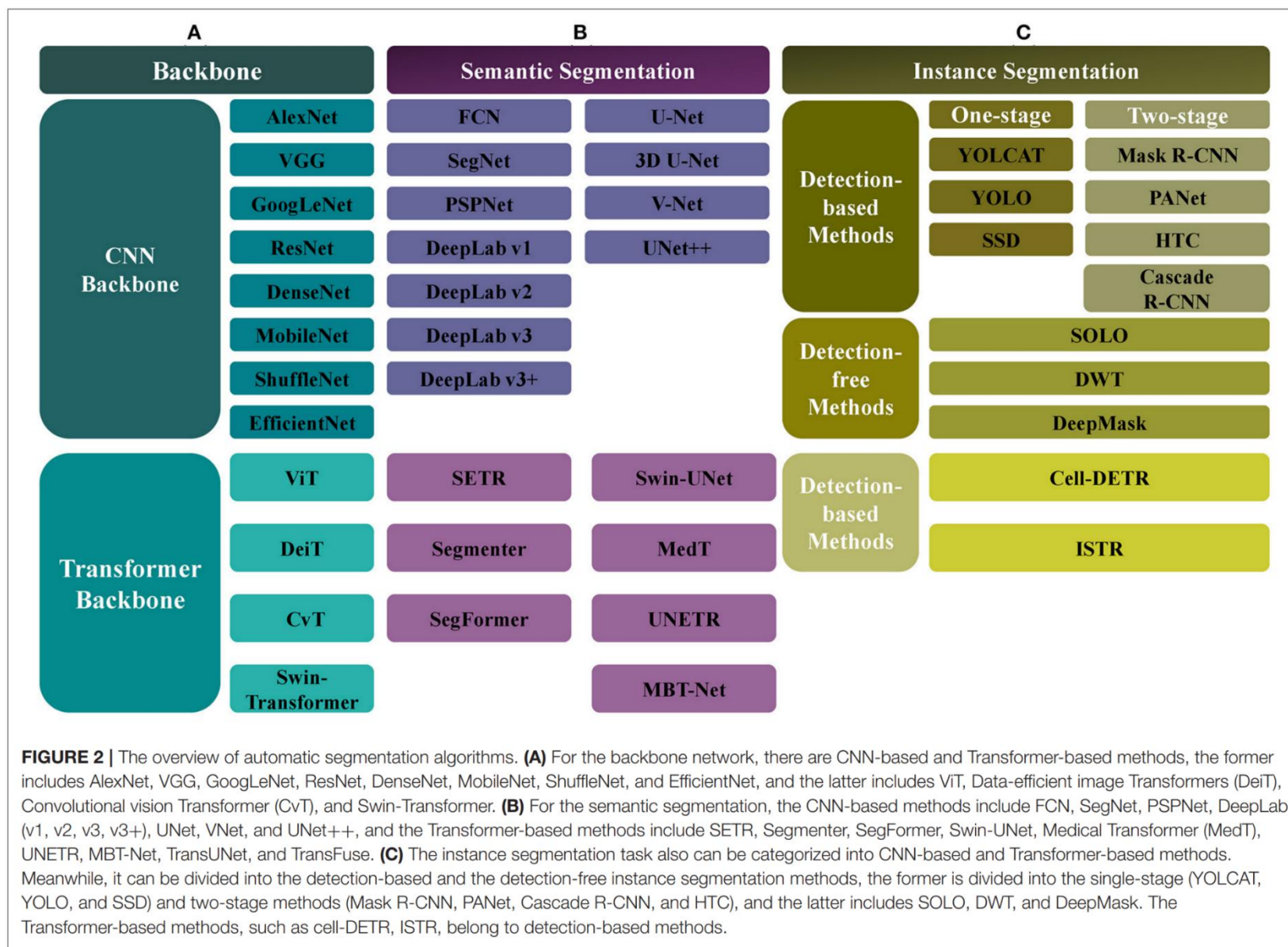


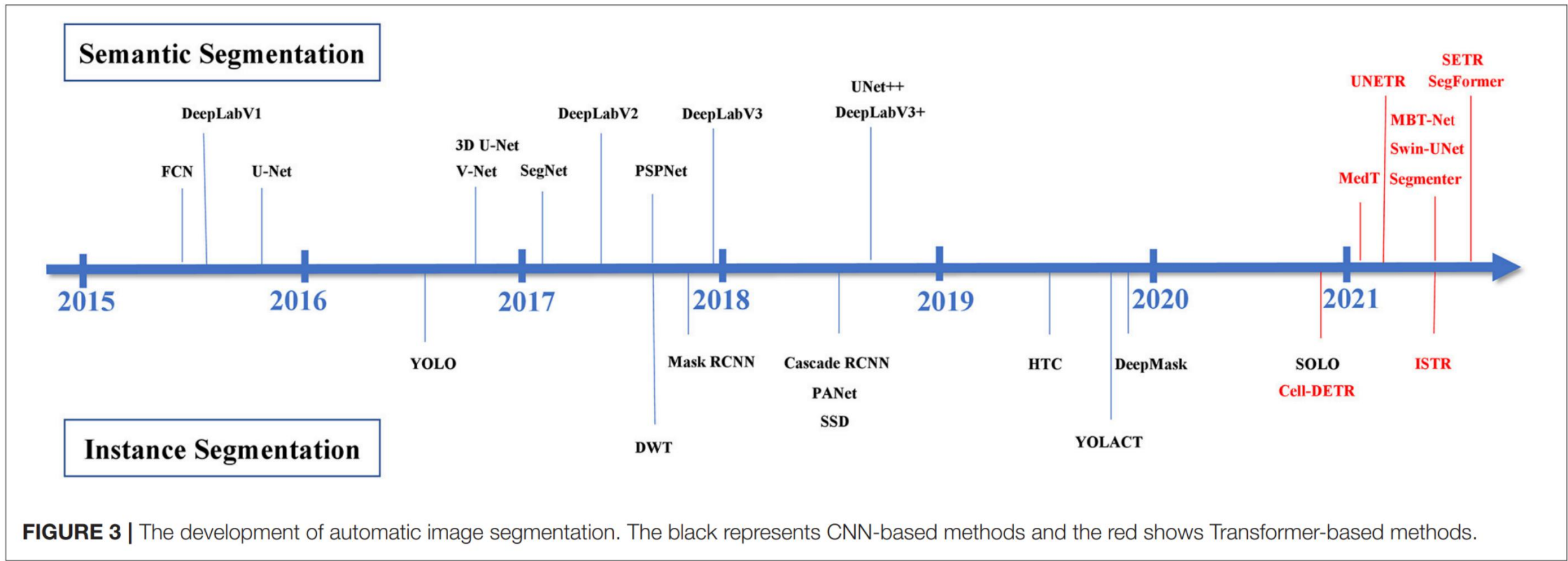
- Image segmentation: the process of dividing a visual input into segments.
- For example, in an image containing cars and trees, segmentation would classify all pixels belonging to trees as a single "tree" class and all pixels for cars as a "car" class.
- It is important to note that semantic segmentation does not distinguish between separate instances of the same object class; that task is handled by a different method called *instance segmentation*.



Convolutional Neural Networks for Image Segmentation



Convolutional Neural Networks for Image Segmentation



- <https://docs.pytorch.org/vision/0.22/models.html#semantic-segmentation>

Semantic Segmentation

• WARNING

The segmentation module is in Beta stage, and backward compatibility is not guaranteed.

The following semantic segmentation models are available, with or without pre-trained weights:

- DeepLabV3
- FCN
- LRASPP

Here is an example of how to use the pre-trained semantic segmentation models:

```
from torchvision.io.image import decode_image
from torchvision.models.segmentation import fcn_resnet50, FCN_ResNet50_Weights
from torchvision.transforms.functional import to_pil_image

img = decode_image("gallery/assets/dog1.jpg")

# Step 1: Initialize model with the best available weights
weights = FCN_ResNet50_Weights.DEFAULT
model = fcn_resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()

# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and visualize the prediction
prediction = model(batch)["out"]
normalized_masks = prediction.softmax(dim=1)
class_to_idx = {cls: idx for (idx, cls) in enumerate(weights.meta["categories"])}
mask = normalized_masks[0, class_to_idx["dog"]]
to_pil_image(mask).show()
```


Fully Convolutional Network (FCN) - the first deep learning model developed specifically for this purpose

Fully Convolutional Networks for Semantic Segmentation

Jonathan Long* Evan Shelhamer* Trevor Darrell
UC Berkeley
{jonlong, shelhamer, trevor}@cs.berkeley.edu

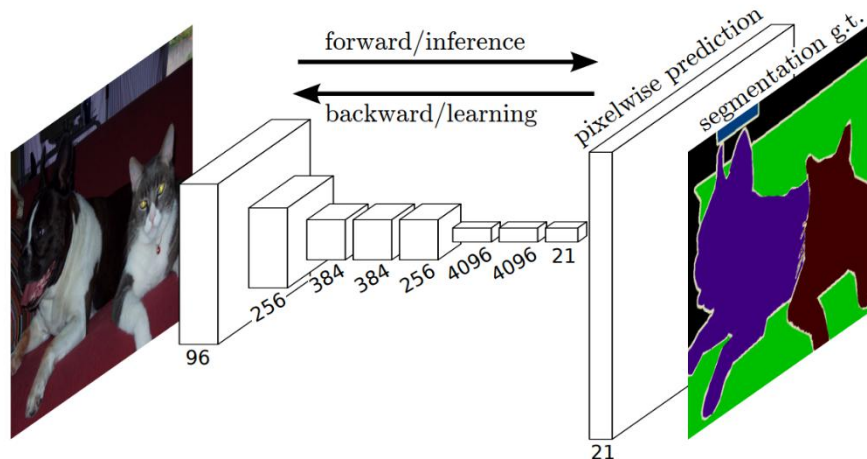


Figure 1. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation.

