Convolutional Neural Networks (CNN)

Dr. Muneendra Ojha

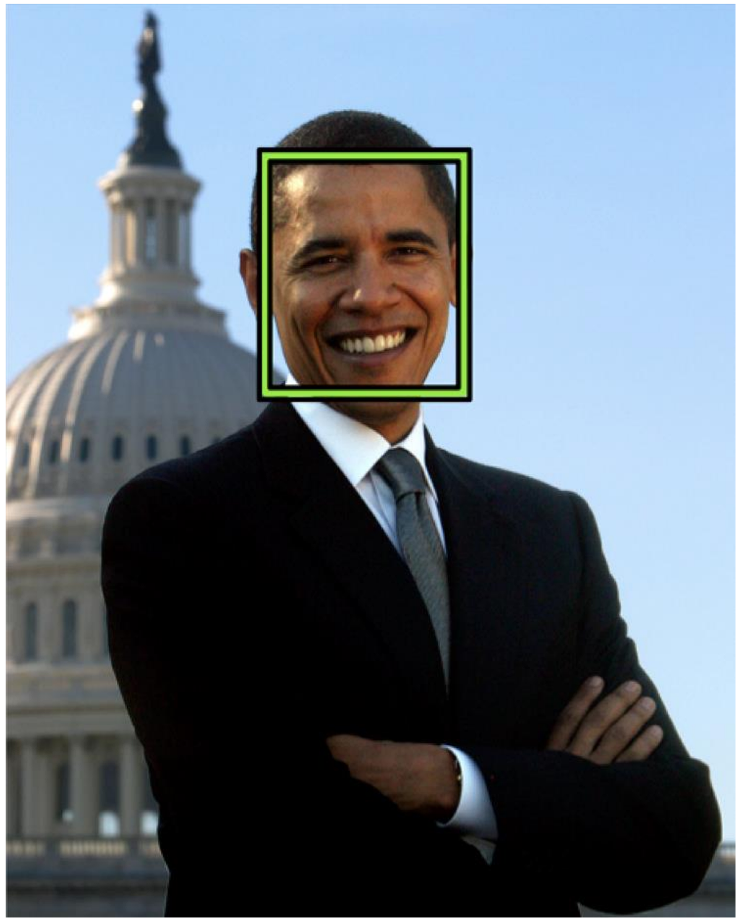
Assistant Professor

Department of Information Technology

Indian Institute of Information Technology - Allahabad

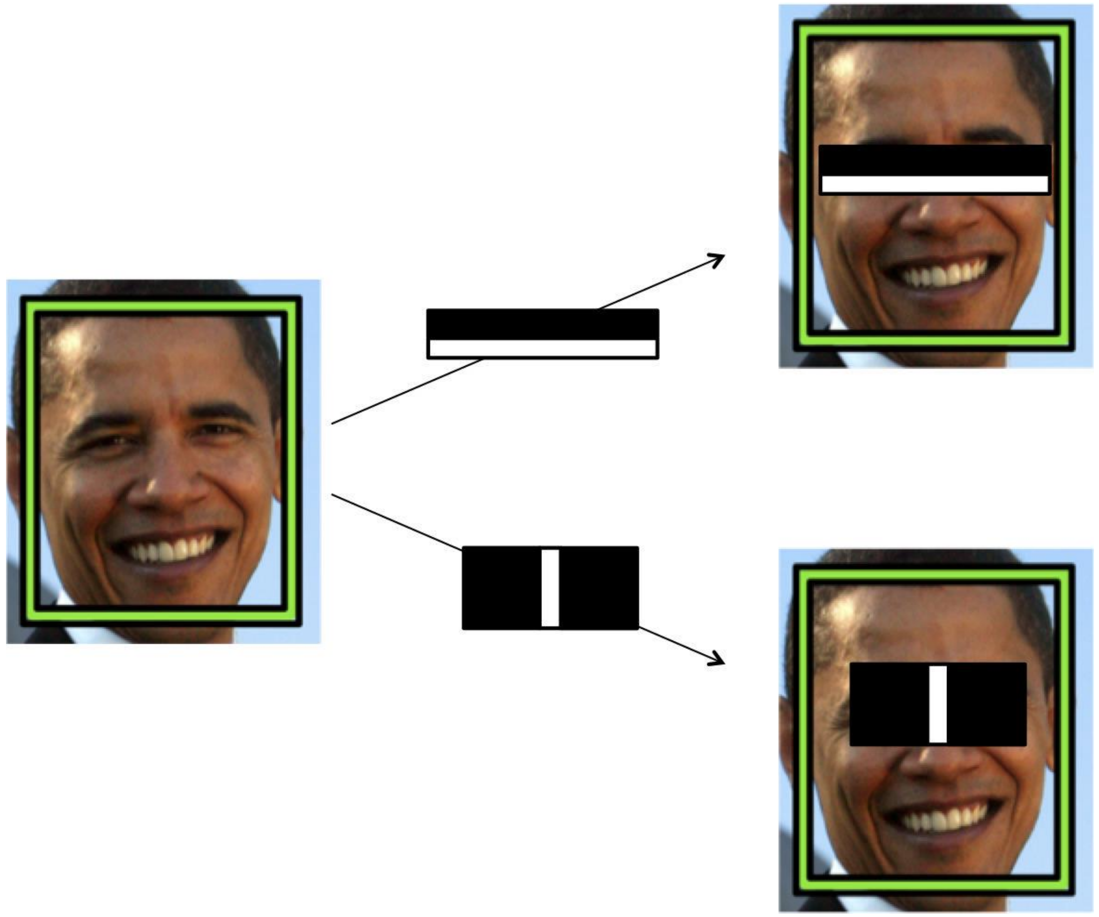
The Shortcomings of Feature Selection

- Take a randomly selected image and tell me if there is a human face in this picture.
- This is exactly the problem that Paul Viola and Michael Jones tackled in their seminal paper published in 2001.



Paul Viola, and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features." Computer Vision and Pattern Recognition, 2001. CVPR 2001. *Proceedings of the 2001 IEEE Computer Society Conference* on. Vol. 1. IEEE, 2001.

The Shortcomings of Feature Selection



- Faces had certain patterns of light and dark patches that can be exploited.
- For example, there is a difference in light intensity between the eye region and the upper cheeks.
- There is also a difference in light intensity between the nose bridge and the two eyes on either side.

The Shortcomings of Feature Selection

- By themselves, each of these features is not very effective at identifying a face.
- But when used together (through a classic machine learning algorithm known as boosting, described in the original manuscript), their combined effectiveness drastically increases.
- On a dataset of 130 images and 507 faces, the algorithm achieves a 91.4% detection rate with 50 false positives

The Shortcomings of Feature Selection

- The performance was unparalleled at the time, but there are fundamental limitations of the algorithm.
- If a face is partially covered with shade, the light intensity comparisons no longer work.
- Moreover, if the algorithm is looking at a face on a crumpled flier or the face of a cartoon character, it would most likely fail.

The Shortcomings of Feature Selection

- The problem is the algorithm hasn’t really learned that much about what it means to “see” a face.
- Beyond differences in light intensity, our brain uses a vast number of visual cues to realize a human face, including contours, relative positioning of facial features, and color.
- If there are slight discrepancies in visual cues (e.g. if parts of the face are blocked or if shade modifies light intensities), our visual cortex can still reliably identify faces.

Regular Fully-Connected Neural Networks

- Regular Neural Nets don’t scale well to full images.
- For example, an image of size, 200x200x3, would lead to neurons having 200*200*3 = 120,000 weights.
- As number of layers increase, the parameters would add up quickly!
- Therefore, full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

Convolution

- If you have any prior experience with computer vision, you’ve already applied convolutions
    - Apply blurring or smoothing to an image: Convolution.
    - Edge detection: Convolution
    - Sharpen an image: Convolution.
- Convolutions are one of the most critical, fundamental building-blocks in computer vision and image processing.

Convolution

- In terms of deep learning, an (image) convolution is an ***element-wise multiplication of two matrices followed by a sum***.

- What a ***convolution*** is:
  1. Take two matrices (which *both have the same dimensions*)
  2. Multiply them, element-by-element (*not the dot product, just a simple multiplication*)
  3. Sum the elements together

The “Big Matrix” and “Tiny Matrix" Analogy

- An image is a matrix with a width (# of columns) and height (# of rows), depth (# of channels)

- We can think of an image as big matrix and a *kernel or convolutional matrix* as a tiny matrix.

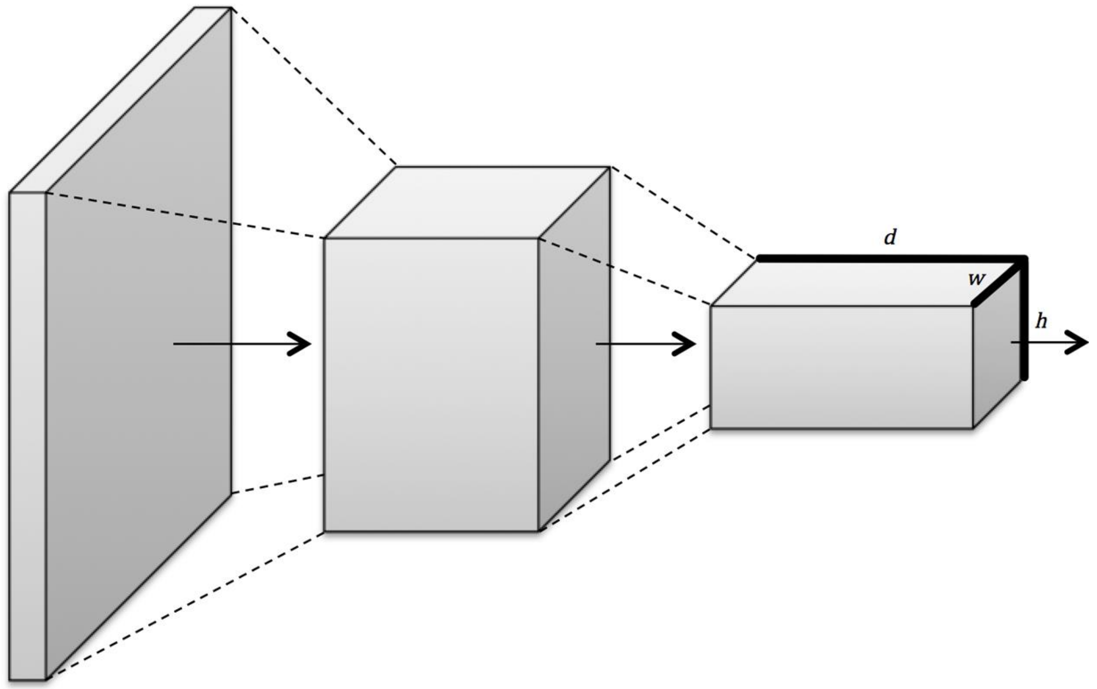
- This tiny matrix sits on top of the big image and slides from left-to-right and top-to-bottom, applying convolution at each (x, y)-coordinate of the original image.

Convolutional Layer

- The convolutional network takes advantage of the fact that we're analyzing images: Which remains static
- Inspired by how human vision works, layers of a convolutional network have neurons arranged in three dimensions, having a width, height, and depth.
- Neurons in a convolutional layer are connected to a small, local region of the preceding layer, to avoid the wastefulness of fully-connected neurons.

