Pet Segmentation UNet

Peilin Rao, Jake Ekoniak, Ananya Sampat, Dylan Shah, Nalin Chopra

Abstract

In this project, a modified U-Net architecture is used for pet segmentation using the Oxford-IIIT Pet Dataset. The standard U-Net is compared with its fast variant, which incorporates depthwise separable convolutions and bilinear upsampling to reduce the number of model parameters while retaining the segmentation quality. An extensive hyperparameter tuning and experimentation with different loss functions reveal that our approach is accurate and efficient and, thus, suitable for fine-grained segmentation tasks in limited resource settings.

1 Introduction

Image segmentation is one of the most important problems in computer vision with numerous applications, including medical imaging, autonomous systems, and wildlife monitoring. In this project, we consider the problem of pet segmentation which consists in proper delimitation of cats and dogs in images using the Oxford-IIIT Pet Dataset. This dataset contains a rich set of high-quality images with pixel level annotations which makes it a suitable benchmark for segmentation models. This type of problem is especially convenient to solve with the help of CNN. Their capability to learn feature hierarchies with local filters and shared parameters enables them to learn both high level features and low level details. These properties make CNNs suitable for complex segmentation tasks where the spatial information is very important. Thus, our approach is based on the U-Net architecture that is built on the encoder-decoder principle and skip connections to preserve relevant spatial features in the process of segmentation. For enhancing the efficiency, our model has incorporated some changes such as depth wise separable convolutions and bilinear upsampling. These modifications decrease the number of parameters and the training time without compromising the accuracy of the model. In general, the present study is aimed at designing a optimal design of a pet segmentation model based on CNNs and improved U-Net framework.

2 Background

In this section, we will offer background information, both historical and mathematical, about the CNN and U-Net architectures.

2.1 CNN Overview

2.1.1 Motivation and Background

Originally introduced by Yann LeCun in his 1998 paper "Gradient-Based Learning Applied to Document Recognition", the Convolutional Neural Network (CNN) is a foundational model in machine learning, particularly for computer vision[1]. CNNs excel at processing grid-like data, such as images, by automatically learning hierarchical features. Unlike traditional neural networks, CNNs use localized filters and parameter sharing, making them computationally efficient

and easier to train. Key components include convolutional layers for feature extraction, activation functions for non-linearity, pooling layers for dimensionality reduction, and fully connected layers for final output. Their translation invariance and parameter efficiency make CNNs highly effective for tasks like image classification, object detection, and segmentation, solidifying their role as a cornerstone of modern deep learning.

2.1.2 CNN Technical Description

Mathematically, a convolution between two functions f and g is defined as

$$(f * g)(x) = \int f(t)g(x-t)dt$$

Intuitively, this measures how much one function overlaps with another when one is shifted by some value t. In image processing, we typically use a discrete 2D convolution. In this case, the continuous integral is replaced by a summation and we consider two dimensions instead of one. Specifically, if I is an inpute image of size $(H \times W)$ and K is kernel (filter) of size $(k \times k)$, then the 2D discrete convolution at position (x, y) is given by:

$$(I * K)(x, y) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(x+m, y+n)\, K(m, n).$$

Here, the kernel "slides" over the image one (or more) pixels at a time, multiplying the corresponding region of the image by the kernel values and then summing the result to provide a single output value. Repeating this for all valid (x, y) positions yields the feature map, which highlights the specific patterns in the image as dictated by the kernel. Although the operation described above is often referred to as a 'convolution' in deep learning frameworks, in strict mathematical terms it more closely resembles a cross correlation. Nevertheless, the terminology "convolution" is what is standard in the context of CNNs. This process is demonstrated pictorially in Figure1.

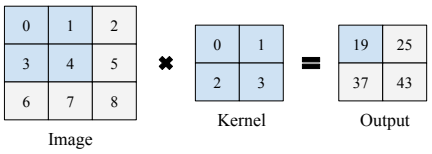


Figure 1: Discrete 2D Convolution

Because this convolution operation is fundamentally a linear transformation, most convolutional layers include a subsequent activation function (such as ReLU). This nonlinearity serves the same purpose as the nonlinear activation function in a standard MLP.

In the example shown, the output feature map has a smaller spatial dimension than the input. When this occurs, it is referred to as a downsample convolution. If preserving the spatial dimension is required, dummy values can be added around the edge of the image, a process known as padding, to ensure the output has the same dimension.

Beyond convolutions, pooling operations are another way to reduce the size of the feature maps. These techniques are similar to downsample convolutions but they are not learned, instead their output is deterministic. Popular pooling techniques are max pooling and average pooling, which select the maximum value of a local path or the average value of the local patch respectively, demonstrated in Figure 2.

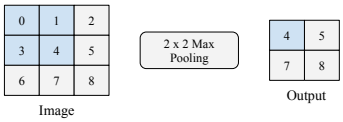


Figure 2: 2D Max Pooling Downsample

In most CNN architectures, multiple convolution, activation, and pooling layers are stacked together. Conceptually, each "pixel" in the deeper feature map encodes information about many regions of the original image, those regions fall within its receptive field. After several layers of convolution and pooling, the receptive field can cover a large portion of the input, potentially the whole image. This enables the network to capture both local structure the global context.

When these layers are applied repeatedly, the spatial dimensions of the feature maps shrink, but essential information is preserved or emphasized through the learned filters. Finally, once the last convolution (and optional pooling) layer has produced one or more feature maps, these maps are either pooled again or flattened into a vector and passed into a Multilayer Perceptron (MLP), referred to as the dense layers of the network. These dense layers perform tasks like classification or regression based on the task of the network.

