Deep Learning — Assignment 3

Third assignment for the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Luka Mucko, Luca Poli

Group: 46

Instructions:

- Fill in your names and the name of your group.
- Answer the questions and complete the code where necessary.
- Keep your answers brief, one or two sentences is usually enough.
- Re-run the whole notebook before you submit your work.
- Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file.
- The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Objectives

In this assignment you will

1. Experiment with convolutional neural networks
2. Train a convolutional neural network on a speech dataset
3. Investigate the effect of dropout and batch normalization
4. Define and train a residual neural network

Required software

If you haven't done so already, you will need to install the following additional libraries:

- `torch` and `torchvision` for PyTorch,
- `d2l` , the library that comes with Dive into deep learning book,
- `python_speech_features` to compute MFCC features.

All libraries can be installed with `pip install` .

```
In [1]: %matplotlib inline
        import os
        import numpy as np
        import matplotlib.pyplot as plt
        from d2l import torch as d2l
        import torch
        from torch import nn
        from scipy.io import wavfile

        # Fix the seed, so outputs are exactly reproducible
        torch.manual_seed(12345)
```

Out[1]: `<torch._C.Generator at 0x7fc19d7675b0>`

3.1 Convolution and receptive fields (9 points)

We will first define some helper functions to plot the receptive field of a node in a network.

```
In [2]: def show_image(img, title=None, new_figure=True):
            if new_figure:
                plt.figure(figsize=(5, 5))
            im = plt.imshow(img, interpolation='none', aspect='equal', cmap='gray')
            ax = plt.gca();

            # plot pixel numbers and grid lines
            ax.set_xticks(np.arange(0, img.shape[1], 1))
            ax.set_yticks(np.arange(0, img.shape[0], 1))
            ax.set_xticklabels(np.arange(0, img.shape[1], 1))
            ax.set_yticklabels(np.arange(0, img.shape[0], 1))
            ax.set_xticks(np.arange(-.5, img.shape[1], 1), minor=True)
            ax.set_yticks(np.arange(-.5, img.shape[0], 1), minor=True)
            ax.grid(which='minor', color='gray', linestyle='-', linewidth=1.5)

            # hide axis outline
            for spine in ax.spines.values():
                spine.set_visible(False)

            if title is not None:
                plt.title(title)

        # set all weights in the network to one,
        # all biases to zero
        def fill_weights_with_ones(network):
            for name, param in network.named_parameters():
                if 'weight' in name:
                    param.data = torch.ones_like(param.data)
                elif 'bias' in name:
                    param.data = torch.zeros_like(param.data)
            return network

        def compute_receptive_field(network, input_size=(15, 15), binary=True):
            assert isinstance(network, torch.nn.Sequential), 'This only works with torch.nn
            for layer in network:
```

```python
        if not isinstance(layer, (torch.nn.Conv2d, torch.nn.AvgPool2d)):
            raise Exception('Sorry, this visualisation only works for Conv2d and Av

    # initialize weights to ones, biases to zeros
    fill_weights_with_ones(network)

    # find the number of input and output channels
    input_channels = None
    output_channels = None
    for layer in network:
        if isinstance(layer, torch.nn.Conv2d):
            if input_channels is None:
                # first convolution layer
                input_channels = layer.in_channels
            output_channels = layer.out_channels
    if input_channels is None:
        input_channels = 1

    # first, we run the forward pass to compute the output shape give the input

    # PyTorch expects input shape [samples, channels, rows, columns]
    x = torch.zeros(1, input_channels, *input_size)
    x.requires_grad = True

    # forward pass: apply each layer in the network
    y = x
    y.retain_grad()
    ys = [y]
    for layer in network:
        y = layer(y)
        # keep track of the intermediate values so we can plot them later
        y.retain_grad()
        ys.append(y)

    # second, we run the backward pass to compute the receptive field

    # create gradient input: zeros everywhere, except for a single pixel
    y_grad = torch.zeros_like(y)
    # put a one somewhere in the middle of the output
    y_grad[0, 0, (y_grad.shape[2] - 1) // 2, (y_grad.shape[3] - 1) // 2] = 1

    # compute the gradients given this single one
    y.backward(y_grad)

    # receptive field is now in the gradient at each layer
    receptive_fields = []
    for y in ys:
        # the gradient for this layer shows us the receptive field
        receptive_field = y.grad
        if binary:
            receptive_field = receptive_field > 0
        receptive_fields.append(receptive_field)
    return receptive_fields

def plot_receptive_field(network, input_size=(15, 15), binary=True):
    receptive_fields = compute_receptive_field(network, input_size, binary)
```

```python
    # plot the gradient at each layer
    plt.figure(figsize=(4 * len(receptive_fields), 4))
    for idx, receptive_field in enumerate(receptive_fields):
        plt.subplot(1, len(receptive_fields), idx + 1)
        # the last element of ys contains the output of the network
        if idx == len(receptive_fields) - 1:
            plot_title = 'output (%dx%d)' % (receptive_field.shape[2], receptive_fi
        else:
            plot_title = 'layer %d input (%dx%d)' % (idx, receptive_field.shape[2],
        # plot the image with the receptive field (sample 0, channel 0)
        show_image(receptive_field[0, 0], new_figure=False, title=plot_title)
        if not binary:
            plt.colorbar(fraction=0.047 * receptive_field.shape[0] / receptive_fiel

def receptive_field_size(network, input_size=(15, 15), binary=True):
    receptive_fields = compute_receptive_field(network, input_size, binary)
    return torch.count_nonzero(torch.flatten(receptive_fields[0][0,0]))
```
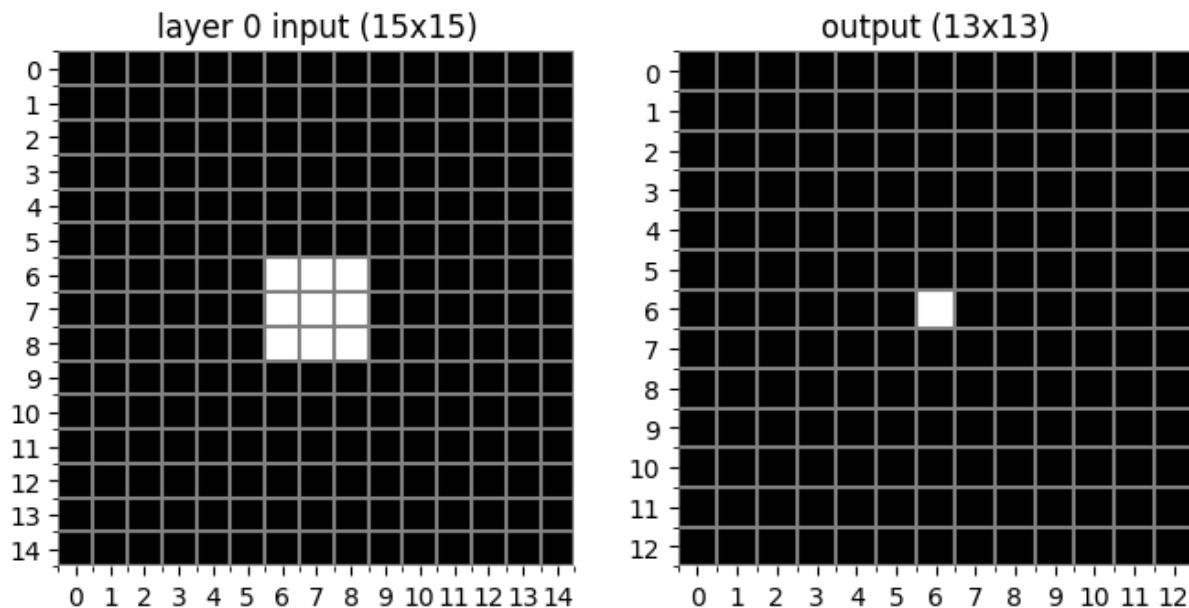
Using these functions, we can define a network and plot the receptive field of a pixel in the output.

(a) Run the code to define a network with one 3×3 convolution layer and plot the images.

In [3]:
```python
net = torch.nn.Sequential(
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3)),
)
plot_receptive_field(net, input_size=(15, 15))
```


Read these images as follows:

- On the left, you see the input size of the network (here: 15 x 15 pixels) and the receptive field for one pixel in the output.

- On the right, you see the output size of the network (here: 13 x 13 pixels).

To visualize the receptive field of this network, we used the following procedure:

- We selected one pixel of the output (shown as the white pixel in the center in the image on the right).
- We computed the gradient for this pixel and plotted the gradient with respect to the input (the image on the left).
- This shows you the receptive field of the network: the output for the pixel we selected depends on these 9 pixels in the input.

