# Data Analysis and Machine Learning 4

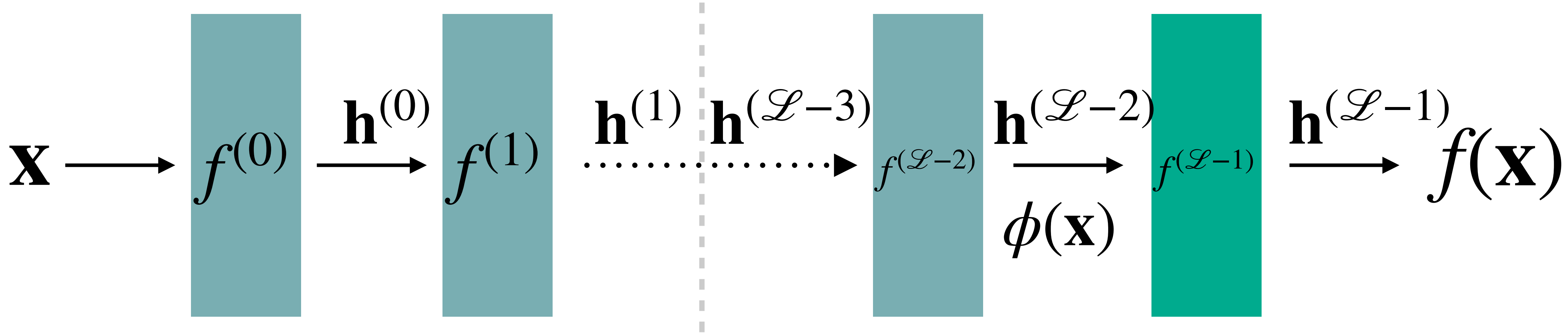
## Week 10: Convolutional neural networks

**Elliot J. Crowley, 27th March 2023**

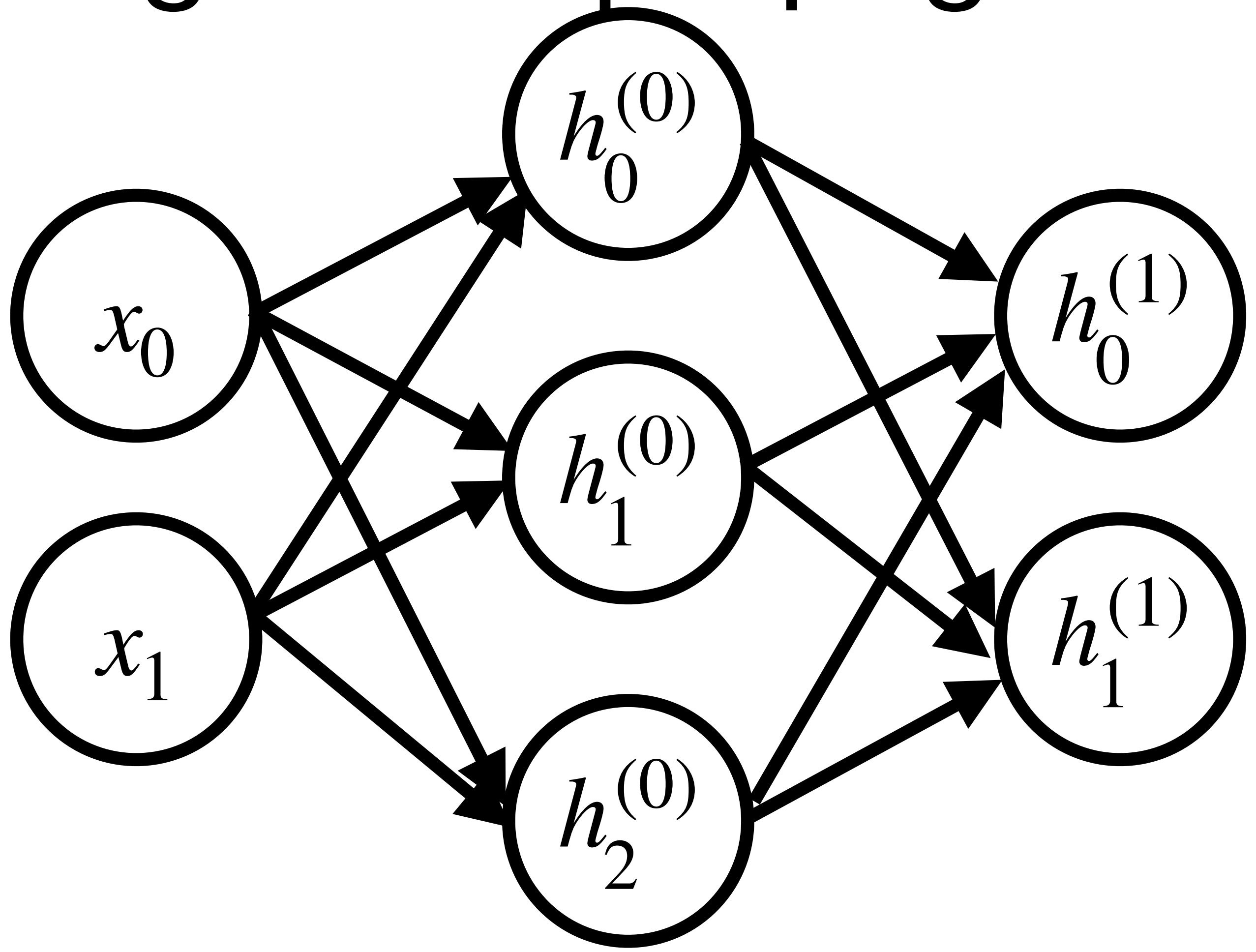
THE UNIVERSITY of EDINBURGH

# Recap

- We learnt about deep neural networks (DNNs) as models that incorporate feature learning into a given task



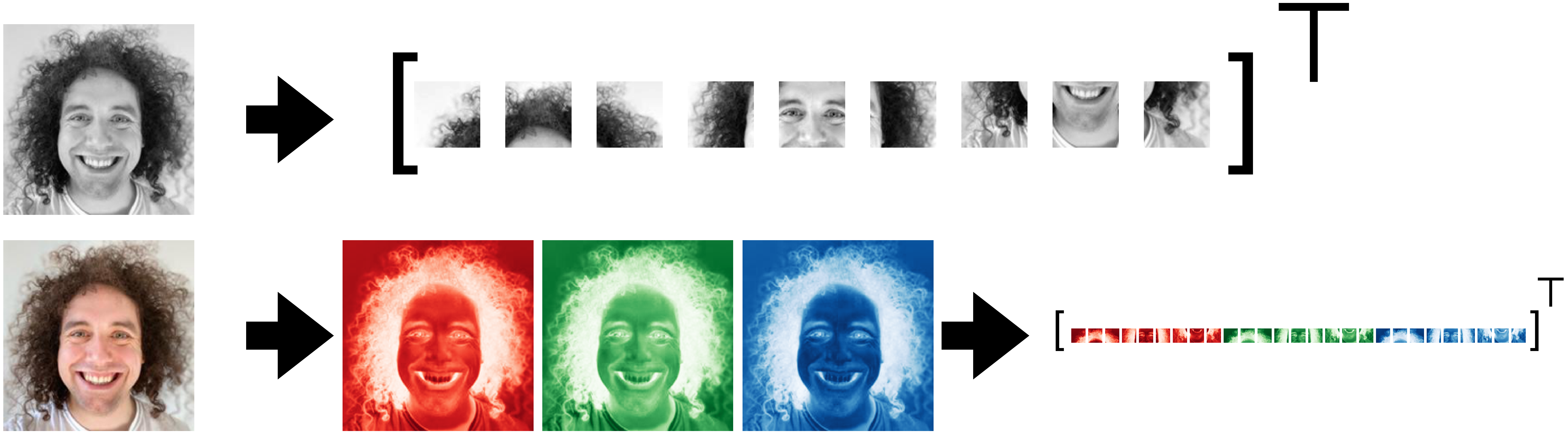

- We examined MLPs and how to learn their weights using the gradients obtained through backpropagation



# Convolutional Neural Networks (ConvNets)

# Images

- So far we have represented all our data points as vectors $\mathbf{x} \in \mathbb{R}^D$
- This makes sense with tabular data. Each dimension has a distinct meaning
- Does it make sense to vectorise images?



# Location, location, location

- Objects can be in different places and at different scales across images
- If you vectorise then you are rarely comparing like-for-like at each dimension

# Structure

- Objects have a spatial structure. The position of relative parts is important
- We lose this information if we vectorise

# Locality

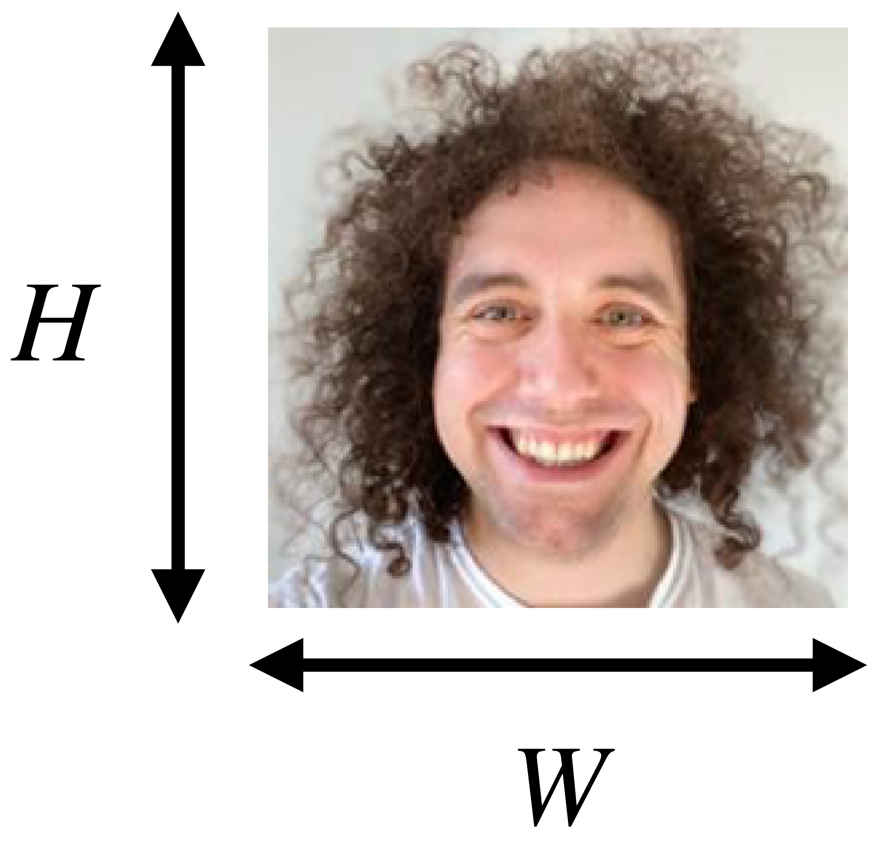
- In an image, pixels near each other tend to relate to the same object
- We lose any sense of locality when we vectorise images

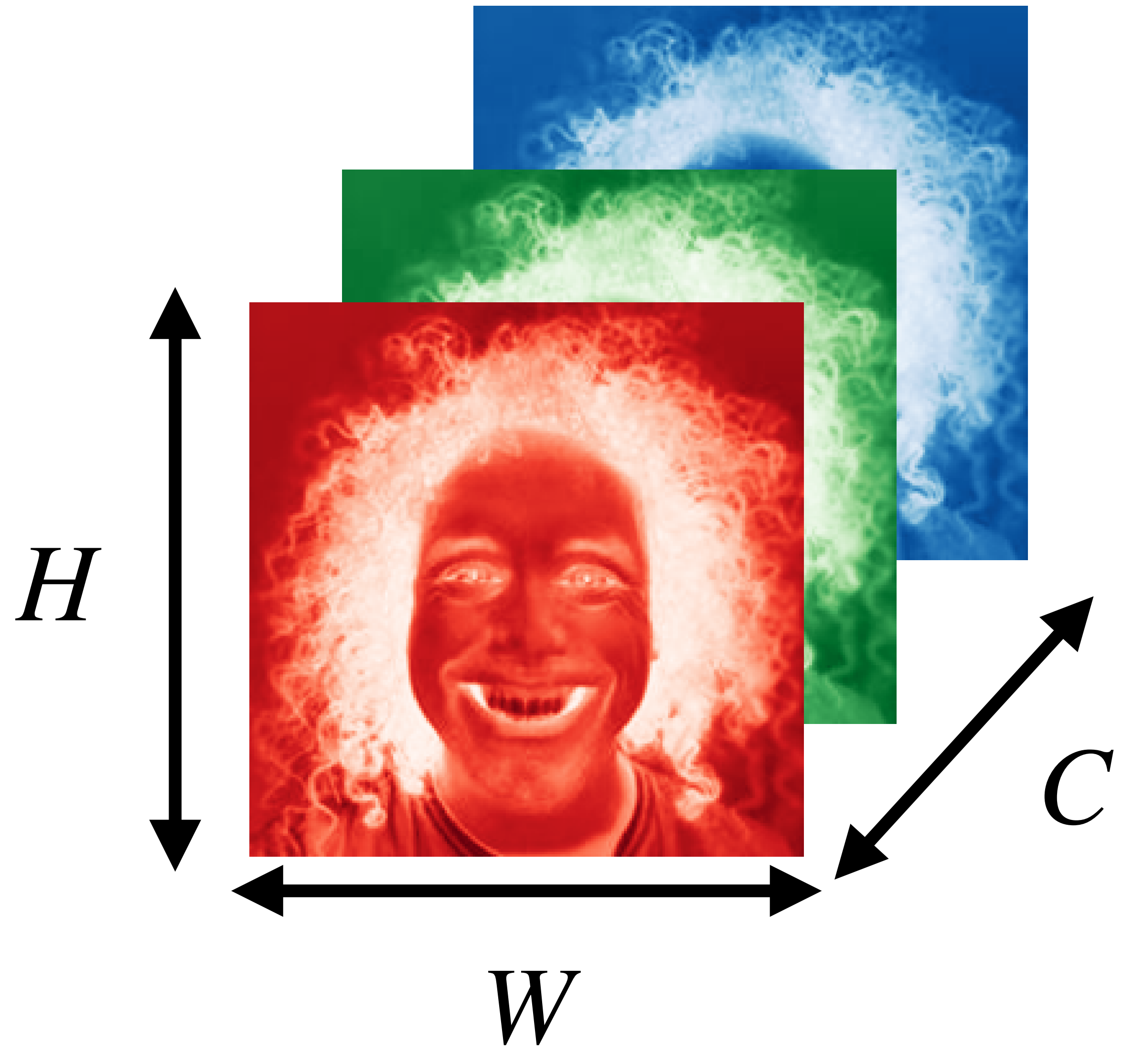
We count “person” as an object. This isn’t meant to be derogatory!