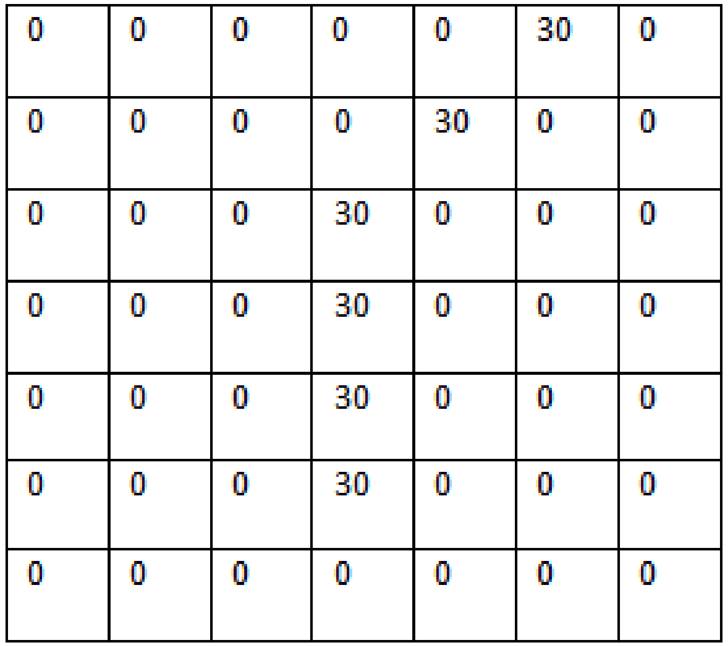
Convolutional Layer

- Convolutional layers arrange neurons in three dimensions, so layers have width, height, and depth



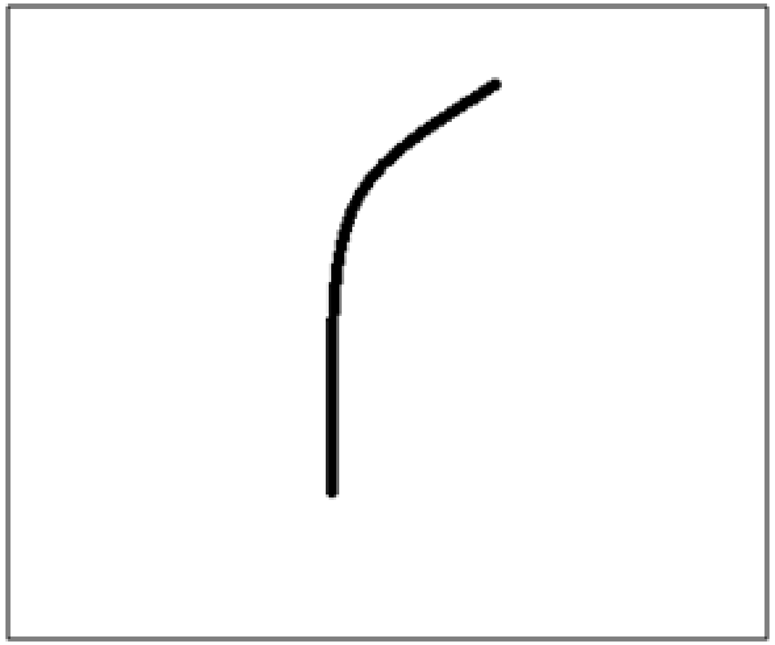
Filters and Feature Maps

- The first concept that arose was that of a ***filter***
- It turns out that here, Viola and Jones were actually pretty close.
- A ***filter*** is essentially a feature detector



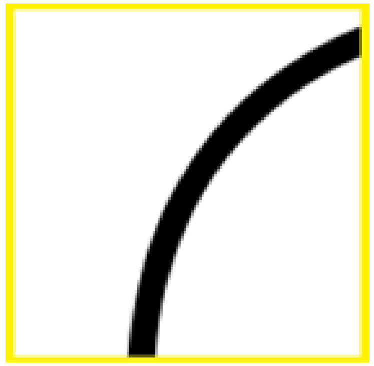
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter



Visualization of a curve detector filter

Filters – High Level Perspective



Visualization of the receptive field

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

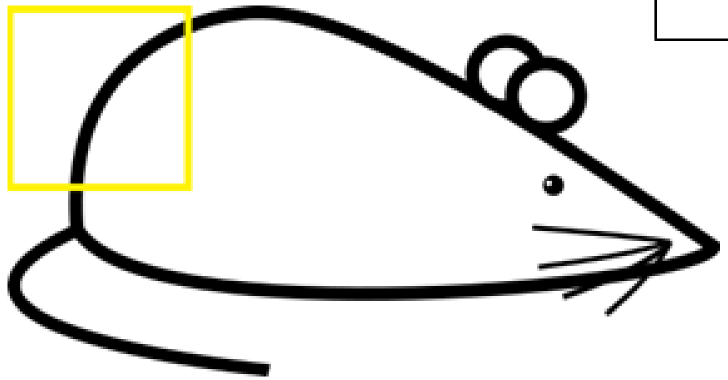
Pixel representation of the receptive field

*

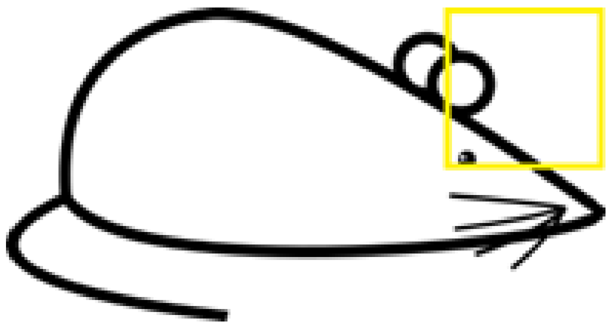
0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)



Filters – High Level Perspective



Visualization of the filter on the image

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

Pixel representation of receptive field

$*$

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Pixel representation of filter

Multiplication and Summation = 0

Kernels

- Let’s think of an *image as a big matrix* and a *kernel as a tiny matrix*
- kernel looks like:

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- Most kernels applied to deep learning and CNNs are NxNsquare matrices
- This takes advantage of optimized linear algebra libraries that operate most efficiently on square matrices.

Kernels

- Use odd kernel size to ensure there is a valid integer (x,y)-coordinate at the center of the image.

A Hand Computation Example of Convolution

- Convolution requires three components
    1. An input image
    2. A kernel matrix that we are going to apply to the input image
    3. An output image to store the output of the image convolved with the kernel

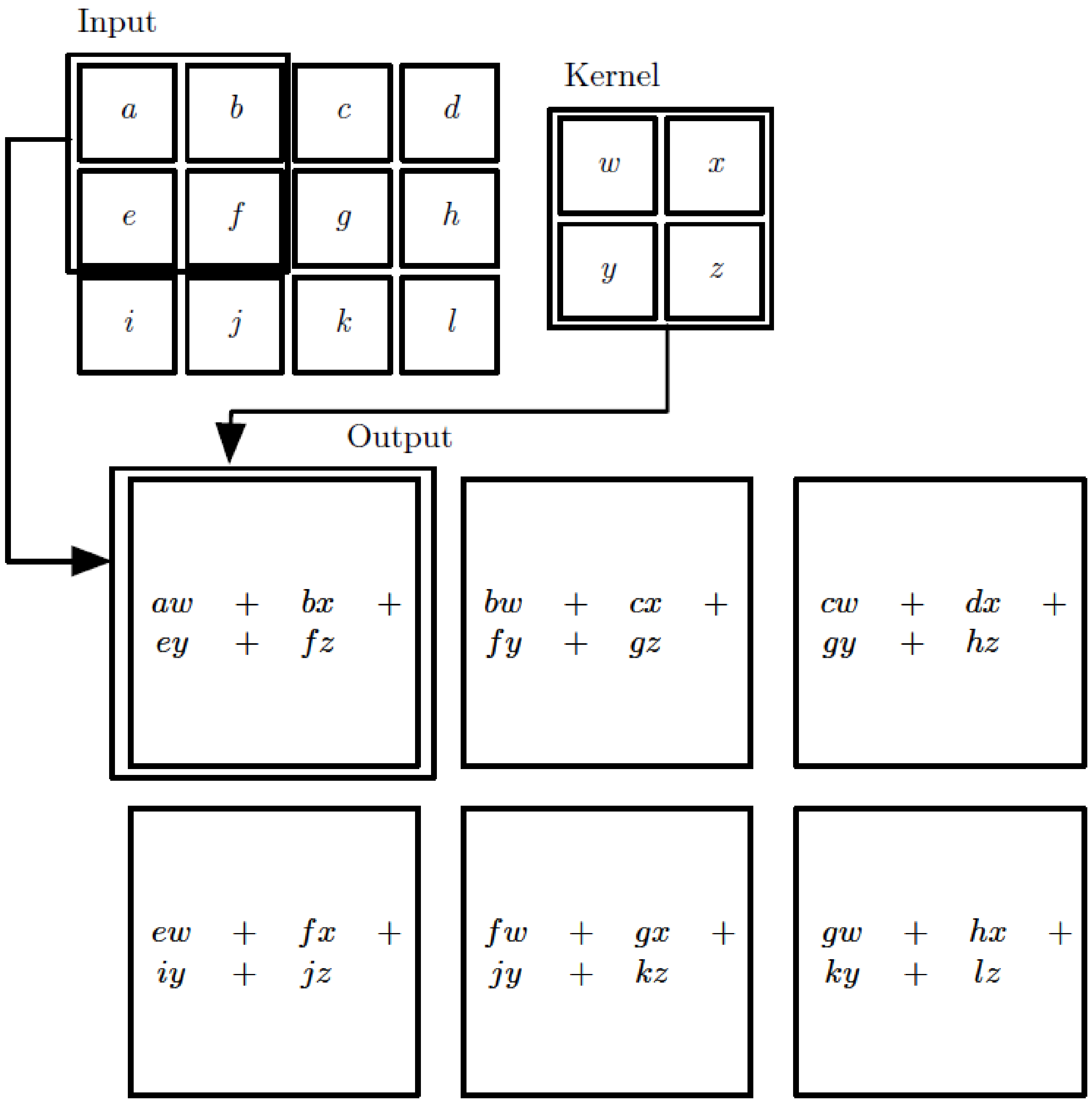
A Hand Computation Example of Convolution

- Example of convolving (denoted mathematically as the * operator) a 3x3 region of an image with a 3x3 kernel used for blurring:

$$O_{i,j} = \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \star \begin{bmatrix}93 & 139 & 101\\ 26 & 252 & 196\\ 135 & 230 & 18\end{bmatrix} = \begin{bmatrix}1/9\times 93 & 1/9\times 139 & 1/9\times 101\\ 1/9\times 26 & 1/9\times 252 & 1/9\times 196\\ 1/9\times 135 & 1/9\times 230 & 1/9\times 18\end{bmatrix}$$

$$O_{i,j} = \sum\begin{bmatrix}10.3 & 15.4 & 11.2\\ 2.8 & 28.0 & 21.7\\ 15.0 & 25.5 & 2.0\end{bmatrix} \approx 132.$$

