UNIVERSITY OF LIVERPOOL

University of Liverpool, Dept of Computer Science

Applied AI: CA2

Rhys Cooper[1] and Thomas Williams[2]

[1]Student ID: 201680340; Email: sgrcoop2@liverpool.ac.uk
[2]Student ID: 201450922; Email: sgtwill6@liverpool.ac.uk

May 2023

Abstract

In this report, we focus on the task of semantic segmentation using the Cam101 dataset. We compare the performance of three popular pretrained models: DeepLabV3, FCN (Fully Convolutional Network), and LRASPP (Lite R-ASPP).

1 Introduction

The Cam101 dataset is a collection of 101 images capturing everyday driving scenes, each with dimensions of 960x720 pixels and formatted as uncompressed 24-bit colour PNG. Each image has a corresponding true segmented mask, in which each pixel's RGB value (between 0 and 255) corresponds to one of the 32 object classes commonly found in a driving scenario. These classes encompass a diverse range of objects, including buildings, pedestrians, vehicles, and road signs, posing significant challenges for accurate and fine-grained segmentation.

To achieve accurate and efficient segmentation results, we employ three pretrained models: DeepLabV3, FCN, and LRASPP. DeepLabV3 is a state-of-the-art model that utilizes atrous convolution and dilated convolutions to capture multi-scale contextual information effectively. FCN is another widely used model that employs fully convolutional layers to enable end-to-end segmentation. Lastly, LRASPP is a lightweight variant of the popular R-ASPP (Residual Atrous Spatial Pyramid Pooling) architecture, known for its excellent trade-off between accuracy and computational efficiency.

1.1 DeepLab-V3 Model Structure

The encoding component of the model comprises an initial 'Conv2d' layer, in order to extract low level features, which is then followed by max-pooling layers that down-sample the feature map to successively smaller dimensions. After this, bottleneck structures within the sequential layers, with narrow inputs and outputs, employ the use of atrous convolution. Atrous convolution remedies the issue of compromised accuracy on dense prediction tasks that arises from the hierarchical feature extraction inherit in DCNNs (Chen et al). The larger the atrous rate, the larger the model's field of view and the more information can be extracted. For this model,the atrous rate gradually doubles as it passes through the atrous convolution layers of the model (ibid).

The output from the atrous convolution layers is passed into the atrous spatial pyramid pooling (ASPP) from the the previous DeepLab-V2 model. The purpose of this component is to aggregate the features extracted at different levels into composite feature maps. The final part of the 'DeepLabHead' translates the feature maps into a final semantic segmentation mask.

This model is constructed by importing a DeepLab-V3 model with a ResNet-101 backbone and ResNet-101 pre-trained weights that have been trained on a subset of COCO 2017 dataset. We then modify this preexisting model for this particular semantic segmentation task by changing the last layer of the model's classifier and auxiliary classifier heads to be a 2D convolutional layer, but with an output shape equal to the number of classes in the Cam101 dataset. Following this, we add a softmax layer as the last layer of the auxiliary classifier head, and then we freeze the layers of the model other than the classifier heads, as shown in Table 6 in the Appendix with True/False in the 'Trainable' column.

1.2 FCN Model Structure

The core premise of the FCN is taking image classification nets like AlexNet and modifying the fully connected layers to be fully convolutional layers, hence allowing for semantic segmentation.

Similar to DeepLab-V3, the initial layers of the model involve a 'Conv2d' layer that extracts

low level features, which is then passed to a max-pooling layer that down samples the feature map to a smaller dimension. Each sequential layer contains exclusively sub-sampling, convolution, and up-sampling, which allow the layers to perform pixel wise prediction. The feature maps extracted at different levels are combined using the 'skip architecture' (Shelhamer et al, 2016). This involves connecting the low level features extracted during the initial layers with the higher level features extracted during the latter levels.

This model is constructed by importing an FCN model, and as with DeepLabV3, importing it with a Resnet-101 backbone and Resnet-101 pre-trained weights that have been trained on a subset of COCO train2017. We then modify this preexisting model for this particular semantic segmentation task by changing the last layer of the model's classifier and auxiliary classifier heads to be a 2D convolutional layer, but with an output shape equal to the number of classes. The softmax layer of the classifier head is again located in the auxiliary classifier head, and then the layers of the model other than the classifier heads are frozen, as shown in Table 7 in the Appendix with True/False in the 'Trainable' column.

1.3 LRASPP Model Structure

After Conv2d for low level feature extraction, the last layer is a Hardswish layer, not present in the other models, which is used to replace ReLU as a non-linear activation function which drastically improves the accuracy (Howard et al, 2019). Following this, the inverted residual layers involve squeeze and excite bottlenecks of fixed size, with atrous pooling applied only to the last block (ibid). In comparison to the ASPP present in the DeepLab classifier head, this model uses Lite-Reduced ASPP which involves global-average pooling. With this LRASPPHead, the sequential 'Cbr' stands for convolution, batch, ReLU, whilst the sequential 'scale' uses AdaptiveAvgPool2d to stitch together feature maps of different scales.

This model is constructed by importing an LRASPP model with a MobileNetV3 backbone and MobileNetV3 'Large' pre-trained weights. As with the previous models, we change the last layer of the model's classifier and auxiliary classifier heads to be a 2D convolutional layer, but with an output shape equal to the number of classes. The softmax layer of the classifier head is again located in the auxiliary classifier head, and then the layers of the model other than the classifier heads are frozen, as shown in Table 8 in the Appendix with True/False in the 'Trainable' column.