(b) Use this method to plot the receptive field of a pixel in the output of a convolution layer with a kernel size of 5×5. **(1 point)**

```
In [4]:  # TODO Plot the receptive field of a 5x5 convolution.
         net = torch.nn.Sequential(
             torch.nn.Conv2d(1, 1, kernel_size=(5, 5)),
         )
         plot_receptive_field(net, input_size=(15, 15))
```


layer 0 input (15x15)   output (11x11)
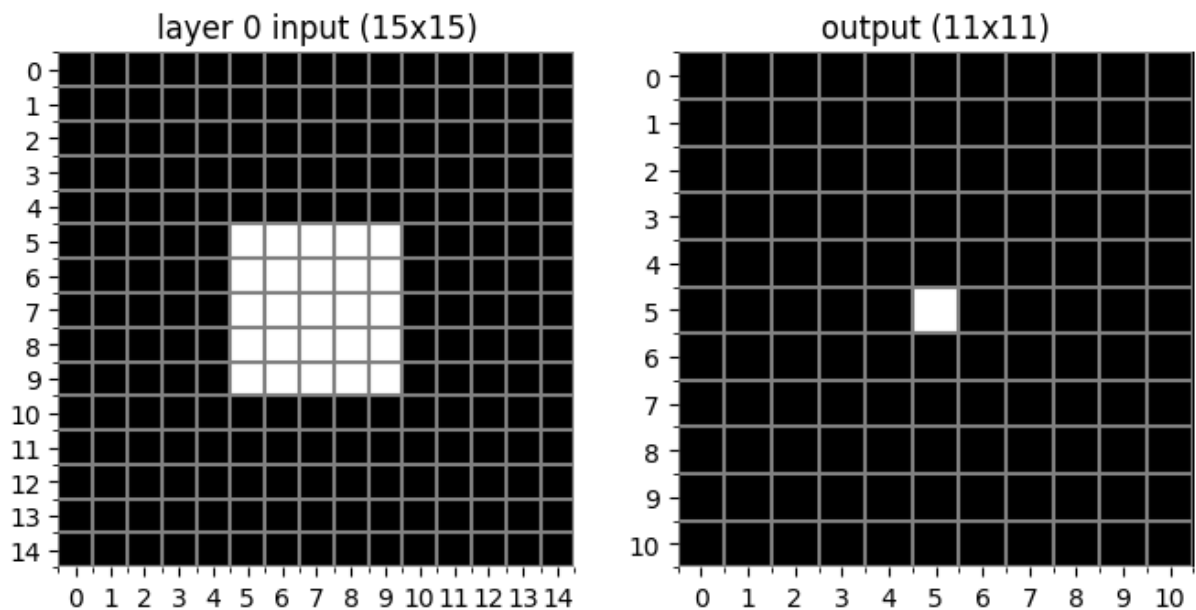
If you look at the result, you will see that two things have changed: the receptive field and the output size.

(c) How do the receptive field size and the output size depend on the kernel size? Give a formula. **(1 point)**

Receptive field size R given kernel size K and stride S
R=(K-1)S+1

Output size O given input size I, padding P and stride S:
$$O = \frac{I-K+2P}{S} + 1$$

Counting the number of parameters

In the previous question, you saw how the receptive fields of a 3x3 convolution differs from a 5x5 kernel convolution. But this is not the only difference: there is also a difference in the number of parameters in the network.

We can count the number of parameters in the network by computing the number of elements (e.g., the weights and biases in a convolution kernel) in the parameter list of the PyTorch network.

We'll define a small helper function to do this:

```
In [5]: def print_parameter_count(network):
            # sum the number of elements in each parameter of the network
            count = sum([param.data.numel() for param in network.parameters()])
            print('%d parameters' % count)
```

(d) Use the function to count the number of parameters for a 3x3 convolution.

```
In [6]: net = torch.nn.Sequential(
            torch.nn.Conv2d(1, 1, kernel_size=(3, 3)),
        )
        plot_receptive_field(net, input_size=(15, 15))
        print_parameter_count(net)
```
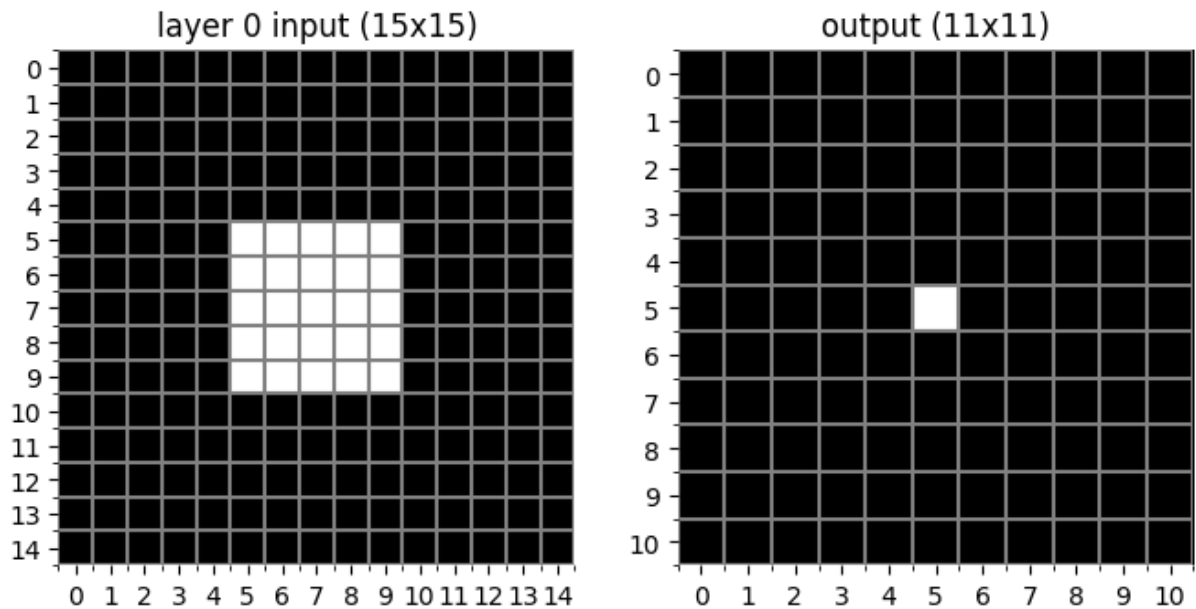
```
10 parameters
```
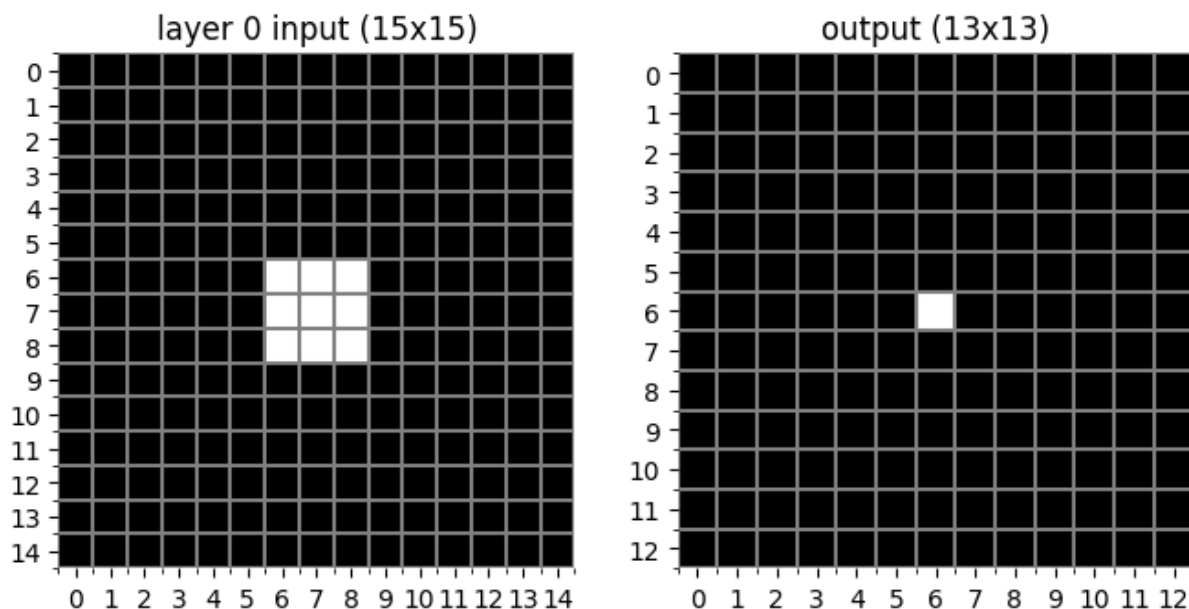

(e) Do the same to count the number of parameters for a 5x5 convolution.   (1 point)

```
In [7]: # TODO count the number of parameters of a 5x5 convolution
        net = torch.nn.Sequential(
            torch.nn.Conv2d(1, 1, kernel_size=(5, 5)),
        )
```

```
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

26 parameters

layer 0 input (15x15)

output (11x11)

(f) Explain the results by showing how to *compute* the number of parameters for the 3x3 and 5x5 convolutions. **(1 point)**

Number of parameters of convolutional layer of size K with 1 in/out channel is $k \times k + 1$

For these computations we used convolution layers with one input and one output channel.

We can also compute the results for a layer with a different number of channels.

(g) Define a network with a 5x5 convolution, 2 input channels and 3 output channels. Print the number of parameters.

In [8]:
```
# TODO count the number of parameters of a 5x5 convolution with 2 input channels an
net = torch.nn.Sequential(
    torch.nn.Conv2d(2, 3, kernel_size=(5, 5)),
)
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

153 parameters

layer 0 input (15x15) | output (11x11)
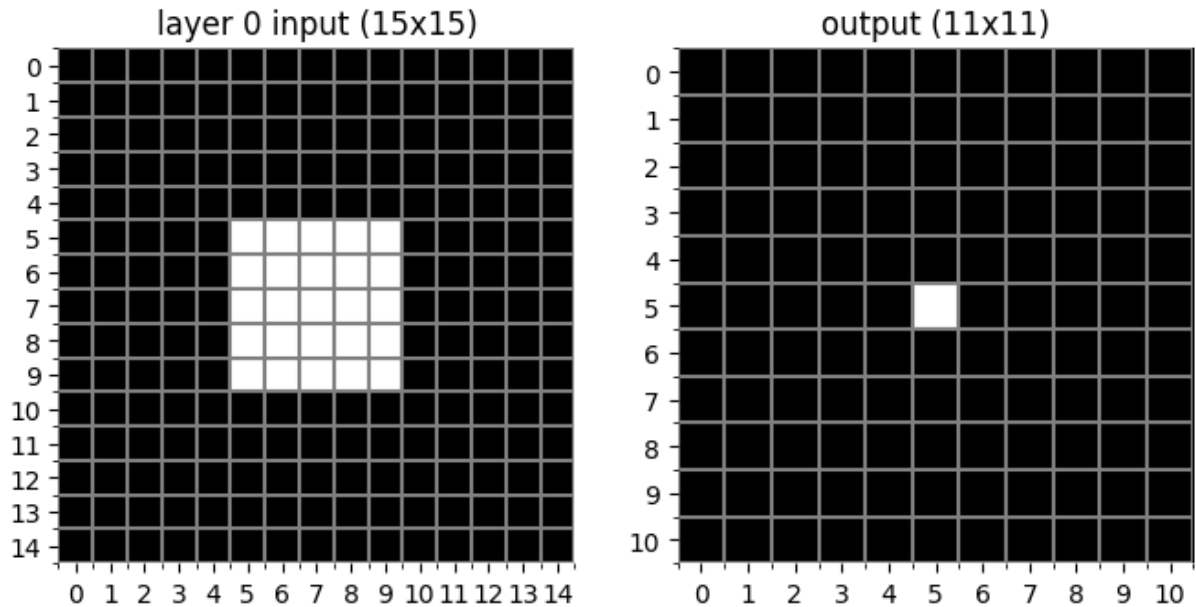
(h) Show how to compute the number of parameters for this case.   **(1 point)**

in = 2
out = 3
k = 5
n_weights = (in*out*k*k)+out*1

Preserving the size of the input image

The PyTorch documentation for `torch.nn.Conv2d` describes the parameters that you can use to define a convolutional layer. We will explore some of those parameters in the next questions.

