Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Lecture 5

Data Science and Computation PhD + Master in Bioinformatics

University of Bologna

Communications

Calendar: updates

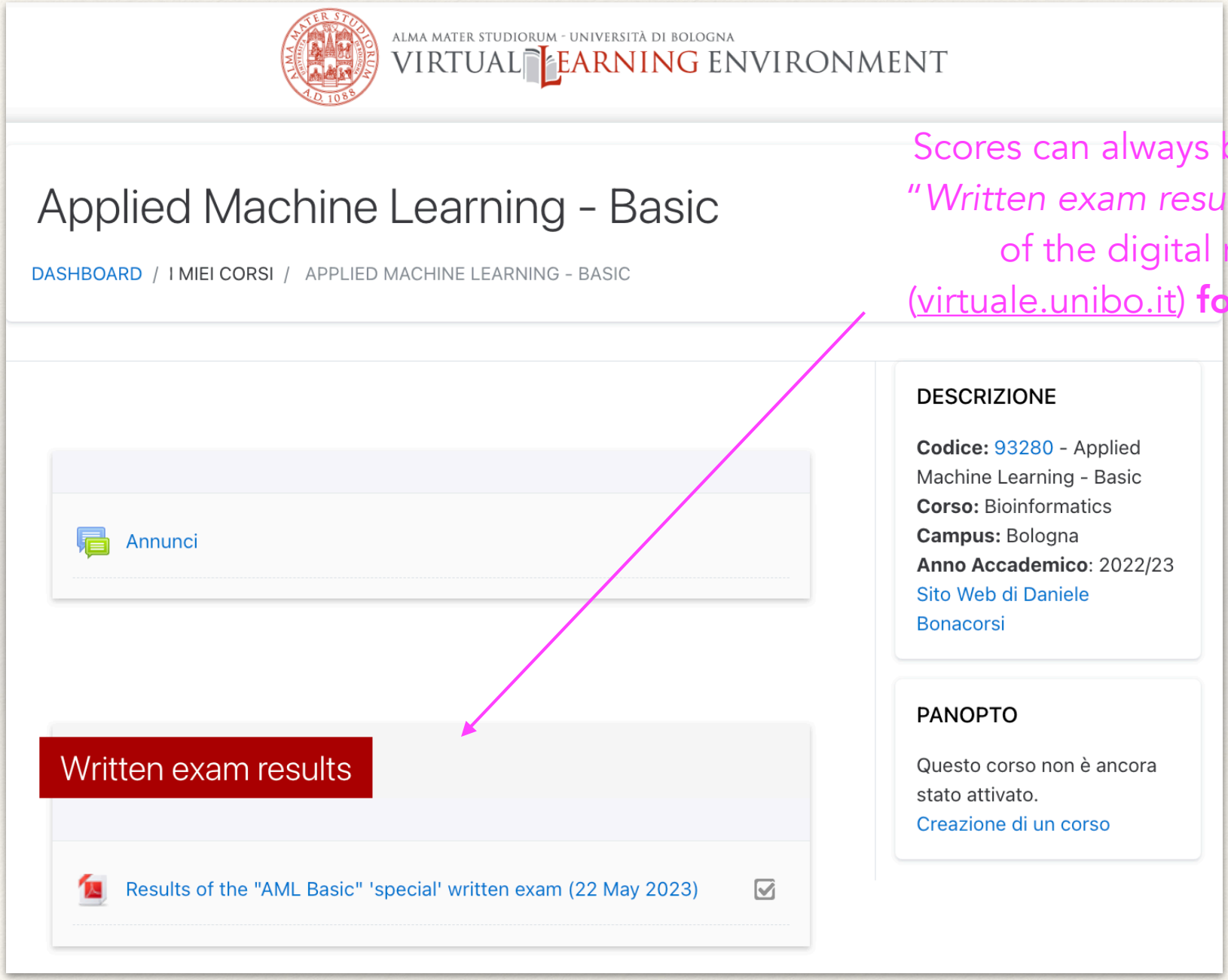
AML - Advanced: 4 May - 16 June

- 12 lectures → 9 * 4hrs + 4 * 3 hrs = 48 hrs → **6 CFU**

thu May 4 14-18 → DONE
fri May 5 14-18 → DONE
thu May 11 14-18 → DONE
fri May 12 14-17 → DONE
thu May 18 14-18 → cancelled
fri May 19 14-17 → cancelled
thu May 25 14-18 → week in progress
fri May 26 14-17 → week in progress
thu Jun 1 14-18
thu Jun 8 14-18
thu Jun 9 14-18
Thu Jun 15 14-18
fri Jun 16 14-17

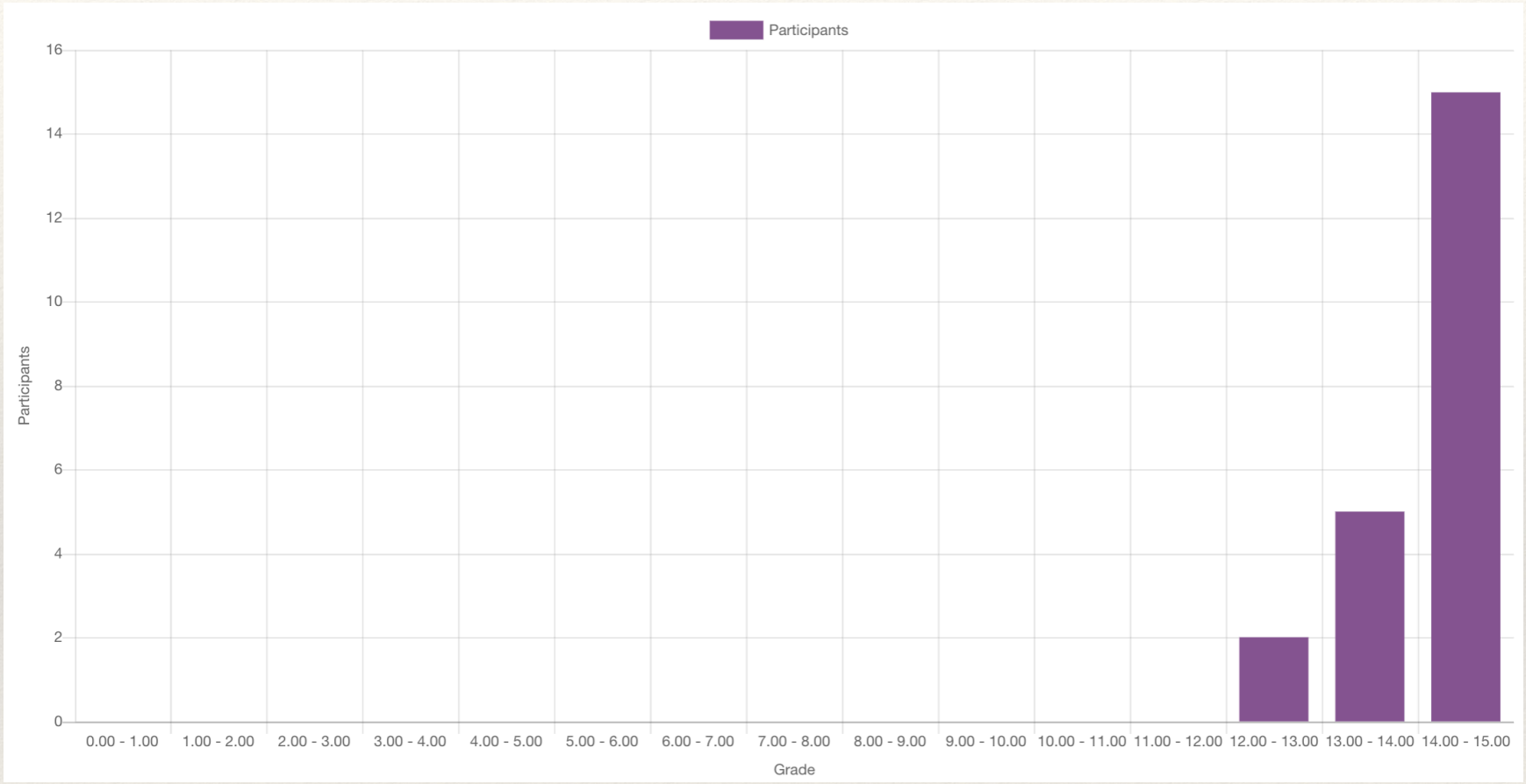
Dates are susceptible to change:
in case, you will be notified in advance.

Where to find **written exam results**



Scores can always be found in the "*Written exam results*" sub-section of the digital repository (virtuale.unibo.it) **for each module**

Results of the AML Basic "prova in-itinere"



Good! → Brief discussion at the lecture (see next)

Some comments

Few remarks:

- Results are, on average, **good and encouraging**
- No exam correction (explanation at the lecture), but write me if you want to know more about your errors
- "*How should I perceive my score?*"
  - Better to have a score than nothing, but remember this was a 'special' written exam date, so if you did not attend, no worries, there will be dates in Jun, Jul, Sep 2023, then Jan, Feb 2024.. Perhaps, even additional exam dates..
  - If you got what you expected, good - then perhaps you are confident enough to know what to do..
  - If you got a higher-than-expected score, be happy, and/or try another test to make sure this was not an overestimate (you have nothing to loose)
  - If you got a lower-than-expected score, no panic: study more and attend another test (it might be an underestimate, or not..)
- For the future, tune yourself with my feedback and my desire to change a bit, soon..: **this test was easy, with very basic questions selected, plenty of time to complete it (recorded even a <7 mins completion), so plenty of time to recheck, you had notes available, no penalties applied..** stay tuned!

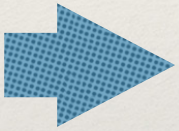
NNs - building blocks

(cont'd)

Our first NN

(looked at for the second time, though)

Instant recap on the notebook.



More on: Training a NN

Training an ANN

Let's focus again (and more) on this question:

- How do you compute the weights. I.e. **how do you train a NN?**

A parenthesis

NNs are a pretty old class of algorithms that have been originally inspired by one goal: **build a machine that can mimic the brain**.

We need to get a sense of how this basic logic drove their conceptual development, though, what we can expect them to ultimately do.

Coding the brain

A divagation..

[*] CAVEAT: not intended to be a neuroscience-compliant explanation!

The origins of NNs is as algorithms that try to mimic the brain.

Think about the senses. The brain can learn to process images (= see), process acoustic waves (= hear), process other signals (e.g. sense of touch)… quite complex! and different tasks! and you are not even close to the end of the list of what brains can do (e.g. think?)

If you mimic by "coding", it is - if you want to mimic the brain - as if you need to write lots of different pieces of software… or, you just make a hypothesis. You speculate that the way the brain manages to do all of these different things is not worth like a thousand different software programs, but instead encoding **a single learning algorithm** [*].

Brain = one single learning algorithm? A divagation..

[*] CAVEAT: not intended to be a neuroscience-compliant explanation!

The "**one single learning algorithm**" hypothesis is currently just an hypothesis. But it is not with 0 evidence.

E.g. auditory cortex. You hear voices because your ears mechanically get a vibration and route this signal to the auditory cortex that process it as a sound signal. Same for visual, or somatosensory brain areas.

Neuroscientists have found out that if one "cuts the wire" from the ears to the auditory cortex, the brain re-wires ("neuro-rewiring experiments"): e.g. the signal from the eyes to the optic nerve eventually gets routed to the auditory cortex. If you do it turns out that the auditory cortex somehow "learns to see". To a surprisingly large extent, it seems somehow as if we can plug in almost any sensor to almost any part of the brain and the brain will learn to deal with it [*].

In summary

So, if the same piece of physical brain tissue may be able to process sight or sound or touch, then maybe there is just one learning algorithm behind, that can process any of them.

So - if the former is true - in trying to emulate the brain, instead of implementing a thousand different algorithms, maybe what we need is to figure out **some approximation of whatever the brain's one-and-only-one learning algorithm is, and implement it**.

This might be our best shot at making real progress towards the AI dream of building someday truly intelligent machines.

But note: **the NN logical/AI-goal motivation behind is not the reason why we do use them; we do use them because they work!**

- but it is nice to keep a window open to what this might imply..

Training a **MLP**: the **BP algo**

For many years, researchers struggled to find a way to train MLPs.

1986, Rumelhart/Hinton/Williams introduced the **back-propagation** (**BP**) **training algorithm** [Ref-RumelhartHintonWilliams] - still used today

- it is simply GD using an efficient technique for computing the gradients automatically
- call "BP" because it does two **propagation**s through the network (one forward, one **back**ward), and is able to compute the gradient (*) of the network's error wrt every single model parameter. I.e. it can find out how each connection weight and each bias term should be tweaked in order to reduce the network error
  - Automatically computing gradients is called **automatic differentiation** (**autodiff**). Various techniques exist, BP uses **reverse-mode autodiff**. It is fast and precise, and is well suited when the function to differentiate has many variables (e.g. connection weights) and few outputs (e.g. one loss)
- once it has these gradients, it just performs a regular GD step
- the whole process is iteratively repeated - in **epochs** - until the network converges

So… GD vs BP ?!

Which is the difference between GD and BP algorithms, **in simple terms**?

Think of the hiker example.

GD: the big hill represents the error or loss function of a problem. The goal is to find the lowest point on the hill (*which corresponds to the minimum error*). GD is like the hiker following a path towards the lowest point on the hill, by taking small steps in the direction of the steepest slope, and hence he gets down-hill… eventually the goal would be to reach the bottom of the hill (*which represents the minimum error*). One path, executed just once, might work but might not be the optimal, though.

BP: (in a NN only) it is a way for the N to figure out which direction the hiker had better go, ultimately (*the NN needs to know how each weight and bias in the NN contributes to the error*). It's like a trail of breadcrumbs left by the hiker on its way, moved and "tuned" again in the reverse direction now, going back up-hill. The hiker starts at the bottom of the hill and leaves breadcrumbs along the way as it climbs up (they represent the gradients, which tell the NN how much each weight and bias needs to change to decrease the error). By following the adjusted breadcrumbs, at the next GD step, the hiker can follow a better path (*i.e. the NN can adjust its weights and biases using GD to eventually reach the minimum error in next rounds*).

In brief, **GD is the process of taking small steps to find the lowest point** (i.e. an optimization algorithm that, in a NN, iteratively updates weights/biases based on the computed gradients), while **BP is the process of figuring out how each step (change in weights/biases) should be in order to reach the actual lowest point** (i.e. the algorithm used to calculate the gradients of the error function with respect to weights/biases, using the chain rules of calculus, enabling GD to perform the weight updates).

- In training a NN, they are both needed

Training a **MLP**: the **GD+BP algo** (in one sentence)

GD+BP algo in one sentence:

For each training instance, the BP algo first makes a prediction (**forward** pass), measures the error, then goes through each layer in reverse to measure the error contribution from each connection (**backwards**/reverse pass), and finally slightly tweaks the connection weights to reduce the error (in a GD step).

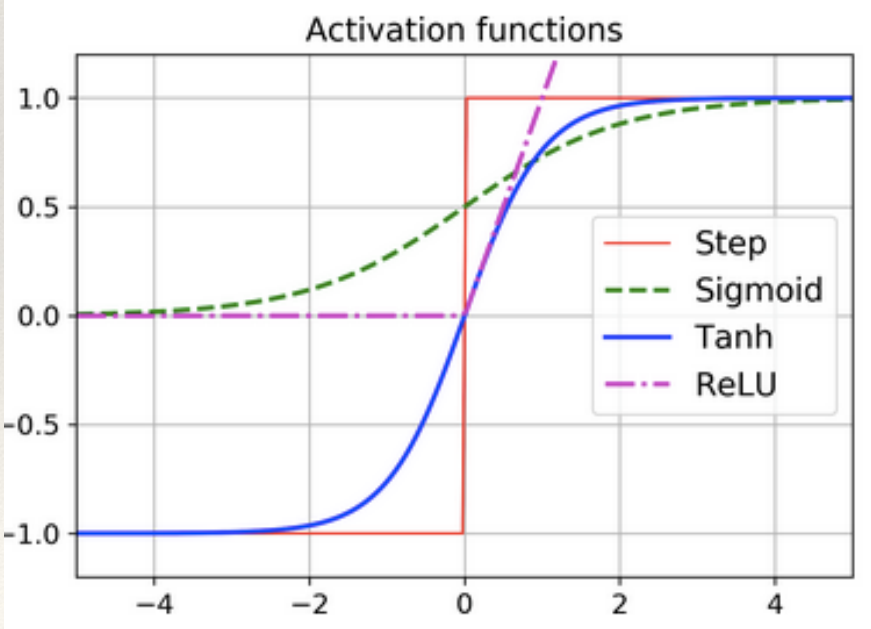
Training a **MLP**: the **GD+BP algo** (more details)

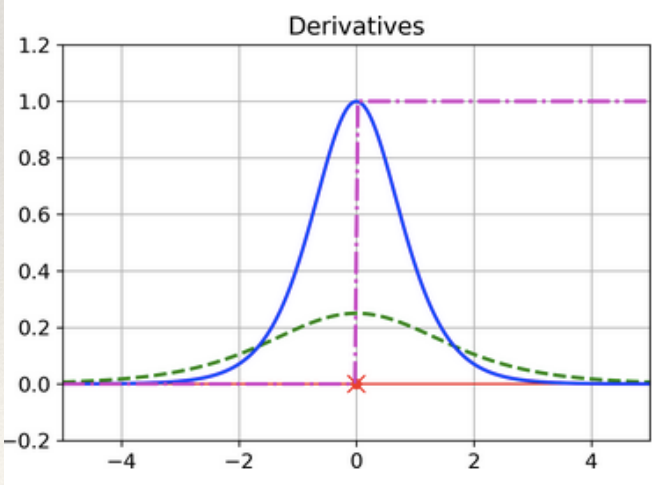
- it divides the training set in mini batches. It handles one mini-batch at a time (with e.g. 32 instances each). It goes through the full training set multiple times. Each pass on all of it is called an **epoch**
- each mini-batch is passed to the network's input layer, which just sends it to all units of the first hidden layer
- for every instance in this minibatch, BP computes the output of all the neurons in this layer
- the result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the output layer
- This is the **forward pass**: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- BP measures the **network's output error** (i.e. using a loss function that compares the desired output and the actual output of the network, and returns some measure of the error)
- BP computes how much each **output** (layer) connection contributed to the error. This is done analytically by applying the (calculus) chain rule (which makes this step fast and precise)
- BP then measures how much of these error contributions came from each connection in the layer below, again using the chain rule - and repeat this until the algorithm reaches the input layer.
- This is the **backwards pass**. This reverse pass efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network
- Finally, BP performs a GD step to tweak all the connection weights in the network, using the error gradients it just computed

BP and **activation functions**

Choice of the **activation functions**

- the BP authors made a key change to the MLP architecture: they replaced the step function with the **logistic** function, **σ(z) = 1 / (1 + exp(–z))**. BP uses GD which can thus make a progress at every step
  - Essential to avoid having only flat segment (no gradients, i.e. GD unable to move on a flat surface
- Other functions would have work well too: **hyperbolic tangent** function **tanh(z) = 2σ(2z) – 1**, **REctified Linear Unit** function: **ReLU(z) = max(0, z)**





BP and **initialisation**

Initialisation matters.

