1_CNN

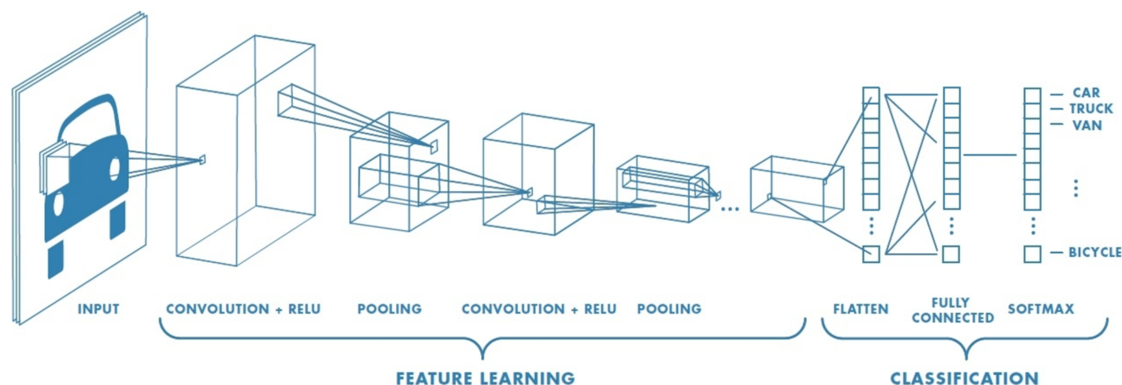
May 28, 2024

1 Convolutional Neural Networks

As the last topic of the neural network part of our course, we would like to have a look at convolutional neural networks (CNN). CNNs are very similar to ordinary Neural Networks — they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. CNNs are frequently employed for visual recognition. I.e. we use them in the Molecular Nanophotonics Group for the real-time detection of single particles.

CNNs are biologically-inspired models inspired by research of D. H. Hubel and T. N. Wiesel. They proposed an explanation for the way in which mammals visually perceive the world around them using a layered architecture of neurons in the brain, and this in turn inspired engineers to attempt to develop similar pattern recognition mechanisms in computer vision. In their hypothesis, within the visual cortex, complex functional responses generated by "complex cells" are constructed from more simplistic responses from "simple cells'. For instances, simple cells would respond to oriented edges etc, while complex cells will also respond to oriented edges but with a degree of spatial invariances. You may find the following youtube video interesting with that respect.

1.1 Layout of a CNN



"A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function." What it actually means is that, each layer is associated with converting the information from the values, available in the previous layers, into some more complex information and pass on to the next layers for further generalization.

1. **Convolution Block:** Consists of the Convolution Layer and the Pooling Layer. This layer forms the essential component of Feature-Extraction

2. **Fully Connected Block** Consists of a fully connected simple neural network architecture. This layer performs the task of Classification based on the input from the convolutional block.

1.2 Convolutional Layer

The convolutional layer is related to the extraction of specific features from an image for example. For this pupose it applies a filter with a specific mathematical procedure to the image. This filter is known from image processing to have a filter kernel.

The image can be represented as a simple matrix with data entries. In general there might be three layers for the colors red green and blue but we will assume just a single layer here. In the same way, the kernel for the convolution is a simple image, just smaller. Often kernel sizes of 3x3 matrix entries are used. The image below shows two examples.