In the previous plot, you may have noticed that the output (13x13 pixels) was slightly smaller than the input (15x15 pixels).
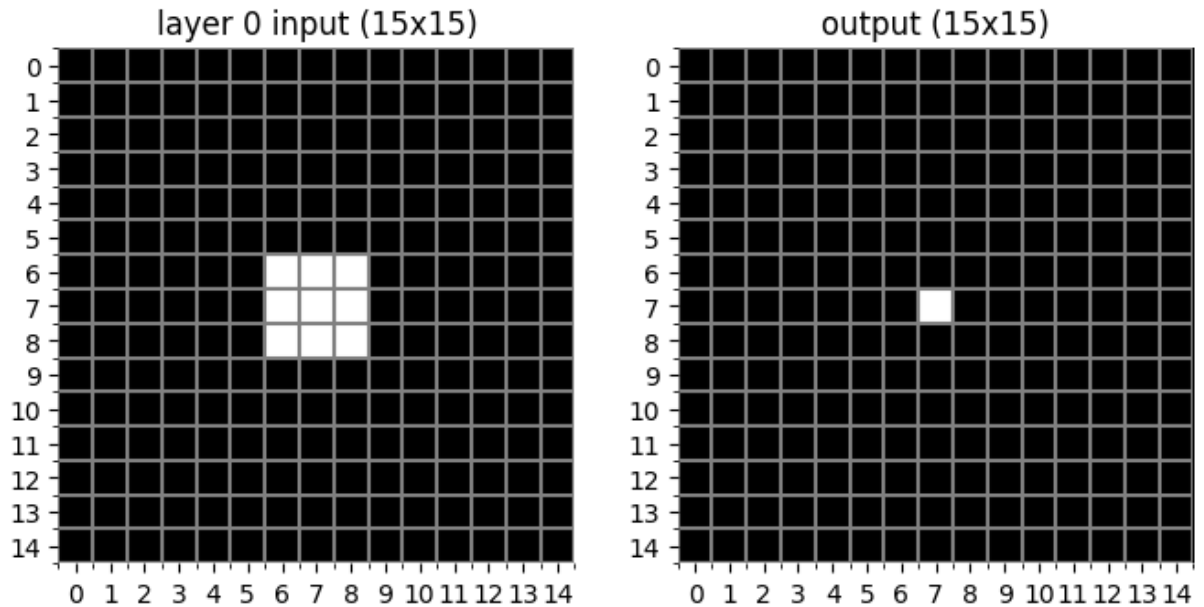
(i) Define a network with a single 3x3 convolutional layer that produces an output that has the same size as the input.   **(1 point)**

Use 1 input and 1 output channel.

```
In [9]:   # TODO Define a network with a 3×3 kernel size that takes a 15×15 input image
          #       and produces a 15×15 output image.
          net = torch.nn.Sequential(
              torch.nn.Conv2d(1, 1, kernel_size=(3, 3), stride=1, padding=1),
          )
          plot_receptive_field(net, input_size=(15, 15))
          print_parameter_count(net)

          10 parameters
```

## layer 0 input (15x15)   output (15x15)

(j) Define a network with a single 5x5 convolutional layer that preserves the input size.

(1 point)

```
In [10]:  # TODO Define a network with a 5×5 kernel size that takes a 15×15 input image
          #      and produces a 15×15 output image.
          net = torch.nn.Sequential(
              torch.nn.Conv2d(1, 1, kernel_size=(5, 5), stride=1, padding=2),
          )
          plot_receptive_field(net, input_size=(15, 15))
          print_parameter_count(net)
```

```
26 parameters
```

layer 0 input (15x15)   output (15x15)
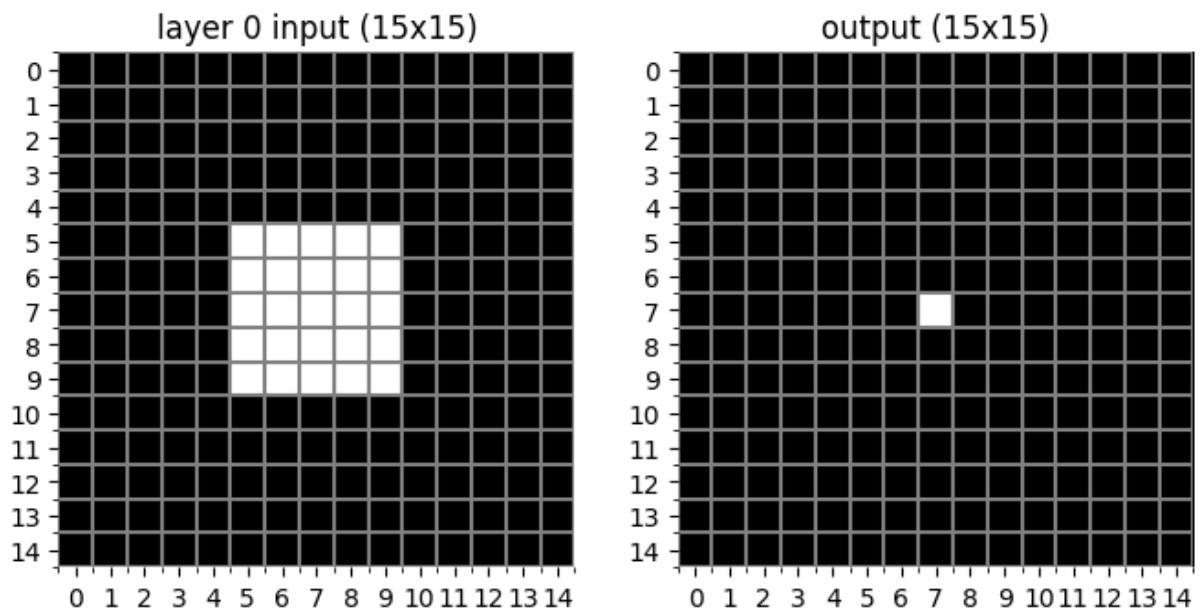
Play around with some other values to see how this parameter behaves.

Multiple layers

As you have just seen, one way to increase the size of the receptive field is to use a larger convolution kernel. But another way is to use more than one convolution layer.

(k) Define a network with two 3x3 convolutions, preserving the image size. Show the receptive field and the number of parameters.   **(1 point)**

For this visualisation, do not use any activation functions, and use 1 channel everywhere.

```python
# TODO define a network with two 3x3 convolutions
net = torch.nn.Sequential(
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3)),
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3))
)
print(net)
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

```
Sequential(
  (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1))
  (1): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1))
)
20 parameters
```
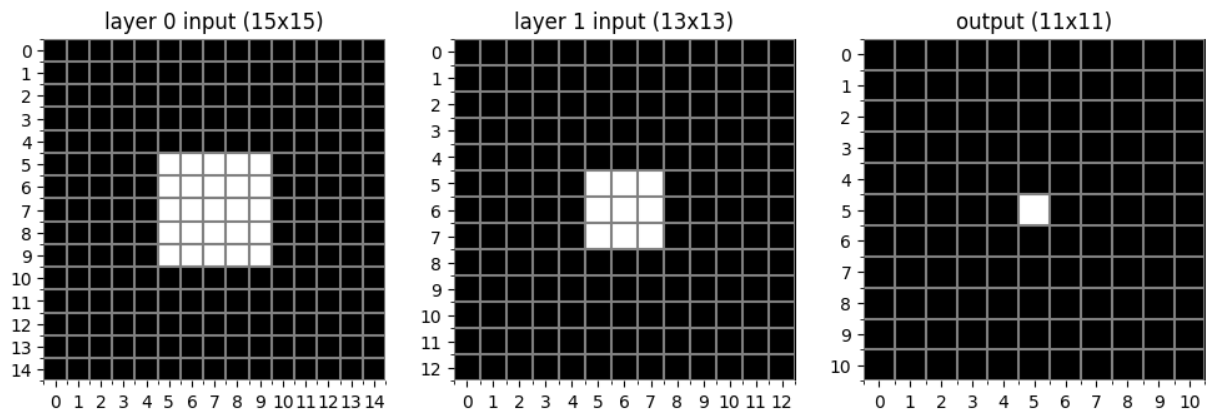

Since we now have two layers, the visualization shows an extra image. From right to left, we have:

- Right: the output size and a single active pixel.
- Middle: the receptive field for the single output pixel between the first and second convolution.
- Left: the receptive field for the single output pixel in the input image.

We have now tried two ways to increase the receptive field size: increasing the kernel size, and using multiple layers.

(l) Compare the number of parameters required by the two options. Which one is more parameter-efficient?   **(1 point)**

Double 3x3 convolution has 20 parameters while 5x5 convolution has 26 parameters so 3x3 is more parameter-efficient.

3.2 Variations on convolution (8 points)

Pooling

We can also increase the size of the receptive field by using a pooling layer.

(a) Construct a network with a 3x3 convolution (preserving the input size) followed by a 2x2 average pooling. Plot the receptive field and print the number of parameters.

(1 point)

Use 1 input and 1 output channel.

