

Deep learning

Unpacking Transformers, LLMs and image generation

Session 5

Syllabus

Session	Lecture	TP
1	Intro to DL Gradient descent and backprop	Intro to micrograd (*) fit the x^2 function
2	DL fundamentals I <ul style="list-style-type: none">• Backprop• Loss functions• Neural Probabilistic Language Model (Bengio 2003)	Bigram model and MLP for next-character prediction (*) extend to tri-gram
3	DL fundamentals II <ul style="list-style-type: none">• Activation function• Regularization• Initialization• Residual networks• Normalization Recurrent Neural Networks	Backprop ninja MLP in pytorch (*) add batchnorm to TP2
4	Attention and Transformers	GPT from scratch
5	DL for computer vision: convnets, <u>unets</u>	Convnets for CIFAR-10
6	VAE & Diffusion models	1D diffusion model 2D diffusion model (*) train on GPU Quiz

More depth is not always better

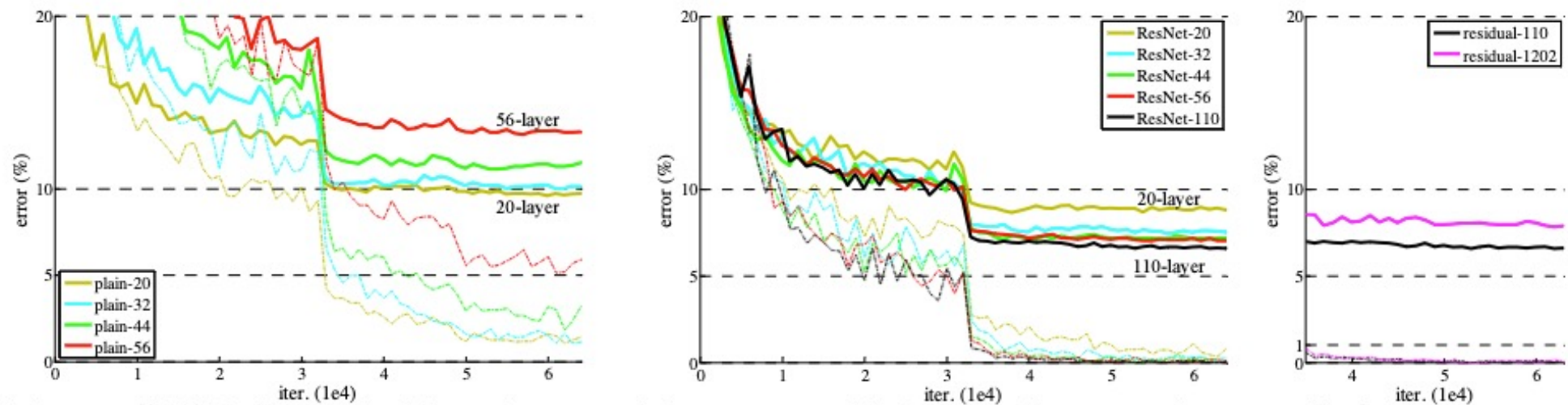


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left:** plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle:** ResNets. **Right:** ResNets with 110 and 1202 layers.

Do better architectures learn better than deeper models?

