DarrenNet: Fully Convolutional Network for Semantic Segmentation

Brandon Szeto [*]
Jacobs School of Engineering
University of California, San Diego
San Diego, CA 92122
`bszeto@ucsd.edu`

Darren Yu [†]
Jacobs School of Engineering
University of California, San Diego
San Diego, CA 92122
`damiyu@ucsd.edu`

Nathaniel Thomas [‡]
Jacobs School of Engineering
University of California, San Diego
San Diego, CA 92122
`nathomas@ucsd.edu`

Abstract

This abstract provides a succinct overview of our work focused on semantic segmentation using the PASCAL VOC-2007 dataset. Our approach involves leveraging a novel deep learning architecture tailored for semantic segmentation tasks. We meticulously preprocess the dataset, addressing challenges such as class imbalance and data augmentation to enhance model generalization. Through experimentation, we explore hyperparameter tuning strategies to optimize performance. Our results showcase a notable improvement in both percent correct and Mean Intersection over Union (IoU) metrics, underscoring the efficacy of our methodology. Additionally, we uncover intriguing insights into the model's behavior, shedding light on its strengths and potential areas for refinement. This work showcases a framework for accurate and nuanced semantic segmentation.

[*] 160 lbs
[†] 130 lbs
[‡] 200 lbs

Preprint. Under review.

Introduction

Semantic segmentation is the task of assigning each pixel in an image a specified label. A single image can be partitioned into multiple meaningful segments corresponding to a given object or region of interest. Labels are assigned to individual pixels, marking boundaries and shapes. Altogether, this task has many applications in computer vision including but not limited to autonomous driving, video surveillance, and object recognition.

In recent years, convolutional neural networks (CNNs) have proven to be an effective approach to semantic segmentation. Such models learn hierarchical features by capturing information using convolutional blocks. Convolutions and related techniques such as weight initialization, batch normalization, and pooling help avoid the flat, dense layers of a traditional fully connected neural network allowing for less computation and improved feature maps.

Xavier Weight Initialization

In our architecture, we use Xavier weight initialization. In Uniform Xavier Initialization, a layer's weights are chosen from a random uniform distribution bounded between

$$\pm\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{1}$$

where n_i is the number of incoming network connections and n_{i+1} is the number of outgoing network connections.

In Normal Xavier Initialization, a layer's weights are chosen from a normal distribution with

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_i + n_{i+1}}} \tag{2}$$

where n_i is the number of incoming network connections and n_{i+1} is the number of outgoing network connections.

The Xavier initialization was created in response to the problem of vanishing and exploding gradients in deep neural networks in the context of symmetric nonlinearities (sigmoid, tanh). The intuition is that the weights should not be intialized randomly, but rather proportional to the size of two connected layers. As a result, the variance of activations and gradients would be maintained through the layers of a deep network.

Kaiming Weight Initialization

With the rise of ReLU, the nonlinearity can no longer be assumed to be symmetric. As such, the assumptions made by the Xavier Weight Initialization fall apart. In 2015, He et. al demonstrated that a ReLU layer typically has a standard deviation close to

$$\sqrt{\frac{n_i}{2}} \tag{3}$$

where n_i is the number of incoming network connections. As such, weights initially chosen from a normal distribution should be weighted by $\sqrt{\frac{n_i}{2}}$, with bias tensors initalized to zero.

Batch Normalization

During the training of a neural network, at each layer, inputs (activations) change as the parameters of the previous layers get updated. Batch normalization normalizes the activations of each layer by subtracting the mean and dividing by the standard deviation of the activations within a mini-batch. We implement batch normalization using PyTorch's `nn` module implementation. For example, we define a layer used for batch normalization as

```
self.bnd1 = nn.BatchNorm2d(32)
```

This ensures that the inputs to each layer have a consistent distribution, which helps in stabilizing the training process.

Pixel Accuracy

Pixel accuracy is a straightforward evaluation metric commonly used in image classification tasks to measure the percentage of correctly classified pixels in an image. It provides a simple measure of overall accuracy without considering class imbalance.

Pixel accuracy is calculated as:

$$\text{Pixel Accuracy} = \frac{\text{Number of Correctly Classified Pixels}}{\text{Total Number of Pixels}} \tag{4}$$

Where:

- Number of Correctly Classified Pixels: The count of pixels for which the predicted class matches the ground truth class.
- Total Number of Pixels: The total count of pixels in the image.

