

Lecture 07
Deep Learning 02
Convolutional Neural Networks

2026-04-10

Sébastien Valade



1. Introduction

2. How the brain recognizes images

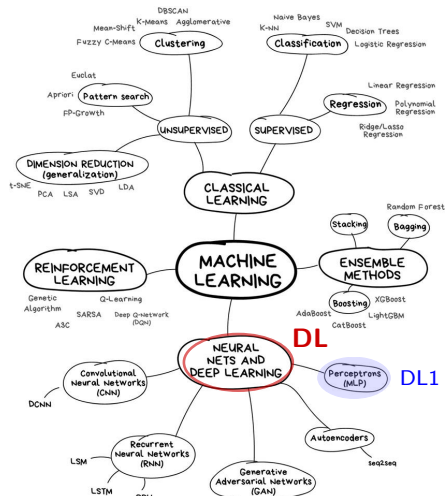
3. CNN building blocs

4. Transfer learning

5. Training workflow

6. Application

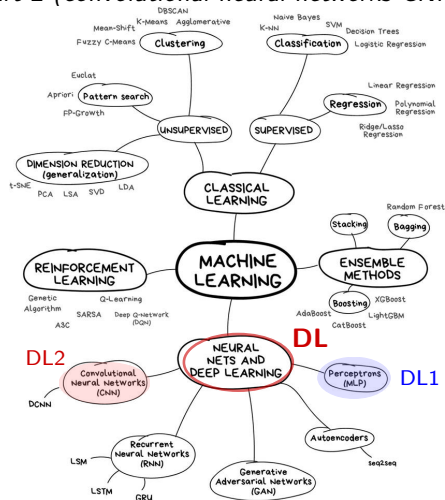
Last week: **Neural Networks** Part-1 (*multilayer perceptrons MLP*) - DL1



source

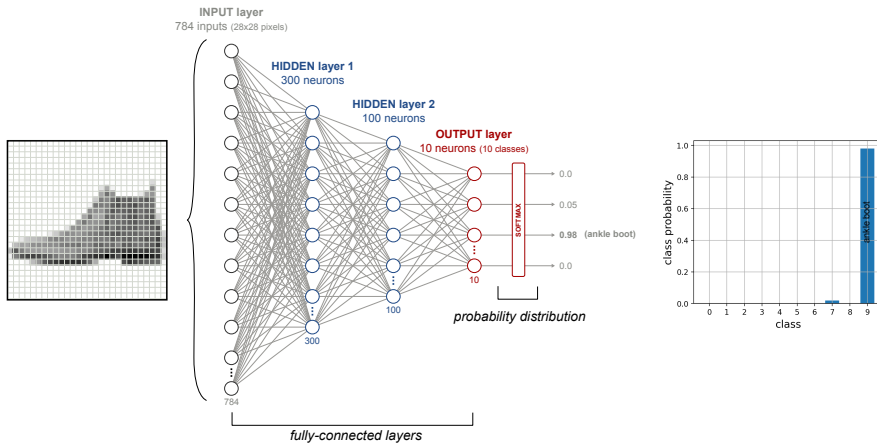
Last week: **Neural Networks** Part-1 (*multilayer perceptrons MLP*) - DL1

This week: **Neural Networks** Part-2 (*convolutional neural networks CNN*) - DL2



Last lecture

⇒ we trained a fully-connected neural network (MLP) to classify a “simple” dataset



Last lecture

⇒ this “simple” network with only 2 hidden layers, handling “simple” images (28x28 pixels, 1-channel), and “few classes” (10 classes) has a total of **266 610 parameters** to be trained

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235,500
dense_1 (Dense)	(None, 100)	30,100
dense_2 (Dense)	(None, 10)	1,010

Total params: **266,610** (1.02 MB)

Trainable params: **266,610** (1.02 MB)

Non-trainable params: **0** (0.00 B)

Last lecture

- ⇒ this “simple” network with only 2 hidden layers, handling “simple” images (28x28 pixels, 1-channel), and “few classes” (10 classes) has a total of **266 610 parameters** to be trained

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235,500
dense_1 (Dense)	(None, 100)	30,100
dense_2 (Dense)	(None, 10)	1,010

Total params: **266,610** (1.02 MB)
Trainable params: **266,610** (1.02 MB)
Non-trainable params: **0** (0.00 B)

- ⇒ limits of fully-connected FC networks:

1. hard to scale to larger images or more complex classification tasks

*EX: 128x128 RGB image with 1st hidden-layer of 300 neurons = $(128*128*3)*300 = >14$ millions parameters*

2. spatial structure of images are not respected (2D array flattened to 1D array)

Last lecture

- ⇒ this “simple” network with only 2 hidden layers, handling “simple” images (28x28 pixels, 1-channel), and “few classes” (10 classes) has a total of **266 610 parameters** to be trained

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235,500
dense_1 (Dense)	(None, 100)	30,100
dense_2 (Dense)	(None, 10)	1,010

Total params: **266,610** (1.02 MB)
 Trainable params: **266,610** (1.02 MB)
 Non-trainable params: **0** (0.00 B)

- ⇒ limits of fully-connected FC networks:

1. hard to scale to larger images or more complex classification tasks

*EX: 128x128 RGB image with 1st hidden-layer of 300 neurons = $(128*128*3)*300 = >14$ millions parameters*

2. spatial structure of images are not respected (2D array flattened to 1D array)

- ⇒ do we *really* need to connect all the pixels together?

would rather need sparse connections: fewer weights, nearby regions related, & far apart regions not related

This lecture: from MLPs to CNNs

1. Introduction

2. How the brain recognizes images

1. Perception by the visual cortex
2. Reproducing brain perception with neural networks

3. CNN building blocs

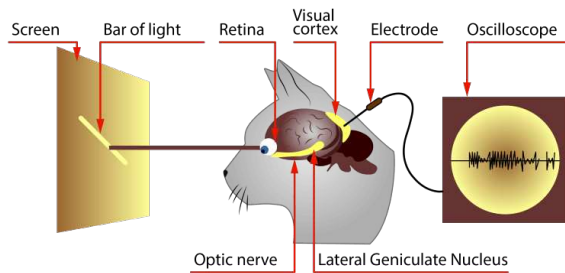
4. Transfer learning

5. Training workflow

6. Application

Perception by the visual cortex

- ⇒ **Convolutional Neural Networks (CNNs)** emerged from the study of the brain's visual cortex
- ⇒ Experiments on cats & monkeys gave insights on *how perception works* (Hubel & Wiesel 1958¹, 1959², 1968³)
NB: the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work



source

¹ Hubel D. (1959) "Single Unit Activity in Striate Cortex of Unrestrained Cats", The Journal of Physiology

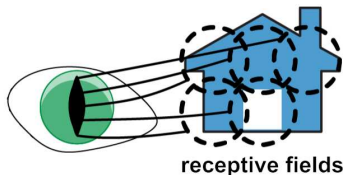
² Hubel D. & Wiesel T. (1959) "Receptive Fields of Single Neurons in the Cat's Striate Cortex", The Journal of Physiology

³ Hubel D. & Wiesel T. (1968) "Receptive Fields and Functional Architecture of Monkey Striate Cortex", The Journal of Physiology

Perception by the visual cortex

⇒ Insights on *how perception works*:

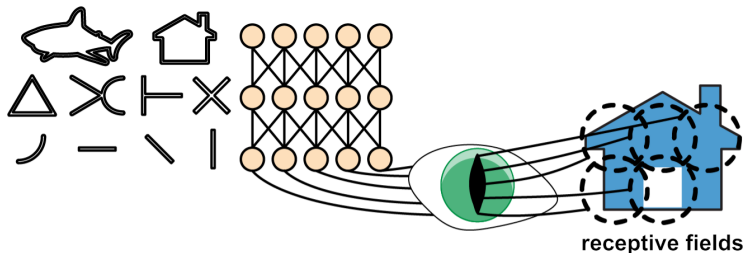
1. biological neurons respond to specific patterns in regions (a.k.a. **receptive fields**) of the visual field



Perception by the visual cortex

⇒ Insights on *how perception works*:

1. biological neurons respond to specific patterns in regions (a.k.a. **receptive fields**) of the visual field
2. the visual cortex is organized in layers: as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields
 - neurons in low-level layers have small receptive fields and react to simple patterns (e.g., edges)
 - NB: *two neurons may have the same receptive field but react to different line orientations*
 - neurons in high-level layers have larger receptive fields and react to more complex patterns that are combinations of the lower-level patterns (e.g., triangles, rectangles, ... → e.g., house, face, ...)