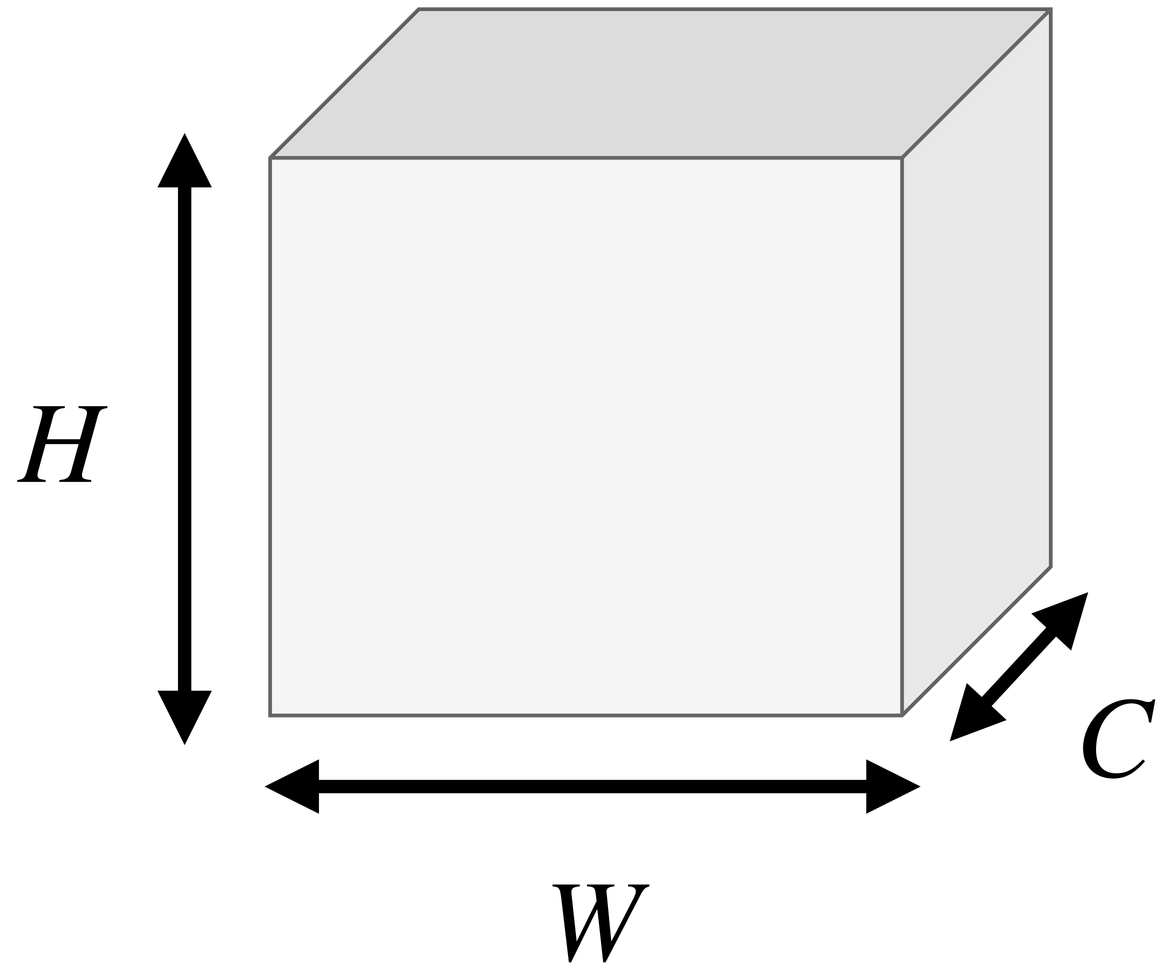
# Spatial information is important

- Let's keep the image in its original form! This is a **cube(/oid)** with dimensions $H \times W \times C$ where $C$ is the number of colour channels (almost always 3)
- We can represent this mathematically using a 3D tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$
- In Python this is just a 3D array

Having channels first is PyTorch notation

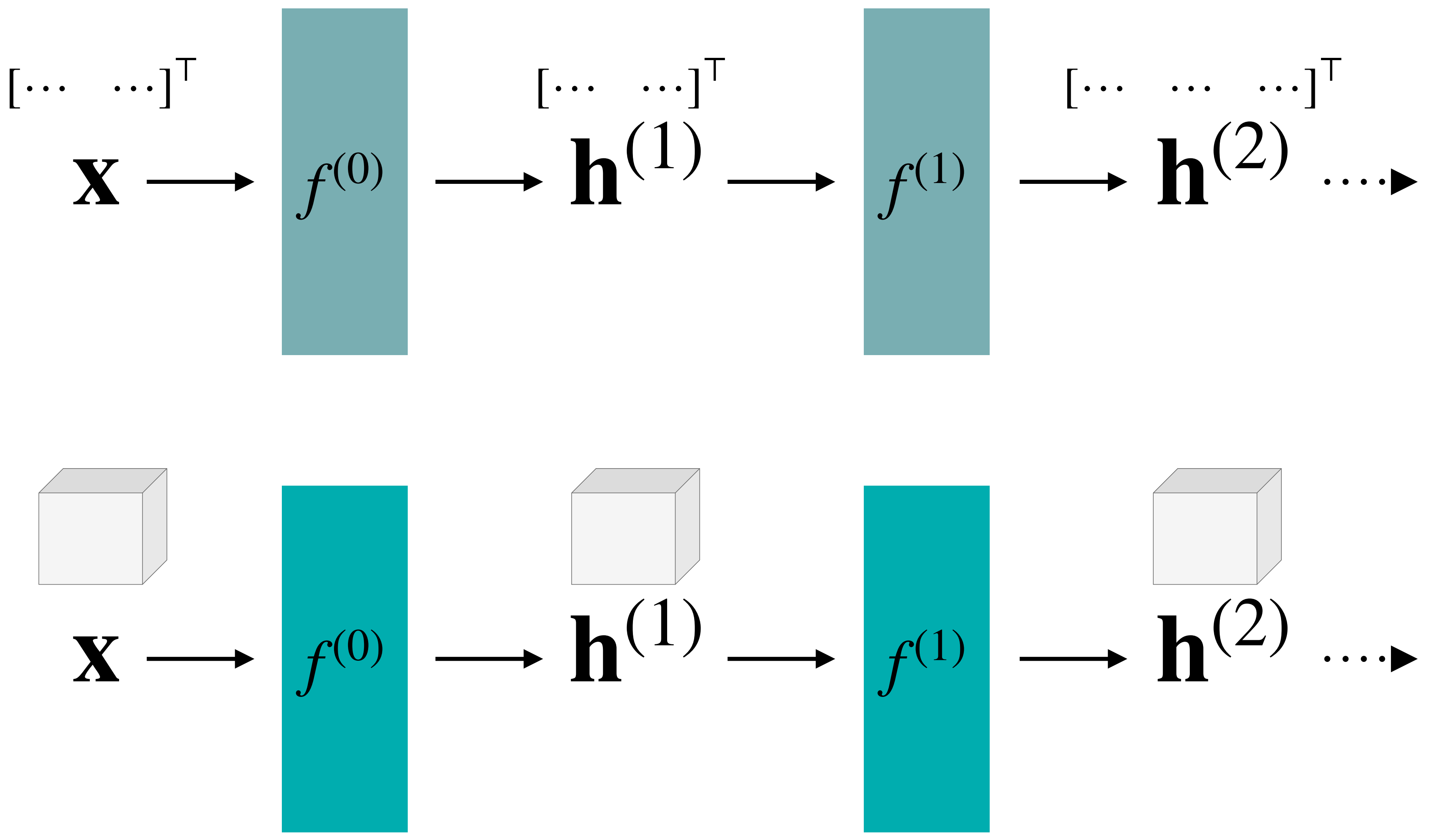


# We can't use an MLP any more :(

- We want to use a DNN $f(\mathbf{x}) = f^{(\mathscr{L}-1)} \circ f^{(\mathscr{L}-2)} \circ \ldots \circ f^{(1)} \circ f^{(0)}(\mathbf{x})$ on images
- The **fully-connected layers** that make up an MLP work on vectors
- We need a new functional layer that works for 3D tensors




$$\mathbf{h}^{(l)} = f^{(l)}(\mathbf{h}^{(l-1)}) = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

$$\mathbf{h}^{(l)} = f^{(l)}(\mathbf{h}^{(l-1)}) = g(?)$$

# Convolutions

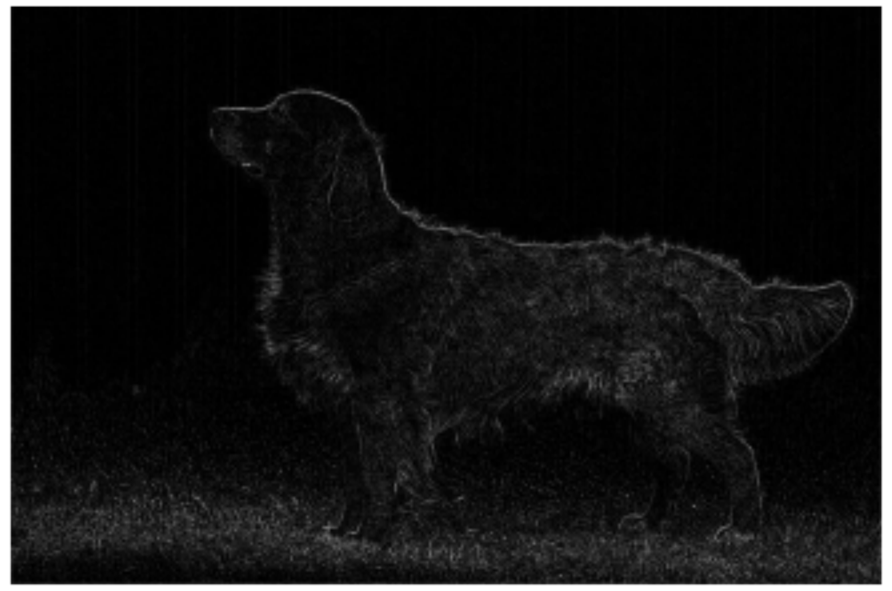
- We don't just want a functional layer that *works*
- We want something that is computationally efficient and suitable given all the things we know about images
- **2D convolutions** fit this brief and are used heavily in image processing
- These populate most layers in Convolutional neural networks (ConvNets)

# 2D Convolution with a single filter

- The 2D convolutions in ConvNets consist of multiple filters
- Let's see how 2D convolution with a single filter works
- We will consider a 2D input (e.g. a grayscale image) for now
- For these, a filter is a $k \times k$ matrix where $k$ is the kernel size

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} *$$

$$=$$



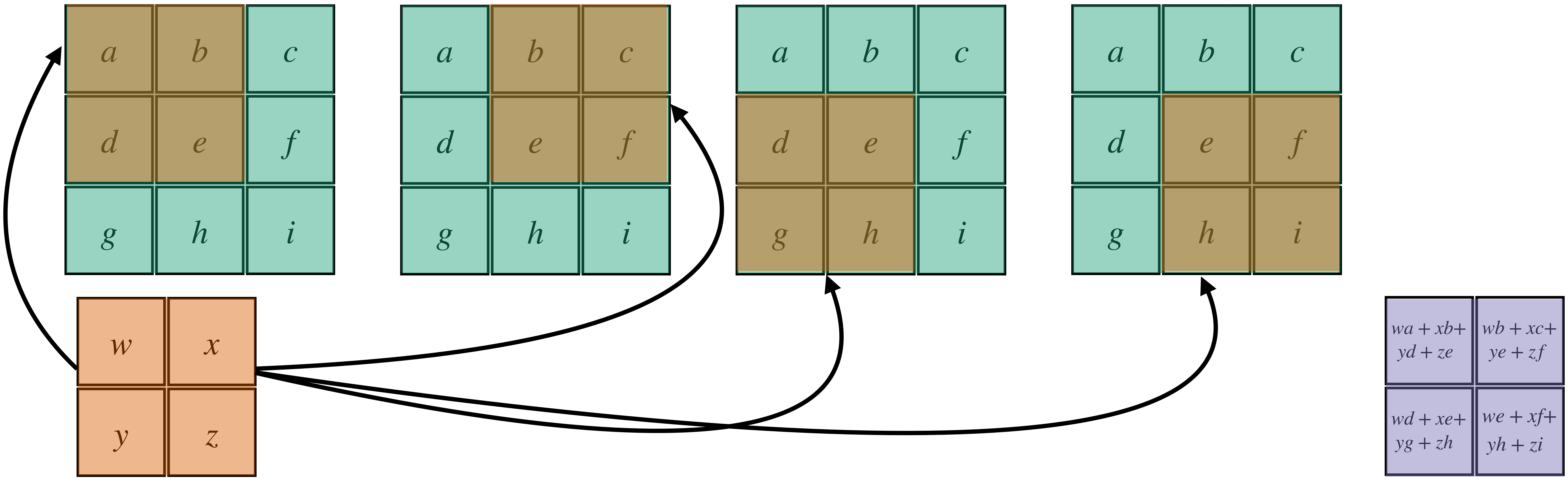
# 2D Convolution with a single filter

- We place the filter over the input and slide it around to every possible position
- At each position, we take the dot product between the filter and the overlapping input elements
- This result is stored in the corresponding position of the output matrix

$$\begin{bmatrix} w & x \\ y & z \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} wa + xb + yd + ze & wb + xc + ye + zf \\ wd + xe + yg + zh & we + xf + yh + zi \end{bmatrix}$$

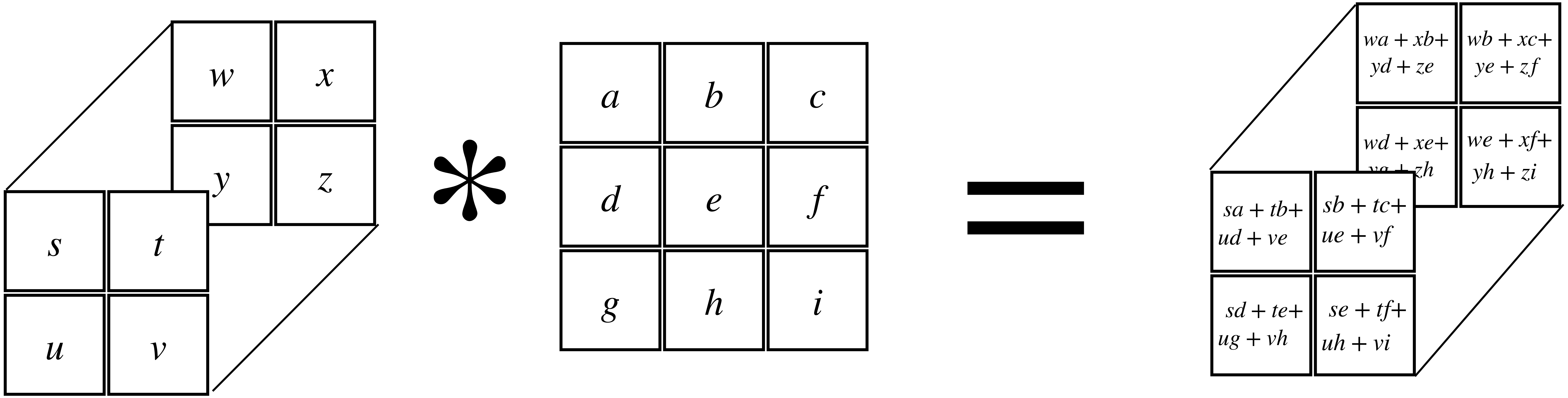
# 2D Convolution with a single filter

- There are four possible places this filter can go
- These correspond to the four elements of the output matrix
- We take the dot product at each position and store it in the output