While pixel accuracy provides a quick measure of overall performance, it may not be the most informative metric for tasks with class imbalance or when individual classes are of interest.

Intersection over Union (IoU)

Intersection over Union (IoU), also known as the Jaccard index, is a popular evaluation metric in semantic segmentation tasks. It measures the overlap between the predicted segmentation mask and the ground truth mask for each class. IoU provides a more nuanced understanding of model performance by considering both true positives and false positives.

For a given class c, IoU is calculated as:

$$IoU_c = \frac{\text{Area of Intersection between predicted and ground truth mask for class } c}{\text{Area of Union between predicted and ground truth mask for class } c} \tag{5}$$

The overall IoU for all classes is often computed as the mean or weighted mean of individual class IoU scores:

$$IoU = \frac{1}{N}\sum_{c=1}^{N} IoU_c \tag{6}$$

Where:

- N is the total number of classes.
- IoU_c is the IoU score for class c.

IoU ranges from 0 to 1, with higher values indicating better segmentation accuracy. A value of 1 indicates perfect overlap between the predicted and ground truth masks, while a value of 0 indicates no overlap.

IoU is particularly useful for tasks where precise localization and segmentation accuracy are essential, such as medical image analysis, object detection, and scene understanding. It provides insights into how well the model captures the spatial extent of different classes within the image.

Related Works

U-Net

U-Net **?** has proven to be highly effective for biomedical image segmentation tasks. Its architecture features a contracting path to capture context and an expansive path for precise localization. The innovative use of skip connections facilitates the flow of fine-grained details across different layers, enhancing the model's ability to accurately delineate object boundaries.

ERFNet

ERFNet **?** introduces a novel factorized convolutional layer that significantly reduces the number of parameters while maintaining expressive power. This reduction in parameters enables faster inference without compromising performance, making it well-suited for deployment in resource-constrained environments.

ResNet

ResNet **?** addresses the challenges of training very deep neural networks by employing residual connections. These connections enable the direct flow of information across layers, mitigating the vanishing gradient problem and facilitating the training of extremely deep networks.

Relevance to our model

We test the performance of the above models on the image segmentation task using the PASCAL VOC-2007 dataset. We look for differences in the models and their relative performance in comparison to the basic fully connected network that we started with.

Methods

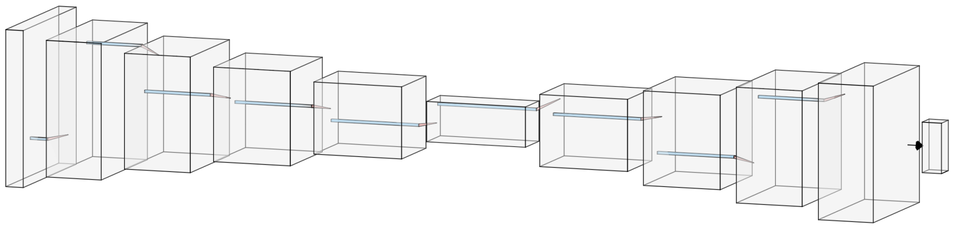

Figure 1: Visualization of System Architecture

Baseline

Our baseline architecture consists of an encoder, decoder, classifier, and activation. We used the Adam gradient descent optimizer

Encoder

We have five convolutional layers that increases the depth of the orginal 3 channels to 32 to 64 to 128 to 256 to 512 each using a size 3 kernel, padding of 1, a stride of 2, and no dilation. Each layer sees a small decrease in the height and width of the layer according to the expression $\frac{W-F+2P}{2}+1$. The outputs of each convolutional layer are subsequently passed through a ReLU activation function and a batch normalization layer.

Decoder

We have five deconvolutional, or upsampling layers that decreases the depth of the final 512 deep layer output from the encoder. This is decreases from 512 to 256 to 128 to 64 to 32 each using a size 3 kernel, padding of 1, a stride of 2, and no dilation. Each layer sees a small decrease in the height and width of the layer according to the expression $S(W-1)+F-2P$. Similarly, the outputs of each deconvolutional layer are subsequently passed through a ReLU activation function and a batch normalization layer.

Classifier and Activation

In our final layer, we have a 1×1 convolutional kernel working as a classifier, and a softmax activation layer. This layer projects a probability distribution stream over the 21 classes.

