Machine Learning & its Application

Convolutional Networks

04-05/10/2023
Done by: Issa Hammoud

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Introduction

- Dense layers are not suitable for image applications.
- We lose the spatial relation in an image when flattening it.
- We also create a huge vector hard to process.
- Instead, we use convolutional networks.
- We will explore convolution operation and how did it integrate machine learning.
- We will then explore solutions for image-to-image applications.

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Convolution

- Before talking about convolution we need to talk about images.
- An image is a 2D visual representation of a scene, stored in an array.
- We have different type of images:
  - Color and gray images.
  - Depth images.
  - Thermal images etc..
- We will focus here on color and gray images only.
- Color and gray images are usually stored on 8 bits.

Convolution

- Storing images on 8 bits mean that each pixel can take values in the range [0-255].
- RGB images represent all colors using a combination of red, green and blue.
- Gray images have only a single channel.
- The pixel value represents its intensity:
  - In gray images, the value represents its blackness or whiteness (0 black, 255 white).
  - In color images, the value represent the intensity of each color in each channel.
- We can convert a color image into a gray image.
- We can do a lot of processing on images to extract useful information

Image Processing

- Convolution is a mathematical operation that is widely used in practice.
- We can implement many image processing algorithms with it.
- It is just a moving kernel applied locally. It is defined for 1d and 2d signals.

Image Processing

- Let's take the first image processing application: blurring.

color image

blurry image

Image Processing

- To blur an image, we can replace each pixel with its neighbors average value.
- This can be implemented by a uniform kernel convolved with the image.

$$K = \begin{bmatrix} \frac{1}{n^2} & \frac{1}{n^2} & \cdots & \frac{1}{n^2} \\ \frac{1}{n^2} & \frac{1}{n^2} & \cdots & \frac{1}{n^2} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{1}{n^2} & \frac{1}{n^2} & \cdots & \frac{1}{n^2} \end{bmatrix}; \; K \in R^{(n,\,n)}$$

Image Processing

- Another application is to detect edges in images.
- On a edge, we have a big difference of color intensities. We can compute the absolute difference between consecutive pixels.

absolute difference in x direction

absolute difference in y direction

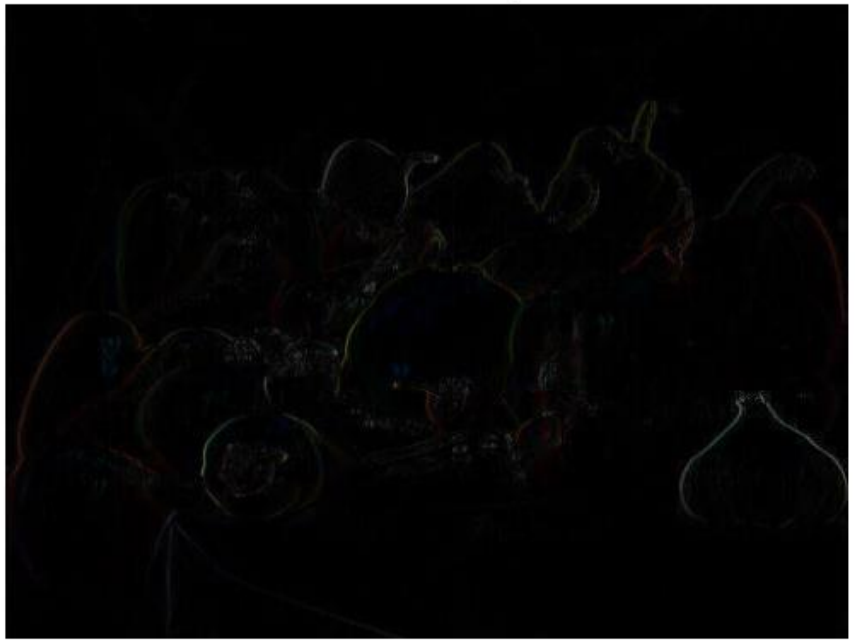


Image Processing

- We can compute the norm to get a single edge image.

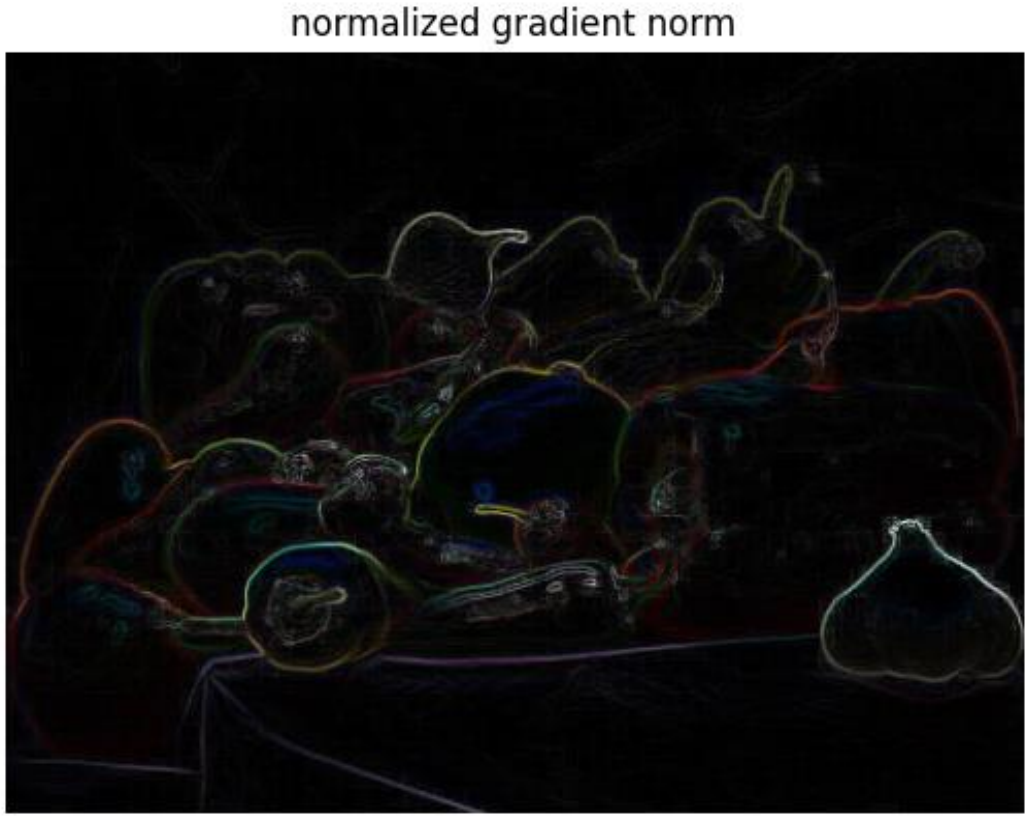


Image Processing

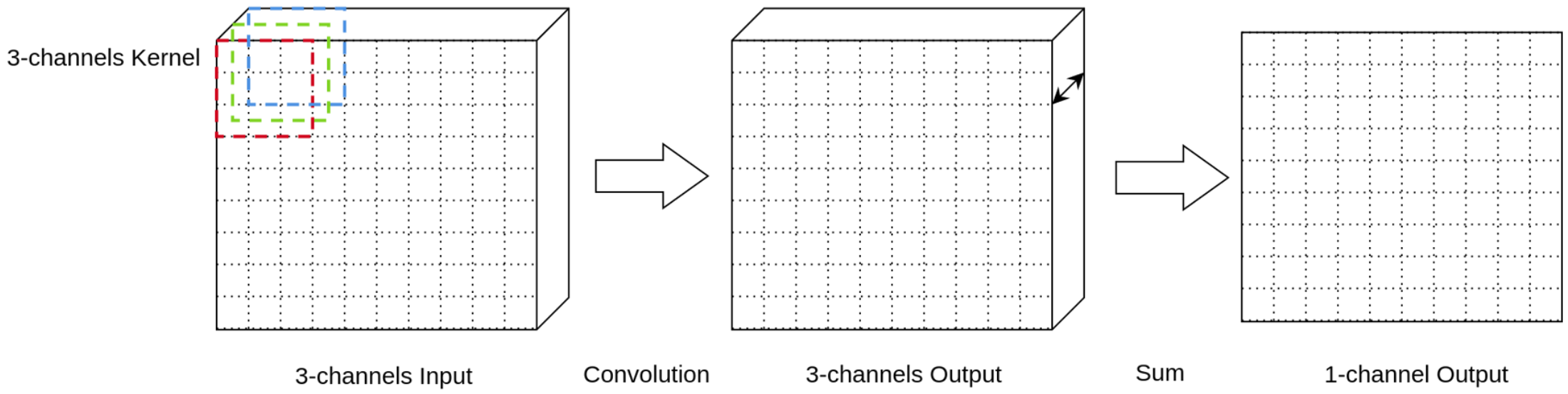
- Interestingly, we can implement the difference in x and y with convolutions.

$$K_x = \begin{bmatrix} -1 & 1 \end{bmatrix};\ K_y = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

- Convolutional kernels can extract meaningful information from an image.
- Instead of using defined kernels, we want to let the model find the kernel values.
- Convolution layers replace dense layers in deep learning for image applications.

Convolution in Neural Networks

- In neural networks, we learn the values of a convolution kernel.
- In addition, we sum all the channels together after convolving with the kernel.



Convolution in Neural Networks

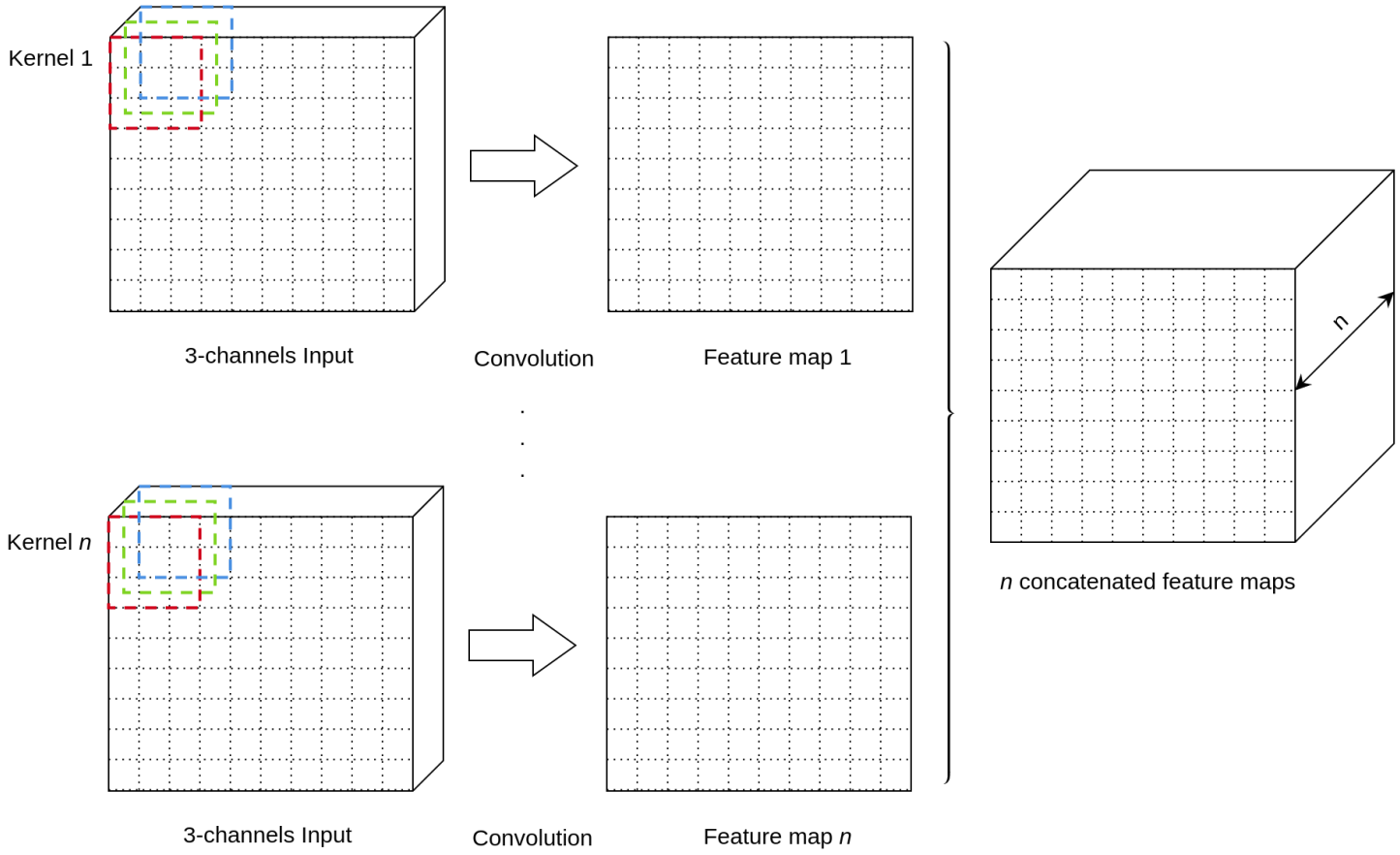
- With dense layers, there is only one parameter to set: the number of units.
- Convolution has multiple parameter to set:
  - Kernel Size.
  - Number of Filters.
  - Padding.
  - Strides.
- We will see what does each one mean.

