Lecture 12
Deep Learning 02
Convolutional Neural Networks

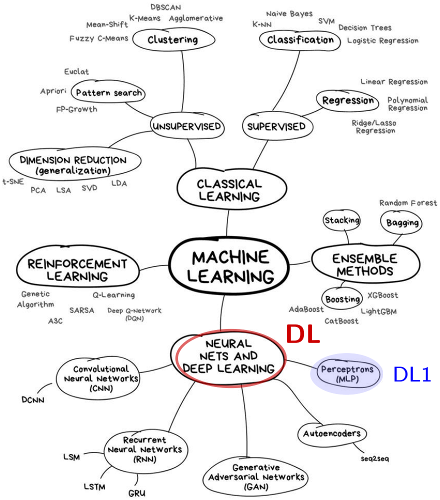
2024-11-19

Sébastien Valade

Universidad Nacional Autónoma de México

Introduction

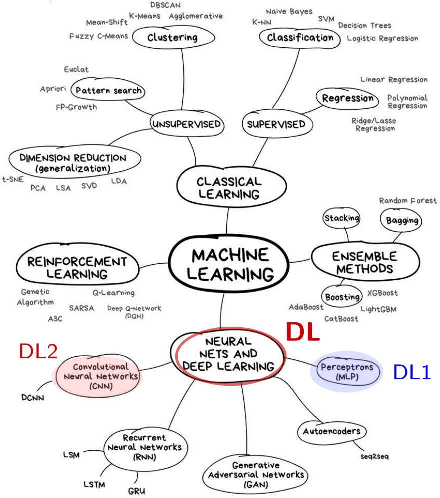
Last week: Neural Networks Part-1 *(multilayer perceptrons MLP)* - DL1



source

Introduction

Last week: Neural Networks Part-1 *(multilayer perceptrons MLP)* - DL1
This week: Neural Networks Part-2 *(convolutional neural networks CNN)* - DL2

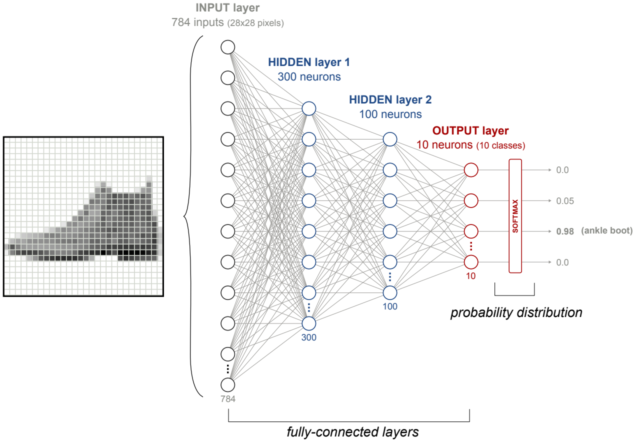


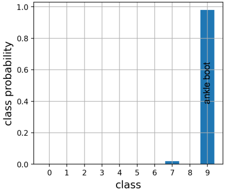
source

Introduction

Last lecture

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





Introduction

Last lecture

$\Rightarrow$ 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)

Introduction

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)

Introduction

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?

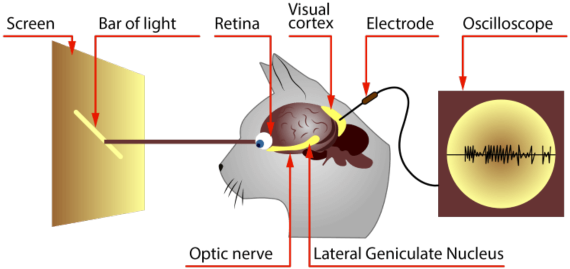
would rather need sparse connections: fewer weights, nearby regions related, & far apart regions not related

This lecture: from MLPs to CNNs

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 1958[1],1959[2],1968[3])*
NB: the authors received the Nobel Prize in Physiology or Medicine in 1981 for their work



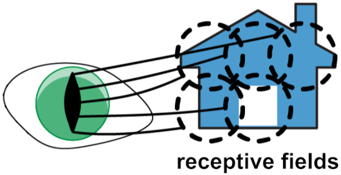
source

[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*:

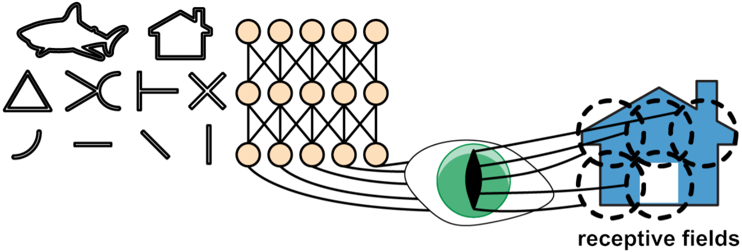
1. biological neurons respond to specific patterns in regions (a.k.a. **receptive fields**) of the visual field



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, ...)

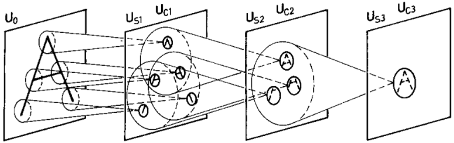


from: Géron A. (2022)

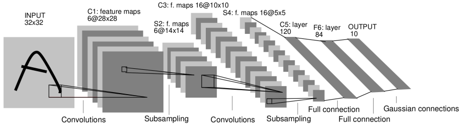
Reproducing brain perception with neural networks

⇒ These studies of the visual cortex inspired the **neocognitron** *(Fukushima 1980*[4]*)*, which gradually evolved into what we now call **convolutional neural networks CNNs**, a.k.a. ConvNets, (LeCun et al. 1998[5])

Fukushima 1980



LeCun et al. 1998

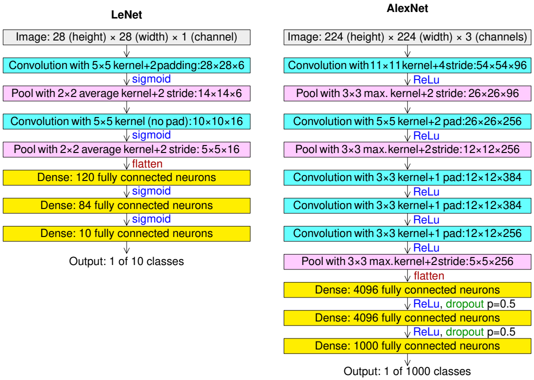


[4] Fukushima, K. (1980) "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", Biological Cybernetics

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

Building blocs (overview) of a CNN

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



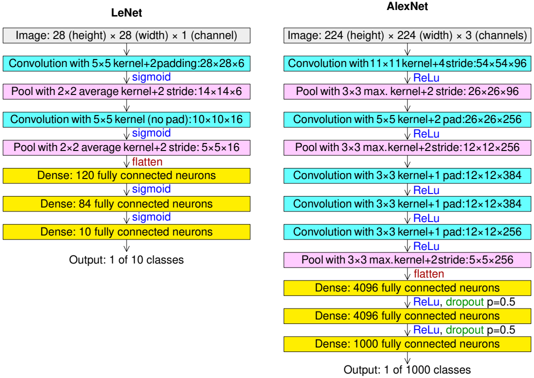
source: wiki

[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 blocs (overview) of a CNN

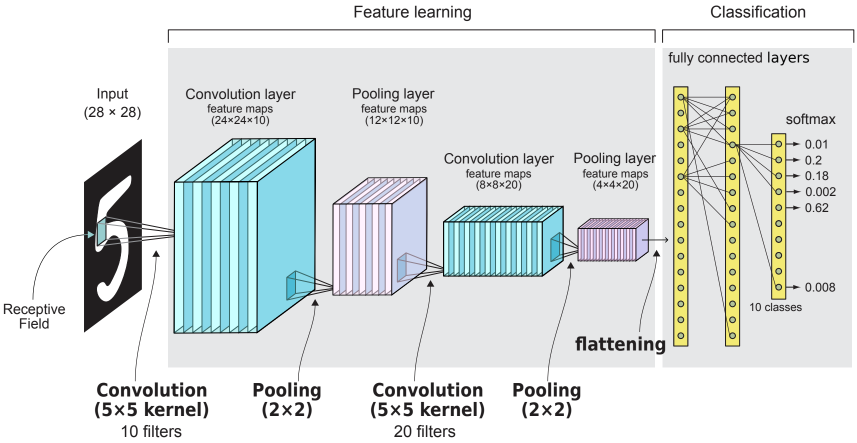
⇒ consider 2 milestones CNN architectures: **LeNet-5** *(LeCun et al. 1998)* and **AlexNet** *(Krizhevsky et al. 2012)*



