Urban Scene Segmentation with U-Net

Sélim Ben Abdallah

Abstract

This study aims to explore how U-Net [6] can be applied to the segmentation of urban scenes. In the first section, I introduce the U-Net architecture. In the second section, I describe how I trained the model on the Cityscapes [2] dataset and I present the results obtained.

Note: I wrote this article myself and used ChatGPT-5 to help correct sentences and improve the clarity of certain sections.

1 The U-Net architecture

Introduced in 2015, U-Net [6] is a widely used model for several tasks, such as image segmentation (by learning the mapping between input image pixels and ground-truth segmentation masks), low-to-high resolution upscaling (by reconstructing detailed high-resolution images from low-resolution inputs), and even as a backbone denoising network for diffusion models (by predicting and removing noise from images during the denoising process).

The U-Net consists of an encoder–decoder architecture. The encoder is responsible for extracting features from an input image, while the decoder upsamples the intermediate features to produce the final output. The encoder and decoder are symmetrical and connected by residual connections (i.e. direct links that by-pass one or more intermediate layers).

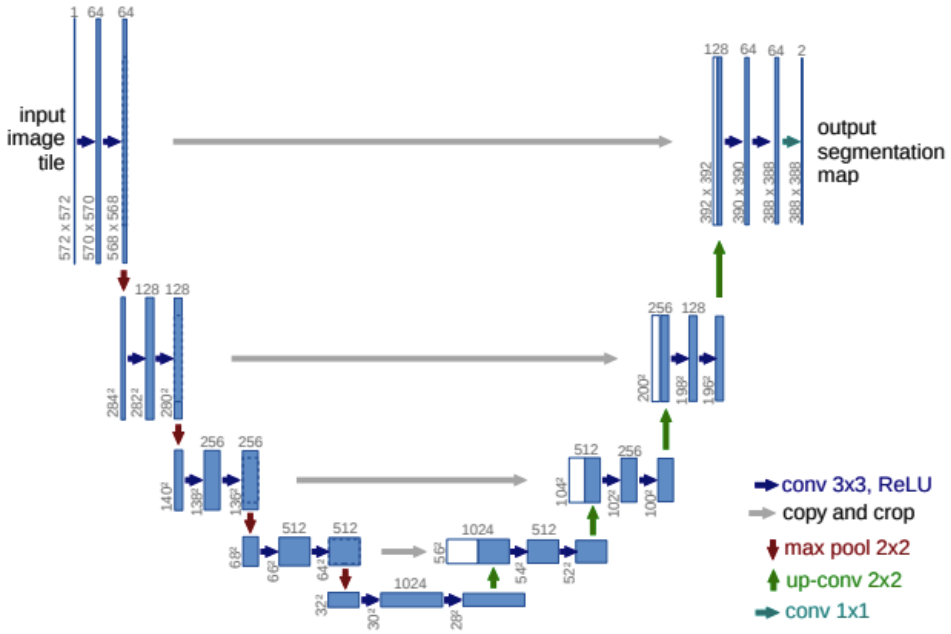


Figure 1: U-Net architecture (example for 32x32 pixels in the lowest resolution) [6]

For a given image, input features are passed through an encoder consisting of repeated convolutional and max-pooling layers that extract intermediate features. These features are then upsampled by the corresponding decoders, with the correspondence between encoder and decoder layers at different

scales indicated by the gray arrows in **Figure 1**. The final layer produces the output, a segmentation mask for instance. The loss between the predicted mask and the ground-truth mask is then calculated, and the gradients are backpropagated through the model to update the weights and improve U-Net's predictions.

The encoder consists of downsampling layers a different levels. Each level comprises two 3×3 convolutional layers followed by non-linear activation functions (ReLU) and a 2×2 max-pooling layer (stride 2) for downsampling. At each downsampling step, the number of feature channels is doubled, while the unpadded convolution layers extract increasingly refined intermediate features.

