Using Superpixelation to improve image segmentation with the UNet architecture

JACK ROBERTS

Compiled May 19, 2024

The goal of this independent study is to understand the impact of using superpixels on image segmentation tasks in deep learning. Image segmentation is the process of partitioning an image into distinct and meaningful regions. Superpixels are a group of pixels that share common characteristics such as color similarity and proximity. This paper will test the hypothesis that providing the U-Net with not only the raw image but also some additional context provided by an encoding of its superpixelation will help improve the U-Net's accuracy in segmenting the image.

http://dx.doi.org/10.1364/ao.XX.XXXXXX

1. INTRODUCTION

A. Image Segmentation

In the realm of computer vision and image processing, the ability to comprehend and extract meaningful information from visual data is central.

At its core, image segmentation can be understood as the partitioning of an image into subgroups of pixels called regions or segments, each representing a coherent and semantically meaningful entity. This helps to reduce the complexity of an image making analysis easier [2].

Figure 1 shows an example segmentation of a tiger and its background.

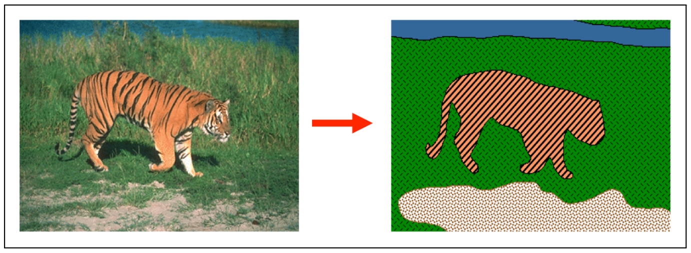


Fig. 1. Image Segmentation Example

The importance of image segmentation spans across various domains, including but not limited to medical imaging, autonomous driving, surveillance, remote sensing, and industrial inspection. In medicine, for instance, precise segmentation of anatomical structures from medical scans facilitates accurate diagnosis, treatment planning, and surgical guidance. Similarly, in autonomous driving systems, segmenting objects in the vehicle's surroundings enables real-time analysis of the environment, paving the way for safer navigation and collision avoidance.

The architecture we will be utilizing in this paper is the U-Net, a convolutional network proposed for biomedical image segmentation.

B. The Convolution Operation

A central component of the U-Net architecture is the convolution operation. It is central for its ability to extract hierarchical features from input images, enabling the network to capture spatial relationships and patterns crucial for accurate segmentation.

The convolution between a matrix A and a kernel B is another matrix C, which is denoted as follows:

$$(A \otimes B)_{i,j} = C_{i,j} = \sum_{k=0}^{p-1} \sum_{l=0}^{q-1} B_{k,l} \cdot B_{i-k,j-l} \tag{1}$$

The operation essentially involves "scanning" matrix A with kernel B, multiplying all the elements of the window in A with B and then summing the results up.

The resulting C matrix of a convolution operation has a size that depends on A and B. We can control this final size in two ways:

- **Convolutional stride** refers to the step you take when moving from one window to the next when "scanning".
- **Padding** refers to the addition of a frame of zero valued entries around matrix A. The implementation of the U-Net in this paper uses padding of 1. A stride of 2 is used for the transpose convolution (up-conv) operations (see section D).

Figure 2 shows a convolution as a neural network layer.

There you can see how, no matter the size of the matrix A, the number of weights to learn is always the kernel's width * kernel's height + 1 (for the bias weight). As the kernel "scans" across matrix A, the same ten weights are updated depending on which overlap with A. This opens up the possibility of learning many filters without a big computational burden [3].