During backpropagation in a convolutional neural network (CNN), both the weights in the dense layers and the kernel values are updated. This enables the kernels to learn how best to capture critical patterns. This process was visualized in a paper "Visualizing and Understanding Convolutional Networks", where it was found that the initial convolutions learn to recognize larger patterns, such as colors and shapes (as shown in Figure 3), while later convolutions focus on more fine-grained details.

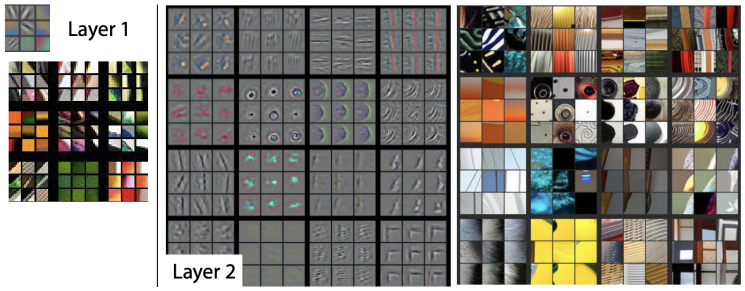


Figure 3: Visualization of CNN Features[2]

2.2 UNet Overview

2.2.1 Motivation and Background

Convolutional Neural Networks (CNNs) have incredibly successful across various computer vision tasks. Moreover, much work has been done to optimize them, with architectures such as AlexNet[3], ResNet[4], and EfficientNet[5] marking significant milestones. However, one significant challenge with the traditional CNN approach is tasks that demand precise spatial understanding. While CNNs excel at capturing high-level context, such as classifying a discrete object in a picture(as in ImageNet[6]), they struggle to preserve fine-grained details. One such task that requires this is image segmentation.

A solution for such problems can with the paper "U-Net: Convolutional Networks for Biomedical Image Segmentation" by Olaf Ronneberger and colleagues[7]. U-Net introduced

an architecture that, through upsampling, is able to recover spatial resolution that is lost during downsampling. This allows the network to make detailed, pixel-level predictions, which is critical for many applications, from the medical domain to autonomous vehicles.

One important technique to introduce when discussing U-Nets is upsampling. In the same way that downsampling is a transformation from a higher dimension space to a lower one, as we saw in Figure 2, upsampling is a transformation from a lower dimension space to a higher one. Some common approaches include, nearest neighbors interpolation, bilinear upsampling, and transposed convolutions. Mathematically, a transposed convolution of an image I and kernel K can be represented as follows:

$$(I *^T K)(x, y) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I\left(\left\lfloor \frac{x+p-m}{s} \right\rfloor, \left\lfloor \frac{y+p-n}{s} \right\rfloor\right) K(m, n)$$

where k is the kernel size, s is the stride, and p is the padding. If I has height H and width W, then the dimension of the output feature map is generally

$$((H-1) \times s - 2p + K) \times ((W-1) \times s - 2p + k).$$

As can be seen, the transposed convolution is not an exact inverse of the standard convolution, but is better described as a learnable upsampling operation that approximates the inverse of the standard convolution.

2.2.2 U-Net Architecture

The U-Net architecture is a type of encoder-decoder architecture, where the model is divided into two main parts: the encoder(consisting of multiple encoder blocks) and the decoder(comprising multiple decoder blocks). The process begins with the encoder.

Each encoder block consists of several dimension-preserving convolutions followed by a non-linear activation function. These convolutions extract features while maintaining spatial resolution. Next, a downsampling step is performed before the next encoder block. This process is repeated as many times as desired, and the number of encoder blocks used can describe the "depth" of the model in question. Due to the successive downsampling layers, the dimension of the feature map becomes smaller and smaller. Eventually, the network reaches a bottleneck layer. This serves as the bridge between the encoder and the decoder, and is where the feature map has the smallest spatial dimension.

From the bottleneck, the decoder part of the network begins, mirroring the structure of the encoder. Each decoder block consists of upsampling (often via transposed convolution) to progressively recover spatial resolution lost during downsampling. Additionally, skip connections (horizontal arrows in the U-Net diagram) connect each encoder block to its corresponding decoder block by concatenating the encoder's features with the decoder's, helping the network retain fine-grained spatial details.

Unlike a standard CNN, the U-Net does not conclude with fully connected (dense) layers. Instead, the network ends with a 1×1 convolutional layer, producing segmentation maps where each pixel corresponds to one or more target classes. An example architecture from the original U-Net paper is provided in Figure 4.

3 Dataset

We used the Oxford-IIIT Pet Dataset. This dataset is a widely recognized benchmark for image segmentation and classification tasks. It comprises 7,349 high-quality images of cats and dogs, spanning 37 distinct breeds. Each image is annotated with pixel-level segmentation

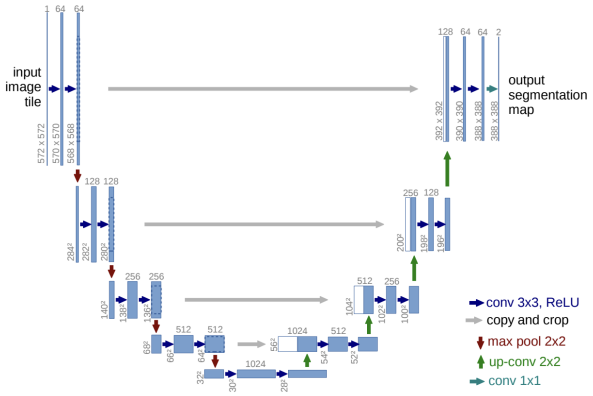


Figure 4: UNet Architecture[7].

masks, which include three classes: the foreground pet, the background, and a trimap region representing ambiguous or transitional pixels. The dataset is evenly distributed between cat and dog images, with approximately 200 images per breed, ensuring a balanced representation. An example of two data instances, a cat and a dog along with their segmentation maps, is displayed in Figure 5.

