Fully Convolutional Networks for Semantic Segmentation of the India Driving Dataset

Allen Zeng*
University of California, San Diego
azeng@ucsd.edu

Shubham Chaudhary*
University of California, San Diego
shchaudh@ucsd.edu

Tara Mirmira*
University of California, San Diego
tmirmira@ucsd.edu

Zonglin Di*
University of California, San Diego
zodi@ucsd.edu

Abstract

This report explores the use of several deep learning architectures for the task of performing image semantic segmentation on the India Driving Dataset. First, we implemented a baseline fully convolution network (FCN) and evaluated its performance using the metrics of pixel accuracy and mean Intersection over Union (mIoU). Optimizing for Cross-Entropy loss for this model, we achieved 78.26% accuracy and 0.6430 mIoU. Using data augmentation and optimizing for Cross-Entropy loss for this model, we achieved 74.96% accuracy and 0.6002 mIoU. We experimented UNet with the cross-entropy loss Without data augmentation and optimizing for Dice loss on the baseline model, we achieved 58.77% accuracy and 0.4185 mIoU. We also experimented with using other architectures for this problem. We implemented a U-Net model, achieving 68.22% accuracy and 0.5173 mIoU. We also implemented a Res-FCN model and optimized for class-weighted Cross-Entropy loss, achieving 80.21% accuracy and 0.6695 mIoU. We also experimented with a Res-UNet architecture, optimizing for class-weighted Cross-Entropy loss.

1 Introduction

Semantic segmentation has a large impact on ensuring safety in autonomous driving. In this report, we explore the use of fully convolutional networks (FCNs) for the task of image semantic segmentation. For an image, this task involves labeling per-pixel the corresponding class of the object to which the pixel belongs. In contrast to object classification or bounding box detection, this results in a dense prediction of class in an image. Semantic segmentation finds many uses in the computer vision space, but we specifically apply a randomly-sampled subset of the India Driving Dataset [1]. This dataset consist of images obtained from a 1080p-resolution, front-facing camera mounted onto a car driving through Hyderabad and Bangalore, India. The dataset consists of 26 distinct categories and one unlabeled category.

In order to segment the images of this dataset, we use a simple FCN, the U-Net model [2], the Res50-FCN model [3], and some novel architectures based on those models. We use batch normalization [4] with Xavier weight initialization [5], and the Adam Optimizer [6] to accelerate learning. When training deep neural networks, the distribution of each layers' inputs changes during training as weights in the previous layers are updated. Batch normalization is used to reduce this internal covariate shift during training. During training via mini-batch stochastic gradient descent, batch

*For equal contribution

normalization attempts to keep the mean and variance of inputs consistent across mini-batches. It maintains two parameters that can be learned, γ and β, for each activation that scale and shift the normalized mini-batch. For some i-th minibatch at the k-th activation $x_i^{(k)}$, normalized to zero mean and unit variance $\hat{x}_i^{(k)}$, batch normalization performs

$$y_i^{(k)} = \text{BN}_{\gamma,\beta}\left(x_i^{(k)}\right) = \gamma\hat{x}_i^{(k)} + \beta$$

As shown in the batch normalization paper [4], this is a transform and the parameters γ and β can be learned using back propagation. In Xavier weight initialization [5], the network's weights are initialized by sampling from a uniform distribution bounded by

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

where n_i represents layer fan-in and n_{i+1} represents layer fan-out. This initialization has the effect of maintaining similar variances of weights across the networks' layers throughout training, which in turn speeds up network convergence. In Adam optimization [6], first-order gradient-based optimization is augmented with adaptive estimates of the lower-order moments. A benefit of using Adam is that the hyper-parameters typically require little tuning, and helps reach convergence faster than plain stochastic gradient descent.

2 Related Work

There are well established architectures for performing semantic segmentation, such as *U-Net* [2] and *ResNet* [3]. In our work we implement a baseline fully convolutional network (FCN) model, models inspired by U-Net, and models inspired by ResNet. Other architectures that perform semantic segmentation include *DeepLab* [7], which is notable for introducing atrous/dilated convolutions with enlarged receptive fields, and *Xception* [8], which is notable for introducing depthwise separable convolutions.

3 Methods

3.1 Baseline

We trained the basic fully convolutional network provided in the starter code on the original data (i.e. without any data augmentation). To address the memory limitations of the GPU, we performed a center crop on the images to reduce them to 512 x 512 images. With this size, we were able to use a batch size of 16. We also tried cropping the images to 256 x 256 and used a batch size of 64, but we achieved better results with the 512 x 512 images. For the results on the 512 x 512 images presented in Table 1, we used a learning rate of 0.01, a weight decay of 0.00001, a multiplicative factor of learning rate decay (gamma) of 0.5, and a step size of 50. We trained the model for 50 epochs. After training the model for 50 eopchs, we found that the validation loss was lowest at epoch 37. The values for the training dataset are shown at epoch 50. The values for the validation and test dataset are shown at epoch 37. A graph of the training and validation loss curves is presented in Figure 2.

Figure 1: Effect of data augmentation

To improve the performance of the baseline model, we implemented data augmentation, in which we performed random horizontal flips, slight rotations and random cropping to the images present in the dataset. All these operations were applied independently with probability 0.5. The angle of rotation was uniformly distributed in the range of $-25°$ to $+25°$. Random cropping was performed

by uniformly sampling a point in the image that would be the top left point of the final image. This point was selected to allow cropping to the final desired size. We also added Gaussian blur with probability 0.5 The baseline model was then trained on both the original data as well as the modified data. Figure 1 demonstrates the effect of data augmentation on some images of training set. As before, we cropped the images to 512 x 512 and used a batch size of 16. For the results presented in Table 2, we used a learning rate of 0.01, a weight decay of 0.00001, a multiplicative factor of learning rate decay (gamma) of 0.5, and a step size of 50. After training the model for 50 epochs, we found that the validation loss was lowest at epoch 40. The values for the training dataset are shown at epoch 50. The values for the validation and test dataset are shown at epoch 40. A graph of the training and validation loss curves is presented in Figure 3.

To improve the performance of the baseline model by correcting for the class imbalance problem, we tried implementing the weighted loss version of the traditional cross entropy loss. The weights were calculated by iterating over the labels from the training set and counting the number of pixels in each class C_i. If class C_i was seen $counts_i$ times in the training set, the corresponding cross entropy loss weight for class C_i would be calculated in Equation 1. Because the class imbalance is so severe, so we did not use $\frac{1}{counts_i}$ directly. The statistic can be found in Table 8 We chose not to use data augmentation in the training of the baseline model with weighted loss for two reasons. First, we found data augmentation hurt model performance. Second, we wanted to evaluate the impact of weighted loss on the baseline model's performance so we did not want to additionally change the data variable as part of this experiment. As before, we cropped the images to 512 x 512 and used a batch size of 16. For the results presented in Table 3, we used a learning rate of 0.01, a weight decay of 0.00001, a multiplicative factor of learning rate decay (gamma) of 0.5, and a step size of 50. After training the model for 50 epochs, we found that the validation loss was lowest at epoch 50. The values for the training dataset are shown at epoch 50. The values for the validation and test dataset are shown at epoch 50. A graph of the training and validation loss curves is presented in Figure 4.

We also tried correcting the class imbalance problem by implementing and using dice loss as our loss function. To calculate the loss on a dataset of size N consisting of images of size $H \times W$ with pixels from C classes, we performed the following operations:

(1) Numerator: For each $C \times H \times W$ matrix of predicted probabilities and each $C \times H \times W$ matrix of targets, we computed an element-wise multiplication. This represents a logical and of the two matrices to quantify the similarities. We multiplied this value by 2. Output: a matrix of size $C \times H \times W$.