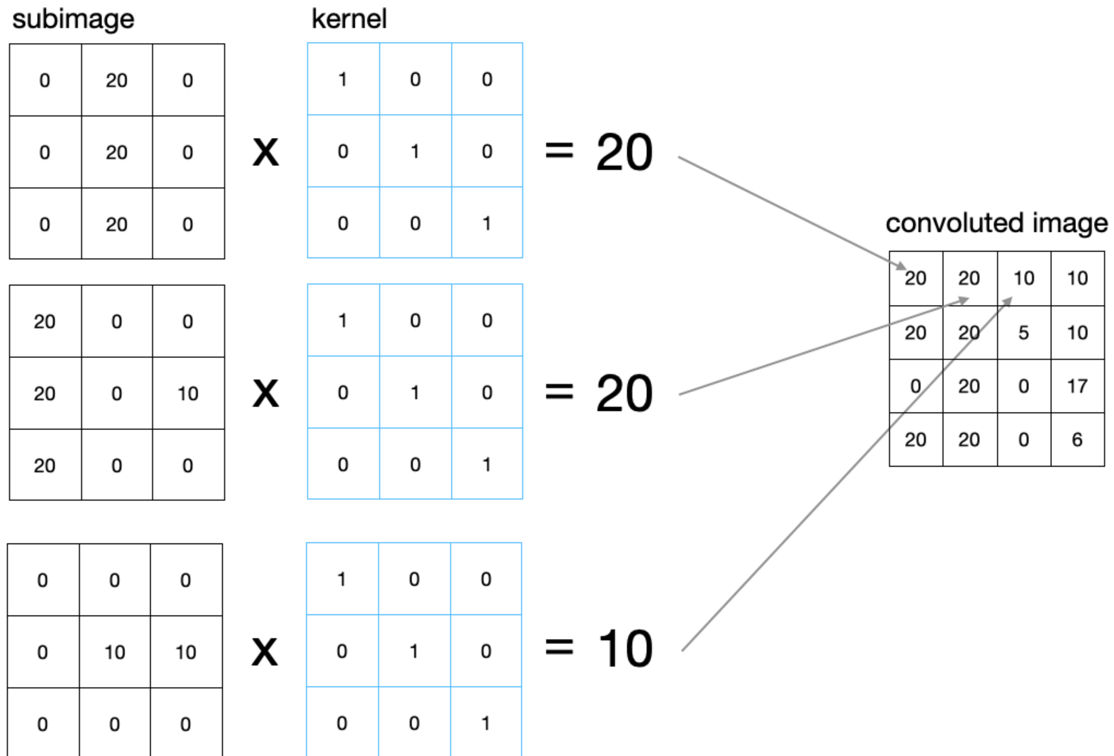
image

0	20	0	0	0	0
0	20	0	10	10	10
0	20	0	0	0	0
0	20	0	0	5	0
0	20	0	0	0	12
0	20	0	0	0	6

kernel

1	0	0
0	1	0
0	0	1

If the image is now convoluted with the kernel, a subimage of the same size as the kernel is selected from the image and multiplied elementwise with the kernel. The resulting matrix entries are then summed up to yield a single number, which corresponds to a single pixel in the corresponding convoluted image. The example above results in a 4x4 matrix, as there are just 4 different possible positions of the kernel along the 6x6 image.



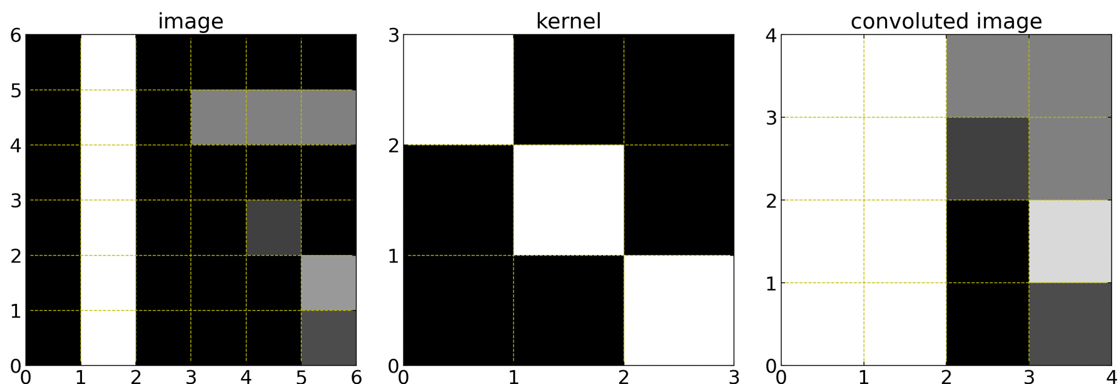
The above example shows, that the maximum value for the convolution result would be given if the original pixel numbers would be in a diagonal. The specific kernel, we have chosen is enhancing diagonal features in the image. Correspondingly other features would be enhanced by other kernels. Many CNNs apply in one step multiple kernels to the image. The results of these operation are called feature maps. The code below demonstrates the convolution with the `scipy` module.

