Analysis of Object Segmentation Architectures: UNet, Autoencoder, CLIP and Point-Based Segmentation

Julia López Gómez Benjamin Henríquez Soto

Abstract

This report evaluates the performance and robustness of four different semantic segmentation models—U-Net, Promt-Based U-Net, a two-phase Autoencoder, and a CLIP-based model—on a processed version of the Oxford-IIIT Pet Dataset. We investigate architectural differences, prompt conditioning, loss functions, and evaluate robustness to common perturbations. While U-Net provided a strong baseline, leveraging pre-trained CLIP features (CLIP+U-Net) significantly improved accuracy, and our prompt-guided model (PointCLIPUNet) achieved strong performance and supported interactive segmentation through a UI. Overall, results demonstrate that the tested architectures can perform well better but require regularisation to maintain robustness.

1. Introduction

Semantic segmentation consists of assigning a class label to each pixel in an image and is essential in applications like autonomous driving, medical imaging, and scene understanding. In this project, we segment pet images into three classes: background, cat, and dog, using a processed version of the Oxford-IIIT Pet Dataset.

Pixel-wise classification presents several challenges: strong intra-class variation (e.g., different dog and cat breeds), fine-grained details (e.g., fur, whiskers), and class imbalance (e.g., more dog than cat images). Overfitting is a risk, given the limited dataset size.

We implement and evaluate four segmentation approaches: a baseline U-Net, a U-Net with autoencoder pre-training, a CLIP-based segmentation model, and a prompt-guided variant that accepts point heatmaps. Each model is assessed using Dice score, Intersection over Union (IoU), and pixel accuracy on a fixed test set. We also compare their robustness under image perturbations.

Section 2 describes the dataset and preprocessing steps, including data augmentation techniques. Section 3 details the architecture and training pipeline for each model. Section 4.3 presents the user interface for the prompt-based model, allowing users to segment images interactively and also visualise the results of the rest of our models. Section 4 presents the evaluation metrics and results, followed by a discussion of the findings and a summary of our work in Section 6.

2. Data Augmentation and Preprocessing

All experiments were conducted using a processed version of the Oxford-IIIT Pet Dataset, pre-divided into a combined `TrainVal` split and a separate `Test` set. Each sample consists of an RGB(A) image and a corresponding semantic segmentation mask labelling each pixel as either background, cat, or dog.

2.1. Data Preprocessing

Prior to training, we conducted a thorough analysis of the dataset. A total of 7 training and 16 test samples were removed due to corrupted or invalid masks (typically masks containing only background pixels). After this cleanup, we retained 3,673 training images and 3,694 test images. Breed representation was found to be uniformly distributed across both splits. However, we assessed that the number of dog-labeled images doubled that of cat-labeled ones, with similar proportions on both the train and test sets, indicating a potential class imbalance problem. Additionally, we converted RGBA images to standard RGB by discarding the alpha channel, ensuring consistency across the dataset.

Masks included void pixels labelled with the class index 255. Rather than ignoring these pixels, which could complicate training and evaluation, we decided to implement a nearest-neighbour fill strategy to assign each pixel the class of the closest valid pixel.

To ensure consistency in input dimensions, all images and masks were standardised to a fixed resolution of 256×256 prior to training. Instead of direct resizing, we adopted a two-step process where each image was first scaled such that its shortest side matched the target dimension, preserving the aspect ratio, to then apply zero-padding to create a square input (Figure 1). This method was selected to avoid distortions and preserve relative object shapes without losing information. For validation and testing, the input images were processed similarly (resised to 256×256 using scaling and padding), but we preserved ground-truth masks at their

Figure 1. Preprocessing pipeline: To the left, the original image and mask of a Bombay cat. To the right, the processed image and mask after scaling and padding. The resized mask is filled using the nearest- neighbour interpolation to fill void pixels (255) with the class of the closest valid pixel.

original resolution to ensure a fair comparison with the provided baseline. Finally, all input images were normalised using ImageNet statistics to ensure stability during training.

This preprocessing pipeline was applied consistently across training, validation, and test sets.

2.2. Data Augmentation

To mitigate overfitting and improve generalisation, we applied several data augmentation techniques, which have shown benefits when working with limited datasets. They expand the training distribution and encourage the model to learn from new representations.

The data augmentation pipeline used during the training included horizontal flipping with a probability of 0.5 to help the model generalise across left-right symmetries. We also included random rotations limited to ±20° (also with p=0.5) to simulate pose variability. All perturbations were applied using default parameters unless otherwise specified.

To introduce spatial deformation, we used elastic transformations and grid distortion, both applied with a probability of 0.2. These help the model become robust to local warping and shape variation. To address appearance variation, we included colour jitter (p = 0.3) and Gaussian noise (p = 0.3), enhancing robustness to lighting and sensor noise.

All augmentations were implemented using the Albumentations library, ensuring that transformations to the input image were consistently mirrored in the corresponding segmentation mask. The augmentations described can be visualised in Figure 2.

3. Network Design and Implementation

This section outlines the modular training and evaluation pipeline common to all models. Although the architectures differ (U-Net, Autoencoder, CLIP-based, and Point-Based) the infrastructure supporting them is unified. The design emphasises flexibility, transparency, and reusability across different model platforms.

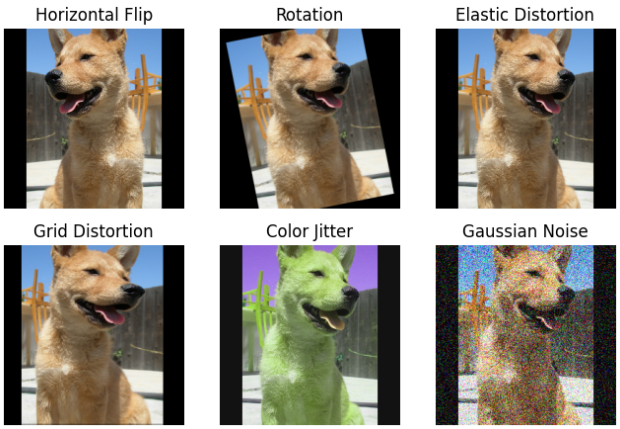


Figure 2. Examples of data augmentation applied to training data.

Configuration and Modularity. All experiments are executed via a command-line interface. The main training script (`train.py`) accepts arguments for core hyperparameters such as learning rate, batch size, number of epochs, dropout rate, optimiser type, scheduler, class weights, validation split ratio, and model type. This allows users to prototype different architectures and configurations with minimal code change.