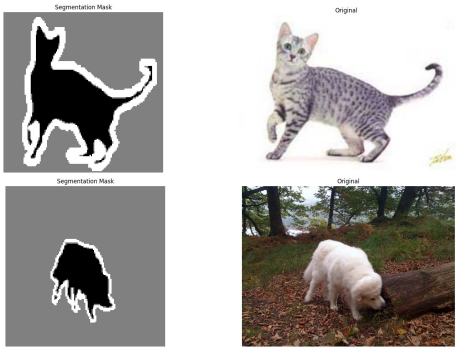


Figure 5: Data Instances [6].

At the time of its introduction, the dataset addressed a critical need for large-scale, pixel-wise labeled data, which was notably lacking during the development of pioneering architectures like U-Net. This makes the Oxford-IIIT Pet Dataset one of the most valuable resources for advancing research in image segmentation. Its fine-grained annotations, diverse pet poses, and varying levels of occlusion and background complexity make it an ideal benchmark for evaluating state-of-the-art methods in semantic segmentation, instance segmentation, and object recognition. This dataset has no immediate ethical concerns that require discussion.

4 Model

Due to the a large training time, we decided to implement our U-Net with depthwise convolutions and bilinear upsampling. These modifications make our model significantly faster to train while introducing little performance loss. A mathematical description of both is found below.

4.1 Inception Model

In the CNN design, conventional convolution operations require large amount of memory to store the kernels. Large memory consumption not only impose difficulty on training, but also makes it hard for the trained model to run on mobile devices. Separable convolution framework called Inception modules can significantly reduce memory by separating one kernel operation to two steps: cross-channel and spatial processing [8].

For standard convolution, given an input image tensor $X \in \mathbb{R}^{H\times W\times M}$, the output ($Y \in \mathbb{R}^{H_{\text{out}}\times W_{\text{out}}\times N}$), is computed by

$$Y_{n,i,j} \;=\; \sum_{m=1}^{M}\sum_{r=1}^{k_h}\sum_{s=1}^{k_w} X_{m,i+r-1,\,j+s-1}\, W_{n,m,r,s},$$

where M and N are the number of input and output channels, and (k_h, k_w) is the kernel size. The Inception modules decompose this operation into two stages. First, a 1×1 convolution compress M channels in X to an intermediate feature map with P channels:

$$\widetilde{X} \in \mathbb{R}^{H\times W\times P}, \quad \widetilde{X}_{p,i,j} \;=\; \sum_{m=1}^{M} X_{m,i,j}\, W^{(1\times 1)}_{p,m}, \quad p = 1,\ldots,P.$$

Then, the P channels are equally divided into B groups, and for each group, a separate spatial convolution with same kernel size is applied:

$$\widetilde{Y}^{(b)}_{q_b,i,j} \;=\; \sum_{p\in S_b}\sum_{r=1}^{k_h}\sum_{s=1}^{k_w} \widetilde{X}_{p,i+r-1,\,j+s-1}\, W^{(b)}_{q_b,p,r,s}.$$

The output of all groups are then concatenated:

$$Y_{n,i,j} = \text{concat}\Big(\widetilde{Y}^{(1)}_{n,i,j}, \widetilde{Y}^{(2)}_{n,i,j}, \ldots, \widetilde{Y}^{(B)}_{n,i,j}\Big).$$

The parameter B is the parameter for the level of decoupling between cross-channel and spatial processing, ranging from $B = 1$ (almost standard convolution) to $B = P$ (each channel is processed separately). When the number of channel groups B is increased, the number of parameters of the spatial convolution is reduced. The parameter count for a standard convolution is $N \times M \times k_h \times k_w$, where M and N are the numbers of input and output channels. The Inception style module has a form of an initial 1×1 convolution that compress the M channels to P channels, followed by a grouped spatial convolution. If the P channels are divided into B groups (where each group has approximately P/B channels), then each spatial filter acts on a smaller set and costs approximately

$$\frac{P^2 \times k_h \times k_w}{B}.$$

Thus, If B increases, the term $\frac{P^2\times k_h\times k_w}{B}$ decreases. Hence, there are fewer parameters.

The extreme case is when $B = P$ (i.e. each channel is convolved independently). This operation is also known as depthwise separable convolution and only requires $M\times k_h\times k_w$ parameters.

4.2 Depthwise Separable Convolution

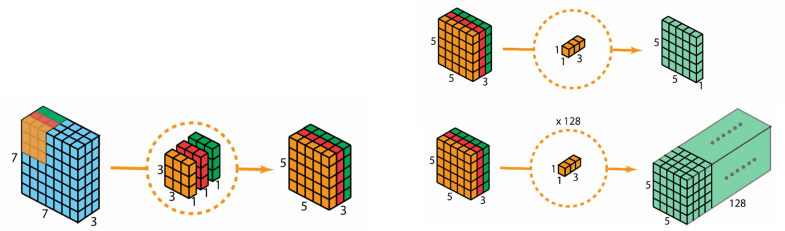
In real world application, the depthwise convolution has some slight variation from the inception module[9]. The depthwise convolution first applies a spatial filter to each channel independently:

$$D_{m,i,j} = \sum_{r=1}^{k_h}\sum_{s=1}^{k_w} X_{m,i+r-1,\,j+s-1}\, W^{(\text{spatial})}_{m,r,s},$$

producing $D \in \mathbb{R}^{H_{\text{out}} \times W_{\text{out}} \times M}$. The channel mixing is then done using a pointwise 1×1 convolution:

$$Y_{n,i,j} = \sum_{m=1}^{M} D_{m,i,j} \, W_{n,m}^{(\text{channel})}.$$

The parameter count is then: $M \times k_h \times k_w + M \times N$, which is much smaller than that of a standard convolution. It is demonstrated in Figure 6.



