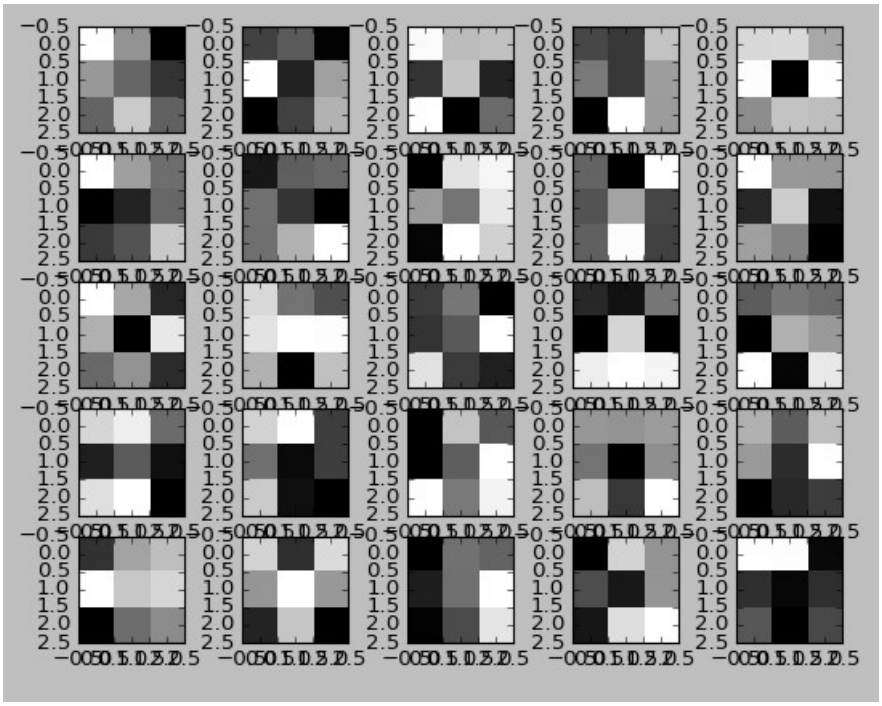
# S3

**Due** Wednesday by 5:30am **Points** None **Available** Jan 29 at 6:27am - Feb 5 at 5:30am 7 days

## Our First Neural Network

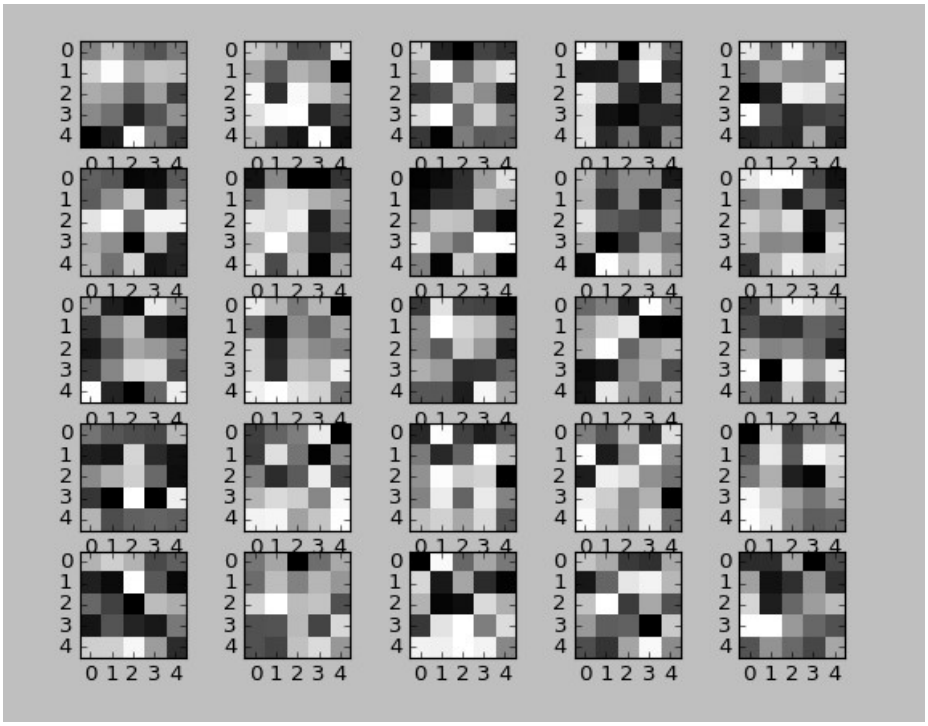
### Kernels

Let's look at what 3x3 kernels are trying to extract:

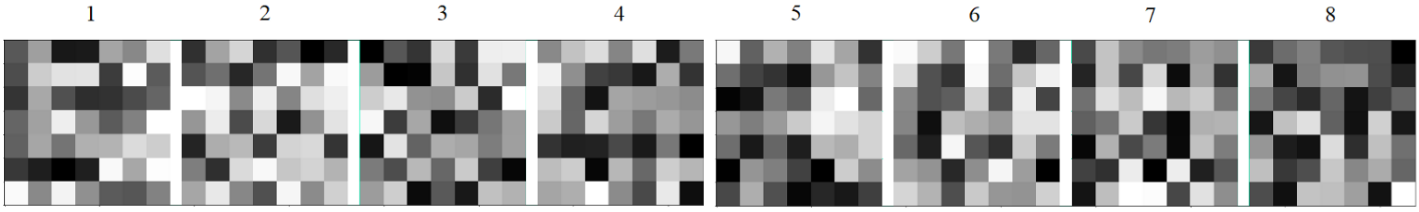


We really cannot find any pattern here. 3x3 is a very small area to actually gather any Perceivable information (for human eyes), especially when we are referring to an image of size 400x400.