Convolutions

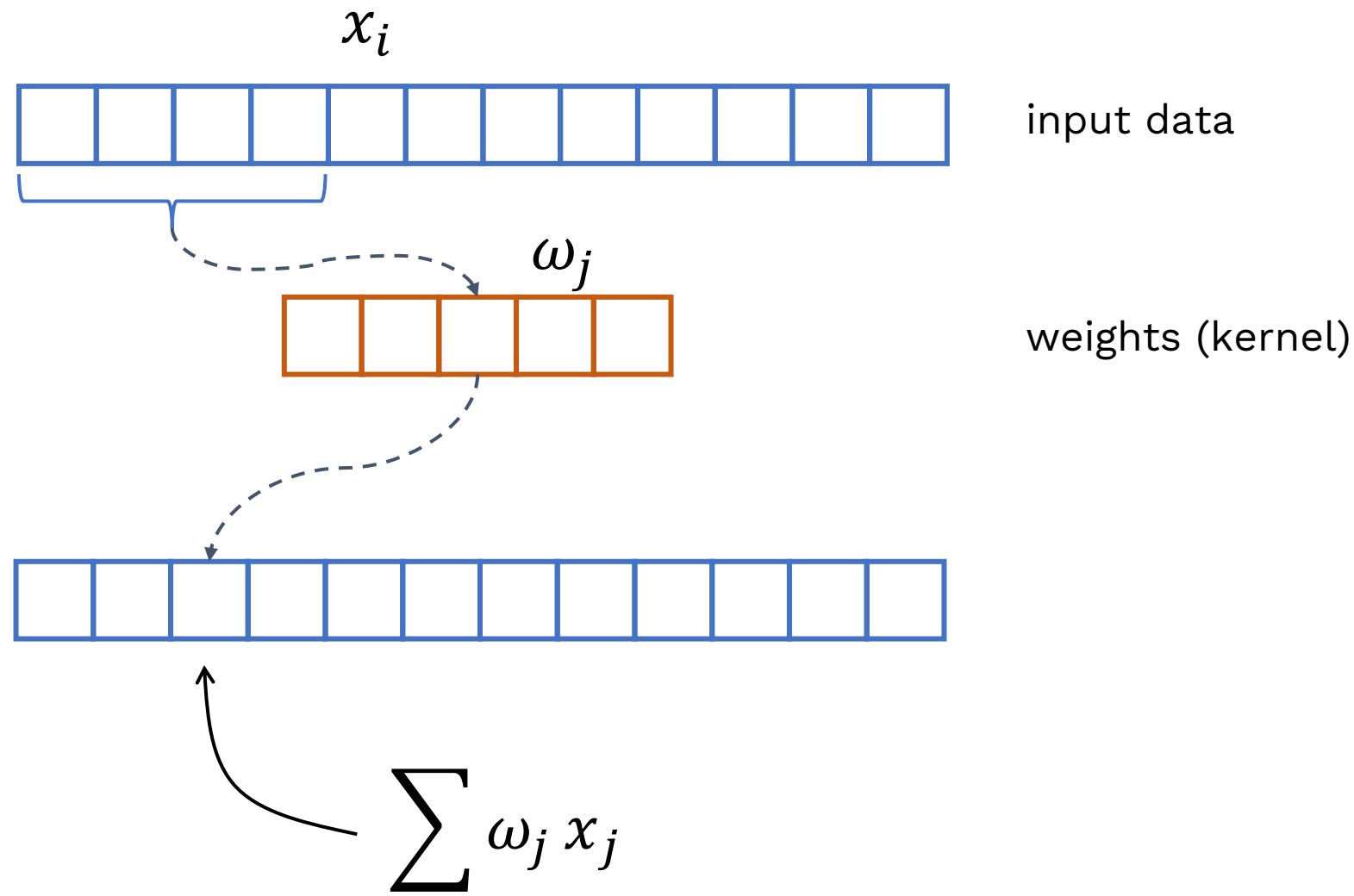


input data

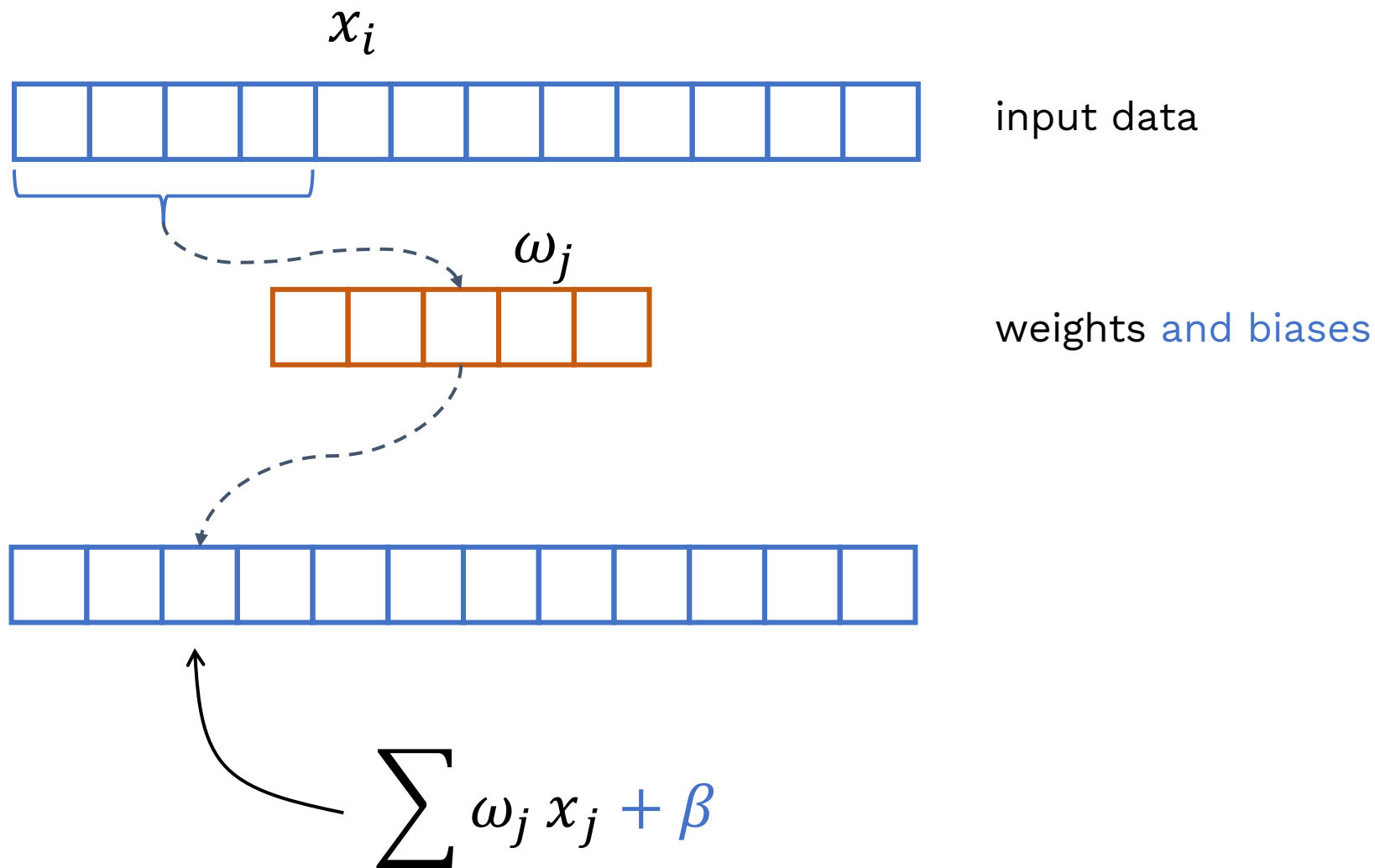


weights

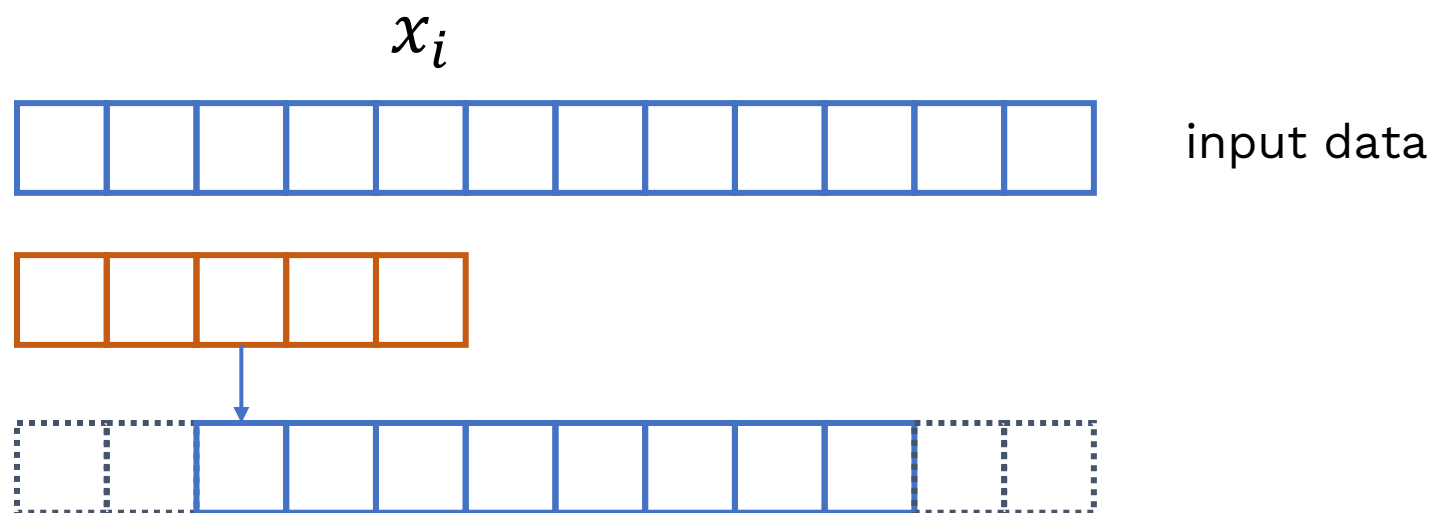
Convolutions



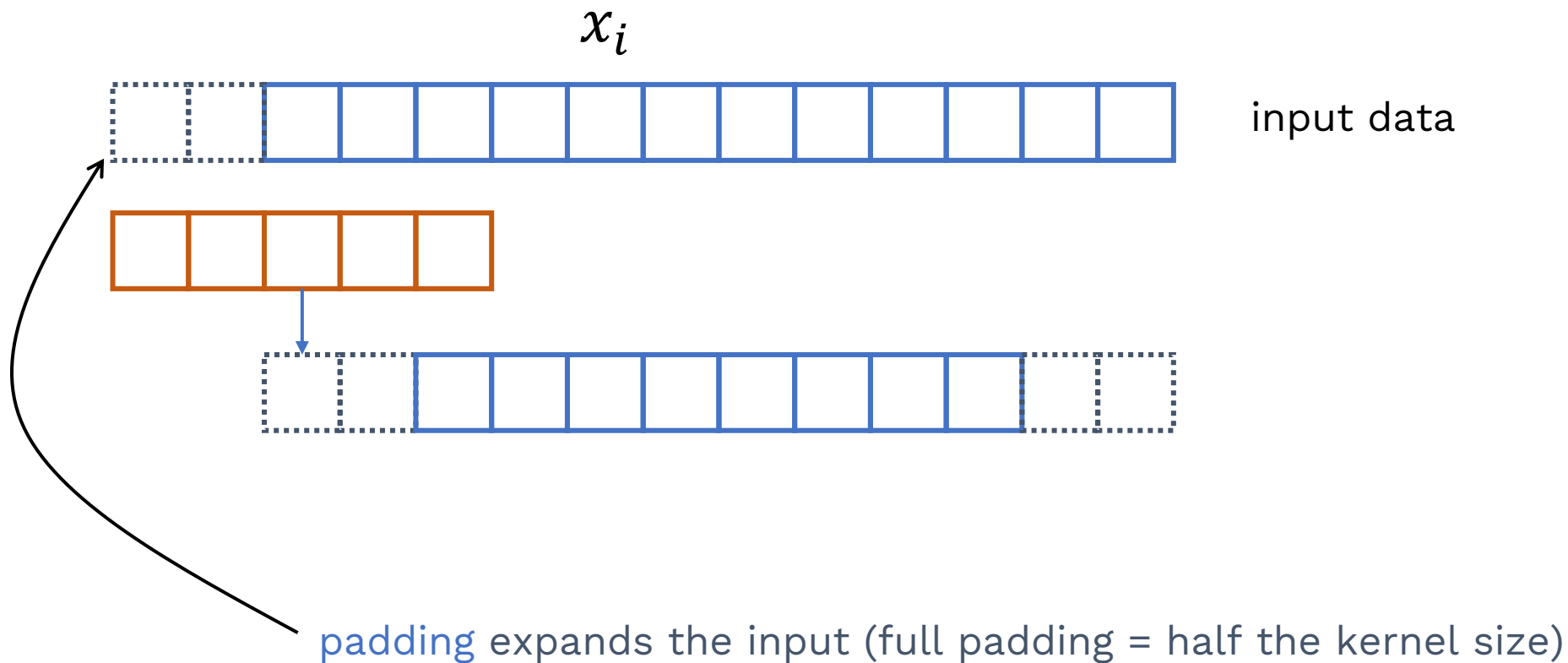
Convolutions



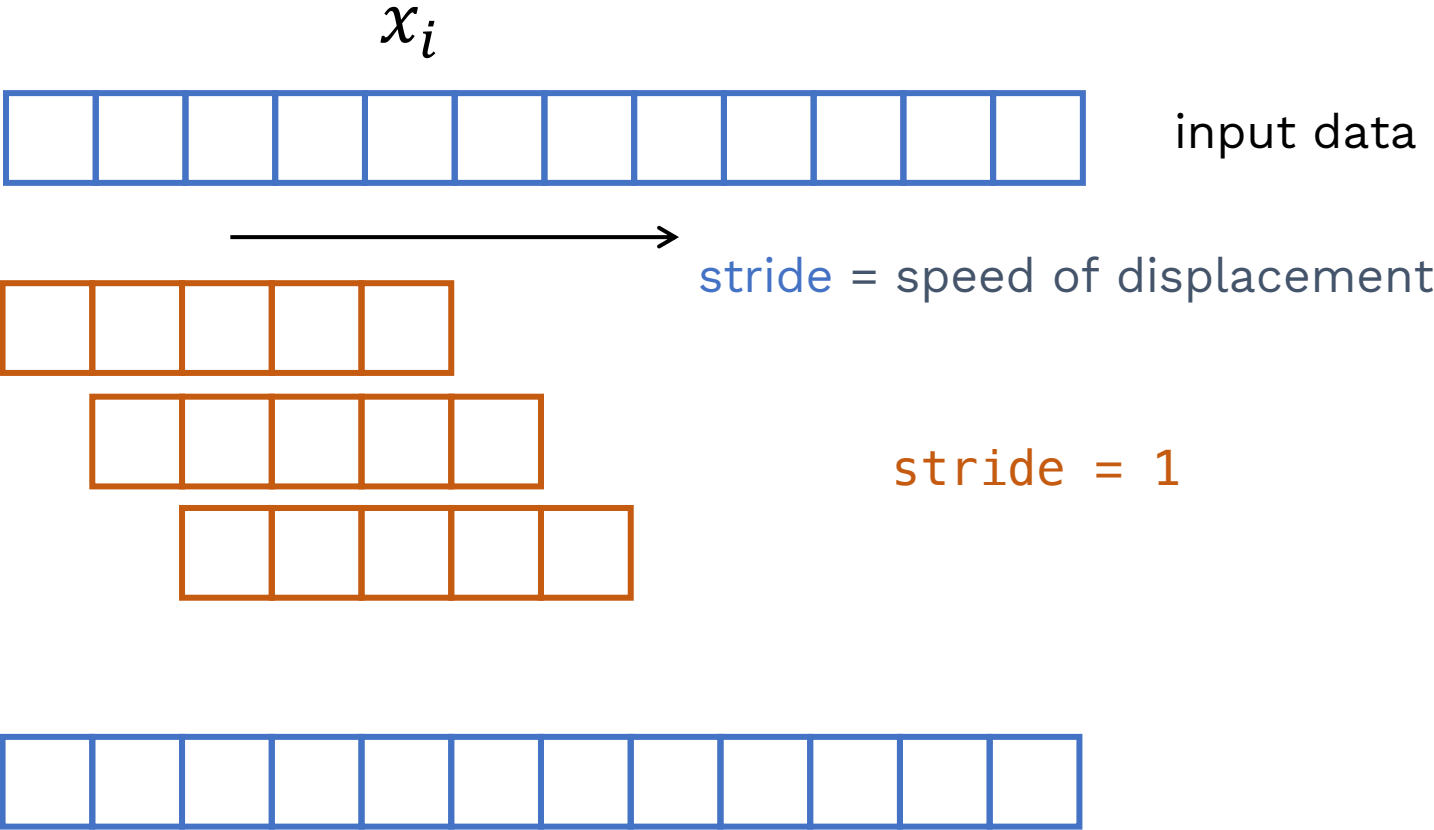
Convolutions



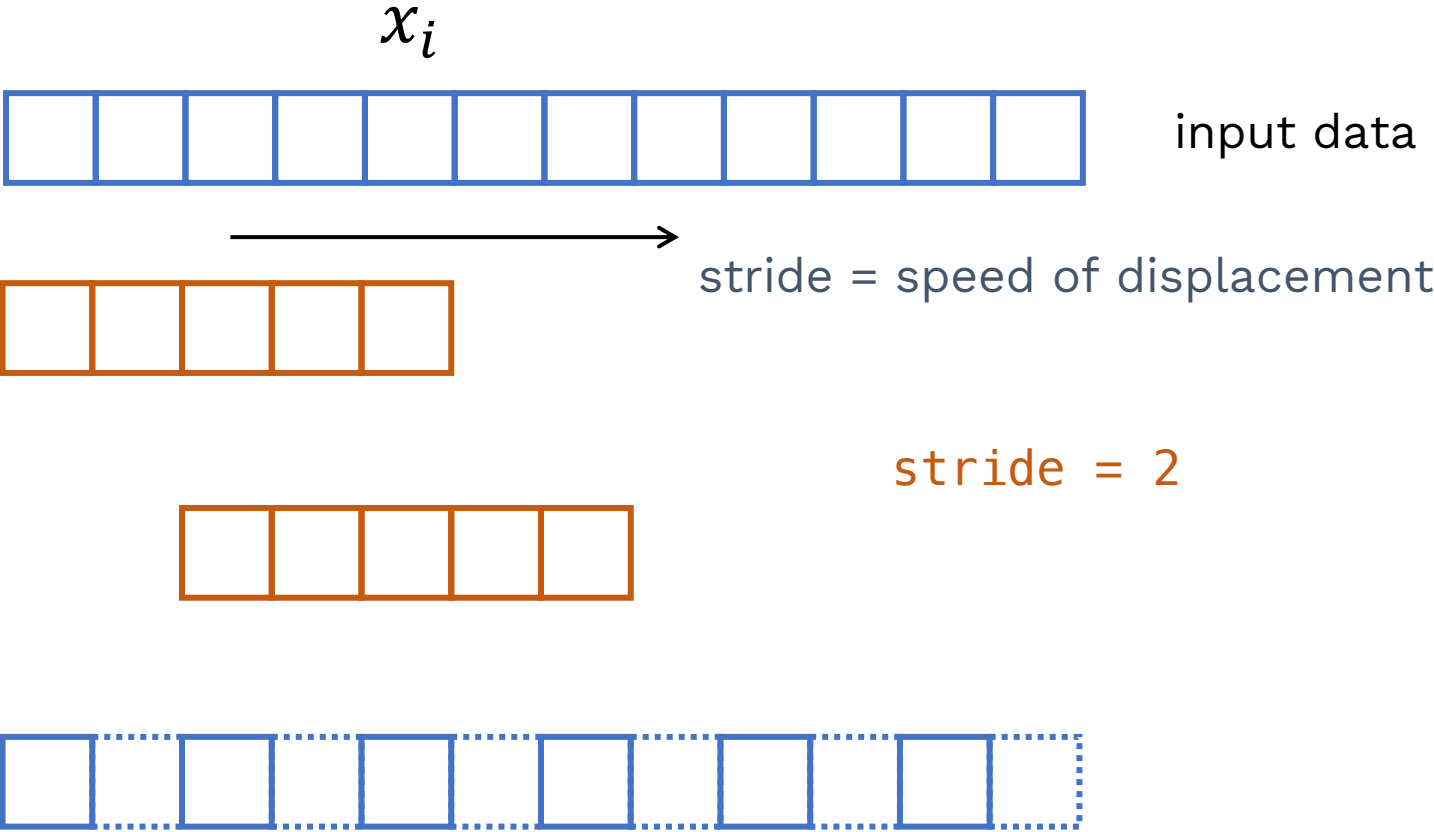
Convolutions



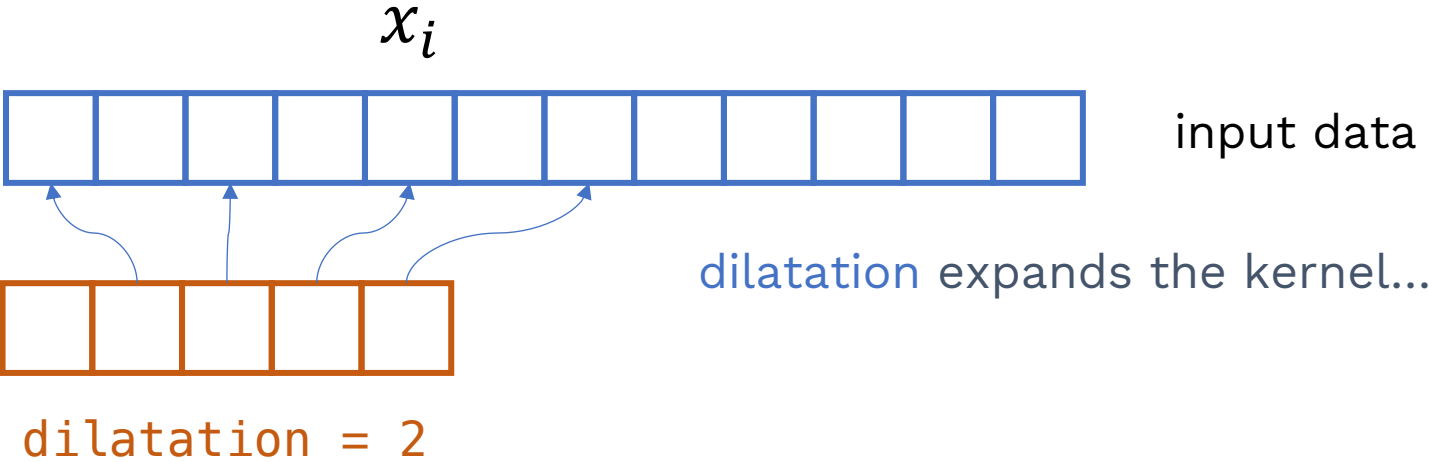
Convolutions



Convolutions



Convolutions



... and effectively requires further padding



Convolutions



$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$



Convolutions

input dimension

output dimension



```
m = nn.Conv1d(1, 1, kernel_size=5, stride=1, dilation=1, bias=False)
```

```
input = torch.ones((1,16))
```

```
m(input).shape
```

```
torch.Size([1, 12])
```

1D Convolutions

input dimension

output dimension

```
m = nn.Conv1d(1, 1, kernel_size=5, stride=1, dilation=1, bias=False)
```

x_i



L_{in}



L_{out}

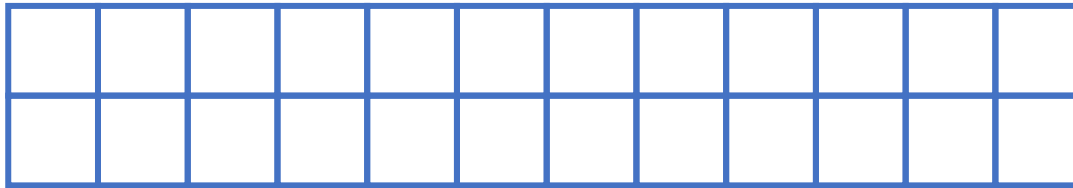
1D Convolutions

input dimension

output dimension

```
m = nn.Conv1d(2, 1, kernel_size=5, bias=False)
```

