Convolutional neural networks

Deep learning course for industry

Mitko Veta

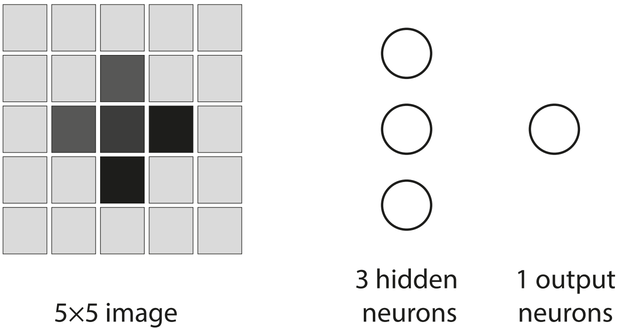
Eindhoven University of Technology
Department of Biomedical Engineering

2020

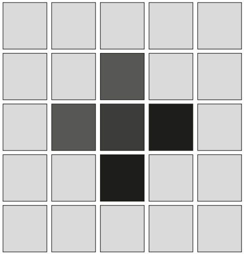
Learning goals

- Demonstrate how deep neural networks can be modified to be more suitable for image data.
- Introduce the Keras neural networks API

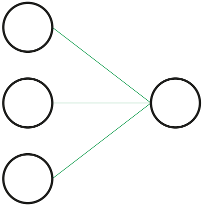
Images as inputs to a neural network



Images as inputs to a neural network



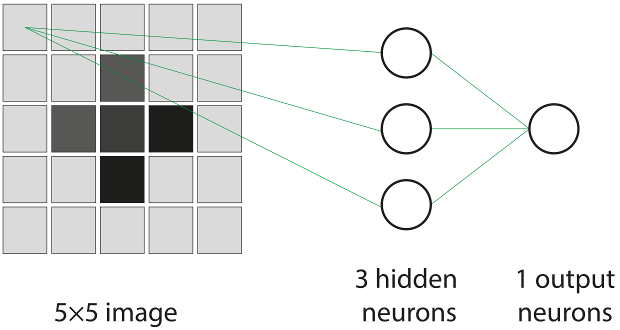
5×5 image



3 hidden neurons

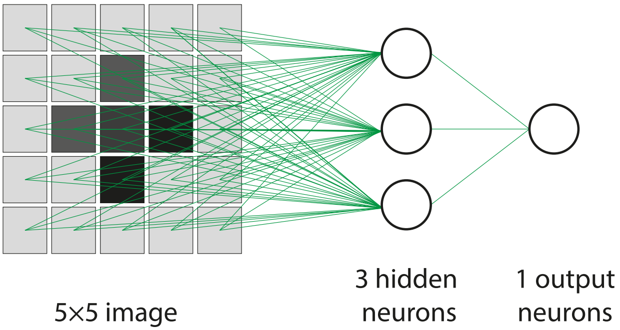
1 output neurons

Images as inputs to a neural network

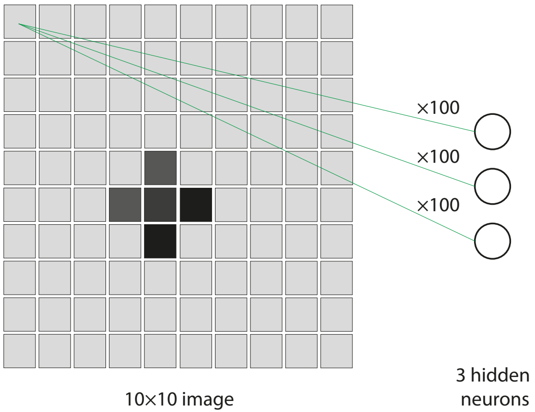


The biases $w_{i,0}$ that are not shown.

