Fully Convolutional Networks for Semantic Segmentation

Joseph Warmus
University of California, San Diego
jwarmus@ucsd.edu

Eric He
University of California, San Diego
zuhe@ucsd.edu

Brydon Brancart
University of California, San Diego
bbrancar@ucsd.edu

Yigit Korkmaz
University of California, San Diego
ykorkmaz@ucsd.edu

Marcus Schaller
University of California, San Diego
mschalle@ucsd.edu

1 Abstract

Convolutional Neural Networks are extremely common tools used in computer vision. They are particularly useful for segmenting objects within an image. This paper will discuss different techniques used to accomplish this task including varying architectures and methods that can improve performance of a model. This paper uses a basic FCN which has a baseline IoU and Accuracy of 47.21% and 90.08% respectively. By applying data augmentation and class weighting this improved to 55.61% and 89.00% respectively. Transfer learning was tried which improved values to 65.85% and 95.06%. A U-net architecture was tried which achieved an IoU of 65.76% and accuracy of 90.21%. Finally, a custom architecture was tried which had an IoU of 65.00% and 92.83%.

2 Introduction

Semantic segmentation is very important in many fields. One in particular is autonomous driving. Using Fully Convolutional Networks (FCNs), the on-board systems are able to differentiate the road from the sidewalk or a person from a recycling bin. However, without good enough accuracy then the computations being done to segment the image are totally useless. This is why different techniques must be done to maximize the accuracy of the models that are generated from FCN research.

3 Related Work

Krizhevsky et al's paper on ImageNet classification was an important breakthrough in the performance of deep convolutional nets and provided a foundation for our work in this project [7]. This paper gave way to much more research in the fields of computer vision and neural networks. Long et al's 2014 paper on fully convolutional networks provided background on the topic of fully convolutional networks in semantic segmentation [8]. It showed how merging layers of different sizes can help improve semantic segmentation definition and improve detection of smaller objects. At the time many researchers were trying to improve the depth of the neural network but it was found that at some point improving depth no longer improves accuracy but rather can hurt it due to over-fitting.

He et al's 2015 paper on residual learning showed that by including residual connections within the network it can prevent this over-fitting. [5].

4 Methods

4.1 Baseline Model

4.1.1 Architecture

Since this baseline model is a fully convolutional network, with last convolutional layer having a 1x1 kernel and acting like a fully connected layer, ReLU function is selected as the activation function for the hidden layers to prevent negative values. The reason behind this is that they prevent the exponential growth in computation required for neural networks and they eliminate the vanishing gradient problem. Computational cost of adding extra ReLUs increases linearly. Also, as the last layer is convolutional, ReLU is used again before the last layer as well.

The general structure of the model is as follows. The model first uses convolutional layers to extract image features, applying batch normalization after each layer. Then, the model applies transposed convolutions to transform(upsample) the dimensions of feature maps to the dimensions(width and height) of the image while reducing the feature size every time. Again, batch normalization is applied after each layer. At the end, a final convolutional layer is used to get the output image in the right size, and extract different class activations using the features in the previous layer. In other words, last layer takes the batch normed ReLU(output) of previous layer and converts it to a WxHx10 image, where the third dimension represents activations to the each of the classes. Because of the batch normalization each of values in those 10 dimensions will be [0,1] for a pixel. Furthermore, note that, this problem is essentially a classification problem with 10 different classes. Therefore multi-class cross entropy loss is preferred for the loss. Each individual pixel is compared with their one-hot-encoded targets and then average over all pixels is taken.

During training, in order to prevent overfitting, we implemented early stopping and used adamW optimizer with weight decay. Studies show that using an adaptive optimizer with weight decay, reduces overfitting and in general they are better at generalizing. In this optimizer, weight decay does not end up in the moving averages, being only proportional to the weight. Therefore, this optimizer performs better at generalising than adam optimizer with l2 regularization [4].