x_i



L_{in}



L_{out}

```
m.weight.shape  
> [1, 2, 5]
```

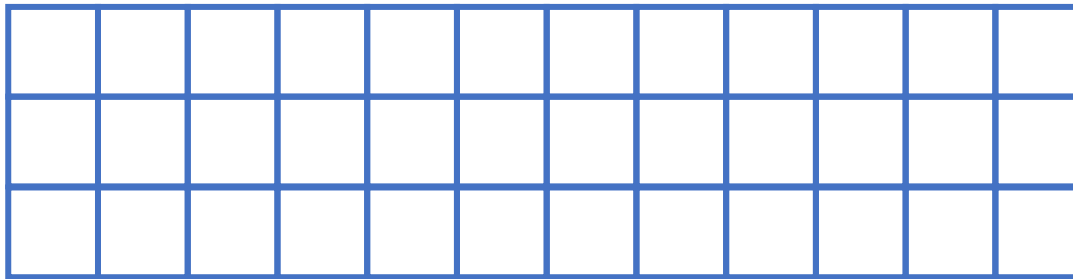

1D Convolutions

input dimension

output dimension

```
m = nn.Conv1d(2, 1, kernel_size=5, bias=False)
```

x_i



L_{in}

```
m.weight.shape  
> [1, 2, 5]
```



RuntimeError: weight of size [1, 2, 5], expected input[1, 3, 7] to have 2 channels, but got 3 channels instead

1D Convolutions

input dimension

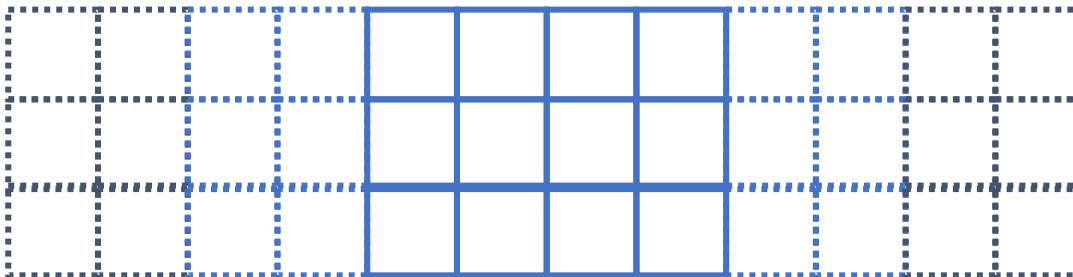
output dimension

```
m = nn.Conv1d(1, 3, kernel_size=5, stride=1, dilation=1, bias=False)
```

x_i



L_{in}



L_{out}

```
m.weight.shape  
> [3, 1, 5]
```

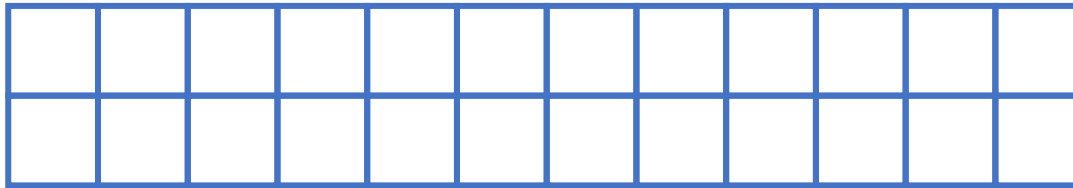
1D Convolutions

input dimension

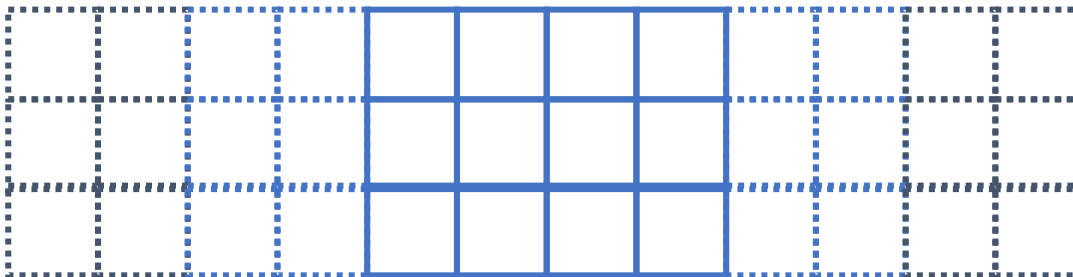
output dimension

```
m = nn.Conv1d(2, 3, kernel_size=5, bias=False)
```

x_i



L_{in}



L_{out}

```
m.weight.shape  
> [3, 2, 5]
```

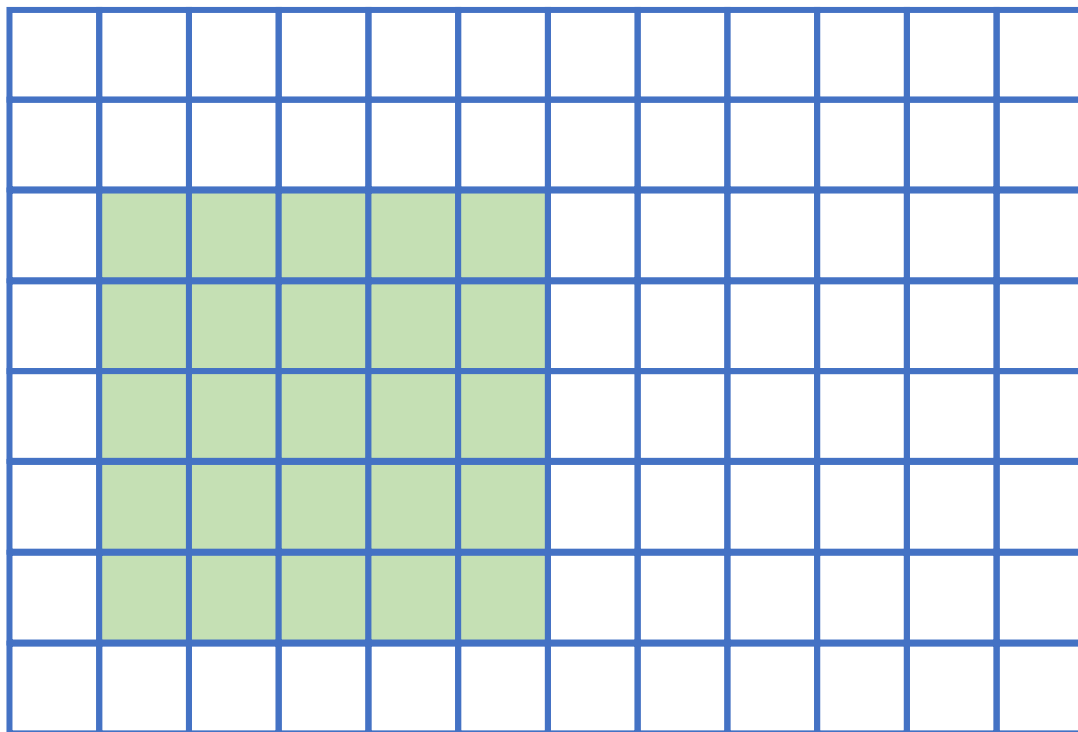
2D Convolutions

input dimension

output dimension

```
m = nn.Conv2d(1, 1, kernel_size=5, bias=False)
```

x_i



`input.shape`
> [N, Cin, H, W]

`m.weight.shape`
> [1, 1, 5, 5]

