

Introduction to Deep Learning (I2DL)

Exercise 10: Semantic Segmentation

Overview

- Exercise 9 Recap
 - Case study of one submitted solution
- Semantic Segmentation
 - Loss Function
 - Architecture and Upsampling
- Submission Details
 - Deadline: July 08, 2020 23.59



Case Study: Leaderboard

Leaderboard: Submission 9

Rank	User	Score	Pass
#1	s0672	942.66	✓
#2	s0463	940.88	✓
#3	s0770	792.80	✓
#4	s0303	722.08	✓
#5	s0587	689.02	✓
#6	s0747	656.89	✓
#7	s0555	654.95	✓
#8	s0400	615.63	✓
#9	s0322	607.35	✓
#10	s0288	602.19	✓

Case Study: Model

```
self.model = nn.Sequential(  
    nn.Conv2d(1, 32, (3, 3), stride=1, padding=2),  
    # nn.BatchNorm2d(32),  
    # nn.Dropout2d(0.2),  
    nn.PReLU(),  
  
    nn.MaxPool2d(3),  
  
    nn.Conv2d(32, 64, (3, 3), stride=1, padding=2),  
    # nn.BatchNorm2d(64),  
    # nn.Dropout2d(),  
    nn.PReLU(),  
  
    nn.MaxPool2d(3, stride=2),  
  
    nn.Conv2d(64, 64, (3, 3), stride=1, padding=1),  
    # nn.BatchNorm2d(64),  
    # nn.Dropout2d(0.3),  
    nn.PReLU(),  
  
    nn.MaxPool2d(2, stride=2),  
  
    nn.Conv2d(64, 128, (2, 2), stride=1, padding=1),  
    # nn.BatchNorm2d(128),  
    # nn.Dropout2d(0.3),  
    nn.PReLU(),
```

Classic ConvNet architecture:

- Feature extraction
- Classification

```
        Flatten(),  
        nn.Linear(10368, 256),  
        # nn.BatchNorm1d(256),  
        nn.Dropout(0.1),  
        nn.PReLU(),  
  
        nn.Linear(256, 30),  
)
```

Case Study: Model Summary

```
#!pip install torchsummary
import torchsummary

torchsummary.summary(model, (1, 96, 96))
```

Layer (type)	Output Shape	Param #
Conv2d-1	[1, 32, 98, 98]	320
PReLU-2	[1, 32, 98, 98]	1
MaxPool2d-3	[1, 32, 32, 32]	0
Conv2d-4	[1, 64, 34, 34]	18,496
PReLU-5	[1, 64, 34, 34]	1
MaxPool2d-6	[1, 64, 16, 16]	0
Conv2d-7	[1, 64, 16, 16]	36,928
PReLU-8	[1, 64, 16, 16]	1
MaxPool2d-9	[1, 64, 8, 8]	0
Conv2d-10	[1, 128, 9, 9]	32,896
PReLU-11	[1, 128, 9, 9]	1
Flatten-12	[1, 10368]	0
Linear-13	[1, 256]	2,654,464
Dropout-14	[1, 256]	0
PReLU-15	[1, 256]	1
Linear-16	[1, 30]	7,710

Total params: 2,750,819
Trainable params: 2,750,819
Non-trainable params: 0

Input size (MB): 0.04
Forward/backward pass size (MB): 6.72
Params size (MB): 10.49
Estimated Total Size (MB): 17.25

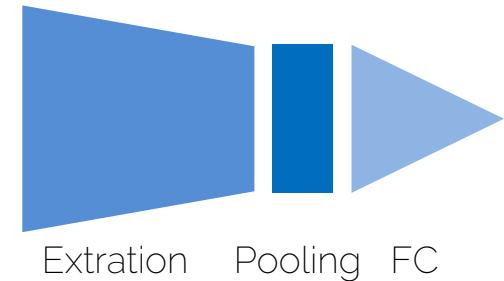
(9x9x128 = 10368)

```
        Flatten(),
        nn.Linear(10368, 256),
        # nn.BatchNorm1d(256),
        nn.Dropout(0.1),
        nn.PReLU(),

        nn.Linear(256, 30),
```

Case Study: Smaller Linear Layer?

