# Lecture 11: Convolutional neural networks

**Announcements:**

- HW #4 is due **Monday, Feb 19**, uploaded to Gradescope. To submit your Jupyter Notebook, print the notebook to a pdf with your solutions and plots filled in. You must also submit your .py files as pdfs.

- The midterm exam is during class (2-3:50pm) on Wednesday, Feb 21, 2024.

  - You are allowed 4 cheat sheets (each an 8.5 x 11 inch paper). You can fill out both sides (8 sides total). You can put whatever you want on these cheat sheets.
  - The midterm will cover material up to and including ~~this Wednesday~~ today's lecture (Feb 14).
  - Past exams are uploaded to Bruin Learn (under "Modules" —> "past exams").
  - You may bring a calculator to the exam.
  - You may do the exam in pen or pencil.

- Midterm exam review session: Thursday, Feb 15, 6-9pm at WG Young CS50.

- No class or OH Monday (for Jonathan; TAs will still hold Monday OH over Zoom.)
- My OH on Wednesday, Feb 21, are canceled. (We will be scanning exams)
- CAE, be sure to schedule your accomodations!

# Adam

## Adam (cont.)

Initialize $\mathbf{v} = 0$ as the "first moment", and $\mathbf{a} = 0$ as the "second moment." Set $\beta_1$ and $\beta_2$ to be between 0 and 1. (Suggested defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.) Initialize $\nu$ to be sufficiently small. Initialize $t = 0$. Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Time update: $t \leftarrow t + 1$
- First moment update (momentum-like):

$$\mathbf{v} \leftarrow \beta_1 \mathbf{v} + (1 - \beta_1)\mathbf{g}$$

$\mathbb{E}[g]$

- Second moment update (gradient normalization):

$$\mathbf{a} \leftarrow \beta_2 \mathbf{a} + (1 - \beta_2)\mathbf{g} \odot \mathbf{g}$$

$\mathbb{E}[g^2]$

- Bias correction in moments:

$$\tilde{\mathbf{v}} = \frac{1}{1 - \beta_1^t}\mathbf{v}$$

$$\tilde{\mathbf{a}} = \frac{1}{1 - \beta_2^t}\mathbf{a}$$

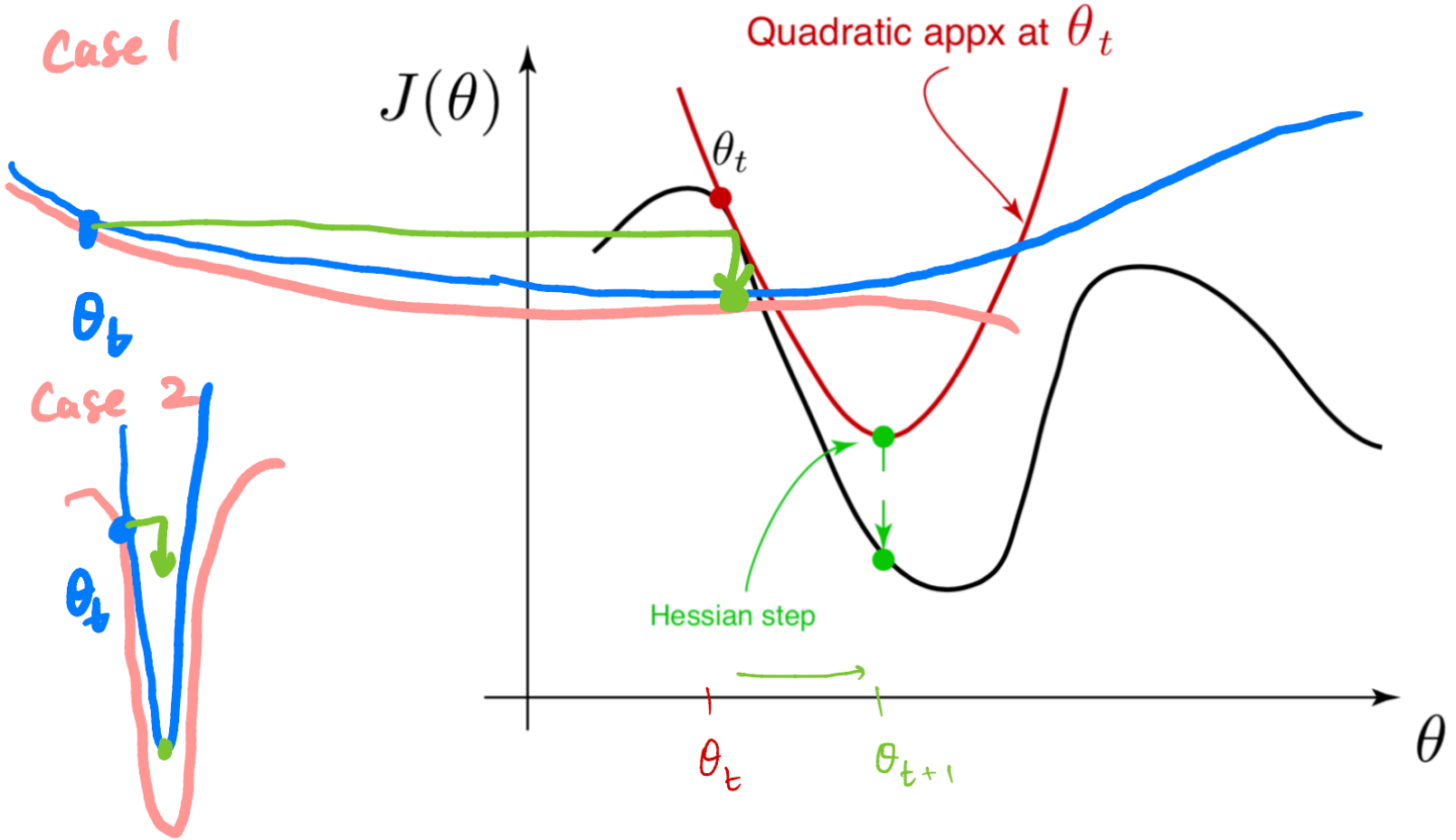
- Gradient step:

$$\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\tilde{\mathbf{a}}} + \nu} \odot \tilde{\mathbf{v}}$$

A resource on bias correction: https://www.coursera.org/lecture/deep-neural-network/bias-correction-in-exponentially-weighted-averages-XjuhD

## First order vs second order methods (cont)

It is possible to also use the *curvature* of the cost function to know how to take steps. These are called second-order methods, because they use the second derivative (or Hessian) to assess the curvature and thus take appropriate sized steps in each dimension. See following picture for intuition:



## Newton's method

Newton's method, by using the curvature information in the Hessian, does not require a learning rate.

Until stopping criterion is met:

- Compute gradient: $\mathbf{g}$
- Compute Hessian: $\mathbf{H}$
- Gradient step:

$$\theta \leftarrow \theta - \mathbf{H}^{-1}\mathbf{g}$$

1. Memory: storing H is expensive

$n = 10^6$ :

grad: 3.8 MBytes

Hessian: 3.6 TB

2. Inverting the Hessian: $O(n^3)$

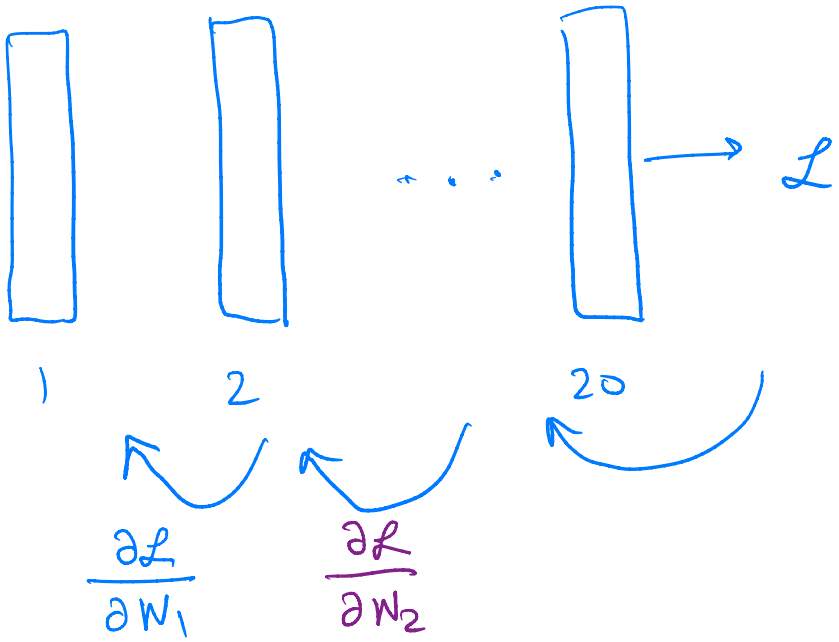
3. Hessian typically requires very large batch sizes

# Challenges in gradient descent

- **Exploding gradients.**
  Sometimes the cost function can have "cliffs" whereby small changes in the parameters can drastically change the cost function. (This usually happens if parameters are repeatedly multiplied together, as in recurrent neural networks.) Because the gradient at a cliff is large, an update can result in going to a completely different parameter space. This can be ameliorated via gradient clipping, which upper bounds the maximum gradient norm.