Image array

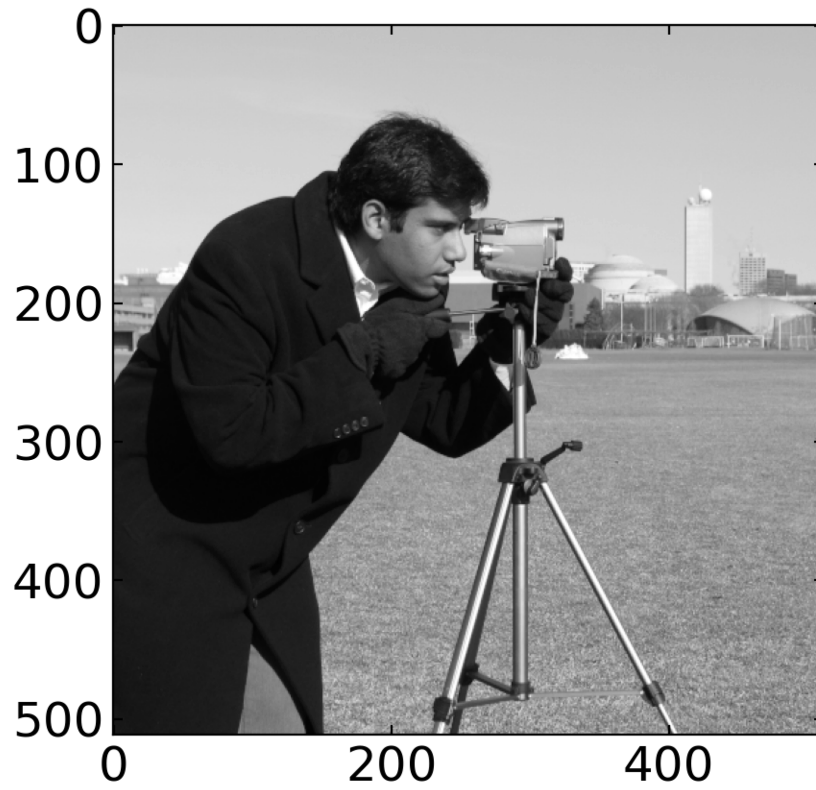
Convolution kernel

Convolution

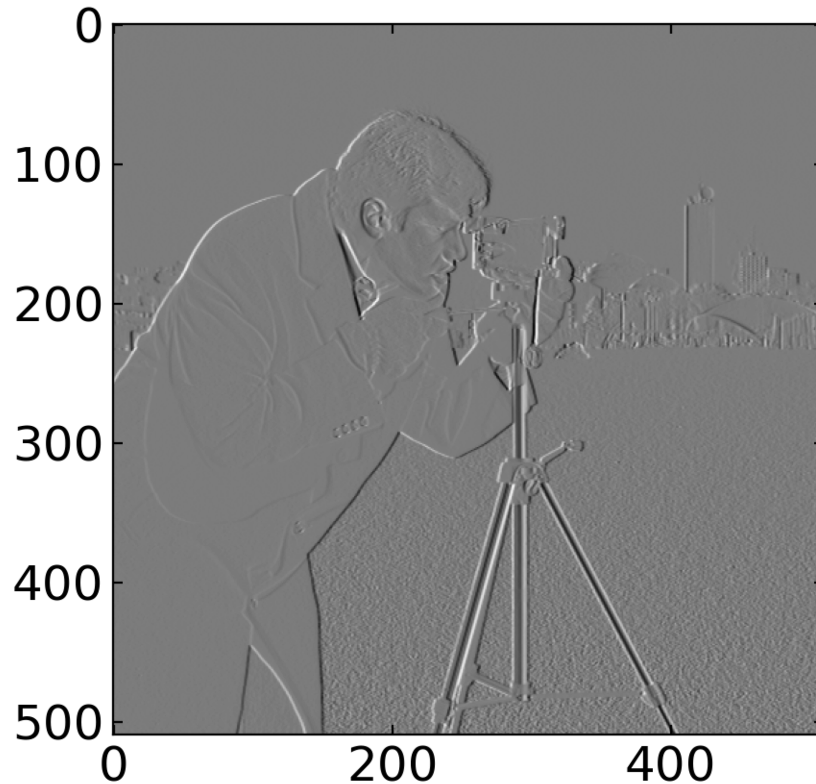
Plotting



<matplotlib.image.AxesImage at 0x7f9224975e50>



<matplotlib.image.AxesImage at 0x7f92247c9490>



1.2.1 Padding

If we try to visualize the operation of convolution, in our head, as the filter matrix moves over the whole image, we find that the no of times, the values of the cells lying within the matrix is considered for the operation is more than the no. of times, the values of the cells in the corners or at the borders, are accounted for. This implies that the values at the corners or around the borders are not being given equal weightage. To overcome this, we add another row and column, of only 0, at all the sides of the image matrix. This idea is known as padding. In actual sense, these values being '0' wouldn't supply any extra information, but will help into accounting the previously less-accounted for values to be given more weightage. As a result of the padding, the output matrix is now of the same size as the input matrix.

