Report for Assignment 2 - Roberto Carminati, Robert Sauerborn

What other architectures for segmentation apart from the SegFormer have you learned in the lecture and how do they differ from the SegFormer?

Fully Convolutional Network (FCN)

- **Architecture:** Utilizes a series of convolutional layers with downsampling and upsampling.
- **Function:** High-resolution input is downsampled through pooling or strided convolutions and then upsampled using linear interpolation or transposed convolution.
- **Differences from SegFormer:** FCNs do not use transformers; instead, they rely entirely on convolutional operations for feature extraction and segmentation

U-Net

- **Architecture:** Consists of an encoder-decoder structure with skip connections between corresponding layers of the encoder and decoder.
- **Function:** The encoder captures context through downsampling, while the decoder enables precise localization using upsampling.
- **Differences from SegFormer:** U-Net emphasizes skip connections between encoder and decoder stages, whereas SegFormer uses a hierarchical transformer structure without explicit skip connections.

Feature Pyramid Networks (FPN)

- **Architecture:** Comprises a backbone network for feature extraction and a top-down pathway that combines features from different levels of the backbone.
- **Function:** Enhances the object detection and segmentation by leveraging features at multiple scales.
- **Differences from SegFormer:** FPNs focus on merging multi-scale features from a convolutional backbone, while SegFormer processes patches with transformers to handle multiple scales.

Key Differences Between SegFormer and Other Architectures

- Transformer Utilization: SegFormer employs a hierarchical transformer architecture for feature extraction, while the others rely primarily on convolutional layers.
- Decoder Design: SegFormer uses a lightweight decoder with multi-layer perceptrons (MLPs) for feature transformation and upsampling, differing from the more complex decoder structures in architectures like U-Net and FCN.

- Patch Embedding: SegFormer embeds patches with overlapping regions and processes them through transformer layers, which is distinct from the spatial downsampling and upsampling strategies in FCN and U-Net.

These differences highlight SegFormer's innovative approach in leveraging transformers for segmentation tasks, architectures that process patches with overlapping regions, avoiding explicit downsampling and upsampling while still capturing multi-scale information. This approach shows the evolution in segmentation strategies to balance efficiency and precise localization.

Why are those architectures using down- and up-sampling instead of keeping the same resolution?

The use of downsampling and upsampling in segmentation architectures is crucial for achieving computational efficiency, enhanced feature extraction, and effective handling of varying spatial resolutions.

Computational Efficiency

Downsampling reduces the computational load and memory usage by decreasing the spatial dimensions of feature maps. This makes processing more manageable, particularly in deeper network layers, allowing for complex operations without a significant increase in computational demands.

Enhanced Feature Extraction

Downsampling increases the receptive field of the network, capturing broader contextual information and spatial dependencies. This hierarchical feature learning ensures that early layers focus on fine details, while deeper layers capture abstract, high-level concepts, improving the network's understanding of complex patterns.

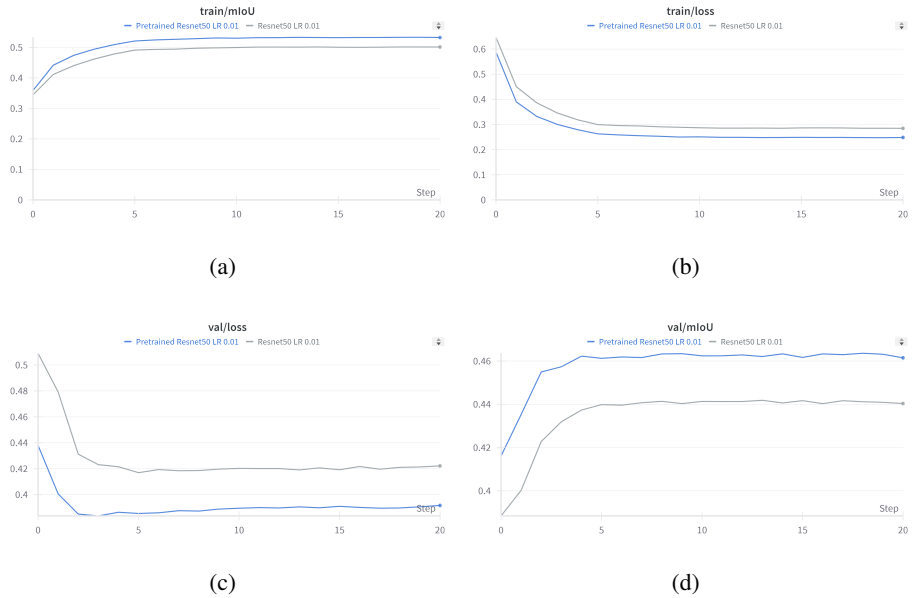
Multi-scale Information Handling

Segmentation tasks require handling objects at various scales within an image. By processing inputs at multiple resolution levels, downsampling and subsequent upsampling enable the network to capture features relevant to differently sized objects. This multi-scale approach allows the network to combine high-level semantic information with low-level spatial details, crucial for accurate segmentation.

Effective Learning and Generalization

Downsampling helps prevent overfitting by encouraging the network to learn general and robust features instead of memorizing fine-grained details. Pooling operations, such as max pooling, summarize important features and reduce sensitivity to small translations or distortions in the input image.