We show that a fully convolutional network (FCN), trained end-to-end, pixels-to-pixels on semantic segmentation exceeds the state-of-the-art without further machinery. To our knowledge, this is the first work to train FCNs end-to-end (1) for pixelwise prediction and (2) from supervised pre-training. Fully convolutional versions of existing networks predict dense outputs from arbitrary-sized inputs. Both learning and inference are performed whole-image-at-a-time by dense feedforward computation and backpropagation. In-network upsampling layers enable pixelwise prediction and learning in nets with subsampled pooling.

Fully Convolutional Network (FCN) - the first deep learning model developed specifically for this purpose

4.1. From classifier to dense FCN

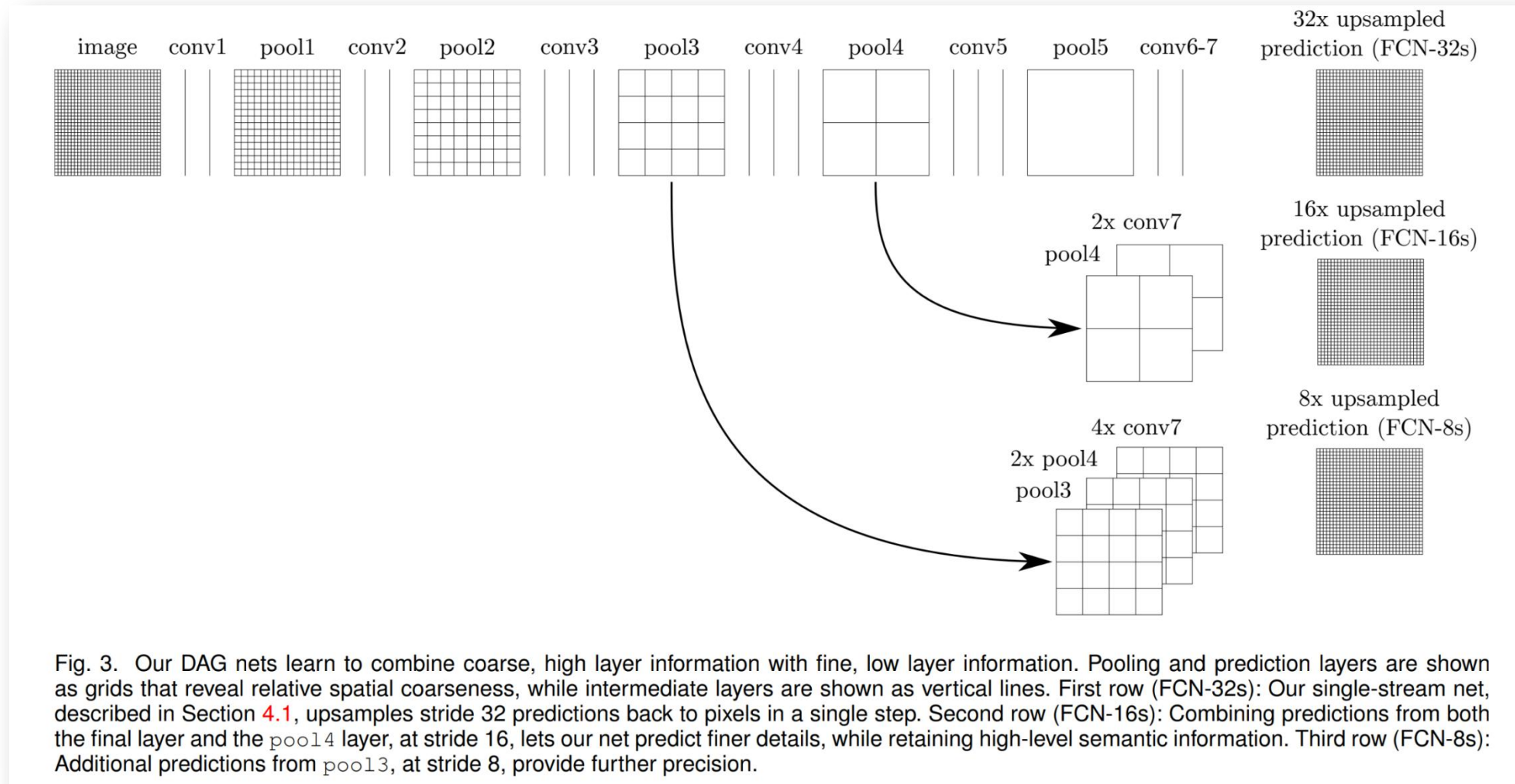
We begin by convolutionalizing proven classification architectures as in Section 3. We consider the AlexNet³ architecture [19] that won ILSVRC12, as well as the VGG nets [31] and the GoogLeNet⁴ [32] which did exceptionally well in ILSVRC14. We pick the VGG 16-layer net⁵, which we found to be equivalent to the 19-layer net on this task. For GoogLeNet, we use only the final loss layer, and improve performance by discarding the final average pooling layer. We decapitate each net by discarding the final classifier layer, and convert all fully connected layers to convolutions. We append a 1×1 convolution with channel dimension 21 to predict scores for each of the PASCAL classes (including background) at each of the coarse output locations, followed by a deconvolution layer to bilinearly upsample the coarse outputs to pixel-dense outputs as described in Section 3.3. Table 1 compares the preliminary validation results along with the basic characteristics of each net. We report the best results achieved after convergence at a fixed learning rate (at least 175 epochs).

3.3. Upsampling is backwards strided convolution

Another way to connect coarse outputs to dense pixels is interpolation. For instance, simple bilinear interpolation computes each output y_{ij} from the nearest four inputs by a linear map that depends only on the relative positions of the input and output cells.

In a sense, upsampling with factor f is convolution with a *fractional* input stride of $1/f$. So long as f is integral, a natural way to upsample is therefore *backwards convolution* (sometimes called *deconvolution*) with an *output* stride of f . Such an operation is trivial to implement, since it simply reverses the forward and backward passes of convolution.

Fully Convolutional Network (FCN) - the first deep learning model developed specifically for this purpose



Fully Convolutional Network (FCN) - the first deep learning model developed specifically for this purpose

FCN

The FCN model is based on the [Fully Convolutional Networks for Semantic Segmentation](#) paper.

• WARNING

The segmentation module is in Beta stage, and backward compatibility is not guaranteed.

Model builders

The following model builders can be used to instantiate a FCN model, with or without pre-trained weights. All the model builders internally rely on the `torchvision.models.segmentation.FCN` base class. Please refer to the [source code](#) for more details about this class.

```
fcn_resnet50(*[, weights, progress, ...])
```

Fully-Convolutional Network model with a ResNet-50 backbone from the [Fully Convolutional Networks for Semantic Segmentation](#) paper.

```
fcn_resnet101(*[, weights, progress, ...])
```

Fully-Convolutional Network model with a ResNet-101 backbone from the [Fully Convolutional Networks for Semantic Segmentation](#) paper.

DeepLabV1, DeepLabV2, DeepLabV3, DeepLabV3+

<https://arxiv.org/pdf/1412.7062>

<https://arxiv.org/pdf/1606.00915>

DeepLabv1 + DeepLabv2

- It was the first to introduce atrous (or dilated) convolution. This technique allows for explicit control over the resolution of computed features and effectively enlarges the field of view of filters without increasing the number of parameters or computational cost. This enables the model to capture context at multiple scales.
- It uses a Deep Convolutional Neural Network (DCNN), specifically VGG-16 or ResNet-101, as its feature extractor.
- The Atrous Spatial Pyramid Pooling (ASPP) module. ASPP probes a convolutional feature layer with filters at multiple sampling rates, which allows for robust segmentation of objects at various scales.
- To sharpen the boundaries of segmented objects, it uses Conditional Random Fields (CRF) as a post-processing step.