- convolutional layers ⇒ extract features
  → *receptive field, filter kernel/depth, feature maps, padding, stride*
- pooling layers ⇒ downsample
  → *pooling kernel, stride*
- dense layers ⇒ classify
  → *fully connected layers*
- activation function ⇒ achieve non-linearity
  → *sigmoid, ReLu, ...*
- flatten layer ⇒ matrix/tensor to vector
- dropout layer ⇒ prevent overfitting (regularization)
  → *in the fully connected layers*

Building blocs (overview) of a CNN

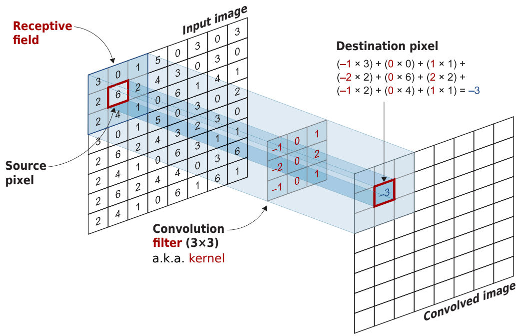
⇒ illustration of the building blocs of convolutional networks



Modified after: Elgendy (2020)

CONV Convolutional layer

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



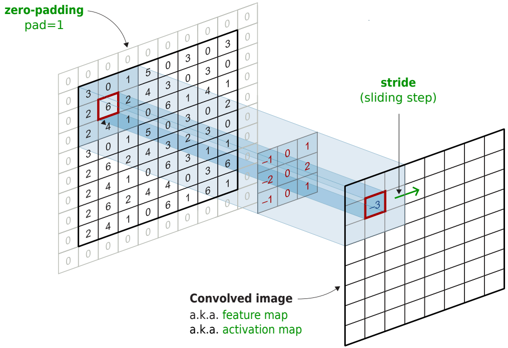
Modified after: Elgendy (2020)

- **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

CONV Convolutional layer

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



⇒ The filter slides over the entire image

- **stride** = sliding step
  ⇒ *hyperparameter to be set*
- **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

CONV Convolutional layer

$\Rightarrow$ What's different with respect to Lecture 03?

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

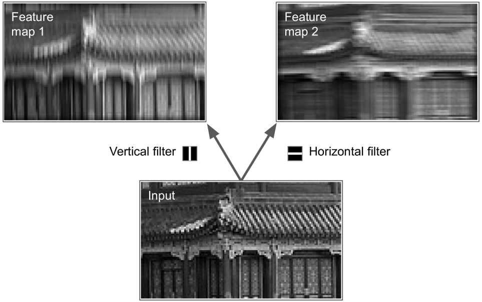
CONV Convolutional layer

⇒ 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) ⇒ 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) ⇒ horizontal lines get enhanced



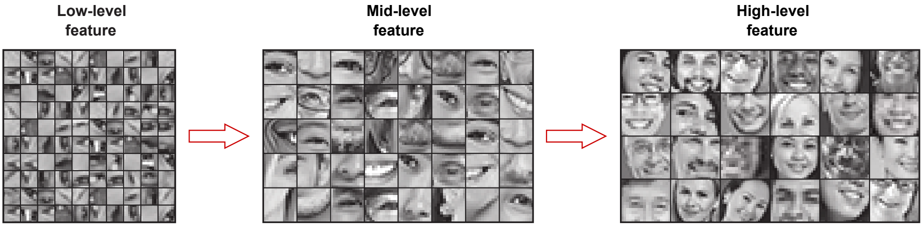
From: Géron (2022)

CONV Convolutional layer

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

EX: simplified version of how CNNs learn faces

- → *1st layer learns basic features (lines & edges)*
- → *2nd layer assembles those into recognizable shapes (corners & circles)*
- → *deeper layers learn more complex shapes such (eyes, ears, etc.)*

