3, 1, 2)
    intersection = (pred * target).sum(dim=(2, 3))
    union = pred.sum(dim=(2, 3)) + target.sum(dim=(2, 3))
    dice = (2.0 * intersection + smooth) / (union + smooth)
    return 1 - dice.mean()

# Initialize Model, Loss, Optimizer
segmentation_model = SegmentationDecoder().to(device)
# segmentation_criterion = nn.CrossEntropyLoss()
# segmentation_criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs, masks)

segmentation_criterion = lambda outputs, masks: dice_loss(outputs, masks)

print("Loss: segmentation_criterion = lambda outputs, masks: nn.CrossEntropyLoss()(outputs, masks) + dice_loss(outputs
    , masks)")

segmentation_optimizer = optim.Adam(segmentation_model.parameters(), lr=learningRate)
scheduler = torch.optim.lr_scheduler.StepLR(segmentation_optimizer, step_size=5, gamma=0.5)
scaler = torch.cuda.amp.GradScaler()

# Training model
for epoch in tqdm(range(epochsNum), desc="Training Progress"):
    segmentation_model.train()
    running_loss = 0.0

    print(f"Epoch [{epoch+1}/{epochsNum}] Running:")

    for images, clip_features, labels in segmentation_dataloader:
        images, clip_features, labels = images.to(device), clip_features.to(device), labels.to(device)

        segmentation_optimizer.zero_grad()

        # Mixed precision training
        with torch.cuda.amp.autocast():
            outputs = segmentation_model(images, clip_features)
            loss = segmentation_criterion(outputs, labels)

        # Scaling loss and backpropagating
        scaler.scale(loss).backward()
        scaler.step(segmentation_optimizer)
        scaler.update()

        running_loss += loss.item()

    segmentation_model.eval()
    val_loss = 0
    with torch.no_grad():
        for images, clip_features, labels in val_segmentation_dataloader:
            images, clip_features, labels = images.to(device), clip_features.to(device), labels.to(device)
            outputs = segmentation_model(images, clip_features)
            loss = segmentation_criterion(outputs, labels)
            val_loss += loss.item()

    print(f'Epoch [{epoch+1}/{epochsNum}], Validation Loss : {val_loss / len(val_segmentation_dataloader)}')

    scheduler.step()

    avg_loss = running_loss / len(segmentation_dataloader)
    print(f"Epoch [{epoch+1}/{epochsNum}], Training Loss: {avg_loss:.4f}")
    print(f"Saved here: Data/Output/Models_DICEONLY/PROMPTED_CLIP_DiceOnly_epoch_{epoch+1}.pth")
    torch.save(segmentation_model.state_dict(), f'Data/Output/Models_DICEONLY/PROMPTED_CLIP_DiceOnly_epoch_{epoch+1}.
        pth')