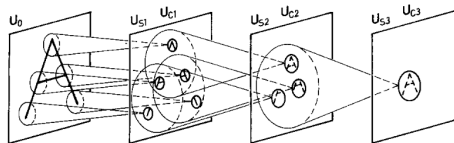


from: Géron A. (2022)

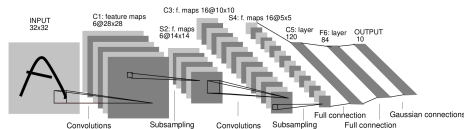
Reproducing brain perception with neural networks

⇒ These studies of the visual cortex inspired the **neocognitron** (Fukushima 1980⁴), which gradually evolved into what we now call **convolutional neural networks CNNs**, a.k.a. ConvNets, (LeCun et al. 1998⁵)

Fukushima 1980



LeCun et al. 1998



⁴ Fukushima, K. (1980) "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", Biological Cybernetics

⁵ LeCun, Y. et al. (1998) "Gradient-Based Learning Applied to Document Recognition", Proceedings of the IEEE 86, no. 11

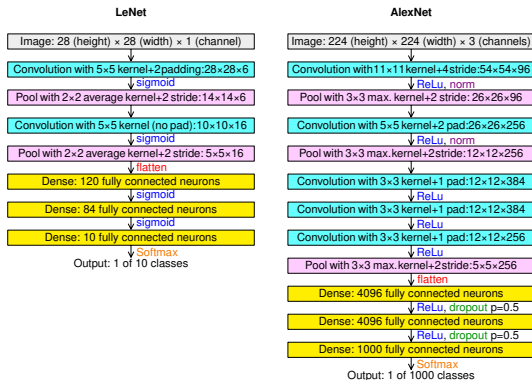
1. Introduction
2. How the brain recognizes images
- 3. CNN building blocs**
 1. Building blocs (overview)
 2. Convolutional layer
 3. Pooling layer
 4. Flatten layer
 5. Fully-connected layer
 6. Normalization layer
 7. Dropout layer
 8. Summary (cheat-sheet)
 9. Residual block
4. Transfer learning

3.1. Building blocs (overview)

1. **CONV** convolutional layers CONV (+ activation function **ReLU**)
2. **POOL** pooling layers POOL
3. **FLAT** flatten layer
4. **FC** fully-connected layers (dense layers) (+ activation functions **ReLU** & **softmax**)
5. **BN** batch normalization layer
6. **DO** dropout layer
7. layers in advanced architectures:
 - RES** residual blocks (ResNet)
 - ICP** inception modules (GoogLeNet)

Building blocks of a CNN

⇒ consider 2 milestones CNN architectures: **LeNet-5** (*LeCun et al. 1998*)⁶ and **AlexNet** (*Krizhevsky et al. 2012*)⁷



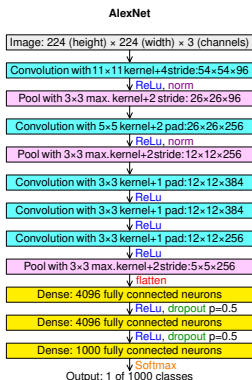
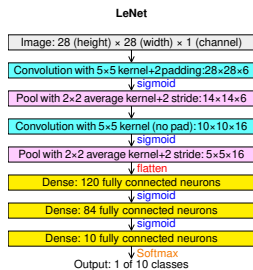
modified after: [wiki](#)

⁶ LeCun, Y. et al. (1998) "Gradient-Based Learning Applied to Document Recognition", Proceedings of the IEEE 86, no. 11

⁷ Krizhevsky, Alex, et al. (2012) "ImageNet Classification with Deep Convolutional Neural Networks", Proceedings of the 25th NeurIPS Conference

Building blocks of a CNN

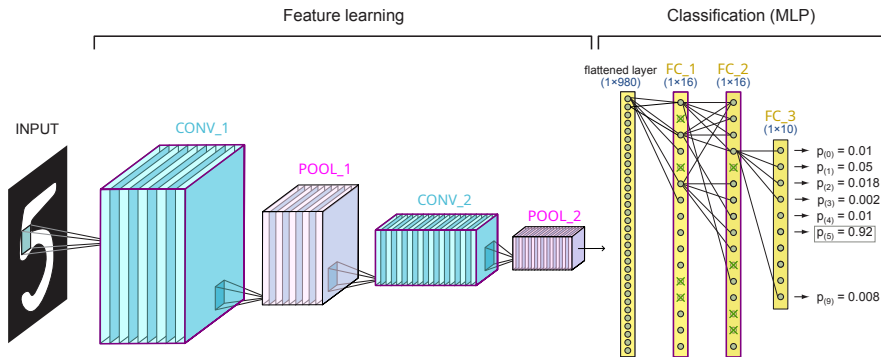
⇒ consider 2 milestones CNN architectures: **LeNet-5** (*LeCun et al. 1998*) and **AlexNet** (*Krizhevsky et al. 2012*)



- convolutional layers** CONV ⇒ extract features
→ receptive field, filter kernel/depth, feature maps, padding, stride
- pooling layers** POOL ⇒ downsample
→ pooling kernel, stride
- dense layers** FC ⇒ classify
→ fully connected layers
- activation functions
→ sigmoid, ReLU, etc. ⇒ achieve non-linearity
→ softmax ⇒ achieve class probability distribution
- normalization layer** ⇒ normalize activations
→ batch norm BN
- flatten layer** ⇒ matrix/tensor to vector
- dropout layer** ⇒ prevent overfitting (regularization)
→ in the fully connected layers

Building blocks of a CNN

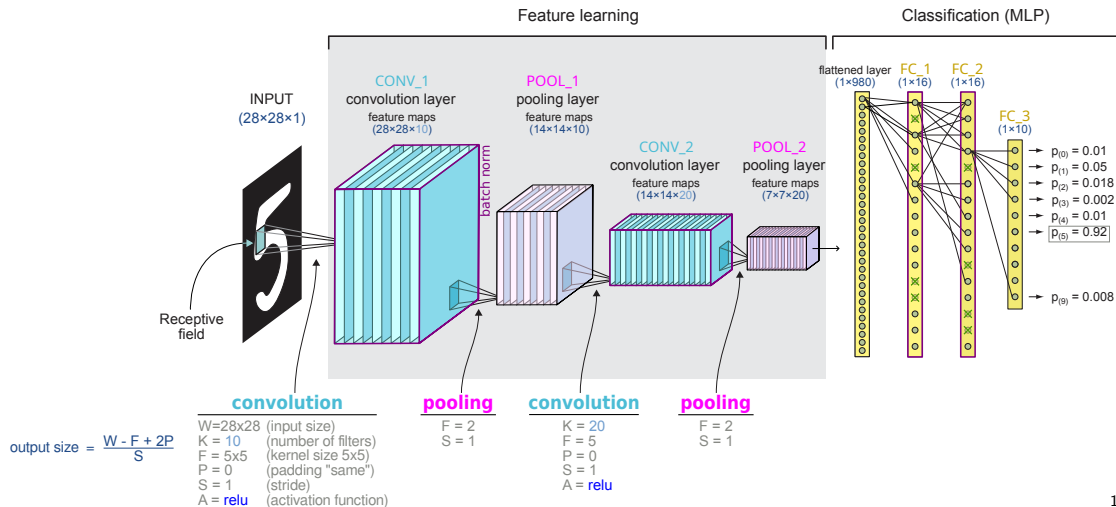
⇒ illustration of the building blocs of convolutional networks (*modified after Elgendy (2020)*)



3.1. Building blocs (overview)

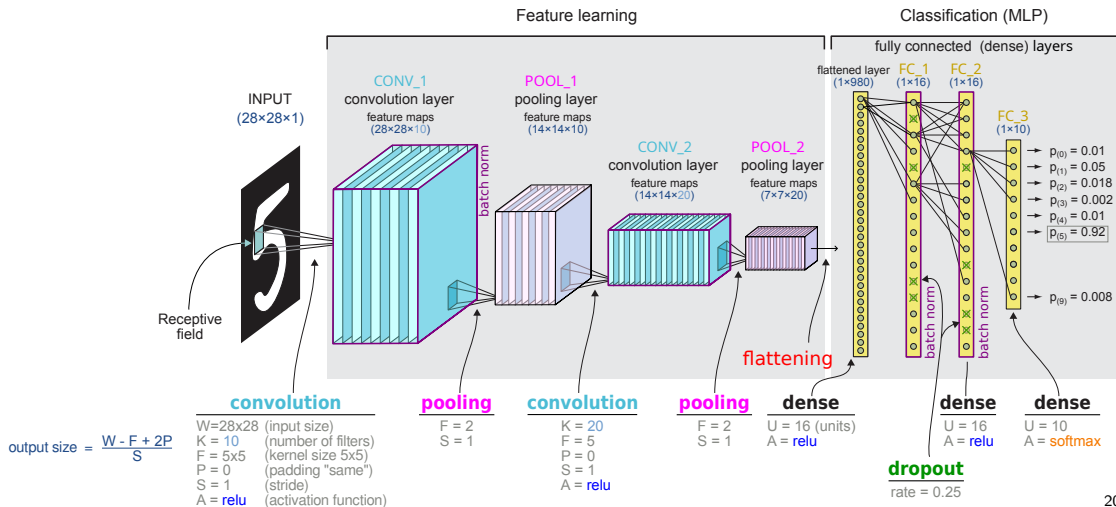
Building blocks of a CNN

⇒ illustration of the building blocs of convolutional networks (*modified after Elgendy (2020)*)



Building blocks of a CNN

⇒ illustration of the building blocs of convolutional networks (*modified after Elgendy (2020)*)



