Inteligencia Artificial

UTN – FRVM

5º Año Ing. en Sistemas de Información

Agenda

- Neural Networks
  - SGD
  - Backpropagation
  - ConvNets

Vanilla GD

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

- **Mini-batch gradient descent.** In large-scale applications, the training data can have on order of millions of examples. Hence, it seems wasteful to compute the full loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over **batches** of the training data. For example, in current state of the art ConvNets, a typical batch contains 256 examples from the entire training set of 1.2 million. This batch is then used to perform a parameter update:

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic GD

- The extreme case of this is a setting where the mini-batch contains only a single example.
- This process is called **Stochastic Gradient Descent (SGD)** (or also sometimes **on-line** gradient descent).
- This is relatively less common to see because in practice due to **vectorized code** optimizations it can be computationally much more efficient to evaluate the gradient for 100 examples, than the gradient for one example 100 times.
- Even though SGD technically refers to using a single example at a time to evaluate the gradient, the term SGD is used when referring to **mini-batch gradient descent** (i.e. mentions of MGD for “Minibatch Gradient Descent”, or BGD for “Batch gradient descent” are rare to see), where it is usually assumed that mini-batches are used.
- The size of the mini-batch is a **hyperparameter** but it is not very common to cross-validate it. It is usually based on memory constraints (if any), or set to some value, e.g. 32, 64 or 128. We use powers of 2 in practice because many vectorized operation implementations work faster when their inputs are sized in powers of 2.

Backpropagation

Backpropagation: a simple example

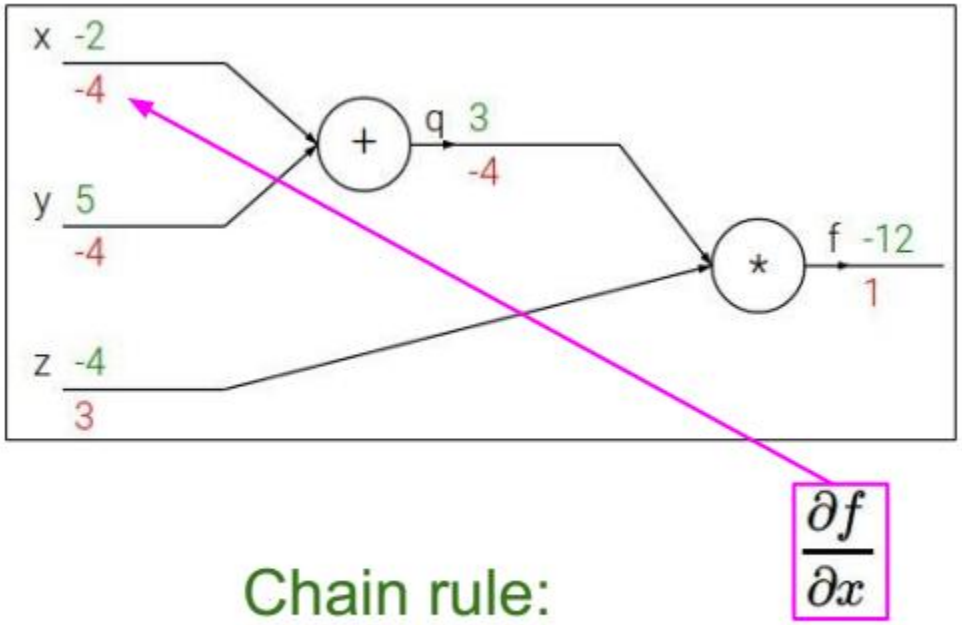
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z = -4

$$\boxed{q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1}$$

$$\boxed{f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q}$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



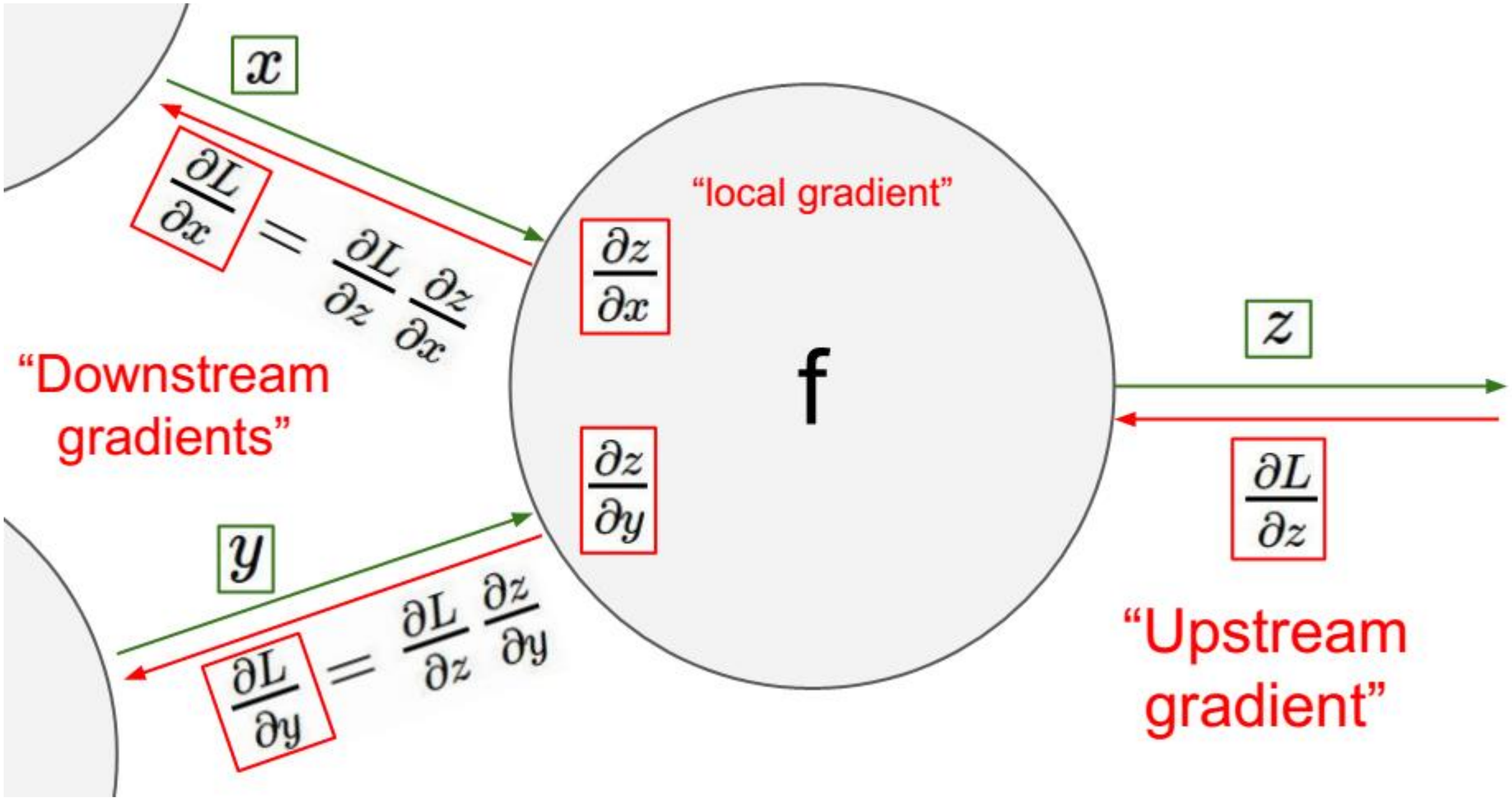
Chain rule:

$$\boxed{\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}}$$

Upstream gradient

Local gradient

Backpropagation



x
$\boxed{\frac{\partial L}{\partial x}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
z
"Downstream gradients"
$\frac{\partial z}{\partial y}$
y
$\boxed{\frac{\partial L}{\partial y}} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
$\frac{\partial L}{\partial z}$
"Upstream gradient"

Convolutional Neural Networks (CNNs / ConvNets)

- Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases.
- Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity.
- The whole network still expresses a **single differentiable score function**: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.
- ConvNet architectures make the explicit assumption that **the inputs are images**, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

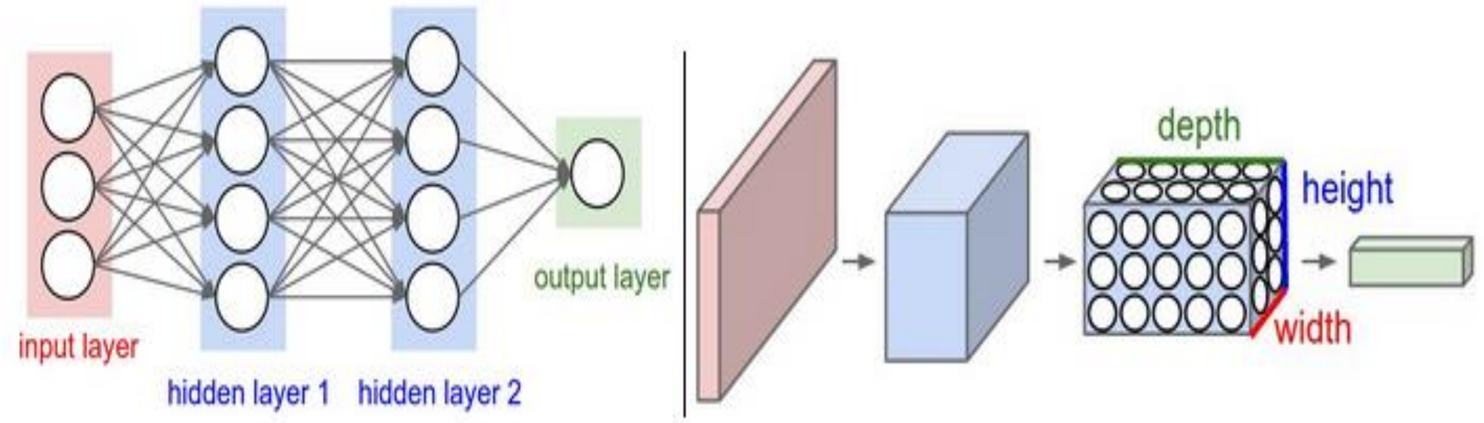
Architecture Overview

- *3D volumes of neurons*. Convolutional Neural Networks take advantage of the fact that the input consists of images

- The layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**.

- For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). The neurons in a layer will only be connected to a small region of the layer before it.
- Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.

Architecture Overview



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Layers used to build ConvNets

- A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function.
- Types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

  - INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
  - CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
  - RELU layer will apply an elementwise activation function, such as the max(0,x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
  - POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
  - FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10.

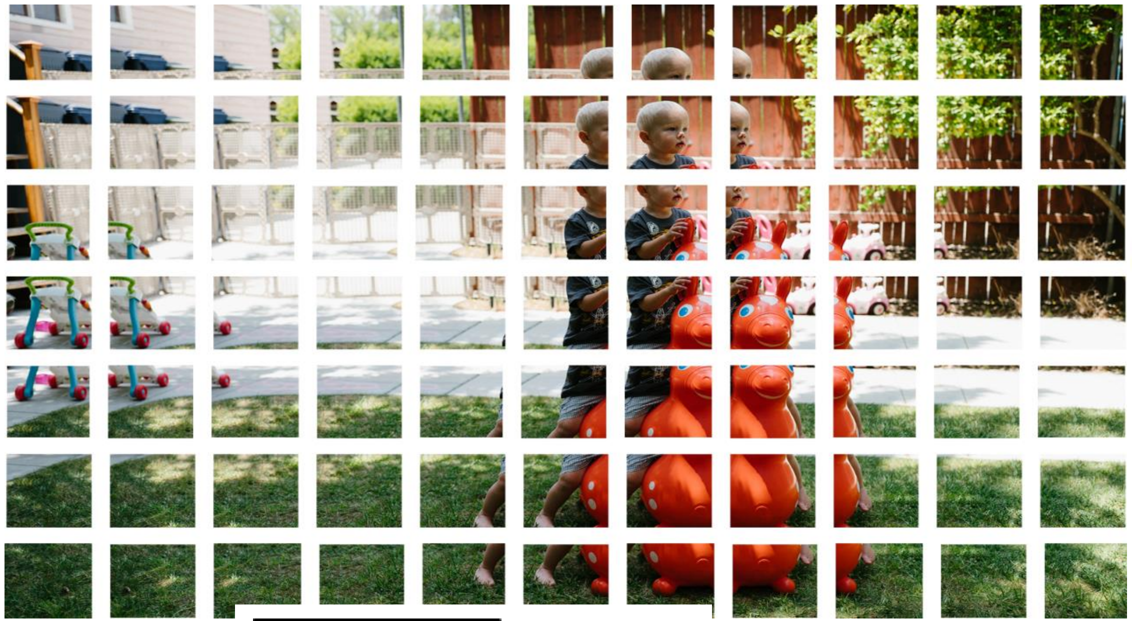
ConvNets

- Instead of feeding entire images into our neural network as one grid of numbers, we’re going to do something a lot smarter that takes advantage of the idea that an object is the same no matter where it appears in a picture.