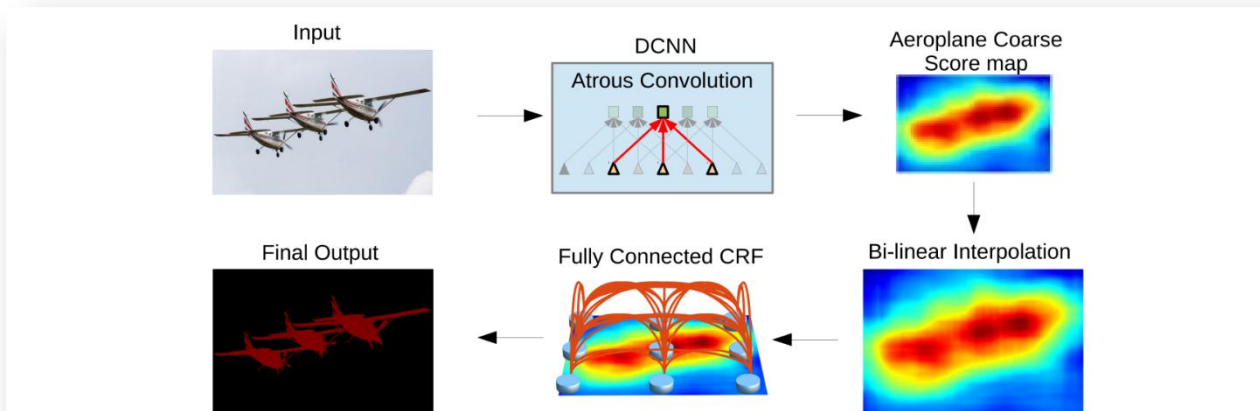


Fig. 1: Model Illustration. A Deep Convolutional Neural Network such as VGG-16 or ResNet-101 is employed in a fully convolutional fashion, using atrous convolution to reduce the degree of signal downsampling (from 32x down 8x). A bilinear interpolation stage enlarges the feature maps to the original image resolution. A fully connected CRF is then applied to refine the segmentation result and better capture the object boundaries.

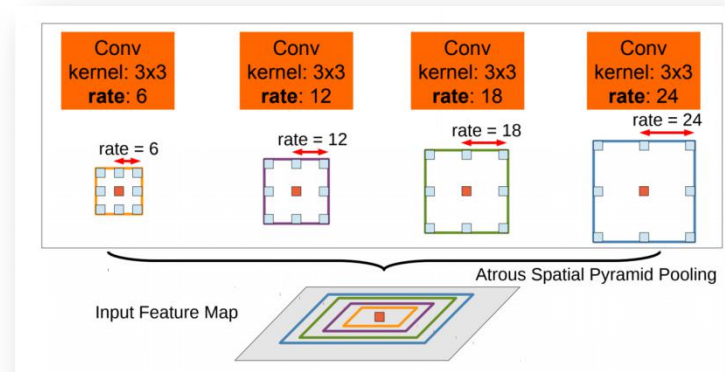


Fig. 4: Atrous Spatial Pyramid Pooling (ASPP). To classify the center pixel (orange), ASPP exploits multi-scale features by employing multiple parallel filters with different rates. The effective Field-Of-VIEWS are shown in different colors.

DeepLabV1, DeepLabV2, DeepLabV3, DeepLabV3+

<https://arxiv.org/pdf/1706.05587>

<https://arxiv.org/abs/1802.02611>

DeepLabv3 + DeepLabv3+

- **(DeepLabv3)** It improved the ASPP module by incorporating batch normalization and image-level features (via global average pooling) for even better multi-scale feature extraction. This refines the segmentation of objects of different sizes. Architecture: It utilizes deeper and more efficient backbone networks, such as a modified ResNet or Xception.
- **(DeepLabv3+)** It extends DeepLabv3 with a simple yet effective decoder module to refine segmentation results, particularly along object boundaries. This creates an encoder-decoder style architecture.

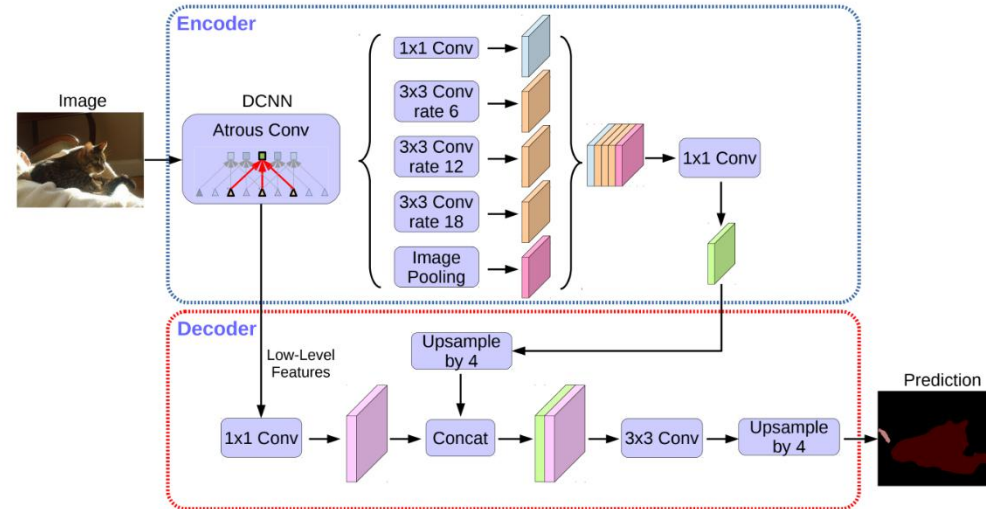


Fig. 2. Our proposed DeepLabv3+ extends DeepLabv3 by employing a encoder-decoder structure. The encoder module encodes multi-scale contextual information by applying atrous convolution at multiple scales, while the simple yet effective decoder module refines the segmentation results along object boundaries.

DeepLabV1, DeepLabV2, DeepLabV3, DeepLabV3+

Docs > Models and pre-trained weights > DeepLabV3

DeepLabV3

The DeepLabV3 model is based on the [Rethinking Atrous Convolution for Semantic Image Segmentation](#) paper.

- WARNING**

The segmentation module is in Beta stage, and backward compatibility is not guaranteed.

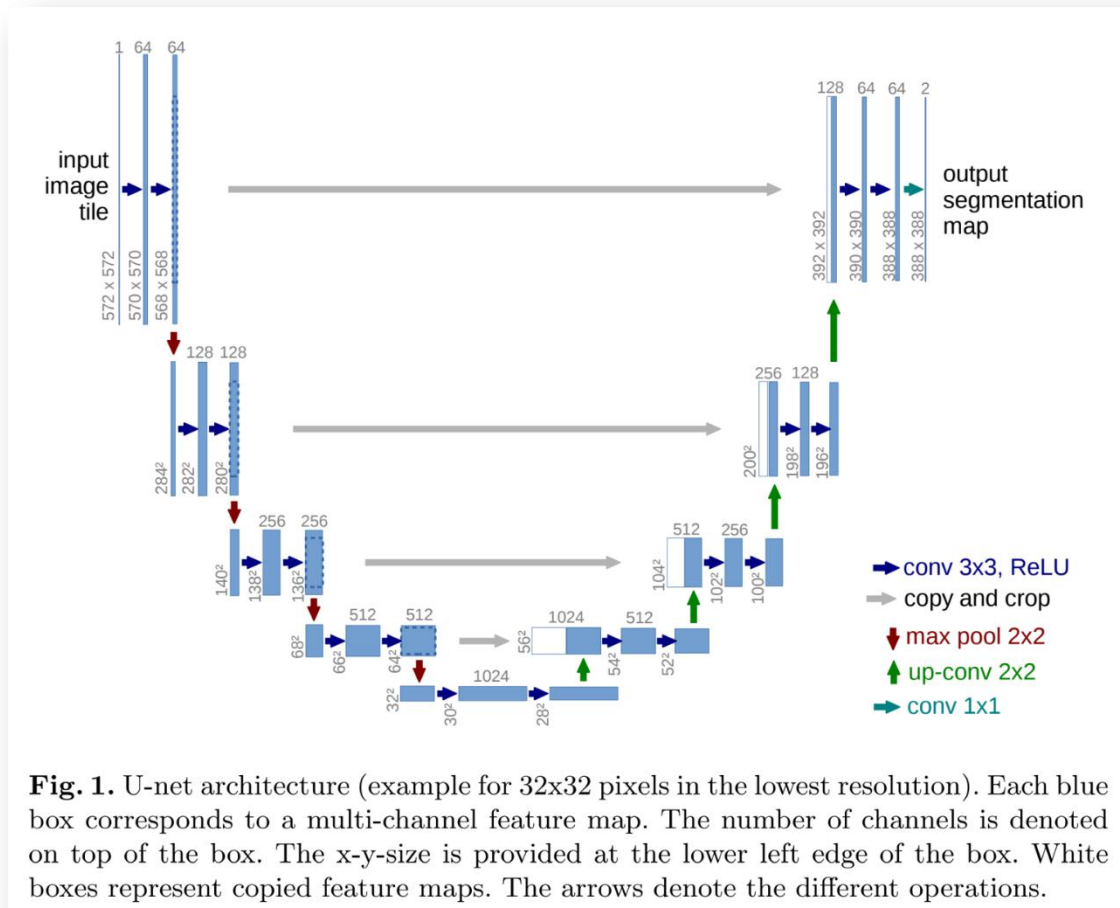
Model builders

The following model builders can be used to instantiate a DeepLabV3 model with different backbones, with or without pre-trained weights. All the model builders internally rely on the `torchvision.models.segmentation.deeplabv3.DeepLabV3` base class. Please refer to the [source code](#) for more details about this class.

