Image classification

Deep learning - BMDC 2025-2026

Gheorghe Cosmin Silaghi

Universitatea Babeș-Bolyai

October 14, 2025

Overview

Short history

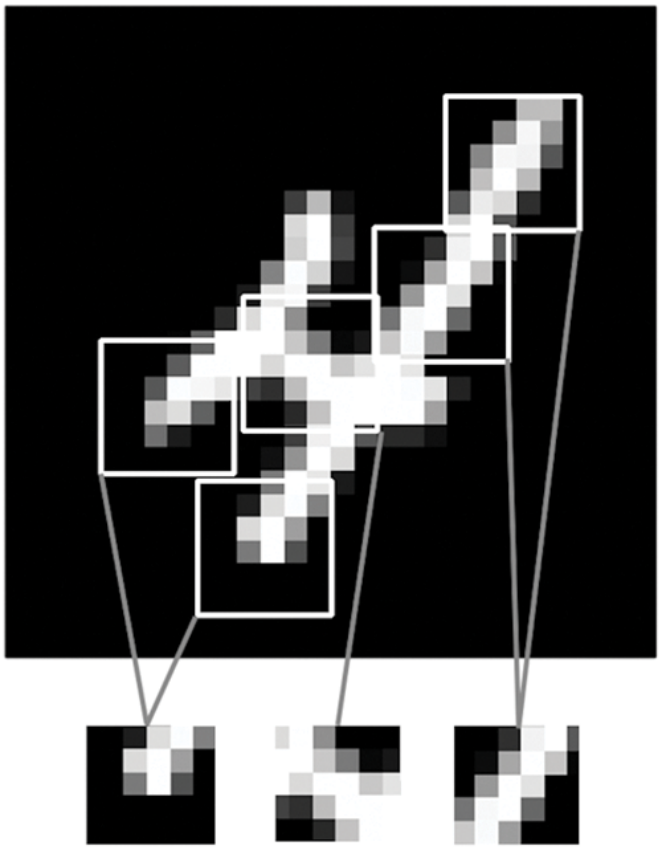
- The first big success story of DL
- Dan Ciresan used CNNs to win ICDAR 2011 and IJCNN 2011 competitions
- fall 2012: G. Hinton's group winning the ImageNet large scaole visual recognition challenge
- 2013-2014 DL still faced skepticism from many senior computer vision researchers
- 2016: DL becomes dominant in computer vision
- today we are interacting with DL-based vision models via Google Photos, Google image search, the camera on the phone, YouTube, OCR software etc.
- DL-based CV models are at the heart of autonomous driving, robotics, AI-assisted medical diagnosis, autonomous retail checkout systems, autonomous farming etc.

Introduction to CNN

- the first CNN to classify MNIST digits
- Conv2D takes as input sensors of shape `(image_height, image_width, image_channels)`.
- The output of every Conv2D plus MaxPooling2D layers is a 3D tensor of shape (height, width, channels)
- ! some DL libraries flip the location of channels in the image tensors: you would pass (channels, height, width) - e.g. the PyTorch ecosystem
- in Keras this is configurable:
  `keras.config.set_image_data_format("channels_first")`