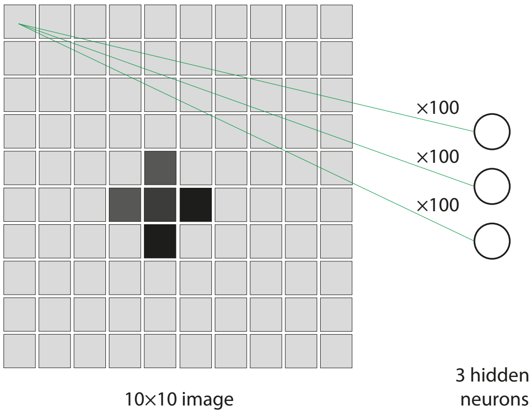
Images as inputs to a neural network



The number of parameters explodes with larger image sizes



The number of parameters explodes with larger image sizes

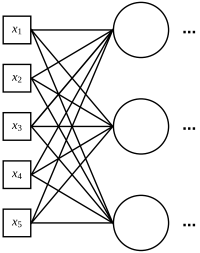


$$\# \text{ parameters} = (\text{height} \times \text{width} \times \# \text{ channels} + 1) \times \# \textit{neurons}$$

The "+1" comes from the biases $w_{i,0}$.

Towards convolutional neural networks

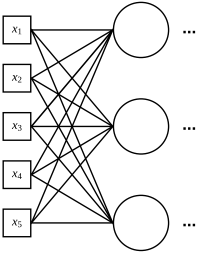
Example (1-D image for simplicity): 5×1 input image, 3 hidden neurons.



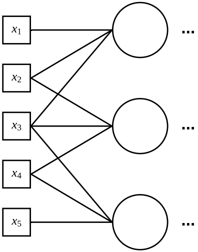
full connectivity: 15 parameters

Towards convolutional neural networks

Example (1-D image for simplicity): 5×1 input image, 3 hidden neurons.



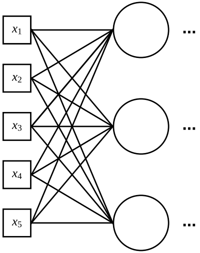
full connectivity: 15 parameters



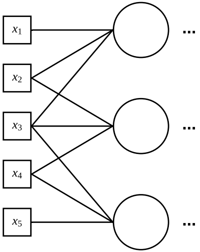
sparse connectivity: 9 parameters

Towards convolutional neural networks

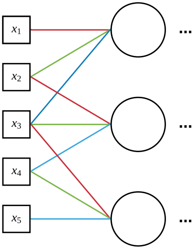
Example (1-D image for simplicity): 5×1 input image, 3 hidden neurons.



full connectivity: 15 parameters



sparse connectivity: 9 parameters

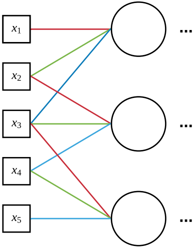


shared weights: 3 parameters

Note: the poor biases are again, ignored, but there are three of them in each case

Towards convolutional neural networks

Let the outputs of the three neurons be $\sigma(a_1), \sigma(a_2), \sigma(a_3)$. Then:



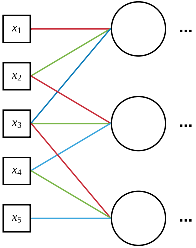
$$a_1 = x_1 w_1 + x_2 w_2 + x_3 w_3$$

$$a_2 = x_2 w_1 + x_3 w_2 + x_4 w_3$$

$$a_3 = x_3 w_1 + x_4 w_2 + x_5 w_3$$

Towards convolutional neural networks

Let the outputs of the three neurons be $\sigma(a_1), \sigma(a_2), \sigma(a_3)$. Then:



$$a_1 = x_1 w_1 + x_2 w_2 + x_3 w_3$$

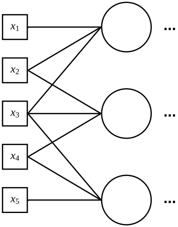
$$a_2 = x_2 w_1 + x_3 w_2 + x_4 w_3$$

$$a_3 = x_3 w_1 + x_4 w_2 + x_5 w_3$$

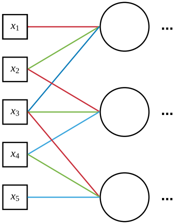
$$[a_1, a_2, a_3] = [x_1, x_2, x_3, x_4, x_5] * [w_3, w_2, w_1]$$

, where $*$ is the convolution operator, thus a **convolutional layer**.