$$\|g\| > \text{clip}$$

$$g \leftarrow g \cdot \frac{\text{clip}}{\|g\|}$$



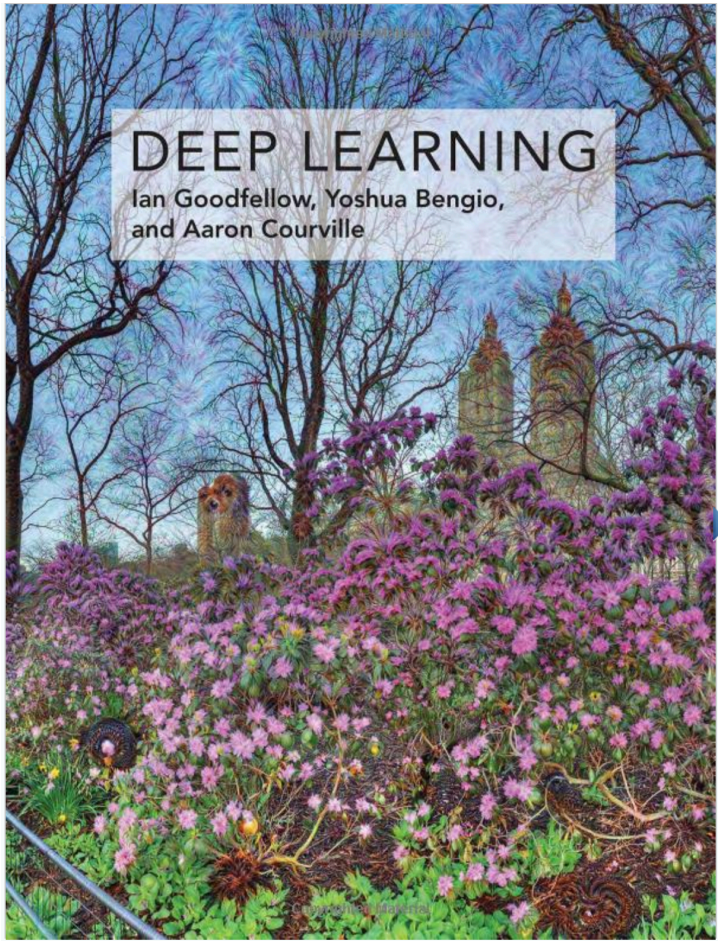
# Challenges in gradient descent

- **Vanishing gradients.**
  Like in exploding gradients, repeated multiplication of a matrix $\mathbf{W}$ can cause vanishing gradients. Say that each time step can be thought of as a layer of a feedforward network where each layer has connectivity $\mathbf{W}$ to the next layer. By layer $t$, there have been $\mathbf{W}^t$ multiplications. If $\mathbf{W} = \mathbf{U}\Lambda\mathbf{U}^{-1}$ is its eigendecomposition, then $\mathbf{W}^t = \mathbf{U}\Lambda^t\mathbf{U}^{-1}$, and hence the gradient along eigenvector $\mathbf{u}_i$ is shrunk (or grown) by the factor $\lambda_i^t$. Architectural decisions, as well as appropriate regularization, can deal with vanishing gradients.

# Convolutional neural networks

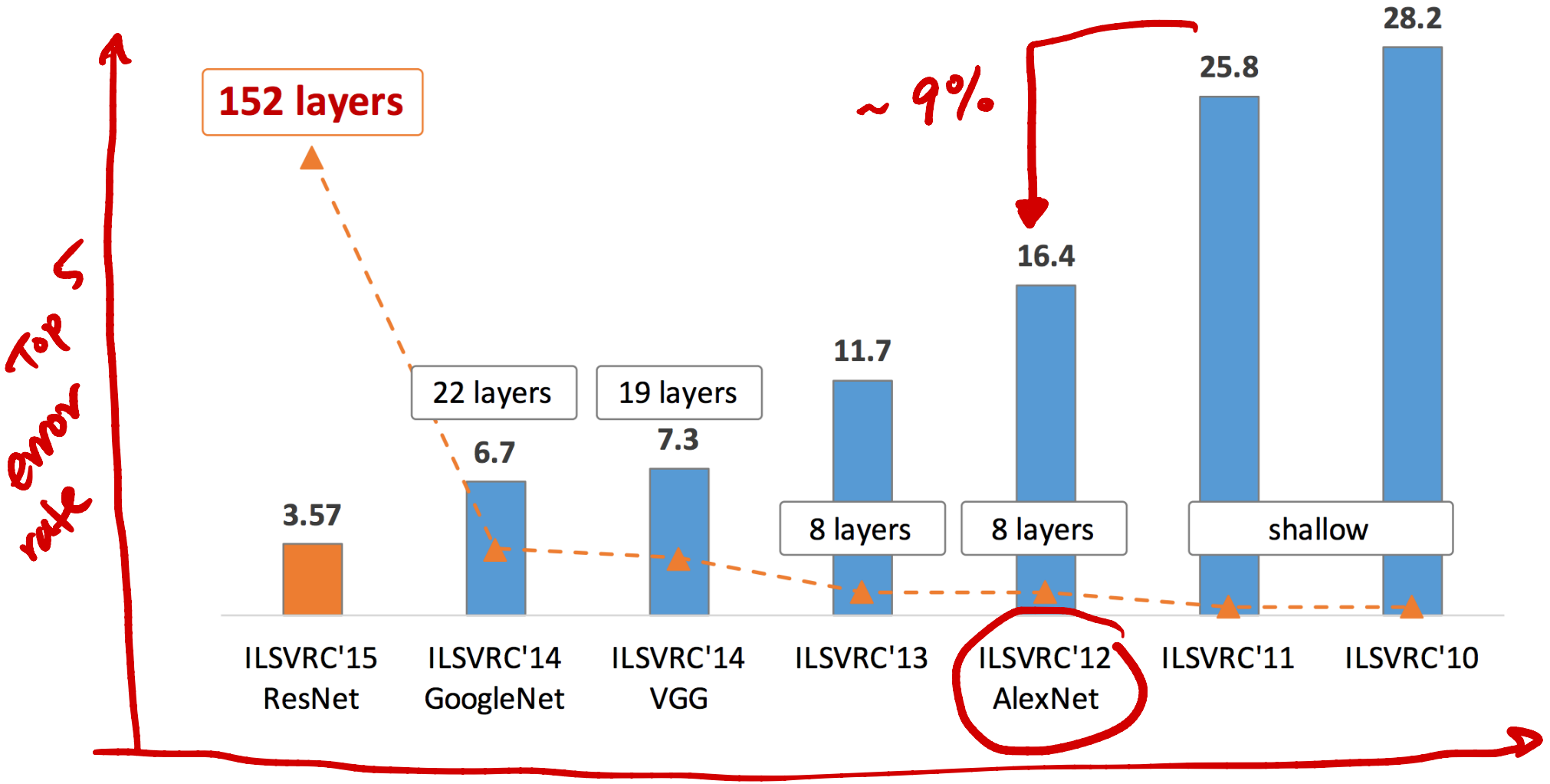
Reading:

Deep Learning, Chapter 9



# Background: convolutional neural networks

The CNN played a large role in this last "revival" of neural networks (i.e., the past 5 years).

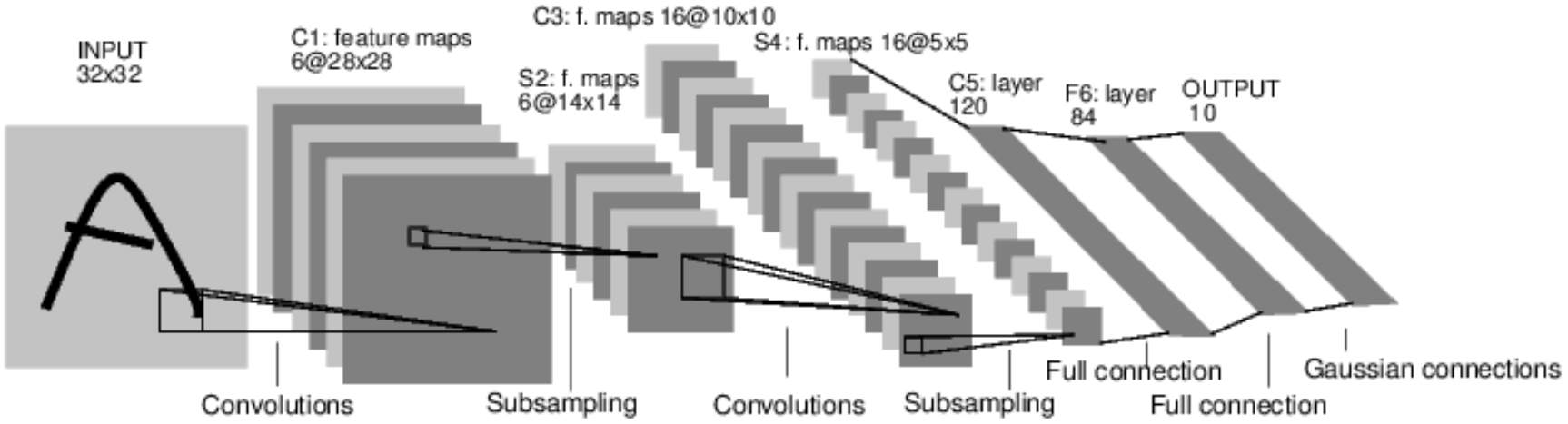


# Background: convolutional neural networks

Fukushima Neocognitron 1980's

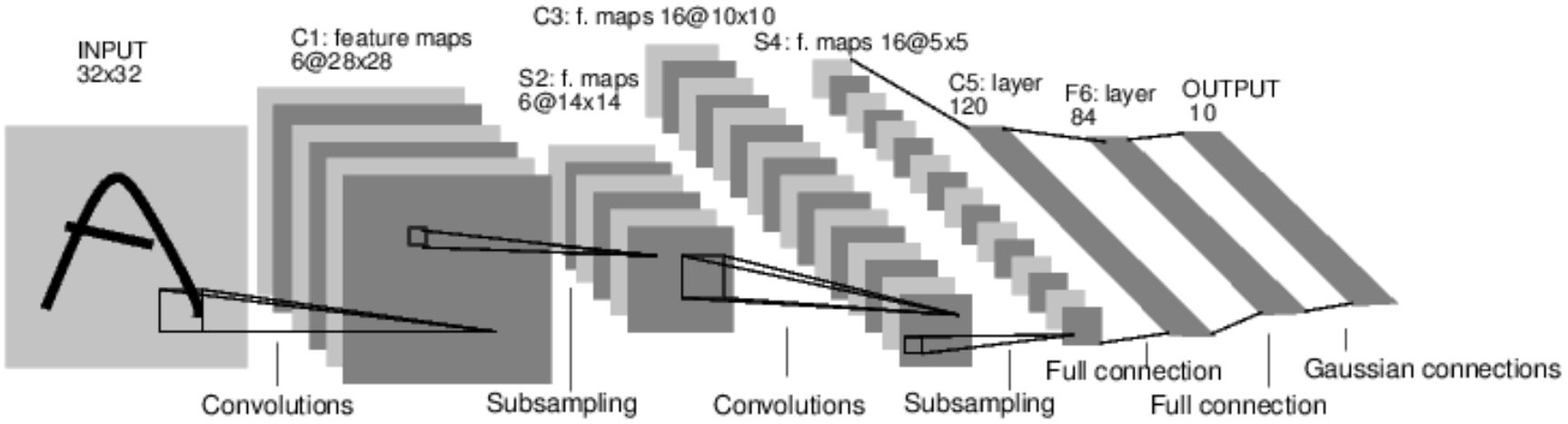
CNNs have been around since the 1990's. In 1998, Yann LeCun introduced LeNet, which is the modern convolutional neural network.

LeNet



# Background: convolutional neural networks

How do we arrive at this architecture? What are the principles that lead us to this?



