# COMPUTER VISION
## MEI/1

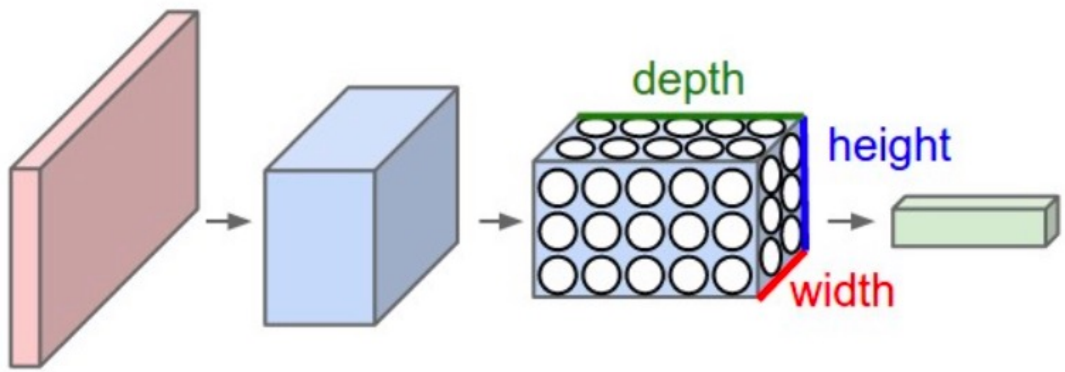
University of Beira Interior, Department of Informatics

Hugo Pedro Proença

hugomcp@di.ubi.pt, **2024/25**

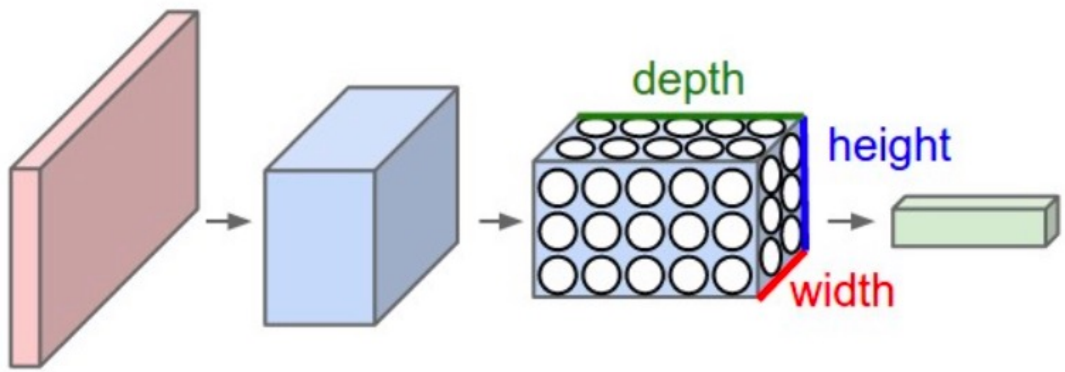
# Convolutional Neural Networks (CNNs)

- CNNs are a type of Neural Networks that have been augmenting their popularity in most tasks related to Computer Vision
  - E.g., Image Segmentation, Classification.
- The property of **shift invariance** gives them the **biological inspiration** of the human visual system and keeps the number of weights relatively small, making learning a feasible task.
- In opposition to traditional Feed-forward nets, neurons in CNNs are arranged in **three dimensions**.