It is important to **initialise all the hidden layers' connection weights randomly**, or else training will fail

- e.g. if all weights and biases stay initialised to 0, then all neurons in a given layer will be identical, thus BP will affect them in the same way, and they will remain identical → no actual training!
- you need to initialise, so you break the symmetry, and allow BP to train a diverse team of neurons

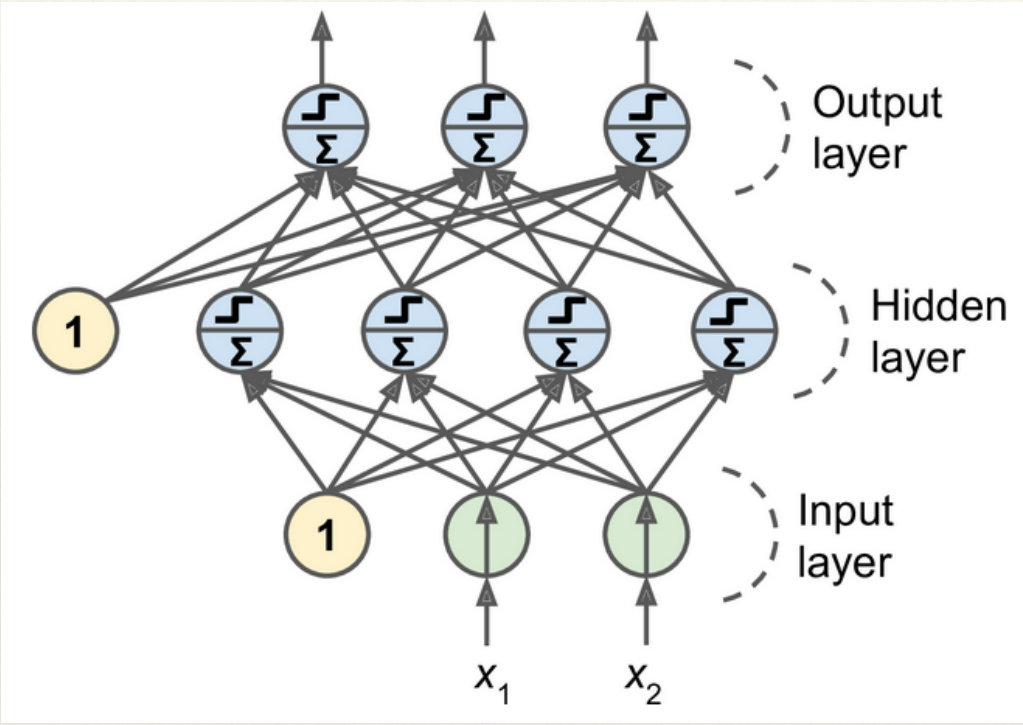
MLP for regression and MLP for classification

Good. So, **what** can you do with MLPs? And **how** (*) you do it?

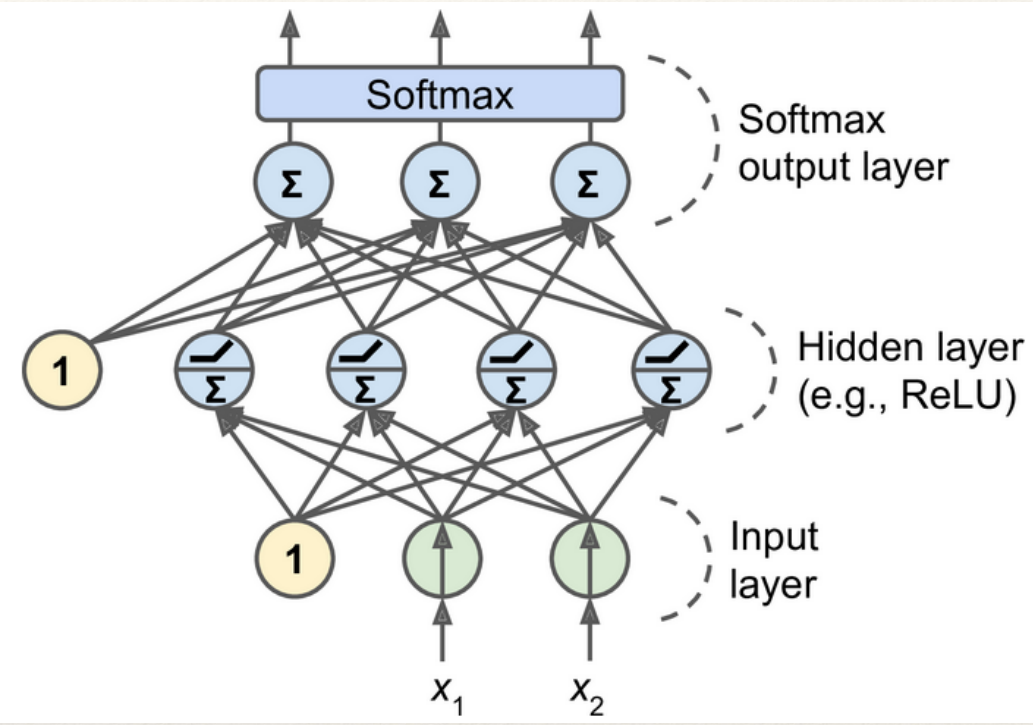
(*) CAVEAT: VERY difficult to give you "working recipes" for all possible applied-ML situations… but let's try to consolidate some basic heuristics and hints

MLP Architectures

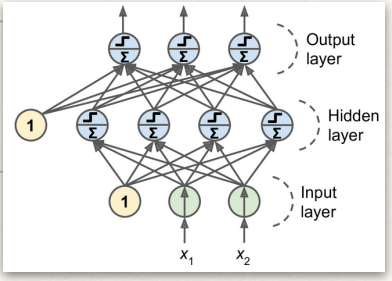
Regression MLP



Classification MLP



Regression MLP



MLPs can be used for **regression** tasks

1 vs many output neurons depending on **univariate vs multivariate regression**

- univariate regression e.g. predict a house price → need 1 output neuron
- multivariate regression e.g. predict center of a 2D object → need 2 output neuron

In general, no activation function for the output neurons, so free to output any range of values. But:

- if you want prediction to be positive, use **ReLU** in the output layer
  - or (alternatively) a smooth variant of ReLU i.e. the **softplus** (softplus(z) = log(1 + exp(z), i.e. close to 0 when z is negative, close to z when z is positive)
- if you want prediction to fall within a given range of values, use the **logistic** function or the **hyperbolic tangent**, and scale the labels to the appropriate range (0 to 1 for the logistic function, or –1 to 1 for the hyperbolic tangent)

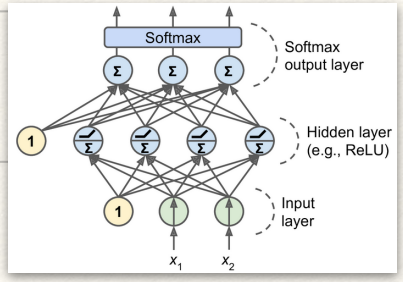
The **loss function** to use during training is typically the **MSE** (=(1/n) * Σ(predicted - true)^2)

- but if you have many outliers in the training set, **MAE** (= (1/n) * Σ|predicted - true|), works better
  - or use the **Huber** loss, which is a combination of both (quadratic when the error is smaller than a threshold δ - typically 1, but linear when the error is larger than δ. This makes it less sensitive to outliers than MSE, and it is often more precise and converges faster than MAE

Regression MLP: architecture

Hyperparameter	typical value
# input neurons	One per input feature (e.g. 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem. Typically 1 to 5.
# neurons per hidden layer	Depends on the problem. Typically 10 to 100.
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU)
Output activation	None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLP



MLPs can be used for **classification** tasks

1 vs many output neurons depending on **single-label vs multi-label binary classification**, all using the **logistic activation function**

- dedicate one output neuron for each positive class: the output(s) will be a number between 0 and 1, which you can interpret as the estimated probability P of the positive class, with the estimated probability of the negative class being 1-P
  - single-label binary classification: e.g. spam/not-spam
  - multi-label binary classification: e.g. spam/not-spam together with urgent/not-urgent
    - note that classes might be non-exclusive: in this case, probabilities will not add up to 1

1 output neuron per class in case of **multiclass classification**, using the **softmax** activation function for the whole output layer

- you need to have one output neuron per class, and you should use the softmax activation function to ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1 (required if the classes are exclusive)
  - e.g. classes 0 through 9 for MNIST classification: each instance can belong only to a single class, there are several, and they are exclusive

The **loss function**: since we are predicting probability distributions, the **cross-entropy** (aka **log loss**) is generally a good choice

Classification MLP: architecture

Hyperpar.	single-label binary classif.	multi-label binary classif.	multiclass classif.
Input and hidden layers	*Same as regression*	*Same as regression*	*Same as regression*
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross-Entropy	Cross-Entropy	Cross-Entropy

Convolutional Neural Networks (**CNN**)

Introduction

CNN (or **ConvNet**)

It is arguably the most popular Deep Learning architecture

- so dominant that it has a role by itself in the surge of interest in DL in the last decade

It started with AlexNet (2012) and grew exponentially since then

- from AlexNet 8 layers to ResNet 152 layers architectures in 3 years (!)

Areas of application

- it dominates computer vision: excellent in any 2D/3D image related tasks
- successfully applied also to recommender systems, NLP, and more

CNN (or **ConvNet**)

Main advantages of CNNs:

- **superhuman accuracy**, beating humans in most image classification tasks
- it performs **automatic feature** extractions, i.e. detects the important features without any human supervision (feed it with pics of cats and dogs and it will learn distinctive features for each class by itself)
- it is **computationally efficient**. It uses convolution and pooling operations and performs parameter sharing. This enables CNN models to run on any device, making them universally attractive.

Magic? Not really!

- → we will see how it works… but first… some thoughts on "vision".

Sensory modules (and → vision)

Humans better than machines in most tasks, but only recently

- IBM's Deep Blue supercomputer beat the chess world champion Garry Kasparov in 1996
- only very recently computers were able to reliably perform seemingly trivial tasks such as detecting a cat or a dog in a picture, or recognising spoken words.

Why are these tasks so effortless to us humans?

- perception largely takes place outside the realm of our consciousness
  - within specialised visual, auditory, and other sensory modules in our brains
  - you cannot choose not to see, or not to perceive
  - To understand perception, one needs to study in depth how the sensory modules work

Vision → CNNs

CNNs emerged from the study of the brain's visual cortex

- not a new technique: used in image recognition since the 80's
- only recently: exponential increase in computational power + amount of training data
- ⟹ boost for training DNNs !

CNNs have objectively managed to achieve **superhuman performance on some complex visual tasks**. They power:

- image search services
- self-driving cars
- automatic video classification systems
- … and more

Moreover, CNNs are **not restricted to visual perception**:

- they are also successful at many other tasks, such as voice recognition and natural language processing (NLP)

A couple of typical (visual) tasks

We will discuss **some of the best CNN architectures**, as well as some visual tasks, e.g.:

- **object detection** (classifying multiple objects in an image and placing bounding boxes around them)
- **semantic segmentation** (classifying each pixel according to the class of the object it belongs to)

The Architecture of the Visual Cortex

Receptive fields and hierarchy of neurons

Crucial insights into the structure of the visual cortex from a series of experiments by Hubel and Wiesel in 1958 and 1959, on cast (and a few years later on monkeys)

- *[Ref-Hubel] [Ref-HubelWiesel1] [Ref-HubelWiesel2]*
- Nobel Prize in Physiology or Medicine in 1981 for their work

In brief, most relevant observations:

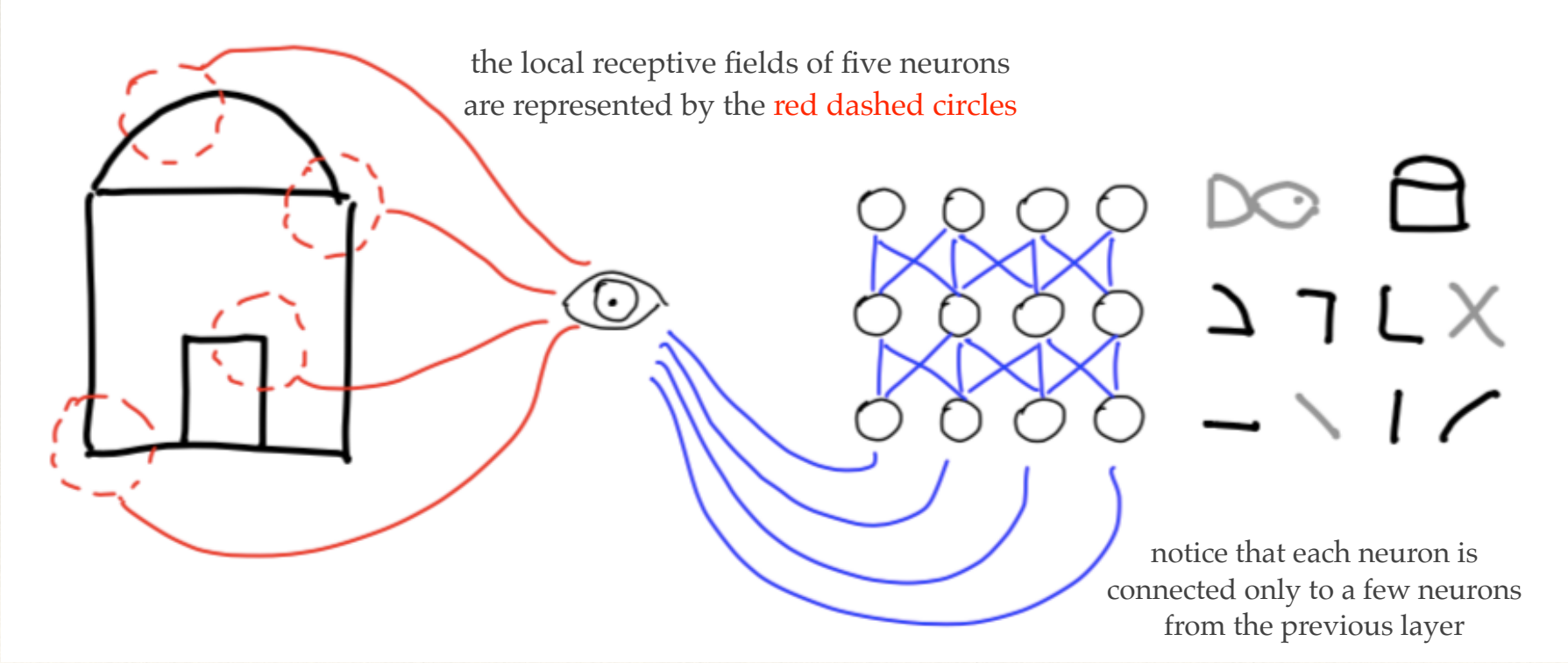
- "neurons in general get active on small visual areas": many neurons in the visual cortex have a small local "**receptive fields**", i.e. they react only to visual stimuli located in a limited region of the visual field (see next). The receptive fields of different neurons may overlap, of course, and altogether they tile the whole visual field
- "neurons get active on patterns": some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field - previous bullet - but react to different line orientations)
- "some neurons work on larger visual areas: they have larger receptive fields, and they react to **more complex patterns** that are combinations of the lower-level patterns

These observations led a classification of higher-level and lower-level neurons, and to the idea that higher-level ones are based on the outputs of neighboring lower-level ones (see Figure). This powerful architecture is **able to detect all sorts of complex patterns in any area of the visual field**.

More brain modules, increased complexity

In summary:

- most biological neurons in the visual cortex respond to specific visual patterns in small regions of the human visual field ("receptive fields")
- as the visual signal makes its way through consecutive brain modules, neurons respond to **more complex patterns** in **larger receptive fields**



CCN and LeNet

These studies of the visual cortex inspired the **neocognitron**, introduced in 1980 *[Ref-Fukushima]*, which gradually evolved into what we now call a **convolutional neural network**.

An important milestone was *[Ref-LeCun1]* in 1998, that introduced the famous **LeNet-5** architecture

- widely used by banks to recognize handwritten check numbers

It has some building blocks that we discussed already..

- e.g. fully connected layers and sigmoid activation functions

.. but it also introduces two new building blocks: **convolutional layers** and **pooling layers**.

CNN vs fully connected

Before continuing…

Why not simply use a deep neural network (from the MLP model) with fully connected layers for image recognition tasks?

CNN vs fully connected

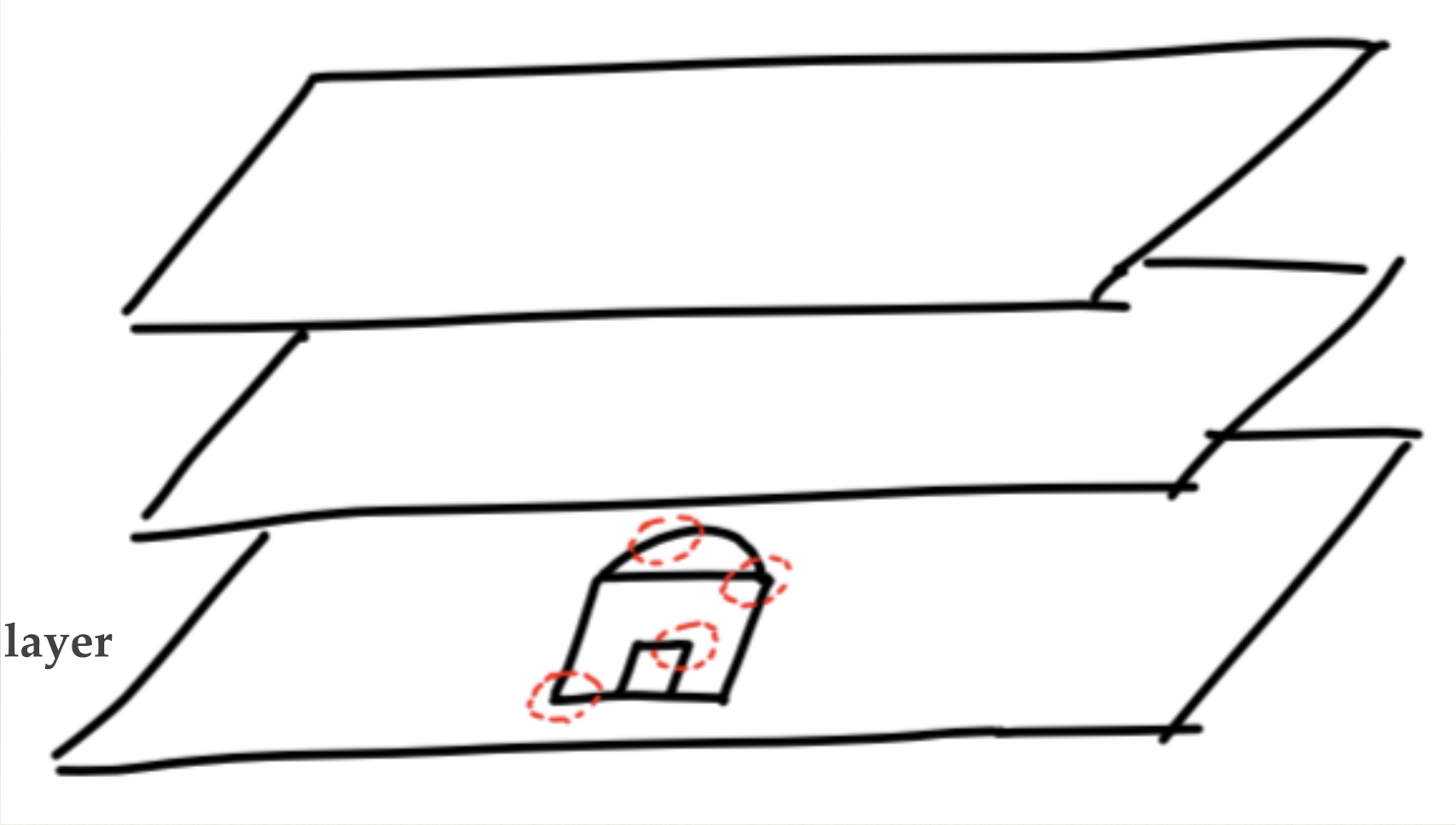
Before continuing…

Why not simply use a deep neural network (from the MLP model) with fully connected layers for image recognition tasks?