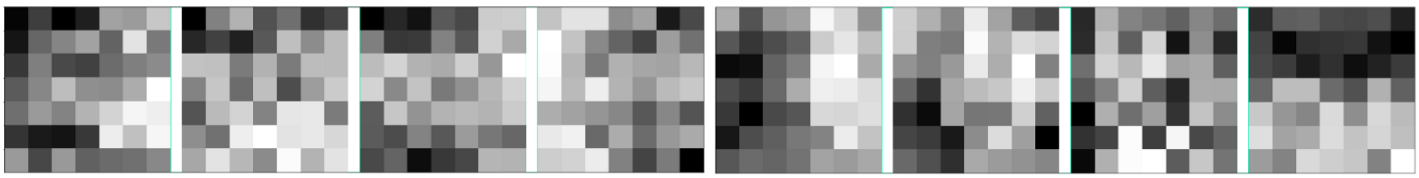
How about 5x5?



Even at 5x5 (for large image sizes) we can't make our a lot of stuff. In fact, at 7x7 we might also fail (look below):



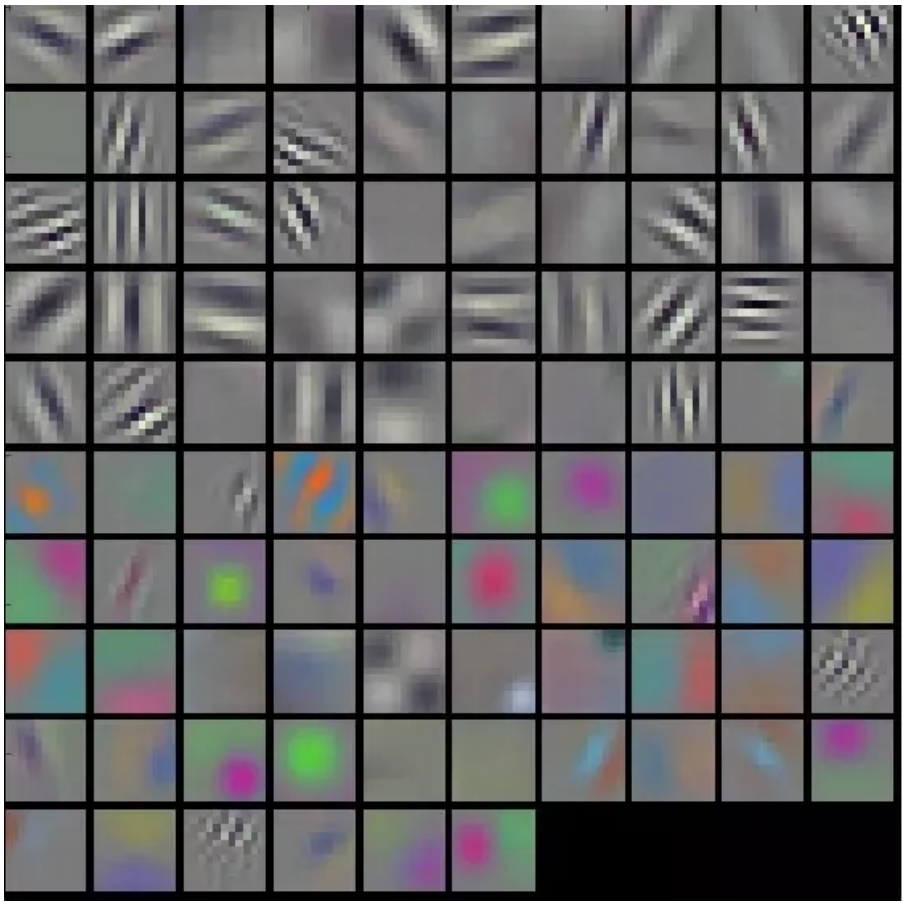

7x7 Kernels before training



7x7 Kernels after training

# 11x11

It is only at the receptive field of around 11x11 when we'd be able to make out things. Look what textures are extracting at 11x11.