4.1.2 Batch Normalization

Batch normalization is a process that is used to make neural networks faster and more stable. In this process, every layer performs normalization on the input coming from the previous layer. That way, issue of not having the output in the same scale with the normalized input will be solved up to some degree. Also, using batch norm smoothens the loss function and reduces the internal covariate shift by making input of every layer distributed around the same mean and standard deviation. The procedure for batch normalization is as follows;

For a layer with with d-dimensional input, $x = (x^{(1)}, ..., x^{(d)})$

$$\mu^{(k)} = \frac{1}{m}\sum_{i=1}^{m} x_i^{(k)} \tag{1}$$

$$\sigma^{2(k)} = \frac{1}{m}\sum_{i=1}^{m}(x_i^{(k)} - \mu^{(k)})^2 \tag{2}$$

where $k \in \{1, ..., d\}$. Using this, the input to a layer is normalized first;

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu^{(k)}}{\sqrt{\sigma^{2(k)} + \epsilon}} \tag{3}$$

ϵ is added to prevent division by zero. Then finally using this normalized $\hat{x}$;

$$y_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)} \tag{4}$$

Where γ and β are learned during the training the ensure the accurate normalization. This procedure is repeated for each batch in the set[6].

4.1.3 Weight Initialization

Weight initialization plays a critical role in preventing gradient exploding or vanishing problem. In this work, we applied Xavier initialization method, where each layer's weights are selected from a random uniform distribution bounded between;

$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} \tag{5}$$

where n_i represents the number of inputs to that layer and n_{i+1} represents the number of outputs going from that layer[2].

By using this method, the variance of activations is maintained. This constant variance helps preventing exploding and/or vanishing gradient problem[1].

4.2 Improved Baseline Model

4.2.1 Augmenting the dataset

To improve the performance of the baseline model, we applied random transformations such as horizontal flipping, vertical flipping and rotation to the training set images (and labels). Main idea in this augmentation is to reduce overfitting by showing network different examples. Keep in mind that the original sample size is not increased, instead all of these transformations are applied randomly with a fixed probability p and the network has been trained for $\frac{old\_epochs}{p}$ this time.

4.2.2 Weighted Loss

As mentioned above, multi-class cross entropy loss is used in this problem. However, evaluation of each pixel and then averaging this value for all pixels means equally learning each pixel, ignoring the frequency of different classes. This can be a problem when there is a class imbalance, i.e. one(or some) class is more dominant than the others, which is also the case in our problem. To solve this issue, we decided to use weighted cross entropy loss, which is a loss method that weights different classes' losses during computation[8]. This helps with the overfitting caused by class imbalance. The main idea in weighting methods is to give the class with fewer samples a bigger weight. In order to do this, we tried 3 different methods. The first method is called basic weighting where each weight is calculated as follows;

$$w_c = 1 - \frac{n_c}{\sum_i n_i} \tag{6}$$

Where n_i represents the number of pixels belonging to i^{th} class.

Second method used is the inverse number of samples, which can be expressed as;

$$w_c = \frac{\sum_i n_i}{n_c} \tag{7}$$

Finally, third method is the inverse square number of samples, which can be expressed as;

$$w_c = \frac{\sum_i n_i}{\sqrt{n_c}} \tag{8}$$

Note that in each of these methods, although their scales differ, classes with fewer samples get bigger weights.

4.3 Experimental model

4.3.1 Residual U-net Architecture

The experimental model designed in this paper is a merger of the Res-net and the U-net architectures. The architecture consist of 6 encoding blocks of similar dimensions to the U-Net architecture each of which include a skip connection. The decoder mimics the U-net decoder and contains bridge connections from the encoder so as to capture the finer details of each image. Additionally, each of the encoder deconvolutions include a skip connection. The following tables will describe the encoder and decoder blocks.

Table 1: The input block of the Residual Unet.

Input Block	
Network Element	**Network Element Details**
Convolution	Input: 3x384x768, Output: 32x192x384, Kernel Size: 3x3, Stride: 2, Padding: 1
Batch Normilizaton	32 features
Activation	ReLU
Convolution	Input: 32x192x384, Output 32x192x384, Kernel Size: 3x3, Stride: 1, Padding : 1
Addition	
Skip Connection	Input: 3x384x786, Output 32x192x384, Kernel Size: 3x3, Stride: 2, Padding: 1

Table 2: The general encoder block of the Residual Unet.

Encoder Block	
Network Element	**Network Element Details**
Batch Normalizaton	D features
Activation	ReLU
Convolution	Input: DxHxW, Output: 2DxH/2xW/2, Kernel Size: 3x3, Stride: 2, Padding: 1
Batch Normilizaton	2D features
Activation	ReLU
Convolution	Input: 2DxH/2xW/2, Output 2DxH/2xW/2, Kernel Size: 3x3, Stride: 1, Padding : 1
Addition	
Skip Connection	Input: DxHxW, Output: 2DxH/2xW/2, Kernel Size: 3x3, Stride: 2, Padding: 1

Table 3: The general decoder block of the Residual Unet.