2 Design Rationale

Our overarching approach is to use 'base-level' optimiser and learning rate schedulers (based on generally accepted values) to first explore unfreezing various layers of the pre-trained model to find the optimal amount of 'fine-tuning' required. The rationale behind this is that the higher level feature extraction that occurs within the final layers of the pre-existing model can have their parameters retrained in order to become specific to the 32 classes of this semantic segmentation task. After this, we implement hyperparameter tuning on the optimisers and classifiers to use these in models going forward. Finally, we explore the best combination of 3 optimisers (SGD, Adam, and RMSProp), the results of which are shown in Table 4. Please note that all accuracies in this section correspond to the 'Dice' evaluation metric, as opposed to IoU and PixAcc. We chose this because after comparing early predicted masks to both PixAcc and IoU accuracy scores, we felt the Dice metric gave a more fair and balanced idea of how successful the prediction was. However, although we

kept this consistent throughout hyperparameter tuning, we switched to just evaluating the models using IoU and PixAcc in the final testing stage.

2.1 Hyperparameter Search: Depth Of Trainable Layers

Tables 1, 2, and 3 below show the results of our fine-tuning exploration. We can see that going 'too deep' into the model overwrites the benefit of using pre-trained weights, since it causes the model to forget previously learned representations which are useful to the current task. Furthermore, unfreezing more layers introduces more trainable parameters that require updating during backpropagation, and as such comes with the added cost of longer training times. The optimal number of unfrozen layers is shown in bold for each model.

Trainable Backbone Layers	Validation Accuracy	Validation Loss
4	**64.57**%	**37.50**
3, 4	64.98 %	42.58

Table 1: DeepLab-V3

Trainable Backbone Layers	Validation Accuracy	Validation Loss
4	**64.31** %	**30.39**
3, 4	57.35%	34.16

Table 2: FCN

Trainable Inverted residual backbone Layers	Validation Accuracy	Validation Loss
16	41.65 %	57.67
15, 16	49.64%	41.18
14, 15, 16	**58.23%**	**36.76**
13, 14, 15, 16	56.65 %	33.15

Table 3: LRASPP

2.2 Hyperparameter Search

We now discuss the various hyperparameters that were explored during the hyperparameter optimisation stage, including optimisers, learning rate schedulers, and batch size. We discuss the impact each change has on overall model accuracy as well as on under/over-fitting.

Optimiser: We found the different optimisers resulted in a significant difference of reported accuracy and loss across all models. We tuned some of the hyperparameters within each optimiser, for example gamma and learning rate, and found that the optimum values lied within generally accepted ranges, settling on `lr=0.0001`, `weight_decay=0.001`, and `momentum=0.9`, among others. The results for different optimisers across different models are shown in Table 4. Using optimisers with weight decay acts as an added regularisation term, which we found improved model performance across all models and resulted in increased accuracy by reducing overfitting, since it adds a penalty term to the loss function based on the magnitudes of the model's weights.

Learning Rate and Step Size: For DeepLabV3, we experimented with learning rate sizes of increasing orders of magnitude whilst the step rate of the scheduler was kept constant at 4.

The optimal learning rate hyperparameter was 0.0001 which produced a validation accuracy of 86.51% , whereas 0.001 scored 58.94% and 0.01 scored 22.69%. An interpretation for this is that a small learning rate is needed to reach a global loss minima, whilst the larger learning rate overshoots the global minima. This was the same case for the FCN with the optimal learning rate hyperparameter value of 0.0001 producing an accuracy of 84.34 %. However, for the LRASPP model, the optimal learning rate was 0.001 and had to be coupled with a decreased step rate of 3 to give an accuracy of 36.59%.

Batchsize: In general, we aimed for a reasonably large batch size as it was thought that introducing a more diverse set of image samples to perform a forwards and backwards pass on would allow the model to learn from a more general and representative sample, and that the gradient found from a larger batch size would also be closer to the true gradient, thus allowing for faster optimisation. Furthermore, larger batch sizes can slightly improve training time, but at the cost of computational resources – we found that increasing the batch size too far caused a RAM overload. However, smaller batch sizes make the gradient update process more stochastic, as the model updates its parameters more frequently and thus cannot learn to overfit on the noise of any particular image; in this way it acts as a form of regularisation to prevent overfitting. Through our experimentation we found the best balance of batch size to be between 5-10% of the size of the dataloader.

Overfitting Presence and Mitigation: The image augmentation stage is a method used to prevent overfitting. The increased dataset size and the added transformations lead to variation and noise in training images, which provides the model with increased diversity that allows it to learn more general features and thus limit overfitting. Specifically, we included horizontal flip and rotate to allow for the models to learn features independent of their orientation, and a random saturation to help the model learn visual representations independent of the natural level of light (since all Cam101 images are during the same slightly cloudy day). Examples of original and augmented images are shown in Figure 2 and Figure 3 of the Appendix. Ideally, we would have used over 200 augmented images, but were limited due to RAM constraints to only 100.

In addition to this, all three models include a batch normalisation layer within the backbone with both improves training speed and introduces some noise to the networks by adding small variations to the activations, which helps prevent overfitting. However, one downside is that the models are unable to process batch sizes of 1 due to this layer. In order to prevent an error, we added a function in the `__init__` of the `TrainTest()` class which verifies if the combination of dataset size, number of k-folds, and input batch size will result in a final batch size of 1 being taken from the dataloader during training, amending the batch size if this is the case.

Finally, by printing out the training and validation loss/accuracies across each epoch, we aimed to track when the model began to overfit on the training data by noting when the graph of validation accuracy began to decrease despite continued improvement in the graph of training accuracy. This regularisation technique, 'Early Stopping', prevents overfitting by not training beyond this number of epochs. However, we were limited due to time and computational constraints in regard to how many epochs we could use, and as such never saw a divergence of validation and training accuracy that would indicate the need to implement Early Stopping.

Table 4 below presents the average validation accuracy and loss for different optimisers for the 3 models. Each optimiser uses the best hyperparameters just discussed. The optimiser

that produces the highest validation accuracy is highlighted, and is subsequently selected to be used for each model in the final testing stage.

