Convolutional Neural Networks for Image Processing

Image Processing with Neural Networks :

images as data : data.shape → (2832, 4256, 3)

height — 2832, width — 4256, number of channels (RGB) — 3

data[250, 3500] → array([0.74, 0.078, 0.145]) → it's a red pixel!

row idx — 250, column idx — 3500; R — 0.74, G — 0.078, B — 0.145

input and modify images :

```
import matplotlib.pyplot as plt
data = plt.imread('image.jpg')
data[:,:,1]=0  → set green channel as zero
data[:,:,2]=0  → set blue channel as zero
plt.imshow(data)   → input image with only RED channel.
plt.show()
```

change an image : data[200:1200, 200:1200, :] = [0,1,0] → make it green!

in black & white images → brighter pixels have higher values → 255 white, → 0 black

represent categorical image labels mathmatically : One-hot Encoding

```
categories = np.array(["t-shirt", "dress", "shoe"])
n_categories = 3
ohe_labels = np.zeros((len(labels), n_categories))
for ii in range(len(labels)):
    jj = np.where(categories == labels[ii])
    ohe_labels[ii, jj] = 1
```

★ One-hot encoding can be used for test predictions

```
test array([[1,0,0],      prediction([[1,0,0],   ⇒ (test * prediction).sum()
            [0,1,0],                  [0,0,1],
            ⋮                          ⋮
```

(test * prediction).sum() → num of corret classifications

s.a.m

Keras for image classification:

from Keras.models import Sequential

model = Sequential()

train_data.shape → (50, 28, 28, 1)

number of images: 50 · h: 28 · w: 28 · C (BW): 1

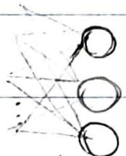
from keras.layers import Dense

model.add(Dense(10, activation = 'relu', input_shape = (784,)))

10 → number of units; 784 = 28 × 28

model.add(Dense(10, activation = 'relu'))

model.add(Dense(3, activation = 'softmax')) → multi-class



model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])

train_data = train_data.reshape((50, 784))

model.fit(train_data, train_labels, validation_split = 0.2, epochs = 3)

validation_split = 0.2 → to prevent overfitting → validation

epochs = 3 → go over all data 3 times

→ training : model adjusts weights through backpropagation and gradient descend

test_data = test_data.reshape ((10, 784))

model.evaluate (test_data, test_labels)

Using Convolutions:

Natural images have spatial corelations → e.g. pixels along a contour or edge

Convolutions

array = np.array([0, 0, 0, 0, 0, 1, 1, 1])

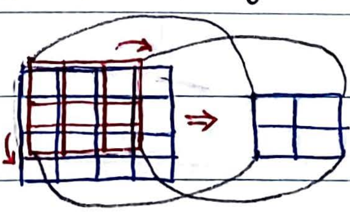
kernel = np.array([-1, 1]) → 1-D Convolution → image : 2D Convolution

Conv = np.array([0, ... , 0])

$\text{kernel} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$ ⇒ Left Edge Detector!

for ii in range(8):

  Conv(ii) = (kernel * array[ii : ii+2]).sum() → Conv ⇒ array([0, 0, 0, 0, 1, 0])



Kernel : $\begin{bmatrix} -1 & 1 & -1 \\ -1 & 1 & -1 \\ -1 & 1 & -1 \end{bmatrix}$ → finds vertical lines

$\begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ -1 & -1 & -1 \end{bmatrix}$ → horizontal lines

$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ → light spot in the dark

$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ → dark spot in the light

s.a.m

from keras.layers import Conv2D

learning is the same as Dense layer but with fewer weights

Conv2D (10, kernel_size = 3, activation = 'relu')

number of filters

3x3

Convolution has one weight for each pixel in the kernel

from keras.models import Sequential

num parameters = 10 x 3 x 3 = 90

from keras.layers import Dense, Conv2D, Flatten

model = Sequential()

model.add (Conv2D (10, kernel_size = 3, activation = 'relu', input_shape = (img_rows, img_columns, 1)))

num channels

model.add (Flatten()) → connection between Convolution and Dense layer

→ flatten the conv output into a one dimentional array

model.add (Dense (3, activation = 'softmax'))

→ three classes

model.compile (...)