Convolution in Neural Networks

- The kernel size is the spatial size of the learnable kernel (k_x, k_y).
- However, the total size of the kernel is $k_x * k_y * c$, where c is the input channels.
- Number of filters *n* is the number of kernels to use in a layer.
- Each kernel (or filter) will generate a single "feature" image.
- We concatenate those features by channel to create a feature map.
- A bias term of the same size of the features is used as well.

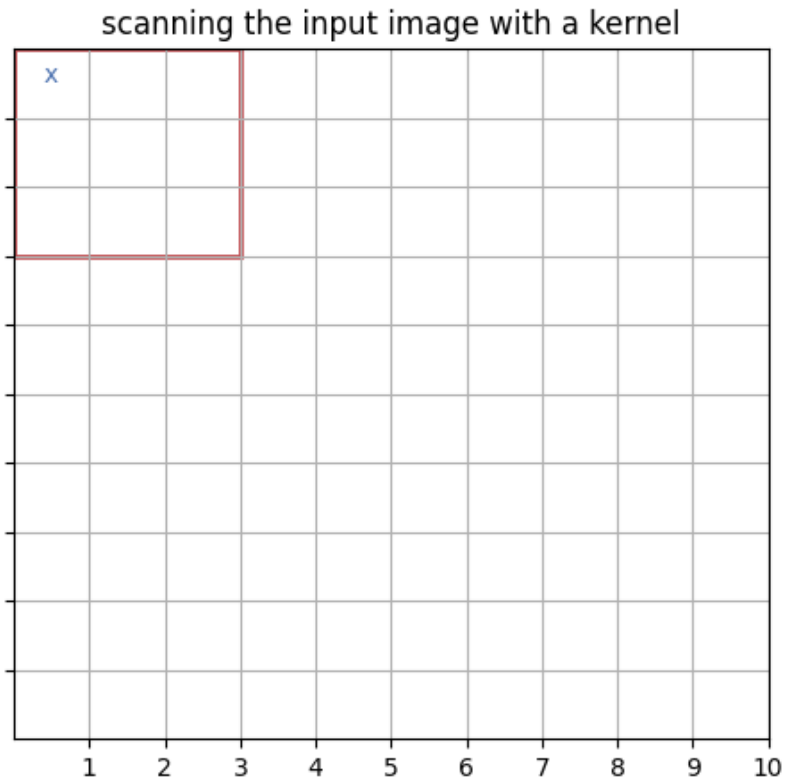
Convolution in Neural Networks

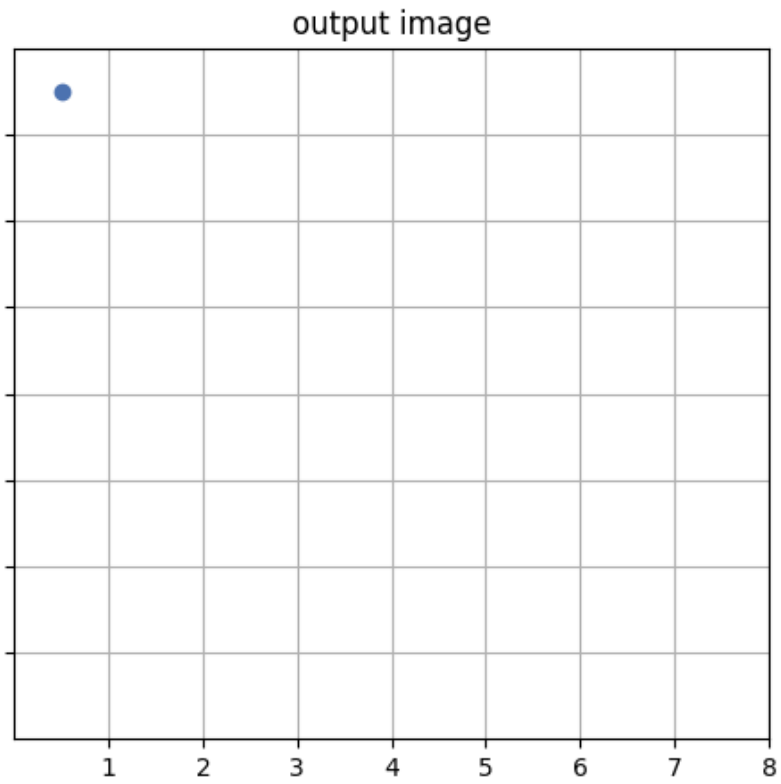
- The total number of parameters of a convolutional layer is: $k_x * k_y * c * n$.



Convolution in Neural Networks

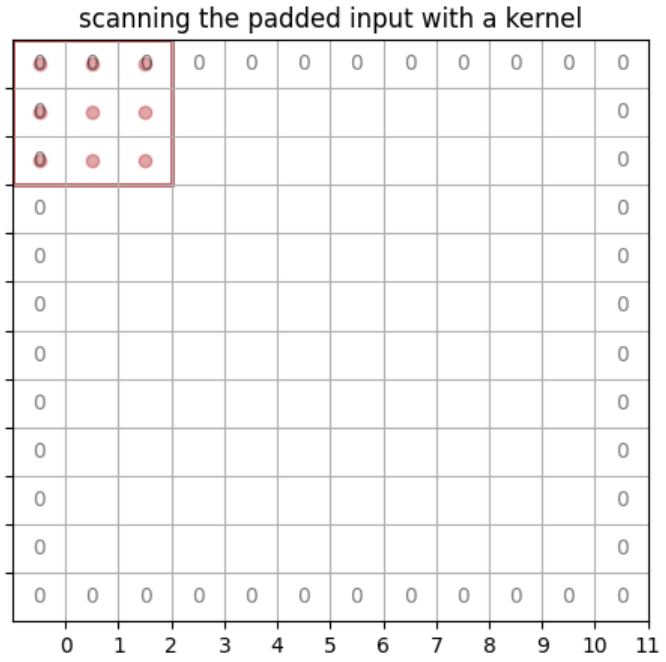
- The convolution operation outputs an image smaller than its input.

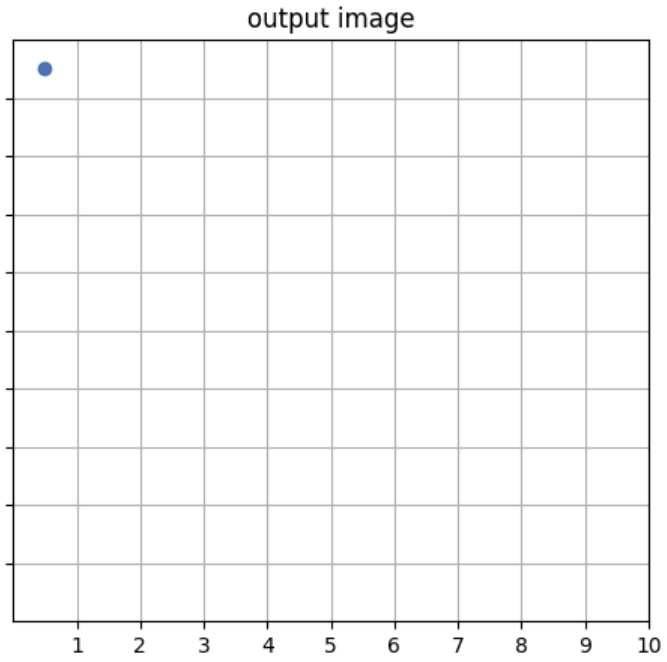




Convolution in Neural Networks

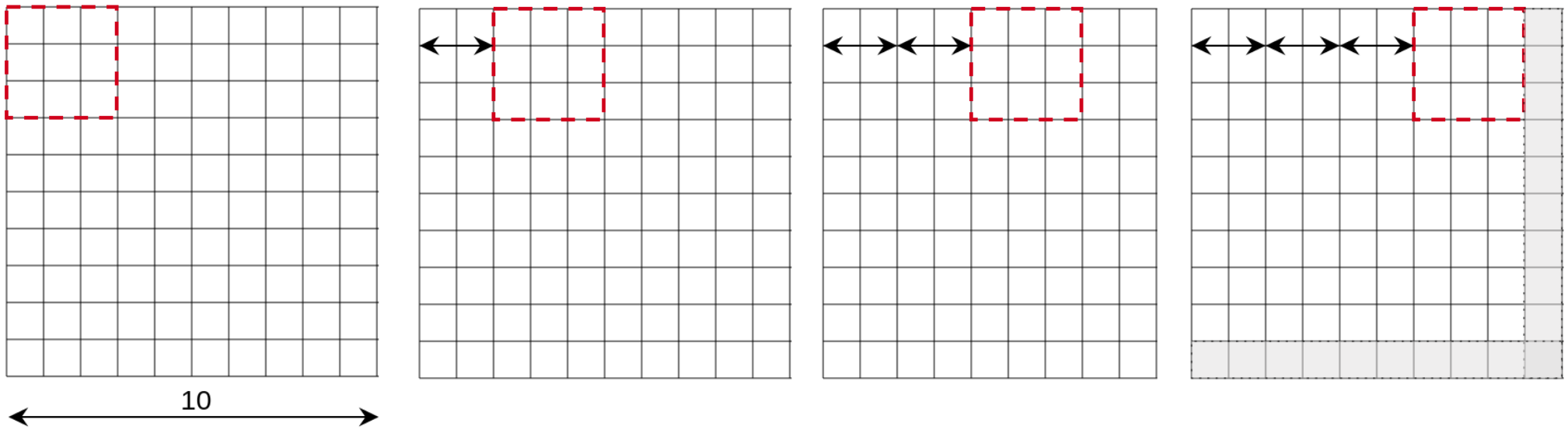
- However, we would to conserve the same output size, thus we pad the input.
- We call this padding "same". If no padding applied, it is called "valid".