1.2.2 Striding

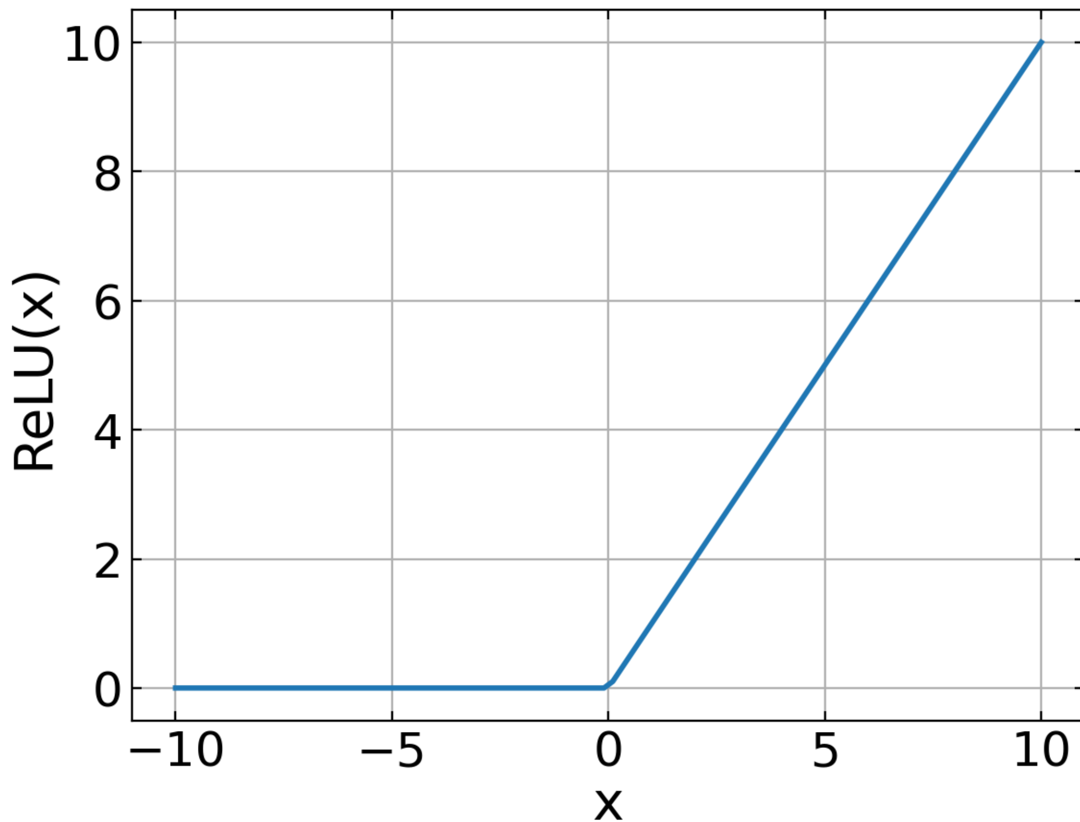
In 'strided' convolution, instead of shifting the filter one-row or one-column at a time, we shift it, maybe, 2 or 3 rows or columns, each time. This is generally done to reduce the no of calculation and also reduce the size of the output matrix. For large image, this doesn't results in loss of data, but reduces computation cost on a large scale.

1.3 RELU Activation

The resulting feature maps are finally passed through and activation function as we identified before. ReLU or Rectified Linear Unit is applied on all the cells of all the output-matrix. The function is defined as:

$$f(x) = \begin{cases} 0 \text{ for x} < 0 \\ x \text{ for x} \geq 0 \end{cases}$$

The shape of the ReLU function is shown below. It supresses essentially all output values which are negative.