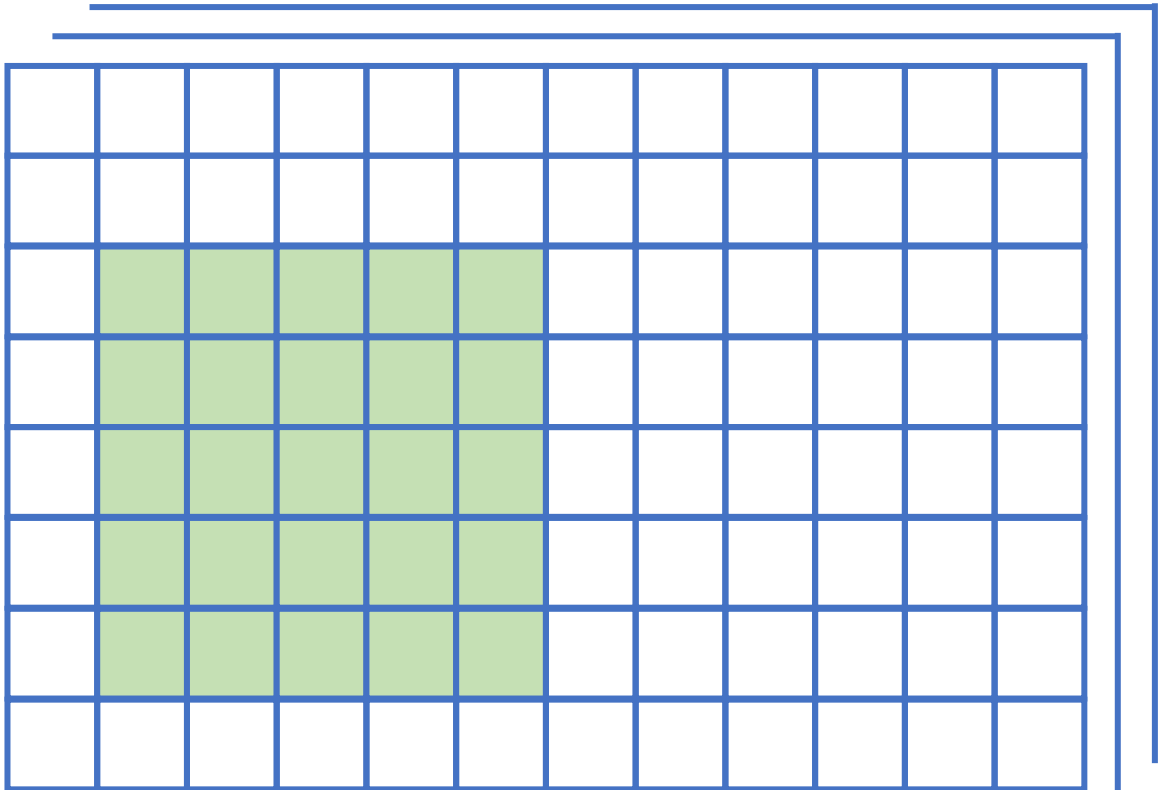
2D Convolutions

input dimension

output dimension

```
m = nn.Conv2d(3, 3, kernel_size=5, bias=False)
```

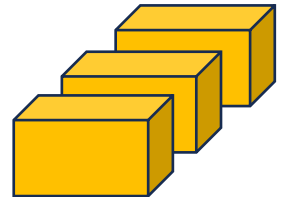
x_i



L_{in}

`input.shape`
> [N, Cin, H, W]

`m.weight.shape`
> [3, 3, 5, 5]



2D Convolutions

```
m = nn.Conv2d(Cin, Cout, kernel_size=5, stride=2, padding=2)
```

```
input = torch.ones(N, Cin, H, W)
```

batch size

Image size (HxW)

number of channels = input dimension

```
output = torch.ones(N, Cout, Hout, Wout)
```

$$L_{out} = \left\lceil \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel_size} - 1) - 1}{\text{stride}} + 1 \right\rceil$$

2D Convolutions

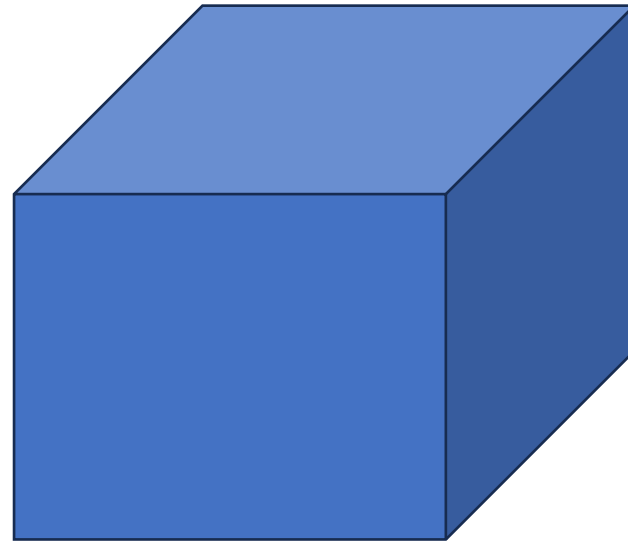
input dimension

output dimension

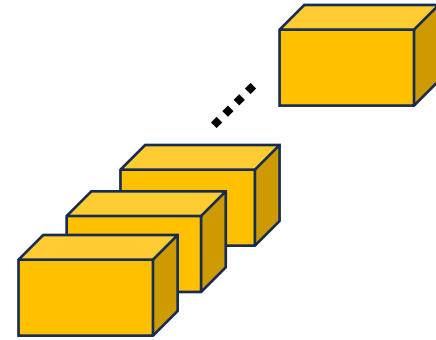
```
m = nn.Conv2d(3, 64, kernel_size=5, padding=2, stride=2)
```



`input.shape`
> [12, 3, 224, 224]



`output.shape`
> [12, 64, 112, 112]



`m.weight.shape`
> [64, 3, 5, 5]

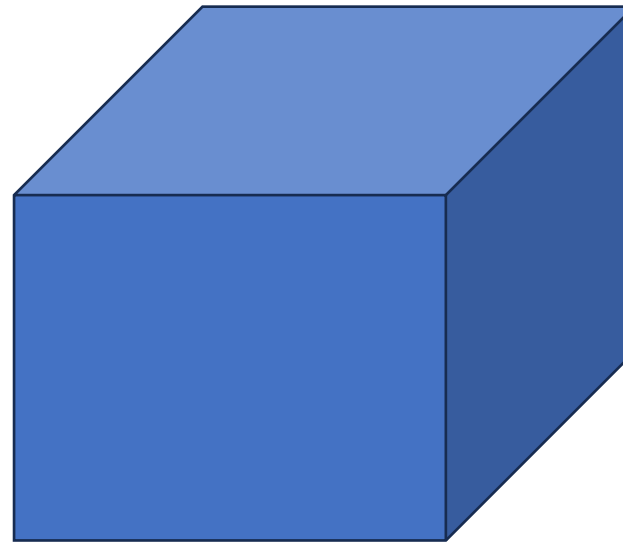
2D Convolutions

Kernels are the same across the image (inductive bias). Shape is independent of image size.

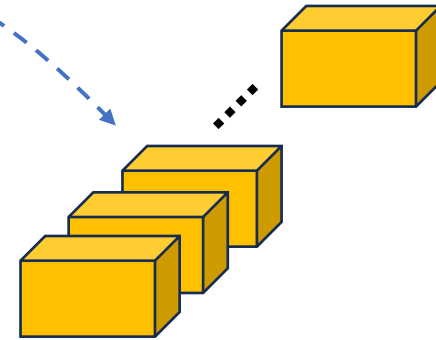
```
m = nn.Conv2d(3, 64, kernel_size=5, padding=2, stride=2)
```



`input.shape`
> [12, 3, 224, 224]



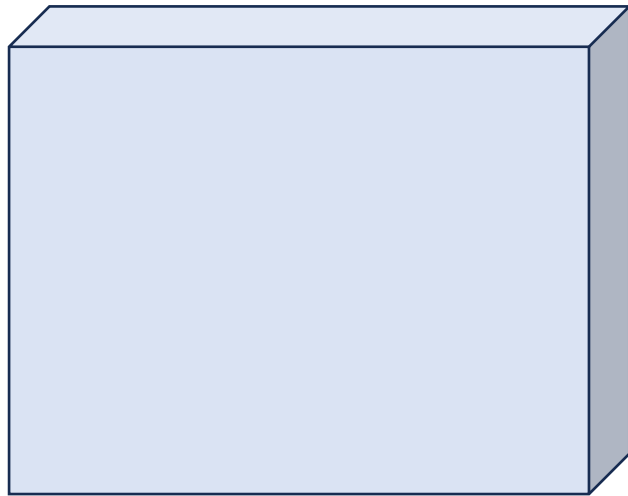
`output.shape`
> [12, 64, 112, 112]



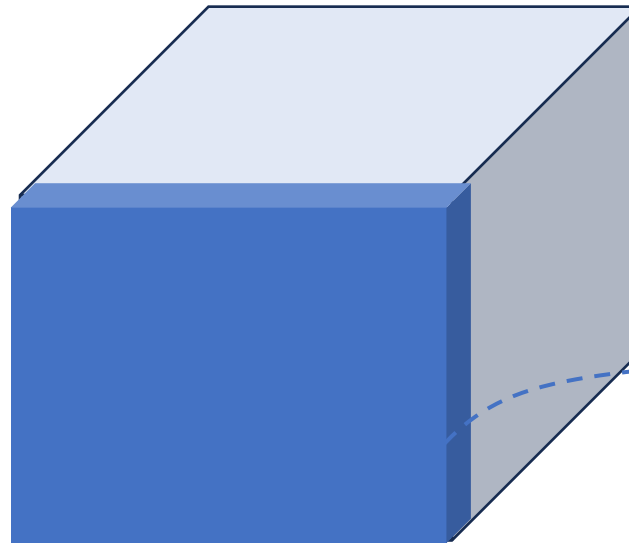
`m.weight.shape`
> [64, 3, 5, 5]

2D Convolutions

```
m = nn.Conv2d(3, 64, kernel_size=5, padding=2, stride=2)
```

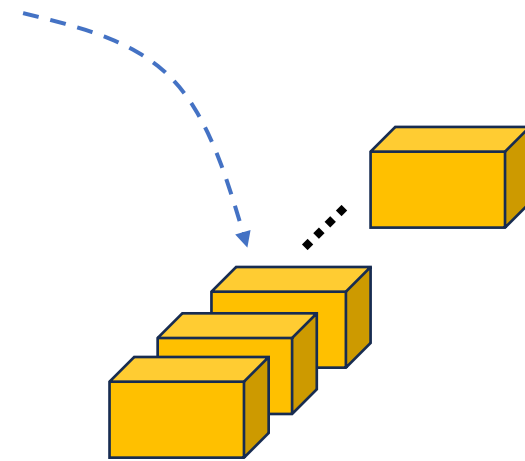


input.shape
> [12, 3, 224, 224]



output.shape
> [12, 64, 112, 112]

filter / kernels

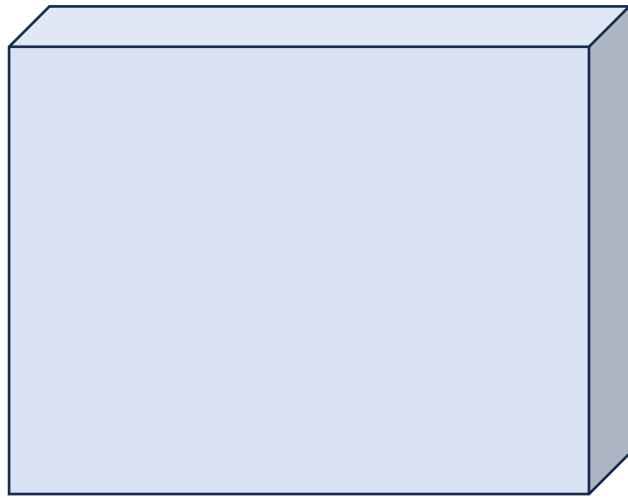


m.weight.shape
> [64, 3, 5, 5]

activation maps

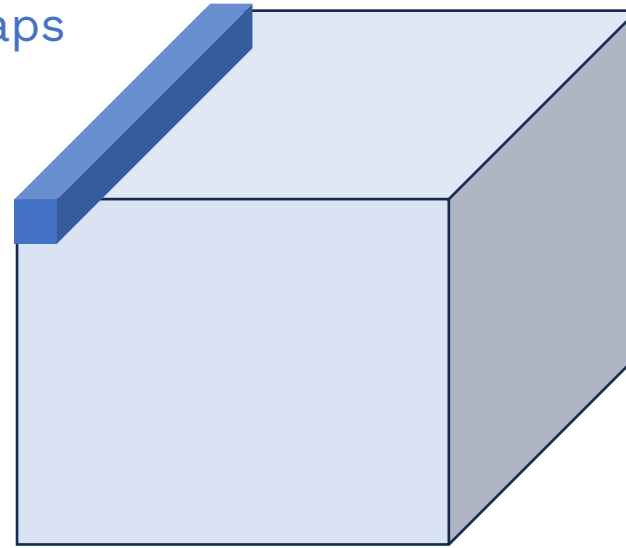
2D Convolutions

```
m = nn.Conv2d(3, 64, kernel_size=5, padding=2, stride=2)
```