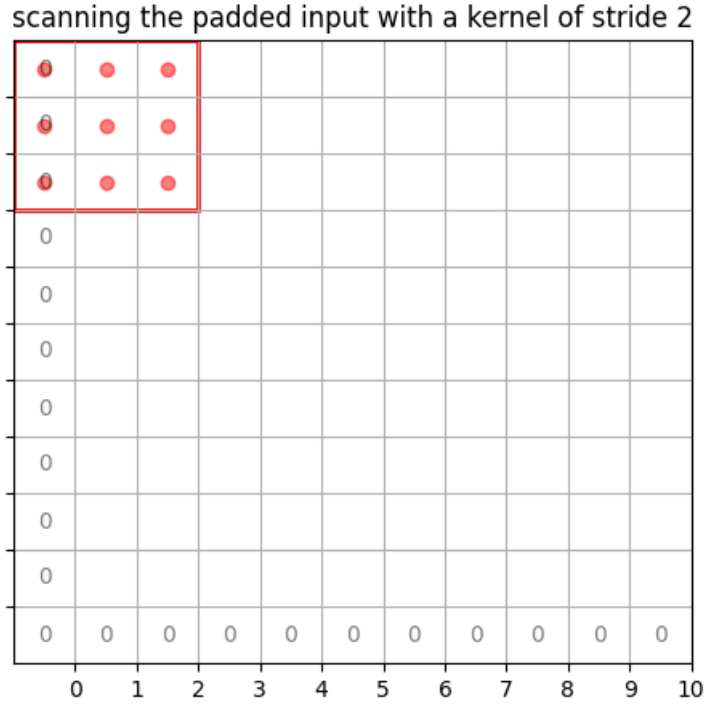
Convolution in Neural Networks

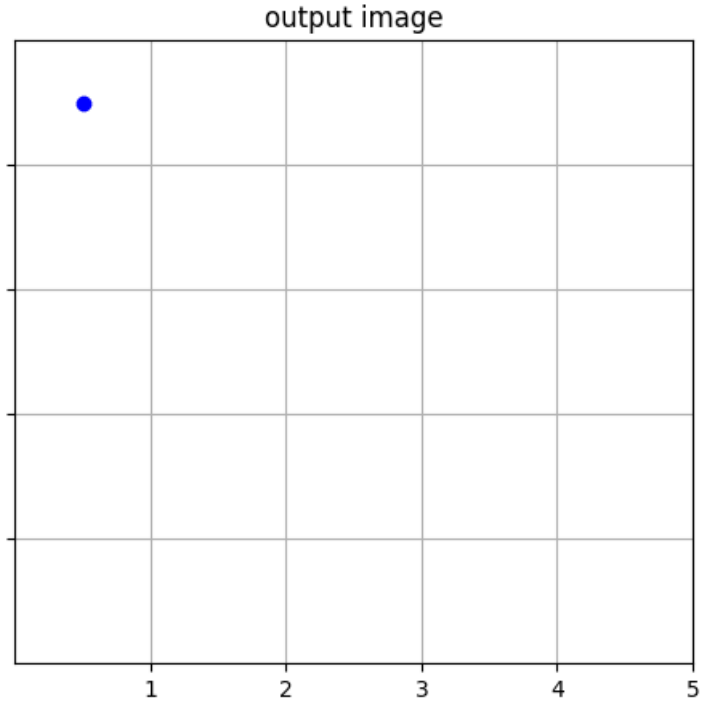
- We presented the case when the kernel move 1 step at a time.
- This step is called the stride, it can be bigger as well.
- What remains from the image is dropped.



Convolution in Neural Networks

- Strides > 1 reduce the output size. We call such convolution strided convolution.
- Using strides = 2 and padding will give an output of half size the input.





Convolution vs Dense Layers

- There are several advantages of using convolution for images:
  - We conserve the image structure and local correlation instead of flattening.
  - We reduce the number of **parameters** needed by **sharing** them across the image.
- A convolutional layer is equivalent to a sparse dense layer with shared weights.

$$Image\ I = \begin{bmatrix} I_{11} & I_{12} & \dots & I_{1n} \\ I_{21} & I_{22} & \dots & I_{2n} \\ \dots & \dots & \dots & \dots \\ I_{n1} & I_{n2} & \dots & I_{nn} \end{bmatrix};\ Kernel\ K = \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix};\ O = I{*}K \rightarrow O = \begin{bmatrix} I_{11} & I_{12} & \dots & I_{1n} \\ I_{21} & I_{22} & \dots & I_{2n} \\ \dots & \dots & \dots & \dots \\ I_{n1} & I_{n2} & \dots & I_{nn} \end{bmatrix} * \begin{bmatrix} k_1 & k_2 \\ k_3 & k_4 \end{bmatrix}$$

We can implement a convolution, $O = I{*}K$, using a dense layer multiplying the Flattened Image FI with a matrix A, as follows:

$$FI = \begin{bmatrix} I_{11} & I_{12} & \dots & I_{1n} & I_{21} & I_{22} & \dots & I_{2n} & \dots & I_{n1} & I_{n2} & \dots & I_{nn} \end{bmatrix};\ A = \begin{bmatrix} k_1 & 0 & 0 & \dots & 0 \\ k_2 & k_1 & 0 & \dots & 0 \\ 0 & k_2 & k_1 & \dots & \dots \\ 0 & 0 & k_2 & \dots & k_1 \\ \dots & \dots & \dots & \dots & k_2 \\ k_3 & 0 & 0 & \dots & 0 \\ k_4 & k_3 & 0 & \dots & 0 \\ 0 & k_4 & k_3 & \dots & \dots \\ \dots & \dots & k_4 & \dots & k_3 \\ 0 & 0 & 0 & \dots & k_4 \end{bmatrix}$$

Outline

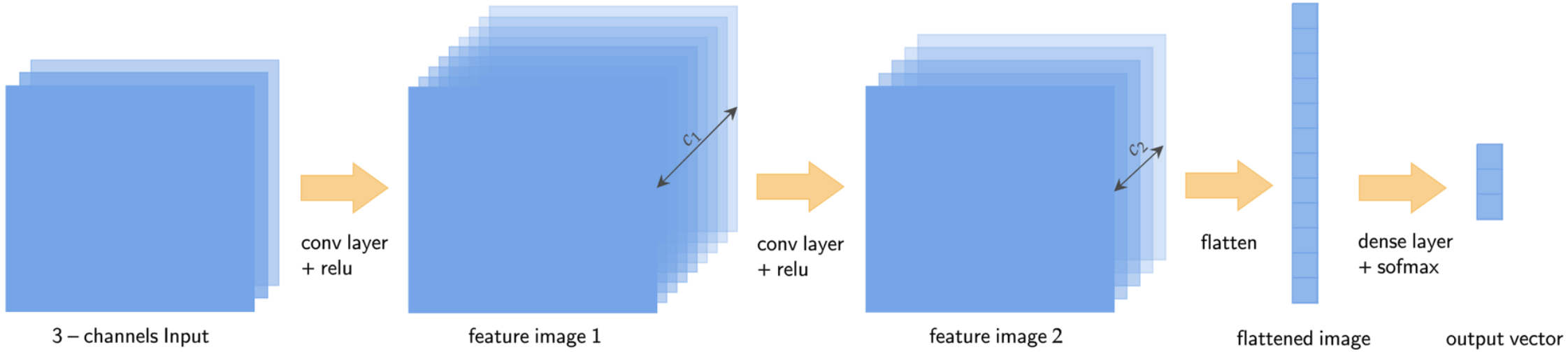
- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Convolution Neural Networks

- We presented how a convolutional layer works. Let's look at convolutional networks.
- Imagine we want to classify images into 3 classes, i.e. multiclass classification.
- We can replace the dense layers in a network with convolutional layers.
- However, we still need at the end a dense layer to apply the linear classifier.
- The convolutional "backbone" is used to extract useful features.
- Then on top we apply one or multiple dense layer.

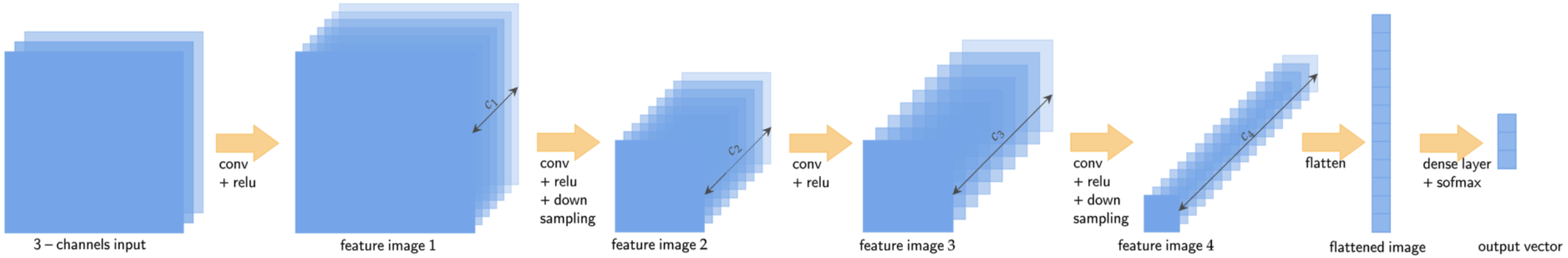
Convolution Neural Networks

- We need to flatten the last convolutional layer in order to apply a dense layer.
- However, we get to the same initial problem with a huge dense layer!
- Thus, we should gradually reduce the feature size before flattening.



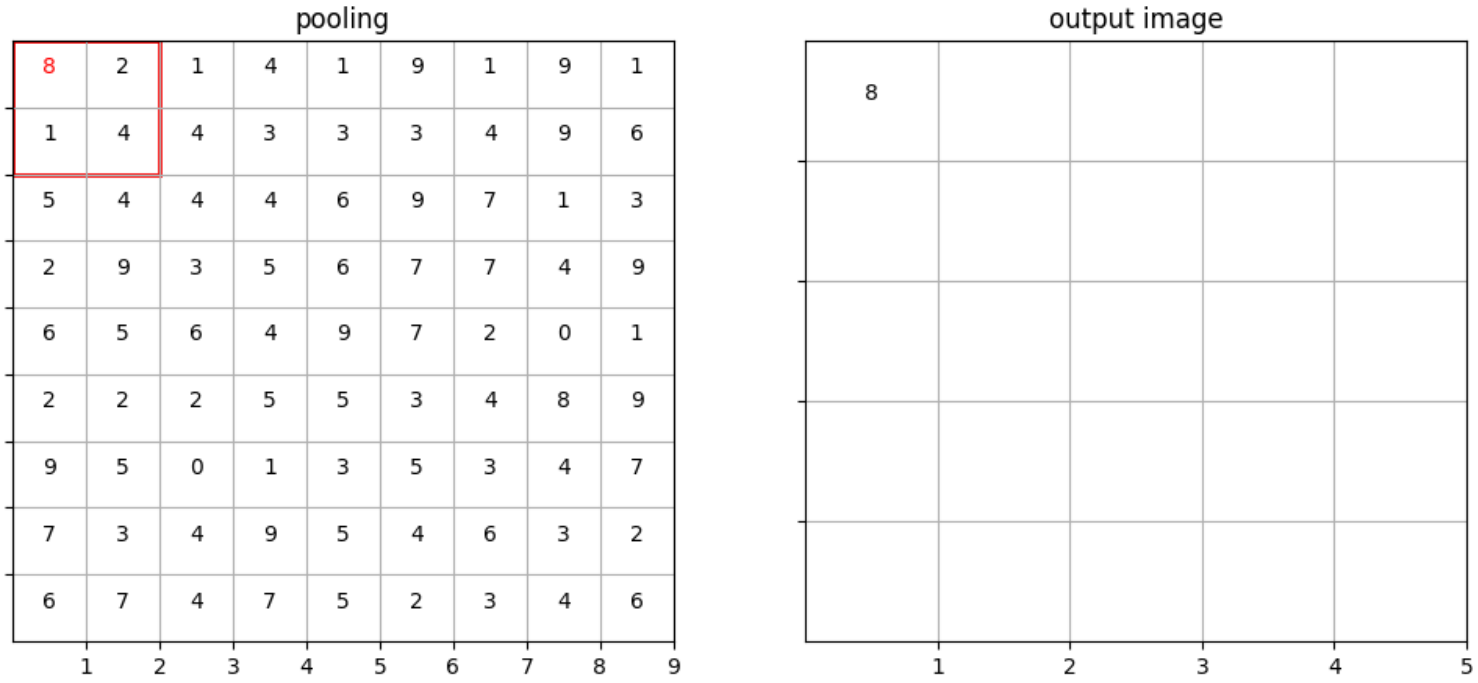
Convolution Neural Networks

- One way to gradually reduce the size if to use strided convolutions (stride > 1).
- With strides=2, we will reduce at each stage the spatial dimension.
- This will also make also the convolution faster.