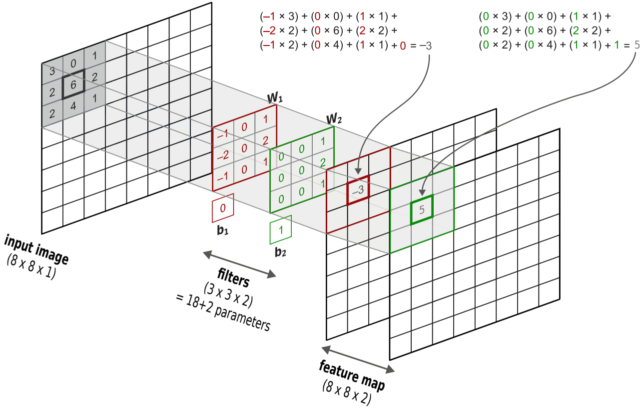


Modified after: Elgendy (2020)

CONV Convolutional layer

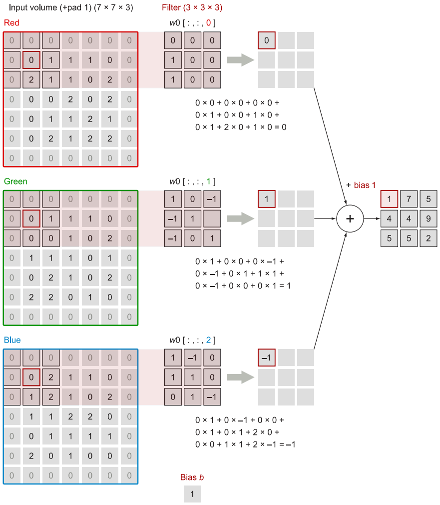
⇒ each convolutional layer usually has >1 filters!

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 (3x3 kernel = 9 neurons)



CONV Convolutional layer

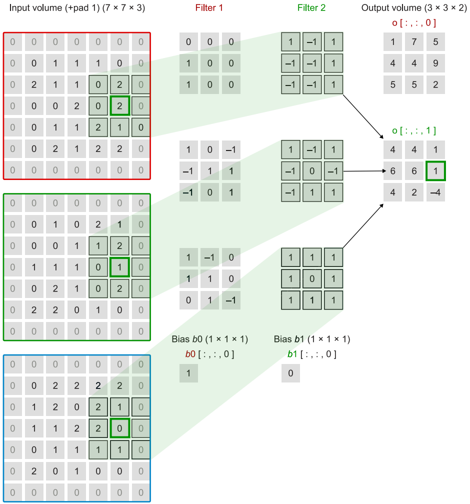
$\Rightarrow$ what if we are handling 1 filter, but input image with $>$ 1 channel (e.g. RGB)?



Modified after: Elgendy (2020)

CONV Convolutional layer

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



Modified after: Elgendy (2020)
See animation at: stanford.edu

CONV Convolutional layer

$\Rightarrow$ the output size of the feature map is determined by the following formula:

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

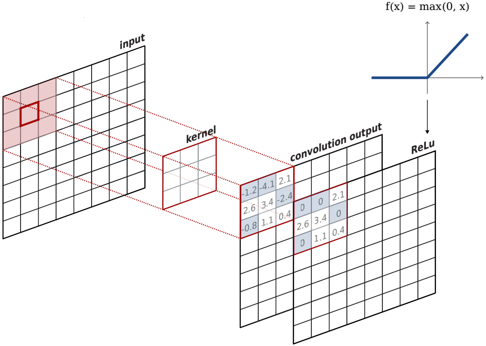
$$\text{where: } \begin{cases} W & \text{input size} \\ F & \text{filter size} \\ P & \text{padding} \\ S & \text{stride} \end{cases}$$

EX: input size W=7x7, filter size F=3x3, pad=0, stride S=1 $\Rightarrow$ output size = 5x5