1. Convolutional layer to reduce size to 1x1
 - Here: 9x9 kernel, 128 filters, no padding
 $\Rightarrow 1 \times 1 \times 128 = 128$
2. Global Average Pooling (GAP)
 - Here: 9x9 kernel $\Rightarrow 128$
 - Disadvantage: lose spatial relations
3. Flatten
 - Solutions: first use 1x1 convolutions



Case Study: With 1x1 Conv

```
# After adding 1x1 layers
# nn.Conv2d(128, 16, (1, 1), stride=1, padding=0),
# Flatten(),
# nn.Linear(9*9*16, 256),
# ...
torchsummary.summary(model, (1, 96, 96))
```

Layer (type)	Output Shape	Param #
=====		
Conv2d-1	[1, 32, 98, 98]	320
PReLU-2	[1, 32, 98, 98]	1
MaxPool2d-3	[1, 32, 32, 32]	0
Conv2d-4	[1, 64, 34, 34]	18,496
PReLU-5	[1, 64, 34, 34]	1
MaxPool2d-6	[1, 64, 16, 16]	0
Conv2d-7	[1, 64, 16, 16]	36,928
PReLU-8	[1, 64, 16, 16]	1
MaxPool2d-9	[1, 64, 8, 8]	0
Conv2d-10	[1, 128, 9, 9]	32,896
PReLU-11	[1, 128, 9, 9]	1
Conv2d-12	[1, 16, 9, 9]	2,064
Flatten-13	[1, 1296]	0
Linear-14	[1, 256]	332,032
Dropout-15	[1, 256]	0
PReLU-16	[1, 256]	1
Linear-17	[1, 30]	7,710
=====		

```
Total params: 430,451
Trainable params: 430,451
Non-trainable params: 0
-----
Input size (MB): 0.04
Forward/backward pass size (MB): 6.66
Params size (MB): 1.64
Estimated Total Size (MB): 8.34
-----
```

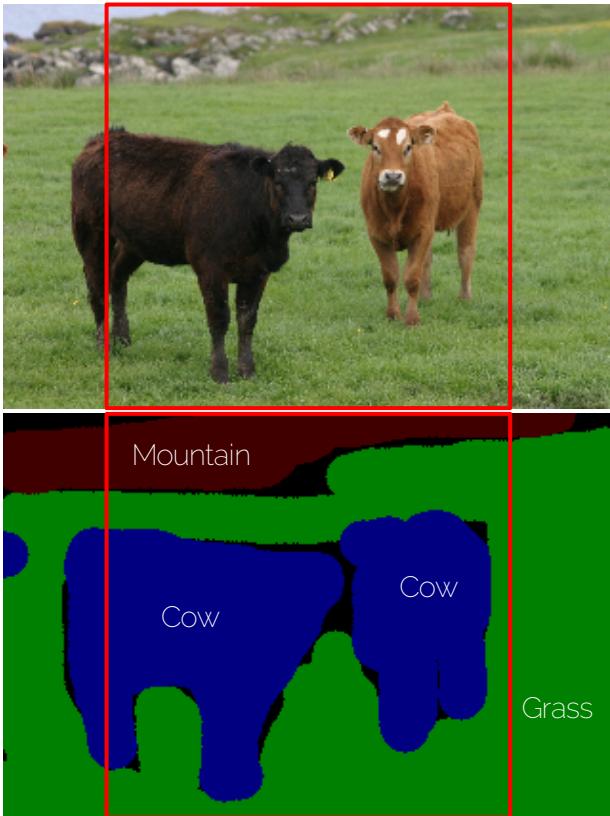
Next steps:
Make deeper and use residual connection to make it train

Case Study: Hyperparameters

```
hparams = {  
    "lr": 0.0001,  
    "batch_size": 512,  
    # TODO: if you have any model arguments/hparams, define them here  
}
```

