Real Time Semantic Segmentation using Efficient Neural Network

Rishie Raj
Maryland Applied Graduate Engineering
University of Maryland
College Park, MD 20740
rraj27@umd.edu

Uthappa Madettira
Maryland Applied Graduate Engineering
University of Maryland
College Park, MD 20740
uthu@umd.edu

Abstract

Semantic segmentation is a computer vision task that involves labeling each pixel in an image with a corresponding class. Real time semantic segmentation builds on the foundations of semantic segmentation. Popular deep neural network architectures for semantic segmentation do not transcend well for real time semantic segmentation because of their large inference time. We have implemented Efficient neural network (E-Net) which is specifically designed for tasks requiring low latency operation. The network has less parameters and better inference time than the popular U-Net and SegNet architectures that are used for segmentation task.

1. Introduction

Semantic segmentation is a popular computer vision task in the field of autonomous driving, medical imaging and urban planning. It provides pixel level categorization in which each and every pixel in an input image is labeled for a particular class thereby providing the spatial and contextual relationships within an image. Real time semantic segmentation is derived from an image segmentation tasks with additional time constraints. Achieving real-time performance involves significant challenges, including optimizing computational efficiency, reducing latency, and maintaining high accuracy despite the constraints.

Several advanced architectures have been developed to address these challenges and enhance the performance of real-time semantic segmentation. Also the availability of larger datasets and computationally-powerful machines have helped these networks perform good for image segmentation. Models like Fully Convolutional Networks (FCNs), U-Net, SegNet, DeepLab are the popular ones for image segmentation task. FCNs replace fully connected layers in traditional CNNs with convolutional layers to maintain spatial information. They were the first to show that CNNs could be trained end-to-end for pixel-wise prediction. U-Net architecture, employs a encoder-decoder structure with skip connections to recover spatial resolution lost during down-sampling. Similar to U-Net, SegNet uses an encoder-decoder architecture but focuses on efficient up-sampling using indices from max-pooling layers. All these architectures are based on VGG16 architecture, which is a very large model designed for multi-class classification. These networks have large number of parameters and have huge inference time making them not suitable for real time tasks. Also deep architectures with numerous parameters can consume substantial memory. Enet which is a lightweight network is designed specifically for real time applications and achieves high performance with low computational requirements.

ENet adopts an encoder-decoder architecture specifically optimized for speed and efficiency. The encoder compresses the input image into a lower-dimensional representation, capturing essential features while reducing the computational load. The decoder then upsamples this representation to produce the final pixel-wise segmentation map. By using techniques such as early downsampling, dilated convolutions, and factorized filters, ENet reduces the number of parameters and floating-point operations required. This streamlined approach allows ENet to perform up to 18 times faster than comparable architectures like SegNet, with significantly fewer computational resources.

2. Architecture

ENet is designed for efficient real-time semantic segmentation, focusing on speed and low computational overhead. The ENet architecture has the following building blocks:

2.1. Building Blocks

2.1.1 Initial Block

The ENet architecture begins with the Initial Block, which is designed for efficient downsampling and feature expansion. It consists of two parallel branches: the main branch

and the extension branch. The main branch performs a 3x3 convolution with a stride of 2, which effectively reduces the spatial dimensions of the input while producing 13 feature maps. Concurrently, the extension branch applies a max-pooling operation with the same stride, generating 3 feature maps. These two outputs are concatenated to form a 16-channel feature map. This concatenation is followed by batch normalization to ensure the stability of the network and a non-linear activation function, either ReLU or PReLU, depending on the specified parameter. This combination allows for efficient processing and ensures that critical spatial information is retained in the early stages of the network.