# Biological inspiration for the CNN

## Biological inspiration

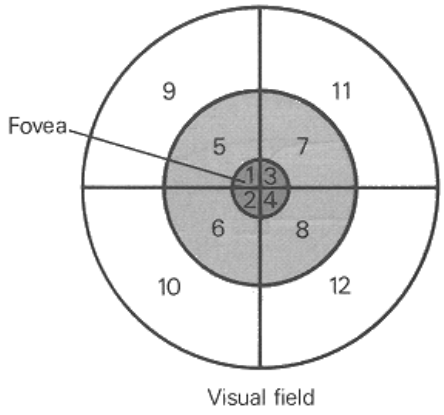
Principles of the convolutional neural network are inspired from neuroscience. Seminal work by Hubel and Wiesel in cat visual cortex in the 1960's (Nobel prize awarded in the 1980's) found that V1 neurons were tuned to the movement of oriented bars. Hubel and Wiesel also defined "simple" and "complex" cells, the computational properties of which were later well-described by linear models and rectifiers (e.g., Movshon and colleagues, 1978). Three key principles of neural coding in V1 are incorporated in convolutional neural networks (CNNs):

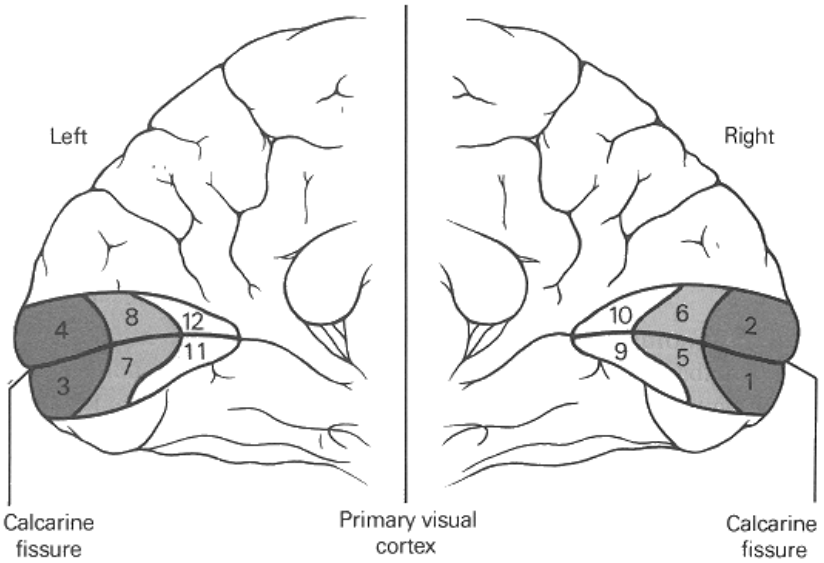
- V1 has a retinotopic map.
- V1 is composed of simple cells that can be adequately described by a linear model in a spatially localzed receptive field.
- V1 is composed of complex cells that respond to similar features as simple cells, but importantly are largely invariant to the position of the feature.

If interested, you can YouTube some videos of their experiments. Warning: they do show cats in an experimental apparatus.

# Biological inspiration for the CNN

- Retinotopic map: CNNs are spatially organized, so that nearby cells act on nearby parts of the input image.



# Biological inspiration for the CNN

relu()

affine transformation
convolution

- Simple cells: CNNs use spatially localized linear filters, which are followed by thresholding.
- Complex cells: CNNs use pooling units to incorporate invariance to shifts of the position of the feature.

"pool"

# Biological inspiration for the CNN

Do CNN's compute like the visual system?

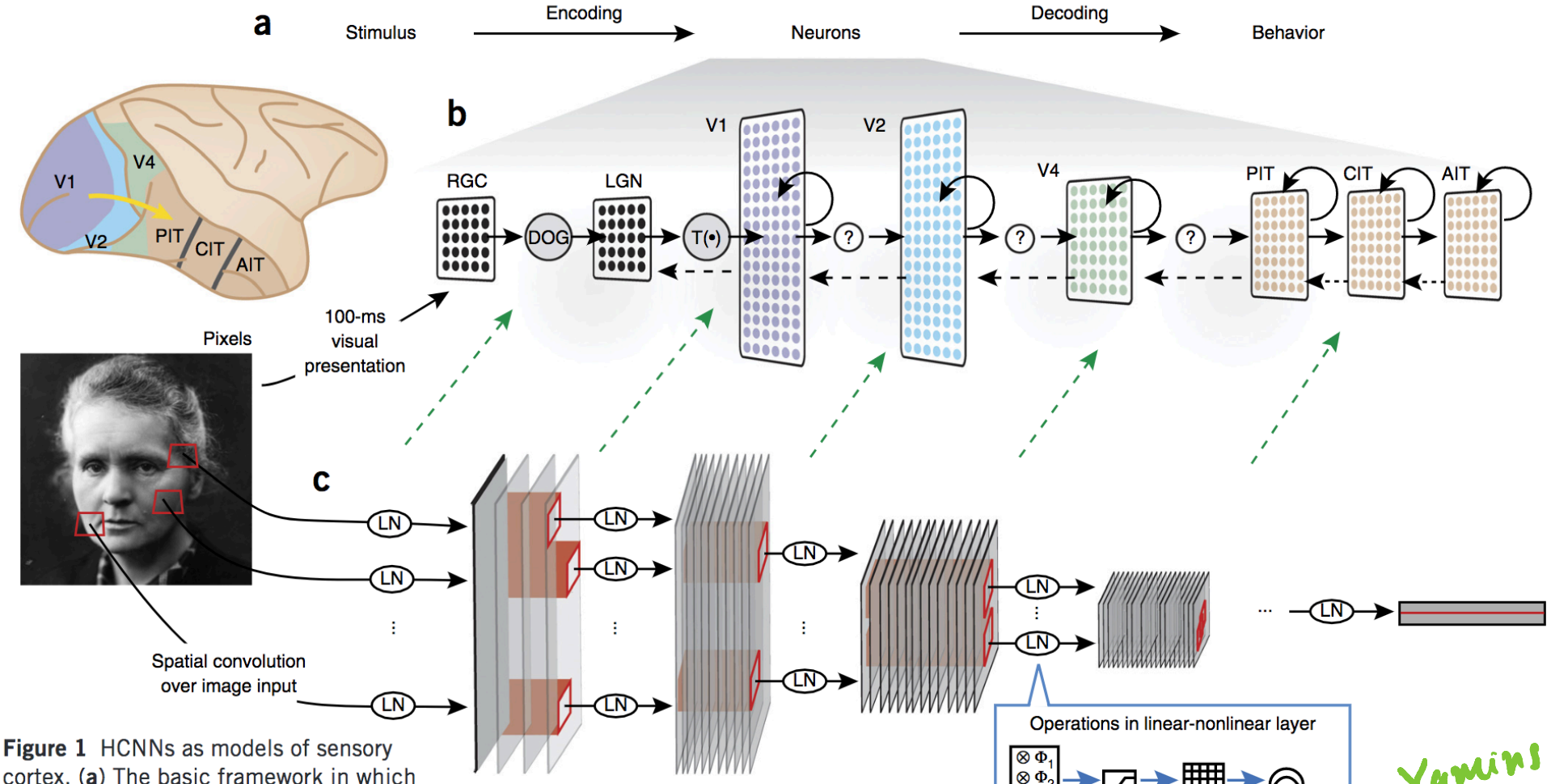
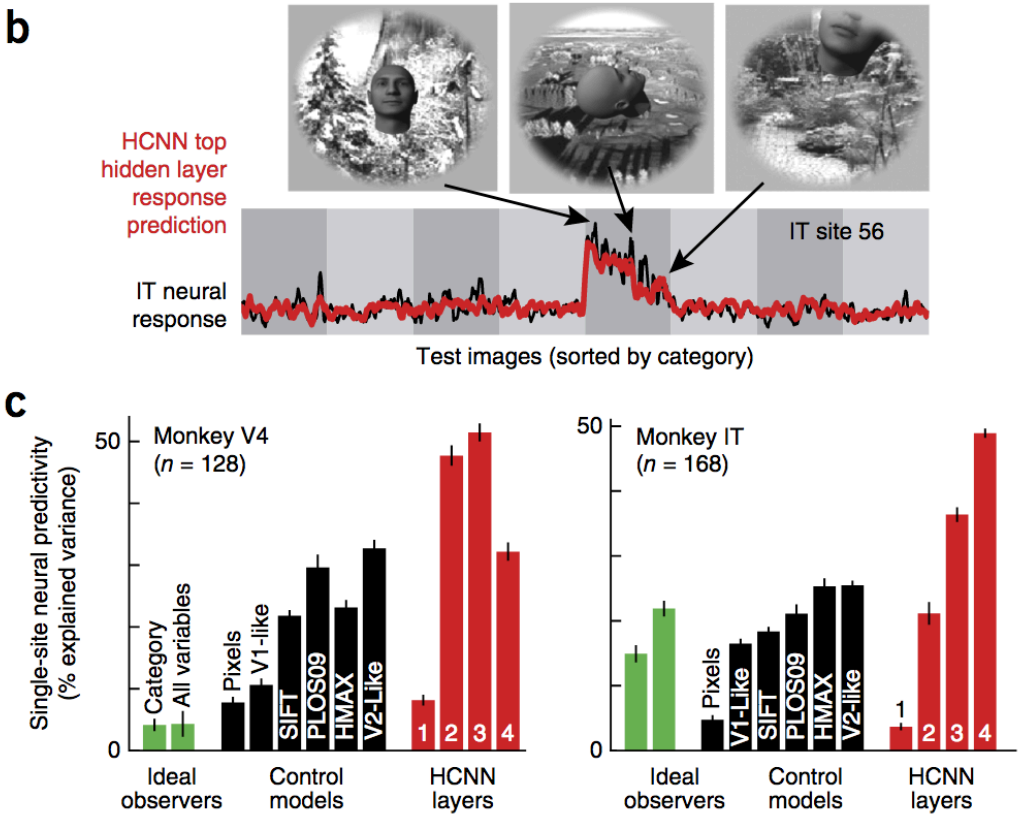



**Figure 1** HCNNs as models of sensory cortex. (**a**) The basic framework in which sensory cortex is studied is one of encoding—the process by which stimuli are transformed into patterns of neural activity—and decoding, the process by which neural activity generates behavior. HCNNs have been used to make models of the encoding step; that is, they describe

Yamins DiCarlo 2014

# Biological inspiration for the CNN

Do CNN's compute like the visual system?



# Biological inspiration for the CNN

Do CNN's compute like the visual system?