- We set the pixel located at the coordinate (i,j) of the output image O to 132


Input
a
b
c
d
e
f
g
h
i
j
k
l
Kernel
w
x
y
z
Output
$aw + bx + ey + fz$
$bw + cx + fy + gz$
$cw + dx + gy + hz$
$ew + fx + iy + jz$
$fw + gx + jy + kz$
$gw + hx + ky + lz$

Layer Types

- Many types of layers used to build Convolutional Neural Networks:
  - Convolutional (CONV)
  - Activation (ACT or RELU, where we use the same of the actual activation function)
  - Pooling (POOL)
  - Fully-connected (FC)
  - Batch normalization (BN)
  - Dropout (DO)

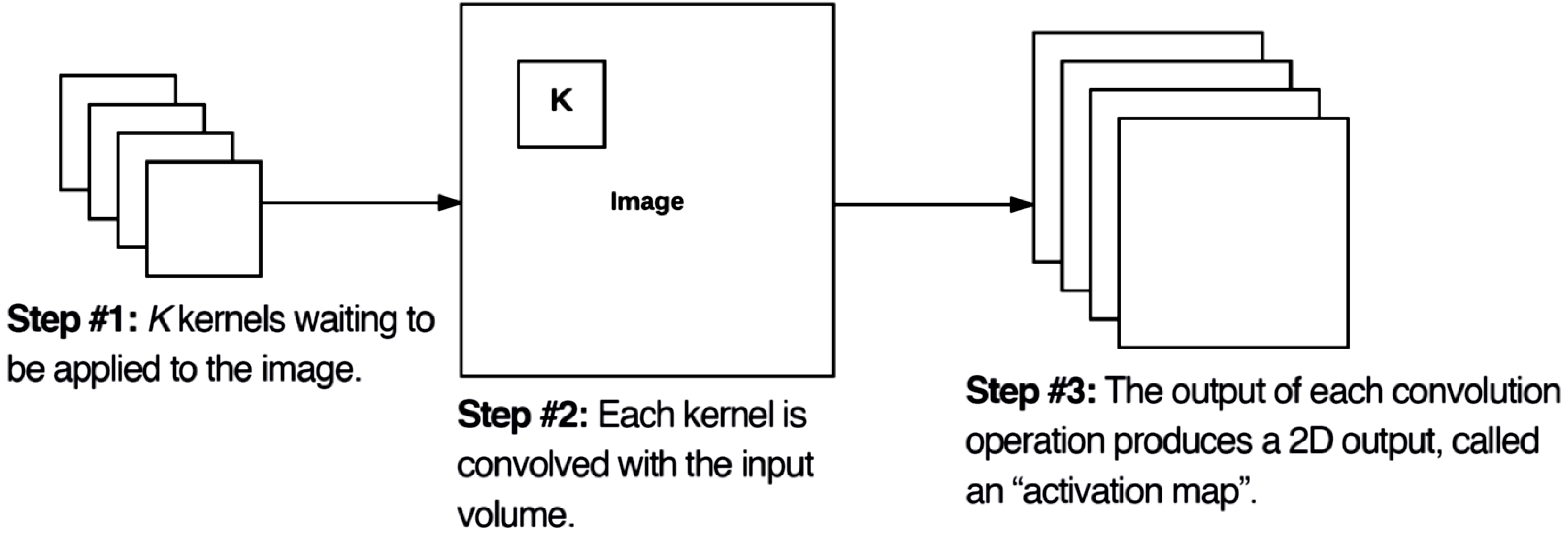
Convolutional Layers

- The CONV layer is the core building block of a Convolutional Neural Network.

- The CONV layer parameters consist of a set of K learnable filters (i.e., “kernels”), where each filter has a width and a height, and are nearly always square.

- For inputs to the CNN, the depth is the number of channels in the image (i.e., a depth of three when working with RGB images, one for each channel).

Convolutional Layers


K
Image
Step #1: *K* kernels waiting to be applied to the image.
Step #2: Each kernel is convolved with the input volume.
Step #3: The output of each convolution operation produces a 2D output, called an "activation map".

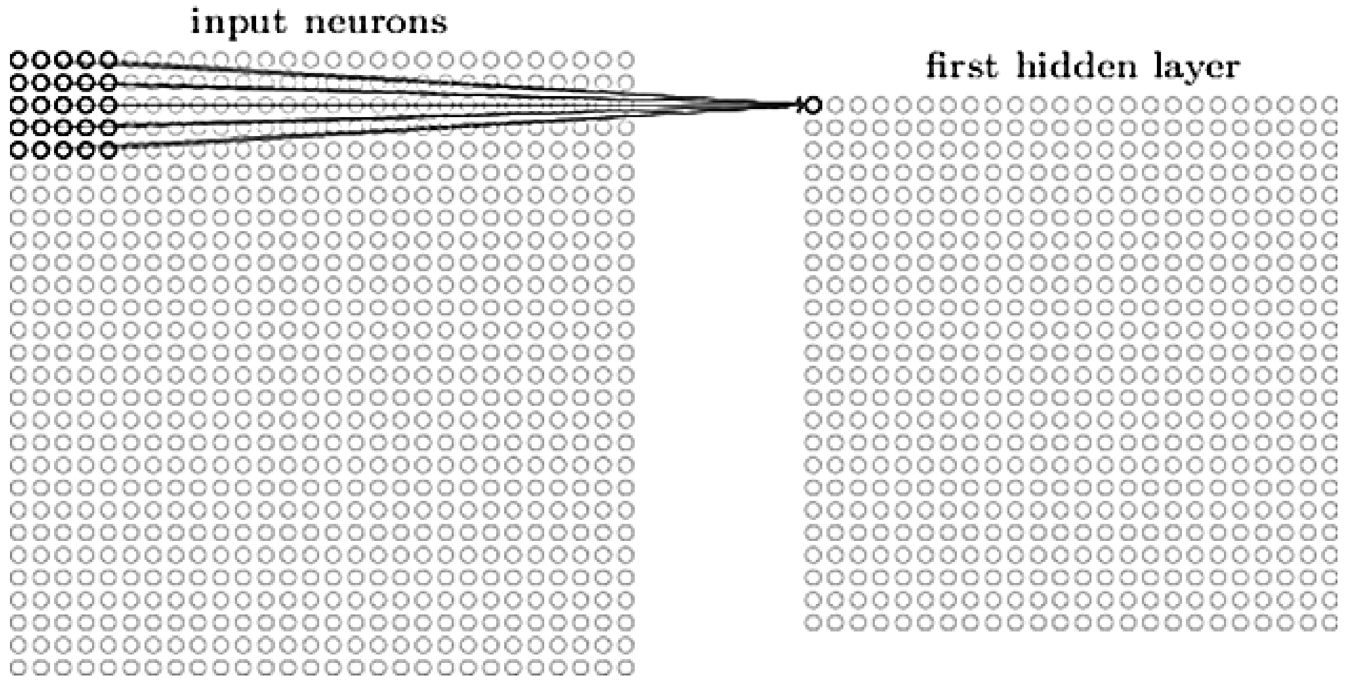
Left: At each convolutional layer in a CNN, there are K kernels applied to the input volume.
Middle: Each of the K kernels is convolved with the input volume.
Right: Each kernel produces an 2D output, called an activation/feature map.

CNN Building Blocks

- In CNNs, we choose to connect each neuron to only a local region of the input volume – we call the size of this ***local region*** as the ***receptive field*** of the neuron.

Local Connectivi

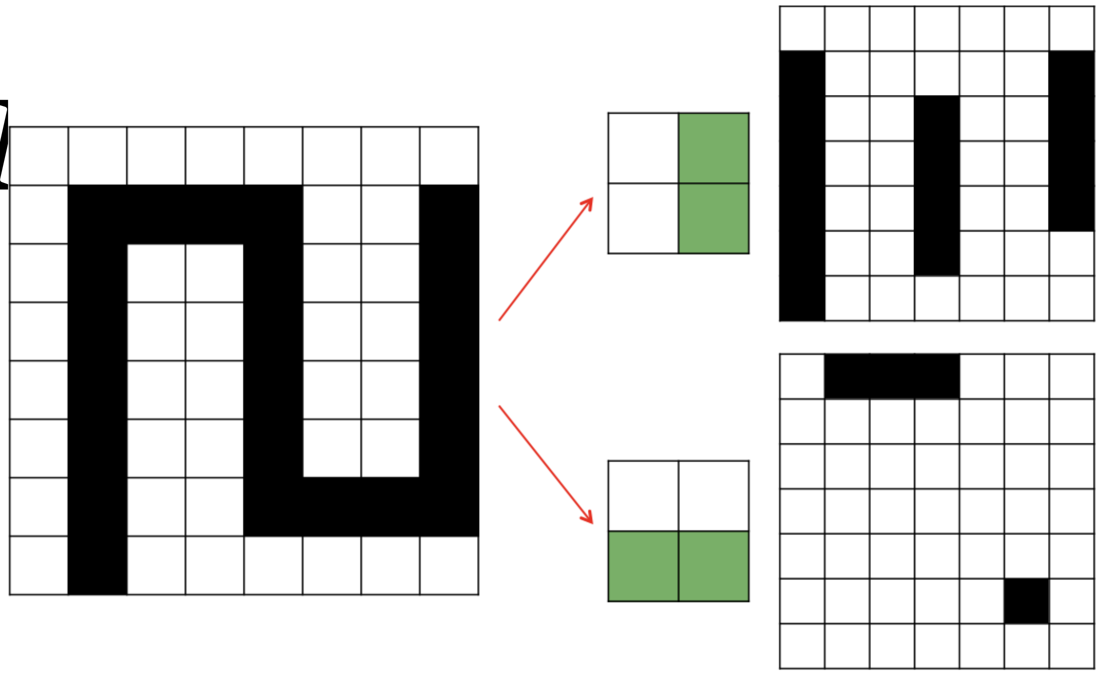
- We connect each neuron to onl local region of the input volum
- The extent of the connectivity along the depth axis is always equal to the depth of the input volume
- The *spatial extent* of this connectivity is a *hyperparameter* called the *receptive field* of the neuron (equivalently this is the *filter size*).



Visualization of 5 x 5 filter convolving around an input volume and producing an activation map

Filters and Feature

- To detect vertical lines, we use the feature detector on the top
- Slide it across the entirety of the image, and check if we have a match.

