

# S2

---

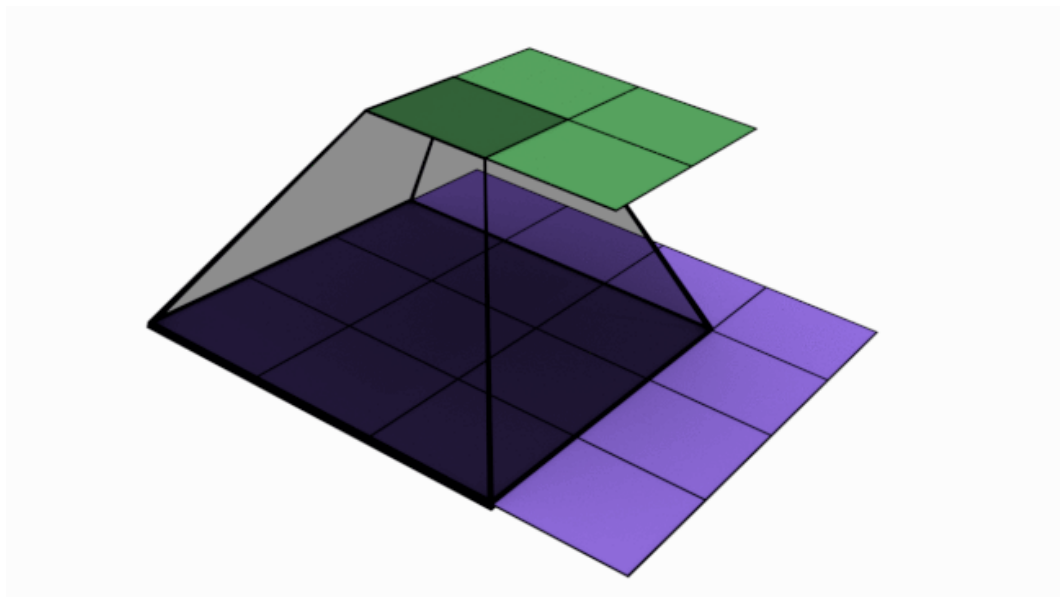
**Due** Wednesday by 5:30am    **Points** 0    **Available** Jan 22 at 6:25am - Jan 29 at 5:30am 7 days

---

## Session 2 | Neural Architecture

Let's re-look at some of the important concepts we covered in the last session.

### Concepts From Session 1



Here we see a 3x3 kernel convolving on an image/channel of size 4x4.

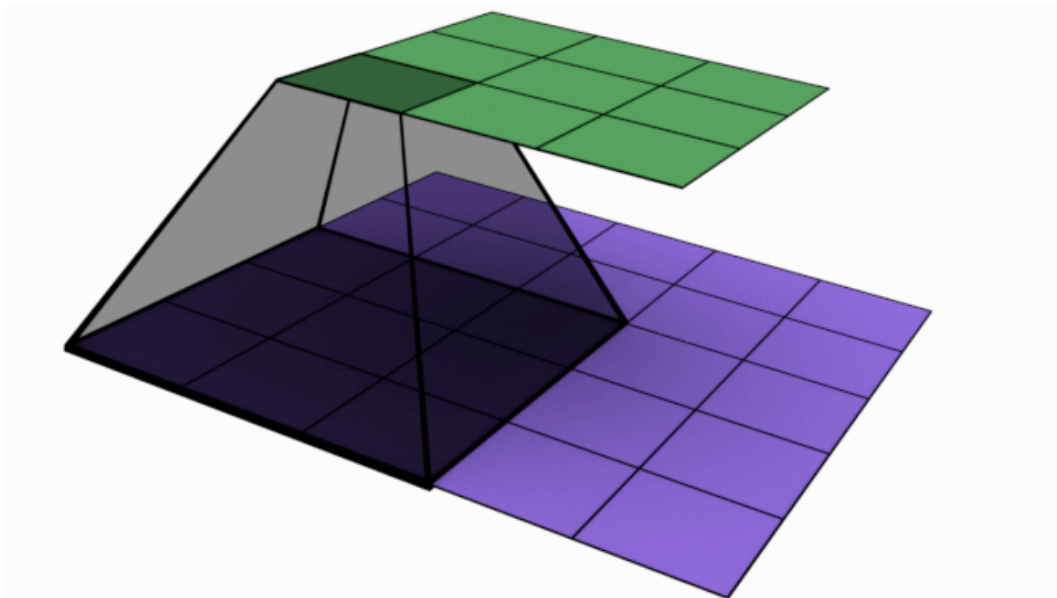
Every purple pixel we see represents a value, so in total, we are looking at 16 values. These values are generated from the image on which we are convolving, so we have no control over them.

The dark purple 3x3 moving box is our kernel. We initialize it (and all other kernels) randomly. We do have control over the values in the kernel, and that is what we want we (backpropagation) would be changing, such that they become the feature extractor (like a vertical edge detector).

Whenever our kernel is stopping, we are looking at 9 multiplications, we then sum all these 9 values, and pass the sum to the green channel. This green channel is called an output channel.

Whenever we perform a convolution with 3x3 kernel on an image of 4x4 in size, the output channel would have a resolution of 2x2. We essentially lose 2 pixels in x as well as the y-axis.