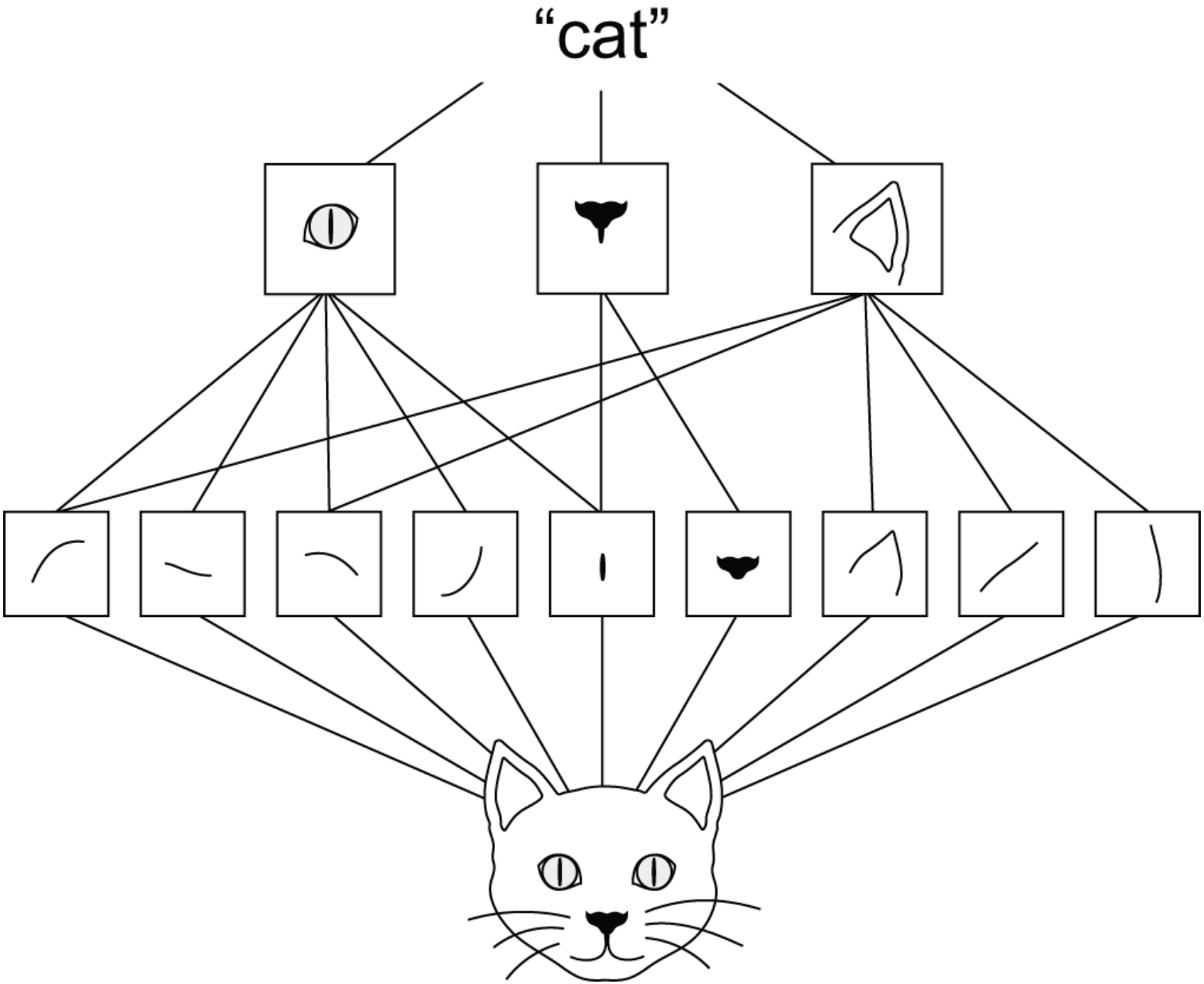
The convolution operation

- Dense layers learn global patterns in their input space (i.e. a MNIST pattern involving **all** the pixels in the image)
- the convolution layers learn local patterns, in our case found in a small 2D 3x3 pixels window from the input



Key characteristics of ConvNets

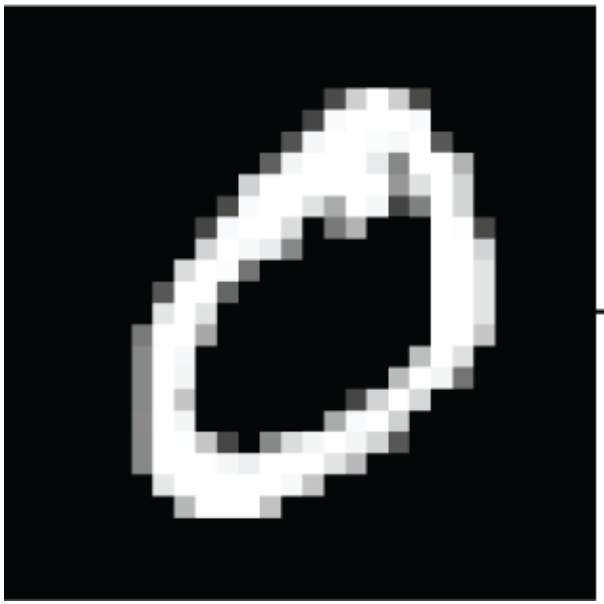
- *The patterns they learn are translation invariant*. After learning a pattern in the lower-right corner of a picture, the ConvNet can recognize it anywhere - *the visual world is fundamentally translation invariant*
- They can learn *spatial hierarchies of patterns*: the first layer will learn a small local pattern, such as edges. The second layer will learn larger patterns made of features produced by the first layer etc - *the visual world is fundamentally spatial hierarchical*


“cat”

How convolutions operate?