Model instantiation is handled dynamically based on arguments. The segmentation heads are standardised to predict three classes: background, cat, and dog.

Data Handling. The process of dataset loading is abstracted into a custom PyTorch `Dataset` class (`SegmentationDataset`), which manages image-mask pairs and applies model-appropriate preprocessing. The dataset is split into training, validation, and test sets. Training data undergoes augmentations (Section 2.1), while validation and test data are only resized and normalised.

The dataloaders use multiprocessing workers to accelerate loading and apply collation strategies that ensure all batches are of consistent shape.

Training Loop and Features. The training pipeline is designed to run on both CPUs and CUDA-compatible GPUs, with automatic detection. All models are trained using the AdamW optimiser by default, with the option to select Adam, SGD, or RMSProp. A Cosine Annealing with Warm Restarts scheduler modulates the learning rate to encourage convergence and escape local minima.

Class imbalance is handled across models using weighted Cross-Entropy, penalising the background class (as it predominates in the images), and assigning higher weights to cats because of their lower representation in the dataset. The weights are set to 0.3 for background, 2.2 for cats, and 0.8 for dogs.

To improve training stability, especially in early epochs, gradient clipping is applied. Mixed precision training via `torch.cuda.amp` is supported and reduces GPU memory footprint without sacrificing numerical stability. Dropout layers are selectively injected into the U-Net architecture (and inherited by CLIP and Point-based models) via a tunable dropout rate parameter, allowing regularisation during training.

Generally, all models are trained for 100 epochs, with early stopping based on validation IoU and batch size of 16, although higher values were tried for specific instances.

Loss Functions and Metrics Loss functions are selected dynamically based on the model type and training phase. U-Net, Point-based, and CLIP-based models use a combination of Cross-Entropy Loss and Dice Loss with optional class weighting. The Autoencoder uses reconstruction loss (MSE) in its pretraining phase and switches to segmentation loss in its second phase.

Evaluation metrics include mean Dice score, per-class Dice, Intersection-over-Union (IoU), per-class IoU and pixel accuracy. These are computed using utility functions in `metrics.py`, and results are aggregated at both batch and epoch levels.

Logging and Checkpointing Training progress is logged through both the terminal and Weights & Biases (W& B) (1), where loss and metrics are visualised per epoch. Checkpointing is optionally enabled; model weights are defaulty saved at the end of each epoch if the current validation IoU exceeds the best seen so far, serving as a backup and allowing for early stopping.

Code Reusability and Design The pipeline is designed to be modular and extensible. All models implement a consistent interface, and training logic delegates device transfer, loss computation, and metric logging to shared functions. This makes it easy to plug in new models or evaluation strategies without modifying the training loop.

The implementation is written in Pytorch, and we used Poetry, a dependency management tool, to manage the environment and package dependencies. This ensures that all required libraries are installed in a consistent manner, and it allows for easy reproduction of the environment across different machines. The code is structured in a modular way, with separate files for data loading, model definitions, training loops, and evaluation metrics. This allows for easy extension and modification of individual components without affecting the overall pipeline.

a) U-Net for Semantic Segmentation

U-Net is a convolutional neural network architecture originally designed for biomedical image segmentation. Its defining feature is the encoder-decoder structure with symmetric skip connections that enable precise localisation. The encoder progressively reduces spatial resolution (image size) while increasing feature depth (the number of channels) to capture high-level semantics. The decoder gradually upsamples these features to reconstruct pixel-level predictions. At each step, skip connections from the encoder layers are concatenated with corresponding decoder layers, recovering spatial detail lost during downsampling.

Formally, U-Net maps an input image $X \in \mathbb{R}^{H\times W\times 3}$ to a segmentation mask $Y \in \{0,1,2\}^{H\times W}$, converting RGB inputs to per-pixel class labels. The output of the network is a tensor of shape $H \times W \times 3$, containing raw class logits for each pixel. These are converted into class probabilities via a softmax layer.

The loss function used is a combination of pixel-wise Cross-Entropy (CE) loss and Dice loss, aiming to balance precision and recall, especially under class imbalance.

The CE loss is defined as:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c}) \tag{1}$$

where $p_{i,c}$ is the predicted softmax probability for class c at pixel i, and $y_{i,c}$ is the one-hot encoded ground truth label. N is the total number of pixels in the image, and C the number of classes (background, cat, dog). The CE loss penalises incorrect class predictions, encouraging the model to assign high probabilities to the correct class.

The Dice loss for class c is defined as:

$$\mathcal{L}_{\text{Dice}} = 1 - \frac{2\sum_i p_{i,c}y_{i,c} + \epsilon}{\sum_i p_{i,c} + \sum_i y_{i,c} + \epsilon} \tag{2}$$

with ϵ as a small constant to ensure numerical stability. The final loss is a weighted sum:

$$\mathcal{L}_{\text{total}} = \alpha\cdot\mathcal{L}_{\text{CE}} + \beta\cdot\mathcal{L}_{\text{Dice}} \tag{3}$$

where we set $\alpha = \beta = 1$. This hybrid loss encourages both accurate classification and overlap, helping mitigate class imbalance.

Implementation and Design Choices. Our U-Net implementation follows a symmetric structure with five downsampling and five upsampling stages, corresponding to the encoder and decoder, respectively (Figure 3). Each encoder stage reduces spatial dimensions by a factor of 2 and doubles the number of feature channels while the decoder reverses this pattern. The U-Net architecture consists of:

- `DoubleConv` (blue/orange): two sequential 3×3 convolutions (with 1-padding to preserve spatial dimensions), each followed by Batch Normalisation and ReLU activation. Dropout is optionally added after

