# BVM 2019 / Tutorial: **Hands-On Deep Learning Using PyTorch** (Part 1)

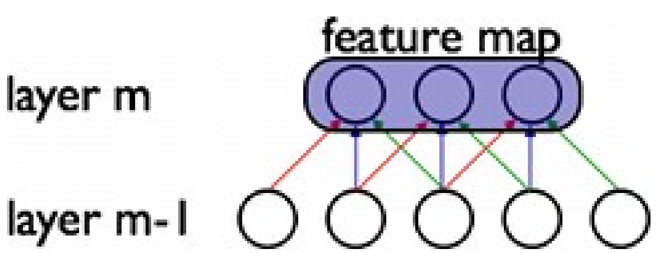
**Christian Lucas**

Prof. Mattias P. Heinrich Group
Institute of Medical Informatics
University of Lübeck

# Convolutional Neural Networks (CNN)

- key idea: **learn convolutional kernels for features extraction**

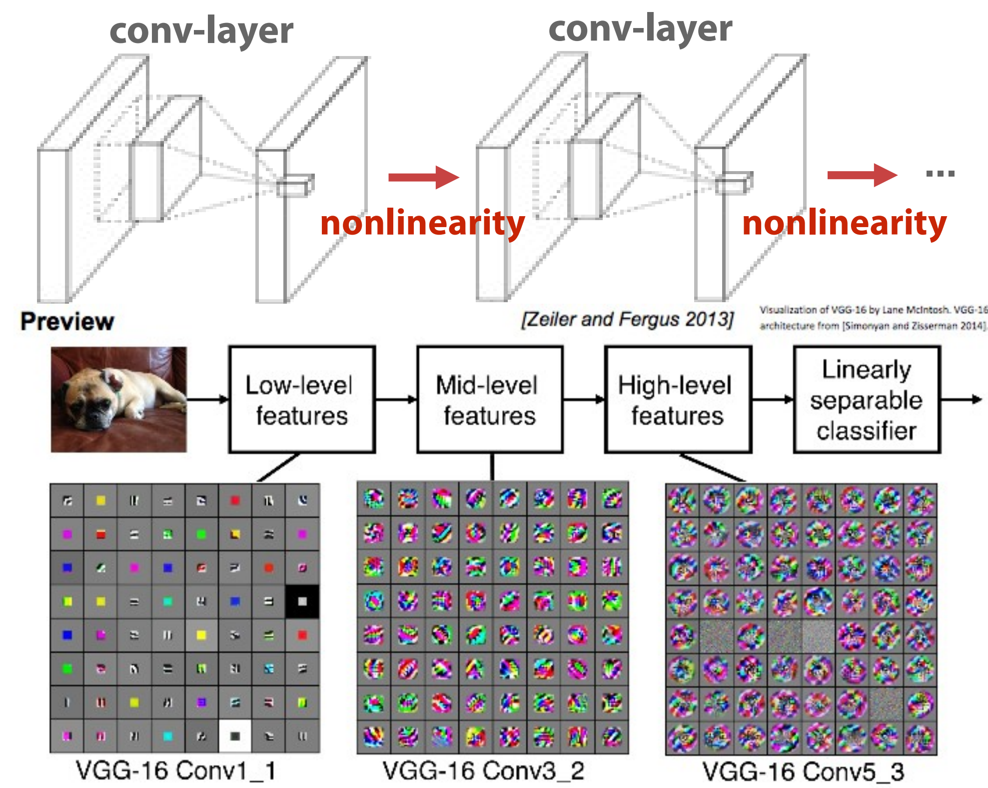
**→ use weight sharing**
**= sparse connectivity**



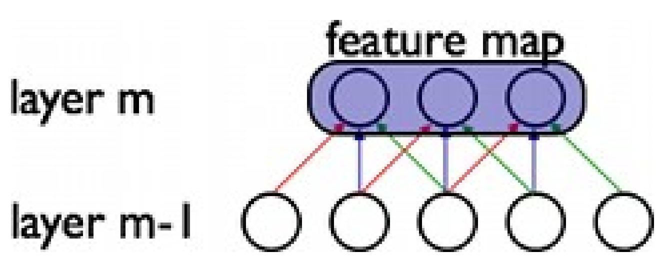

three weights
$w_1$, $w_2$, $w_3$
each used in 3 neurons

# Convolutional Neural Networks (CNN)

- key idea: **learn convolutional kernels for features extraction**
- **Jointly learn everything within a deep architecture**
  → Hierarchical feature transformations are jointly learned with classifier



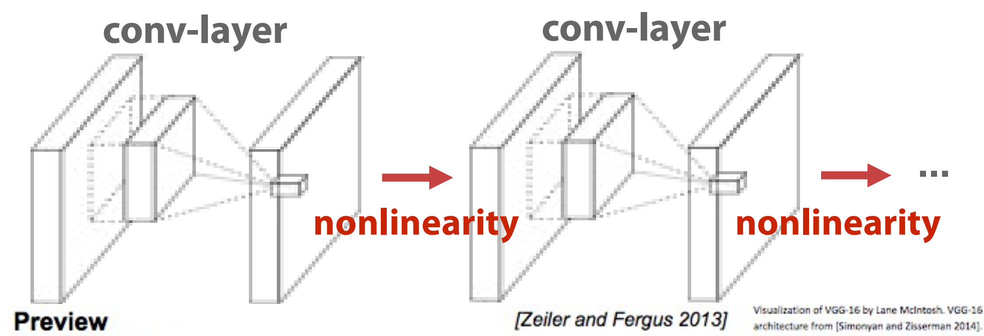

**→ use weight sharing**
**= sparse connectivity**

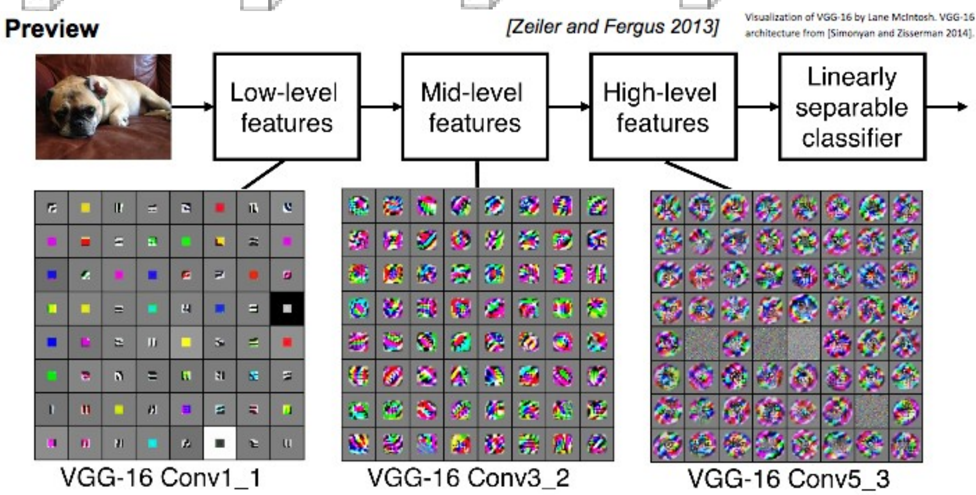



three weights
$w_1$, $w_2$, $w_3$
each used in 3 neurons

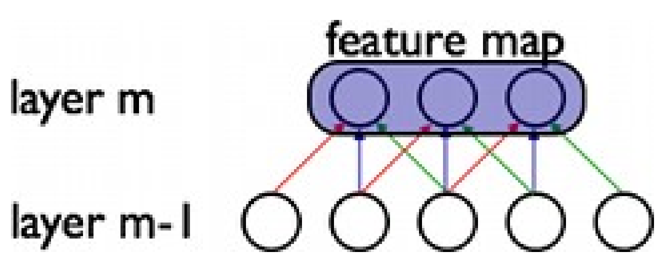
# Convolutional Neural Networks (CNN)

- key idea: **learn convolutional kernels for features extraction**
- **Jointly learn everything within a deep architecture**
  → Hierarchical feature transformations are jointly learned with classifier