Decoder Block	
Network Element	**Network Element Details**
Upsample Conv	Input: DxHxW, Output: Dx2Hx2W, Kernel Size 3x3: Stride: 2, Padding: 1
Concatenate	Concatenate Upsample and Corresponding Encoder Block
Batch Normalizaton	D + D/2 Features
Activation	ReLU
Convolution	Input: (D+D/2)xHxW, Output: D/2xHxW, Kernel Size: 3x3, Stride: 1, Padding: 1
Batch Normilizaton	D/2 Features
Activation	ReLU
Convolution	Input: D/2xHxW, Output 2DxH/2xW/2, Kernel Size: 3x3, Stride: 1, Padding : 1
Addition	Add above and below convolution
Convolution	Input: (D+D/2)xHxW, Output: D/2xHxW, Kernel Size: 3x3, Stride: 1, Padding: 1

4.3.2 Residual U-net Regularization Techniques

In order to address the class imbalance it was decided to use basic weighting as it was proven to be the most effective on the vanilla FCN. Inverse number of samples was also tested but no improvement was found. Surprisingly, it was also discovered the using data augmentation was also damaging the model's overall performance. Given further time it would be interesting to try and remedy this. The ideal learning rate was found to be 0.005 with a weight decay of 0.001. However, since the model took hours to train due to it's overall size it further gradient descent optimization should be considered. A batch size of 16 was found to be the largest that could be ran on the given GPU. The weights were initialized using the Xavier initialization method.

Table 4: The architecture of the Residual U-net. See the above tables for individual block details.

Full Network Architecture	
Network Element	**Dimensions (Refer to above tables for individual block details)**
Input/Encoder Block 1	Input: 3x384x768, Output: 32x192x384
Encoder Block 2	Input 32x192x384, Output: 64x96x192
Encoder Block 3	Input: 64x96x192, Output: 128x48x96
Encoder Block 4	Input: 128x48x96, Output: 256x24x48
Encoder Block 5	Input: 256x24x48, Output: 512x12x24
Encoder Block 6	Input: 512x12x24, Output: 1024x6x12
Decoder Block 1	Input: 1024x6x12, Output: 512x12x24
Decoder Block 2	Input: 512x12x24, Output: 256x24x48
Decoder Block 3	Input: 256x24x48, Output 128x48x96
Decoder Block 4	Input: 128x48x96, Output: 64x96x192
Decoder Block 5	Input: 64x96x192, Output: 32x192x384
Upsample	Input: 32x192x384, Output 32x384x768, kernel: 3x3, stride: 2
Classifier	Input 32x384x768, Output 10x384x768, kernel: 1x1, stride: 1

4.4 Transfer Learning

Transfer learning is the method that uses large pre-trained network as starting point and finetune the model for specific tasks. This method greatly speeds up the training time as seen from our experiment while the early pre-trained layers gave great internal representation to produce good results.

In this part, we used ResNet34 as the encoder and the same decoder as the baseline model. To match the dimensions of the decoder output, we dropped the last two layers(pooling and fc) of the Res34. For the weight initialization of the decoder, we used the Xavier uniform for the weights and normal for the bias. We also compared SGD optimizer which is the one used in the original resnet paper and Adam optimizer for gradient descent. In addition, we ran the model both by freezing the pretrained parameters and unfreezing the parameters to see which one produces better outcome.

The deal with the class imbalance problem, we used the cross entropy with standard weighting as loss function as explained in the above section. Each class is given $w_c = 1 - \frac{n_c}{\sum_i n_i}$ weight.

The model architecture is as follows:

Table 5: Transfer Learning Model Architecture with Resnet 34

Layer	**dimensions**
Encoder(pretained resnet34)	
conv	$3 \rightarrow 64$, kernel size=(7, 7), stride=(2, 2), padding=(3, 3), Batch Norm, ReLU
MaxPool	kernel size = 3, stride = 2, padding = 1
Sequential Layer 4	
conv1	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	$64 \rightarrow 64$, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm

Table 6: Transfer Learning Model Architecture with Resnet 34

Sequential Layer 5	
conv1	64 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv(downsample)	64 → 128,kernel size=(1, 1), stride=(2, 2), BatchNorm
conv1	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	128 → 128, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
Sequential Layer 6	
conv1	128 → 256, kernel size=(3,3), stride=(2, 2), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv(downsample)	128 → 256,kernel size=(1, 1), stride=(2, 2), BatchNorm
conv1	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	256 → 256, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
Sequential Layer 7	
conv1	256 → 512, kernel size=(3,3), stride=(2, 2), padding=(1, 1), Batch Norm, ReLU
conv2	512 → 512, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv(downsample)	256 → 512,kernel size=(1, 1), stride=(2, 2), BatchNorm
conv1	512 → 512, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	512 → 512, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
conv1	512 → 512, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm, ReLU
conv2	512 → 512, kernel size=(3,3), stride=(1, 1), padding=(1, 1), Batch Norm
ReLU	
Decoder	
deconv1	512 → 512, kernel (3, 3), stride=(2, 2), padding=(1, 1), Batch Norm
deconv2	512 → 256, kernel (3, 3), stride=(2, 2), padding=(1, 1), Batch Norm
deconv3	256 → 128, kernel (3, 3), stride=(2, 2), padding=(1, 1), Batch Norm
deconv4	128 → 64, kernel (3, 3), stride=(2, 2), padding=(1, 1),Batch Norm
deconv5	64 → 32, kernel (3, 3), stride=(2, 2), padding=(1, 1),Batch Norm
Conv (classifier)	32 → 10, kernel (1, 1), stride=(1, 1)

4.5 U-Net for Image Segmentation

Alongside the previous models, we also implemented Ronneberger and Fischer's U-Net model for image segmentation. This involved implementing the original model, and then augmenting it to improve performance. [9]

4.5.1 Original U-Net architecture

In this model, a contracting encoder is paired with an expanding decoder. Convolutional layers with max pooling contract the input, while deconvolution is used to re-expanding it for output. The encoder consists of four blocks of two 3x3 convolutional layers, each with ReLU activation, followed by a 2x2 max-pooling layer. With max pooling, translation invariance is achieved up to some degree and also images are downscaled by extracting most important features. The result of the encoder is a fifth ”bottom” layer in the U-shaped network. The decoder is constructed from four

blocks of a deconvolutional layer followed by two 3x3 convolutional layers with ReLU activation functions. At each decoder block, the output of encoder layers 4-2 are concatenated to the input prior to implementing the layer. This results in the concatenation of the encoder output 4 with decoder input 1, encoder output 3 with decoder input 2, encoder output 2 with decoder input 3, and finally encoder output 1 with decoder input 4. In total, the network consists for nine layers. Following the paper's use of drop-out, drop-out layers were added after each block.

4.5.2 U-Net architecture alterations and dice-loss

In order to improve the performance of U-Net and accommodate the specifics of the segmentation task, five changes were made to the original architecture. First, padding was applied to the layers. This was done to ensure that the input and output images were of the same size, a necessary condition for matching the size of the target tensor when calculation the loss. Second, batch-normalization was added to each convolutional layer prior to the ReLU activation. Third, drop out layers was added to each convolutional layer following the ReLU activation. Fourth, the adam optimizer was used over the original paper's stochastic-gradient descent implementation. Fifth, dice-loss was used to calculate network loss as opposed to the original use of cross entropy loss. As drop-out and dice-loss were not discussed in the previous section, they will be outlined here.

4.5.3 Drop-out

To ensure that U-Net's large number of layers did not result in over-fitting to the test set, drop-out was added following each block. Drop-out is a form of regularization in which neurons are randomly dropped from the layer output by ignoring the output of these neurons.

4.5.4 Dice-loss

Dice-loss is a measure of loss that accounts for class-imbalances by accounting for both the number of correctly predicted classes and the number of incorrectly predicted classes. [3]

For each class, we calculate the individual dice-score.

$$DiceScore_c = \frac{2 * TP}{TP + TN + FP + FN} \tag{9}$$

These scores are then averaged. As a higher value indicates greater prediction, the final dice score is subtracted from 1 to create the dice-loss.

$$Loss = 1 - \frac{\sum Score_c}{nclasses} \tag{10}$$

By accounting for both correctly predicted pixels and those that were missed, we can better capture how the model is failure to properly predict pixels in rarer classes.