Building blocks of a CNN

⇒ illustration of the implementation in Keras (Tensor Flow)

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization

# Build model object
model = Sequential()

# CONV_1 (with BatchNorm)
model.add(Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)))
model.add(BatchNormalization())

# POOL_1
model.add(MaxPooling2D(pool_size=(2, 2)))

# CONV_2 (with BatchNorm)
model.add(Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'))
model.add(BatchNormalization())

# POOL_2
model.add(MaxPooling2D(pool_size=(2, 2)))

# FLATTEN
model.add(Flatten())

# FC_1 (with BatchNorm & Dropout)
model.add(Dense(16, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(rate=0.25))

# FC_2 (with BatchNorm & Dropout)
model.add(Dense(16, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(rate=0.25))

# FC_3 (with softmax)
model.add(Dense(10, activation='softmax'))

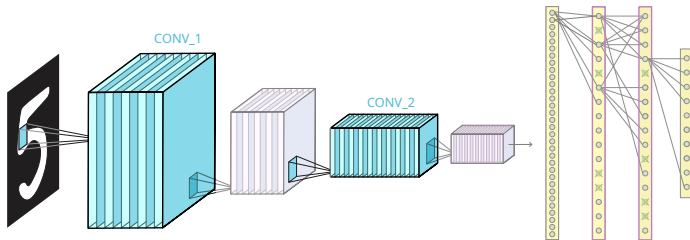
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 10)	260
batch_normalization (BatchNormalization)	(None, 28, 28, 10)	40
max_pooling2d (MaxPooling2D)	(None, 14, 14, 10)	0
conv2d_1 (Conv2D)	(None, 14, 14, 20)	5,020
batch_normalization_1 (BatchNormalization)	(None, 14, 14, 20)	80
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 20)	0
flatten (Flatten)	(None, 980)	0
dense (Dense)	(None, 16)	15,696
batch_normalization_2 (BatchNormalization)	(None, 16)	64
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 16)	272
batch_normalization_3 (BatchNormalization)	(None, 16)	64
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 10)	170

Total params: 21,666 (84.63 KB)
 Trainable params: 21,542 (84.15 KB)
 Non-trainable params: 124 (496.00 B)

Let's explain what each layer does

1. **CONV** Convolutional layer

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

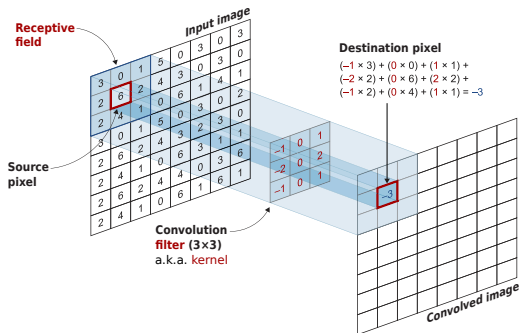
    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(10, activation='softmax')
])

```

1. **CONV** Convolutional layer

⇒ **convolution**: reminder of Lecture 03 (filtering)



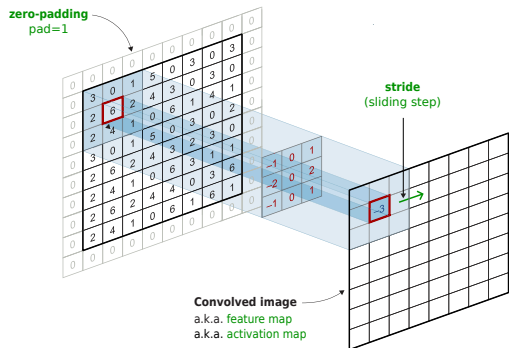
Modified after: Elgendy (2020)

- **filter kernel** = matrix of weights applied to extract features from the input
 ⇒ weights are learned by CNN during training!
 ⇒ hyperparameters to be set:
 - size (3×3 , 5×5 , ...)
 - depth (number of filters)
- **destination pixel** = weighted sum of pixel-values in the receptive field and the filter-weights
- **receptive field** = area of the image that the filter convolves

NB: strictly speaking, convolutional layers actually use cross-correlations, which are very similar to convolutions

1. **CONV** Convolutional layer

⇒ **convolution**: reminder of Lecture 03 (filtering)



⇒ The filter slides over the entire image

- **stride** = sliding step
 ⇒ *hyperparameter to be set*
 ⇒ *increasing stride decreases size of feature map*
- **zero-padding** = add zeros around the input image to keep output the same size
 ⇒ *hyperparameter to be set*
- **feature map** (a.k.a. **activation map**) = convolved image

1. **CONV** Convolutional layer

⇒ What's different with respect to Lecture 03?

- in CNNs, the filter weights are randomly initialized and the values are learned by the network
- in doing so, the network learns to extract useful features from the image

3.2. Convolutional layer

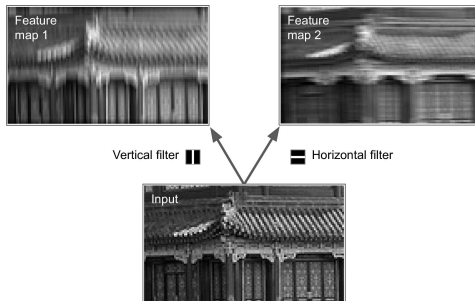
1. **CONV** Convolutional layer

⇒ What's different with respect to Lecture 03?

- in CNNs, the filter weights are randomly initialized and the values are learned by the network
- in doing so, the network learns to extract useful features from the image

⇒ Example: image modified by 2 possible filters

- vertical filter (7×7 matrix of 0s with 1s in central column) $\Rightarrow$ vertical lines get enhanced (since all inputs in the receptive field are multiplied by 0, except for those in the central vertical line)
- horizontal filter (7×7 matrix of 0s with 1s in central row) $\Rightarrow$ horizontal lines get enhanced



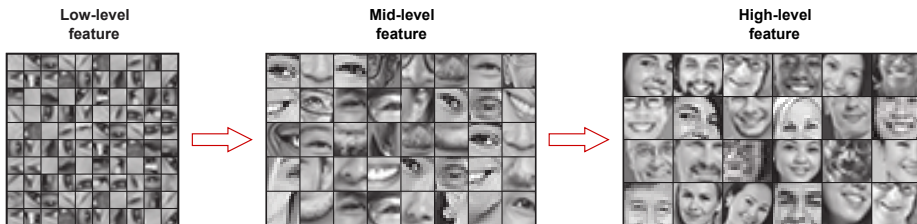
From: Géron (2022)

1. CONV Convolutional layer

⇒ stacking convolutional layers allows the network to learn progressively learn more complex features

EX: simplified version of how CNNs learn faces

- First layers ⇒ learn low-level features (lines, corners, circles, etc.)
- Intermediate layers ⇒ assemble low-level features into mid-level features (eyes, ears, etc.)
- Deep layers ⇒ learn high-level features (faces)

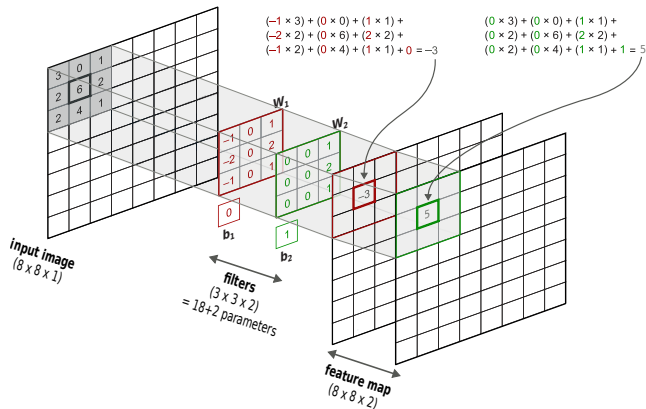


Modified after: Elgendy (2020)

1. **CONV** Convolutional layer

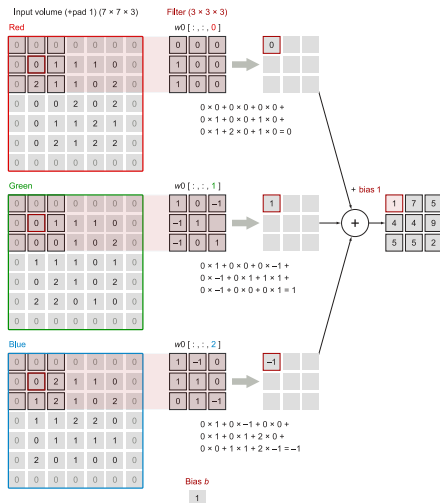
⇒ each convolutional layer usually has >1 filters!

NB: increasing the number of filters in a hidden convolutional layer of a CNN, is equivalent to increasing the number of neurons in a hidden fully-connected layer of a MLP (3×3 kernel = 9 neurons)



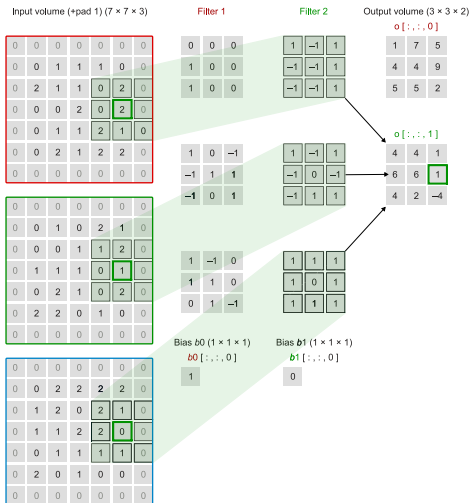
1. **CONV** Convolutional layer

⇒ what if we are handling 1 filter, but input image with > 1 channel (e.g. RGB)?



1. **CONV** Convolutional layer

⇒ what if we are handling > 1 filter with input image having > 1 channel (e.g. RGB)?



1. **CONV** Convolutional layer

⇒ the output size of the feature map is determined by the following formula:

$$\text{output size} = \frac{W - F + 2P}{S} + 1$$

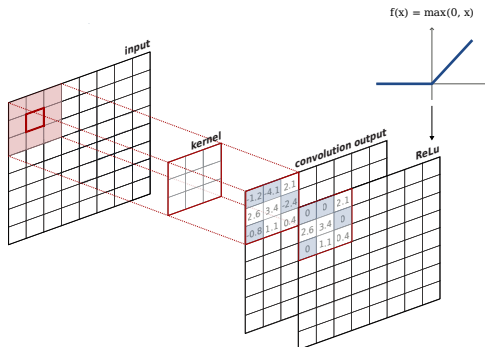
$$\text{where: } \begin{cases} W & \text{input size} \\ F & \text{filter size} \\ P & \text{padding} \\ S & \text{stride} \end{cases}$$

EX: input size $W=7 \times 7$, filter size $F=3 \times 3$, pad=0, stride $S=1 \Rightarrow$ output size = 5×5

3.2. Convolutional layer

1. **CONV** Convolutional layer → activation function

- ⇒ similar to what we saw for MLPs, **activation functions** are used to *introduce non-linearity in the network*
NB: a convolutional layer performs a linear operation, so stacking multiple convolutional layers without any activation functions would be equivalent to a single convolutional layer, unable to learn anything complex
- ⇒ most common activation function is **ReLU** (Rectified Linear Unit): $f(x) = \max(0, x)$
NB: ReLU is preferred because of its computational efficiency and its ability to avoid the vanishing gradient problem

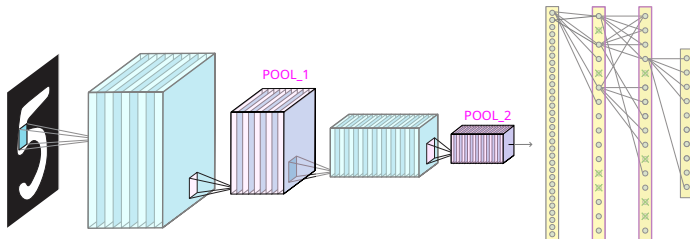


3.2. Convolutional layer

1. **CONV** Convolutional layer

⇒ final comments on the advantages of *convolutional layers* with respect to *fully-connected layers*:

- neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their receptive fields
⇒ *this allows the network to extract small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layers*
- all neurons in a feature map share the same parameters, which dramatically reduces the number of parameters in the model
⇒ *once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location. In contrast, once a fully connected neural network has learned to recognize a pattern in one location, it can recognize it in that particular location*

2. **POOL** Pooling layer (subsampling)

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(10, activation='softmax')
])

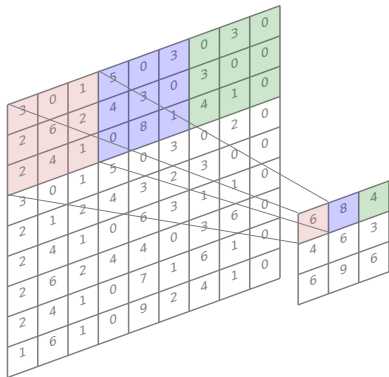
```