→ here we do not reshape input data, because we want to have the spatial relationships → so we should define input shape in the Conv2D

model.fit (..., batch_size = 10)

also

model.evaluate (...) → output: [0.547, 1.0]

→ cross entropy loss

→ accuracy!

zero padding : add zeros to base image to have same size for the feature map after convolution

model.add (Conv2D (..., padding = 'valid')) → default

padding = 'same' → zero padding applied!

Stride : size of the step we take with the kernel over the input image

model.add (Conv2D (..., strides = 1)) → strides > 1 → output image is smaller than input

calculate the size of output : $O = \frac{I - K + 2P}{S} + 1$

input size

kernel size

size of zero padding

strides

★ Dilated Convolution → useful when we want to aggregate information across multiple scales

model.add (Conv2D (...), dilation = 2)

Going Deeper :

input_shape keyword argument is required for first layer only. other layers do not need this argument to be specified.

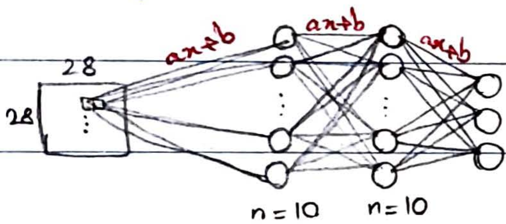
· Deep Network → more than one convolutional layer

why adding more layers ? → | Features in early layers → simple features
| Features in intermediate layers → more complex features (simples objects)
| Features in late layers → specific types of objects

hierarchical representation of data

Deep Networks → more computational cost
→ may require more data

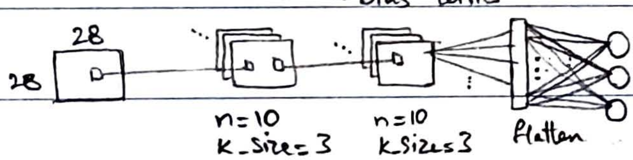
How many parameters in a model? ⇒ model.summary() → description of model + total trainable and non-trainable parameters



Fully Conneeted model

$784\times10 + 10 \quad 10\times10+10 \quad 3\times10+3 \rightarrow$ 7993 (trainable) parameters

bias terms



Convolution model
(padding='same', stride=1)

$9\times10 + 10 \quad 9\times10\times10+10 \quad - \quad 28\times28\times10\times3+3 \rightarrow$ 24533 (trainable) parameter

kernel values (3x3) · no. kernels · bias term

★ Convolution networks do not necessarily reduce number of parameters. In FC networks most number of parameters are for the first dense layer. In contrast, in CNNs most parameters come from the last layers. (mainly the first dense layer after the flattening)

pooling operations: Convolve an image with a kernel. On each part, replace the related part of the image with the maximum value available on that part. (max pooling)

if the kernel is 2x2 → the image dimensions reduce to a quarter after pooling.

```
result = np.zeros((im.shape[0] // 2, im.shape[1] // 2))
result[0,0] = np.max(im[0:2, 0:2])
result[0,1] = np.max(im[0:2, 2:4])
...
```

```
for ii in range(result.shape[0]):
    for jj in range(result.shape[1]):
        result[ii, jj] = np.max(
            im[ii*2:ii*2+2,
               jj*2:jj*2+2])
```

⇓ in Keras

```
from keras.layers import MaxPool2D
model = Sequential()
model.add(Conv2D(~))
model.add(MaxPool2D(2))
model.add(Conv2D(~))
model.add(MaxPool2D(2))
...
```

put pooling layer after each Conv layer

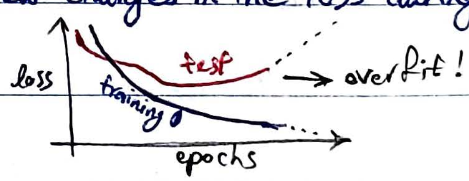
(2) → kernel size

model.Summary()
↓
number of parameters reduced from ~30000 to ~2000 !