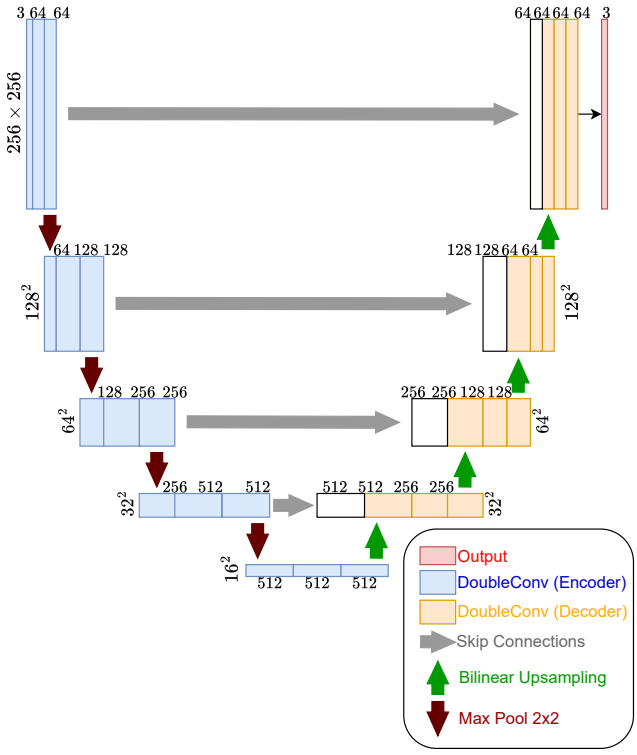


Figure 3. U-Net architecture used for segmentation. The encoder downsamples to a bottleneck of size 16×16 (for input size 256×256), and the decoder upsamples back to the original resolution. Skip connections are shown in blue. All models are based on reimplementations of this base architecture.

each block. The output is a feature map with double or half the number of channels.

- `Down` (red arrow): 2×2 max pooling reduces spatial resolution by half, feeding into a `DoubleConv` block, which doubles the number of channels.
- `Up` (green arrow): upsampling is performed using bilinear interpolation (or optionally transposed convolution), followed by concatenation with the corresponding encoder feature map via skip connection, and a DoubleConv block, halving the number of channels.
- `Skip connections` (grey arrows): connect encoder and decoder layers at the same spatial resolution, preserving localisation information.
- `OutConv` (red): a 1×1 convolution maps the final feature map to three output logits (one per class).

In the double convolution, Batch Normalisation helps to stabilise training, and ReLU activation introduces non-linearity, mapping negative values to zero. This helps to mitigate the vanishing gradient problem and introduces sparsity in the activations. The optional Dropout layer is used to reduce overfitting by randomly disabling a fraction of neurons during training. Skip connections are crucial for preserving spatial information, allowing the decoder to access high-resolution features from the encoder. This is particularly important in segmentation tasks where precise localisation is required.

We include optional Dropout (0–0.3) to regularise training and mitigate overfitting. Upsampling mode is configurable, though we found bilinear interpolation followed by convolution to be more stable. We also experimented with different optimisers and learning rates, with AdamW yielding the best results with a learning rate of 0.0001.

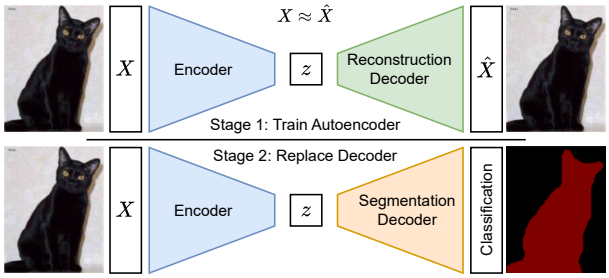


Figure 4. Segmentation pipeline with autoencoder pretraining.

Results. U-Net achieved successful results, reaching mean test IoU scores of 0.67-0.68, with a mean Dice score of 0.74-0.75. The model performed well on the test set, with 0.92-0.93 of pixel accuracy.

b) Autoencoder Pretraining for Segmentation

Autoencoders are unsupervised neural networks where their input is the same as their output: they learn to reconstruct their input by passing it through an encoder-decoder architecture. The encoder compresses the input X into a lower-dimensional representation z (latent space), while the decoder reconstructs the original input $\hat{X}$ from this representation. This is similar to the U-Net architecture, but without the need for labelled masks, eliminating skip connections, and replacing the segmentation head (what classifies the pixels) with a reconstruction decoder (rebuilds the input image) (See Figure 4). The goal is to minimise the reconstruction error, typically using a loss function like Mean Squared Error (MSE).

The encoder is expected to learn meaningful low-level visual features (e.g., edges, textures, shapes), which can later be reused for other applications like segmentation.

Implementation. Our autoencoder consists of an encoder, a reconstruction decoder, and a segmentation decoder. The encoder and decoders are implemented as convolutional neural networks, with the encoder compressing the input image into a compact latent representation and the

decoders reconstructing the image or generating segmentation maps.

The **encoder** compresses input RGB images into a 1024-dimensional latent vector using stacked convolutional layers with LeakyReLU activations and a final linear projection. The **reconstruction decoder** then upsamples this latent vector through a series of transposed convolutions to reconstruct the original image. For **segmentation**, the same encoder is reused, but the reconstruction decoder is replaced with a segmentation decoder of identical structure, except for its final output layer, which produces per-pixel class logits instead of RGB values.

We do not include skip connections in the autoencoder, as the goal is to compress and reconstruct global representations, not preserve spatial information like U-Net. This promotes learning generalisable features that can transfer across tasks. All layers use Leaky ReLU activations to prevent neuron inactivity during training.

Training Strategy. **Stage 1.** The model is first trained to minimise a reconstruction loss between the input image X and its reconstruction $\hat{X}$. We use mean squared error (MSE) as the loss function:

$$\mathcal{L}_{\text{recon}} = \frac{1}{N}\sum_{i=1}^{N} |X_i - \hat{X}_i|^2 = \frac{1}{N}\|X - \hat{X}\|_2^2 \qquad (4)$$

where N is the number of pixels per image. The model is trained to minimise this loss in the first half of the training schedule.

Stage 2. After pretraining, we discard the reconstruction decoder and retain the encoder weights. These are frozen or fine-tuned, depending on the experiment. The segmentation decoder is then trained on the labelled segmentation task using the same hybrid loss as in the U-Net (Eq. 3).

Results. For this model, we trained our experiments with an increasing number of epochs (100, 150, 200), larger batch sizes (32, 64, 128), and different learning rates to accommodate both the autoencoder pretraining and the segmentation training and to assess the impact of these hyperparameters. However, we found that the model was overfitting and struggling to improve performance, and changing these parameters did not yield significant improvements. Altogether, this was our worst-performing model, with test IoU scores of 0.56-0.57 and mean Dice scores of 0.63-0.64.

c) CLIP Features for Segmentation

CLIP (Contrastive Language–Image Pretraining) is a vision-language model trained on 400 million image–text pairs to align visual and textual information in a shared embedding space. It consists of a Vision Transformer (ViT) for images and a Transformer for text, both trained jointly such that text and image pairs that match (e.g., "a dog" and *an image of a dog*) are pulled closer together in embedding space, while mismatched ones are pushed apart. Although CLIP is typically used for image classification and zero-shot tasks, its image encoder produces rich visual representations that can be reutilised for downstream tasks such as semantic segmentation.