Model		Validation Accuracy (%)	Validation Loss
DeepLabV3	SGD	6.61	3.344
	Adam	87.90	1.230
	RMSProp	37.01	0.355
FCN	SGD	35.32	3.156
	Adam	87.10	0.932
	RMSProp	28.07	0.351
LRASPP	SGD	25.46	2.07
	Adam	52.45	0.365
	RMSProp	23.71	0.304

Table 4: Comparison of Accuracy and Loss for Different Models and Optimisers

3 Model Performance Analysis

3.1 Accuracy and Loss on Training and Test Sets

Model	Test IoU Accuracy	Test Pixel Accuracy	Train Loss	Validation Loss	Test Loss
DeepLab-v3	13.15%	88.28 %	1.257	1.236	1.223
FCN	18.28%	97.49 %	0.924	0.910	0.902
LRASPP	8.55%	89.42 %	0.385	0.368	0.354

Table 5: Evaluation metrics and loss of best models

DeepLab-V3: The pixel accuracy value for training and validation, as shown in Figure 4a) of the Appendix, converges the fastest for the DeepLabV3 model relative to the others. This is likely due to the main benefit of the DeepLabV3 model is that the heavy use of atrous convolution allows for high resolution feature maps that can include large-scale features due to the high spatial boundaries.

However, a downside caused by the use of atrous convolution is that it may face difficulty in segmenting objects that are small and are not perceived well by the models larger receptive field caused by the increasing atrous rate. This is reflected in the predicted mask of Appendix Figure 4c), where in comparison to the prediction maps of the other two models, lacks semantic segmentation ability for smaller objects. The fact this prediction mask predicts the fewest number of segments compared to the other models yet still achieves a high pixel accuracy raises a limitation of pixel accuracy as an evaluation metric: spatial boundaries of objects and partial object segmentation are not taken into account. The IoU evaluation metric overcomes some of these limitations.

FCN: The FCN outperformed the other two models on both evaluation metrics whilst achieving the second highest loss. Inspecting the semantic segmentation success of the predicted masks in Figure 5c) reveals a respectable prediction on par with that of LRASPP model. A benefit of the FCN is that the aforementioned 'skip' architecture operates end-to-end, meaning no peripheral operations are required to arrive at a segmentation map, in

addition to this structure allowing for the concatenated features to not have to be stored in the memory (Shelhamer, Long and Darrell, 2017), thus reducing memory requirements. However, one limitation is the the lack of down pooling within the FCN's operations, which means it is computationally demanding, especially if the original image is of high resolution.

LRASPP: The LRASPP has the second highest Pixel accuracy and the lowest loss, as shown in Table 5. A visual inspection of the semantic segmentation success of the predicted mask in Figure 6c) in the Appendix shows good performance – producing significantly more segments than the DeepLabV3 – despite the model achieving the worst IoU accuracy.

In general, a benefit of the LRASPP model is that the MobileNetV3 provides the computational efficiency to allow it to be deployed on user mobile devices.; this efficiency is made possible by the use of a Lite-reduced atrous spatial pyramid pooling component. However, the LRASPP does not have as many trainable parameters compared to similar models, and although this makes a noticeable improvement in training time, it creates a limit on the complexity that it can learn. By passing the parameter `verbose=True` when defining each model class, a `torchmetrics` summary will be printed out which displays the total number of parameters in the model, alongside the enabled trainable parameters. In comparison to the above models, DeepLabV3 has 61 million total parameters, FCN has 54 million, and LRASPP just 3 million.

3.2 Loss Function and Predicted Mask Analysis

Throughout our hyperparameter tuning we used just one loss function, CrossEntropyLoss, to maintain a fair comparison and avoid shifting goalposts. The loss function seemed appropriate to the task at hand, and was easy to implement. However, after testing the best version of each model, we were somewhat disappointed at the quality of the final predicted masks. Upon inspection, it is clear that none of the models do particularly well in creating a mask with 'fine-grained' segmentation, and are not able to capture the smaller class objects such as road signs, road markings, or (often) even bicycles. We note the validation and testing loss for all models is relatively small, as shown in the preceding Table 5, which generally indicates successful model performance; however, we realised that perhaps there is a problem with the loss function itself.

Given that the dataset contains a small number of sequential images down the same road, many of the dataset classes are not actually observed (e.g. 'Animal', 'Bridge', 'Motorcycle-Scooter'), and, on the flip side, a massive proportion of pixel instances are dominated by a handful of classes, such as 'Road' and 'Sky'. This means that the dataset is highly unbalanced, so when the loss function calculates the loss with an equal weighting to all classes, the model is not incentivised to improve mask predictions for smaller classes, since it is doing so well on the majority of pixel instances. This explains why the loss for all models is so low despite subpar performance – especially given that we were using pre-trained models. After research, we found that in order to circumvent this issue we must assign a list of `class_weights` to the loss function, such that it penalises a loss of non-frequent classes with higher severity. This approach essentially prevents the model from underfitting – it allows the model to be trained in such a way that it is able to learn the smaller complexities of all the different classes in the dataset, instead of only training on the high-frequency classes.