Understanding and Improving Deep CNNs

is learning progressive as expected? → view changes in the loss during learning

Learning curves




```
training = model.fit(train_data, train_labels, ...)
import matplotlib.pyplot as plt
plt.plot(training.history['loss'])
plt.plot(training.history['val_loss'])
plt.show()
```

↳ "training" has a dictionary to store the learning curves contains functions which can be executed at the end of each training epoch.

★ Storing the optimal parameters before the model starts overfitting?

```
from keras.callbacks import ModelCheckpoint
```
→ stores the weights

file type

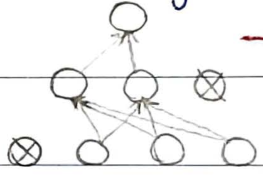
```
checkpoint = ModelCheckpoint('weights.hdf5', monitor='val_loss',
                             save_best_only = True )
callbacks_list = [checkpoint]
```

it only gets lists

```
model.fit (train_data, train_labels, ..., callbacks = callbacks_list)
- - - -
model.load_weights ('weights.hdf5')
model.predict_classes (test_data) → array([2, 1, 2, 0, ...])
```

Neural network regularization : to prevent overfitting

→ Dropout: In each learning step, select a subset of units and ignore them in the forward pass, and, in the back-propagation of error → introduced in 2014



```
from keras.layers import Dropout
  ⋮
model.add (Conv2D(...))
model.add (Dropout (0.25))
  ⋮
```
→ proportion of the units to be neglected (randomly changes the selected units in each learning step)

→ Batch Normalization : Rescale the outputs of each layer → in 2015

```
from keras.layers import BatchNormalization
  ⋮
model.add (Conv2D(~))
model.add (BatchNormalization ())
  ⋮
```

→ Be careful! Sometimes models become worse when using these two together.

The disharmony between batch normalization and dropout

Interpreting the model: CNNs are black boxes! → it will evolve rapidly, soon!

model.layers → [<.. Conv>,
<.. Conv>,
<.. Flatten>,
<dense>]

Conv1 = model.layers[0]

weights1 = Conv1.get_weights() → len(weights) = 2

kernels1 = weights1[0] → kernels1.shape = $(\underbrace{3,3}_{3\times3\text{ kernels}},\overset{\text{num\_channels}}{1},\underset{\text{num\_kernels}}{2})$

kernels1_1 = kernels1[:, :, 0, 0] → kernels1_1.shape = (3,3)

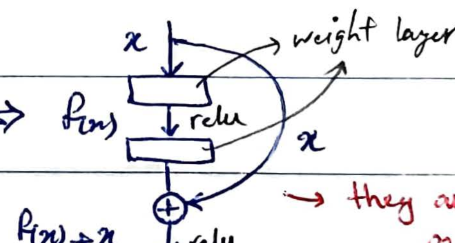
plt.imshow(kernels1_1) →  → but, what it does?!

test_image = test_data[3, :, :, 0] ← Convolve that with one image

filtered_image = Convolution(test_image, kernels1_1) → we coded the function before! (few pages back)

plt.imshow(filtered_image)

Further reading: https://distill.pub/2017/feature-visualization/
by Chris Olah

what next?
→ deeper networks
→ Residual networks ⇒ 

→ they are surprisingly effective!
→ Transfer learning → great when you do not have a lot of data.
→ Fully Convolutional Networks (FCNs) ⇒ get image and output another image (convolution + deconvolution)
→ Generative Adversarial Networks (GANs) ⇒ create new images!