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



Convolutional layer

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

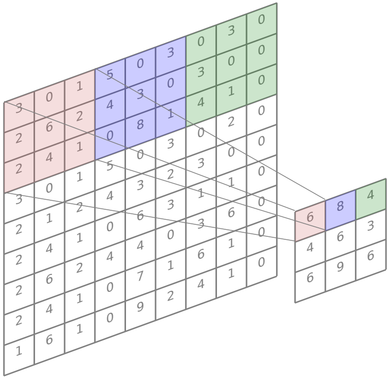
- neurons in the first convolutional layer are *not connected* to every single pixel in the input image, but *only to pixels in their receptive fields*
  ⇒ *this allows the network to extract small low-level features in the first hidden layer, then assemble them into larger higher-level features in the 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*

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

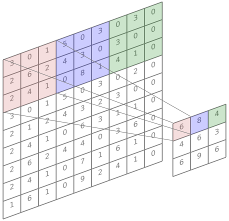


POOL **Pooling layer** (subsampling)

⇒ increasing the number of convolutional layers increases the number of parameters to be learned

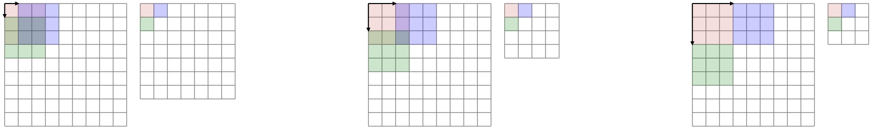
⇒ **pooling layers** are used to *reduce (subsample) the spatial dimensions of the feature map* while keeping the most important information. Types of pooling layers: **max pooling** and **average pooling**

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

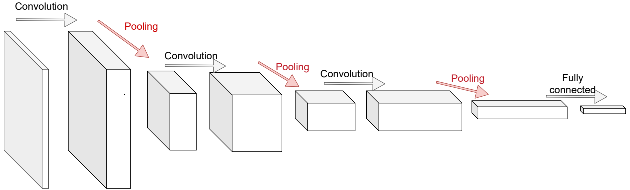
stride = 1   stride = 2   stride = 3



POOL **Pooling layer** (subsampling)

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

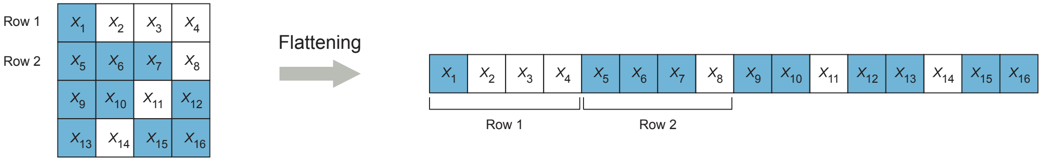
From: Elgendy (2020)



NB: stride during convolution also allow to reduce the size of the feature map; many authors have suggested that pooling operations could be removed in favor of adjusting stride/padding in the convolutional layer.

Flatten layer

⇒ Flattening is used to convert an image matrix (2D) or tensor (n·2D) into a vector (1D)
NB: during the flatenning process, the 2D information is entirely lost.



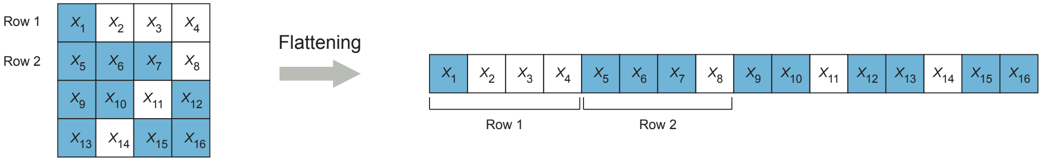
Modified after: Elgendy (2020)

⇒ Where are the flatten layers?

→ 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

Flatten layer

⇒ Flattening is used to convert an image matrix (2D) or tensor (n·2D) into a vector (1D)
NB: during the flatenning process, the 2D information is entirely lost.



Modified after: Elgendy (2020)

⇒ Where are the flatten layers?