#### Convolving a 3x3 channel on 5x5



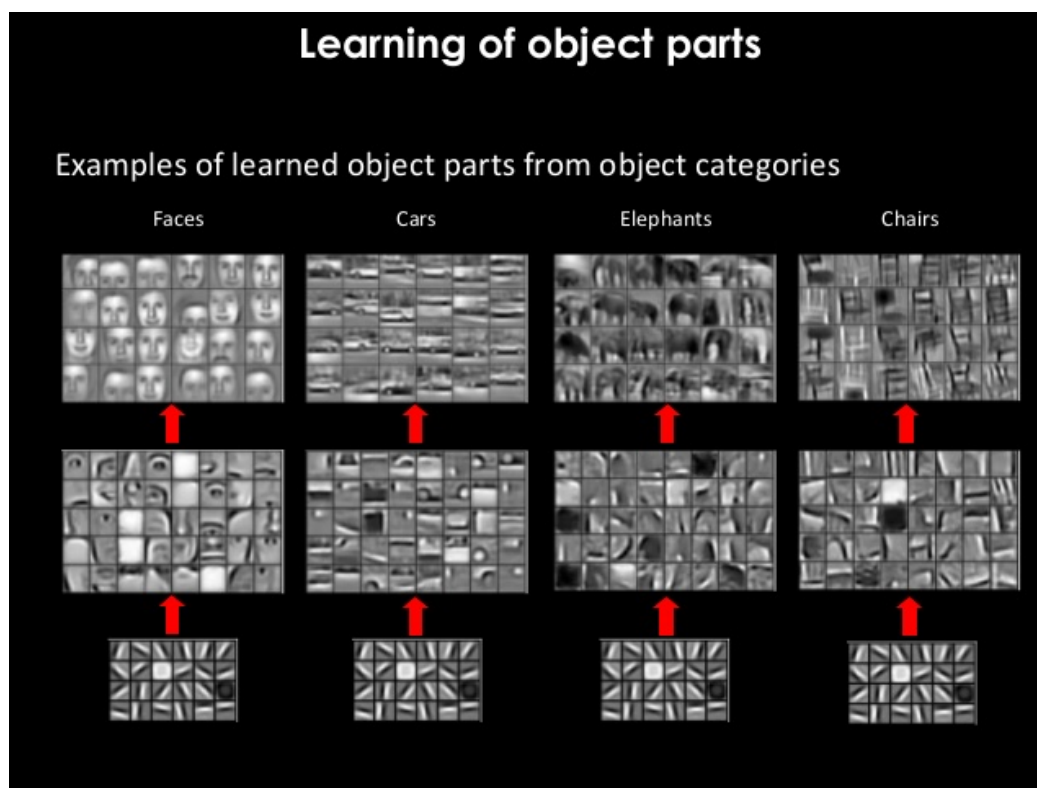
Similarly, if we convolve a 3x3 kernel on a 5x5 image, the output channel we would create will have a resolution of 3x3. This is true only in the case when:

1. we are not using any padding (we are not adding additional 0s (what else can we add?) on the boundaries of our input image, changing its resolution, say from 5x5 to 7x7), and
2. we are not using a stride of more than 1.

In the images above, you see, that whenever the kernel moves, it skips just 1 pixel. If it were to skip 2 pixels, that would be called a stride of 2.

## Why do we add layers?

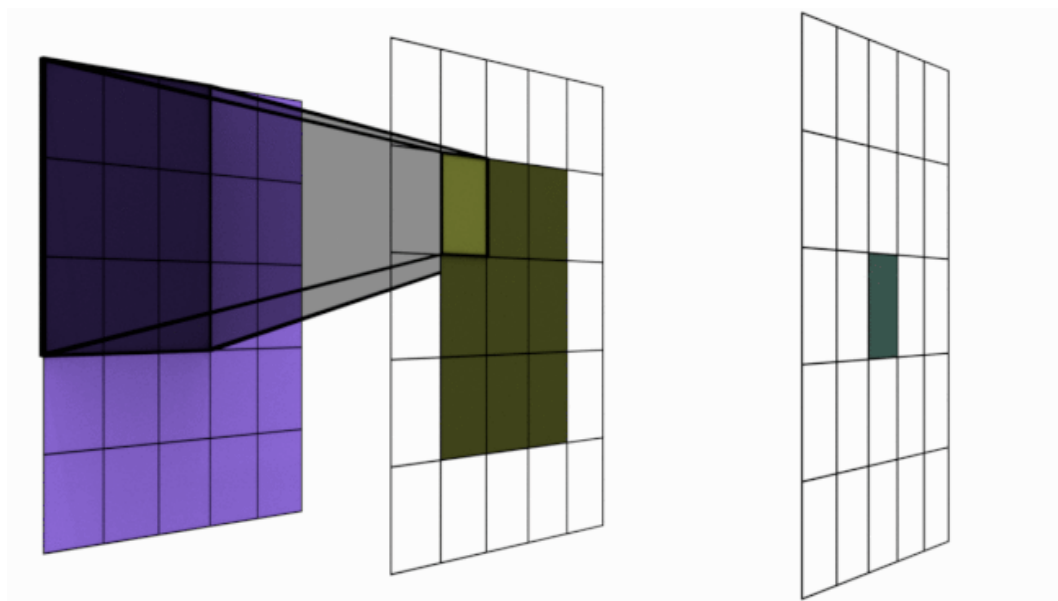
We add layers in a DNN for multiple reasons:



1. we have an objective (say detecting an object), and we can do that easily if we could detect the parts of the objects. Parts of the objects can be built from some patterns, and these patterns in-turn can be made from textures. To make any kind of texture, we would need edges and gradients. We add layers to procedurally do exactly this. We expect that our first layers would be able to extract simple features like edges and gradients. Next layers would then build slightly complex features like textures, and the patterns. Then later layers could build parts of objects, which can then be combined into objects. This can be seen in the image above.
2. we progressively add layers, the receptive field of the network slowly increases. If we are using 3x3 kernels, then each pixel in the second layers has only "seen" (receptive field) 3x3 pixels. Before the network can take any decision, the whole image needs to be processed. We add layers to achieve this. Also, consider the fact that required or important edges and gradients can be made or **seen within 11x11 pixels** in an image of 400x400. But say, we were looking at a face, the parts of the face would take much more area (or the number of pixels).



## Receptive Field



Here we see our first layer as a 5x5 image. We are convolving this 5x5 image with a kernel of size 3x3, and hence the resulting output resolution will be a channel with 3x3 pixels/values. When we convolve on this 3x3 channel with a kernel of size 3x3, we will get only 1 output. We have added 2 layers here.

To get the final output of 1 or 1x1, we could have used a 5x5 kernel directly. This means that using a 3x3 kernel twice is equivalent to using a 5x5 kernel. This also means that two layers of 3x3 have a resulting receptive field of 5x5.

As we have discussed in the class, we want the final global receptive field (at the final prediction layer or output layer) to be equal to the size of the image. This is important as the network needs to "see" the whole image before it can predict exactly what the image is all about.

This would mean that we need to add as many layers are required to reach the final receptive field equal to the size of the **object**. Since we have decided to consider the size of the object to be equal to the size of the image (for first few sessions), our final receptive field is going to be the size of the image. (We know this is not true, images can have objects of any size, but we need to consider this restriction to build our concepts. Later we would work on what needs to be done to remove this restriction).

### The Convolution Mathematics

3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

We can see above that our 3x3 kernel has these values:

```
0 1 2
2 2 0
0 1 2
```

Whenever our kernel is stopping on a 3x3 area, we are looking at 9 multiplications and the sum of the resulting 9 multiplications being passed on to the output (green) channel as shown in the image above.

The values in the output channel can be considered as the "confidence" of finding a particular feature. Higher the value, higher the confidence, and lower (or more negative) the value, **"higher"** the confidence of the **non-existence** of the feature.

Some examples of edge detectors would be:

-1	-1	-1
2	2	2
-1	-1	-1
Horizontal lines		

-1	2	-1
-1	2	-1
-1	2	-1
Vertical lines		

-1	-1	2
-1	2	-1
2	-1	-1
45 degree lines		

2	-1	-1
-1	2	-1
-1	-1	2
135 degree lines		

When we use the *horizontal edge detector kernel* with the values, as shown above, we get the following result:



Let's look at this through some numbers. Let us look at how a vertical edge would look like in an image:

0.2 0.2 **0.9** 0.2 0.5  
 0.1 0.1 **0.9** 0.3 0.2  
 0.0 0.2 **0.8** 0.1 0.1  
 0.2 0.3 **0.9** 0.1 0.2  
 0.1 0.1 **0.9** 0.3 0.2

The values shown in **bold** represents a vertical line in this image

Let us define our vertical kernel as:

-1 **2** -1

-1 **2** -1

-1 **2** -1

After convolving the values we get are:

-2.0 **4.3** -2.3

-1.7 **4.1** -2.1

-1.7 **4.1** -2.1

We can clearly see in this example that the central vertical values in the 3x3 output layer above, the detection of the vertical line. **Not only we have detected the vertical line, we are also passing on an image/channel which shows a vertical line.**

Spend a few moments to think about this bold line.