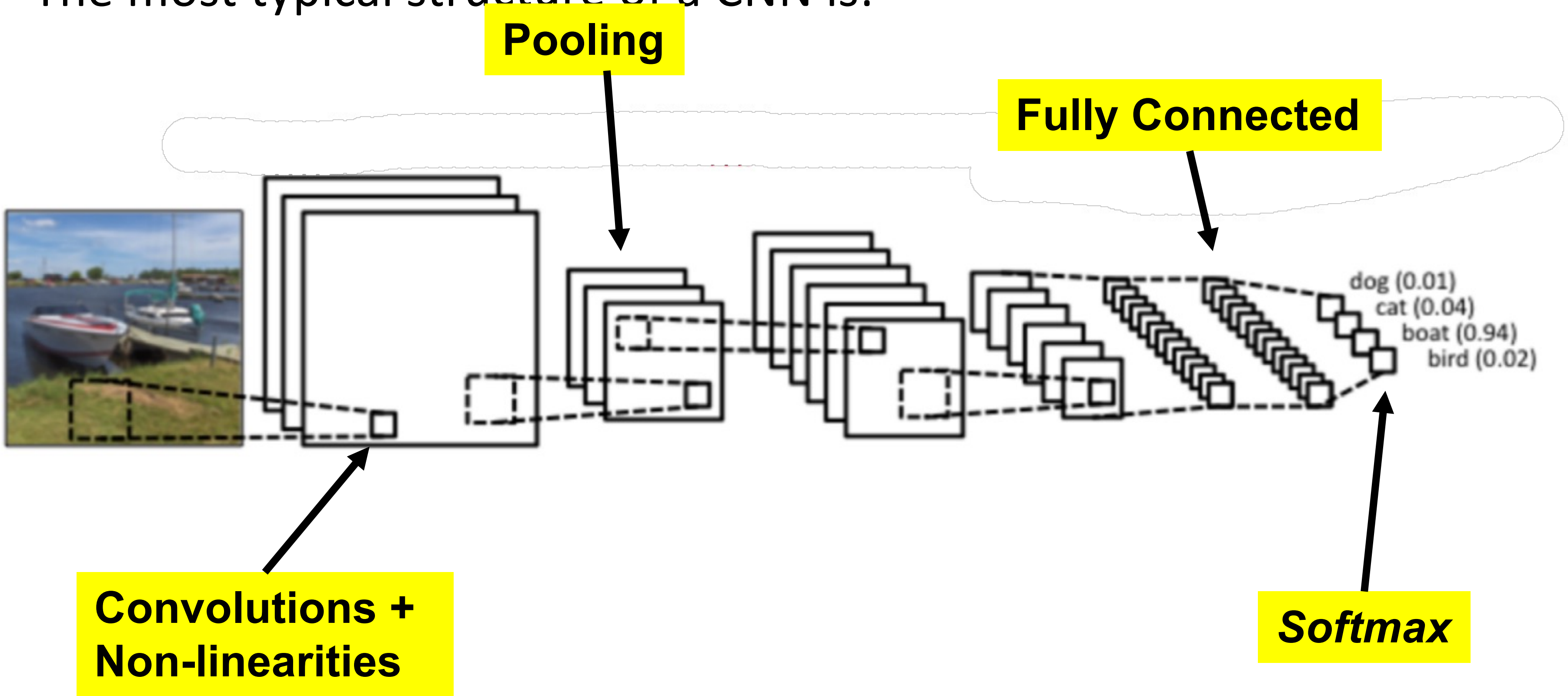
# Convolutional Neural Networks (CNNs)

- Each layer of a CNN transforms a 3D input into a 3D output.
- This pioneering work in CNNs was due to Yann LeCun (LeNet5) after many previous successful iterations since 1988.
- Initially, the **LeNet** architecture was used mainly for character recognition tasks such as reading zip codes, digits...
- The efficacy of CNNs in visual tasks is the main reason behind the popularity of deep learning. They are powering major advances in computer vision, with  applications for robotics, security and medical diagnosis.



# Convolutional Neural Networks (CNNs)

- The most typical structure of a CNN is:




These operations are the basic building blocks of *most CNNs*, so understanding how these work is an important step to understand the functioning of these powerful models.

# Convolutional Neural Networks (CNNs)

- **Convolution**
  - This block computes the convolution between an input map **x** with a bank of k multi-dimensional filters **f,** to obtain the results **y.**

$$\mathbf{x} \in \mathbb{R}^{H \times W \times D}, \quad \mathbf{f} \in \mathbb{R}^{H' \times W' \times D \times D''}, \quad \mathbf{y} \in \mathbb{R}^{H'' \times W'' \times D''}.$$