print("Training complete!")
```

A.3.8 performance_evaluation.py: Evaluating Models on Test Set

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Performance evaluation
# Date: 21.03.2025


"""0. Preliminary code"""
print("PERFORMANCE EVALUATION:")

# Setting code-behaviour varaibles
classNum = 4
```

```
exludedClasses = [3]
class_to_pixel = {0: 0, 1: 38, 2: 75, 3: 255}
pixel_to_class = {v: k for k, v in class_to_pixel.items()}

image_dir = 'Data/Input/Test/color/'
label_dir = 'Data/Input/Test/label/'
heatmap_dir = 'Data/Input/Test/heatmaps/'

# (to skip testing of a model, set any relevant desired epoch to 0)
models_dir = 'Data/Output/Models_DICEONLY/'
unet_epoch = 0 # CROSS : 29, DICE: 22 path:  /home/s2677266/CVis/Data/Output/Models/UNET_CROSS_2_EPOCH_
autoencoder_epoch = 0 # 95
segmentation_decoder_epoch = 0 #42
clip_decoder_epoch = 0 #42
prompted_clip_decoder_epoch = 0  #42

pathName = f'Data/Output/Models_DICEONLY/Not_Frozen_Val_Segmentation_95_Decoder_DiceOnly_epoch_96.pth'

# Importing useful libraries
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from PIL import Image
from torchvision import transforms
from tqdm import tqdm
import open_clip

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
image_files = [f for f in os.listdir(image_dir) if f.endswith(('.jpg', '.png'))]

# Defining data loading
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
])
def load_image_and_label(image_path, label_path):
    """Load an image and its corresponding label, apply necessary transformations."""
    image = Image.open(image_path).convert('RGB')
    label = Image.open(label_path).convert('L') # label(grayscale)
    input_tensor = transform(image).unsqueeze(0).to(device)
    label_array = np.array(label)
    return input_tensor, label_array

# Defining IoU function
def compute_iou(pred, target, num_classes=classNum, exclude_classes=[]):
    """
    Compute Intersection over Union (IoU) for multi-class segmentation using accumulated dataset-wide counts,
    with the option to exclude specific classes from the computation.

    :param pred: Predicted segmentation map (H, W), with class indices.
    :param target: Ground truth segmentation map (H, W), with class indices.
    :param num_classes: Number of classes in segmentation.
    :param exclude_classes: List of class indices to exclude from IoU calculation.
    :return: IoU scores per class and mean IoU.
    """

    intersection_counts = np.zeros(num_classes)
    union_counts = np.zeros(num_classes)

    for cls in range(num_classes):
        if cls in exclude_classes:
            continue

        pred_inds = pred == cls
        target_inds = target == cls

        intersection_counts[cls] += torch.logical_and(pred_inds, target_inds).sum().item()
        union_counts[cls] += torch.logical_or(pred_inds, target_inds).sum().item()

    iou_per_class = [
        intersection_counts[cls] / union_counts[cls] if union_counts[cls] > 0 else float('nan')
        for cls in range(num_classes)
        if cls not in exclude_classes
    ]

    mean_iou = np.nanmean(iou_per_class) if iou_per_class else float('nan')

    return iou_per_class, mean_iou

# Defining Dice coefficient function
def compute_dice_coefficient(pred, target, num_classes=classNum, exclude_classes=[]):
    """
    Compute Dice Coefficient for multi-class segmentation,
    with the option to exclude specific classes from the computation.

    :param pred: Predicted segmentation map (H, W), with class indices.
    :param target: Ground truth segmentation map (H, W), with class indices.
    :param num_classes: Number of classes in segmentation.
    :param exclude_classes: List of class indices to exclude from Dice calculation.
    :return: Dice scores per class and mean Dice score.
    """

    dice_per_class = []
```

```
    for cls in range(num_classes):
        if cls in exclude_classes:
            continue

        pred_inds = pred == cls
        target_inds = target == cls
        intersection = 2.0 * torch.logical_and(pred_inds, target_inds).sum().item()
        union = pred_inds.sum().item() + target_inds.sum().item()

        if union == 0:
            dice_per_class.append(float('nan'))
        else:
            dice_per_class.append(intersection / union)

    mean_dice = np.nanmean(dice_per_class) if dice_per_class else float('nan')

    return dice_per_class, mean_dice

# Defining pixel accuracy function
def compute_pixel_accuracy(pred, target, exclude_classes=[]):
    """
    Compute Pixel Accuracy,
    with the option to exclude specific classes from the computation.

    :param pred: Predicted segmentation map (H, W), with class indices.
    :param target: Ground truth segmentation map (H, W), with class indices.
    :param exclude_classes: List of class indices to exclude from accuracy calculation.
    :return: Pixel accuracy score.
    """

    exclude_mask = torch.zeros_like(target, dtype=torch.bool)
    for cls in exclude_classes:
        exclude_mask |= (target == cls)

    valid_pixels = ~exclude_mask
    correct = (pred[valid_pixels] == target[valid_pixels]).sum().item()
    total = valid_pixels.sum().item()

    return correct / total if total > 0 else float('nan')


"""1. UNet testing"""
if unet_epoch!=0:
    print("\nTesting␣UNet...")

    # Loading model
    class UNet(nn.Module):
        def __init__(self, num_classes=classNum):
            super(UNet, self).__init__()

            # Encoder
            self.enc1 = self.conv_block(3, 64)
            self.pool1 = nn.MaxPool2d(2)

            self.enc2 = self.conv_block(64, 128)
            self.pool2 = nn.MaxPool2d(2)

            self.enc3 = self.conv_block(128, 256)
            self.pool3 = nn.MaxPool2d(2)

            self.enc4 = self.conv_block(256, 512)

            # Decoder
            self.up5 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
            self.dec5 = self.conv_block(512, 256)

            self.up6 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
            self.dec6 = self.conv_block(256, 128)

            self.up7 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
            self.dec7 = self.conv_block(128, 64)

            self.final_conv = nn.Conv2d(64, num_classes, kernel_size=1) # output layer

        def conv_block(self, in_channels, out_channels):
            return nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Dropout(0.2)
            )

        def forward(self, x):
            # Encoder
            c1 = self.enc1(x)
            p1 = self.pool1(c1)

            c2 = self.enc2(p1)
            p2 = self.pool2(c2)

            c3 = self.enc3(p2)
```

```
            p3 = self.pool3(c3)

            c4 = self.enc4(p3)

            # Decoder
            u5 = self.up5(c4)
            u5 = torch.cat([u5, c3], dim=1)
            c5 = self.dec5(u5)

            u6 = self.up6(c5)
            u6 = torch.cat([u6, c2], dim=1)
            c6 = self.dec6(u6)

            u7 = self.up7(c6)
            u7 = torch.cat([u7, c1], dim=1)
            c7 = self.dec7(u7)

            outputs = self.final_conv(c7)

            return outputs

    model = UNet().to(device)
    model.load_state_dict(torch.load(f'/home/s2677266/CVis/Data/Output/Models_DICEONLY/UNET_DiceOnly_EPOCH_88.pth',
        map_location=device))
    model.eval()

    print("(model loaded)")

    # Evaluating image by image
    all_intersection = np.zeros(3)
    all_union = np.zeros(3)
    all_dice = []
    all_pixel_accs = []
    for image_file in tqdm(image_files, desc="Processing images"):
        image_path = os.path.join(image_dir, image_file)
        label_path = os.path.join(label_dir, image_file.replace(".jpg", ".png"))
        input_tensor, label_image = load_image_and_label(image_path, label_path)

        # Inference
        with torch.no_grad():
            output_tensor = model(input_tensor)
        output_tensor = F.softmax(output_tensor, dim=1) # softmax to get predicted class indices
        predicted_classes = torch.argmax(output_tensor, dim=1).squeeze(0).cpu()
        predicted_classes = F.interpolate( # resizing predicted mask to original label size
            predicted_classes.unsqueeze(0).unsqueeze(0).float(),
            size=(label_image.shape[0], label_image.shape[1]),
            mode='nearest'
        ).squeeze(0).squeeze(0).to(dtype=torch.uint8)
        label_tensor = torch.from_numpy(label_image).to(dtype=torch.uint8) # convertion of label image to class
            indices
        label_class_map = torch.zeros_like(label_tensor, dtype=torch.uint8)
        for pixel_value, class_idx in pixel_to_class.items():
            label_class_map[label_tensor == pixel_value] = class_idx

        # Computing metrics
        iou_per_class, _ = compute_iou(predicted_classes, label_class_map, exclude_classes=exludedClasses)
        _, mean_dice = compute_dice_coefficient(predicted_classes, label_class_map, exclude_classes = exludedClasses)
        pixel_acc = compute_pixel_accuracy(predicted_classes, label_class_map, exclude_classes = exludedClasses)

        for cls in range(classNum):
            if cls in exludedClasses:
                continue
            if not np.isnan(iou_per_class[cls]):
                all_intersection[cls] += torch.logical_and(predicted_classes == cls, label_class_map == cls).sum().
                    item()
                all_union[cls] += torch.logical_or(predicted_classes == cls, label_class_map == cls).sum().item()

        all_dice.append(mean_dice)
        all_pixel_accs.append(pixel_acc)

    # Displaying overall scores
    final_iou_per_class = [
        all_intersection[cls] / all_union[cls] if all_union[cls] > 0 else float('nan')
        for cls in range(classNum) if cls not in exludedClasses
    ]
    overall_mean_iou = np.nanmean(final_iou_per_class)
    overall_mean_dice = np.nanmean(all_dice)
    overall_pixel_accuracy = np.mean(all_pixel_accs)

    print(f"Per-Class IoU: {final_iou_per_class}")
    print(f"Overall Mean IoU: {overall_mean_iou:.4f}")
    print(f"Overall Mean Dice Coefficient: {overall_mean_dice:.4f}")
    print(f"Overall Pixel Accuracy: {overall_pixel_accuracy:.4f}")


"""2. Autoencoder testing"""
if autoencoder_epoch*segmentation_decoder_epoch!=0:
    print("\nTesting Autoencoder...")

    # Loading model
    class Autoencoder(nn.Module):
        def __init__(self):
            super(Autoencoder, self).__init__()

            # Encoder
```

```
            self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, stride=2),  # /2 size
            nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
        )

            # Decoder
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(128, 256, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
                nn.LeakyReLU(0.2),
                nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
                nn.LeakyReLU(0.2),
                nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),  # x2 size
                nn.LeakyReLU(0.2),
                nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
                nn.LeakyReLU(0.2),
                nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1), # output layer
                nn.Sigmoid()
            )

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

    class ResidualBlock(nn.Module):
        def __init__(self, in_channels, out_channels):
            super(ResidualBlock, self).__init__()
            self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1)
            self.bn1 = nn.BatchNorm2d(out_channels)
            self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1)
            self.bn2 = nn.BatchNorm2d(out_channels)
            self.relu = nn.ReLU(inplace=True)

        def forward(self, x):
            residual = x
            x = self.relu(self.bn1(self.conv1(x)))
            x = self.bn2(self.conv2(x))
            x += residual # skip connection
            return self.relu(x)

    class SegmentationDecoder(nn.Module):
        def __init__(self, encoder):
            super(SegmentationDecoder, self).__init__()

            for param in encoder.parameters():
                param.requires_grad = True

            self.encoder = encoder

            # Decoder (with residual blocks and skip connections)
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(128, 128, kernel_size=3, stride=2, padding=1, output_padding=1), # x2 size
                ResidualBlock(128, 128),
                nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, output_padding=1),  # x2 size
                ResidualBlock(64, 64),
                nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, output_padding=1),   # x2 size
                ResidualBlock(32, 32),
                nn.Conv2d(32, 32, kernel_size=3, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(inplace=True),
                nn.Conv2d(32, 4, kernel_size=1)
            )
        def forward(self, x):
            x = self.encoder(x)
            x = self.decoder(x)
            return x

    autoencoder = Autoencoder().to(device)
    autoencoder.load_state_dict(torch.load(models_dir+f'updated_complex_working_autoencoder_epoch_{autoencoder_epoch}.
         pth', map_location=device))
    autoencoder.eval()

    # Loading segmentation model
    segmentation_model = SegmentationDecoder(autoencoder.encoder).to(device)
    segmentation_model.load_state_dict(torch.load(pathName, map_location=device))
    segmentation_model.eval()

    print("(model␣loaded)")

    # Evaluating image by image
    all_intersection = np.zeros(3)
    all_union = np.zeros(3)
    all_dice = []
    all_pixel_accs = []
    for image_file in tqdm(image_files, desc="Processing␣images"):
        image_path = os.path.join(image_dir, image_file)
```

```
        label_path = os.path.join(label_dir, image_file.replace(".jpg", ".png"))
        input_tensor, label_image = load_image_and_label(image_path, label_path)

        # Inference
        with torch.no_grad():
            output_tensor = segmentation_model(input_tensor)
        output_tensor = F.softmax(output_tensor, dim=1)
        predicted_classes = torch.argmax(output_tensor, dim=1).squeeze(0).cpu()
        predicted_classes = F.interpolate( # resizing predicted mask to original label size
            predicted_classes.unsqueeze(0).unsqueeze(0).float(),
            size=(label_image.shape[0], label_image.shape[1]),
            mode='nearest'
        ).squeeze(0).squeeze(0).to(dtype=torch.uint8)
        label_tensor = torch.tensor(label_image, dtype=torch.uint8)
        label_class_map = torch.zeros_like(label_tensor)
        for pixel_value, class_idx in pixel_to_class.items():
            label_class_map[label_tensor == pixel_value] = class_idx

        # Computing metrics
        iou_per_class, _ = compute_iou(predicted_classes, label_class_map, exclude_classes=exludedClasses)
        _, mean_dice = compute_dice_coefficient(predicted_classes, label_class_map, exclude_classes = exludedClasses)
        pixel_acc = compute_pixel_accuracy(predicted_classes, label_class_map, exclude_classes = exludedClasses)

        for cls in range(classNum):
            if cls in exludedClasses:
                continue
            if not np.isnan(iou_per_class[cls]):
                all_intersection[cls] += torch.logical_and(predicted_classes == cls, label_class_map == cls).sum().
                    item()
                all_union[cls] += torch.logical_or(predicted_classes == cls, label_class_map == cls).sum().item()

        all_dice.append(mean_dice)
        all_pixel_accs.append(pixel_acc)

    # Displaying overall scores
    final_iou_per_class = [
        all_intersection[cls] / all_union[cls] if all_union[cls] > 0 else float('nan')
        for cls in range(classNum) if cls not in exludedClasses
    ]
    overall_mean_iou = np.nanmean(final_iou_per_class)
    overall_mean_dice = np.nanmean(all_dice)
    overall_pixel_accuracy = np.mean(all_pixel_accs)

    print(f"Per-Class␣IoU:␣{final_iou_per_class}")
    print(f"Overall␣Mean␣IoU:␣{overall_mean_iou:.4f}")
    print(f"Overall␣Mean␣Dice␣Coefficient:␣{overall_mean_dice:.4f}")
    print(f"Overall␣Pixel␣Accuracy:␣{overall_pixel_accuracy:.4f}")


"""3. CLIP testing"""
if clip_decoder_epoch!=0:
    print("\nTesting␣CLIP...")

    # Loading model
    class ClipDecoder(nn.Module):
        def __init__(self, clip_feature_dim=512, num_classes=classNum):
            super(ClipDecoder, self).__init__()

            # Project CLIP features
            self.clip_fc = nn.Linear(clip_feature_dim, 256)

            # CNN Feature Extractor
            self.encoder = nn.Sequential(
                nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
            )

            # Decoder
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
                nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
            )

        def forward(self, image, clip_features):
            cnn_features = self.encoder(image)
            clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
            clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
            fusion = torch.cat([cnn_features, clip_features], dim=1)
            segmentation_output = self.decoder(fusion)
            return segmentation_output

    model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
    model = model.to(device)
    tokenizer = open_clip.get_tokenizer("ViT-B-32")

    # Loading segmentation model
    clip_segmentation_model = ClipDecoder().to(device)
    # clip_segmentation_model.load_state_dict(torch.load(models_dir+f'New_clip_segmentation_model_epoch_{
        clip_decoder_epoch}.pth', map_location=device))
```

```
    clip_segmentation_model.load_state_dict(torch.load(models_dir+f'CLIP_DiceOnly_epoch_{clip_decoder_epoch}.pth',
        map_location=device))
    clip_segmentation_model.eval()

    print("(model loaded)")

    # Evaluating image by image
    all_intersection = {cls: 0 for cls in range(classNum)}
    all_union = {cls: 0 for cls in range(classNum)}
    all_dice = []
    all_pixel_accs = []
    for image_file in tqdm(image_files, desc="Processing images"):
        image_path = os.path.join(image_dir, image_file)
        label_path = os.path.join(label_dir, image_file.replace('.jpg', '.png'))
        _, label_image = load_image_and_label(image_path, label_path)

        # CLIP feature extraction
        image_clip = Image.open(image_path).convert("RGB")
        image_clip_tensor = preprocess_val(image_clip).unsqueeze(0).to(device)
        with torch.no_grad():
            clip_features = model.encode_image(image_clip_tensor)
        clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)

        # Inference
        image_seg = Image.open(image_path).convert("RGB")
        image_tensor = transforms.ToTensor()(image_seg).unsqueeze(0).to(device)
        with torch.no_grad():
            output_tensor = clip_segmentation_model(image_tensor, clip_features)
        pred_mask = output_tensor.squeeze(0) # output conversion to predicted segmentation mask
        pred_mask = torch.argmax(pred_mask, dim=0).cpu()
        pred_mask = F.interpolate( # resizing predicted mask to original label size
            pred_mask.unsqueeze(0).unsqueeze(0).float(),
            size=(label_image.shape[0], label_image.shape[1]),
            mode='nearest'
        ).squeeze(0).squeeze(0).to(dtype=torch.uint8)
        label_tensor = torch.from_numpy(label_image).to(dtype=torch.uint8)
        gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)
        for pixel_value, class_idx in pixel_to_class.items():
            gt_mask_class[label_tensor == pixel_value] = class_idx

        # Computing metrics
        iou_per_class, _ = compute_iou(pred_mask, gt_mask_class, classNum, exclude_classes=exludedClasses)
        _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
        mean_dice = float(mean_dice)
        pixel_acc = compute_pixel_accuracy(pred_mask, gt_mask_class, exclude_classes=exludedClasses)

        for cls in range(classNum):
            if cls in exludedClasses:
                continue
            if not np.isnan(iou_per_class[cls]):
                all_intersection[cls] += torch.logical_and(pred_mask == cls, gt_mask_class == cls).sum().item()
                all_union[cls] += torch.logical_or(pred_mask == cls, gt_mask_class == cls).sum().item()

        all_dice.append(mean_dice)
        all_pixel_accs.append(pixel_acc)

    # Displaying overall scores
    final_iou_per_class = [
        all_intersection[cls] / all_union[cls] if all_union[cls] > 0 else float('nan')
        for cls in range(classNum) if cls not in exludedClasses
    ]
    overall_mean_iou = np.nanmean(final_iou_per_class)
    overall_mean_dice = np.nanmean(all_dice)
    overall_pixel_accuracy = np.mean(all_pixel_accs)

    print(f"IoU per class: {final_iou_per_class}")
    print(f"Overall Mean IoU: {overall_mean_iou:.4f}")
    print(f"Overall Mean Dice Coefficient: {overall_mean_dice:.4f}")
    print(f"Overall Pixel Accuracy: {overall_pixel_accuracy:.4f}")


"""4. Prompted CLIP testing"""
if prompted_clip_decoder_epoch!=0:
    print("\nTesting prompted CLIP...")

    # Loading model
    class PromptedClipDecoder(nn.Module):
        def __init__(self, clip_feature_dim=512, num_classes=classNum):
            super(PromptedClipDecoder, self).__init__()

            # Project CLIP features
            self.clip_fc = nn.Linear(clip_feature_dim, 256)

            # CNN Feature Extractor (with 4 input channels: RGB + Heatmap)
            self.encoder = nn.Sequential(
                nn.Conv2d(4, 64, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
            )

            # Decoder
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
```

```
                nn.ReLU(),
                nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
                nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
            )

        def forward(self, image, clip_features):
            cnn_features = self.encoder(image)
            clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
            clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
            fusion = torch.cat([cnn_features, clip_features], dim=1)
            segmentation_output = self.decoder(fusion)
            return segmentation_output

    model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
    model = model.to(device)
    tokenizer = open_clip.get_tokenizer("ViT-B-32")

    # Loading segmentation model
    prompted_clip_segmentation_model = PromptedClipDecoder().to(device)
    prompted_clip_segmentation_model.load_state_dict(torch.load(models_dir+f'PROMPTED_CLIP_DiceOnly_epoch_␣␣{
        prompted_clip_decoder_epoch}.pth', map_location=device))
    prompted_clip_segmentation_model.eval()

    print("(model␣loaded)")

    # Evaluating image by image
    all_intersection = {cls: 0 for cls in range(classNum)}
    all_union = {cls: 0 for cls in range(classNum)}
    all_dice = []
    all_pixel_accs = []
    for image_file in tqdm(image_files, desc="Processing␣images"):
        image_path = os.path.join(image_dir, image_file)
        label_path = os.path.join(label_dir, image_file.replace('.jpg', '.png'))
        heatmap_path = os.path.join(heatmap_dir, image_file.split('.')[0]+'_heatmap.png')
        _, label_image = load_image_and_label(image_path, label_path)

        # CLIP feature extraction
        image_clip = Image.open(image_path).convert("RGB")
        image_clip_tensor = preprocess_val(image_clip).unsqueeze(0).to(device)
        with torch.no_grad():
            clip_features = model.encode_image(image_clip_tensor)
        clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)

        # Inference
        image_seg = Image.open(image_path).convert("RGB").resize((500, 500))
        heatmap = Image.open(heatmap_path).convert('L')
        heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
        image_tensor = transforms.ToTensor()(image_seg).unsqueeze(0).to(device)
        image_tensor = torch.cat([image_tensor, heatmap], dim=1)
        with torch.no_grad():
            output_tensor = prompted_clip_segmentation_model(image_tensor, clip_features)
        pred_mask = output_tensor.squeeze(0) # output conversion to predicted segmentation mask
        pred_mask = torch.argmax(pred_mask, dim=0).cpu()
        pred_mask = F.interpolate( # resizing predicted mask to original label size
            pred_mask.unsqueeze(0).unsqueeze(0).float(),
            size=(label_image.shape[0], label_image.shape[1]),
            mode='nearest'
        ).squeeze(0).squeeze(0).to(dtype=torch.uint8)
        label_tensor = torch.from_numpy(label_image).to(dtype=torch.uint8)
        gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)
        for pixel_value, class_idx in pixel_to_class.items():
            gt_mask_class[label_tensor == pixel_value] = class_idx

        # Computing metrics
        iou_per_class, _ = compute_iou(pred_mask, gt_mask_class, classNum, exclude_classes=exludedClasses)
        _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
        mean_dice = float(mean_dice)
        pixel_acc = compute_pixel_accuracy(pred_mask, gt_mask_class, exclude_classes=exludedClasses)

        for cls in range(classNum):
            if cls in exludedClasses:
                continue
            if not np.isnan(iou_per_class[cls]):
                all_intersection[cls] += torch.logical_and(pred_mask == cls, gt_mask_class == cls).sum().item()
                all_union[cls] += torch.logical_or(pred_mask == cls, gt_mask_class == cls).sum().item()

        all_dice.append(mean_dice)
        all_pixel_accs.append(pixel_acc)

    # Displaying overall scores
    final_iou_per_class = [
        all_intersection[cls] / all_union[cls] if all_union[cls] > 0 else float('nan')
        for cls in range(classNum) if cls not in exludedClasses
    ]
    overall_mean_iou = np.nanmean(final_iou_per_class)
    overall_mean_dice = np.nanmean(all_dice)
    overall_pixel_accuracy = np.mean(all_pixel_accs)

    print(f"IoU␣per␣class:␣{final_iou_per_class}")
    print(f"Overall␣Mean␣IoU:␣{overall_mean_iou:.4f}")
    print(f"Overall␣Mean␣Dice␣Coefficient:␣{overall_mean_dice:.4f}")
    print(f"Overall␣Pixel␣Accuracy:␣{overall_pixel_accuracy:.4f}")
```