However, we found that there although the generally accepted approach to calculating the class weights is to take the inverse of the frequency of each class in the dataset, there are multiple ways of doing this, with no clear optimal solution. This meant that adjusting the

class weights amounted to an entirely new hyperparameter, and we found that it became one of, if not the, most significant hyperparameter of all. Multiple approaches to calculating the class weights were used, including the following code snippet (here, `classes_count` is a list of length 32 corresponding to the number of pixels in the training dataset belonging to each class), as well as a Laplace Smoothing approach not shown. Further, the variables X and Y can be varied and result in wildly different final accuracy and losses.

```
X, Y = 100, 50
# 1. Either this:
class_weights = torch.where(classes_count > 0,  X - 100*(classes_count /
    sum(classes_count).item()), Y)

# 3. Or this:
class_weights = torch.where(classes_count > 0,  (X*max(classes_count)/sum(
    classes_count).item()) - Y*(classes_count / sum(classes_count).item()),
     (X*max(classes_count)/sum(classes_count).item()))
```

Listing 1: class_weights function

To our dismay, the reported accuracies and losses were not better than previously found during hyperparameter exploration. However, many of the predicted masks showed a significant improvement in quality, despite lower accuracies. This makes it difficult to find the best `class_weights`, since in order to evaluate each list of class weights, an entire model needs to be trained and tested and predicted mask visually compared. Nonetheless, after some experimentation, we found an improved mask by using the LRASPP model with RMSprop optimiser and the second `class_weights` function shown in the code. This model scored a final testing IoU Accuracy of 4.57%, final Pixel Accuracy of 73.06%, and final loss of 3.921. However, the final predicted mask seems to show significant improvement on the other models – it is able to capture finer details such as road markings, close and far away traffic lights, a hedge, and even a very distant (albeit somewhat smudged) bus! The original image, true mask, and predicted mask is show in Figure 1 below.

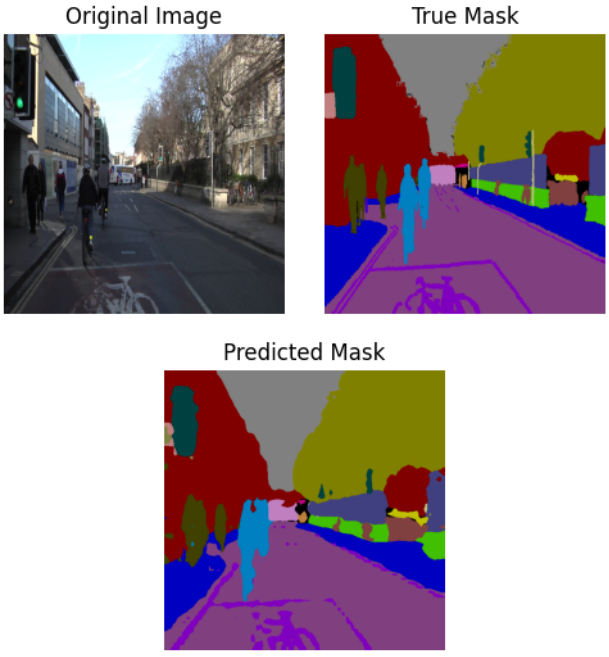


Figure 1: Original image and mask, and predicted mask from a LRASPP model trained using weighted loss

4 Concluding remarks

Upon reflection, we feel confident with importing pre-trained models and modifying them for future computer vision segmentation tasks. Our knowledge and skills with parts of the PyTorch library have developed. We also found reading the original papers of the models interesting since they were cutting edge developments in the field of computer vision.

We were proud of our ability to create our own modified weighted cross entropy loss function. If done again, we would like to expand on the exploration of weighted loss functions by more systematically exploring the optimal function to calculate `class_weights`, and would like to further develop our programming knowledge and skills by attempting to implement our own Dice Coefficient as a loss function. We would also be curious to see how different versions of the same type of model would compare, for example DeepLab-V2 vs DeepLab-V3. A final avenue of potential improvement would be buying more RAM and seeing how the use of more augmented images and larger batch sizes is able to improve predicted mask quality.

5 Reference List

Chen, L.-C., Papandreou, G., Schroff, F. and Adam, H. (2017). Rethinking Atrous Convolution for Semantic Image Segmentation. CoRR, [online] abs/1706.05587. Available at: http://arxiv.org/abs/1706.05587.

Howard, A., Sandler , M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q.V. and Adam, H. (2019). Searching for MobileNetV3. CoRR, [online] abs/1905.02244. Available at: http://arxiv.org/abs/1905.02244.

Shelhamer, E., Long, J. and Darrell, T. (2017). Fully Convolutional Networks for Semantic Segmentation. IEEE transactions on pattern analysis and machine intelligence, [online] 39(4), pp.640–651. doi:https://doi.org/10.1109/TPAMI.2016.2572683.

6 Appendix

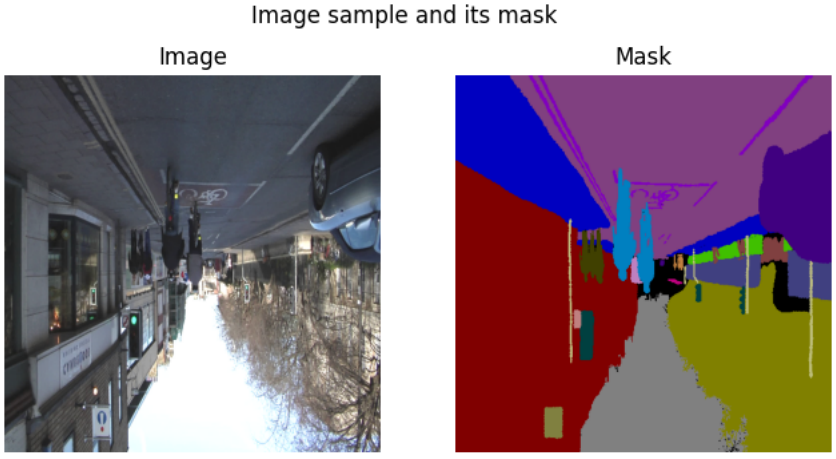


Figure 2: Augmented image from `test_set`, brightness increased, rotated and vertically flipped