Motivation (or rather a justification)

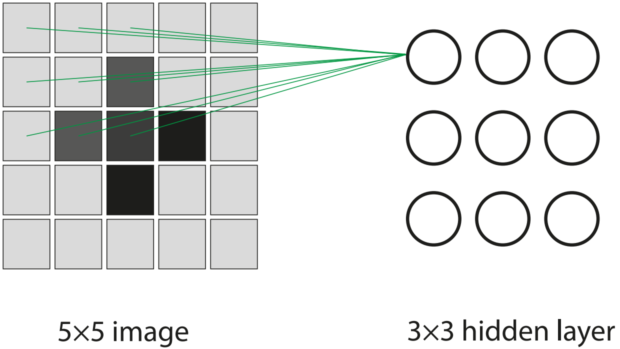


sparse connectivity
motivation: the features appear locally

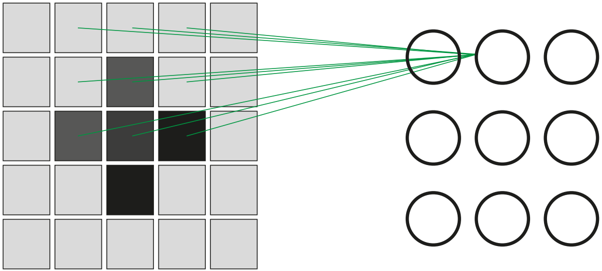


shared weights
motivation: the features repeat throughout the image

Towards convolutional neural networks in 2D



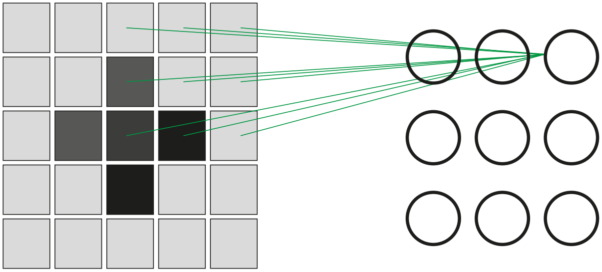
Towards convolutional neural networks in 2D



5×5 image

3×3 hidden layer

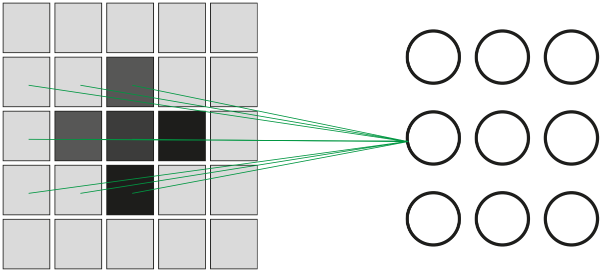
Towards convolutional neural networks in 2D



5×5 image

3×3 hidden layer

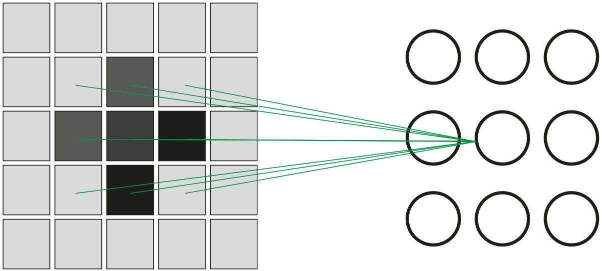
Towards convolutional neural networks in 2D