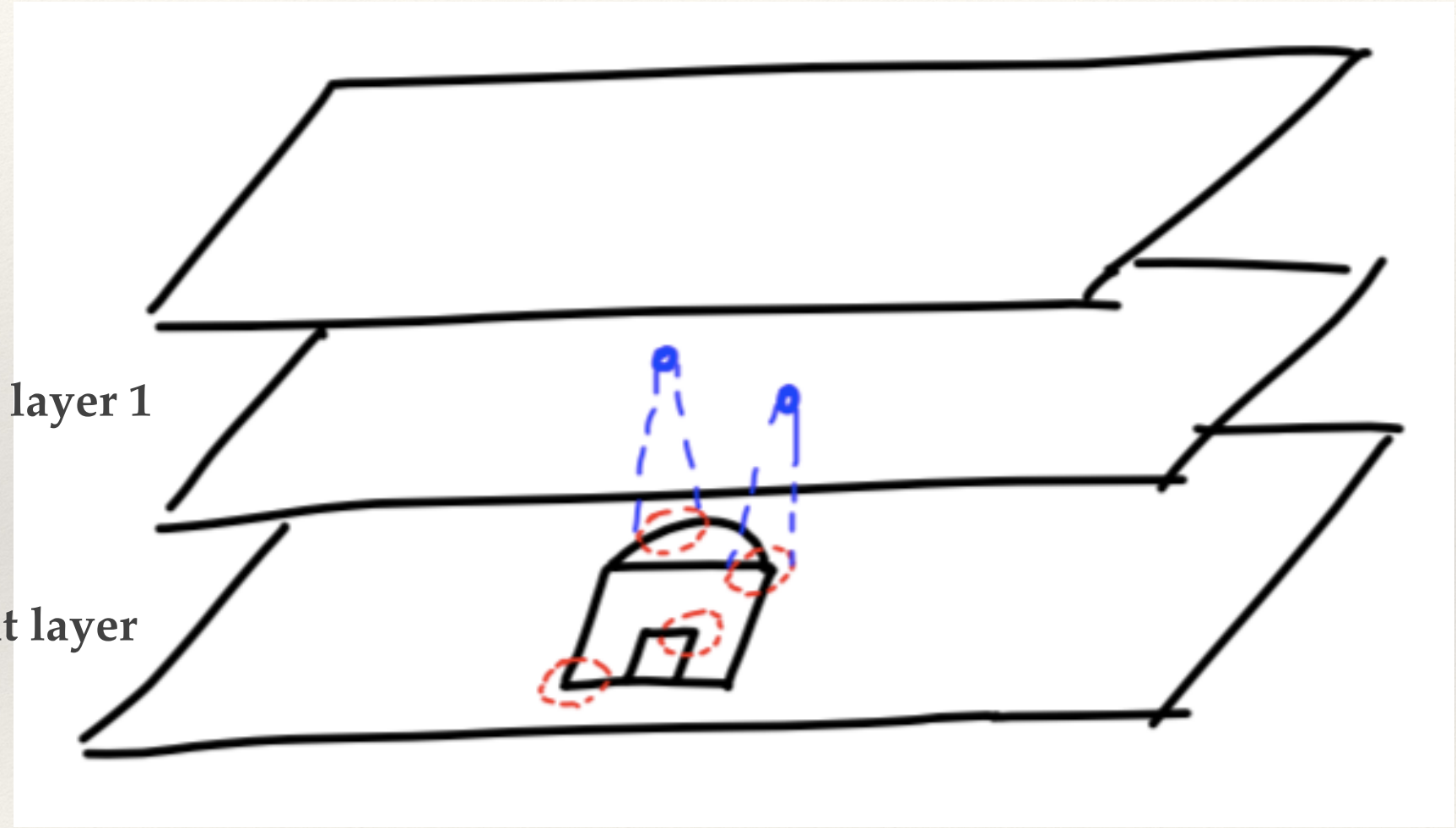
Unfortunately, although this works fine for small images (e.g. MNIST), it breaks down for larger images, as a huge number of parameters would be required

- e.g. MNIST has 28x28 images. A 100×100 image has 10,000 pixels, so with a NN with 1,000 neurons in the first layer and beyond, one easily get to millions of connections, and that's just the first layer
- CNNs solve this problem using partially connected layers and weight sharing

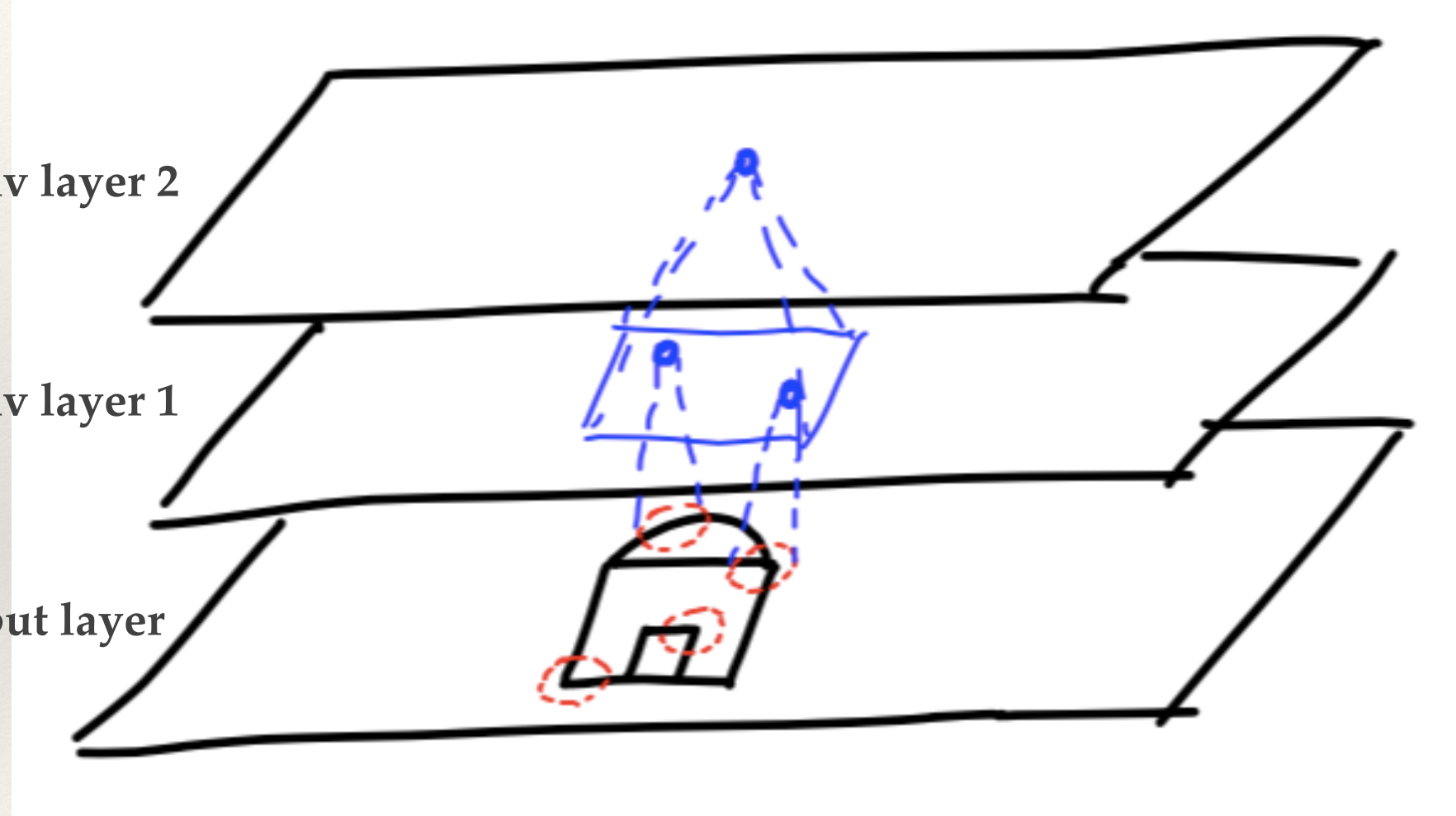
Develop some intuition first..



Develop some intuition first..



Develop some intuition first..



Even in this simplified and intuitive explanation, one can grasp that **this architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on**.

- This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition

Note: until now, all multilayer neural networks we looked at had layers composed of a long line of neurons, and we had to flatten input images to 1D before feeding them to the neural network. Now each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs

How CNNs works

Basic idea

If I work on images/videos, I want **translational invariance**

- translations should not affect recognition

One way to encode this is to implement **weight sharing**

- i.e. you apply the same weight matrices to different locations in the image

These concepts are implemented using **convolutions**.

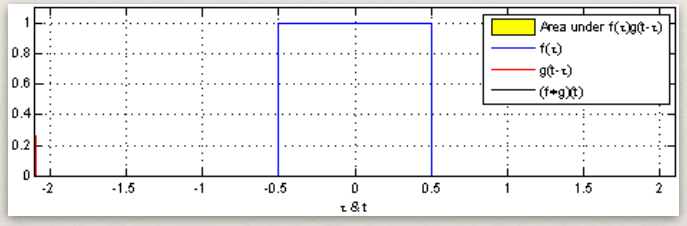
Convolution

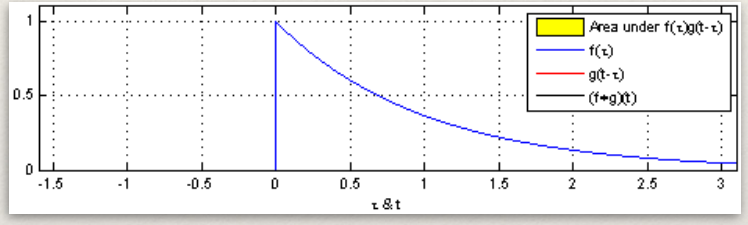
For continuous functions:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

For discrete functions:

$$(f * g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n-m)dm$$





The **convolution** (referred to the operation as well as to the result) is a mathematical operation on 2 functions *f* and *g* that produces a third function that - in words - expresses how the shape of one is modified by the other.

- basically the integral of their product after one is reversed and shifted
- symmetric - but → see next, for how we treat it..

In a nutshell, it is a mathematical operation to merge two sets of information.

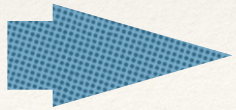
Convolution

We do not have functions over all $\mathbb{R}$, but we take ***f*** as one "**signal**" (data) of finite length (think of an 1D array whose elements are only in part ≠0), and another function ***g*** as a "**filter**" (or "kernel")

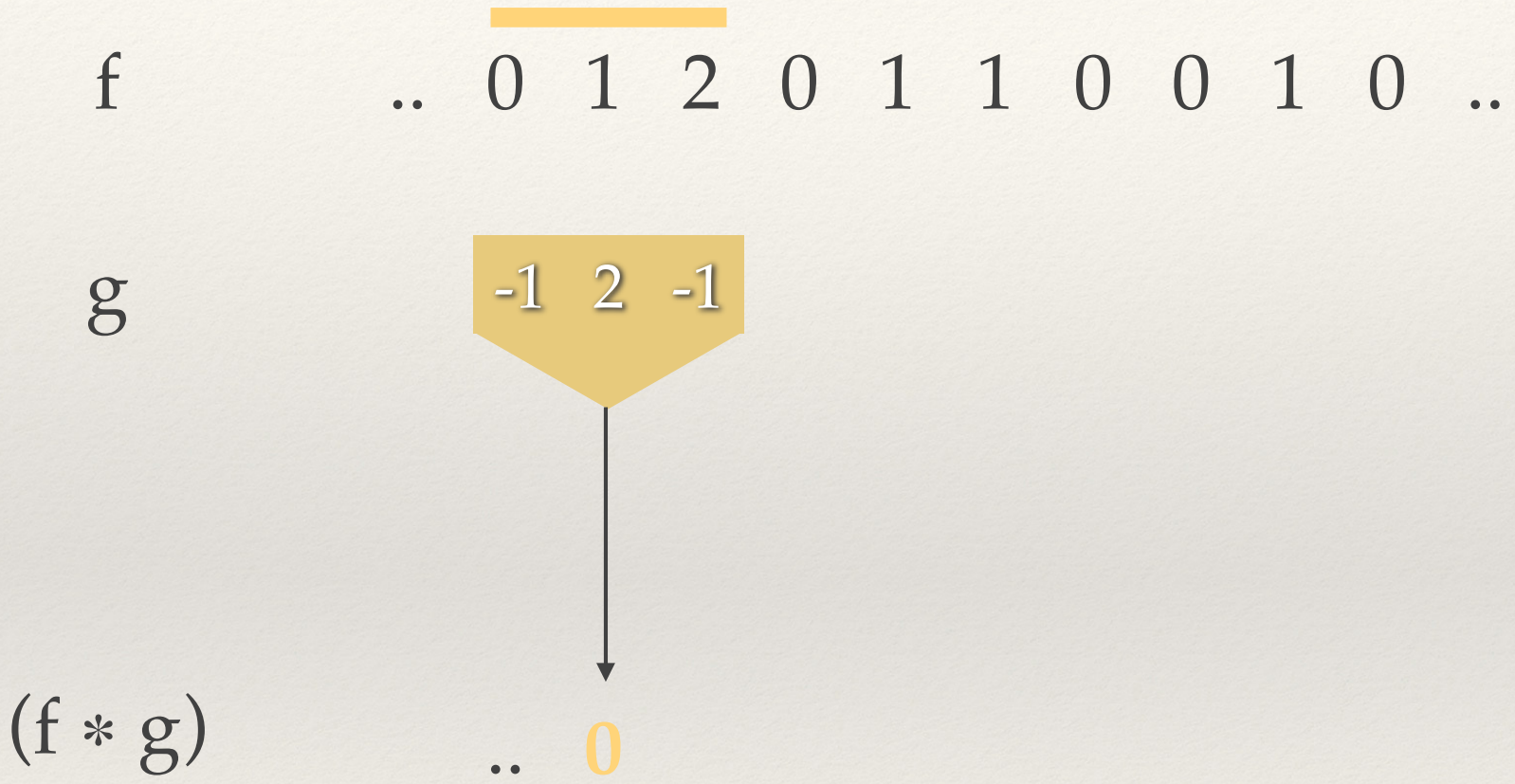
- signal → an image (simple or a 3D MRI image), a sound wave, a time series, ..
- filter → a discrete laplacian

I overlay the filter ***g*** on the signal ***f*** and I slide it along the series

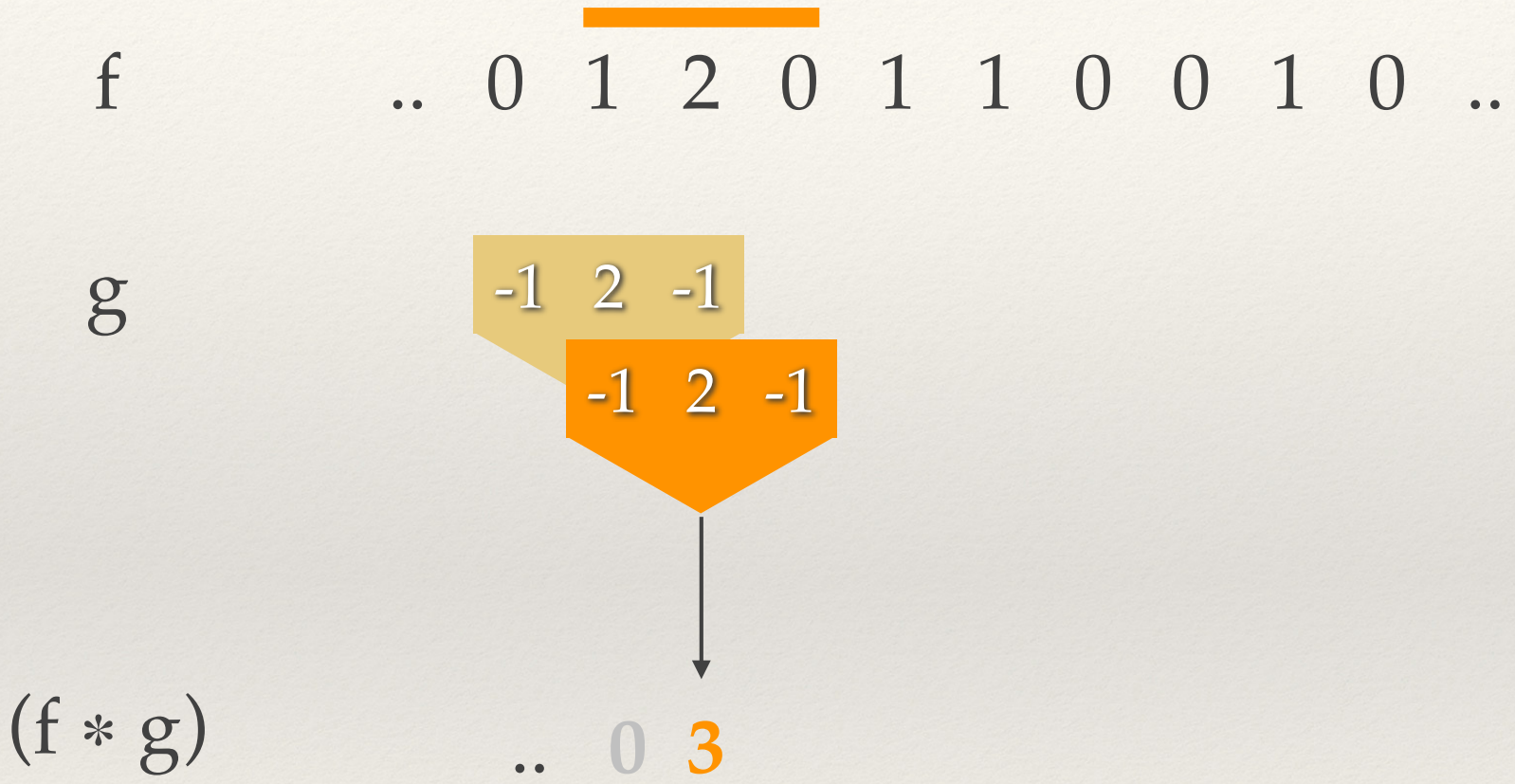
f	..	0	1	2	0	1	1	0	0	1	0	..
g		-1	2	-1								



Convolution (1D)