(2) Denominator: We squared the elements of the predicted probabilities matrix and added this to the squared elements of the targets matrix. Output: a matrix of size $C \times H \times W$. This represents the union of the two matrices to quantify the size of the differences as well as the similarities.

(3) We then performed an element-wise division of the numerator and denominator matrices to get another $C \times H \times W$ matrix. We summed all the elements of this matrix and divided by $N * C$ to normalize the loss. This value represents the set of pixels the model predicted correctly so we subtract this value from 1 to calculate dice loss.

Like with weighted cross entropy, we chose not to use data augmentation for this experiment. We cropped the images to 256 x 256 and used a batch size of 32. For the results presented in Table 4, we used a learning rate of 0.01, a weight decay of 0.00001, a multiplicative factor of learning rate decay (gamma) of 0.5, and a step size of 50. After training the model for 50 epochs, we found that the validation loss was lowest at epoch 50. The values for the training dataset are shown at epoch 50. The values for the validation and test dataset are shown at epoch 50. A graph of the training and validation loss curves is presented in Figure 5.

3.2 Experiments

For transfer learning, we changed the backbone from the basic encoder to a ResNet-18 model [3]. We implemented ResNet-18 ourselves rather from the *torchvision* library. For model weight, we used it from *torchvision* site and load it with a non-strict flag. We initialized the encoder part using

the ImageNet weight and the decoder part is initialized using bi-linear insertion way. We only used the last 3 stages of the encoder for the following decoder. The details of FCN with ResNet-18 can be found in 12.

For the inspiration part, due to the time limit and the GPU limit, we combined the residual block and U-Net together. We replaced the standard U-Net encoder with ResNet-50. Different from FCN with ResNet-18, we used ResNet-50 from *torchvision* directly. We used 6 stages of ResNet-50. The smallest feature map is $2048 \times 32 \times 32$. In training stage, we initialized the encoder with ImageNet pre-trained weight. The details of U-Net with ResNet-50 can be found in 14.

$$
\begin{aligned}
sum &= \sum_{i=1}^{27} counts_i \\
weights_i &= \frac{0.15 * \log sum}{counts_i}
\end{aligned}
\tag{1}
$$

Table 1: Evaluation Metrics for Baseline Model

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.7915	0.6547	0.8531	0.0619	0.4855	0.1739	0.9160
Validation	0.7815	0.6411	0.8691	0.0439	0.4869	0.1441	0.9035
Test	0.7826	0.6430	0.8776	0.0514	0.4621	0.1380	0.9145

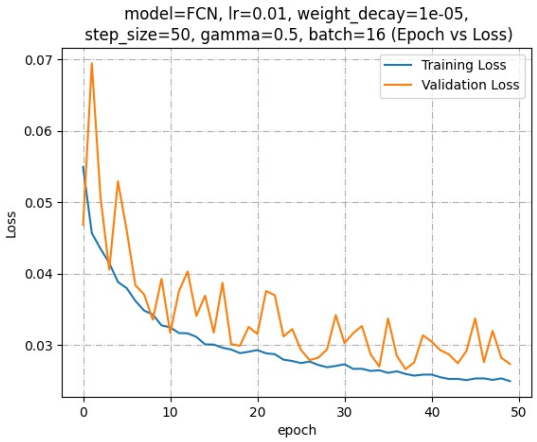

Figure 2: Training and Validation Loss for FCN Baseline Model without Data Augmentation

Table 2: Evaluation Metrics for FCN Baseline Model with Data Augmentation

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.7748	0.6332	0.8754	0.0	0.4180	0.1716	0.9117
Validation	0.7737	0.6309	0.8691	0.0	0.4397	0.1821	0.9098
Test	0.7496	0.6002	0.8575	0.0	0.3779	0.1038	0.9048

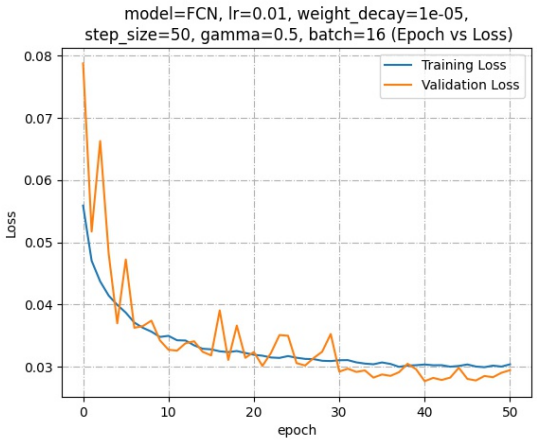

Figure 3: Training and Validation Loss for FCN Baseline Model with Data Augmentation

Table 3: Evaluation Metrics for FCN Baseline Model with Weighted Cross Entropy Loss

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.7609	0.6138	0.8442	0.14 75	0.5282	0.2406	0.9202
Validation	0.7807	0.6400	0.8822	0.1409	0.5213	0.2132	0.9204
Test	0.7805	0.6409	0.8864	0.1549	0.5025	0.2115	0.9238

Table 4: Evaluation Metrics for FCN Baseline Model with Dice Loss

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.6859	0.5219	0.8145	0.02349	0.4541	0.1871	0.8529
Validation	0.5945	0.4250	0.7275	0.0315	0.3947	0.1153	0.8781
Test	0.5877	0.4185	0.7210	0.0355	0.3919	0.1151	0.8758

3.3 Experimentation

We use `torchsummary` package to show the details of the network. The details of baseline model is Table 9 in the Appendix.

4 Results

4.1 Evaluation metrics

First, we descibe the evaluation metrics used in our experiments. We used two methods. One is pixel accuracy and the other is mean Intersection over Union. Pixel accuracy is the ratio of total correct predictions and total number of samples. However, this method cannot describe the *class imbalance*. For example, if there are 3 classes in one picture, whose pixel ratio is 80%, 10%, 10%. If the classifier predicts all the pixels as the first class, the pixel accuracy is still 80%. Thus, this is not a good method to evaluate imbalance.

Therefore, we use another metric called mean Intersection over Union (mIoU). Per-class IoU can be calculated as $IoU = \frac{TP}{TP+FP+FN}$ where TP, FP, FN refer to the number of pixels classified as true positive, false positive, and false negative, respectively. Similarly, mIoU can be calculated from

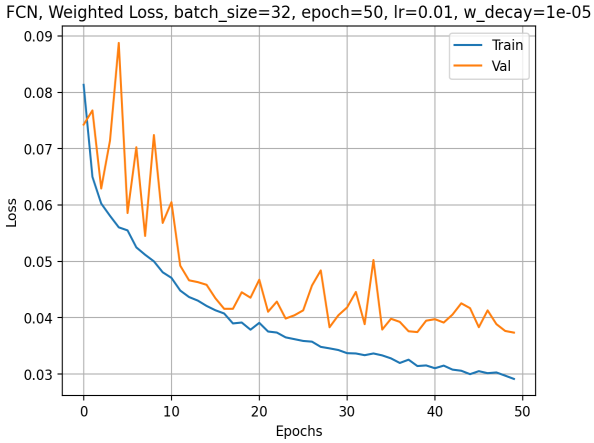

Figure 4: Training and Validation Loss for FCN Baseline Model with Weighted Cross Entropy Loss