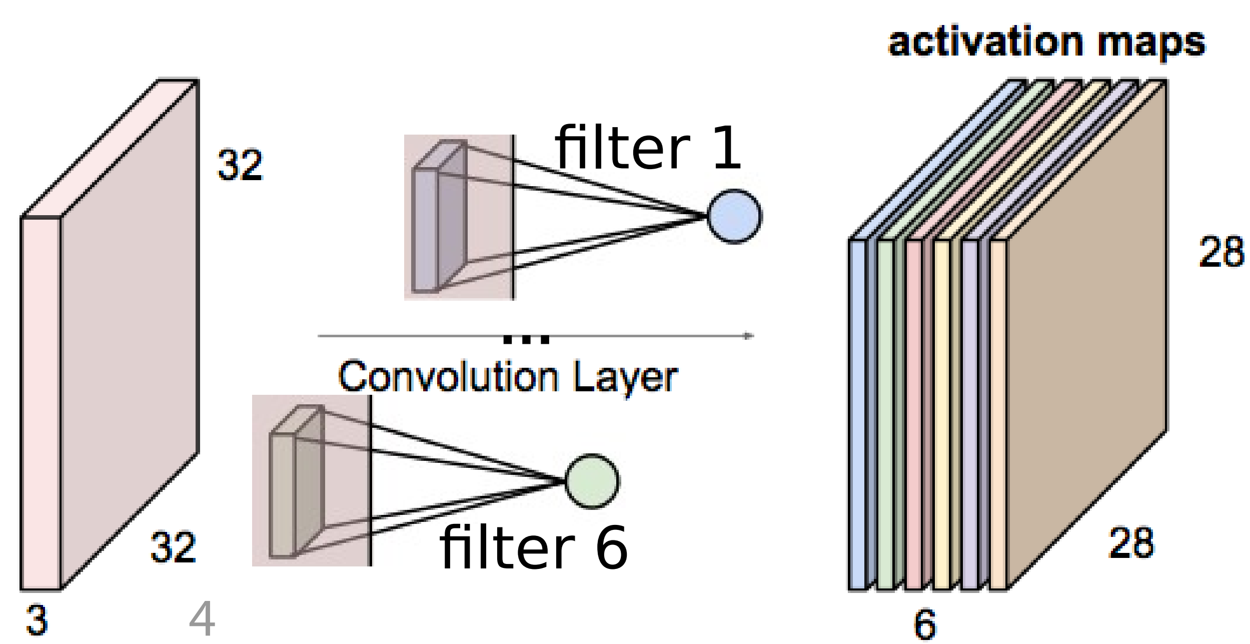

**→ use weight sharing**
**= sparse connectivity**



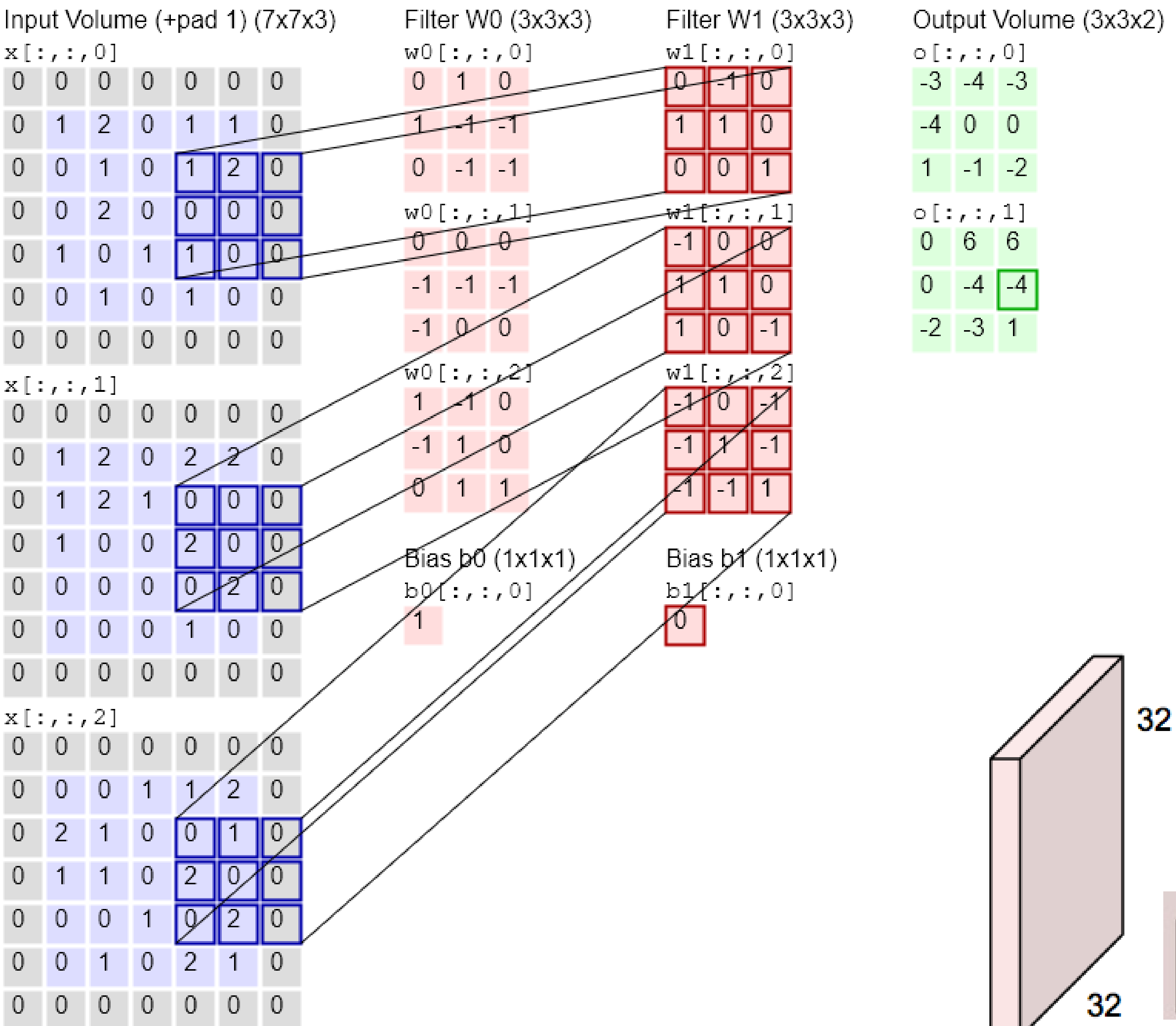

three weights
$w_1$, $w_2$, $w_3$
each used in 3 neurons

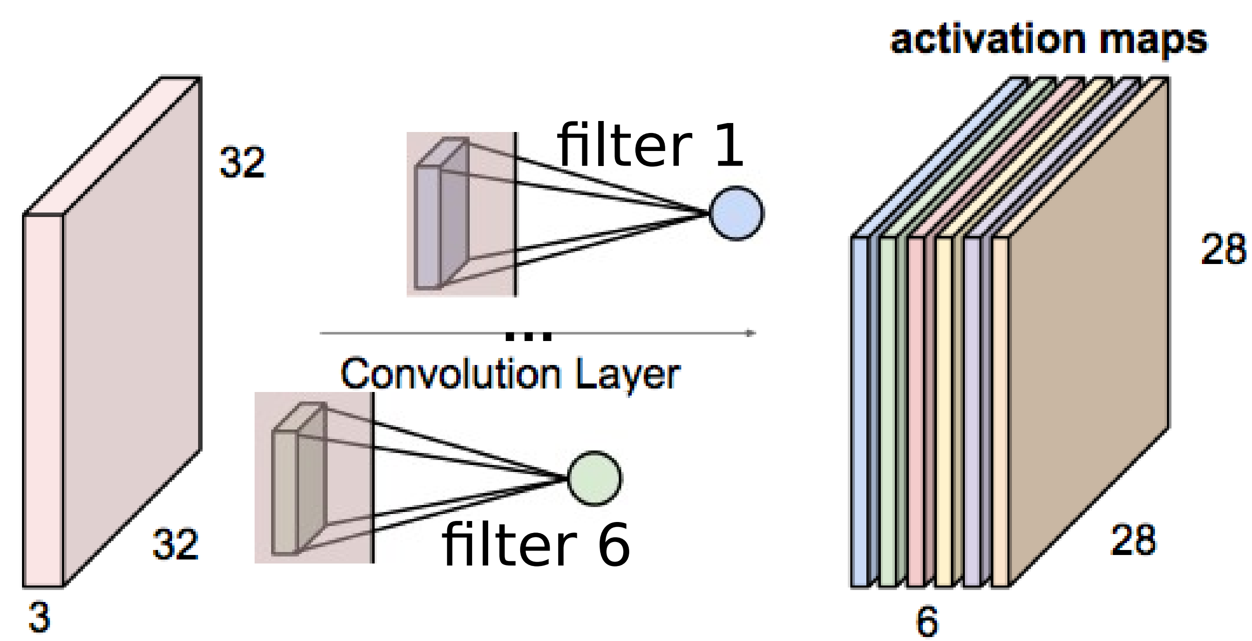
each filter is **applied to multi-channel patch** (e.g. 8x8x3)