<code>deeplabv3_mobilenet_v3_large(*[, weights, ...])</code>	Constructs a DeepLabV3 model with a MobileNetV3-Large backbone.
<code>deeplabv3_resnet50(*[, weights, progress, ...])</code>	Constructs a DeepLabV3 model with a ResNet-50 backbone.
<code>deeplabv3_resnet101(*[, weights, progress, ...])</code>	Constructs a DeepLabV3 model with a ResNet-101 backbone.

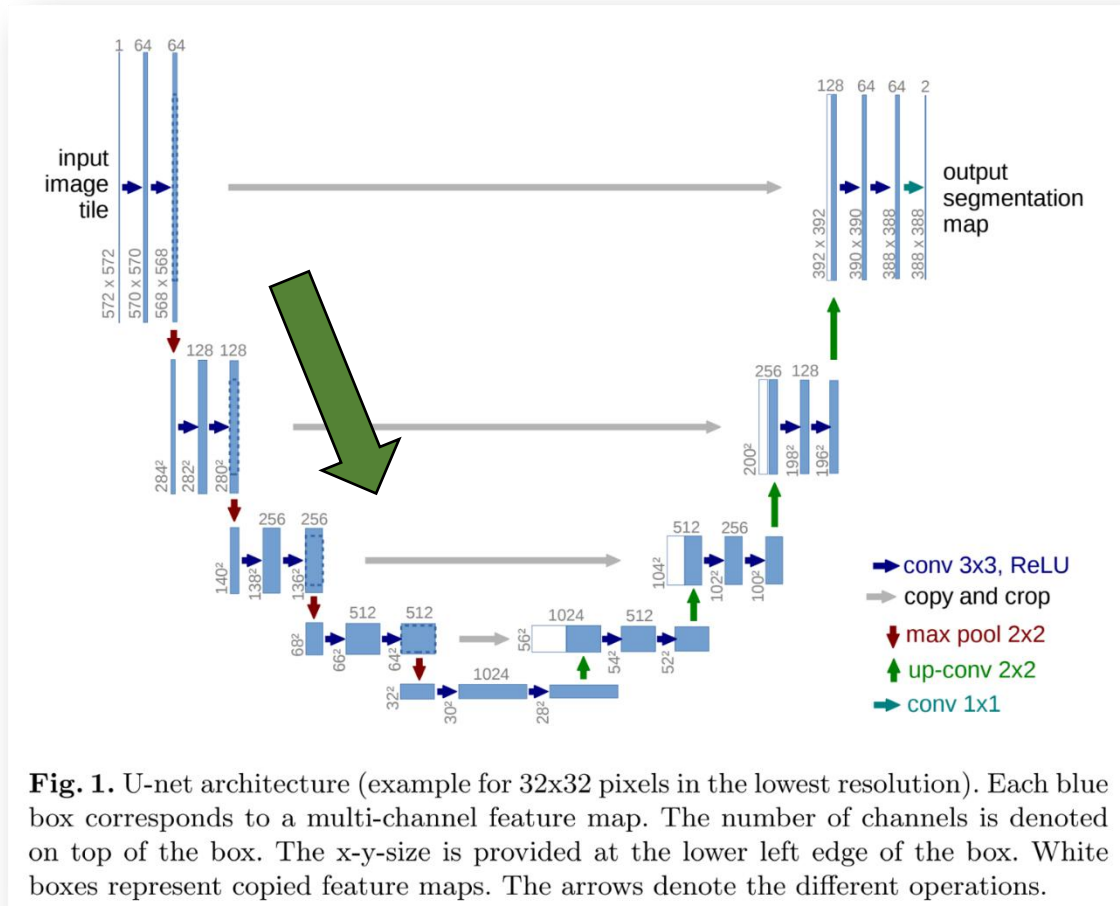
U-Net

<https://arxiv.org/pdf/1505.04597>



U-Net

<https://arxiv.org/pdf/1505.04597>



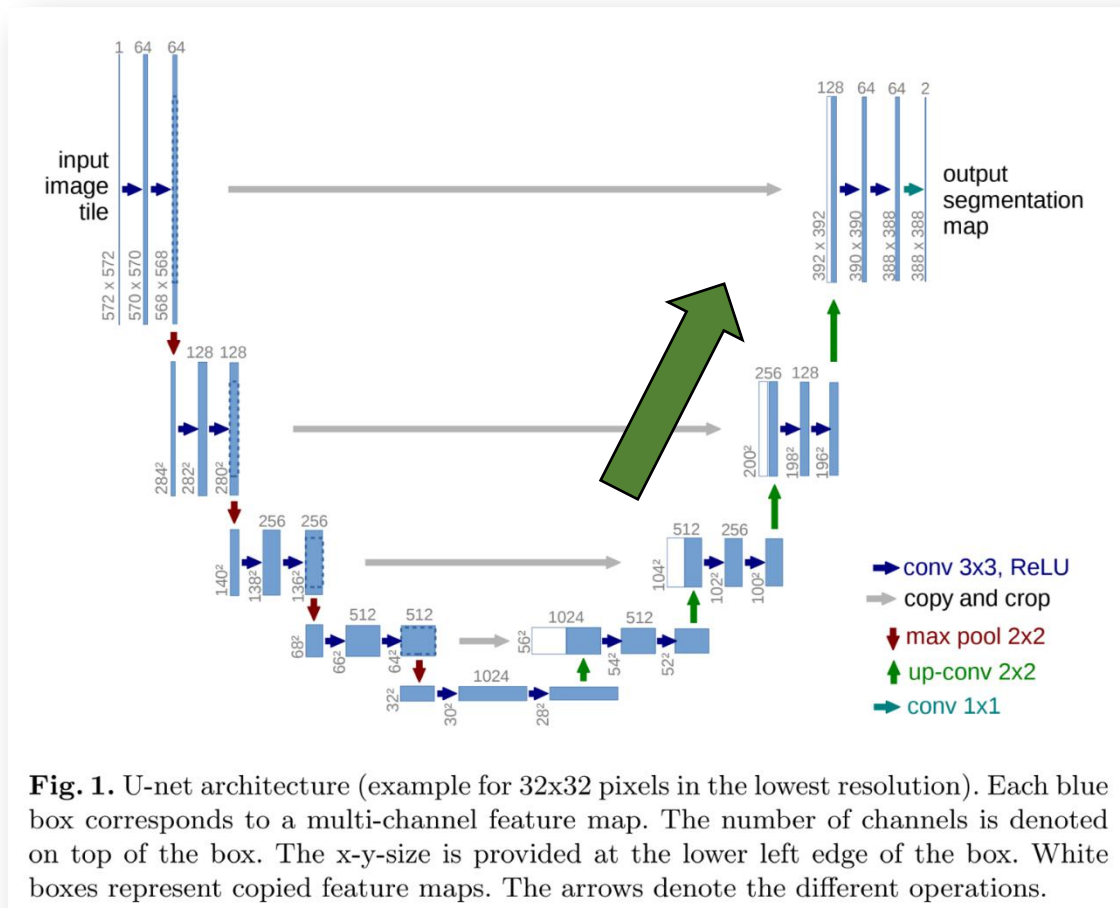
2 Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net