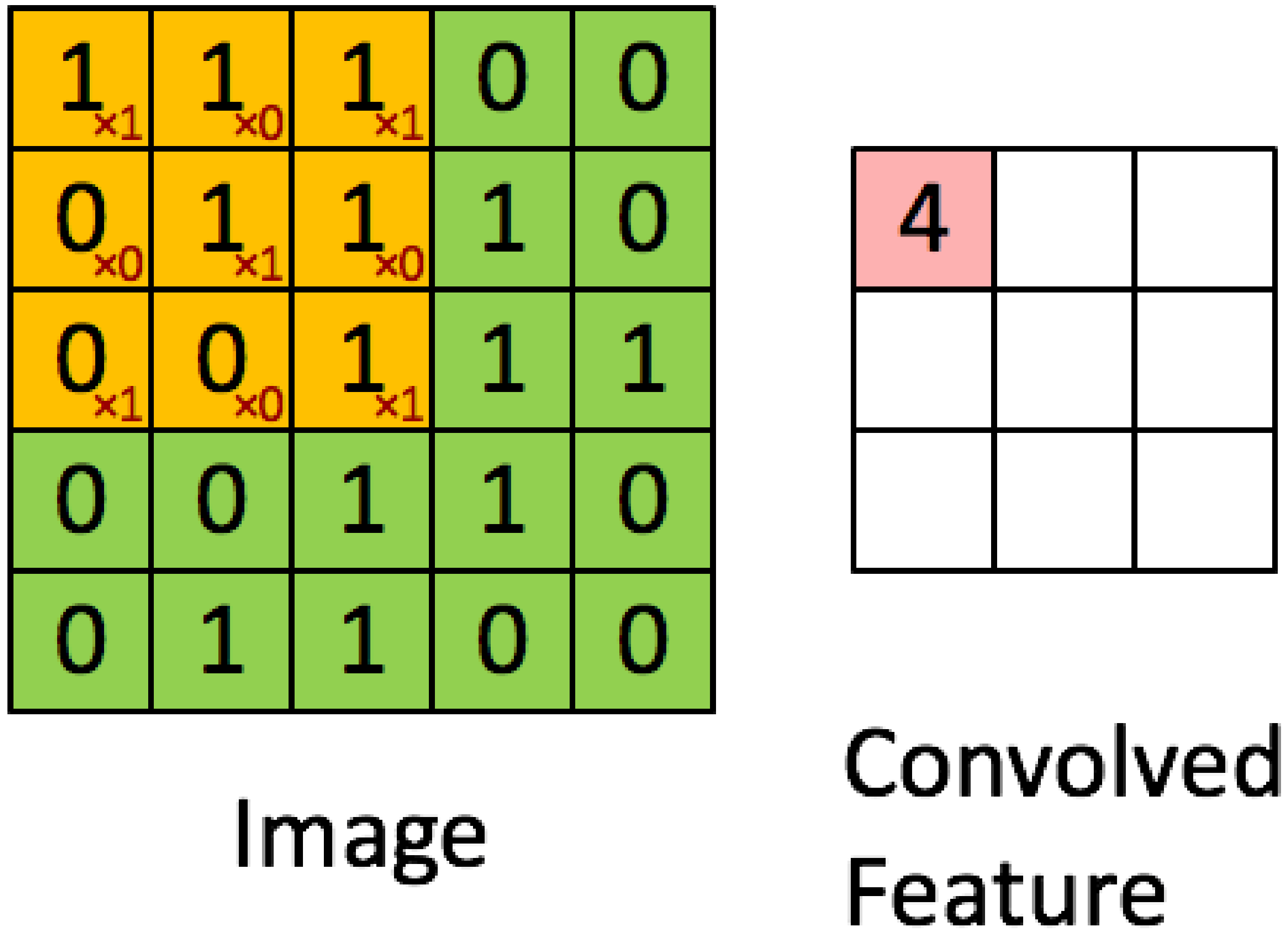


- We keep track of our answers in the matrix in the top right.
- If there's a match, we shade the appropriate box black.
- If there isn't, we leave it white.

- This ***result is our feature map***

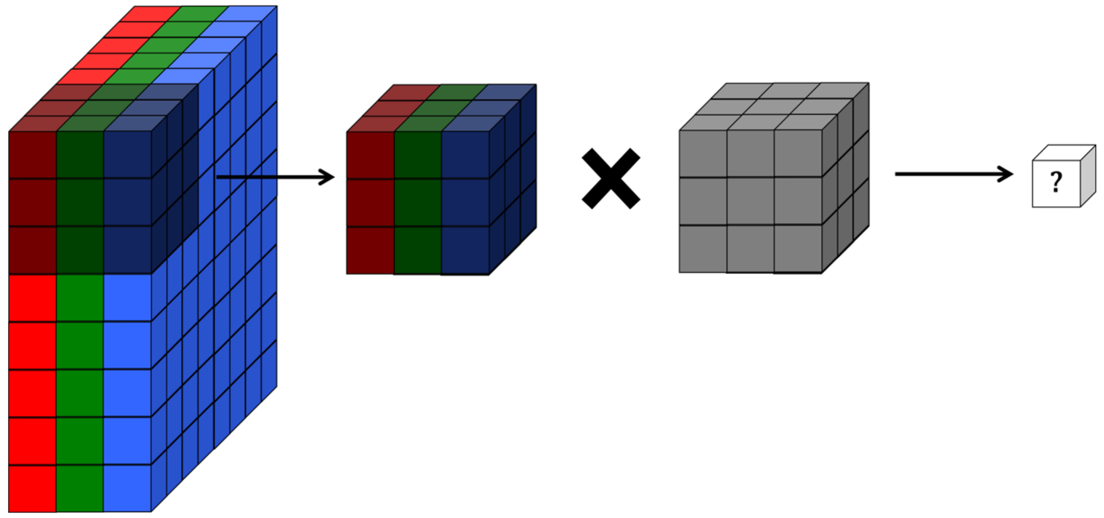
Filters and Feature Maps

- This operation is called a convolution.
- We take a ***filter*** and we multiply it over the entire area of an input image.
- In this scheme, layers of neurons in a feedforward neural net represent either the original image or a feature map.
- ***Filters represent combinations of connections that get replicated*** across the entirety of the input


1 ×1
1 ×0
1 ×1
0
0
0 ×0
1 ×1
1 ×0
1
0
0 ×1
0 ×0
1 ×1
1
1
0
0
1
1
0
0
1
1
0
0
Image
4
Convolved Feature

Local Connectivity

- It is important to emphasize again this *asymmetry in how we treat the spatial dimensions* (*width and height*) and the *depth* dimension



- The connections are local in space (along width and height), but always full along the entire depth of the input volume.

Filters and Feature Maps

- ***Example 1:*** Suppose that the input volume has size [32x32x3].

- If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter).

- The extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Filters and Feature Maps

- ***Example 2:*** Suppose an input volume had size [16x16x20].
- Using a receptive field size of 3x3, every neuron in the Conv Layer would have a total of 3*3*20 = 180 connections to the input volume.
- Notice that, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

Spatial arrangement

- There are three hyperparameters that control the size of the output volume:
  - **Spatial extent**
  - **Depth**
  - **Stride**
  - **Padding**

Spatial extent

- The ***spatial extent*** is a ***hyperparameter*** called the ***receptive field*** of the neuron (equivalently this is the ***filter size***).

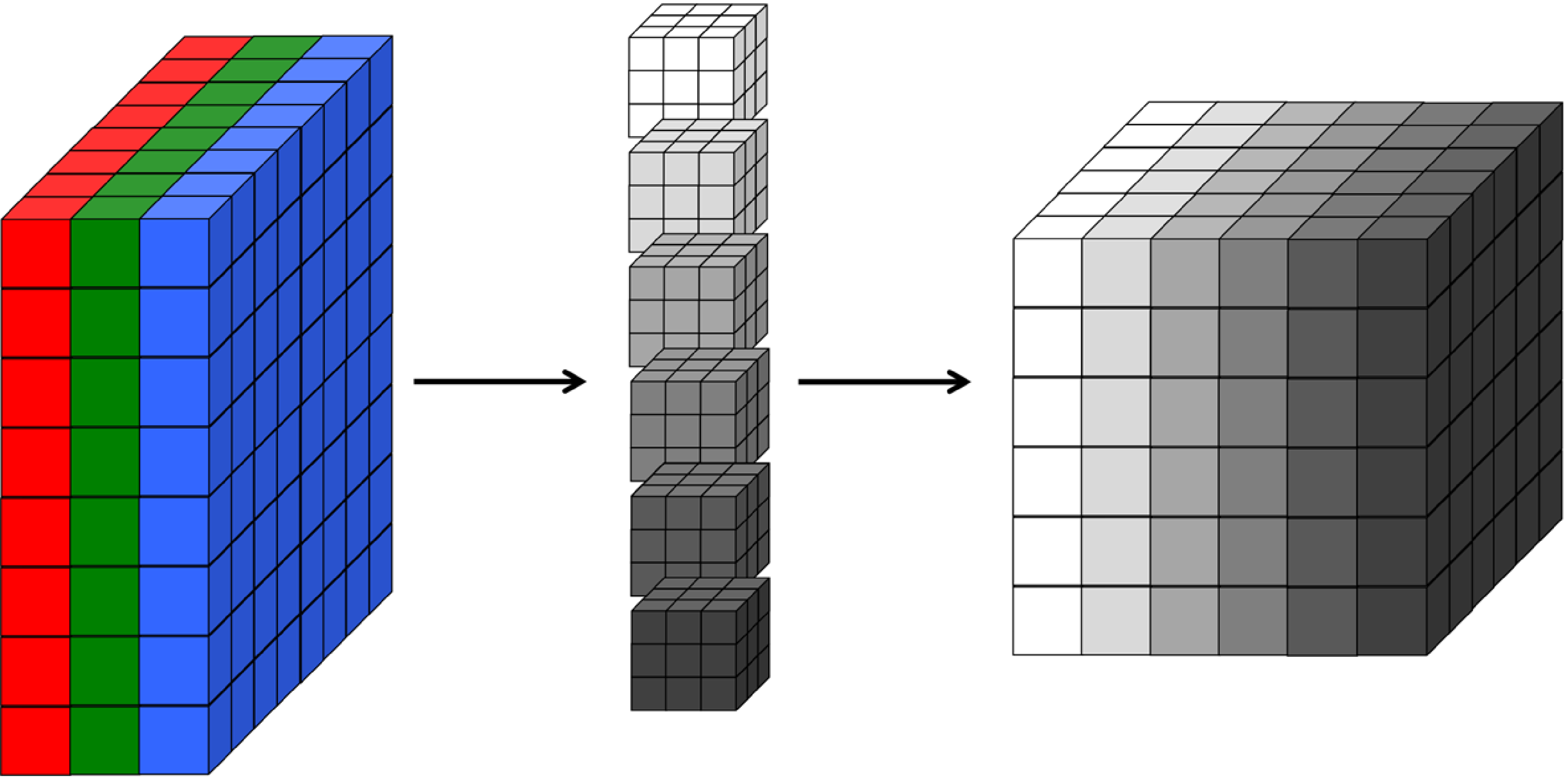
- The *spatial extent e*, is equal to the filter’s height and width.

Depth

- Two types of Depth:
  - Input Depth
  - Output Depth

- Input Depth: Number of color channels
- Output Depth: Depth of an output volume corresponds to the number of filters used.