filter bank **generates multiple feature/activation maps**



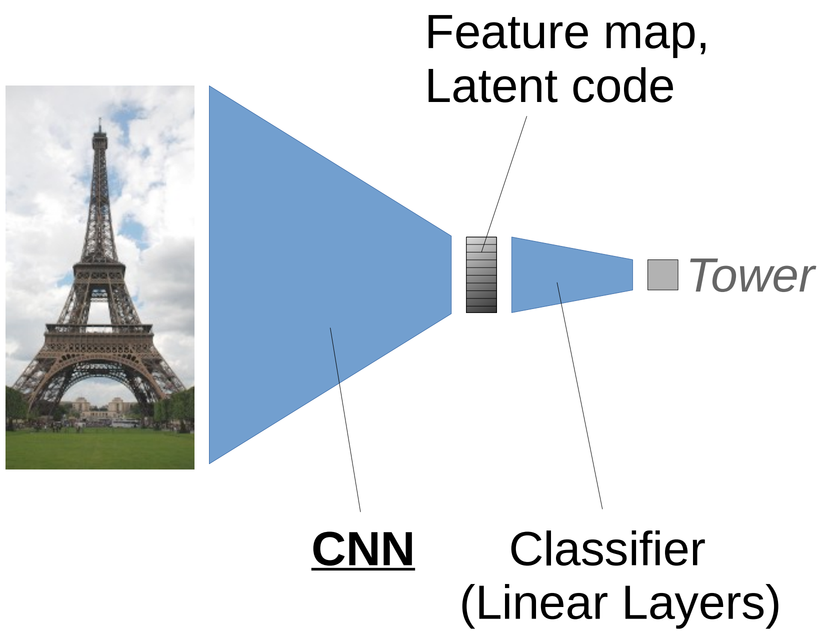
# Convolutional Neural Networks (CNN)