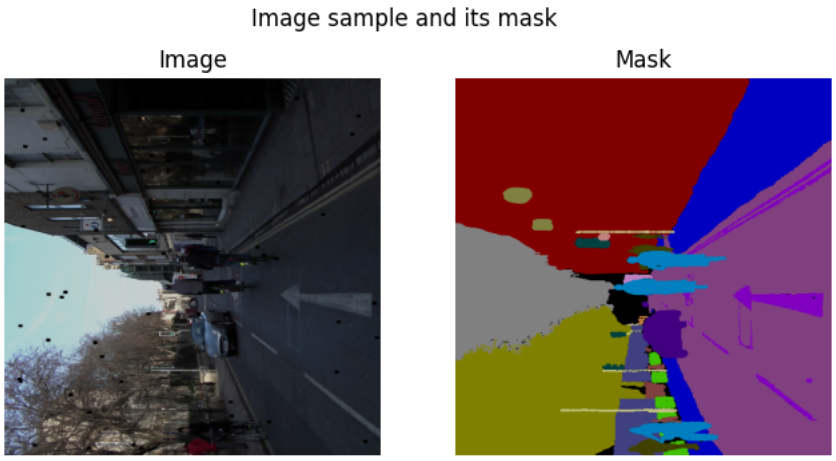


Figure 3: Augmented image from `test_set`, saturation decreased, rotated, and with added noise

Layer	Input Shape	Output Shape	bottlenecks	Trainable
DeepLabV3 (DeepLabV3)	[10, 3, 520, 520]	[10, 32, 520, 520]		Partial
| IntermediateLayerGetter (backbone)	[10, 3, 520, 520]	[10, 2048, 65, 65]	–	False
| Conv2d (conv1)	[10, 3, 520, 520]	[10, 64, 260, 260]	–	False
| BatchNorm2d (bn1)	[10, 64, 260, 260]	[10, 64, 260, 260]	–	False
| ReLU (relu)	[10, 64, 260, 260]	[10, 64, 260, 260]	–	–
| MaxPool2d (maxpool)	[10, 64, 260, 260]	[10, 64, 130, 130]	–	–
| Sequential (layer1)	[10, 64, 130, 130]	[10, 256, 130, 130]	3	False
| Sequential (layer2)	[10, 256, 130, 130]	[10, 512, 65, 65]	4	False
| Sequential (layer3)	[10, 512, 65, 65]	[10, 1024, 65, 65]	18	False
| Sequential (layer4)	[10, 1024, 65, 65]	[10, 2048, 65, 65]	3	False
| DeepLabHead (classifier)	[10, 2048, 65, 65]	[10, 32, 65, 65]	–	True
| ASPP (0)	[10, 2048, 65, 65]	[10, 256, 65, 65]	–	True
| ModuleList (convs)	–	–	–	True
| Sequential (project)	[10, 1280, 65, 65]	[10, 256, 65, 65]	–	True
| Conv2d (1)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| BatchNorm2d (2)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Softmax	[10, 32, 65, 65]	[10, 32, 65, 65]	–	True
| FCNHead (aux classifier)	[10, 1024, 65, 65]	[10, 32, 65, 65]	–	True
| Conv2d (0)	[10, 1024, 65, 65]	[10, 256, 65, 65]	–	True
| BatchNorm2d (1)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| ReLU (2)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Dropout (3)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Conv2d (4)	[10, 256, 65, 65]	[10, 32, 65, 65]	–	True
| Softmax	[10, 32, 65, 65]	[10, 32, 65, 65]	–	True

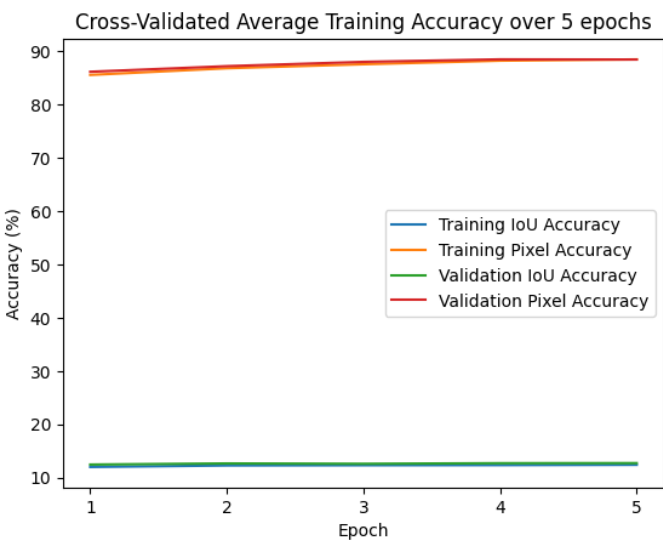
Table 6: DeepLabV3 Model Structure

Layer	Input Shape	Output Shape	bottlenecks	Trainable
FCN	[10, 3, 520, 520]	[10, 32, 520, 520]		Partial
| IntermediateLayerGetter (backbone)	[10, 3, 520, 520]	[10, 2048, 65, 65]	–	False
| Conv2d (conv1)	[10, 3, 520, 520]	[10, 64, 260, 260]	–	False
| BatchNorm2d (bn1)	[10, 64, 260, 260]	[10, 64, 260, 260]	–	False
| ReLU (relu)	[10, 64, 260, 260]	[10, 64, 260, 260]	–	–
| MaxPool2d (maxpool)	[10, 64, 260, 260]	[10, 64, 130, 130]	–	–
| Sequential (layer1)	[10, 64, 130, 130]	[10, 256, 130, 130]	3	False
| Sequential (layer2)	[10, 256, 130, 130]	[10, 512, 65, 65]	4	False
| Sequential (layer3)	[10, 512, 65, 65]	[10, 1024, 65, 65]	23	False
| Sequential (layer4)	[10, 1024, 65, 65]	[10, 2048, 65, 65]	3	False
| FCNHead (classifier)	[10, 1024, 65, 65]	[10, 32, 65, 65]	–	True
| Conv2d (0)	[10, 1024, 65, 65]	[10, 256, 65, 65]	–	True
| BatchNorm2d (1)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| ReLU (2)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Dropout (3)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Conv2d (4)	[10, 256, 65, 65]	[10, 32, 65, 65]	–	True
| Softmax	[10, 32, 65, 65]	[10, 32, 65, 65]	–	True
| FCNHead (aux classifier)	[10, 1024, 65, 65]	[10, 32, 65, 65]	–	True
| Conv2d (0)	[10, 1024, 65, 65]	[10, 256, 65, 65]	–	True
| BatchNorm2d (1)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| ReLU (2)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Dropout (3)	[10, 256, 65, 65]	[10, 256, 65, 65]	–	True
| Conv2d (4)	[10, 256, 65, 65]	[10, 32, 65, 65]	–	True
| Softmax	[10, 32, 65, 65]	[10, 32, 65, 65]	–	True