Convolution Neural Networks

- Another way to reduce the spatial dimensions is pooling.
- Pooling compute an average, or max value, in a kernel size. It is not learnable.



Convolution Neural Networks

- Downsampling, whether with pooling or striding, is a common pattern.
- It is also related to an important notion called "*receptive field*" covered later.
- An alternative to flattening is computing an average (or max) value per channel.
- It can help reduce the size of the output vector before the dense layer.
- We can develop our first CNN to classify images.

Convolution Neural Networks

```python
def cnn_with_pooling(input_image, flatten=True):
  """
  input_image: a 4D array of type float32 and shape (batch, height, width, channels)
  return: a 2D array of type float32 and shape (batch, 3)
  """
  # Conv2D is a class that implements the convolutional layer
  # filter is the number of filters (or feature maps) to use
  conv_1 = tf.keras.layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(input_image)
  conv_2 = tf.keras.layers.Conv2D(filters=8, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(conv_1)
  downsample_1 = tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv_2)

  conv_3 = tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(downsample_1)
  conv_4 = tf.keras.layers.Conv2D(filters=16, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(conv_3)
  downsample_2 = tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv_4)

  conv_5 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(downsample_2)
  conv_6 = tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='same', activation='relu')(conv_5)
  downsample_3 = tf.keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2, 2), padding='same')(conv_6)

  if flatten:
    vector = tf.keras.layers.Flatten()(downsample_3)
  else:
    vector = tf.keras.layers.GlobalAveragePooling2D()(downsample_3)
  dense = tf.keras.layers.Dense(units=3)(vector)
  output = tf.nn.softmax(dense, axis=-1)
  return output
```

Convolution Neural Networks

- Let's compute the parameters number and output shape with an image of size (100, 100, 3).

Layer Name	Number of Parameters	Output Shape
conv_1	3*3*3*8 + 8 = 224	(100, 100, 8)
conv_2	3*3*8*8 + 8 = 584	(100, 100, 8)
downsample_1	0	(50, 50, 8)
conv_3	3*3*8*16 + 16 = 1168	(50, 50, 16)
conv_4	3*3*16*16 + 16 = 2320	(50, 50, 16)
downsample_2	0	(25, 25, 16)
conv_5	3*3*16*32 + 32 = 4640	(25, 25, 32)
conv_6	3*3*32*32 + 32 = 9248	(25, 25, 32)
downsample_3	0	(13, 13, 32)
vector	0	(5408,)
dense	5408 * 3 = 16227	(3,)
output	0	(3,)
Total	34411	-

Convolution Neural Networks

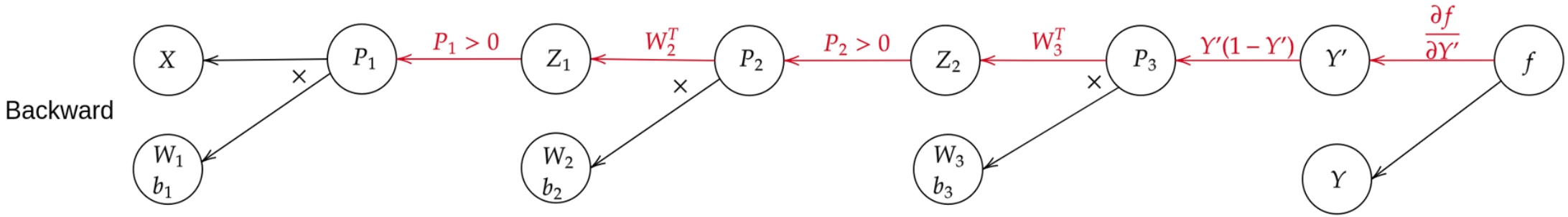
- There are 2 things to remark:
  1. The convolutional layer parameters depend only the number of channels, not the spatial size.
  2. Convolutional layer are not very parameter-intensive, so we can add many of them w/o overfitting.
- The second point implies that we can develop deep networks (a lot of layers).
- However, some problems may arise:
  - Gradient vanishing or explosion: the gradients become too small or too big.
  - Slow convergence and instability: training becomes slow and harder to reach good local minima.
- To solve gradient explosion we can clip it by a maximum value or norm.
- The other problems need more advanced techniques to solve them.

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Residual Connections

- Residual connections were proposed to solve the problem of vanishing gradients.
- With more depth, gradient we may become too small in early layers, even with ReLU.
- Gradient is accumulated from last to first layer and thus can become too small.

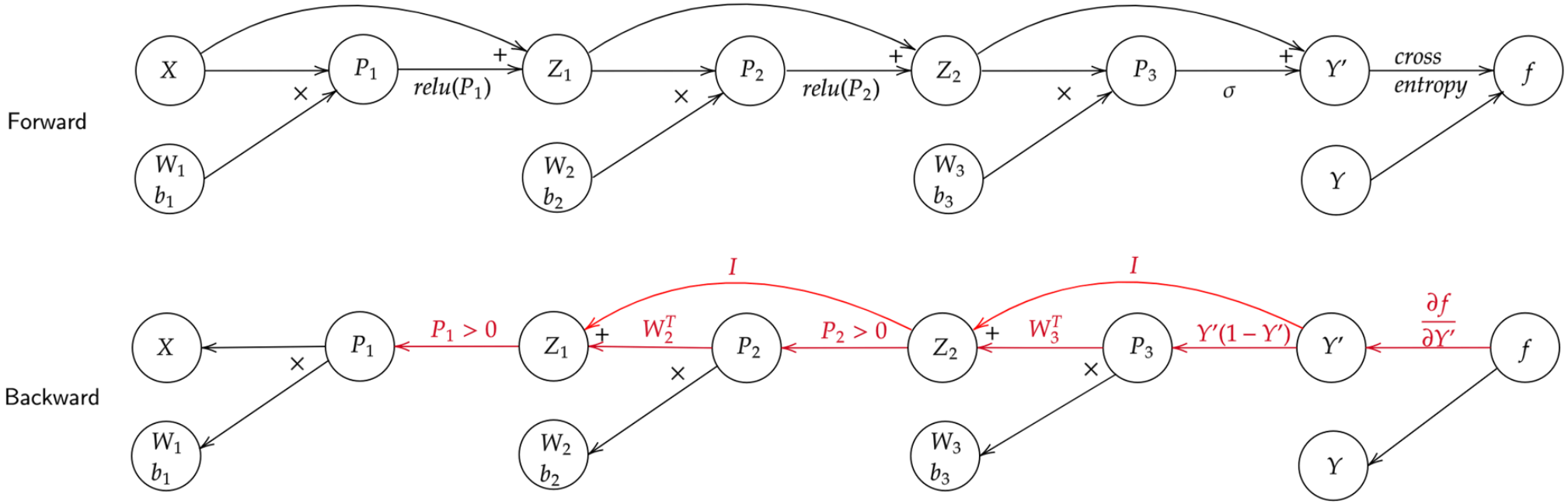


$$grad\_W_1 = \frac{\partial P_1}{\partial W_1} @ grad = grad @ X_1^T \qquad grad\_W_2 = \frac{\partial P_2}{\partial W_2} @ grad = grad @ Z_1^T \qquad grad\_W_3 = \frac{\partial P_3}{\partial W_3} @ grad = grad @ Z_2^T$$

$$grad\_b_1 = \frac{\partial P_1}{\partial b_1} @ grad = grad \qquad grad\_b_2 = \frac{\partial P_2}{\partial b_2} @ grad = grad \qquad grad\_b_3 = \frac{\partial P_3}{\partial b_3} @ grad = grad$$

Residual Connections

- Residual connections ensure a useful gradient at all layers by adding shortcuts.

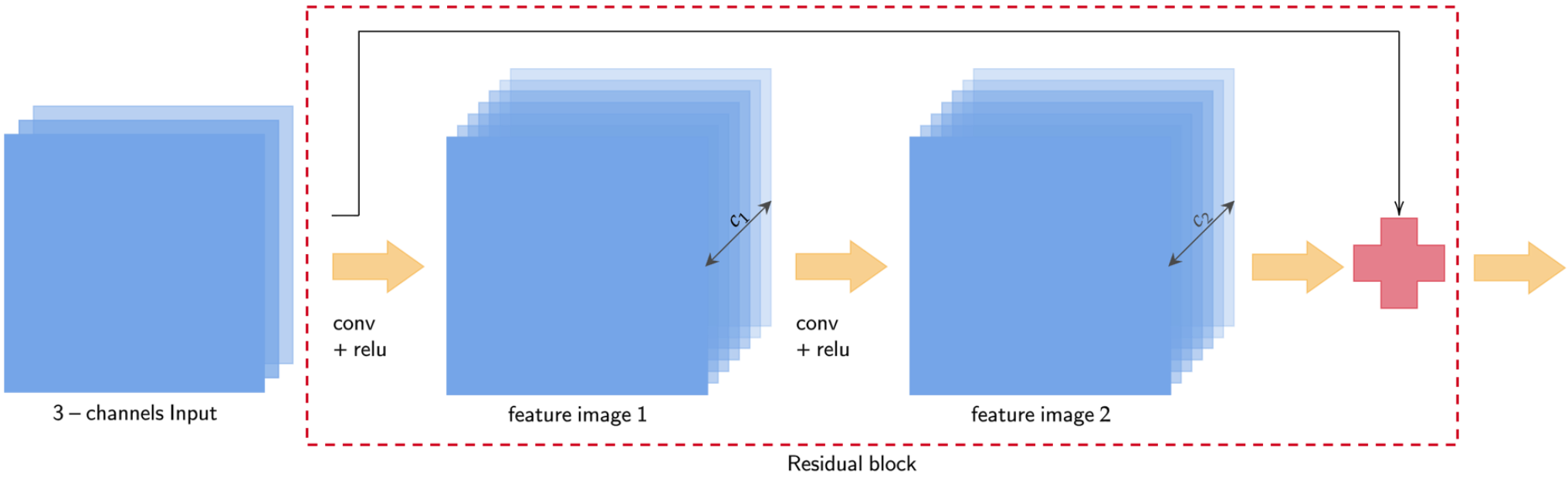


$$grad\_W_1 = \frac{\partial P_1}{\partial W_1} @ grad = grad @ X_1^T \qquad grad\_W_2 = \frac{\partial P_2}{\partial W_2} @ grad = grad @ Z_1^T \qquad grad\_W_3 = \frac{\partial P_3}{\partial W_3} @ grad = grad @ Z_2^T$$

$$grad\_b_1 = \frac{\partial P_1}{\partial b_1} @ grad = grad \qquad grad\_b_2 = \frac{\partial P_2}{\partial b_2} @ grad = grad \qquad grad\_b_3 = \frac{\partial P_3}{\partial b_3} @ grad = grad$$

