Introduction to image segmentation

DEEP LEARNING FOR IMAGES WITH PYTORCH

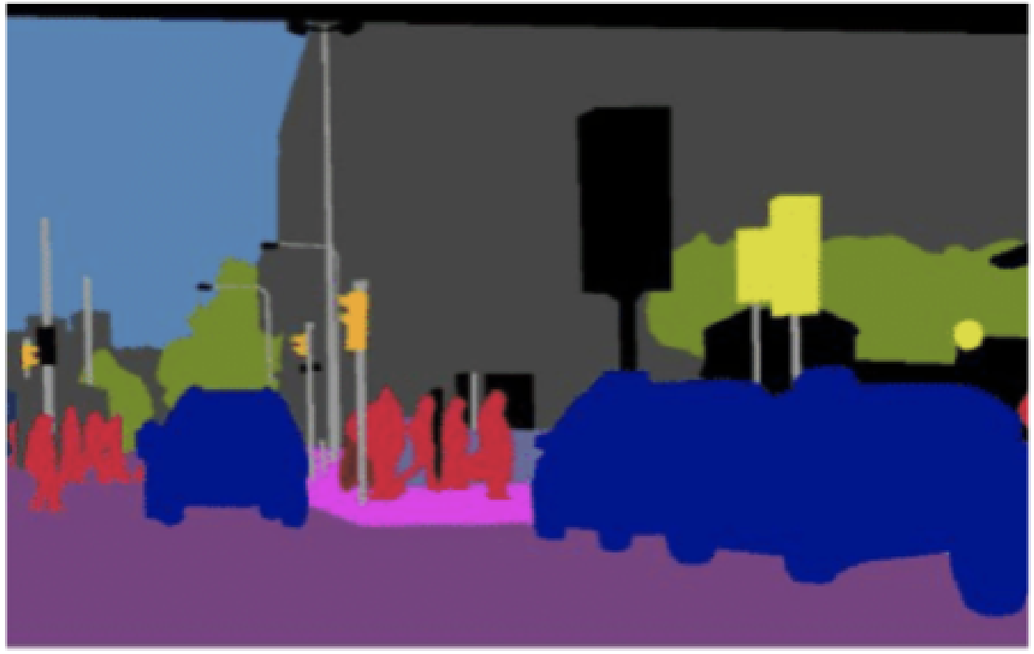
Michal Oleszak
Machine Learning Engineer

datacamp

Image segmentation

- **Image segmentation** partitions the image into multiple segments on the pixel level
- Each pixel in an image is assigned to a particular segment
- Three types of segmentation:
  - Semantic segmentation
  - Instance segmentation
  - Panoptic segmentation

Semantic segmentation



- Each pixel classified into a class
- All pixels belonging to the same class are treated equally

Instance segmentation

- Distinguishes between different instances of the same class
- Background often not segmented

Panoptic segmentation

- Combines semantic and instance segmentations
- Assigns a unique label to each instance of an object
- Classifies background at pixel level

Data annotations

```
image = Image.open("images/British_Shorthair_36.jpg")
mask = Image.open("annots/British_Shorthair_36.png")

transform = transforms.Compose([
  transforms.ToTensor()
])
image_tensor = transform(image)
mask_tensor = transform(mask)

print(f"""Image shape: {image_tensor.shape}
          Mask shape: {mask_tensor.shape}""")
```

```
Image shape: torch.Size([3, 333, 500])
Mask shape: torch.Size([1, 333, 500])
```

Understanding the mask

- Dataset documentation:

  > Pixel Annotations: 1: Foreground 2: Background 3: Not classified

- Unique mask values:

```
mask_tensor.unique()
```

```
tensor([0.0039, 0.0078, 0.0118])
```

- Pixel values are divided by `255` :
  - `1 / 255 = 0.0039` - object
  - `2 / 255 = 0.0.0078` - background
  - `3 / 255 = 0.0118` - unclassified

Creating a binary mask

```
binary_mask = torch.where(
    mask_tensor == 1/255,
    torch.tensor(1.0),
    torch.tensor(0.0),
)

to_pil_image = transforms.ToPILImage()
mask = to_pil_image(binary_mask)
plt.imshow(mask)
```

- `torch.where()` :
  - Condition
  - Value to use if condition met
  - Value to use otherwise
- Transform mask to PIL image
- Display mask image

Segmenting the object

```
object_tensor = image_tensor * binary_mask

to_pil_image = transforms.ToPILImage()
object_image = to_pil_image(object_tensor)

plt.imshow(object_image)
```

- Multiply image with the binary mask
- Transform object to PIL image
- Display object image

Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH

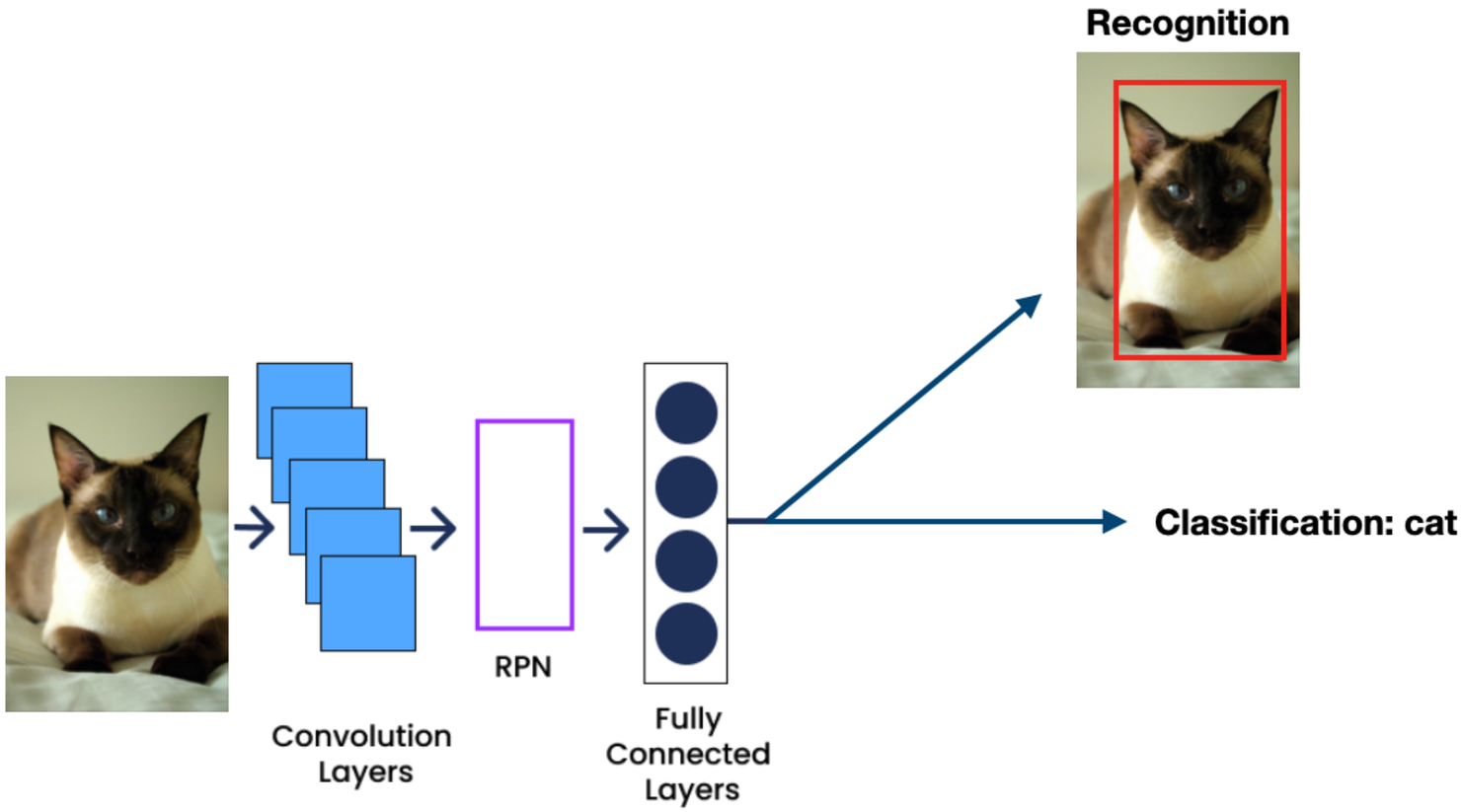
Instance segmentation with Mask R-CNN

DEEP LEARNING FOR IMAGES WITH PYTORCH

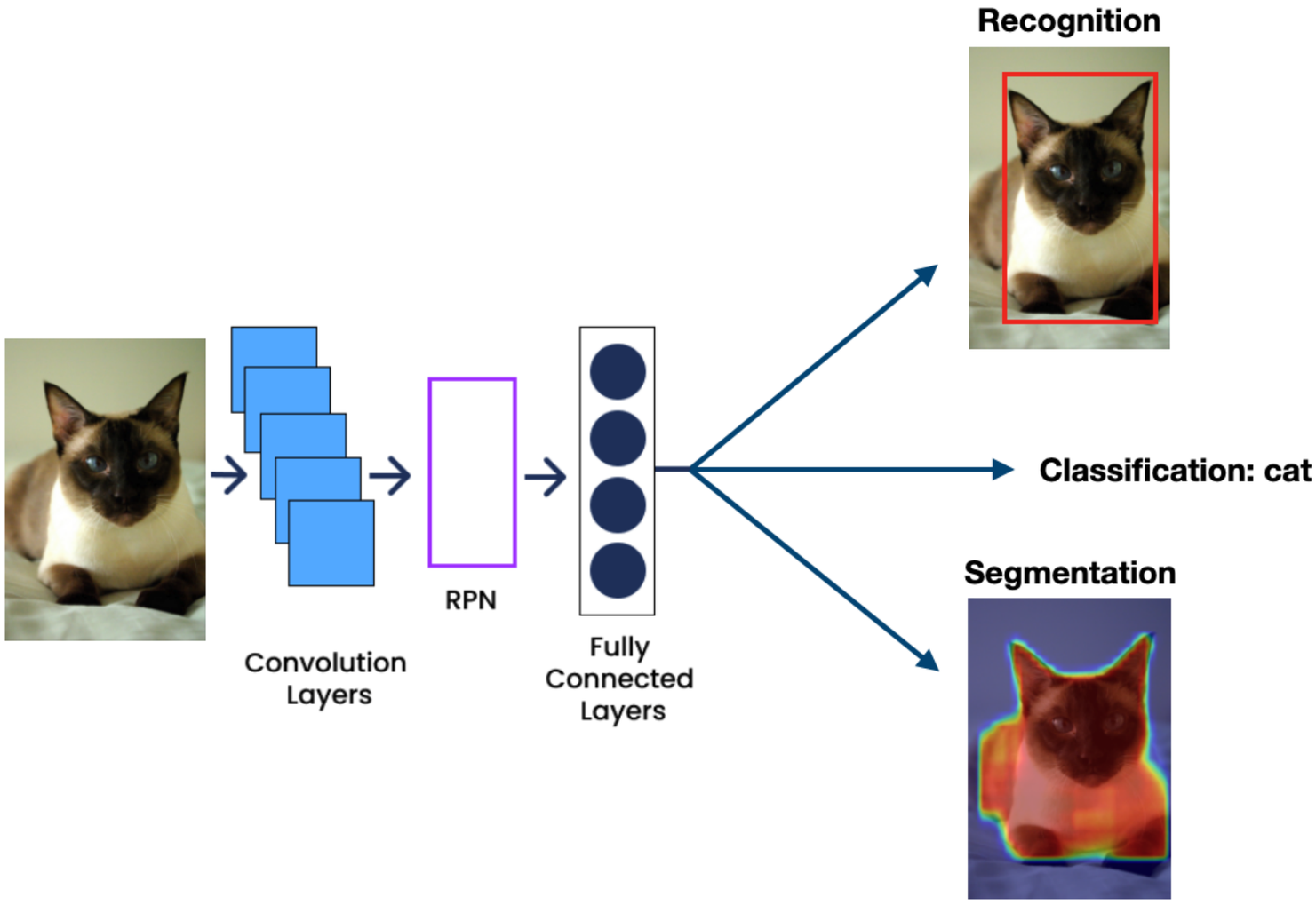
Michal Oleszak
Machine Learning Engineer

datacamp

Faster R-CNN


Recognition
Classification: cat
RPN
Convolution Layers
Fully Connected Layers

Mask R-CNN


Recognition
Classification: cat
Segmentation
RPN
Convolution Layers
Fully Connected Layers

Pre-trained Masked R-CNN in PyTorch

```
from torchvision.models.detection import \
maskrcnn_resnet50_fpn

model = maskrcnn_resnet50_fpn(pretrained=True)
model.eval()

image = Image.open("cat_and_laptop.jpg")
transform = transforms.Compose([
    transforms.ToTensor()
])
image_tensor = transform(image).unsqueeze(0)

with torch.no_grad():
    prediction = model(image_tensor)
```

- Import the Mask R-CNN model
- Load pre-trained model
- Load test image and transform to tensor

- Pass image tensor to the model

Model outputs

- Labels

```
prediction[0]["labels"]
```

```
tensor([
    17, 73, 76, 73, 67, 42, 63, 84,73, 65,
    17, 73, 73, 73, 84, 72, 76, 76,17, 15
])
```

- Class names

```
print(class_names[17], class_names[73])
```

```
cat laptop
```

- Class probabilities

```
prediction[0]["scores"]
```

```
tensor([
    0.9981, 0.9672, 0.9061, 0.6893, 0.3729,
    ...,
    0.0745, 0.0705, 0.0623, 0.0610, 0.0508
])
```

- Masks

```
prediction[0]["masks"]
```

```
tensor([[[[0., 0., 0.,  ..., 0., 0., 0.],
          ...]]]])
```

Soft masks

- Unique mask values

```
prediction[0]["masks"].unique()
```

```
tensor([0.0000e+00, 5.9713e-08, ...,
        9.9989e-01, 9.9990e-01])
```

- Mask R-CNN masks:
  - Values between 0 and 1
  - Represent the model's confidence that each pixel belongs to the object
  - Provide more nuanced information than binary masks
  - Can be binarized by thresholding if needed