Improvements Over Baseline

Data Augmentation

To enhance the robustness of our model, we applied data augmentation techniques to our dataset. This involved performing various transformations on the input images, such as mirror flips, rotations, and crops. During the process, we must ensure that the same transformations are applied to the corresponding labels to maintain data integrity throughout the augmentation process. We found that data augmentation improved on the average Jaccard index on our models by around 0.03-0.05.

Cosine Annealing

In order to optimize the learning rate dynamically throughout the training process, we implemented the cosine annealing learning rate scheduler. This technique adjusts the learning rate in a cosine-shaped manner, effectively annealing it towards zero as training progresses. By aligning the learning rate adjustments with the number of epochs, we aim to improve the convergence and generalization capabilities of our model. Training our models with cosine annealing had a minimal improvement in both our accuracy and Jaccord index.

Class Imbalance

To mitigate the challenges posed by class imbalance, particularly addressing rare classes, we employed strategies to alleviate this issue. One approach is to apply a weighted loss criterion, which assigns higher weights to the infrequent classes during the optimization process. By doing so, we incentivize the network to pay more attention to these underrepresented classes, thus improving its ability to accurately classify them. Otherwise, the model could simply learn to label the entire image the background and still achieve decent pixel accuracy. The introduction of weight decay had a considerable improvement in our Jaccard index and was used in all our models moving on.

Experimentation

Below are the descriptions (in table format) and regularization techniques used for each architecture.

DarrenNet Architecture

Layer	In	Out	Padding	Kernel	Stride	Activation
Initial Block		16	0	3x3	1	ReLU
Downsampler 1	16	64	0	3x3	2	ReLU
Non-bottleneck 1	64	64	0	3x3	1	ReLU
Non-bottleneck 2	64	64	0	3x3	1	ReLU
Non-bottleneck 3	64	64	0	3x3	1	ReLU
Non-bottleneck 4	64	64	0	3x3	1	ReLU
Non-bottleneck 5	64	64	0	3x3	1	ReLU
Downsampler 2	64	128	0	3x3	2	ReLU
Non-bottleneck 6	128	128	0	3x3	1	ReLU
Non-bottleneck 7	128	128	0	3x3	1	ReLU
Non-bottleneck 8	128	128	0	3x3	1	ReLU
Non-bottleneck 9	128	128	0	3x3	1	ReLU
Non-bottleneck 10	128	128	0	3x3	1	ReLU
Non-bottleneck 11	128	128	0	3x3	1	ReLU
Non-bottleneck 12	128	128	0	3x3	1	ReLU
Non-bottleneck 13	128	128	0	3x3	1	ReLU
Upsampler 1	128	64	0	3x3	1	ReLU
Non-bottleneck 14	64	64	0	3x3	1	ReLU
Non-bottleneck 15	64	64	0	3x3	1	ReLU
Upsampler 2	64	16	0	3x3	1	ReLU
Non-bottleneck 16	16	16	0	3x3	1	ReLU
Non-bottleneck 17	16	16	0	3x3	1	ReLU
Output Conv	16	21	0	2x2	2	-

Table 1: DarrenNet Architecture

Layer Name	In	Out	Padding	Kernel Size	Stride	Activation
conv1	n_{channel}	n_{channel}	(1, 0)	3x1	1	ReLU
conv2	n_{channel}	n_{channel}	(0, 1)	1x3	1	ReLU
conv3	n_{channel}	n_{channel}	(1×dilated, 0)	3x1	1	ReLU
conv4	n_{channel}	n_{channel}	(0, 1×dilated)	1x3	1	ReLU
bn1						
bn2						
dropout						2D

Table 2: Non-bottleneck (DarrenNet)

Layer Name	In	Out	Padding	Kernel Size	Stride	Activation
conv	in_{channel}	$out_{\text{channel}} - in_{\text{channel}}$	1	3x3	2	
pool				2x2	2	
bn	out_{channel}	out_{channel}				

Table 3: DownsamplerBlock (DarrenNet)

Layer Name	In	Out	Padding	Kernel Size	Stride	Activation
conv	in_{channel}	out_{channel}	(1, 1)	3x3	2	
bn	out_{channel}	out_{channel}				

Table 4: UpsamplerBlock (DarrenNet)

DarrenNet Techniques

In our DarrenNet architecture, we use horizontal and vertical flipping, Kaiming weight initialization, cosine annealing, and add extra penalty to loss inversely proportional to the frequency of a given class pixel.