In our project, we explore two CLIP-based segmentation architectures:

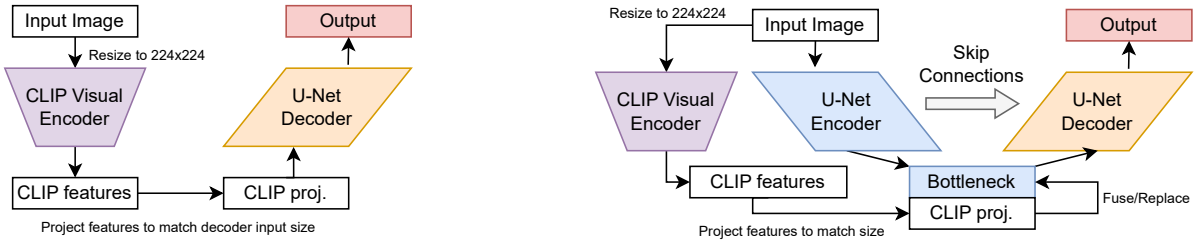
- **CLIP-only model:** See Figure 5a. The CLIP image encoder processes the input image and produces a global 512-dimensional embedding. This is projected into a spatial tensor that serves as the bottleneck for a standard U-Net, and is used as input to a segmentation decoder. Unlike the original U-Net, this model does not include skip connections, relying entirely on the global CLIP features to inform the reconstruction of a segmentation mask.

- **CLIP+U-Net model:** See Figure 5b. This model combines CLIP features with features from a U-Net encoder. The image is simultaneously passed through the CLIP encoder and a U-Net encoder; the outputs are fused, either by replacing or summing, to form a joint representation. The original U-Net skip connections are preserved and used by the decoder, which follows the same architecture as in our baseline U-Net.

In both architectures, the CLIP model is kept frozen during training, as its weights are pre-trained on a vast corpus and are considered general-purpose visual descriptors. The fusion mechanism in the CLIP+U-Net model is controlled via a `fuse_clip` flag: when enabled, the projected CLIP bottleneck is added to the U-Net encoder output before decoding instead of replaced.

Implementation. To ensure compatibility with CLIP, images are resized to 224×224 before extracting their CLIP features. For encoding, they maintain their original 256×256 dimension for consistency with other models. The resulting 512-dimensional CLIP feature vector is passed through a linear layer and reshaped into a spatial tensor of shape $512 \times 16 \times 16$, matching the spatial resolution of the U-Net bottleneck (although these are our standard dimensions, these values are calculated dynamically in our code implementation). This tensor is then passed to a U-Net decoder consisting of four upsampling blocks and a final 1×1 convolution, as explained in Section 3.

The decoder mirrors our baseline U-Net architecture and supports both with-skip and no-skip modes. In the CLIP-only model, skip connections are disabled; in the CLIP+U-Net model, they are reused to recover spatial details from the encoder. Dropout is optionally applied after each upsampling block to regularise training.



(a) CLIP-only segmentation model.

(b) CLIP+U-Net model.

Figure 5. 5a CLIP-only: The input image is passed through a frozen CLIP encoder, and the output (CLIP features) is projected to match the input dimension of the U-Net decoder and fed to it. 5b CLIP+U-Net: The image is passed through both a U-Net encoder and a frozen CLIP encoder. The CLIP encoder produces CLIP features from the image, which are projected to match the dimension of the U-Net encoder output (bottleneck). CLIP projected features are either added to the U-Net bottleneck (`fuse_clip` = True) or used to replace it (fuse_clip = False). The output is fed into the U-Net decoder, and skip connections are used to recover spatial details from the U-Net encoder.

Both models are trained in the same way as the U-Net (Section 3), with a weighted combination of cross-entropy Dice loss (Eq. 3). The training follows the shared common pipeline described earlier, using the AdamW optimiser, cosine annealing scheduler, and mixed-precision training. Augmentations, including flipping, rotation, grid and elastic distortion, are used to promote generalisation.

Despite obtaining better results with the CLIP+U-Net model, we noticed that it was overfitting. To address that, we ran experiments adding further augmentations (e.g., Gaussian noise, colour jitter) and increasing the dropout rate. However, these changes did not yield improvements, yet (slightly) lowered the performance.

Results. The CLIP-only model tests the hypothesis that general-purpose visual features (like CLIP) can provide a starting point for semantic segmentation prediction. However, our experiments suggest that while the CLIP features are semantically meaningful, the absence of skip connections limits performance, as the CLIP-only model struggles to bootstrap learning, similar to the Autoencoder.

The CLIP+U-Net model addresses this by incorporating both semantic (CLIP visual encoder) and spatial ()features. The improvement over CLIP-only highlights the importance of retaining local information. Furthermore, the fact that both this model and our baseline U-Net outperform models without skip connections (CLIP-only and autoencoder) suggests that low-level spatial context plays a critical role in segmentation.

CLIP+U-Net is our best-performing model, achieving average test IoU scores of 0.71-0.73, with a mean Dice score of 0.81-0.83. CLIP-only only achieves 0.56-0.67 average IoU.

d) Prompt-based Segmentation

Point-based segmentation refers to a class of interactive segmentation algorithms that use minimal supervision (typically a single user click) to guide the segmentation process. In our approach, we extend the CLIP+U-Net architecture by incorporating this point-based prompt as an additional input channel, forming a model we call **PointCLIP+U-Net**.

The goal of this model is to introduce human guidance into the segmentation pipeline, enabling the network to focus on specific objects or regions of interest based on a user-defined point. This is especially useful in ambiguous scenes or in multi-object images, where the model alone may struggle to resolve which object to segment.

Point Encoding. The user prompt is encoded as a *Gaussian heatmap*, which softly highlights a single location in the image. For a given input RGB image $X \in \mathbb{R}^{3\times H\times W}$ and user click at location (x_0, y_0), we generate a heatmap $h \in \mathbb{R}^{1\times H\times W}$ such that each pixel (x, y) in h is assigned:

$$h(x,y) = \exp\left(-\frac{(x-x_0)^2 + (y-y_0)^2}{2\sigma^2}\right)$$

where σ controls the spread, this produces a smooth spatial attention signal centred around the prompt. The heatmap is concatenated to the RGB image to form a 4-channel input: $\hat{X} \in \mathbb{R}^{4\times H\times W}$.