- **Step 1: Break the image into overlapping image tiles**

ConvNets

By doing this, we turned our original image into 77 equally-sized tiny image tiles.



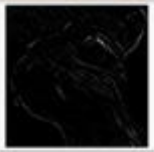
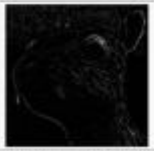
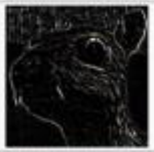
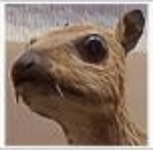
1	0	1
0	1	0
1	0	1

1	1	1	0	0
0	1	1	1	0
0	0	$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$
0	0	$1_{\times 0}$	$1_{\times 1}$	$0_{\times 0}$
0	1	$1_{\times 1}$	$0_{\times 0}$	$0_{\times 1}$

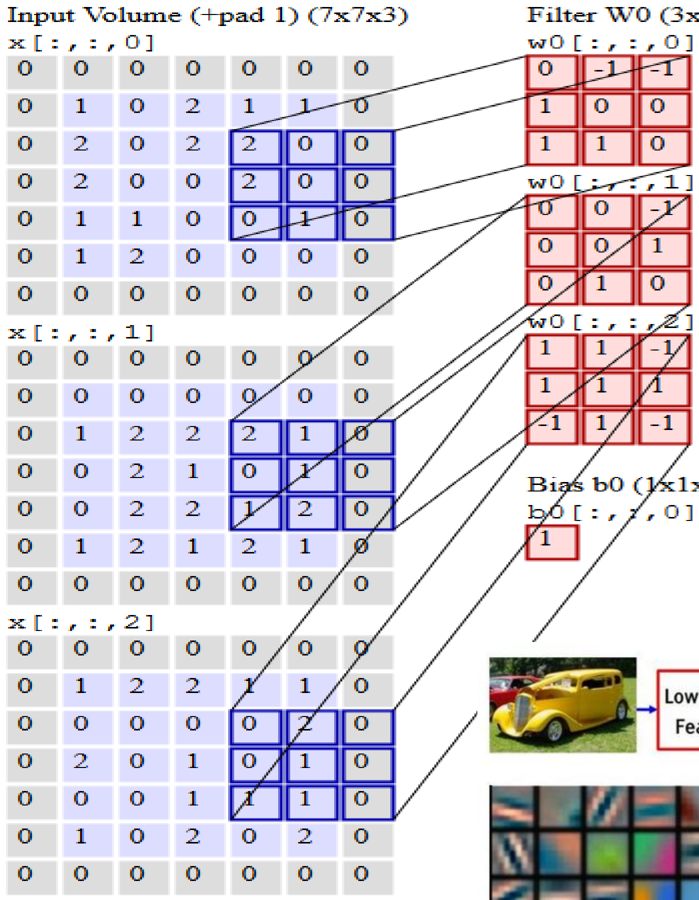
Image

4	3	4
2	4	3
2	3	4

Convolved Feature

Operation	Filter	Convolved Image
Identity	$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix}$	
Edge detection	$\begin{bmatrix}1&0&-1\\0&0&0\\-1&0&1\end{bmatrix}$	
	$\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}$	
	$\begin{bmatrix}-1&-1&-1\\-1&8&-1\\-1&-1&-1\end{bmatrix}$	
Sharpen	$\begin{bmatrix}0&-1&0\\-1&5&-1\\0&-1&0\end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	

Ejemplo…



Input Volume (+pad 1) (7x7x3)

x[:,:,0]

0	0	0	0	0	0	0
0	1	0	2	1	1	0
0	2	0	2	2	0	0
0	2	0	0	2	0	0
0	1	1	0	0	1	0
0	1	2	0	0	0	0
0	0	0	0	0	0	0

x[:,:,1]

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	1	2	2	2	1	0
0	0	2	1	0	1	0
0	0	2	2	1	2	0
0	1	2	1	2	1	0
0	0	0	0	0	0	0

x[:,:,2]

0	0	0	0	0	0	0
0	1	2	2	1	1	0
0	0	0	0	0	2	0
0	2	0	1	0	1	0
0	0	0	1	1	1	0
0	1	0	2	0	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

w0[:,:,0]

0	-1	-1
1	0	0
1	1	0

w0[:,:,1]

0	0	-1
0	0	1
0	1	0

w0[:,:,2]

1	1	-1
1	1	1
-1	1	-1

Bias b0 (1x1x1)

b0[:,:,0]

1

Filter W1 (3x3x3)

w1[:,:,0]

-1	0	1
1	1	-1
1	1	-1

w1[:,:,1]

0	0	1
1	-1	0
0	-1	-1

w1[:,:,2]

1	0	1
0	1	0
1	1	-1

Bias b1 (1x1x1)

b1[:,:,0]

0

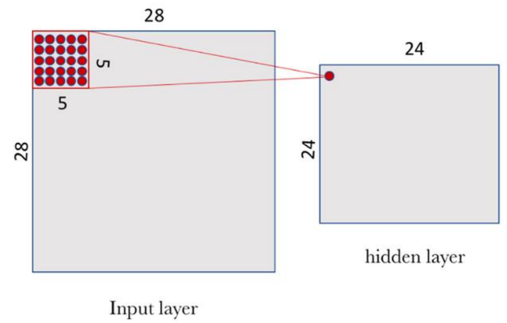
Output Volume (3x3x2)

o[:,:,0]