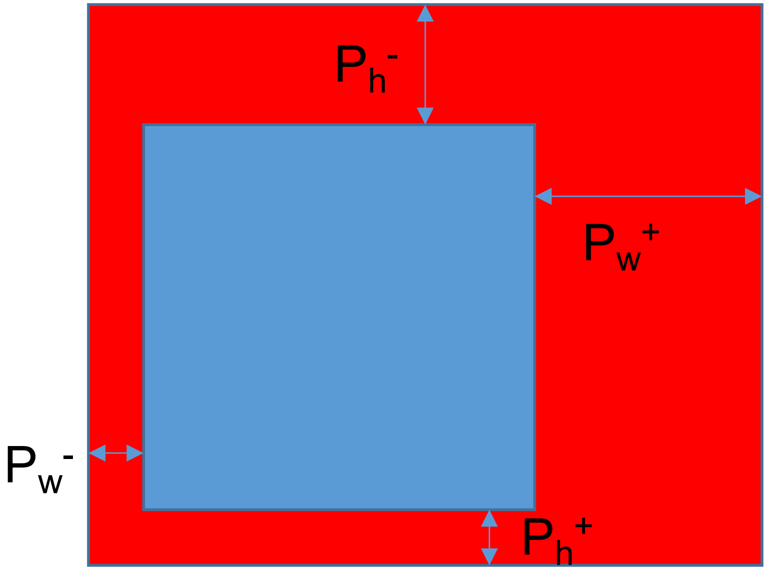
- Formally, the outputs **y** are given by:

$$y_{i''j''d''} = b_{d''} + \sum_{i'=1}^{H'} \sum_{j'=1}^{W'} \sum_{d'=1}^{D} f_{i'j'd} \times x_{i''+i'-1,j''+j'-1,d',d''}.$$

# Convolutional Neural Networks (CNNs)

- Convolution (padding and stride)
  - Usually it is possible to specify top, bottom, left, right paddings ($P_h^-$, $P_h^+$, $P_w^-$, $P_w^+$) of the input array and subsampling strides ($S_h$,$S_w$) of the output array.

$$y_{i''j''d''} = b_{d''} + \sum_{i'=1}^{H'} \sum_{j'=1}^{W'} \sum_{d'=1}^{D} f_{i'j'd} \times x_{S_h(i''-1)+i'-P_h^-, S_w(j''-1)+j'-P_w^-, d', d''}.$$




The output size is given by:

$$H'' = 1 + \left\lfloor \frac{H - H' + P_h^- + P_h^+}{S_h} \right\rfloor.$$

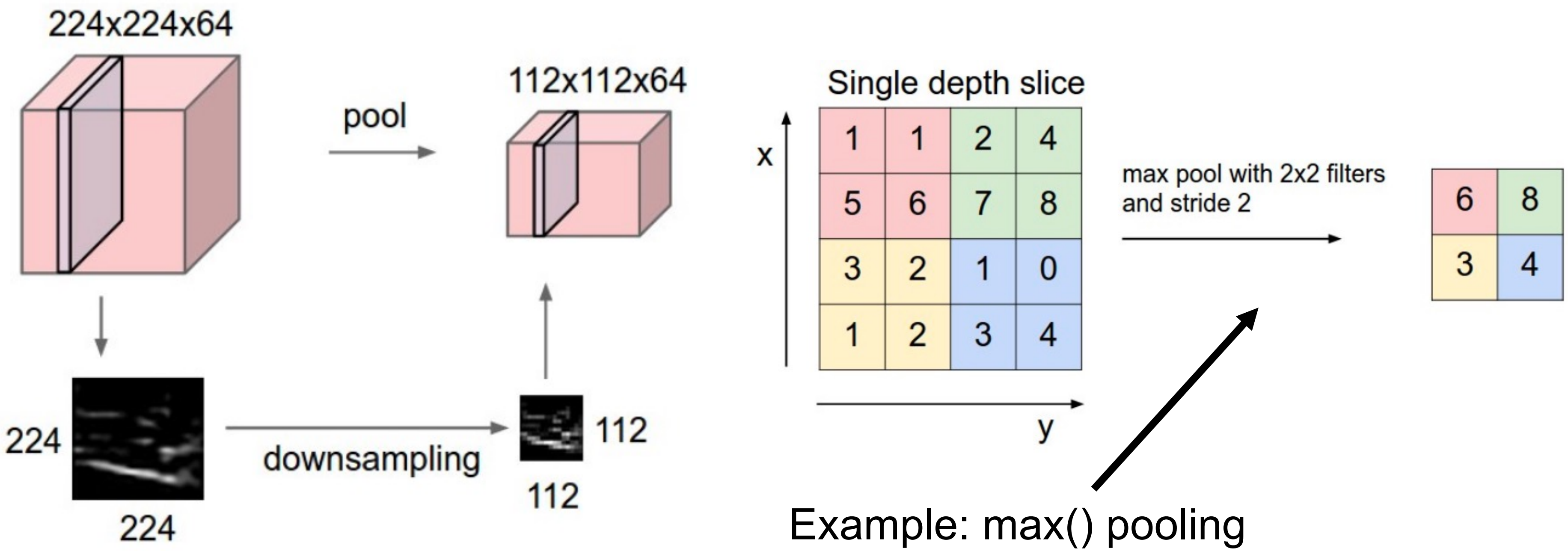
# Convolutional Neural Networks (CNNs)