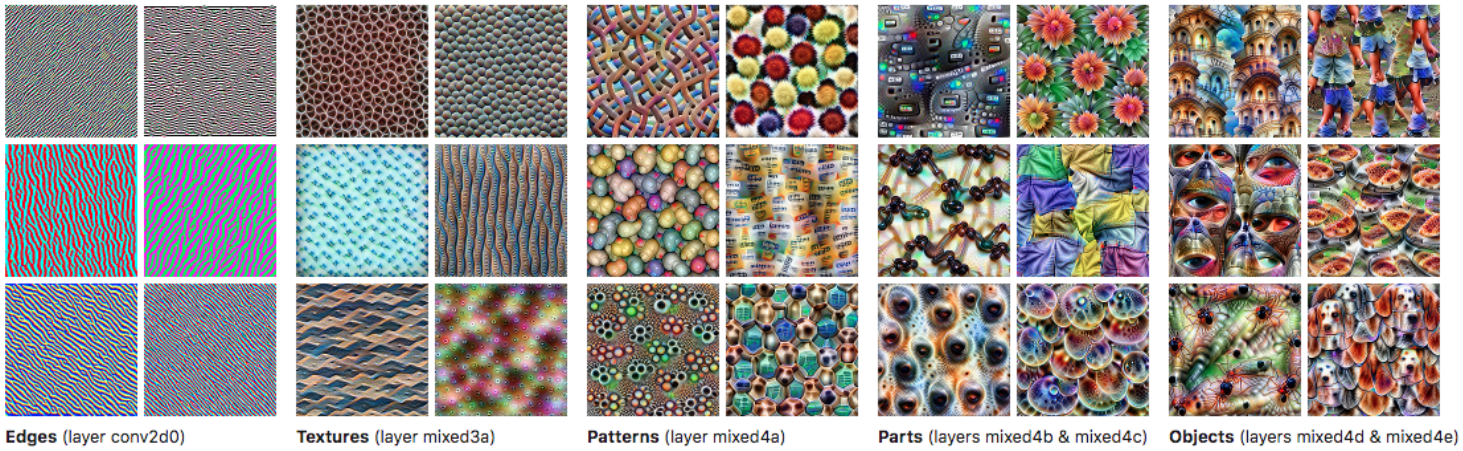


## Viewing Edges & Gradients, Textures, Patterns, Parts, Objects

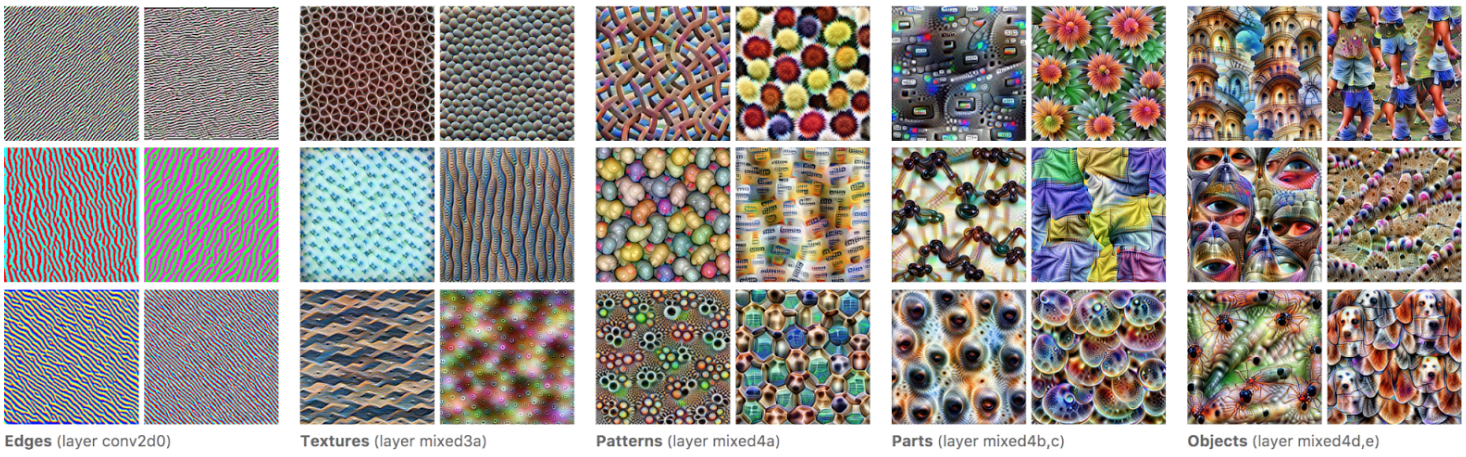
All this while we have been discussing that we extract edges and gradients, textures, patterns, part of objects and then objects. But is it true? In fact, if this is true, then we should be able to visualize our kernels and see that for ourselves!

And indeed that is the case.

Look at the image below:

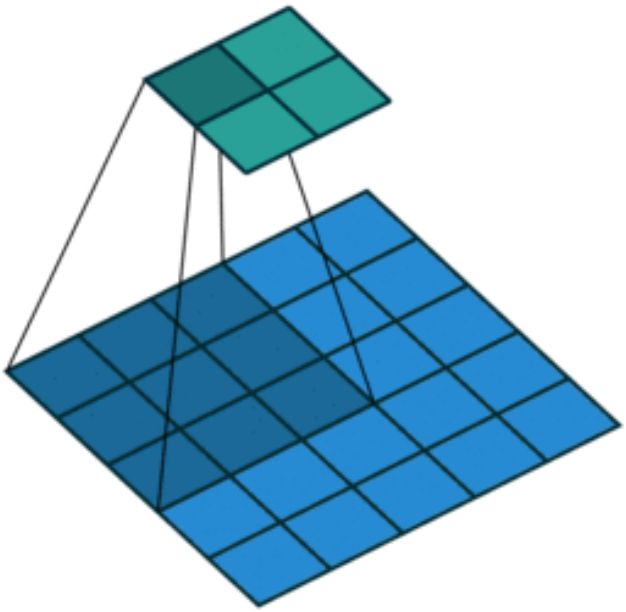



## Let's go on a visualization tour



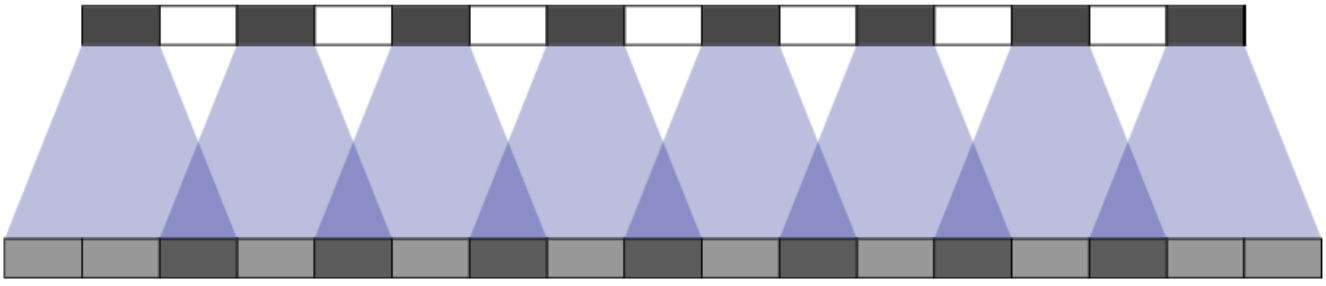

Please visit this link: [https://distill.pub/2017/feature-visualization/ (https://distill.pub/2017/feature-visualization/)](https://distill.pub/2017/feature-visualization/) to check out the other visualizations.

## Checkboard Issue



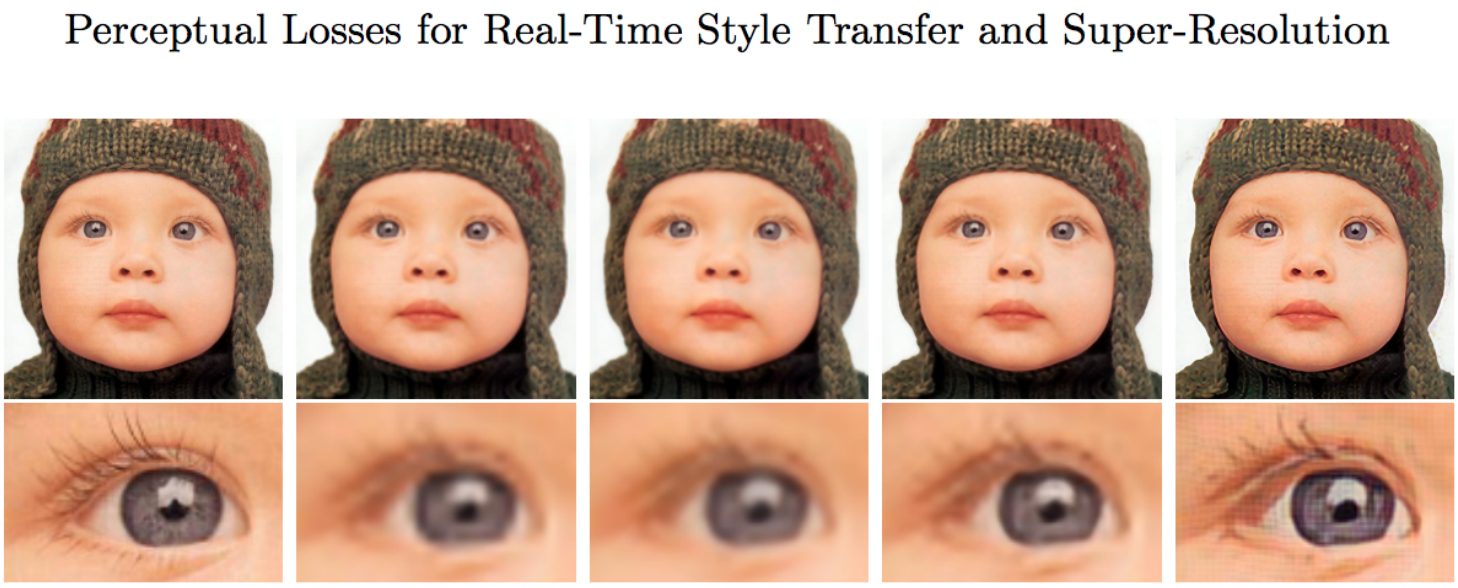
We learned in the last lecture that when we convolve with the standard way (stride of 1), we cover most of the pixels 9 times (we covered only 1 pixel 9 times on a channel of size 5x5, but as the channel size increases, we cover more and more pixels 9 times, for e.g. on a channel of size 400x400, 396x396 pixels would be covered 9 times).

But, when we convolve with a stride of more than 1, we would be covering some pixels more than once. And this is not good, as we are creating an island of extra information spread around in a repeating pattern. The image below would help:

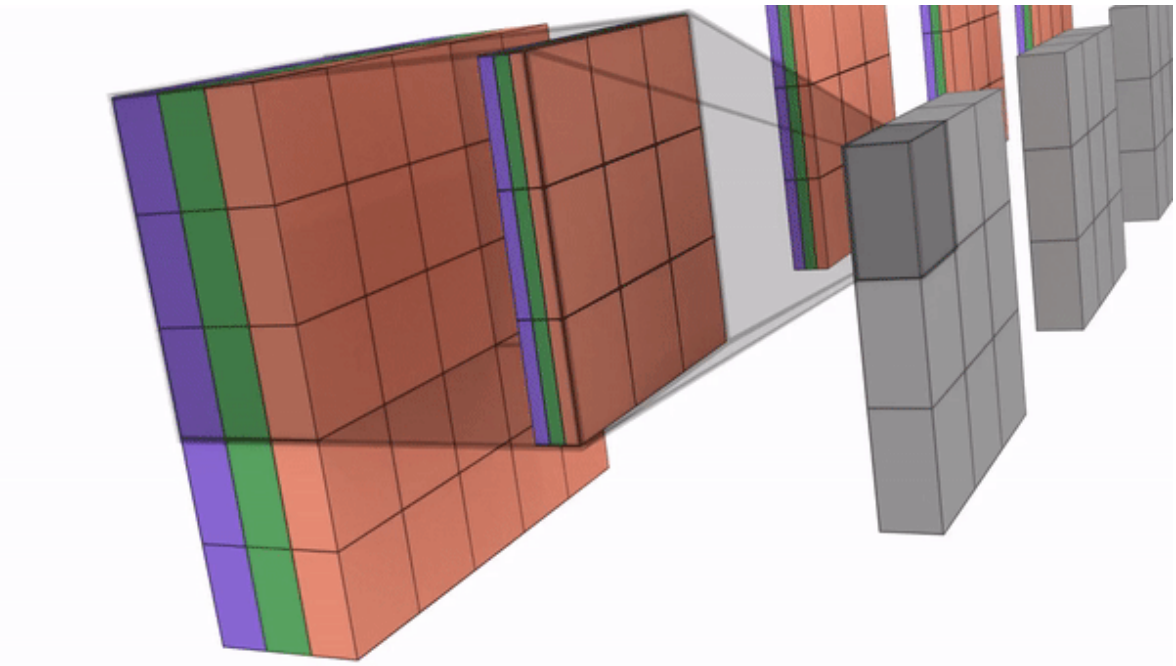


As you can see, we are spreading the information unevenly. We are actually blurring the inputs as can be seen in the image below:

We will come across this checkerboard issue again in super-Resolution algorithms. Observe the last image in the sequence of images below:




## What a 3x3x3x4 Kernel Convolution is?



It should be clear that we have a 3x3x**3**x**4** kernel above.

## Where were we in Session 2?

We stopped here:

**400x400**x3 | (3x3x3)x32 | 398x398x32 RF of 3x3
398x398x32 | (3x3x32)x64 | 396x396x64 RF of 5X5
396x396x64 | (3x3x64)x128 | 394x394x128 RF of 7X7
394x394x128 | (3x3x128)x256 | 392x392x256 RF of 9X9
392x392x256 | (3x3x256)x**512** | 390x390x512 RF **of 11X11**
**MaxPooling**
195x195x512 | (?x?x512)x32 | ?x?x32 RF of 22x22

.. 3x3x32x64 RF of 24x24
.. 3x3x64x128 RF of 26x26

.. 3x3x128x256 RF of 38x28
.. 3x3x256x**512** **RF of 30x30**

## Some points to consider before we proceed:

1. In the network above, the most important numbers for us are:
    1. **400x400**, as that defines where our edges and gradients would form
    2. **11x11**, as that's the receptive field which we are trying to achieve before we add transformations (like channel size reduction using MaxPooling)
    3. **512** kernels, as that is what we would need at a minimum to describe all the edges and gradients for the kind of images we are working with (ImageNet)
2. We have added 5 layers above, but that is **inconsequential** as our aim was to reach the **11x11** receptive field. *For some other dataset we might have reach required RF for edges&gradients, say after 4 or 3 layers*
3. We are using 3x3 because of the benefits it provides, our ultimate aim is to reach the receptive field of 11x11
4. We are following 32, 64, 128, 256, 512 kernels, but there are other possible patterns. We are choosing this specific one as this is expressive enough to build 512 final kernels we need, and since this is an experiment we could, later on, reduce the kernels depending on what hardware we pick for deployment.
5. The receptive field of 30x30 is again important for us because that is where we are "hoping" form textures.
6. The 5 Convolution Layers we see above form "**Convolution Block**".

The question we left unanswered in the last session was, how should we reduce the number of kernels. We cannot just add 32, 3x3 kernels as that would re-analyze all the 512 channels and give us 32 kernels. This is something we used to do before 2014, and it works, but intuitively we need something better. We have 512 features now, instead of evaluating these 512 kernels and coming out with 32 new ones, it makes sense to combine them to form 32 mixtures. That is where 1x1 convolution helps us.

Think about these few points:

1. Wouldn't it be better to merge our 512 kernels into 32 richer kernels which could extract multiple features which come together?
2. 3x3 is an expensive kernel, and we should be able to figure out something lighter, less computationally expensive method.
3. Since we are merging 512 kernels into 32 complex one, it would be great if we **do not pick** those features which are not required by the network to predict our images (like backgrounds).