<https://arxiv.org/pdf/1505.04597>



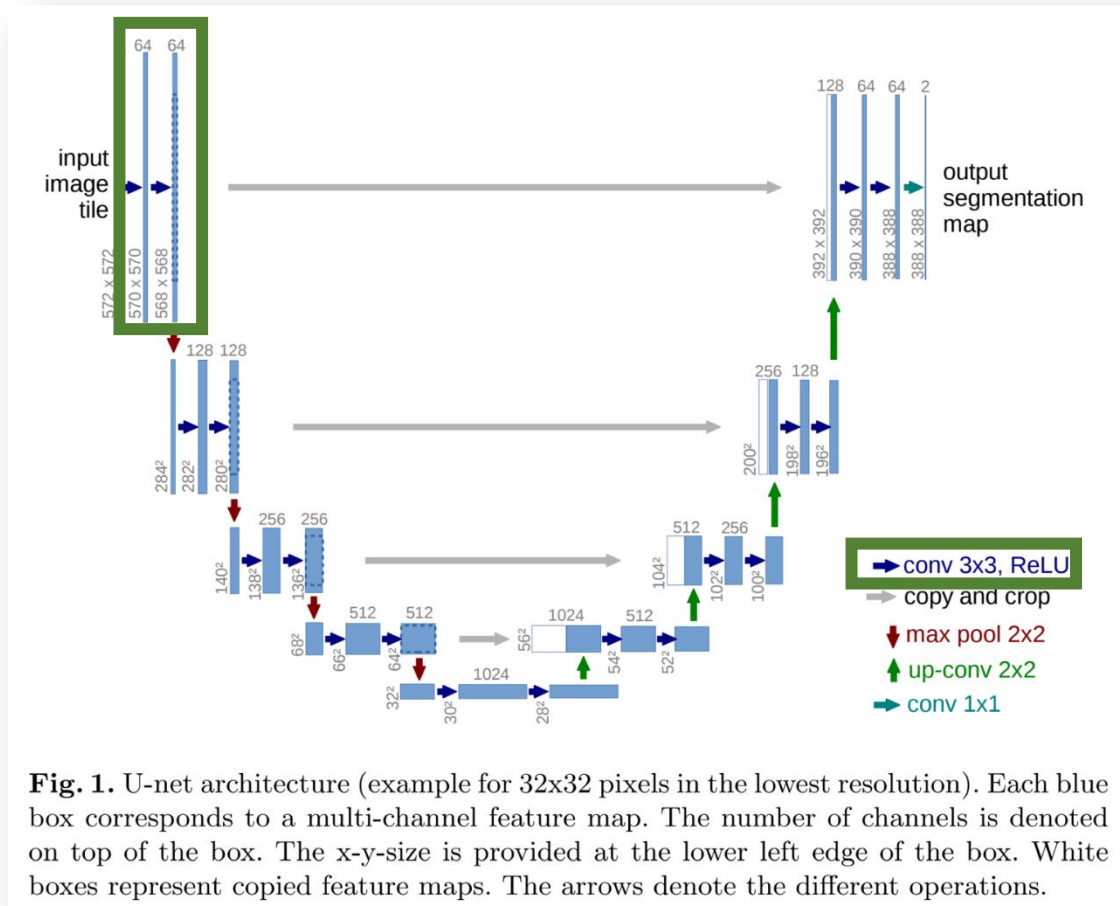
2 Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net

<https://arxiv.org/pdf/1505.04597>



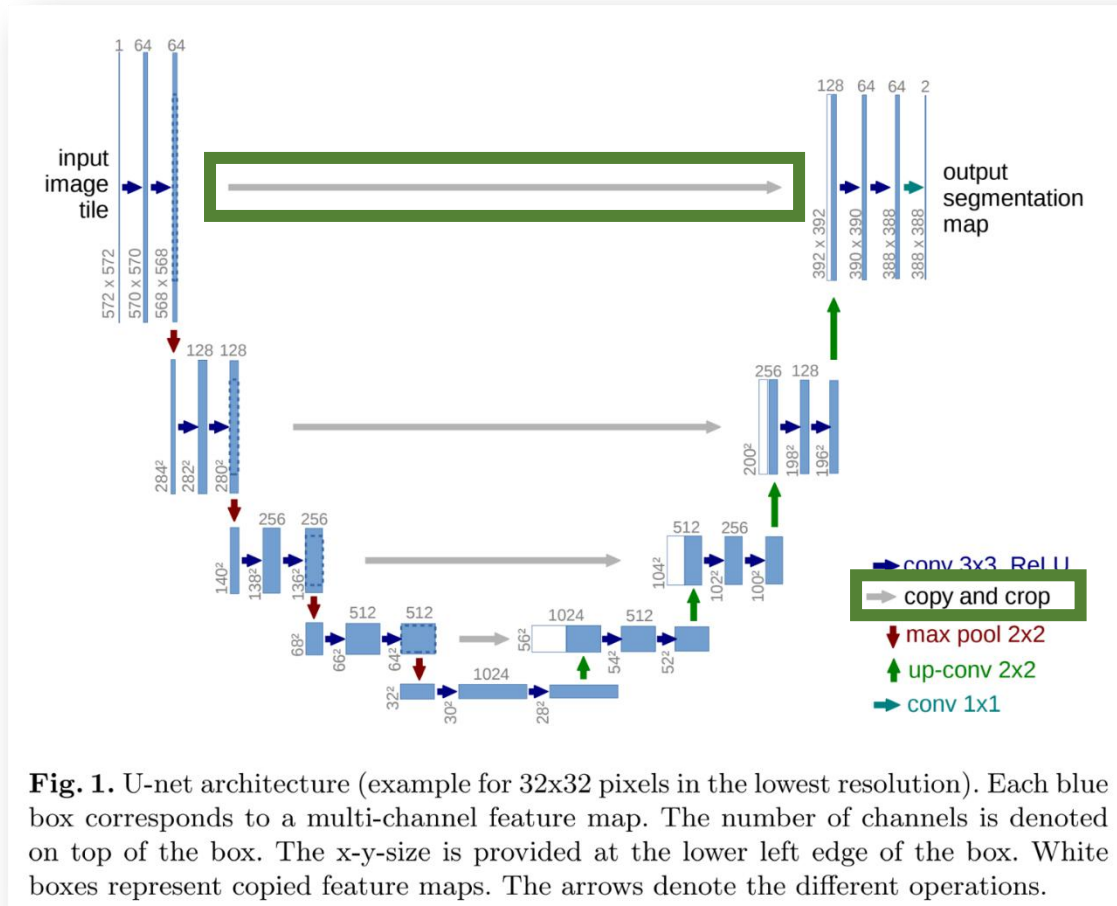
2 Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions) each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net

<https://arxiv.org/pdf/1505.04597>



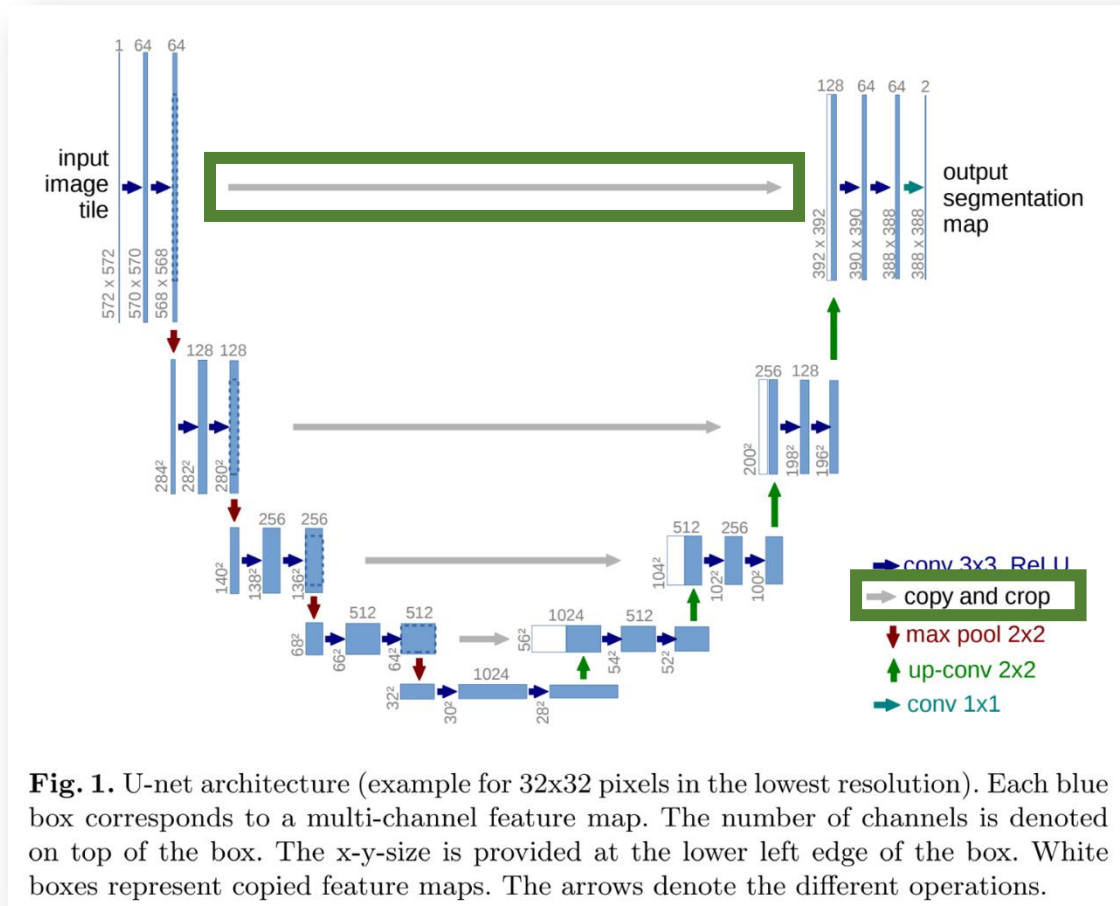
2 Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a **concatenation** with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net