3.3. Pooling layer

2. **POOL** Pooling layer (subsampling)

- ⇒ increasing the number of convolutional layers increases the number of parameters to be learned
- ⇒ pooling layers are used to *reduce (subsample) the spatial dimensions of the feature map* while keeping the most important information. Types of pooling layers: max pooling and average pooling

EX: 3×3 max pooling filter with stride=3, reducing the feature map from 9×9 to 3×3

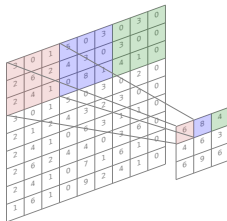


3.3. Pooling layer

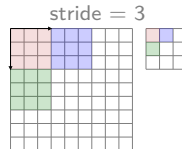
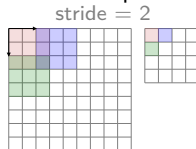
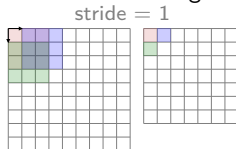
2. **POOL** Pooling layer (subsampling)

- ⇒ increasing the number of convolutional layers increases the number of parameters to be learned
- ⇒ pooling layers are used to *reduce (subsample) the spatial dimensions of the feature map* while keeping the most important information. Types of pooling layers: max pooling and average pooling

EX: 3×3 max pooling filter with stride=3, reducing the feature map from 9×9 to 3×3



- ⇒ changing the stride will change the size of the feature map



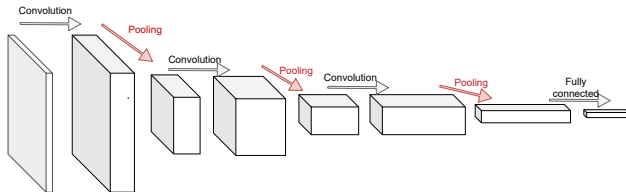
3.3. Pooling layer

2. **POOL** Pooling layer (subsampling)

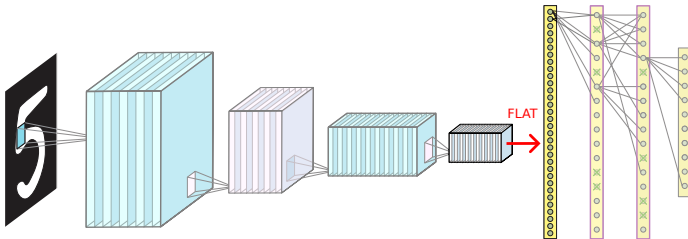
⇒ pooling layers reduce image resolution while keeping the image's important features
(think of it as an image-compressing program)



From: Elgendy (2020)



NB: stride during convolution also allow to reduce the size of the feature map; pooling operations are not strictly necessary and could be removed in favor of adjusting stride/padding in the convolutional layer (e.g. Valade et al. 2019, 2023).

3. **FLAT** Flatten layer

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

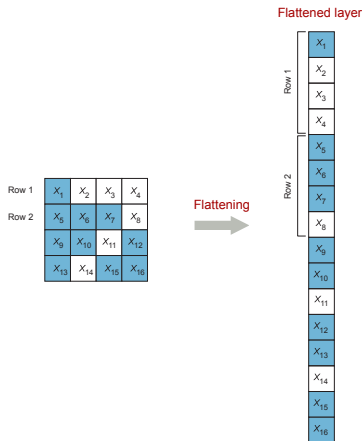
    Dense(10, activation='softmax')
])

```

3. **FLAT** Flatten layer

⇒ Flattening is used to convert an image matrix (2D) or tensor (n-2D) into a vector (1D)

NB: during the flattening process, the 2D information is entirely lost.



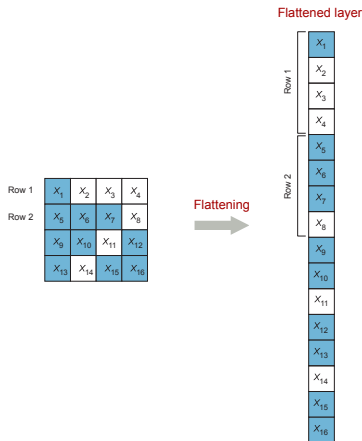
⇒ Where are the flatten layers positionned?

- in MLPs, the *input image* is flattened into a vector and parsed to the FC layers for classification
- in CNNs, the *feature maps* (learned in the convolutional layers CONV) are flattened, and the feature vector is fed to the FC layers for classification

3. **FLAT** Flatten layer

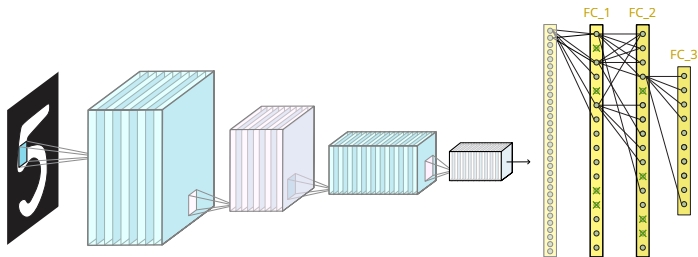
⇒ Flattening is used to convert an image matrix (2D) or tensor (n-2D) into a vector (1D)

NB: during the flattening process, the 2D information is entirely lost.



⇒ Where are the flatten layers positionned?

- in MLPs, the *input image* is flattened into a vector and parsed to the FC layers for classification
- in CNNs, the *feature maps* (learned in the convolutional layers CONV) are flattened, and the feature vector is fed to the FC layers for classification

4. **FC** Fully-Connected layer (a.k.a. **dense layer**)

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(10, activation='softmax')
])

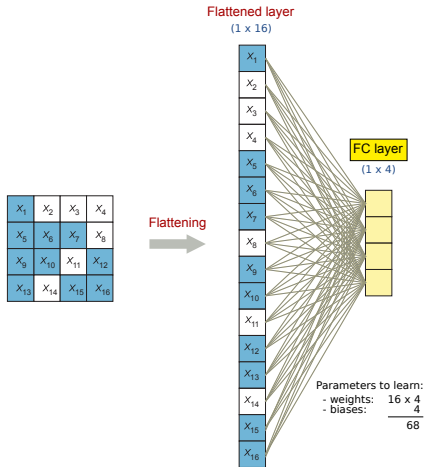
```