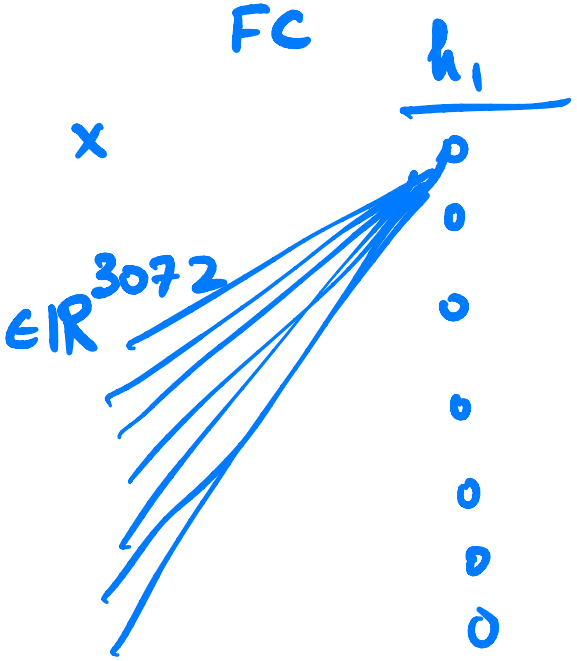
## Limitations of biological analogies

There are several limitations in these analogies. For example,

- CNNs have no feedback connections; neural populations are recurrently connected.
- The brain foveates a small region of the visual space, using saccades to focus on different areas.
- The output of the brain is obviously more than just image category classification.
- Pooling done by complex cells is an approximate way to incorporate invariance.

## Motivation for convolutional neural networks

- For images, fully connected networks require many parameters. In CIFAR-10, the input size is $32 \times 32 \times 3 = 3072$, requiring $3072$ weights for each neuron. For a more normal sized image that is $200 \times 200$, each neuron in the *first layer* would require $120000$ parameters. This quickly gets out of hand. This network, with a huge number of parameters, would not only take long to train, but may also be prone to overfitting.




3072 · # of artificial neurons in layer 1.

## The convolution operation

The convolution of two functions, $f(t)$ and $g(t)$, is given by:

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

In discrete time, this is given by:

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

# Correlation and convolution

Note, however, that in general CNNs don't use *convolution*, but instead use *cross-correlation*. Colloquially, instead of "flip-and-drag," CNNs just "drag." For real-valued functions, cross-correlation is defined by:

$$(f \star g)(n) = \sum_{m=-\infty}^{\infty} f(m)g(n+m)$$

We'll follow the field's convention and call this operation convolution.

## Convolution in 2D

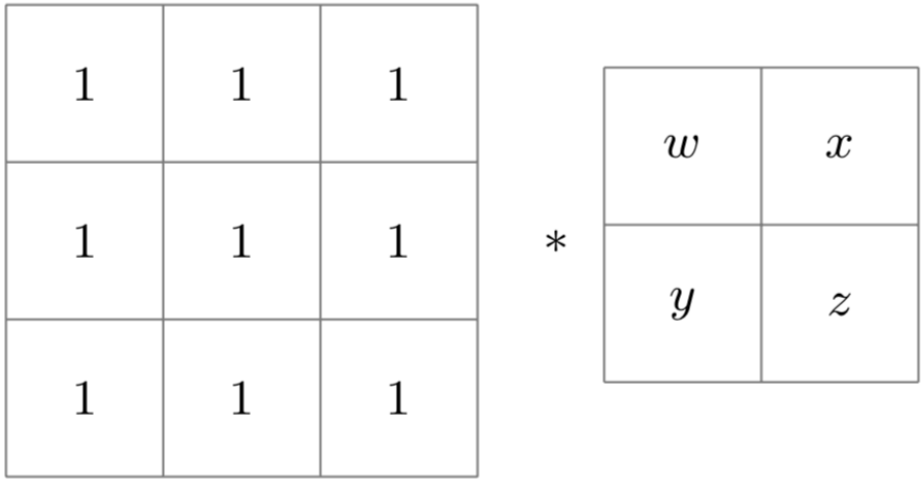
The 2D convolution (formally: cross-correlation) is given by:

$$(f * g)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n) g(i+m, j+n)$$

B A

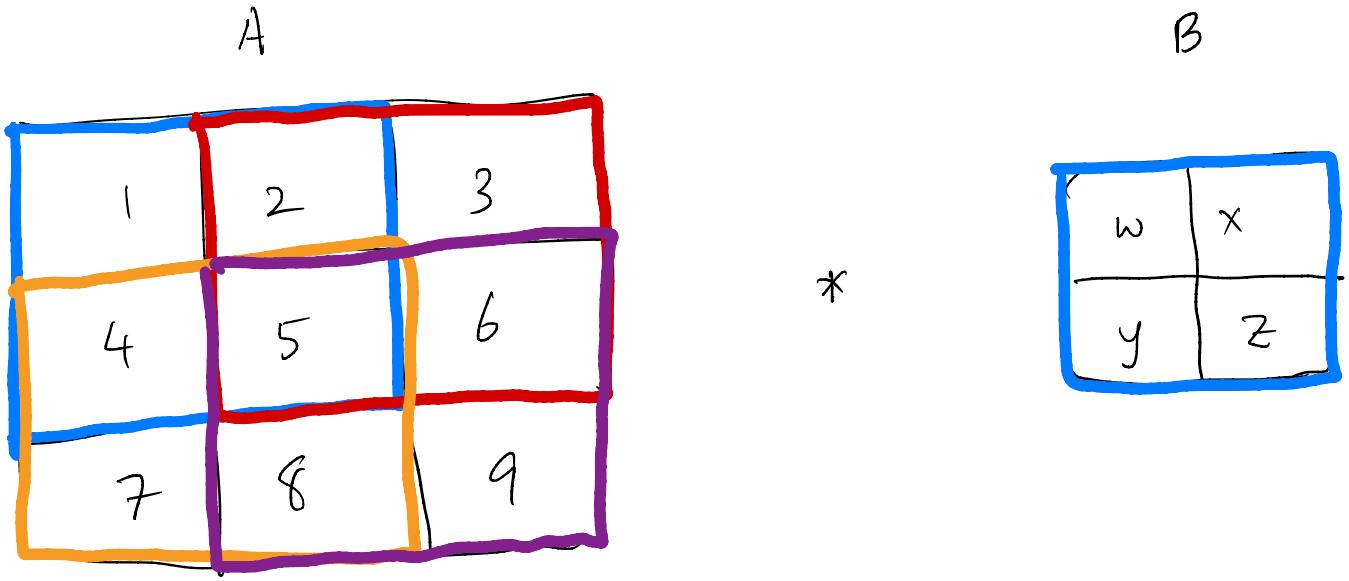
This generalizes to higher dimensions as well. Note also: these "convolutions" are not commutative.

# Convolution in 2D example

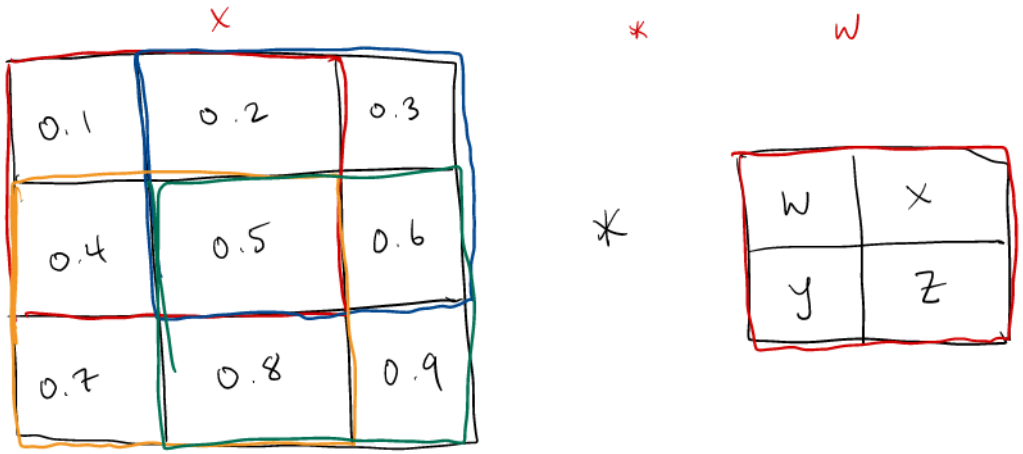


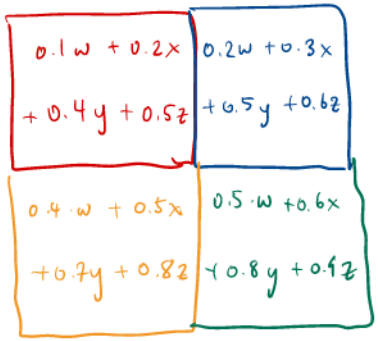

***** All convolutions in this class will be VALID convolutions (the filter and inputs must be totally overlapping). *****

# Convolution in 2D example



# Convolution in 2D

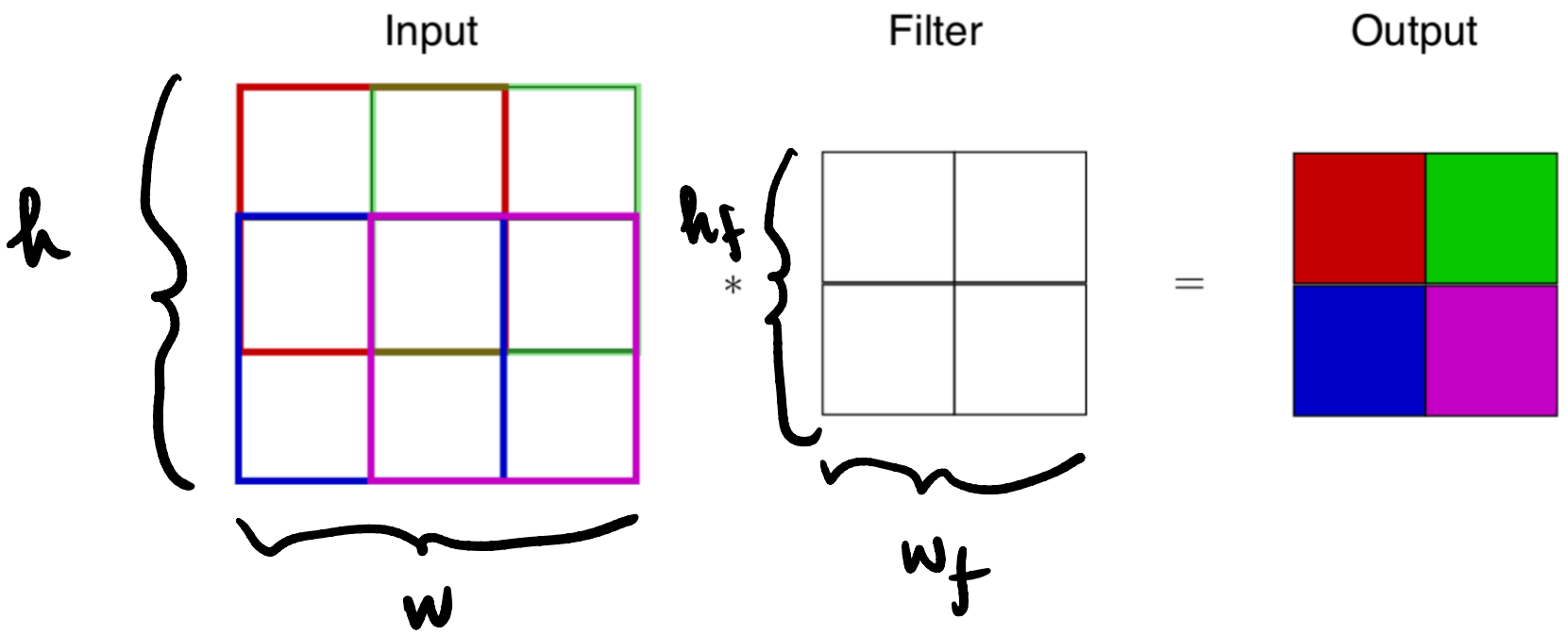



"VALID CONVOLUTION"