Table 7: U-Net Model Architecture
Stride = 1 unless mentioned, image dimension = 768 x 384 for all inputs and outputs

Network	**Dimensions, Padding, Activation, Non-Linearity**
Encoder Block 1.1	Conv: 3 → 64, Same Padding, Batch Norm, ReLU activation, Drop out
Encoder Block 1.2	Conv: 64 → 64, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 1.3	MaxPool - kernel: 2x2
Encoder Block 2.1	Conv: 64 → 128, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 2.2	Conv: 128 → 128, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 2.3	MaxPool - kernel: 2x2
Encoder Block 3.1	Conv: 128 → 256, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 3.2	Conv: 256 → 256, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 3.3	MaxPool - kernel: 2x2
Encoder Block 4.1	Conv: 256 → 512, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 4.2	Conv: 512 → 512, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 4.3	MaxPool - kernel: 2x2
Bottom Block 5.1	Conv: 512 → 1024, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Encoder Block 5.2	Conv: 1024 → 1024, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 6.1	DeConv: 1024 → 512, Kernel: 2x2, stride=2
Decoder Block 6.2	Concatenate block 4.2 output and 6.1 output: 512 + 512 = 1024
Decoder Block 6.3	Conv: 1024 → 512, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 6.4	Conv: 512 → 512, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 7.1	DeConv: 512 → 256, Kernel: 2x2, stride=2
Decoder Block 7.2	Concatenate block 3.2 output and 7.1 output: 256 + 256 = 512
Decoder Block 7.3	Conv: 512 → 256, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 7.4	Conv: 256 → 256, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 8.1	DeConv: 256 → 128, Kernel: 2x2, stride=2
Decoder Block 8.2	Concatenate block 2.2 output and 8.1 output: 128 + 128 = 256
Decoder Block 8.3	Conv: 256 → 128, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 8.4	Conv: 128 → 128, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 9.1	DeConv: 128 → 64, Kernel: 2x2, stride=2
Decoder Block 9.2	Concatenate block 1.2 output and 9.1 output: 64 + 64 = 128
Decoder Block 9.3	Conv: 128 → 64, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Decoder Block 9.4	Conv: 64 → 64, Same Padding, Kernel: 3x3, Batch Norm, ReLU activation, Drop out
Output Block 9.5	Conv: 64 → 10, Same Padding, Kernel: 1x1

5 Results

In this section, results of different models will be discussed. The following legend will be used for the segmented test images.

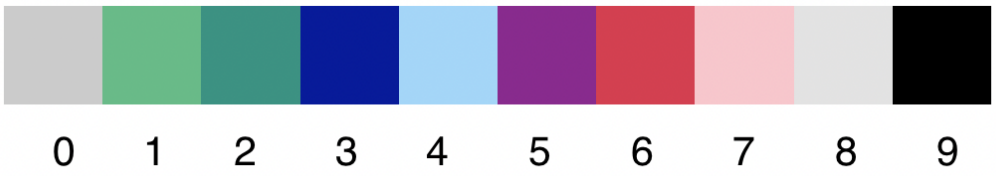
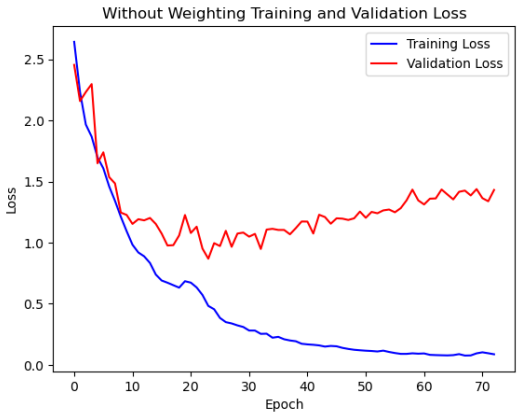


Figure 1: Legend for segmentation images.

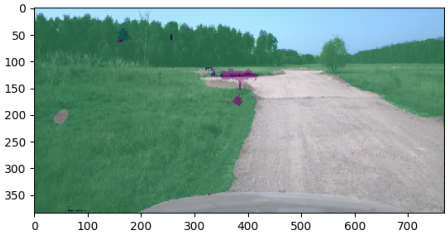
5.1 Baseline Model

Using a learning rate of 0.001, batch size of 32, we trained the baseline model for 100 epochs with early stopping enabled. We achieved a validation IoU of 47.21%, accuracy of 90.08% and loss of

1.362124. The graphs for training and validation loss as well as the segmented version of the first test image can be found below.