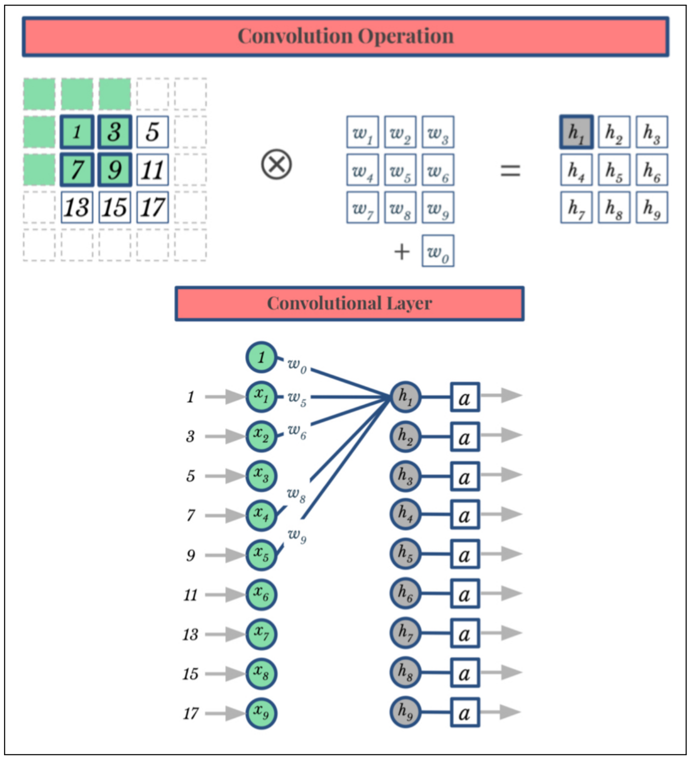


Fig. 2. Convolution as a Neural Network Layer

C. Max-pooling Operation

Another important operation and layer in Deep Learning is the Pooling layer. The idea is similar to the convolution operation in that the initial A matrix is swept by a kernel but instead, pooling is a non-learnable operation. Max-pooling picks the maximum of each window rather than learning a set of weights. Max-pooling is an important part of the encoding process of the U-Net since it reduces the dimensions of a tensor while preserving important information [3].

Figure 3 shows an example of 2x2 Max-pooling with stride 2.

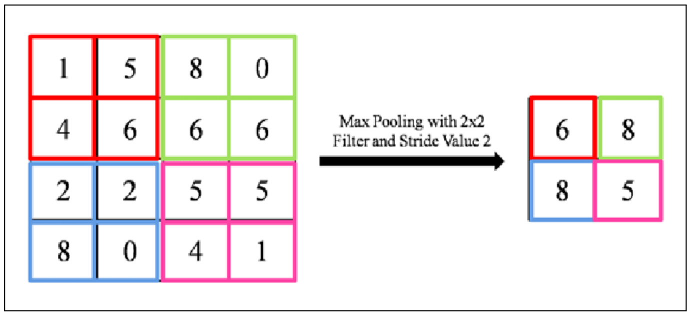


Fig. 3. 2x2 Max-pooling with stride 2

D. Transpose Convolution Operation

Finally, the transpose convolution operation is another key feature of the U-Net architecture. It is typically used to increase the size (length and width) of a tensor and so features in the decoder path of the U-Net. The transpose convolution between two matrices A and B is denoted as follows [2]:

$$A \otimes^T B$$

It is computed as follows:

- Multiplying each element of A by the whole of matrix B (the kernel here).
- Placing the resulting matrix in a new matrix according with a predefined stride, summing entries whenever there is an overlap.

An example transpose convolution operation can be seen in figure 4.

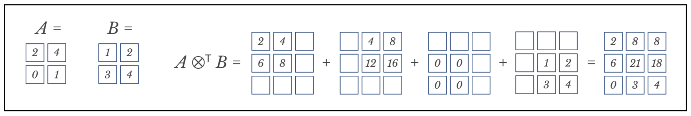


Fig. 4. Transpose Convolution Operation

E. U-Net

The U-Net architecture is distinguished by its U-shaped design, comprising an encoder path followed by a symmetric decoder path connected at each layer by skip connections. At the heart of U-Net are convolutional layers responsible for feature extraction and spatial information preservation [6].

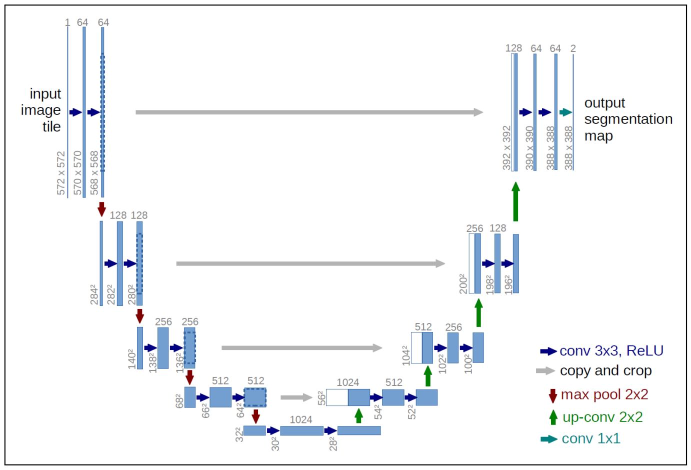


Fig. 5. Image Segmentation Example

Figure 5 shows the U-Net architecture. Each strip is a tensor and the arrows represent tensor operations. The numbers on top of the tensors represent the number of channels and the numbers to the side are the height and width. Note the paper feeds an RGB image (3 channels) as input; however, we will be experimenting with an input tensor with more channels containing information from the image's superpixelation.