<https://arxiv.org/pdf/1505.04597>



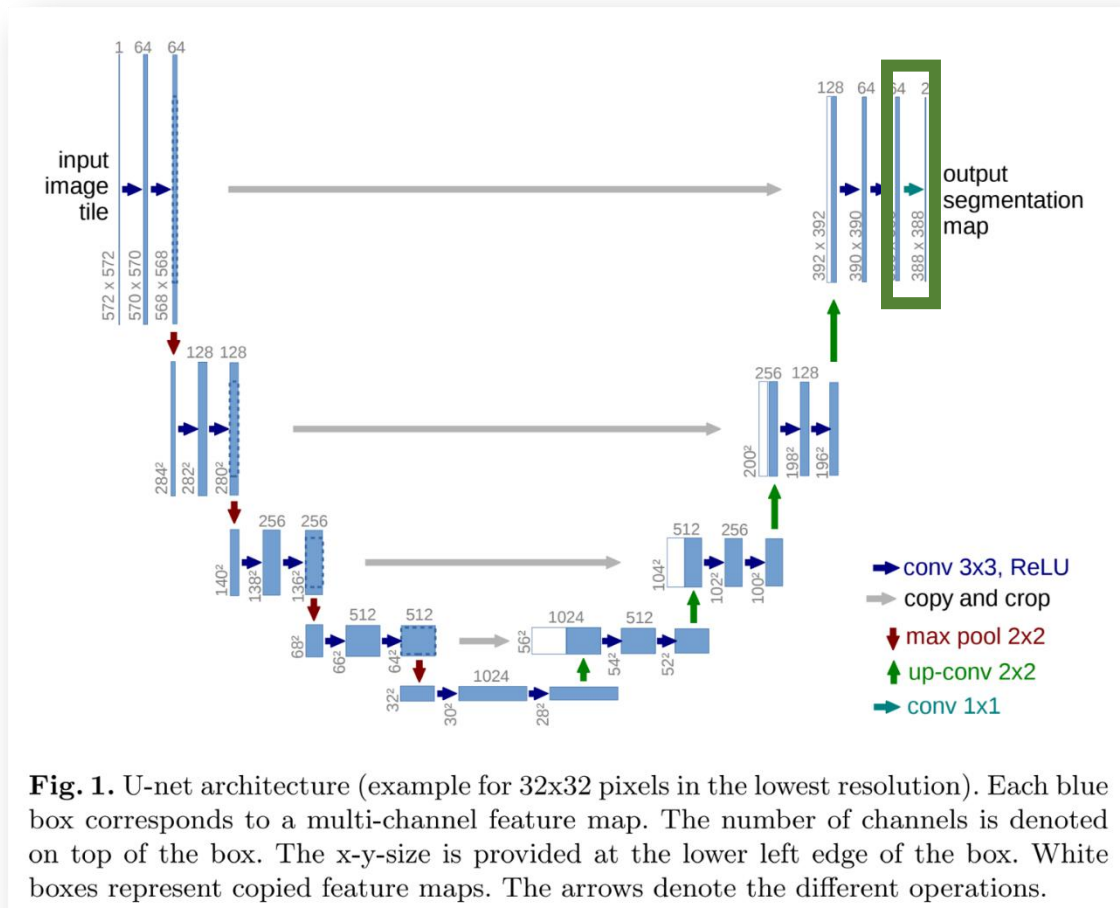
2 Network Architecture

The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a **concatenation** with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net

<https://arxiv.org/pdf/1505.04597>



2 Network Architecture

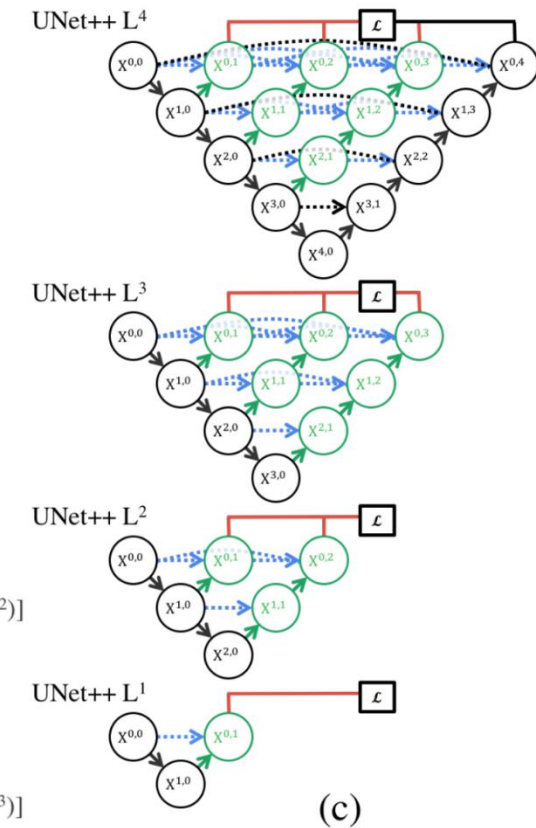
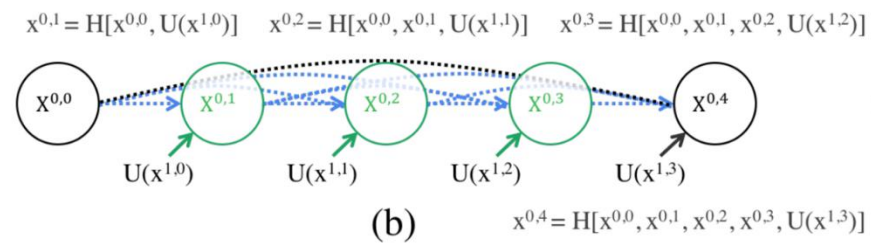
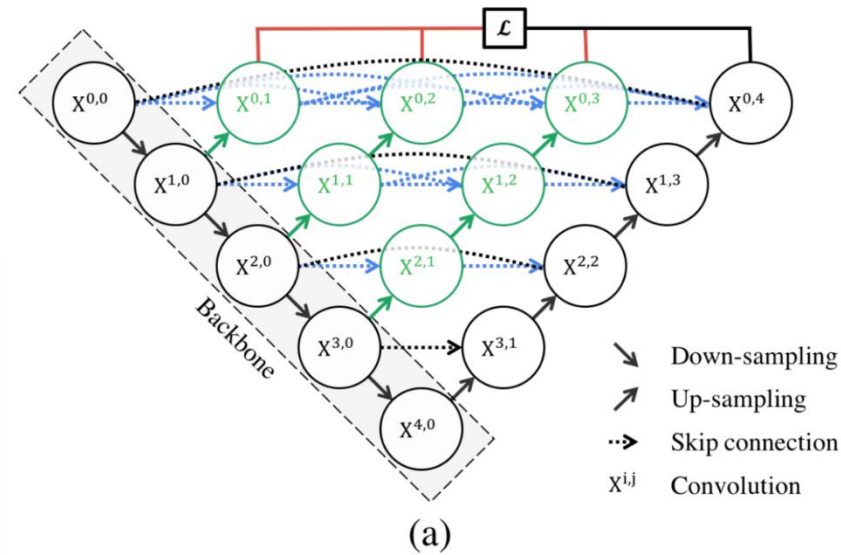
The network architecture is illustrated in Figure 1. It consists of a contracting path (left side) and an expansive path (right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step we double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution (“up-convolution”) that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

To allow a seamless tiling of the output segmentation map (see Figure 2), it is important to select the input tile size such that all 2x2 max-pooling operations are applied to a layer with an even x- and y-size.

U-Net ++

<https://arxiv.org/abs/1807.10165>

Fig. 1: (a) UNet++ consists of an encoder and decoder that are connected through a series of nested dense convolutional blocks. The main idea behind UNet++ is to bridge the semantic gap between the feature maps of the encoder and decoder prior to fusion. For example, the semantic gap between $(X^{0,0}, X^{1,3})$ is bridged using a dense convolution block with three convolution layers. In the graphical abstract, black indicates the original U-Net, green and blue show dense convolution blocks on the skip pathways, and red indicates deep supervision. Red, green, and blue components distinguish UNet++ from U-Net. (b) Detailed analysis of the first skip pathway of UNet++. (c) UNet++ can be pruned at inference time, if trained with deep supervision.