A.3.9 robustness_exploration.py: Robustness Exploration

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Robustness exploration on the prompted CLIP model
# Date: 31.03.2025


"""0. Preliminary code"""

print("ROBUSTNESS EXPLORATION WITH DIFFERENT PERTURBATION TYPES (PROMPTED CLIP):")

# Setting code-behaviour varaibles
classNum = 4
pixel_to_class = {0: 0, 38: 1, 75: 2, 255: 3}
exludedClasses = [3]    # classes excludeed from metric calculation (3 not to consider the outline)
maxImages = -1          # number of input images to load from the defined set (0 to consider them all)
digits = 3              # digits to approximate decimal results in, when printed

# Defining paths
model_dir = f'Data/Output/Models2/PROMPTED_CLIP_DICE2_epoch_42.pth'
images_dir = 'Data/Input/Test/color/'
masks_dir = 'Data/Input/Test/label/'
heatmaps_dir = 'Data/Input/Test/heatmaps/'
output_plots_dir = 'Data/Output/Visuals/Prompted_CLIP_Robustness/'

# Importing useful libraries
import os
import numpy as np
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import open_clip
import matplotlib.pyplot as plt
import cv2
from skimage.util import random_noise

device = "cuda" if torch.cuda.is_available() else "cpu"

image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]

# Loading pre-trained prompted CLIP model
CLIP_model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
CLIP_model = CLIP_model.to(device)
tokenizer = open_clip.get_tokenizer("ViT-B-32")

class PromptedClipDecoder(nn.Module):
    def __init__(self, clip_feature_dim=512, num_classes=classNum):
        super(PromptedClipDecoder, self).__init__()

        # Project CLIP features
        self.clip_fc = nn.Linear(clip_feature_dim, 256)

        # CNN Feature Extractor (with 4 input channels: RGB + Heatmap)
        self.encoder = nn.Sequential(
            nn.Conv2d(4, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
            nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
        )

    def forward(self, image, clip_features):
        cnn_features = self.encoder(image)
        clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
        clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
        fusion = torch.cat([cnn_features, clip_features], dim=1)
        segmentation_output = self.decoder(fusion)
        return segmentation_output

# Loading segmentation model
model = PromptedClipDecoder().to(device)
model.load_state_dict(torch.load(model_dir, map_location=device))
model.eval()

def compute_dice_coefficient(pred, target, num_classes=classNum, exclude_classes=[]):
    """
    Compute Dice Coefficient for multi-class segmentation with the option to exclude specific classes.