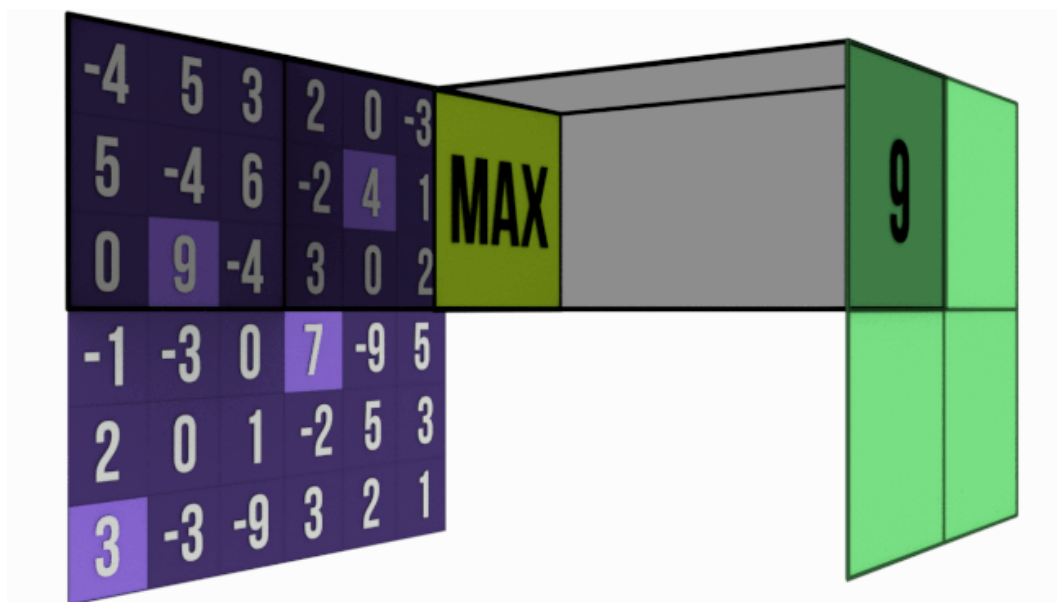
**How many layers would we need to move from 400x400 image to 1x1?**

As we saw in the last lecture, we need to add around 200 layers (as we add each layer we reduce the size of the image/channel by 2, so  $400/2 = 200$ , gives us the number of layers we need to add).

Now, these are an insanely large number of layers. We can do much better than this.

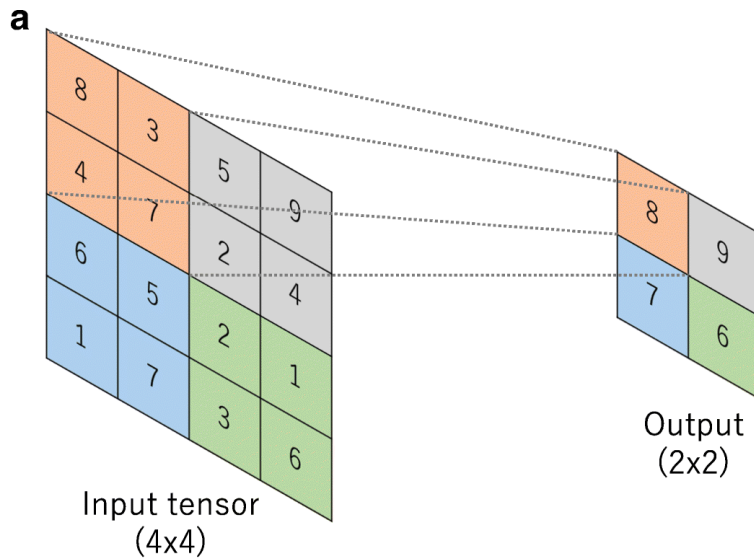
## MaxPooling

As we learned in the last lecture, we can use something called MaxPooling to solve this, as shown in the image below:



We saw this image and discussed that we rarely (rather never) use MaxPooling with 3x3, but rather use 2x2.