1)

3.5. Fully-connected layer

4. **FC** Fully-Connected layer (a.k.a. **dense** layer)

⇒ **fully-connected (dense) layers**: each neuron of the FC layer is connected to **all inputs** (i.e., all pixels of the flattened feature map)

Example:

⇒ input: flattened feature map of size (1×16)

⇒ output: dense layer with 4 neurons (1×4)

⇒ parameters to learn:

- weights: 16×4

- biases: 4

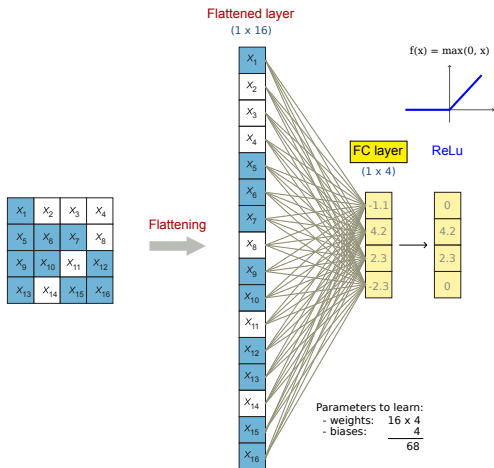
→ total number of parameters: $16 \times 4 + 4 = 68$

→ general formula: $N_{\text{params}} = n_{\text{in}} \times n_{\text{out}} + n_{\text{out}}$

3.5. Fully-connected layer

4. **FC** Fully-Connected layer (a.k.a. **dense** layer)

⇒ **fully-connected (dense) layers**: each neuron of the FC layer is connected to **all inputs** (i.e., all pixels of the flattened feature map)

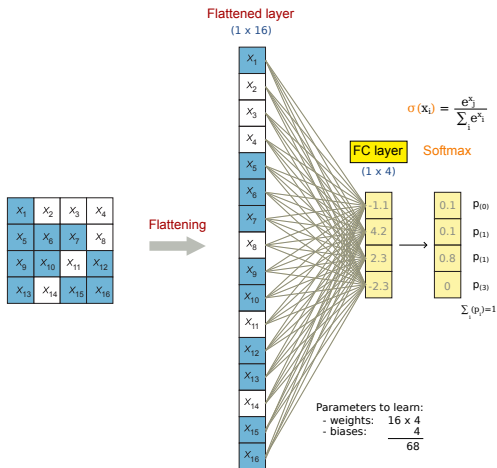
ReLU activation function

⇒ applied to the output of the FC layer to introduce non-linearity in the network

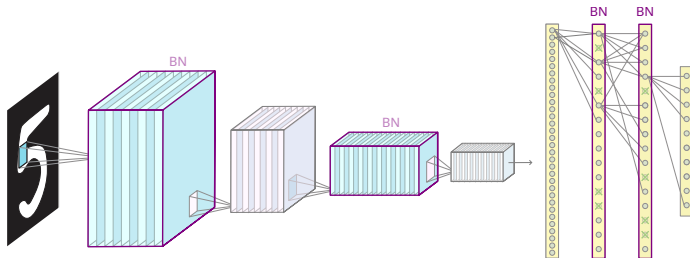
3.5. Fully-connected layer

4. **FC** Fully-Connected layer (a.k.a. **dense** layer)

⇒ **fully-connected (dense) layers**: each neuron of the FC layer is connected to **all inputs** (i.e., all pixels of the flattened feature map)

Softmax activation function

⇒ applied to the output of the FC layer to convert into probabilities

5. **BN** Batch normalization layer

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

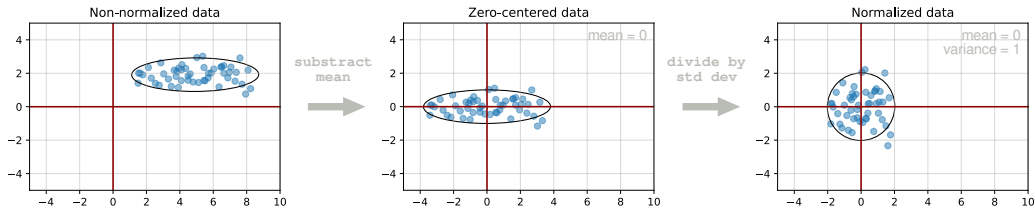
    Dense(10, activation='softmax')
])

```

5. **BN** Batch normalization layer

⇒ **Batch norm** applies a *normalization process* to the values in the hidden layers

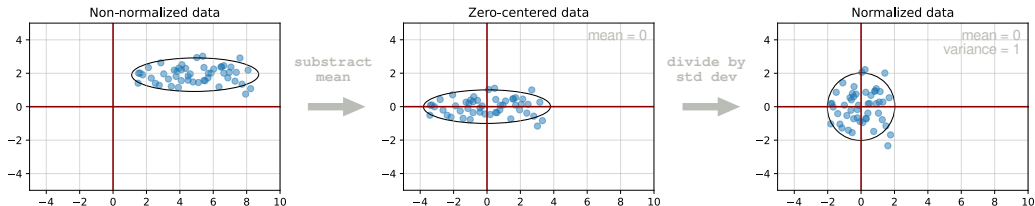
⇒ *normalization* consists in standardizing the distribution of the activations (i.e., $\text{mean}=0$, $\text{variance}=1$) by using the mean and variance of the current batch of data



***NB:** this normalization process is similar to what is done in the input layer as preprocessing step (see section on data preprocessing), but applied to the activations of the hidden layers instead of the input features*

5. **BN** Batch normalization layer

- ⇒ **Batch norm** applies a *normalization process* to the values in the hidden layers
- ⇒ *normalization* consists in standardizing the distribution of the activations (i.e., $\text{mean}=0$, $\text{variance}=1$) by using the mean and variance of the current batch of data

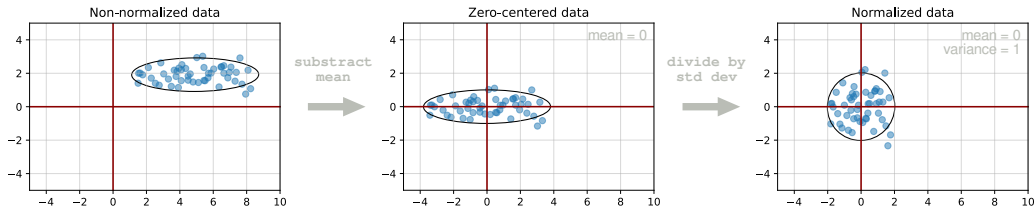


NB: this normalization process is similar to what is done in the input layer as preprocessing step (see section on data preprocessing), but applied to the activations of the hidden layers instead of the input features

- ⇒ this weakens the coupling of the learning process between earlier and later layers, allowing each layer of the network to learn more independently ⇒ stabilizes training and allows higher learning rates

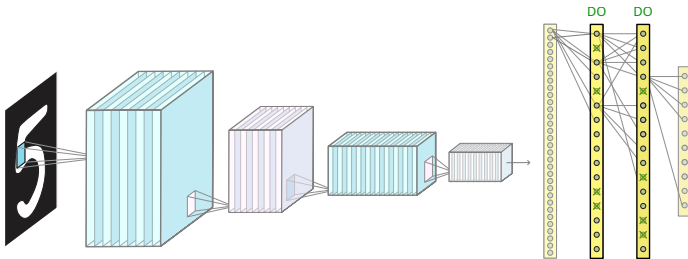
5. **BN** Batch normalization layer

- ⇒ **Batch norm** applies a *normalization process* to the values in the hidden layers
- ⇒ *normalization* consists in standardizing the distribution of the activations (i.e., $\text{mean}=0$, $\text{variance}=1$) by using the mean and variance of the current batch of data



NB: this normalization process is similar to what is done in the input layer as preprocessing step (see section on data preprocessing), but applied to the activations of the hidden layers instead of the input features

- ⇒ this weakens the coupling of the learning process between earlier and later layers, allowing each layer of the network to learn more independently ⇒ stabilizes training and allows higher learning rates
- ⇒ typical order: **CONV|FC** → **ReLU** → **BN**

6. **DO** Dropout layer (regularization)