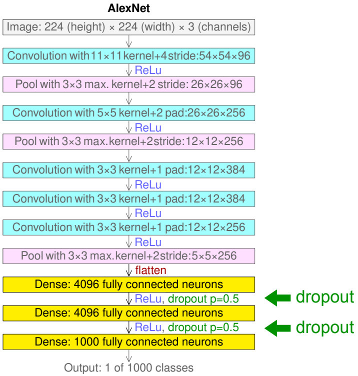
- → in MLPs, the *input image* is flattened into a vector and parsed to the FC layers for classification
- → in CNNs, the *feature maps* (learned in the convolutional layers CONV) are flattened, and the feature vector is fed to the FC layers for classification

3.5. Dropout layer

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)*

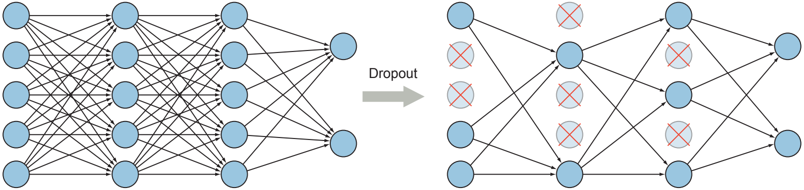


Dropout layer (regularization)

⇒ **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



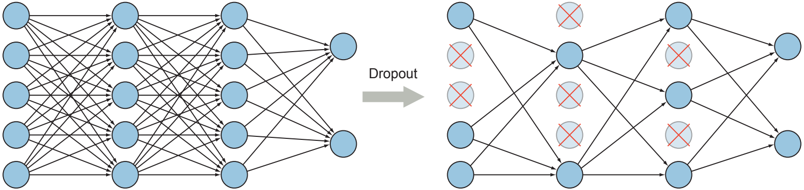
From: Elgendy (2020)

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)

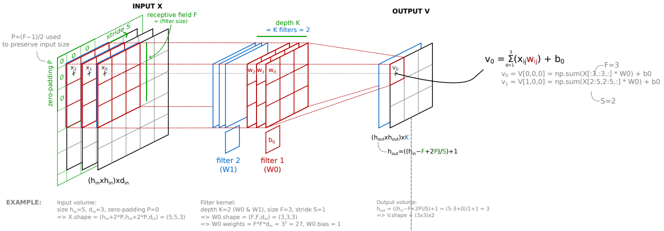
→ *this forces all nodes to learn without relying specific nodes* (since they can be dropped at any point)
⇒ spreads out the weights among all neurons (avoiding neurons to become too "strong" or too "weak")
⇒ makes the network more resilient (less dependent on specific nodes)

Summary (cheat-sheet)

Convolutional layer

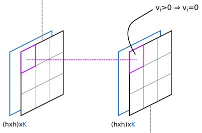
⇨ 1 filter applies a convolution between filter-weights and pixel-values in the receptive field (multiply each pixel by corresponding weight and summing gives the center pixel value in new image)

⇨ convolution layer = K filters



activation function (ReLu)

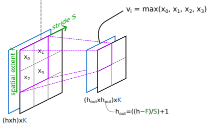
⇨ applies threshold



Pooling layer (max-pooling)

⇨ reduces dimentionality (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 => filter moves 2 pix at a time => 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

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"
→ *compitition held between 2010-2017 using ImageNet as benchmark for image classification & segmentation tasks → several CNN architectures have been developed to win the competition*

ILSVRC competition

⇒ CNN networks having won the ILSVRC competition:

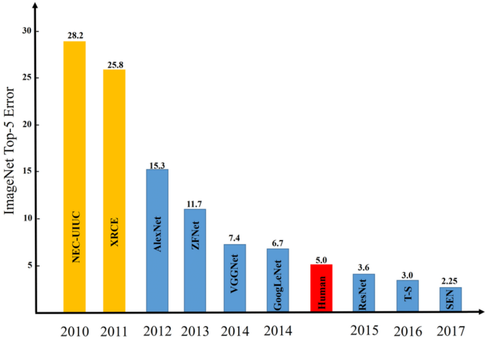


Image credit: Z. Alyafeai, L. Ghouti

ILSVRC competition

⇒ CNN networks having won the ILSVRC competition:

⇒ performance comes with a computational cost!