```
In [12]:  # TODO define a network with two 3x3 convolutions
          net = torch.nn.Sequential(
              torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=1),
              torch.nn.AvgPool2d(kernel_size=(2, 2))
          )
          print(net)
          plot_receptive_field(net, input_size=(15, 15))
          print_parameter_count(net)
```

```
Sequential(
  (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): AvgPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0)
)
10 parameters
```
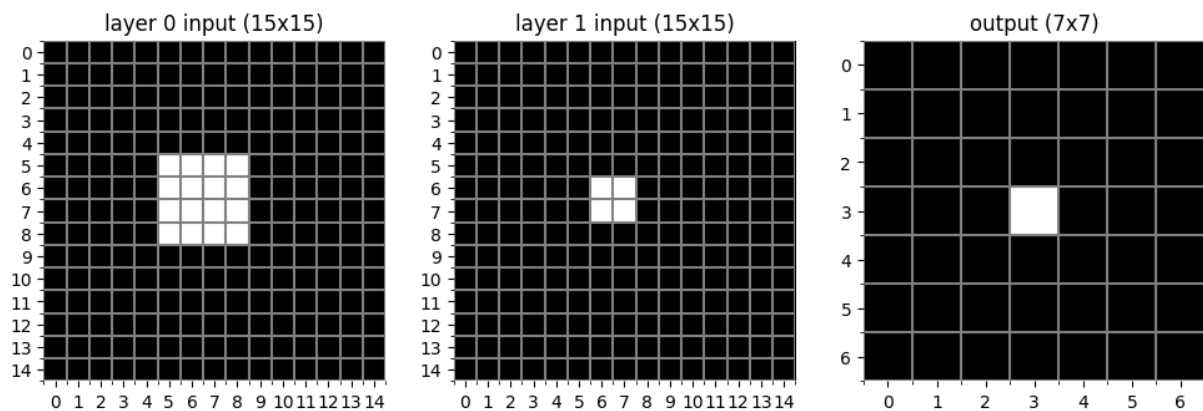

(b) Explain the number of parameters in this convolution + pooling network. (1 point)

We have 10 parameters, which came from the Conv2d layer (1x1x3x3 +1 bias). The pooling layer does not have any learnable parameters.
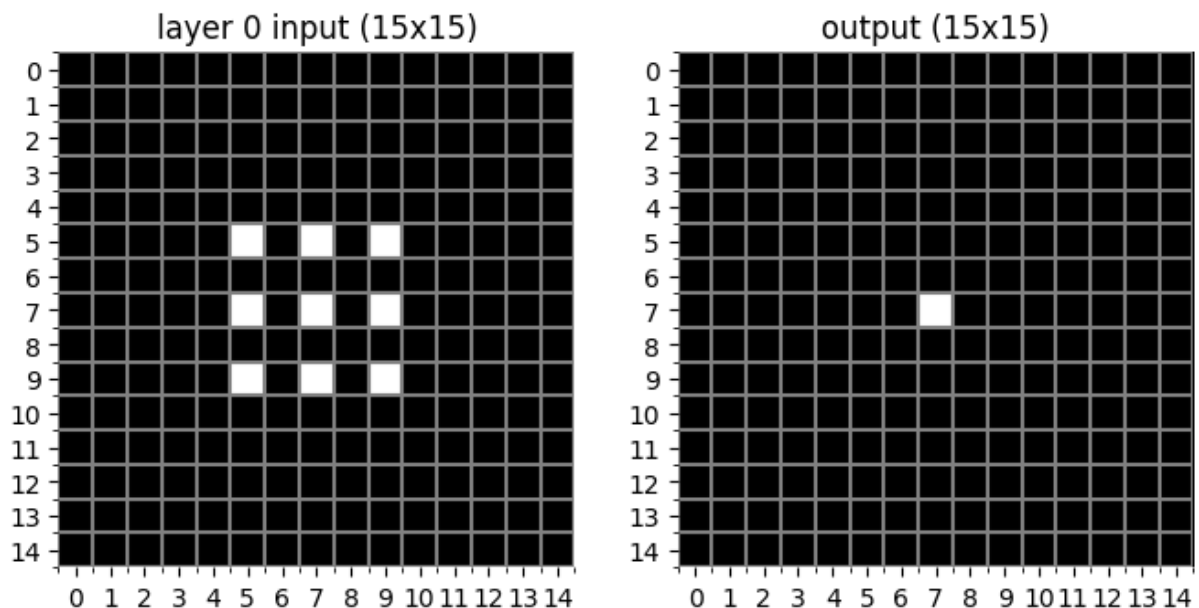
Dilation

A third option to increase the receptive field is *dilation*.

(c) Define a network with 3x3 convolution with dilation that preserves the input size.

(1 point)

```
In [13]:  # TODO define a network with one 3x3 convolution with dilation
          net = torch.nn.Sequential(
              torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=2, dilation=2),
          )
          # the output should also be 15x15 pixels
          print(net)
          plot_receptive_field(net, input_size=(15, 15))
          print_parameter_count(net)
```

```
Sequential(
  (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(1, 1), padding=(2, 2), dilation=(2,
2))
)
10 parameters
```


(d) Explain how dilation affects the receptive field.   **(1 point)**

With dilation each pixel in the receptive field is now 2 pixels apart instead of 1. So the receptive field increases without increasing the number of parameters.

Using strides

By default, convolution layers use a stride of 1.
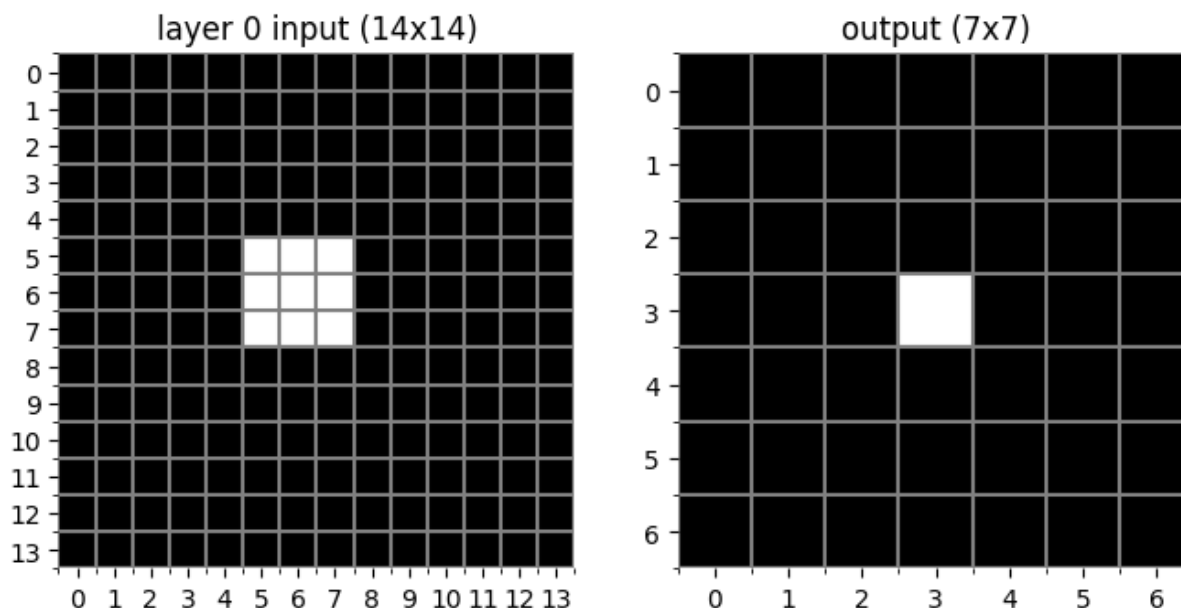
(e) Change the network to use a stride of 2 and plot the result.   **(1 point)**

```
In [14]:  # TODO increase the stride to 2
          net = torch.nn.Sequential(
              torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=(1, 1), stride=(2, 2)),
          )
          print(net)
```

```
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
```

```
Sequential(
  (0): Conv2d(1, 1, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
)
10 parameters
```


(f) Explain the new output size and compare the result with that of pooling. **(1 point)**

With stride 2 the output size is 7x7, because the stride cause the filter to skip some pixel (so the output dimension is halved). This is the same as the pooling layer, but in this case we have learnable parameters.

(g) Explain how the stride affects the receptive field of this single convolution layer.

(1 point)

The stride causes the filter to skip some pixel, the receptive field dimension is the same, but the receptive field in the output are more sparsely distributed (by a factor of 2).

(h) Explain the number of parameters for this network. **(1 point)**

The stride does not affect the number of parameters, so we still have 10 parameters. But it can effect the number of parameters in future layers (because the output size is smaller).

3.3 Combining layers (7 points)

As you have seen, there are multiple ways to increase the receptive field. You can make interesting combinations by stacking multiple layers.

Let's try a few ways to make networks with a large receptive field. For each of the questions in this section:

- Create a network where a pixel in the output has a 9x9 receptive field.
- Use 3 input channels and 3 output channels in every layer.
- In convolution layers, try to preserve the input size as much as possible.

(a) Make a network with a single convolution that satisfies the above conditions.

(1 point)

```
In [15]: # TODO
         net = torch.nn.Sequential(
             torch.nn.Conv2d(3, 3, kernel_size=(9,9), padding=4)
         )
         print(net)
         plot_receptive_field(net, input_size=(14, 14))
         print_parameter_count(net)
         assert receptive_field_size(net) == 9*9, "Receptive field of output pixel should be
```