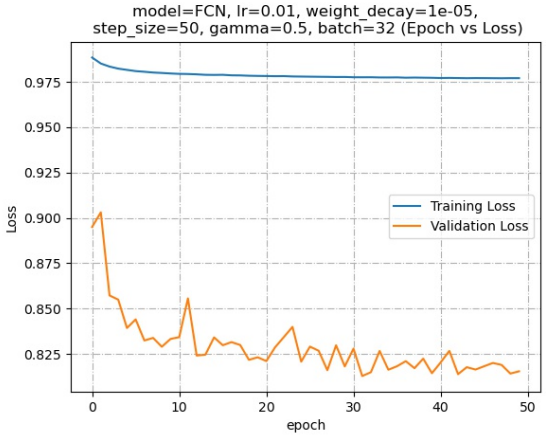

Figure 5: Training and Validation Loss for FCN Baseline Model with Dice Loss

all of the class's counts of TP, FP, FN. Typically, mIoU is be calculated as either a ratio of sums of the counts or as a direct averaging of calculated per-class IoUs. We have chosen the former, and so we formulate mIoU as

$$mIoU = \frac{\sum_k TP_k}{\sum_k TP_k + FP_k + FN_k}$$

where k refers to the class index. We chose this method of mIoU calculation because it values "correctness" as the total number of pixels correctly classified, regardless of class, divided by the union of correctly and incorrectly classified pixels. So the more pixels are classified, the better. The other mIoU formulation, $mIoU = \frac{1}{n}\sum_k IoU_k$, weighs each class's "correctness" the same regardless of the size of the class – so classes with fewer examples have a more significant impact on this mIoU metric than on our chosen mIoU metric. For our task of semantic segmentation on the India Driving Dataset, we believe it is important to measure the total number of pixels classified correctly regardless of class size, so the ratio of sums version was chosen. In the following sections, the mIoU values reported are all calculated using the ratio of sums formula.

4.2 Baseline Without Data Augmentation

The results for the baseline model trained on the original dataset (i.e. without any data augmentation) can be found in Table 1 and Figure 2. The visualization of segmentation output on test set has been presented in 6.

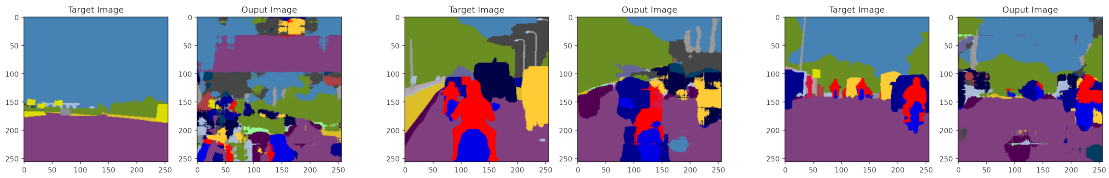

Figure 6: Performance of FCN on test set

4.3 Baseline With Data Augmentation

The results for the baseline model trained with data augmentation can be found in Table 2 and Figure 3.

4.4 Baseline With Weighted Loss

The results for the baseline model trained with weighted cross entropy loss can be found in Table 3 and Figure 4. The visualization of segmentation output on test set has been presented in 7.

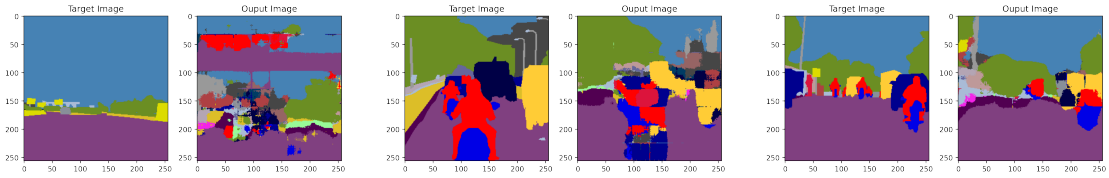

Figure 7: Performance of FCN with dice loss on test set

4.5 Baseline With Dice Loss

The results for the baseline model trained with weighted dice loss can be found in Table 4 and Figure 5. The visualization of segmentation output on test set has been presented in 8.

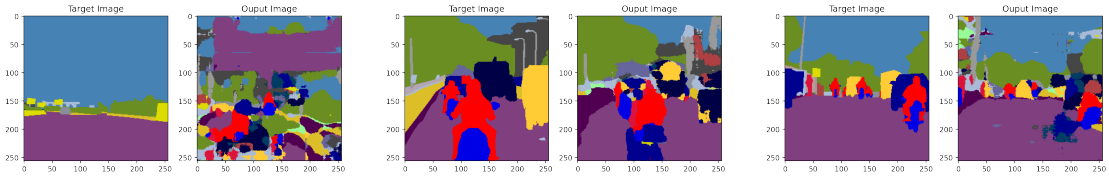

Figure 8: Performance of FCN with dice loss on test set

4.6 U-Net

The architecture of U-Net [2] is similar to the baseline model. Apart from using convolutional layers as encoder and decoder, it also concatenates the feature maps constructed during encoding to the up-sampled feature map while decoding. The concatenation of these feature maps improves the localized information and leads to better segmentation. In the original paper, the feature maps from encoder segment are cropped so as to allow concatenation with smaller feature maps of decoder segment, this ultimately leads to a smaller output than the original image. In order to circumvent this problem we have instead padded the decoder feature maps so as to match the size of encoder feature maps. In this section we present the results of U-Net trained on the dataset of image sizes 512×512, with learning rate 0.01, batch size 8 and trained for 50 epochs. The validation statistics are presented in Table 5 and the change in training and validation losses with increase in epochs in Figure 9. The segmentation results of this model on some images of test set have been demonstrated in Figure 10.

Table 5: Evaluation Metrics for U-Net Model with Cross Entropy Loss

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.7662	0.6209	0.8527	0.03 85	0.4836	0.0963	0.9227
Validation	0.8020	0.6688	0.8971	0.0552	0.5579	0.1148	0.9396
Testing	0.8000	0.6664	0.9006	0.0625	0.5329	0.1196	0.9388

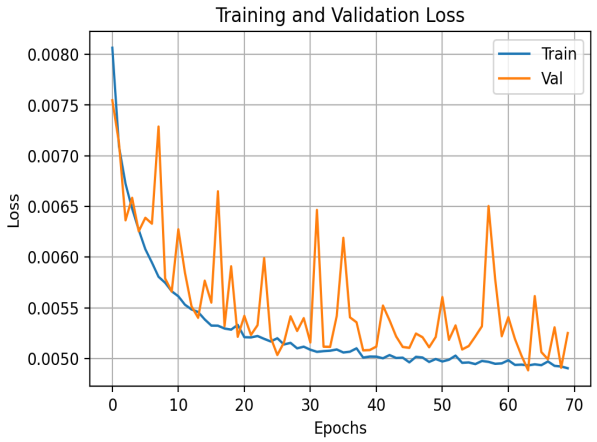

Figure 9: Training and Validation Loss for U-Net

4.7 U-Net With Weighted Loss

In this section we present the results of training a U-Net model with weighted loss. The training and validation losses have been presented in Figure 11 and the visualization of output segmentations have been presented in Figure 11

4.8 FCN with ResNet-18 backbone

For this experiment, we used $ResNet - 18$ as the backbone. The hyper-parameters is the same as FCN baseline given in . The batchsize is 64. The input size is 512×512. The results of Res-FCN can be found in Table 7. Its training loss and validation loss curve is in Fig. 14 and the illustration can be found in 13

Table 6: Experiment Result for Res-FCN Baseline Model without Data Augmentation and weighted loss

Dataset Type	Average Pixel Accuracy	Average IoU	IoU Road	IoU Sidewalk	IoU Car	IoU Billboard	IoU Sky
Training	0.8321	0.7122	0.9119	0.1933	0.6924	0.2572	0.9461
Validation	0.8046	0.6723	0.8891	0.1469	0.6115	0.2226	0.9410
Test	0.8021	0.6695	0.8887	0.1941	0.6306	0.2145	0.9409

4.9 U-Net with ResNet-50 Backbone

As mentioned in Section 3.2, we replaced the backbone of UNet with a resnet-50 mode. We trained the model using batch size 8, learning rate 0.01 and weight_decay 1e-5.