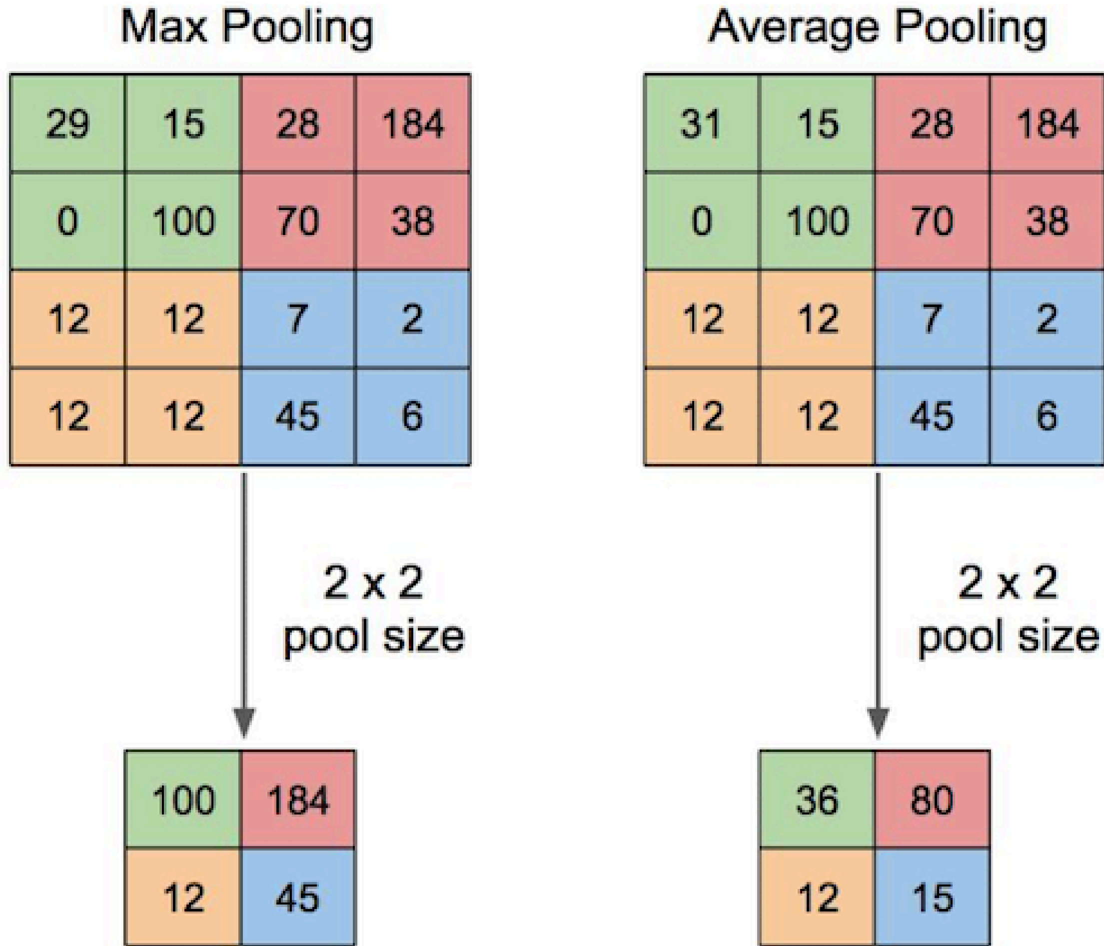


The basic intuition to derive from here is that, after convolution, if a particular convolution function results in ‘0’ or a negative value, it implies that the feature is not present there and we denote it by ‘0’, and for all the other cases we keep the value. Together with all the operations and the functions applied on the input image, we form the first part of the Convolutional Block.

1.4 Pooling Layer

The Pooling layer consist of performing the process of extracting a particular value from a set of values, usually the max value or the average value of all the values. This reduces the size of the output matrix. For example, for MAX-POOLING, we take in the max value among all the values