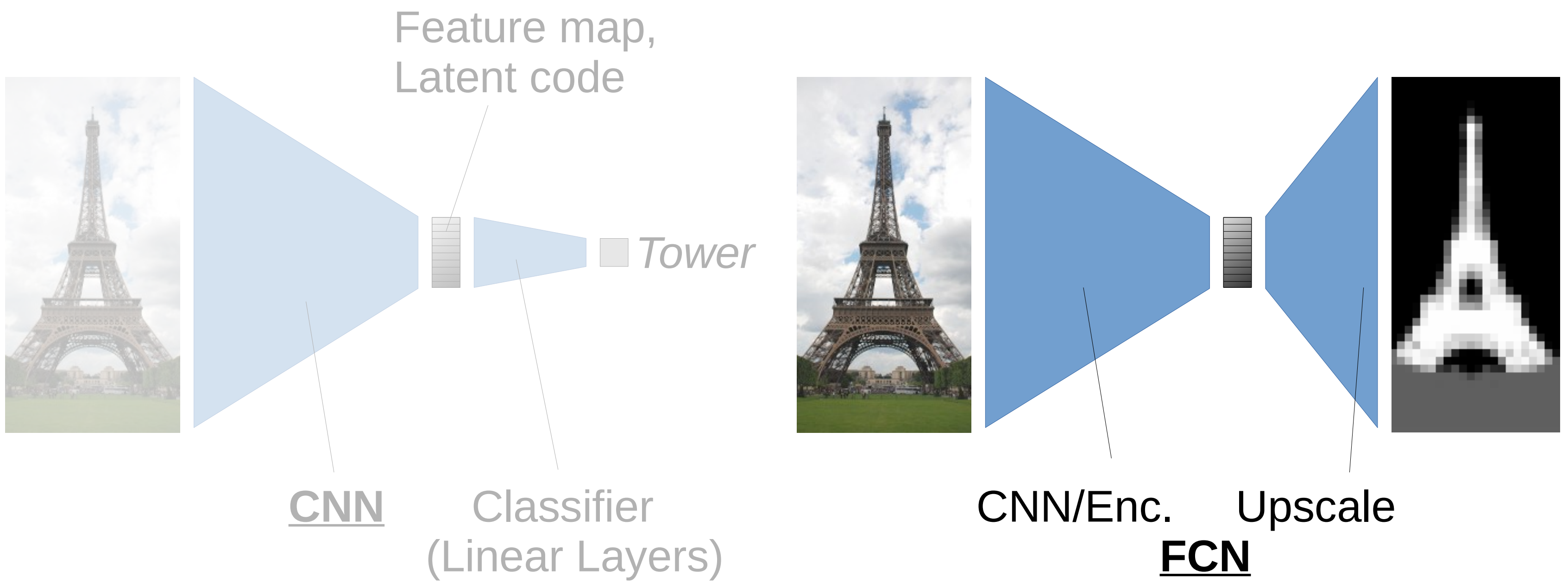




http://cs231n.github.io/convolutional-networks/

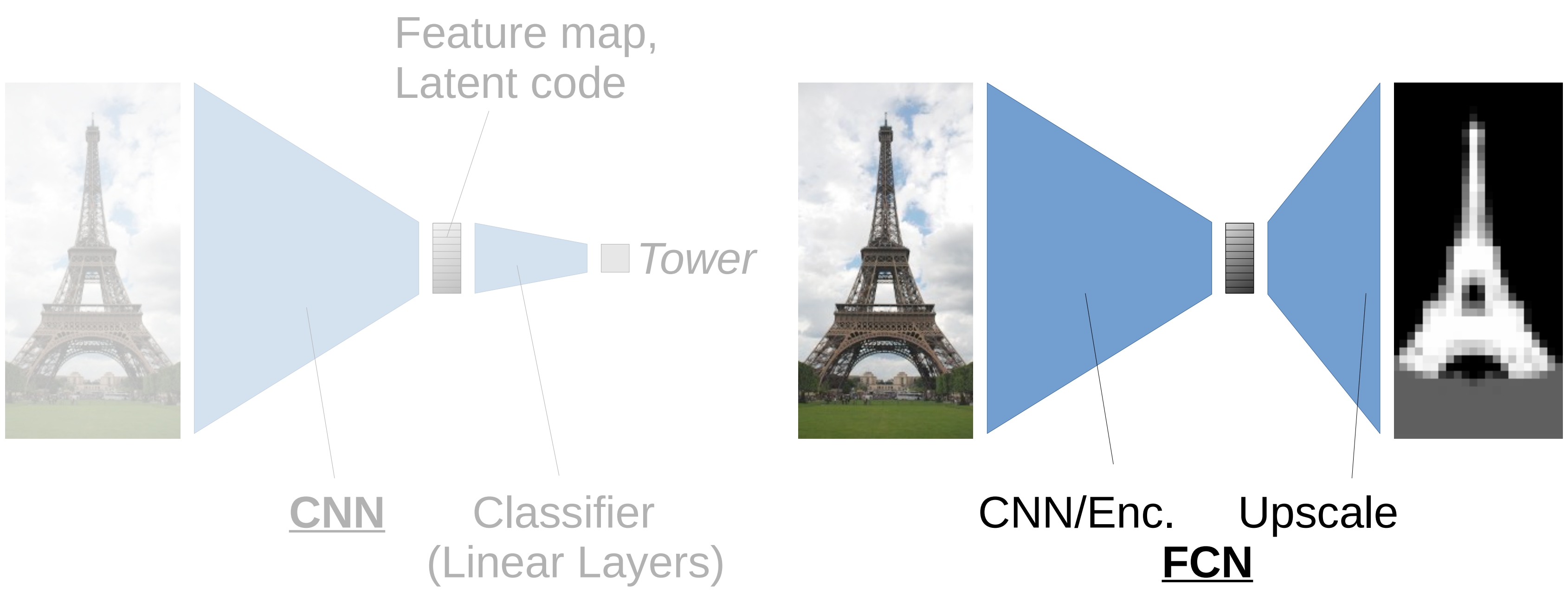
# Semantic Segmentation

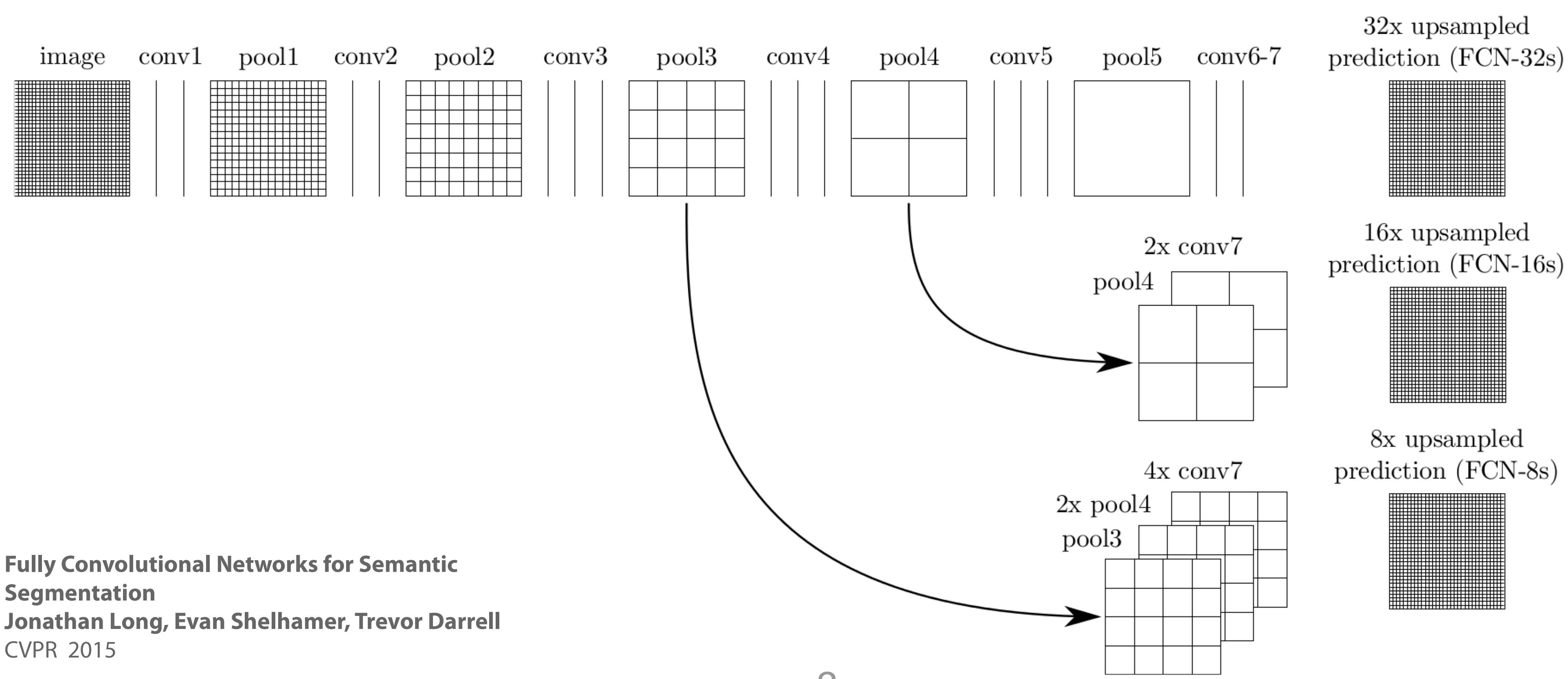


# Semantic Segmentation



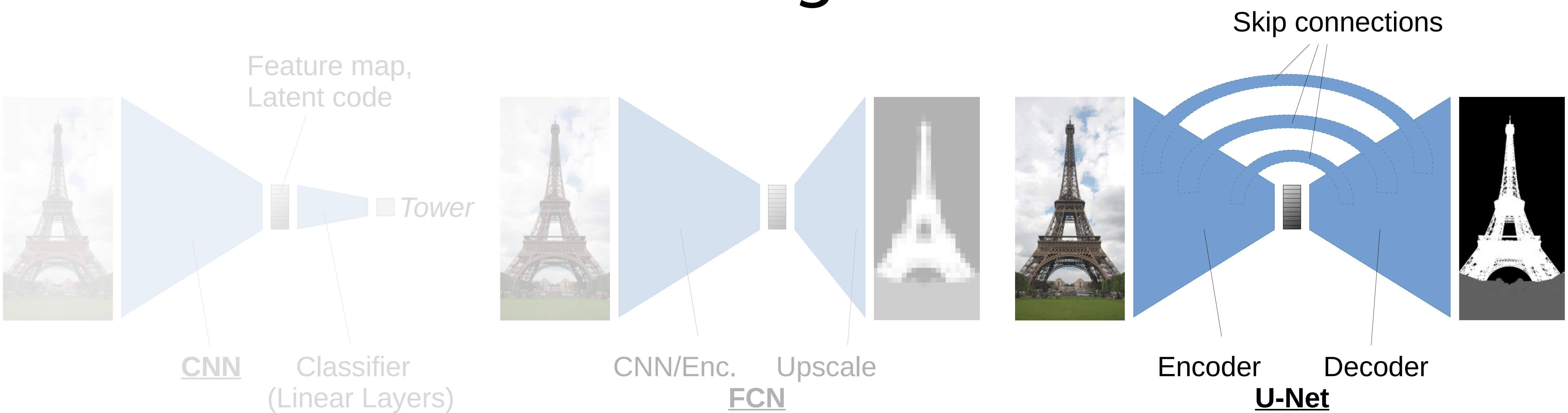
# Semantic Segmentation



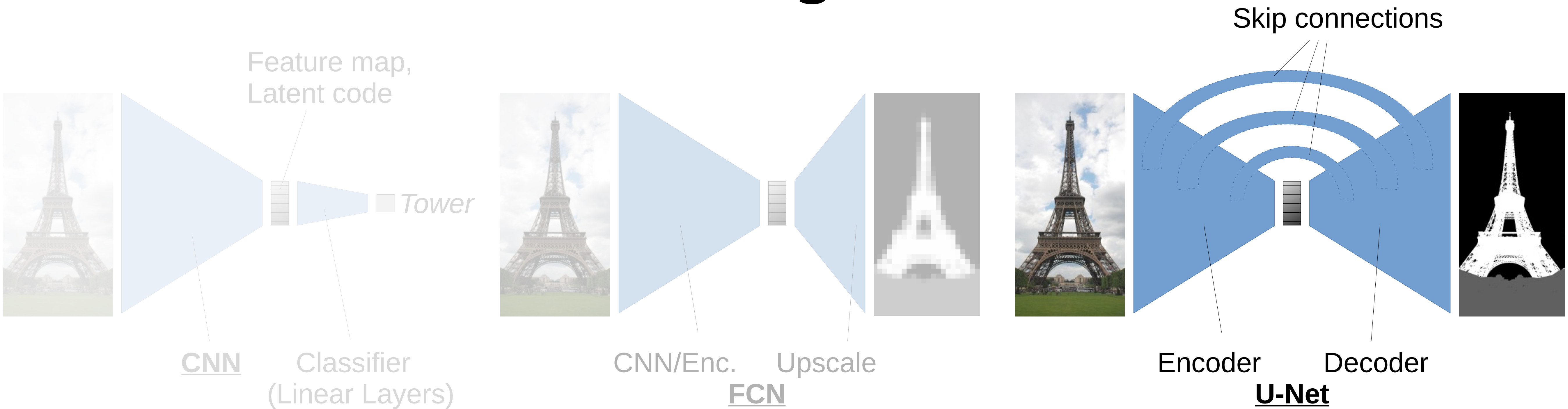



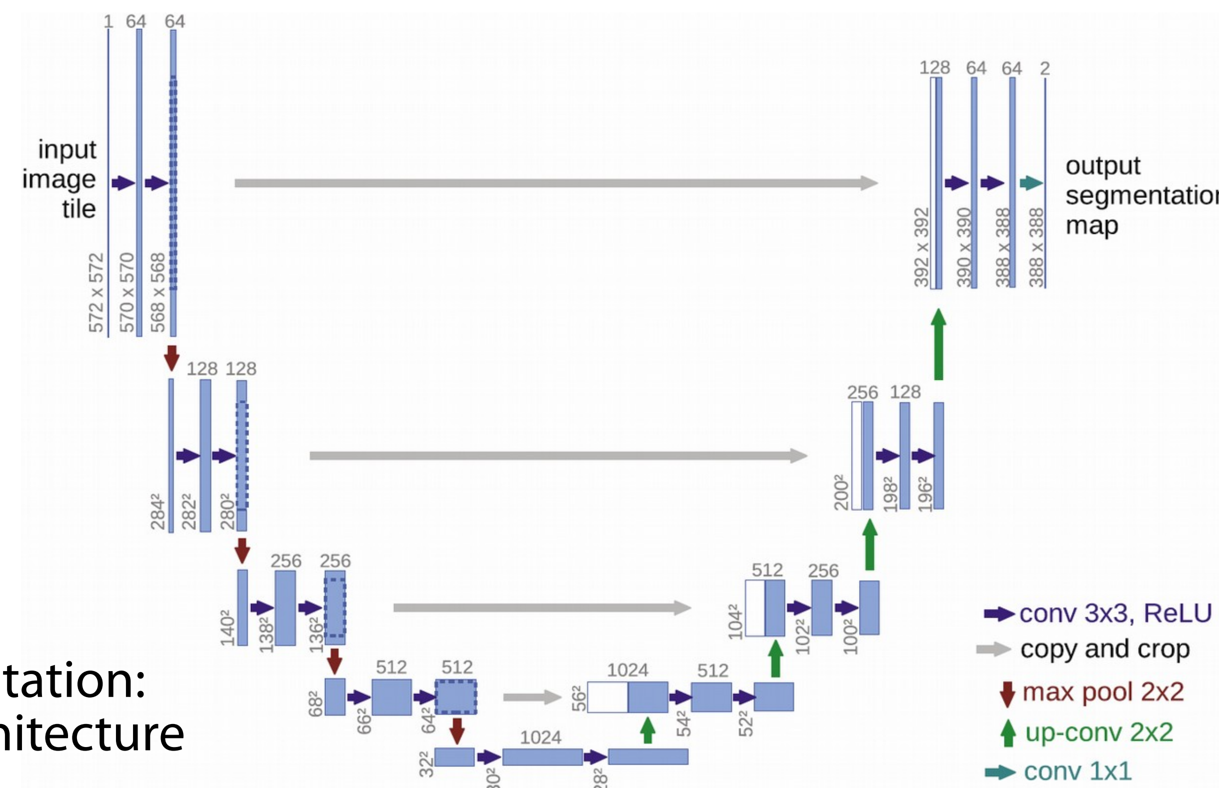

**Fully Convolutional Networks for Semantic Segmentation**
**Jonathan Long, Evan Shelhamer, Trevor Darrell**
CVPR 2015

# Semantic Segmentation



# Semantic Segmentation

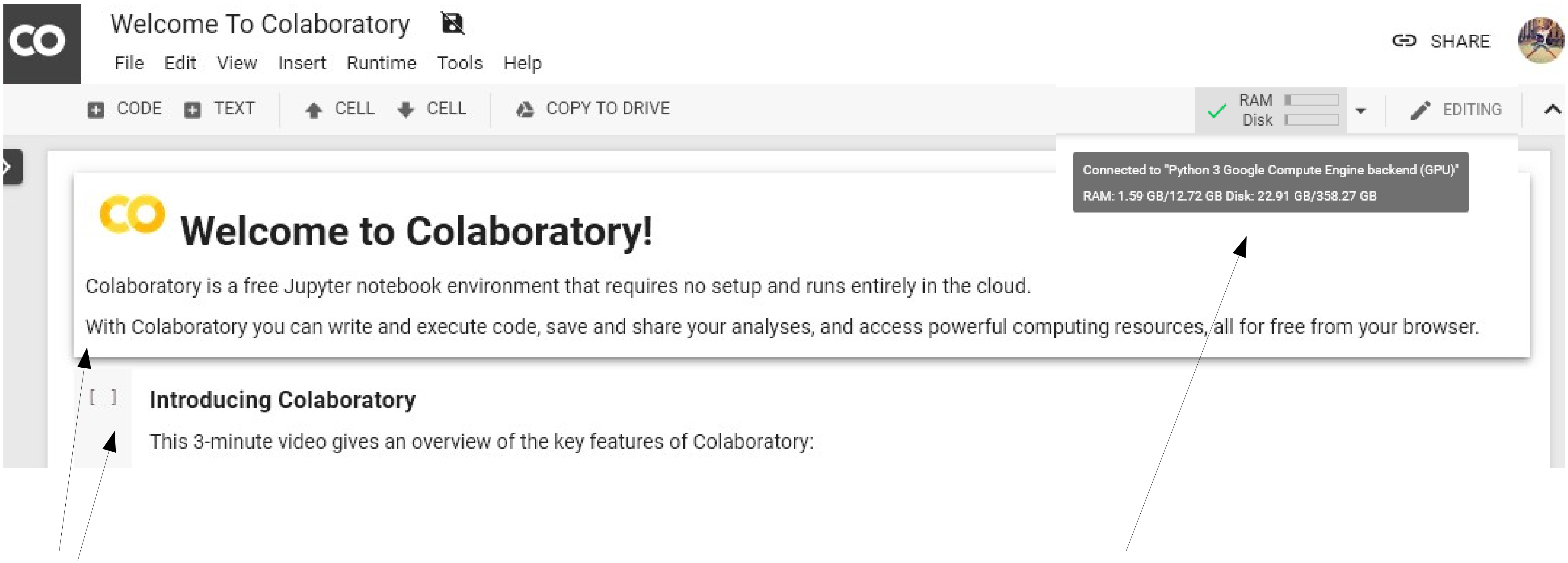



Swiss army knife for segmentation:
Unified multi-resolution architecture

**U-Net: Convolutional Networks for Biomedical Image Segmentation**
**Olaf Ronneberger, Philipp Fischer, Thomas Brox**
MICCAI 2015, Lecture Notes in Computer Science (Vol. 9351), pp. 234-241, Springer

# Colab




Notebook cells

12GB GPU memory

`.cuda()`

- PyTorch 1.0 pre-installed
- Access to git: `!git clone https://github.com/name/repository.git`
  `import repository.pythonfile`
- Install other packages: `!pip install package` or `!setup.py install`

# Data

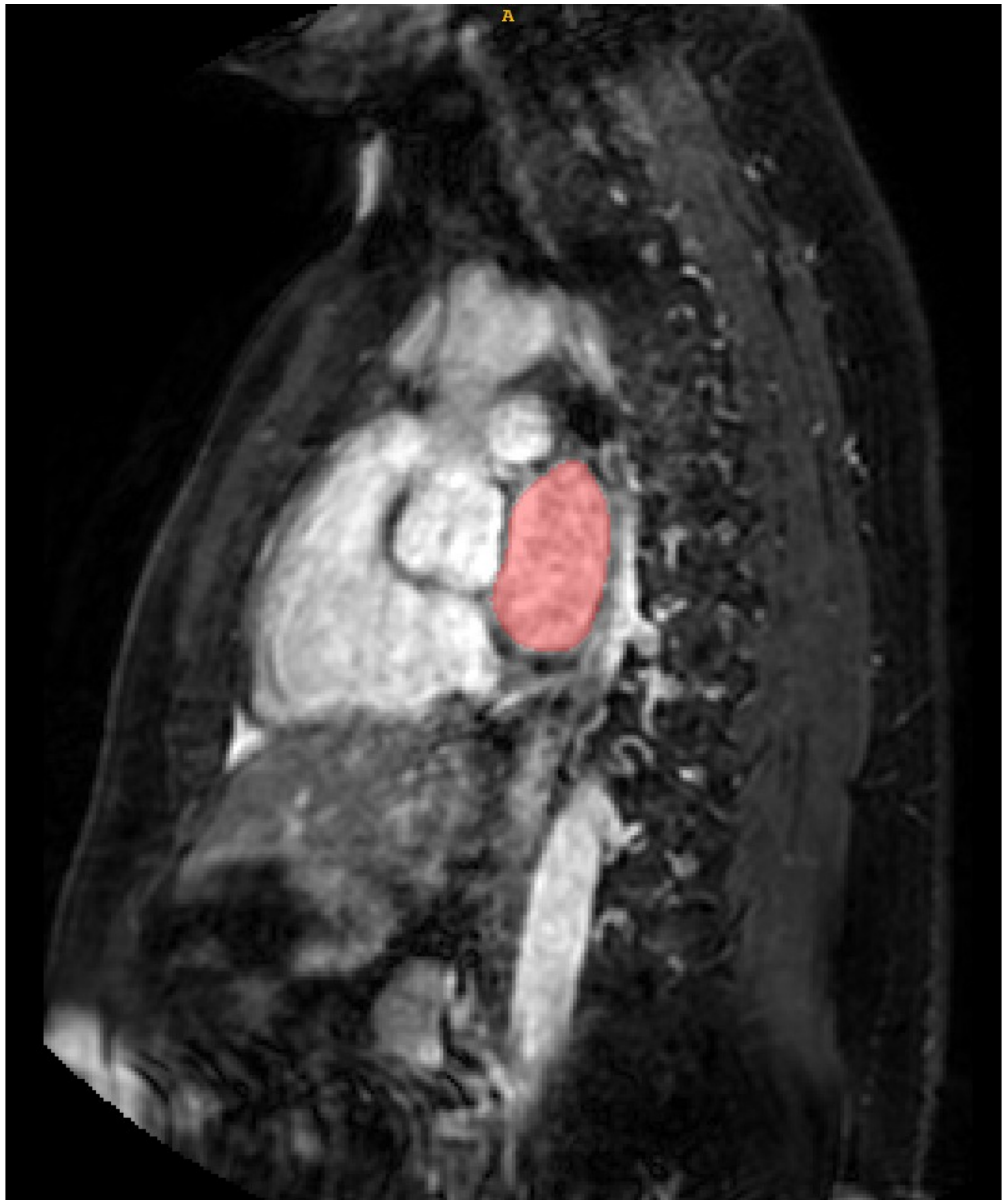
- Medical Segmentation Decathlon Task 02: **Heart data**