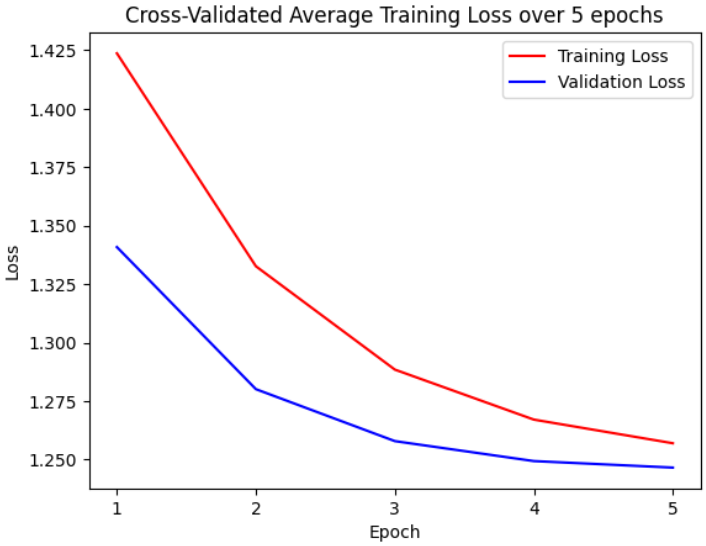
Table 7: FCN Model Structure

Layer	Input Shape	Output Shape		Trainable
LRASPP	[10, 3, 520, 520]	[10, 32, 520, 520]		Partial
| IntermediateLayerGetter (backbone)	[10, 3, 520, 520]	[10, 960, 33, 33]	–	False
| Conv2dNormActivation (0)	[10, 3, 520, 520]	[10, 16, 260, 260]	–	False
| Conv2d (0)	[10, 3, 520, 520]	[10, 16, 260, 260]	–	True
| BatchNorm2d (1)	[10, 16, 260, 260]	[10, 16, 260, 260]	–	True
| Hardswish (2)	[10, 16, 260, 260]	[10, 16, 260, 260]	–	True
| InvertedResidual (1)	[10, 16, 260, 260]	[10, 16, 260, 260]	–	False
| InvertedResidual (2)	[10, 16, 260, 260]	[10, 24, 130, 130]	–	False
| InvertedResidual (3)	[[10, 24, 130, 130]	[10, 24, 130, 130]	–	False
| InvertedResidual (4)	[10, 24, 130, 130]	[10, 40, 65, 65]	–	False
| InvertedResidual (5)	[10, 40, 65, 65]	[10, 40, 65, 65]	–	False
| InvertedResidual (6)	[10, 40, 65, 65]	[10, 40, 65, 65]	–	False
| InvertedResidual (7)	[10, 40, 65, 65]	[10, 80, 33, 33]	–	False
| InvertedResidual (8)	[10, 80, 33, 33]	[10, 80, 33, 33]	–	False
| InvertedResidual (9)	[10, 80, 33, 33]	[10, 80, 33, 33]	–	False
| InvertedResidual (10)	[10, 80, 33, 33]	[10, 80, 33, 33]	–	False
| InvertedResidual (11)	[10, 80, 33, 33]	[10, 112, 33, 33]	–	False
| InvertedResidual (12)	[10, 112, 33, 33]	[10, 112, 33, 33]	–	False
| InvertedResidual (13)	[10, 112, 33, 33]	[10, 112, 33, 33]	–	False
| InvertedResidual (14)	[10, 160, 33, 33]	[10, 160, 33, 33]	–	False
| InvertedResidual (15)	[10, 160, 33, 33]	[10, 160, 33, 33]	–	False
| Conv2dNormActivation (16)	[10, 160, 33, 33]	[10, 960, 33, 33]	–	False
| Conv2d (0)	[10, 160, 33, 33]	[10, 960, 33, 33]	–	True
| BatchNorm2d (1)	[10, 960, 33, 33]	[10, 960, 33, 33]	–	True
| Hardswish (2)	[10, 960, 33, 33]	10, 960, 33, 33	–	True
| LRASPHead (classifier)	[10, 40, 65, 65]	[10, 32, 65, 65]	–	True
| Sequential (cbr)	[10, 960, 33, 33]	[10, 128, 33, 33]	–	True
| Conv2d (0)	[10, 960, 33, 33]	[10, 128, 33, 33]	–	True
| BatchNorm2d (1)	[10, 128, 33, 33]	[10, 128, 33, 33]	–	True
| ReLU (2)	[10, 128, 33, 33]	[10, 128, 33, 33]	–	True
| Sequential (scale)	[10, 960, 33, 33]	[10, 128, 1, 1]	–	True
| AdaptiveAvgPool2d (0)	[10, 960, 33, 33]	[10, 960, 1, 1]	–	True
| Conv2d (1)	[10, 960, 1, 1]	[10, 128, 1, 1]	–	True
| Sigmoid (2)	[10, 128, 1, 1]	[10, 128, 1, 1]	–	True
| Conv2d (high classifier)	[10, 40, 65, 65]	[10, 32, 65, 65]	–	True
| Conv2d (low classifier)	[10, 128, 65, 65]	[10, 32, 65, 65]	–	True