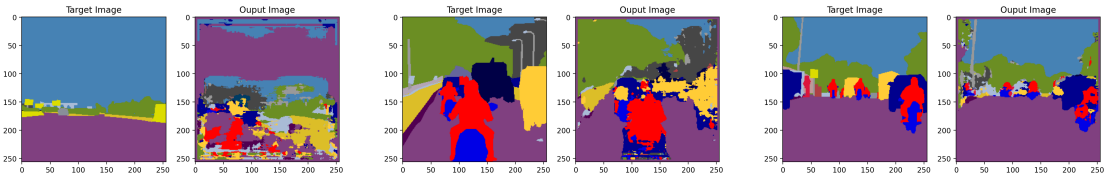

Figure 10: Performance of U-Net on test set

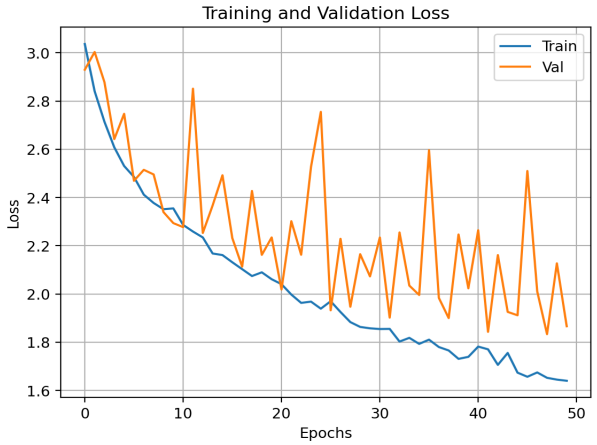

Figure 11: Training and Validation Loss for U-Net with weighted loss

5 Discussion

The FCN baseline model trained on only the original data performs comparably well with the baseline model trained on an augmented dataset. The FCN baseline model trained on augmented data performs slightly worse. This suggests that the FCN model does not provide sufficient invariance to the movement of pixels. Further, we see that data augmentation worsened the performance on some rare classes such as sidewalk and billboard more than it worsened the performance on more common classes. This suggests data augmentation may be a helpful technique for generalization, but does not address the class imbalance issue. To address class imbalance, modification of the loss function performs best. We can see that the baseline model trained with weighted cross entropy loss reduces the overall average pixel accuracy as compared to the baseline model trained with unweighted cross entropy loss. However, the usage of weighted cross entropy loss significantly imroves the accuracy of the model on rare classes such as sidewalk and billboard without hurting the accuracy of popular classes. A similar situation arises when using dice loss as the loss function. While the performance of the FCN baseline model trained with dice loss is much worse than the FCN baseline model trained with cross entropy loss, we can see that the performance difference between popular classes and rare classes has been reduced when dice loss was used. The graph of training and validation loss is reversed for the FCN model trained with dice loss because training loss is higher than validation loss, while typically one expects training loss to be slightly lower than validation loss. This is likely due to a difference in the ratio of popular to rare classes in the training and validation datasets. In dice loss, the loss of a particular pixel has a factor of the softmax probability of the pixel being its correct value to reward higher certainty. If the validation set has fewer rare class pixels, then, due to higher certainty in popular pixels, it will have a lower loss than the training set. Weighted cross entropy performs better than dice loss. Dice loss is commonly used in medical imaging tasks so it is possible that it does not work as well for our task.

From the model architecture perspective, we can find that skip connection from the U-Net can keep details better than FCN, which will send more information from the encoder using concatenation operation. Thus, the performance of U-Net is better than FCN. A better backbone can also encode the information better so as to improve the performance which can be found in FCN with ResNet-18

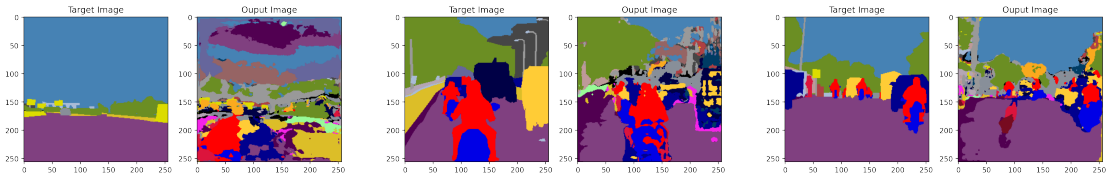

Figure 12: Performance of U-Net with weighted loss on test set

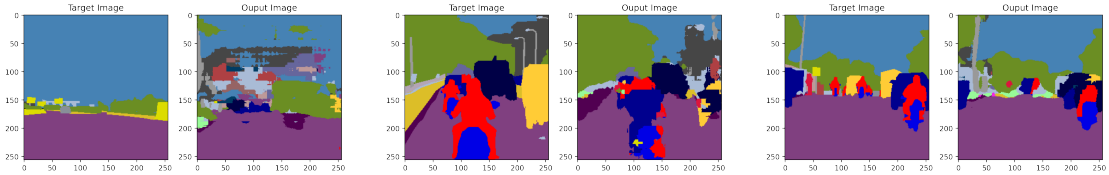

Figure 13: Performance of U-Net with weighted loss on test set

backbone compared with the original FCN. For ResUnet, we used 25 epochs due to the time limit. However, in the encoder-decoder architecture, the feature map from the encoder should not too small like 1×1, which will hurt the performance in decoding part.

As for data augmentation, we selected random rotation, random crop and horizontal flip. There is no big improvement from data augmentation. The reason may be that we only trained 50 epoch for every model due to the time limit, which the model has not converged. Another reason may be that we did not choose proper augmentation methods. Since there is no significant improvement in model performance using data augmentation, we trained the other models without data augmentation so the experiments ran faster.

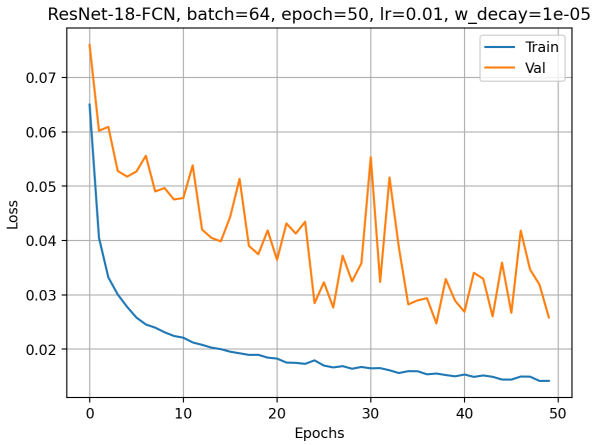

Figure 14: Training and Validation Loss for FCN with ResNet 18 backbone

Table 7: Experiment Result for Res-FCN Baseline Model without Data Augmentation and weighted loss

Dataset Type	**Average Pixel Accuracy**	**Average IoU**	**IoU Road**	**IoU Sidewalk**	**IoU Car**	**IoU Billboard**	**IoU Sky**
Training	0.7548	0.6060	0.8794	0	0.3284	0.2572	0.8000
Validation	0.7495	0.5985	0.8759	0	0.3261	0	0.7581
Test	0.7619	0.6144	0.8887	0.8744	0.3149	0	0.7491