```
Sequential(
  (0): Conv2d(3, 3, kernel_size=(9, 9), stride=(1, 1), padding=(4, 4))
)
732 parameters
```
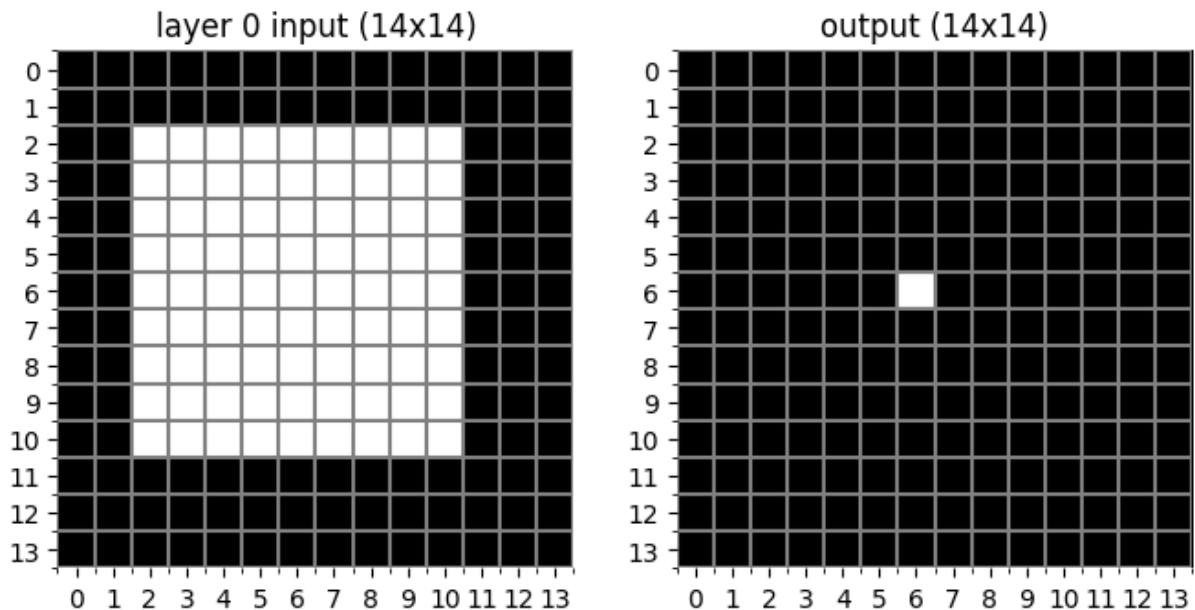

Many popular network architectures use a sequence of 3x3 convolutions.

(b) Use only 3x3 convolutions.   **(1 point)**

```
In [16]: # TODO
         net = torch.nn.Sequential(
             torch.nn.Conv2d(3, 3, kernel_size=(3,3),padding=1),
             torch.nn.Conv2d(3, 3, kernel_size=(3,3),padding=1),
             torch.nn.Conv2d(3, 3, kernel_size=(3,3),padding=1),
             torch.nn.Conv2d(3, 3, kernel_size=(3,3),padding=1)
         )
         print(net)
         plot_receptive_field(net, input_size=(14, 14))
```

```
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel should be
```

```
Sequential(
  (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (2): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
)
336 parameters
```


(c) Use a 2x2 average pooling layer in combination with one or more 3x3 convolutions. (Use any kernel size) **(1 point)**
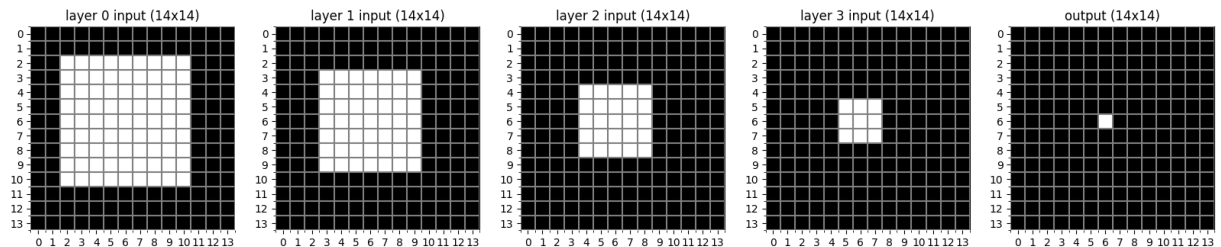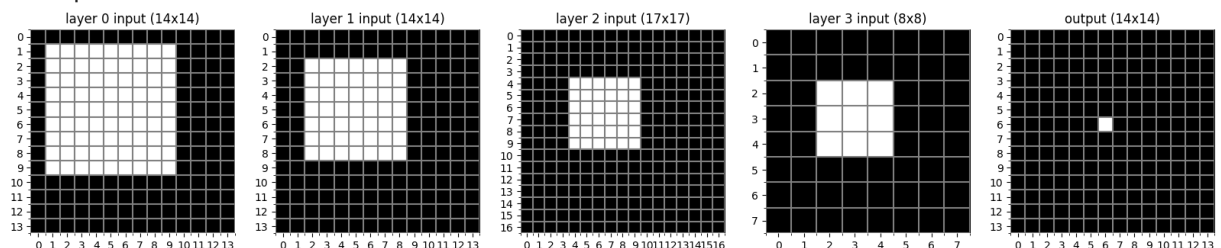
In [17]:
```python
# TODO
net = torch.nn.Sequential(
    torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=1),
    torch.nn.Conv2d(3, 3, kernel_size=(2,2), padding=2),
    torch.nn.AvgPool2d(kernel_size=(2,2)),
    torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=4),
)
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel should be
```

```
Sequential(
  (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2d(3, 3, kernel_size=(2, 2), stride=(1, 1), padding=(2, 2))
  (2): AvgPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0)
  (3): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4))
)
207 parameters
```


(d) Copy the previous convolution + pooling network and replace the pooling layer with a strided convolution layer. **(1 point)**

In [18]:
```python
# TODO
net = torch.nn.Sequential(
    torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=1),
```
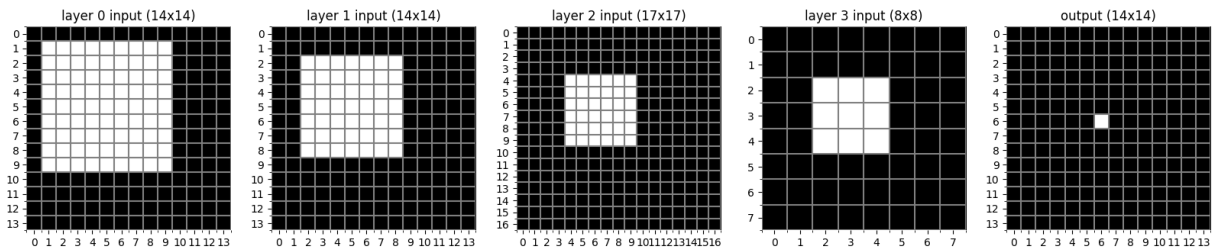
```python
        torch.nn.Conv2d(3, 3, kernel_size=(2,2), padding=2),
        torch.nn.Conv2d(3,3, kernel_size=(2,2), stride=2),
        torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=4),
    )
    print(net)
    plot_receptive_field(net, input_size=(14, 14))
    print_parameter_count(net)
    assert receptive_field_size(net) == 9*9, "Receptive field of output pixel should be
```

```
Sequential(
  (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): Conv2d(3, 3, kernel_size=(2, 2), stride=(1, 1), padding=(2, 2))
  (2): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
  (3): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(4, 4))
)
246 parameters
```


(e) Construct a network with exactly two 3x3 convolutions. Use dilation to get a receptive field of 9x9 pixels. **(1 point)**

```python
In [19]: # TODO
         net = torch.nn.Sequential(
             torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=1, dilation=3),
             torch.nn.Conv2d(3, 3, kernel_size=(3,3), padding=3)
         )
         print(net)
         plot_receptive_field(net, input_size=(14, 14))
         print_parameter_count(net)
         assert receptive_field_size(net) == 9*9, "Receptive field of output pixel should be
```

```
Sequential(
  (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), dilation=(3, 3))
  (1): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(3, 3))
)
168 parameters
```

layer 0 input (14x14)   layer 1 input (10x10)   output (14x14)
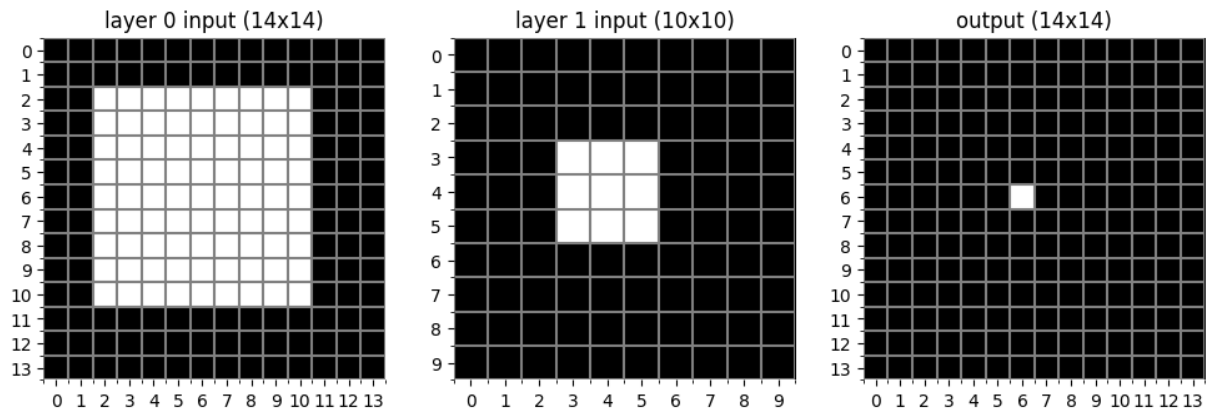
(f) For each of the methods, list the number of layers, the number of parameters, and the size of the output of the network:

Method	Layers	Parameters	Output size
One 9x9 convolution	1	732	14x14
Many 3x3 convolutions	4	336	14x14
With pooling	4	207	14x14
With strided convolution	4	246	14x14
With dilation	2	168	14x14

(g) Compare the methods in terms of the number of parameters.   (1 point)

One 9x9 convolution has the most parameters.
4 3x3 convolutions have only 336 parameters.
Pooling layer doesn't have parameters and for the sake of constraints we had to use a 2x2 convolution so the number of parameters is even lower than 4x(3x3).
We replaced the pooling operator by a 2x2 convolution which gave us a larger number of parameters than the pooling one.
Dilation doesn't add new parameters so we have just 2x(3x3) convolutions.

(h) Compare the methods in terms of the output size. How much downsampling do they do?   (1 point)

One convolutional layer of k×k reduces the 2d image by (k-1) in both dimensions.
2×2 pooling reduces the image size n to $\left\lceil \frac{n}{2} \right\rceil$
Higher stride reduces the output size (O=(I-K+2P)/S + 1)
Dilation does not affect the output size

3.4 Padding in very deep networks (2 points)

Without padding, the output of a convolution is smaller than the input. This limits the depth of your network.

(a) How often can you apply a 3x3 convolution to a 15x15 input image?