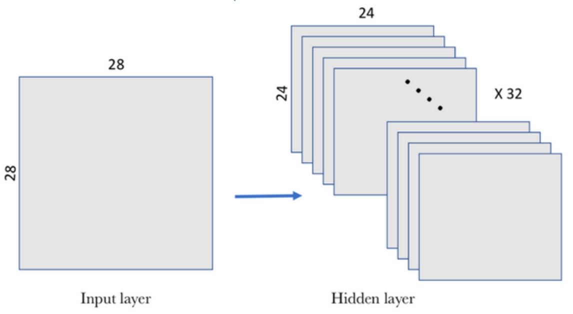
7	10	9
2	-1	9
0	6	4

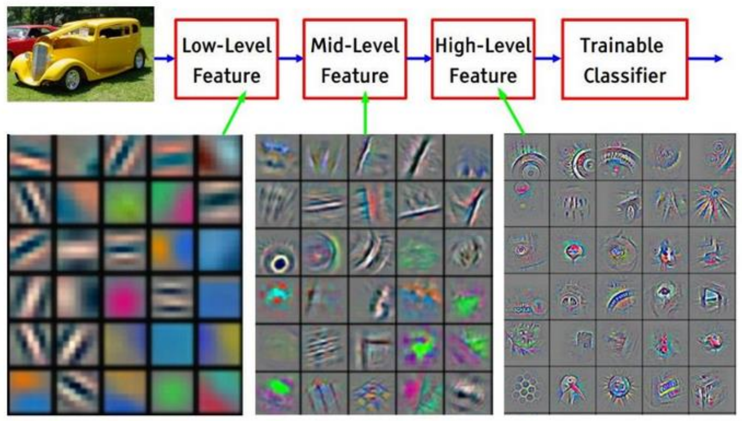
o[:,:,1]

1	-1	6
4	2	1
2	6	4



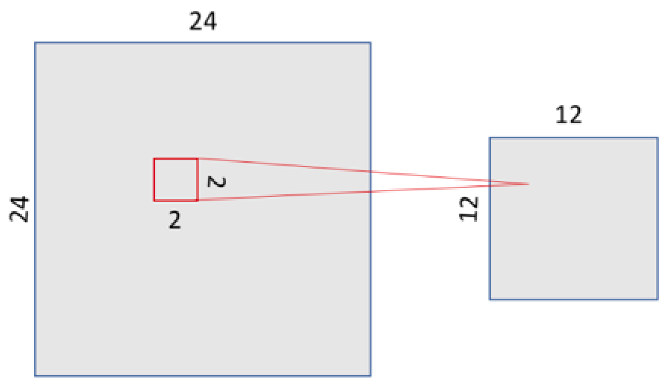
N-filters

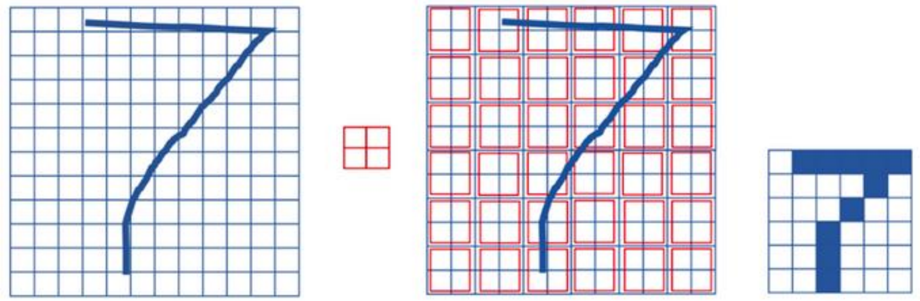


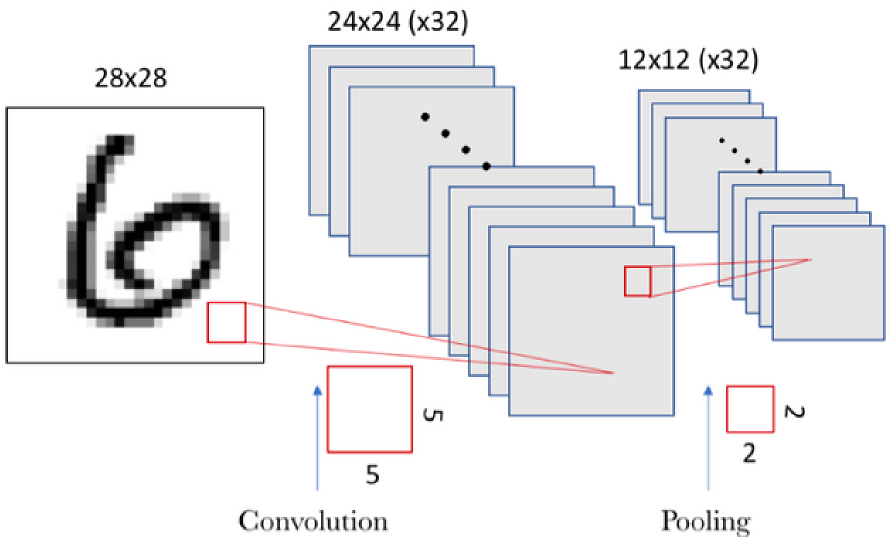


Padding?
Stride?