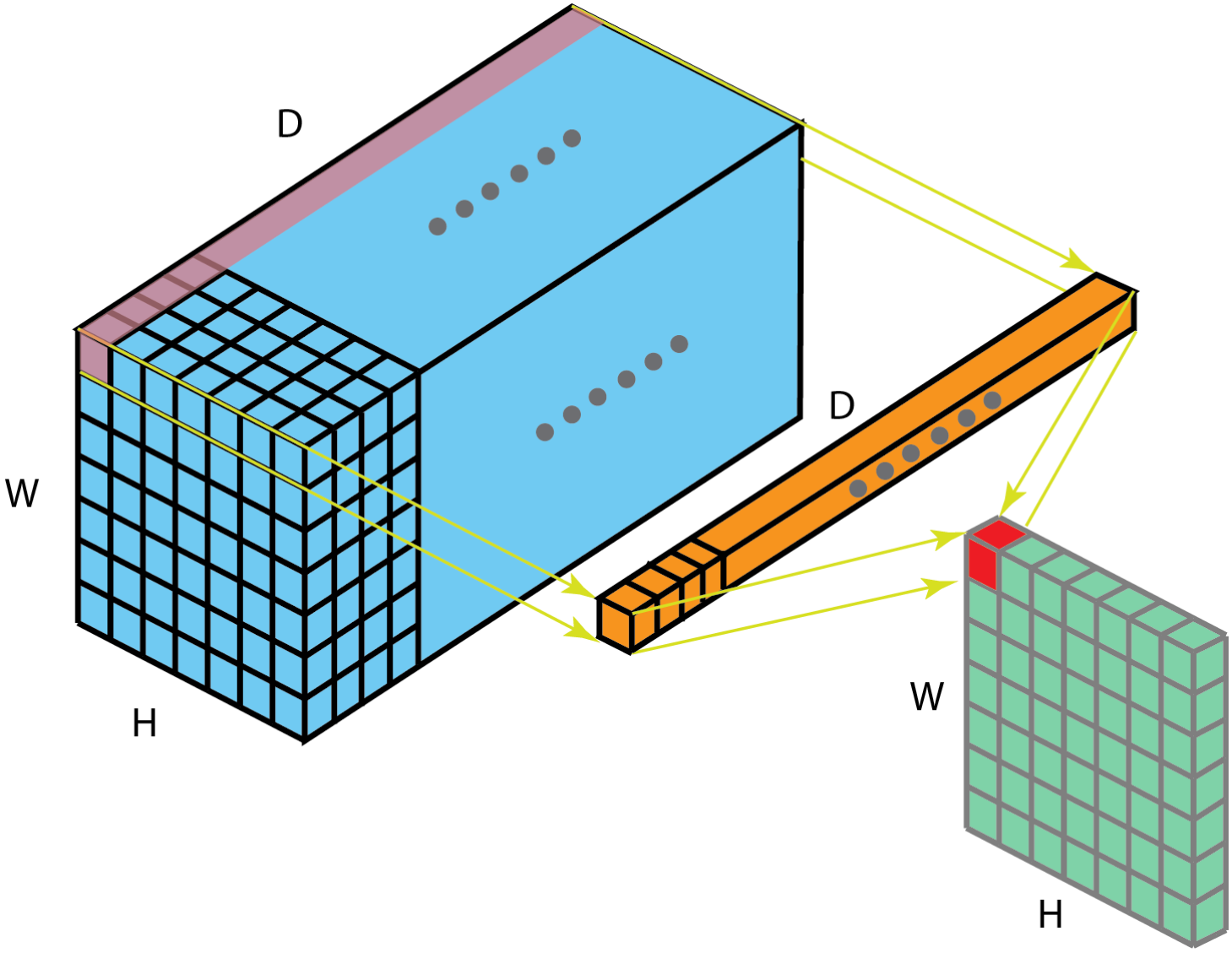
**1x1 provides all these features.**

1. 1x1 is computation less expensive.

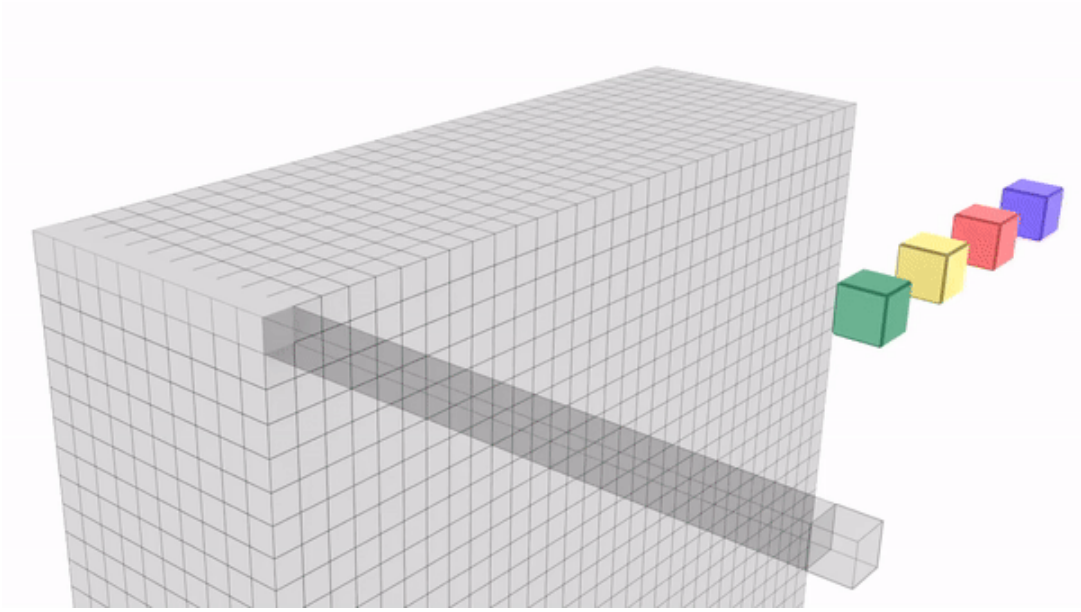
2. 1x1 is not even a proper convolution, as we can, instead of convolving each pixel separately, multiply the whole channel with just 1 number
3. 1x1 is merging the pre-existing feature extractors, creating new ones, keeping in mind that those features are found together (like edges/gradients which make up an eye)
4. 1x1 is performing a weighted sum of the channels, so it can so happen that it decides not to pick a particular feature that defines the background and not a part of the object. This is helpful as this acts like filtering. Imaging the last few layers seeing only the dog, instead of the dog sitting on the sofa, the background walls, painting on the wall, shoes on the floor and so on. If the network can filter out unnecessary pixels, later layers can focus on describing our classes more, instead of defining the whole image.