Displaying soft masks

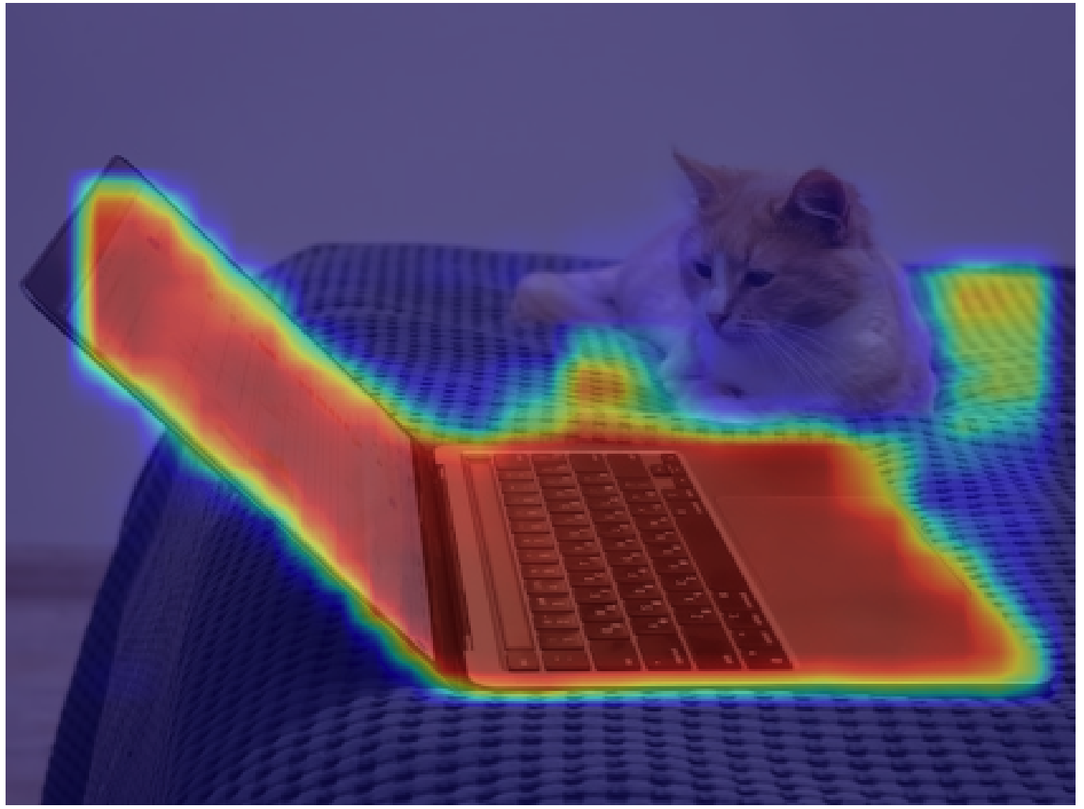
```
masks = prediction[0]["masks"]
labels = prediction[0]["labels"]

for i in range(2):
    plt.imshow(image)
    plt.imshow(
        masks[i, 0],
        cmap="jet",
        alpha=0.5,
    )
    plt.title(
        f"Object: {class_names[labels[i]]}"
    )
    plt.show()
```

- Extract masks and labels from prediction
- Iterate over top two objects, plotting the original image
- For each object, plot the semi-transparent mask
- Add title and display

Displaying soft masks

Object: cat

Object: laptop



Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH

Semantic segmentation with U-Net

DEEP LEARNING FOR IMAGES WITH PYTORCH

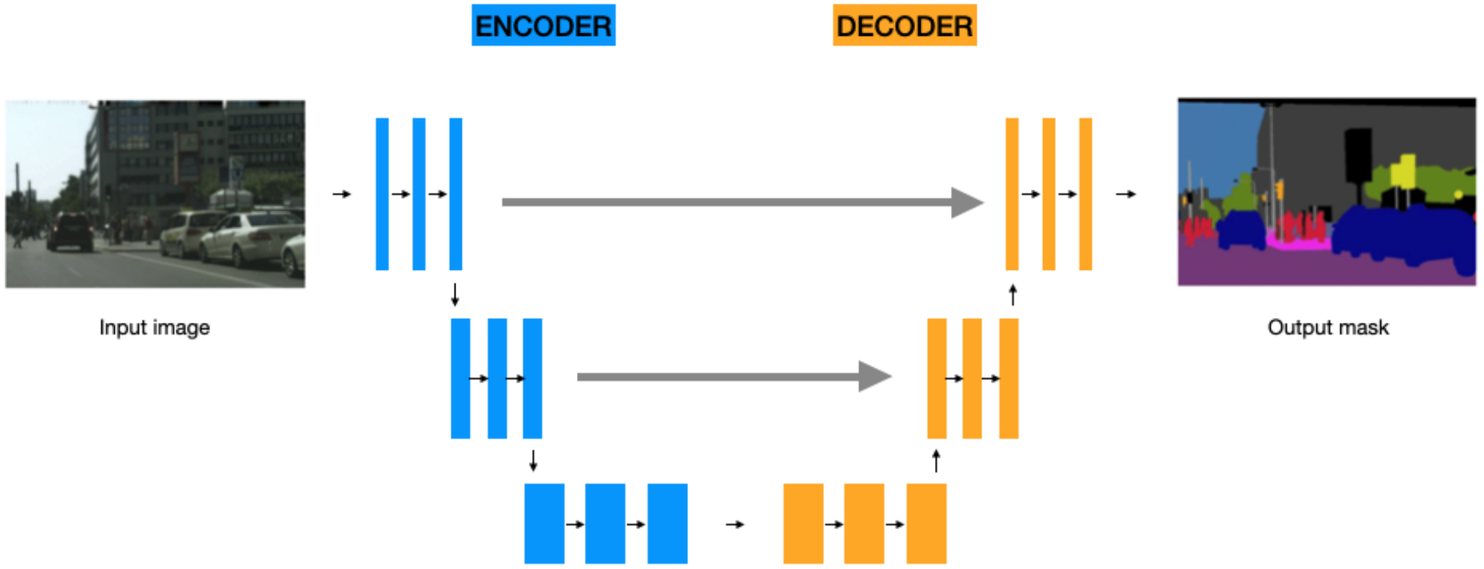
Michal Oleszak
Machine Learning Engineer