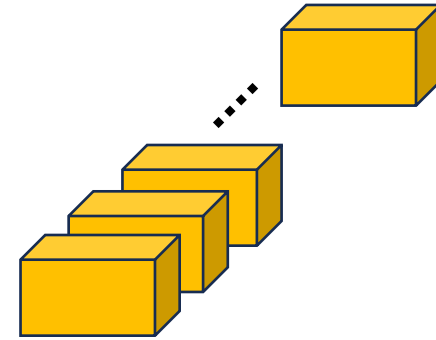


`input.shape`
> [12, 3, 224, 224]

feature maps



`output.shape`
> [12, 64, 112, 112]



`m.weight.shape`
> [64, 3, 5, 5]

ResNet architecture

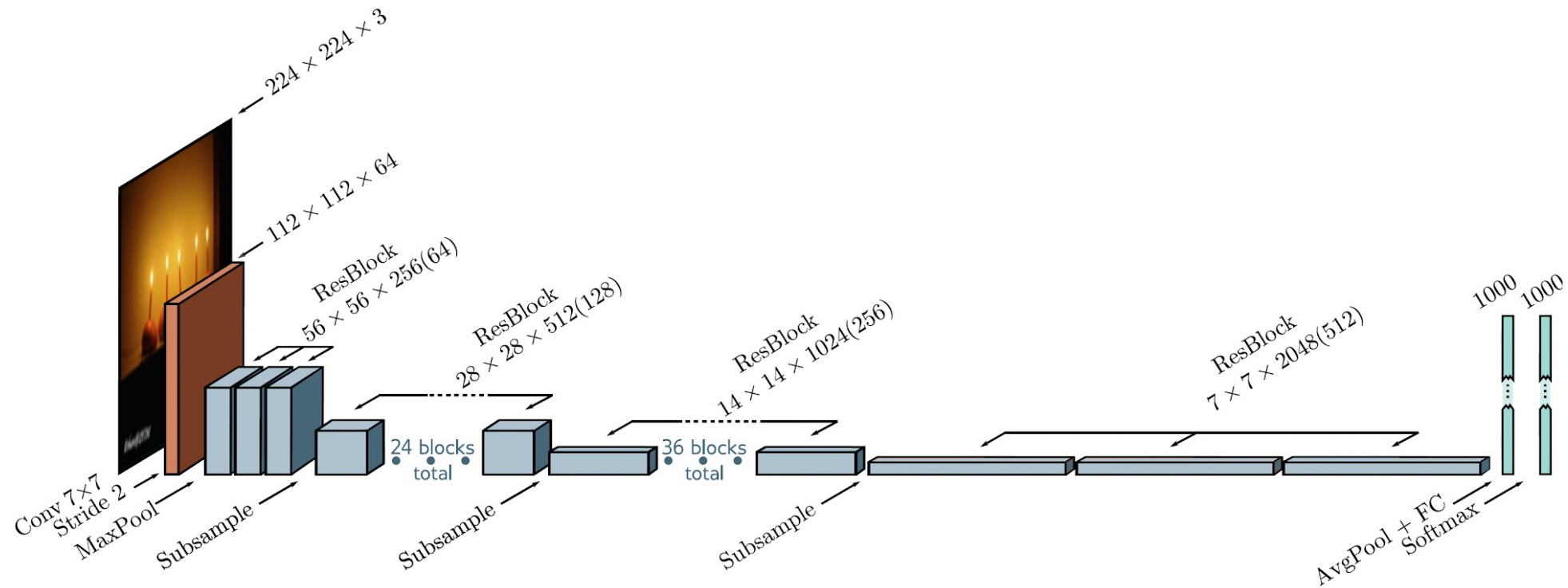
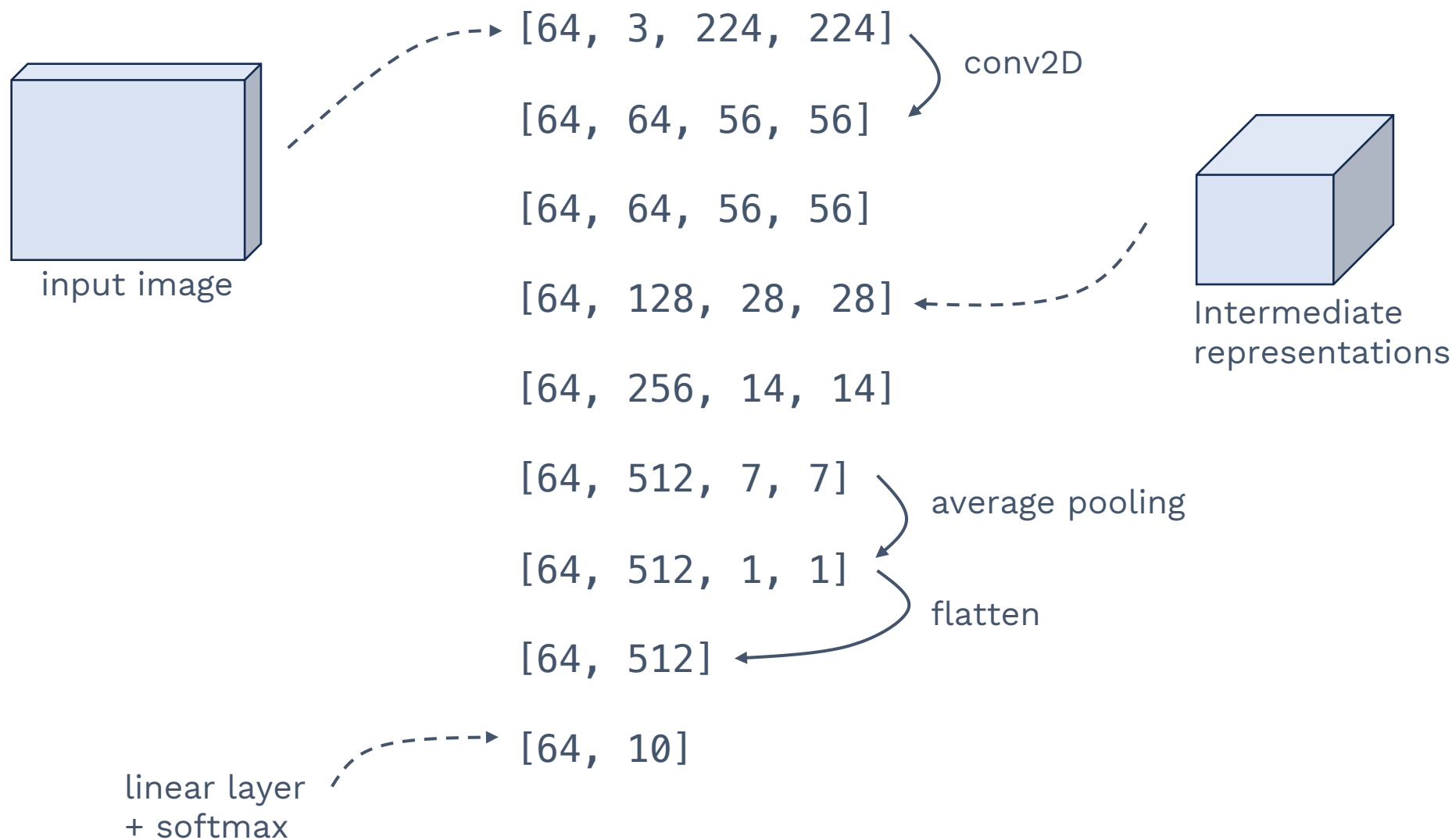


Figure 11.8 ResNet-200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling and accompanying increases in the number of channels. The network concludes with average pooling across all spatial positions and a fully connected layer that maps to pre-softmax activations.

2D Convolutions



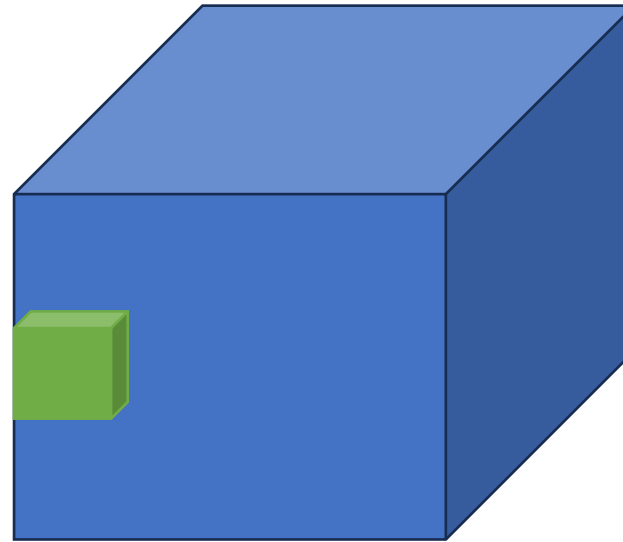
Max Pooling

```
m = nn.MaxPool2d(kernel_size = 3)
input = torch.ones((11, 3, 112, 112))
```

```
output.shape
> [11, 3, 37, 37]
```



default stride = kernel size



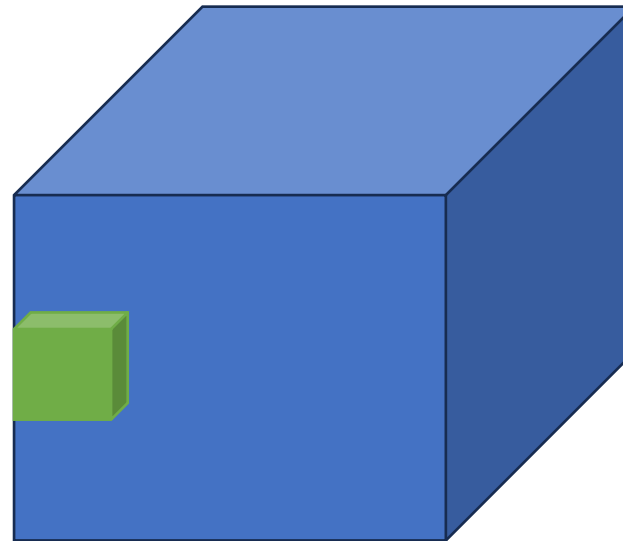
Average Pooling

```
m = nn.AvgPool2D(kernel_size = 3)
input = torch.ones((11, 3, 112, 112))
```

```
output.shape
> [11, 3, 37, 37]
```



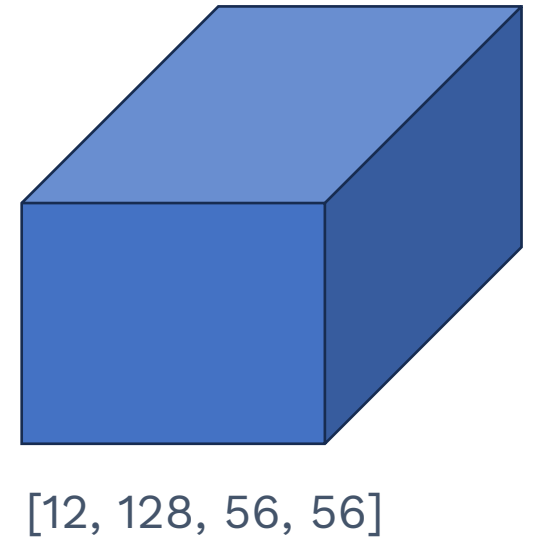
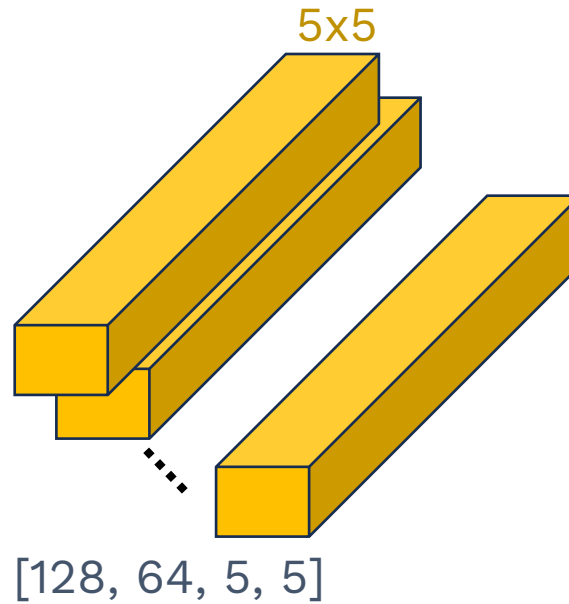
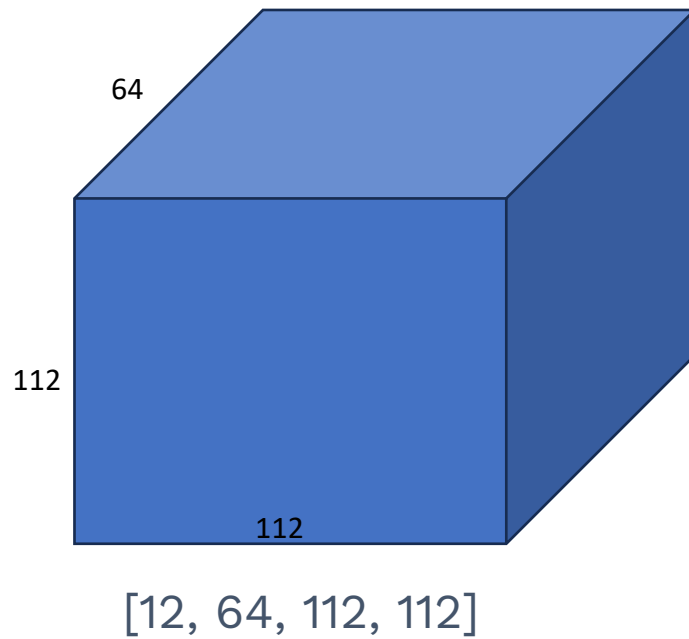
default stride = kernel size