In conclusion, downsampling and upsampling are essential techniques in segmentation architectures, balancing computational efficiency, robust feature extraction, and multi-scale information handling, leading to improved segmentation accuracy and performance.



(a) (b)

(c) (d)

Figure 1: Pretrained Resnet50 vs. Resnet50

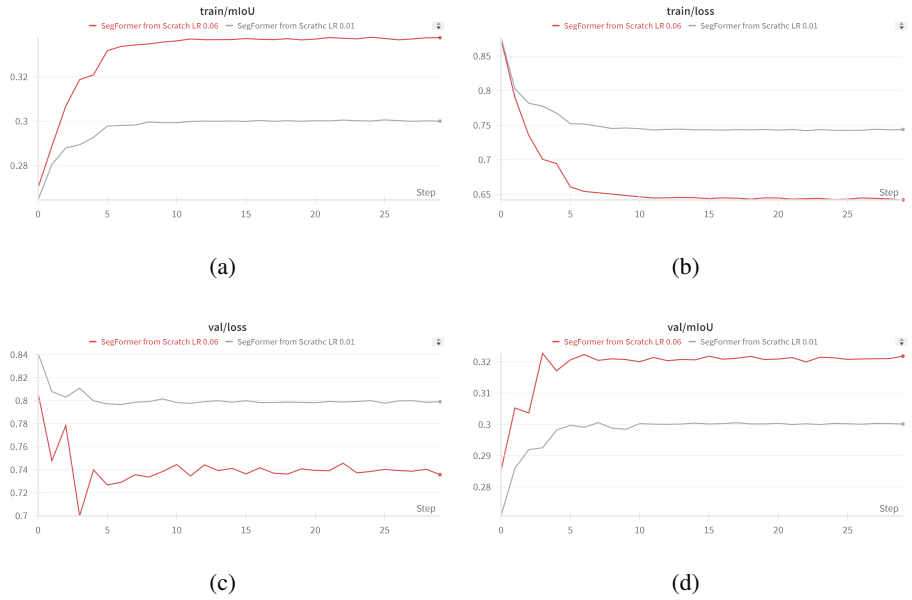
What is the purpose of pre-training and fine-tuning? For which use-cases is it especially beneficial?

Pre-training and fine-tuning are techniques used to enhance the performance and efficiency of neural networks. Pre-training involves training a model on a large, general dataset to learn useful features and patterns, providing a strong foundation for further learning. Fine-tuning then adapts this pre-trained model to a specific task or dataset by continuing the training process, focusing on the new, more specialized data. This approach is especially beneficial in fields like natural language processing, computer vision, and speech recognition, where pre-trained models can significantly improve performance and reduce training time. Pre-training and fine-tuning allow for more efficient and effective model development, making it easier to achieve high performance on a wide range of tasks.

What have you observed in your experiments above, how has using pre-trained weights influenced the performance and why?

We used the model Resnet with 21 epochs and a learning rate of 0.0014 on the "Oxford pets dataset" first with the Resnet50 model from scratch obtaining a training mIoU of 0.5012 and a validation mIoU of 0.4404. And afterwards we used the same model with the same parameters but with pretrained weights from Pytorch obtaining a training mIoU of 0.5324 and a validation mIoU of 0.4615. As visible in the Figure 1 using the pretrained model the mIoU improve relevantly especially on the validation set. This let us train the model for less epoch and use a smaller learning rate.

In our second experiment, again working on the "Oxford pets dataset", we implemented a SegFormer model. We trained it for 30 epoch first with a learning rate of 0.001 obtaining a training mIoU of 0.3 and a validation mIoU of 0.3001 and after with a learning rate of 0.006 obtaining a training mIoU of 0.3377 and a validation mIoU of 0.3218. (Figure 2) With SegFormer from scratch we obtain a worse performance than with both the experiments with Resnet50.



(a) (b)

(c) (d)

Figure 2: SegFormer from Scratch LR = 0.06 vs. SegFormer from Scratch LR = 0.01

(a) (b) (c) (d)

Figure 3: Firsts samples and mask generated by the model Segformer

The use of pretrained weights shows the best performance, letting us use a smaller learaning rate just to fine tune the model, and train the model for a smaller number of epochs. Training a model for vision computing is also very expensive if we think about resources and at the power of the computers. Using pretrained weights make working on this topics more accessible to everyone and also improve the research speed on this topic faster.

Finally we visualize the predicted masks of the SegFormer model of the firsts 2 validation samples. (Figure 3)

The self-attention used in the SegFormer

In the SegFormer model, the self-attention mechanism is optimized using a spatial reduction technique, where a spatial reduction (sr) layer is applied before the key-value transformation. This reduces the spatial resolution of the feature maps, making the self-attention computation more efficient by reducing the number of operations. Here the relevant lines doing this in our code:

```
if self.sr_ratio > 1:
x_ = x.permute(0, 2, 1).reshape(B, C, H, W)
x_ = self.sr(x_).reshape(B, C, -1).permute(0, 2, 1)
x_ = self.norm(x_)
kv = self.kv(x_).reshape(B, -1, 2, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1,
```

```
else:
    kv = self.kv(x).reshape(B, -1, 2, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4
k, v = kv[0], kv[1]
```

These lines show the use of the spatial reduction (sr) layer, which effectively reduces the spatial dimensions before computing the key and value matrices, thereby reducing the computational load of the self-attention mechanism.