The architecture outputs a tensor with K channels where K is the number of segment classes.

The encoder path captures high-level features of the input image through successive downsampling operations, using max-pooling layers. This process progressively reduces the spatial dimensions of the input image while increasing the number of channels, "compressesing" the visual information into its most important features.

Conversely, the decoder path, gradually upsamples the tensors. This step is achieved through transpose convolutions or upsampling layers, which restore the spatial resolution while integrating contextual information from the encoder path provided by the skip connections.

Furthermore, skip connections establish direct connections between corresponding encoder and decoder layers. These connections mitigate the issue of vanishing gradients and add extra information to the decoder that might be lost during down or up sampling [2].

F. Superpixelation

Superpixelation is a technique in image processing where pixels with similar characteristics are grouped together to form larger, more coherent units called superpixels. These superpixels serve as a more abstract representation of the image, simplifying it while retaining essential details. By organizing pixels into these cohesive clusters, superpixelation facilitates more efficient and effective image analysis and processing tasks compared to working with individual pixels. Essentially, it's a way to create a more manageable and structured view of an image by grouping similar pixels together as seen in figure 6.

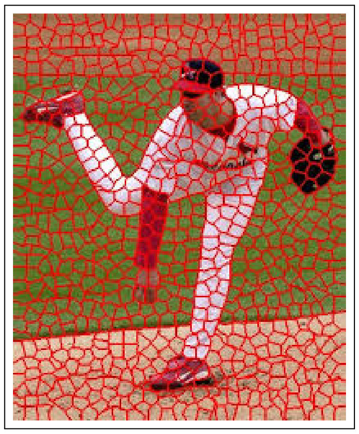


Fig. 6. Superpixelation Example

Simple Linear Iterative Clustering (SLIC) is the algorithm used in this paper for superpixel generation [1]. SLIC clusters pixels based on their color similarity and proximity in the image plane. It accomplishes this by optimizing a distance metric in a 5-dimensional space comprising of color space and spatial space. The 5 dimensional space is labxy space, where lab is the pixel color vector in CIELAB color space and xy is the pixel position. This new 5D distance measure works as follows:

$$
\begin{aligned}
C_k &= [R_k, G_k, B_k, x_k, y_k]^T \\
d_{RGB} &= \sqrt{(R_j - R_k)^2 + (G_j - G_k)^2 + (B_j - B_k)^2} \\
d_{xy} &= \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2} \\
D_s &= d_{RGB} + \frac{m}{S} d_{xy}
\end{aligned}
$$

Where $m \in [1, 20]$ is a variable that allows us to control the compactness of a superpixel (higher values of m correspond to a more compact superpixel).

$$S = \sqrt{\frac{\text{number of superpixels in input image}}{\text{number of superpixels used to segment input image}}}$$

The SLIC algorithm can be seen in algorithm 1.

Algorithm 1. SLIC algorithm

```
1: Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]$ by sampling pixels at regular grid steps S.
2: Perturb cluster centers in an n x n neighbourhood, to the lowest gradient position.
3: Repeat
4: **for** each cluster $C_k$ **do**
5:     Assign the best matching pixels from a 2S x 2S square neighbourhood around the cluster center according to the distance measure
6: Compute new cluster centers and residual error E L1 distance between previous centers and recomputed centers
7: until E $\leq$ threshold
8: Enforce connectivity
```

2. METHODOLOGY

A. Using Superpixels to Improve Segmentation

As discussed, superpixels are clusters of pixels grouped by similarities in their position and color. The idea behind this independent study is to use superpixels to help improve the accuracy of image segmentation.

Figure 7 shows how the superpixel borders match closely with the border of the image segmentation. The idea is that by providing the U-Net with information that closely approximates the image's borders, we can drive the segmentation process to a more accurate solution.