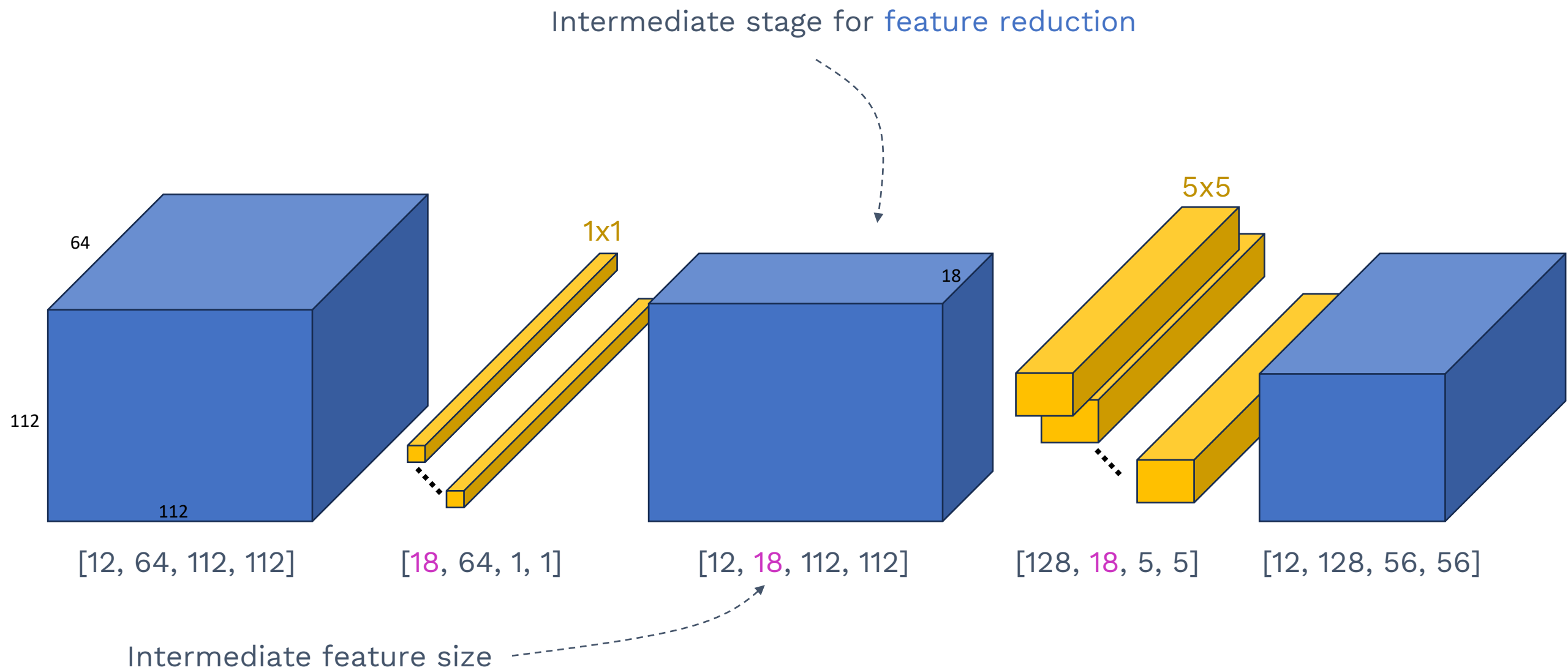
Convolutions are expensive!

204,800 parameters!

$128 \times 64 \times 5 \times 5 \times (112 \times 112) = 2,569,011,200$ operations!



The 1x1 convolution trick

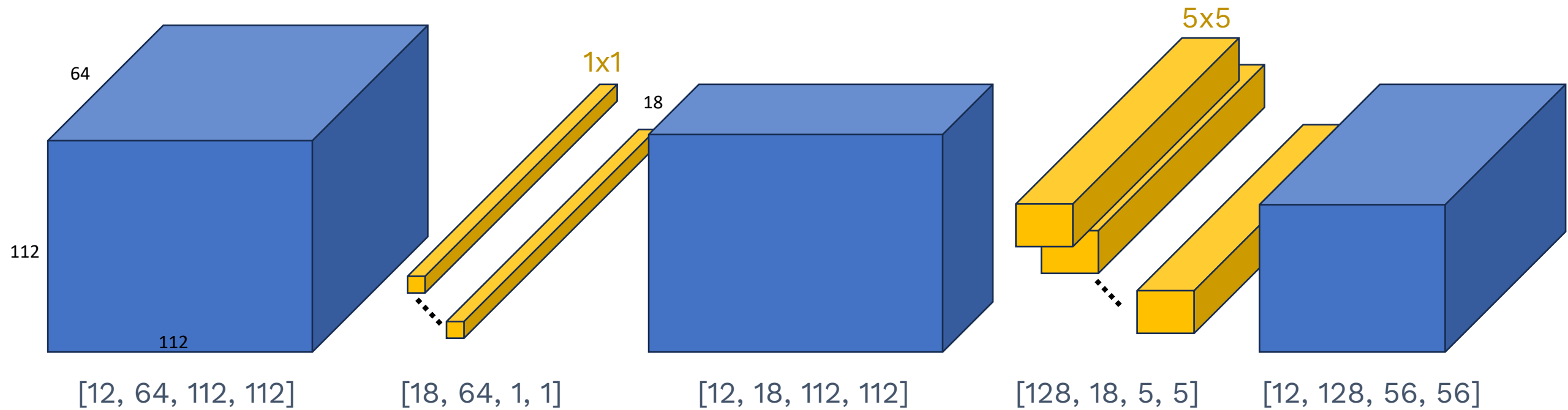


The 1x1 convolution trick

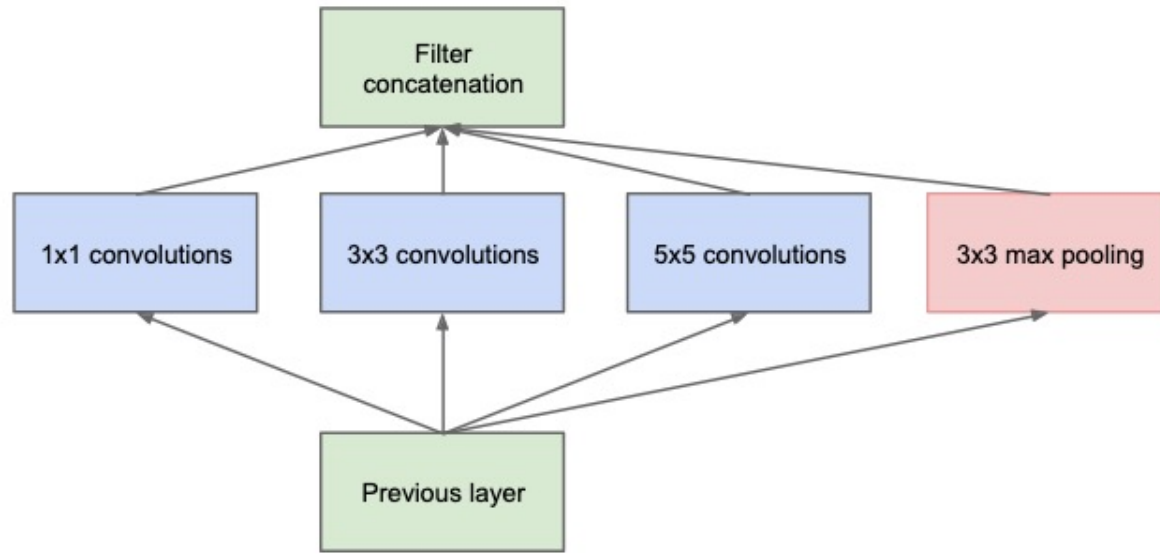
58,752 parameters

3x less parameters!

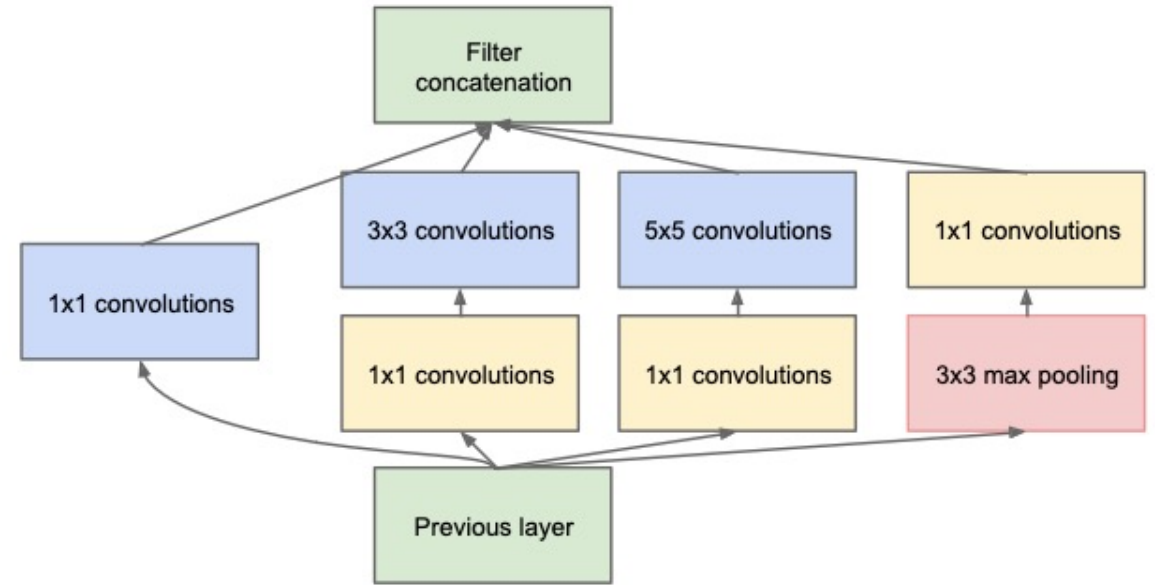
$18 \times 64 \times 1 \times 1 \times 112 \times 112 = 14,450,688$ operations 3x faster!
 $128 \times 18 \times 5 \times 5 \times 112 \times 112 = 722,534,400$ operations
736,985,088 operations in total