  ([www.medicaldecathlon.com](www.medicaldecathlon.com))

- Segmentation of left atrium („linker Vorhof“)

Cardiac

**Target:** Left Atrium
**Modality:** Mono-modal MRI
**Size:** 30 3D volumes (20 Training + 10 Testing)
**Source:** King's College London
**Challenge:** Small training dataset with large variability



# Data Loading

```
class MyDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.root_dir)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, idx, '.nii.gz'])
        image = io.imread(img_name)
        sample = {'image': image, 'index': idx}

        if self.transform:
            sample = self.transform(sample)

        return sample
```

`torch.utils.data.Dataset`: class representing a dataset

`__len__` returns the size of the dataset for `len(dataset)`

`__getitem__` for indexing (`dataset[i]` for i-th sample)

- Memory efficient implementation *on-demand* in `__getitem__`
- Samples as a dict `{'image': image, 'label': label}`
- Argument `transform` for any required pre-processing

# Data Loading

```
class MyDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform

    def __len__(self):
        return len(self.root_dir)

    def __getitem__(self, idx):
        img_name = os.path.join(self.root_dir, idx, '.nii.gz'])
        image = io.imread(img_name)
        sample = {'image': image, 'index': idx}

        if self.transform:
            sample = self.transform(sample)

        return sample
```

`torch.utils.data.Dataset`: class representing a dataset

`__len__` returns the size of the dataset for `len(dataset)`

`__getitem__` for indexing (`dataset[i]` for $i^{th}$ sample)

- Memory efficient implementation *on-demand* in `__getitem__`
- Samples as a dict `{'image': image, 'label': label}`
- Argument `transform` for any required pre-processing

⋮

- `Sampler` for subsets (e.g., training set and validation set)

```
class ToTensor(object):
    def __call__(self, sample):
        image, idx = sample['image'], sample['index']
        image = image.transpose((2, 0, 1))
        return {'image': torch.from_numpy(image), index': idx}
```

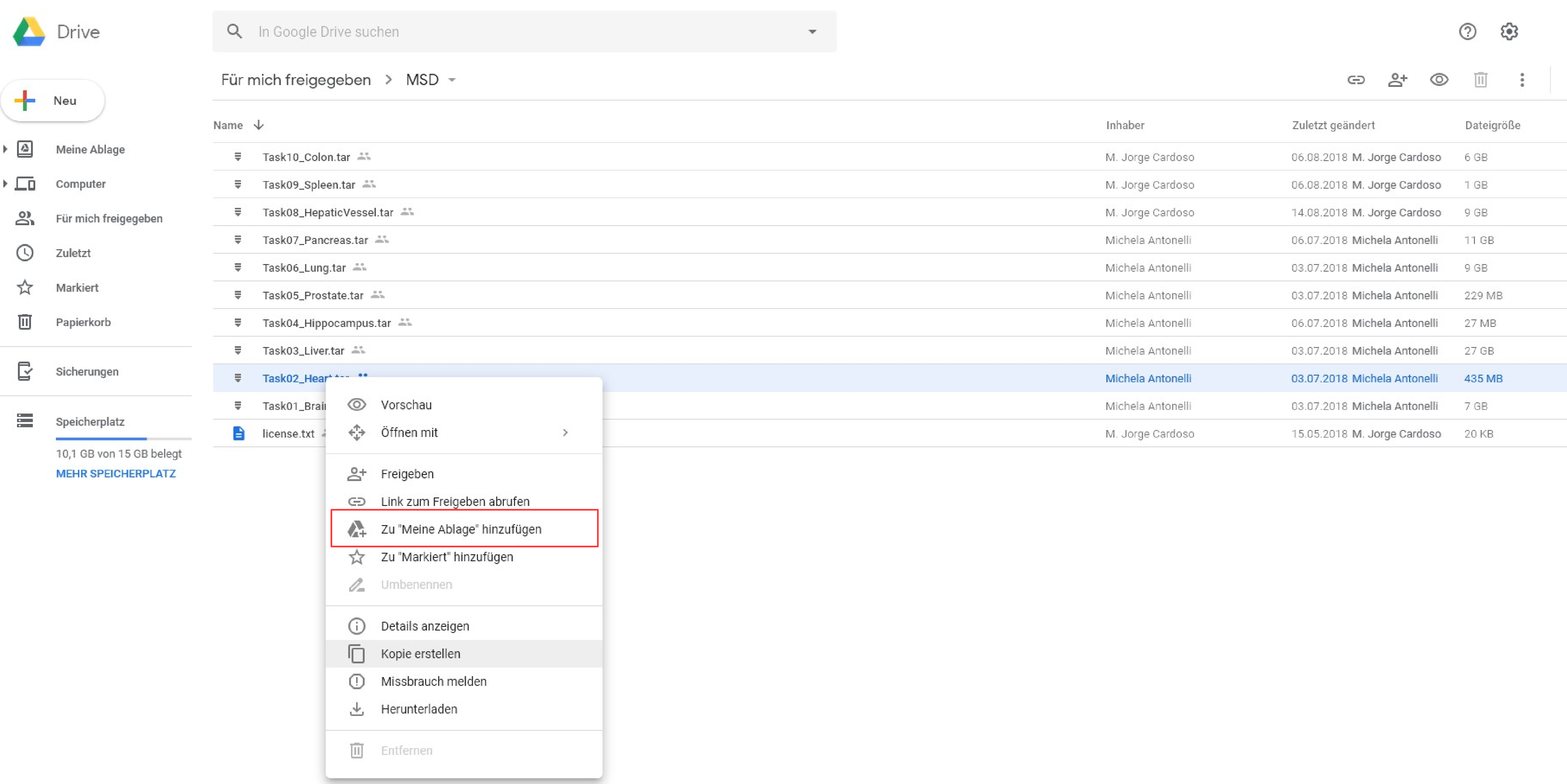
```
transformed_dataset = MyDataset(root_dir='/data/',
                                transform=transforms.Compose([
                                  Rescale(256),
                                  ToTensor()
                                ]))

dataloader = DataLoader(transformed_dataset,
                        batch_size=4,
                        shuffle=True,
                        num_workers=4)

for i_batch, sample_batched in enumerate(dataloader):
    # do whatever you‘d like to do
```

`torch.utils.data.DataLoader` is an iterator which provides:

- Batching the data
- Shuffling the data
- Load the data in parallel using multiprocessing workers.