```

model = Sequential([
    Conv2D(filters=10, kernel_size=(5, 5), strides=1, padding='same', activation='relu', input_shape=(28,28,1)),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Conv2D(filters=20, kernel_size=(5, 5), strides=1, padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),

    Flatten(),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(16, activation='relu'),
    BatchNormalization(),
    Dropout(rate=0.25),

    Dense(10, activation='softmax')
])

```

1)

6. **DO** Dropout layer (regularization)

⇒ Dropout is a popular regularization technique used to prevent overfitting of deep neural networks

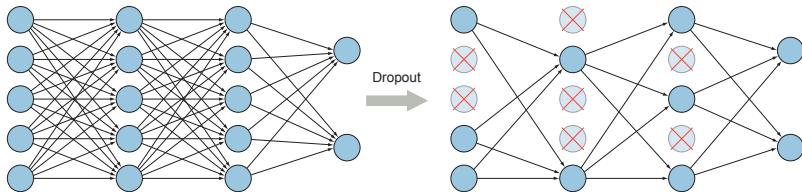
→ dropout layers are introduced between the fully connected layers (at the end of the network architecture)

6. **DO** Dropout layer (regularization)

⇒ **Dropout** is a popular regularization technique used to prevent overfitting of deep neural networks

→ dropout layers are introduced between the fully connected layers (at the end of the network architecture)

→ dropout randomly "turns off" a percentage of neurons (nodes) making up a layer of the network
⇒ these neurons are *not included* in the forward or backward pass



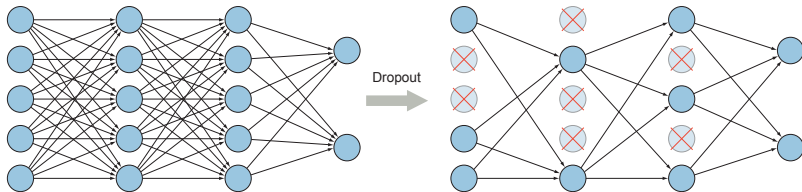
From: Elgendy (2020)

6. **DO** Dropout layer (regularization)

⇒ **Dropout** is a popular regularization technique used to prevent overfitting of deep neural networks

→ dropout layers are introduced between the fully connected layers (at the end of the network architecture)

→ dropout randomly “turns off” a percentage of neurons (nodes) making up a layer of the network
⇒ these neurons are *not included* in the forward or backward pass



From: Elgendy (2020)

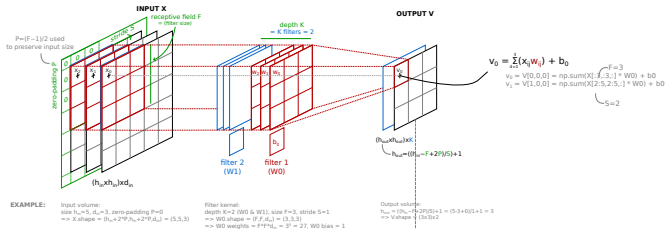
→ this forces all nodes to learn without relying specific nodes (since they can be dropped at any point)
⇒ spreads out the weights among all neurons (avoiding neurons to become too “strong” or too “weak”)
⇒ makes the network more resilient (less dependent on specific nodes)

3.8. Summary (cheat-sheet)

Summary (cheat-sheet)

Convolutional layer

- ⇒ 1 filter applies a convolution between filter-weights and pixel-values in the receptive field (multiply each pixel by corresponding weight and summing gives the center pixel value in new image)
- ⇒ convolution layer = K filters



activation function (ReLU)

- ⇒ applies threshold

Pooling layer (max-pooling)

- ⇒ reduces dimensionality (for memory issue)
- ⇒ enables network to see image at large scale

Hyperparameters

- receptive field (F)

- = filter size

NB: usually an odd number, so that it is centered on a central pixel

- depth (K)

- = number of filters

NB: depth column = set of neurons that are all looking at the same region of the input

- stride (S)

- = number of pixels the filter slides across the image at each step

EX: stride 2 $\Rightarrow$ filter moves 2 pix at a time $\Rightarrow$ produces smaller outputs

- zero-padding (P)

- = pad the input volume with zeros around the border

- dropout rate (p)

- = percentage of the input units to drop

NB: hyper-parameters control the output volume size:

width & height = $((W-F+2P)/S) + 1$ where W = input width/height
 depth = K

6. **RES** Residual block

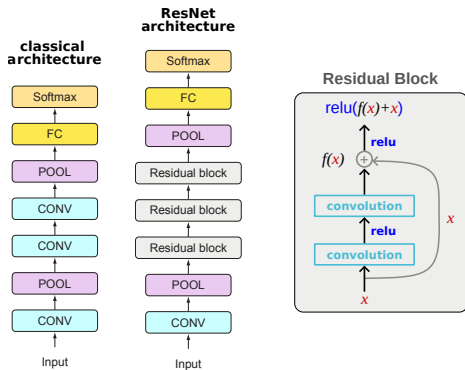
⇒ **Residual blocks** introduced to address the problem of vanishing gradients and allow for training very deep networks (i.e., ResNet has 152 layers: $8\times$ deeper than VGG-19 (19 layers), $7\times$ deeper than GoogLeNet (22 layers))

- **vanishing gradient problem**: as the network backpropagates the gradient of the error from the final layer back to the first layer, it is multiplied by the weight matrix at each step; thus the gradient can decrease exponentially quickly to zero, leading to a vanishing gradient phenomenon that prevents the earlier layers from learning.
- residual blocks introduce skip connections that allow the input to bypass one or more layers and be added to the output of those layers (He et al. 2015⁸, ResNet)

⁸ He et al. (2016) "Deep Residual Learning for Image Recognition", Proceedings of the IEEE 86, no. 11

6. **RES** Residual block

⇒ **Residual blocks** introduced to address the problem of vanishing gradients and allow for training very deep networks



Implementation of the residual block