5×5 image

3×3 hidden layer

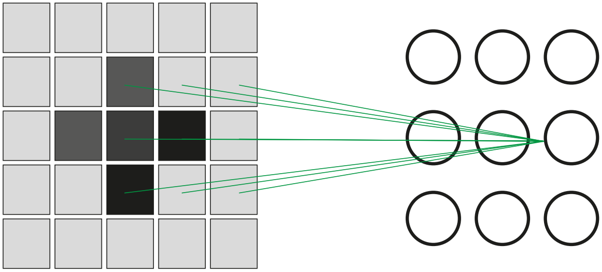
Towards convolutional neural networks in 2D



5×5 image

3×3 hidden layer

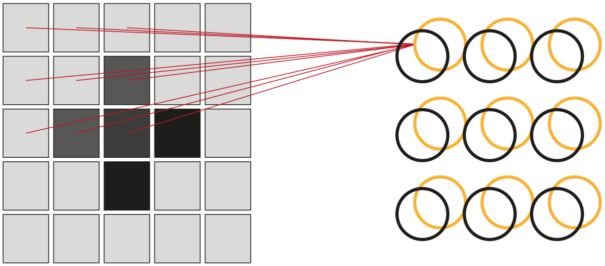
Towards convolutional neural networks in 2D



5×5 image

3×3 hidden layer

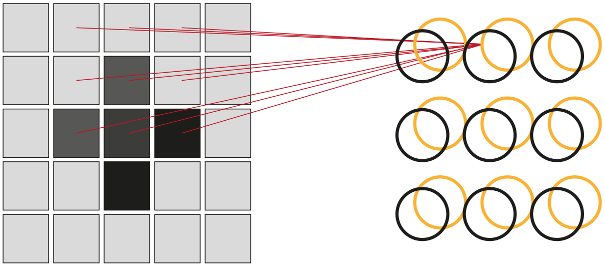
Adding a second feature map



5×5 image

3×3×2 hidden layer
(2 feature maps)

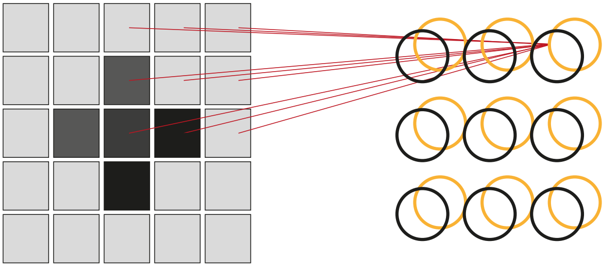
Adding a second feature map



5×5 image

3×3×2 hidden layer
(2 feature maps)

Adding a second feature map



5×5 image

3×3×2 hidden layer
(2 feature maps)

Convolution with padding

Figure source: https://github.com/vdumoulin/conv_arithmetic

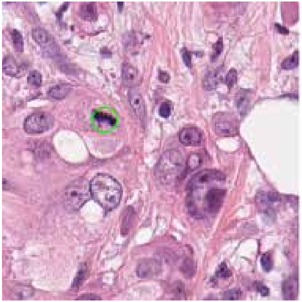
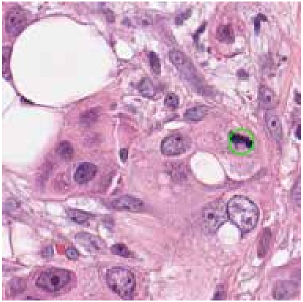
Computing the output size

$$\text{output size} = \frac{\text{input size} - \text{kernel size} + 2 \times \text{padding}}{\text{stride}} + 1$$

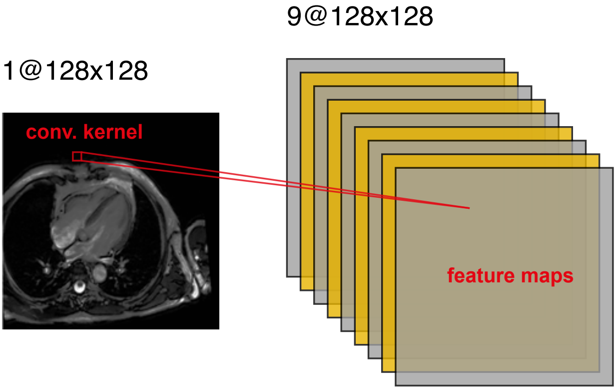
In this example: input size $= 5$, kernel size $= 3$, padding $= 1$, stride $= 1$.
The output size is $(5 - 3 + 2 \times 1)/1 + 1 = 5$.