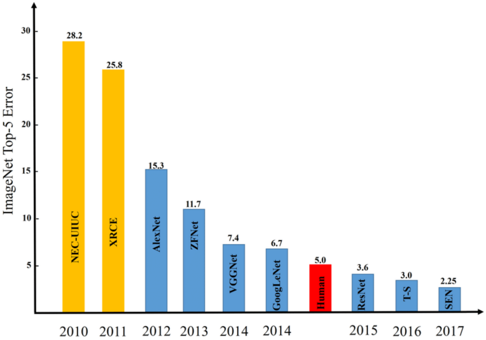
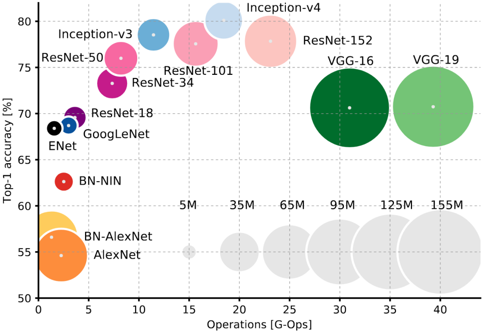


Image credit: Z. Alyafeai, L. Ghouti



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

$\Rightarrow$ 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)

$\Rightarrow$ explanation of the differences in architectures is beyond the scope of this lecture

$\rightarrow$ *see e.g. "Mohamed Elgendy (2020) Deep Learning for Vision Systems (Manning Editions)"*

$\rightarrow$ *see e.g. "Aurélien Géron (2022) Hands-On Machine Learning (O'Reilly Editions)"*

Transfer learning using pretrained CNNs

$\Rightarrow$ desining and training your own network *from scratch* can be difficult (or impossible without enough data)

$\rightarrow$ *training "from scratch" means the model starts with zero knowledge, i.e. with random initialization of weights*

Transfer learning using pretrained CNNs

$\Rightarrow$ desining and training your own network *from scratch* can be difficult (or impossible without enough data)

$\rightarrow$ *training "from scratch" means the model starts with zero knowledge, i.e. with random initialization of weights*

$\Rightarrow$ **transfer learning** allows to **fine-tune a pretrained model**

$\rightarrow$ *a pretrained model is a network that has been previously trained on a large dataset, typically on a large-scale image classification task*

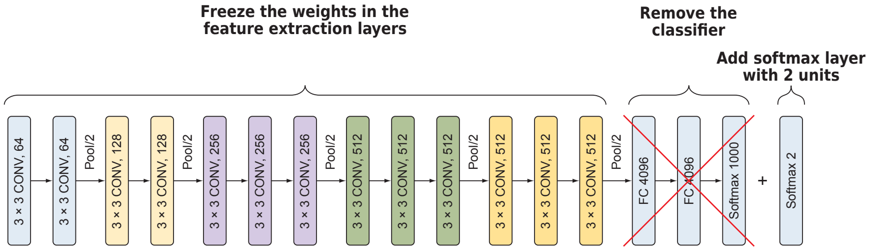
$\rightarrow$ *fine-tuning means starting from a pretrained model, then retraining parts of the model on a new dataset to adapt the model to the new task*

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** 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)

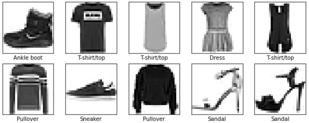
Last week: MLP for MNIST-fashion dataset classification task

```
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:]
```

1.1 Load data
- training dataset
- validation dataset
- test dataset

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

1.2 Preprocess data
- scale pixel intensities to 0–1




```
# 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()
```

2.1 Build model
- set layer type/order

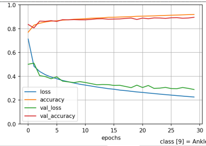
```
Model: "sequential"

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
```

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

2.2 Compile model
- set loss function
- set optimizer
- set metrics

```
# Train model
history = model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
                    epochs=30, # nb of times X_train is seen 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()
```

3. Train model
- learn layer parameters (weights/biases)
- plot training history (check for overfitting)

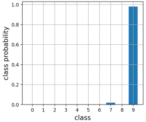


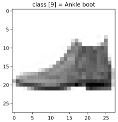

```
# Evaluate model
test_loss, test_acc = model.evaluate(X_test,  y_test)
print('Test accuracy:', test_acc)
```