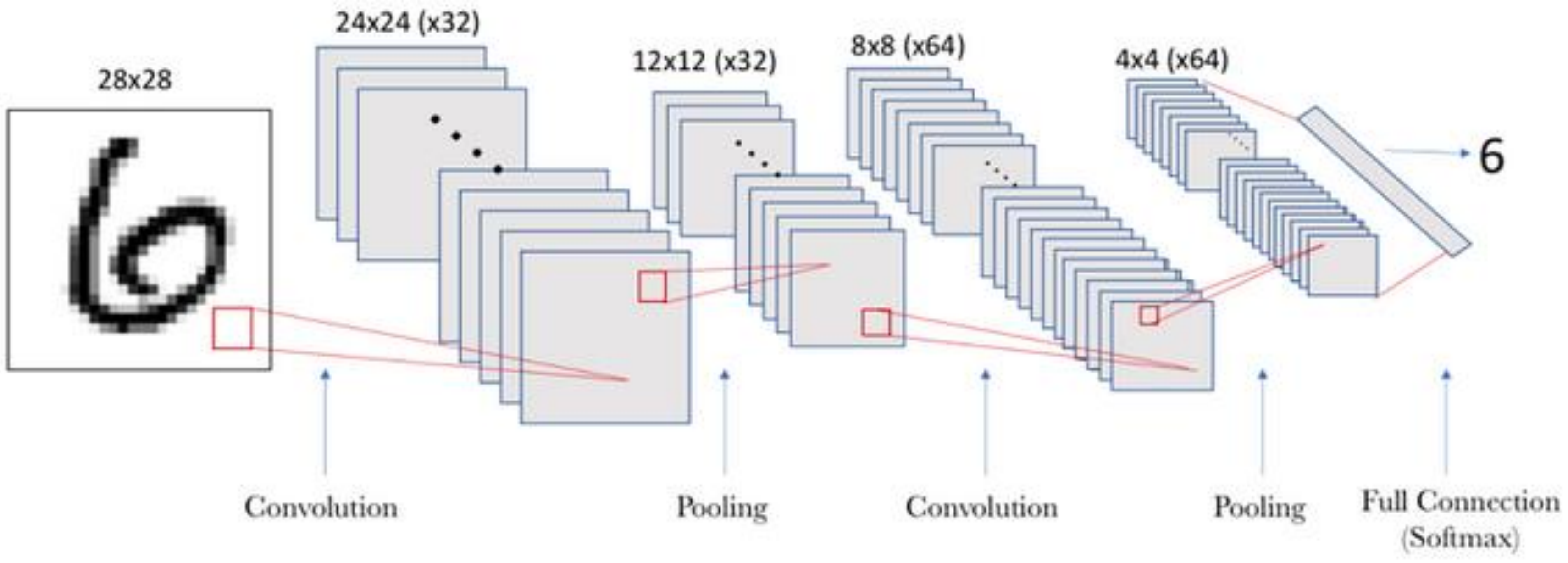
Pooling


24
24
2
2
12
12

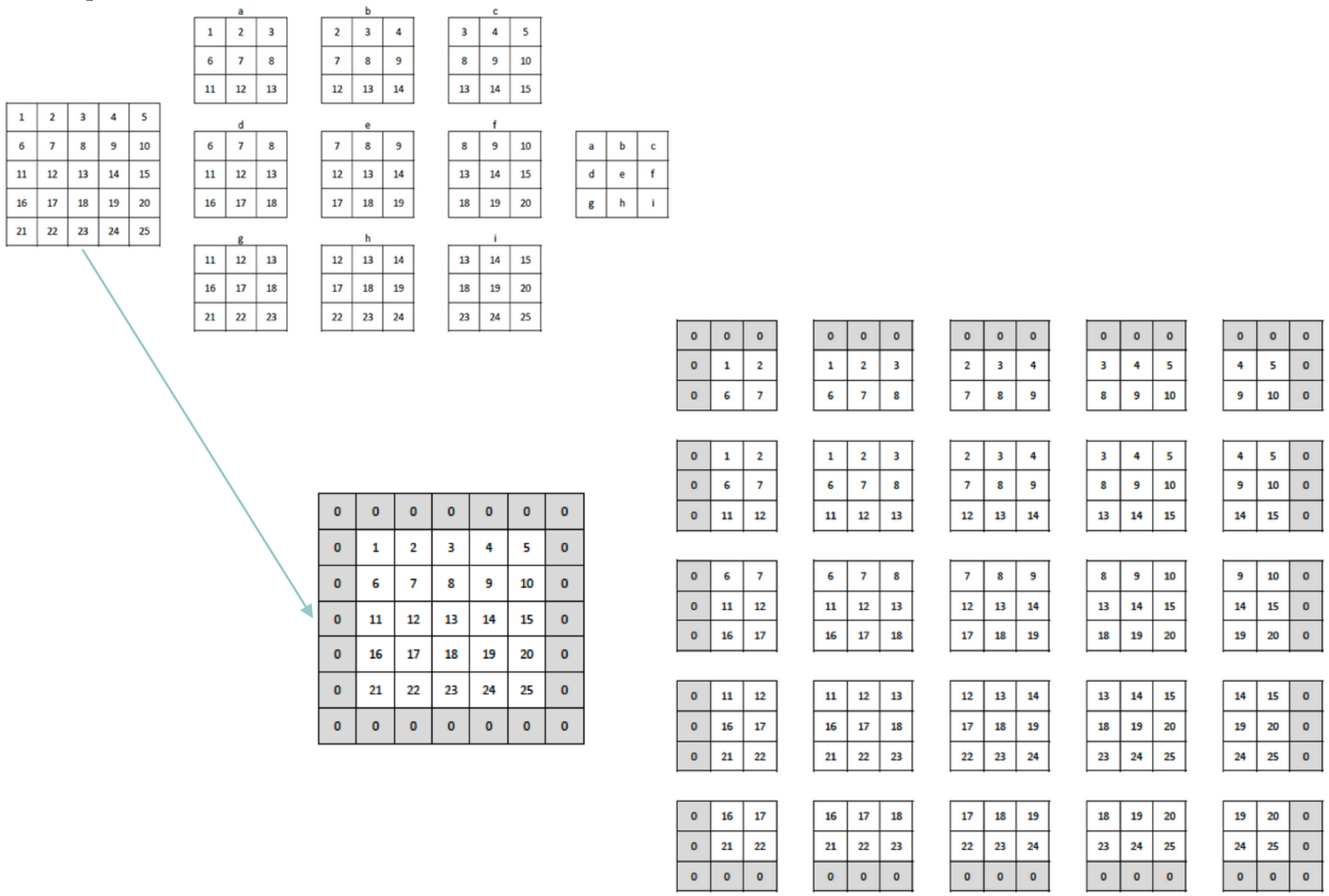



28x28
24x24 (x32)
12x12 (x32)
5
5
Convolution
2
2
Pooling

Conv+Pooling+Dense

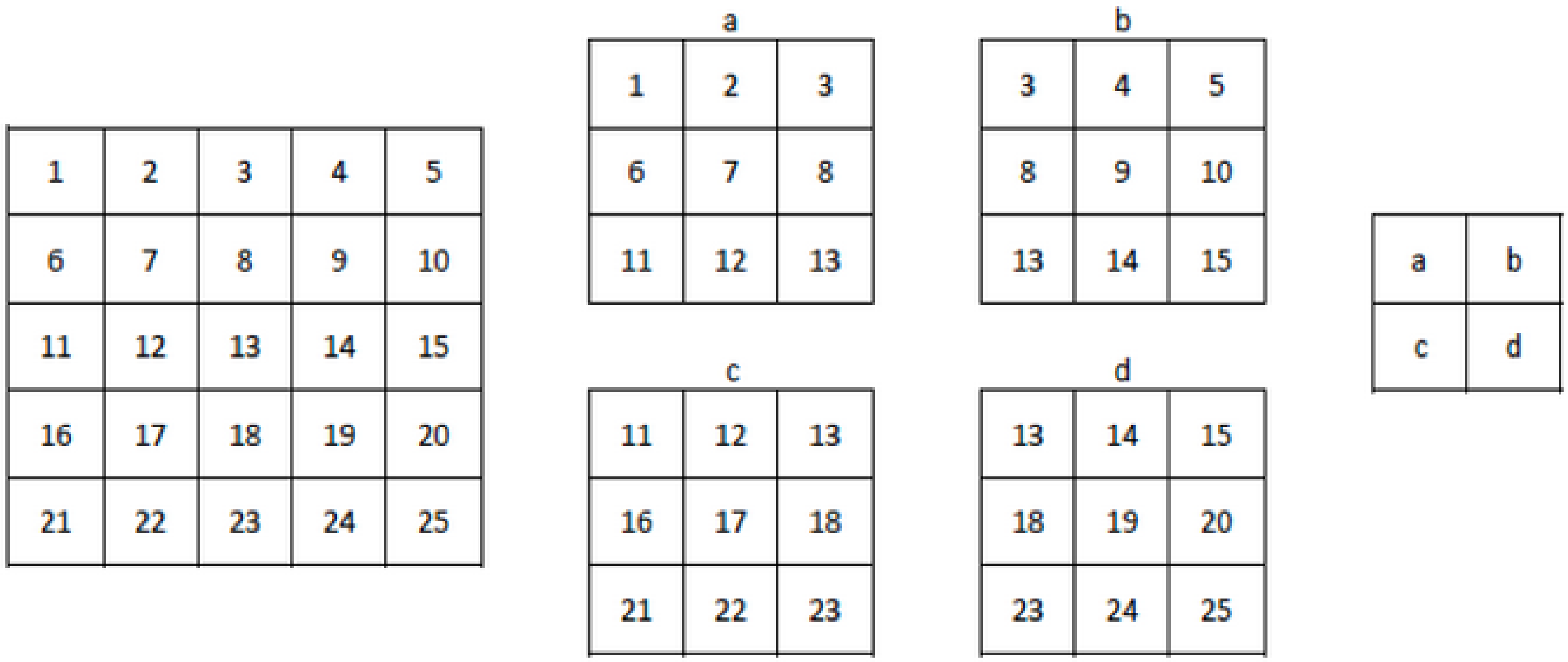


Padding



Stride

=2



1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

a

1	2	3
6	7	8
11	12	13

b

3	4	5
8	9	10
13	14	15

c

11	12	13
16	17	18
21	22	23

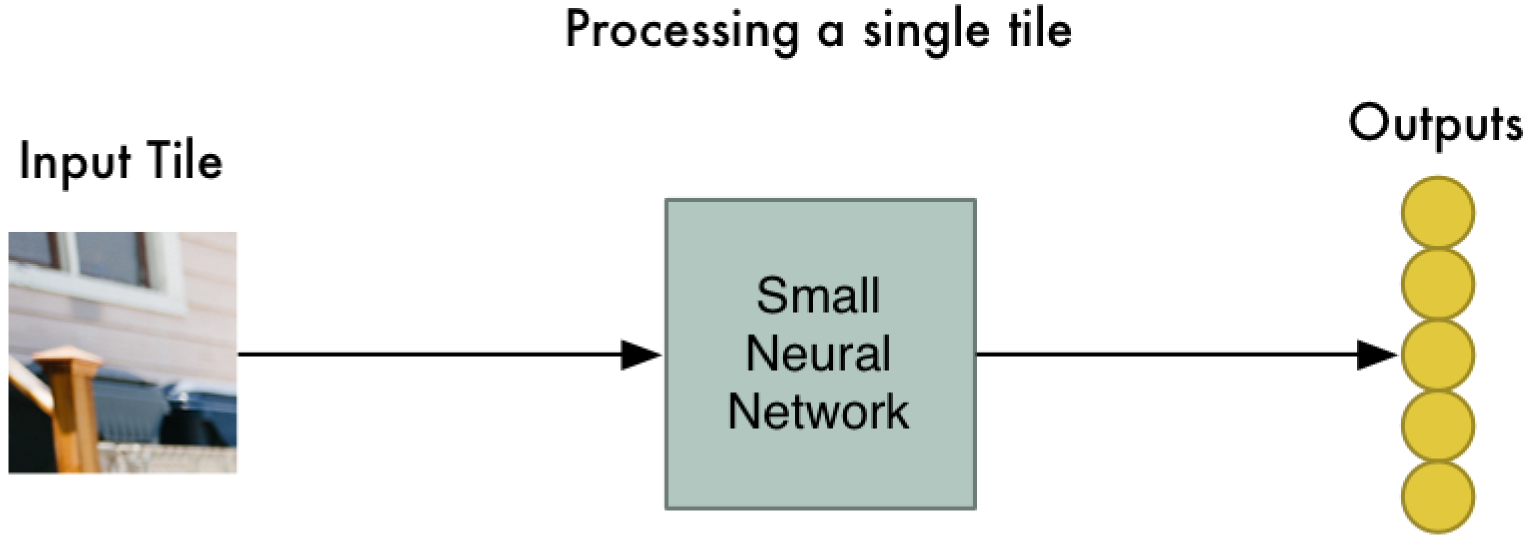
d

13	14	15
18	19	20
23	24	25

a	b
c	d

ConvNets

- **Step 2: Feed each image tile into a small neural network**

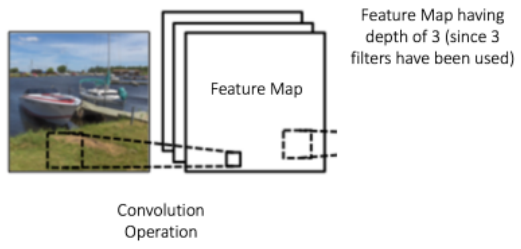


However, **there's one big twist**: We'll keep the **same neural network weights** for every single tile in the same original image. In other words, we are treating every image tile equally. If something interesting appears in any given tile, we'll mark that tile as interesting.