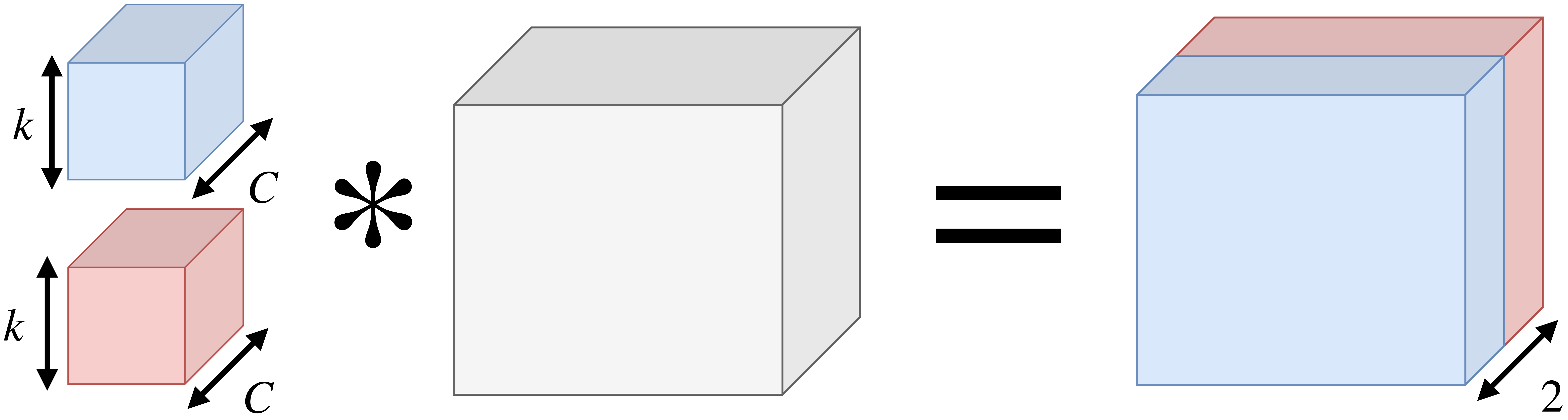
# 2D Convolution with multiple filters

- One filter gave us one output matrix
- Two filters gives us two output matrices that we stack to form a tensor
- And so on…



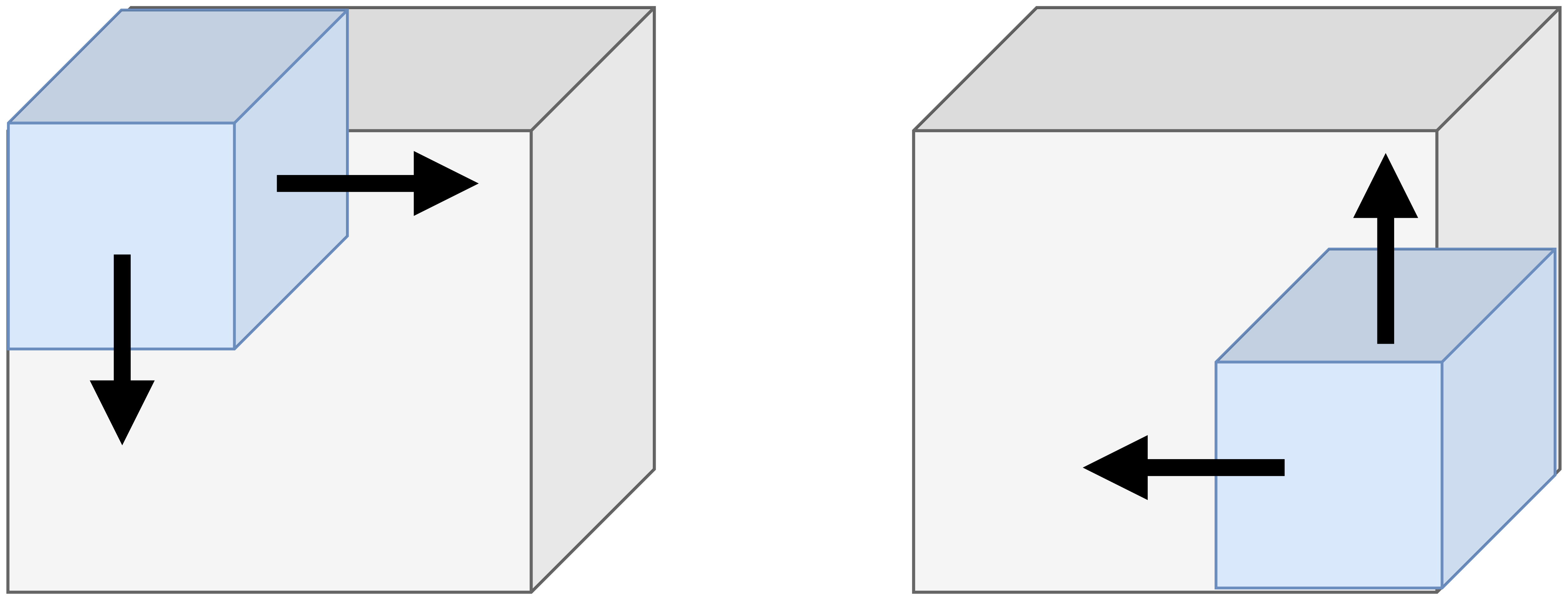
# What happens if the inputs are 3D?

- We can still perform a 2D convolution on a 3D input
- We get one output matrix per filter as before
- The only difference is that the filters are $C \times k \times k$ tensors (cubes)



# Cube in a cube

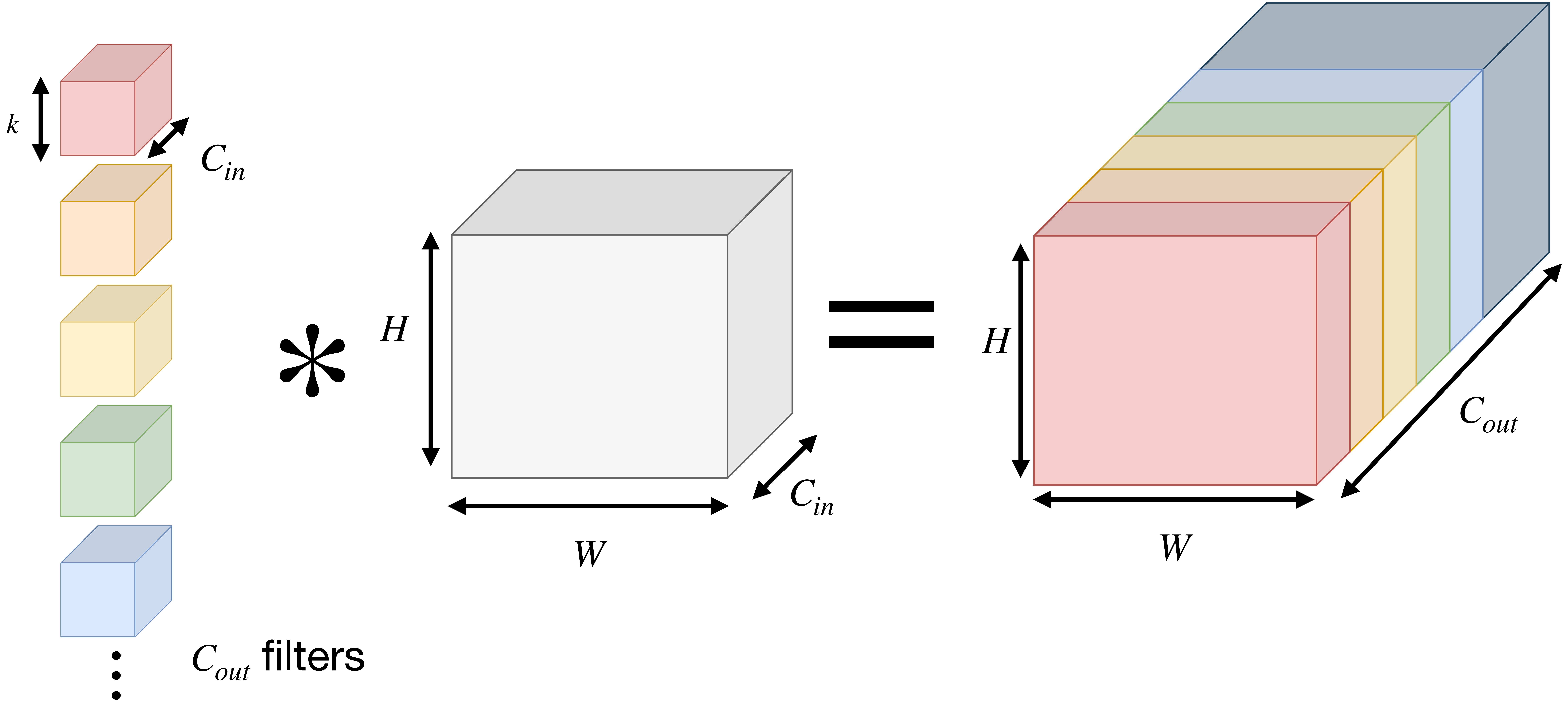
- Picture sliding the filter cube around inside the input cube
- It can't move along the $z$ axis because the cubes have the same depth
- It can only move left/right and up/down
- At each position, you take a dot product and store it in a matrix



# Padding

- It is common practice to pad the input with zeros so that the input and output have the same height and width after a convolution
- We will assume that this always happens hereon for ease

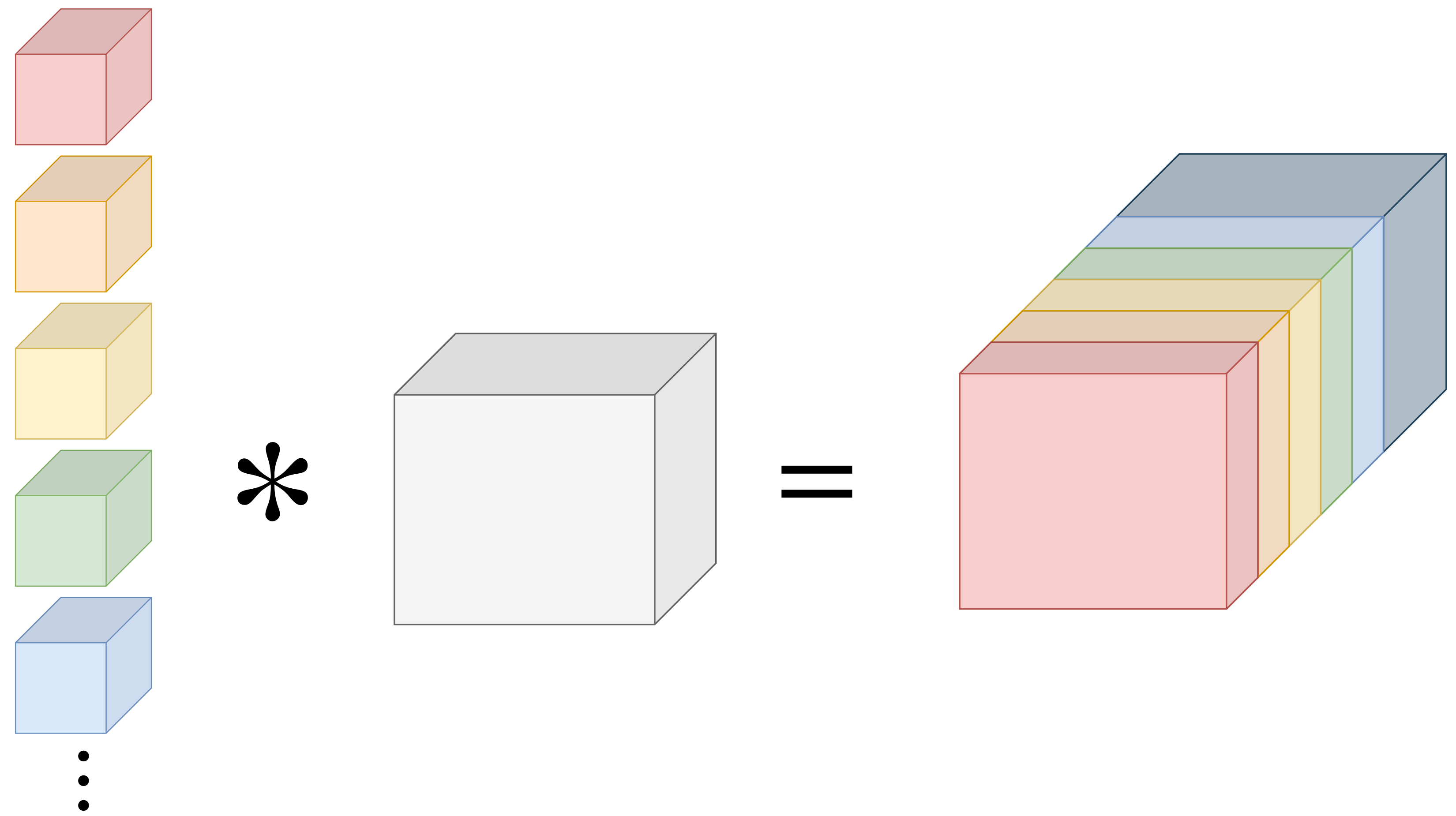
# Conv2D




$\mathbf{W} \in \mathbb{R}^{C_{out} \times C_{in} \times k \times k}$ $\quad$ $\mathbf{x} \in \mathbb{R}^{C_{in} \times H \times W}$ $\quad$ $\mathbf{y} \in \mathbb{R}^{C_{out} \times H \times W}$

# Convolutional layers

- After all that, we can finally unveil what a convolutional layer looks like!
- In an MLP we had $\mathbf{h}^{(l)} = f^{(l)}(\mathbf{h}^{(l-1)}) = g(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$
- A convolutional layer looks like $\mathbf{h}^{(l)} = f^{(l)}(\mathbf{h}^{(l-1)}) = g(\mathbf{W}^{(l)} * \mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$
- That's it!