```
In [20]:  # find the maximum number of layers
          number_of_times = 25

          # create a 15x15 input
          x = torch.zeros(1, 1, 15, 15)
          print('input size: %dx%d' % (x.shape[2], x.shape[3]))

          # create a 3x3 convolution
          conv = torch.nn.Conv2d(1, 1, kernel_size=(3, 3))

          #for n in range(number_of_times):
              # apply another convolution
          #     x = conv(x)
           #    print('layer %d, output size: %dx%d' % (n + 1, x.shape[2], x.shape[3]))
```

```
input size: 15x15
```

Earlier in this assignment, you have used padding to address this problem. This seems ideal.

(b) Copy the previous code, add some padding, and show that we can now have an infinite number of layers.

(We are computer scientists and not mathematicians, so for the purpose of this question we'll consider 'infinite' to be equal to 25.)

```
In [21]:  # find the maximum number of layers
          number_of_times = 25

          # create a 15x15 input
          x = torch.zeros(1, 1, 15, 15)
          print('input size: %dx%d' % (x.shape[2], x.shape[3]))

          # create a 3x3 convolution
          conv = torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=(1, 1))

          for n in range(number_of_times):
              # apply another convolution
              x = conv(x)
              print('layer %d, output size: %dx%d' % (n + 1, x.shape[2], x.shape[3]))
```

```
input size: 15x15
layer 1, output size: 15x15
layer 2, output size: 15x15
layer 3, output size: 15x15
layer 4, output size: 15x15
layer 5, output size: 15x15
layer 6, output size: 15x15
layer 7, output size: 15x15
layer 8, output size: 15x15
layer 9, output size: 15x15
layer 10, output size: 15x15
layer 11, output size: 15x15
layer 12, output size: 15x15
layer 13, output size: 15x15
layer 14, output size: 15x15
layer 15, output size: 15x15
layer 16, output size: 15x15
layer 17, output size: 15x15
layer 18, output size: 15x15
layer 19, output size: 15x15
layer 20, output size: 15x15
layer 21, output size: 15x15
layer 22, output size: 15x15
layer 23, output size: 15x15
layer 24, output size: 15x15
layer 25, output size: 15x15
```

(c) Does it really work like this? Have a look at the following experiment.

- We simulate a convolution network with 25 convolution layers, with 3x3 kernels and the right amount of padding.
- We set the weights to 1/9 (so that the sum of the 3x3 kernel is equal to 1) and set the bias to zero.
- We give this network a 15x15-pixel input filled with ones.
- We plot the output of layers 5, 10, 15, 20, and 25.

In [22]:
```python
# create a 15x15 input filled with ones
x = torch.ones(1, 1, 15, 15)

# create a 3x3 convolution
conv = torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=(1, 1))

# set weights to 1/9 (= sum to one), bias to zero
conv.weight.data = torch.ones_like(conv.weight.data) / 9
conv.bias.data = torch.zeros_like(conv.bias.data)

plt.figure(figsize=(10, 2))
for n in range(1, 26):
    # apply another convolution
    x = conv(x)
    # print('layer %d, output size: %dx%d' % (n + 1, x.shape[2], x.shape[3]))
    if n % 5 == 0:
        plt.subplot(1, 5, n // 5)
        plt.imshow(x[0, 0].detach().numpy(), cmap='gray')
```

```
        plt.axis('off')
        plt.title('layer %d' % n)
```

| layer 5 | layer 10 | layer 15 | layer 20 | layer 25 |
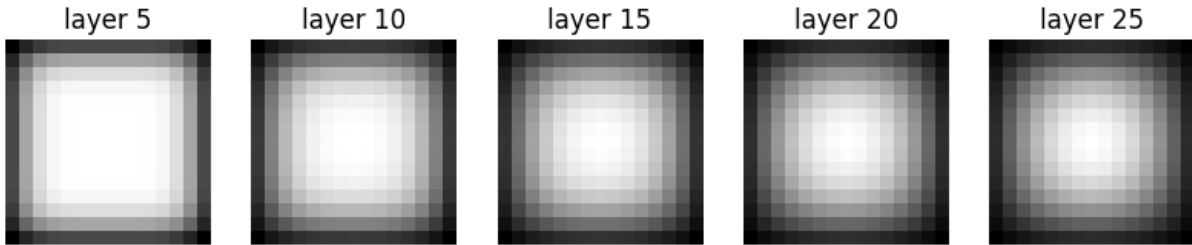
(d) Explain the pattern that we see in the output of the final layers. How does this happen, and what does this mean for our very deep networks?   **(2 points)**

In the output we can see that the image at each iteration become more blurred, and the pixels become more similar to each other. This happens because the convolution operation is working like a simple averaging filter (caused by the weight with values 1/9). So ad each layer the network spread and smooth the input image. So this repeated application of the same filter is useless, and can cause the lost of informations.

3.5 Spoken digits dataset (4 points)

Time for some practical experiments. The d2l book uses a dataset of images as a running example (FashionMNIST). In this assignment we will investigate CNNs in a completely different domain: speech recognition.

The dataset we use is the free spoken digits dataset, which can be found on https://github.com/Jakobovski/free-spoken-digit-dataset. This dataset consists of the digits 0 to 9, spoken by different speakers. The data comes as .wav files.

(a) Use the commands below (or a similar tool) to download the dataset. You can also use `git clone` to clone the repository mentioned above.

```
In [22]:   #! mkdir -p free-spoken-digit-dataset
           #! wget -O - https://github.com/Jakobovski/free-spoken-digit-dataset/archive/refs/h
```

Below is a function to load the data. We pad/truncate each sample to the same length. The raw audio is usually stored in 16 bit integers, with a range -32768 to 32767, where 0 represents no signal. Before using the data, it should be normalized. A common approach is to make sure that the data is between 0 and 1, between -1 and 1, or zero-mean unit-variance. Not all of these work well on this data, so later on, if your network doesn't seem to learn anything: try a different method to see if that works better.

(b) Update the below code to normalize the data to a reasonable range.   **(1 point)**

```
In [5]:   samplerate = 8000
          def load_waveform(file, size = 6000):
```

```
    samplerate, waveform = wavfile.read(file)
    # Take first 6000 samples from waveform. With a samplerate of 8000 that corresp
    # Pad with 0s if the file is shorter
    waveform = np.pad(waveform,(0,size))[0:size]
    # Normalize waveform
    mean = np.mean(waveform, axis=0)
    std = np.std(waveform, axis=0) + 1e-9
    waveform = (waveform - mean) / std
    return waveform
```

The following code loads all .wav files in a directory, and makes it available in a pytorch dataset.

(c) Load the data into a variable `data`.

```
In [6]: class SpokenDigits(torch.utils.data.Dataset):
            def __init__(self, data_dir):
                digits_x = []
                digits_y = []
                for file in os.listdir(data_dir):
                    if file.endswith(".wav"):
                        waveform = load_waveform(os.path.join(data_dir, file))
                        label = int(file[0])
                        digits_x.append(waveform)
                        digits_y.append(label)
                # convert to torch tensors
                self.x = torch.from_numpy(np.array(digits_x, dtype=np.float32))
                # add an extra dimension to represent the "channels" (we start with 1 chann
                self.x = self.x.unsqueeze(1)
                self.y = torch.from_numpy(np.array(digits_y))

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

            def __getitem__(self, idx):
                return self.x[idx], self.y[idx]

        data = SpokenDigits('free-spoken-digit-dataset/recordings')
```

```
In [7]: data.x
```

```
Out[7]:  tensor([[[ 6.3627e-03,  1.1801e-02,  1.2578e-02,  ...,  1.4698e-04,
                    1.4698e-04,  1.4698e-04]],

                 [[-1.8913e-01, -1.8913e-01, -4.5162e-01,  ...,  7.3366e-02,
                    7.3366e-02,  7.3366e-02]],

                 [[ 1.9949e-02,  1.8095e-03,  1.3903e-02,  ...,  1.8095e-03,
                    1.8095e-03,  1.8095e-03]],

                 ...,

                 [[-1.2523e-01,  2.6425e-01, -2.4717e-02,  ...,  4.1042e-04,
                    4.1042e-04,  4.1042e-04]],

                 [[-7.7714e-02,  2.0091e-01, -4.1508e-02,  ...,  2.0674e-04,
                    2.0674e-04,  2.0674e-04]],

                 [[-2.2171e-01,  1.4809e-01,  1.6967e-04,  ...,  1.6967e-04,
                    1.6967e-04,  1.6967e-04]]])
```

(d) Describe the dataset: how many samples are there, how many features does each sample have? How many classes are there? **(1 point)**

```
In [8]:  print("Number of samples: ", data.x.size()[0])
         print("Number of features: ", data.x.size()[2])
         print("Number of classes:", data.y.unique().size()[0])
```
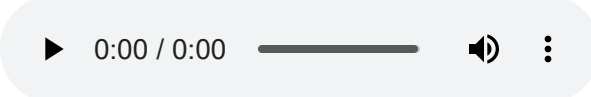
```
Number of samples:  3000
Number of features:  6000
Number of classes: 10
```

Here is code to play samples from the dataset to give you an idea what it "looks" like.

Note: If this step doesn't work in your notebook, then you can ignore it.