The decoder, i.e. the expansive path, consists of a multi-level series of upsampling layers. Each level begins with an upsampling of the feature map followed by a 2×2 convolution that divides by the number of feature channels by two. The result is concatenated with the correspondingly cropped feature map from the encoder via residual connections. This is followed by two 3×3 convolutional layers, each with ReLU activation. Cropping is applied to compensate for the loss of border pixels in each convolution. The final layer is a 1×1 convolution that maps each 64-channel feature vector to the desired number of output classes. In total, the network contains 23 convolutional layers.

The encoder and decoder are connected via two types of connections: residual connections and the bottleneck. The residual connections copy features from the corresponding encoder layer, crop them, and concatenate them with the matching decoder layer. The encoder's features are effective at identifying which pixels belong to a given object, while the decoder's features help localize the area of the image where the object is found. Combining these two sources of information through the residual connections enables precise object segmentation. The bottleneck is the transition between the encoder and decoder. At this stage, the features from the encoder are downsampled using a 2×2 max-pooling layer, then pass through two 3×3 convolutional layers each followed by ReLU activations, and are finally upsampled with a 2×2 convolution.

As described in the U-Net paper [6], the loss function (also named the energy function) is computed as a pixel-wise softmax over the final feature map followed by a cross-entropy loss function. The softmax function for pixel position $x \in \Omega$ and class $k \in \{1, \dots, K\}$ is defined as

$$p_k(x) = \frac{\exp(a_k(x))}{\sum_{k'=1}^{K} \exp(a_{k'}(x))},$$

where $a_k(x)$ denotes the activation in feature channel k at pixel position x. This function approximates the maximum operation by assigning $p_k(x) \approx 1$ for the class k with the highest activation $a_k(x)$, and $p_k(x) \approx 0$ for the other classes.

The cross-entropy loss penalizes deviations between the predicted probabilities and the ground-truth labels at each pixel, and is given by

$$\mathcal{L} = -\sum_{x \in \Omega} \sum_{k=1}^{K} y_k(x) \log p_k(x),$$

where $y_k(x)$ is the ground-truth label for class k at pixel x. Minimizing this loss encourages the network to assign high probabilities to the correct class at every pixel position.

2 Image Segmentation with U-Net

U-Net [6] was initially introduced for medical imaging. In this study, I used the Cityscapes dataset [2], which contains 5,000 annotated images of European cities captured from the driver's perspective.

2.1 Architectural design

I reimplemented the architecture presented in **Figure 1**, with few modifications. The first one is adding batch normalization layers after each 3×3 convolutional layer to stabilize the training process. Also, the original U-Net implementation [6] used unpadded convolutions, which reduced the spatial

size by two pixels per side (one pixel per side for each convolution). This shrinking effect required cropping the encoder feature maps before concatenation in the skip connections to ensure matching dimensions. To avoid this, I used a padding of one pixel in my convolutional layers, which preserves the spatial size between input and output and removes the need for cropping. For an input dimension of 64, the Double Convolution design is as follows :

```
DoubleConv(
  (doubleconv): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
  )
)
```

The rest of the architecture follows the paper implementation, with four encoding stages, a bottleneck, skip connections, and four decoding stages. The last layer is a 1×1 convolution that outputs a feature map of shape (batch size, out channels, height, width) which represents the logits per class and per pixel. I then use an argmax function along the channel dimension to pick the highest-scoring class index at each pixel, producing the segmentation mask.

```
UNet(
  (enc_level_1): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool_level_1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (enc_level_2): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool_level_2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (enc_level_3): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool_level_3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (enc_level_4): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (pool_level_4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (bottleneck): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(1024, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (upsample_level_4): ConvTranspose2d(1024, 512, kernel_size=(2, 2), stride=(2, 2))
  (dec_level_4): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (upsample_level_3): ConvTranspose2d(512, 256, kernel_size=(2, 2), stride=(2, 2))
  (dec_level_3): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (upsample_level_2): ConvTranspose2d(256, 128, kernel_size=(2, 2), stride=(2, 2))
  (dec_level_2): DoubleConv(
    (doubleconv): Sequential(
      (0): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
    )
  )
  (upsample_level_1): ConvTranspose2d(128, 64, kernel_size=(2, 2), stride=(2, 2))
  (conv_final): Conv2d(64, 19, kernel_size=(1, 1), stride=(1, 1))
)
```