# Why convolutions?

- They are much more parameter-efficient than the matrices seen MLPs
- e.g. Let's have a simple 2 layer MLP for classifying my face vs. other faces
- Let's go with hidden dimension 100. How many parameters does $\mathbf{W}^{(0)}$ use?

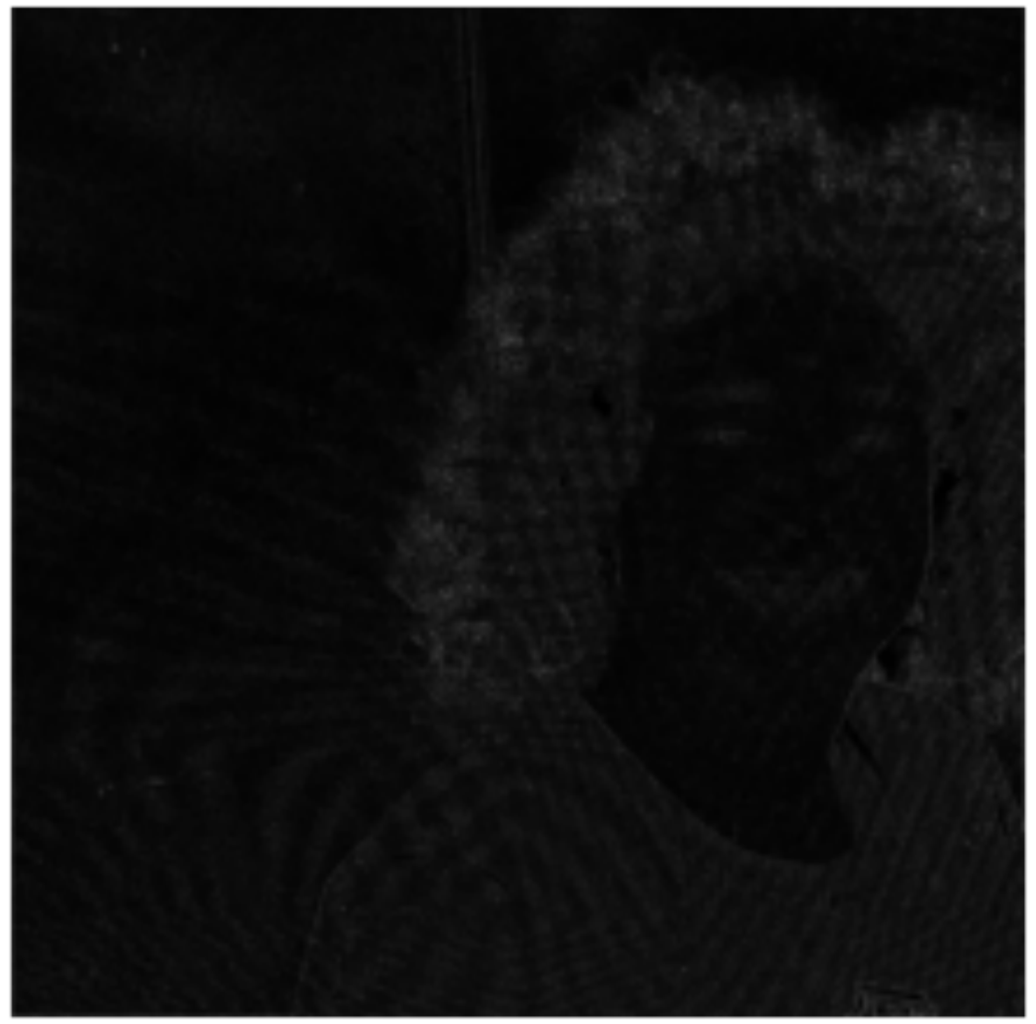
$\mathbf{x} \in \mathbb{R}^{3 \times 224 \times 224}$ $\xrightarrow{\text{vectorise}}$ $[\;]$ $\mathbf{x} \in \mathbb{R}^{150528}$

$\mathbf{W}^{(0)}$ needs to be $100 \times 150528$

That's 15 million parameters!

# Why convolutions?

- Filters are applied to the whole image, they aren't tied to a certain region
- This means they can deal with objects moving: they'll produce a similar output response, just at a different location
- They are **equivariant** to translation



# Pooling layers

- There's one last thing to cover before we can look at a whole ConvNet
- Pooling layers — these reduce the spatial resolution of their input by aggregating nearby elements
- Let's look at an example on an 2D input of a max pooling layer with $k = 2$

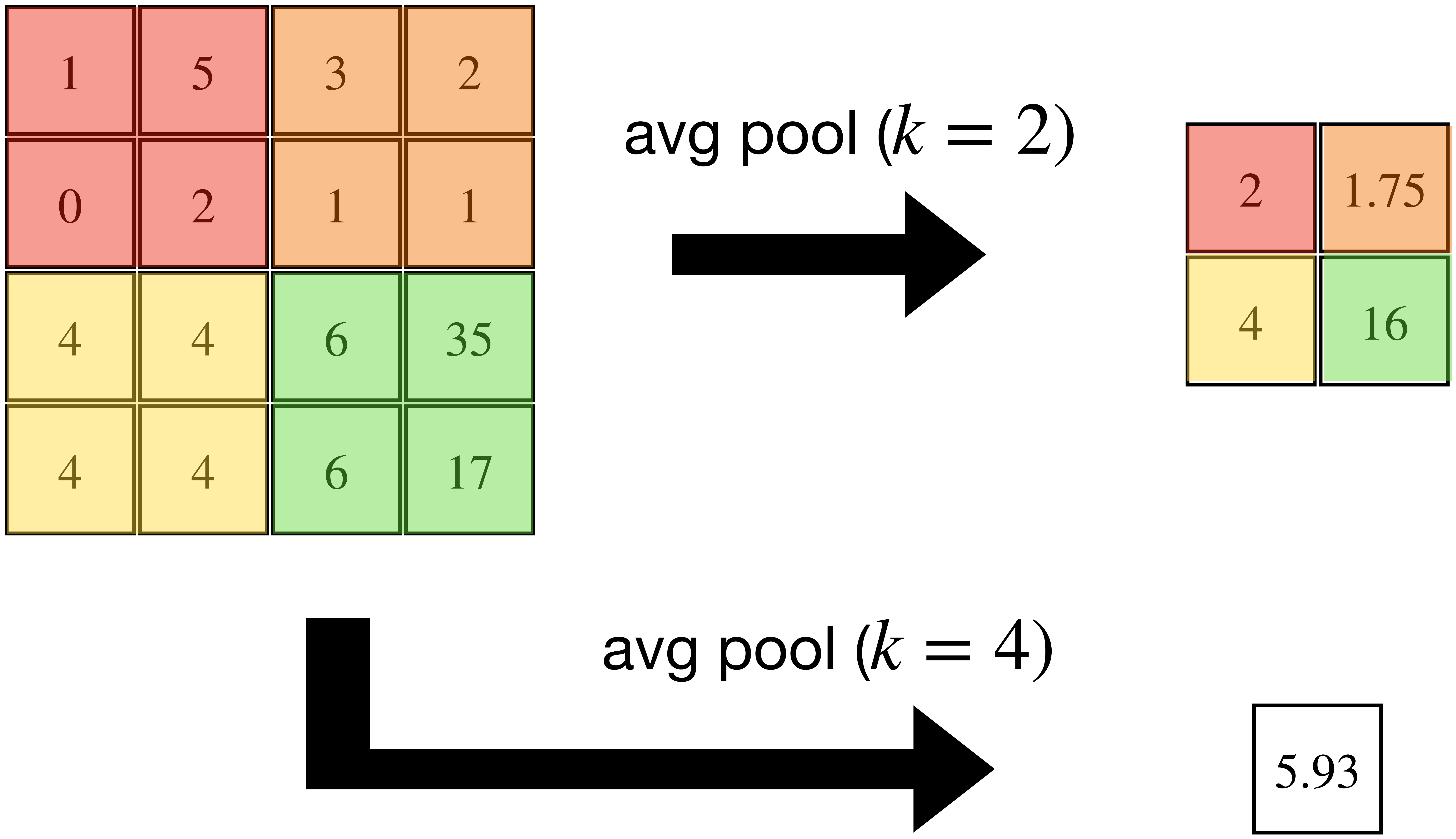
$$\begin{bmatrix}1&5&3&2\\0&2&1&1\\4&4&6&35\\4&4&6&17\end{bmatrix} \xrightarrow{\text{max pool } (k=2)} \begin{bmatrix}5&3\\4&35\end{bmatrix}$$

**The input has been split into $2 \times 2$ blocks**

**The output matrix contains the maximum value within each block**

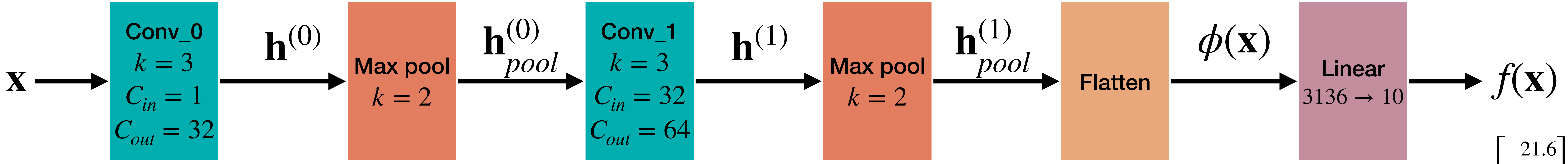
**It's spatial resolution has been halved**

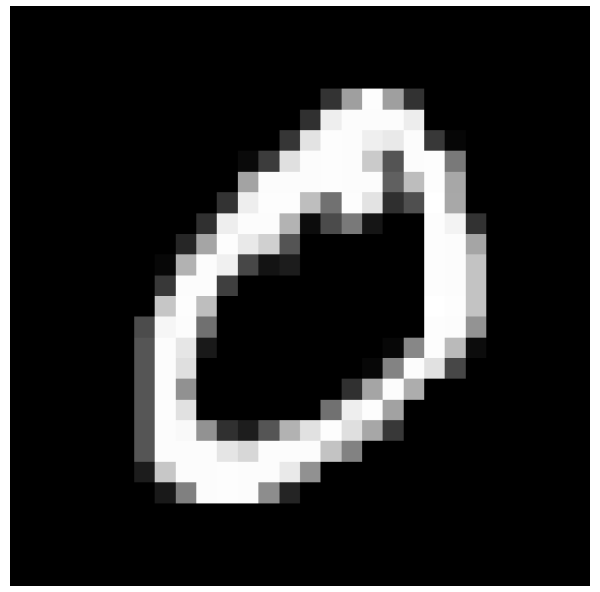
# Average pooling



# ConvNets

- ConvNets consist of assorted convolutional and pooling layers, and end with one or more fully-connected layers, the last of which is (usually) **linear**
- Let's look at a small ConvNet architecture trained to classifying MNIST digits
- The 10D output gives the logits for each class 0,1,2,3,…



The input (left) is a $28 \times 28$ grayscale image

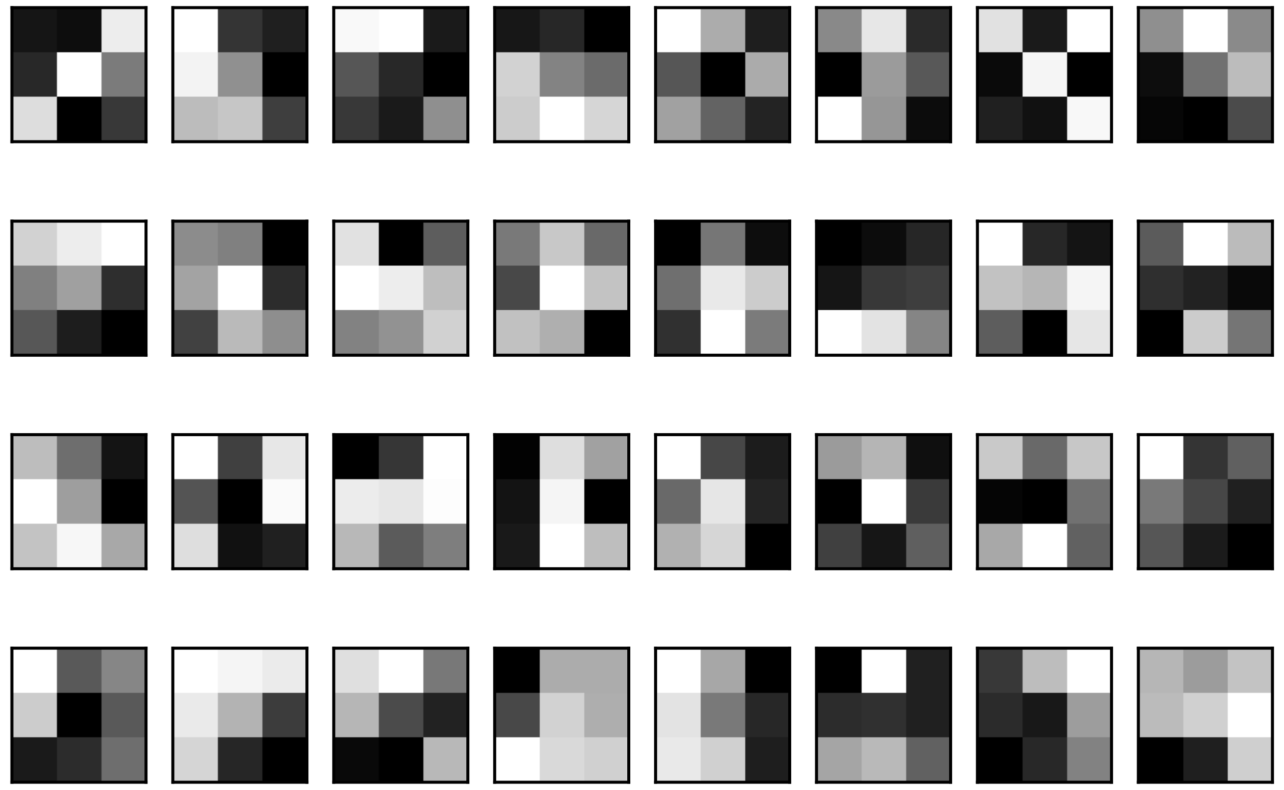
The output (right) is a vector of logits for each class