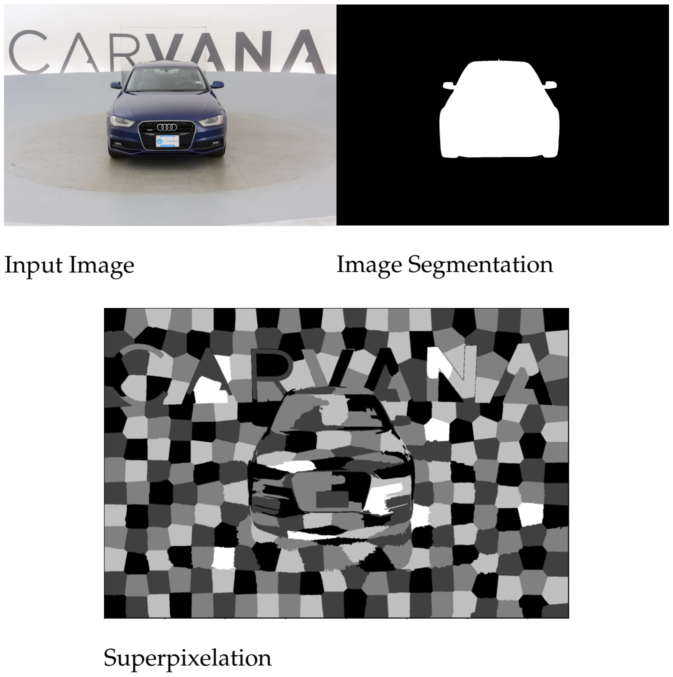


Fig. 7. Superpixel borders compared to segmentation border

This paper will endeavour to see if feeding the input image into the U-Net along with an encoding of its superpixelation will improve the accuracy of image segmentation. To encode the superpixelation we will use one-hot encoding.

B. One Hot Encoding

One-hot encoding is a method used to convert categorical variables into a numerical format suitable for machine learning algorithms. Each category within a categorical variable is represented by a binary vector, where only one element is active (set to 1), indicating the presence of that category, while all other elements are inactive (set to 0) [4].

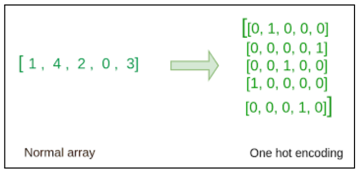


Fig. 8. One-hot encoding

However, it's important to note that one-hot encoding may lead to high-dimensional data, especially with variables having a large number of categories, which can impact computational efficiency and model performance.

The risk of high-dimensional data is very relevant when trying to one-hot encode a superpixelation because there are hundreds of segments for each image. To work around this issue we can take inspiration from the graph coloring problem.

C. The Graph Coloring Problem

The graph coloring problem is the task of coloring the vertices of a graph in such a way that no two adjacent vertices have the same color. See figure 9 for an example of this.

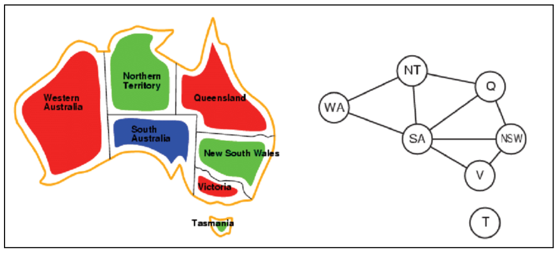


Fig. 9. Graph Coloring Example

We can apply the graph coloring problem to a superpixelation such that each superpixel is assigned a "color" that it does not share with its neighbors. This will still capture the boundary relationship between superpixels, crucial to informing the image segmentation, without requiring hundreds of channels when one-hot encoded. The algorithm used in this paper to compute the color map is greedy so does not always converge to the optimal solution of four colors [5].

The process first begins by computing the adjacency matrix for the superpixelation where *segments_slic* is a tensor representing the superpixels:

```
def create_adj_matrix(segments_slic, kernel_size, n_sp,
    ↪ width, length):
    G = np.zeros((n_sp, n_sp)) # represents neighbouring
    ↪ relationship between superpixels
    for seg in np.unique(segments_slic):
        mask = segments_slic == seg

        xy = np.where(mask)
        max_x, min_x = np.max(xy[0]), np.min(xy[0])
        max_y, min_y = np.max(xy[1]), np.min(xy[1])
        min_x = max(0, min_x - kernel_size)
        min_y = max(0, min_y - kernel_size)
        max_x = min(width, max_x + kernel_size)
        max_y = min(length, max_y + kernel_size)
        region_of_interest = mask[min_x:max_x, min_y:max_y
    ↪ ]

        dilated = ndimage.binary_dilation(
    ↪ region_of_interest)
        diff = dilated - region_of_interest.astype(int)
        neig = np.unique(segments_slic[min_x:max_x, min_y:
    ↪ max_y][diff != 0])
        G[seg, neig] = 1
    return G
```

From this function we now have a matrix G containing information about which superpixels connect to eachother.