datacamp

Semantic segmentation

- No distinction between different instances of the same class
- Useful for medical imaging or satellite image analysis
- Popular architecture: **U-Net**

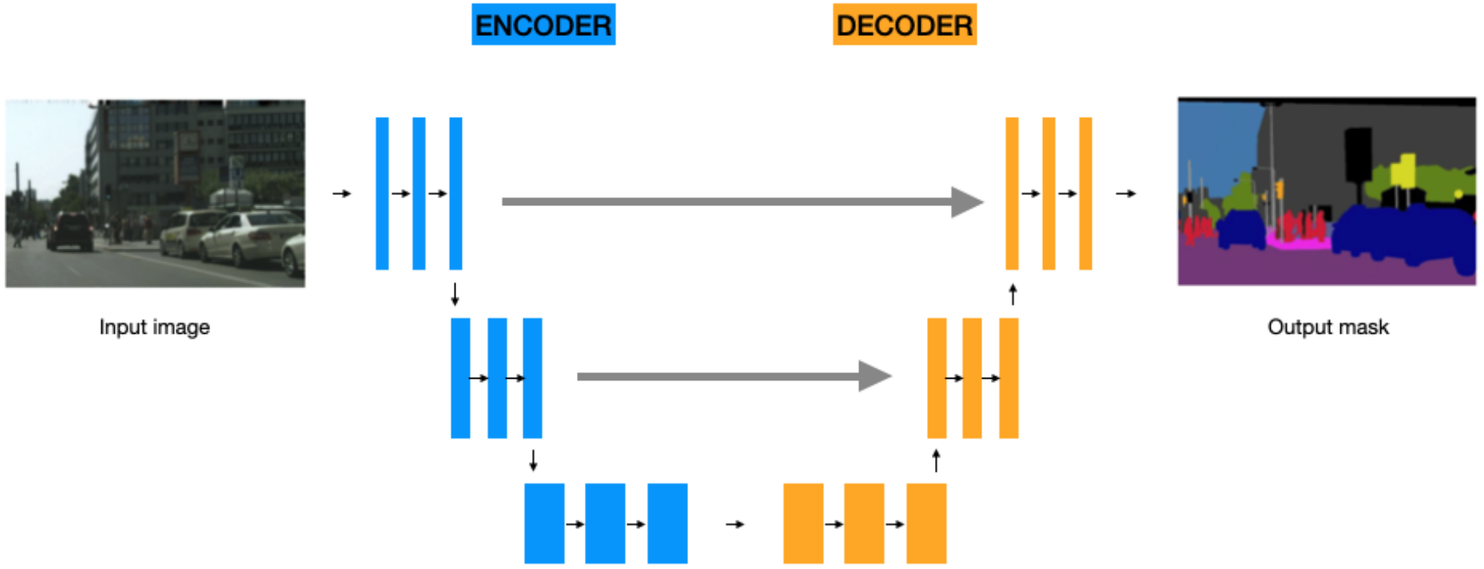
U-Net architecture



Encoder:

- Convolutional and pooling layers
- Downsampling: reduces spatial dimensions while increasing depth