We can classify our input as $\arg\max f(\mathbf{x})$

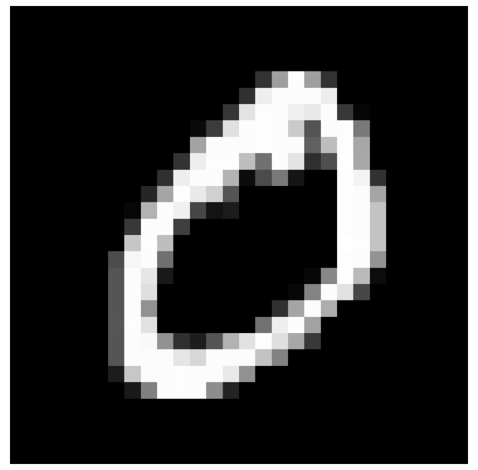
$$\begin{bmatrix} 21.6 \\ -12.3 \\ 5.3 \\ -7.0 \\ -2.9 \\ -6.9 \\ 4.3 \\ -6.6 \\ 0.9 \\ 0.4 \end{bmatrix}$$

# Conv_0

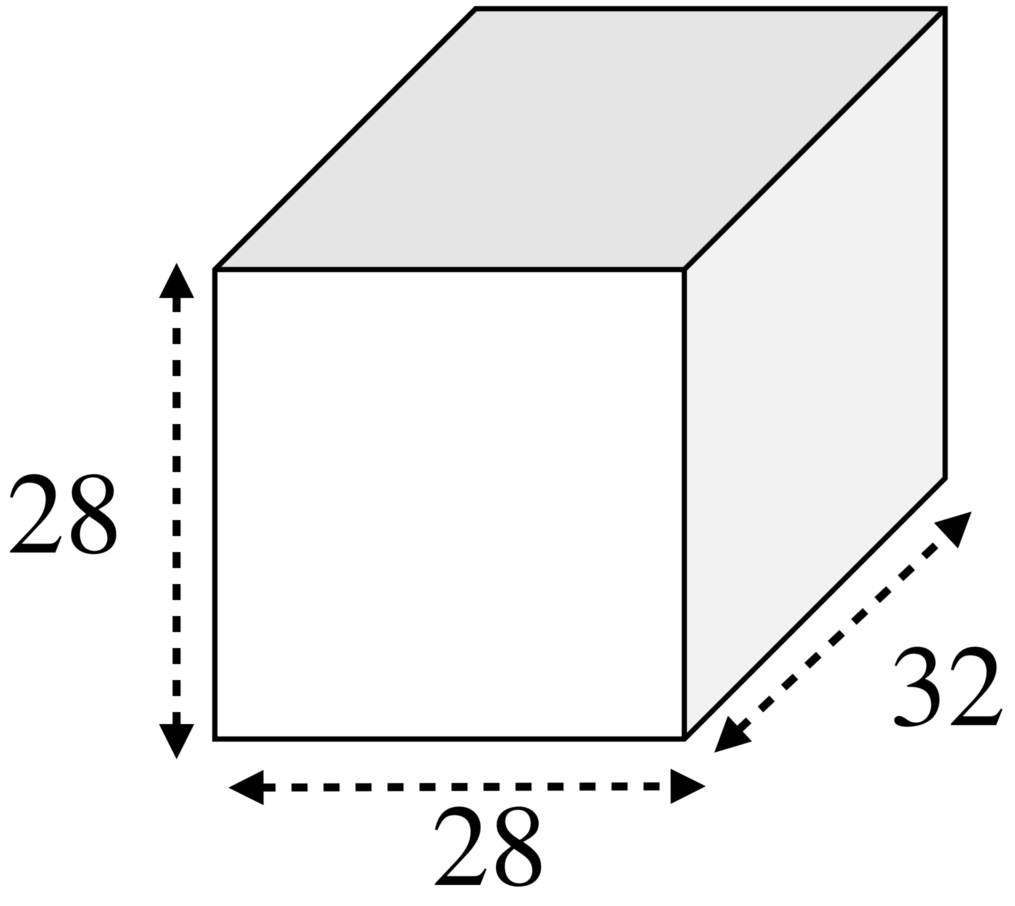
- This layer takes our (padded) image input and applies 32 filters
- We then add a bias to each output channel and apply a ReLU non-linearity



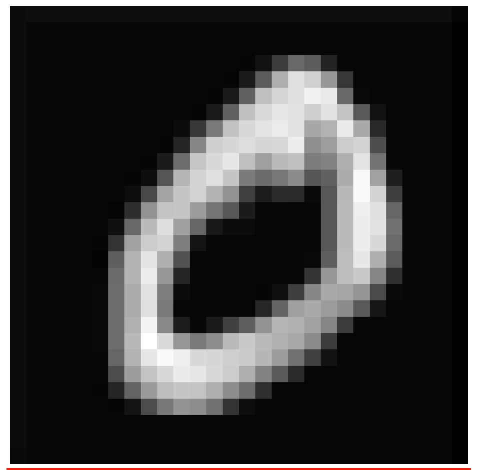
$\mathbf{W}^{(\mathbf{0})} \in \mathbb{R}^{32 \times 1 \times 3 \times 3}$



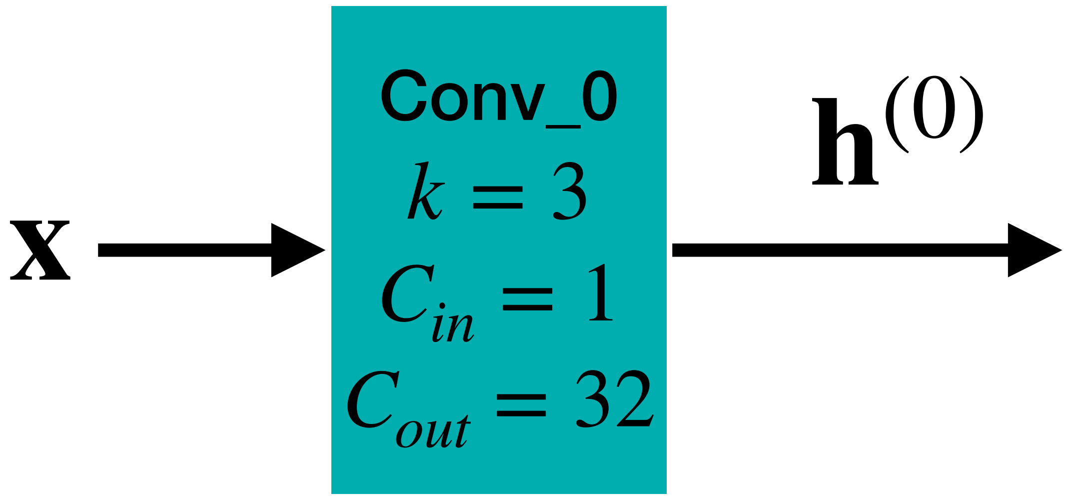
$\mathbf{x} \in \mathbb{R}^{1 \times 28 \times 28}$




$\mathbf{h}^{(0)} \in \mathbb{R}^{32 \times 28 \times 28}$

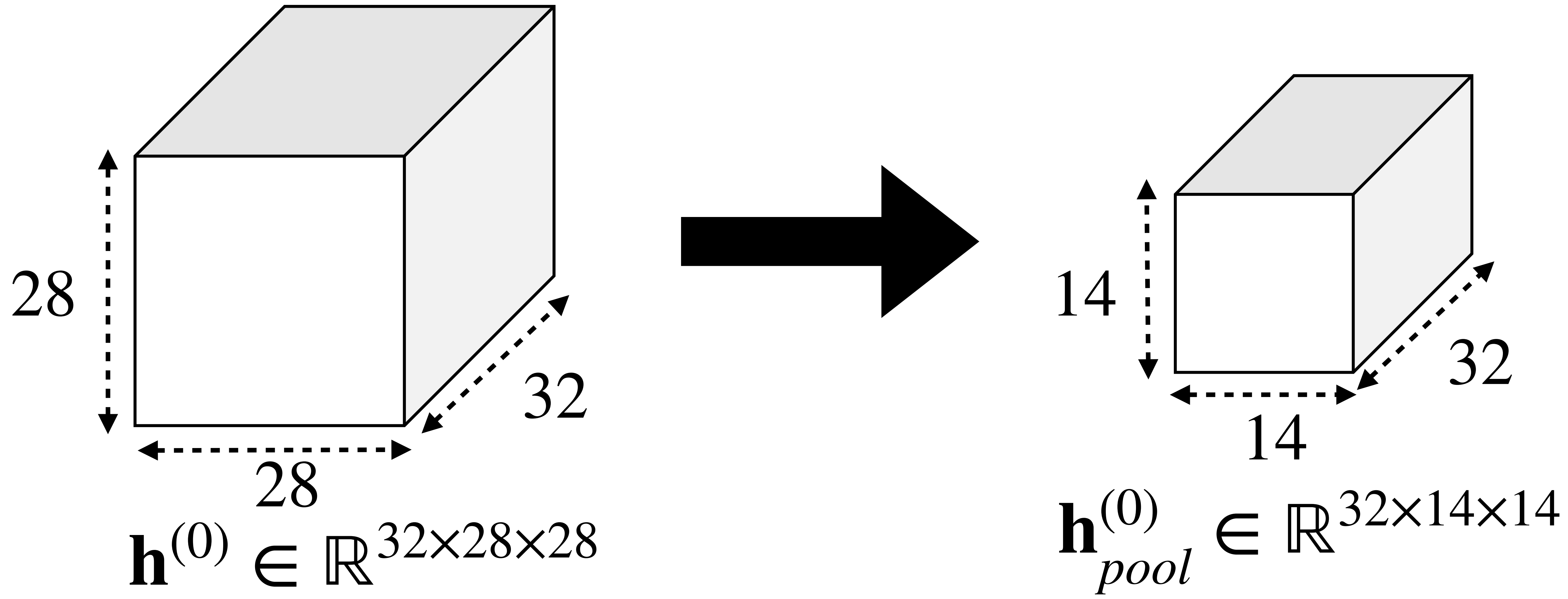


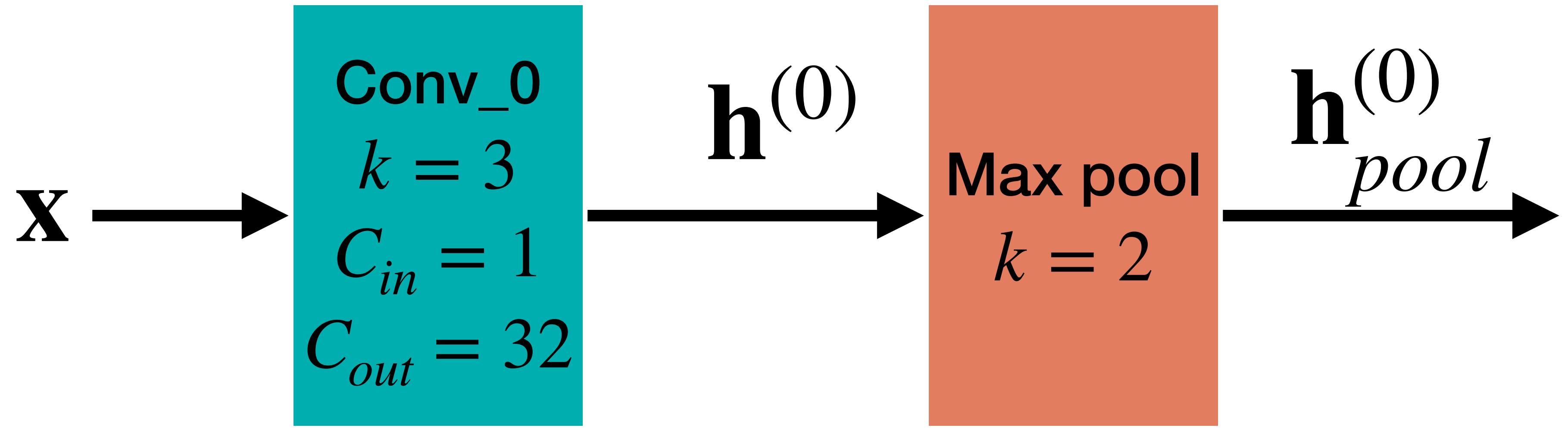
These are the average activations across all channels



# Max pool

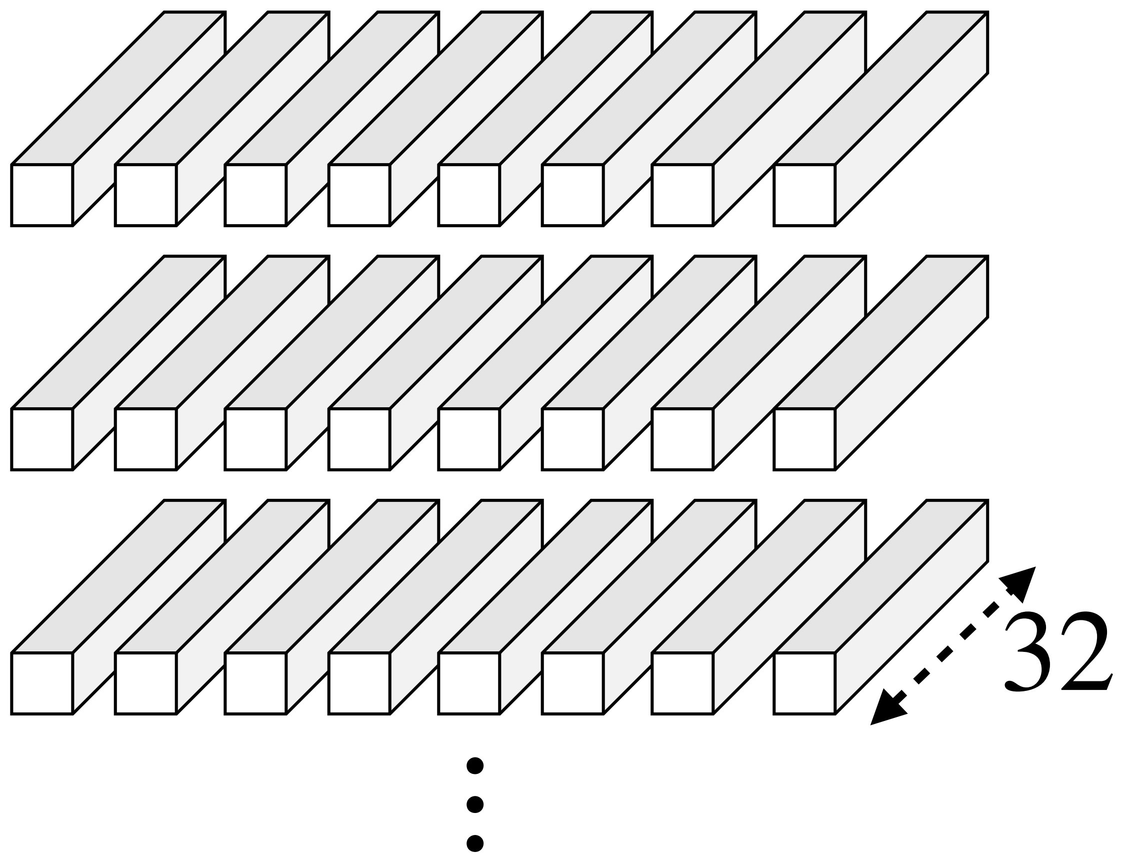
- This reduces spatial resolution
- The purpose of this is to build **translation invariance** into our representations