of say a 2 X 2 part of the matrix. Thus, we are actually taking in the values denoting the presence of a feature in that section of the image. In this way we are getting rid of unwanted information regarding the presence of a feature in a particular portion of the image and considering only what is required to know. It is common to periodically insert a Pooling layer in-between successive convolutional blocks in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network.



1.5 Output Size

All of the given operations are tentatively changing the input matrix size into an output matrix size.

$$n_{\text{out}} = \left[\frac{n_{\text{in}} + 2p - k}{s}\right] + 1$$

where

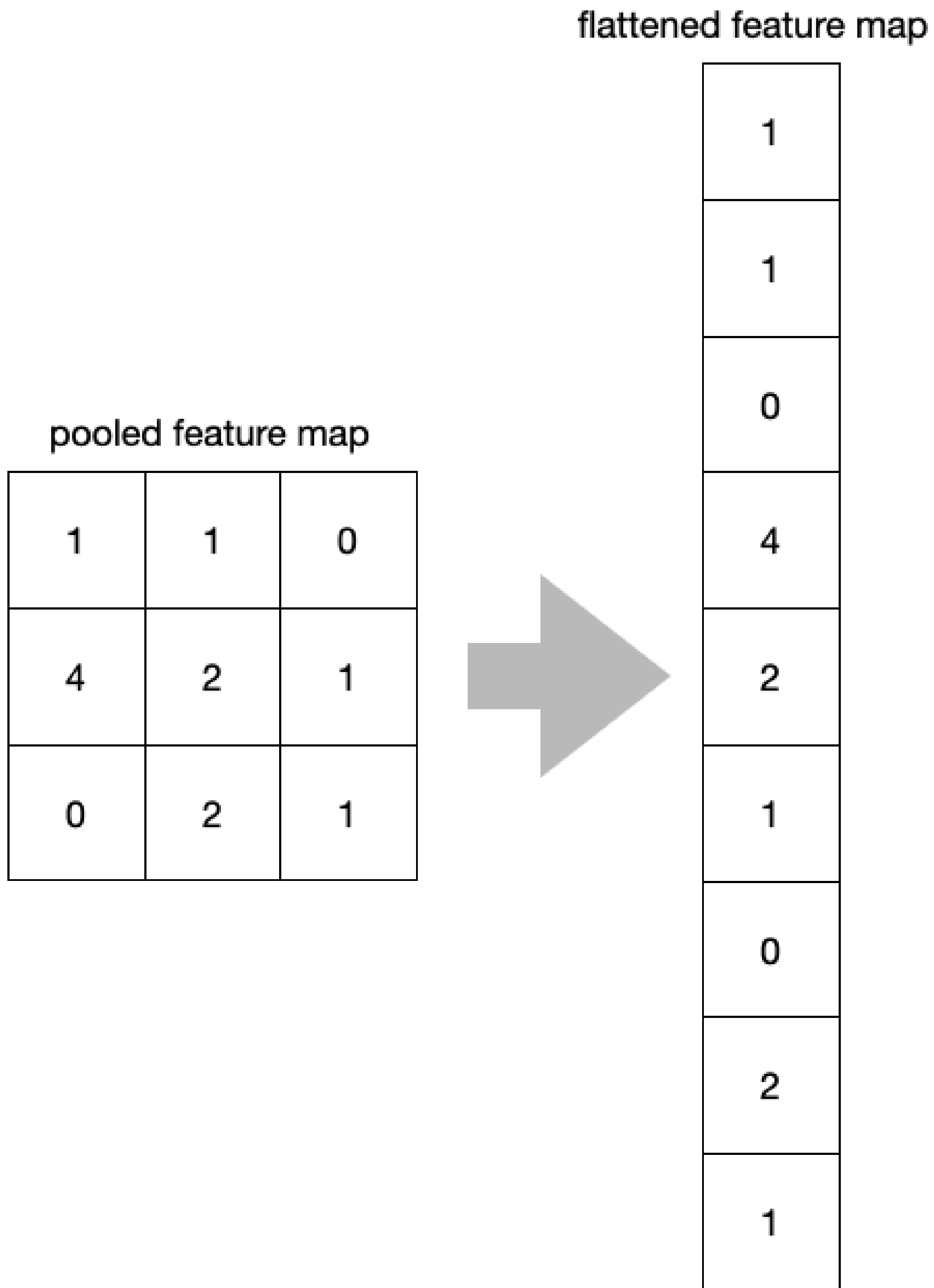
- n_{in}: number of input features
- n_{out}: number of output
- k: convolution kernel size