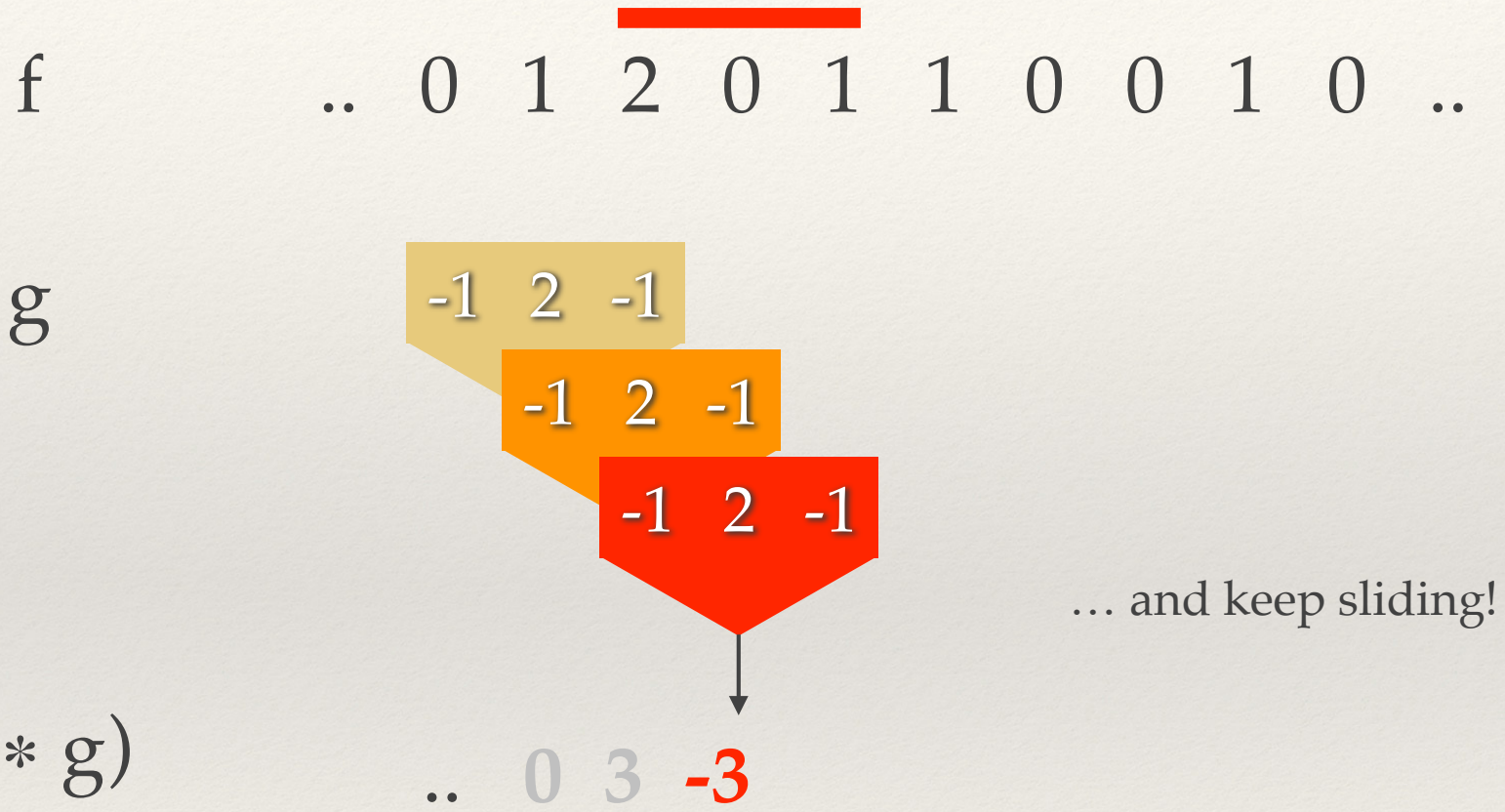


Convolution (1D)



Convolution (1D)

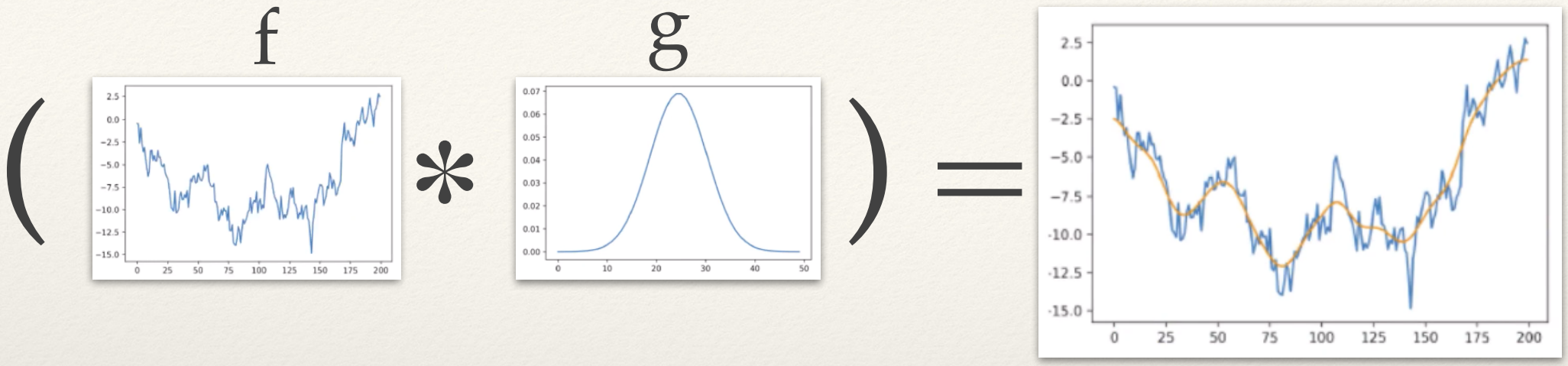
[elaboration based on A.Mueller original]



… and keep sliding!

Warning: the borders.. len(f∗g) = len(f)-2 → more later on this

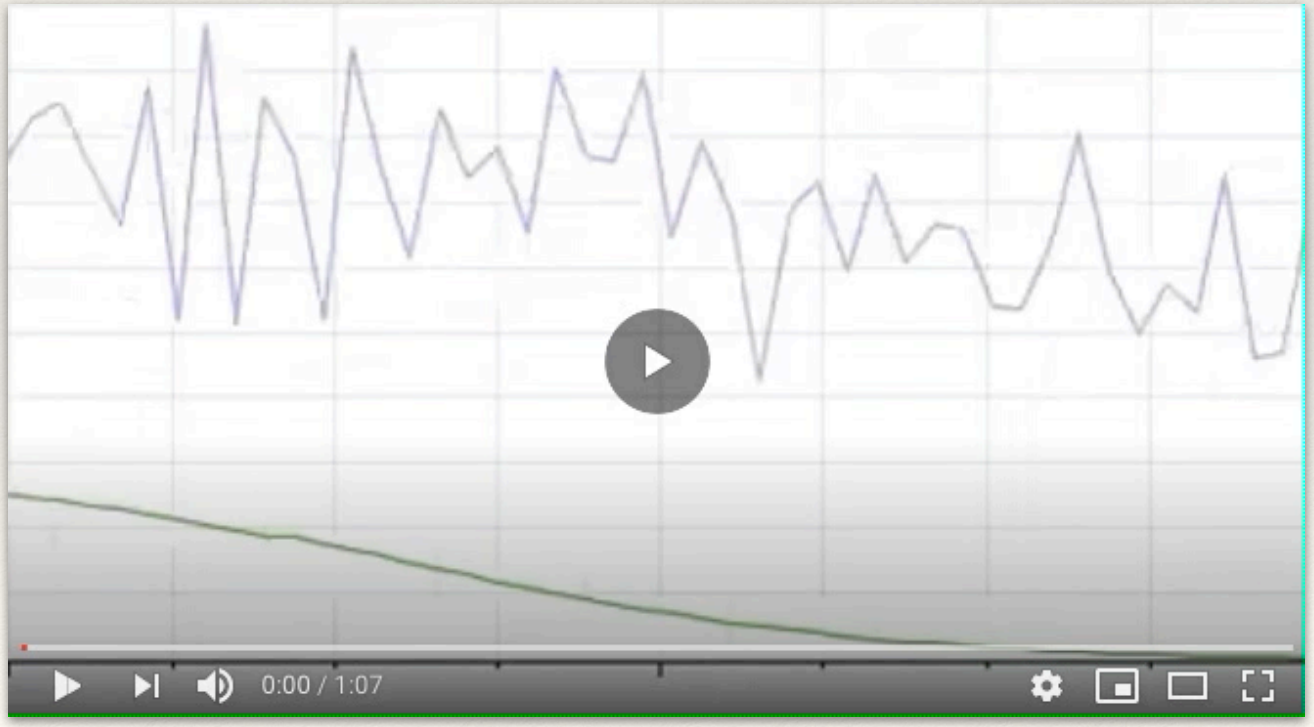
Convolution: 1D example, "smoothing"



Convolutional gaussian kernel

Smoothing of noisy sensor readings through convolution with a Gaussian kernel

Can you guess how to control how hard do I smooth the input?



Convolution (2D)

input

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

filter/kernel

1	0	1
0	1	0
1	0	1

3x3 convolution
(due to the shape of the filter)

feature map

4	3	4
2	4	3
2	3	4

We perform the convolution operation by sliding the **filter** over the **input**. At every location, we do element-wise matrix multiplication, we sum the result, and fill the **feature map**.

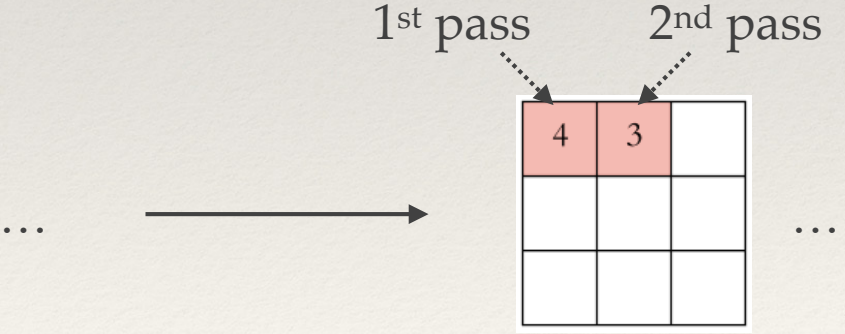
The area where the convolution operation takes place (green area) is called the **receptive field** (also 3x3, due to the size of the filter)

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

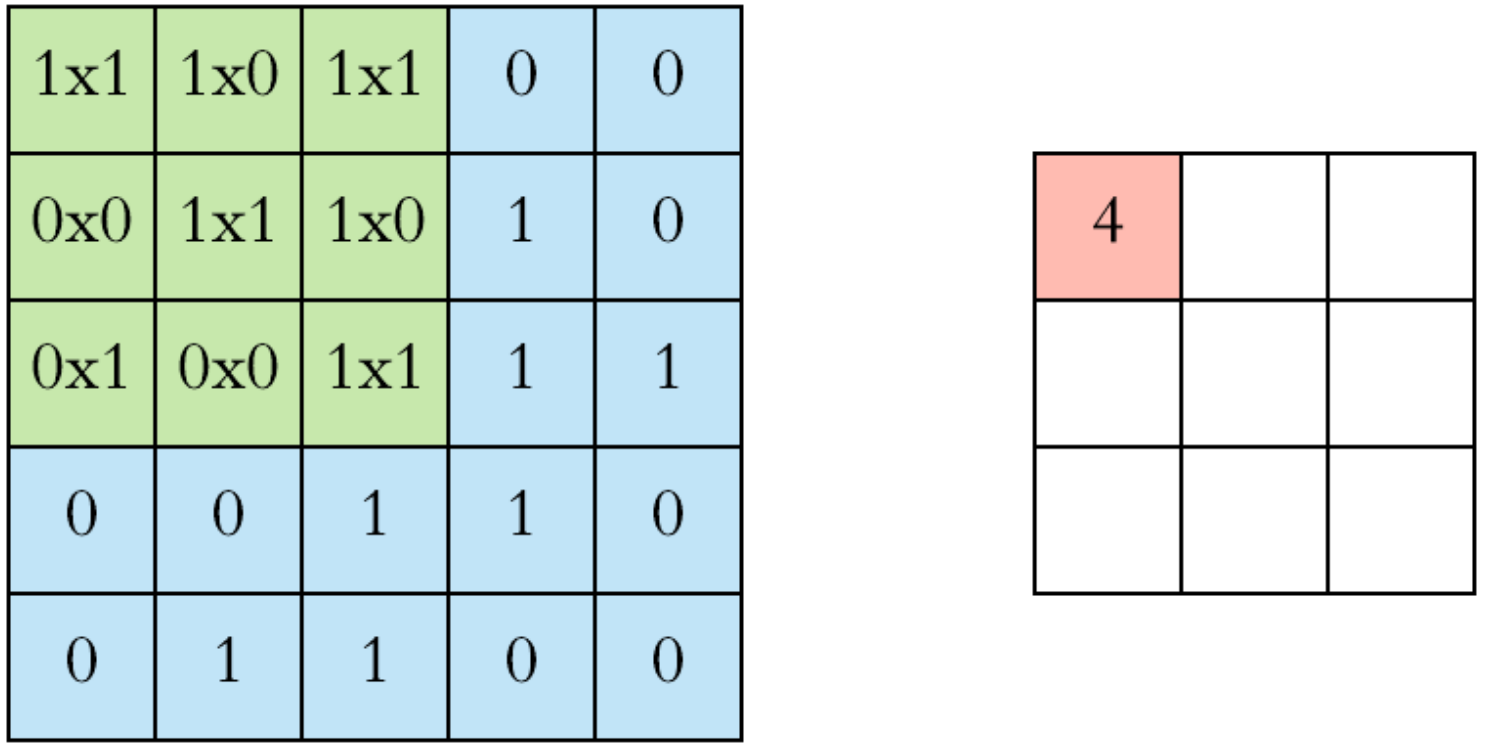
1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

input $*$ filter (1st and 2nd pass..)

1st pass 2nd pass



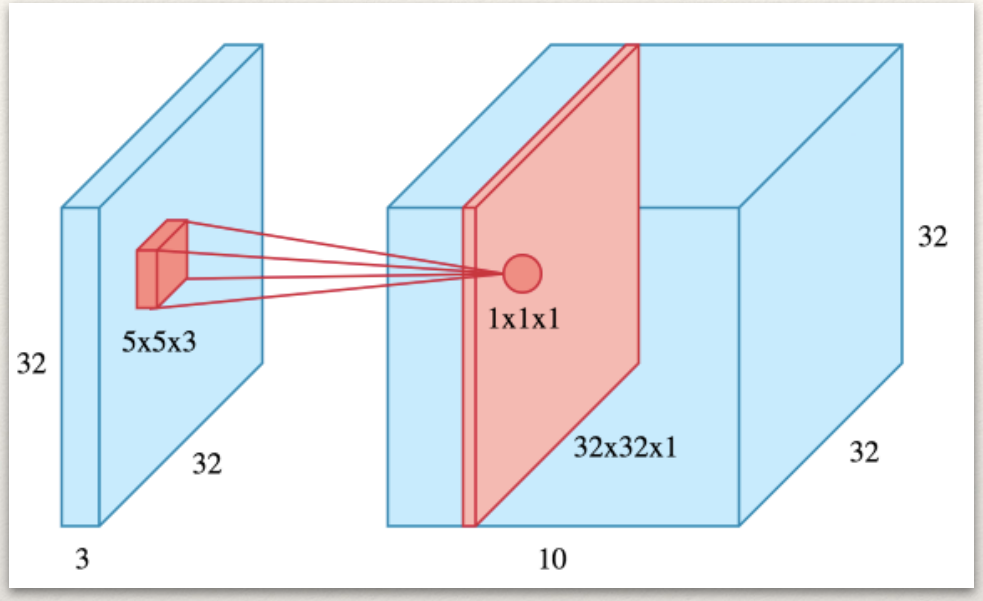
Convolution (2D)

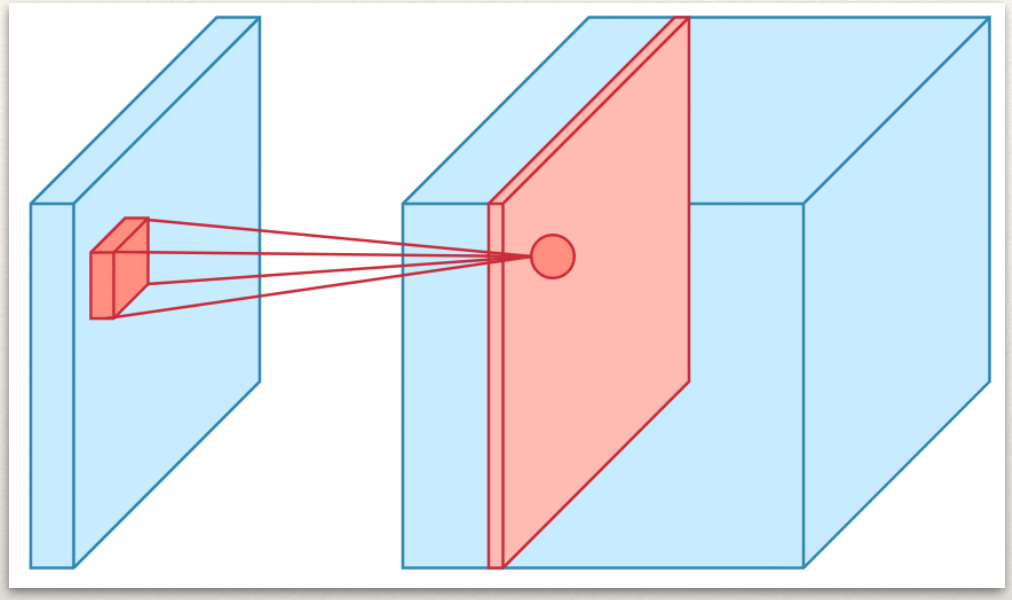


Convolution (3D)

Real image convolutions are performed in 3D, as a single image is represented as a 3D matrix with dimensions of height, width (as before) but also depth, i.e. RGB color channels

- a dataset of images is a 4D tensor





Note: padding used here. More later.