# Convolution in 2D

***** All convolutions in this class will be VALID convolutions (the filter and inputs must be totally overlapping). *****



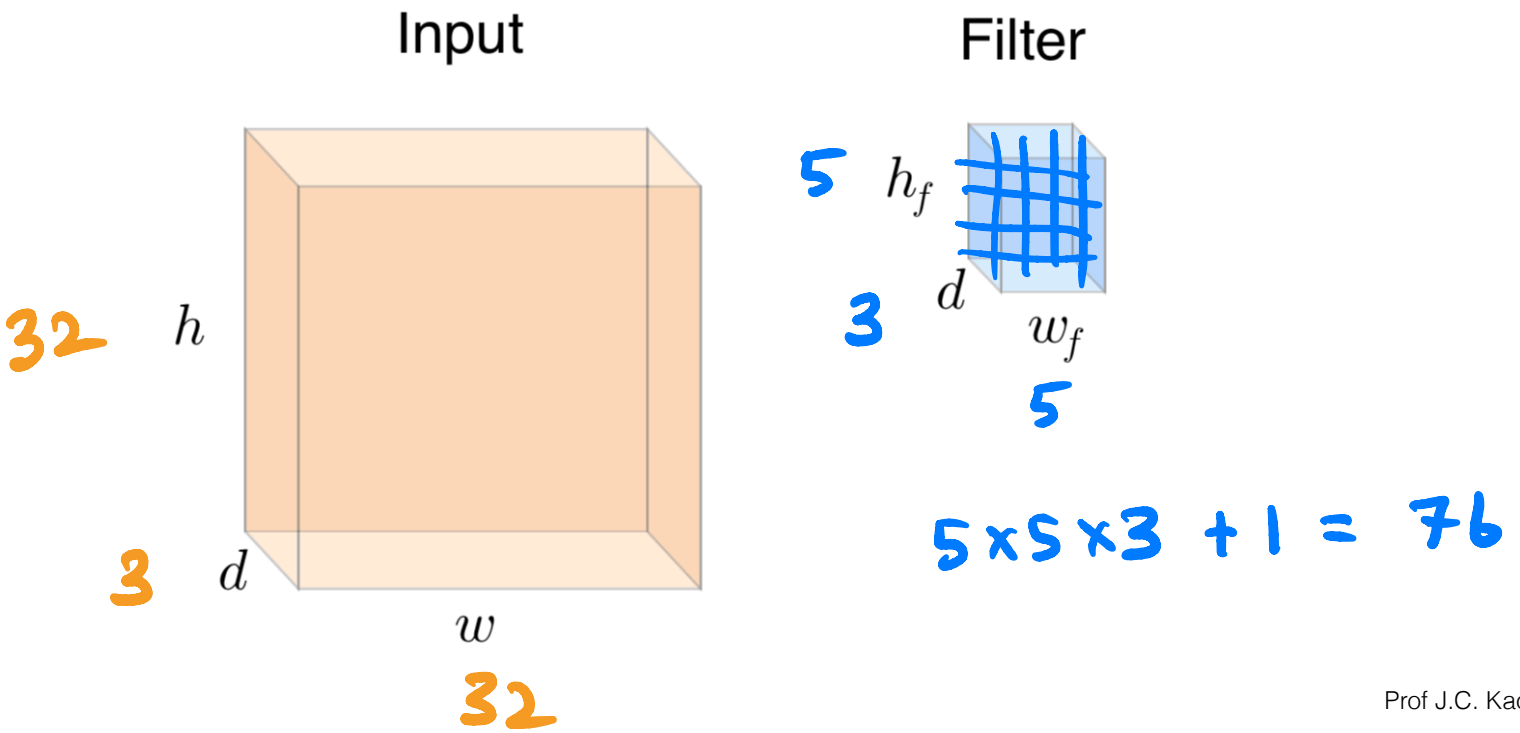

$$(w - w_f + 1, h - h_f + 1)$$

# The convolutional layer of a CNN

## Convolutional layer

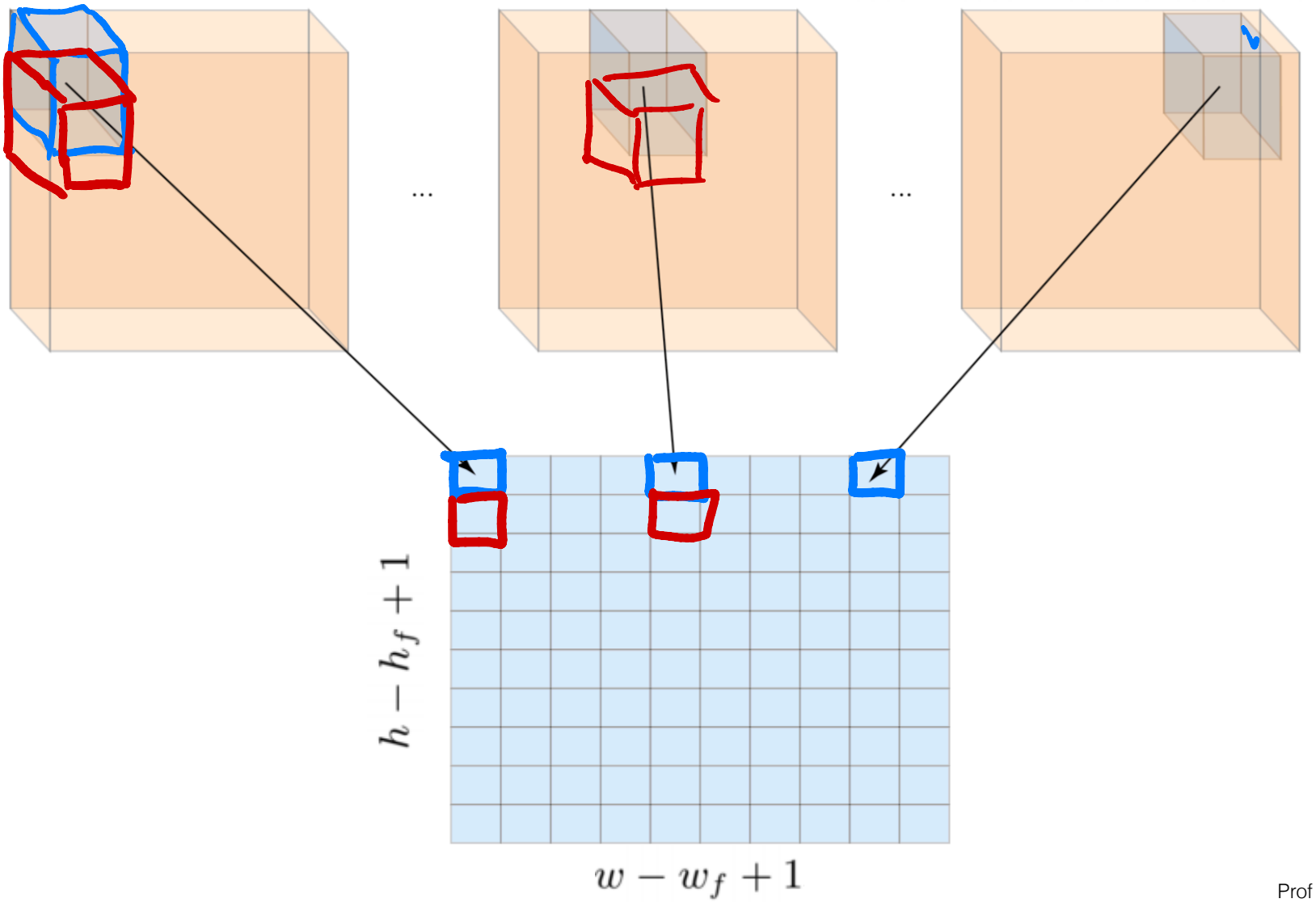
This convolution operation (typically in 3D, since images often come with a width and height as well as *depth* for the R, G, B channels) defines the "convolutional layer" of a CNN. The convolutional layer defines a collection of filters (or activation maps), each with the same dimension as the input.

- Say the input was of dimensionality $(w, h, d)$.
- Say the filter dimensionality is $(w_f, h_f, d)$. So that the filter operates on a small region of the input, typically $w_f < w$.
- The depths being equal means that the output of this convolution operation is 2D.