The Inception architecture



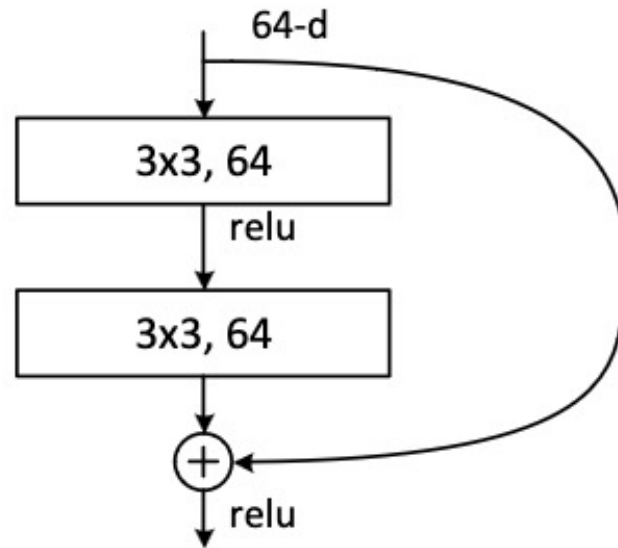
(a) Inception module, naïve version



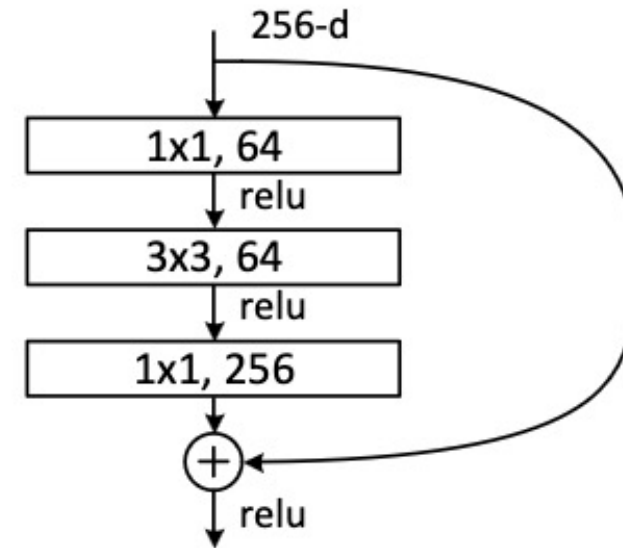
(b) Inception module with dimension reductions

Figure 2: Inception module

The ResNet architecture



Naïve block



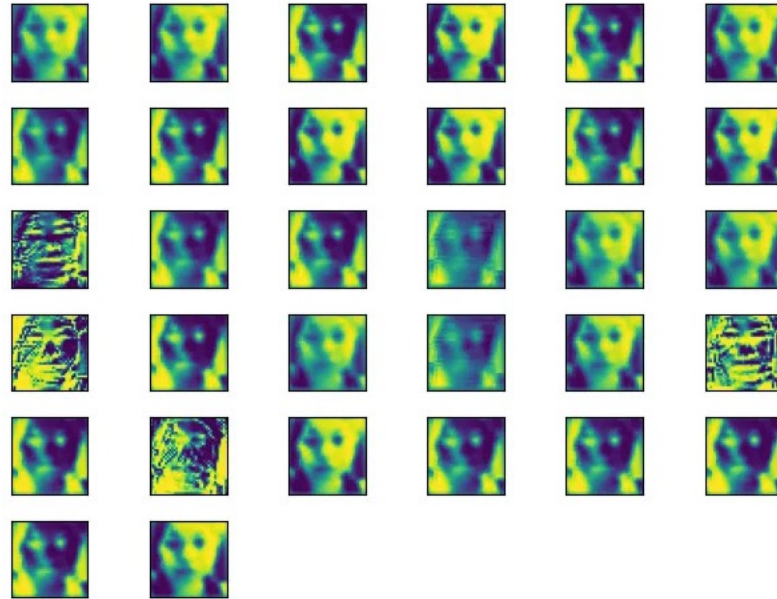
Bottleneck block

The 1x1 convolution trick: do we maintain information?

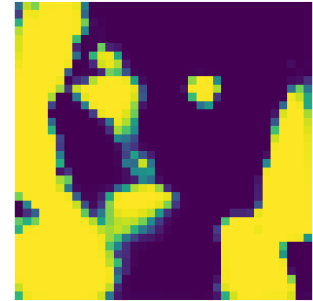
Task: skin detector



Source image



Activation maps in 32D



Activation maps with 1D pooling

The 1x1 convolution trick

1x1 convolution layers are useful to:

1. Reduce the computational load
2. Reduce feature maps dimensionality
3. Add non-linearity along the channel dimension
4. Create smaller models with the same level of accuracy

The ResNet blocks

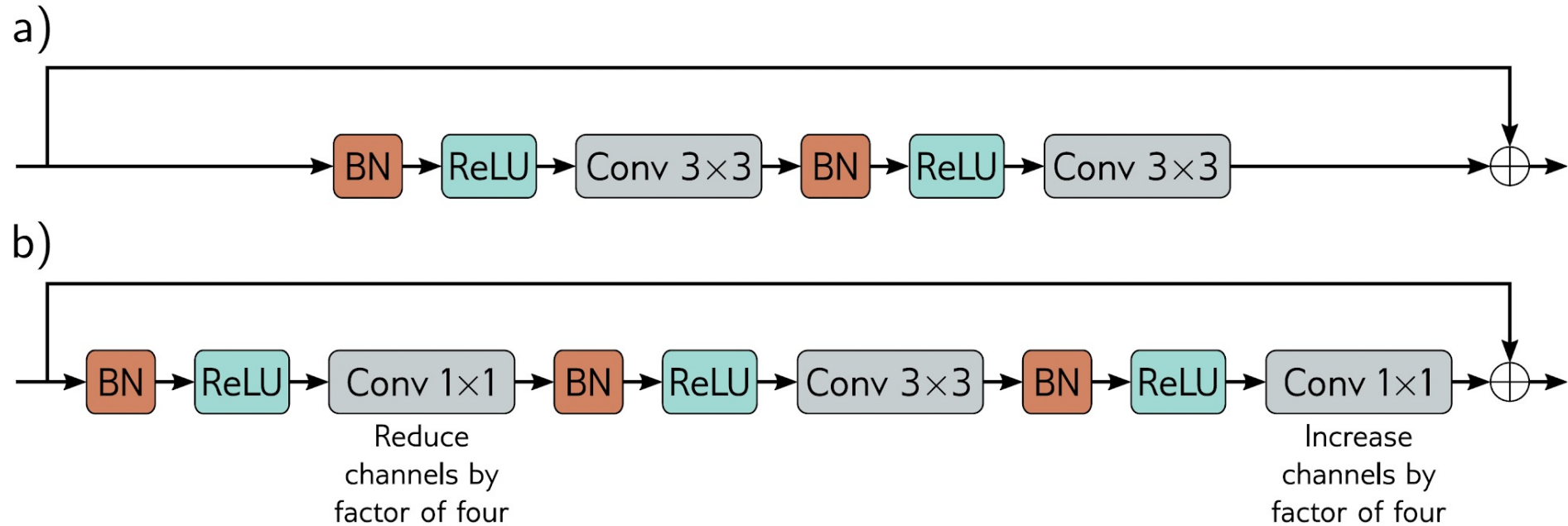
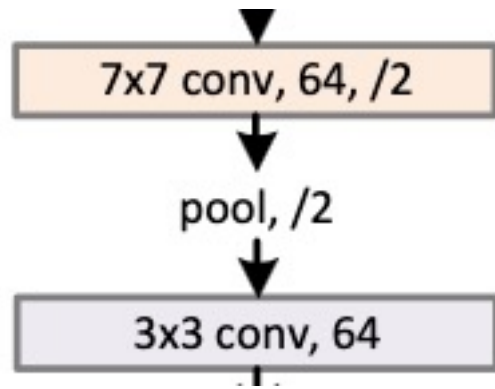


Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses fewer parameters. It contains three convolutions. The first 1×1 convolution reduces the number of channels. The second 3×3 convolution is applied to the smaller representation. A final 1×1 convolution increases the number of channels again so that it can be added back to the input.

The ResNet blocks



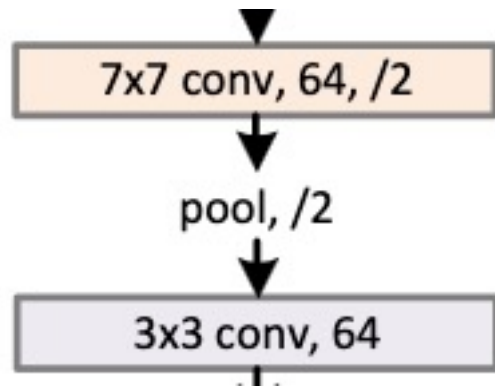
```
nn.Conv2d(..., 64, kernel_size=7, stride=2, padding=3)  
nn.BatchNorm2d(64)  
nn.ReLU()
```

An arrow points from the first line of code, `nn.Conv2d(..., 64, kernel_size=7, stride=2, padding=3)`, to the "7x7 conv, 64, /2" box in the diagram.

The ResNet blocks



The ResNet blocks



```
nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)  
nn.BatchNorm2d(64)  
nn.ReLU()
```

An arrow points from the text "nn.BatchNorm2d(64)" to the "3x3 conv, 64" box in the diagram.

Tips

You can stack modules in a layer with `nn.Sequential()`

```
net = [nn.Conv2d(12, 14, kernel_size=3), nn.Conv2d(12, 14, kernel_size=3)]  
m = nn.Sequential(*net)  
sum([p.numel() for p in m.parameters()])
```

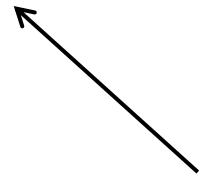
3052

Tips

But **beware**, the following does not work!

```
net = [nn.Conv2d(12, 14, kernel_size=3)] * 2  
m = nn.Sequential(*net)  
sum([p.numel() for p in m.parameters()])
```

1526



You would expect 3052 here!

Tips

Instead, use a for loop

```
net = []  
for _ in range(2):  
    net += [nn.Conv2d(12, 14, kernel_size=3)]  
m = nn.Sequential(*net)  
sum([p.numel() for p in m.parameters()])
```

3052

Practical 5: ResNet from scratch

Step 1: a stack of convnets (PlainNet)

Step 2: a standard ResNet

Step 3: a ResNet with bottlenecks

Learning:

With standard convnets, more depth does not mean better performance.

With ResNet, better performance with as many parameters.

With bottlenecks, better performance with more depth.

Practical 5: ResNet from scratch

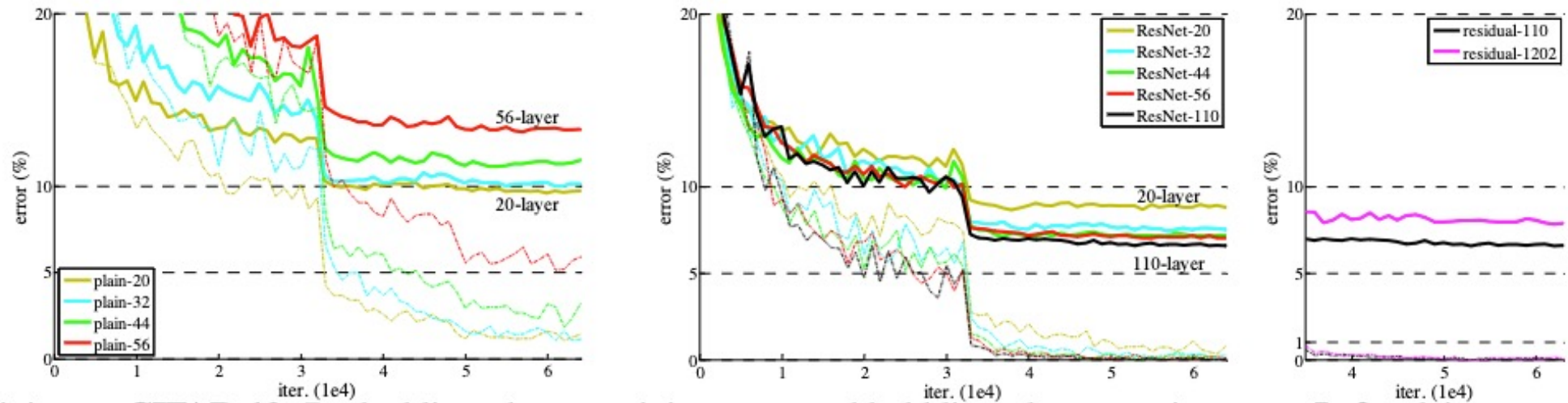


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left:** plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle:** ResNets. **Right:** ResNets with 110 and 1202 layers.