Motivation (or rather justification) for CNNs

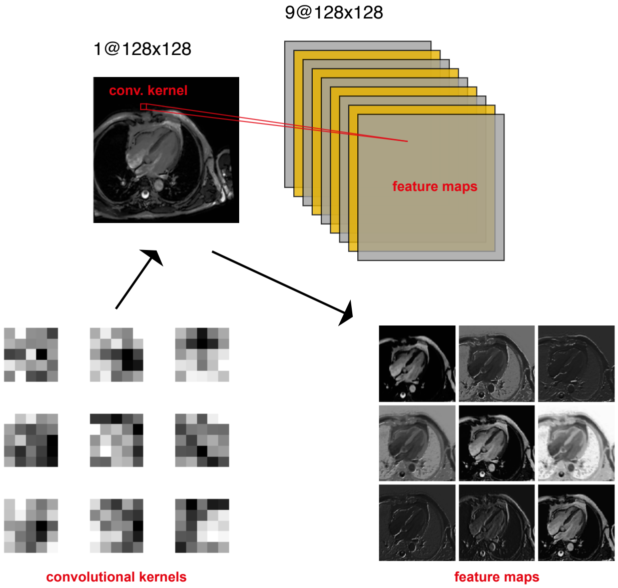
The features of interest can appear at different locations in the image.

Kernels and feature maps



Kernels and feature maps



Motivation (or rather a consequence) for deep CNNs

The network learns low-level features in the first layers, and builds up towards more complex features in the deeper layers: intensity $\rightarrow$ edges and colour blobs $\rightarrow$ junctions $\rightarrow$ shapes $\rightarrow$ etc.

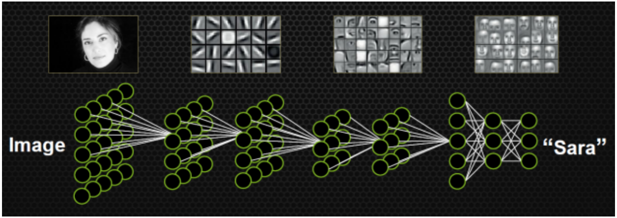


Figure source: nvidia.com

Equivariance and invariance to translation

The convolutional layers are **equivariant with translation**: as the input is translated, the output is translated in a predictable manner.

Equivariance and invariance to translation

The convolutional layers are **equivariant with translation**: as the input is translated, the output is translated in a predictable manner.

A desired property of neural networks for classification is **invariance**: as the input is translated, the output remains the same.

Partial translational invariance of CNNs is achieved with the max-pooling operator.

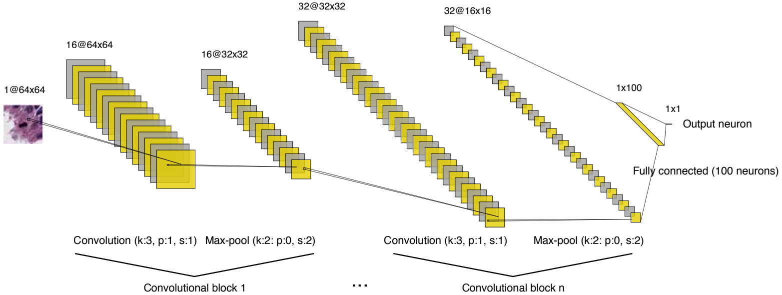
Note: there are other types of invariance e.g. rotational.

Max-pooling

A max-pool with a 2×2 kernel stride and size 2 (most common form) will reduce the image size by 2 in each dimension (a useful side-effect).