# Hands on!

- Go to MSD drive and add **Task02_Heart.tar** to your drive




- Open the first jupyter notebook in your `colab.research.google.com`

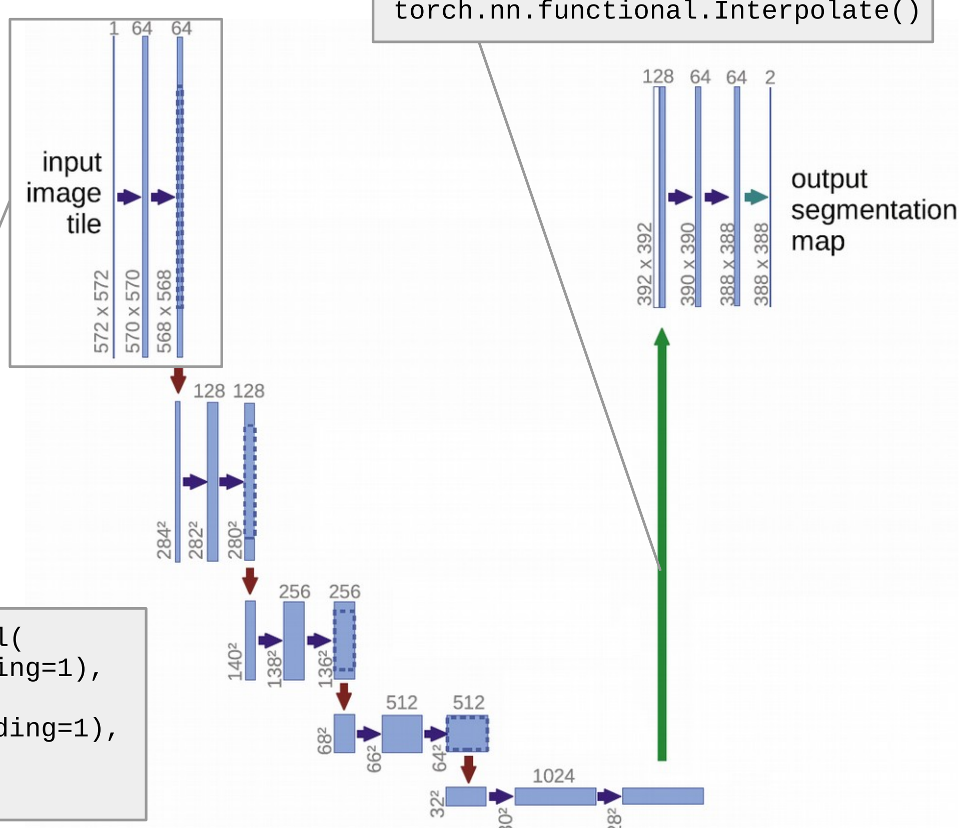
# Hands on!

- **Semantic segmentation** of the left atrium in the heart data via Deep Learning

- Task 1: Implement **Fully-Convolutional** Architecture
  - ✓ Double convolutional blocks
  - ✓ 2 poolings, and 1 final upsampling
  - ✓ Double the channel number, half the spatial resolution
  - ✓ To keep it easy: Use padding, so size of feature map after convolution is same as before

- Task 2: Extend FCN to **U-Net** (Symmetric decoder, more upsamplings, skip connections)

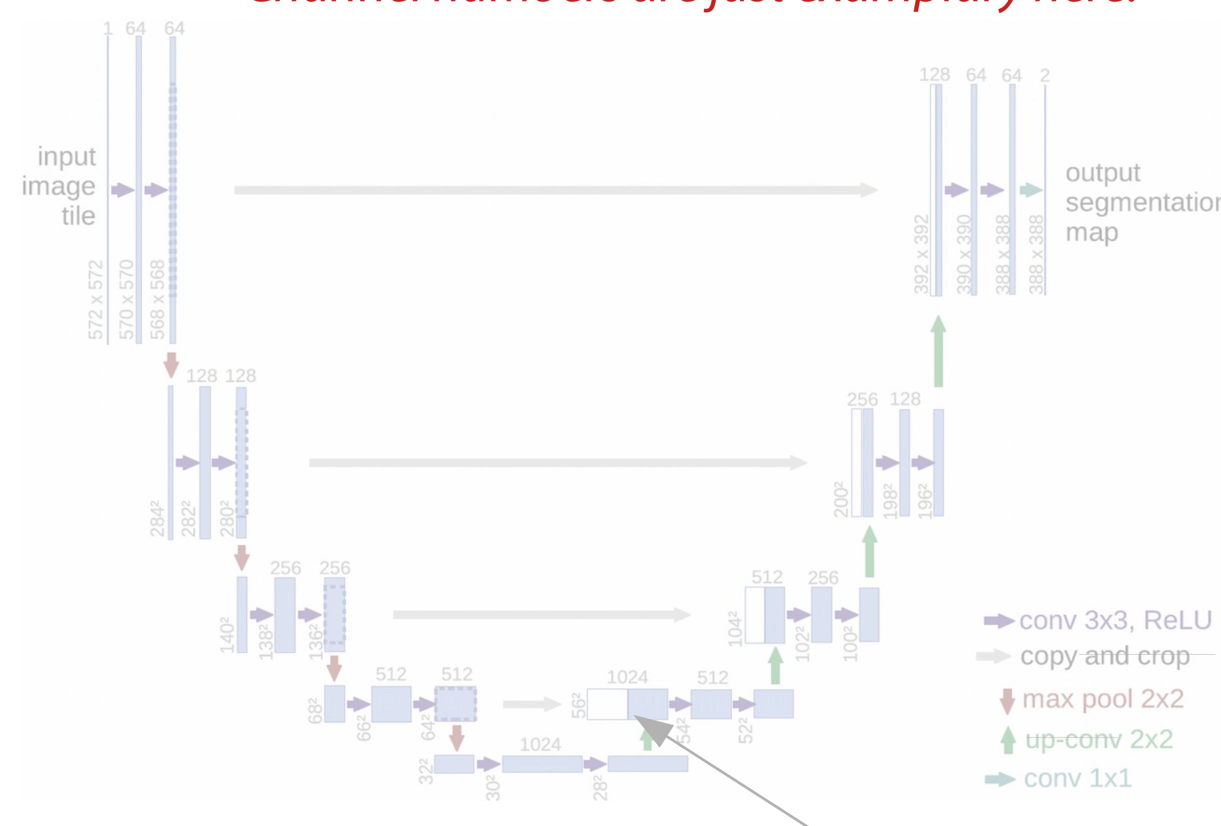
```
torch.nn.functional.Interpolate()
```




```
self.block0 = nn.Sequential(
  nn.Conv3d(1, 64, 3, padding=1),
  nn.ReLU(),
  nn.Conv3d(64, 64, 3, padding=1),
  nn.ReLU()
)
```

Task 1

*Channel numbers are just examplary here!*




```
torch.cat(out3, out4_upsampled)
```

Task 2

# Task 1: Solution

```
class FCN(nn.Module):
  def __init__(self):
    super().__init__()

    self.block0 = nn.Sequential(nn.BatchNorm3d(1),
                                nn.Conv3d(1, 20, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(20),
                                nn.Conv3d(20, 20, 3, padding=1),
                                nn.ReLU()
                               )

    self.mp01 = nn.MaxPool3d(2, 2)

    self.block1 = nn.Sequential(nn.BatchNorm3d(20),
                                nn.Conv3d(20, 40, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(40),
                                nn.Conv3d(40, 40, 3, padding=1),
                                nn.ReLU()
                               )

    self.mp12 = nn.MaxPool3d(2, 2)

    self.block2 = nn.Sequential(nn.BatchNorm3d(40),
                                nn.Conv3d(40, 80, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(80),
                                nn.Conv3d(80, 80, 3, padding=1),
                                nn.ReLU()
                               )

    self.block3 = nn.Sequential(nn.Conv3d(80, 40, 1),
                                nn.ReLU(),
                                nn.Conv3d(40, 1, 1),                 # 1 output channel segmentation
                               )

  def forward(self, inputs):

    output0 = self.block0(inputs)
    output1 = self.mp01(output0)
    output1 = self.block1(output1)
    output2 = self.mp12(output1)
    output2 = self.block2(output2)

    return self.block3(F.interpolate(output2, scale_factor=4))       # 4x upsampling to recover from 2x 2-MaxPooling
```