The set of neurons that are all looking at the same region of the input are referred to as a ***depth column***



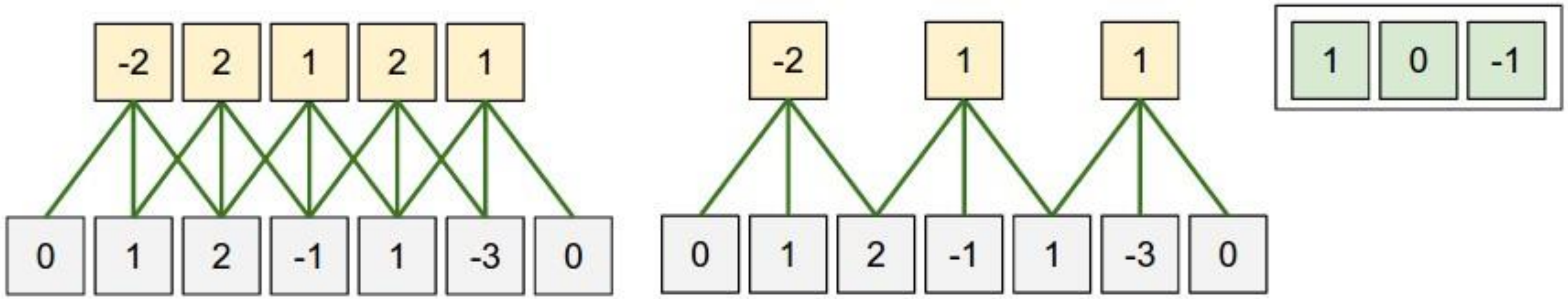
each filter corresponds to a slice in the resulting output volume

Stride

- **Stride:** The number of pixels with which we slide the filter.
- This description is similar to a sliding window that slides from left-to-right and top-to-bottom across an image

Stride

- When the stride is 1 then we move the filters *one pixel at a time*.
- When the stride is 2 then filters jump *2 pixels at a time* as we slide them around.
- This will produce smaller output volumes spatially



Spatial size of the output volume

- The spatial size of the output volume is given by:

$$\frac{(W - F + 2P)}{S} + 1$$

- Where,
    - *W: input volume size*
    - *F: receptive field size*
    - *S: stride*
    - *P: amount of zero padding*

Constraints on strides

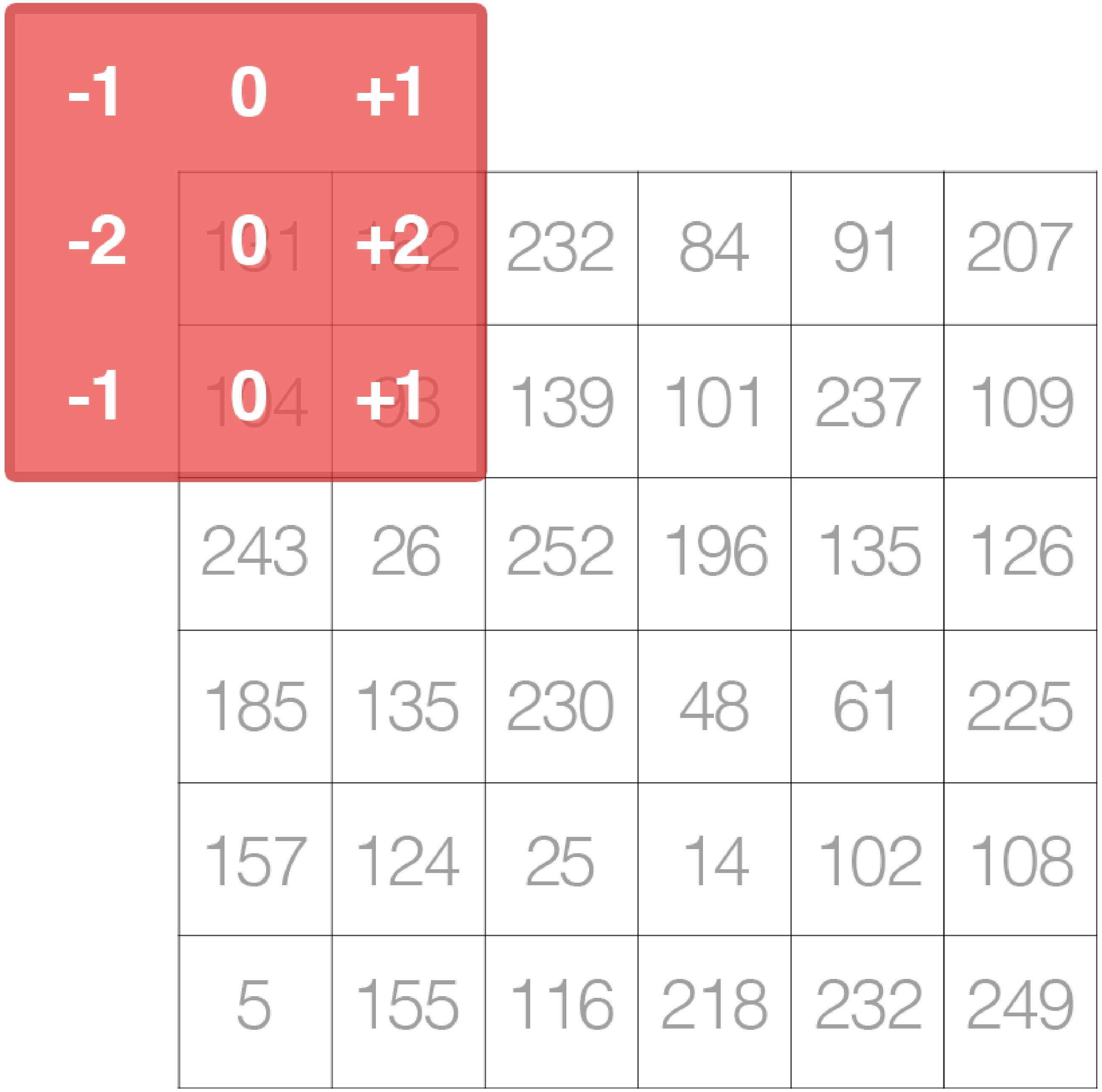
- Note that the spatial arrangement hyperparameters have mutual constraints.
- When the input has size $W = 10$, no zero-padding is used $P = 0$, and the filter size is $F = 3$, then it would be impossible to use stride $S = 2$.

$$\frac{(W - F + 2P)}{S} + 1 = \frac{10 - 3 + 0}{2} + 1 = 4.5$$

- 4.5 is not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input.

Zero-Padding

- Using zero-padding, we can "pad" our input along the borders such that our output volume size matches our input volume size.



Full Description of the Convolutional Layer

- A convolutional layer takes in an input volume.

- This input volume has the following characteristics:
  - *Width* w_{in}
  - *Height* h_{in}
  - *Depth* d_{in}
  - *Zero padding* p

Full Description of the Convolutional Layer

- This volume is processed by a ***total of k filters***, which represent the weights and connections in the convolutional network.
- Filters have ***hyperparameters*** described as follows:
  - Their ***spatial extent e***, which is equal to the filter's height and width.
  - Their ***stride s***,
  - The ***bias b***

Full Description of the Convolutional Layer

This results in an output volume with the following characteristics:

- Its activation function f, applied to the incoming logit of each neuron in the output volume
- Its *width:*

$$w_{out} = \left\lceil \frac{w_{in} - e + 2p}{s} \right\rceil + 1$$

- Its *height:*

$$h_{out} = \left\lceil \frac{h_{in} - e + 2p}{s} \right\rceil + 1$$

Full Description of the Convolutional Layer

- The m^{th} "depth slice" of the output volume, where $1 \leq m \leq k$, corresponds to the function f applied to the sum of the m^{th} filter convoluted over the input volume and the bias b_m.

- This means that per filter, we have $d_{in}e^2$ parameters.

- In total, that means the layer has $kd_{in}e^2$ parameters and k biases.

Parameter Sharing: Example

- The Krizhevsky et al. architecture accepted images of size $[227 \times 227 \times 3]$.
- On the first Convolutional Layer, it used neurons with receptive field size $e = 11$, stride $s = 4$ and no zero padding $p = 0$.
- Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of K=96, the Conv layer output volume had size $[55 \times 55 \times 96]$.

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks.” Advances in Neural Information Processing Systems 25 (NIPS 2012).

Parameter Sharing: Example

- Each of the [55 × 55 × 96] neurons in this volume was connected to a region of size [11 × 11 × 3] in the input volume.
- Using the real-world example above, we see that there are 55 x 55 x 96 = 290,400 neurons in the first Conv Layer, and each has 11 x 11 x 3 = 363 weights and 1 bias.
- Altogether, **290400 x 364 = 105,705,600** parameters on the first layer of the ConvNet.

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." Advances in Neural Information Processing Systems 25 (NIPS 2012).

Parameter Sharing

- We can reduce the number of parameters by making one reasonable assumption:
  - If one feature is useful to compute at some spatial position (x_1,y_1), then it should also be useful to compute at a different position (x_2,y_2).
- Thus, denoting a single 2-dimensional slice of depth as a **depth slice** we are going to constrain the neurons in each depth slice to use the same weights and bias.

Parameter Sharing: Example

- With parameter sharing scheme, the first Conv Layer would now have only 96 unique set of weights (one for each depth slice)
- A total of $96 \times 11 \times 11 \times 3 = 34848$ unique weights, or 34,944 parameters (+96 biases).
- Alternatively, all 55x55 neurons in each depth slice will now be using the same parameters.

Example Demo

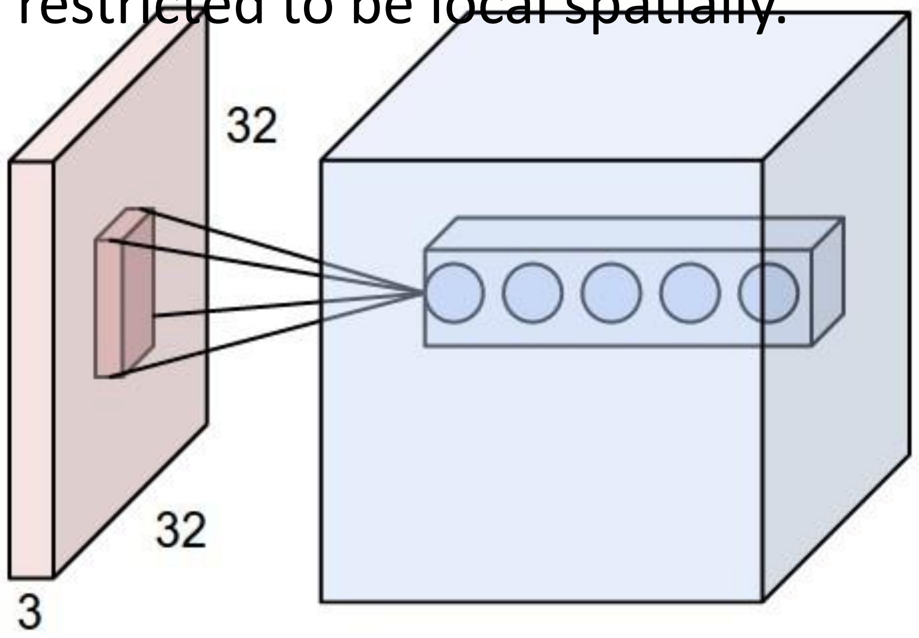
(http://cs231n.github.io/assets/conv-demo/index.html)