Table 8: Number of Pixel and Weight for Different Class in the Training Set

Class Index	Class Name	Number of Pixels	Loss Weight
0	road	2,198,453,584	1.0
1	drivable fallback	519,609,102	1.0
2	sidewalk	28,929,892	3.766
3	non-drivable fallback	126,362,539	2.292
4	person/animal	58,804,972	3.057
5	rider	59,032,134	3.053
6	motorcycle	94,293,190	2.585
7	bicycle	2,569,952	6.187
8	autorickshaw	101,519,794	2.511
9	car	163,659,019	2.033
10	truck	105,075,839	2.476
11	bus	47,400,301	3.272
12	vehicle fallback	30,998,126	3.697
13	curb	133,835,645	2.234
14	wall	135,950,687	2.219
15	fence	41,339,482	3.409
16	guard rail	15,989,829	4.359
17	billboard	104,795,610	2.479
18	traffic sign	8,062,798	5.044
19	traffic light	450,973	7.927
20	pole	94,820,222	2.579
21	obs-str-bar-fallback	341,805,135	1.296
22	building	557,016,609	1.0
23	bridge/tunnel	71,159,311	2.866
24	vegetation	1,465,165,566	1.0
25	sky	1,823,922,767	1.0
26	unlabeled	2,775,322	6.110

Individual Contributions

Tara Mirmira: I implemented dice loss and helped with the implementation of weighted cross entropy loss and debugging of the training procedure. I contributed to the running of experiments, collection of results, and writing of the report.

Zonglin Di: I implemented the training loop and the experimental architectures like the ResNet model and the U-Net with ResNet backbone model. I contributed to running the experiments, collecting results, and writing of the report. I finished the results of FCN, weight FCN, UNet, ResFCN, ResUNet because I have 4 3090 GPU locally.

Allen Zeng: I implemented the baseline model, evaluation metrics, and helped debug the training and validation procedure by generating a synthetic dataset. I ran some of the experiments and contributed to the writing of the report.

Shubham Chaudhary: I implemented data augmentation and helped debug the training loop. Implemented one version of UNet. Contributed in running experiments, wrote code for visualizing the model results, and report writing.

References

[1] Girish Varma, Anbumani Subramanian, Anoop Namboodiri, Manmohan Chandraker, and C V Jawahar. Idd: A dataset for exploring problems of autonomous navigation in unconstrained environments. In *IEEE Winter Conf. on Applications of Computer Vision (WACV)*, 2019.

[2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[5] James Dellinger. Weight initialization in neural networks: A journey from the basics to kaiming.

[6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[7] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017.

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

Appendix

Table 9: Details about FCN Baseline Architecture

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 256, 256]	896
ReLU-2	[-1, 32, 256, 256]	0
BatchNorm2d-3	[-1, 32, 256, 256]	64
Conv2d-4	[-1, 64, 128, 128]	18,496
ReLU-5	[-1, 64, 128, 128]	0
BatchNorm2d-6	[-1, 64, 128, 128]	128
Conv2d-7	[-1, 128, 64, 64]	73,856
ReLU-8	[-1, 128, 64, 64]	0
BatchNorm2d-9	[-1, 128, 64, 64]	256
Conv2d-10	[-1, 256, 32, 32]	295,168
ReLU-11	[-1, 256, 32, 32]	0
BatchNorm2d-12	[-1, 256, 32, 32]	512
Conv2d-13	[-1, 512, 16, 16]	1,180,160
ReLU-14	[-1, 512, 16, 16]	0
BatchNorm2d-15	[-1, 512, 16, 16]	1,024
ConvTranspose2d-16	[-1, 512, 32, 32]	2,359,808
ReLU-17	[-1, 512, 32, 32]	0
BatchNorm2d-18	[-1, 512, 32, 32]	1,024
ConvTranspose2d-19	[-1, 256, 64, 64]	1,179,904
ReLU-20	[-1, 256, 64, 64]	0
ConvTranspose2d-22	[-1, 128, 128, 128]	295,040
ReLU-23	[-1, 128, 128, 128]	0
BatchNorm2d-24	[-1, 128, 128, 128]	256
ConvTranspose2d-25	[-1, 64, 256, 256]	73,792
ReLU-26	[-1, 64, 256, 256]	0
BatchNorm2d-27	[-1, 64, 256, 256]	128
ConvTranspose2d-28	[-1, 32, 512, 512]	18,464
ReLU-29	[-1, 32, 512, 512]	0
BatchNorm2d-30	[-1, 32, 512, 512]	64
Conv2d-31	[-1, 27, 512, 512]	891

Table 10: Details about UNet Architecture

Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 512, 512]	1,792	Conv2d-53	[-1, 128, 256, 256]	147,584
BatchNorm2d-2	[-1, 64, 512, 512]	128	BatchNorm2d-54	[-1, 128, 256, 256]	256
ReLU-3	[-1, 64, 512, 512]	0	ReLU-55	[-1, 128, 256, 256]	0
Conv2d-4	[-1, 64, 512, 512]	36,928	ConvTranspose2d-56	[-1, 64, 512, 512]	32,832
BatchNorm2d-5	[-1, 64, 512, 512]	128	Conv2d-57	[-1, 64, 512, 512]	73,792
ReLU-6	[-1, 64, 512, 512]	0	BatchNorm2d-58	[-1, 64, 512, 512]	128
MaxPool2d-7	[-1, 64, 256, 256]	0	ReLU-59	[-1, 64, 512, 512]	0
Conv2d-8	[-1, 128, 256, 256]	73,856	Conv2d-60	[-1, 64, 512, 512]	36,928
BatchNorm2d-9	[-1, 128, 256, 256]	256	BatchNorm2d-61	[-1, 64, 512, 512]	128
ReLU-10	[-1, 128, 256, 256]	0	ReLU-62	[-1, 64, 512, 512]	0
Conv2d-11	[-1, 128, 256, 256]	147,584	Conv2d-63	[-1, 27, 512, 512]	1,755
BatchNorm2d-12	[-1, 128, 256, 256]	256			
ReLU-13	[-1, 128, 256, 256]	0			
MaxPool2d-14	[-1, 128, 128, 128]	0			
Conv2d-15	[-1, 256, 128, 128]	295,168			
BatchNorm2d-16	[-1, 256, 128, 128]	512			
ReLU-17	[-1, 256, 128, 128]	0			
Conv2d-18	[-1, 256, 128, 128]	590,080			
BatchNorm2d-19	[-1, 256, 128, 128]	512			
ReLU-20	[-1, 256, 128, 128]	0			
MaxPool2d-21	[-1, 256, 64, 64]	0			
Conv2d-22	[-1, 512, 64, 64]	1,180,160			
BatchNorm2d-23	[-1, 512, 64, 64]	1,024			
ReLU-24	[-1, 512, 64, 64]	0			
Conv2d-25	[-1, 512, 64, 64]	2,359,808			
BatchNorm2d-26	[-1, 512, 64, 64]	1,024			
ReLU-27	[-1, 512, 64, 64]	0			
MaxPool2d-28	[-1, 512, 32, 32]	0			
Conv2d-29	[-1, 1024, 32, 32]	4,719,616			
BatchNorm2d-30	[-1, 1024, 32, 32]	2,048			
ReLU-31	[-1, 1024, 32, 32]	0			
Conv2d-32	[-1, 1024, 32, 32]	9,438,208			
BatchNorm2d-33	[-1, 1024, 32, 32]	2,048			
ReLU-34	[-1, 1024, 32, 32]	0			
ConvTranspose2d-35	[-1, 512, 64, 64]	2,097,664			
Conv2d-36	[-1, 512, 64, 64]	4,719,104			
BatchNorm2d-37	[-1, 512, 64, 64]	1,024			
ReLU-38	[-1, 512, 64, 64]	0			
Conv2d-39	[-1, 512, 64, 64]	2,359,808			
BatchNorm2d-40	[-1, 512, 64, 64]	1,024			
ReLU-41	[-1, 512, 64, 64]	0			
ConvTranspose2d-42	[-1, 256, 128, 128]	524,544			
Conv2d-43	[-1, 256, 128, 128]	1,179,904			
BatchNorm2d-44	[-1, 256, 128, 128]	512			
ReLU-45	[-1, 256, 128, 128]	0			
Conv2d-46	[-1, 256, 128, 128]	590,080			
BatchNorm2d-47	[-1, 256, 128, 128]	512			
ReLU-48	[-1, 256, 128, 128]	0			
ConvTranspose2d-49	[-1, 128, 256, 256]	131,200			
Conv2d-50	[-1, 128, 256, 256]	295,040			
BatchNorm2d-51	[-1, 128, 256, 256]	256			
ReLU-52	[-1, 128, 256, 256]	0			