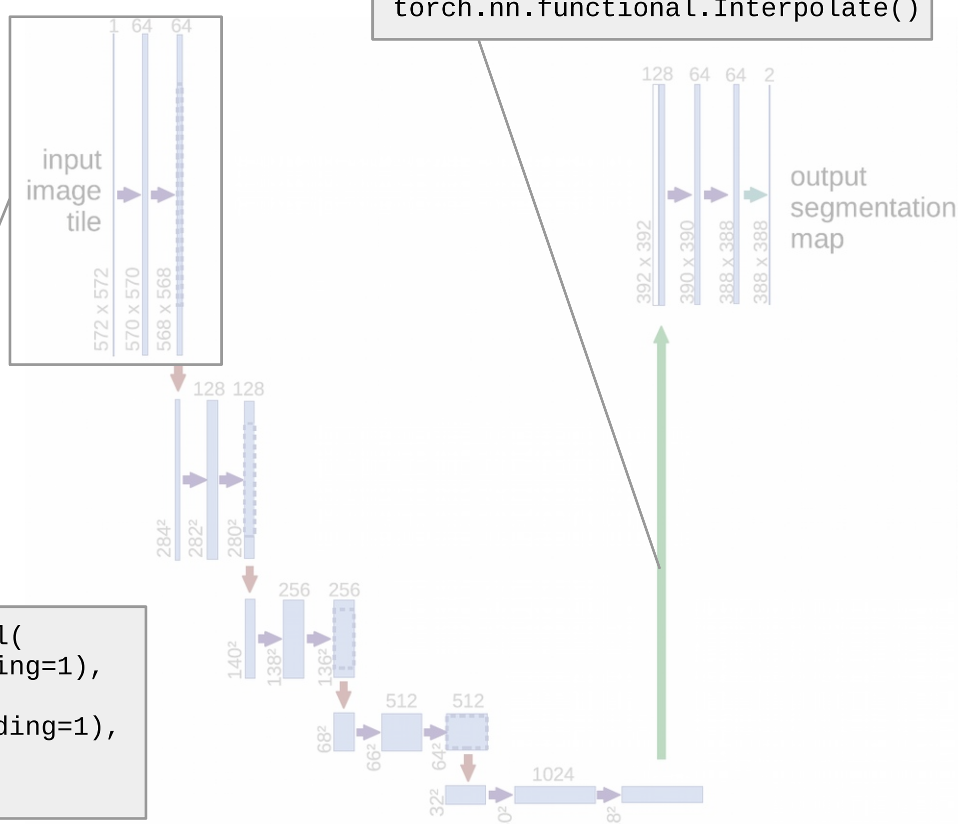
# Hands on!

- **Semantic segmentation** of the left atrium in the heart data via Deep Learning

- Task 1: Implement **Fully-Convolutional** Architecture
  - ✓ Double convolutional blocks
  - ✓ 2 poolings, and 1 final upsampling
  - ✓ Double the channel number, half the spatial resolution
  - ✓ To keep it easy: Use padding, so size of feature map after convolution is same as before

- Task 2: Extend FCN to **U-Net** (Symmetric decoder, more upsamplings, skip connections)

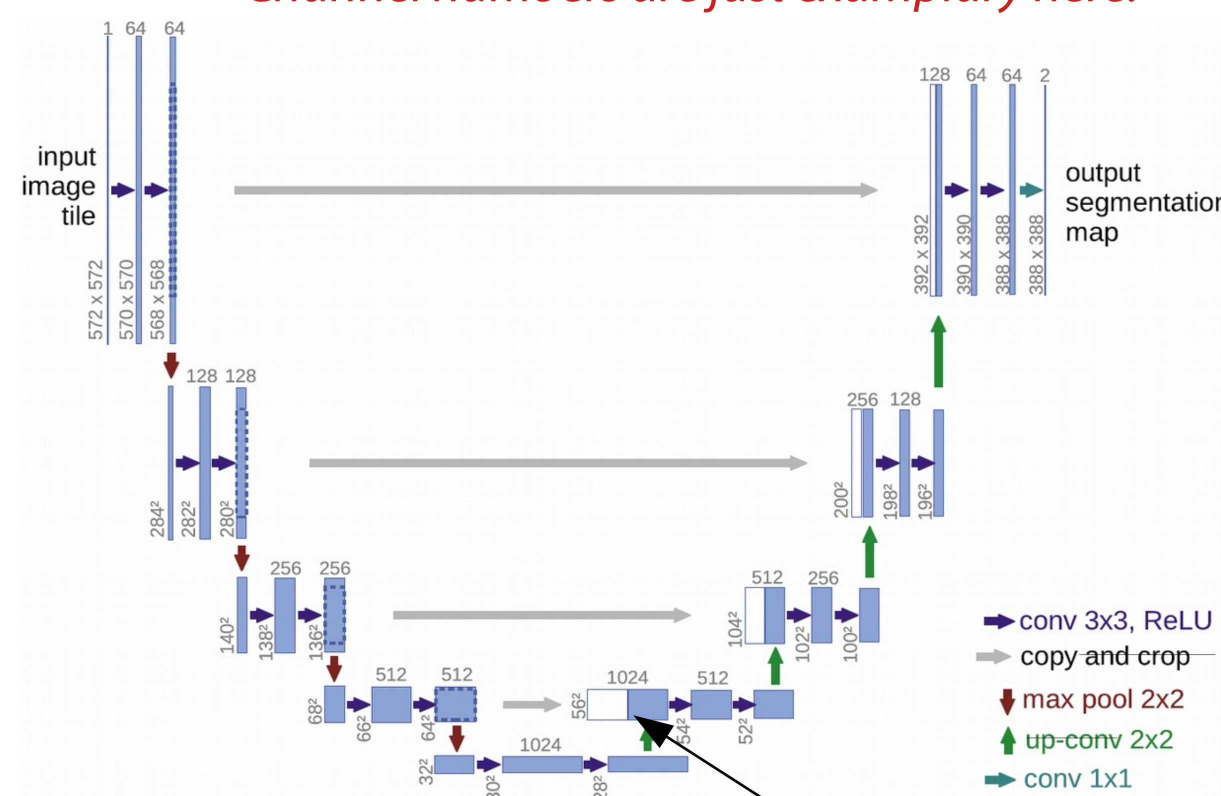
```
torch.nn.functional.Interpolate()
```




```
self.block0 = nn.Sequential(
  nn.Conv3d(1, 64, 3, padding=1),
  nn.ReLU(),
  nn.Conv3d(64, 64, 3, padding=1),
  nn.ReLU()
)
```

Task 1

*Channel numbers are just examplary here!*




```
torch.cat(out3, out4_upsampled)
```

Task 2

# Task 2: Solution

```
class Unet(nn.Module):
  def __init__(self):
    super().__init__()
    self.block0 = […]
    self.mp01 = […]
    self.block1 = […]
    self.mp12 = […]

    self.block2 = nn.Sequential(nn.BatchNorm3d(40),
                                nn.Conv3d(40, 80, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(80),
                                nn.Conv3d(80, 80, 3, padding=1),
                                nn.ReLU()
                               )

    self.block3 = nn.Sequential(nn.BatchNorm3d(120),          # 120 = upsample(block2) + skip(block1) = 80 + 40
                                nn.Conv3d(120, 80, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(80),
                                nn.Conv3d(80, 40, 3, padding=1),
                                nn.ReLU()
                               )

    self.block4 = nn.Sequential(nn.BatchNorm3d(60),           # 60 = upsample(block3) + skip(block0) = 40 + 20
                                nn.Conv3d(60, 40, 3, padding=1),
                                nn.ReLU(),
                                nn.BatchNorm3d(40),
                                nn.Conv3d(40, 20, 3, padding=1),
                                nn.ReLU()
                               )

    self.block5 = nn.Sequential(nn.Conv3d(20, 10, 1),
                                nn.ReLU(),
                                nn.Conv3d(10, 1, 1)
                               )

  def forward(self, inputs):

    output0 = self.block0(inputs)
    output1 = self.mp01(output0)
    output1 = self.block1(output1)
    output2 = self.mp12(output1)
    output2 = self.block2(output2)

    output3 = F.interpolate(output2, scale_factor=2)
    output3 = self.block3(torch.cat([output3, output1], dim=1))
    output4 = F.interpolate(output3, scale_factor=2)
    output4 = self.block4(torch.cat([output4, output0], dim=1))

    return self.block5(output4)
```