- p: padding size
- s: convolution stride size

For our above example $n_{\text{in}} = 6$, $k = 3$, $p = 0$ and $s = 1$ from which $n_{\text{out}} = 4$ follows.

1.6 Flattening

After multiple convolution layers and downsampling operations, the 3D representation of the image is converted into a feature vector that is passed into a multi-layer perceptron to output probabilities. The following image describes the flattening operation



1.7 Dropout

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate)

such that the sum over all inputs is unchanged.

1.8 Fully Connected Layer

This layer forms the last block of the CNN architecture, related to the task of classification. This is essentially a fully connected Simple Neural Network as we constructed it in the lecture before. It is consisting of two or three hidden layers and an output layer generally implemented using 'Softmax Regression', that performs the work of classification among a large no of categories. The structure of the output layer therefore depends on what the output should be.

2 Example CNN with Keras

We will use the knowledge gained above to create a convolutional neural network for the character recognition we started already in the last lecture with conventional neural networks. We refer to the same MNIST example as before. As the convolutional networks are more complex, we will use tensorflow and keras as the frontend to program that. Using this, the implementation is straight forward.

2.1 Prepare the data

First we have to load the data and devide that into training and testing data.

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

2.2 Build the network

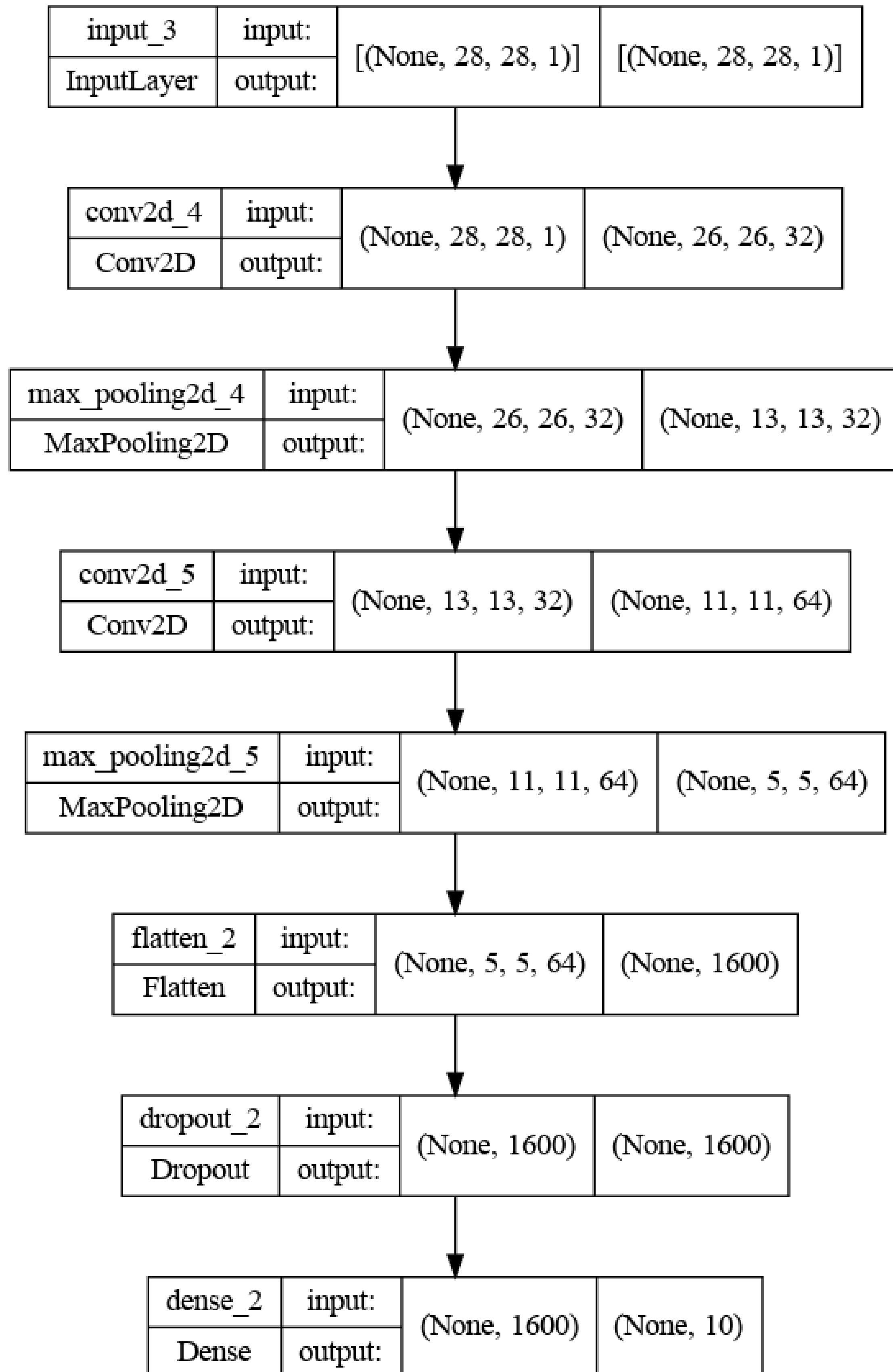
Our network shall consist of a

- input layer with 28 x 28 pixels
- convolutional layer with 32 kernels of 3 x 3 pixels and a ReLU activation
- pooling layer taking the maximum of 2 x 2 pixels
- convolutional layer with 64 kernels of 3 x 3 pixels and a ReLU activation
- pooling layer taking the maximum of 2 x 2 pixels
- flattening layer
- dropout layer which choses randomly 50% of the input
- dense output layer with the number of classes, which is 10

This network is easily setup in kears by the following commands

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_4 (Conv2D)           (None, 26, 26, 32)        320       
                                                                 
 max_pooling2d_4 (MaxPooling  (None, 13, 13, 32)       0         
 2D)                                                             
```

```
 conv2d_5 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_5 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten_2 (Flatten)         (None, 1600)              0

 dropout_2 (Dropout)         (None, 1600)              0

 dense_2 (Dense)             (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
```


input_3
InputLayer
input:
output:
[(None, 28, 28, 1)]
[(None, 28, 28, 1)]
conv2d_4
Conv2D
input:
output:
(None, 28, 28, 1)
(None, 26, 26, 32)
max_pooling2d_4
MaxPooling2D
input:
output:
(None, 26, 26, 32)
(None, 13, 13, 32)
conv2d_5
Conv2D
input:
output:
(None, 13, 13, 32)
(None, 11, 11, 64)
max_pooling2d_5
MaxPooling2D
input:
output:
(None, 11, 11, 64)
(None, 5, 5, 64)
flatten_2
Flatten
input:
output:
(None, 5, 5, 64)
(None, 1600)
dropout_2
Dropout
input:
output:
(None, 1600)
(None, 1600)
dense_2
Dense
input:
output:
(None, 1600)
(None, 10)

2.3 Train the network

Next we want to train the network. To do so, we first need to compile the model with a specific loss function, which is the `categorical_crossentropy` method, which we mentioned already last lecture. The compiled model can then be trained for a specific amount of epochs. It is important to have a part of the training data splitted apart to use that as the validation data. This validation data will also be used to calculated the loss but the loss is not minimized on that data but rather on the training data. What could happen during training is now that the training loss decreases but the validation loss increases. This in general means that the network is **overfitted** to the training data. It just reflects the training data but is not able to infer any other information accurately from it.