2.2 Results

The model was trained for 50 epochs using cross-entropy loss and the Adam optimizer with a fixed learning rate of 1e-4. The goal was not to obtain the most optimized U-Net model; I did not use weight decay or experiment with other optimizers. Instead, the aim was to train an initial model capable of qualitatively achieving its segmentation objective. The original images (1024×2048) were resized to

256×512 to reduce memory usage.

Parameters	Value
GPU (Google Colab)	NVIDIA A100
Image size	256*512
Batch size	2
Total Epochs	50
Learning Rate	1e-4
Loss	Cross Entropy
Optimizer	Adam

Table 1: Hyperparameters - U-Net training

As presented in **Figure 2 (left)**, the training loss decreases consistently across all 50 epochs, indicating that the model is learning and the Adam optimizer is functioning properly. The validation loss also decreases but plateaus after 15 epochs and begins to fluctuate after around 20 epochs. This suggests that the model is not overfitting but is also not improving significantly, possibly due to a lack of regularization. Potential improvements could include replacing the ReLU layers with LeakyReLU activations or introducing weight decay into the optimizer.

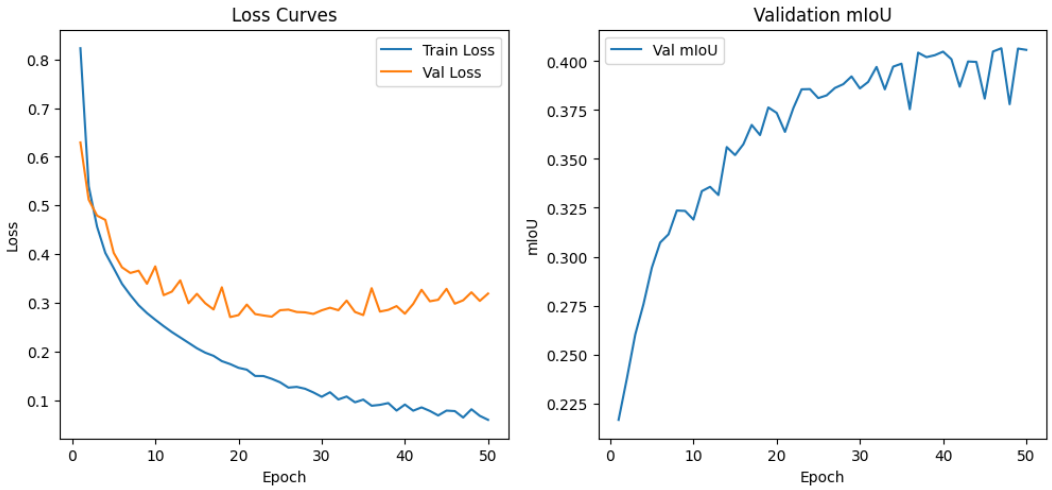
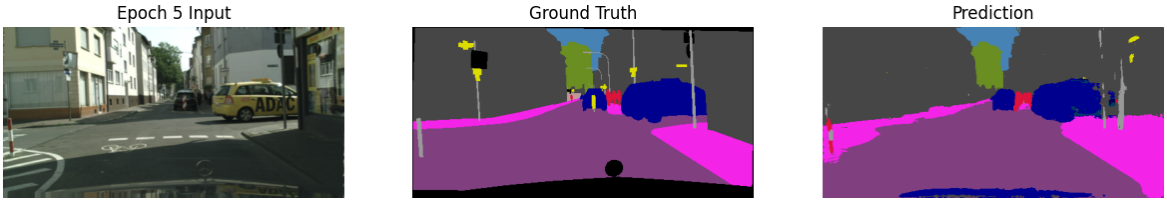