```
In [9]:  from IPython.display import Audio
         def play(sample):
             print(f'Label: {sample[1]}')
             return Audio(sample[0][0].numpy(), rate=samplerate)
         play(data[0])
```

```
Label: 2
```

Out[9]:

▶   0:00 / 0:00   ━━━━━━━━━━━━   🔈   ⋮

Before continuing, we split the dataset into a training and a test set.

```
In [10]:  train_prop = 2/3
          train_count = int(len(data) * train_prop)
          train, test = torch.utils.data.random_split(data, [train_count, len(data) - train_c
```

The code above uses 2/3 of the data for training.

(e) Discuss an advantage and disadvantage of using more of the data for training.

(2 points)

By using more data for training we can build a more complex model (with more parameters) and we can avoid overfitting. But we have less data for testing, so we can have a less accurate evaluation of the model.

Finally, we split the data into batches:

```
In [11]: data_params = {'batch_size': 32}
         train_iter = torch.utils.data.DataLoader(train, **data_params)
         test_iter  = torch.utils.data.DataLoader(test,  **data_params)
```

3.6 One-dimensional convolutional neural network (8 points)

We will now define a network architecture. We will use a combination of convolutional layers and pooling. Note that we use 1d convolution and pooling here, instead of the 2d operations used for images.

(a) Complete the network architecture, look at the d2l book chapter 7 and chapter 8 for examples. **(2 points)**

```
In [31]: def build_net():
             return torch.nn.Sequential(
                 nn.Conv1d(1, 4, kernel_size=5), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 # TODO: Add three more convolutional layers, ReLU layers and pooling layers
                 #       doubling the number of channels each time
                 # TODO: Your code here.
                 nn.Conv1d(4, 8, kernel_size=5), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 nn.Conv1d(8, 16, kernel_size=5), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 nn.Conv1d(16, 32, kernel_size=5), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),

                 nn.Flatten(),
                 nn.Linear(11872, 128), nn.ReLU(),
                 nn.Linear(128, 64), nn.ReLU(),
                 nn.Linear(64, 10))
```

(b) The first fully connected layer has input dimension 11872, where does that number come from? **(1 point)**

Each convolution layer with kernel size 5 reduce the input size by 4, and each avg pooling halve the input size (because of the stride 2), while the relu does not change the size. So after each step (Conv, AvgPool, ReLU) output_size = (input_size - 4)/2. We start with an input

dimension of 6000, after the convolution and pooling layers we got an output size of 371, and after the flatten layer we have 371x32=11872.

(c) How many parameters are there in the model? I.e. the total number of weights and biases. **(1 point)**

```
In [32]:   # TODO: Compute the number of parameters
           print_parameter_count(build_net())
```

1532090 parameters

(d) Suppose that instead of using convolutions, we had used only fully connected layers, while keeping the number of features on each hidden layer the same. How many parameters would be needed in that case approximately? **(1 point)**

```
In [33]:   print_parameter_count(torch.nn.Sequential(
               nn.Linear(1, 4), nn.ReLU(),
               # TODO: Add three more convolutional layers, ReLU layers and pooling layers
               #       doubling the number of channels each time
               # TODO: Your code here.
               nn.Linear(4, 8), nn.ReLU(),
               nn.Linear(8, 16), nn.ReLU(),
               nn.Linear(16, 32), nn.ReLU(),

               nn.Flatten(),
               nn.Linear(11872, 128), nn.ReLU(),
               nn.Linear(128, 64), nn.ReLU(),
               nn.Linear(64, 10)))
```

1529386 parameters

If we use only fully connected layers we will have 1529386 parameters, which is similar to the 1532090 of the convolutional network.

The FashionMNIST dataset used in the book has 60000 training examples. How large is our training set? How would the difference affect the number of epochs that we need? Compare to chapter 7.6 and chapter 8.1 of the book.

(e) How many epochs do you think are needed? **(1 point)**

```
In [4]:   lr, num_epochs = 0.1, 75 # TODO: change
```

We will use the code from (a previous edition of) the d2l book to train the network. In particular, the `train` function, defined in chapter 7.6. This function is reproduced below:

```
In [18]:   def train(net, train_iter, test_iter, num_epochs, lr, device = d2l.try_gpu()):
               """Train a model with a GPU (defined in Chapter 6)."""
               print('training on', device)
               net.to(device)
               optimizer = torch.optim.SGD(net.parameters(), lr=lr)
               loss = nn.CrossEntropyLoss()
               animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
```

```
                        legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device, torch.long)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

(f) Now train the network.

```
In [267…   train(build_net(), train_iter, test_iter, num_epochs=num_epochs, lr=lr, device="cpu
```

```
loss 0.000, train acc 1.000, test acc 0.698
2665.0 examples/sec on cpu
```
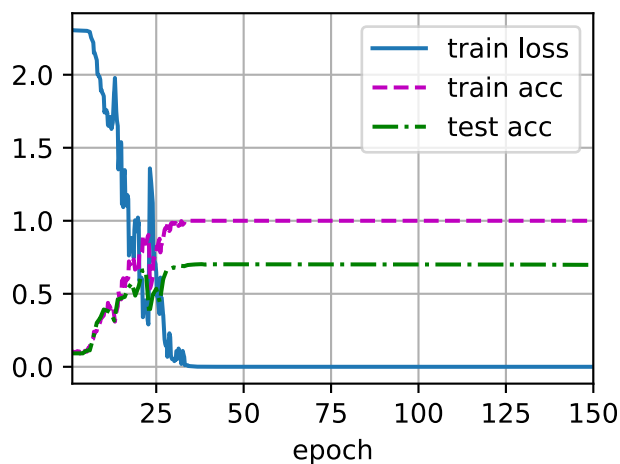

(g) Did the training converge?   **(2 point)**

If the training has not converged, maybe you need to change the number of epochs and/or the learning rate.

Hint: This is a non-trivial problem, so your network might take some time to learn. Don't give up too quickly, it might take 50-100 epochs before you see any significant changes in the loss curves.

TODO: Document the runs that you have performed and their results in the table below.

Experiment	epochs	lr	train accuracy	test accuracy	converged?
experiment 1	150	0.01	1	0.5	yes
experiment 2	150	0.15	1	0.6	yes
experiment 2	150	0.1	1	0.7	yes
experiment 2	150	0.2	1	0.598	yes

3.7 Questions and evaluation (6 points)

(a) Does the network look like it is overfitting or underfitting? Explain how see this.
(1 point)

TODO Your answer here.

(b) Is what we have here a good classifier? Could it be used in a realistic application? Motivate your answer. **(1 point)**

TODO: discuss your answer

(c) Do you think there is enough training data compared to the dimensions of the data and the number of parameters? Motivate your answer. **(1 point)**

TODO Your answer here.

(d) How could the classifier be improved? Give at least 2 suggestions. **(1 point)**

TODO Your answer here.

(e) The free spoken digits datasets has recordings from several different speakers. Is the test set accuracy a good measure of how well the trained network would perform for recognizing digits spoken by a new, unknown speaker? And if not, how could that be tested instead? **(2 points)**

TODO Your answer here.

3.8 Variations (8 points)

One way in which the training might be improved is with dropout or with batch normalization.

(a) Make a copy of the network architecture from 3.6a below, and add dropout.

(1 point)

Hint: see chapter 8.1 for an example that uses dropout.

```python
In [264...  def build_net_dropout():
                return torch.nn.Sequential(
                    nn.Conv1d(1, 4, kernel_size=5), nn.ReLU(),
                    nn.AvgPool1d(kernel_size=2, stride=2),
                    # TODO: Add three more convolutional layers, ReLU layers and pooling layers
                    #        doubling the number of channels each time
                    # TODO: Your code here.
                    nn.Conv1d(4, 8, kernel_size=5), nn.ReLU(),
                    nn.AvgPool1d(kernel_size=2, stride=2),
                    nn.Conv1d(8, 16, kernel_size=5), nn.ReLU(),
                    nn.AvgPool1d(kernel_size=2, stride=2),
                    nn.Conv1d(16, 32, kernel_size=5), nn.ReLU(),
                    nn.AvgPool1d(kernel_size=2, stride=2),

                    nn.Flatten(),
                    nn.Linear(11872, 128), nn.ReLU(), nn.Dropout(0.3),
                    nn.Linear(128, 64), nn.ReLU(), nn.Dropout(0.3),
                    nn.Linear(64, 10))

            train(build_net_dropout(), train_iter, test_iter, num_epochs=100, lr=0.075, device=
```
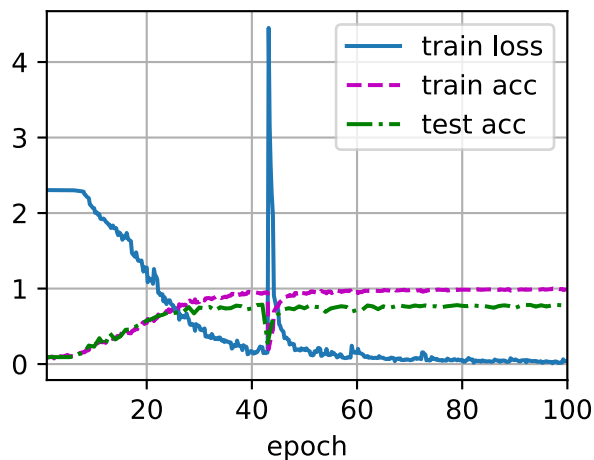
```
loss 0.047, train acc 0.988, test acc 0.770
2605.1 examples/sec on cpu
```


(b) How does dropout change the results? Does this match what you saw on the simple network last week?

(1 point)

TODO Your answer here.

(c) Make a copy of the original network architecture, and add batch normalization to all convolutional and linear layers.