- Default learning rate
- Experiment with batch normalization / Dropout
- Forms of ReLU activations (PReLU, ELU)
- Appropriate weight initialization

Semantic Segmentation



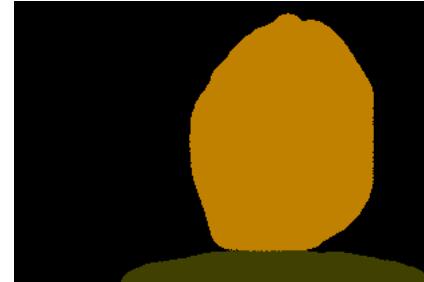
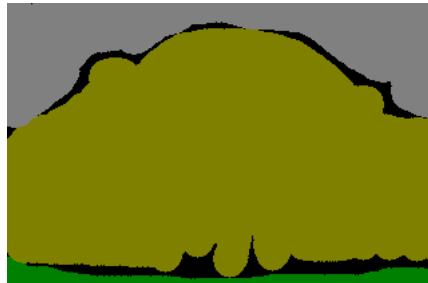
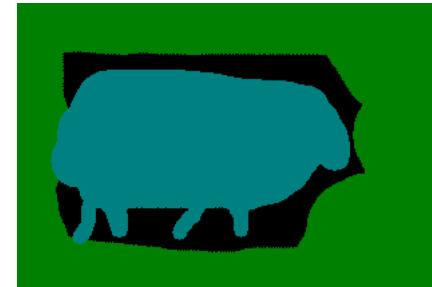
Input:
(3, w, h) RGB image

Output:
(23, w, h) segmentation map
with scores for every class
in every pixel

Semantic Segmentation Labels

<i>object class</i>	<i>R</i>	<i>G</i>	<i>B</i>	<i>Colour</i>
<i>void</i>	0	0	0	
<i>building</i>	128	0	0	
<i>grass</i>	0	128	0	
<i>tree</i>	128	128	0	
<i>cow</i>	0	0	128	
<i>horse</i>	128	0	128	
<i>sheep</i>	0	128	128	
<i>sky</i>	128	128	128	
<i>mountain</i>	64	0	0	

"void" for unlabelled pixels



Metrics: Loss Function

- Averaged per pixel cross-entropy loss

```
for (inputs, targets) in train_data[0:4]:  
    inputs, targets = inputs, targets  
    outputs = dummy model(inputs.unsqueeze(0))  
    loss = torch.nn.CrossEntropyLoss(ignore_index=1, reduction='mean')  
    losses = loss(outputs,targets.unsqueeze(0))  
    print(losses)
```

- ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.

Metrics: Accuracy

- Only consider pixels which are not „void“

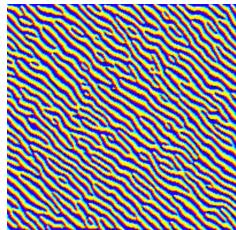
```
def evaluate_model(model):
    test_scores = []
    model.eval()
    for inputs, targets in test_loader:
        inputs, targets = inputs.to(device), targets.to(device)

        outputs = model.forward(inputs)
        , preds = torch.max(outputs, 1)
        targets_mask = targets >= 0
        test_scores.append(np.mean((preds == targets)[targets_mask].data.cpu().numpy()))

    return np.mean(test_scores)
print("Test accuracy: {:.3f}".format(evaluate_model(dummy_model)))
```

Semantic Segmentation Task

- Input shape: $(N, \text{num_channels}, H, W)$
Output shape: $(N, \text{num_classed}, H, W)$
- We want to:
 - Maintain dimensionality (H, W)
 - Get features at different spatial resolutions