- **Spatial Pooling**
  - The typical blocks are the max and sum pooling, respectively computing the maximum and the summed response of each feature channel in a H' x W' patch.
- Pooling progressively reduces the spatial size of the input representation.
  - This reduces the number of parameters and, therefore, controls over fitting;
  - Also, it makes the network invariant to small transforms, distortions and translations in the input image (a small distortion in input will not change the output of pooling).

$$y_{i''j''d} = \max_{1 \leq i' \leq H', 1 \leq j' \leq W'} x_{i''+i'-1, j''+j'-1, d}. \qquad y_{i''j''d} = \frac{1}{W'H'} \sum_{1 \leq i' \leq H', 1 \leq j' \leq W'} x_{i''+i'-1, j''+j'-1, d}.$$

# Convolutional Neural Networks (CNNs)

- **Pooling**
  - Note that Pooling down samples the input volume only spatially;
  - The input depth is equal to the output depth;
  - The pooling operation is often considered **deprecated**. To reduce the size of the representation, in is possible to use larger strides in the convolution layers.



# Convolutional Neural Networks (CNNs)

- **Batch Normalization**
  - Deep networks suffer from internal covariate shift—changes in the distribution of each layer’s inputs during training.
    - This slows down training and makes it harder to tune hyperparameters.
  - Batch Normalization (BN) addresses this by normalizing the input of each layer so that it has a mean of 0 and variance of 1, which stabilizes and accelerates learning.
- Typically applied **after a convolutional or fully connected layer and before the non-linearity (activation)**.
- In CNNs, BN is applied **per feature map**, i.e., the same mean and variance are used across all spatial locations in a channel.
- BN behaves differently during **training** and **inference**:
  - **Training**: uses batch statistics.
  - **Inference**: uses running averages of $\mu$ and $\sigma$

# Convolutional Neural Networks (CNNs)

- Given an input mini-batch B = $\{x_1, x_2, \ldots x_m\}$ of size “m”, Batch Normalization is applied for each feature map:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Obtained from all elements in the feature map.
E.g, having (M=2,C=1,H=2,W=2) it will calculate a single mean value, for all spatial and depth slements

$$\sigma = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu)^2$$

- The transformed features are given by:

$$y_i = \gamma \frac{x_i - \mu}{\sqrt{\sigma + \epsilon}} + \beta$$

where $\gamma, \beta$ are learnable parameters

# Convolutional Neural Networks (CNNs)

Consider a batch of **2 samples**, each with **1 channel** and a **2×2 feature map**:

**Sample 1:**

$$x^{(1)} = \begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$$

$$x^{(2)} = \begin{bmatrix} 5.0 & 7.0 \\ 6.0 & 8.0 \end{bmatrix}$$

**Sample 2:**

You are told the BN layer uses:

- $\gamma=1.5$
- $\beta=0.0$
- $\epsilon=0$

- Apply the BN procedure and obtain the transformed feature maps.