From there we can pass the adjacency matrix into the following function to generate a dictionary assigning each superpixel segment to a "color". The algorithm works as follows:

1. Represent the graph with an adjacency matrix.
2. Count the degree of each node and define the possible colors.
3. Sort the nodes from largest to smallest degree.
4. Starting from the node with largest degree, assign it a color available in its color set.
5. Remove that color from the color set of adjacent nodes.
6. Repeat until all nodes assigned a color.

```
def generate_colored_G(G):
    # count degree of all node.
    degree =[]
    for i in range(len(G)):
        degree.append(sum(G[i]))

    # instantiate the possible color
    colorDict = {}
    for i in range(len(G)):
        colorDict[i]=[0,1,2,3,4,5]


    # sort the node depends on the degree
    sortedNode=[]
    indeks = []

    # use selection sort
    for i in range(len(degree)):
        _max = 0
        j = 0
        for j in range(len(degree)):
            if j not in indeks:
                if degree[j] > _max:
                    _max = degree[j]
                    idx = j
        indeks.append(idx)
        sortedNode.append(idx)

    # The main process
    solution={}
    for n in sortedNode: # starting from the node of
    ↪ highest degree
        # setTheColor = list of available colors
        setTheColor = colorDict[n]
        # assign the color for current node to be the
    ↪ first available color (GREEDY)
        solution[n] = setTheColor[0]
        adjacentNode = G[n]
        for j in range(len(adjacentNode)):
            # if its a neighbour and it currently has the
    ↪ color just used by node n
            if adjacentNode[j]==1 and (setTheColor[0] in
    ↪ colorDict[j]):
                # remove that color from the list so that
    ↪ we don't end up having some colored neighbours
                colorDict[j].remove(setTheColor[0])
    return solution
```

With a new superpixel representation where each segment is assigned a "color" such that it does not share that "color" with any neighbours we can now use one hot encoding to create a tensor of shape [C,W,H] where C, the number of channels, is the number of distinct colors used to fill the map, and W,H are the dimensions of the image.

```
segments_slic_1d = segments_slic.flatten()
# solution is the dictionary mapping each superpixel
    ↪ segment to a "color"
color_pix = np.array([solution[segment] for segment in
    ↪ segments_slic_1d])
color_pix = color_pix.reshape(segments_slic.shape)
out = F.one_hot(torch.tensor(color_pix)).permute(2, 0, 1)
```

The resulting tensor has a channel for each "color" used and has 1's where that particular color appears in the superpixelation and 0's elsewhere. Before we can stack these channels onto our input image and train the U-Net, the greedy nature of our coloring algorithm has to be addressed. Since it is greedy it does not produce optimal results and so some superpixelations use 5 colors and others use 6. The U-Net needs to know the dimensions of its input so, in order to ensure a consistent number of channels, an extra channel containing all zeros was added to the tensors with only 5 channels to ensure all superpixelation tensors shared the same dimensions.

D. The dataset

For this project, the U-Net was trained and tested on the Carvana dataset. This dataset contains a large number of car images. Each car has exactly 16 images, each one taken at different angles. The training set contains a manually cutout mask for each image. For training, we used data augmentation so the images underwent some transformations to improve the model's accuracy. Training images were resized to be 160x240, rotated, flipped, and normalised to compensate for variations between samples.

Some examples of the types of cars in the dataset can be see in figure 10.