Architecture. Since CLIP+U-Net was our best-performing model, the point-based architecture (Figure 6) is based on it, with the following modifications:

- The RGB image and heatmap are concatenated and passed through a modified U-Net encoder that accepts 4-channel input. The encoder produces skip connections and a bottleneck feature map.
- The RGB image (without the heatmap) is resized to 224×224 and passed to the frozen CLIP encoder to extract a 512-dimensional embedding, which is projected and reshaped to match the bottleneck shape.

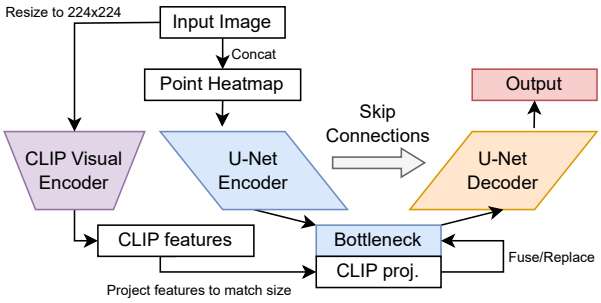


Figure 6. PointCLIP+U-Net architecture. A Gaussian point heatmap is concatenated with the input image and passed to a 4-channel U-Net encoder. CLIP features are extracted from the RGB image and fused with the encoder bottleneck. The decoder upsamples the fused features using skip connections to produce the segmentation mask.

- The projected CLIP features and the encoder bottleneck are either added or fused depending on a flag (`fuse_clip`).
- The U-Net decoder upsamples the fused bottleneck using skip connections to predict the segmentation mask.

To avoid losing information about the point heatmap by completely replacing the U-Net bottleneck with CLIP features, we add/fuse the projected CLIP features to the bottleneck. This allows the model to leverage both local (U-Net) and global (CLIP) information.

Training. The PointCLIP+U-Net model is trained using the same hybrid loss as our U-Net baseline (Equation 3), combining Cross-Entropy (CE) and Dice loss. To prioritise confident class separation, we modified the loss weights to $\alpha = 3$ for CE loss and $\beta = 0.3$ for Dice loss, emphasizing per-pixel classification accuracy. Although the weights were chosen empirically, they are inspired by the weighting strategy used in focal loss variants, where class confidence is heavily rewarded for overcoming imbalanced predictions.

During training, a single point is sampled from the ground-truth mask to simulate user interaction. This point is then encoded into a Gaussian heatmap and concatenated with the input image as a fourth channel. The model receives this additional input and learns to attend to the target region indicated by the prompt. We use the same data augmentation and training configuration as in previous models.

Results. PointCLIP+U-Net achieves 0.70-0.71 IoU and 0.81-0.82 Dice score, improving segmentation quality compared to models without skip connections (e.g., CLIP-only, autoencoder), and outperforming the U-Net. However, it does not achieve as high per-class average IoU as the CLIP+U-Net model.

4. Evaluation and Comparison

This section presents a comprehensive evaluation of the models discussed in Section 3. We report segmentation performance on the test set using Dice score, Intersection-over-Union (IoU), and pixel accuracy, and compare the results across the four main models, U-Net, Autoencoder, CLIP-only, and CLIP+U-Net.

4.1. Evaluation Metrics

We report three common segmentation metrics, each computed per class (background, cat, dog) and averaged over classes. Let Y_c and $\hat{Y}_c$ be the ground-truth and predicted masks for class c, respectively. The metrics are defined as follows:

Dice Score measures overlap between predicted and ground-truth masks:

$$\text{Dice}_c = \frac{2|\hat{Y}_c \cap Y_c| + \epsilon}{|\hat{Y}_c| + |Y_c| + \epsilon}$$

It is sensitive to both false positives and false negatives.

Intersection over Union (IoU) or Jaccard index:

$$\text{IoU}_c = \frac{|\hat{Y}_c \cap Y_c| + \epsilon}{|\hat{Y}_c \cup Y_c| + \epsilon}$$

Pixel Accuracy computes the percentage of correctly predicted pixels:

$$\text{Pixel Accuracy} = \frac{\sum_i \mathbb{1}(\hat{y}_i = y_i)}{N}$$

To ensure fair comparison, predictions are interpolated back to the original image size before metric computation.

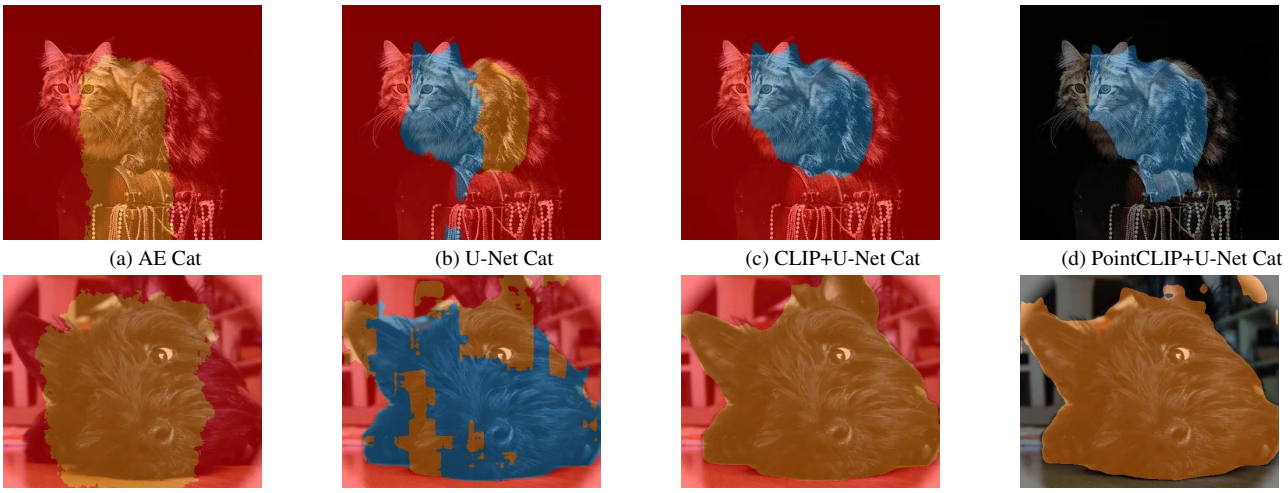
4.2. Quantitative Results

Table 1 reports the test performance of the four segmentation models, the best-performing of each type. All models outperform the provided baseline (mean IoU = 0.33), with CLIP+U-Net achieving the best results overall, reaching a Dice score of 0.83 and a mean IoU of 0.72.