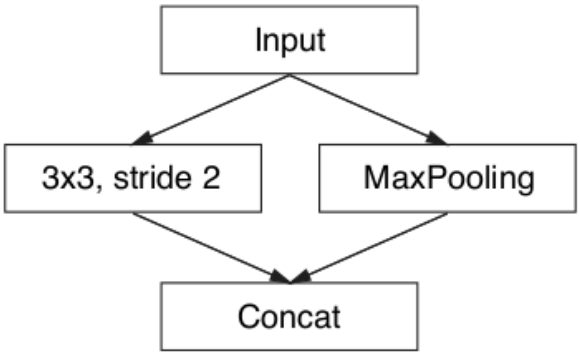

Figure 1. Initial Block

2.1.2 Regular Bottleneck

Regular bottleneck is the fundamental building block of ENet, crucial for maintaining the balance between computational efficiency and the network's representational power. Each regular bottleneck consists of a main branch, which acts as a shortcut connection, and an extension branch that performs the main convolutions. The extension branch starts with a 1x1 convolution that reduces the number of channels by a specified internal ratio, known as projection. This step is followed by a 3x3 convolution, which can be regular, dilated (to increase the receptive field without additional computational cost), or asymmetric (split into two convolutions, such as 5x1 and 1x5, to reduce computational overhead while maintaining a large receptive field). The extension branch concludes with another 1x1 convolution that restores the original number of channels, known as expansion. To prevent overfitting, dropout is applied at this stage. The output from the extension branch is then added to the main branch output, batch normalized, and passed through a ReLU or PReLU activation function. This structure allows the network to efficiently learn complex representations while maintaining low computational costs.

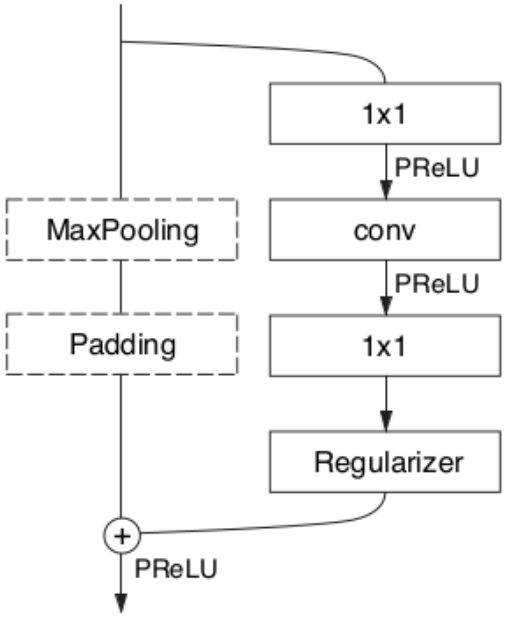

Figure 2. Standard Bottleneck Block

2.1.3 Downsampling Bottleneck

The downsampling bottleneck is designed to reduce the spatial dimensions of the feature map while increasing the number of channels. This is crucial for capturing more abstract features at different levels of the network. In the main branch, a max-pooling operation with a stride of 2 is applied, and the pooling indices are stored for use in the corresponding upsampling bottleneck later in the network. The extension branch begins with a 2x2 convolution with stride 2, which projects the input into a lower-dimensional space. This is followed by a 3x3 convolution, which can be dilated or asymmetric, similar to the regular bottleneck. The extension branch concludes with a 1x1 convolution that expands the feature map back to the desired number of channels. Batch normalization and dropout are applied to ensure stability and prevent overfitting. The outputs from the main and extension branches are concatenated, batch normalized, and activated using PReLU, providing a rich, abstract representation of the input data at reduced spatial dimensions

2.1.4 Upsampling Bottleneck

The upsampling bottleneck mirrors the downsampling bottleneck, focusing on increasing the spatial resolution of the feature maps. In the main branch, a 1x1 convolution is applied to the input to project it into a lower-dimensional space, followed by max unpooling using the indices stored during the downsampling stage. The extension branch begins with a 1x1 convolution for projection, followed by

a transposed convolution (also known as deconvolution) which effectively upsamples the feature map. Another 1x1 convolution is then applied to expand the feature map back to the desired number of channels. Batch normalization and dropout are used to maintain stability and prevent overfitting. The outputs from the main and extension branches are then added together, batch normalized, and passed through a PReLU activation function, ensuring that the upsampled feature maps are both accurate and computationally efficient.