((a)) Depthwise first step ((b)) Depthwise second step

Figure 6: Depthwise convolution steps [10].

4.3 Bilinear Upsampling

Another way of speeding up training is by using bilinear upsampling instead of transposed convolutions/deconvolutions in the decoder. This is done because bilinear upsampling uses fixed interpolation weights rather than learned parameters.

The pixel intensity of a point (x, y) in the upsampled domain is interpolated by bilinear interpolation as a weighted sum of the intensities of the four nearest neighbors of the original grid. If we denote the intensities of the four nearest pixels (x_1, y_1), (x_2, y_1), (x_1, y_2), and (x_2, y_2) as $I(x_1, y_1)$, $I(x_2, y_1)$, $I(x_1, y_2)$, and $I(x_2, y_2)$. Then linear interpolation (similar to LaGrange interpolation) would be:

$$I(x, y) = (1 - \alpha)(1 - \beta)\, I(x_1, y_1) + \alpha(1 - \beta)\, I(x_2, y_1) + (1 - \alpha)\beta\, I(x_1, y_2) + \alpha\beta\, I(x_2, y_2)$$

where

$$\alpha = \frac{x - x_1}{x_2 - x_1}, \quad \beta = \frac{y - y_1}{y_2 - y_1}$$

Bilinear upsampling, shown in Figure 7 is a simpler method that does not involve any learned parameters. However, we found that the decreases in our accuracy were negligible while the increase in training speed were significant. We attribute this to the presence of skip connections, which help the model retain features that might be lost in upsampling.

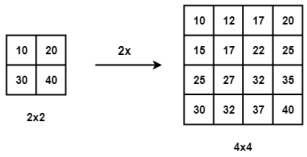


Figure 7: Bilinear Interpolation [11].

4.4 Model Architecture

The modified Unet with depthwise convolution and bilinear upsampling is composed of a number of convolutional layers, batch normalization, and ReLU activations, interspersed with certain specialized blocks, e.g. FastDoubleConv and FastDecoderBlock. With a total of 3,898,462 parameters and an estimated memory footprint of 327.31 MB, the design is a good balance between complexity and resource requirement. The full model architecture is in the appendix: 3

Grid search in Table 4, as will explain later in section 6, test different hyperparameters. The combination of hyperparameter that achieve the highest IoU is shown below:

Table 1: Hyperparameter Configuration

Batch Size	Criterion	LR	Num Epochs	Optimizer	Weight Decay	Score
32	dice	0.0100	100	adam	0.0000	0.5980

5 Methodology

In this section, we will go over our methodology. This includes our preprocessing phase and choice of hyperparameters. All preprocessing and training was done with PyTorch (version 2.6.0) with a 4070 GPU.

5.1 Preprocessing

Our preprocessing pipeline was developed using PyTorch's Dataset and DataLoader classes. Specifically, a custom dataset class was implemented to retrieve data instances directly from the source while applying the necessary masks and transformations. This dataset class serves as the foundation for three distinct DataLoaders—training, testing, and validation, which were generated via a random split comprising 72%, 20%, and 8% of the total dataset, respectively.

The transformations we utilized for our project were: resizing, random resized cropping, horizontal flipping, color jittering, and normalization. A more in-depth description of these can be seen below.

5.1.1 Resizing

Resizing images is an essential preprocessing step that ensures all inputs to the model are of uniform size. By standardizing dimensions, models can process batches efficiently while reducing computational overhead. Resizing can sometimes lead to a loss of important image details, but it remains crucial for training stability. Moreover, resizing to a smaller dimension is an easy technique to decrease training time. During our hyperparameter search, images were resized to $(48, 48)$ to allow for fast training, while our final model was trained on the dataset images resized to $128, 128$ for greater accuracy.

5.1.2 Random Resized Cropping

Random cropping involves extracting a subregion of an image and resizing it back to the original dimensions. This process encourages the model to learn different portions of the image, preventing over-reliance on specific features present in only one region. By exposing the model to multiple perspectives of an object, it improves recognition and generalization to new images. While we tested random resized cropping, we did not implement it in our final model.

5.1.3 Horizontal Flipping

Horizontal flipping simulates changes in viewpoint by mirroring the image along the vertical axis. This technique is particularly useful in object detection and classification tasks where left-right orientation does not affect the object's identity, such as human faces or animals. However, caution must be taken in applications like medical imaging, where such transformations may distort critical anatomical structures. Horizontal flipping was applied per image with a 50% chance.

5.1.4 Color Jittering

Color jittering modifies brightness, contrast, saturation, and hue randomly, ensuring that the model does not become overly reliant on specific color distributions. This technique is useful for improving robustness to lighting variations and different camera conditions, making the model more adaptable to real-world settings. Our color jitter was conducted with the following parameters: brightness = 0.2, contract = 0.2, saturation = 0.2, hue = 0.1.