Model	Mean IoU	Dice Score	Pixel Acc.
U-Net	0.68	0.75	0.87
Autoencoder	0.56	0.63	0.74
CLIP-only	0.58	0.71	0.80
CLIP+U-Net	**0.72**	**0.83**	**0.88**

Table 1. Test performance across all models.

The CLIP+U-Net model outperforms the U-Net baseline by 4-8% in mean IoU and Dice score, demonstrating the effectiveness of combining CLIP features with U-Net's



(a) AE Cat (b) U-Net Cat (c) CLIP+U-Net Cat (d) PointCLIP+U-Net Cat

(e) AE Dog (f) U-Net Dog (g) CLIP+U-Net Dog (h) PointCLIP+U-Net Dog

Figure 7. Comparison of segmentation methods on cat (top) and dog (bottom) images.

spatial information. The autoencoder model performs the worst, likely due to its lack of skip connections and reliance on reconstruction pretraining. The CLIP-only model, while leveraging powerful semantic features, fails to match the performance of CLIP+U-Net, highlighting the importance of spatial information in segmentation tasks. Pixel accuracy remains high for all models, likely due to background class dominance.

Model	Background	Cat	Dog
U-Net	**0.84**	0.68	0.52
Autoencoder	0.75	0.67	0.27
CLIP-only	0.68	**0.69**	0.36
CLIP+U-Net	0.82	0.68	**0.66**

Table 2. Per-class IoU scores on test set.

Table 2 highlights key differences in class-wise performance across models. U-Net achieves the highest background IoU (0.84), likely due to its dominant representation in the dataset. CLIP-only, despite its weaker overall performance, performs best on the cat class (0.69), possibly reflecting the semantic strength of CLIP features for fine-grained recognition. However, its lack of spatial grounding leads to poor localisation on the dog class (0.36). The autoencoder shows degraded performance across all classes, particularly dogs (0.27), supporting our earlier observation that omitting skip connections and relying solely on global reconstruction is problematic for segmentation detail.

CLIP+U-Net achieves high scores for both background (0.82) and dogs (0.66), while maintaining strong cat performance (0.68). These results reinforce that combining pretrained semantic features (CLIP) with spatial detail from skip connections (U-Net encoder) leads to better generalisation across diverse classes.

4.2.1 Qualitative Comparison

Figure 7 presents segmentation outputs on representative cat and dog examples across the four models. Consistent with the per-class IoU results (Table 2), the CLIP+U-Net and PointCLIP+U-Net models achieve significantly better spatial coherence and boundary alignment, particularly for dog masks.

The autoencoder consistently underperforms in both examples. U-Net demonstrates success in coverage, but mixes dog and cat classes. In the dog image, the detection is accurate, but half of it is detected as a cat. The addition of CLIP features dramatically improves boundary precision and semantic awareness.

4.2.2 Impact of Pretraining and Architecture

Our experiments reveal key insights about the role of pretraining and architectural design in segmentation performance. The autoencoder-based model, which uses unsupervised pretraining on reconstruction, learns general low-level features but struggles to capture semantic boundaries without skip connections. Despite training for up to 200 epochs with increased batch sizes (up to 128), its performance plateaued early, suggesting limited capacity for dense segmentation.

The inclusion of pretrained CLIP features significantly boosts performance, but only when combined with spatial guidance. The CLIP-only model performs poorly on its own, lacking spatial resolution and skip connections. In contrast, the CLIP+U-Net hybrid outperforms all baselines,

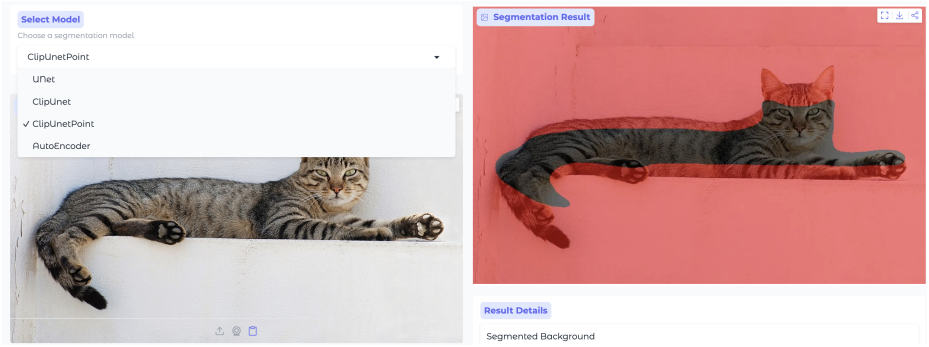


Figure 8. User interface. Highlighting the background after clicking on the uploaded image. The dropdown menu is displayed.

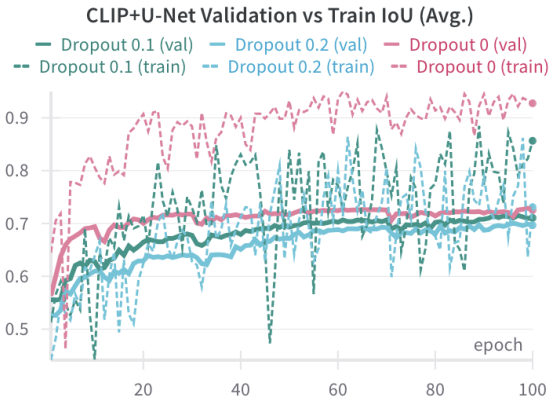


Figure 9. Train vs validation IoU comparison for CLIP+U-Net with different dropout.

indicating that pre-trained need further information for effective segmentation.

We also examined generalisation. While the CLIP+U-Net model achieved the best test scores, training/validation curves showed signs of overfitting. We retrained the model using dropout regularisation (rates 0.1 and 0.2), which yielded slightly lower test scores but greatly improved generalisation. This is depicted in Figure 9, where we can observe how 0 dropout has the highest training IoU.