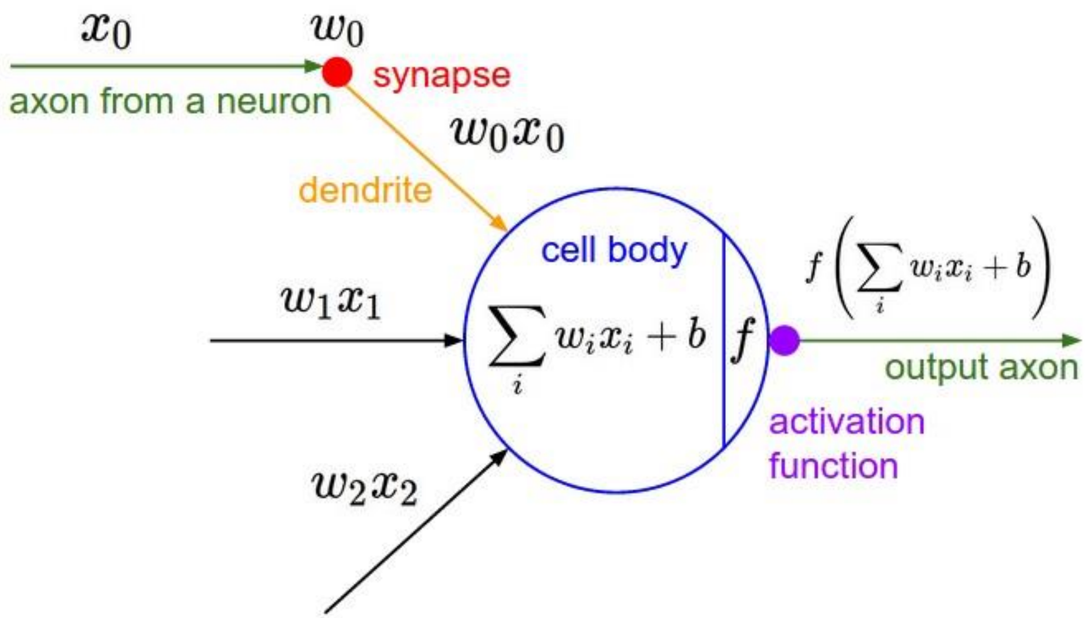
Parameter Sharing

- Sometimes the parameter sharing assumption may not make sense.
- One practical example is when the input are faces that have been centered in the image.
- One might expect that different eye-specific or hair-specific features should be learned in different spatial locations.
- In that case it is common to relax the parameter sharing scheme, and simply call the layer a ***Locally-Connected Layer***.

Left: Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels).

Right: The feed-forward mechanism remains unchanged: compute a dot product of weights with the input followed by a non-linearity, but the connectivity is now restricted to be local spatially.





Activation Layers

- After each CONV layer in a CNN, a nonlinear activation function is applied, such as ReLU, ELU, or any of the other Leaky ReLU variants.
- Activation layers are not technically “layers” and are sometimes omitted from network architecture diagrams as it’s assumed that an activation immediately follows a convolution.
- As an example, consider the following network architecture: INPUT => CONV => RELU => FC.

Pooling Layers

- It is common to insert POOL layers in-between consecutive CONV layers in a CNN architectures:

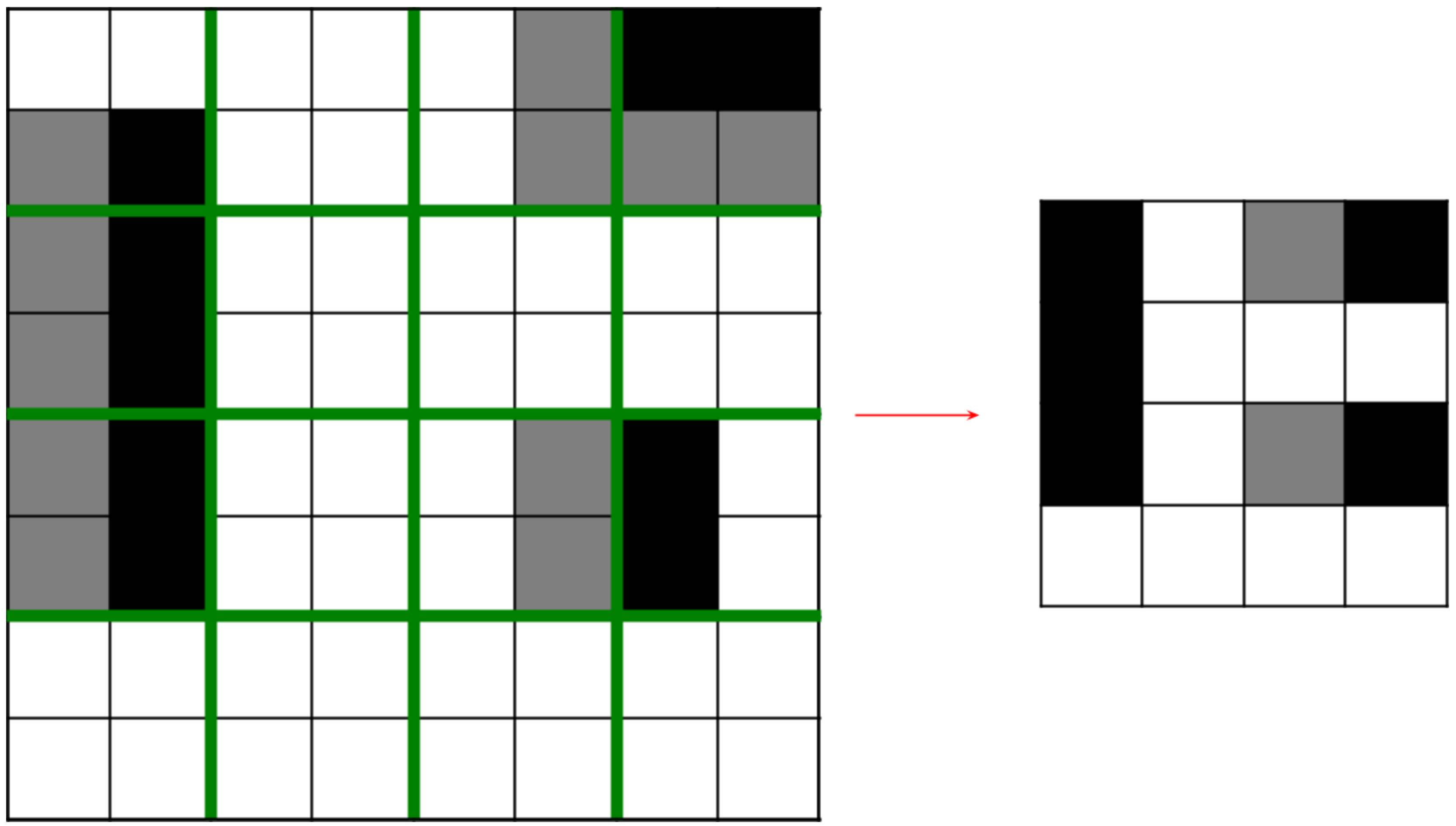
INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC

- The primary function of the POOL layer is to progressively reduce the spatial size (i.e., width and height) of the input volume.
- This helps reduce the number of parameters in the network
- Pooling also helps in controlling overfitting.

Pooling Layers

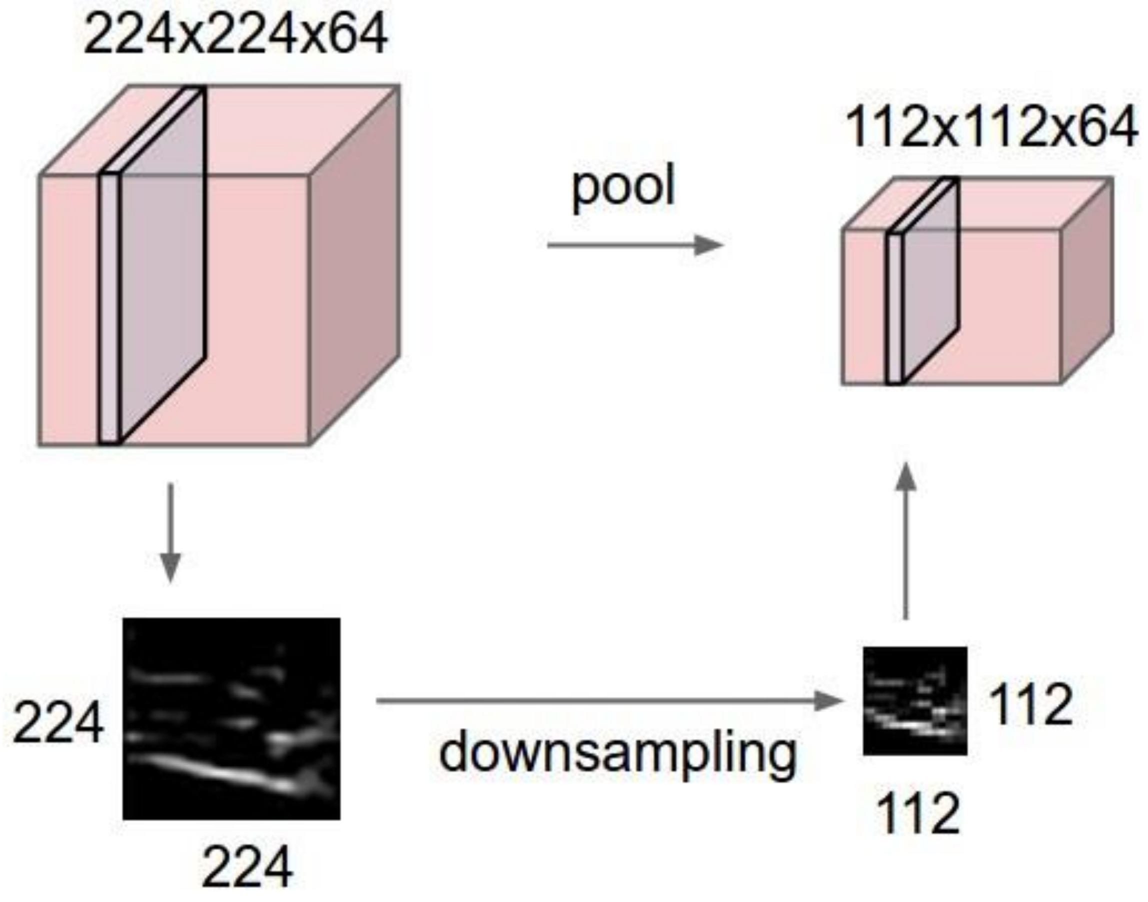
- POOL layers operate on each of the depth slices of an input independently using either the max or average function.

- Max pooling is typically done in the middle of the CNN architecture to reduce spatial size, whereas average pooling is normally used as the final layer of the network (e.x., GoogLeNet, SqueezeNet, ResNet) where we wish to avoid using FC layers entirely.