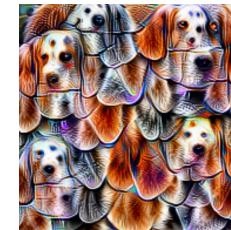
Edges



Textures



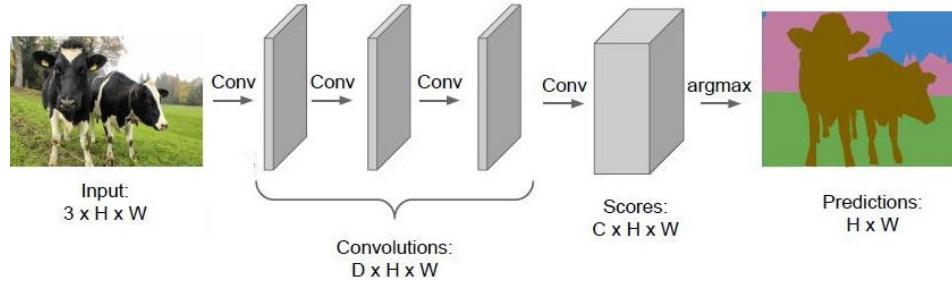
Parts



Objects

Naive Solution

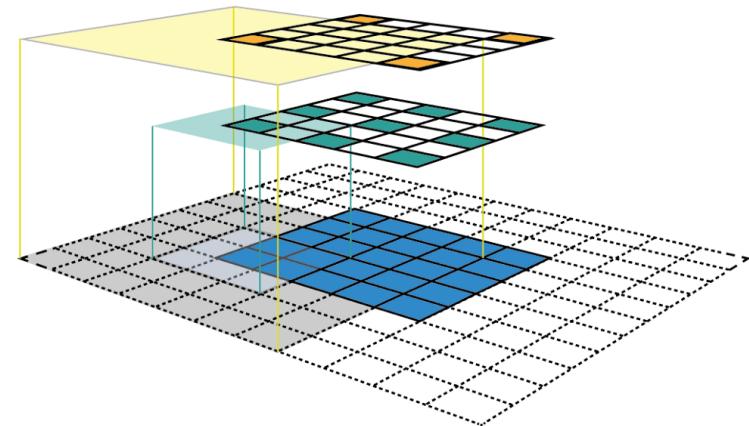
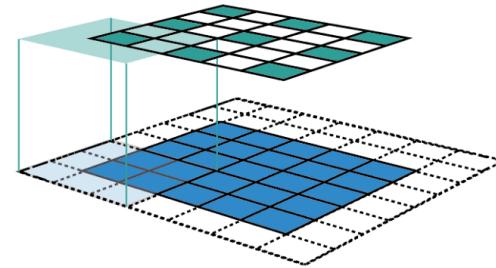
- Keep dimensionality constant throughout the network
- Use increasing filter sizes



- Problem:
 - Increased memory consumption
 - Filter size would be the same
e.g., 128 filters a $(64 \times 3 \times 3)$ \rightarrow 73k params
 - But we have to save inputs and outputs for every layer
e.g., 128 filters a $(64 \times W \times H)$ \rightarrow millions of params!

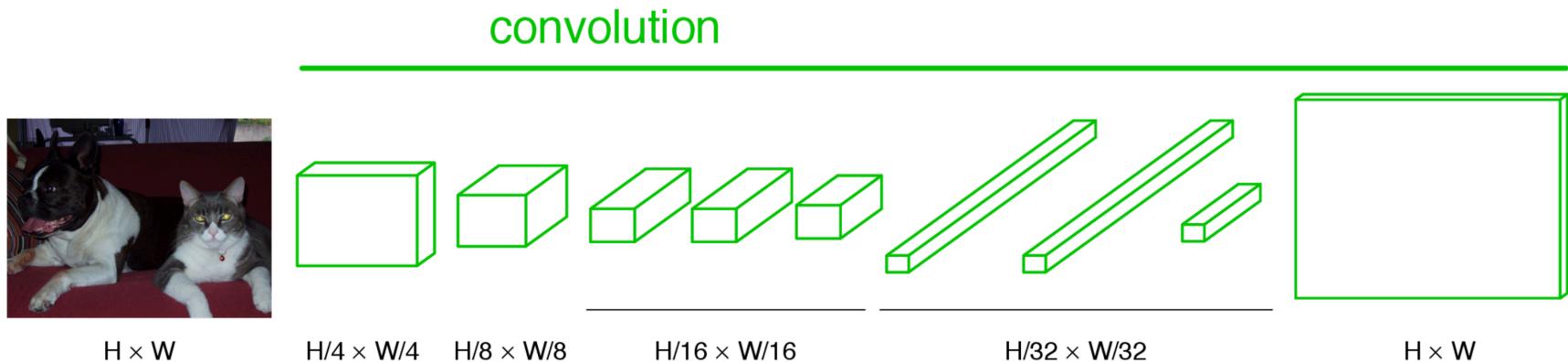
Excursion: Receptive Field (RF)