# Convolutional Neural Networks (CNNs)

- **Non-Linearity**
  - There are two basic non-linear activation functions used in CNNS: “ReLU” (Rectified Linear Units) and “Sigmoid”.

$$y_{ijd} = \max\{0, x_{ijd}\}.$$

$$y_{ijd} = \sigma(x_{ijd}) = \frac{1}{1 + e^{-x_{ijd}}}.$$

  - As advantages with respect to each other, Sigmoid is consider not to blow up activation, while ReLU **does not vanishes the gradient**
    - In the case of Sigmoid, when the input grows to infinitely large, the derivative tends to 0.
  - However, in the case of ReLU, there is no mechanism to constrain the output of the neuron, as the input is often the output)

# Convolutional Neural Networks (CNNs)

- **Fully Connected layers**
  - Neurons in a fully connected layer have full connections to all activations in the previous layer, as in a regular feed-forward network.
  - In practical terms, these neurons resemble pretty much the neurons in ”Convolution” layers.
    - The only difference between fully connected and Convolution layers is that the neurons in the former layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters.
    - However, the neurons in both layers still compute dot products, so their functional form is identical.
  - For example, an FC layer with K=4096 that is looking at some input volume of size 7×7×512 can be expressed as a Convolution layer with F=7 x 7 x 4096 (padding 0, stride 1).
  - In other words, we are setting the filter size to be exactly the size of the input volume;
  - Hence the output will simply be 1×1×4096.

# Convolutional Neural Networks (CNNs)

- **Softmax**
  - Can be seen as the combination of an activation function (exponential) and a normalization operator.
  - It is usually applied as the transfer function of the last layer of the CNN, where the idea is to push up the maximum value of the responses to “1”, and all the other values to “0”.
  - In practice, it simulates the probability of the input corresponding to each category, represented by a neuron in the output layer.

$$y_{ijk} = \frac{e^{x_{ijk}}}{\sum_{t=1}^{D} e^{x_{ijt}}}.$$

# Convolutional Neural Networks (CNNs)

- Most of the data memory used by CNNs is used in the early Convolutional layers (where spatial resolution is maximal), whereas most of the parameters of the network are in the fully connected layers.
  - Example **VGGNet,** one of the well known and succeeded architectures:

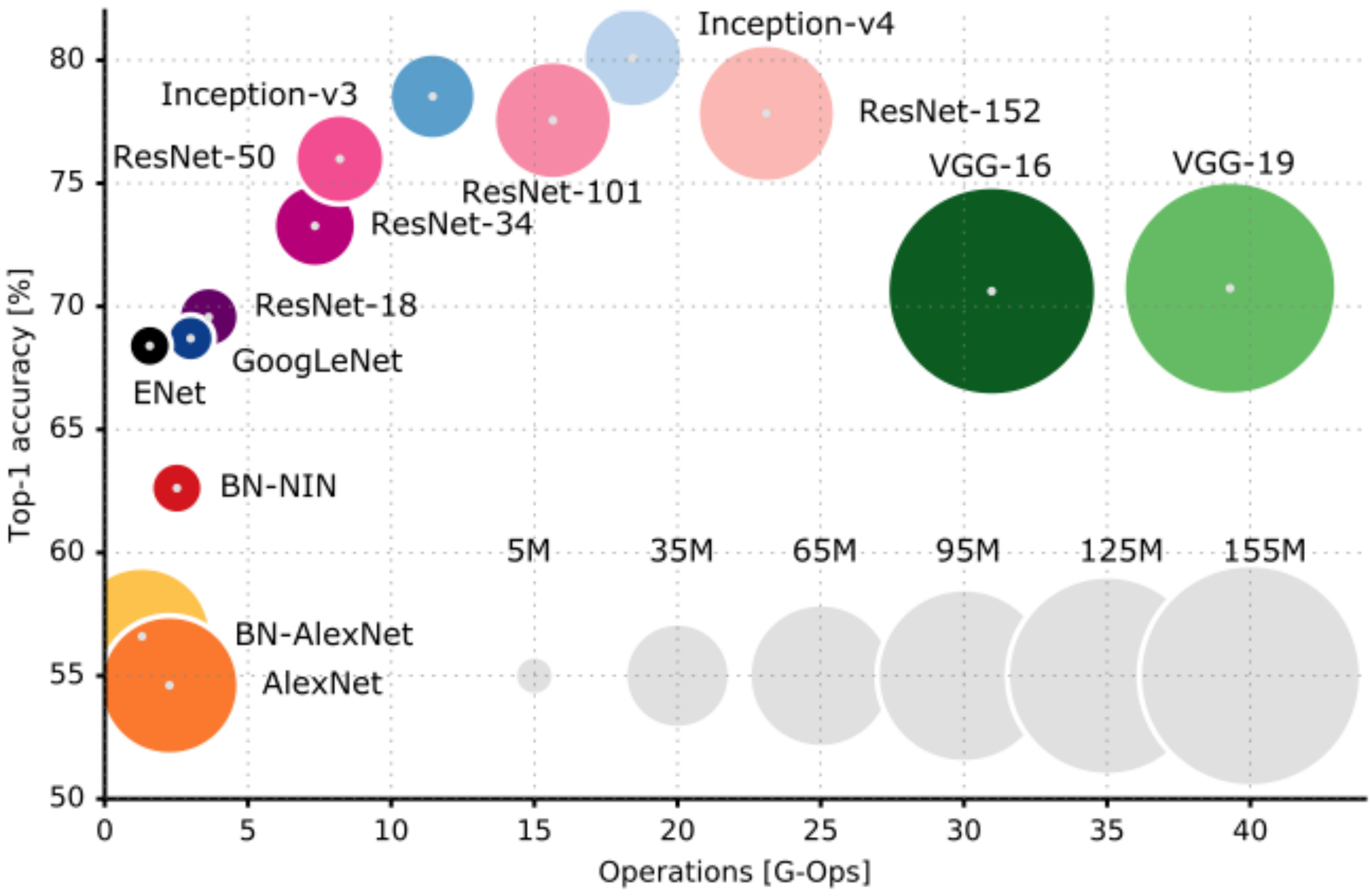
INPUT: [224x224x3] memory: 224*224*3=150K weights: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K weights: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M weights: (3*3*128)*128 = 147,456 POOL2: [56x56x128] memory: 56*56*128=400K weights: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K weights: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K weights: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K weights: 0
FC: [1x1x4096] memory: 4096 weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 weights: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 weights: 4096*1000 = 4,096,000