Transfer Learning Architecture

Layer	In	Out	Padding	Kernel	Stride	Activation
encoder						
deconv1	512	256	1	3x3	2	ReLU
deconv1	512	256	1	3x3	2	ReLU
deconv2	256	128	1	3x3	2	ReLU
deconv3	128	64	1	3x3	2	ReLU
deconv4	64	32	1	3x3	2	ReLU
conv	32	21	1	3x3	2	Softmax

Table 5: Transfered Encoder to Basic FCN Architecture

Transfer Learning Techniques

UNet Architecture

Layer	In	Out	Padding	Kernel	Stride	Activation
conv1	3	64	1	3x3	1	ReLU
conv2	64	64	1	3x3	1	ReLU
pool1	64	64	0	2x2	2	
conv3	64	128	1	3x3	1	ReLU
conv4	128	128	1	3x3	1	ReLU
pool2	128	128	0	2x2	2	
conv5	128	256	1	3x3	1	ReLU
conv6	256	256	1	3x3	1	ReLU
pool3	256	256	0	2x2	2	
conv7	256	512	1	3x3	1	ReLU
conv8	512	512	1	3x3	1	ReLU
pool4	512	512	0	2x2	2	
conv9	512	1024	1	3x3	1	ReLU
conv10	1024	1024	1	3x3	1	ReLU
deconv1	1024	512	0	2x2	2	ReLU
conv11	1024	512	1	3x3	1	ReLU
conv12	512	512	1	3x3	1	ReLU
deconv2	512	256	0	2x2	2	ReLU
conv13	512	256	1	3x3	1	ReLU
conv14	256	256	1	3x3	1	ReLU
deconv3	256	128	0	2x2	2	ReLU
conv15	256	128	1	3x3	1	ReLU
conv16	128	128	1	3x3	1	ReLU
deconv4	128	64	0	2x2	2	ReLU
conv17	64	64	1	3x3	1	ReLU
conv18	64	64	1	3x3	1	ReLU
output	64	21	0	1x1	1	ReLU