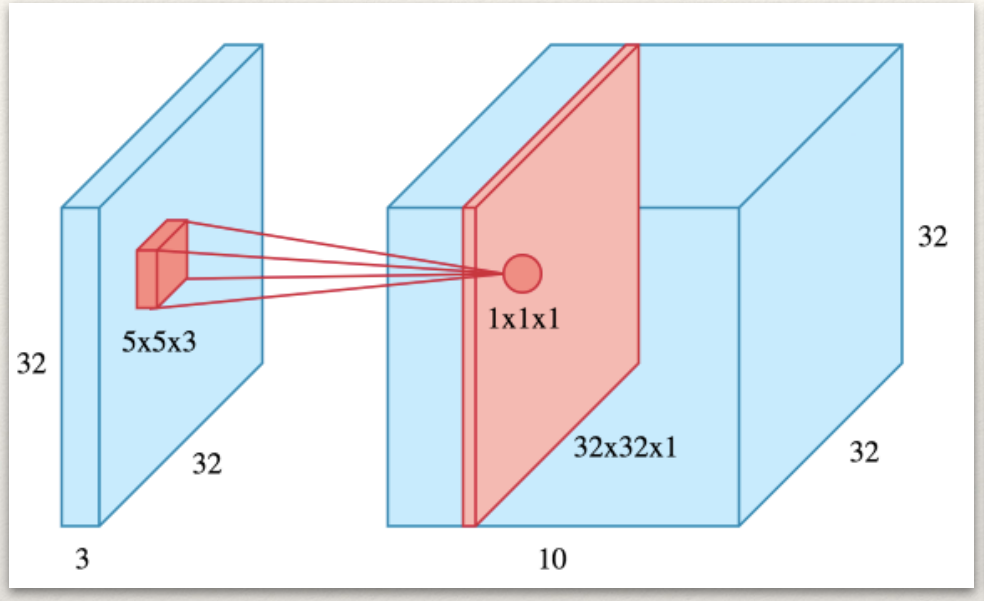
Take a 32x32x3 (color) image as input.

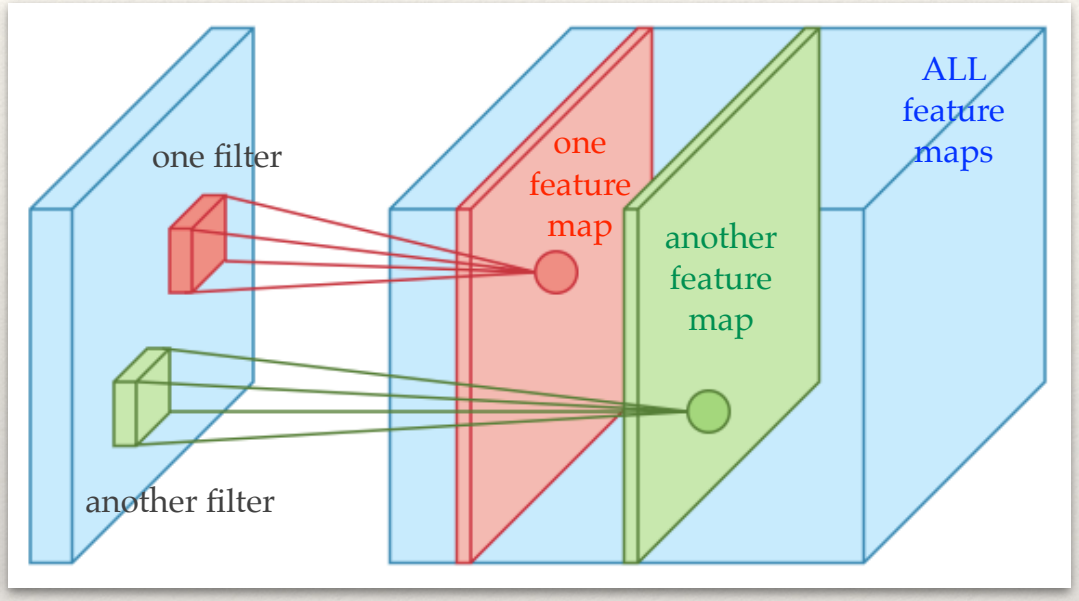
Apply one 5x5x3 filter. Compute (3D matrix mult) at one location, produce a scalar, fill one position in the feature map, then continue and slide through the input: we get a 32x32x1 feature map for one filter.

Convolution (3D)

Real image convolutions are performed in 3D, as a single image is represented as a 3D matrix with dimensions of height, width (as before) but also depth, i.e. RGB color channels

- a dataset of images is a 4D tensor





Note: padding used here. More later.

Take a 32x32x3 (color) image as input.

Apply one 5x5x3 filter. Compute (3D matrix mult) at one location, produce a scalar, fill one position in the feature map, then continue and slide through the input: we get a 32x32x1 feature map for one filter.

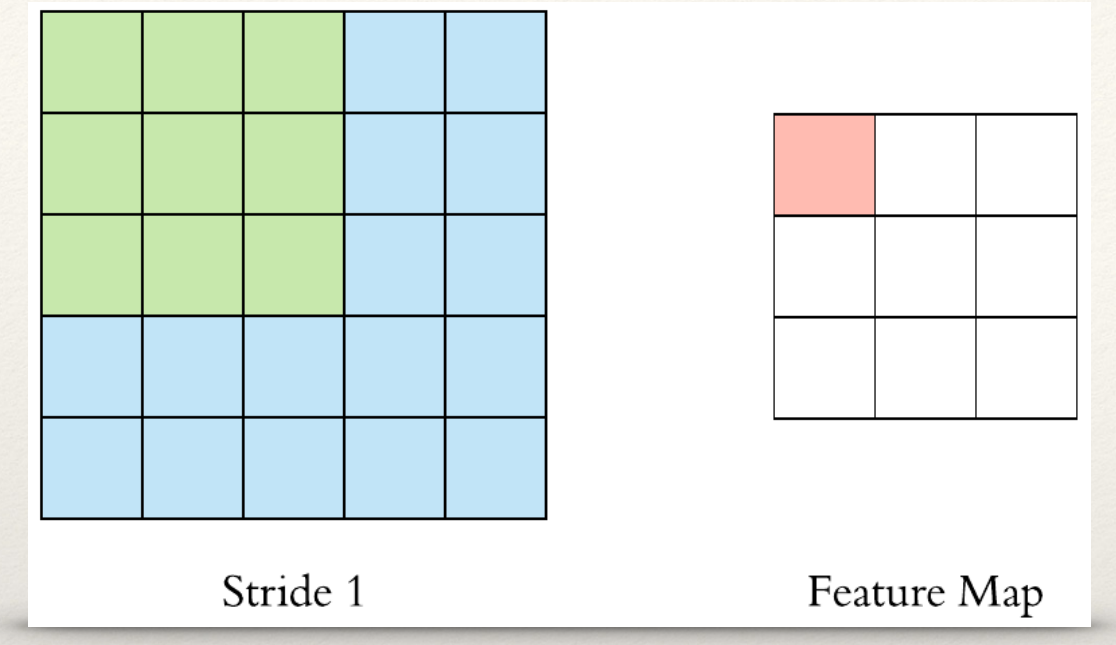
Apply more filters (say, 10 in total). You get 10 32x32x1 feature maps. Stacking them along the depth dimension give us the final output of the convolution layer: a volume of size 32x32x10, shown as the large blue box on the right. Note that the convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

Non linearity is coded in!

(it is omitted in the plots/animations for simplicity, but it is worthwhile to state it explicitly now that:)

All weighted sums go through an activation function (e.g. ReLU): a CNN is no different. So, the actual values in the final feature maps (one per filter) are **not actually the sums of the products, but the output of the activation function applied to them**

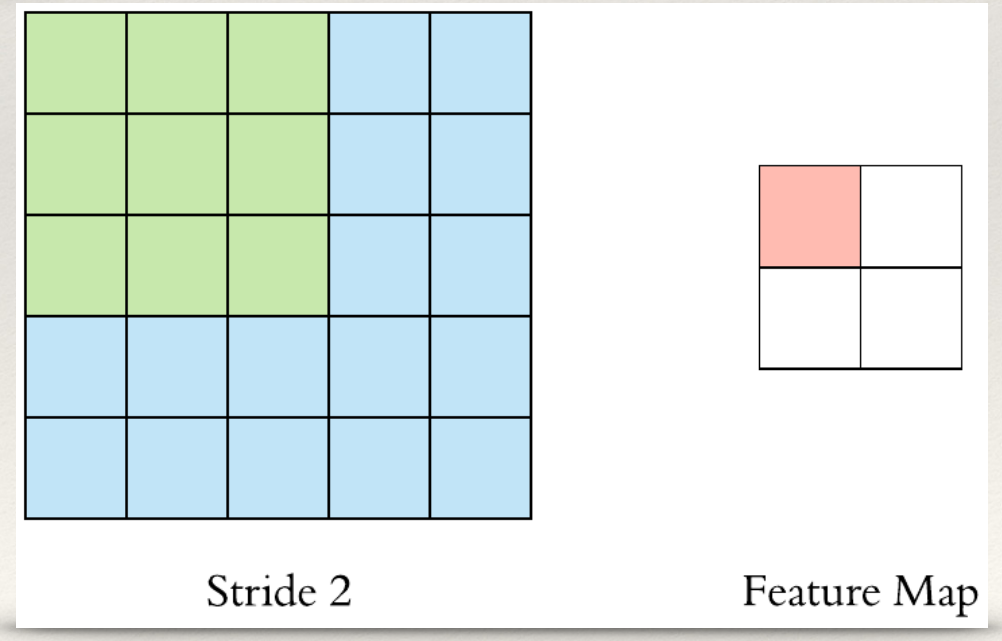
Stride



The **stride** specifies how much we move the convolution filter at each step. Default is 1.

Bigger strides yields less overlap between the receptive fields, and makes the resulting feature map smaller (we skip over some potential locations).

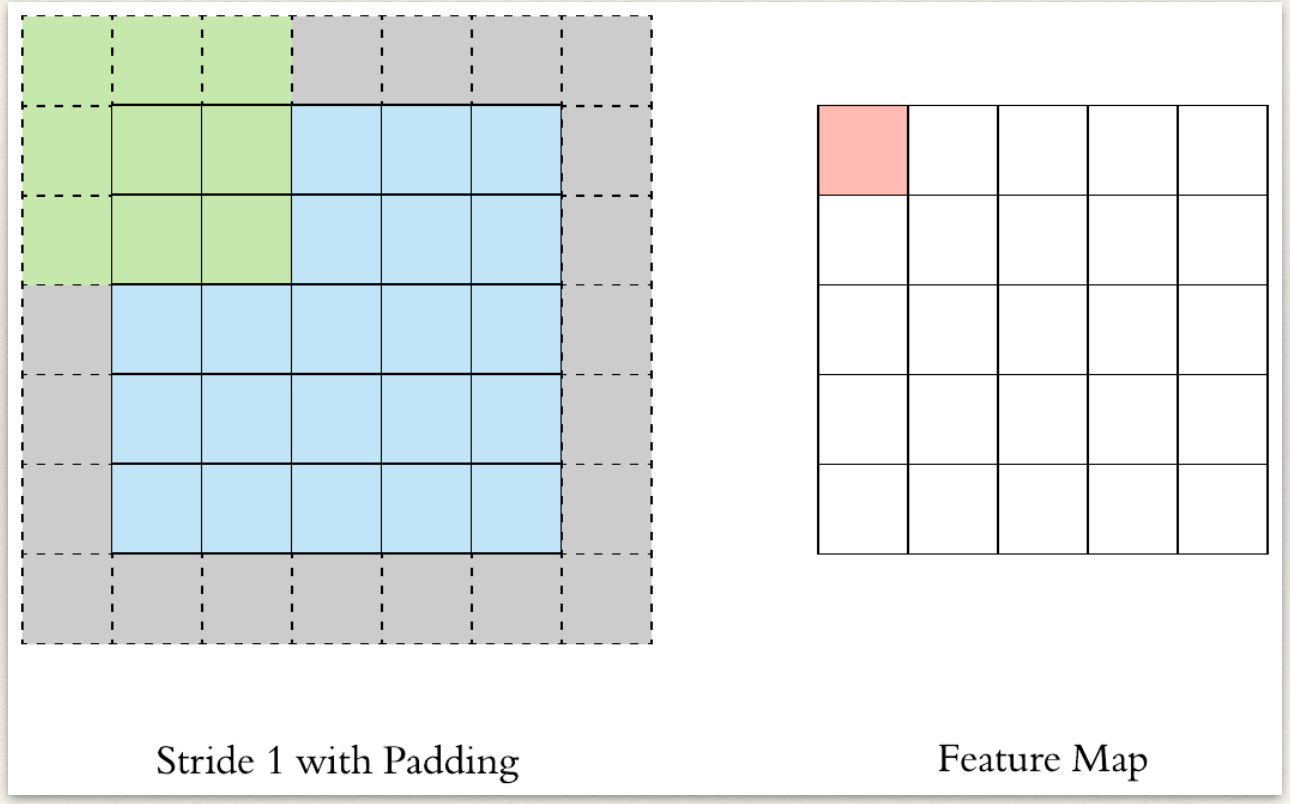
NOTE: feature maps are always smaller than the input. *Does it need to be always like this? No!* → next



Padding

If you constrain the convolution filter to be contained in the input, feature maps are smaller than the input (and you can make it even smaller by increasing the stride). This is often not desirable in CNNs as it yields smaller feature maps shrinking at each layer.

To maintain the same dimension of the input, we can use **padding**.

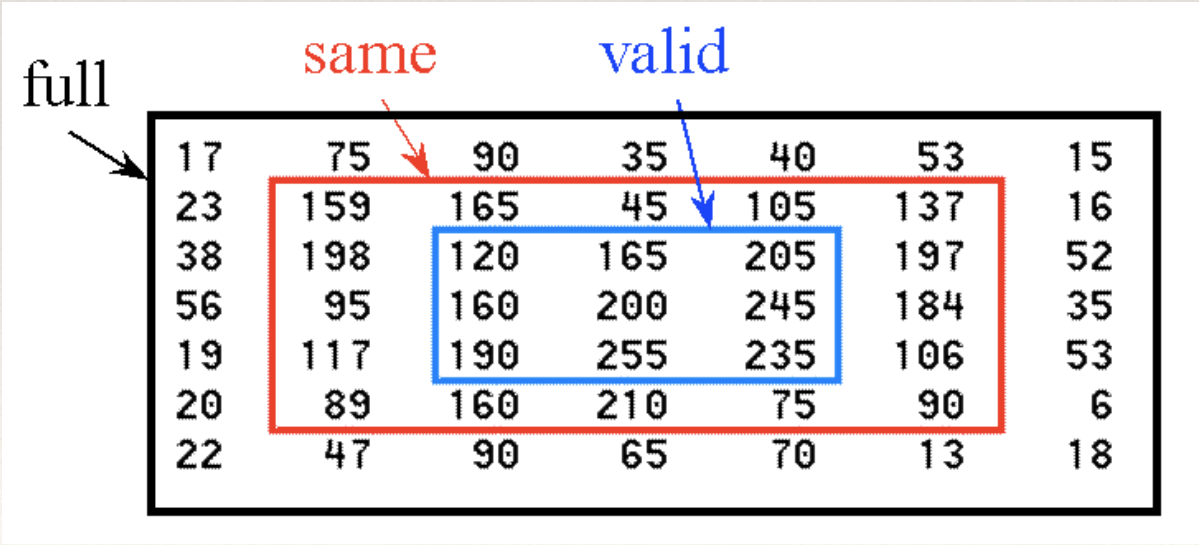


Convolution: the borders

Borders in the convolution:

- **Full**: outcome is the full convolution
- **Same**: pad with the numbers on the edge. Outcome is the central part of the convolution, of the same size as the original input
- **Valid**: no padding. Outcome restricted to only those parts of the convolution that are computed w/o the zero-padded edges

In 2D (e.g. images):



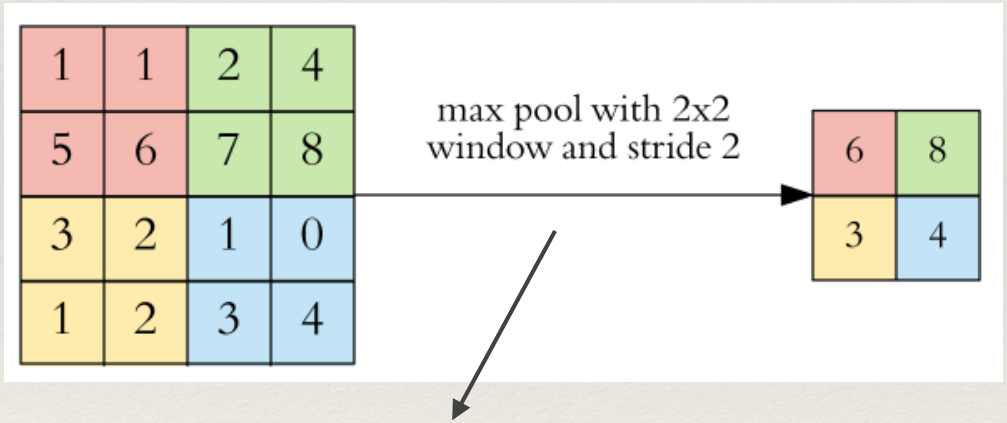
Pooling layers

After one (some) convolution operation(s) we apply **pooling** to reduce the dimensionality.