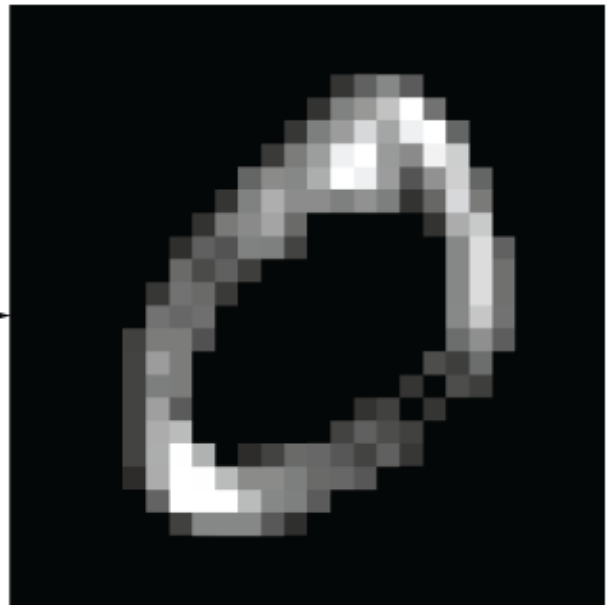
- takes as input a feature map with two spatial axes and a depth axis.
- the convolution operation extracts patches from the input feature map and applies some transformation to all of these patches and produce an output feature map.
- the output depth its arbitrary: is a parameter of the convolution. they stand for **filters** and encode specific aspects of the input data
- the Conv2D outputs a feature map of (26, 26, 64), computing 64 filters over its input. each of these 64 output channels is a reponse map of that filter over the input, the response of that filter pattern at different locations in the input

Original input



Single filter

Response map, quantifying the presence of the filter's pattern at different locations