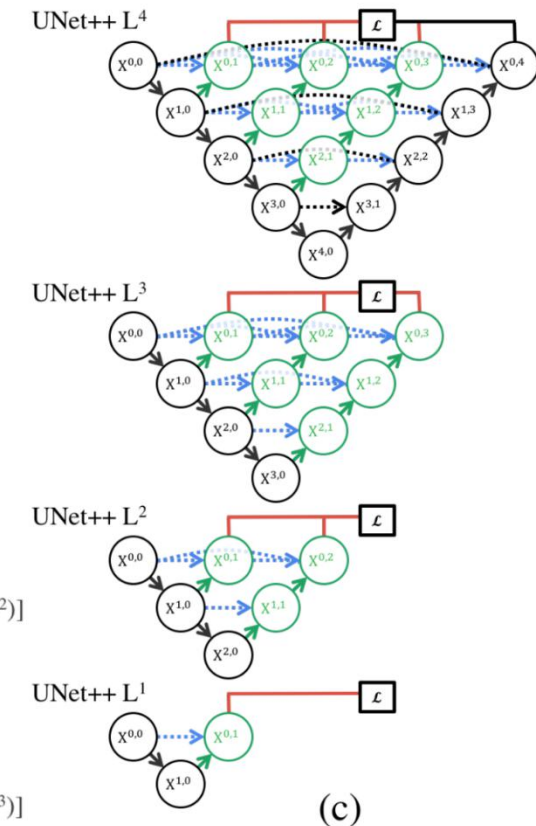
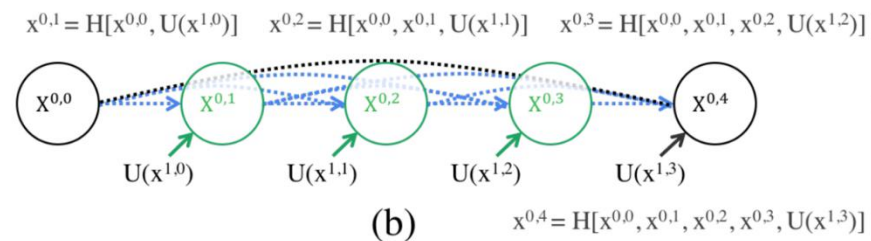
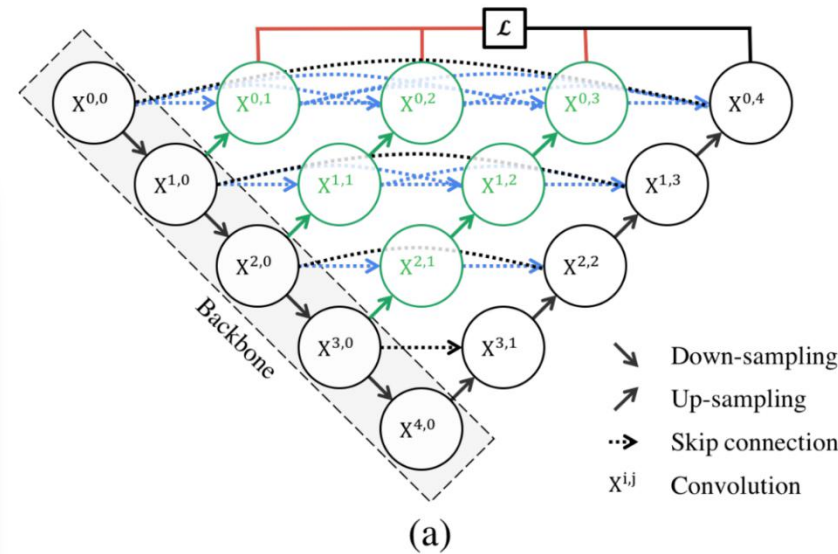


U-Net ++

<https://arxiv.org/abs/1807.10165>

3.1 Re-designed skip pathways

Re-designed skip pathways transform the connectivity of the encoder and decoder sub-networks. In U-Net, the feature maps of the encoder are directly received in the decoder; however, in UNet++, they undergo a dense convolution block whose number of convolution layers depends on the pyramid level. For example, the skip pathway between nodes $X^{0,0}$ and $X^{1,3}$ consists of a dense convolution block with three convolution layers where each convolution layer is preceded by a concatenation layer that fuses the output from the previous convolution layer of the same dense block with the corresponding up-sampled output of the lower dense block. Essentially, the dense convolution block brings the semantic level of the encoder feature maps closer to that of the feature maps awaiting in the decoder. The hypothesis is that the optimizer would face an easier optimization problem when the received encoder feature maps and the corresponding decoder feature maps are semantically similar.



No New-Net

<https://arxiv.org/pdf/1809.10483>

2.2 Network architecture

U-Net [14] is a successful encoder-decoder network that has received a lot of attention in the recent years. Its encoder part works similarly to a traditional classification CNN in that it successively aggregates semantic information at the expense of reduced spatial information. Since in segmentation, both semantic as well as spatial information are crucial for the success of a network, the missing spatial information must somehow be recovered. U-Net does this through the decoder, which receives semantic information from the bottom of the 'U' (see Fig. 1) and recombines it with higher resolution feature maps obtained directly from the encoder through skip connections. Unlike other segmentation networks, such as FCN [12] and previous iterations of DeepLab [13] this allows U-Net to segment fine structures particularly well.

Our network architecture is an instantiation of the 3D U-Net [15] with minor modifications. Following our successful participation in 2017 [6], we stick with our design choice to process patches of size $128 \times 128 \times 128$ with a batch size of two. Due to the high memory consumption of 3D convolutions with large patch sizes, we implemented our network carefully to still allow for an adequate number of feature maps. By reducing the number of filters right before upsampling and by using inplace operations whenever possible, this results in a network with 30 feature channels at the highest resolution, which is nearly double the number we could train with in our previous model (using the same 12 GB NVIDIA Titan X GPU). Due to our choice of loss function, traditional ReLU activation functions did not reliably produce the desired results, which is why we replaced them with leaky ReLUs (leakiness 10^{-2}) throughout the entire network. With a small batch size of 2, the exponential moving averages of mean and variance within a batch learned by batch normalization [24] are unstable and do not reflect the feature map activations at test time very well. We found instance normalization [23] to provide more consistent results and therefore used it to normalize all feature map activations (between convolution and nonlinearity). For an overview over our segmentation architecture, please refer to Fig. 1.

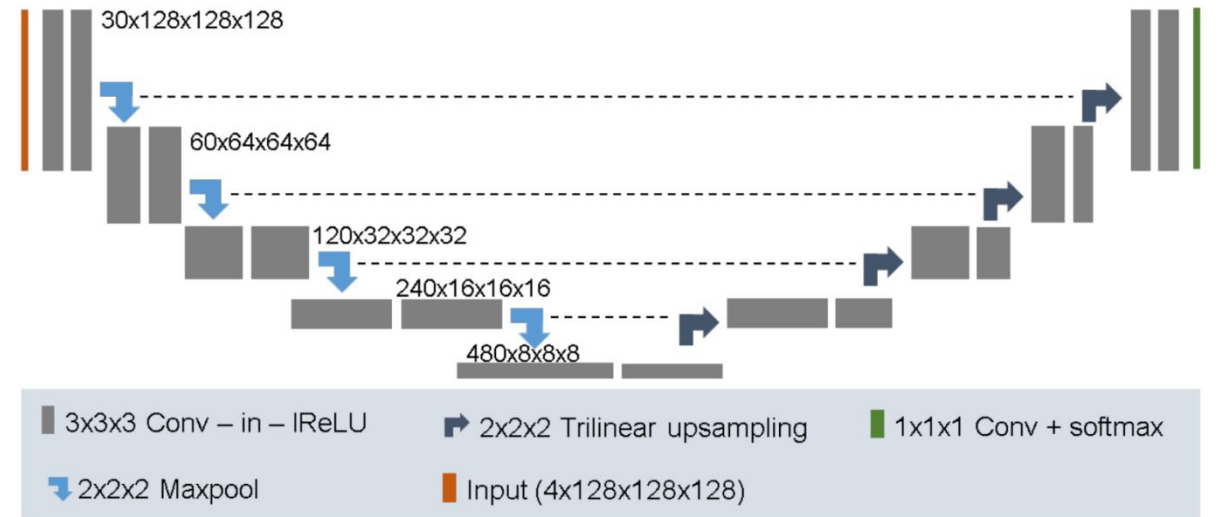


Fig. 1. We use a 3D U-Net architecture with minor modifications. It uses instance normalization [23] and leaky ReLU nonlinearities and reduces the number of feature maps before upsampling. Feature map dimensionality is noted next to the convolutional blocks, with the first number being the number of feature channels.