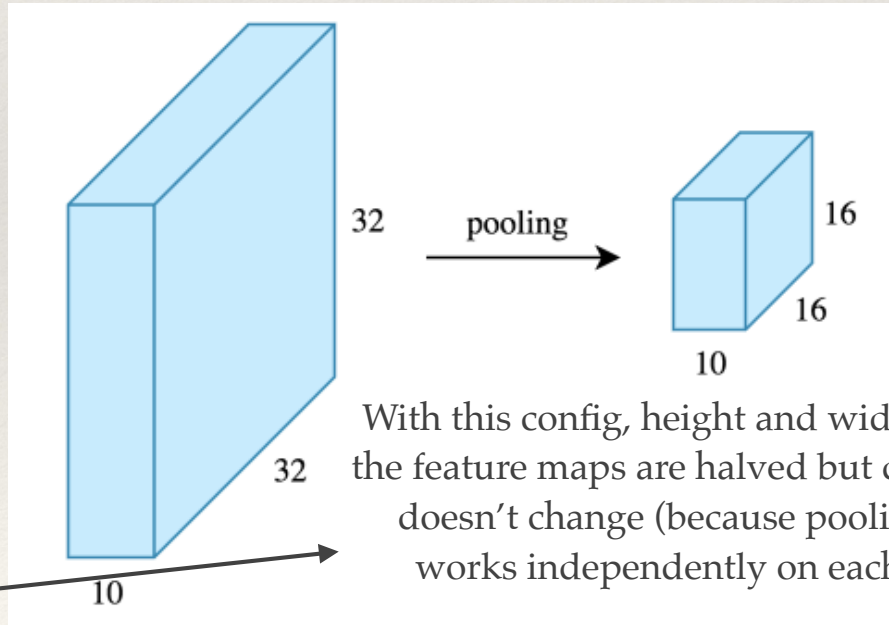
Goal is to reduce the number of parameters, as it 1) shortens the training time and 2) fights overfitting. We do so by adding a **pooling layer**

- Remember that Deep Learning is all about "clever compression", i.e. find low dimensional representations of the input that disentangle the distinguishing variation factors in your data)

Note that a pooling layer downsamples each feature map independently, reducing the height and width, keeping the depth intact.



Different colours are different pooling windows.
Max pooling takes the max in the pooling window.
We need to specify the window size and the stride (e.g. in this case, they avoid overlapping).



With this config, height and width of the feature maps are halved but depth doesn't change (because pooling works independently on each)

Total nb params reduced by to 1/4 !

A clarification: Padding vs Pooling?

Isn't it contradicting to use **padding** to avoid to reduce dimensionality (as convolution with filters shrinks the output) and to use **pooling** after some convolutional layers to reduce dimensionality?

Answer is that you actually want to reduce dimensionality, but want also to do it cleverly!

A clarification: **Padding** vs Pooling?

Isn’t it contradicting to use **padding** to avoid to reduce dimensionality (as convolution with filters shrinks the output) and to use **pooling** after some convolutional layers to reduce dimensionality?

Answer is that you actually want to reduce dimensionality, but want also to do it cleverly!

Padding is used also to ensure that input pixels on the corners and edges of the input are not "disadvantaged" in affecting the output, you do not want to loose information from the edges/corners of the input matrix

- Without padding, a pixel on the corner of an images overlaps with just one filter region, while a pixel in the middle of the image overlaps with many filter regions. Hence, the pixel in the middle affects more units in the next layer and therefore has a greater impact on the output.

The shrinking induced by convolutions with no padding is not ideal and if you have a really deep net you would quickly end up with very low dimensional representations that loose most of the relevant information in the data. Instead you want to shrink your dimensions in a smart way, which is achieved by **Pooling**.

- In particular, **Max Pooling** has been found to work well. This is really an empirical result, i.e. there isn't a lot of theory to explain why this is the case.
- Intuition: by taking the max over nearby activations, you still retain the information about the presence of a particular feature in this region, while losing information about its exact location. This can be good or bad. Good because it buys you translation invariance, and bad because exact location may be relevant for you problem.

A clarification: Padding vs **Pooling**?

Isn't it contradicting to use **padding** to avoid to reduce dimensionality (as convolution with filters shrinks the output) and to use **pooling** after some convolutional layers to reduce dimensionality?

Answer is that you actually want to reduce dimensionality, but want also to do it cleverly!

Padding is used also to ensure that input pixels on the corners and edges of the input are not "disadvantaged" in affecting the output, you do not want to loose information from the edges/corners of the input matrix

- Without padding, a pixel on the corner of an images overlaps with just one filter region, while a pixel in the middle of the image overlaps with many filter regions. Hence, the pixel in the middle affects more units in the next layer and therefore has a greater impact on the output.

The shrinking induced by convolutions with no padding is not ideal and if you have a really deep net you would quickly end up with very low dimensional representations that loose most of the relevant information in the data. Instead you want to shrink your dimensions in a smart way, which is achieved by **Pooling**.

- In particular, **Max Pooling** has been found to work well. This is really an empirical result, i.e. there isn't a lot of theory to explain why this is the case.
- Intuition: by taking the max over nearby activations, you still retain the information about the presence of a particular feature in this region, while losing information about its exact location. This can be good or bad. Good because it buys you translation invariance, and bad because exact location may be relevant for you problem.

Typical config for a CNN?

In CNN architectures, typically..

- convolution is done with 3x3 windows, stride 1 and with padding
- pooling is performed with 2x2 windows, stride 2 and no padding

(note: “typically”..)

Hyperparameters?

Hence, degrees of freedom so far are:

- **filter size**: not so many options. Typically 3x3 filters, but 5x5 or 7x7 are also often used (depending on the application). All filters are 3D (we often omit thinking of depth, as it is not changing through layers)
  - An apparently useless 1x1 filter might be used (not treated here)
- **filter count**: quite some variability. Any power of 2 between 2^5 and 2^{10}. More filters give a more powerful model, but more params expose to risks of overfitting. Best practices indicate to start with small # filters in the initial layers and progressively increase as we go deeper into the NN.
- **stride**: default is 1.
- **padding**: yes, usually it is used.

Any more layers in a CNN?

Yes!

After convolution + pooling layers, in a CNN architecture we add a couple of **fully connected (FC) layers**

Layers need to be properly connected, though:

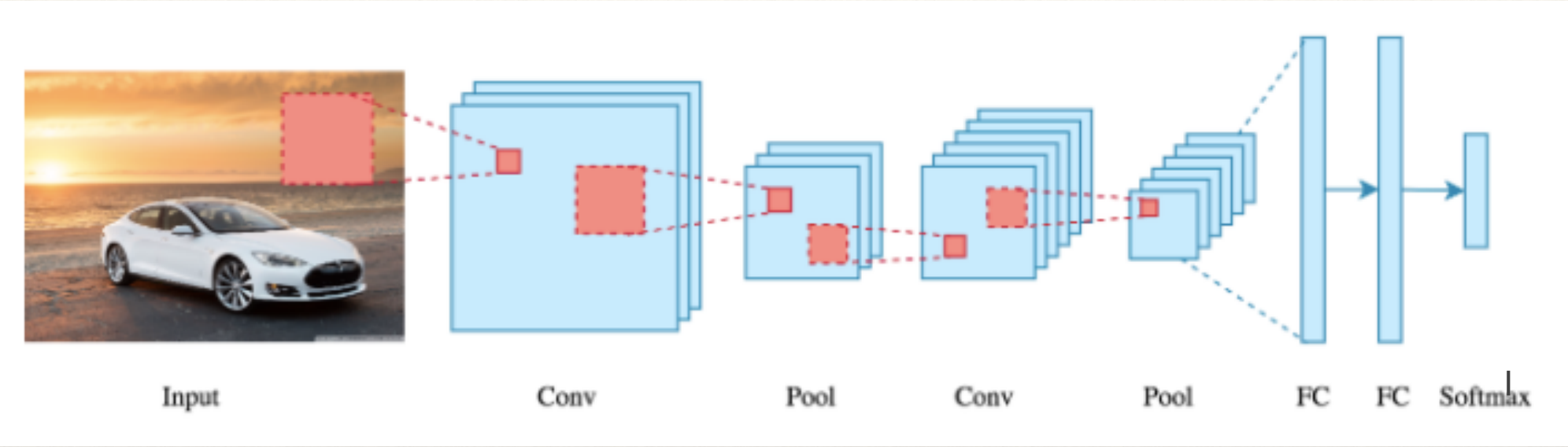
- Conv/Pooling layers are 3D volumes..
- .. but a FC layer expects to get a 1D vector of numbers

So, the output of the final pooling layer need to be **flattened** to 1D and then sent as input to the final FC layers

- note that flattening is just rearranging data, nothing fancy!

Final (basic) CNN architecture

All CNN models follow a similar architecture, which we just built, one component at a time.



feature extraction part

classification part

Now that we know how it is done, can we develop an intuition of what each part does?

The 2 parts of our final (basic) CNN architecture

1. **Feature extraction** via Conv layers:

   - it is not easy to automatic detect meaningful features in an image given only image+label ! Conv layers are able to learn such complex features by building on top of each other

     - first layers detect edges; the next layers combine them to detect shapes; to following layers merge this information to infer perhaps a specific object (e.g. a cat's tail). Note that a CNN does not know what a tail is - but, by seeing many of these objects in images, it learns to detect it as a feature once it sees one.

As such, Cons layers are the horsepowers of a CNN! Most of the work is done here! Then, next is:

2. **Classification** via FC layers:

   - act as a classifier on top of such features, i.e. learn how to use the features produced by Conv layers in order to correctly classify the images

     - namely, they assign a probability for the input image being e.g. a cat

Applied Machine Learning - Advanced

Prof. Daniele Bonacorsi

Hands-on: MNIST2B

Data Science and Computation PhD + Master in Bioinformatics

University of Bologna

Hands-on: MNIST2B

Classification with Keras: a CNN

[credits to: A. Geron, "Hands-On Machine Learning With Scikit-Learn and Tensorflow"]

Coding material

AML2223Adv_HandsOn_MNIST2B.ipynb

→ go to the "**MNIST2B**" notebook

w/o NN

w/ MLP

w/ CNN

Exercise:

Some different dataset with CNN

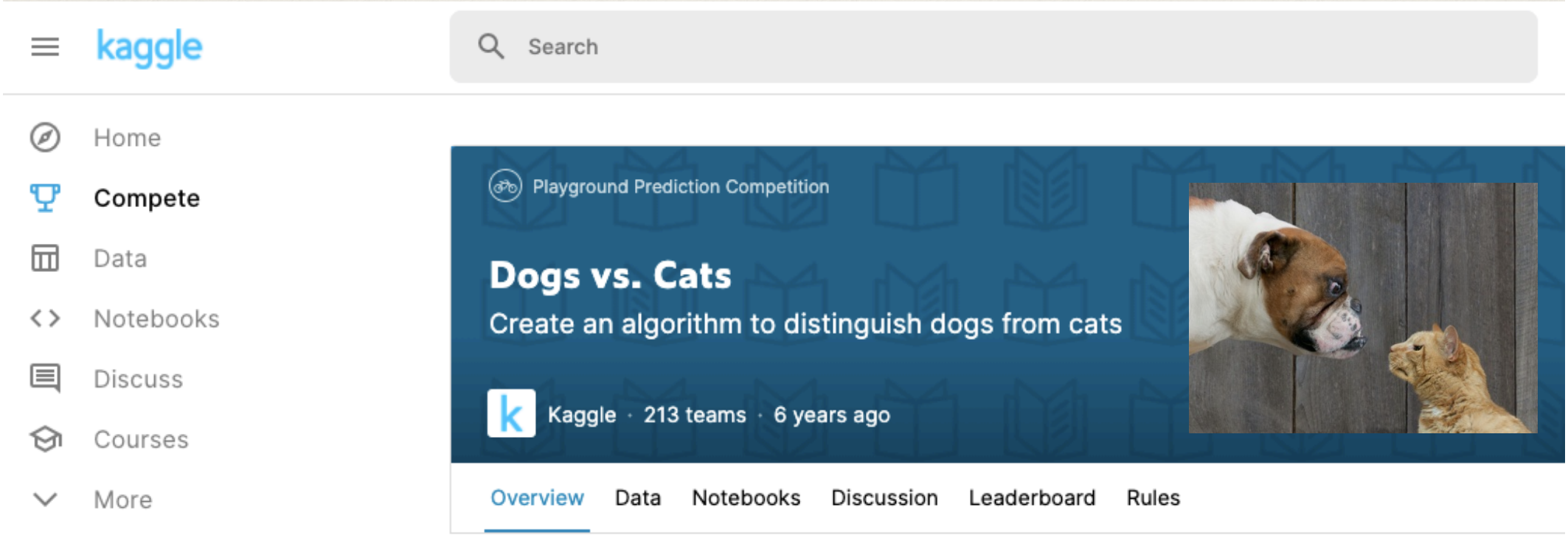
Kaggle!

Goal: use CNN, but not on MNIST. Profit of this to have a look at dropout, data augmentation, etc

Implementation example in Keras

We will use a cats vs dogs dataset from a Kaggle competition:

- https://www.kaggle.com/c/dogs-vs-cats/data
- *"The training archive contains 25,000 images of dogs and cats. Train your algorithm on these files and predict the labels for test1.zip (1 = dog, 0 = cat)"*



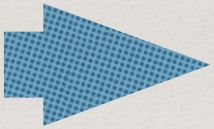
Which CNN architecture?

Our CNN architecture:

- the input is e.g. an image of a cat or dog
- 4 Conv + pooling layers, followed by 2 FC layers
- the output is binary

[just an example architecture]

Can you "visualize" what we are going to implement?

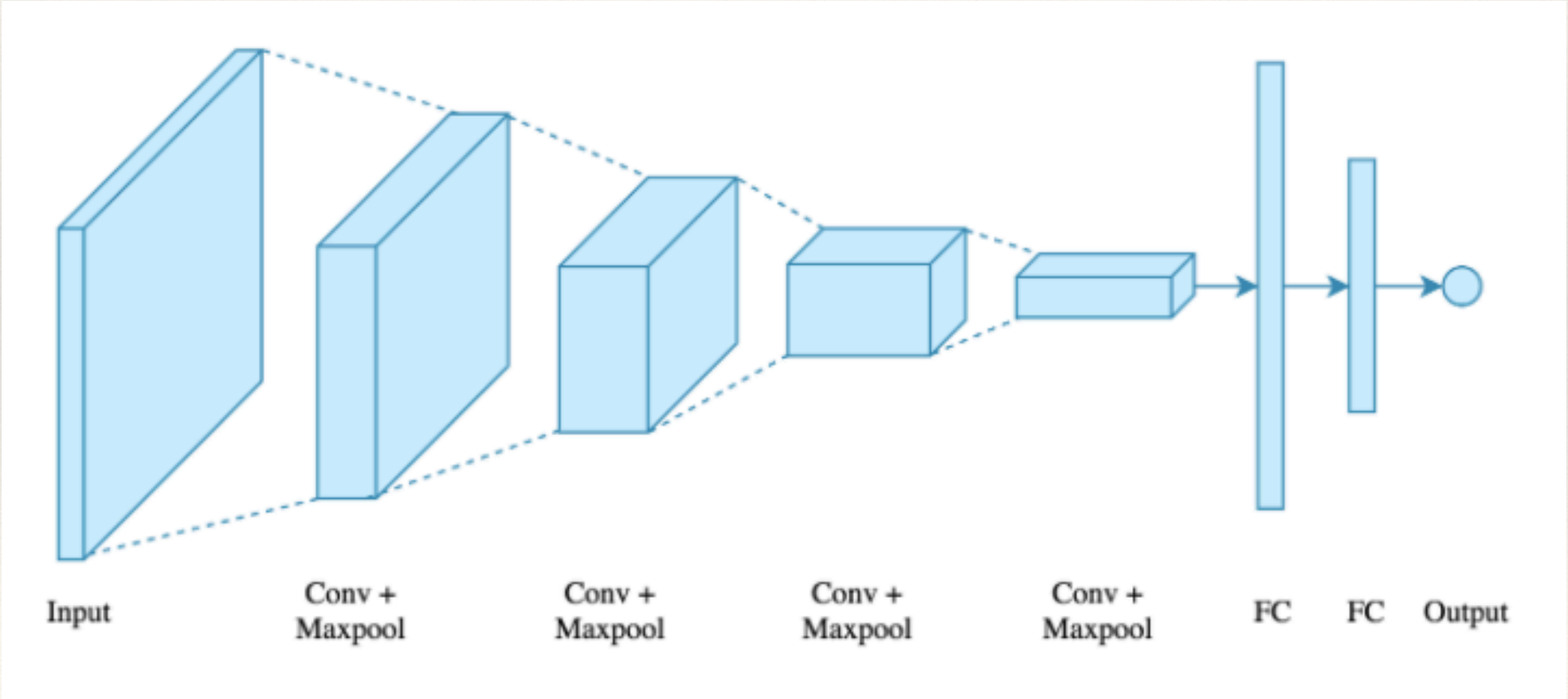


Which CNN architecture?