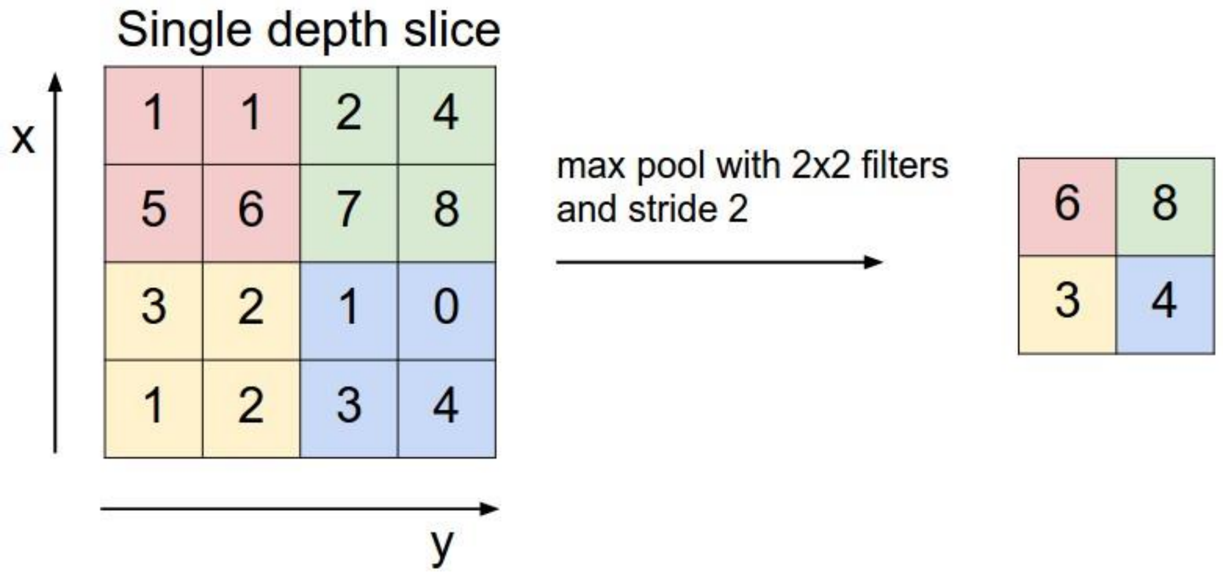


Create a cell for each tile, compute the maximum value in the tile, and propagate this maximum value into the corresponding cell of the condensed feature map

Max Pooling



224x224x64
pool
112x112x64
224
224
downsampling
112
112



Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters and stride 2
6 8
3 4

Max Pooling

- POOL layers Accept an input volume of size
$$W_{input} \times H_{input} \times D_{input}$$
- They then require two parameters:
  - The receptive field size F (also called the “pool size”).
  - The stride S.

Max Pooling

- Applying the POOL operation yields an output volume of size

$$W_{output} \times H_{output} \times D_{output}$$

Where:

$$W_{output} = \left(\frac{W_{input} - F}{S}\right) + 1$$

$$H_{output} = \left(\frac{H_{input} - F}{S}\right) + 1$$

$$D_{output} = D_{input}$$

Max Pooling

In practice, we tend to see two types of max pooling variations:

1. $F = 3$; $S = 2$ which is called overlapping pooling and normally applied to images/input volumes with large spatial dimensions.
2. $F = 2$; $S = 2$ which is called non-overlapping pooling. This is the most common type of pooling and is applied to images with smaller spatial dimensions.

Fully-connected Layers

- Neurons in FC layers are fully-connected to all activations in the previous layer
- FC layers are always placed at the end of the network
- It’s common to use one or two FC layers prior to applying the softmax classifier

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC

Getting rid of pooling

- Many people dislike the pooling operation
- Several researchers propose to discard the pooling layer in favor of architecture that only consists of repeated CONV layers.
- To reduce the size of the representation they suggest using larger stride in CONV layer once in a while.
- Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs).

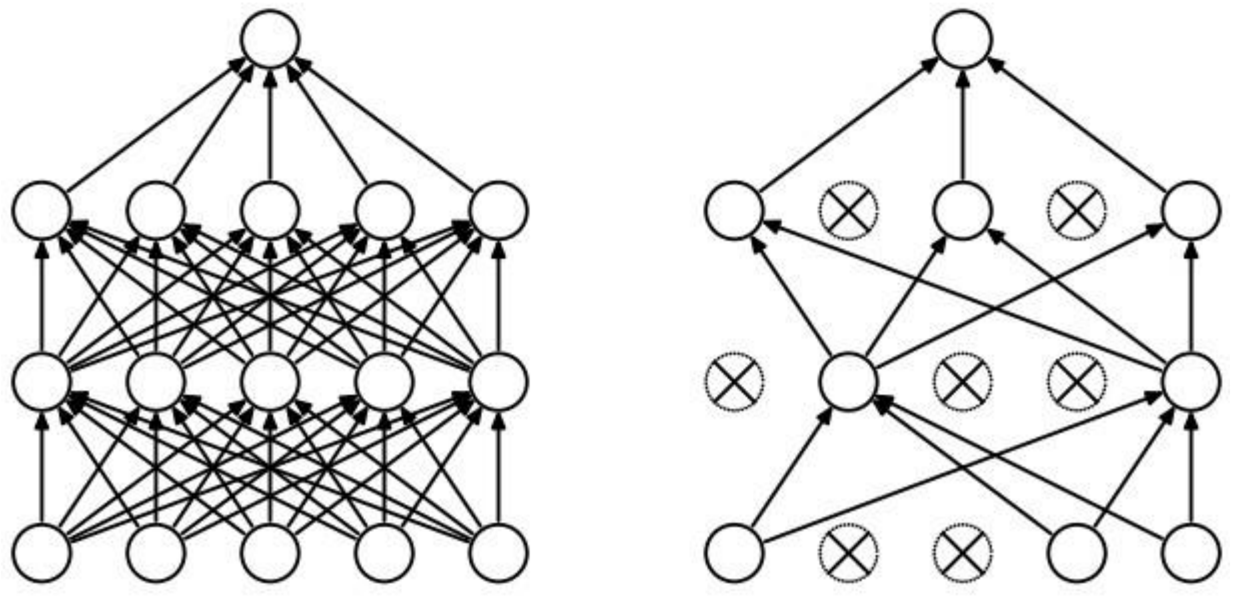
Dropout

- *"dropout" refers to dropping out units (both hidden and visible) in a neural network*
- Dropout is a form of regularization
- Aims to help prevent overfitting by increasing testing accuracy,
- For each mini-batch in the training set, dropout layers, with probability p, ***randomly disconnect inputs from the preceding layer to the next layer*** in the network architecture

Dropout

- We randomly disconnect with probability p=0.5

After the forward and backward pass are computed for the minibatch, we re-connect the dropped connections, and then sample another set of connections to drop.


(a) Standard Neural Net (b) After applying dropout.

Dropout

- Dropout is applied reduce overfitting by altering the network architecture at training time.
- Randomly dropping connections ensures that no single node in the network is responsible for “activating” when presented with a given pattern.
- Instead, dropout ensures there are multiple, redundant nodes that will activate when presented with similar inputs – this helps our model to generalize.

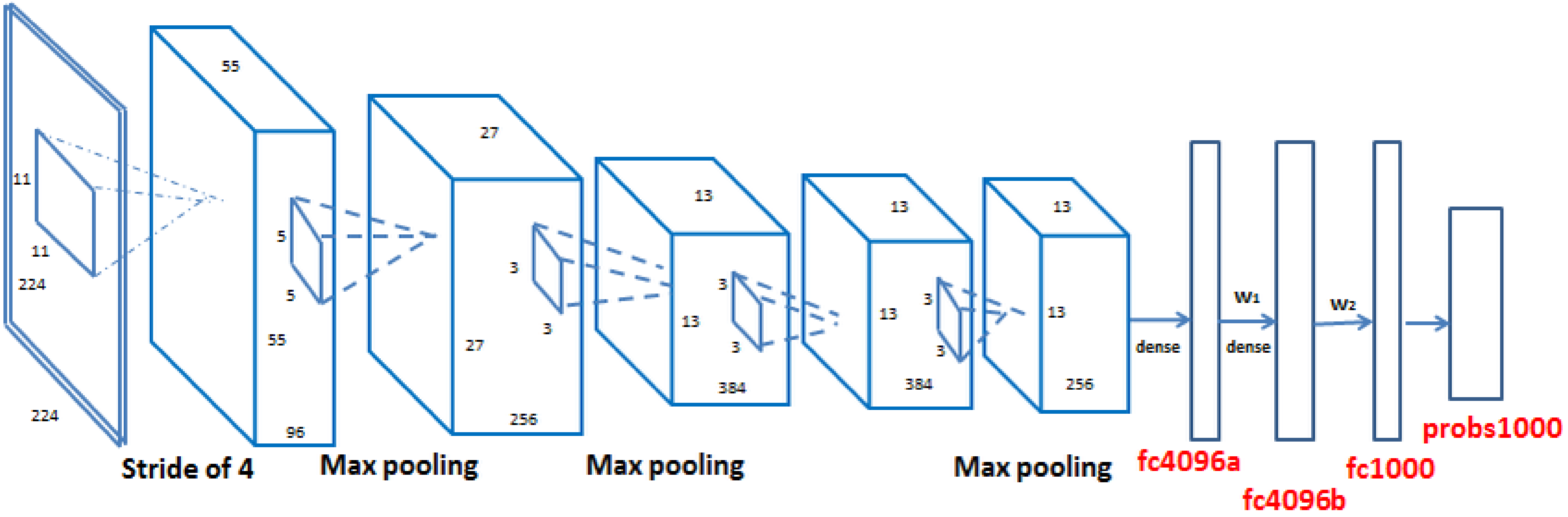
Dropout

- It is most common to place dropout layers with p = 0.5 in-between FC layers of an architecture
- where the final FC layer is assumed to be the softmax classifier:
- … CONV => RELU => POOL => FC => DO => FC => DO => FC
- However, we may also apply dropout with smaller probabilities (i.e., p = 0.10-0.25) in earlier layers of the network as well

Layer Patterns

- The most common form of CNN architecture is to stack a few CONV and RELU layers, following them with a POOL operation.
- Repeat this sequence until the volume width and height is small, at which point we apply one or more FC layers.
- INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC
- Common choices for each reputation include:
  - 0 <= N <= 3
  - M >= 0
  - 0 <= K <= 2

AlexNet consists of **5 Convolutional Layers** and **3 Fully Connected Layers**



Layer Patterns

- AlexNet-like [94] CNN architecture which has multiple CONV =>RELU => POOL layer sets, followed by FC layers:

INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU] * 3 => POOL =>[FC => RELU => DO] * 2 => SOFTMAX

- For deeper network architectures, such as VGGNet [95], we'll stack two (or more) layers before every POOL layer

INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU] * 2 => POOL =>[CONV => RELU] * 3 => POOL => [CONV => RELU] * 3 => POOL =>[FC => RELU => DO] * 2 => SOFTMAX

Layer Patterns

- Generally, we apply deeper network architectures when we
  1. have lots of labeled training data and
  2. the classification problem is sufficiently challenging.

- Stacking multiple CONV layers before applying a POOL layer allows the CONV layers to develop more complex features before the destructive pooling operation is performed.

Case studies

- **LeNet**. The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's.
- **AlexNet**. The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton.
- **ZF Net**. The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net).
- **GoogLeNet**. The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google.
- **VGGNet**. The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet.
- **ResNet**. Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015.

Activation Function

- An activation function is the function that takes the combined input z, applies a function on it, and passes the output value.
- The activation function, determines the state of a neuron by indicating a trigger or no-trigger.

why do we need an activation function

Can't we just pass the value of z as the final output?