**b**

```

0 0 0 0 0 0 2 1 0 0 0 0 0 0 1 9 18 3 0 1 8 0 0 0 0 0 0
0 0 0 0 0 0 0 0 11 0 28 20 0 0 0 4 23 8 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 13 7 0 0 0 13 8 1 0 0 0 0 7 14 7 0
0 0 0 0 0 0 9 18 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 12 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 30 172 384 509 440 190 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 38 423 562 482 695 957 992 977 748 286 0 0 0 0 0 0
4 7 6 0 0 0 0 0 0 169 632 940 969 812 588 474 498 427 170 18 25 0 0 0 0
0 5 6 3 8 20 97 168 84 0 0 175 481 300 41 0 0 0 0 119 283 203 56 10 0
0 0 0 2 16 39 184 307 158 0 0 0 0 0 0 0 0 0 0 446 635 342 40 0 0
5 0 0 0 0 9 142 398 404 101 0 0 0 0 0 0 0 0 0 295 617 474 99 0 0
4 5 0 0 0 0 83 421 719 522 99 0 0 0 0 0 440 709 520 142 10 20 10 0
0 4 10 10 5 1 13 139 490 818 651 197 0 0 0 122 446 553 268 53 37 19 0 0
0 1 5 10 11 8 4 2 22 37 15 0 0 7 9 4 0 0 0 0 0 5 8 6 0 0
5 1 0 0 0 0 0 0 0 0 0 186 548 758 522 27 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 437 818 984 956 671 167 0 0 0 0 4 0 0 0
0 0 0 0 0 0 0 0 80 476 835 805 454 261 503 804 631 138 0 0 0 0 0 0
0 0 0 0 0 0 0 0 140 451 456 158 0 0 19 128 156 32 0 0 0 0 0 0
0 0 0 0 0 0 111 230 132 0 0 0 0 0 0 0 0 0 77 143 80 17 9 0 0
0 0 0 0 0 0 155 424 390 0 0 0 0 0 0 0 0 62 301 235 42 0 0 0
0 0 0 0 0 107 443 698 536 195 0 0 0 0 0 115 523 663 325 48 4 0 0
0 0 0 0 0 18 176 548 898 964 759 433 158 31 114 428 781 848 510 126 14 24 10 0
0 0 0 0 0 18 144 449 793 977 999 996 990 857 480 140 32 11 0 0 0 0
0 0 0 0 0 2 11 6 45 244 574 841 956 891 583 219 33 2 9 0 0 0 0
0 0 0 0 0 0 0 3 11 0 0 0 0 1 0 0 0 0 13 5 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 15 0 0 0 21 1 15 11 0 0 0 23 0 0 8 9 0 0
0 0 0 0 0 0 0 0 0 0 6 0 0 0 13 13 9 0 0 20 3 0 4 20 0 0
0 0 0 0 0 0 0 0 5 0 4 12 0 0 5 6 13 7 1 11 8 0 0 19 0 0
0 0 0 0 0 0 0 0 7 0 0 0 0 0 77 163 109 56 75 48 0 0 29 6 0 0
0 0 0 0 0 0 0 0 0 0 0 0 151 314 218 144 283 285 45 10 58 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 78 146 102 119 363 585 427 136 30 8 0 0
0 0 4 2 0 0 0 0 0 0 0 28 116 110 9 0 125 466 792 494 51 19 0 0
0 0 11 5 0 0 0 0 3 187 0 50 451 252 28 0 0 98 847 865 135 24 0 0
0 0 12 1 0 0 0 0 0 379 663 0 14 554 192 10 0 0 817 972 183 28 0 0
0 3 8 0 0 0 0 0 0 527 936 228 21 218 47 0 0 87 854 842 160 20 0 0
0 5 6 0 0 0 0 0 0 273 737 639 259 68 0 0 0 456 844 494 62 21 0 0
0 0 3 4 1 0 0 0 0 310 577 509 200 0 0 0 409 798 560 117 19 32 0 0
0 0 0 3 7 4 0 0 0 198 341 201 0 0 0 316 685 668 178 0 38 22 0 0
2 0 0 0 5 0 0 0 0 0 68 63 0 0 325 736 650 238 17 3 19 12 0 0
6 1 0 0 1 2 2 0 0 0 0 17 0 12 359 774 647 212 6 3 4 6 0 0
3 1 0 0 0 0 0 0 0 0 0 170 224 0 1 368 661 620 206 2 8 8 0 0
0 0 0 0 0 0 0 0 0 0 241 467 461 0 0 372 850 624 58 4 16 0 0
0 0 0 0 0 0 0 0 74 412 593 474 276 0 0 728 901 229 3 17 0 0
0 0 0 0 0 0 0 0 382 816 564 181 53 0 0 626 989 348 6 19 0 0
0 0 0 0 0 0 0 0 418 746 445 200 104 0 0 711 988 288 7 11 0 0
0 0 0 0 0 0 0 0 37 271 395 351 165 0 0 34 847 871 154 13 11 0 0
0 0 0 0 0 0 0 0 0 122 166 74 0 0 94 449 817 513 44 14 20 0 0
0 0 0 0 0 0 0 0 0 122 238 407 616 463 151 18 6 10 0 0
0 0 0 0 0 0 0 0 0 0 0 0 222 451 419 302 104 18 19 2 1 0 0
0 0 0 0 0 0 0 0 14 0 0 0 122 243 135 25 10 11 0 1 8 0 0
0 0 0 0 0 0 0 0 9 0 0 1 0 0 22 26 0 0 3 6 0 3 13 0 0
0 0 0 0 0 0 0 18 47 0 0 123 6 0 65 0 0 0 75 1 0 19 0 0 0 0
0 0 0 0 0 0 0 0 57 0 0 31 7 0 0 0 0 28 0 0 46 0 0 0 0
0 0 0 0 0 0 0 52 0 0 85 0 0 40 84 63 68 0 15 12 0 41 0 0 0
0 0 0 0 0 0 0 103 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 202 711 807 870 324 0 0 0 0 222 0 0 0 0 0
0 0 0 0 0 0 0 0 0 555 352 51 0 9 228 405 624 572 0 0 0 0 0 0
0 0 0 0 0 0 0 365 504 195 0 73 195 318 331 339 170 240 552 85 11 0 0 0
8 22 7 0 0 0 0 560 0 364 332 506 251 0 0 335 302 24 831 0 0 0 0 0
0 10 5 0 0 19 0 628 156 349 0 995 0 0 0 0 325 85 658 0 0 0 0 0
0 6 0 0 3 31 0 699 177 366 0 0 0 0 0 0 340 167 628 0 0 0 0 0
8 0 0 0 0 0 41 240 278 388 0 0 0 0 0 468 247 538 142 0 0 0 0
1 7 0 0 0 0 657 165 272 409 0 0 0 547 244 365 19 0 109 0 0 0 0
0 0 13 12 0 0 549 204 180 587 216 139 434 240 301 580 0 117 0 0 0 0
0 0 0 13 15 11 0 568 229 0 179 102 109 141 0 0 29 0 0 0 0 0
5 0 0 0 0 0 183 34 0 21 21 255 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 626 228 193 357 523 390 201 473 366 0 14 0 0 0 0
0 0 0 0 0 665 456 203 319 637 0 0 628 243 437 578 0 60 0 0 0
0 0 0 0 0 488 84 497 92 0 0 695 110 464 0 0 0 0 0 0 0
0 0 0 0 0 699 249 249 0 0 0 0 0 333 545 155 68 0 0 0 0
0 0 0 0 0 302 294 299 0 0 0 0 155 258 458 0 0 0 0 0 0
0 0 0 0 0 13 563 160 452 600 0 0 428 318 219 794 0 0 0 0 0
0 0 0 0 0 165 501 234 163 402 605 586 370 217 78 535 73 0 71 0 0
0 0 0 0 0 8 547 613 302 73 38 124 301 684 21 0 0 0 0 0 0
0 0 0 0 0 0 186 703 783 841 121 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 26 26 0 0 0 0 0 0 0 3 36 0 1 0 0 0 0
0 0 0 0 0 3 23 12 37 93 0 36 0 0 0 0 0 44 0 0 0