Our CNN architecture:

- the input is e.g. an image of a cat or dog
- 4 Conv + pooling layers, followed by 2 FC layers
- the output is binary

[just an example architecture]



The code for this exercise

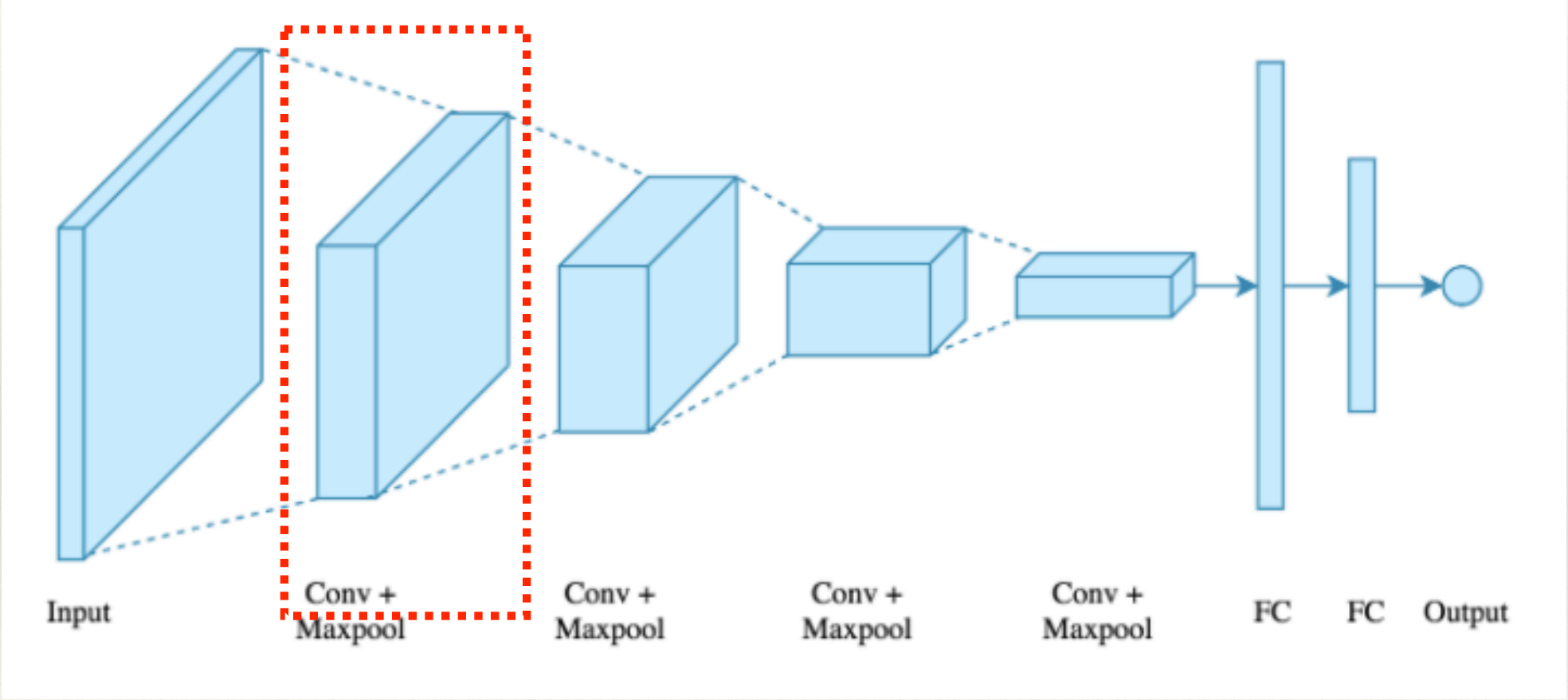
```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1',
                 input_shape=(150, 150, 3)))
model.add(MaxPooling2D((2, 2), name='maxpool_1'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', name='conv_2'))
model.add(MaxPooling2D((2, 2), name='maxpool_2'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_3'))
model.add(MaxPooling2D((2, 2), name='maxpool_3'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_4'))
model.add(MaxPooling2D((2, 2), name='maxpool_4'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu', name='dense_1'))
model.add(Dense(256, activation='relu', name='dense_2'))
model.add(Dense(1, activation='sigmoid', name='output'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Conv2D

```
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1',
                 input_shape=(150, 150, 3)))
```

- a method to create a 2D convolutional layer
  - https://keras.io/api/layers/convolution_layers/
- Config: 32 filters of size 3x3, use ReLU non-linearity as activation, we enable padding (same), stride=1 (default), we give a name to the layer, we specify the input shape (150x150 color pictures)

150x150x3 32 3x3 filters

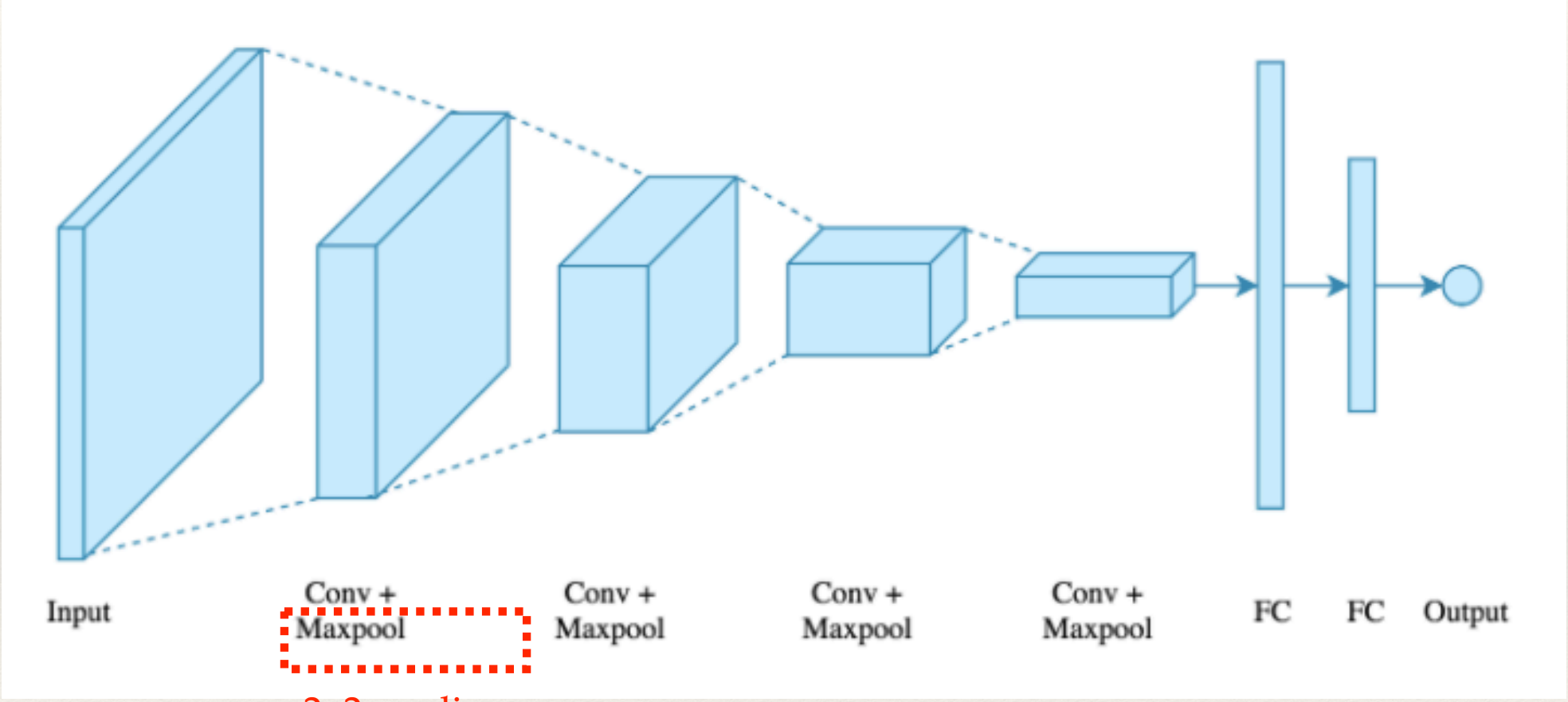


MaxPooling2D

```
model.add(MaxPooling2D((2, 2), name='maxpool_1'))
```

- a method to create a maxpooling layer
- Config: only argument is the window size, we use a 2x2 window (most common). Stride length (by default) is equal to the window size, which is 2 here.

150x150x3 32 3x3 filters

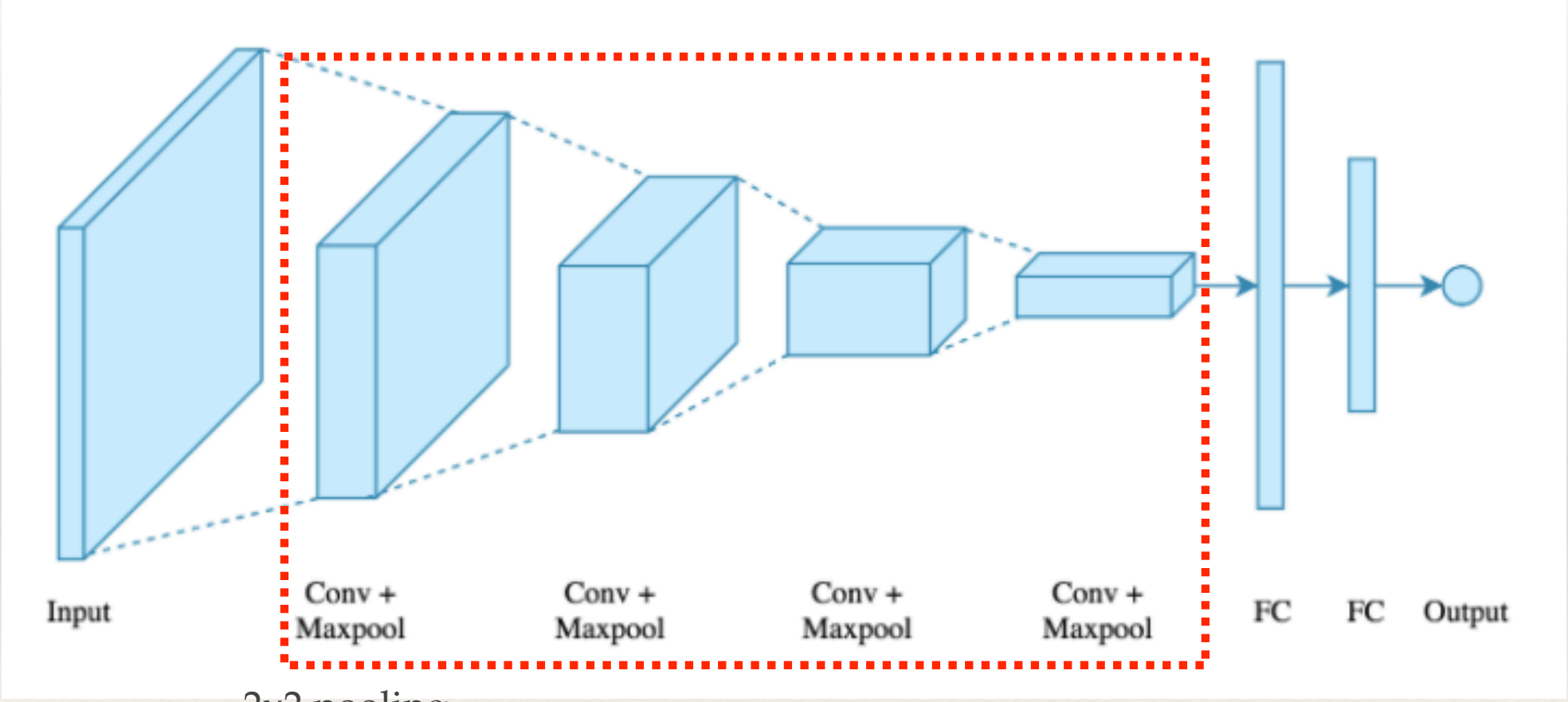


2x2 pooling

We do it (similarly, apart from sizes..) 4 times

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1',
                 input_shape=(150, 150, 3)))
model.add(MaxPooling2D((2, 2), name='maxpool_1'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', name='conv_2'))
model.add(MaxPooling2D((2, 2), name='maxpool_2'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_3'))
model.add(MaxPooling2D((2, 2), name='maxpool_3'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_4'))
model.add(MaxPooling2D((2, 2), name='maxpool_4'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu', name='dense_1'))
model.add(Dense(256, activation='relu', name='dense_2'))
model.add(Dense(1, activation='sigmoid', name='output'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

150x150x3 32 3x3 filters



2x2 pooling

(the plot does not precisely reflect the code here..)

Flatten

```
model.add(Flatten())
```

- bridge from Conv+pooling layers to FC layers

Dropout

```
model.add(Dropout(0.5))
```

- see next →

FC

```
model.add(Dense(512, activation='relu', name='dense_1'))
model.add(Dense(256, activation='relu', name='dense_2'))
```

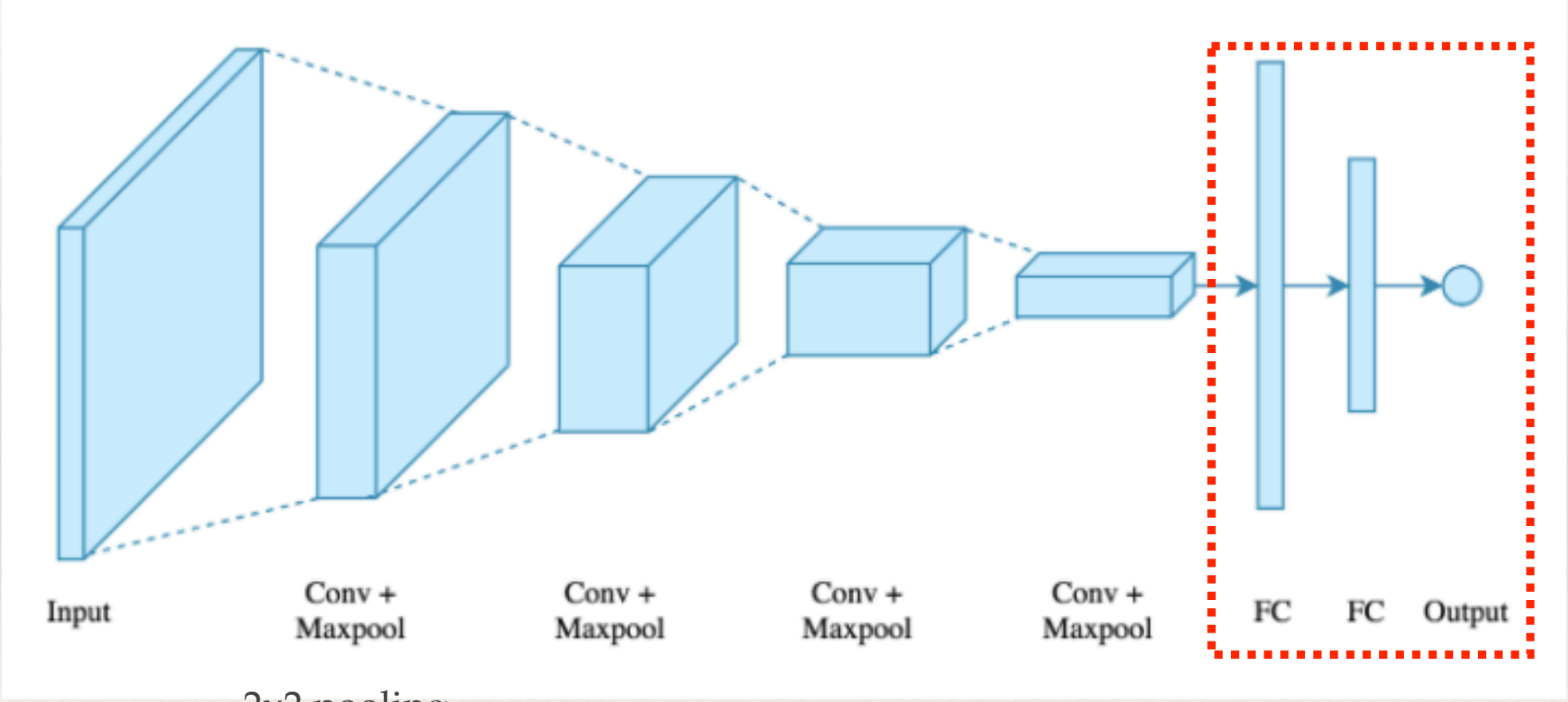
- 2 layers, 512 and 256 respectively

Output

```
model.add(Dense(1, activation='sigmoid', name='output'))
```

- binary classification, so simple sigmoid (i.e. no softmax)

150x150x3 32 3x3 filters



2x2 pooling

Drop-out [1/3]

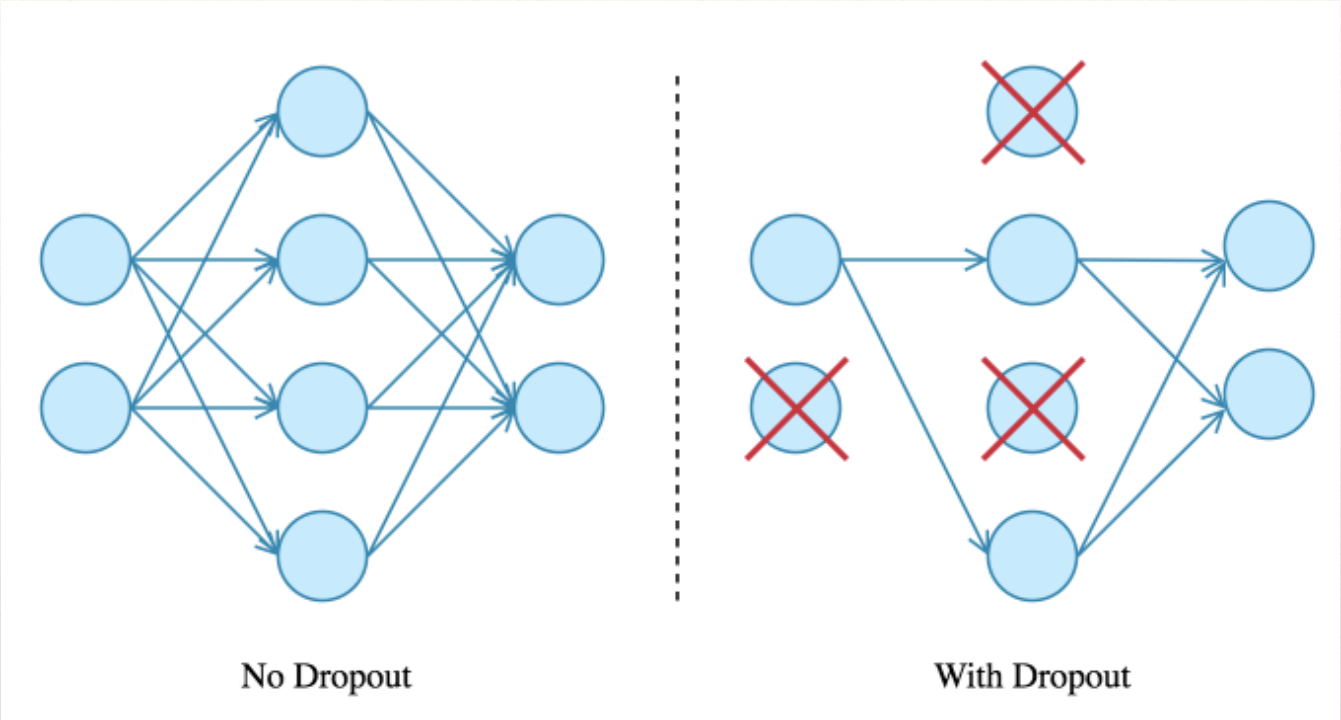
Dropout is a technique used to prevent overfitting

- well, by far, the *most popular* regularisation technique for deep NN
  - together with batch normalisation (see later →)
- remarkably useful: even state-of-the-art models at ~95% accuracy may get a +2% accuracy boost just by adding dropout

The idea

- during training time, at each iteration, a neuron is temporarily "dropped", i.e. disabled, but not 100% sure, instead with probability *p*. So, you have a probability p that all the inputs (outputs) to (from) this neuron will be disabled at the current iteration. Resampling at every training step may re-activate it at the next step
- the hyperparameter *p* is called the **dropout-rate**: typically a number around 0.5, i.e. 50% of the neurons being dropped out.

Drop-out [2/3]



Crazy, no? We are disabling neurons on purpose and the NN works better?!

Yes! Dropout prevents the NN to be too dependent on a small # of neurons, and forces every neuron to be able to operate independently.

- *analogy: the competence of key individuals in a team might not be an asset and does not make the team resilient*
- a concept applied also elsewhere
  - constraining the code size of the autoencoder allows to learn more intelligent representations..

Drop-out [3/3]

Few (obvious?) remarks:

- Dropout can be applied to input or hidden layer nodes but **not the output nodes**
- The edges in and out of the dropped out nodes are disabled, but remember this applies only at one training step: nodes which are dropped out **change at each training step**
- We **don't apply dropout during test** time after the network is trained, we do so only in training

Fit the model