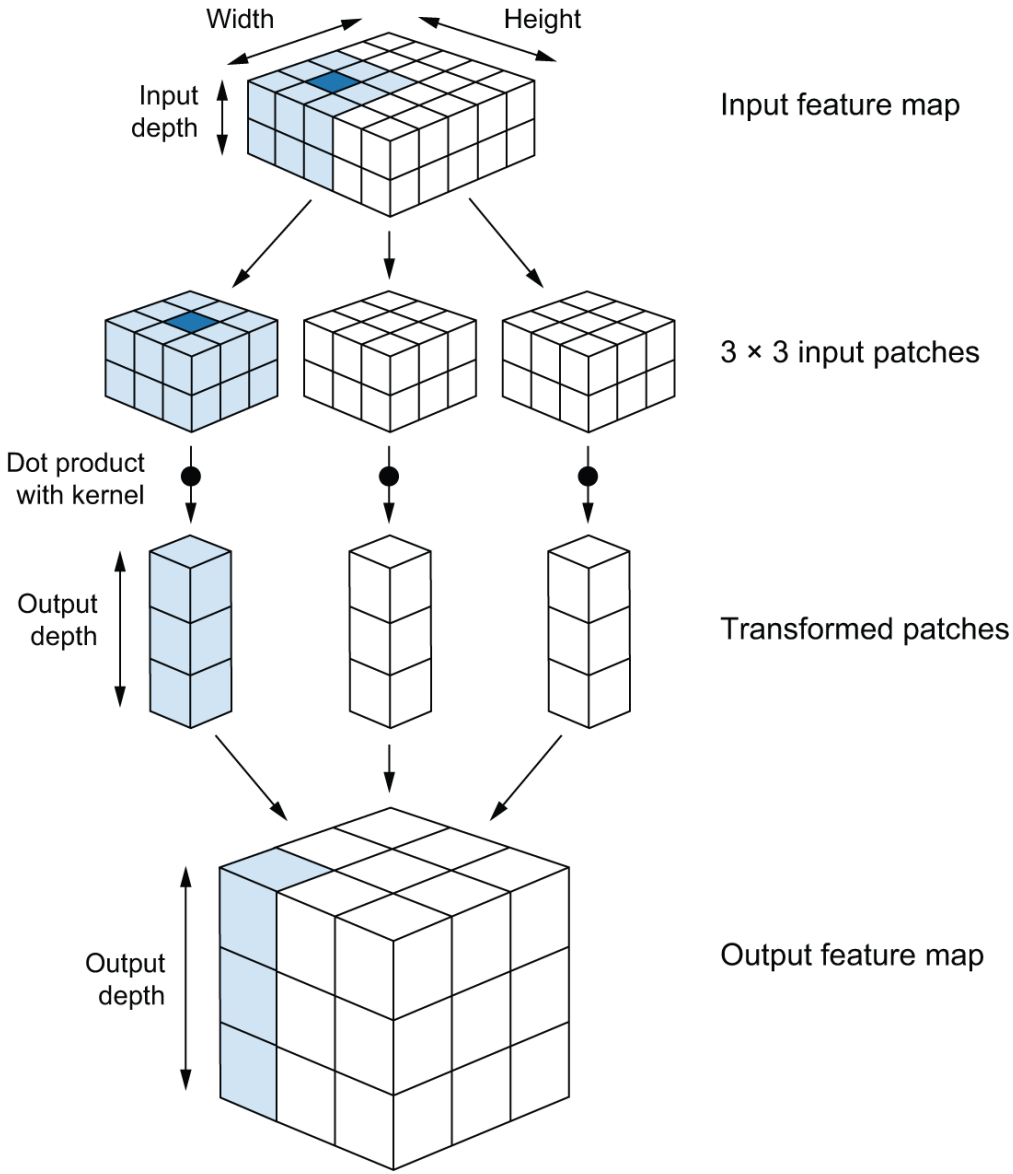


Conv2D parameters

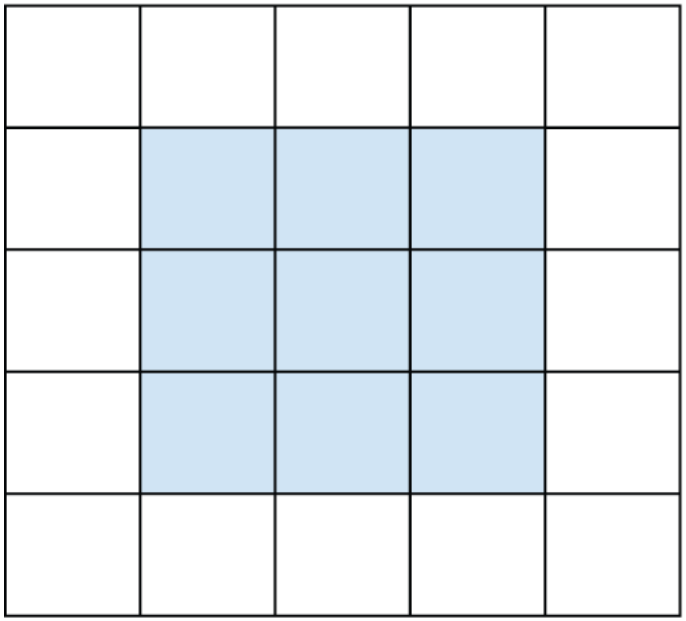
- *size of the patches* extracted from the inputs (typically 3x3 or 5x5)
- *depth of the output feature map* - number of filters computed by the convolution. in MNIST we started with a depth of 32 and ended with a depth of 64.

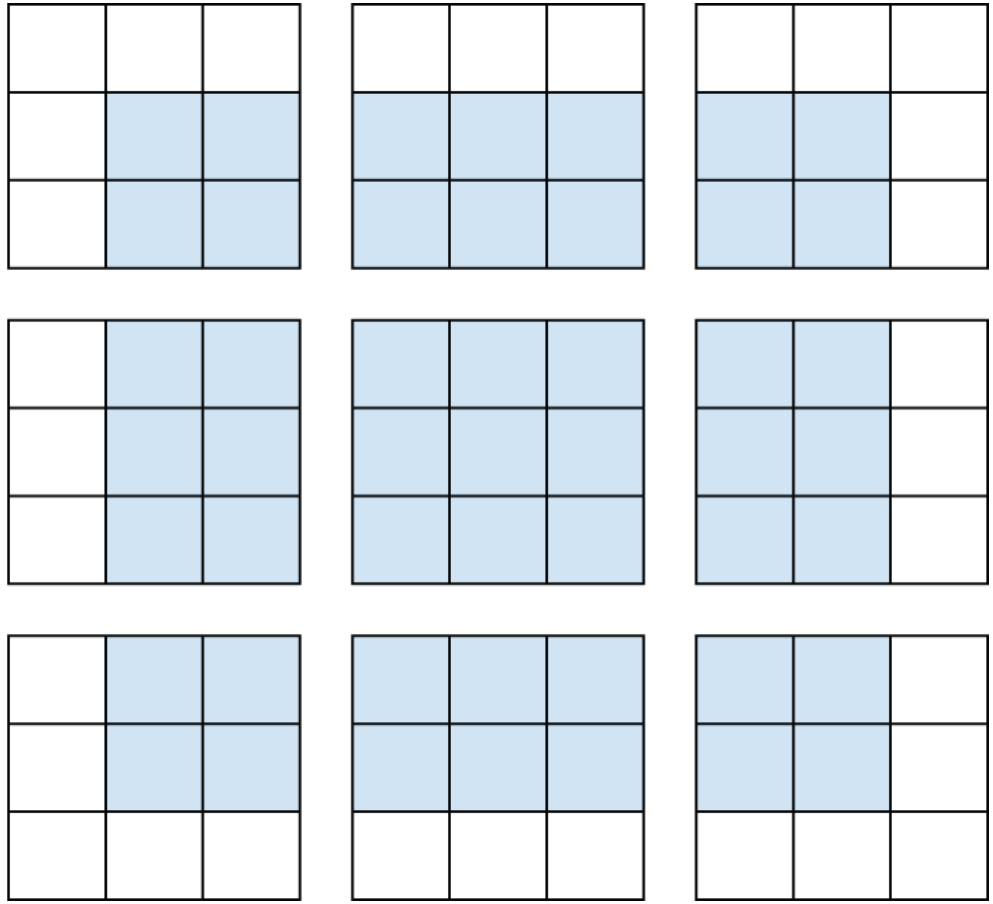
Functioning of the convolution operation