Table 6: UNet Architecture

UNet Techniques

Results

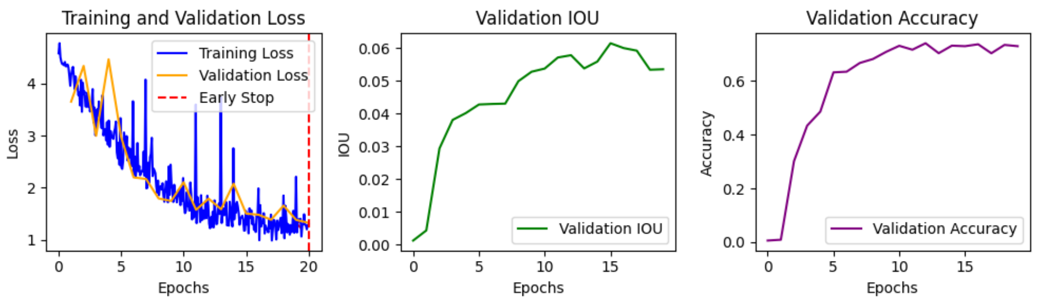

Figure 2: Baseline Augmentation

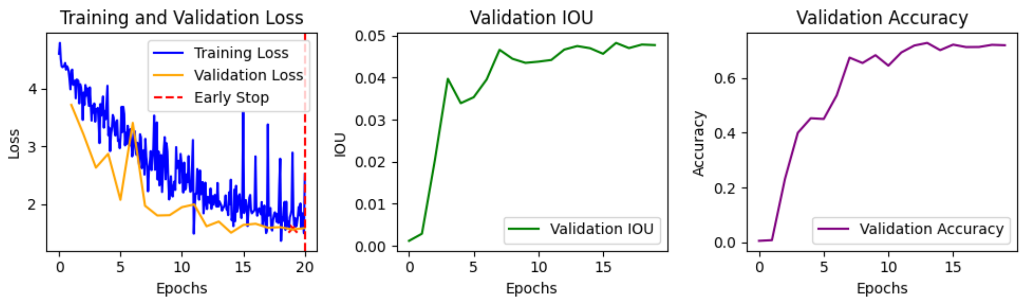

Figure 3: Baseline Augmentation

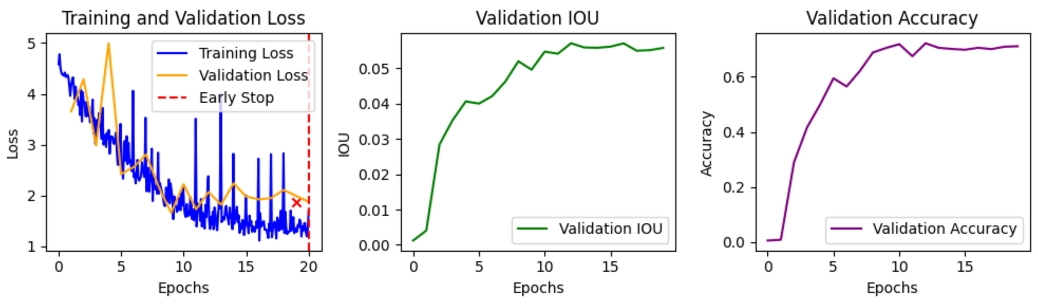

Figure 4: Baseline Cosine

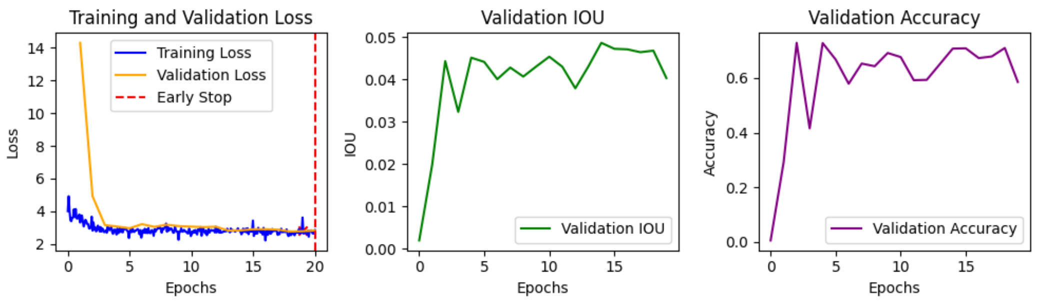

Figure 5: Baseline Weights

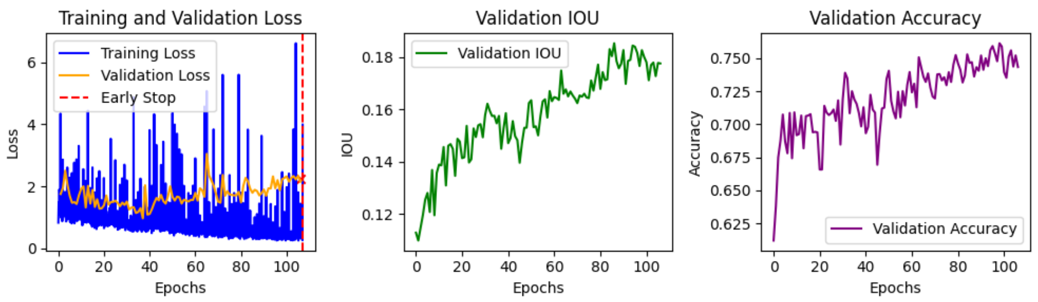

Figure 6: DarrenNet Augment Affine

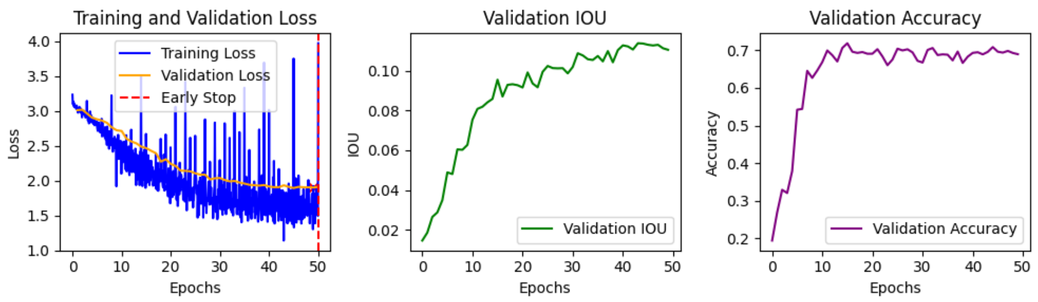

Figure 7: DarrenNet Transfer

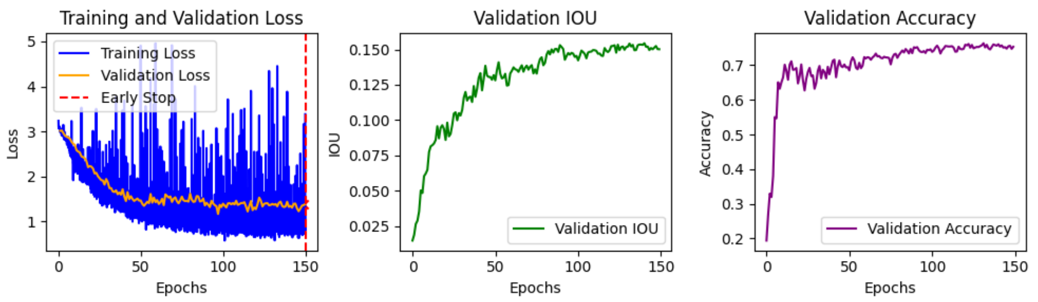

Figure 8: DarrenNet Transfer Augment

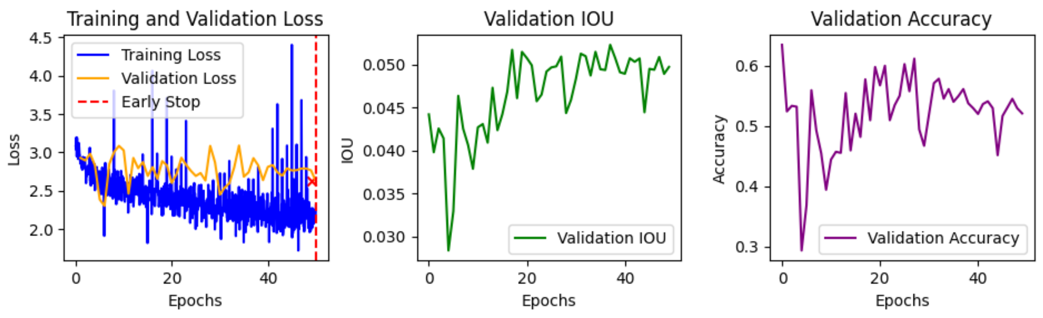

Figure 9: UNet

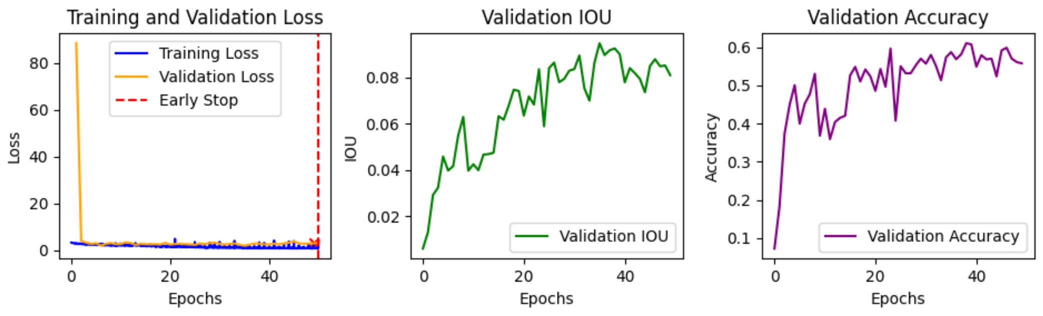

Figure 10: UNet Transfer

Discussion

Baseline: - Benefits - Drawbacks - Why we got the results we got

Improved baseline: - Benefits - Drawbacks - Why we got the results we got

DarrenNet: - Benefits - Drawbacks - Why we got the results we got

Transfer Darrennet: - Benefits - Drawbacks - Why we got the results we got

UNet: - Benefits - Drawbacks - Why we got the results we got

Performance Differences (between implementations)

Insights from loss curves, tables, visualizations

Contributions

Brandon Szeto: Basic FCN, model saving and loading, cosine annealing, average IoU and pixel accuracy, architecture research and writeup abstract, introduction, related works, methods.

Darren Yu: Group mascot, CUDA support, model inference image preview, plot training versus validation loss, writeup baseline, DarrenNet, and clean/style code.

Nathaniel Thomas: Custom CLI, Apple silicon support, dataset augmentation, class imabalance problem, transfer learning using ERFNet, ResNet, and UNet.

References