A “typical” CNN architecture for 2D image classification

Note that the convolution is a linear operation so non-linearities (such as ReLU) are still needed.

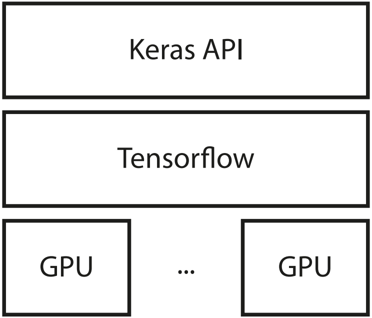


Training CNNs

Example by Andrej Karpathy.

What is Keras?

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.



What is Keras?

Two API implementations: keras.io (reference implementation) and from Tensorflow.

```python
import keras
import tensorflow as tf

layer = keras.layers.Dense # used in slides
layer = tf.keras.layers.Dense # used in the exercises
```

Implementing neural networks in Keras

The core data structure of Keras is a **model**, a way to organize layers.

The simplest type of model is the **Sequential** model, a linear stack of layers. Covers majority of use cases.

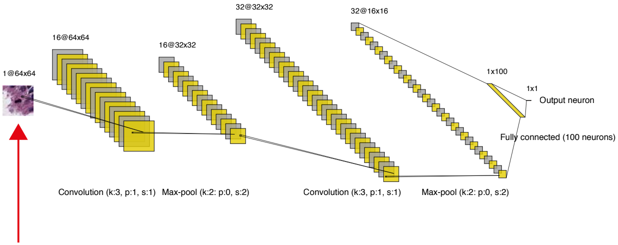
For more complex architectures, you should use the Keras **functional API**, which allows to build arbitrary graphs of layers. Covers almost all use cases.

Functional API

Most common use cases:

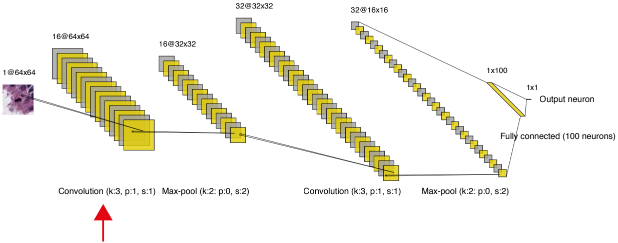
- Models with multiple input and/or multiple outputs
- Models with shared layers

Example implementation



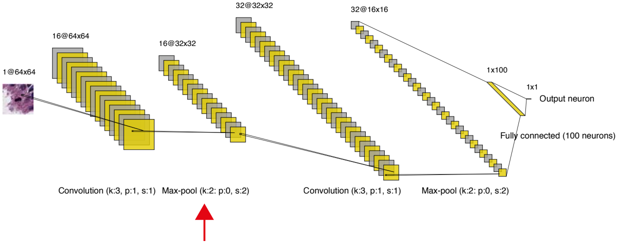

```
# placeholder for RGB images
inputs = keras.Input(shape=[64, 64, 1])
```

Example implementation



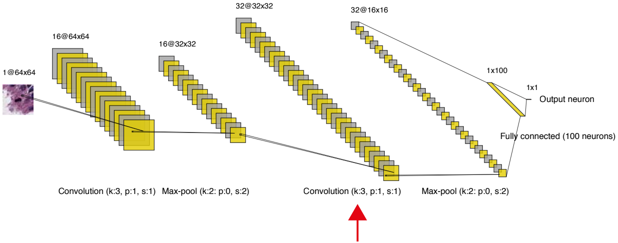

```
conv1 = keras.layers.Conv2D(
    filters      = 16,
    kernel_size  = 3,
    strides      = (1, 1),
    padding      = "same",
    activation   = "relu")
```