(a) Training and Validation Loss



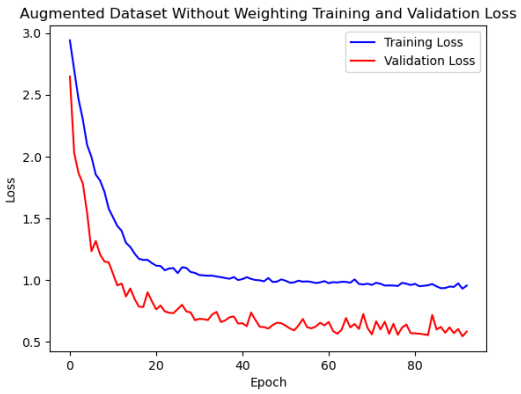
(b) Segmented Test Image

Figure 2: Training and Validation Loss Graph for Baseline model, Segmented Test Set Image

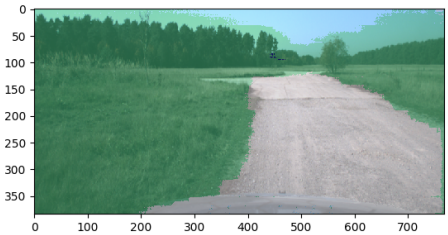
5.2 Improved Baseline Model

5.2.1 Augmented Dataset

After augmenting the dataset, we trained the baseline model for 300 epochs now, with a early stopping limit of 30 epoch. The same batch size and learning rate is used. This time, we achieved a validation IoU of 48.72%, accuracy of 86.79% and loss of 0.566520. The graphs for training and validation loss as well as the segmented version of the first test image can be found below.



(a) Training and Validation Loss

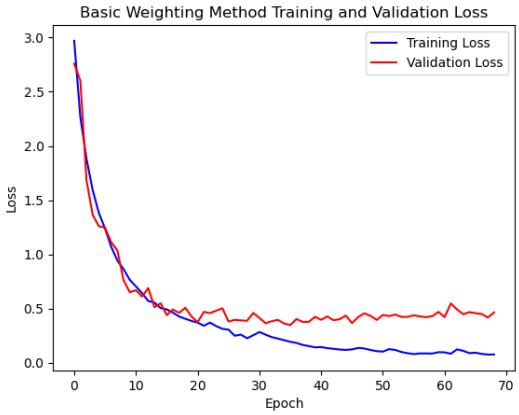


(b) Segmented Test Image

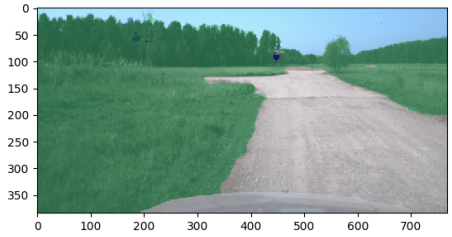
Figure 3: Training and Validation Loss Graph for Baseline model with augmented dataset, Segmented Test Set Image

5.2.2 Weighted Loss

After trying 3 different weighting methods, we achieved a validation IoU of 50.52%, accuracy of 91.80% and loss of 0.430165 for basic weighting method, a validation IoU of 50.44%, accuracy of 90.83% and loss of 0.288386 for inverse number of samples (INS) and finally, we achieved a validation IoU of 45.15%, accuracy of 87.23% and loss of 0.559670 for inverse square root number of samples (ISNS) method. Because of the better performance of basic weighting method, we decided to use that one. The graphs for training and validation loss for the basic weighting method as well as the segmented version of the first test image can be found below.



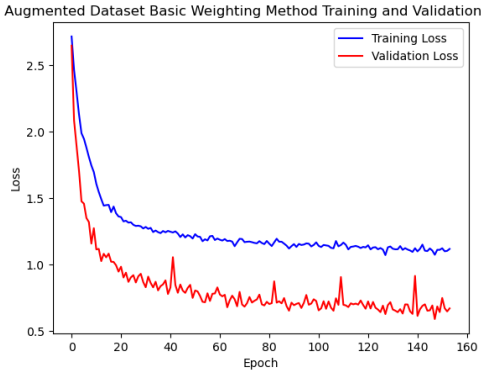
(a) Training and Validation Loss



(b) Segmented Test Image

Figure 4: Training and Validation Loss Graph for Baseline model with basic weighting method, Segmented Test Set Image

When we combined basic weighting method with augmented dataset, we achieved a validation IoU of 55.61%, accuracy of 89.00% and loss of 0.673270. The training was done for maximum of 300 epochs, with early stopping enabled, learning rate of 0.001 and batch size of 32. The graphs for training and validation loss as well as the segmented version of the first test image can be found below.