- the convolution works by sliding these windows of size 3x3 or 5x5 over the 3D input feature map, stopping at every possible location and extracting the 3D patch of the surrounding feature of shape (window_height, window_width, input_depth).
- next, each such 3D patch is transformed in a 1D vector of shape (output_depth) via the tensor product with the learned weights matrix (the convolution kernel) - the same kernel is reused across every patch
- all these vectors (one per patch) are spatially reassembled into a 3D output map of shape (height, width, output_depth).
- the vector $output[i, j, :]$ comes from the 3D patch $input[i-1 : i+1, j-1 : j+1, :]$


Width
Height
Input depth
Input feature map
3 × 3 input patches
Dot product with kernel
Output depth
Transformed patches
Output depth
Output feature map

The border effect

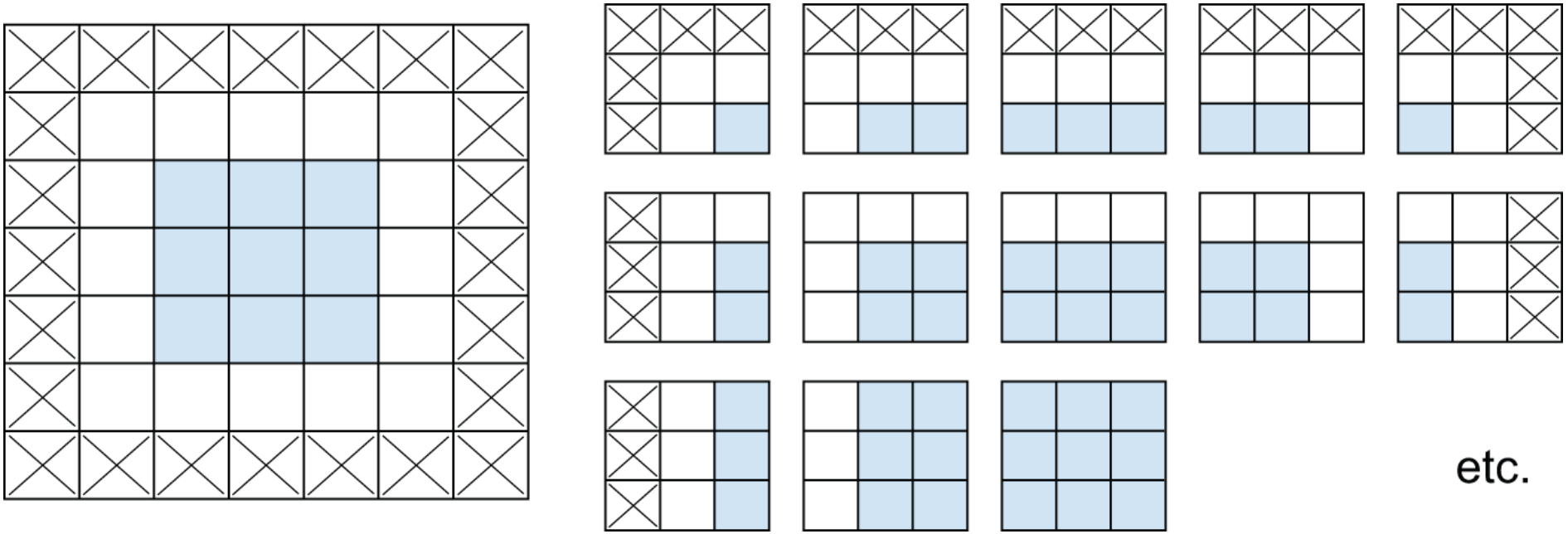




- by sliding a 3x3 window over a 28x28 image, we get a 26x26 output image.