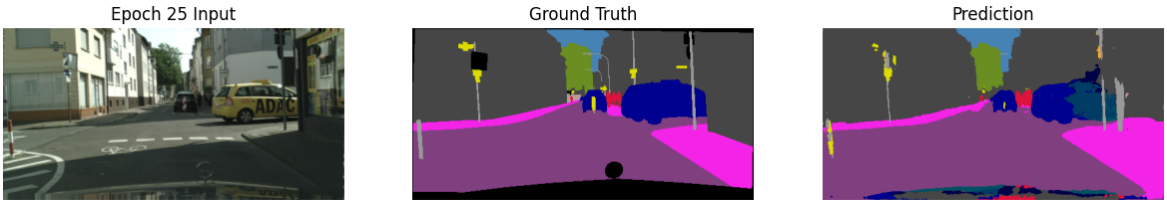
Figure 2: Train/Val Losses (left) and mean IoU (Val set) [6]

As presented in **Figure 2 (right)**, the mean Intersection over Union (mIoU), a metric that measures the overlap between predicted and ground-truth regions, increases during the first 25 epochs before plateauing and fluctuating around 0.40. This indicates that the model has learned a useful representation, but the quality of the predicted segmentation masks remains limited. One possible reason for this plateau is the downscaling of the original images from 1024×2048 to 256×512, which may have led to the loss of fine details crucial for precise segmentation. However, the predicted segments are still adequate to confirm that the model has correctly learned the most relevant representations as seen in **Figure 3**.

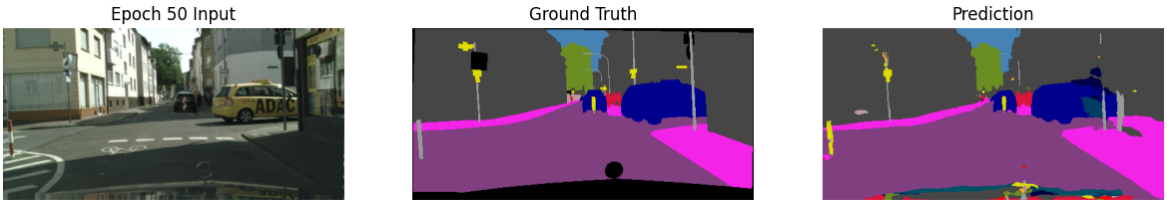
We can observe that the model refined its predictions between epoch 5 and epoch 50. For instance, the road sign (on the left side of the image), which was initially misclassified as a person at epoch 5, is correctly segmented as a road sign at epoch 50. The road segmentation (colored in dark purple) also appears more precise after 50 epochs. Interestingly, the model even attempted to segment the reflections of objects on the car at epoch 50. However, the shadows in the input image made it difficult for the model to accurately segment the sidewalk on the right side of the image. In addition, traffic lights and street lamps were not effectively segmented.



(a) Prediction at 5 epochs



(b) Prediction at 25 epochs



(c) Prediction at 50 epochs

Figure 3: Predicted Segmentation masks at 5, 25 and 50 epochs

I also tested the model on images that were not included in the training or validation sets, and the predicted segmentation masks appear quite accurate, as shown in **Figure 4**.