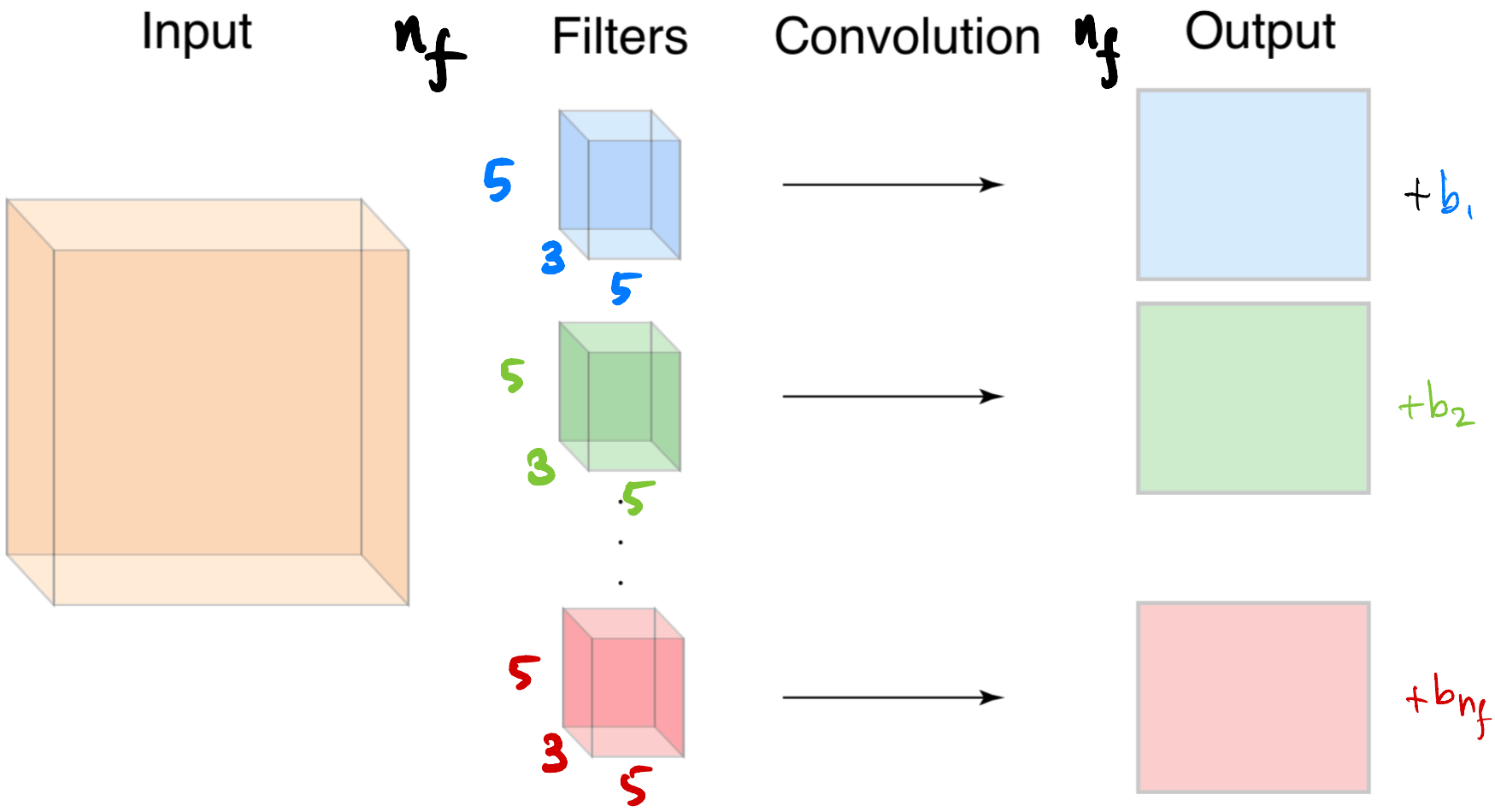
## Convolutional layer (cont.)

After performing the convolution, the output is $(w - w_f + 1, h - h_f + 1)$



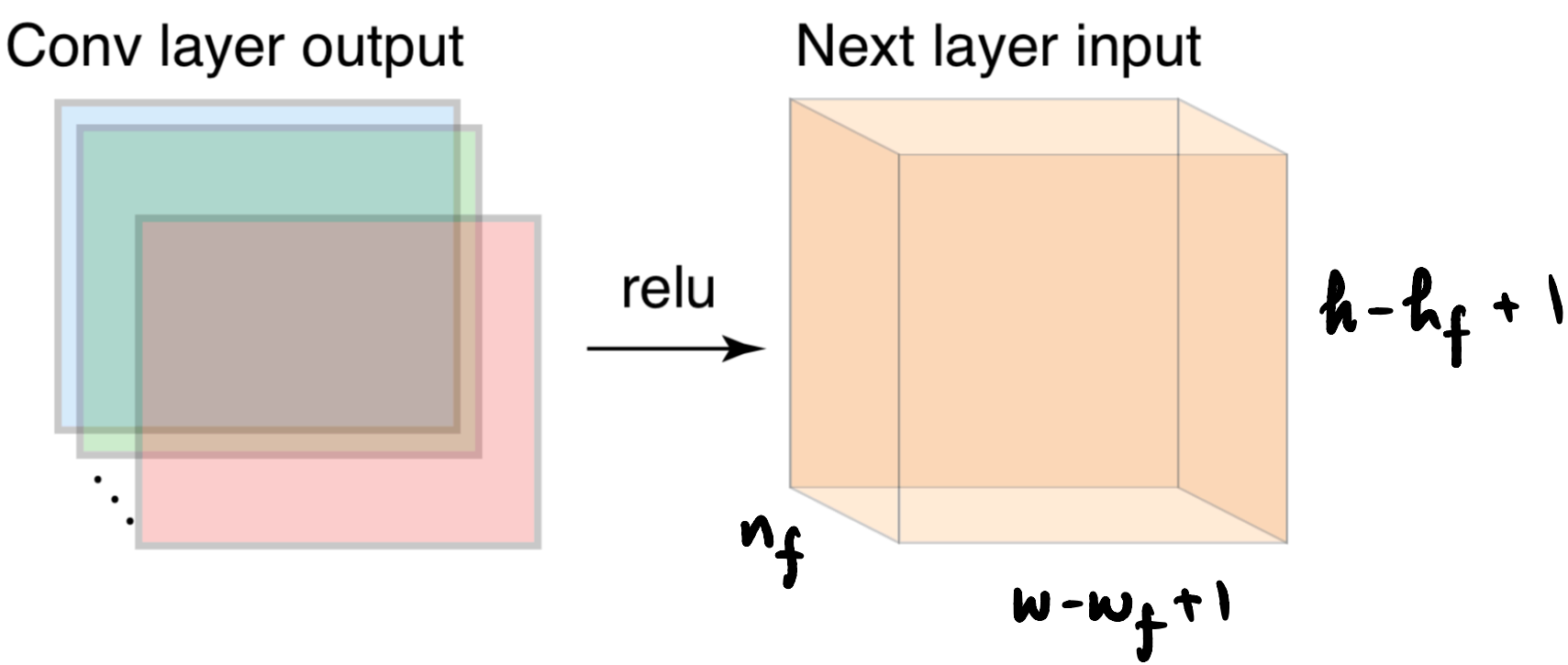
## Convolutional layer (cont.)

Now, we don't have just one filter in a convolutional layer, but multiple filters.
We call each output (matrix) a slice.



## Convolutional layer (cont.)

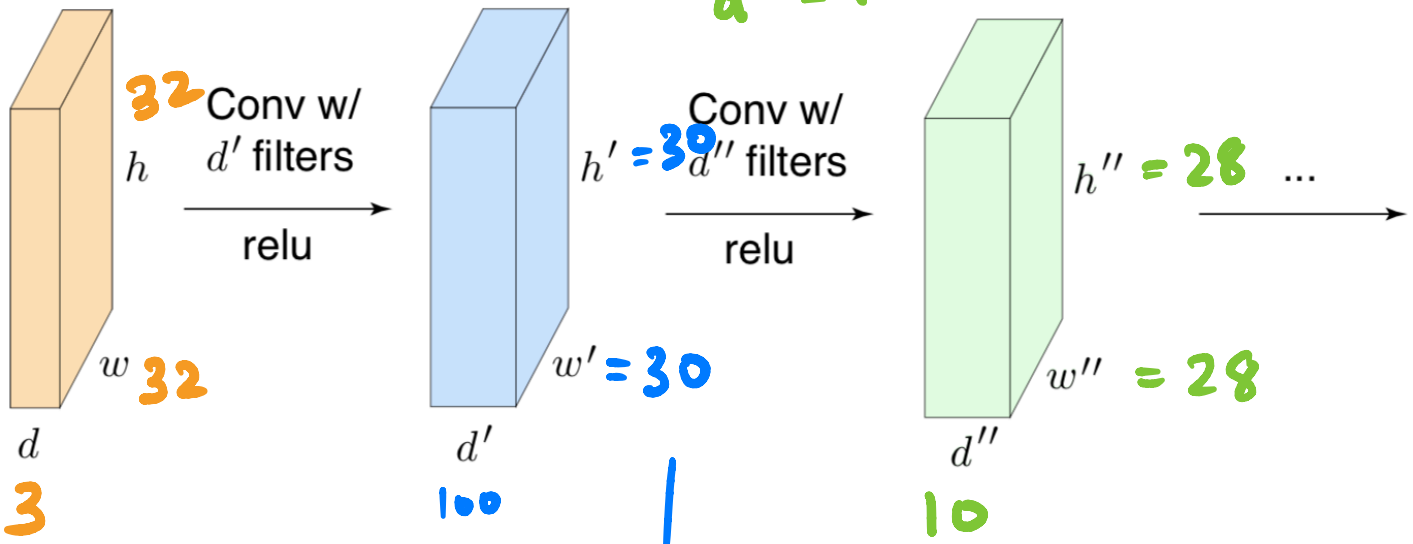
The output slices of the convolution operations with each filter are composed together to form a $(w - w_f + 1, h - h_f + 1, n_f)$ tensor, where $n_f$ is the number of filters. The output is then passed through an activation nonlinearity, such as $\mathrm{ReLU}(\cdot)$. This then acts as the input to the next layer.



# The convolutional layer of a CNN

## Composition of convolutional layers

These layers can then be composed, which comprise a large component of the CNN.