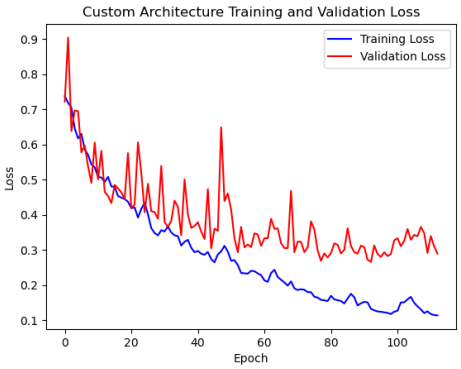
(a) Training and Validation Loss

(b) Segmented Test Image

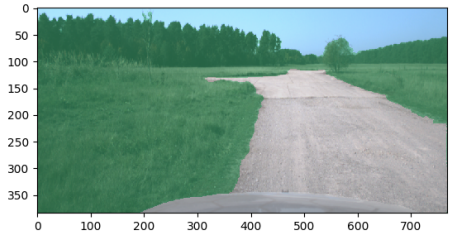
Figure 5: Training and Validation Loss Graph for Baseline model with augmented dataset and basic weighting method, Segmented Test Set Image

5.3 Experimental Residual U-net

The final results of the experimental residual U-net designed in this paper was an accuracy of 92.83% and a IoU of 65.00%. With further parameter tuning it is likely these results can be further improved. After experimenting with learning rates it was determined that 0.005 was ideal. In total it took 115 epochs to train at slightly over 2 hours to train on an Nvidia P100



(a) Training and Validation Loss

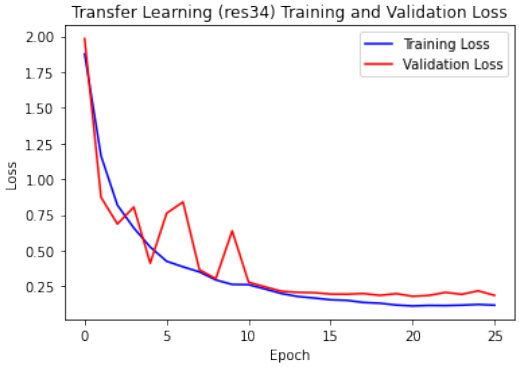


(b) Segmented Test Image

Figure 6: Training and Validation Loss Graph for Experimental model, Segmented Test Set Image

5.4 Transfer Learning

Using transfer learning with resnet34 as encoder and basic function decoder, we are able to achieve IoU of 63.11 %, pixel-accuracy of 94.78% and loss of 0.211075. We found ADAM converges much faster than SGD with better performance. In addition, unfreezing the parameters from the pretrained model gives about 4% improvement in pixel accuracy as compared to freezing the pretrained parameters. The implementation of standard weighting method, however, did not improve the performance and was therefore not used in the reported model. Due to GPU out-of-memory limitation, the model was trained with batch size of 8 and we found learning rate of 0.001 to have best performance.



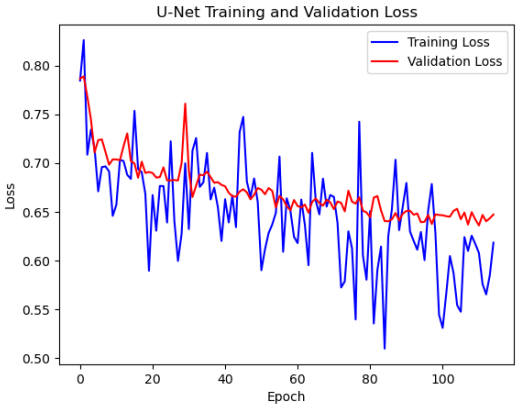
(a) Training and Validation Loss

(b) Segmented Test Image

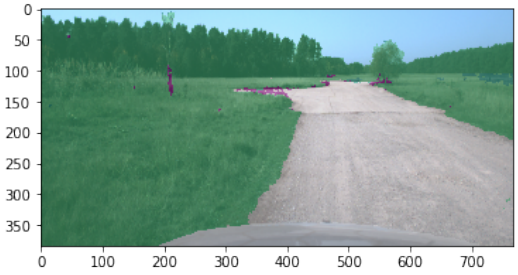
Figure 7: Training and Validation Loss Graph for Transfer Learning Model Res34

5.5 U-Net

The final results for the U-Net model were a validation IoU of 56.75%, pixel-accuracy of 90.2%, and a dice-loss of .64. After experimenting with multiple learning rates, a learning rate of .0009 was found to achieve the highest performance. The larger size of the U-Net model limited GPU space, requiring a batch-size of 4. Experimentation was performed with mixed-precision, however this yielded no improvements to the above metrics. No data-augmentation was applied to the training set in order to allow a comparison to the baseline-model.