    :param pred: Predicted segmentation map (H, W), with class indices.
    :param target: Ground truth segmentation map (H, W), with class indices.
    :param num_classes: Number of classes in segmentation.
    :param exclude_classes: List of class indices to exclude from Dice calculation.
    :return: Dice scores per class and mean Dice score.
    """

```

```
    dice_per_class = []
    for cls in range(num_classes):
        if cls in exclude_classes:
            continue

        pred_inds = pred == cls
        target_inds = target == cls
        intersection = 2.0 * torch.logical_and(pred_inds, target_inds).sum().item()
        union = pred_inds.sum().item() + target_inds.sum().item()

        if union == 0:
            dice_per_class.append(float('nan'))
        else:
            dice_per_class.append(intersection / union)

    mean_dice = np.nanmean(dice_per_class) if dice_per_class else float('nan')

    return dice_per_class, mean_dice


"""a. Gaussian noise"""

def apply_gaussian_noise(image, std_dev):
    """Applies Gaussian noise to an image with a given standard deviation."""
    noise = np.random.normal(0, std_dev, image.shape)
    noisy_image = np.clip(image + noise, 0, 255).astype(np.uint8)
    return noisy_image

def evaluate_gaussian_noise(model, images_dir, masks_dir, std_dev_levels, device, save_as:str="a_gaussian_noise"):
    """Evaluates the model with Gaussian noise perturbation and plots Dice score vs. noise level."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for std_dev in std_dev_levels:
        total_dice = []
        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            noisy_image = apply_gaussian_noise(image, std_dev)
            image_tensor = transforms.ToTensor()(Image.fromarray(noisy_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)

            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(std_dev_levels, dice_scores, marker='o')
    plt.xlabel("Gaussian Noise Standard Deviation")
    plt.ylabel("Mean Dice Score")
    plt.title("Gaussian Noise")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\na) Gaussian Noise...", end=' ')
std_dev_levels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
dice_scores = evaluate_gaussian_noise(model, images_dir, masks_dir, std_dev_levels, device)
print("Done!")
print(" -"*49)
print("| st. deviations  =", " | ".join([f"{value:{digits+2}}" for value in std_dev_levels]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")
```

```
188
189 """b. Gaussian blurring"""
190
191 def apply_gaussian_blur(image, ksize):
192     """Applies Gaussian blur to an image with a given kernel size."""
193     if ksize % 2 == 0:
194         ksize += 1
195     kernel = np.array([
196         [1/16, 2/16, 1/16],
197         [2/16, 4/16, 2/16],
198         [1/16, 2/16, 1/16]
199     ])
200     blurred_image = cv2.filter2D(image, -1, kernel)
201     return blurred_image
202
203 def evaluate_gaussian_blur(model, images_dir, masks_dir, max_blur_level, device, save_as:str="b_gaussian_blurring"):
204     """Evaluates the model with Gaussian blurring perturbation and plots Dice score vs. blur level."""
205     image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
206     dice_scores = []
207
208     for blur_level in range(max_blur_level + 1):
209         total_dice = []
210
211         counter = 0
212         for img_file in image_files:
213             counter += 1
214             if counter == maxImages:
215                 break
216             img_path = os.path.join(images_dir, img_file)
217             mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
218             heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')
219
220             blurred_image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
221             for i in range(blur_level):
222                 blurred_image = apply_gaussian_blur(blurred_image, 3)
223             image_tensor = transforms.ToTensor()(Image.fromarray(blurred_image)).unsqueeze(0).to(device)
224             heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
225             heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
226             image_tensor = torch.cat([image_tensor, heatmap], dim=1)
227
228             gt_mask = Image.open(mask_path).convert("L")
229             gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
230             label_array = np.array(gt_mask)
231
232             label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
233             gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)
234
235             for pixel_value, class_idx in pixel_to_class.items():
236                 gt_mask_class[label_tensor == pixel_value] = class_idx
237
238             image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)
239
240             with torch.no_grad():
241                 clip_features = CLIP_model.encode_image(image_clip_tensor)
242                 clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
243                 model_output = model(image_tensor, clip_features)
244                 pred_mask = model_output.squeeze(0)
245                 pred_mask = torch.argmax(pred_mask, dim=0).cpu()
246
247             _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)
248             total_dice.append(mean_dice)
249
250         dice_scores.append(np.mean(total_dice))
251
252     plt.figure()
253     plt.plot(range(max_blur_level + 1), dice_scores, marker='o')
254     plt.xlabel("Gaussian Blur Level (number of iterative 3x3 mask applications)")
255     plt.ylabel("Mean Dice Score")
256     plt.title("Gaussian Blurring")
257     plt.grid()
258     plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")
259
260     return dice_scores
261
262 print("\nb) Gaussian Blurring...", end=' ')
263 max_blur_level = 9
264 dice_scores = evaluate_gaussian_blur(model, images_dir, masks_dir, max_blur_level, device)
265 print("Done!")
266 print(" -"*49)
267 print("| blurring steps =", " | ".join([f"{value:{digits+2}}" for value in list(range(max_blur_level+1))]), "|")
268 print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
269 print(" -"*49,"\n")
270
271
272 """c. Contrast increase"""
273
274 def apply_contrast_change(image, contrast_factor):
275     """Applies contrast increase to an image by multiplying pixel values by a contrast factor."""
276     contrast_image = image * contrast_factor
277     contrast_image = np.clip(contrast_image, 0, 255).astype(np.uint8)
278     return contrast_image
279
280 def evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device, save_as:str="c_contrast_increase"
      , invert_axis:bool=False):
281     """Evaluates the model with increased contrast perturbation and plots Dice score vs. contrast factor."""
```

```
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for contrast_factor in contrast_factors:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            contrast_image = apply_contrast_change(image, contrast_factor)
            image_tensor = transforms.ToTensor()(Image.fromarray(contrast_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(contrast_factors, dice_scores, marker='o')
    plt.xlabel("Contrast Factor")
    plt.ylabel("Mean Dice Score")
    plt.title("Contrast Decrease" if invert_axis else "Contrast Increase")
    plt.grid()
    if invert_axis: plt.gca().invert_xaxis()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\nc) Contrast Increase...", end=' ')
contrast_factors = [1.0, 1.01, 1.02, 1.03, 1.04, 1.05, 1.1, 1.15, 1.2, 1.25]
dice_scores = evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device)
print("Done!")
print("-"*49)
print("| contrast factor =", " | ".join([f"{value:{digits+2}}" for value in contrast_factors]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("-"*49,"\n")


"""d. Contrast decrease"""
# (using contrast increase functions)

print("\nd) Contrast Decrease...", end=' ')
contrast_factors = [1.0, 0.95, 0.90, 0.85, 0.80, 0.60, 0.40, 0.30, 0.20, 0.10]
dice_scores = evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device, save_as="
    d_contrast_decrease", invert_axis=True)
print("Done!")
print("-"*49)
print("| contrast factor =", " | ".join([f"{value:{digits+2}}" for value in contrast_factors]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("-"*49,"\n")


"""e. Brightness Increase"""

def apply_brightness_change(image, brightness_offset):
    """Applies brightness change by adding an offset to pixel values."""
    bright_image = image.astype(np.int16) + brightness_offset
    bright_image = np.clip(bright_image, 0, 255).astype(np.uint8)
    return bright_image

def evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device, save_as:str="
    e_brightness_increase", invert_axis:bool=False):
    """Evaluates the model with brightness perturbation and plots Dice score vs. brightness offset."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []
```

```
    for brightness_offset in brightness_offsets:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            bright_image = apply_brightness_change(image, brightness_offset)
            image_tensor = transforms.ToTensor()(Image.fromarray(bright_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(brightness_offsets, dice_scores, marker='o')
    plt.xlabel("Brightness␣Offset")
    plt.ylabel("Mean␣Dice␣Score")
    plt.title("Brightness␣Decrease" if invert_axis else "Brightness␣Increase")
    plt.grid()
    if invert_axis: plt.gca().invert_xaxis()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\ne)␣Brightness␣Increase...", end='␣')
brightness_offsets = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
dice_scores = evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device)
print("Done!")
print("␣-"*49)
print("|␣bright.␣offset␣␣=", "␣|␣".join([f"{value:{digits+2}}" for value in brightness_offsets]), "|")
print("|␣mean␣Dice␣score␣=", "␣|␣".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("␣-"*49,"\n")


"""f. Brightness Decrease"""
# (using brightness increase functions)

print("\nf)␣Brightness␣Decrease...", end='␣')
brightness_offsets = [0, -5, -10, -15, -20, -25, -30, -35, -40, -45]
dice_scores = evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device, save_as="
     f_brightness_decrease", invert_axis=True)
print("Done!")
print("␣-"*49)
print("|␣bright.␣offset␣␣=", "␣|␣".join([f"{value:{digits+2}}" for value in brightness_offsets]), "|")
print("|␣mean␣Dice␣score␣=", "␣|␣".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("␣-"*49,"\n")


"""g. Square occlusion"""

def apply_occlusion(image, square_size):
    """Applies occlusion by replacing a random square region with black pixels."""
    h, w, _ = image.shape
    if square_size > 0:
        x = np.random.randint(0, w - square_size + 1)
        y = np.random.randint(0, h - square_size + 1)
        image[y:y+square_size, x:x+square_size] = 0
    return image

def evaluate_occlusion(model, images_dir, masks_dir, occlusion_sizes, device, save_as:str="g_square_occlusion"):
    """Evaluates the model under increasing occlusion perturbation."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for square_size in occlusion_sizes:
```

```
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            occluded_image = apply_occlusion(image, square_size)
            image_tensor = transforms.ToTensor()(Image.fromarray(occluded_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(occlusion_sizes, dice_scores, marker='o')
    plt.xlabel("Occlusion Size (edge length in pixels)")
    plt.ylabel("Mean Dice Score")
    plt.title("Square Occlusion")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")
    return dice_scores

print("\ng) Square Occlusion...", end=' ')
occlusion_sizes = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
dice_scores = evaluate_occlusion(model, images_dir, masks_dir, occlusion_sizes, device)
print("Done!")
print("-" * 49)
print("| oc. square size =", " | ".join([f"{value:{digits+2}}" for value in occlusion_sizes]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("-" * 49, "\n")