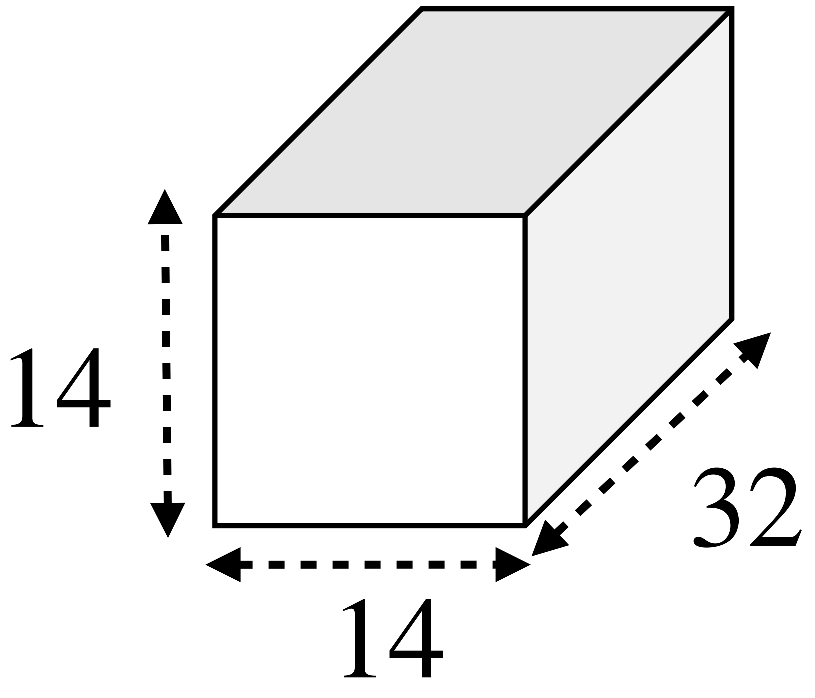



# Conv_1

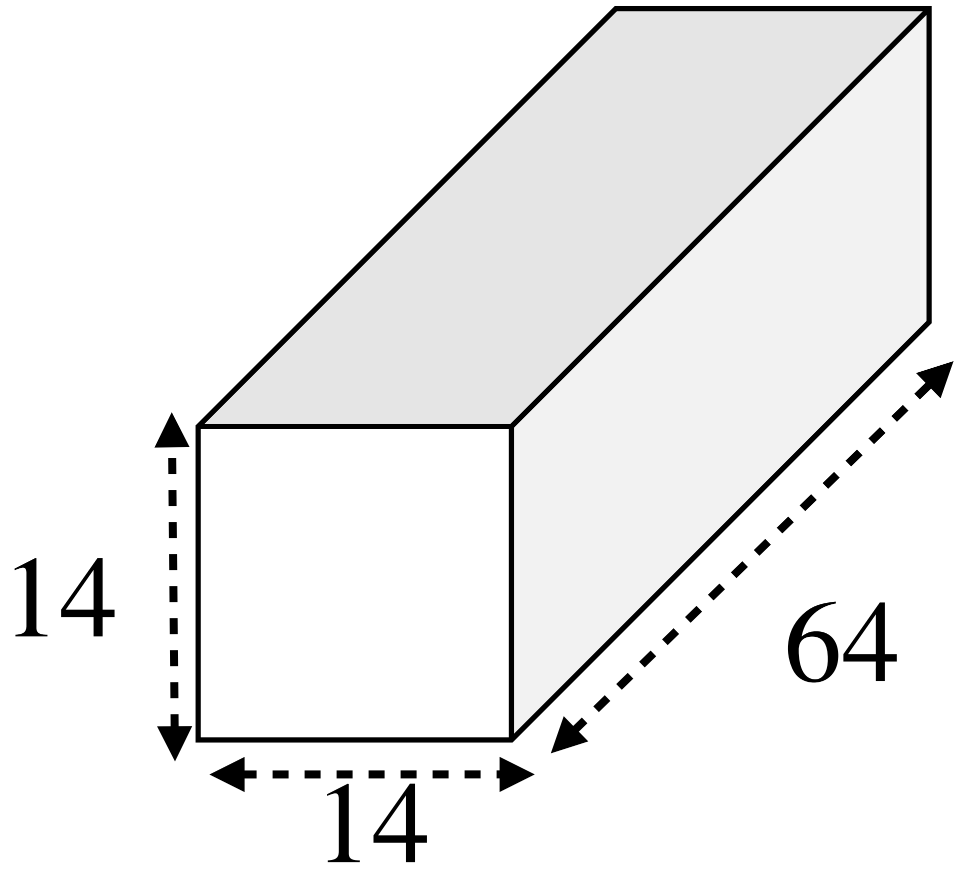
- This layer takes our (padded) pooled represented and applies 64 filters
- We then add a bias to each output channel and apply a ReLU non-linearity



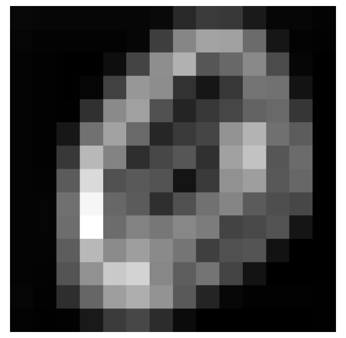

$\mathbf{W}^{(1)} \in \mathbb{R}^{64\times32\times3\times3}$



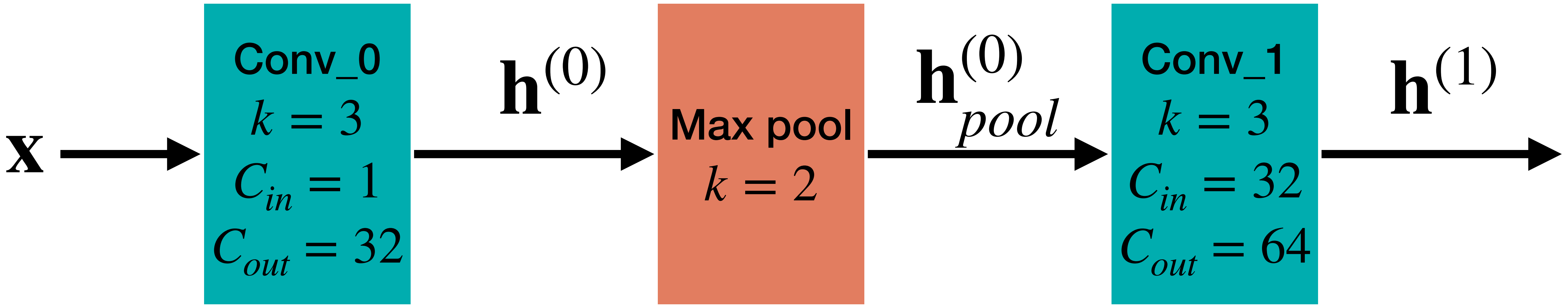

$\mathbf{h}^{(0)}_{pool} \in \mathbb{R}^{32\times14\times14}$




$\mathbf{h}^{(1)} \in \mathbb{R}^{64\times14\times14}$

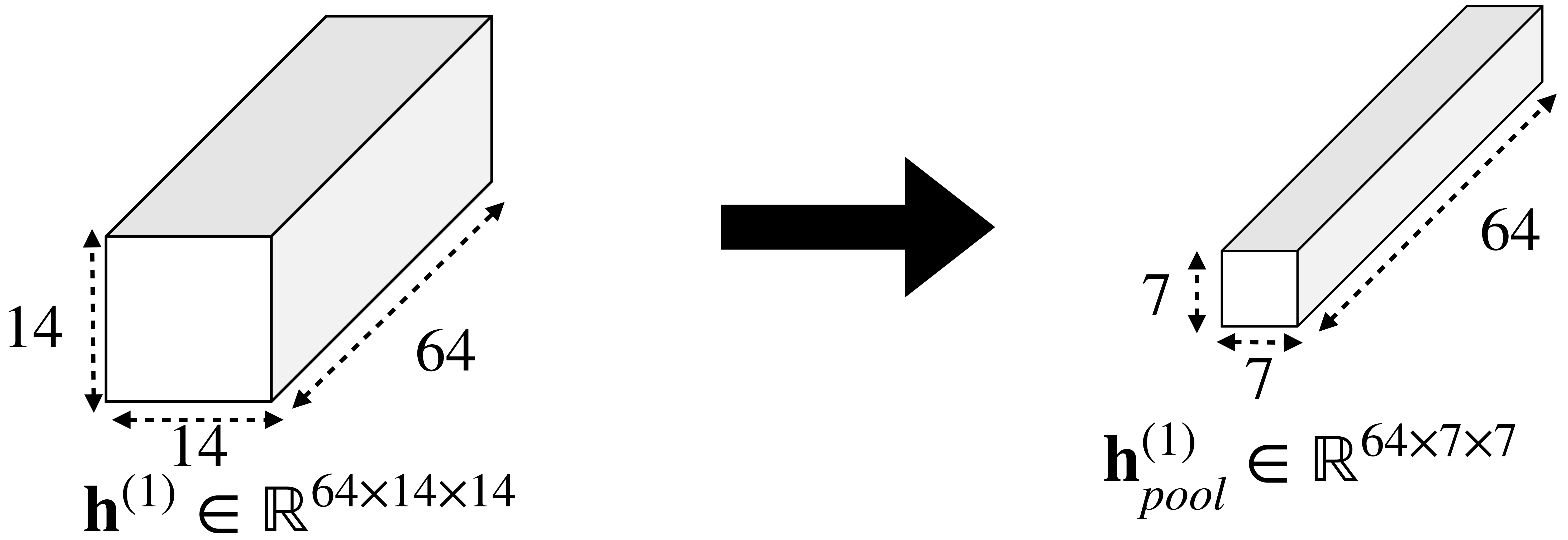


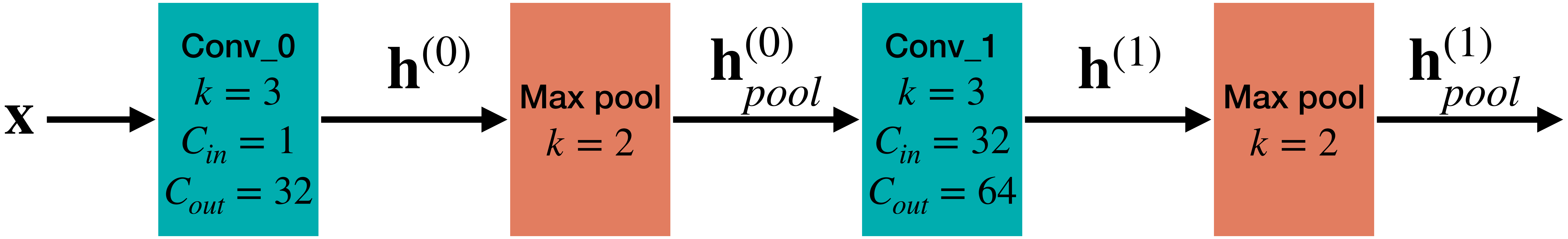
These are the average activations across all channels



# Max pool

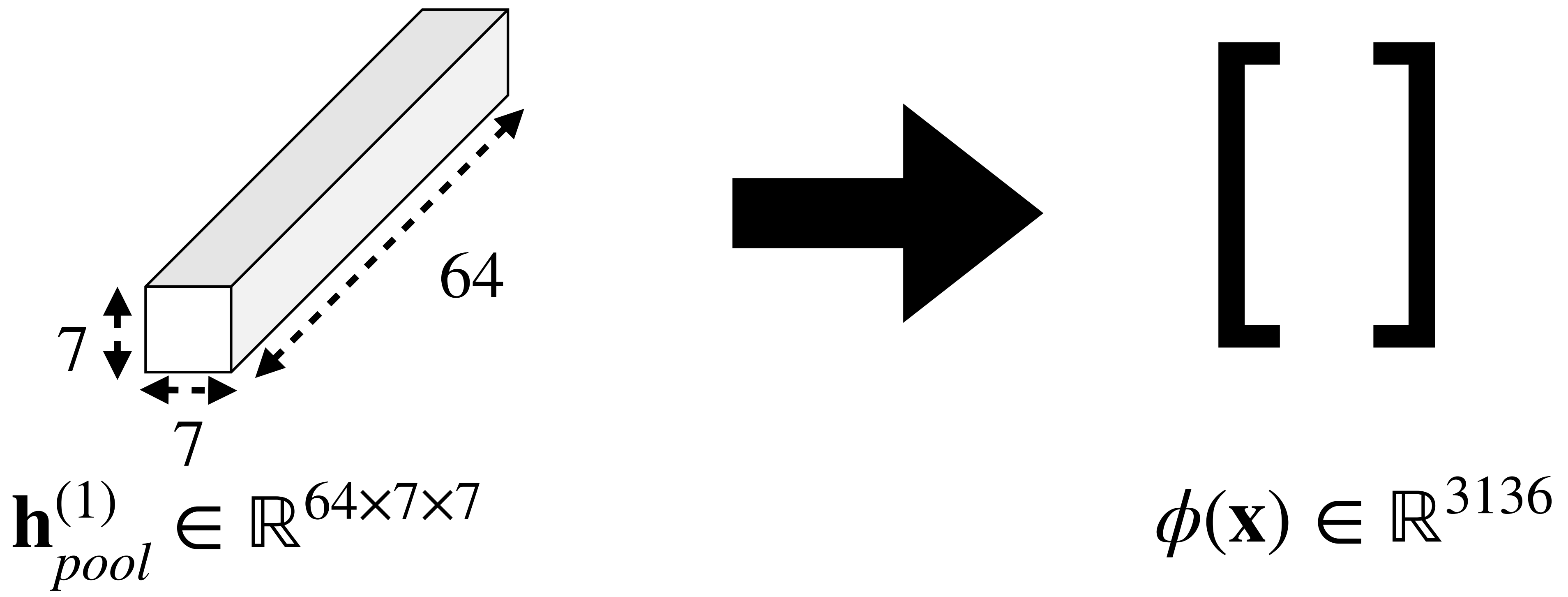
- This reduces spatial resolution again…