5.1.5 Normalization of Image Data

Normalization is a fundamental preprocessing step that adjusts image pixel intensities to match a predefined distribution. In this project, we adopt the widely used ImageNet normalization scheme: channel-wise means of [0.485, 0.456, 0.406] and standard deviations of [0.229, 0.224, 0.225]. This process stabilizes numerical computations and ensures consistent gradient behavior during training.

5.2 Tuning the Loss Function

In our implementation of the U-Net model for pet segmentation, selecting an appropriate loss function was critical for optimizing segmentation performance. Loss functions measure the difference between predicted segmentation masks and ground truth masks, guiding the model's learning process. In this section, we discuss the loss functions used in our project, including Cross-Entropy Loss, Intersection over Union (IoU), Dice Loss, and Jaccard Loss.

5.2.1 Cross-Entropy Loss

Cross-Entropy Loss is widely used for classification tasks, including pixel-wise classification in image segmentation. It measures the dissimilarity between the predicted probability distribution and the ground truth distribution. For a given pixel i and class c, the cross-entropy loss is defined as:

$$\mathcal{L}_{CE} = -\sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}), \tag{1}$$

where C is the number of classes, $y_{i,c}$ is a binary indicator (1 if the true class is c, 0 otherwise), $\hat{y}_{i,c}$ is the predicted probability for a class c at pixel i. While, cross-entropy is effective for pixel-wise classification, it does not explicitly consider spatial structure, which is crucial for segmentation.

5.2.2 Intersection over Union (IoU)

Intersection over Union (IoU), also called the Jaccard Index, is a common evaluation metric in segmentation tasks. It measures the overlap between predicted and ground truth masks. Given a predicted mask P and a ground truth mask G, IoU is defined as:

$$IoU = \frac{|P \cap G|}{|P \cup G|} = \frac{\sum PG}{\sum P + \sum G - \sum PG}. \tag{2}$$

Since IoU is a measure rather than a loss function, we adapt it by defining the Jaccard Loss.

5.2.3 Jaccard Loss

Jaccard Loss is derived from the IoU metric and is used to optimize the model for better spatial alignment between prediction and ground truth. It is defined as:

$$\mathcal{L}_{Jaccard} = 1 - \frac{\sum PG + \epsilon}{\sum P + \sum G - \sum PG + \epsilon}, \tag{3}$$

where ϵ is a small smoothing term to prevent division by zero. This loss function optimizes IoU, which is particularly useful for segmentation tasks.

5.2.4 Dice Loss

Dice Loss is another overlap-based loss function inspired by the Dice Coefficient, a similarity measure between sets. It is particularly effective for imbalanced segmentation tasks. The Dice Coefficient is given by:

$$Dice = \frac{2|P \cap G|}{|P| + |G|} = \frac{2\sum PG}{\sum P + \sum G}. \tag{4}$$

To use it as a loss function, we define Dice Loss as:

$$\mathcal{L}_{Dice} = 1 - \frac{2\sum PG + \epsilon}{\sum P + \sum G + \epsilon}, \tag{5}$$

where ϵ is a smoothing term to stabilize training.

5.2.5 Tuning the Loss Function

To improve segmentation performance, we experimented with different loss functions. Initially, we used Cross-Entropy Loss, but it does not directly optimize for spatial consistency. To address this, we incorporated Dice Loss and Jaccard Loss, emphasizing region-based similarity rather than pixel-wise classification.

Dice Loss and Jaccard Loss penalize errors based on the intersection-over-union principle, making them more suitable for segmentation tasks where class imbalance exists. These losses helped improve segmentation accuracy by enforcing higher overlap between predictions and ground truth masks.

5.3 Hyperparameter Choice

Hyperparameters were found using a grid-search implemented from scratch. Due to the high training time of the U-Net architecture, this search was conducted with a random subset of the dataset (20%) and with images resized to $(48, 48)$. Limited testing with our full size model confirms this approach, showing that performance on the smaller model mirrors that of the larger one.

6 Results and Discussion

In this section we will discuss our findings. Specifically, we will discuss our Fast U-Net, where we utilize depthwise separable convolutions and bilinear upsampling, against a standard U-Net(normal convolutions and transposed convolutions).

6.1 Comparison between Standard U-Net and Fast U-Net

Inspecting the segmentation masks (shown in Figure 10) visually, the standard U-Net has more accurate outlines of the objects with better defined boundaries. On the other hand, the Fast U-Net, although it is able to capture the general region of the object, has some scattered artifacts. These artifacts can be attributed to the lightweight design of the model.

6.2 Model Size and Parameter Reduction

The use of depthwise separable convolutions results to a significant decrease in the number of model parameters. The standard U-Net has about 31 million parameters while the Fast U-Net has about 3.9 million parameters. The reduction of more than 27 million parameters is made with a fairly small penalty in terms of segmentation accuracy, which makes it easier to set the hyperparameters and conduct two fold cross validation grid search on 4070 GPU.

6.3 Grid Search Performance

The Fast U-Net architecture was extensively optimized through a systematic search over several important hyperparameters, including the learning rate, weight decay, batch size, loss function, and the number of training epochs. Out of the 48 experiments conducted, the highest IoU value of around 0.598 was achieved, and the median IoU value was 0.538, as shown in Table 4. No single hyperparameter was found to be the best among all others in terms of affecting the performance since most of the configurations produced IoU values in the range of 0.50–0.60.