Table 12: Details about Res-FCN Architecture

Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 256, 256]	9,408	Conv2d-54	[-1, 512, 16, 16]	2,359,296
BatchNorm2d-2	[-1, 64, 256, 256]	128	BatchNorm2d-55	[-1, 512, 16, 16]	1,024
ReLU-3	[-1, 64, 256, 256]	0	Conv2d-56	[-1, 512, 16, 16]	131,072
MaxPool2d-4	[-1, 64, 128, 128]	0	BatchNorm2d-57	[-1, 512, 16, 16]	1,024
Conv2d-5	[-1, 64, 128, 128]	36,864	ReLU-58	[-1, 512, 16, 16]	0
BatchNorm2d-6	[-1, 64, 128, 128]	128	BasicBlock-59	[-1, 512, 16, 16]	0
ReLU-7	[-1, 64, 128, 128]	0	Conv2d-60	[-1, 512, 16, 16]	2,359,296
Conv2d-8	[-1, 64, 128, 128]	36,864	BatchNorm2d-61	[-1, 512, 16, 16]	1,024
BatchNorm2d-9	[-1, 64, 128, 128]	128	ReLU-62	[-1, 512, 16, 16]	0
ReLU-10	[-1, 64, 128, 128]	0	Conv2d-63	[-1, 512, 16, 16]	2,359,296
BasicBlock-11	[-1, 64, 128, 128]	0	BatchNorm2d-64	[-1, 512, 16, 16]	1,024
Conv2d-12	[-1, 64, 128, 128]	36,864	ReLU-65	[-1, 512, 16, 16]	0
BatchNorm2d-13	[-1, 64, 128, 128]	128	BasicBlock-66	[-1, 512, 16, 16]	0
ReLU-14	[-1, 64, 128, 128]	0	Conv2d-67	[-1, 512, 8, 8]	2,359,296
Conv2d-15	[-1, 64, 128, 128]	36,864	BatchNorm2d-68	[-1, 512, 8, 8]	1,024
BatchNorm2d-16	[-1, 64, 128, 128]	128	ReLU-69	[-1, 512, 8, 8]	0
ReLU-17	[-1, 64, 128, 128]	0	Conv2d-70	[-1, 512, 8, 8]	2,359,296
BasicBlock-18	[-1, 64, 128, 128]	0	BatchNorm2d-71	[-1, 512, 8, 8]	1,024
Conv2d-19	[-1, 128, 64, 64]	73,728	Conv2d-72	[-1, 512, 8, 8]	262,144
BatchNorm2d-20	[-1, 128, 64, 64]	256	BatchNorm2d-73	[-1, 512, 8, 8]	1,024
ReLU-21	[-1, 128, 64, 64]	0	ReLU-74	[-1, 512, 8, 8]	0
Conv2d-22	[-1, 128, 64, 64]	147,456	BasicBlock-75	[-1, 512, 8, 8]	0
BatchNorm2d-23	[-1, 128, 64, 64]	256	Conv2d-76	[-1, 512, 8, 8]	2,359,296
Conv2d-24	[-1, 128, 64, 64]	8,192	BatchNorm2d-77	[-1, 512, 8, 8]	1,024
BatchNorm2d-25	[-1, 128, 64, 64]	256	ReLU-78	[-1, 512, 8, 8]	0
ReLU-26	[-1, 128, 64, 64]	0	Conv2d-79	[-1, 512, 8, 8]	2,359,296
BasicBlock-27	[-1, 128, 64, 64]	0	BatchNorm2d-80	[-1, 512, 8, 8]	1,024
Conv2d-28	[-1, 128, 64, 64]	147,456	ReLU-81	[-1, 512, 8, 8]	0
BatchNorm2d-29	[-1, 128, 64, 64]	256	BasicBlock-82	[-1, 512, 8, 8]	0
ReLU-30	[-1, 128, 64, 64]	0	Conv2d-83	[-1, 30, 8, 8]	15,360
Conv2d-31	[-1, 128, 64, 64]	147,456	BatchNorm2d-84	[-1, 30, 8, 8]	60
BatchNorm2d-32	[-1, 128, 64, 64]	256	ResNet Output-85	[[-1, 128, 64, 64],	0
ReLU-33	[-1, 128, 64, 64]	0		[-1, 256, 32, 32],	
BasicBlock-34	[-1, 128, 64, 64]	0		[-1, 512, 16, 16]]	
Conv2d-35	[-1, 256, 32, 32]	294,912	Conv2d-86	[-1, 27, 16, 16]	13,851
BatchNorm2d-36	[-1, 256, 32, 32]	512	ReLU-87	[-1, 27, 16, 16]	0
ReLU-37	[-1, 256, 32, 32]	0	BatchNorm2d-88	[-1, 27, 16, 16]	54
Conv2d-38	[-1, 256, 32, 32]	589,824	Conv2d-89	[-1, 27, 32, 32]	6,939
BatchNorm2d-39	[-1, 256, 32, 32]	512	ReLU-90	[-1, 27, 32, 32]	0
Conv2d-40	[-1, 256, 32, 32]	32,768	BatchNorm2d-91	[-1, 27, 32, 32]	54
BatchNorm2d-41	[-1, 256, 32, 32]	512	ConvTranspose2d-92	[-1, 27, 32, 32]	11,691
ReLU-42	[-1, 256, 32, 32]	0	ReLU-93	[-1, 27, 32, 32]	0
BasicBlock-43	[-1, 256, 32, 32]	0	BatchNorm2d-94	[-1, 27, 32, 32]	54
Conv2d-44	[-1, 256, 32, 32]	589,824	Conv2d-95	[-1, 27, 64, 64]	3,483
BatchNorm2d-45	[-1, 256, 32, 32]	512	ReLU-96	[-1, 27, 64, 64]	0
ReLU-46	[-1, 256, 32, 32]	0	BatchNorm2d-97	[-1, 27, 64, 64]	54
Conv2d-47	[-1, 256, 32, 32]	589,824	ConvTranspose2d-98	[-1, 27, 64, 64]	11,691
BatchNorm2d-48	[-1, 256, 32, 32]	512	ReLU-99	[-1, 27, 64, 64]	0
ReLU-49	[-1, 256, 32, 32]	0	BatchNorm2d-100	[-1, 27, 64, 64]	54
BasicBlock-50	[-1, 256, 32, 32]	0	ConvTranspose2d-101	[-1, 27, 512, 512]	186,651
Conv2d-51	[-1, 512, 16, 16]	1,179,648	ReLU-102	[-1, 27, 512, 512]	0
BatchNorm2d-52	[-1, 512, 16, 16]	1,024	BatchNorm2d-103	[-1, 27, 512, 512]	54
ReLU-53	[-1, 512, 16, 16]	0			