SegFormer

<https://arxiv.org/pdf/2105.15203>

SegFormer: Simple and Efficient Design for Semantic Segmentation with Transformers

Enze Xie¹, Wenhai Wang², Zhiding Yu³, Anima Anandkumar^{3,4}, Jose M. Alvarez³, Ping Luo¹

¹The University of Hong Kong ²Nanjing University ³NVIDIA ⁴Caltech

Abstract

We present SegFormer, a simple, efficient yet powerful semantic segmentation framework which unifies Transformers with lightweight multilayer perceptron (MLP) decoders. SegFormer has two appealing features: 1) SegFormer comprises a novel hierarchically structured Transformer encoder which outputs multiscale features. It does not need positional encoding, thereby avoiding the interpolation of positional codes which leads to decreased performance when the testing resolution differs from training. 2) SegFormer avoids complex decoders. The proposed MLP decoder aggregates information from different layers, and thus combining both local attention and global attention to render powerful representations. We show that this simple and lightweight design is the key to efficient segmentation on Transformers. We scale our approach up to obtain a series of models from SegFormer-B0 to SegFormer-B5, reaching significantly better performance and efficiency than previous counterparts. For example, SegFormer-B4 achieves 50.3% mIoU on ADE20K with 64M parameters, being $5\times$ smaller and 2.2% better than the previous best method. Our best model, SegFormer-B5, achieves 84.0% mIoU on Cityscapes validation set and shows excellent zero-shot robustness on Cityscapes-C. Code will be released at: github.com/NVlabs/SegFormer.

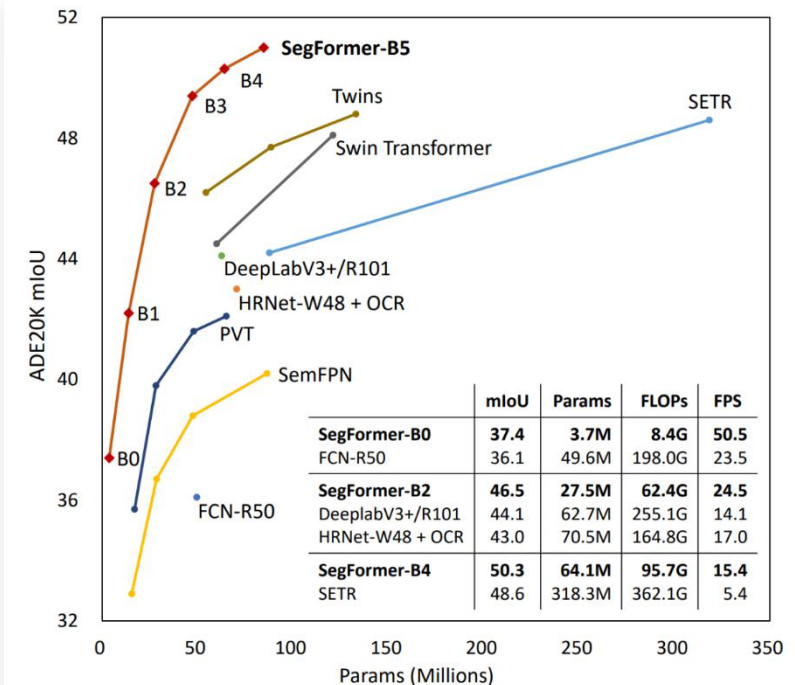


Figure 1: **Performance vs. model efficiency on ADE20K.** All results are reported with single model and single-scale inference. SegFormer achieves a new state-of-the-art 51.0% mIoU while being significantly more efficient than previous methods.

SegFormer

<https://arxiv.org/pdf/2105.15203>

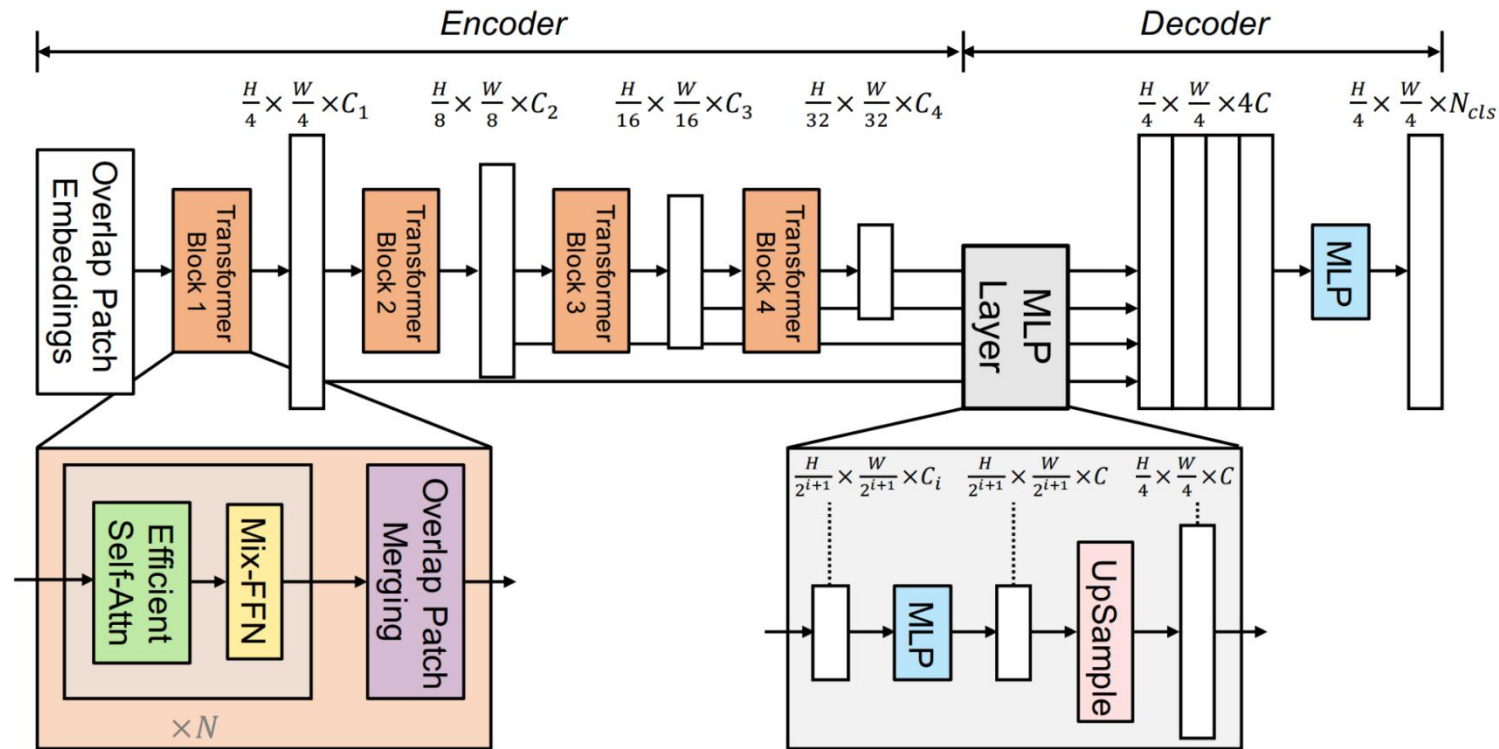


Figure 2: **The proposed SegFormer framework** consists of two main modules: A hierarchical Transformer encoder to extract coarse and fine features; and a lightweight All-MLP decoder to directly fuse these multi-level features and predict the semantic segmentation mask. “FFN” indicates feed-forward network.

Where to go from here?

https://github.com/qubvel-org/segmentation_models.pytorch

<https://pypi.org/project/segmentation-models-pytorch/>

segmentation-models-pytorch 0.5.0

[pip install segmentation-models-pytorch](#)

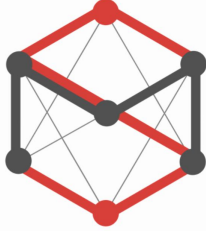
Released: Apr 17, 2025

Image segmentation models with pre-trained backbones. PyTorch.

Navigation

- Project description
- Release history
- Download files

Project description



Segmentation Models

Python library with Neural Networks for Image Semantic Segmentation based on [PyTorch](#).

BUILD	PASSING	COVERAGE	78%	DOCS	PASSING
PYPI	V0.5.0	PYTORCH	1.9+	PYTHON	3.9+
LICENSE	MIT	DOWNLOADS	257K/MONTH		

The main features of the library are:

- Super simple high-level API (just two lines to create a neural network)
- 12 encoder-decoder model architectures (Unet, Unet++, Segformer, DPT, ...)
- 800+ pretrained convolution- and transform-based encoders, including [timm](#) support
- Popular metrics and losses for training routines (Dice, Jaccard, Tversky, ...)
- ONNX export and torch script/trace/compile friendly

Verified details

These details have been [verified by PyPI](#)


Project links

- Homepage

GitHub Statistics

- Repository
- Stars: 10624
- Forks: 1758
- Open issues: 59
- Open PRs: 6

Maintainers

-  qubvel

Models and encoders

Architectures

Architecture	Paper	Documentation	Checkpoints
Unet	paper	docs	
Unet++	paper	docs	
MAnet	paper	docs	
Linknet	paper	docs	
FPN	paper	docs	
PSPNet	paper	docs	
PAN	paper	docs	
DeepLabV3	paper	docs	
DeepLabV3+	paper	docs	
UPerNet	paper	docs	checkpoints
Segformer	paper	docs	checkpoints
DPT	paper	docs	checkpoints