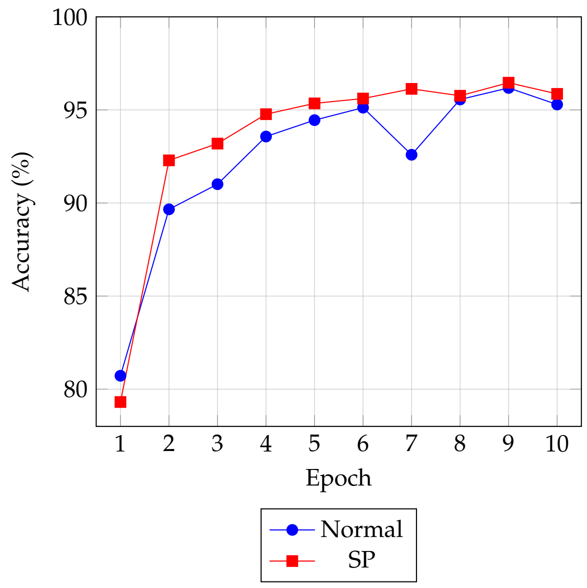
E. U-Net implementation

In order to leave some room for improvement, the U-Net implementation used in this paper was altered to reduce its accuracy in image segmentation. This was done by reducing the number of "layers" in the architecture by having only two max pooling operations in the encoding portion and then a corresponding two convolutional transpose layers in the decoding portion. The number of channels per layer was also scaled down with the initial input tensor being converted to 16 channels rather than the 64 stated in the original paper. This scaling was then consistent with the paper going from 16 to 32 to 64 and so on. This once again served to cap the model's performance and allow the benefit of superpixels to be apparent if existent.

3. RESULTS

During training, after each epoch, the accuracy of the model on the test dataset is calculated. Figure 11 shows this progression across 10 epochs:

Fig. 10. Example cars from the Carvana data set



Epoch	Normal	SP
1	80.8	79.4
2	89.7	92.3
3	91.0	93.2
4	93.6	94.8
5	94.5	95.4
6	95.1	95.6
7	92.6	96.1
8	95.6	95.8
9	96.2	96.5
10	95.3	95.9

Fig. 11. Comparison of testing accuracy between Normal and SP methods over 10 training epochs.

Figures 12, 13, and 14 show some segmentations using superpixels and not to help visualise the effect:

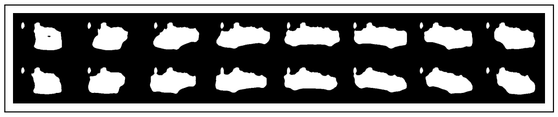


Fig. 12. Predicted segmentation (normal image input)

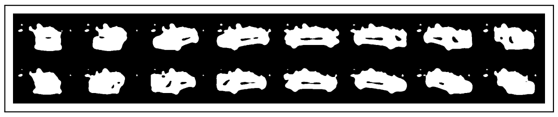


Fig. 13. Predicted segmentation (image input with superpixels)

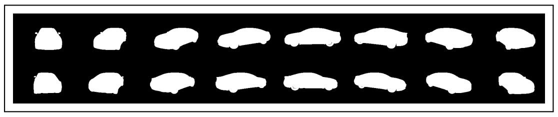


Fig. 14. True segmentation

In testing, the average losses for the standard input image were 0.114 compared to average losses of 0.106 for the images inputted with their superpixelation encoding.

The results demonstrate an improvement in segmentation accuracy when utilizing superpixel information. The addition of superpixel channels provided the U-Net model with valuable boundary and spatial information, leading to more precise segmentation outputs. The time trade-off must however be noted. In this implementation, the superpixel encoding was computed online due to memory constraints on the Bowdoin HPC and tensor (.pt) files taking up far more space than a regular image (.jpg). This resulted in far slower segmentation with each regular epoch taking roughly 2 minutes and 15 seconds compared to 5 hours and 19 minutes for each epoch where the images are adjusted online. The solution here however is very simple, the superpixel tensors should be computed offline and uploaded in their final form before running the model to avoid unnecessary online computation. This would allow the U-Net to run as quickly as normal but with the improved accuracy provided with the superpixels.

4. CODE

The code for this project can be found at the following link: https://github.com/JackRobs25/IS.

5. EXTENSION

To take this project further it would be interesting to feed the superpixel encoding into each layer of the encoding process of the U-Net as opposed to just the original input. By passing this extra information in at each step of the encoding process the model would be reminded of its target more frequently. The U-Net's skip connections would also feed this valuable information to each step of the decoding path as well. It would be interesting to see if this would improve the accuracy of the U-Net further.

REFERENCES

1. Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2274–2282, 2012.
2. Jeova Farias. Lec 13: Intro to image segmentation. University Lecture, 2023.
3. Jeova Farias. Lec6: Convolutional neural networks. University Lecture, 2023.
4. GeeksforGeeks. Machine Learning | One-Hot Encoding, 2022. Accessed: May 17, 2024.
5. Febi Mudiyanto. Solve Graph Coloring Problem with Greedy Algorithm and Python, 2022.
6. Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, 2015.