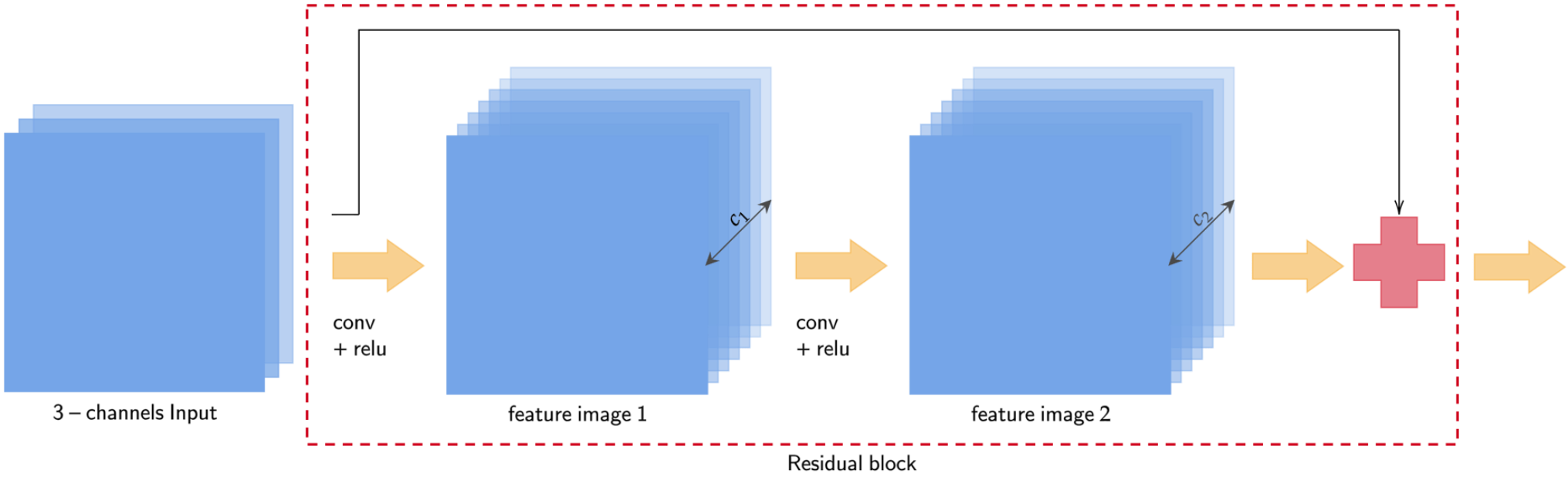
Residual Connections

- It is called residual because output = f(input) + input → f(input) = output – input.
- We are learning the difference between output and input.



Residual Connections

- We can have multiple convolutions inside a residual block.
- But how can we sum the input with the output if they don't have the same shape?



Residual Connections

- The output block has more channel than the input in general (so we can't sum).
- In addition, we may down-sample so the spatial resolution may change as well.
- Thus, instead of directly adding the input, we can add a convolution layer in between.
- We can create a function to create a residual block. It should take:
  - A list of kernel size for each convolution in the block.
  - A list of filters for each convolution in the block.
  - A list of strides for each convolution in the block.

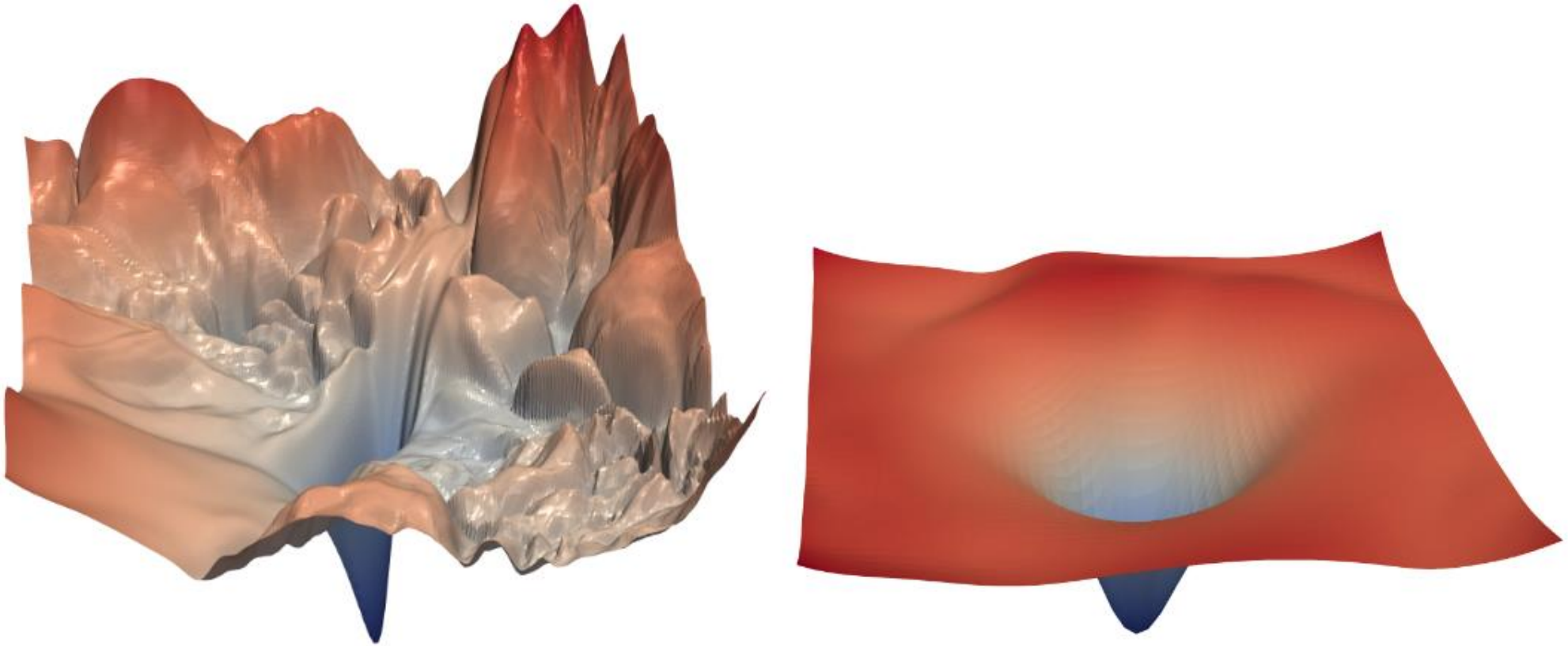
Residual Connections

```
def residual_block_v1(x, feature_maps, strides, kernel_size):
  """
  x: a 4D array of type float32 and shape (batch, height, width, channels)
  feature_maps: list containing the number of feature maps to use
  strides: list containing the strides to use
  kernel_size: list containing the kernel size to use
  return: a 4D array of type float32
  """
  residual = x
  # check if the output dimension will be different from the input.
  # we always suppose that the output channels are bigger than the input channels
  if 2 in strides or x.shape[-1] < feature_maps[-1]:
    residual = tf.keras.layers.Conv2D(
           filters=feature_maps[-1], kernel_size=1,
           strides=max(strides), padding='same', activation='relu')(x)

  conv = x
  for i in range(len(feature_maps)):
    conv = tf.keras.layers.Conv2D(
           filters=feature_maps[i], kernel_size=kernel_size[i],
           strides=strides[i], padding='same', activation='relu')(conv)

  return tf.keras.layers.Add()([conv, residual])
```

Residual Connections

- Residual connections solve gradient vanishing problem, and enable deep networks.
- However, deep networks means in general high non-convexity.
- How is it possible to train deep network with gradient descent?
- To understand what happens inside, researcher developed a visualization technique.
- It is called loss landscape visualization and its purpose is to visualize how the loss change w.r.t parameters (in a reduced dimension like 3D).
- As we will see, residual connections make the loss smoother and easier to optimize.

Residual Connections



(a) without skip connections

(b) with skip connections

Image source: https://arxiv.org/pdf/1712.09913.pdf

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Batch Normalization

- Batch normalization is proposed to stabilize the training and enable faster convergence.
- It enables the use of big learning rate without the risk of divergence and instability.
- Despite it is great success, we still don't have a big understanding why it works.
- It is used with convolutional layers, but also with dense layers as well.
- Every batch normalization layer is composed of 2 steps:
  - Normalization: eliminate the mean and variance of a batch of input.
  - Re-Parameterization: assign new mean and variance that are learnable.

Batch Normalization

- It is summarized in the following equations.
- Note that it is applied for a batch of images not single one.

for a given hidden layer, we denote its output as H_B ; H_B of shape $(B, \dots)$ is a batch of activation, each activation is denoted H_i of shape $(1, \dots)$

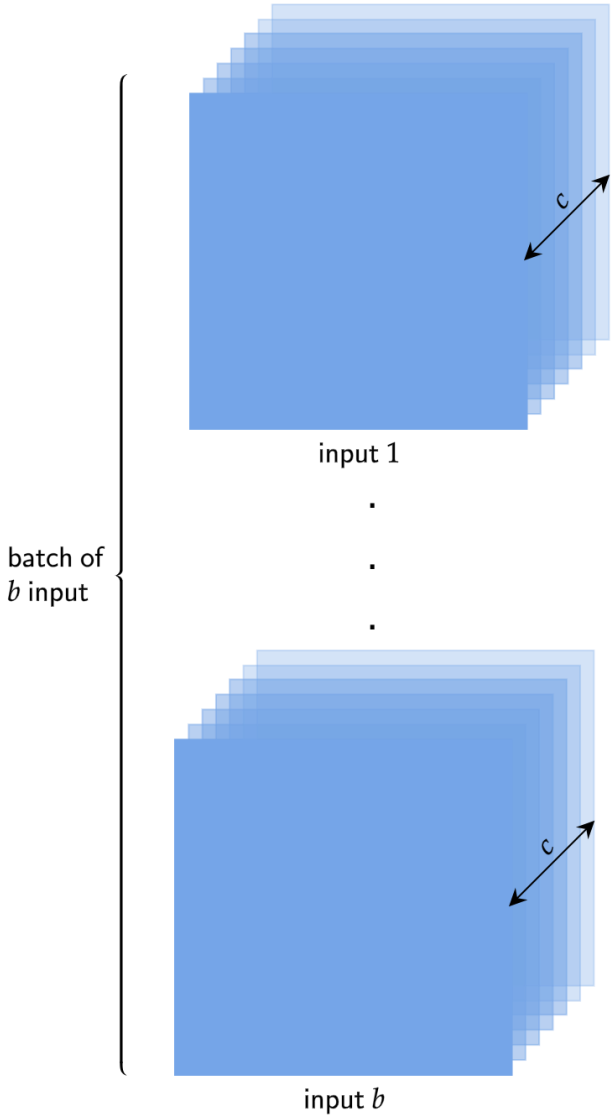
Batch Normalization performs the following :

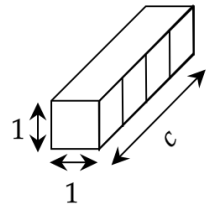
Normalization : $\widehat{H}_B = \frac{H_B - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$; $\mu_B = \frac{1}{B}\sum_i^B H_i$; $\sigma_B^2 = \frac{1}{B}\sum_i^B (H_i - \mu_B)^2$; μ_B and σ_B has the same shape as H_i ; ϵ small value for stability

Re – Parameterization : $\widetilde{H}_B = \gamma \odot \widehat{H}_B + \beta$; γ new learnable standard deviation, β new learnable mean

Batch Normalization

- For convolutional layers, we compute μ and σ over (batch, height, width).
- But multiple questions arise:
  - How to apply it during inference with a single image?
  - Inference will vary based on the batch size.
- Thus, we use batch norm differently in training and inference.





size of μ / σ / γ / β
with convolutional layers

Batch Normalization

- Instead of computing μ and σ for a batch during inference, we will:
  - Estimate a moving mean and variance during training.
  - Apply it in the formula instead of μ and σ.
- Batch normalization doesn't change during training, we just compute an additional quantity.