Finally, extensive hyperparameter searches with variations in batch size (32–128), dropout, class weights, and augmentations confirmed that architectural changes (e.g., CLIP fusion, skip connections) had far more impact on performance than tuning these parameters alone. This highlights the importance of model structure over shallow optimisation tweaks.

4.3. User Interface

To complement our prompt-based segmentation model, we developed an interactive user interface that allows users to visualise and test all segmentation models explored in this project. While the coursework encouraged the development of a point-based interface, we extended this idea by integrating support for our best-performing models: `UNet`, `ClipUnet`, `ClipUnetPoint`, and `AutoEncoder`.

The interface, shown in Figure 8, allows users to upload an image from the camera, clipboard or file browser. The user selects a model from a dropdown menu and, when using `ClipUnetPoint`, interactively click on a region of interest. The click is converted into a Gaussian heatmap prompt and passed to the model for inference. The resulting segmentation mask is overlaid on the image and displayed to the user in real-time.

This application was deployed using Gradio and HuggingFace Spaces and serves as both a demonstrator for our models and a practical interface for experimentation. While our primary implementation supports point-based prompting, the modular design allows for future extension to other prompt types (e.g., bounding boxes or scribbles). The application can be accessed in this link[1], and a video demonstration is available (2), showing how the interface works, how it guides the user through the segmentation process, and how to use the different models.

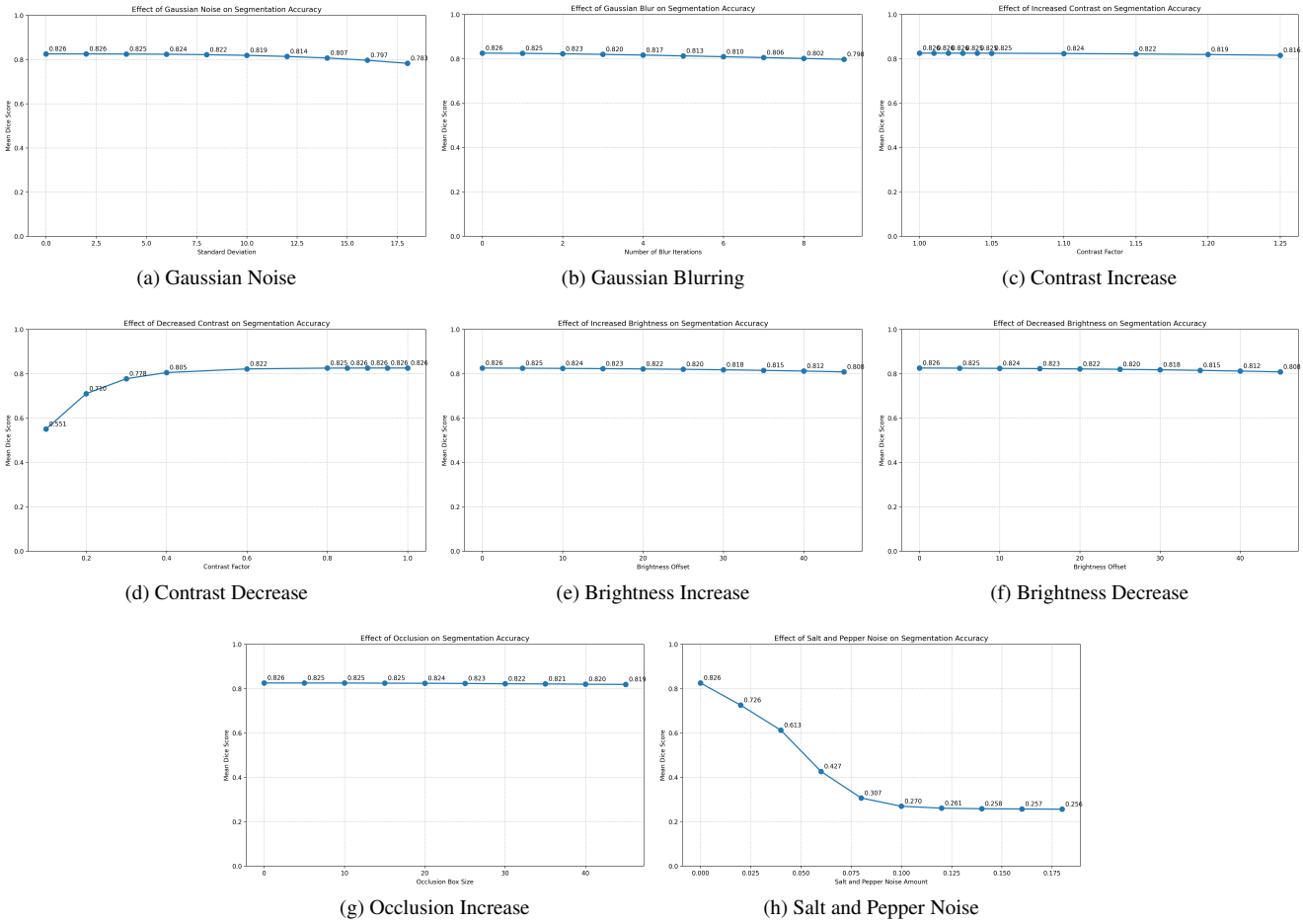
5. Robustness

To assess generalisation beyond clean data, we evaluate the robustness of the CLIP+U-Net model under common visual perturbations. We apply eight types of noise and image corruptions to the test set and measure the impact on segmentation performance using the mean Dice score.

Figure 10 shows the mean Dice scores of the CLIP+U-Net model under different perturbations. Overall, the model demonstrates strong robustness, maintaining scores in the range of 80–82% across most perturbations, comparable to its unperturbed test performance (mean Dice: 82.3%).

Most perturbations lead only to small drops in the Dice score (1–2%), reflecting the model's strong generalisation

[1] https://huggingface.co/spaces/bhenriquezsoto/point-based-segmentation-1



(a) Gaussian Noise (b) Gaussian Blurring (c) Contrast Increase

(d) Contrast Decrease (e) Brightness Increase (f) Brightness Decrease

(g) Occlusion Increase (h) Salt and Pepper Noise

Figure 10. Robustness of the CLIP+U-Net model under eight perturbation types applied to the test set. Mean Dice scores are plotted against increasing perturbation strength, showing how model performance degrades under challenging conditions.

capability. Two notable outliers appear: **salt-and-pepper noise** and **contrast decrease**. Salt-and-pepper noise introduces high-frequency pixel corruption that disrupts local textures and boundaries, causing Dice scores to fall significantly, down to 25%. This sensitivity is likely due to the lack of such high-frequency, non-continuous noise in the training augmentations. While Gaussian noise and elastic deformation were included, they do not replicate the effect of salt-and-pepper noise. In contrast, the model performs better as contrast decreases. At maximum contrast reduction, the Dice score peaks at 82.6%, outperforming the score at lower contrast shifts (as low as 55%). A possible explanation is that lower contrast may suppress irrelevant texture and background clutter, simplifying the segmentation task and enabling the model to focus more on global object structure.