```
X_shortcut = X
X = Conv2D(filters = F1, kernel_size = (3, 3), strides = (1,1))(X)
X = Activation('relu')(X)
X = Conv2D(filters = F1, kernel_size = (3, 3), strides = (1,1))(X)
X = Add()([X, X_shortcut])
X = Activation('relu')(X)
```

1. Introduction
2. How the brain recognizes images
3. CNN building blocs
- 4. Transfer learning**
 1. ImageNet & ILSVRC
 2. Famous CNN architectures
 3. Transfer learning using pretrained CNNs
5. Training workflow
6. Application

Data is key

- ⇒ the deeper the network, the more powerful it can be, but the more data it needs to be trained
- ⇒ **ImageNet dataset**: *large-scale image dataset* with 1.2 million images and 1,000 classes ([ImageNet](#))
 - Deng, J. et al. (2009) *ImageNet: A Large-Scale Hierarchical Image Database*. CVPR



Data is key

- ⇒ the deeper the network, the more powerful it can be, but the more data it needs to be trained
- ⇒ **ImageNet dataset**: *large-scale image dataset* with 1.2 million images and 1,000 classes ([ImageNet](#))
 - Deng, J. et al. (2009) *ImageNet: A Large-Scale Hierarchical Image Database*. CVPR



- ⇒ **ILSVRC competition**: “ImageNet Large Scale Visual Recognition Competition”
 - competition held between 2010-2017 using ImageNet as benchmark for image classification & segmentation tasks → several CNN architectures have been developed to win the competition

4.1. ImageNet & ILSVRC

ILSVRC competition

⇒ CNN networks having won the ILSVRC competition:

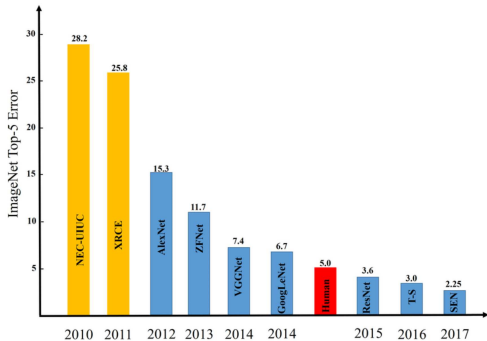


Image credit: Z. Alyafeai, L. Ghouti

4.1. ImageNet & ILSVRC

ILSVRC competition

- ⇒ CNN networks having won the ILSVRC competition:
- ⇒ performance comes with a computational cost!

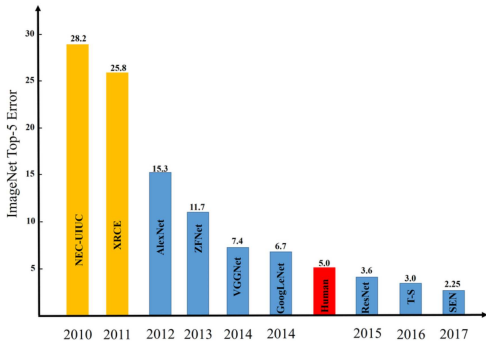
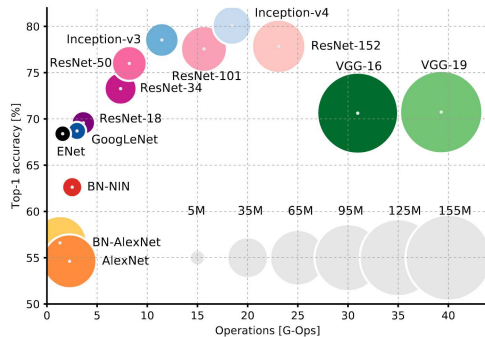


Image credit: Z. Alyafeai, L. Ghouti



From: Cansiani et al. (2017)

Most famous CNN architectures

⇒ most famous CNN networks achieving very good performances on the *ImageNet* dataset:

- LeNet-5 (1998)
- AlexNet (2012)
- GoogLeNet (2014)
- ResNet (2015)
- Xception (2016)
- SEnet (2017)

Most famous CNN architectures

⇒ most famous CNN networks achieving very good performances on the *ImageNet* dataset:

- LeNet-5 (1998)
- AlexNet (2012)
- GoogLeNet (2014)
- ResNet (2015)
- Xception (2016)
- SEnet (2017)

⇒ details on differences in architecture:

→ see e.g. “Mohamed Elgendy (2020) *Deep Learning for Vision Systems* (Manning Editions)”

→ see e.g. “Aurélien Géron (2022) *Hands-On Machine Learning* (O’Reilly Editions)”

Transfer learning using pretrained CNNs

- ⇒ desining and training your own network *from scratch* can be difficult (or impossible without enough data)
- training “from scratch” means the model starts with zero knowledge, i.e. with random initialization of weights

Transfer learning using pretrained CNNs

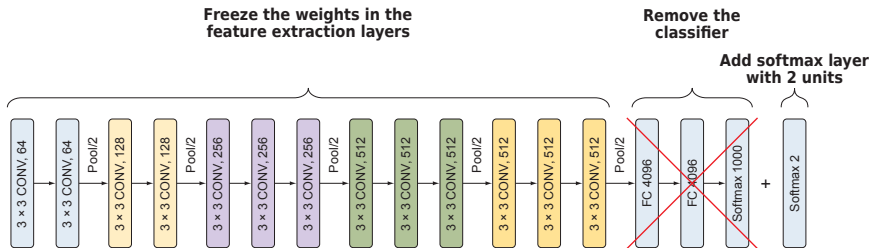
- ⇒ designing and training your own network *from scratch* can be difficult (or impossible without enough data)
 - training “from scratch” means the model starts with zero knowledge, i.e. with random initialization of weights

- ⇒ **transfer learning** allows to **fine-tune a pretrained model**
 - a pretrained model is a network that has been previously trained on a large dataset, typically on a large-scale image classification task
 - fine-tuning means starting from a pretrained model, then retraining parts of the model on a new dataset to adapt the model to the new task

4.3. Transfer learning using pretrained CNNs

Transfer learning using pretrained CNNs

- ⇒ designing and training your own network *from scratch* can be difficult (or impossible without enough data)
 - training “from scratch” means the model starts with zero knowledge, i.e. with random initialization of weights
- ⇒ **transfer learning** allows to **fine-tune a pretrained model**
 - a pretrained model is a network that has been previously trained on a large dataset, typically on a large-scale image classification task
 - fine-tuning means starting from a pretrained model, then retraining parts of the model on a new dataset to adapt the model to the new task
- ⇒ EX: suppose we want to train a model that classifies images in 2 categories (e.g. bananas and apples)
 - instead of collecting hundreds of thousands of images for each class, labeling them, and training a network from scratch, we can applying transfer learning to a VGG16 network



Modified after: Elgendy (2020)

1. Introduction
2. How the brain recognizes images
3. CNN building blocs
4. Transfer learning
- 5. Training workflow**
 1. Data preprocessing
 2. Data splitting
 3. Training control
 4. Overfitting vs underfitting
 5. Model hyperparameters & evaluation

6. Application

1. Preprocessing data

⇒ image resizing

NB: neural networks (MLP, CNN) require all images to be the same shape (e.g. 224×224 for VGG16, 299×299 for Xception)

⇒ image grayscaling (optional)

NB: color images add computational complexity (more parameters), so convert to grayscale if the problem does not require color (e.g., can the human eye identify the object in grayscale?)

⇒ image normalization

NB: normalize the pixel values to the $[0-1]$ range to boost learning performance and make the network converge faster

⇒ image augmentation (optional)

NB: create variants (flip, rotate, crop, color jitter) to increase dataset size and robustness

2. Splitting data for train/validation/test

⇒ Training ML model:

- **training dataset** (70%) ⇒ *used to train the model and update the weights (learning)*
- **test dataset** (30%) ⇒ *used to evaluate the model's performance on unseen data (generalization)*

⇒ Training DL model:

- **training dataset** (70%) ⇒ *used to train the model and update the weights (learning)*
- **validation dataset** (15%) ⇒ *used to tune hyperparameters and decide when to stop training (early stopping)*
- **test dataset** (15%) ⇒ *used to evaluate the model's performance on unseen data (generalization)*

Note: use *shuffle* before splitting to avoid ordering bias

2. Splitting data for train/validation/test

⇒ Training ML model:

- **training dataset** (70%) ⇒ *used to train the model and update the weights (learning)*
- **test dataset** (30%) ⇒ *used to evaluate the model's performance on unseen data (generalization)*

⇒ Training DL model:

- **training dataset** (70%) ⇒ *used to train the model and update the weights (learning)*
- **validation dataset** (15%) ⇒ *used to tune hyperparameters and decide when to stop training (early stopping)*
- **test dataset** (15%) ⇒ *used to evaluate the model's performance on unseen data (generalization)*

Note: use *shuffle* before splitting to avoid ordering bias

Pseudo code for model training:

```
for each epoch for each training data instance:  
- propagate error through the network and adjust the weights  
- calculate the accuracy and error over training data  
  
for each validation data instance:  
- calculate the accuracy and error over the validation data
```

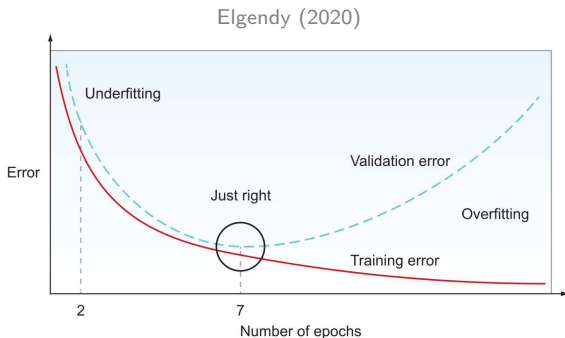
3. Epochs & early stopping

- ⇒ **batch size** = number of samples per update ⇒ *hyperparameter set to define how many samples are propagated through the network before updating the weights*
- ⇒ **epoch** = one full pass over the training set (training iteration)
⇒ *hyperparameter set to define how many iterations our network continues training*

$$\text{iterations per epoch} = \frac{N_{\text{train}}}{\text{batch size}}$$

3. Epochs & early stopping

- ⇒ **learning**: as model iterates over the training data, it learns features and patterns, and improves predictions based on these features (decreasing **training error**)
- ⇒ **overfitting**: occurs when the model starts to memorize the training data instead of learning generalizable features, leading to a decrease in performance on unseen data (validation set)
- ⇒ **early stopping**: stop training when **validation error** starts increasing



3. Epochs & early stopping

⇒ early stopping function in Keras:

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=20)
```

⇒ main parameters:

- **monitor**: metric to monitor during training (e.g. validation loss `val_loss`)

NB: monitor the validation loss (or accuracy) to detect overfitting, not the training loss (or accuracy) which will always improve with more epochs

- **min_delta**: minimum change to qualify as an improvement

NB: define `min_delta` according to the rate of change. The default value of 0 works pretty well in many cases.

- **patience**: number of epochs with no improvement after which training will be stopped

EX: `patience=1` means training will stop at the epoch where the error increases

4. Overfitting vs underfitting

⇒ overfitting: *training error* is good, *validation error* is bad

→ Fixes: *more data, augmentation, regularization, simpler model*

→ Regularization techniques:

- dropout: randomly disable neurons (see dropout layer)
- data augmentation: adds variability (see data preparation)
- L2 weight decay: penalize large weights (next lecture)

4. Overfitting vs underfitting

⇒ **overfitting**: *training error is good, validation error is bad*

→ Fixes: *more data, augmentation, regularization, simpler model*

→ Regularization techniques:

- dropout: randomly disable neurons (see dropout layer)
- data augmentation: adds variability (see data preparation)
- L2 weight decay: penalize large weights (next lecture)

⇒ **underfitting**: *training error is bad, validation error is bad*

→ Fixes: *bigger model, train longer, reduce regularization*

5. Model hyperparameters & evaluation (more details in next lecture)

⇒ Model hyperparameters:

- Number of hidden layers (depth of the network)
- Activation function (e.g. ReLU, sigmoid, softmax)
- Error/loss function (e.g. cross-entropy for classification)
- Optimizer (e.g. SGD, Adam)
- Learning rate (step size for weight updates, optimizer's input parameter)
- Batch size (number of samples per update)
- Epochs (number of training iterations)

⇒ Model evaluation:

- Accuracy (proportion of correct predictions)
- Precision (proportion of true positives among predicted positives)
- Recall (proportion of true positives among actual positives)
- F1-score (harmonic mean of precision and recall)

1. Introduction
2. How the brain recognizes images
3. CNN building blocs
4. Transfer learning
5. Training workflow
- 6. Application**
 1. from MLP to CNN
 2. using TensorBoard

Last week: **MLP** for MNIST-fashion dataset classification task

```
import tensorflow as tf

# Load data
fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# Preprocess data
X_train, X_test, X_valid = X_train/255.0, X_test/255.0, X_valid/255.0

# Build model (using the Sequential API)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(300, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()

# Compile model
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])

# Train model
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                  epochs=30, # nb of times X_train is seen
                  batch_size=32) # nb of images per training instance
print('training instances per epoch = {}'.format(X_train.shape[0] / 32))

# Plot training history
import pandas as pd
pd.DataFrame(history.history).plot()

# Evaluate model
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test accuracy:', test_acc)

# Predict
img = X_test[0,:]
img = (np.expand_dims(img,0)) # add image to a batch
y_proba = model.predict(img).round(2)
y_pred = np.argmax(model.predict(img), axis=-1)

plt.bar(range(10), y_proba[0])
plt.imshow(img[0,:,:], cmap='binary')
plt.title('class {} = {}'.format(y_pred, class_names[np.argmax(y_proba)]))
```

1.1 Load data

- training dataset
- validation dataset
- test dataset

1.2 Preprocess data

- scale pixel intensities to 0-1

2.1 Build model

- set layer type/order

2.2 Compile model

- set loss function
- set optimizer
- set metrics

3. Train model

- learn layer parameters (weights/biases)
- plot training history (check for overfitting)

4. Evaluate model

- evaluate accuracy on test dataset

5. Predict from model

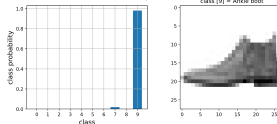
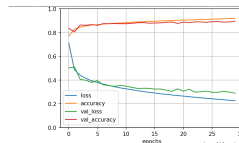
- predict image class using learned model



Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010

Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0



Last week: **MLP** for MNIST-fashion dataset classification task

```
import tensorflow as tf

# Load data
fashion_mnist = tf.keras.datasets.fashion_mnist

(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

# Preprocess data
X_train, X_test, X_valid = X_train/255.0, X_test/255.0, X_valid/255.0

# Build model (using the Sequential API)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(300, activation='relu'),
    tf.keras.layers.Dense(100, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
model.summary()

# Compile model
model.compile(loss='sparse_categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# Train model
```

1.1 Load data

- training dataset
- validation dataset
- test dataset

1.2 Preprocess data

- scale pixel intensities to 0-1

2.1 Build model

- set layer type/order

2.2 Compile model

- set loss function
- set optimizer
- set metrics



Model: "sequential"

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010

Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0

MLP are powerful, but break for large images due to the huge amount of parameters to optimize

EX1: *simple* model above on the *simple* MNIST-fashion dataset (28x28 pix) $\Rightarrow$ 266,610 parameters

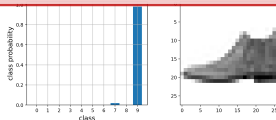
EX2: 100x100 image = 10,000 pixels, with first hidden layer having 1,000 neurons (which is already very restrictive)
 $\Rightarrow 10,000 \times 1,000 = 10$ million connections, only for the first layer!

```
# Predict
img = X_test[0,:,:]
img = (np.expand_dims(img,0)) # add image to a batch
y_proba = model.predict(img).round(2)
y_pred = np.argmax(model.predict(img), axis=-1)

plt.bar(range(10), y_proba[0])
plt.imshow(img[0,:,:], cmap='binary')
plt.title('class {} = {}'.format(y_pred, class_names[np.argmax(y_proba)]))
```

5. Predict from model

- predict image class using learned model

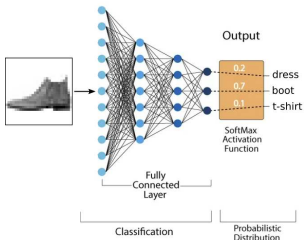


This week: CNN for MNIST-fashion dataset classification task

Build model (MLP)

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

Multi Layer Perceptron (MLP)

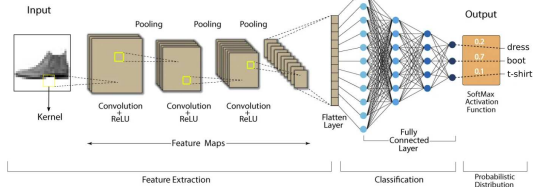


Build model (CNN)

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, 7, activation="relu", padding="same", input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

1)

Convolutional Neural Network (CNN)



This week: CNN for MNIST-fashion dataset classification task

```
# Build model (MLP)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

Model: "sequential_5"

Layer (type)	Output Shape	Param #
flatten_5 (Flatten)	(None, 784)	0
dense_5 (Dense)	(None, 300)	235500
dense_6 (Dense)	(None, 100)	30100
dense_7 (Dense)	(None, 10)	1010
Total params: 266,610		
Trainable params: 266,610		
Non-trainable params: 0		

```
# Build model (CNN)
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, 7, activation="relu", padding="same", input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    tf.keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(64, activation="relu"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

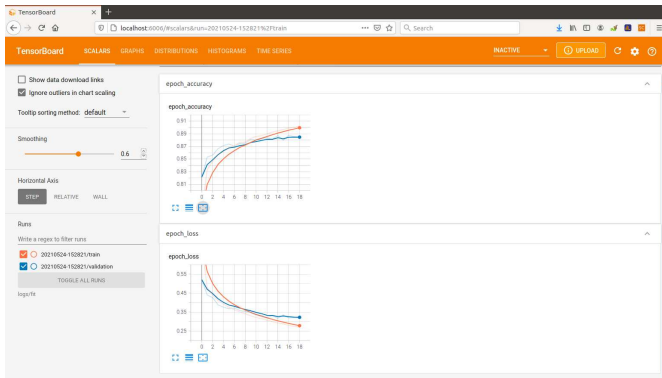
Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 64)	3200
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_2 (Conv2D)	(None, 14, 14, 128)	147584
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 128)	0
conv2d_3 (Conv2D)	(None, 7, 7, 256)	295168
conv2d_4 (Conv2D)	(None, 7, 7, 256)	590880
max_pooling2d_2 (MaxPooling2D)	(None, 3, 3, 256)	0
Flatten (Flatten)	(None, 2304)	0
dense (Dense)	(None, 128)	295040
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 10)	650
Total params: 1,413,834		
Trainable params: 1,413,834		
Non-trainable params: 0		

TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard provides the visualization and tooling needed for machine learning experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases
- etc.



TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard is installed during the TensorFlow conda installation

TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard is installed during the TensorFlow conda installation

⇒ To use it, you should:

1. Add the **tf.keras.callbacks.TensorBoard** callback to the Keras Model.fit() method *(ensures that logs are created and stored)*

```
# Create callback
import datetime
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

# Add callback to model.fit()
history = model.fit(X_train, y_train, callbacks=[tensorboard_callback])
```

TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard is installed during the TensorFlow conda installation

⇒ To use it, you should:

1. Add the **tf.keras.callbacks.TensorBoard** callback to the Keras Model.fit() method *(ensures that logs are created and stored)*

```
# Create callback
import datetime
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

# Add callback to model.fit()
history = model.fit(X_train, y_train, callbacks=[tensorboard_callback])
```

2. Run TensorBoard from command line

```
$ conda activate tf
$ cd <working dir>
$ tensorboard --logdir logs/fit    # set directory used to store logs
```

TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard is installed during the TensorFlow conda installation

⇒ To use it, you should:

1. Add the **tf.keras.callbacks.TensorBoard** callback to the Keras Model.fit() method *(ensures that logs are created and stored)*

```
# Create callback
import datetime
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

# Add callback to model.fit()
history = model.fit(X_train, y_train, callbacks=[tensorboard_callback])
```

2. Run TensorBoard from command line

```
$ conda activate tf
$ cd <working dir>
$ tensorboard --logdir logs/fit    # set directory used to store logs
```

3. Open a web-browser to the address

```
http://localhost:6006/
```

TensorBoard: TensorFlow's visualization toolkit

⇒ TensorBoard is installed during the TensorFlow conda installation

⇒ To use it, you should:

1. Add the **tf.keras.callbacks.TensorBoard** callback to the Keras Model.fit() method *(ensures that logs are created and stored)*

```
# Create callback
import datetime
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

# Add callback to model.fit()
history = model.fit(X_train, y_train, callbacks=[tensorboard_callback])
```

2. Run TensorBoard from command line

```
$ conda activate tf
$ cd <working dir>
$ tensorboard --logdir logs/fit    # set directory used to store logs
```

3. Open a web-browser to the address

```
http://localhost:6006/
```

Nota Bene: you can open it directly from a Jupyter cell (after training has finished however) as follows:

```
%load_ext tensorboard    # Load the TensorBoard notebook extension
%tensorboard --logdir logs # Open TensorBoard in cell
```