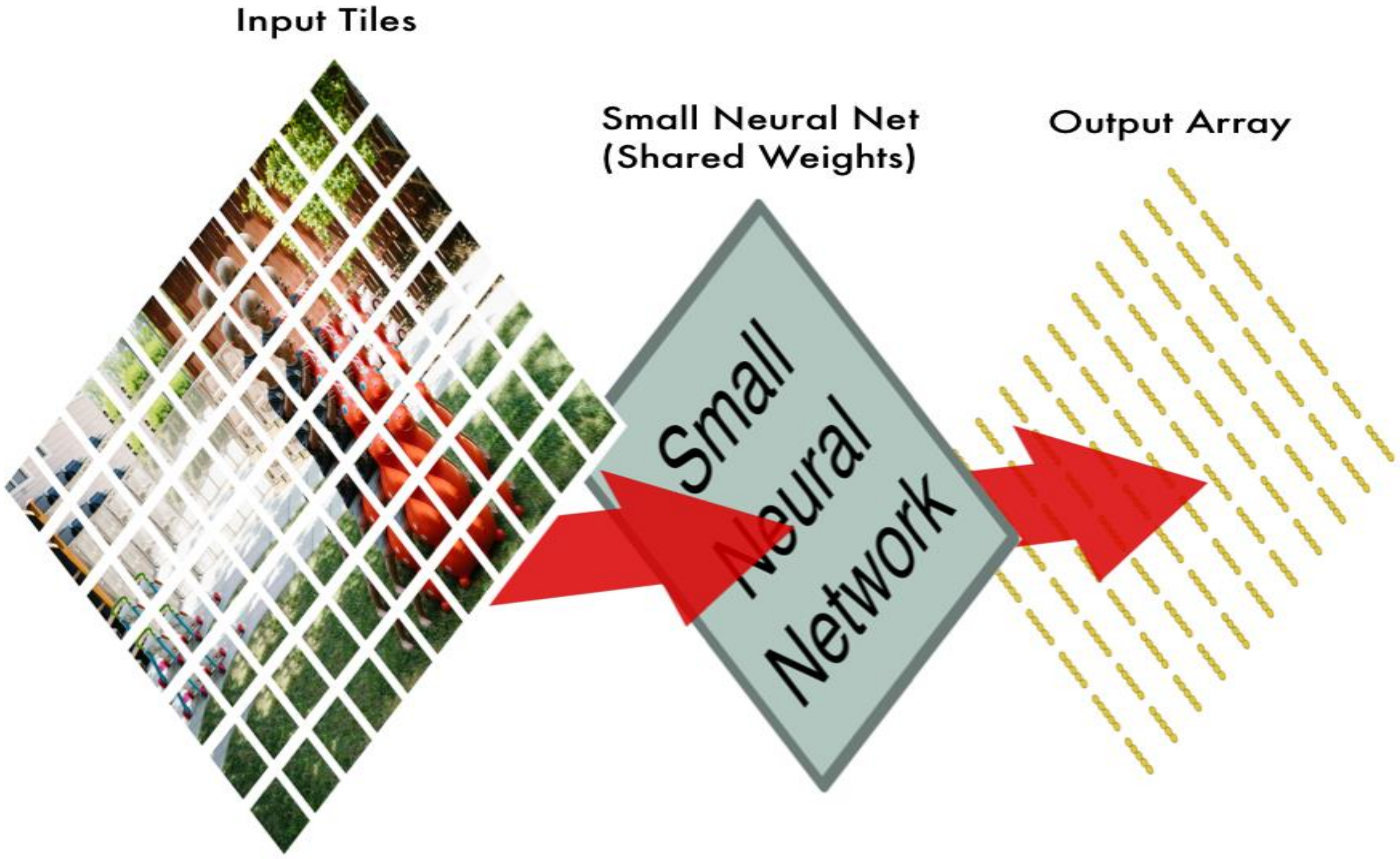
ConvNets

- **Step 3: Save the results from each tile into a new array**

- We don't want to lose track of the arrangement of the original tiles. So we save the result from processing each tile into a grid in the same arrangement as the original image.
- In other words, we've started with a large image and we ended with a slightly smaller array that records which sections of our original image were the most interesting.

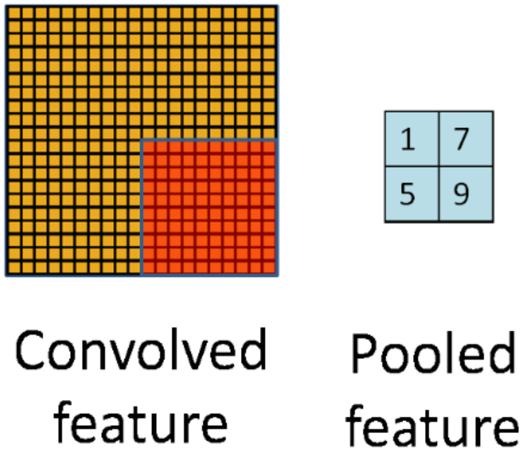

Feature Map having depth of 3 (since 3 filters have been used)
Feature Map
Convolution Operation