Batch Normalization during training :

$$\text{Normalization : } \widehat{H}_B = \frac{H_B - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\text{Moving averages : } \mu_{moving} = \alpha\mu_{moving} + (1-\alpha)\mu_B \; ; \; \sigma_{moving} = \alpha\sigma_{moving} + (1-\alpha)\sigma_B \; ; \; \alpha \in [0,1)$$

$$\text{Re - Parameterization : } \widetilde{H}_B = \gamma \odot \widehat{H}_B + \beta$$

Batch Normalization

- Thus, in inference we used the moving μ and σ estimated during training.
- Like this, we can do inference with different batches and have the same results.

Batch Normalization during testing :

$$\text{Normalization}: \widehat{H} = \frac{H - \mu_{moving}}{\sqrt{\sigma^2_{moving} + \epsilon}}$$

$$\text{Re} - \text{Parameterization}: \widetilde{H}_B = \gamma \odot \widehat{H}_B + \beta$$

Batch Normalization

- Some practical considerations when using batch normalization:
  - Batch norm came most of the time directly after a convolution operation, before activation.
  - If your batch size is too small, batch norm estimates will not be accurate.
  - If you are using multiple GPU, each GPU will have a different batch norm estimates.
  - Think to increase your learning rate with batch normalization (preferably with schedules).
- Other normalization techniques exist like Batch Renormalization and Layer Normalization.

Batch Normalization

- We can write and reuse functions that create convolutional and dense blocks.

```
def conv_bn_relu(x, feature_maps, strides, kernel_size):
  """
  x: a 4D array of type float32 and shape (batch, height, width, channels)
  feature_maps: (int) number of feature maps to use
  strides: (int) the strides to use
  kernel_size: (int) the kernel size to use
  return: a 4D array of type float32
  """
  x = tf.keras.layers.Conv2D(
          filters=feature_maps, kernel_size=kernel_size,
          strides=strides, padding='same')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.ReLU()(x)
  return x


def dense_bn_relu(x, units):
  """
  x: a 2D array of type float32 and shape (batch, features)
  units: (int) number of hidden neurons
  """
  x = tf.keras.layers.Dense(units)(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.ReLU()(x)
  return x
```

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

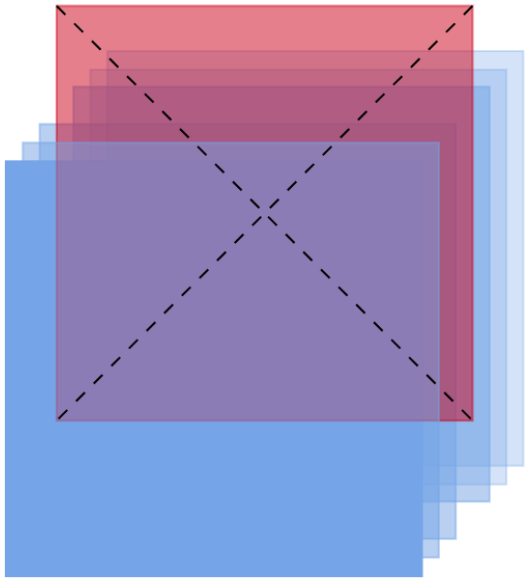
Dropout

- We can use dropout with convolutional layers as well to reduce overfitting.
- However, the standard dropout is not suitable for structured data like image.
- Randomly dropping may hurt spatially correlated information.
- Thus, we replace standard dropout with spatial dropout.
- In such case, we either drop or keep a total feature map (channel).
- This is known as Spatial Dropout 2D. There exist also different forms.

Dropout

- We can implement now a residual block with batch norm and dropout.

```
def residual_block(x, feature_maps, strides, kernel_size, drop_rate):
  """
  x: a 4D array of type float32 and shape (batch, height, width, channels)
  feature_maps: list containing the number of feature maps to use
  strides: list containing the strides to use
  kernel_size: list containing the kernel size to use
  drop_rate: float, the proportion of feature maps to drop.
  return: a 4D array of type float32
  """
  residual = x

  if 2 in strides or x.shape[-1] < feature_maps[-1]:
    residual = conv_bn_relu(x, feature_maps[-1], strides=max(strides), kernel_size=1)

  if drop_rate > 0:
    x = tf.keras.layers.SpatialDropout2D(drop_rate)(x)

  for i in range(len(feature_maps)):
    x = conv_bn_relu(x, feature_maps[i], strides=strides[i], kernel_size=kernel_size[i])

  return tf.keras.layers.Add()([x, residual])
```



randomly drop a feature map

Outline

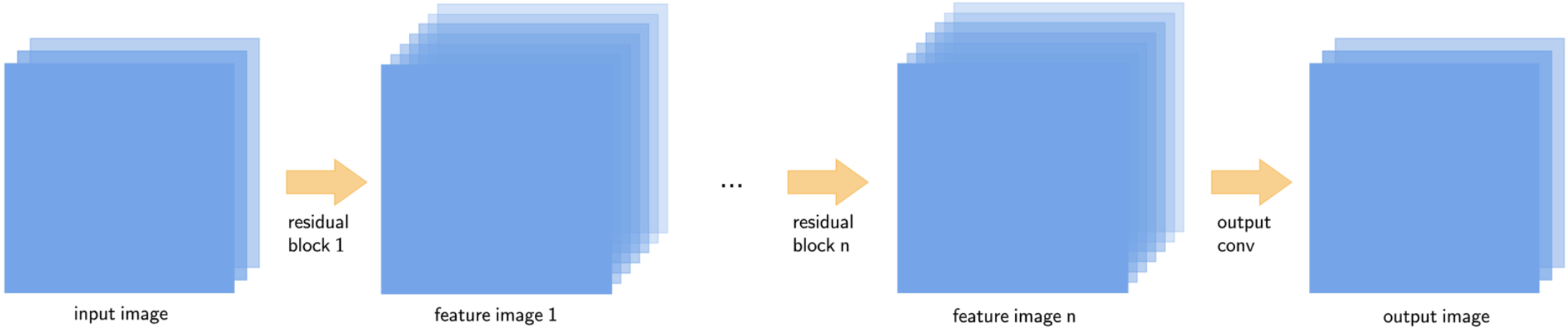
- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Fully Convolutional Networks

- CNN can solve image classification and regression problems.
- However, we can imagine many applications that don't fall directly into the
- For instance, we can't solve image-to-image problems with a CNN.
- We need to design a network that takes an image and outputs an image.
- As there is no need for a linear classifier on top, there is no need for a dense layer.
- We can design a fully convolutional network (FCN).

Fully Convolutional Networks

- Someone can imagine a network that looks like this:



- As there is no need for a dense layer, there is no need for downsampling. True?
- Absolutely not!

Fully Convolutional Networks

- Downsampling reduces the computational time, but this is not the main reason.
- The reason is related to a notion called the network receptive field.
- Let's ask a question: how a local operation like convolution see the whole picture.

RGB

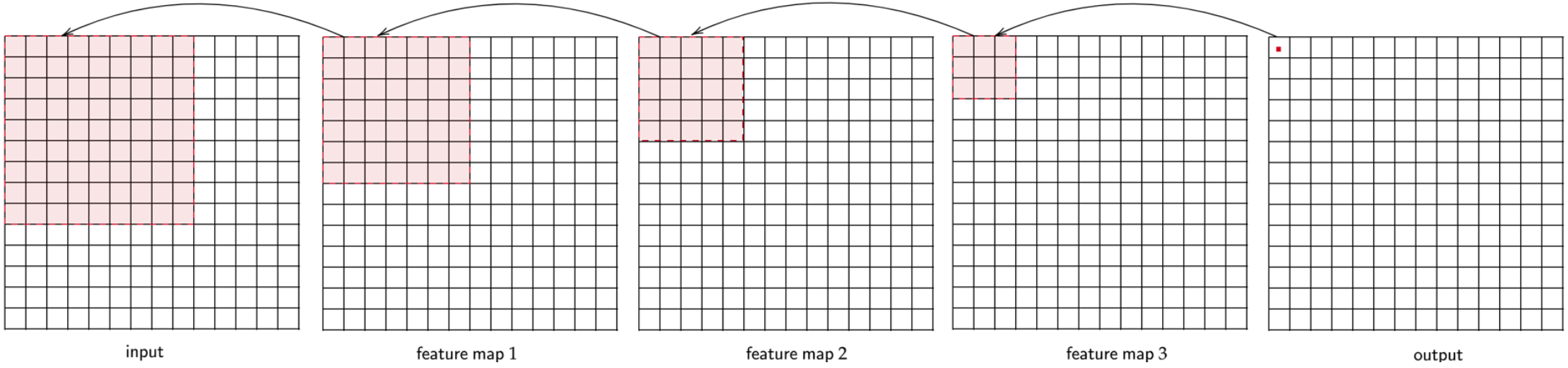
Segmentation Mask

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Receptive Field

- Receptive field is a notion related to convolutional operations.
- It is the size of a region in the input that contributes to a single pixel in the output.



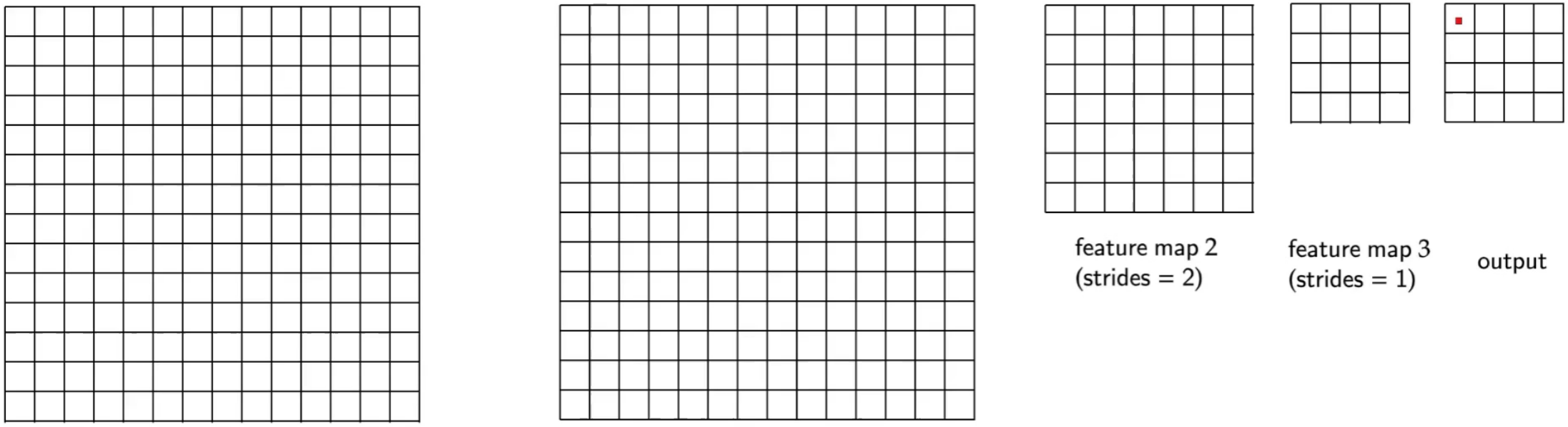
- Each pixel in a layer is connected to a 3x3 pixels in the previous layer.