(1 point)

Hint: see chapter 8.5 for an example.

```
In [170...
def build_net_batchnorm():
    return torch.nn.Sequential(
        nn.Conv1d(1, 4, kernel_size=5), nn.BatchNorm1d(4), nn.ReLU(),
        nn.AvgPool1d(kernel_size=2, stride=2),
        # TODO: Add three more convolutional layers, ReLU layers and pooling layers
        #       doubling the number of channels each time
        # TODO: Your code here.
        nn.Conv1d(4, 8, kernel_size=5), nn.BatchNorm1d(8), nn.ReLU(),
        nn.AvgPool1d(kernel_size=2, stride=2),
        nn.Conv1d(8, 16, kernel_size=5), nn.BatchNorm1d(16), nn.ReLU(),
        nn.AvgPool1d(kernel_size=2, stride=2),
        nn.Conv1d(16, 32, kernel_size=5), nn.BatchNorm1d(32), nn.ReLU(),
        nn.AvgPool1d(kernel_size=2, stride=2),

        nn.Flatten(),
        nn.Linear(11872, 128), nn.BatchNorm1d(128), nn.ReLU(),
        nn.Linear(128, 64), nn.BatchNorm1d(64), nn.ReLU(),
        nn.Linear(64, 10))

train(build_net_batchnorm(), train_iter, test_iter, num_epochs=15, lr=0.1, device="
```

```
loss 0.003, train acc 1.000, test acc 0.873
2314.3 examples/sec on cpu
```
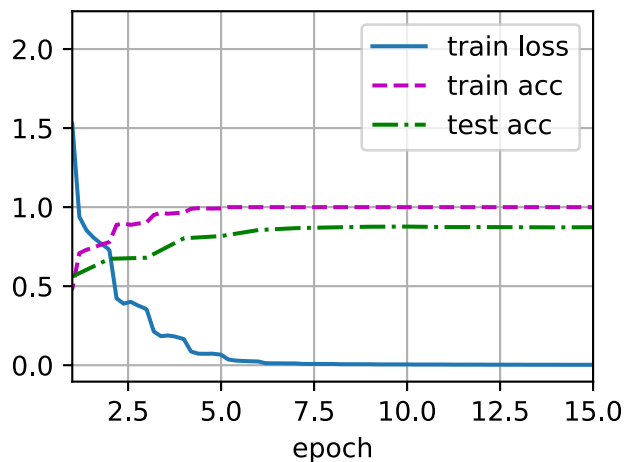

(d) How does batch normalization change the results? Does this match what you saw on the simple network last week?   **(1 point)**

TODO Your answer here.

Residual network

We can also try to use a residual network. The book has code for a 2d resnet in Chapter 8.6.

(e) Copy the `Residual` module here, and adapt it for 1d convolutions. Use a kernel size of 5 for the convolution layers.   **(2 points)**

Use residual blocks each containing two convolutional layers.

```
In [25]: class Residual(nn.Module):
             """The Residual block of ResNet models."""
             def __init__(self, in_channels, out_channels, strides=1):
                 super().__init__()
                 self.conv1 = nn.Conv1d(in_channels, out_channels, kernel_size=5, stride=str
                 self.conv2 = nn.Conv1d(out_channels, out_channels, kernel_size=5, stride=1,
                 self.bn1 = nn.BatchNorm1d(out_channels)
                 self.bn2 = nn.BatchNorm1d(out_channels)
                 self.conv_x = nn.Conv1d(in_channels, out_channels, kernel_size=5, stride=st

             def forward(self, X):
                 Y = nn.functional.relu(self.bn1(self.conv1(X)))
                 Y = self.bn2(self.conv2(Y))
                 Y += self.conv_x(X)
                 return nn.functional.relu(Y)
```

(f) Make a copy of the network architecture from 3.6a, and replace the convolutions with residual blocks.   (1 point)

```
In [26]: def build_resnet():
             return torch.nn.Sequential(
                 Residual(1, 4), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 # TODO: Add three more convolutional layers, ReLU layers and pooling layers
                 #       doubling the number of channels each time
                 # TODO: Your code here.
                 Residual(4, 8), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 Residual(8, 16), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),
                 Residual(16, 32), nn.ReLU(),
                 nn.AvgPool1d(kernel_size=2, stride=2),

                 nn.Flatten(),
                 nn.Linear(11872, 128), nn.ReLU(),
                 nn.Linear(128, 64), nn.ReLU(),
                 nn.Linear(64, 10))

         # TODO train(resnet, train_iter, test_iter, num_epochs)
         train(build_resnet(), train_iter, test_iter, num_epochs=15, lr=0.02, device="cpu")
```

```
loss 0.020, train acc 0.999, test acc 0.897
826.6 examples/sec on cpu
```
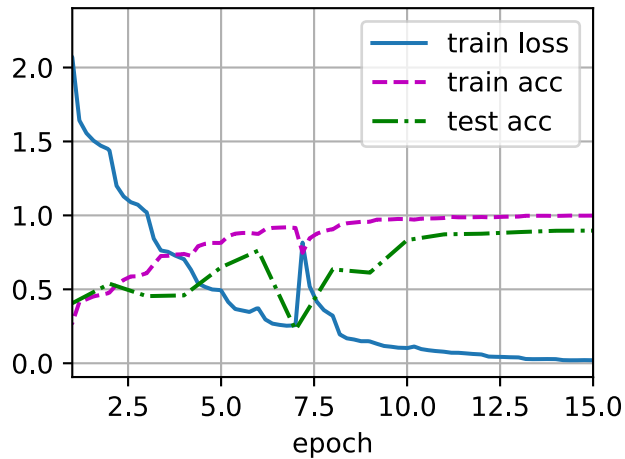
(g) How do residual connections change the results?   **(1 point)**

TODO Your answer here.

3.9 Feature extraction (5 points)

Given enough training data a deep neural network can learn to extract features from raw data like audio and images. However, in some cases it is still necessary to do manual feature extraction, in particular when working with smaller datasets like this one. For speech recognition, a popular class of features are MFCCs.

Here is code to extract these features. You will need to install the `python_speech_features` first.

```python
In [3]: from python_speech_features import mfcc

def load_waveform_mfcc(file, size = 6000):
    samplerate, waveform = wavfile.read(file)
    waveform = np.pad(waveform,(0,size))[0:size] / 32768
    return np.transpose(mfcc(waveform, samplerate))
```

(a) Implement a variation of the dataset that uses these features.   **(2 points)**

```python
In [40]: class SpokenDigitsMFCC(torch.utils.data.Dataset):
    def __init__(self, data_dir, size=6000):
        self.data_dir = data_dir
        self.size = size
        self.digits_x = []
        self.digits_y = []

        for file in os.listdir(data_dir):
            if file.endswith(".wav"):
                waveform = self.load_waveform_mfcc(os.path.join(data_dir, file))
                label = int(file[0])
                self.digits_x.append(waveform)
                self.digits_y.append(label)
```

```python
        # Convert to torch tensors
        self.x = torch.stack(self.digits_x).to(torch.float)
        self.y = torch.tensor(self.digits_y)

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

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

    def load_waveform_mfcc(self, file):
        samplerate, waveform = wavfile.read(file)
        waveform = np.pad(waveform, (0, self.size))[0:self.size] / 32768
        return torch.tensor(np.transpose(mfcc(waveform, samplerate)))


data_mfcc = SpokenDigitsMFCC("free-spoken-digit-dataset/recordings/") # TODO: your
train_count_mfcc = int(len(data_mfcc) * train_prop)
train_mfcc, test_mfcc = torch.utils.data.random_split(data_mfcc, [train_count_mfcc,
train_iter_mfcc = torch.utils.data.DataLoader(train_mfcc, **data_params)
test_iter_mfcc  = torch.utils.data.DataLoader(test_mfcc,  **data_params)

assert next(iter(train_iter_mfcc))[0].shape == torch.Size([data_params['batch_size'
```

The MFCC features will have 13 channels instead of 1 (the `unsqueeze` operation is not needed).

(b) Inspect the shape of the data, and define a new network architecture that accepts data with this shape.   **(1 point)**

```python
In [47]:  def build_net_mfcc():
              return nn.Sequential(
                  nn.Conv1d(13, 16, kernel_size=3, stride=1, padding=1), nn.ReLU(),
                  nn.AvgPool1d(kernel_size=2, stride=2),
                  nn.Conv1d(16, 32, kernel_size=3, stride=1, padding=1), nn.ReLU(),
                  nn.AvgPool1d(kernel_size=2, stride=2),
                  nn.Conv1d(32, 64, kernel_size=3, stride=1, padding=1), nn.ReLU(),
                  nn.AvgPool1d(kernel_size=2, stride=2),
                  nn.Flatten(),
                  nn.Linear(576, 32), nn.ReLU(),
                  nn.Linear(32, 10)  # Output has 10 units (assuming 10 classes for digit rec
              )
```
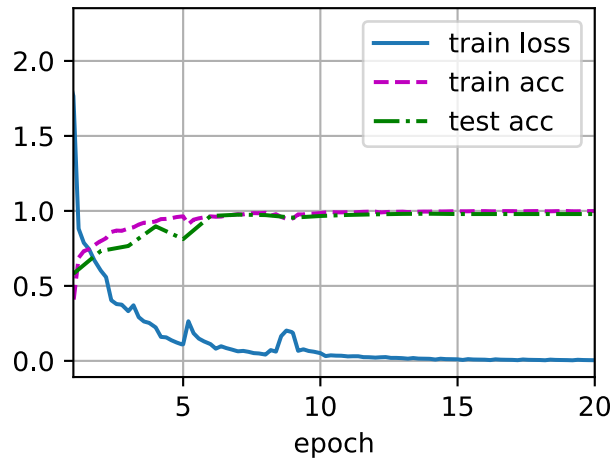
(c) Train the network with the MFCC features.   **(1 point)**

```python
In [50]:  # TODO: Your code here.
          net = build_net_mfcc().to("cpu")
          train(net, train_iter_mfcc, test_iter_mfcc, num_epochs=20, lr=0.05, device="cpu")
```

```
loss 0.004, train acc 0.999, test acc 0.979
13893.8 examples/sec on cpu
```

(d) What would be needed to get a fully neural network approach to work as well as MFCC features? **(1 point)**

TODO Your answer here.

The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 57 points.