"""h. Salt&Pepper Noise"""

def apply_salt_and_pepper_noise(image, noise_amount):
    """Applies salt and pepper noise to an image."""
    noisy_image = random_noise(image, mode='s&p', amount=noise_amount)
    noisy_image = (noisy_image * 255).astype(np.uint8)
    return noisy_image

def evaluate_salt_and_pepper_noise(model, images_dir, masks_dir, noise_levels, device, save_as:str="h_SaP_noise"):
    """Evaluates the model with salt-and-pepper noise perturbation and plots Dice score vs. noise level."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for noise_amount in noise_levels:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            noisy_image = apply_salt_and_pepper_noise(image, noise_amount)
            image_tensor = transforms.ToTensor()(Image.fromarray(noisy_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
```

```
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(noise_levels, dice_scores, marker='o')
    plt.xlabel("Salt␣&␣Pepper␣Noise␣Level")
    plt.ylabel("Mean␣Dice␣Score")
    plt.title("Salt␣&␣Pepper␣Noise")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\nh)␣Salt␣&␣Pepper␣Noise...", end='␣')
noise_levels = [0.00, 0.02, 0.04, 0.06, 0.08, 0.10, 0.12, 0.14, 0.16, 0.18]
dice_scores = evaluate_salt_and_pepper_noise(model, images_dir, masks_dir, noise_levels, device)
print("Done!")
print("␣-"*49)
print("|␣S&P␣noise␣level␣=", "␣|␣".join([f"{value:{digits+2}}" for value in noise_levels]), "|")
print("|␣mean␣Dice␣score␣=", "␣|␣".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("␣-"*49,"\n")


"""i. Imprecise prompt"""

def shift_heatmap(heatmap, distance):
    """Shifts the heatmap by a given distance in a random direction."""
    h, w = heatmap.shape
    angle = np.random.uniform(0, 2 * np.pi)  # Random angle in radians
    dx = int(distance * np.cos(angle))  # X shift
    dy = int(distance * np.sin(angle))  # Y shift

    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    shifted_heatmap = cv2.warpAffine(heatmap, translation_matrix, (w, h), borderValue=0)
    return shifted_heatmap

def evaluate_heatmap_shift(model, images_dir, masks_dir, shift_distances, device, save_as="i_heatmap_shift"):
    """Evaluates the model with shifting heatmaps and plots Dice score vs. shift distance."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for distance in shift_distances:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            image_tensor = transforms.ToTensor()(Image.fromarray(image)).unsqueeze(0).to(device)

            heatmap = np.array(Image.open(heatmap_path).convert('L').resize((500, 500)))
            shifted_heatmap = shift_heatmap(heatmap, distance)
            heatmap_tensor = transforms.ToTensor()(Image.fromarray(shifted_heatmap)).unsqueeze(0).to(device)

            image_tensor = torch.cat([image_tensor, heatmap_tensor], dim=1)

            gt_mask = Image.open(mask_path).convert("L").resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
```

```
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(shift_distances, dice_scores, marker='o')
    plt.xlabel("Heatmap Shift (pixels)")
    plt.ylabel("Mean Dice Score")
    plt.title("Imprecise prompt")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\ni) Imprecise prompt...", end=' ')
shift_distances = [0, 25, 50, 75, 100, 125, 150, 200, 250, 300]
dice_scores = evaluate_heatmap_shift(model, images_dir, masks_dir, shift_distances, device)
print("Done!")
print(" -"*49)
print("| heatmap shift   =", " | ".join([f"{value:{digits+2}}" for value in shift_distances]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")
```

A.3.10 robustness_exploration_prompt_clip.py: Robustness-Exploration for Prompt Clip

```
# Authors: Sahil Mehul Bavishi (s2677266) and Matteo Spadaccia (s2748897)
# Subject: Computer Vision Coursework. Robustness exploration on the prompted CLIP model
# Date: 31.03.2025


"""0. Preliminary code"""

print("ROBUSTNESS EXPLORATION WITH DIFFERENT PERTURBATION TYPES (PROMPTED CLIP):")

# Setting code-behaviour varaibles
classNum = 4
pixel_to_class = {0: 0, 38: 1, 75: 2, 255: 3}
exludedClasses = [3]    # classes excludeed from metric calculation (3 not to consider the outline)
maxImages = -1          # number of input images to load from the defined set (0 to consider them all)
digits = 3              # digits to approximate decimal results in, when printed

# Defining paths
model_dir = f'Data/Output/Models2/PROMPTED_CLIP_DICE2_epoch_42.pth'
images_dir = 'Data/Input/Test/color/'
masks_dir = 'Data/Input/Test/label/'
heatmaps_dir = 'Data/Input/Test/heatmaps/'
output_plots_dir = 'Data/Output/Visuals/Prompted_CLIP_Robustness/'

# Importing useful libraries
import os
import numpy as np
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import open_clip
import matplotlib.pyplot as plt
import cv2
from skimage.util import random_noise

device = "cuda" if torch.cuda.is_available() else "cpu"

image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]

# Loading pre-trained prompted CLIP model
CLIP_model, preprocess_train, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
CLIP_model = CLIP_model.to(device)
tokenizer = open_clip.get_tokenizer("ViT-B-32")

class PromptedClipDecoder(nn.Module):
    def __init__(self, clip_feature_dim=512, num_classes=classNum):
        super(PromptedClipDecoder, self).__init__()

        # Project CLIP features
        self.clip_fc = nn.Linear(clip_feature_dim, 256)

        # CNN Feature Extractor (with 4 input channels: RGB + Heatmap)
        self.encoder = nn.Sequential(
            nn.Conv2d(4, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
        )

        # Decoder
```

```
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
            nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
        )

    def forward(self, image, clip_features):
        cnn_features = self.encoder(image)
        clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
        clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
        fusion = torch.cat([cnn_features, clip_features], dim=1)
        segmentation_output = self.decoder(fusion)
        return segmentation_output

# Loading segmentation model
model = PromptedClipDecoder().to(device)
model.load_state_dict(torch.load(model_dir, map_location=device))
model.eval()

def compute_dice_coefficient(pred, target, num_classes=classNum, exclude_classes=[]):
    """
    Compute Dice Coefficient for multi-class segmentation with the option to exclude specific classes.

    :param pred: Predicted segmentation map (H, W), with class indices.
    :param target: Ground truth segmentation map (H, W), with class indices.
    :param num_classes: Number of classes in segmentation.
    :param exclude_classes: List of class indices to exclude from Dice calculation.
    :return: Dice scores per class and mean Dice score.
    """

    dice_per_class = []
    for cls in range(num_classes):
        if cls in exclude_classes:
            continue

        pred_inds = pred == cls
        target_inds = target == cls
        intersection = 2.0 * torch.logical_and(pred_inds, target_inds).sum().item()
        union = pred_inds.sum().item() + target_inds.sum().item()

        if union == 0:
            dice_per_class.append(float('nan'))
        else:
            dice_per_class.append(intersection / union)

    mean_dice = np.nanmean(dice_per_class) if dice_per_class else float('nan')

    return dice_per_class, mean_dice


"""a. Gaussian noise"""

def apply_gaussian_noise(image, std_dev):
    """Applies Gaussian noise to an image with a given standard deviation."""
    noise = np.random.normal(0, std_dev, image.shape)
    noisy_image = np.clip(image + noise, 0, 255).astype(np.uint8)
    return noisy_image

def evaluate_gaussian_noise(model, images_dir, masks_dir, std_dev_levels, device, save_as:str="a_gaussian_noise"):
    """Evaluates the model with Gaussian noise perturbation and plots Dice score vs. noise level."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for std_dev in std_dev_levels:
        total_dice = []
        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            noisy_image = apply_gaussian_noise(image, std_dev)
            image_tensor = transforms.ToTensor()(Image.fromarray(noisy_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
```

```
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)

            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(std_dev_levels, dice_scores, marker='o')
    plt.xlabel("Gaussian Noise Standard Deviation")
    plt.ylabel("Mean Dice Score")
    plt.title("Gaussian Noise")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\na) Gaussian Noise...", end=' ')
std_dev_levels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
dice_scores = evaluate_gaussian_noise(model, images_dir, masks_dir, std_dev_levels, device)
print("Done!")
print(" "-"*49)
print("| st. deviations  =", " | ".join([f"{value:{digits+2}}" for value in std_dev_levels]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" "-"*49,"\n")


"""b. Gaussian blurring"""

def apply_gaussian_blur(image, ksize):
    """Applies Gaussian blur to an image with a given kernel size."""
    if ksize % 2 == 0:
        ksize += 1
    kernel = np.array([
        [1/16, 2/16, 1/16],
        [2/16, 4/16, 2/16],
        [1/16, 2/16, 1/16]
    ])
    blurred_image = cv2.filter2D(image, -1, kernel)
    return blurred_image

def evaluate_gaussian_blur(model, images_dir, masks_dir, max_blur_level, device, save_as:str="b_gaussian_blurring"):
    """Evaluates the model with Gaussian blurring perturbation and plots Dice score vs. blur level."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for blur_level in range(max_blur_level + 1):
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            blurred_image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            for i in range(blur_level):
                blurred_image = apply_gaussian_blur(blurred_image, 3)
            image_tensor = transforms.ToTensor()(Image.fromarray(blurred_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

```

```
    plt.figure()
    plt.plot(range(max_blur_level + 1), dice_scores, marker='o')
    plt.xlabel("Gaussian Blur Level (number of iterative 3x3 mask applications)")
    plt.ylabel("Mean Dice Score")
    plt.title("Gaussian Blurring")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\nb) Gaussian Blurring...", end=' ')
max_blur_level = 9
dice_scores = evaluate_gaussian_blur(model, images_dir, masks_dir, max_blur_level, device)
print("Done!")
print(" -"*49)
print("| blurring steps  =", " | ".join([f"{value:{digits+2}}" for value in list(range(max_blur_level+1))]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")