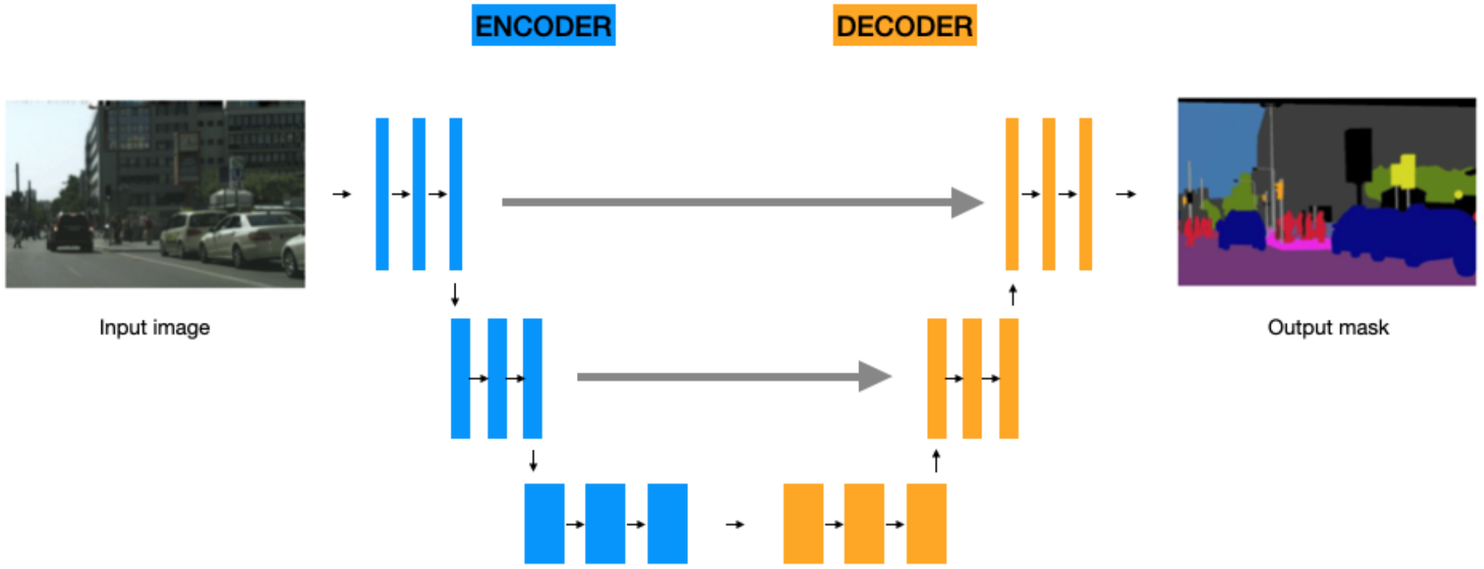
U-Net architecture



Decoder:

- Symmetric to the encoder
- Upsamples feature maps with transposed convolutions

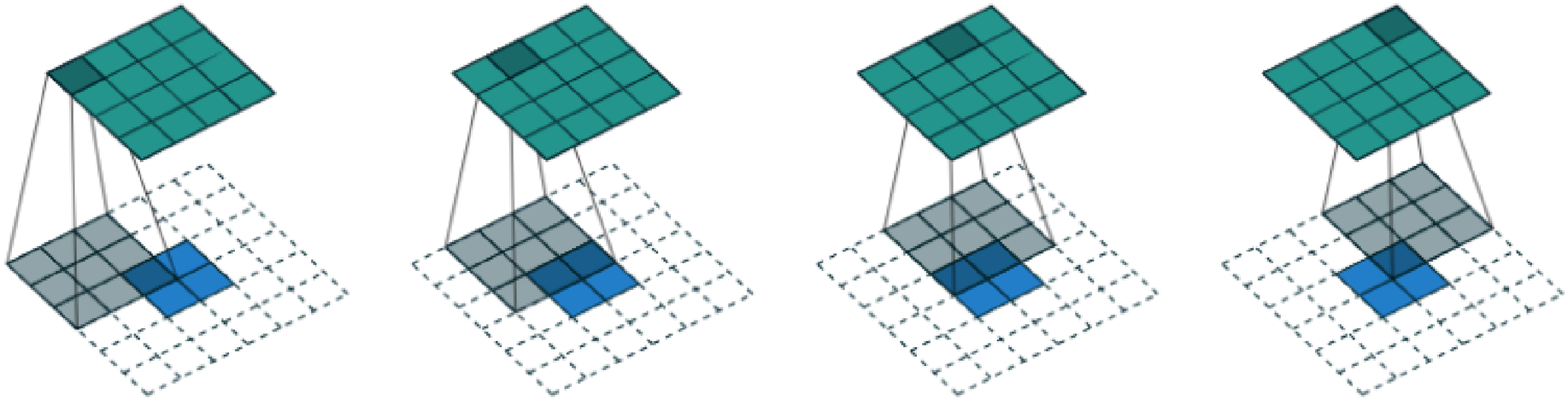
U-Net architecture



Skip connections:

- Links from the encoder to the decoder
- Preserve details lost in downsampling

Transposed convolution



- Upsamples feature maps in the decoder: increases height and width while reducing depth
- Transposed convolution process:
  1. Insert zeros between or around the input feature map
  2. Perform a regular convolution on the zero-padded input

Transposed convolution in PyTorch

```
import torch.nn as nn


upsample = nn.ConvTranspose2d(
    in_channels=in_channels,
    out_channels=out_channels,
    kernel_size=2,
    stride=2,
)
```