Example implementation



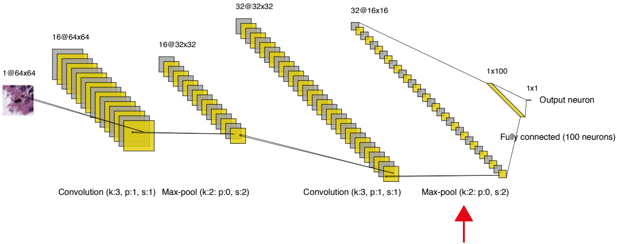

```
maxPool1 = keras.layers.MaxPool2D(
    pool_size = (2, 2),
    strides   = (2,2),
    padding   = "valid")
```

Example implementation



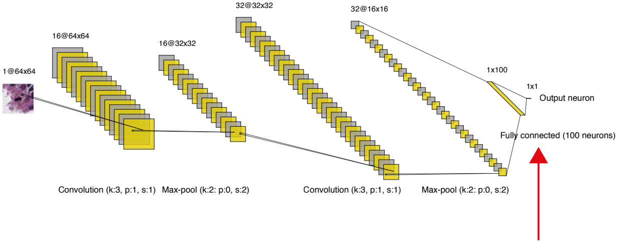

```
conv2 = keras.layers.Conv2D(
    filters      = 32,
    kernel_size  = 3,
    strides      = (1, 1),
    padding      = "same",
    activation   = "relu")
```

Example implementation



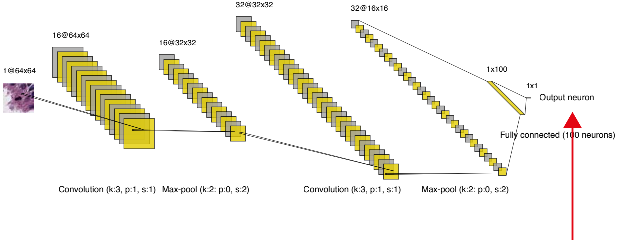

```
maxPool2 = keras.layers.MaxPool2D(
    pool_size = (2, 2),
    strides   = (2,2),
    padding   = "valid")
```

Example implementation



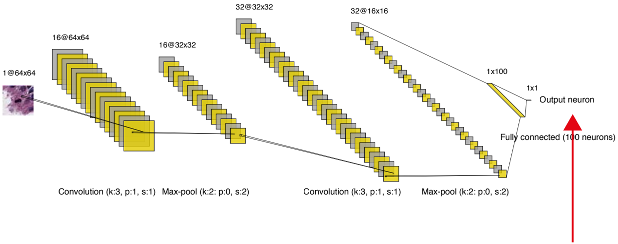

```
dense = keras.layers.Dense(
    units      =  100,
    activation = "ReLU")
```

Example implementation




```
output = keras.layers.Dense(
    units        =  1,
    activation   = "sigmoid")
```

Example implementation



Making the connection:

```
x = maxPool1(conv1(inputs))
x = maxPool2(conv2(x))
x = flatten(x)
x = dense(x)
outputs = output(x)
```

Training a model

```
model = keras.Model(inputs = inputs, outputs = outputs)

loss = keras.losses.categorical_crossentropy
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9)

model.compile(loss=loss, optimizer=optimizer)
```

Training a model

```
model = keras.Model(inputs  = inputs, outputs = outputs)

loss = keras.losses.categorical_crossentropy
optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9)

model.compile(loss=loss, optimizer=optimizer)

# x_train and y_train are Numpy arrays
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

Training a model

Or, feed batches manually in a training loop:

```
for i in range(0, num_iterations):
    x_batch, y_batch = some_batch_generator(i)
    model.train_on_batch(x_batch, y_batch)
```

Model evaluation

```
loss_and_metrics = model.evaluate(x_test, y_test,
        batch_size=128)

classes = model.predict(x_test, batch_size=128)
```

Summary

- Compared to fully connected neural networks, convolutional neural networks have sparse connectivity and weight sharing, which makes them suitable for image data.
- The Keras neural networks API allows for user-friendly implementation and training of (convolutional) neural networks.