"""c. Contrast increase"""

def apply_contrast_change(image, contrast_factor):
    """Applies contrast increase to an image by multiplying pixel values by a contrast factor."""
    contrast_image = image * contrast_factor
    contrast_image = np.clip(contrast_image, 0, 255).astype(np.uint8)
    return contrast_image

def evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device, save_as:str="c_contrast_increase"
    , invert_axis:bool=False):
    """Evaluates the model with increased contrast perturbation and plots Dice score vs. contrast factor."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for contrast_factor in contrast_factors:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            contrast_image = apply_contrast_change(image, contrast_factor)
            image_tensor = transforms.ToTensor()(Image.fromarray(contrast_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes = exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(contrast_factors, dice_scores, marker='o')
    plt.xlabel("Contrast Factor")
    plt.ylabel("Mean Dice Score")
    plt.title("Contrast Decrease" if invert_axis else "Contrast Increase")
    plt.grid()
    if invert_axis: plt.gca().invert_xaxis()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\nc) Contrast Increase...", end=' ')
contrast_factors = [1.0, 1.01, 1.02, 1.03, 1.04, 1.05, 1.1, 1.15, 1.2, 1.25]
dice_scores = evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device)
print("Done!")
print(" -"*49)
print("| contrast factor =", " | ".join([f"{value:{digits+2}}" for value in contrast_factors]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
```

```
print("␣-"*49,"\n")


"""d. Contrast decrease"""
# (using contrast increase functions)

print("\nd)␣Contrast␣Decrease...", end='␣')
contrast_factors = [1.0, 0.95, 0.90, 0.85, 0.80, 0.60, 0.40, 0.30, 0.20, 0.10]
dice_scores = evaluate_contrast_change(model, images_dir, masks_dir, contrast_factors, device, save_as="
    d_contrast_decrease", invert_axis=True)
print("Done!")
print("␣-"*49)
print("|␣contrast␣factor␣=", "␣|␣".join([f"{value:{digits+2}}" for value in contrast_factors]), "|")
print("|␣mean␣Dice␣score␣=", "␣|␣".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("␣-"*49,"\n")


"""e. Brightness Increase"""

def apply_brightness_change(image, brightness_offset):
    """Applies brightness change by adding an offset to pixel values."""
    bright_image = image.astype(np.int16) + brightness_offset
    bright_image = np.clip(bright_image, 0, 255).astype(np.uint8)
    return bright_image

def evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device, save_as:str="
    e_brightness_increase", invert_axis:bool=False):
    """Evaluates the model with brightness perturbation and plots Dice score vs. brightness offset."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for brightness_offset in brightness_offsets:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            bright_image = apply_brightness_change(image, brightness_offset)
            image_tensor = transforms.ToTensor()(Image.fromarray(bright_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(brightness_offsets, dice_scores, marker='o')
    plt.xlabel("Brightness␣Offset")
    plt.ylabel("Mean␣Dice␣Score")
    plt.title("Brightness␣Decrease" if invert_axis else "Brightness␣Increase")
    plt.grid()
    if invert_axis: plt.gca().invert_xaxis()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\ne)␣Brightness␣Increase...", end='␣')
brightness_offsets = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
dice_scores = evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device)
print("Done!")
print("␣-"*49)
print("|␣bright.␣offset␣␣=", "␣|␣".join([f"{value:{digits+2}}" for value in brightness_offsets]), "|")
print("|␣mean␣Dice␣score␣=", "␣|␣".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print("␣-"*49,"\n")
```

```
"""f. Brightness Decrease"""
# (using brightness increase functions)

print("\nf) Brightness Decrease...", end=' ')
brightness_offsets = [0, -5, -10, -15, -20, -25, -30, -35, -40, -45]
dice_scores = evaluate_brightness_change(model, images_dir, masks_dir, brightness_offsets, device, save_as="
     f_brightness_decrease", invert_axis=True)
print("Done!")
print(" -"*49)
print("| bright. offset  =", " | ".join([f"{value:{digits+2}}" for value in brightness_offsets]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")


"""g. Square occlusion"""

def apply_occlusion(image, square_size):
    """Applies occlusion by replacing a random square region with black pixels."""
    h, w, _ = image.shape
    if square_size > 0:
        x = np.random.randint(0, w - square_size + 1)
        y = np.random.randint(0, h - square_size + 1)
        image[y:y+square_size, x:x+square_size] = 0
    return image

def evaluate_occlusion(model, images_dir, masks_dir, occlusion_sizes, device, save_as:str="g_square_occlusion"):
    """Evaluates the model under increasing occlusion perturbation."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for square_size in occlusion_sizes:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            occluded_image = apply_occlusion(image, square_size)
            image_tensor = transforms.ToTensor()(Image.fromarray(occluded_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(occlusion_sizes, dice_scores, marker='o')
    plt.xlabel("Occlusion Size (edge length in pixels)")
    plt.ylabel("Mean Dice Score")
    plt.title("Square Occlusion")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")
    return dice_scores

print("\ng) Square Occlusion...", end=' ')
occlusion_sizes = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
dice_scores = evaluate_occlusion(model, images_dir, masks_dir, occlusion_sizes, device)
print("Done!")
print(" -" * 49)
print("| oc. square size =", " | ".join([f"{value:{digits+2}}" for value in occlusion_sizes]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -" * 49, "\n")


"""h. Salt&Pepper Noise"""

def apply_salt_and_pepper_noise(image, noise_amount):
```

```
    """Applies salt and pepper noise to an image."""
    noisy_image = random_noise(image, mode='s&p', amount=noise_amount)
    noisy_image = (noisy_image * 255).astype(np.uint8)
    return noisy_image

def evaluate_salt_and_pepper_noise(model, images_dir, masks_dir, noise_levels, device, save_as:str="h_SaP_noise"):
    """Evaluates the model with salt-and-pepper noise perturbation and plots Dice score vs. noise level."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for noise_amount in noise_levels:
        total_dice = []

        counter = 0
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            noisy_image = apply_salt_and_pepper_noise(image, noise_amount)
            image_tensor = transforms.ToTensor()(Image.fromarray(noisy_image)).unsqueeze(0).to(device)
            heatmap = Image.open(heatmap_path).convert('L').resize((500, 500))
            heatmap = transforms.ToTensor()(heatmap).unsqueeze(0).to(device)
            image_tensor = torch.cat([image_tensor, heatmap], dim=1)

            gt_mask = Image.open(mask_path).convert("L")
            gt_mask = gt_mask.resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(noise_levels, dice_scores, marker='o')
    plt.xlabel("Salt & Pepper Noise Level")
    plt.ylabel("Mean Dice Score")
    plt.title("Salt & Pepper Noise")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\nh) Salt & Pepper Noise...", end=' ')
noise_levels = [0.00, 0.02, 0.04, 0.06, 0.08, 0.10, 0.12, 0.14, 0.16, 0.18]
dice_scores = evaluate_salt_and_pepper_noise(model, images_dir, masks_dir, noise_levels, device)
print("Done!")
print(" -"*49)
print("| S&P noise level =", " | ".join([f"{value:{digits+2}}" for value in noise_levels]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")


"""i. Imprecise prompt"""

def shift_heatmap(heatmap, distance):
    """Shifts the heatmap by a given distance in a random direction."""
    h, w = heatmap.shape
    angle = np.random.uniform(0, 2 * np.pi)  # Random angle in radians
    dx = int(distance * np.cos(angle))  # X shift
    dy = int(distance * np.sin(angle))  # Y shift

    translation_matrix = np.float32([[1, 0, dx], [0, 1, dy]])
    shifted_heatmap = cv2.warpAffine(heatmap, translation_matrix, (w, h), borderValue=0)
    return shifted_heatmap

def evaluate_heatmap_shift(model, images_dir, masks_dir, shift_distances, device, save_as="i_heatmap_shift"):
    """Evaluates the model with shifting heatmaps and plots Dice score vs. shift distance."""
    image_files = [f for f in os.listdir(images_dir) if f.endswith('.jpg')]
    dice_scores = []

    for distance in shift_distances:
        total_dice = []

        counter = 0
```

```
        for img_file in image_files:
            counter += 1
            if counter == maxImages:
                break
            img_path = os.path.join(images_dir, img_file)
            mask_path = os.path.join(masks_dir, img_file.replace('.jpg', '.png'))
            heatmap_path = os.path.join(heatmaps_dir, img_file.split('.')[0]+'_heatmap.png')

            image = np.array(Image.open(img_path).convert("RGB").resize((500, 500)))
            image_tensor = transforms.ToTensor()(Image.fromarray(image)).unsqueeze(0).to(device)

            heatmap = np.array(Image.open(heatmap_path).convert('L').resize((500, 500)))
            shifted_heatmap = shift_heatmap(heatmap, distance)
            heatmap_tensor = transforms.ToTensor()(Image.fromarray(shifted_heatmap)).unsqueeze(0).to(device)

            image_tensor = torch.cat([image_tensor, heatmap_tensor], dim=1)

            gt_mask = Image.open(mask_path).convert("L").resize((256, 256), Image.NEAREST)
            label_array = np.array(gt_mask)

            label_tensor = torch.from_numpy(label_array).to(dtype=torch.uint8)
            gt_mask_class = torch.zeros_like(label_tensor, dtype=torch.uint8)

            for pixel_value, class_idx in pixel_to_class.items():
                gt_mask_class[label_tensor == pixel_value] = class_idx

            image_clip_tensor = preprocess_val(Image.open(img_path).convert("RGB")).unsqueeze(0).to(device)

            with torch.no_grad():
                clip_features = CLIP_model.encode_image(image_clip_tensor)
                clip_features = clip_features / clip_features.norm(dim=-1, keepdim=True)
                model_output = model(image_tensor, clip_features)
                pred_mask = model_output.squeeze(0)
                pred_mask = torch.argmax(pred_mask, dim=0).cpu()