```

```

0 0 0 2 11 28 20 9 18 23 8 0 0
0 0 0 18 13 7 13 8 0 0 7 14 0
0 0 0 0 0 0 0 0 0 0 0 12 0
0 0 0 38 562 695 992 977 286 0 0
7 6 20 168 169 940 969 588 498 170 283 56 0
5 2 39 398 404 0 0 0 617 635 40 0
5 10 5 421 818 651 0 0 446 709 142 20 0
5 10 11 4 37 15 548 758 27 0 5 8 0
0 0 0 80 835 984 1002 804 138 0 4 0
0 0 0 230 140 456 158 19 156 77 143 17 0
0 0 0 443 698 195 0 0 115 663 325 4 0
0 0 0 176 898 977 101 201 1990 848 126 24 0
0 0 0 2 11 244 841 956 583 33 9 1 0

```

```

0 0 0 0 15 6 21 15 9 23 3 20 0
0 0 0 0 7 12 0 163 109 75 8 29 0
0 0 0 0 0 0 314 218 585 427 58 0
0 11 0 0 3 187 451 252 0 466 865 135 0
0 12 0 527 936 554 192 0 87 972 183 0
0 6 1 0 273 737 639 0 798 844 62 0
0 3 7 0 198 341 0 736 685 178 38 0
3 0 2 2 0 224 12 774 661 206 8 0
5 0 0 0 74 593 474 0 850 901 17 0
5 0 0 0 418 816 200 0 711 988 19 0
0 0 0 0 37 395 351 0 94 847 871 20 0
2 0 0 0 0 222 451 616 151 10 0
6 0 0 0 14 1 0 222 243 25 11 13 0

```

```

0 0 0 18 57 123 31 65 0 75 19 46 0
0 0 0 52 103 85 40 84 68 15 12 41 0
0 0 0 0 555 807 870 624 572 222 0 0
22 7 0 560 364 506 331 339 552 831 11 0
10 5 31 699 366 0 995 0 340 658 0 0
8 0 0 41 657 409 0 547 538 142 109 0
0 13 15 0 549 568 587 434 301 560 117 29 0
5 0 0 0 626 357 523 473 366 14 0 0
0 0 0 665 497 637 0 695 578 60 0 0
0 0 699 299 0 0 155 545 155 68 0 0
0 0 0 13 563 600 605 586 428 794 0 71 0
0 0 0 8 613 703 841 684 21 0 0 0
0 0 0 26 23 37 93 0 36 0 44 0 0

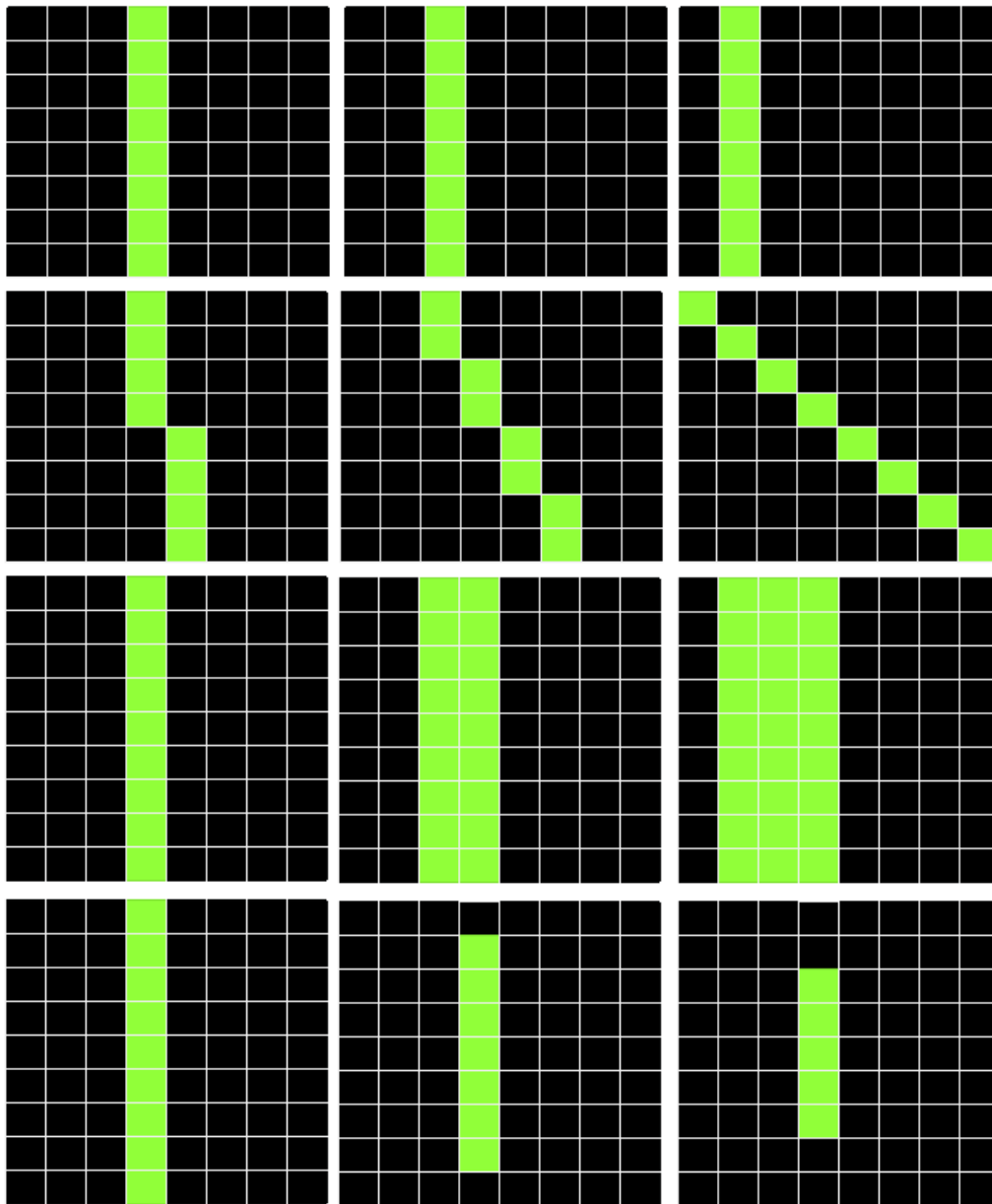
```