```
history = model.fit_generator(train_generator, steps_per_epoch=100, epochs=20,
                              validation_data=validation_generator, validation_steps=50, verbose=1)
```

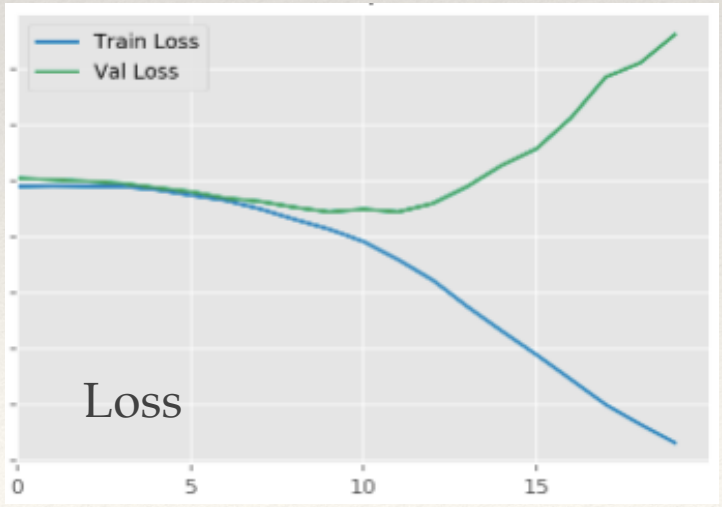
(more on next slide)

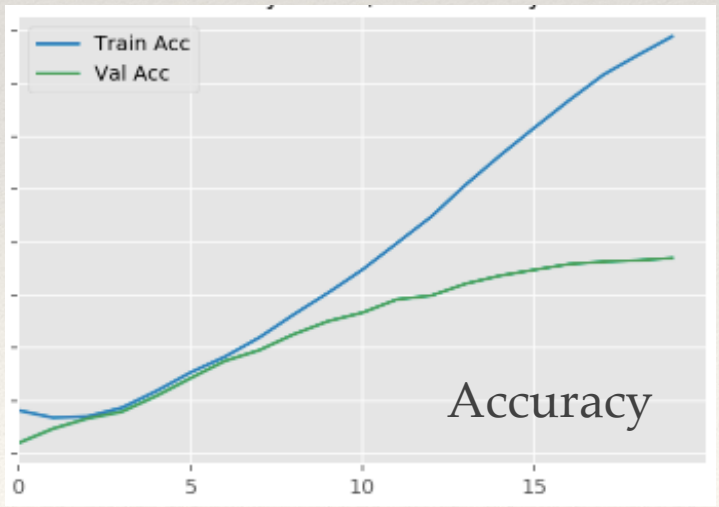
(remember to use a GPU.. → *CPU@colab: ~39m, GPU@colab: ~2m40s*)

(…)

```
Epoch 18/20
100/100 [==============================] - 8s 84ms/step - loss: 0.1790 - accuracy: 0.9275 - val_loss: 0.9475 - val_accuracy: 0.6930
Epoch 19/20
100/100 [==============================] - 8s 84ms/step - loss: 0.1634 - accuracy: 0.9395 - val_loss: 1.9811 - val_accuracy: 0.7020
Epoch 20/20
100/100 [==============================] - 8s 83ms/step - loss: 0.1319 - accuracy: 0.9475 - val_loss: 2.7144 - val_accuracy: 0.7030
```

~95% ~70%





Fit the model

fit_generator is different from the behaviour of fit, where increasing batch_size typically speeds up things

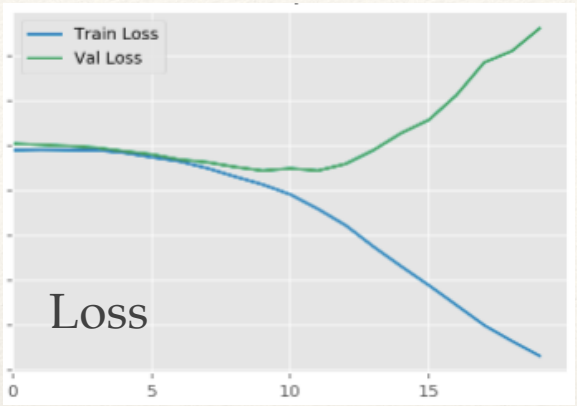
- Keras documentation for fit_generator: https://keras.io/models/sequential/

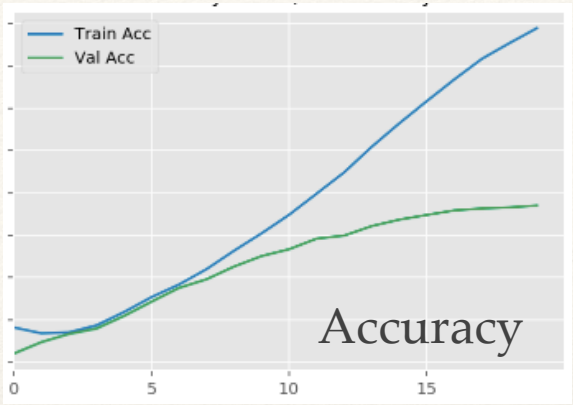
When you use fit_generator, the number of samples processed for each epoch is batch_size * steps_per_epochs.

- steps_per_epoch: total # of steps (batches of samples) to yield from generator before declaring one epoch finished and starting the next epoch. It should typically be equal to the number of unique samples of your dataset divided by the batch size.

In conclusion, when you increase batch_size with fit_generator, you should decrease steps_per_epochs by the same factor, if you want training time to stay the same or lower.

Model behaviour





Clear symptoms of overfitting

- training loss keeps decreasing..
- .. but validation loss starts increasing after epoch ~10

Overfitting despite dropout?!

- well, yes.. we are training for a tough task but on very few examples (**only 1000 images per category**). No matter which regularisation techniques we use: we will overfit on such a small dataset..
  - usually in DL we would need orders of magnitude more than this, e.g. **O(100k) training examples** would start to suffice?

A solution? → **Data augmentation**

The need at the basis of **Data augmentation**

Overfitting is due to training on too few examples → poor generalization performance

- if we had infinite training data, we wouldn't overfit by construction, because we would see (and learn from) every possible instance!

Often not easy to collect more data.. *what can we do with just the data we already have?*

Data augmentation in brief

Data augmentation: a way to generate more training data from our current training set. A sort of "**sample enrichment**", done by artificially increasing the size of the training set by generating many realistic variants of each training instance**.** How? Via random transformation of existing ones.

- this artificially boosts the size of the training set, thus reducing overfitting
- data augmentation can hence be also considered as a regularisation technique

How does this work? How do you create new samples?

Data augmentation (in some more detail)

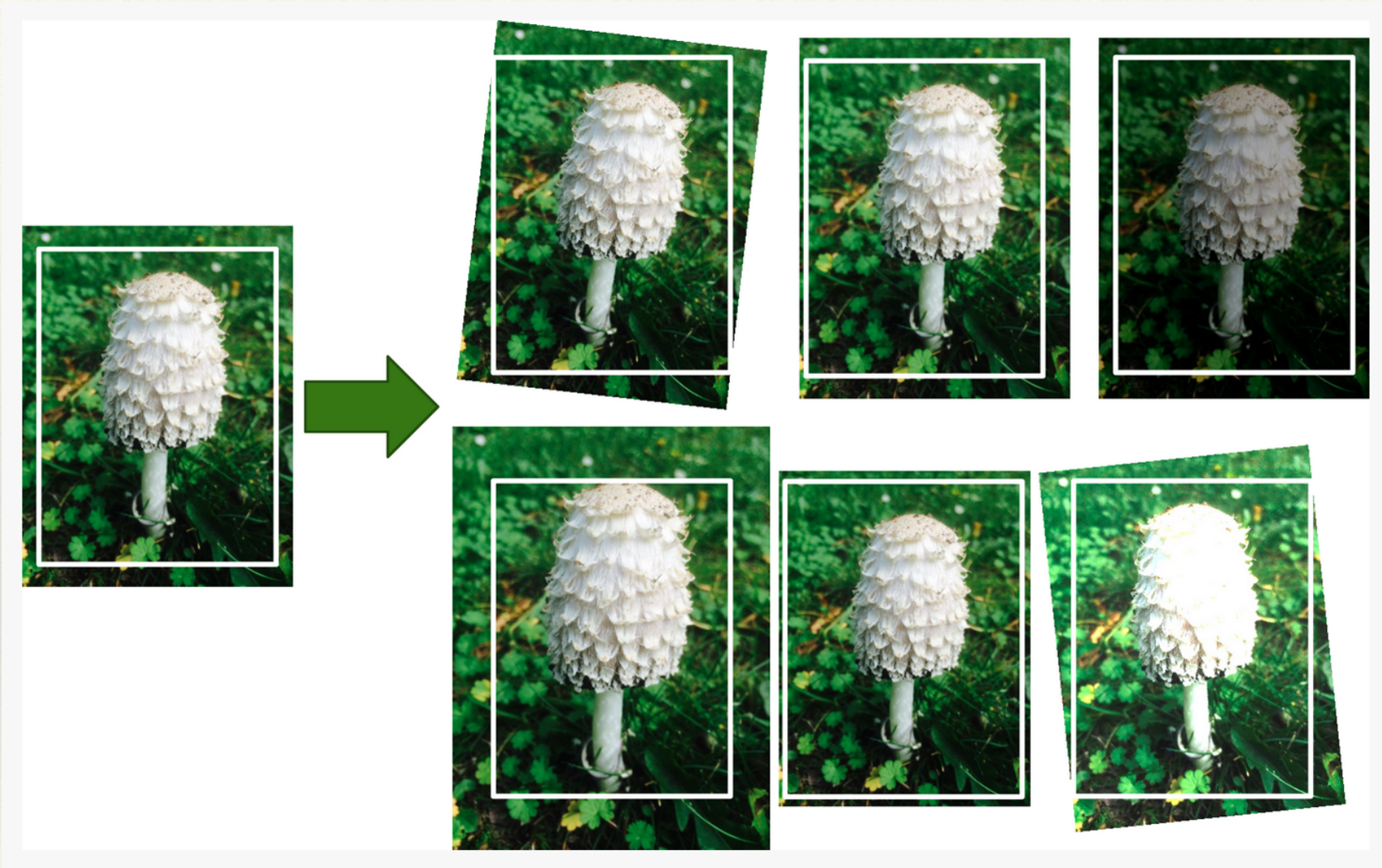
How?

The generated instances should be as realistic as possible

- ideally, given an image from the augmented training set, a human should not be able to tell whether it was augmented or not.
- Simply adding white noise will not help; **the modifications should be learnable** (white noise is not).

E.g. you can slightly shift, rotate, and resize every picture in the training set by various amounts and add the resulting pictures to the training set

- see next slide
- This forces the model to be more tolerant to variations in the position, orientation, and size of the objects in the pictures. For a model that's more tolerant of different lighting conditions, you can similarly generate many images with various contrasts. In general, you can also flip the pictures horizontally (except for text, and other asymmetrical objects). By combining these transformations, you can greatly increase the size of your training set.



Generating new training instances from existing ones

Data augmentation: the implementation

can boost the training set size even by large factors! (e.g. 50x)

(data cleaning tricks are applied, mainly *whitening* and *mean normalization*)

Data augmentation:

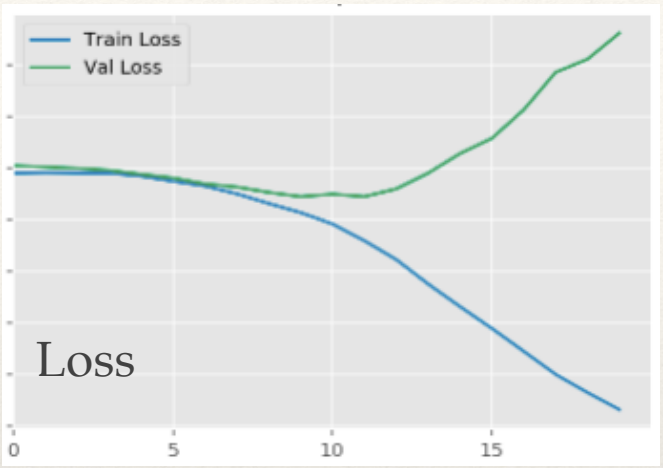
- done basically in every image-based DL model
- implemented dynamically during training time
- done via generation of realistic images, with “learnable" transformations, as we said
  - rotation, shifting, resizing, exposure adjustment, contrast change, etc (simply adding noise won’t help)
- only performed on the training data (we don’t touch the validation or test set!)

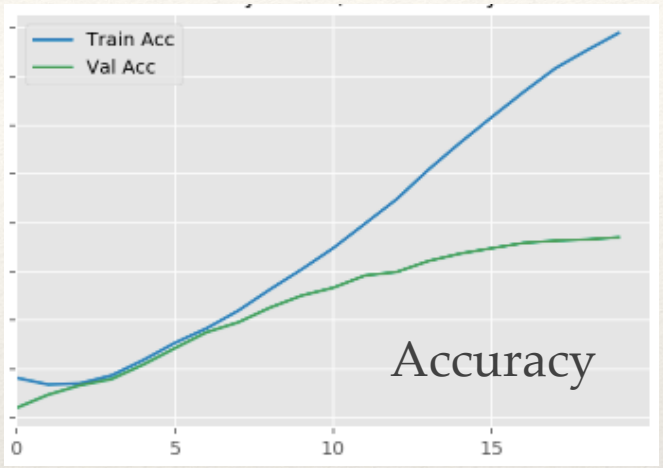
We can hence generate a lot of new samples from even a single training example!

- “this is still a cat image”. We can infer it as a human, so the model should be able to learn that as well.

Impact of data augmentation

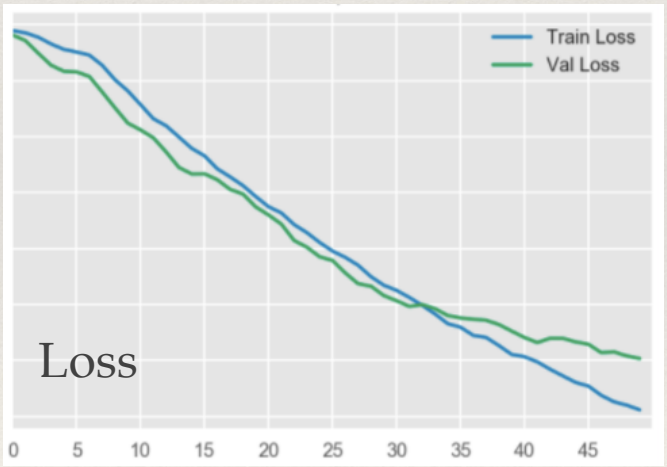
Baseline

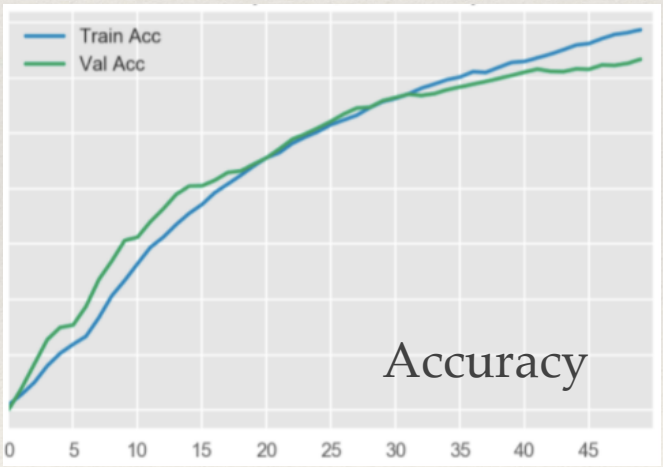




w/ data augmentation

(and more epochs)





CPU@colab: ~3h14m (!!!)

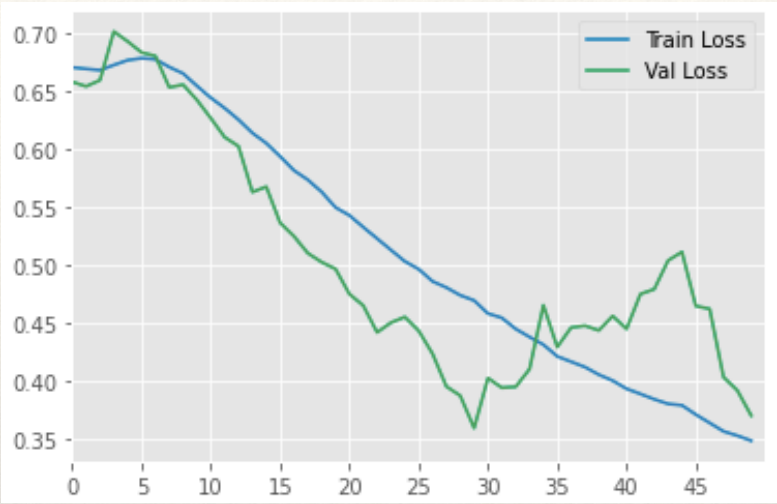
GPU@colab: ~26m

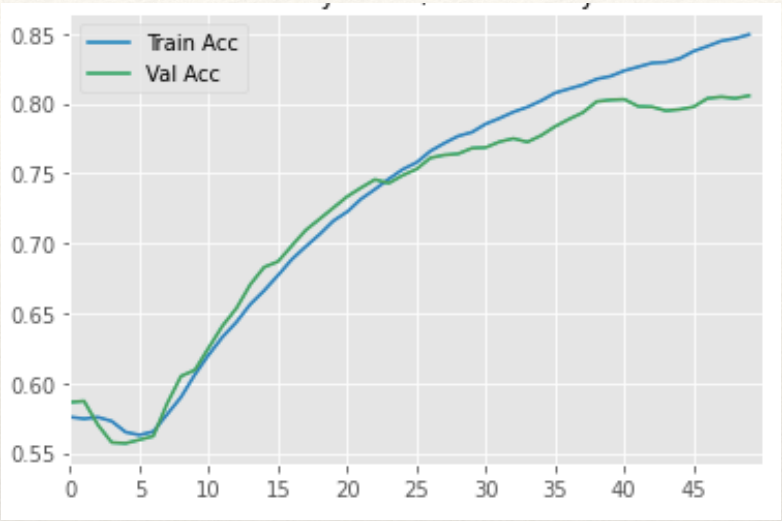
[DISCLAIMER: this is not a benchmark test!]

3h 14m

```
Epoch 58/60
100/100 [==============================] - 193s 2s/step - loss: 0.3773 - accuracy: 0.8301 - val_loss: 0.2505 - val_accuracy: 0.7919
Epoch 59/60
100/100 [==============================] - 192s 2s/step - loss: 0.3851 - accuracy: 0.8248 - val_loss: 0.5514 - val_accuracy: 0.7854
Epoch 60/60
100/100 [==============================] - 192s 2s/step - loss: 0.3945 - accuracy: 0.8156 - val_loss: 0.7709 - val_accuracy: 0.8135
CPU times: user 6h 15min 45s, sys: 2min 55s, total: 6h 18min 40s
Wall time: 3h 14min 13s
```

NOTE: actual results may vary..





Impact of data augmentation

Applying data augmentation to our CNN does not affect the model definition

- we are not touching the model architecture, the only change is how we feed in the data
  - https://keras.io/api/preprocessing/image/

Results are much better now! (see previous slide), because:

- we are training on more images with variety
- we made the model transformation invariant
  - the model saw a lot of shifted/rotated/scaled cats, so it's able to recognize cats better