            _, mean_dice = compute_dice_coefficient(pred_mask, gt_mask_class, exclude_classes=exludedClasses)
            total_dice.append(mean_dice)

        dice_scores.append(np.mean(total_dice))

    plt.figure()
    plt.plot(shift_distances, dice_scores, marker='o')
    plt.xlabel("Heatmap Shift (pixels)")
    plt.ylabel("Mean Dice Score")
    plt.title("Imprecise prompt")
    plt.grid()
    plt.savefig(output_plots_dir + save_as + ".pdf", format="pdf")

    return dice_scores

print("\ni) Imprecise prompt...", end=' ')
shift_distances = [0, 25, 50, 75, 100, 125, 150, 200, 250, 300]
dice_scores = evaluate_heatmap_shift(model, images_dir, masks_dir, shift_distances, device)
print("Done!")
print(" -"*49)
print("| heatmap shift   =", " | ".join([f"{value:{digits+2}}" for value in shift_distances]), "|")
print("| mean Dice score =", " | ".join([f"{d:{digits+2}.{digits}f}" for d in dice_scores]), "|")
print(" -"*49,"\n")
```

A.3.11 flaskApp.py: Backend for UI

```
from flask import Flask, render_template, request, send_from_directory, jsonify
import os
from PIL import Image
import torch
import numpy as np
import torchvision.transforms as transforms
import streamlit as st
import torch
import torch.nn as nn
from torchvision import transforms
from PIL import Image
import open_clip
import numpy as np
import matplotlib.pyplot as plt



app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = 'static'
classesNum = 4
def load_models():
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    clip_model, _, preprocess_val = open_clip.create_model_and_transforms("ViT-B-32", pretrained="openai")
    clip_model = clip_model.to(device)

    class ClipDecoder(nn.Module):
        def __init__(self, clip_feature_dim=512, num_classes=classesNum):
            super(ClipDecoder, self).__init__()

            # Project CLIP features
```

```
            self.clip_fc = nn.Linear(clip_feature_dim, 256)

            # CNN Feature Extractor (Now 4 input channels: RGB + Heatmap)
            self.encoder = nn.Sequential(
                nn.Conv2d(4, 64, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.MaxPool2d(2),
                nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
            )

            # Decoder
            self.decoder = nn.Sequential(
                nn.ConvTranspose2d(384, 64, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                nn.Upsample(size=(256, 256), mode='bilinear', align_corners=True),
                nn.ConvTranspose2d(64, num_classes, kernel_size=3, stride=1, padding=1),
            )

        def forward(self, image, clip_features):
            cnn_features = self.encoder(image)
            clip_features = self.clip_fc(clip_features).view(clip_features.shape[0], 256, 1, 1)
            clip_features = clip_features.expand(-1, -1, cnn_features.shape[2], cnn_features.shape[3])
            fusion = torch.cat([cnn_features, clip_features], dim=1)
            segmentation_output = self.decoder(fusion)

            return segmentation_output

    model_path = 'Data/Output/Models/PROMPTED_CLIP_CROSS_epoch_44.pth'
    segmentation_model = ClipDecoder().to(device)
    segmentation_model.load_state_dict(torch.load(model_path, map_location=device))
    segmentation_model.eval()
    return clip_model, preprocess_val, segmentation_model, device

clip_model, preprocess_val, model, device = load_models()

# Extract CLIP features
def extract_clip_features(image):
    image = preprocess_val(image).unsqueeze(0).to(device)
    with torch.no_grad():
        features = clip_model.encode_image(image)
    features = features / features.norm(p=2, dim=-1, keepdim=True)
    return features.to(device)

def generate_gaussian_heatmap(size, center, intensity=1.0, sigma=50):
    x0, y0 = center
    x = np.arange(size[0])
    y = np.arange(size[1])
    X, Y = np.meshgrid(x, y)
    heatmap = intensity * np.exp(-((X - x0)**2 + (Y - y0)**2) / (2 * sigma**2))
    return heatmap

def segment_image(image, heatmap):
    clip_features = extract_clip_features(image)
    heatmap_resized = Image.fromarray((heatmap * 255).astype(np.uint8)).resize((256, 256))
    heatmap = np.array(heatmap_resized) / 255.0  # Convert back to float array

    # Resize and normalize input image
    transform = transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
    ])

    input_tensor = transform(image).to(device).to(torch.float32)
    heatmap = torch.tensor(heatmap, dtype=torch.float32).unsqueeze(0).to(device)
    input_tensor = torch.cat([input_tensor, heatmap], dim=0).unsqueeze(0)

    with torch.no_grad():
        output = model(input_tensor, clip_features)
    output = torch.argmax(output.squeeze(), dim=0).cpu().numpy()
    return output

def visualize_segmentation(segmented_output, original_size):
    # Define a color palette for 4 classes (0, 1, 2, 3)
    color_map = {
        0: [0, 0, 0],
        1: [51, 102, 153],
        2: [173, 216, 230],
        3: [255, 255, 255]
    }

    # Create a color image based on the segmented output
    colored_output = np.zeros((segmented_output.shape[0], segmented_output.shape[1], 3), dtype=np.uint8)
    for label, color in color_map.items():
        colored_output[segmented_output == label] = color

    # Convert to a PIL image
    segmented_image = Image.fromarray(colored_output)
    segmented_image = segmented_image.resize(original_size, resample=Image.NEAREST)

    # Save the image
    segmented_image_path = os.path.join(app.config['UPLOAD_FOLDER'], 'segmented.png')
    segmented_image.save(segmented_image_path)

    return segmented_image_path
```

```
126
127 @app.route('/')
128 def home():
129     return render_template('index.html')
130
131 @app.route('/upload', methods=['POST'])
132 def upload():
133     try:
134         image = request.files['image']
135         if image:
136             image_path = os.path.join(app.config['UPLOAD_FOLDER'], image.filename)
137             image.save(image_path)
138             return render_template('image_display.html', image_path=image_path)
139         else:
140             return "No image received"
141     except Exception as e:
142         return f"Error: {str(e)}"
143
144 @app.route('/process_click', methods=['POST'])
145 def process_click():
146     try:
147         data = request.json
148         x, y = float(data['x']), float(data['y'])
149         image_path = data['image_path']
150         print(f"Clicked here {x}, {y}")
151         image = Image.open(image_path).convert("RGB")
152         image_size = image.size
153         print(image_size)
154         heatmap = generate_gaussian_heatmap(image_size, (x, y), intensity=1.0, sigma=50)
155         heatmap_path = os.path.join(app.config['UPLOAD_FOLDER'], 'heatmap.png')
156
157         plt.imsave(heatmap_path, heatmap, cmap='gray')
158
159         # print("Came here")
160         segmented_output = segment_image(image, heatmap)
161         segmented_image_path = visualize_segmentation(segmented_output, image_size)
162         print('Reached here')
163         return jsonify({'heatmap': heatmap_path, 'segmented_image': segmented_image_path})
164     except Exception as e:
165         return jsonify({'error': str(e)})
166
167 @app.route('/process_scribble', methods=['POST'])
168 def process_scribble():
169     try:
170         data = request.json
171         points = [(float(p['x']), float(p['y'])) for p in data['points']]
172         image_path = data['image_path']
173
174         print(f"Scribble points: {points}")
175
176         image = Image.open(image_path).convert("RGB")
177         image_size = image.size
178
179         # Generate heatmap from multiple points
180         heatmap = np.zeros(image_size[::-1])  # Create empty heatmap
181         for x, y in points:
182             heatmap += generate_gaussian_heatmap(image_size, (x, y), intensity=1.0, sigma=1)
183
184         heatmap = np.clip(heatmap, 0, 1)  # Normalize
185
186         heatmap_path = os.path.join(app.config['UPLOAD_FOLDER'], 'heatmap.png')
187         plt.imsave(heatmap_path, heatmap, cmap='gray')
188
189         segmented_output = segment_image(image, heatmap)
190         segmented_image_path = visualize_segmentation(segmented_output, image_size)
191
192         return jsonify({'heatmap': heatmap_path, 'segmented_image': segmented_image_path})
193
194     except Exception as e:
195         return jsonify({'error': str(e)})
196
197 @app.route('/process_bounding_box', methods=['POST'])
198 def process_bounding_box():
199     try:
200         data = request.json
201         start_x, start_y = float(data['startX']), float(data['startY'])
202         end_x, end_y = float(data['endX']), float(data['endY'])
203         image_path = data['image_path']
204
205         print(f"Bounding Box: ({start_x}, {start_y}) to ({end_x}, {end_y})")
206
207         image = Image.open(image_path).convert("RGB")
208         image_size = image.size
209
210         # Ensure proper ordering of coordinates
211         x_min, x_max = min(start_x, end_x), max(start_x, end_x)
212         y_min, y_max = min(start_y, end_y), max(start_y, end_y)
213
214         # Generate all points inside the bounding box
215         points = [(x, y) for x in range(int(x_min), int(x_max)) for y in range(int(y_min), int(y_max))]
216
217         heatmap = np.zeros(image_size[::-1])  # Empty heatmap
218         for x, y in points:
219             heatmap += generate_gaussian_heatmap(image_size, (x, y), intensity=1.0, sigma=1)
220
```

```
        heatmap = np.clip(heatmap, 0, 1)  # Normalize

        heatmap_path = os.path.join(app.config['UPLOAD_FOLDER'], 'heatmap.png')
        plt.imsave(heatmap_path, heatmap, cmap='gray')

        segmented_output = segment_image(image, heatmap)
        segmented_image_path = visualize_segmentation(segmented_output, image_size)

        return jsonify({'heatmap': heatmap_path, 'segmented_image': segmented_image_path})

    except Exception as e:
        return jsonify({'error': str(e)}), 500



@app.route('/static/<path:filename>')
def static_files(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'], filename)

if __name__ == '__main__':
    app.run(debug=True)
```

A.3.12 index.html: Frontend Landing Page