# Convolutional Neural Networks (CNNs)

- **VGGNet:**
  - The total memory used is about 4 bytes * 24,000,000 = 93 MB
  - This is required only for the **forward step**
  - In practice, the backward step requires around the double memory;
  - The network has 138,000,000 parameters to be tuned by the back-propagation algorithm.
- It should be noted that the conventional paradigm of a linear list of layers is not the state-of-the-art anymore.
  - Google's Inception architectures and also Residual Networks from Microsoft Research Asia.
  - Both of these feature more intricate and different connectivity structures.
- Most of the COTS (commercial off-the-shelf) models have complex graph-based architectures.

# Convolutional Neural Networks (CNNs)

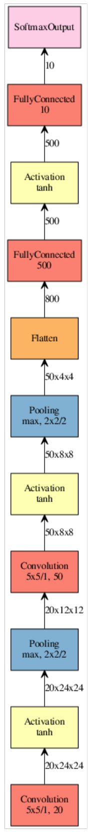
- Accuracy vs. Number of operations for a single forward step. Circumference radii corresponds to the number of parameters




**Source**: https://towardsdatascience.com/neural-network-architectures-156e5bad51ba

# Convolutional Neural Networks (CNNs)

- An illustration of the most popular deep learning architectures is provided in http://josephpcohen.com/w/visualizing-cnn-architectures-side-by-side-with-mxnet/

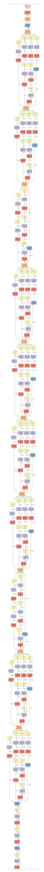



LeNet

AlexNet

VGG

GoogLeNet



Inception

Resnet

# CNNs: Example

- How to create (and instantiate) one CNN (Sequential):

```
def cnn_model(input_shape=(32, 32, 3)):

    model = Sequential()

    model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu', input_shape=input_shape))
    model.add(Conv2D(filters=32, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(Conv2D(filters=64, kernel_size=3, padding='same', activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dense(10, activation='softmax'))


    return model

# ###########################################################
# Instantiate model
model = cnn_model()
model.summary()

# Compile model
model.compile(optimizer='rmsprop’, loss='categorical_crossentropy’, metrics=['accuracy'])
```

(“**Sequential**” objects provide the simplest way. “**Functional**” objects enable additional functionalities)