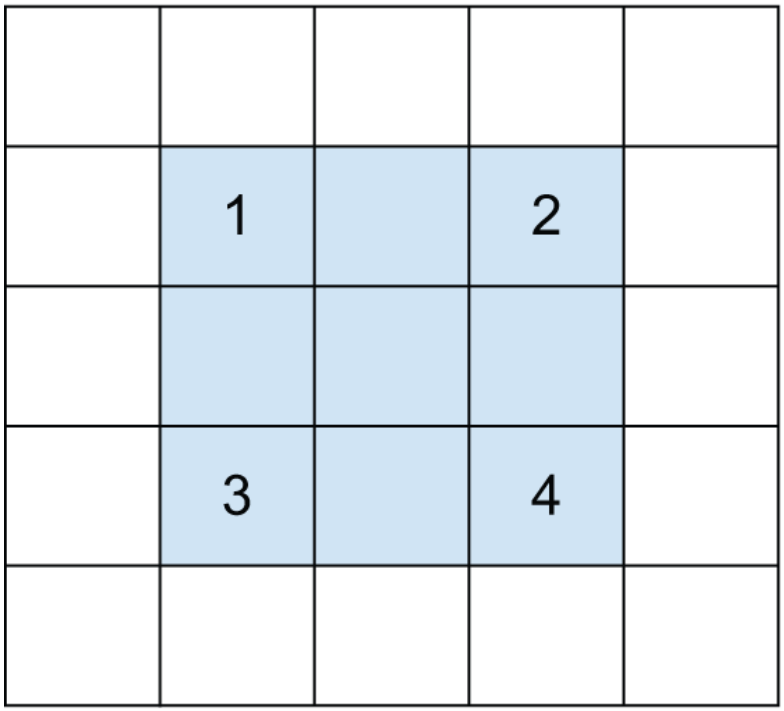
Padding

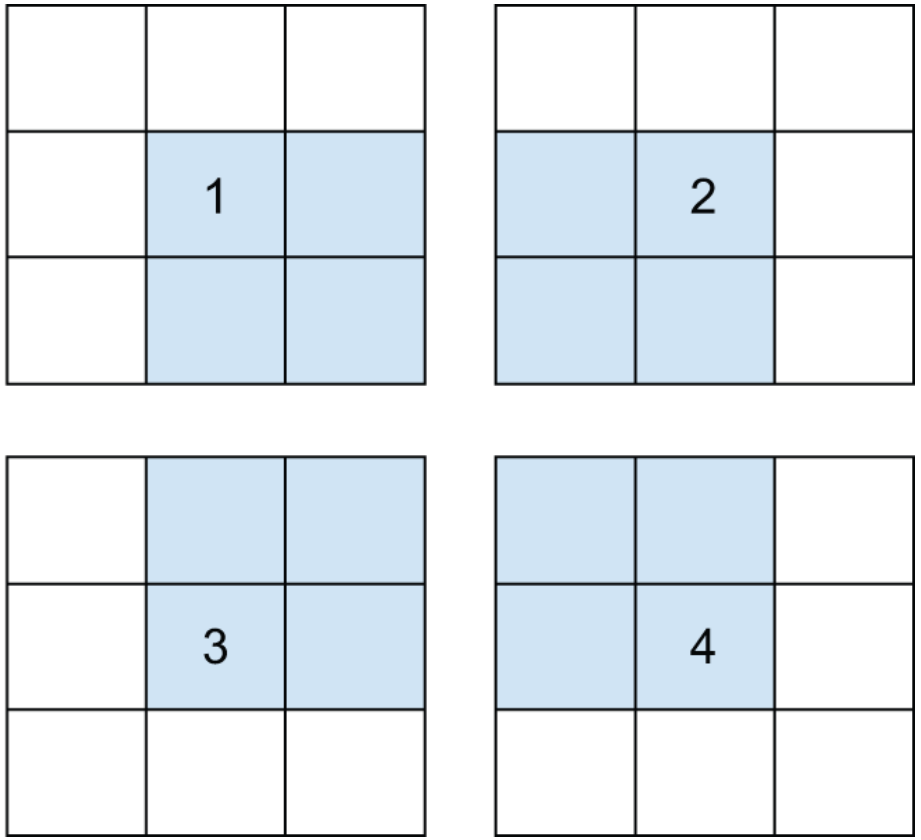
- if we want to get an output with the same spatial dimensions as the input, we can use *padding*: add the appropiate number of rows and columns each side of the input feature map so that is possible to fit centered convolution windows at every input tile.
- **padding** argument: **valid** (i.e. no padding) or **same**.