```
Epoch 1/15
```

```
2023-07-11 14:36:16.805623: W
tensorflow/core/framework/cpu_allocator_impl.cc:82] Allocation of 169344000
exceeds 10% of free system memory.
```

```
422/422 [==============================] - 7s 16ms/step - loss: 0.3624 -
accuracy: 0.8897 - val_loss: 0.0824 - val_accuracy: 0.9793
Epoch 2/15
422/422 [==============================] - 7s 16ms/step - loss: 0.1122 -
accuracy: 0.9652 - val_loss: 0.0614 - val_accuracy: 0.9832
Epoch 3/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0856 -
accuracy: 0.9741 - val_loss: 0.0444 - val_accuracy: 0.9893
Epoch 4/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0699 -
accuracy: 0.9786 - val_loss: 0.0457 - val_accuracy: 0.9873
Epoch 5/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0620 -
accuracy: 0.9808 - val_loss: 0.0425 - val_accuracy: 0.9882
Epoch 6/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0559 -
accuracy: 0.9829 - val_loss: 0.0373 - val_accuracy: 0.9913
Epoch 7/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0514 -
accuracy: 0.9834 - val_loss: 0.0364 - val_accuracy: 0.9908
Epoch 8/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0468 -
accuracy: 0.9847 - val_loss: 0.0294 - val_accuracy: 0.9918
Epoch 9/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0454 -
accuracy: 0.9856 - val_loss: 0.0311 - val_accuracy: 0.9903
Epoch 10/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0415 -
```

```
accuracy: 0.9872 - val_loss: 0.0325 - val_accuracy: 0.9907
Epoch 11/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0412 -
accuracy: 0.9867 - val_loss: 0.0303 - val_accuracy: 0.9918
Epoch 12/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0353 -
accuracy: 0.9889 - val_loss: 0.0301 - val_accuracy: 0.9912
Epoch 13/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0372 -
accuracy: 0.9877 - val_loss: 0.0307 - val_accuracy: 0.9918
Epoch 14/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0329 -
accuracy: 0.9897 - val_loss: 0.0318 - val_accuracy: 0.9910
Epoch 15/15
422/422 [==============================] - 7s 17ms/step - loss: 0.0326 -
accuracy: 0.9896 - val_loss: 0.0316 - val_accuracy: 0.9920
```

```
<keras.callbacks.History at 0x7f9224764040>
```

2.4 Evaluate the trained network

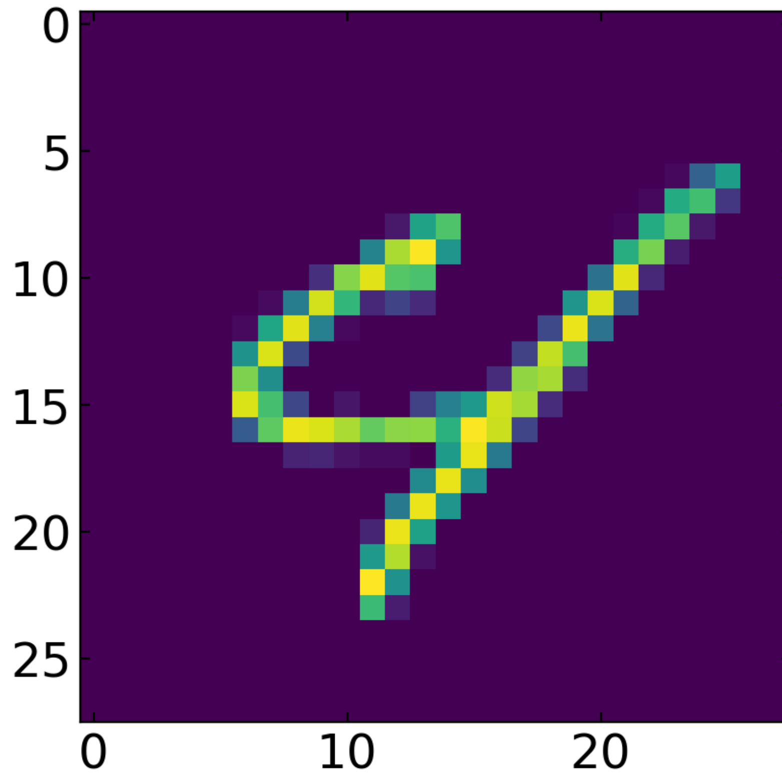
The following is just to tell you how accurate the model is. According to that, we achieve about 99% accuracy.

```
Test loss: 0.02492290735244751
Test accuracy: 0.9919000267982483
```

2.5 Evaluate the accuracy of your visual neural network ;-)

We can now randomly select a number from the tesing (not the training) data and display the prediction and compare that vs. your own built in neural network.

```
My prediction is:  4
```



2.6 Where to go from here?

If you are interested in applying CNNs to classify and locate data, have a look at our framework for single particle tracking in real time.