# CNNs: Example

- How to use (or fine tune) one well known CNN model:
  - Example: Inception.V3

```
def create_inception(tot_classes):
    imgs_input = Input((args['image_height'], args['image_width'], 3))

    if args['fine_tuning'] == 0:
        model_tmp = inception_v3.InceptionV3(input_shape=(args['image_height'], args['image_width'], 3),
                        weights=None, include_top=False)
    else:
        model_tmp = inception_v3.InceptionV3(input_shape=(args['image_height'], args['image_width'], 3),
                        weights='imagenet', include_top=False)
        model_tmp.trainable = False

    x = model_tmp(imgs_input, training=False)

    x = keras.layers.GlobalAveragePooling2D()(x)
    outs = Dense(tot_classes, activation='linear')(x)

    md = Model(inputs=imgs_input, outputs=outs)
    md.compile(optimizer=RMSprop(learning_rate=args['learning_rate']), loss=tf.keras.losses.MeanAbsoluteError())
    return md
```

- This is typically the approach that attains the best results.
  - Not only the architecture was coherently designed, but also the weights were optimized based in huge datasets.

# CNNs: Example

- How to train one CNN:

```
# For small datasets
history = model.fit(X_train, y_train, batch_size=32, epochs=10, verbose=1, validation_split=.3)

# For large datasets
for i in range(tot_batches):
            [X_batch, y_batch] = get_input_batch(i)
            loss = model.train_on_batch(X_batch, y_batch)
```

- Typical preprocessing steps:

```
# Images are typically normalized to the range [0, 1].
X_train = X_train.astype("float32") / 255
X_test  = X_test.astype("float32") / 255


# In classification problems, labels are typically converted to one-hot encoding.
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)
```

# Argument Parsing

```
ap = argparse.ArgumentParser()
ap.add_argument('-d', '--dataset', required=True, help='CSV learning dataset file')
ap.add_argument('-o', '--output_folder', required=True, help='Output folder')
ap.add_argument('-b', '--batch_size', type=int, default=100, help='Learning batch size')
ap.add_argument('-iw', '--image_width', type=int, default=512, help='Image width')
ap.add_argument('-ih', '--image_height', type=int, default=128, help='Image height')
ap.add_argument('-l', '--learning_rate', type=float, default=1e-3, help='Learning rate')
ap.add_argument('-de', '--decay_rate', type=float, default=1e-2, help='Decay rate')
ap.add_argument('-dr', '--dropout_rate', type=float, default=0.25, help='Dropout rate')
ap.add_argument('-e', '--epochs', type=int, default=1000000, help='Tot. epochs')
ap.add_argument('-pl', '--probability_learn', type=float, default=0.7, help='Probability Learning set')
ap.add_argument('-pv', '--probability_validation', type=float, default=0.15, help='Probability Validation set')
args = ap.parse_args()
```

The script is then executed by: “python3 script.py –d ‘data.csv’,...

# Large Dataset Loading

```
def read_csv(dataset):
    # ###########################
    # Load Data in '.csv' format: [ [filename_1, label_1], [filename_2, label_2],...]
    samp = []
    with open(dataset) as f:
        csv_file = csv.reader(f, delimiter=',')
        for row in csv_file:
            samp.append(row)

    random.shuffle(samp)
    return samp
```

The “csv” file should be in the format:
/path/image_1.jpg 1
/path/image_2.jpg 0
/path/image_3.jpg 2

# Dataset Splitting

```
def split_dataset(dt):
    dt_l = []
    dt_v = []
    dt_t = []
    onehot_encoder = OneHotEncoder(sparse=False)
    onehot_encoder.fit(np.asarray([x[-1] for x in dt]).reshape(-1, 1))
    out = onehot_encoder.transform(np.asarray([x[-1] for x in dt]).reshape(-1, 1))
    dt = list(zip(dt, out))

    for el in dt:
        x = random.random()
        if x < args.probability_learn:
            dt_l.append([el[0][0], el[1]])
        elif x < args.probability_learn + args.probability_validation:
            dt_v.append([el[0][0], el[1]])
        else:
            dt_t.append([el[0][0], el[1]])

    return dt_l, dt_v, dt_t
```