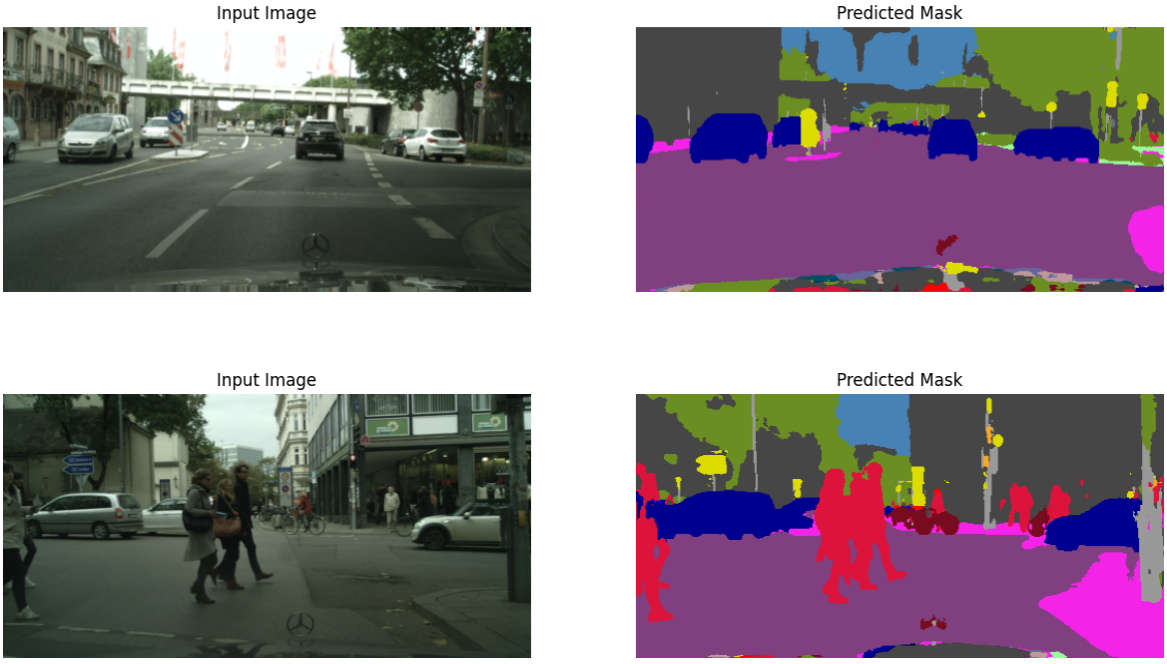


Figure 4: Predicted Segmentation masks on the test set

3 Conclusion

At the time of writing this report (Summer 2025), U-Net [6] is already 10 years old, and multiple other models for image segmentation have since been introduced. Examples include Mask R-CNN [3], an instance segmentation model that extends the Faster R-CNN architecture [5] by adding a mask prediction branch, and SegFormer [7], which leverages the Transformer's attention mechanism to capture long-range dependencies. Despite the emergence of these newer architectures, U-Net remains a strong backbone for modern models, such as RadioUnet [4] for radio map segmentation; or TransUNet [1], which embeds Transformer blocks into the encoder of a U-Net to combine global attention with detailed spatial reconstruction.

In this study, I trained the U-Net model from scratch on the Cityscapes dataset [2]. Even with only 50 training epochs, the model demonstrated efficient and accurate segmentation performance. Further work can include, using more refined metrics than mIoU, varying the learnign rate, and other hyperparameters. Also, model such as Mask R-CNN, YOLOs, or SegFormer can be considered.

4 Code

The code for the U-Net implementation is available on github : **https://github.com/selim-ba/**

References

[1] Jieneng Chen, Yongyi Lu, Qihang Yu, Xiangde Luo, Ehsan Adeli, Yan Wang, Le Lu, Alan L Yuille, and Yuyin Zhou. Transunet: Transformers make strong encoders for medical image segmentation. In *Medical Image Computing and Computer Assisted Intervention – MICCAI 2021*, pages 61–71. Springer, 2021.

[2] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3213–3223, 2016.

[3] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 2961–2969, 2017.

[4] Nicolas Meyer, Khalid Rajab, Minho Choi, Wonju Seong, Sunwoo Kim, Jiwon Jang, and Heejung Chang. Radiounet: Fast segmentation of radio maps with convolutional neural networks. *IEEE Transactions on Wireless Communications*, 20(6):3758–3771, 2021.

[5] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, volume 28, 2015.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *arXiv preprint arXiv:1505.04597*, 2015.

[7] Enze Xie, Wenhai Wang, Zhiding Yu, Anima Anandkumar, Jose M. Alvarez, and Ping Luo. Segformer: Simple and efficient design for semantic segmentation with transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.