Table 14: Details about Res50-Unet Architecture

Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #	Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 256, 256]	9,408	Conv2d-79	[-1, 256, 64, 64]	131,072	BatchNorm2d-157	[-1, 512, 16, 16]	1,024
BatchNorm2d-2	[-1, 64, 256, 256]	128	BatchNorm2d-80	[-1, 256, 64, 64]	512	ReLU-158	[-1, 512, 16, 16]	0
ReLU-3	[-1, 64, 256, 256]	0	ReLU-81	[-1, 256, 64, 64]	0	Conv2d-159	[-1, 2048, 16, 16]	1,048,576
MaxPool2d-4	[-1, 64, 128, 128]	0	Conv2d-82	[-1, 256, 32, 32]	589,824	BatchNorm2d-160	[-1, 2048, 16, 16]	4,096
Conv2d-5	[-1, 64, 128, 128]	4,096	BatchNorm2d-83	[-1, 256, 32, 32]	512	ReLU-161	[-1, 2048, 16, 16]	0
BatchNorm2d-6	[-1, 64, 128, 128]	128	ReLU-84	[-1, 256, 32, 32]	0	Bottleneck-162	[-1, 2048, 16, 16]	0
ReLU-7	[-1, 64, 128, 128]	0	Conv2d-85	[-1, 1024, 32, 32]	262,144	Conv2d-163	[-1, 512, 16, 16]	1,048,576
Conv2d-8	[-1, 64, 128, 128]	36,864	BatchNorm2d-86	[-1, 1024, 32, 32]	2,048	BatchNorm2d-164	[-1, 512, 16, 16]	1,024
BatchNorm2d-9	[-1, 64, 128, 128]	128	Conv2d-87	[-1, 1024, 32, 32]	524,288	ReLU-165	[-1, 512, 16, 16]	0
ReLU-10	[-1, 64, 128, 128]	0	BatchNorm2d-88	[-1, 1024, 32, 32]	2,048	Conv2d-166	[-1, 512, 16, 16]	2,359,296
Conv2d-11	[-1, 256, 128, 128]	16,384	ReLU-89	[-1, 1024, 32, 32]	0	BatchNorm2d-167	[-1, 512, 16, 16]	1,024
BatchNorm2d-12	[-1, 256, 128, 128]	512	Bottleneck-90	[-1, 1024, 32, 32]	0	ReLU-168	[-1, 512, 16, 16]	0
Conv2d-13	[-1, 256, 128, 128]	16,384	Conv2d-91	[-1, 256, 32, 32]	262,144	Conv2d-169	[-1, 2048, 16, 16]	1,048,576
BatchNorm2d-14	[-1, 256, 128, 128]	512	BatchNorm2d-92	[-1, 256, 32, 32]	512	BatchNorm2d-170	[-1, 2048, 16, 16]	4,096
ReLU-15	[-1, 256, 128, 128]	0	ReLU-93	[-1, 256, 32, 32]	0	ReLU-171	[-1, 2048, 16, 16]	0
Bottleneck-16	[-1, 256, 128, 128]	0	Conv2d-94	[-1, 256, 32, 32]	589,824	Bottleneck-172	[-1, 2048, 16, 16]	0
Conv2d-17	[-1, 64, 128, 128]	16,384	BatchNorm2d-95	[-1, 256, 32, 32]	512	Conv2d-173	[-1, 2048, 16, 16]	37,750,784
BatchNorm2d-18	[-1, 64, 128, 128]	128	ReLU-96	[-1, 256, 32, 32]	0	BatchNorm2d-174	[-1, 2048, 16, 16]	4,096
ReLU-19	[-1, 64, 128, 128]	0	Conv2d-97	[-1, 1024, 32, 32]	262,144	ReLU-175	[-1, 2048, 16, 16]	0
Conv2d-20	[-1, 64, 128, 128]	36,864	BatchNorm2d-98	[-1, 1024, 32, 32]	2,048	ConvBlock-176	[-1, 2048, 16, 16]	0
BatchNorm2d-21	[-1, 64, 128, 128]	128	ReLU-99	[-1, 1024, 32, 32]	0	Conv2d-177	[-1, 2048, 16, 16]	37,750,784
ReLU-22	[-1, 64, 128, 128]	0	Bottleneck-100	[-1, 1024, 32, 32]	0	BatchNorm2d-178	[-1, 2048, 16, 16]	4,096
Conv2d-23	[-1, 256, 128, 128]	16,384	Conv2d-101	[-1, 256, 32, 32]	262,144	ReLU-179	[-1, 2048, 16, 16]	0
BatchNorm2d-24	[-1, 256, 128, 128]	512	BatchNorm2d-102	[-1, 256, 32, 32]	512	ConvBlock-180	[-1, 2048, 16, 16]	0
ReLU-25	[-1, 256, 128, 128]	0	ReLU-103	[-1, 256, 32, 32]	0	Bridge-181	[-1, 2048, 16, 16]	0
Bottleneck-26	[-1, 256, 128, 128]	0	Conv2d-104	[-1, 256, 32, 32]	589,824	ConvTranspose2d-182	[-1, 1024, 32, 32]	8,389,632
Conv2d-27	[-1, 64, 128, 128]	16,384	BatchNorm2d-105	[-1, 256, 32, 32]	512	Conv2d-183	[-1, 1024, 32, 32]	18,875,392
BatchNorm2d-28	[-1, 64, 128, 128]	128	ReLU-106	[-1, 256, 32, 32]	0	BatchNorm2d-184	[-1, 1024, 32, 32]	2,048
ReLU-29	[-1, 64, 128, 128]	0	Conv2d-107	[-1, 1024, 32, 32]	262,144	ReLU-185	[-1, 1024, 32, 32]	0
Conv2d-30	[-1, 64, 128, 128]	36,864	BatchNorm2d-108	[-1, 1024, 32, 32]	2,048	ConvBlock-186	[-1, 1024, 32, 32]	0
BatchNorm2d-31	[-1, 64, 128, 128]	128	ReLU-109	[-1, 1024, 32, 32]	0	Conv2d-187	[-1, 1024, 32, 32]	9,438,208
ReLU-32	[-1, 64, 128, 128]	0	Bottleneck-110	[-1, 1024, 32, 32]	0	BatchNorm2d-188	[-1, 1024, 32, 32]	2,048
Conv2d-33	[-1, 256, 128, 128]	16,384	Conv2d-111	[-1, 256, 32, 32]	262,144	ReLU-189	[-1, 1024, 32, 32]	0
BatchNorm2d-34	[-1, 256, 128, 128]	512	BatchNorm2d-112	[-1, 256, 32, 32]	512	ConvBlock-190	[-1, 1024, 32, 32]	0
ReLU-35	[-1, 256, 128, 128]	0	ReLU-113	[-1, 256, 32, 32]	0	UpBlockForUNetWithResNet50-191	[-1, 1024, 32, 32]	0
Bottleneck-36	[-1, 256, 128, 128]	0	Conv2d-114	[-1, 256, 32, 32]	589,824	ConvTranspose2d-192	[-1, 512, 64, 64]	2,097,664
Conv2d-37	[-1, 128, 128, 128]	32,768	BatchNorm2d-115	[-1, 256, 32, 32]	512	Conv2d-193	[-1, 512, 64, 64]	4,719,104
BatchNorm2d-38	[-1, 128, 128, 128]	256	ReLU-116	[-1, 256, 32, 32]	0	BatchNorm2d-194	[-1, 512, 64, 64]	1,024