Studying the times of training runs shows that decreasing the batch size and the number of training iterations leads to a significant decrease in the training time, as shown in Table 5. Furthermore, the use of Dice loss also led to reduced computational complexity. Surprisingly, the IoU metrics did not worsen for these 'faster' training settings, which were quite similar to the initial ones. This result indicates that for large datasets or for computation-resource-limited settings, the strategy of optimizing the training time by decreasing the batch size and the number of iterations may be a reasonable compromise that does not necessarily result in a deterioration of the segmentation quality.

6.4 The Performance of the Optimized Fast U-Net

When the original Fast U-Net was compared to the best tuned version of the network (shown in Figure 9), it was noticed that there was an improvement in the definition of the boundaries and the reduction of the background noise. Although the changes are not significant, the optimized Fast U-Net generated more accurate object contours than the original network as evidenced by the higher IoU values obtained during the grid search.

6.5 ROC Analysis and Some Class Level Findings

The ROC curves for each class of pixels (background, boundary, and main body) are presented in Figure 8. The ROC curves are fairly close to the diagonal and the AUC values are between 0.48 and 0.52. This near random separability on the pixel level is especially evident for thin boundary regions which are hard to classify due to the lack of sufficient number of training samples. In

addition, the prevalence of the background class increases the problem of class imbalance. These observations suggest that future work may benefit from more specific approaches, such as class balancing or the use of region-based loss functions.

7 Conclusion

In summary, we have developed a modified U-Net model for pet segmentation, which is accurate and efficient. We also achieved a significant reduction in model parameters by integrating depthwise separable convolutions and bilinear upsampling while maintaining a reasonable segmentation quality. Further, extensive hyperparameter tuning validated the robustness of our approach. These results further confirm the efficacy of efficient CNN-based architectures for fine-grained segmentation tasks in resource-constrained environments, thus opening new directions for future research and applications. Our code is public on GitHub at `https://github.com/classjek/math156`.

8 Author Contributions

Table 2: Author Contributions

Author	**Contribution**
Peilin Rao	Development of U-Net architecture; implement depthwise Separable convolution and bilinear upsampling; construct model wrapper for training. Write section 4 of report
Jake Ekoniak	Development of preprocessing pipeline and main U-Net architecture. Create smaller U-Net architecture so other group members can conduct hyperparameter search in reasonable time. Write abstract and sections 2 and 3 of report.
Ananya Sampat	Develop and implement image transformations. Write portion of section 5.
Dylan Shah	Develop, test, and implement various loss functions. Write portion of section 5.
Nalin Chopra	Conduct hyperparameter search. Evaluation of model and results. Orchestrate blog post.

9 Acknowledgments

We would like to express our sincere gratitude to Professor Tingwei Meng. We also extend our thanks to the creators and maintainers of the Oxford-IIIT Pet Dataset, whose work provided the essential data for our research.

10 Component List

Jake tried to implement attention[12]. He likely made some mistakes with the implementation as it resulted in little change to the accuracy, so it was not used in the final model. However, this may be due to the fact that a non-sequence-based model simply doesn't always need attention. Besides this, all components were implemented.

References

[1] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition, 1998.

[2] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks, 2013.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks, 2012.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[5] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020.

[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database, 2009.

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

[8] François Chollet. Xception: Deep learning with depthwise separable convolutions. *arXiv preprint*, arXiv:1610.02357, 2016. Accessed 15 Mar. 2025.

[9] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2017.

[10] Crayon Consulting. An introduction to separable convolutions with literature review, 2020. Accessed 15 Mar. 2025.

[11] Atul Kang. Image processing – bilinear interpolation, 2018. Accessed 15 Mar. 2025.

[12] Ozan Oktay, Jo Schlemper, Loic Le Folgoc, Matthew Lee, Mattias Heinrich, Kazunari Misawa, Kensaku Mori, Steven McDonagh, Nils Y Hammerla, Bernhard Kainz, Ben Glocker, and Daniel Rueckert. Attention u-net: Learning where to look for the pancreas, 2018.

A Model Architecture detail

Table 3: Architecture Detail of the FastUnet Model

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 3, 128, 128]	27
Conv2d-2	[-1, 64, 128, 128]	192
BatchNorm2d-3	[-1, 64, 128, 128]	128
ReLU-4	[-1, 64, 128, 128]	0
Conv2d-5	[-1, 64, 128, 128]	576
Conv2d-6	[-1, 64, 128, 128]	4096
BatchNorm2d-7	[-1, 64, 128, 128]	128
ReLU-8	[-1, 64, 128, 128]	0
FastDoubleConv-9	[-1, 64, 128, 128]	0
MaxPool2d-10	[-1, 64, 64, 64]	0
Conv2d-11	[-1, 64, 64, 64]	576
Conv2d-12	[-1, 128, 64, 64]	8192
BatchNorm2d-13	[-1, 128, 64, 64]	256
ReLU-14	[-1, 128, 64, 64]	0
Conv2d-15	[-1, 128, 64, 64]	1152