MaxPooling adds a bit of:

Shift Invariance

Rotational Invariance

Scale Invariance



How many layers would we need now?

400 | 398 | 396 | 394 | 392 | 390 | MP (2x2)

195 | 193 | 191 | 189 | 187 | 185 | MP (2x2)

92 | 90 | 88 | 86 | 84 | 82 | MP (2x2)

41 | 39 | 37 | 35 | 33 | 31 | MP (2x2)

15 | 13 | 11 | 9 | 7 | 5 | 3 | 1

By using MaxPooling we have reduced the layer count from 200 to 27. That's much better.

How many kernels did we add in the first layer?

How many kernels are required?

We would need a set of edges and gradients to be detected to be able to represent the whole image. Through experiments, we have learned that we should use around 32 or 64 kernels in the first layer, increasing the number of kernels slowly. Let's us assume we add 32 kernels in the first layer, 64 in second, 128 in third and so on.

**Our Network would look something like this:**

400x400 | (3x3)x32 | 398x398x32

398x398 | (3x3)x64 | 396x396x64

...

One need to observe here that the input to the second layer is not **398x398** but **398x398x32**, as we added 32 kernels. Each kernel would create its own channel.

## 3x3 is misleading!

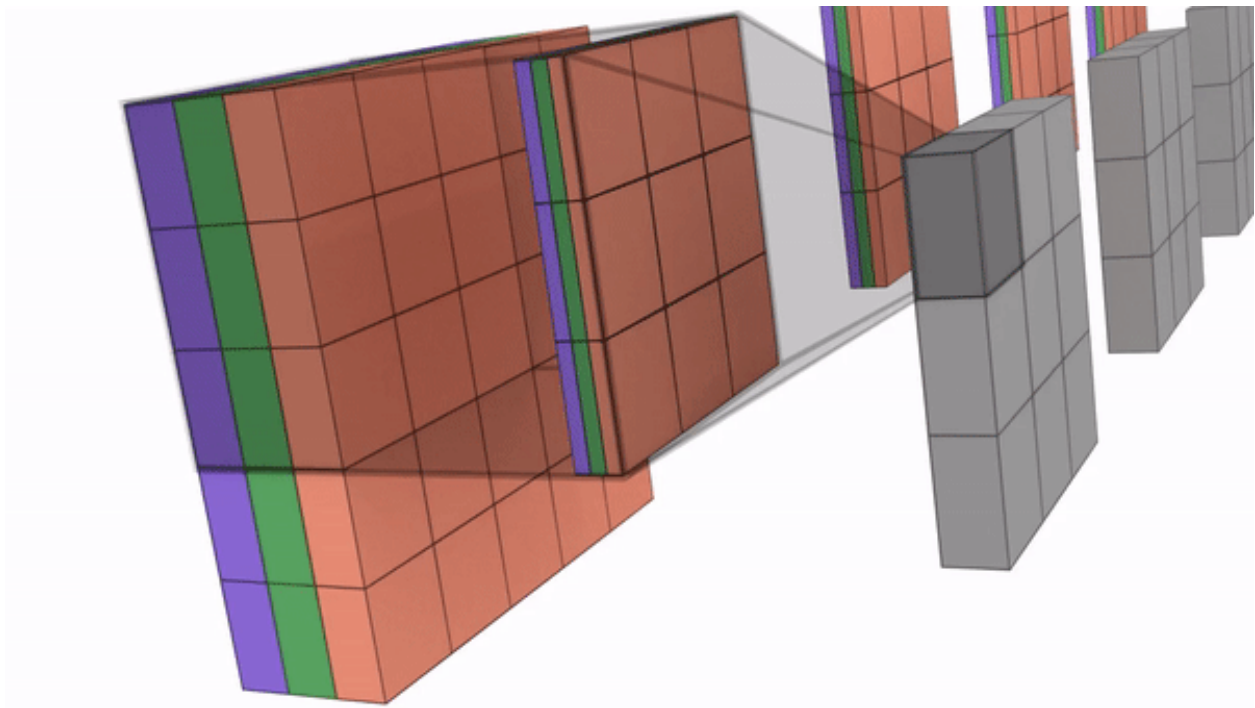
What we meant here is that unless we write the total channels in the 3x3 kernels, we are not representing it properly. We should write our kernel as **3x3x1**. If we were to re-write our network above again, it should be:

$$\begin{aligned}
 &400 \times 400 \times 1 \mid (3 \times 3 \times 1) \times 32 \mid 398 \times 398 \times 32 \\
 &398 \times 398 \times 32 \mid (3 \times 3 \times 32) \times 64 \mid 396 \times 396 \times 64 \\
 &\dots
 \end{aligned}$$

Notice that our kernels in the second layer have 32 channels.

**Our kernels must have an equal number of channels as in the input channel.** Since input has 32 channels in the second layer, our kernel will have 32 channels. Each channel in the kernel (say channel # 23) will look only at 1 channel (channel number 23 in the input).

Let's look at this animation:



In this animation, you can see that each kernel has 3 channels. Three channels are required as the input (5x5) has three channels. We are using 4 kernels here, that means we would have 4 channels in the output. Hence the output is 3x3x4.

If we have an infinite number of channels in the input, our kernels must have infinite channels. This has nothing to do with the number of channels in the output. **Output channels** are equal to the **number of kernels** we use.

## Multi Channel Convolution

Look at this image below, now you can understand how multi-channels are handled. (Please note that the bias is obsolete, and not used/focused on anymore)

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Input Channel #1 (Red)

0	0	0	0	0	0	...
0	167	166	167	169	169	...
0	164	165	168	170	170	...
0	160	162	166	169	170	...
0	156	156	159	163	168	...
0	155	153	153	158	168	...
...	...	...	...	...	...	...

Input Channel #2 (Green)

0	0	0	0	0	0	...
0	163	162	163	165	165	...
0	160	161	164	166	166	...
0	156	158	162	165	166	...
0	155	155	158	162	167	...
0	154	152	152	157	167	...
...	...	...	...	...	...	...

Input Channel #3 (Blue)

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

Kernel Channel #1



308

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

Kernel Channel #2



-498

0	1	1
0	1	0
1	-1	1

Kernel Channel #3



164

+ 1 = -25

Bias = 1

Output				
-25				...
				...
				...
				...
...	...	...	...	...

## Let's build a network again!

We are adding an increasing number of kernels as generally required:

400x400x1 | (3x3)x32 | 398x398x32  
 398x398x32 | (3x3)x64 | 396x396x64  
 396x396x64 | (3x3)x128 | 394x394x128  
 394x394x128 | (3x3)x256 | 392x392x256  
 392x392x256 | (3x3)x512 | 390x390x512  
 MaxPooling  
 195x195x512...

We have a problem here. Even though till now we have used  $32+64+128+256+512$  kernels (which is a small number), we right now have 992 images in our Memory. We solved the issue of large channel size by using MaxPooling, but we need to figure out a way to reduce these number of the channel while making sure, that, we are not defeating the purpose of increasing the number of channels (something we desperately want).

That's the topic of 3rd session!

## Assignment 2:

1. Open this link: [COLABLINK](https://colab.research.google.com/drive/1uJZvJdi5VprOQHROtJIHy0mnY2afjNlx) [\(https://colab.research.google.com/drive/1uJZvJdi5VprOQHROtJIHy0mnY2afjNlx\)](https://colab.research.google.com/drive/1uJZvJdi5VprOQHROtJIHy0mnY2afjNlx)
2. Duplicate this file to your Collaboratory
3. Then:
  1. read the file carefully
  2. add comments to all the cells carefully, explaining exactly what that cell does (for your own good)!
  3. in the cell where the main model is defined:
    1. write receptive field of each layer as a comment
    2. write the input channel dimensions
  4. run each cell one by one
  5. experiment
  6. Once you are done with your experiments, attempt S2 Solution Quiz. You will have 45 minutes to answer questions about this code. You will also be running the code once/twice within this 45 minutes.
  7. Read the S2 - Solution Quiz carefully before attempting it.
4. You have actual quiz called Q2 as well.
5. Fixed deadlines unless WW3 starts.

## Session 2 Video:

*Wednesday Batch*

EVA4 Session 2 - Wednesday Batch