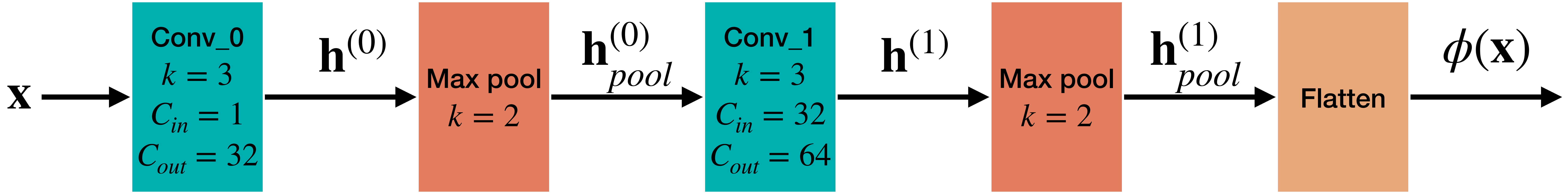



# Flattening

- The last layer of a DNN is a linear layer applied to a feature vector $\phi(\mathbf{x})$
- We are almost there, but our representation is still a tensor
- We simply vectorise, or flatten our representation into a vector




The way this is done doesn't matter as long as it is consistent between data points



# Linear classification

- Finally, we apply a linear transform to our feature vectors

$$f(\mathbf{x}) = \mathbf{W}^{(\mathscr{L}-1)}\phi(\mathbf{x}) + \mathbf{b}^{(\mathscr{L}-1)}$$

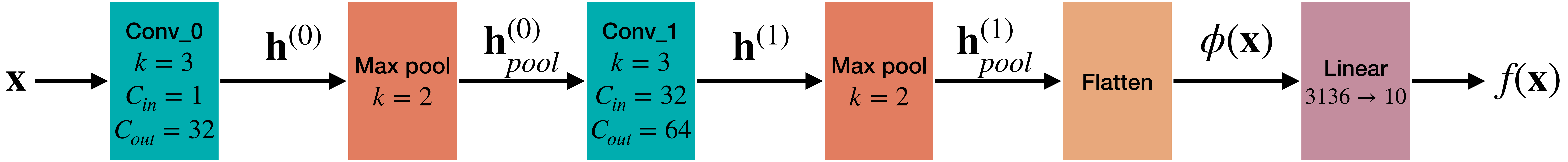
- This gives us a vector that contains the logits for each class

$$\Big[\quad\Big] \;\longrightarrow\; \begin{bmatrix} 21.6 \\ -12.3 \\ 5.3 \\ -7.0 \\ -2.9 \\ -6.9 \\ 4.3 \\ -6.6 \\ 0.9 \\ 0.4 \end{bmatrix}$$

$\phi(\mathbf{x}) \in \mathbb{R}^{3136}$ $\quad f(\mathbf{x}) \in \mathbb{R}^{10}$

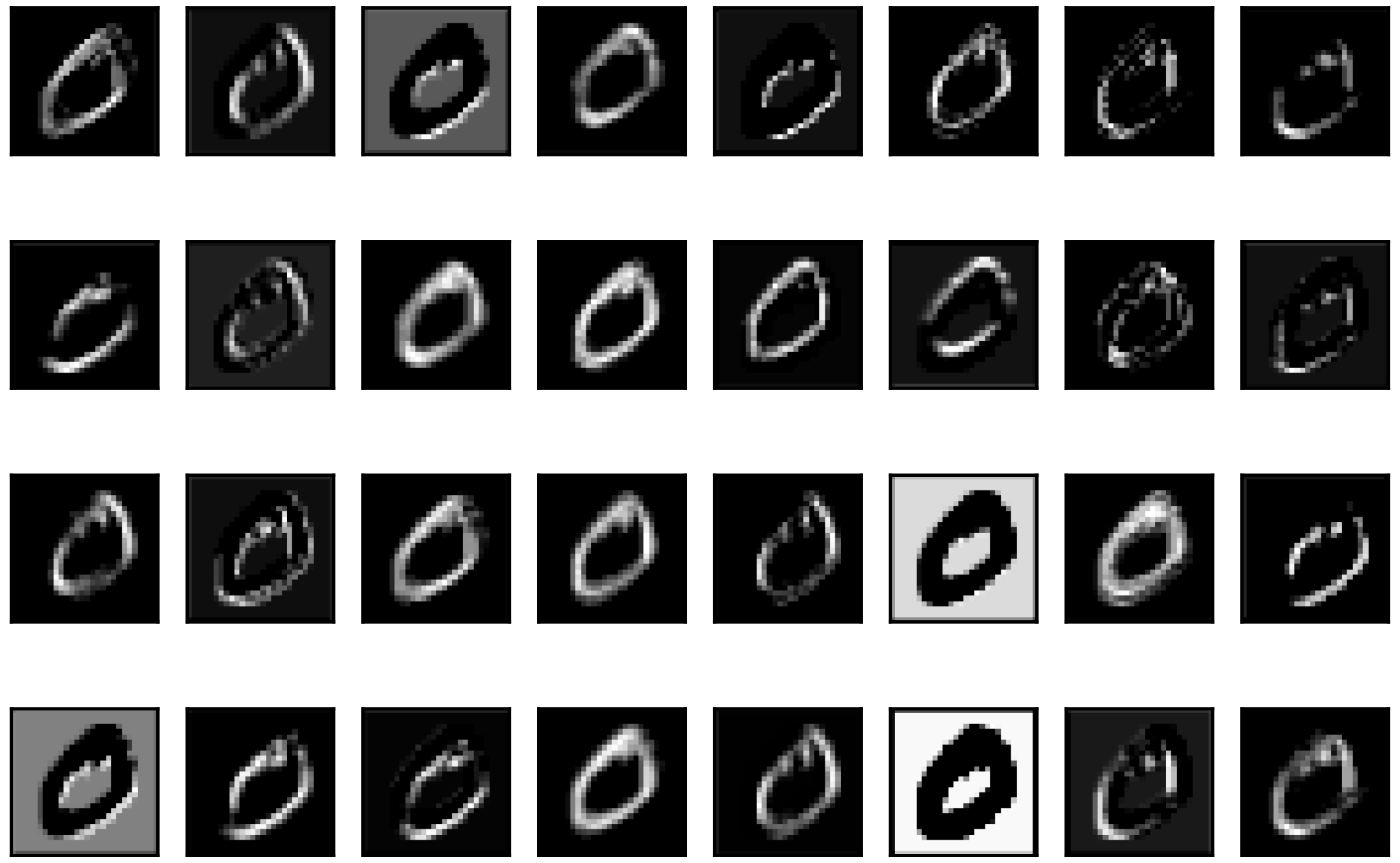
Here the logit for class 0 (21.6) is the largest

It follows that the probability the input is this class is also largest so it is sensible to classify as class 0

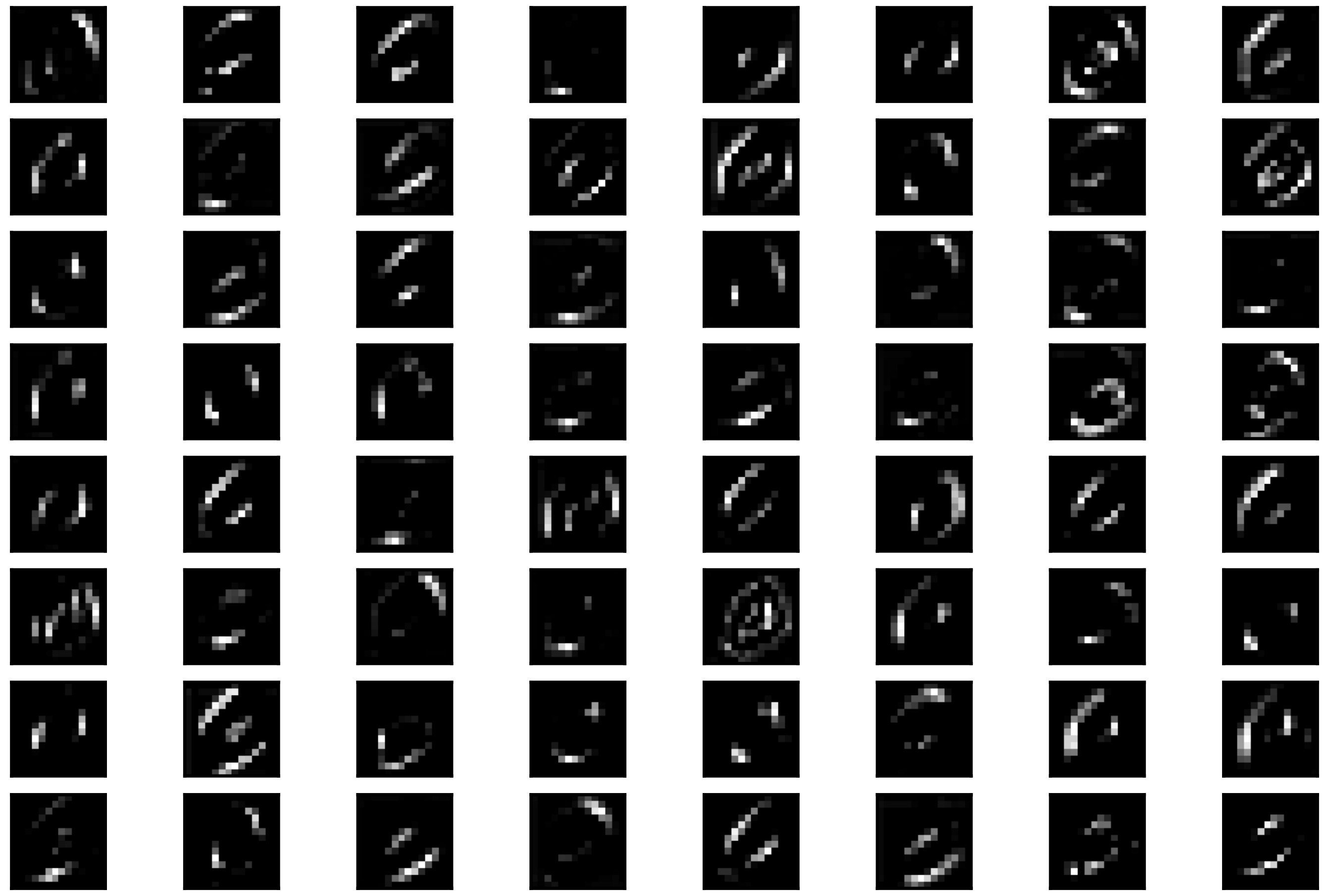


# (Lack of) interpretability

- It's pretty difficult to interpret what exactly is happening
- We can look at all the different channels of $\mathbf{h}^{(0)}$ and $\mathbf{h}^{(1)}$ to try and get an idea
- These models are still very hard to interpret



$\mathbf{h}^{(0)} \in \mathbb{R}^{32\times28\times28}$



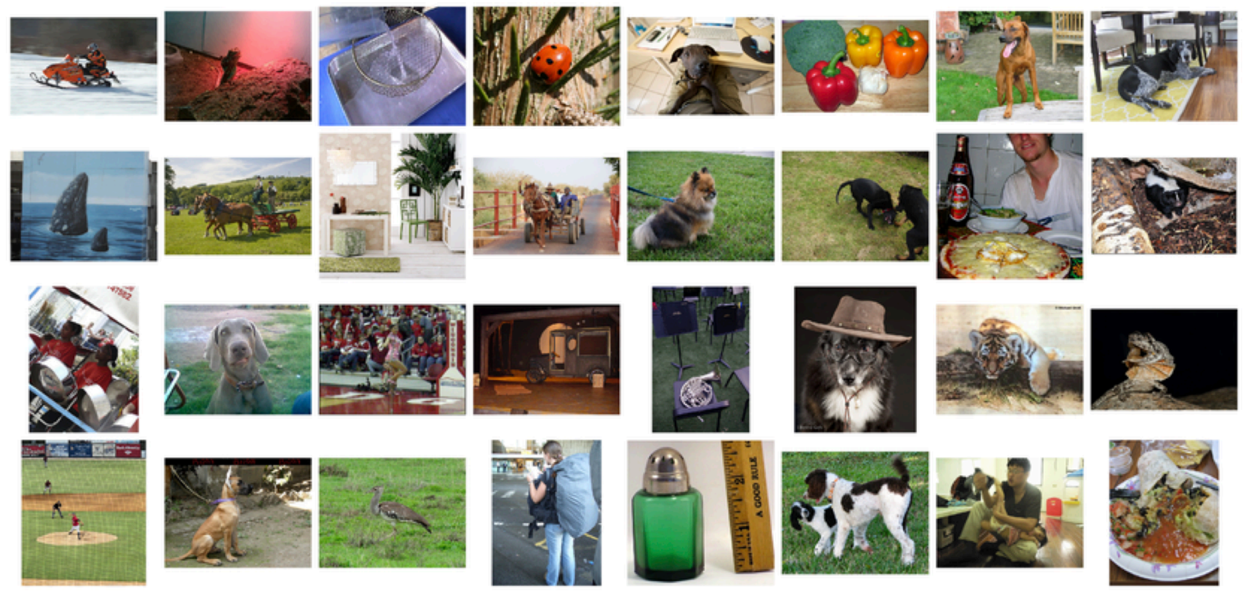
$\mathbf{h}^{(1)} \in \mathbb{R}^{64\times14\times14}$

# GPUs

- Convolutions can be naively implemented in a loop, however loops are slow
- Convolutions are implemented by turning both the input and filters into two big matrices and multiplying them
- Graphics processing units (GPUs) can do matrix-multiplies very fast
- They are essential for training all but the smallest DNNs

# Why bother?

- A benchmark in computer vision is classification performance on ImageNet
- It is a 1000-way classification task with 1 million training images
- For the 2012 ImageNet challenge:
    - The 2nd place model used handcrafted features and got 26.2% top 5-error
    - The 1st place model used a deep ConvNet and got **15.3% top 5-error** (& 36.7% top 1-error)

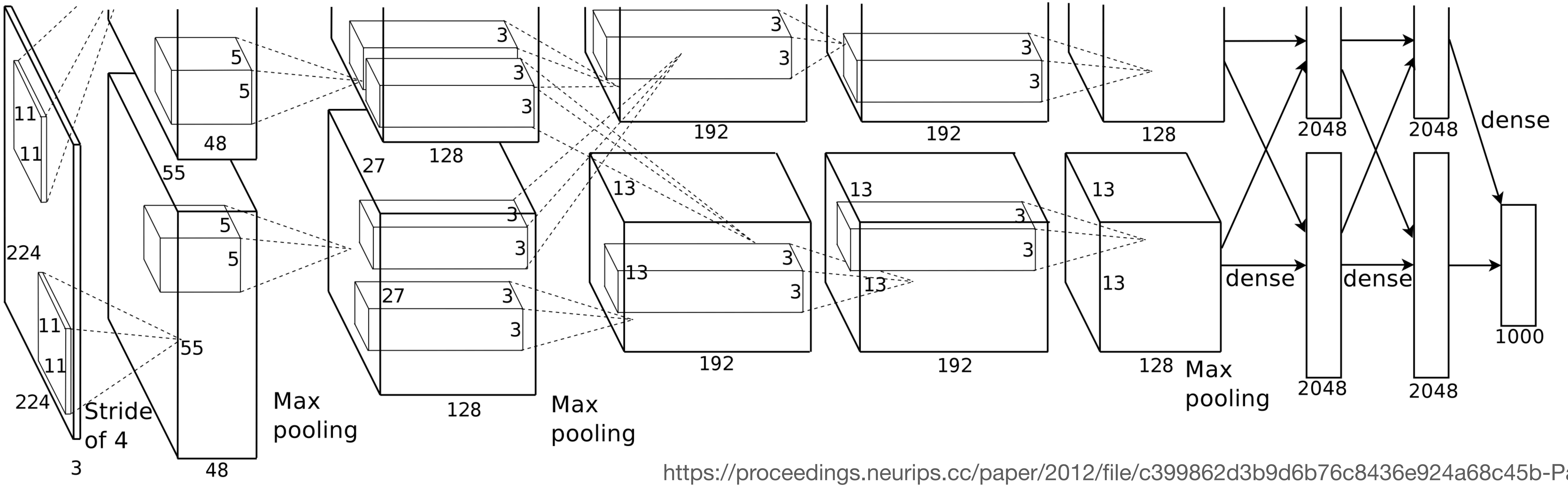


https://arxiv.org/pdf/1409.0575.pdf

# AlexNet (2012)

- The winning entry. It's split into two streams for 2 GPUs because of memory constraints (that no longer exist :) )