U-Net: layer definitions

```
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        self.enc1 = self.conv_block(in_channels, 64)
        self.enc2 = self.conv_block(64, 128)
        self.enc3 = self.conv_block(128, 256)
        self.enc4 = self.conv_block(256, 512)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.upconv3 = nn.ConvTranspose2d(512, 256,
                                          kernel_size=2, stride=2)
        self.upconv2 = nn.ConvTranspose2d(256, 128,
                                          kernel_size=2, stride=2)
        self.upconv1 = nn.ConvTranspose2d(128, 64,
                                          kernel_size=2, stride=2)

        self.dec1 = self.conv_block(512, 256)
        self.dec2 = self.conv_block(256, 128)
        self.dec3 = self.conv_block(128, 64)
        self.out = nn.Conv2d(64, out_channels, kernel_size=1)
```

- Encoder:
  - Convolutional blocks

```
def conv_block(self, in_channels, out_channels):
return nn.Sequential(
  nn.Conv2d(in_channels, out_channels),
  nn.ReLU(inplace=True),
  nn.Conv2d(out_channels, out_channels),
  nn.ReLU(inplace=True)
)
```

  - Pooling layer
- Decoder:
  - Transposed convolutions
  - Convolutional blocks

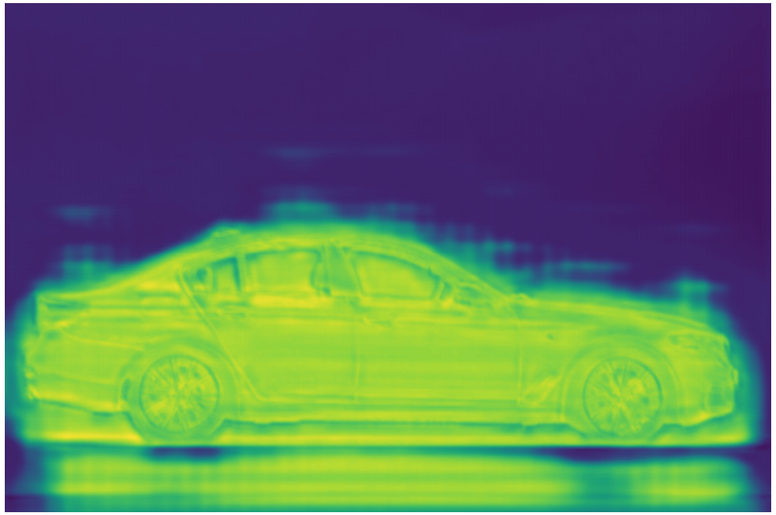
U-Net: forward method

```
def forward(self, x):
    x1 = self.enc1(x)
    x2 = self.enc2(self.pool(x1))
    x3 = self.enc3(self.pool(x2))
    x4 = self.enc4(self.pool(x3))

    x = self.upconv3(x4)
    x = torch.cat([x, x3], dim=1)
    x = self.dec1(x)

    x = self.upconv2(x)
    x = torch.cat([x, x2], dim=1)
    x = self.dec2(x)

    x = self.upconv1(x)
    x = torch.cat([x, x1], dim=1)
    x = self.dec3(x)

    return self.out(x)
```

- Pass input through encoder's convolutional blocks and pooling layers
- Decoder and skip connections:
  - Pass encoded input through transpose convolution
  - Concatenate with corresponding encoder output
  - Pass through convolution block
  - Repeat for all decoder steps
- Return output of the last decoder step

Running inference

```
model = UNet()
model.eval()

image = Image.open("car.jpg")
transform = transforms.Compose([transforms.ToTensor()])
image_tensor = transform(image).unsqueeze(0)

with torch.no_grad():
    prediction = model(image_tensor).squeeze(0)

plt.imshow(prediction[1, :, :])
plt.show()
```



Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH

Panoptic segmentation

DEEP LEARNING FOR IMAGES WITH PYTORCH

Michal Oleszak
Machine Learning Engineer

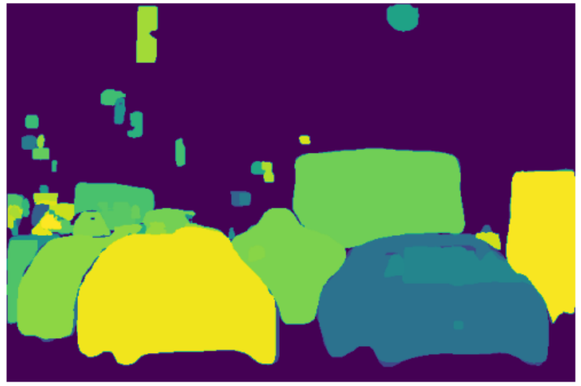
datacamp

Panoptic segmentation challenge

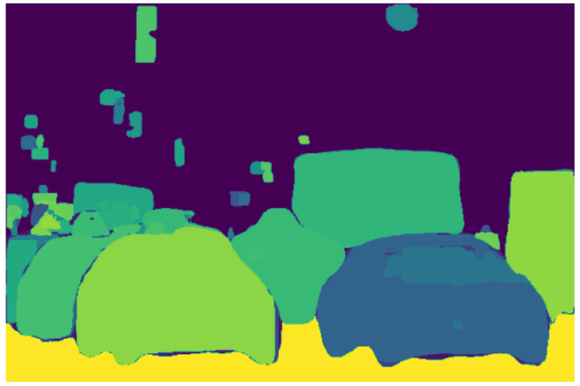
Original image

Semantic segmentation

Instance segmentation



Panoptic segmentation



Panoptic segmentation workflow

- Combining semantic and instance segmentation can be complex:
  - Overlaps
  - Ensuring unique instance IDs
- Our workflow:
  1. Generate semantic masks
  2. Combine them into a single mask
  3. Initialize the panoptic mask as the semantic mask
  4. Generate instance masks
  5. Iterate over instance masks and overlay detected objects onto the semantic mask

Semantic masks

```
model = UNet()

with torch.no_grad():
    semantic_masks = model(image_tensor)
    print(semantic_masks.shape)
```

```
torch.Size([1, 3, 427, 640])
```

```
semantic_mask = torch.argmax(
  semantic_masks, dim=1
)
```

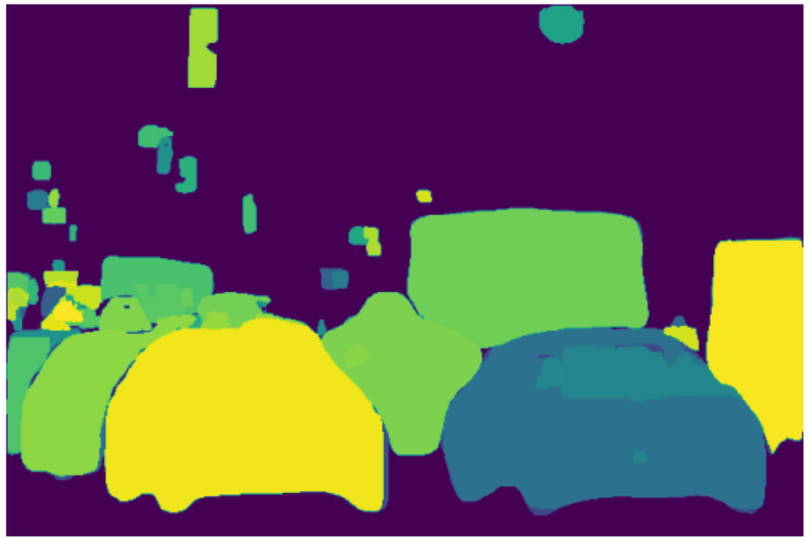
- Instantiate the model
- Produce semantic masks for the input image
- Choose highest-probability class for each pixel

Instance masks

```
model = MaskRCNN()

with torch.no_grad():
    instance_masks = model(image_tensor)[0]["masks"]
    print(instance_masks.shape)
```

```
torch.Size([80, 1, 427, 640])
```

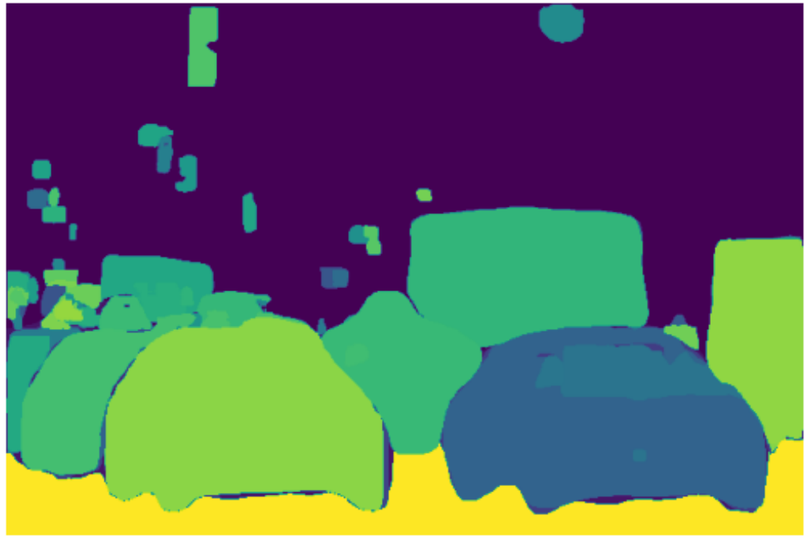
- Load instance segmentation model
- Produce instance masks



Panoptic masks

```
panoptic_mask = torch.clone(semantic_mask)

instance_id = 3
for mask in instance_masks:
    panoptic_mask[mask > 0.5] = instance_id
    instance_id += 1
```

- Initialize panoptic mask as semantic_mask
- Iterate over instance masks
- Set panoptic mask to instance ID where mask > 0.5
- Increase instance ID counter



Let's practice!

DEEP LEARNING FOR IMAGES WITH PYTORCH