(a) Training and Validation Loss



(b) Segmented Test Image

Figure 8: Training and Validation Loss Graph for U-Net model, Segmented Test Set Image

5.6 Table of Results

As a summary, the performance measures of each model can be found in the table below.

Table 8: Different models and their respective performance measures

Network	**IoU**	**Accuracy**	**Loss**
Baseline Model	47.21%	90.08%	1.362124
Data Augmented Baseline Model	48.72%	86.79%	0.566520
Basic Weighting Method Baseline Model	50.52%	91.80%	0.430165
Basic Weighting + Data Augmented Baseline Model	55.61%	89.00%	0.673270
Custom Network	65.00%	92.83%	0.283083
Transfer Learning	63.11%	94.78%	0.211075
U-Net Architecture	56.75%	90.21%	0.640345

6 Discussion

6.1 Baseline Model vs. Improved Baseline Model

As seen from the respective graphs, the methods we implemented to improve baseline method increased its performance, reducing the overfitting. Although applying only data augmentation seems like did not increase the performance, combining it with weighted loss increased the models performance, increasing IoU by 8%. Applying more transformations, such as crops, can solve this issue, since we only applied rotations and flips. When different weighting methods compared, it can be seen that ISNS is the worst performing one. One of the reasons for this can be the scaling of the weights, since we are dividing total number of samples by square root of sample per class, we end up getting very high weights for classes with fewer samples compared to the classes with more samples. On the other hand, INS and basic weighting performed very close to each other, however considering that this scaling issue will happen in INS as well, we decided to move on with basic weighting method. Notice that although the IoU and accuracy gets better, the loss fluctuates between the methods. We believe that this is because we considered IoU as the early stopping criteria during the training and the weighted structure of the loss.

6.2 Custom Architecture

The custom architecture proved to be very effective in improving the performance from both the baseline model and the U-net model. It is clear that by adding these residual connections over-fitting

can be avoided as this implementation improved upon the U-Net network by 8%. Additionally the bridges in the U-net help the model learn more detailed features which may be lost within deeper layers of the network. This can be seen from the 18% improvement over the vanilla model. However, since this network was deeper than the vanilla U-net and FCN models it required more GPU memory to train and the batch size had to be reduced to 16. Having access to a more powerful GPU proved to be instrumental in training the network. By implementing some of the techniques used on the vanilla FCN the accuracy improved even more but given more time it is likely this model could be further improved upon.

6.3 Transfer Learning

Transfer Learning provides a big improvement over the baseline model and has the highest pixel accuracy out of all of the networks. However, its IoU is a bit lower than the custom network that uses Residual U-net Architecture. We saw a significant improvement in training speed as compared to other models, since the pre-trained layers were already able to provide good internal representation and the fine-tuning was much faster with less parameters to learn. Res34 is a much deeper network as compared to the baseline model encoder and it uses skip connection to overcome the vanishing gradient problem. The depth helped the model to obtain better linear separation in the final layer and achieve better performance. We could try using deeper pre-trained models or other architectures in future studies.

6.4 U-Net

The U-Net provided an improvement over the baseline model, however it did not achieve comparable scores to the improved baseline model. This is likely due to the learning rate, as the resulting loss graph indicates a high level of volatility in the training loss. Due to time constraints, the model cannot be re-run with a lower learning rate. Another reason for lower performance than the improved baseline is the lack of data augmentation. This was chosen so as to compare U-Net to the baseline model, however the lower performance as compared to the improved model indicates the importance of using augmented data. Alongside the lower metric scores, the U-Net's larger architecture resulted in a longer training time. While the original model was meant to train on smaller data sets with improved performance over other image segmentation networks, the use of a larger data set led to a training time of over 4 hours on a Tesla T4 GPU. For the benefits of U-Net to be fully realized, a lower learning rate must be complemented by long training times or more computing power.

References

[1] Xavier initializations and regularization. https://cs230.stanford.edu/section/4/. Accessed: 2022-02-16.

[2] J. Dellinger. Weight initialization in neural networks: A journey from the basics to kaiming, April 2019.

[3] S. Du. Understanding dice loss for crisp boundary detection, Feb 2020.

[4] F. M. Graetz. Why adamw matters, Jun 2018.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[6] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[8] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation, 2015.

[9] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.