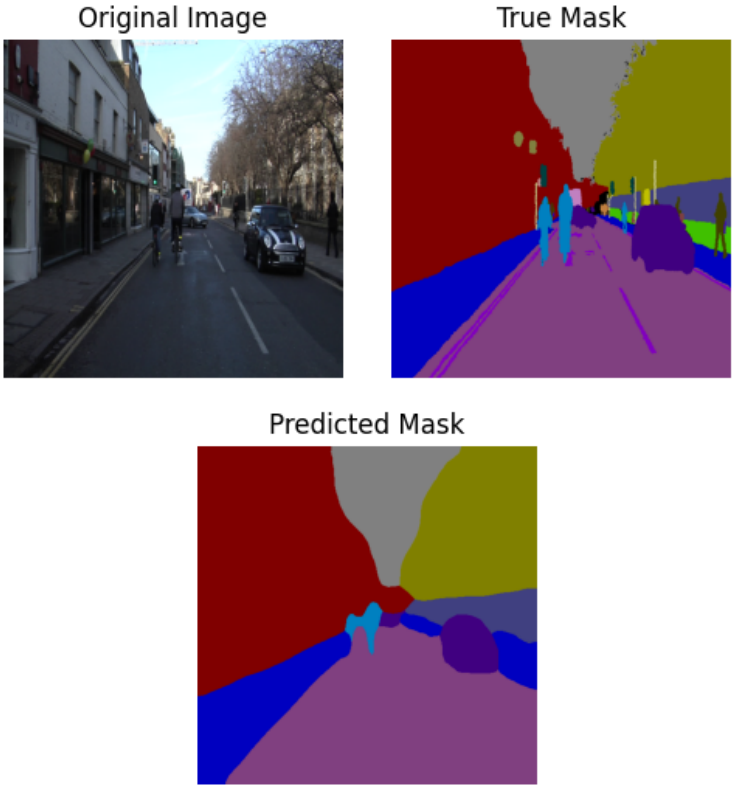
Table 8: LRASPP Model Structure



(a) CV-averaged Training Accuracy Over 5 Epochs

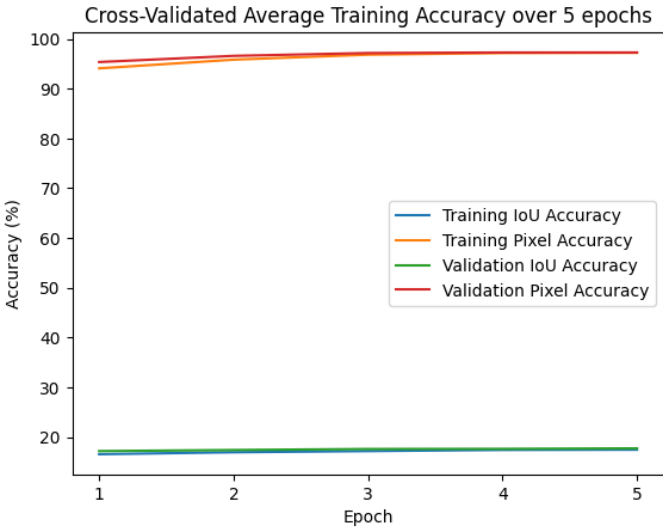


(b) CV-averaged Training Loss Over 5 Epochs

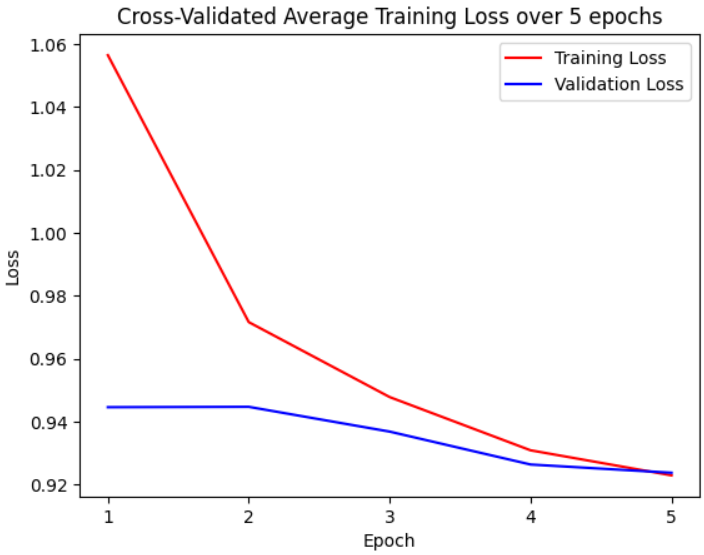


(c) Original image and mask, and a final predicted mask

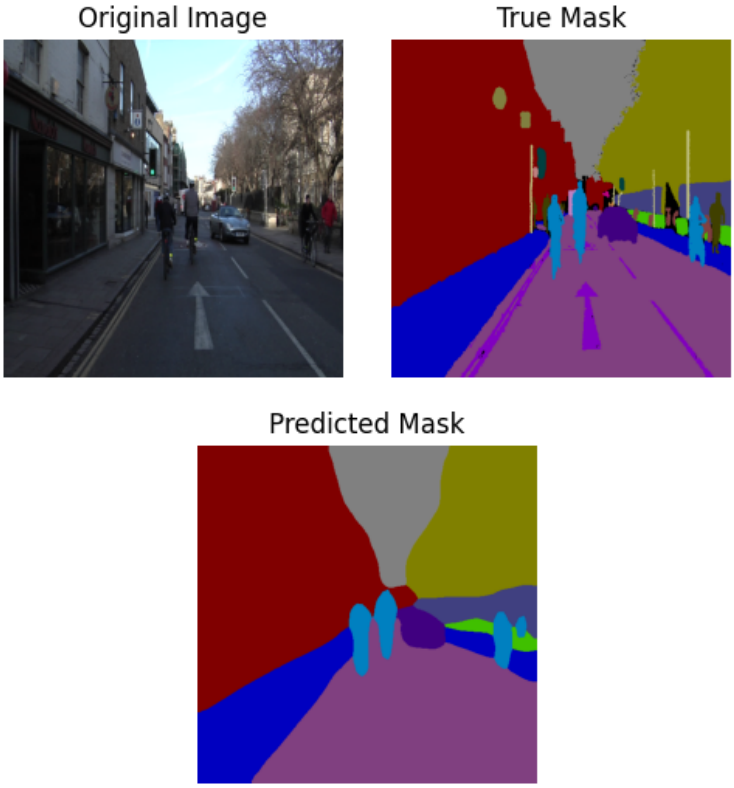
Figure 4: DeepLab-V3 Results



(a) CV-averaged Training Accuracy Over 5 Epochs