Receptive Field

- To increase the receptive field, we can:
  - Either increase the kernel sizes.
  - Either add more layers.
- Both propositions will increase the number of parameters and operations.
- We would like to have something more efficient.
- Here's come down-sampling into play.

Receptive Field

- Down-sampling can exponentially increase the receptive field.



Receptive Field

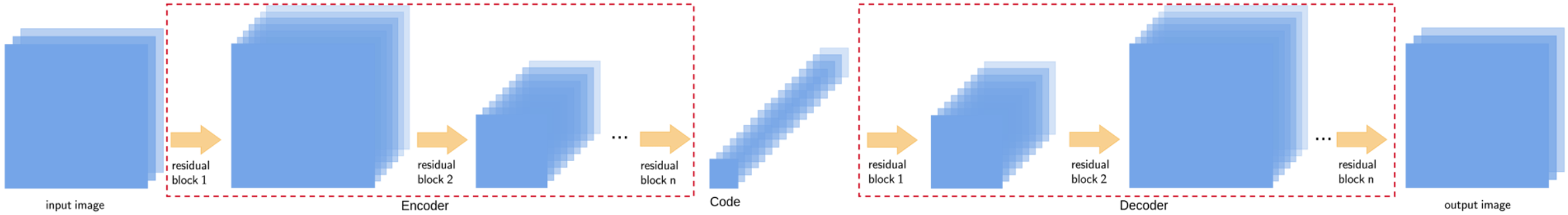
- It is a win-win situation because down-sampling means:
  - Faster training because the feature spatial size is reduced.
  - Bigger receptive field.
- Down-sampling can be implement with pooling or strided convolution.
- The receptive field can be bigger than the input image itself.
- Thus, image-to-image network uses down-sampling to increase their receptive field.
- However, to have an image in the output, we need to upsample. How?

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Encoder-Decoder Architecture

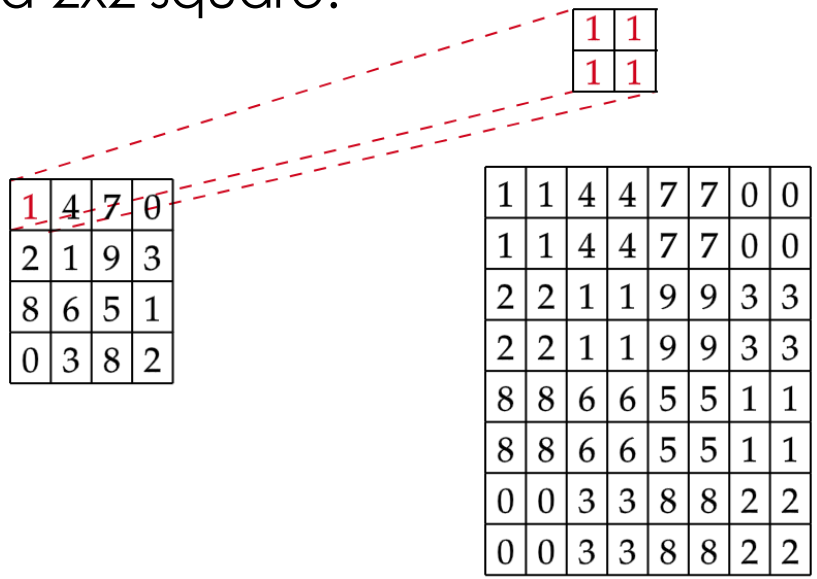
- Encoder-Decoder encodes an image into a "code" and then decode it back.



- We know how to downsample, but how to upsample?
- There is 2 ways:
  - Interpolation.
  - Transpose convolution.

Encoder-Decoder Architecture

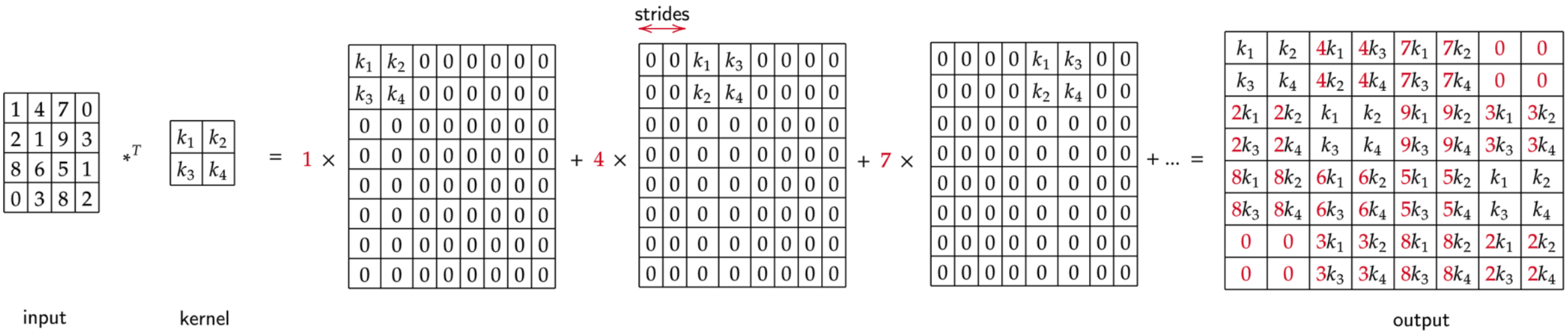
- The most used interpolation in deep learning is nearest interpolation.
- To double the image size just repeat the same value in a 2x2 square.



- It is similar to the idea of down-sampling by pooling.
- Transpose convolution is more similar to strided convolution.

Encoder-Decoder Architecture

- Transpose convolution takes the same parameters as a normal convolution.
- It is usually used only for up-sampling, with a kernel of 2x2 and strides of 2.



Encoder-Decoder Architecture

- A encoder residual block is just similar to the residual block of a CNN.
- We can down-sample using strided convolution (learnable) or pooling.

```
def encoder_residual_block(x, feature_maps, kernel_size, drop_rate, learnable_sampling):
    if learnable_sampling:
      x = conv_bn_relu(x, feature_maps[-1], strides=2, kernel_size=1)
    else:
      x = tf.keras.layers.MaxPool2D(padding='same')(x)
    residual = x
    if x.shape[-1] < feature_maps[-1]:
      additional_channels = feature_maps[-1] - x.shape[-1]
      paddings = [[0, 0], [0, 0], [0, 0], [0, additional_channels]]
      residual = tf.pad(residual, paddings)

    if drop_rate > 0:
      x = tf.keras.layers.SpatialDropout2D(drop_rate)(x)
    for i in range(len(feature_maps)):
      x = conv_bn_relu(x, feature_maps[i], strides=1, kernel_size=kernel_size[i])
    return tf.keras.layers.Add()([x, residual])
```

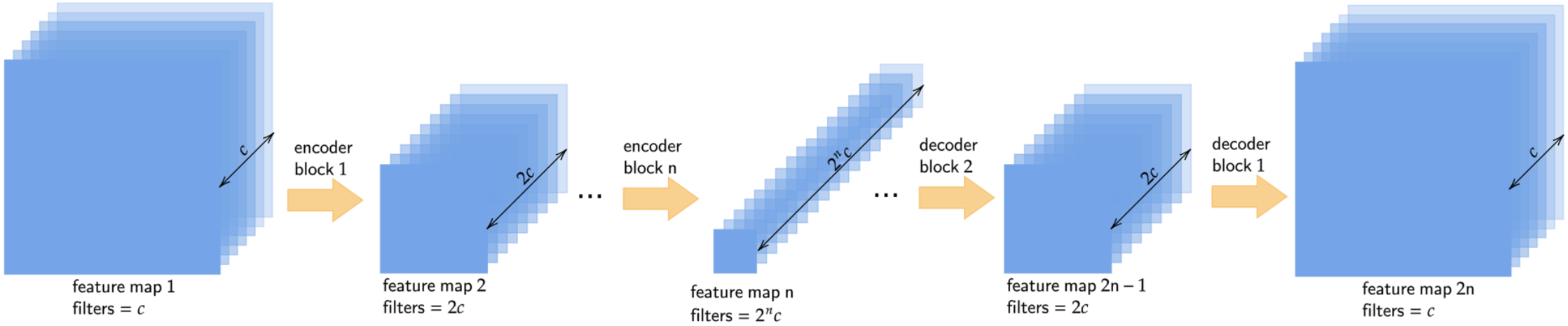
Encoder-Decoder Architecture

- We can implement a decoder residual block similarly to the encoder block.

```
def decoder_residual_block(x, feature_maps, kernel_size, drop_rate, learnable_sampling):
    if learnable_sampling:
      x = tf.keras.layers.Conv2DTranspose(filters=feature_maps[-1],
                                          strides=2,
                                          kernel_size=2)(x)
    else:
      x = tf.keras.layers.UpSampling2D()(x)
    residual = x
    if x.shape[-1] != feature_maps[-1]:
      residual = conv_bn_relu(x, feature_maps[-1], strides=1, kernel_size=1)

    if drop_rate > 0:
      x = tf.keras.layers.SpatialDropout2D(drop_rate)(x)
    for i in range(len(feature_maps)):
      x = conv_bn_relu(x, feature_maps[i], strides=1, kernel_size=kernel_size[i])
    return tf.keras.layers.Add()([x, residual])
```

Encoder-Decoder Architecture

- Encoder-Decoder is computationally efficient, in addition to its big receptive field.
- A common practice is to double the filters each time we reduce the spatial size.