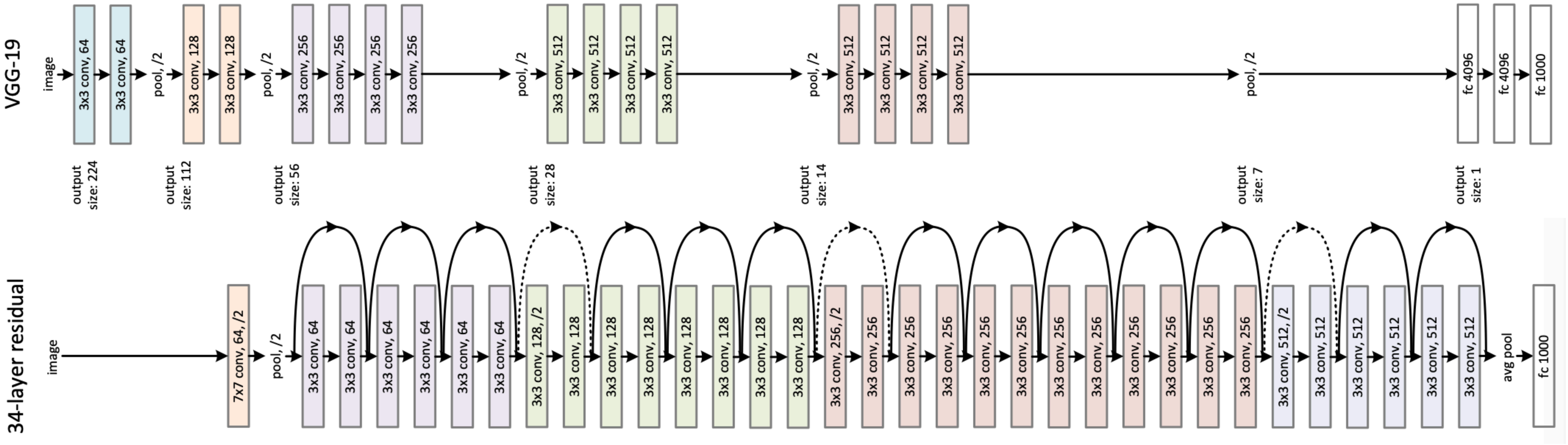
- 5 convolutional layers, 3 max pools (interspersed), and 3 FC layers




https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

# Deeper and deeper on ImageNet

- 2014: A 16 layer (13 conv + 3 FC) VGG net can achieve 8.4% top-5 error
- 2015: ResNets use skip connections to go very deep. A 152 layer ResNet gets a top-5 error of 4.49%

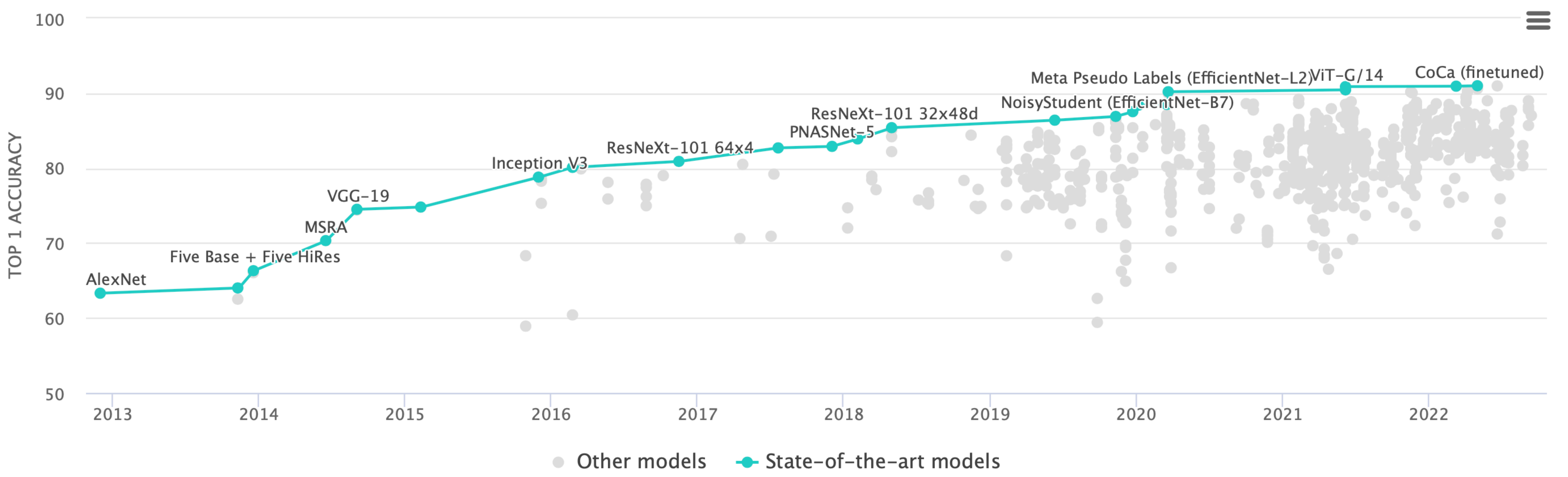



https://arxiv.org/pdf/1512.03385.pdf

# ImageNet top-1 accuracies

Leaderboard Dataset

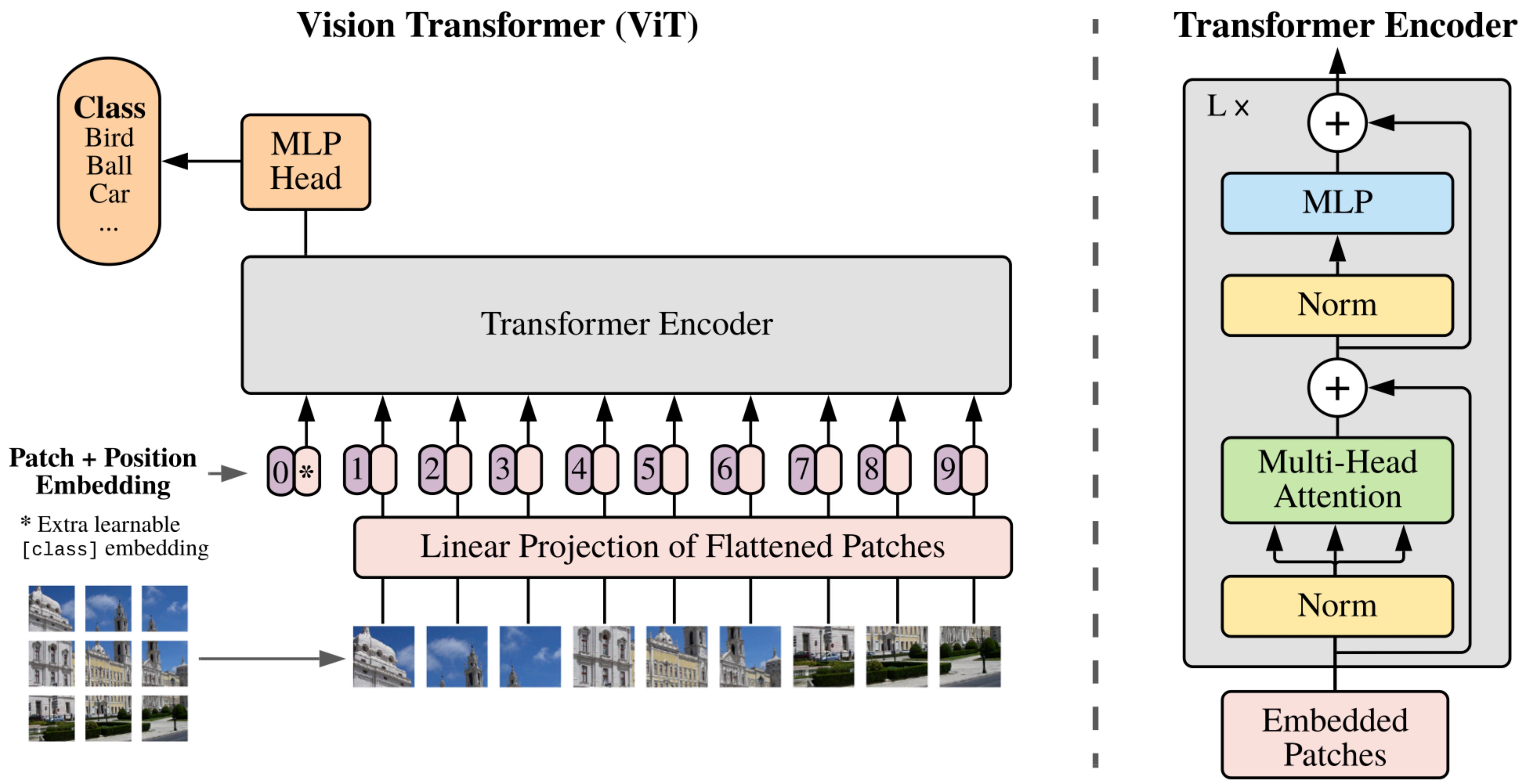
View Top 1 Accuracy by Date for All models




https://paperswithcode.com/sota/image-classification-on-imagenet

# Vision transformers

- ConvNets are no longer state-of-the art in computer vision
- But they are still widespread so learning about them wasn't a waste :)




https://arxiv.org/pdf/2010.11929v2.pdf

# Why not use deep learning for everything?

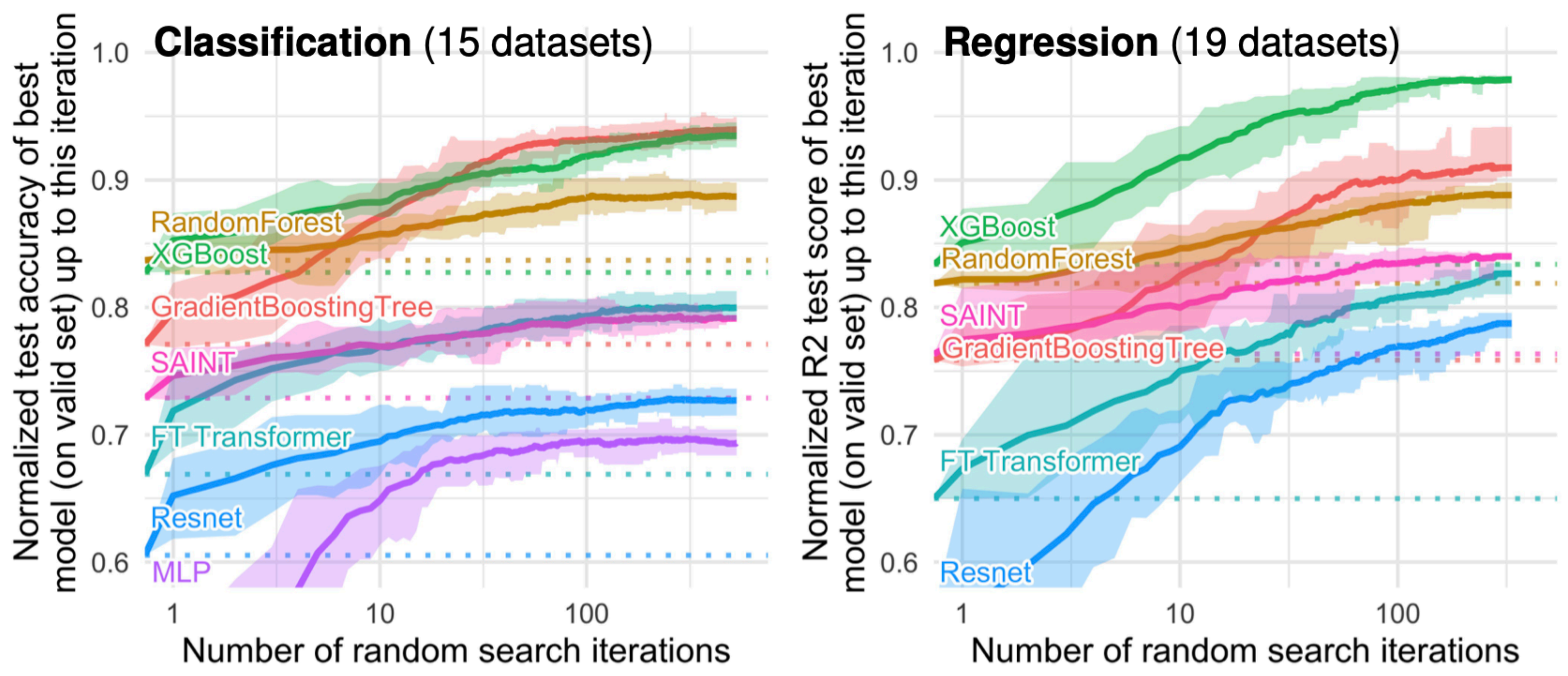
- Deep learning beats other ML approaches for learning on images, text, and audio data

- DNNs are surpassed by decision tree-based models on tabular data

- DNN are near-impossible to interpret, so when this is required a linear model is preferable

- DNNs need lots of data to train from scratch which we may not have!

- We can however use their features for related tasks




https://arxiv.org/pdf/2207.08815.pdf

# Summary

- We have looked at properties of images to justify the need to retain spatial information
- We have seen how 2D convolutions work, and how to performing pooling
- We have looked at a simple ConvNet architecture in detail
- We have had a brief history lesson in the evolution of ConvNets
- We have considered when it is appropriate to use deep learning

# The end (of the lectures)

- You have visualised and analysed data
- You have considered the ethical implications of deploying ML in society
- You have learnt about linear models for classification and regression
- You have learnt about non-parametric and non-linear models
- You have written code to use these models


I hope you enjoyed it!