- Region in input space a feature in a CNN is looking at
- E.g., after 2 (5x5) convolutions with stride 1 we have a receptive field of 9x9
(RF after first conv: 5
RF after second conv: 5+4)



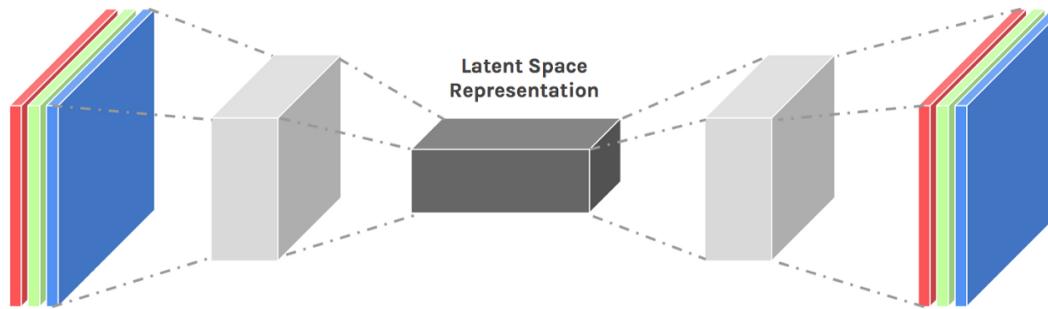
Coming from Classification

- Use strided convolutions and pooling to increase the receptive field
- Upsample result to input resolution



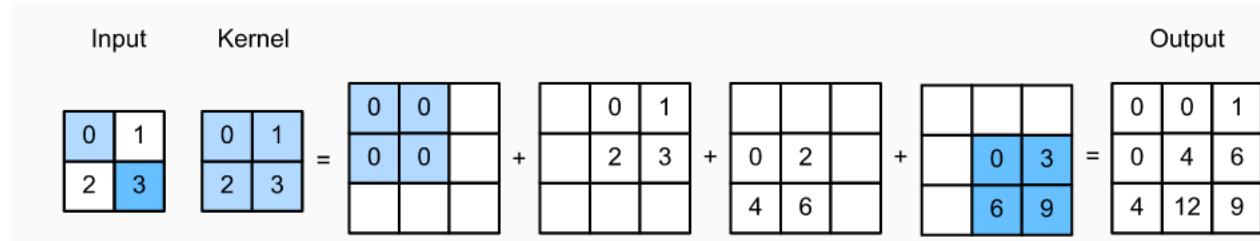
Better Solution

- Slowly reduce size -> slowly increase size
 - Pooling -> Upsampling
 - Strided convolution -> Transposed convolution
- Combine with normal convolutions, bn, dropout, etc.



Transposed Convolutions

- Upsampling with learnable parameters



- Output size computation:

- Regular conv layer:

$$out = \frac{(in - kernel + 2 * pad)}{stride} + 1$$

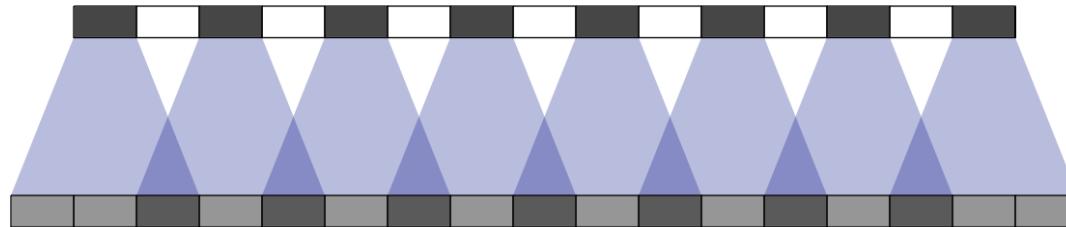
- Transpose convolution for multiples of 2

$$out = (in - 1) * stride - 2 * pad + kernel$$

(Transpose computation not relevant for the exam,
more info here: https://github.com/vdumoulin/conv_arithmetic)

Are transpose convolutions superior?