Convolution strides

- stride: the distance between two successive windows (default to 1).
- strided convolution: the strive is higher than 1.
- using strides=2 means that the width and height of the feature map are downsampled by a factor of 2.





The MaxPooling operation

- its role: to aggressively down-sample feature maps like strided convolutions
- extracts windows from the input feature maps and outputs the max value of each channel.
- is conceptually similar with a convolution, but rather than transforming local patches via a learned linear transformation, they are transformed via the hardcoded max operation.
- MaxPooling is done with 2x2 windows and stride 2, resulting a downsampled feature map by a factor of 2.

Why need for downsampling?

- reduce the size of the feature map such that to make the information they contain less spatially distributed and increasingly contained in the channels,
- induce spatial-filter hierarchies by making successive convolution layers to look at increasingly large windows

Why MaxPooling?

- stides=2, or AveragePooling also do downsampling
- features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map.
- it's more informative to look at the maximal presence of the different features than at their average presence.
- the most effective subsampling strategy is to produce a dense maps of features and then look at the maximal activation of the features over the small patches.

Training a ConvNet for Cats and Dogs dataset

- get the data from Kaggle
- understand the Dataset object
- train a ConvNet from scratch
- use **Augmentation** to regularize the model

Data augmentation

- generates more samples in the training set from the existing samples
- each examples is augmented by a number of random transformations that yield believable-looking images, such that at training, the model not to see the same example twice
- where to add the data augmentation layer:
  - at the start of the model, inside the model. right before the Rescaling layer
  - inside the data pipeline - outside the model. we could apply them to our Dataset object via map.
- data augmentation is a regularization techniques, therefore, it is not applied on the test/validation data.

What is a pre-trained model

- a model previously trained on a large dataset, typically on a large-scale image classification task
- if the original dataset is large-enough, then the spatial hierarchy of features learned by the pre-trained model can act as a generic model of the visual world
- its features can prove useful for many different computer vision problems, even if they imply different classes than those of the original task