```
 <!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Image Segmentation</title>
    <link href="https://fonts.googleapis.com/css2?family=Merriweather:wght@300;400;700&display=swap" rel="stylesheet">
    <style>
        body {
            font-family: 'Merriweather', serif;
            background-color: #eef2f7;
            margin: 0;
            padding: 20px;
            display: flex;
            justify-content: center;
            align-items: center;
            flex-direction: column;
            min-height: 100vh;
        }
        h1, h2 {
            color: #2c3e50;
            margin-bottom: 15px;
            text-align: center;
        }
        form {
            background-color: #ffffff;
            padding: 25px;
            border-radius: 12px;
            box-shadow: 0 6px 12px rgba(0, 0, 0, 0.1);
            margin-bottom: 25px;
            width: 100%;
            max-width: 400px;
            text-align: center;
        }
        label {
            font-weight: bold;
            color: #34495e;
        }
        input[type="file"] {
            margin: 10px 0;
            padding: 8px;
            border: 1px solid #ccc;
            border-radius: 6px;
            width: 100%;
        }
        button {
            background-color: #3498db;
            color: white;
            padding: 12px 18px;
            border: none;
            border-radius: 8px;
            cursor: pointer;
            font-size: 16px;
            transition: background-color 0.3s ease-in-out;
        }
        button:hover {
            background-color: #2980b9;
        }
        .image-container {
            text-align: center;
            margin: 20px 0;
        }
        img {
            max-width: 100%;
            border-radius: 12px;
            box-shadow: 0 6px 12px rgba(0, 0, 0, 0.1);
            cursor: crosshair;
```

```
        }
        .segmented-title {
            margin-top: 30px;
            font-size: 20px;
            font-weight: bold;
        }
    </style>
    <script>
        function sendClick(event) {
            let img = document.getElementById("uploadedImage");
            let rect = img.getBoundingClientRect();
            let x = event.clientX - rect.left;
            let y = event.clientY - rect.top;
            let imagePath = img.getAttribute("src");

            fetch('/process_click', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ x: x, y: y, image_path: imagePath })
            })
            .then(response => response.json())
            .then(data => {
                if (data.segmented_image) {
                    document.getElementById("segmentedResult").src = data.segmented_image;
                } else {
                    alert("Error: " + data.error);
                }
            });
        }
    </script>
</head>
<body>
    <h1>Image Segmentation Tool</h1>
    <form action="/upload" method="POST" enctype="multipart/form-data">
        <label for="image">Select an image:</label>
        <input type="file" name="image" required>
        <button type="submit">Upload</button>
    </form>

    <h2>Click on the image to select coordinates</h2>
    <div class="image-container">
        <img id="uploadedImage" src="{{ image_path }}" onclick="sendClick(event)">
    </div>

    <h2 class="segmented-title">Segmented Image</h2>
    <div class="image-container">
        <img id="segmentedResult" src="">
    </div>
</body>
</html>
```

A.3.13 image_display.html: Frontend Interaction and Result Page

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Image Interaction</title>
    <style>
        canvas {
            position: absolute;
            top: 0;
            left: 0;
            pointer-events: none;
        }
    </style>
    <script>
        let mode = "point";
        let drawing = false;
        let strokePoints = [];
        let startX, startY, endX, endY;
        let canvas, ctx;

        function changeMode() {
            mode = document.getElementById("modeSelect").value;
        }

        function sendClick(event) {
            if (mode !== "point") return;
            let img = document.getElementById("uploadedImage");
            let rect = img.getBoundingClientRect();
            let x = (event.clientX - rect.left).toFixed(2);
            let y = (event.clientY - rect.top).toFixed(2);
            let imagePath = img.getAttribute("src");

            fetch('/process_click', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ x: x, y: y, image_path: imagePath })
            })
            .then(response => response.json())
```

```
            .then(updateResults);
        }

        window.onload = function() {
            let img = document.getElementById("uploadedImage");
            canvas = document.createElement("canvas");
            canvas.width = img.width;
            canvas.height = img.height;
            canvas.style.position = "absolute";
            canvas.style.top = img.offsetTop + "px";
            canvas.style.left = img.offsetLeft + "px";
            document.body.appendChild(canvas);
            ctx = canvas.getContext("2d");
        };

        function startDrawing(event) {
            event.preventDefault(); // Prevent image movement
            let img = document.getElementById("uploadedImage");
            let rect = img.getBoundingClientRect();
            startX = event.clientX - rect.left;
            startY = event.clientY - rect.top;
            drawing = true;
            strokePoints = [];
            ctx.clearRect(0, 0, canvas.width, canvas.height);
        }

        function draw(event) {
            event.preventDefault(); // Prevent dragging behavior
            if (!drawing) return;
            let img = document.getElementById("uploadedImage");
            let rect = img.getBoundingClientRect();
            let x = event.clientX - rect.left;
            let y = event.clientY - rect.top;

            if (mode === "scribble") {
                addPoint(event, true);
            } else if (mode === "bounding_box") {
                ctx.clearRect(0, 0, canvas.width, canvas.height);
                ctx.strokeStyle = "blue";
                ctx.lineWidth = 2;
                ctx.strokeRect(startX, startY, x - startX, y - startY);
                endX = x;
                endY = y;
            }
        }


        function stopDrawing(event) {
            if (!drawing) return;
            drawing = false;
            if (mode === "scribble" && strokePoints.length > 0) {
                sendStroke();
            } else if (mode === "bounding_box") {
                sendBoundingBox();
            }
        }

        function addPoint(event, drawLine) {
            let img = document.getElementById("uploadedImage");
            let rect = img.getBoundingClientRect();
            let x = event.clientX - rect.left;
            let y = event.clientY - rect.top;
            strokePoints.push({ x: x.toFixed(2), y: y.toFixed(2) });

            if (drawLine && strokePoints.length > 1) {
                let lastPoint = strokePoints[strokePoints.length - 2];
                ctx.beginPath();
                ctx.moveTo(lastPoint.x, lastPoint.y);
                ctx.lineTo(x, y);
                ctx.strokeStyle = "red";
                ctx.lineWidth = 2;
                ctx.stroke();
            }
        }

        function sendStroke() {
            let img = document.getElementById("uploadedImage");
            let imagePath = img.getAttribute("src");

            fetch('/process_scribble', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ points: strokePoints, image_path: imagePath })
            })
            .then(response => response.json())
            .then(updateResults);
        }

        function sendBoundingBox() {
            let img = document.getElementById("uploadedImage");
            let imagePath = img.getAttribute("src");

            fetch('/process_bounding_box', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
```

```
                body: JSON.stringify({ startX: startX.toFixed(2), startY: startY.toFixed(2), endX: endX.toFixed(2),
                    endY: endY.toFixed(2), image_path: imagePath })
            })
            .then(response => response.json())
            .then(updateResults);
        }

        function updateResults(data) {
            if (data.segmented_image) {
                document.getElementById("segmentedResult").src = data.segmented_image;
            }
            if (data.heatmap) {
                document.getElementById("heatmapResult").src = data.heatmap;
            }
        }
    </script>
</head>
<body>
    <h1>Image Interaction</h1>
    <label for="modeSelect">Select Mode:</label>
    <select id="modeSelect" onchange="changeMode()">
        <option value="point">Point</option>
        <option value="scribble">Scribble</option>
        <option value="bounding_box">Bounding Box</option>
    </select>
    <br><br>
    <img id="uploadedImage" src="{{ image_path }}"
         onclick="sendClick(event)"
         onmousedown="startDrawing(event)"
         onmousemove="draw(event)"
         onmouseup="stopDrawing(event)"
         onmouseleave="stopDrawing(event)"
         style="max-width: 100%; cursor: crosshair; user-select: none; position: relative;">

    <h2>Heatmap</h2>
    <img id="heatmapResult" src="" style="max-width: 100%;">

    <h2>Segmented Image</h2>
    <img id="segmentedResult" src="" style="max-width: 100%;">
</body>
</html>
```

A.4 Contribution statement

For the production of the present study, the workload was equally distributed between two students, with clearly devided responsibilities and continuous development checks in the form of weekly meetings. Hereafter, the specific distribution of tasks.
S2677266: dataset loading, augmentation, AutoEncoder, CLIP, data annotation, performance evaluation, UI backend, report drafting
S2748897: dataset loading, resizing, UNet, data annotation, prompt-enriched CLIP, robustness exploration, UI frontend, report drafting

References

[1] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*, 2017.

[2] Akanksha Bali and Shailendra Narayan Singh. A review on the strategies and techniques of image segmentation. In *2015 Fifth international conference on advanced computing & communication technologies*, pages 113–120. IEEE, 2015.

[3] Hui Zhang, Jason E Fritts, and Sally A Goldman. Image segmentation evaluation: A survey of unsupervised methods. *computer vision and image understanding*, 110(2):260–280, 2008.

[4] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3523–3542, 2022. Available at: DOI: 10.1109/TPAMI.2021.3059968.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2018.

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

[8] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[10] Diederik P Kingma, Max Welling, et al. Auto-encoding variational bayes, 2013.

[11] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[12] 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. Clip: Contrastive language-image pretraining. Website, 2021. `https://openai.com/index/clip/`.

[13] Ziqin Zhou, Yinjie Lei, Bowen Zhang, Lingqiao Liu, and Yifan Liu. Zegclip: Towards adapting clip for zero-shot semantic segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11175–11185, 2023.

[14] Zdravko Marinov, Paul F Jäger, Jan Egger, Jens Kleesiek, and Rainer Stiefelhagen. Deep interactive segmentation of medical images: A systematic review and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 2024.

[15] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Sam: Segment anything model. Website, 2023. `https://segment-anything.com/`.

[16] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar. Oxford-iiit pet dataset. Website, 2012. `https://www.robots.ox.ac.uk/~vgg/data/pets/`.

[17] Roboflow. You might be resizing your images incorrectly. *Roboflow Blog*, 2023. Accessed: 2025-04-04.

[18] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[19] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, and C. V. Jawahar. Cats and dogs. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3498–3505, 2012.

[20] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 fourth international conference on 3D vision (3DV)*, pages 565–571. Ieee, 2016.

[21] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, et al. Attention u-net: Learning where to look for the pancreas. *arXiv preprint arXiv:1804.03999*, 2018.

[22] Mark Everingham, SM Ali Eslami, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International journal of computer vision*, 111:98–136, 2015.

[23] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.