2.2. ENet Model

The full ENet model is structured to progressively encode the input image into a lower-dimensional representation and then decode it back to the original resolution with pixel-wise classifications.

1. **Initial Block:** The input image is first processed by the initial block, which performs efficient downsampling and feature expansion.

2. **Stage 1 (Encoder):** This stage includes a downsampling bottleneck followed by regular bottlenecks, gradually reducing the spatial dimensions while increasing the depth of the feature maps.

3. **Stage 2 (Encoder):** Further downsampling is performed, followed by regular, dilated, and asymmetric bottlenecks to capture complex features at multiple scales.

4. **Stage 3 (Encoder):** This stage continues with similar bottlenecks without additional downsampling, maintaining the depth and complexity of the feature maps.

5. **Stage 4 (Decoder):** The first upsampling stage uses upsampling bottlenecks to increase the spatial dimensions of the feature maps, making use of the indices stored during downsampling.

6. **Stage 5 (Decoder):** The final upsampling stage restores the feature maps to the original input size, followed by a transposed convolution that produces the final segmentation output

3. Implementation

The ENet architecture that was discussed in the earlier sections was implemented in *Pytorch* in a python notebook. The datasets were downloaded and arranged in a required folder structure. The model was then trained as per the following details:

Name	Type	Output size
initial		$16 \times 256 \times 256$
bottleneck1.0	downsampling	$64 \times 128 \times 128$
$4\times$ bottleneck1.x		$64 \times 128 \times 128$
bottleneck2.0	downsampling	$128 \times 64 \times 64$
bottleneck2.1		$128 \times 64 \times 64$
bottleneck2.2	dilated 2	$128 \times 64 \times 64$
bottleneck2.3	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.4	dilated 4	$128 \times 64 \times 64$
bottleneck2.5		$128 \times 64 \times 64$
bottleneck2.6	dilated 8	$128 \times 64 \times 64$
bottleneck2.7	asymmetric 5	$128 \times 64 \times 64$
bottleneck2.8	dilated 16	$128 \times 64 \times 64$
Repeat section 2, without bottleneck2.0		
bottleneck4.0	upsampling	$64 \times 128 \times 128$
bottleneck4.1		$64 \times 128 \times 128$
bottleneck4.2		$64 \times 128 \times 128$
bottleneck5.0	upsampling	$16 \times 256 \times 256$
bottleneck5.1		$16 \times 256 \times 256$
fullconv		$C \times 512 \times 512$

Figure 3. Structure of the model implementation used

Dataset Type	**Number of Images**
Training	367
Validation	101
Testing	233

Table 1. Dataset split

3.1. Dataset

For training this model we used the *CamVid* dataset which has the below mentioned split of data:

The original images had a resolution of 360×480, but they were resized to 512×512 before feeding them into the network. This is because a convolutional neural network is better at capturing spatial and semantic information in a square-sized image.

3.2. Hyperparameters

The following hyperparameters were tuned to optimize the performance of the model.

1. **Learning Rate**: An optimal learning rate is really important for proper loss convergence of a model because a high learning rate will lead to the model blowing up whereas a low learning rate might not converge the

model. We found that the best results were seen at a learning rate of 5×10^{-4}.

2. **Batch Size**: The batch size decides the number of images that are passed to the model in a single training/validation iteration. We have kept the batch size as 5 for two reasons. One, the gradient descent is performed higher number of times during training and hence the model converges faster. Second, as this is a convolutional network and was trained on an *NVIDIA RTX 3060* graphics card with 6GB of memory, the GPU frequently ran out of memory on higher batch sizes.

3. **Momentum**: For updating parameters during training, we used the ADAM optimizer with a weight decay of 2×10^{-4}. The weight decay is an important hyperparameter which ensures that the gradient descent proceeds in the direction of past trends of gradient. This helps the optimizer minimize the losses much faster.

Finally we ran the model for 100 epochs on the training and validation dataset and saw the losses converging quite nicely in both the training and validation sets.