- Firstly, the range of the output value would be –Infinity to + Infinity, where we won't have a clear way of defining a threshold where activation should happen.
- Secondly, the network will in a way be rendered useless, as it won't really learn.

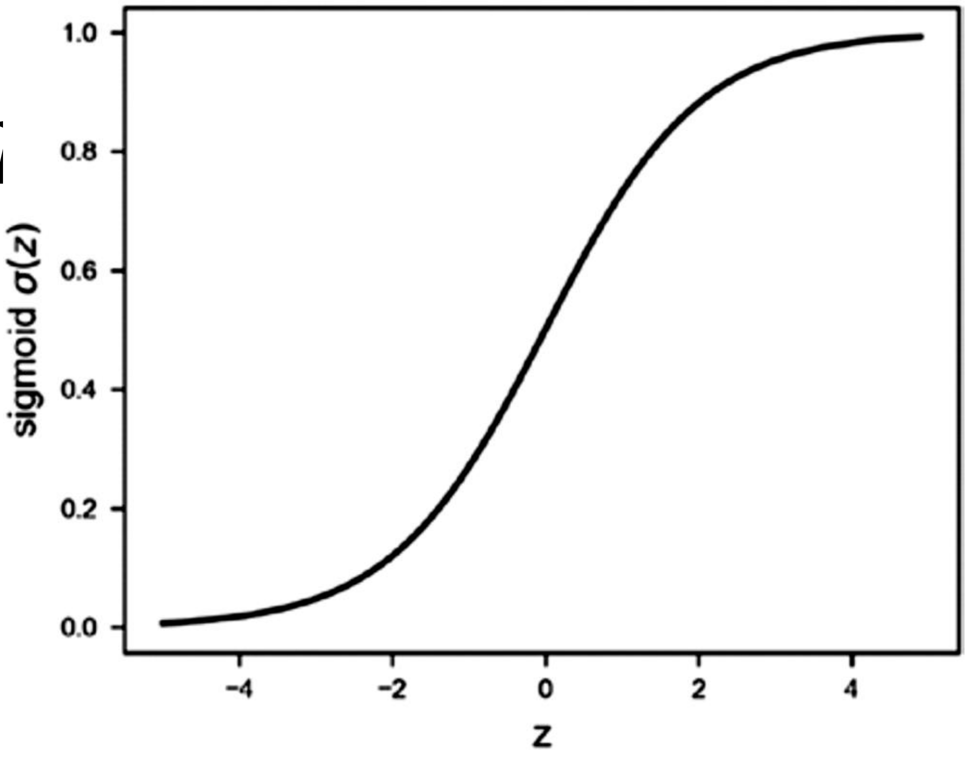
why do we need an activation function

- If activation function is a linear function (basically no activation), then the derivative of that function becomes 0;
- Backpropagation algorithm gives feedback to the network about wrong classifications and thereby helps a neuron to adjust its weights by using a derivative of the function.
- If derivative becomes 0, the network loses out on this learning ability.

why do we need an activation function

- To put it another way, we can say there is really no point of having the DNN, as the output of having just one layer would be similar to having n layers.
- To keep things simple, we would always need a nonlinear activation function (at least in all hidden layers) to get the network to learn properly.

Sigmoid Activation Fu



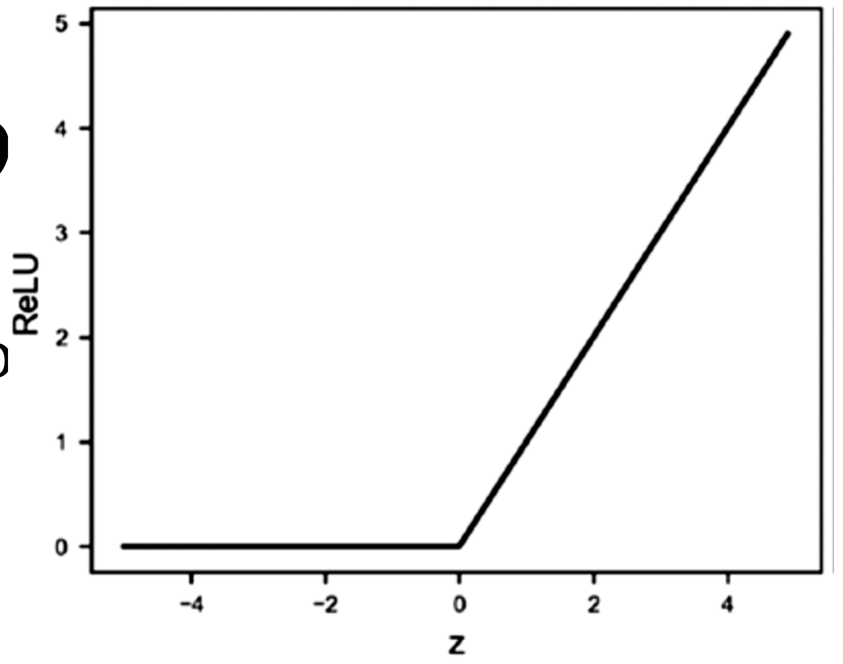
- A sigmoid function is defined as:

$$\sigma = \frac{1}{1 + e^{-z}}$$

- It renders the output between 0 and 1

- It resembles the principle:
  - *lower influence: low output*
  - *higher influence: higher output*

ReLU Activation Functio



- ReLU(Rectified Linear Unit) uses the functio

$$\boldsymbol{f(z) = max(0, z)}$$

- If the output is positive it produces the same value, otherwise it produces 0
- ReLU is a valid nonlinear function and works really well as an activation function
- A direct result of 0 when z is negative, thereby deactivating the neuron

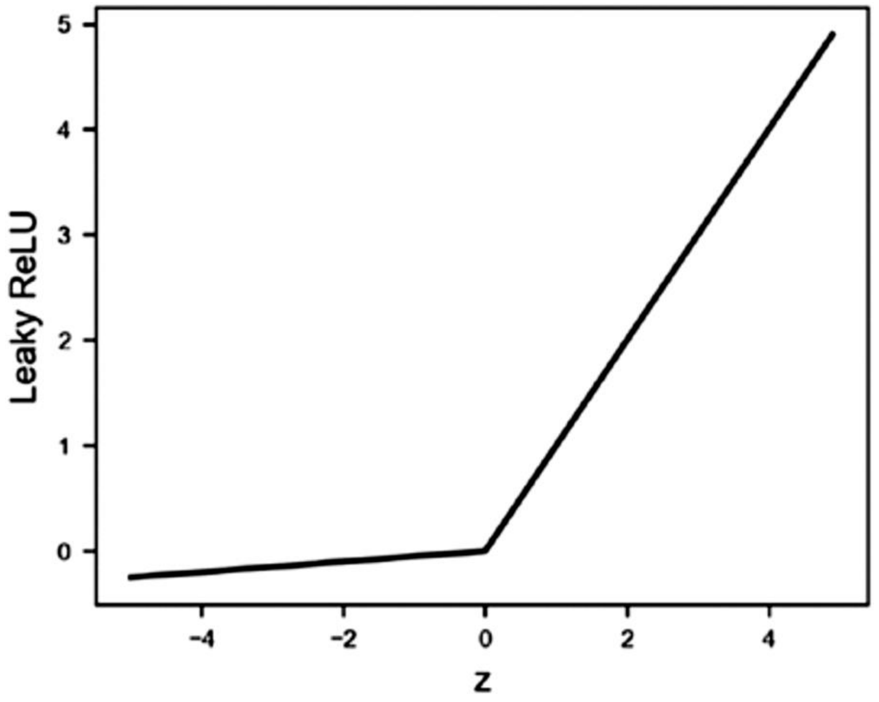
Drawback of ReLU

- Because of the horizontal line with 0 as the output, we can face serious issues sometimes.
- A horizontal line, which is a constant with a derivative of 0 become a bottleneck during training, as the weights will not get updated.
- *Solution*: Leaky ReLU
  - Here the negative value outputs a slightly slanting line instead of a horizontal line, which helps in updating the weights through backpropagation effectively

Leaky ReLU

- Leaky ReLU is defined as

$$f(z) = z\ ;\ when\ z\ > 0$$
$$f(z) = \alpha z\ ;\ when\ z < 0$$



- Where α is a parameter that is defined as a small constant, say 0.005

Model

- The overall structure of a DNN is developed using the model object in Keras.
- This provides a simple way to create a stack of layers by adding new layers one after the other.
- The easiest way to define a model is by using the sequential model

Model

- Creation of a simple sequential model with one layer followed by an activation.
- The layer has 10 neurons and receives an input with 15 neurons and be activated with the ReLU activation function.

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(10, input_dim=15))
model.add(Activation('relu'))
```

Layers

- Dense Layer
- Dropout Layer
- Embedding layers - https://keras.io/layers/embeddings/
- Convolutional layers - https://keras.io/layers/convolutional/
- Pooling layers - https://keras.io/layers/pooling/
- Merge layers - https://keras.io/layers/merge/
- Recurrent layers - https://keras.io/layers/recurrent/
- Normalization layers and many more - https://keras.io/layers/normalization/

The Loss Function

- The loss function is the metric that helps a network understand whether it is learning in the right direction.
- The loss function essentially measures the loss from the target.
- If the chance of passing or failing is defined by the probability, so,
    - 1 indicates pass with 100% certainty
    - 0 would indicate definite fail.

The Loss Function

- Suppose a model learns from the data and predicts a score of 0.87 for the student to pass. So, the actual loss here would be 1.00 – 0.87 = 0.13.
- If it repeats the exercise with some parameter updates in order to improve and now achieves a loss of 0.40, it would understand that the changes it has made are not helping the network to appropriately learn.
- Alternatively, a new loss of 0.05 would indicate that the updates or changes from the learning are in the right direction

some popular loss functions

- Mean Squared Error:

$$\sum_{n=1}^{k} \frac{(actual - predicted)^2}{k}$$

- Kerasequivalent

keras.losses.mean_squared_error(y_actual, y_pred)

Regression: Some popular loss functions

- Mean Absolute Error:average absolute error between actual and predicted

$$\sum_{n=1}^{k} \frac{|actual - predicted|}{k}$$

- Kerasequivalent

keras.losses.mean_absolute_error(y_actual, y_pred)

Regression: Some popular loss functions

- MAPE – Mean absolute percentage error

  keras.losses.mean_absolute_percentage_error

- MSLE – Mean square logarithmic error

  keras.losses.mean_squared_logarithmic_error

Classification: Some popular loss functions

- **Binary cross-entropy:** Defines the loss when the categorical outcomes is a binary variable, that is, with two possible outcomes: (Pass/Fail) or (Yes/No)

$$Loss = -\left[\, y * \log(p) + (1-y) * \log(1-p) \,\right]$$

- Keras equivalent

keras.losses.binary_crossentropy(y_actual, y_predicted)

Classification: Some popular loss functions

- **Categorical cross-entropy:** Defines the loss when the categorical outcomes is a nonbinary, that is, >2 possible outcomes: (Yes/No/Maybe) or (Type 1/ Type 2/… Type n)

$$loss = \sum_{i}^{n} y_i \log_2 y_i$$

- Keras equivalent

keras.losses.categorical_crossentropy(y_actual, y_predicted)