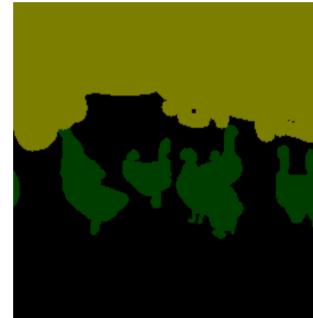
- Short answer: no, not always
- Long answer: possible checkerboard artifacts for general image generation, see
<https://distill.pub/2016/deconv-checkerboard/>



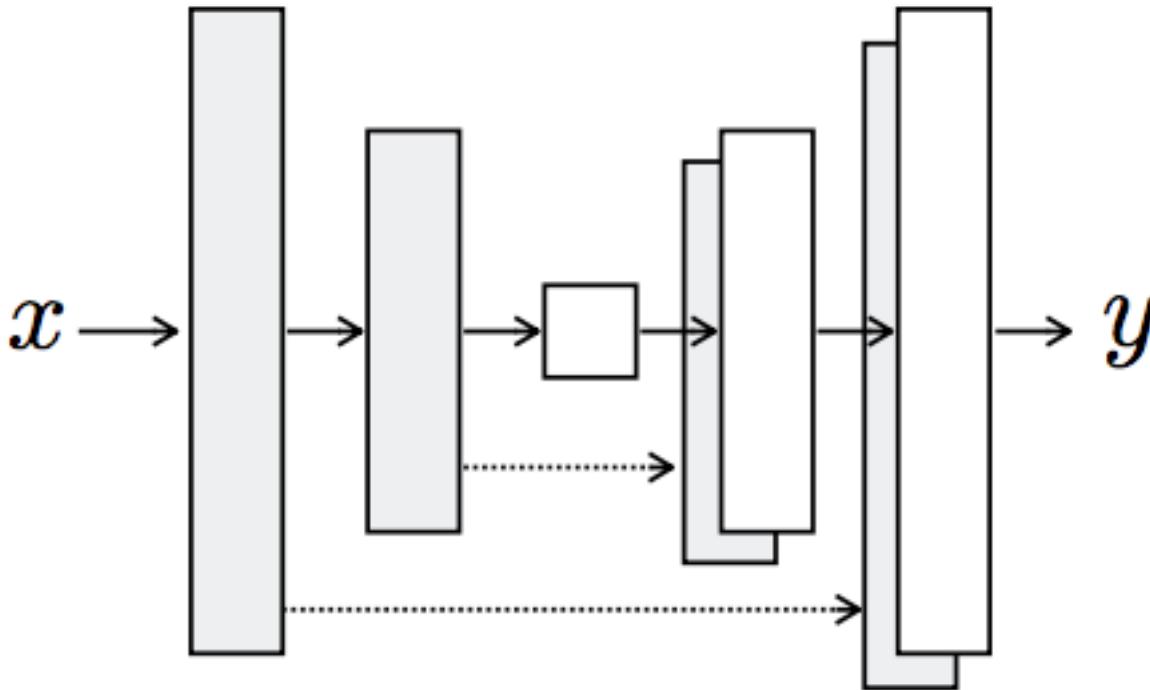
- My personal go-to:
 - Regular upsampling, followed by a convolution layer

Submission: Details

- Submission Start: July 02, 2020 12.00
- Submission Deadline : July 08, 2020 23.59
- Your model's accuracy is all that counts!



Advanced architecture: U-Net & Skip Connections



Good luck &
see you next week