## 1X1 CONVOLUTION

This is how 1x1 convolution looks like:




What you are seeing above is an input image of size 7x7x**D** where D is the number of channels. We remember from the last lecture, that any kernel which wants to convolve, will need to possess the same number of channels. Our 1x1 kernel must have D channels. Simply put, our new kernel has D values, all defined randomly, to begin with. In 1x1 convolution, each channel in the input image would be multiplied with 1 value in the respective channel in 1x1, and then these weighted values would be summed together to create our output. This animation should help:



What you see above is an input of size 32x32x10. We are using 4 1x1 kernels here. Since we have 10 channels in input, our 1x1 kernel also has 10 channels.

## 32x32x**10** | 1x1x**10**x4 | 32x32x**4**

We have reduced the number of channels from 10 to 4. Similarly, we will use 1x1 in our network to reduce the number of channels from 512 to 32. Let's look at the new network:

**400x400**x3 | (3x3x3)x32 | 398x398x32 RF of 3x3

CONVOLUTION BLOCK 1 BEGINS
398x398x32 | (3x3x32)x64 | 396x396x64 RF of 5X5
396x396x64 | (3x3x64)x128 | 394x394x128 RF of 7X7
394x394x128 | (3x3x128)x256 | 392x392x256 RF of 9X9
392x392x256 | (3x3x256)x**512** | 390x390x512 RF **of 11X11**

CONVOLUTION BLOCK 1 ENDS

TRANSITION BLOCK 1 BEGINS
MAXPOOLING(2x2)
195x195x512 | **(1x1x512)x32** | 195x195x32 RF of 22x22
TRANSITION BLOCK 1 ENDS

CONVOLUTION BLOCK 2 BEGINS
195x195x32 |(3x3x32)x64 | 193x193x64 RF of 24x24
193x193x64 |(3x3x64)x128 | 191x191x128 RF of 26x26
191x191x128 |(3x3x128)x256 | 189x189x256 RF of 28x28
189x189x256 |(3x3x256)x512 | 187x187x512 RF of 30x30
CONVOLUTION BLOCK 2 ENDS

TRANSITION BLOCK 2 BEGINS
MAXPOOLING(2x2)
93x93x512 | **(1x1x512)x32** | 93x93x32 RF of 60x60
TRANSITION BLOCK 2 ENDS

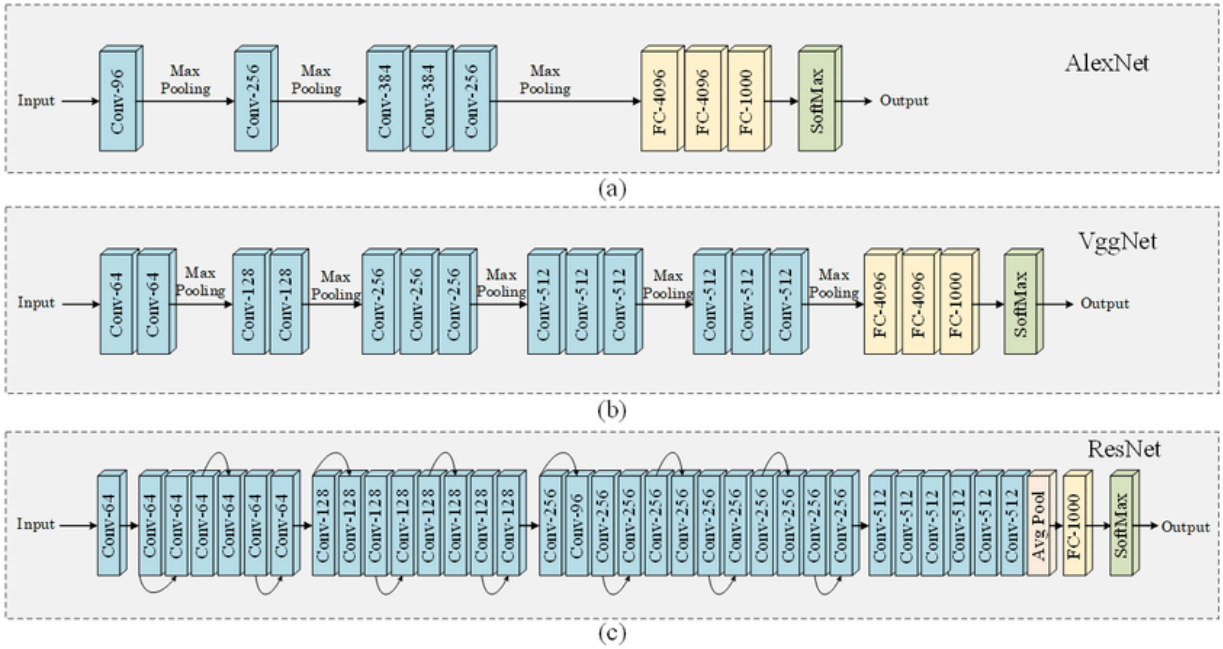
CONVOLUTION BLOCK 3 BEGINS
93x93x32 | (3x3x32)x64 | 91x91x64 RF of 62x62
...