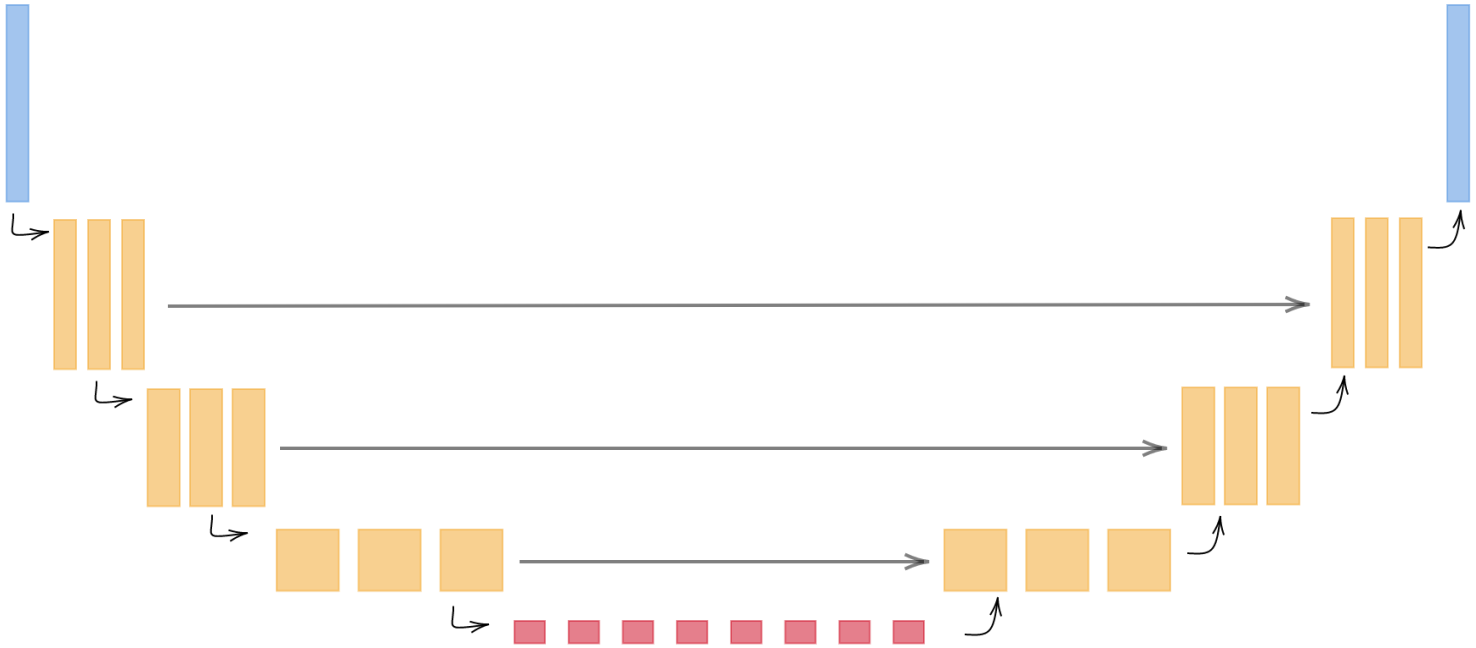


Encoder-Decoder Architecture

```
input_shape = (320, 320, 3)

encoder_fmaps = [[8, 8], [16, 16], [32, 32], [64, 64], [128, 128]]
encoder_kernels = [[3, 3], [3, 3], [3, 3], [3, 3], [3, 3]]

decoder_fmaps = [[128, 128], [64, 64], [32, 32], [16, 16], [8, 8]]
decoder_kernels = [[3, 3], [3, 3], [3, 3], [3, 3], [3, 3]]

code_fmaps = 256
code_kernel = 5
code_stride = 1

drop_rate = 0
learnable_sampling = False

input_image = tf.keras.layers.Input(input_shape)
encoded = input_image
for i in range(len(encoder_fmaps)):
  encoded = encoder_residual_block(encoded, encoder_fmaps[i], encoder_kernels[i], drop_rate, learnable_sampling)

code = conv_bn_relu(encoded, code_fmaps, code_stride, code_kernel)

decoded = code
for i in range(len(encoder_fmaps)):
  decoded = decoder_residual_block(decoded, decoder_fmaps[i], decoder_kernels[i], drop_rate, learnable_sampling)

output_image = tf.keras.layers.Conv2D(filters=3, kernel_size=3, strides=1, padding='same')(decoded)

model = tf.keras.Model(input_image, output_image)
model.summary()
```

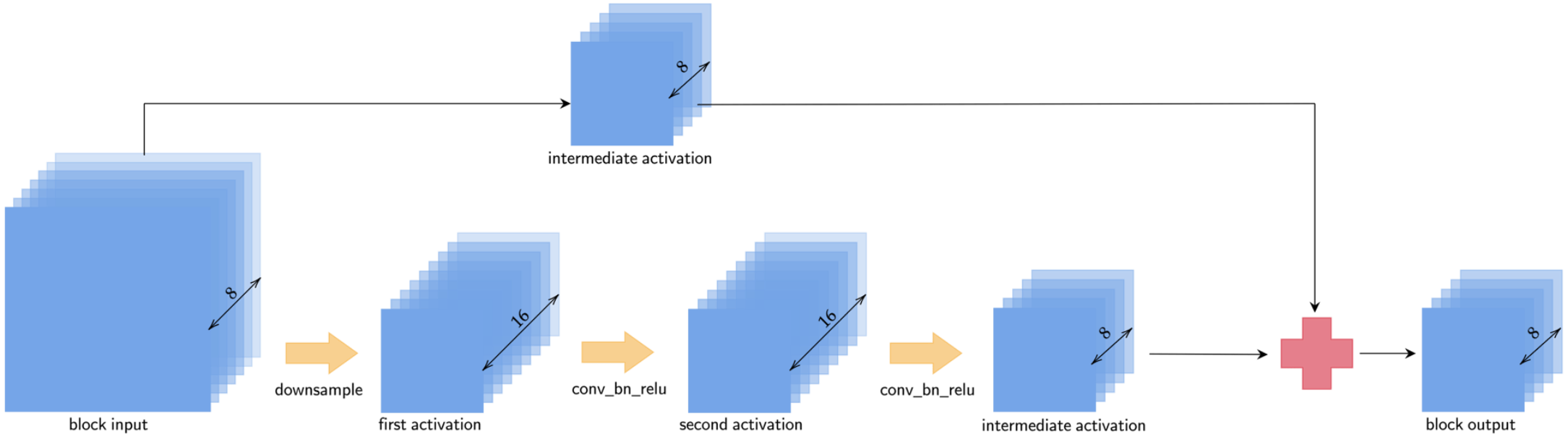
Encoder-Decoder Architecture

- Another development that improved the architecture performance is UNet.
- The idea is to connect encoder to the decoder to ease the reconstruction step.



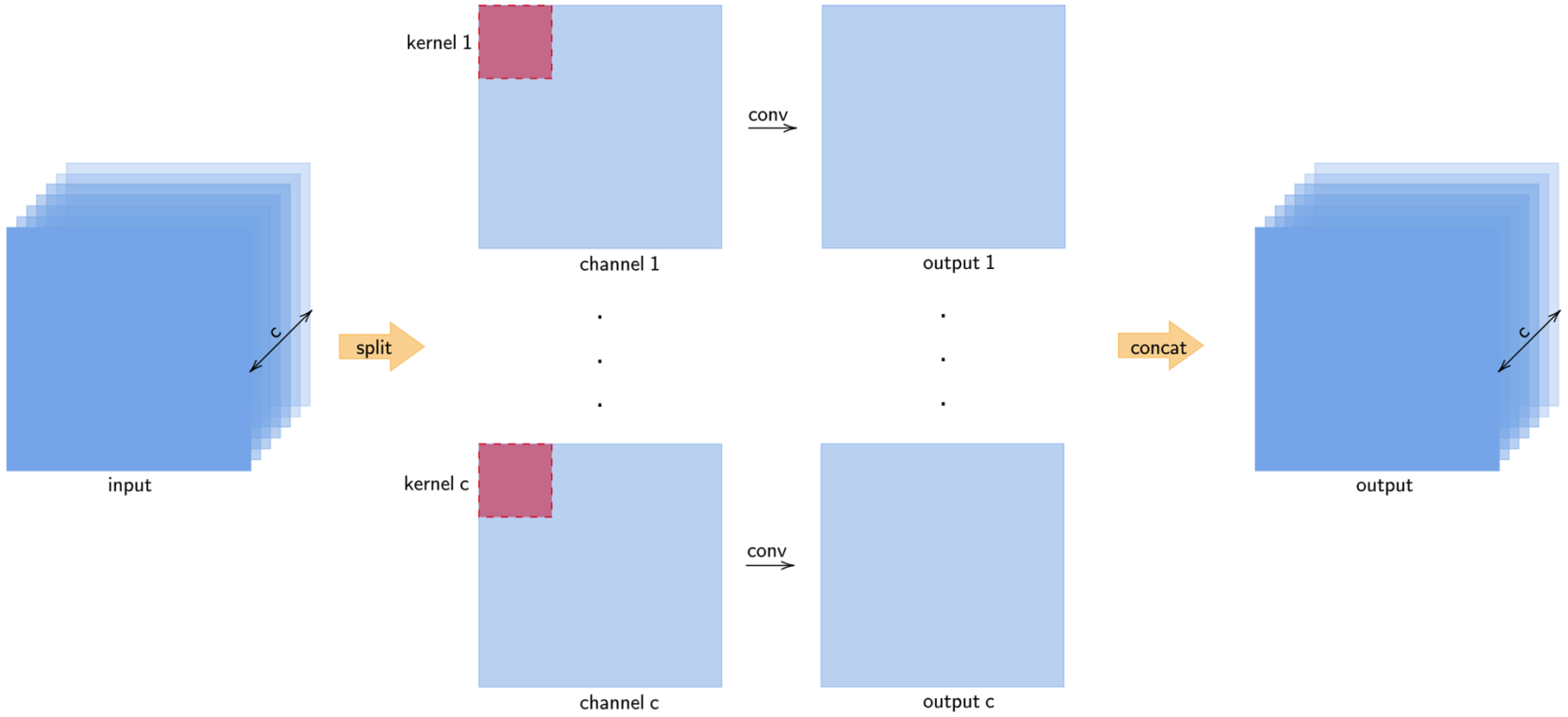
Encoder-Decoder Architecture

- Some tricks were developed for limited resources environment like mobile phones.
- In a network called MobileNet, they introduced "inverted residual block".
- The goal is to reduce the number of parameters of a network.



Encoder-Decoder Architecture

- A special type of convolution was developed for limited resources environment.
- It is called depthwise convolution and it reduces the size of operations of a model.



Encoder-Decoder Architecture

- The number of parameters of a depthwise convolution equals $k_x * k_y * c$.
- We can add a multiplier d (how many filters to apply), thus it becomes: $k_x * k_y * c * d$.

```
def depthwise_conv_bn_relu(x, depth_multiplier, strides, kernel_size):
  x = tf.keras.layers.DepthwiseConv2D(
          depth_multiplier=depth_multiplier, kernel_size=kernel_size,
          strides=strides, padding='same')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.ReLU()(x)
  return x
```

Outline

- Introduction
- Convolution
  - Image Processing
  - Convolution in Neural Networks
  - Convolution vs Dense Layers
- Convolutional Neural Networks
  - Residual connections
  - Batch normalization
  - Dropout
- Fully Convolutional Networks
  - Receptive Field
  - Encoder-Decoder Architecture
- Transfer Learning

Transfer Learning

- Convolutional networks learn generic features about images.
- We extract different levels of details in different layers.
- It was shown empirically that early layers learn generic image characteristics (edge ...)
- On the other hand, later layers are more specialized about the task in hand.
- Thus, we can think of using knowledge learned from a dataset into another.
- This is known as transfer learning.

Transfer Learning

- It is useful in scenarios when we have limited data to learn from.
- In this case, we do the following:
  - We train (or use existing) model on a big and diverse dataset.
  - We finetune the model weights on the dataset of interest in hand.
- Those 2 datasets can be very different in practice.
- Finetuning means updating some or all model layers on the small dataset.
- In the literature, almost all new models are trained on ImageNet for benchmarking.
- ImageNet is a huge dataset for image classification (14M, with 22K classes).

Transfer Learning

- New developments use self-supervised learning to prepare pretrained models.
- Self-supervised learning aims to train an unsupervised dataset in a supervised manner.
- The idea is to augment each image to create a positive pair.
- All other images are supposed negative pairs.
- This helps to overcome the data scarcity and can enable deep learning for such cases.
- This also help in unsupervised application like clustering.
- We can use those network to output a meaningful representation for an image.

Transfer Learning

- Another result of this generic learning of convolution is to do multitask learning.
- A multitask network can predict different but related quantities (like age/gender).