ReLU-39	[-1, 128, 128, 128]	0	Conv2d-117	[-1, 1024, 32, 32]	262,144	ReLU-195	[-1, 512, 64, 64]	0
Conv2d-40	[-1, 128, 64, 64]	147,456	BatchNorm2d-118	[-1, 1024, 32, 32]	2,048	ConvBlock-196	[-1, 512, 64, 64]	0
BatchNorm2d-41	[-1, 128, 64, 64]	256	ReLU-119	[-1, 1024, 32, 32]	0	Conv2d-197	[-1, 512, 64, 64]	2,359,808
ReLU-42	[-1, 128, 64, 64]	0	Bottleneck-120	[-1, 1024, 32, 32]	0	BatchNorm2d-198	[-1, 512, 64, 64]	1,024
Conv2d-43	[-1, 512, 64, 64]	65,536	Conv2d-121	[-1, 256, 32, 32]	262,144	ReLU-199	[-1, 512, 64, 64]	0
BatchNorm2d-44	[-1, 512, 64, 64]	1,024	BatchNorm2d-122	[-1, 256, 32, 32]	512	ConvBlock-200	[-1, 512, 64, 64]	0
Conv2d-45	[-1, 512, 64, 64]	131,072	ReLU-123	[-1, 256, 32, 32]	0	UpBlockForUNetWithResNet50-201	[-1, 512, 64, 64]	0
BatchNorm2d-46	[-1, 512, 64, 64]	1,024	Conv2d-124	[-1, 256, 32, 32]	589,824	ConvTranspose2d-202	[-1, 256, 128, 128]	524,544
ReLU-47	[-1, 512, 64, 64]	0	BatchNorm2d-125	[-1, 256, 32, 32]	512	Conv2d-203	[-1, 256, 128, 128]	1,179,904
Bottleneck-48	[-1, 512, 64, 64]	0	ReLU-126	[-1, 256, 32, 32]	0	BatchNorm2d-204	[-1, 256, 128, 128]	512
Conv2d-49	[-1, 128, 64, 64]	65,536	Conv2d-127	[-1, 1024, 32, 32]	262,144	ReLU-205	[-1, 256, 128, 128]	0
BatchNorm2d-50	[-1, 128, 64, 64]	256	BatchNorm2d-128	[-1, 1024, 32, 32]	2,048	ConvBlock-206	[-1, 256, 128, 128]	0
ReLU-51	[-1, 128, 64, 64]	0	ReLU-129	[-1, 1024, 32, 32]	0	Conv2d-207	[-1, 256, 128, 128]	590,080
Conv2d-52	[-1, 128, 64, 64]	147,456	Bottleneck-130	[-1, 1024, 32, 32]	0	BatchNorm2d-208	[-1, 256, 128, 128]	512
BatchNorm2d-53	[-1, 128, 64, 64]	256	Conv2d-131	[-1, 256, 32, 32]	262,144	ReLU-209	[-1, 256, 128, 128]	0
ReLU-54	[-1, 128, 64, 64]	0	BatchNorm2d-132	[-1, 256, 32, 32]	512	ConvBlock-210	[-1, 256, 128, 128]	0
Conv2d-55	[-1, 512, 64, 64]	65,536	ReLU-133	[-1, 256, 32, 32]	0	UpBlockForUNetWithResNet50-211	[-1, 256, 128, 128]	0
BatchNorm2d-56	[-1, 512, 64, 64]	1,024	Conv2d-134	[-1, 256, 32, 32]	589,824	ConvTranspose2d-212	[-1, 128, 256, 256]	131,200
ReLU-57	[-1, 512, 64, 64]	0	BatchNorm2d-135	[-1, 256, 32, 32]	512	Conv2d-213	[-1, 128, 256, 256]	221,312
Bottleneck-58	[-1, 512, 64, 64]	0	ReLU-136	[-1, 256, 32, 32]	0	BatchNorm2d-214	[-1, 128, 256, 256]	256
Conv2d-59	[-1, 128, 64, 64]	65,536	Conv2d-137	[-1, 1024, 32, 32]	262,144	ReLU-215	[-1, 128, 256, 256]	0
BatchNorm2d-60	[-1, 128, 64, 64]	256	BatchNorm2d-138	[-1, 1024, 32, 32]	2,048	ConvBlock-216	[-1, 128, 256, 256]	0
ReLU-61	[-1, 128, 64, 64]	0	ReLU-139	[-1, 1024, 32, 32]	0	Conv2d-217	[-1, 128, 256, 256]	147,584
Conv2d-62	[-1, 128, 64, 64]	147,456	Bottleneck-140	[-1, 1024, 32, 32]	0	BatchNorm2d-218	[-1, 128, 256, 256]	256
BatchNorm2d-63	[-1, 128, 64, 64]	256	Conv2d-141	[-1, 512, 32, 32]	524,288	ReLU-219	[-1, 128, 256, 256]	0
ReLU-64	[-1, 128, 64, 64]	0	BatchNorm2d-142	[-1, 512, 32, 32]	1,024	ConvBlock-220	[-1, 128, 256, 256]	0
Conv2d-65	[-1, 512, 64, 64]	65,536	ReLU-143	[-1, 512, 32, 32]	0	UpBlockForUNetWithResNet50-221	[-1, 128, 256, 256]	0
BatchNorm2d-66	[-1, 512, 64, 64]	1,024	Conv2d-144	[-1, 512, 16, 16]	2,359,296	ConvTranspose2d-222	[-1, 64, 512, 512]	32,832
ReLU-67	[-1, 512, 64, 64]	0	BatchNorm2d-145	[-1, 512, 16, 16]	1,024	Conv2d-223	[-1, 64, 512, 512]	38,656
Bottleneck-68	[-1, 512, 64, 64]	0	ReLU-146	[-1, 512, 16, 16]	0	BatchNorm2d-224	[-1, 64, 512, 512]	128
Conv2d-69	[-1, 128, 64, 64]	65,536	Conv2d-147	[-1, 2048, 16, 16]	1,048,576	ReLU-225	[-1, 64, 512, 512]	0
BatchNorm2d-70	[-1, 128, 64, 64]	256	BatchNorm2d-148	[-1, 2048, 16, 16]	4,096	ConvBlock-226	[-1, 64, 512, 512]	0
ReLU-71	[-1, 128, 64, 64]	0	Conv2d-149	[-1, 2048, 16, 16]	2,097,152	Conv2d-227	[-1, 64, 512, 512]	36,928
Conv2d-72	[-1, 128, 64, 64]	147,456	BatchNorm2d-150	[-1, 2048, 16, 16]	4,096	BatchNorm2d-228	[-1, 64, 512, 512]	128
BatchNorm2d-73	[-1, 128, 64, 64]	256	ReLU-151	[-1, 2048, 16, 16]	0	ReLU-229	[-1, 64, 512, 512]	0
ReLU-74	[-1, 128, 64, 64]	0	Bottleneck-152	[-1, 2048, 16, 16]	0	ConvBlock-230	[-1, 64, 512, 512]	0
Conv2d-75	[-1, 512, 64, 64]	65,536	Conv2d-153	[-1, 512, 16, 16]	1,048,576	UpBlockForUNetWithResNet50-231	[-1, 64, 512, 512]	0
BatchNorm2d-76	[-1, 512, 64, 64]	1,024	BatchNorm2d-154	[-1, 512, 16, 16]	1,024	Conv2d-232	[-1, 27, 512, 512]	1,755
ReLU-77	[-1, 512, 64, 64]	0	ReLU-155	[-1, 512, 16, 16]	0			
Bottleneck-78	[-1, 512, 64, 64]	0	Conv2d-156	[-1, 512, 16, 16]	2,359,296			