4. Results

Based on the model that we have implemented as described earlier, we were able to achieve a decent performance on real-time semantic segmentation of the test data from the *CamVid* dataset. It contains a total of 599 images which could be processed by our implemented model at 10 FPS (frames per second). This signifies that real-time processing capabilities were displayed by the model. The segmentation mask video which shows this result has been linked here: https://drive.google.com/file/d/1FQidlf5rqZjHdSQhCbD-O594d85F2fhJ/view?usp=sharing.

Our model has performed fairly well on the test set by giving a *Per-Pixel* accuracy of 75%. Per-Pixel accuracy is defined by the following formula:

$$\text{Pixel Accuracy} = \frac{\text{Number of correctly classified pixels}}{\text{Total number of pixels}}$$

However, there is a drawback of per-pixel accuracy metric. It is very sensitive to class imbalances and most large-scale datasets such as the *CamVid* dataset have class imbalances in some of their images. In order to account for this drawback, in image segmentation tasks, other metrics such as *Intersection over Union (IoU)* and *Dice score* are calculated. These metrics are not sensitive to class imbalances as they focus on overlap areas rather than pixel-wise matching. *IoU* in particular, emphasises the precision of delineating object boundaries. They are given by the following formulae:

$$\text{IoU} = \frac{\text{Area of Overlap between G.T. and Pred.}}{\text{Area of Union between G.T. and Pred.}}$$

$$\text{F1/Dice Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The values of these metrics based on our results has been shown in the table below. These results prove that we have achieved a decent performance on the segmentation task, despite making the model extremely lightweight for real-time applications.

Evaluation Metric	**Value**
Pixel Accuracy	75.44%
F1/Dice Score	0.64
IoU	0.515

Table 2. Evaluation Metric Table

Finally the inferences that we ran on the model for the test data gave us the segmentation masks. A comparison of the predicted masks for the input images against the ground truth has been shown below:

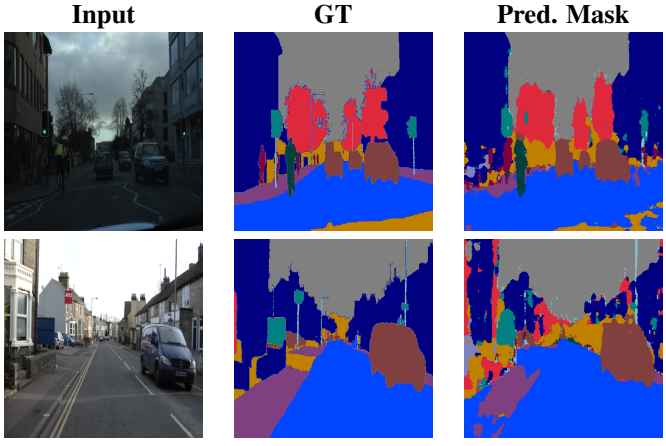

Figure 4. Comparison of input images, ground truths, and prediction masks.

5. Conclusion

In this project, we have implemented the ENet architecture as described earlier. Our aim was to build a model for semantic segmentation task that could be used in real-time.

For this we made the model lightweight by down-sampling aggressively early and performing all the convolution operations on the scaled-down images so that it would be computationally inexpensive. The feature maps were then upsampled rapidly in a few blocks to achieve the segmentation masks of the same size as the input images. The performance metrics that we achieved on the testing dataset were discussed and it can be seen that the results are quite promising.

6. References

[1] Paszke, A. & Chaurasia, A. (2016) ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation *In: arXiv:2112.08088*.

[2] Simoyan, K. & Zisserman, A. (2014) Very deep convolutional networks for large-scale image recognition *In: arXiv:1409.1556*.

[3] Badrinarayanan, V., Handa, A. &Cipolla, A. (2015) Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *In: arXiv:1511.00561*.

[4] Krizhevsky, A. & Sutskever, I. (2016) Imagenet classification with deep convolutional neural networks *In: Advances in Neural Information Processing Systems 25, 2012, pp. 1097–1105*