4. Evaluate model
- evaluate accuracy on test dataset

```
# 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





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 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

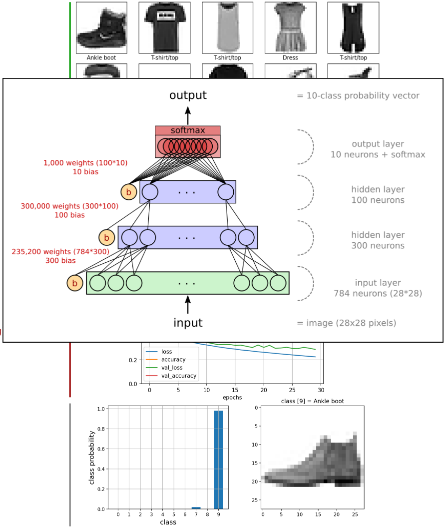
2.1 Build model
- set layer type/order

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



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 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()
```

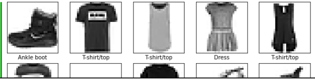
1.1 Load data
- training dataset
- validation dataset
- test dataset

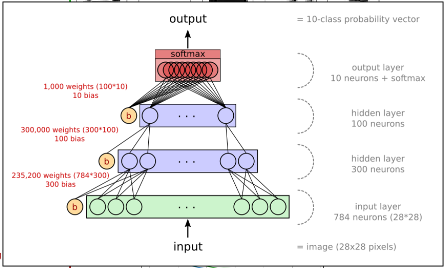
1.2 Preprocess data
- scale pixel intensities to 0–1

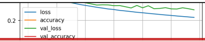
2.1 Build model
- set layer type/order

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







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 x 1,000 = 10 million connections, only for the first layer!

```
plt.title('class {} = {}'.format(y_pred, class_names[np.argmax(y_proba)]))
```

5.1. from MLP to CNN

This week: **CNN** for MNIST-fashion dataset classification task

```python
# 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
_________________________________________________________________
```

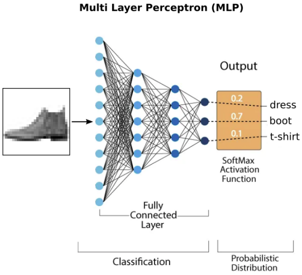
```python
# Build model (CNN)
model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, 7, activation="relu", padding="same", input_shape=[28, 28, 1]),
    tf.keras.layers.MaxPooling2D(2),
    tf.keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    tf.keras.layers.Conv2D(128, 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 (MaxPooling2 (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 256)         295168
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 7, 7, 256)         590080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (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
```

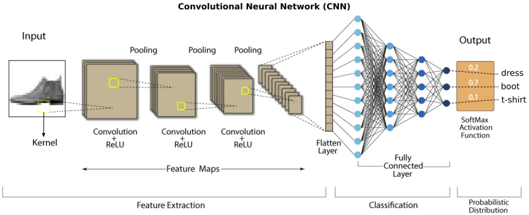
5.1. from MLP to CNN

This week: **CNN** for MNIST-fashion dataset classification task

```python
# 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")
])
```




```python
# 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")
])
```

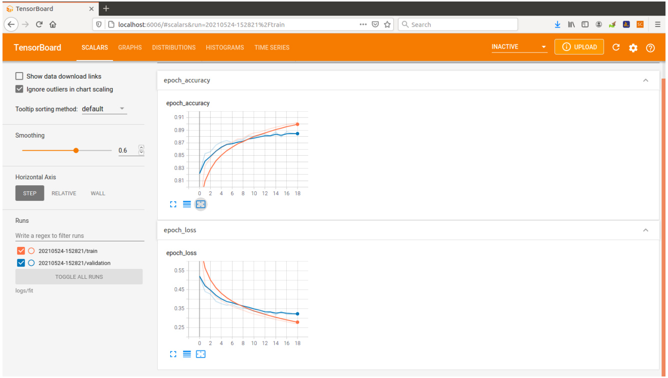


Application

[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.



5.2. using TensorBoard

TensorBoard: TensorFlow's visualization toolkit

$\Rightarrow$ TensorBoard is installed during the TensorFlow conda installation

5.2. using TensorBoard

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)*

```python
# 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])
```

5.2. using TensorBoard

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
```

5.2. using TensorBoard

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
```