Layer (type)	Output Shape	Param #
Conv2d-16	[-1, 128, 64, 64]	16384
BatchNorm2d-17	[-1, 128, 64, 64]	256
ReLU-18	[-1, 128, 64, 64]	0
FastDoubleConv-19	[-1, 128, 64, 64]	0
MaxPool2d-20	[-1, 128, 32, 32]	0
Conv2d-21	[-1, 128, 32, 32]	1152
Conv2d-22	[-1, 256, 32, 32]	32768
BatchNorm2d-23	[-1, 256, 32, 32]	512
ReLU-24	[-1, 256, 32, 32]	0
Conv2d-25	[-1, 256, 32, 32]	2304
Conv2d-26	[-1, 256, 32, 32]	65536
BatchNorm2d-27	[-1, 256, 32, 32]	512
ReLU-28	[-1, 256, 32, 32]	0
FastDoubleConv-29	[-1, 256, 32, 32]	0
MaxPool2d-30	[-1, 256, 16, 16]	0
Conv2d-31	[-1, 256, 16, 16]	2304
Conv2d-32	[-1, 512, 16, 16]	131072
BatchNorm2d-33	[-1, 512, 16, 16]	1024
ReLU-34	[-1, 512, 16, 16]	0
Conv2d-35	[-1, 512, 16, 16]	4608
Conv2d-36	[-1, 512, 16, 16]	262144
BatchNorm2d-37	[-1, 512, 16, 16]	1024
ReLU-38	[-1, 512, 16, 16]	0
FastDoubleConv-39	[-1, 512, 16, 16]	0
MaxPool2d-40	[-1, 512, 8, 8]	0
Conv2d-41	[-1, 512, 8, 8]	4608
Conv2d-42	[-1, 1024, 8, 8]	524288
BatchNorm2d-43	[-1, 1024, 8, 8]	2048
ReLU-44	[-1, 1024, 8, 8]	0
Conv2d-45	[-1, 1024, 8, 8]	9216
Conv2d-46	[-1, 1024, 8, 8]	1048576
BatchNorm2d-47	[-1, 1024, 8, 8]	2048
ReLU-48	[-1, 1024, 8, 8]	0
FastDoubleConv-49	[-1, 1024, 8, 8]	0
Conv2d-50	[-1, 512, 16, 16]	524288
Conv2d-51	[-1, 1024, 16, 16]	9216
Conv2d-52	[-1, 512, 16, 16]	524288
BatchNorm2d-53	[-1, 512, 16, 16]	1024
ReLU-54	[-1, 512, 16, 16]	0
Conv2d-55	[-1, 512, 16, 16]	4608
Conv2d-56	[-1, 512, 16, 16]	262144
BatchNorm2d-57	[-1, 512, 16, 16]	1024
ReLU-58	[-1, 512, 16, 16]	0
FastDoubleConv-59	[-1, 512, 16, 16]	0
FastDecoderBlock-60	[-1, 512, 16, 16]	0
Conv2d-61	[-1, 256, 32, 32]	131072
Conv2d-62	[-1, 512, 32, 32]	4608
Conv2d-63	[-1, 256, 32, 32]	131072
BatchNorm2d-64	[-1, 256, 32, 32]	512
ReLU-65	[-1, 256, 32, 32]	0

Layer (type)	Output Shape	Param #
Conv2d-66	[-1, 256, 32, 32]	2304
Conv2d-67	[-1, 256, 32, 32]	65536
BatchNorm2d-68	[-1, 256, 32, 32]	512
ReLU-69	[-1, 256, 32, 32]	0
FastDoubleConv-70	[-1, 256, 32, 32]	0
FastDecoderBlock-71	[-1, 256, 32, 32]	0
Conv2d-72	[-1, 128, 64, 64]	32768
Conv2d-73	[-1, 256, 64, 64]	2304
Conv2d-74	[-1, 128, 64, 64]	32768
BatchNorm2d-75	[-1, 128, 64, 64]	256
ReLU-76	[-1, 128, 64, 64]	0
Conv2d-77	[-1, 128, 64, 64]	1152
Conv2d-78	[-1, 128, 64, 64]	16384
BatchNorm2d-79	[-1, 128, 64, 64]	256
ReLU-80	[-1, 128, 64, 64]	0
FastDoubleConv-81	[-1, 128, 64, 64]	0
FastDecoderBlock-82	[-1, 128, 64, 64]	0
Conv2d-83	[-1, 64, 128, 128]	8192
Conv2d-84	[-1, 128, 128, 128]	1152
Conv2d-85	[-1, 64, 128, 128]	8192
BatchNorm2d-86	[-1, 64, 128, 128]	128
ReLU-87	[-1, 64, 128, 128]	0
Conv2d-88	[-1, 64, 128, 128]	576
Conv2d-89	[-1, 64, 128, 128]	4096
BatchNorm2d-90	[-1, 64, 128, 128]	128
ReLU-91	[-1, 64, 128, 128]	0
FastDoubleConv-92	[-1, 64, 128, 128]	0
FastDecoderBlock-93	[-1, 64, 128, 128]	0
Conv2d-94	[-1, 3, 128, 128]	195

B Plots and tables for training results

Table 4: Parameter Optimization Results

batch_size	criterion	lr	num_epochs	optimizer_name	weight_decay	score (IoU)	score_rank
32	dice	0.0100	100	adam	0.0000	0.5980	1.0
32	cross_entropy	0.0100	100	adam	0.0001	0.5950	2.0
32	cross_entropy	0.0100	100	adam	0.0000	0.5940	3.0
32	dice	0.0100	50	adam	0.0000	0.5825	4.0
32	dice	0.0100	100	adam	0.0001	0.5810	5.0
32	cross_entropy	0.0010	100	adam	0.0000	0.5750	6.0
32	cross_entropy	0.0100	50	adam	0.0000	0.5745	7.0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
64	cross_entropy	0.0100	50	adam	0.0100	0.2095	48.0