DenseNet

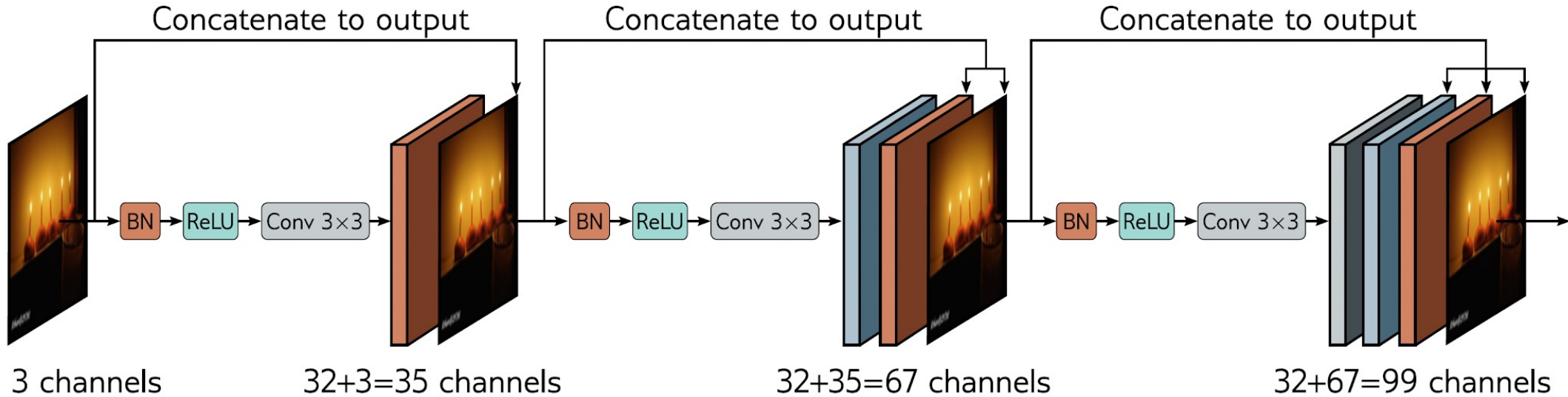


Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation, and both earlier representations are concatenated to this to create a total of 67 channels and so on.

U-Net

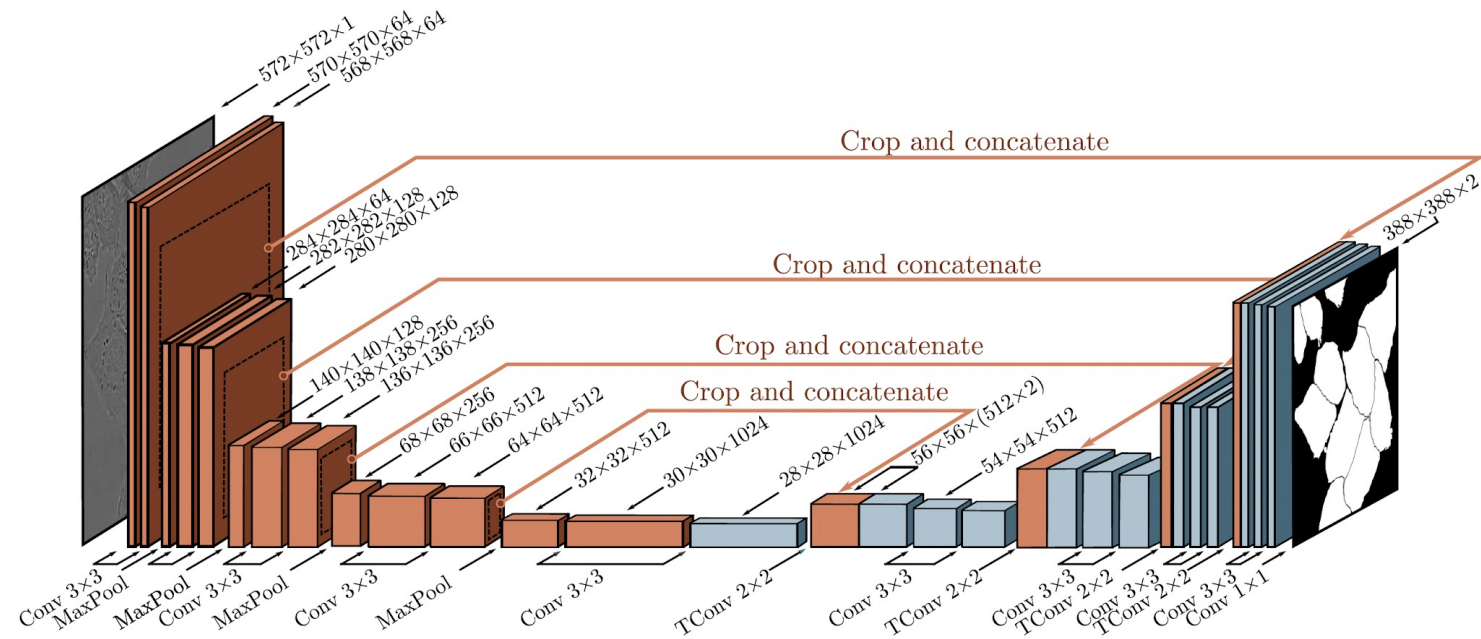


Figure 11.10 U-Net for segmenting HeLa cells. The U-Net has an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled (blue blocks). The encoder uses regular convolutions, and the decoder uses transposed convolutions. Residual connections append the last representation at each scale in the encoder to the first representation at the same scale in the decoder (orange arrows). The original U-Net used “valid” convolutions, so the size decreased slightly with each layer, even without downsampling. Hence, the representations from the encoder were cropped (dashed squares) before appending to the decoder. Adapted from Ronneberger et al. (2015).

U-Net for segmentation

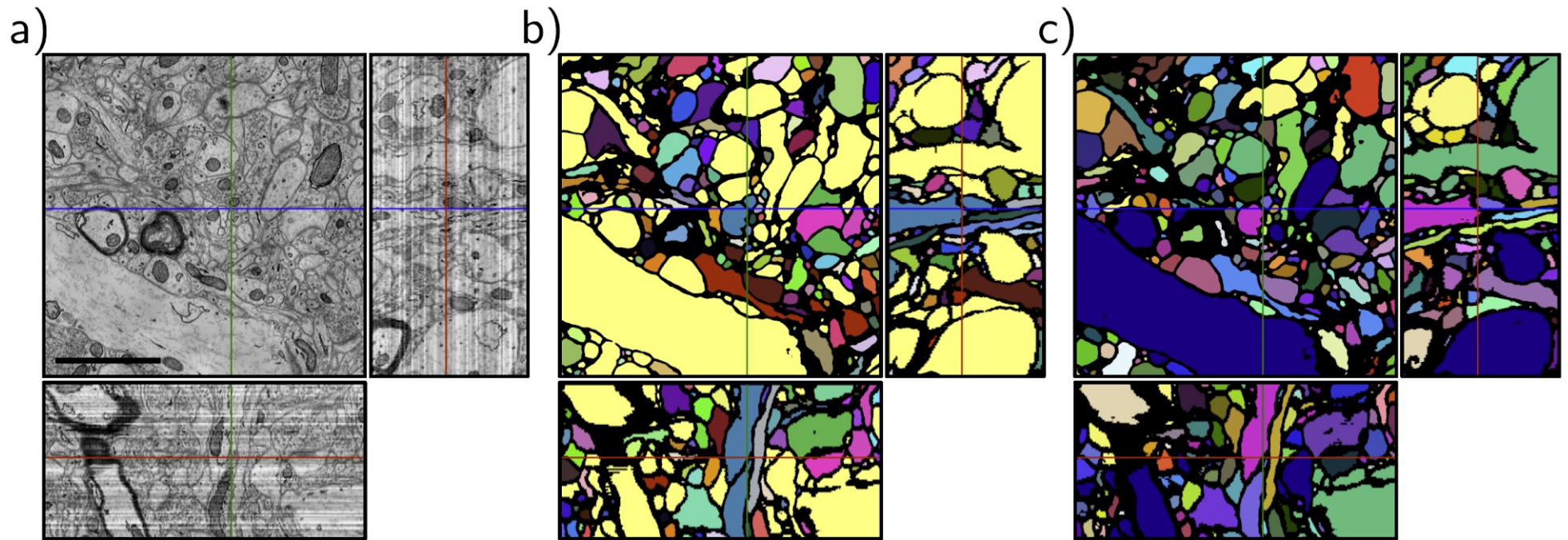


Figure 11.11 Segmentation using U-Net in 3D. a) Three slices through a 3D volume of mouse cortex taken by scanning electron microscope. b) A single U-Net is used to classify voxels as being inside or outside neurites. Connected regions are identified with different colors. c) For a better result, an ensemble of five U-Nets is trained, and a voxel is only classified as belonging to the cell if all five networks agree. Adapted from Falk et al. (2019).

U-Net for pose estimation

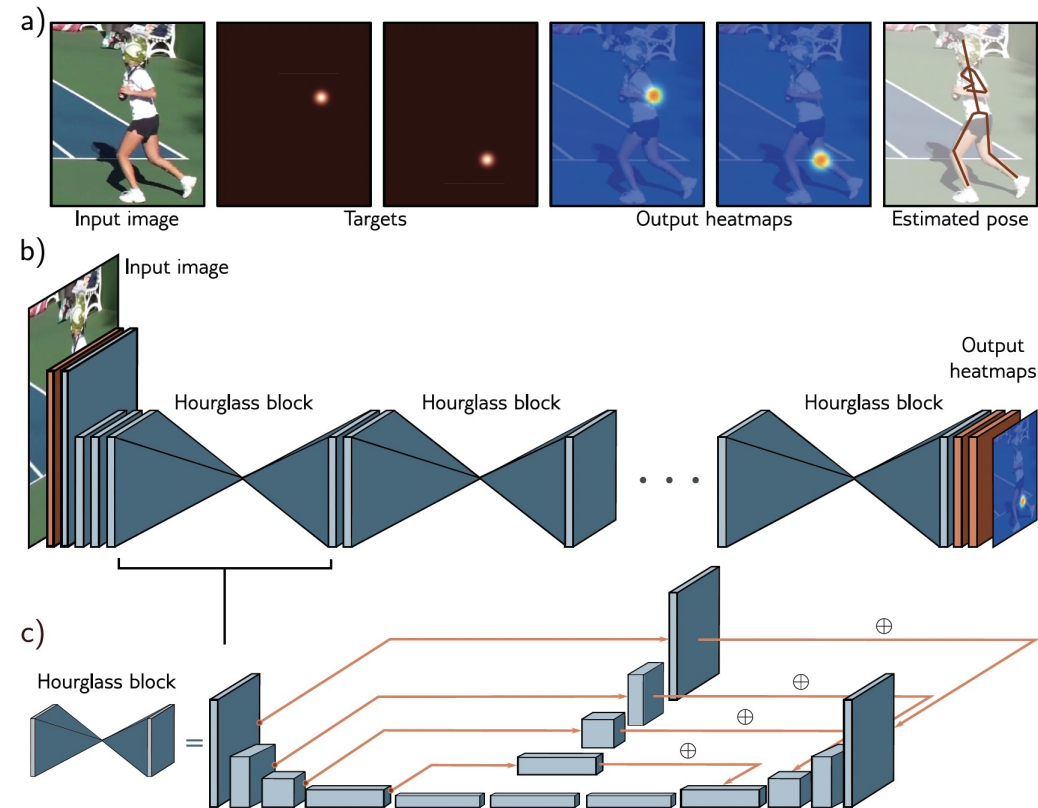


Figure 11.12 Stacked hourglass networks for pose estimation. a) The network input is an image containing a person, and the output is a set of heatmaps, with one heatmap for each joint. This is formulated as a regression problem where the targets are heatmap images with small, highlighted regions at the ground-truth joint positions. The peak of the estimated heatmap is used to establish each final joint position. b) The architecture consists of initial convolutional and residual layers followed by a series of hourglass blocks. c) Each hourglass block consists of an encoder-decoder network similar to the U-Net except that the convolutions use zero padding, some further processing is done in the residual links, and these links add this processed representation rather than concatenate it. Each blue cuboid is itself a bottleneck residual block (figure 11.7b). Adapted from Newell et al. (2016).