These findings suggest that while the model is robust to photometric variations, it remains vulnerable to high-frequency spatial noise. Including training augmentations that mimic salt-and-pepper corruption could help mitigate this limitation.

6. Discussion and Conclusion

Our experiments suggest that architectural choices had a far greater impact on segmentation quality than regularisation or hyperparameter tuning. Likewise, the performance gap between the autoencoder and U-Net-based models highlights the importance of skip connections for spatial precision. CLIP-only models struggled without spatial grounding, reinforcing that pretrained features must be paired with architectures capable of preserving fine detail.

For future work, our findings suggest that network capacity and architecture are the key parameters to look at. While Dropout improved generalisation, it didn't benefit performance, suggesting either a fallout on local minima or insufficient structural capacity to achieve better solutions. Overall, meaningful improvements stemmed from architectural integration, not tuning.

References

[1] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com. 3

[2] UoE/CV Group 50. Interactive segmentation interface demo. YouTube Video, Jun 2024. [https://youtu.be/wJlaTmeq-0M]. 9

A. Code

To create maintainable and extensible code, we use `wandb` and `poetry`. The former allows us to track experiments, record selected parameters, and monitor different metrics. The latter helps prevent issues arising from having different versions of libraries—namely, `tqdm`, `albumentations`, `numpy`, `matplotlib`, `torch`, `torchvision`, `pillow`, `scikit-learn`, `transformers`, `scikit-image`, `wandb`, and `clip`.

With the same goal, instead of having four different trainers for each model, we opted for a unique pipeline that handles model-specific components before returning to the general training pipeline (calculating loss, performing gradient steps, computing metrics, etc.).

A.1. Project Structure

This section provides an overview of the project structure and explains the main objective of each file.

The `Dataset` folder contains all datasets (both color images and masks) for training, validation, and testing. In `robustness_results`, you will find plots showing how the mean Dice score varies with different levels of perturbation, along with examples demonstrating the changes in images for each level.

Within the `src` folder, the Python files are organized as follows:

`app.py` contains the code for the web application (downloading and preloading models, inference, user interface, etc.), available here[2].

`data_loading.py` includes all data augmentations and most pre-processing tasks such as generating a heatmap from a point and retrieving images from the dataset folder.

`metrics.py` defines our evaluation metrics (adaptive multifocal loss, Dice loss, IoU score, pixel accuracy, etc.).

`robustness_test.py` contains the code necessary to perform each of the perturbations described in the project statement, allowing computation of results either individually or all at once, and generates examples and plots of the mean Dice score for the selected perturbations (designed to be run from the terminal).

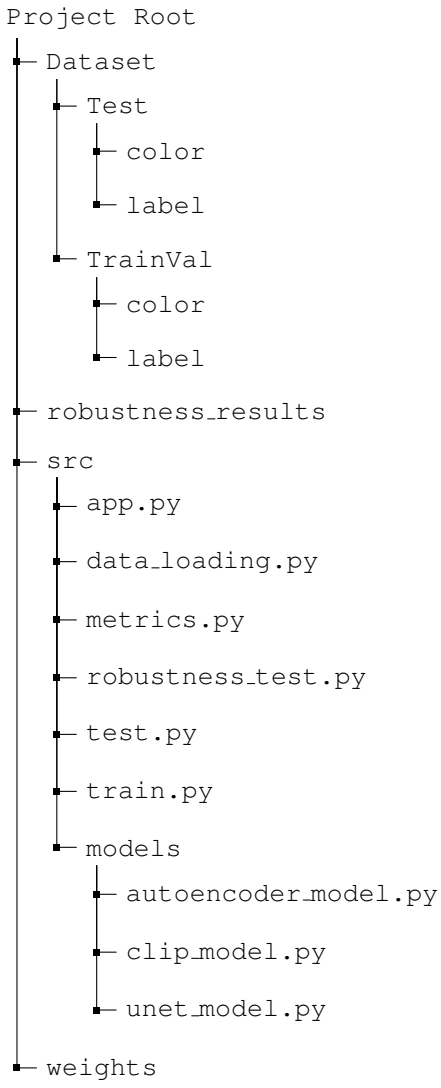
`test.py` includes the code to test a specific model. Provided with a model weight path, it displays in the terminal the overall mean Dice score, IoU score, pixel accuracy, as well as class-specific Dice and IoU scores.

`train.py` is likely the largest module in our project. It integrates various pieces of code from the Python files to create a complete training pipeline, enabling the easy addition of different optimizers, losses, and models. This module is intended to be run from the terminal and supports configurable parameters such as batch size, model selection, number of epochs, optimizer choice, class weights, dropout rate, and more.

The `weights` folder contains the best-performing weights for each of the four models described.

Finally, the `models` folder hosts the Python files defining each model class. In `clip_model.py`, you'll find the CLIP base architecture along with the `clipUNet` class, which uses the UNet architecture. By using the **fuse_clip** parameter, the model can either add the CLIP features to the UNet features or replace them and pass the modified features to the decoder to generate the mask. The `PointCLIPUNet` class is very similar to `CLIPUnet`, except it accepts an additional input (a heatmap generated by the user click) to segment the selected object (provided it falls under categories such as cat, dog, or background). The `autoencoder_model.py` defines the architecture for both encoders and its decoder, and in `unet_model.py` we have both the base UNet model and the point-based variant, which, similar to `PointCLIPUNet`, accepts an extra input (a user-generated heatmap) for segmentation.

[2] https://huggingface.co/spaces/bhenriquezsoto/point-based-segmentation-1


Project Root
Dataset
Test
color
label
TrainVal
color
label
robustness_results
src
app.py
data_loading.py
metrics.py
robustness_test.py
test.py
train.py
models
autoencoder_model.py
clip_model.py
unet_model.py
weights