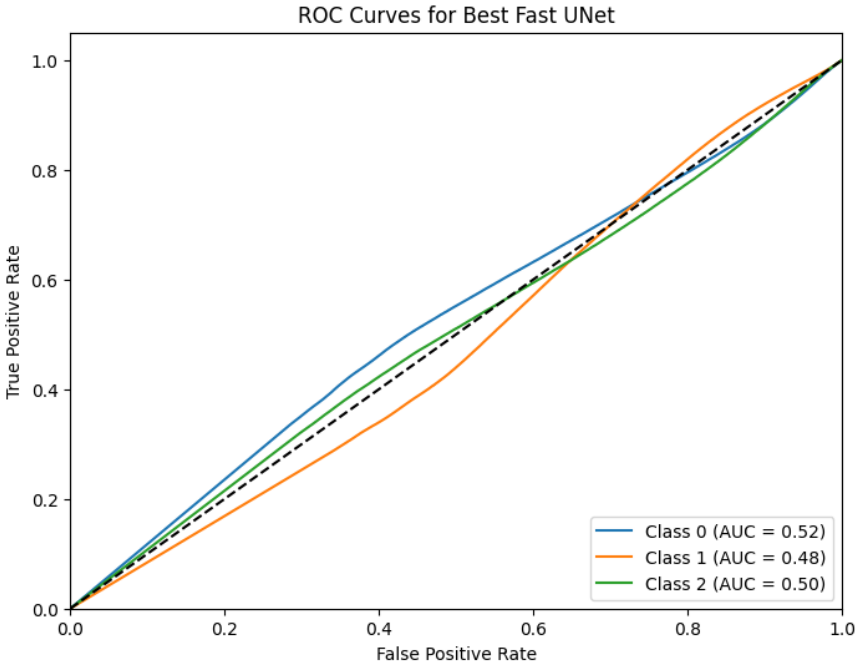


Figure 8: Receiver Operating Characteristic (ROC) curves of the models.

Table 5: Total Time Comparison for Different Parameters

batch_size	criterion	lr	num_epochs	optimizer_name	weight_decay	total_time	time_rank
32	cross_entropy	0.0001	50	adam	0.0000	1.20	1.0
32	dice	0.0010	50	adam	0.0100	1.20	1.0
32	dice	0.0001	50	adam	0.0100	1.20	1.0
32	dice	0.0001	50	adam	0.0001	1.20	1.0
32	dice	0.0001	50	adam	0.0000	1.20	1.0
32	cross_entropy	0.0100	50	adam	0.0100	1.20	1.0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
64	cross_entropy	0.0100	100	adam	0.0000	13.55	47.0

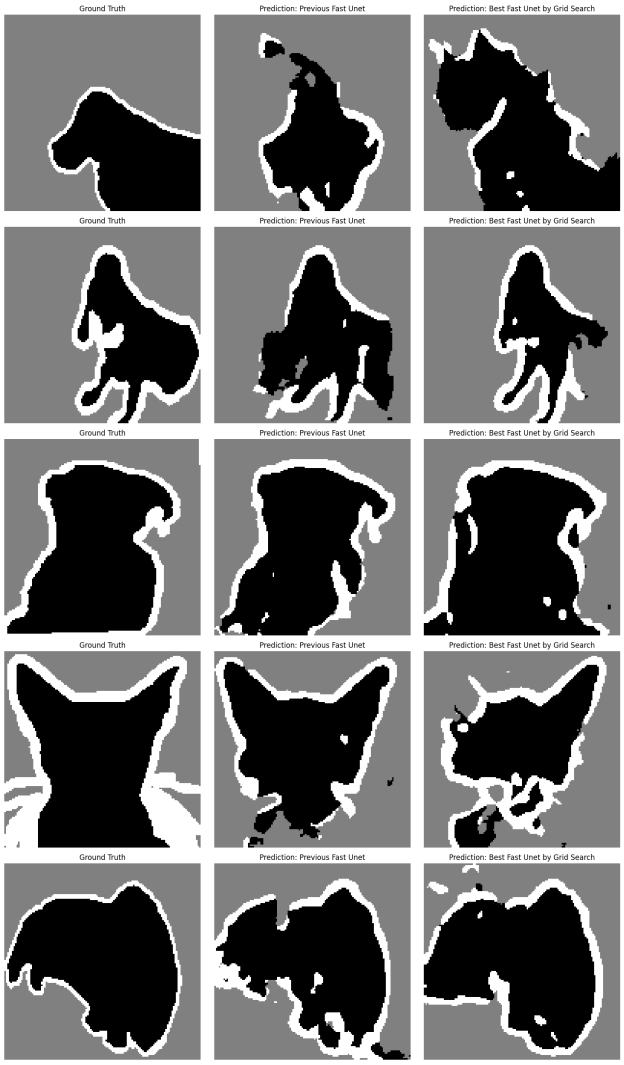


Figure 9: Optimized results of Fast U-Net.

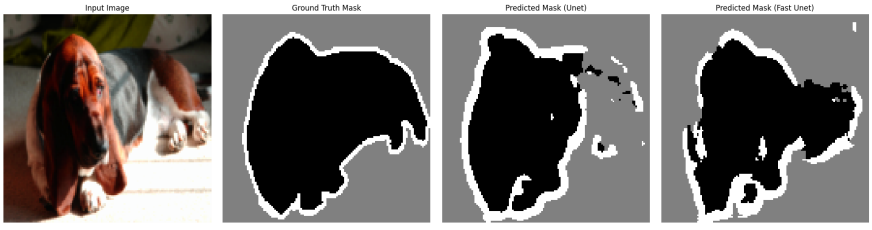


Figure 10: Comparison of U-Net and Fast U-Net performance.