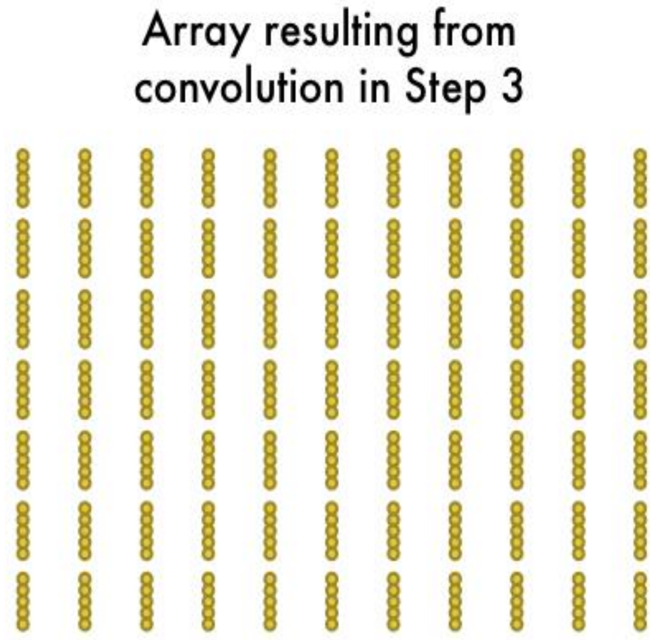
ConvNets


Input Tiles
Small Neural Net
(Shared Weights)
Output Array
Small
Neural
Network

ConvNets

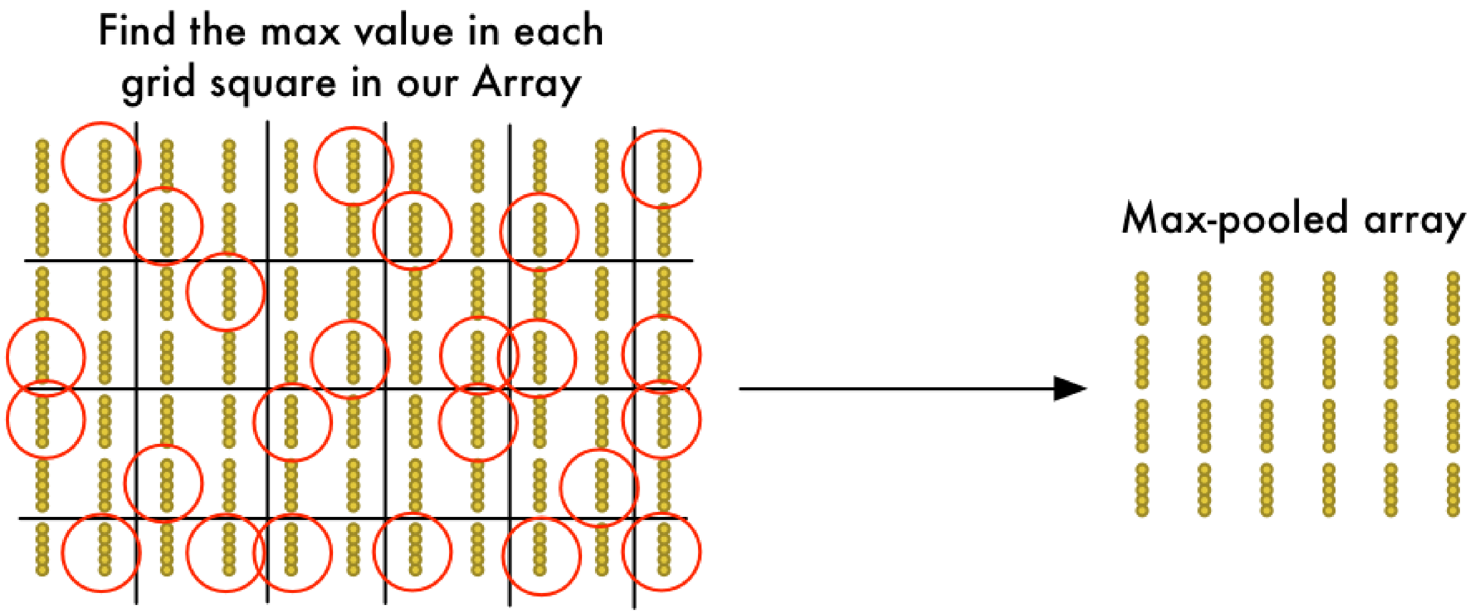
- **Step 4: Downsampling**





To reduce the size of the array, we *downsample* it using an algorithm called max pooling.

ConvNets

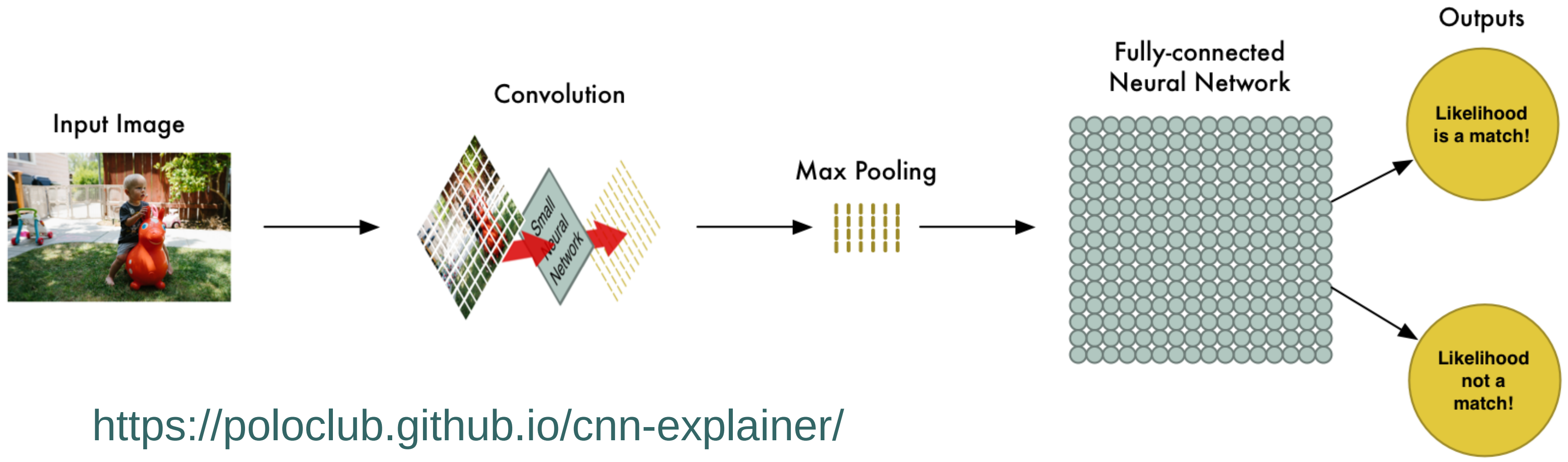


The idea here is that if we found something interesting in any of the four input tiles that makes up each 2x2 grid square, we'll just keep the most interesting bit. This reduces the size of our array while keeping the most important bits.

ConvNets

Final step: Make a prediction

So far, we've reduced a giant image down into a fairly small array. That array is just a bunch of numbers, so we can use that small array as input into *another neural network*. This final neural network will decide if the image is or isn't a match. To differentiate it from the convolution step, we call it a "fully connected" network. So from start to finish, our whole five-step pipeline looks like this:



https://poloclub.github.io/cnn-explainer/