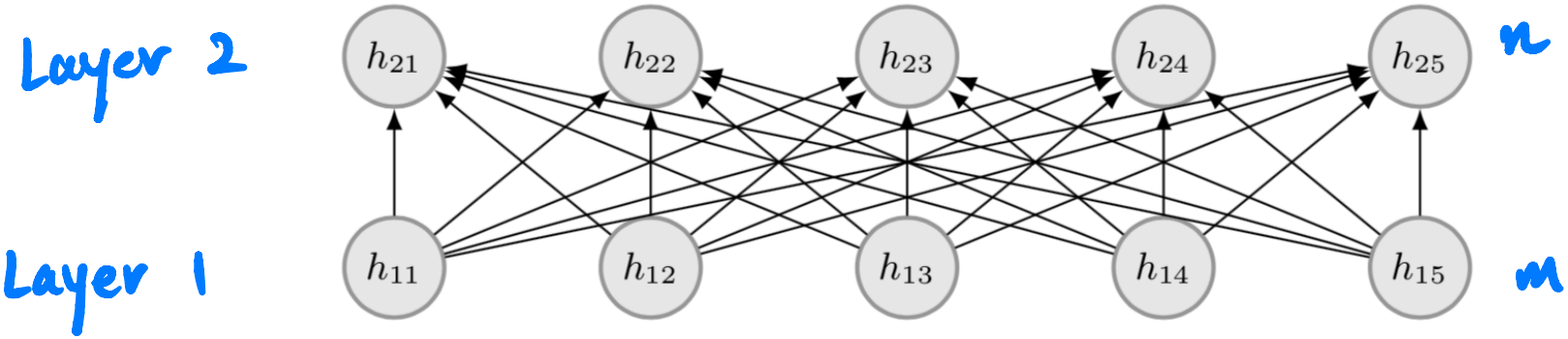


## Convolutional layers have sparse interactions

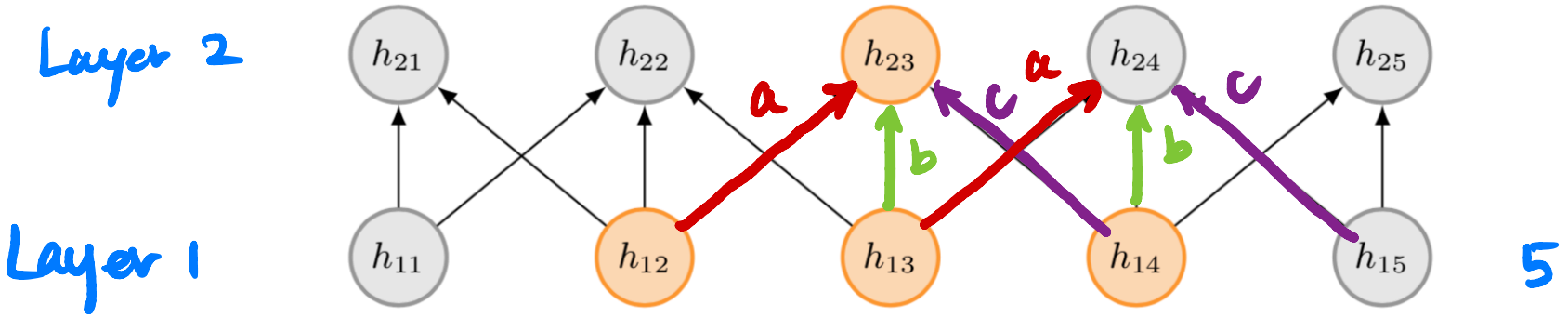
Convolutional layers have *sparse interactions* or *sparse connectivity*. The idea of sparse connectivity is illustrated below, where each output is connected to a small number of inputs.

$Wh_1 + b$

Fully connected layers:




Sparsely connected layers:



## Convolutional layers have sparse interactions (cont.)

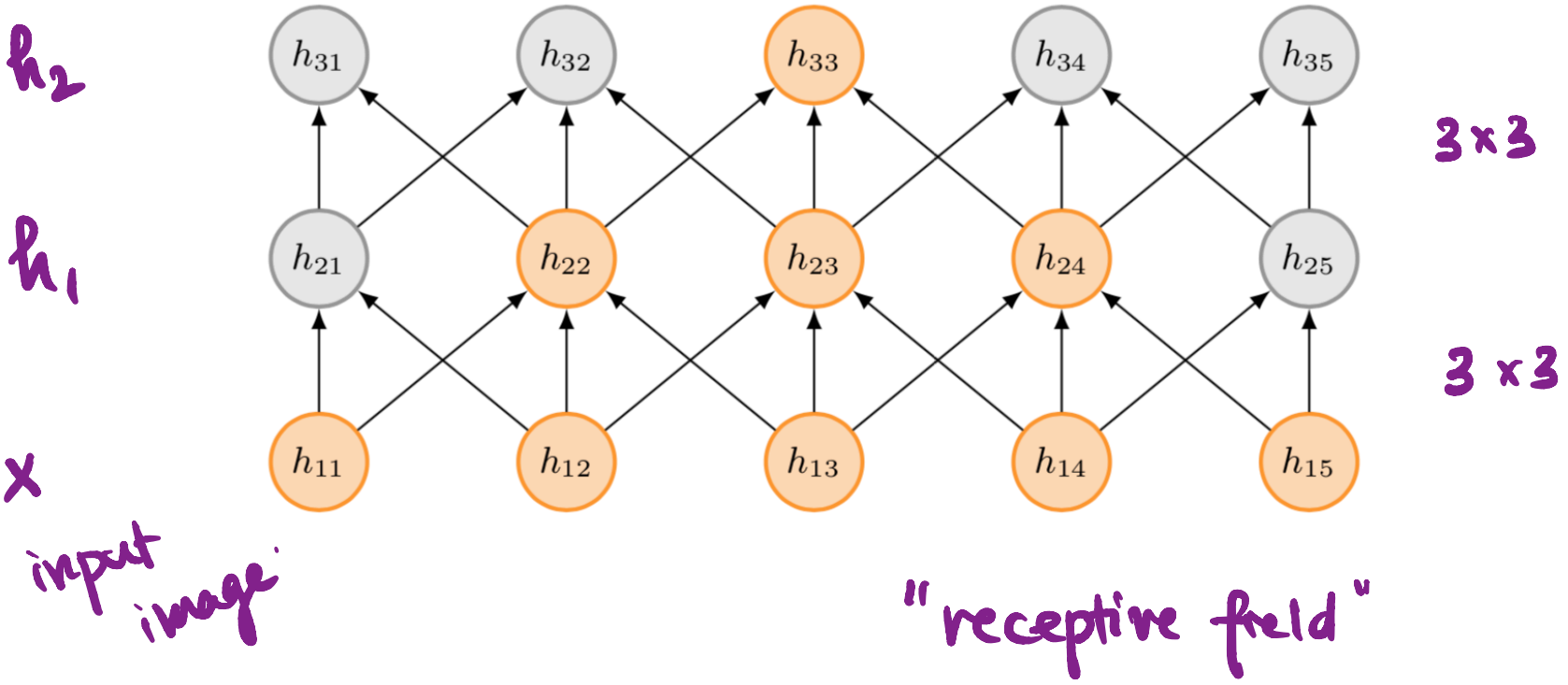
Sparse interactions aid with computation.

- **Sparse interactions reduce computational memory.** In a fully connected layer, each neuron has $w \cdot h \cdot d$ weights corresponding to the input. In a convolutional layer, each neuron has $w_f \cdot h_f \cdot d$ weights corresponding to the input. Moreover, in a convolutional layer, every output neuron in a slice has the *same* $w_f \cdot h_f \cdot d$ weights (more on this later). As there are far fewer parameters, this reduces the memory requirements of the model.
- **Sparse interactions reduce computation time.** If there are $m$ inputs and $n$ outputs in a hidden layer, a fully connected layer would require $\mathcal{O}(mn)$ operations to compute the output. If each output is connected to only $k$ inputs (e.g., where $k = w_f \cdot h_f \cdot d$) then the layer would require $\mathcal{O}(kn)$ operations to compute the output.

## Convolutional layers have sparse interactions (cont.)

A concern of sparse interactions is that information information from different parts of the input may not interact. For example, a self-driving car should know where obstacles are from all over the image to avoid them.

This argues that networks should be composed of more layers, since units in deeper layers indirectly interact with larger portions of the input.



## Convolutional layers share parameters

Convolutional layers have shared parameters (or *tied weights*), in every output neuron in a given slice uses the same set of parameters in the filter.

*Example*: Consider an input that is $(32 \times 32 \times 3)$. We have two architectures; in the fully connected architecture, there are $500$ output neurons at the first layer. In the convolutional neural net there are 4 filters that are all $4 \times 4$. (1) How many output neurons are there in the convolutional neural network, assuming that the convolution is only applied in regions where the filter fully overlaps the kernel? (2) How many parameters are in each model?

(1) input: $(32 \times 32 \times 3)$ convolved w/ filters $(4 \times 4 \times 3)$

$(w - w_f + 1,\ h - h_f + 1) \longrightarrow$ output of 1 filter: $(29, 29)$

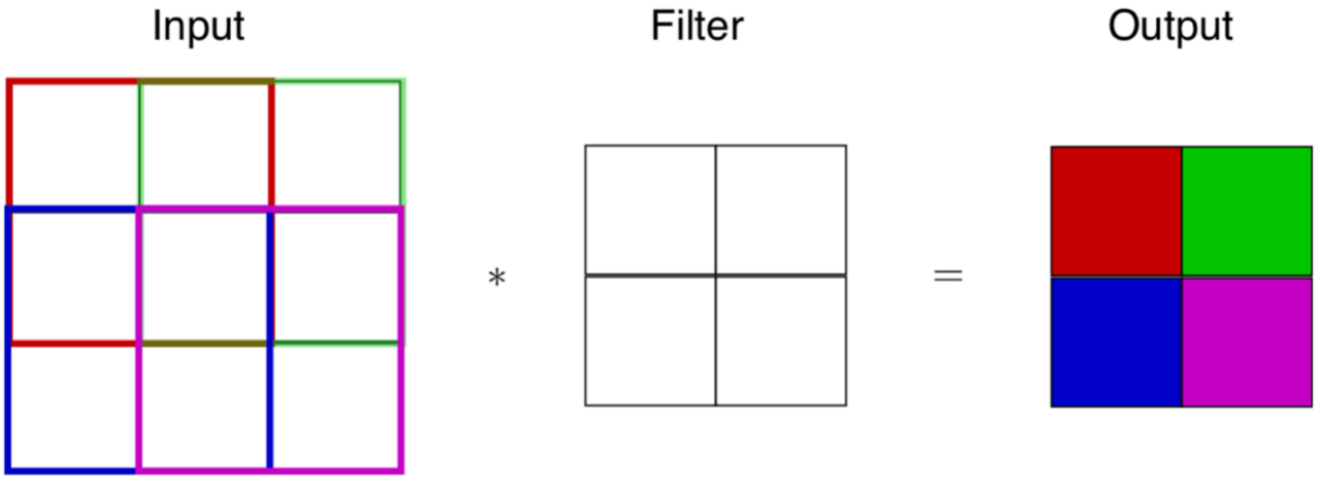
# of neurons: $29 \times 29 \times 4$

(2) FC: $(32 \times 32 \times 3 + 1) \cdot 500 = 1.5$ million params.