Divides the available data into three sub-sets: learning + validation + test

# Data Batch Loading

```
def get_input_batch(gt, idx, augm, tot_c):
  tot = min(args.batch_size, len(gt) - idx)

  imgs = np.zeros((tot, args.image_height, args.image_width, 1)).astype('float')
  labels = np.zeros((tot, tot_c)).astype('float')

  for i in range(tot):
    img = cv2.imread(gt[idx + i][0])
    if augm is not None:
      img = augm.augment_image(img)
    img = cv2.resize(img, (args.image_width, args.image_height))

    imgs[i, :, :, 0] = img[:, :, 0] / 255
    labels[i, :] = gt[idx + i][1]

  return imgs, labels
```

Load one batch of (maximum) ”batch_size” images and the corresponding ground truth

# Create CNN

```
def create_cnn(tot_c):
  imgs_input = Input((args.image_height, args.image_width, 3))

  conv12 = Conv2D(64, kernel_size=3, strides=2, padding="same")(imgs_input)
  conv12_bn = BatchNormalization(momentum=0.8)(conv12)
  conv12_a = LeakyReLU()(conv12_bn)
  drop12 = Dropout(args.dropout_rate)(conv12_a)

  conv13 = Conv2D(128, kernel_size=3, strides=2, padding="same")(drop12)
  conv13_bn = BatchNormalization(momentum=0.8)(conv13)
  conv13_a = LeakyReLU()(conv13_bn)
  drop13 = Dropout(args.dropout_rate)(conv13_a)

  conv14 = Conv2D(256, kernel_size=3, strides=2, padding="same")(drop13)
  conv14_bn = BatchNormalization(momentum=0.8)(conv14)
  conv14_a = LeakyReLU()(conv14_bn)
  # drop14 = conv14_a
  drop14 = Dropout(args.dropout_rate)(conv14_a)

  conv15 = Conv2D(512 , kernel_size=3, strides=2, padding="same")(drop14)
  conv15_bn = BatchNormalization(momentum=0.8)(conv15)
  conv15_a = LeakyReLU()(conv15_bn)
  drop15 = Dropout(args.dropout_rate)(conv15_a)

  conv16 = Conv2D(512, kernel_size=3, strides=2, padding="same")(drop15)
  conv16_bn = BatchNormalization(momentum=0.8)(conv16)
  conv16_a = LeakyReLU()(conv16_bn)
  drop16 = Dropout(args.dropout_rate)(conv16_a)
  pooled = Flatten()(drop16)

  dense1 = Dense(128, activation='relu', kernel_constraint=None)(pooled)
  drop1 = Dropout(args.dropout_rate)(dense1)

  dense2 = Dense(64, activation='relu', kernel_constraint=None)(drop1)
  drop2 = Dropout(args.dropout_rate)(dense2)

  outp = Dense(tot_c, activation='sigmoid', kernel_constraint=None)(drop2)
  out = Softmax()(outp)

  md = Model(inputs=imgs_input, outputs=out)
  md.compile(optimizer=SGD(lr=args.learning_rate, momentum=0.8),
loss=tf.keras.losses.CategoricalCrossentropy())
  md.summary()
  return md
```

Creates a CNN of 27 layers

# Train()

```
i = 0
while i < len(l_s):
        [imgs, gt] = get_input_batch(l_s, i, augmenter, tot_c)
        lo = md.train_on_batch(imgs, gt)
        lo_l.append(lo)
        i += args.batch_size
        print('\r Learn [%d - %d/%d]...' % (epoch, i, len(l_s)), end='')
```

One training epoch

```
i = 0
while i < len(v_s):
        [imgs, gt] = get_input_batch(v_s, i, None, tot_c)
        lo = md.test_on_batch(imgs, gt)
        lo_v.append(lo)
        i += args.batch_size
        print('\r Valid [%d - %d/%d]...' % (epoch, i, len(v_s)), end='')
```

One validation epoch

# Train()

```
ep = range(1, epoch + 1)
fig_1 = plt.figure(1, figsize=(18, 8))
plt.clf()
gs = gridspec.GridSpec(2, 2, figure=fig_1)

ax = fig_1.add_subplot(gs[0, 0])
ax.plot(ep, losses_learn, '-g')
ax.plot(ep, losses_valid, '-r')
ax.grid(True)
ax.title.set_text('Losses’)

...
```

Plot intermediate results