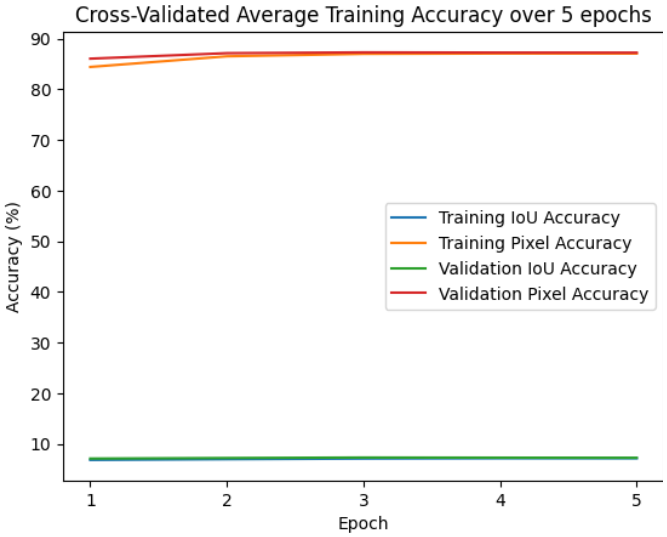


(b) CV-averaged Training Loss Over 5 Epochs

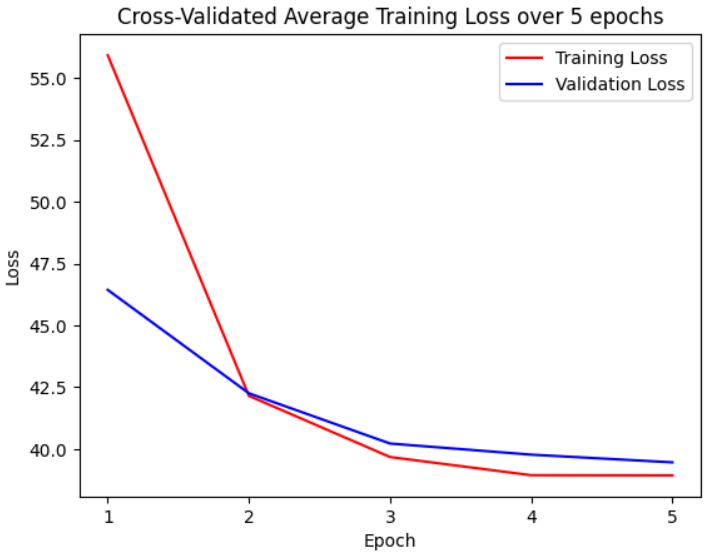


(c) Original image and mask, and a final predicted mask

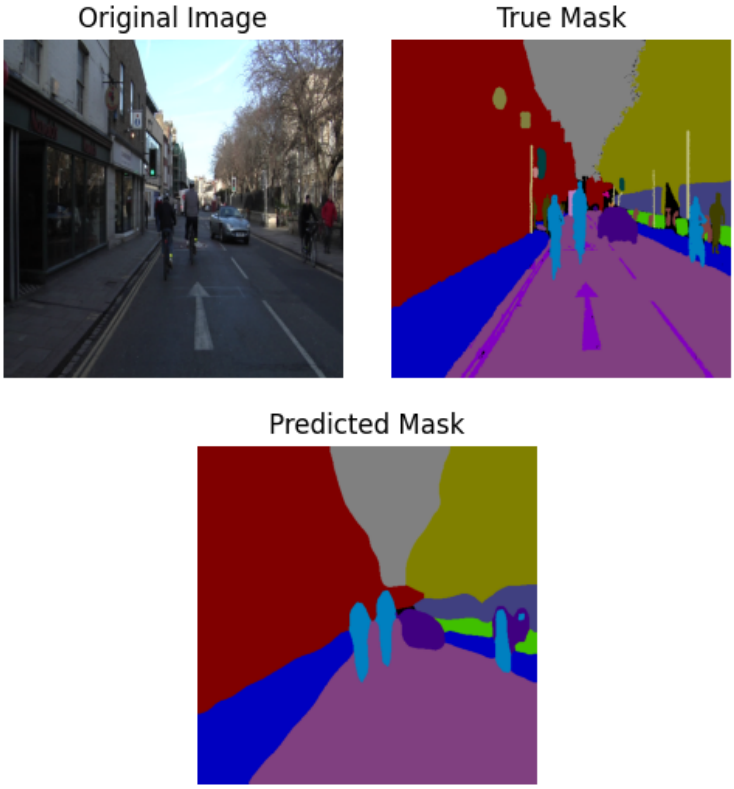
Figure 5: FCN Results



(a) CV-averaged Training Accuracy Over 5 Epochs



(b) CV-averaged Training Loss Over 5 Epochs



(c) Original image and mask, and a final predicted mask

Figure 6: LLRAPP Results