Two ways of using a pre-trained model

1. feature extraction
2. fine tuning

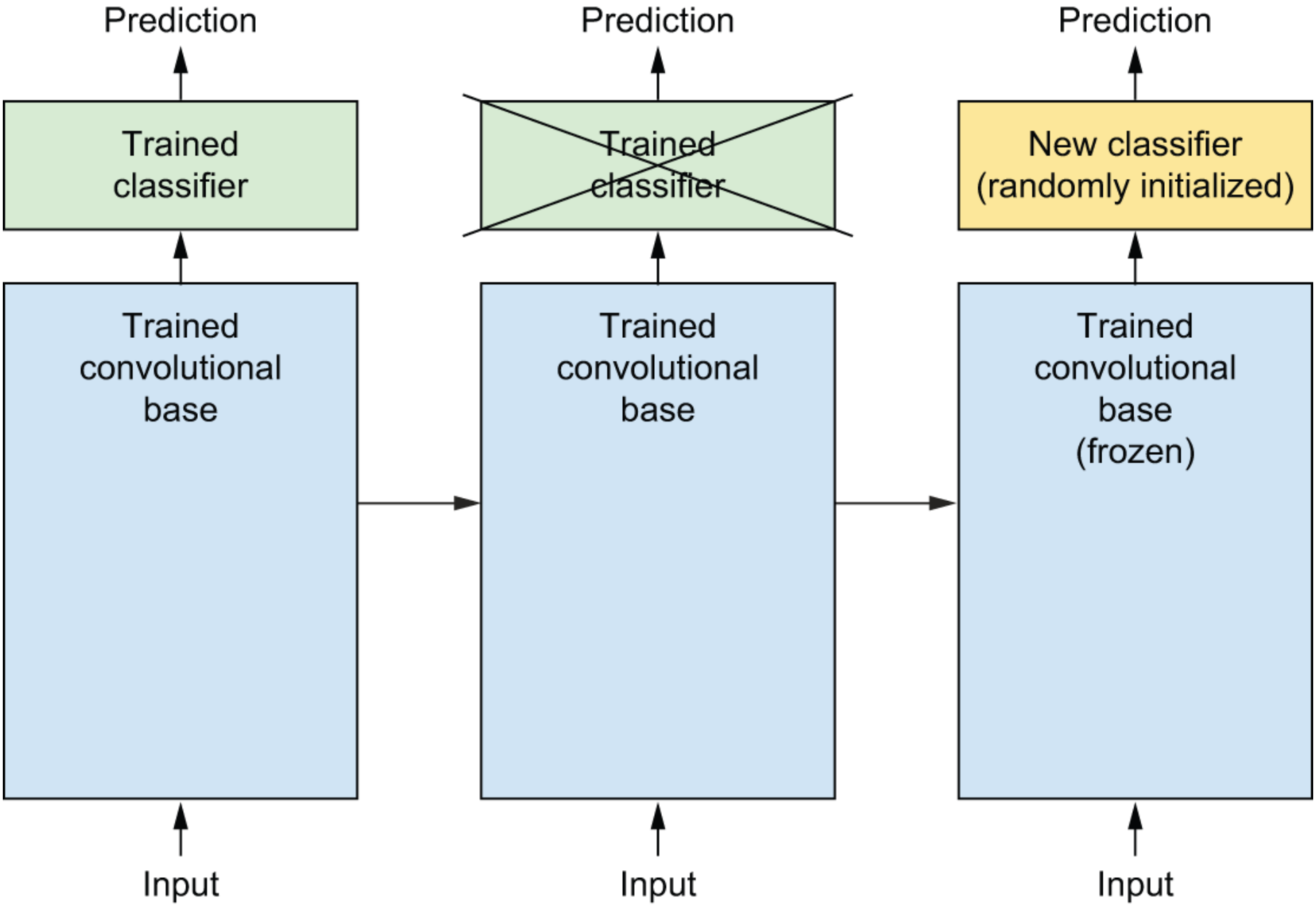
Feature extraction

- using the representations learned by a previsous trained model to extract interesting features from new samples
- these features are run through a new classifier, which is trained from scratch

Parts of a CV classifier:

- *convolutional base*: the successive Conv2D and MaxPooling layers
- the dense connected classifier

- *feature extraction*: take the convolutional base of a previously trained network, run the data through it and train a new classifier on top of the current output


Prediction
Trained classifier
Trained convolutional base
Input
Prediction
Trained classifier
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

- reusing the dense connected classifier should be avoided. why? representations learned by the convolutional base are likely to be more generic and therefore, more reusable
- the feature maps of ConvNets are presence maps of generic concepts over a picture, which are likely to be useful regardless of the computer vision problem at hand
- on opposite, representations learned bt the dense classifier are specific to the set of classes for which the network was trained
- layers that come earlier in the model extract local, highly generic feature maps,
- whereas layers that are higher up extract mode abstract concepts (such as cat ear, dog eye etc).

KerasHub library

- KerasHub contains implementations of popular pretrained model archictures, paired with pre-trained weights that can be downloaded to your machine
- Xception, ResNet, EfficientNet, MobileNet etc.
- ImageConverter layer: rescales our input to match the pretrained checkpoint

Options for feature extraction

Option 1

- run the convolutional base over your dataset and records the output to a NumPy array on the disk
- use these data to construct a standalone dense connected classifier

Option 2

- extend the convolutional base by adding dense layers on top
- running the whole thing end-to-end on the input data
- this option allows using data augmentation
- this technique is more expensive than the first one

Fine tuning

- unfreeze the frozen model based used in feature extraction, and jointly training both the newly added part of the model and the base model
- it slightly adjust the more abstract representations of the model to make them relevant for the problem at hand
- it's only possible to fine tune the convolutional base once the classifier on the top has been trained
- if the classifier is not already trained, then the error signal propagating through the network during training will be too large and the representations previosly learned by the fine-tuned layers will be destroyed

Steps for fine-tuning

1. add the custom network on the top of an already trained base network
2. freeze the base network
3. train the part you added
4. unfreeze the base network
5. jointly train both these layers and the part you added

you should not unfreeze the batch normalization layers!

Partial fine tuning

- you can choose to unfreeze and fine tune all the convolutional base.
- however, dealing with large pre-trained models you may unfreeze some of the top layers of the convolutional base and leave the lower layers frozen.

Why?

- earlier layers in the base encode more generic, reusable features, whereas layers higher up encode more specialized features. It is more useful to fine tune the specialized features, because these are the ones that need to be repurposed for the problem at hand
- the more parameters you are training, the more you are at risk of overfitting. The convolutional base has 15 mil. parameters, thus, it would be risky to attempt to train on a small dataset.