Conv: $(4 \times 4 \times 3 + 1) \cdot 4 = 196$ params

# All convolutions in this class are valid convolutions

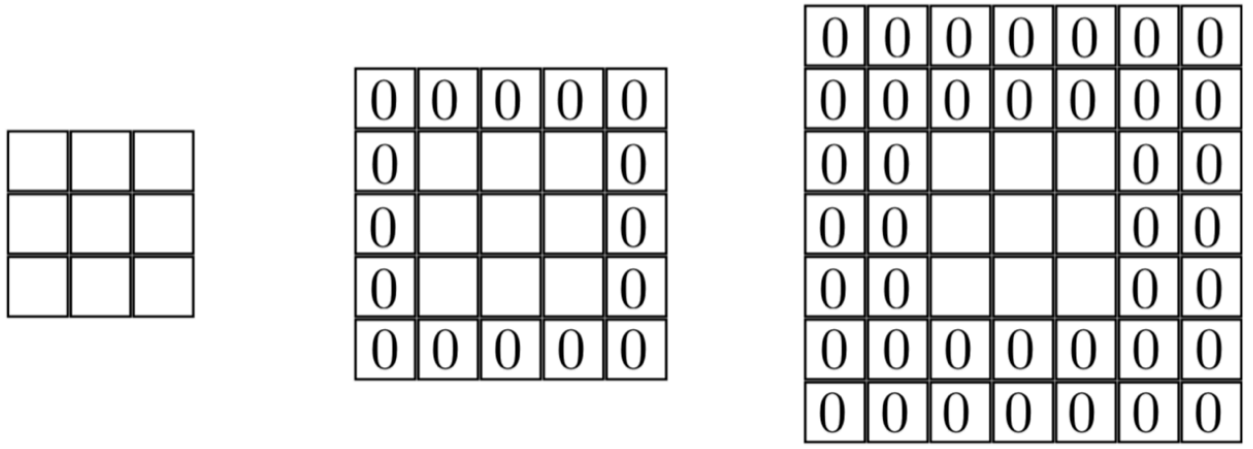
**In this class, all convolutions will be valid convolutions. We explicitly specify the amount of zero padding when we need it.**




Output size: $(w - w_f + 1, h - h_f + 1)$

# Convolutional padding

$\text{pad} = 0$ $\text{pad} = 1$ $\text{pad} = 2$




The output of the convolution is now $(w - w_f + 1 + 2\text{pad}, h - h_f + 1 + 2\text{pad})$.

pad = 1

$w$ $w_f = 3$

$w' = w - 3 + 1 + 2$

$= w$



## Convolution padding (cont.)

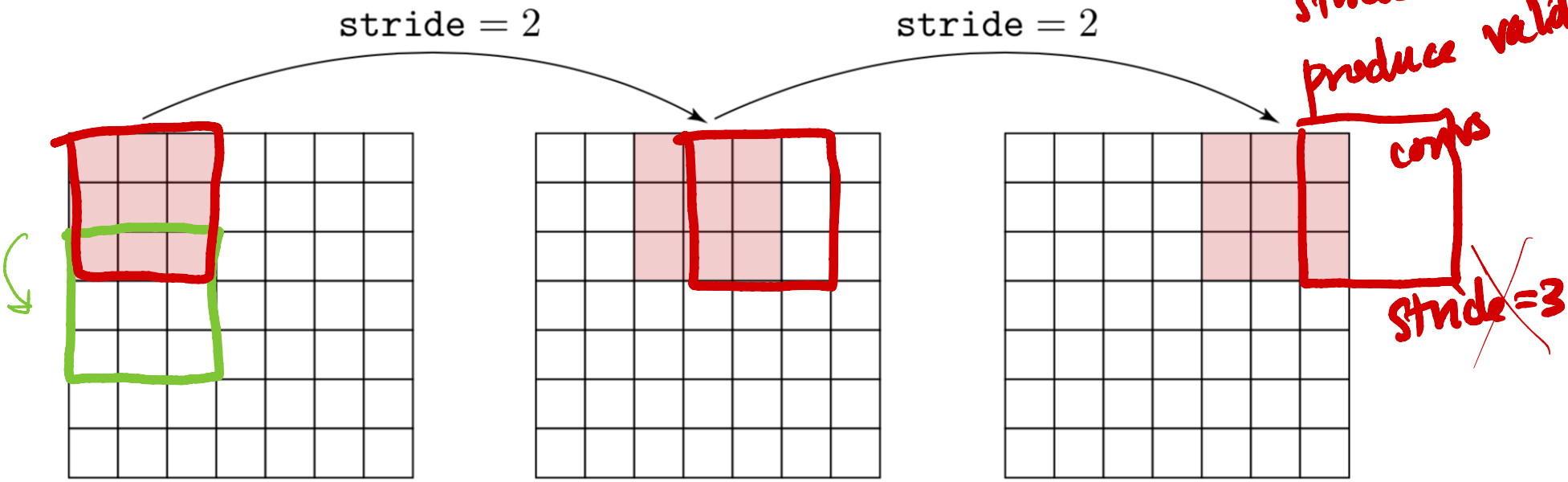
It is worth noting that Goodfellow et al. report that the optimal amount of zero padding (in terms of test accuracy) is somewhere between $\texttt{pad} = 0$ and the pad that causes the output and input to have the same width and height.

# Convolutional stride

$$\left(\frac{w - w_f + 2 \cdot pad}{stride} + 1, \frac{h - h_f + 2 \cdot pad}{stride} + 1\right)$$

## Convolution stride

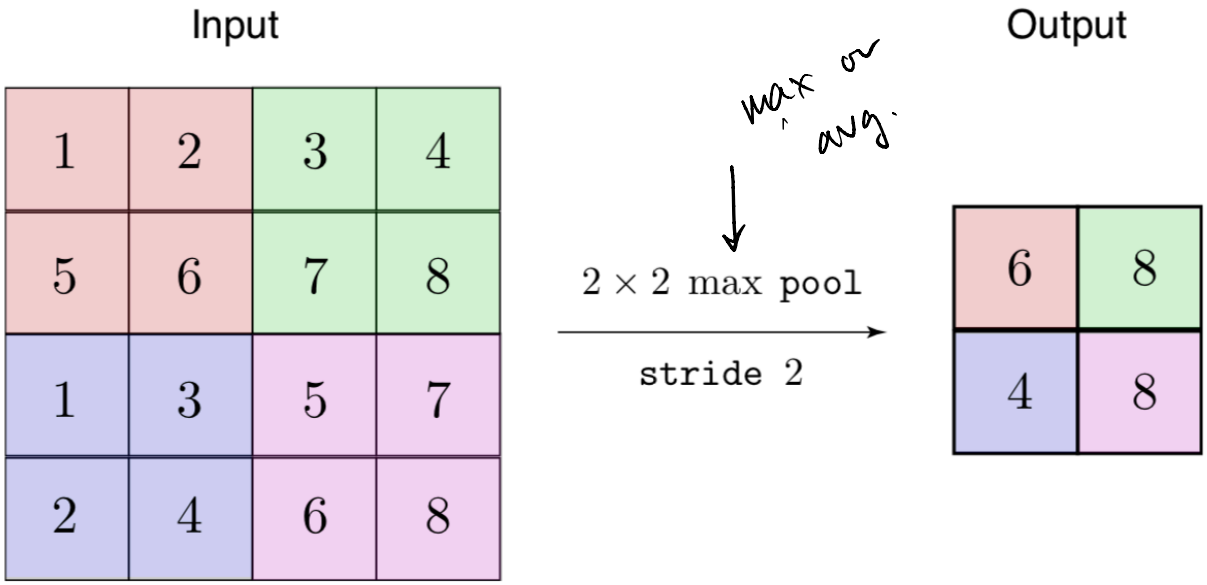
Another variable to control is the stride, which defines how much the filter moves in the convolution. For normal convolution, $\texttt{stride} = 1$, which denotes that the filter is dragged across every part of the input. Consider a $7 \times 7$ input with a $3 \times 3$ filter. If convolution is only applied where the kernel overlaps the input, the output is $5 \times 5$. With $\texttt{stride} = 2$, the output is $3 \times 3$.



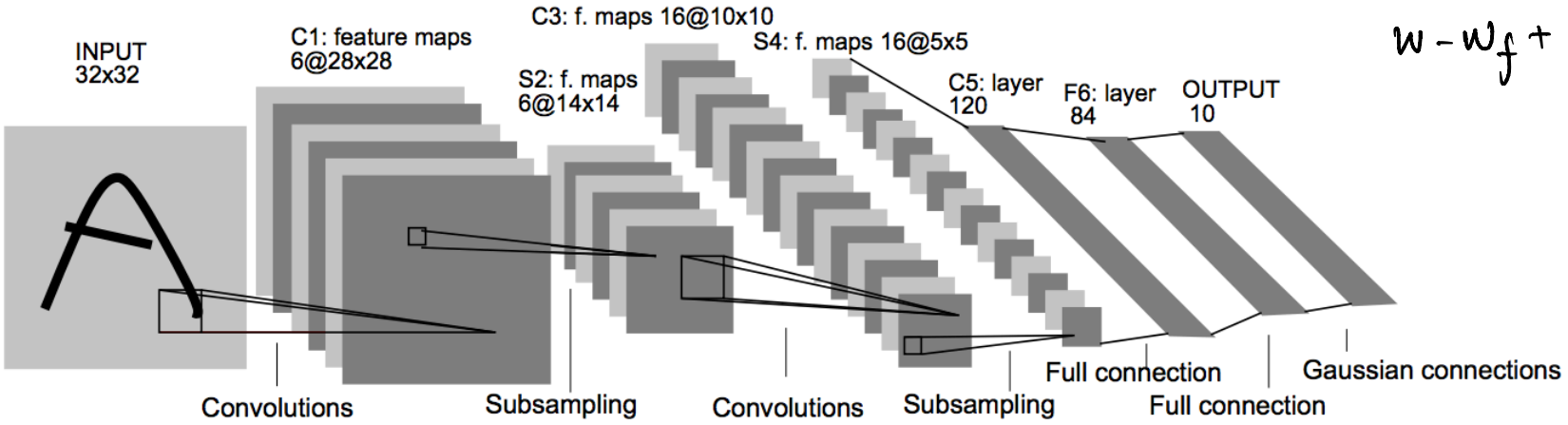
## Pooling layer

CNNs also incorporate pooling layers, where an operation is applied to all elements within the filtering extent. This corresponds, effectively, to downsampling.

The pooling filter has width and height $(w_p, h_p)$ and is applied with a given stride. It is most common to use the $\max()$ operation as the pooling operation.



# Sizing examples




$W - W_f + 1$

output

**C1 contains six 5x5 conv filters.** stride = 1, pad = 0

Size of feature maps at C1? $32 - 5 + 1 = 28$ $(28 \times 28 \times 6)$

Number of parameters in C1 layer? $(5 \times 5 \times 1 + 1) \cdot 6 = 156$ params

**S2 is a 2x2 pooling layer applied at stride 2.**

Size of feature maps at S2? $(14 \times 14 \times 6)$

Number of parameters in S2 layer? 0

**C3 contains sixteen 5x5 conv filters.**

Number of parameters in C3 layer? $(5 \times 5 \times 6 + 1) \cdot 16$

Size of feature maps at C3? $14 - 5 + 1 = 10$ $(10 \times 10 \times 16)$