Notice, that we have kept the first convolution outside of our convolution block, as now we can create a functional block receiving 32 channels and then perform 4 convolutions, giving finally 512 channels, which can then be fed to transition block (hoping to receive 512 channels) which finally reduces channels to 32.

## THE MODERN ARCHITECTURE

What we have covered as architecture is for understanding only. This architecture is called Squeeze and Expand.

Most of the modern architecture follows this architecture:

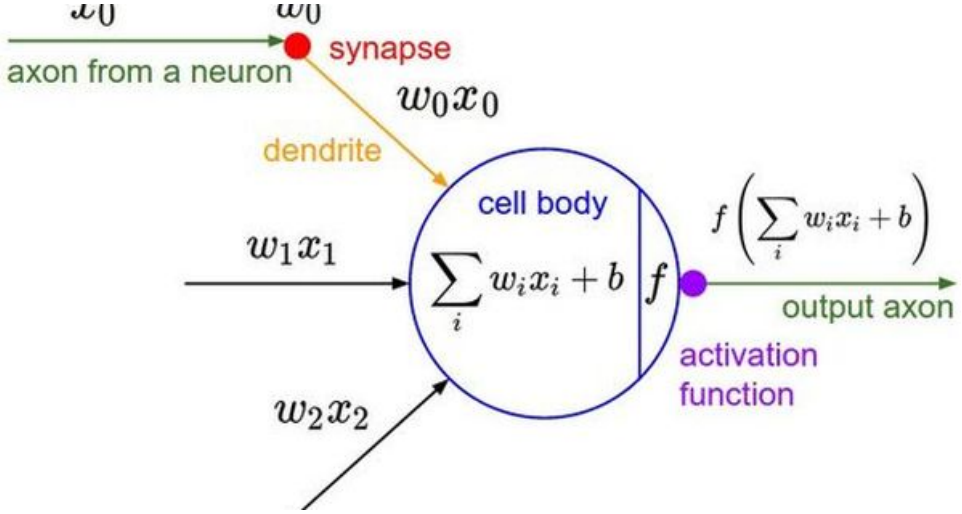



Major Points:

1. ResNet is the latest among the above. You can clearly note 4 major blocks.
2. The total number of kernels increases from 64 > 128 > 256 > 512 as we proceed from the first block to the last (unlike what we discussed where at each block we expand to 512. Both architectures are correct, but 64 ... 512 would lead in lesser computation and parameters.
3. Only the most advanced networks have ditched the FC layer for the GAP layer. In TSAI we will only use GAP Layers.
4. Nearly every network starts from 56x56 resolution!

# Activation Functions

**Their main purpose is to convert an input signal of a node in an ANN to an output signal.**




Since ages, this activation function has kept the AIML community occupied in the wrong directions.

After the convolution is done, we need to take a call with what to do with the values our kernel is providing us. Should we let all pass, or should we change them? This question of choice (of what to do with output) has kept us guessing. And as humans always feel, deciding what to pick and what to remove must have a complicated solution.

## Why do we need an activation function?

If we do not apply an Activation function then the output signal would simply be a simple **linear function**. A *linear function* is just a polynomial of **one degree.** Now, a linear equation is easy to solve but they are limited in their complexity and have less power to learn complex functional mappings from data. A Neural Network without Activation function would simply be a **Linear regression model,** which has limited power and does not perform well most of the time. We want our Neural Network to not just learn and compute a linear function but something more complicated than that.

## Why do we need non-linearity?

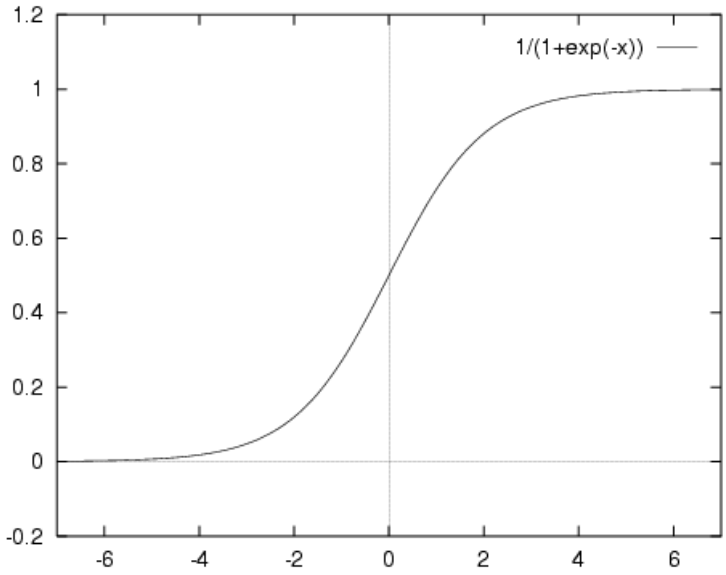
We need a Neural Network Model to learn and represent almost anything and any arbitrary complex function that maps inputs to outputs. Neural-Networks are considered **Universal Function Approximators**. *It means that they can compute and learn any function at all*. Almost any process we can think of can be represented as a functional computation in Neural Networks.

*Hence it all comes down to this, we need to apply an Activation function f(x) so as to make the network more powerful and add the ability to it to learn something complex and complicated form data and represent non-linear complex arbitrary functional mappings between inputs and outputs. Hence using a non-linear Activation we are able to generate non-linear mappings from inputs to outputs.*

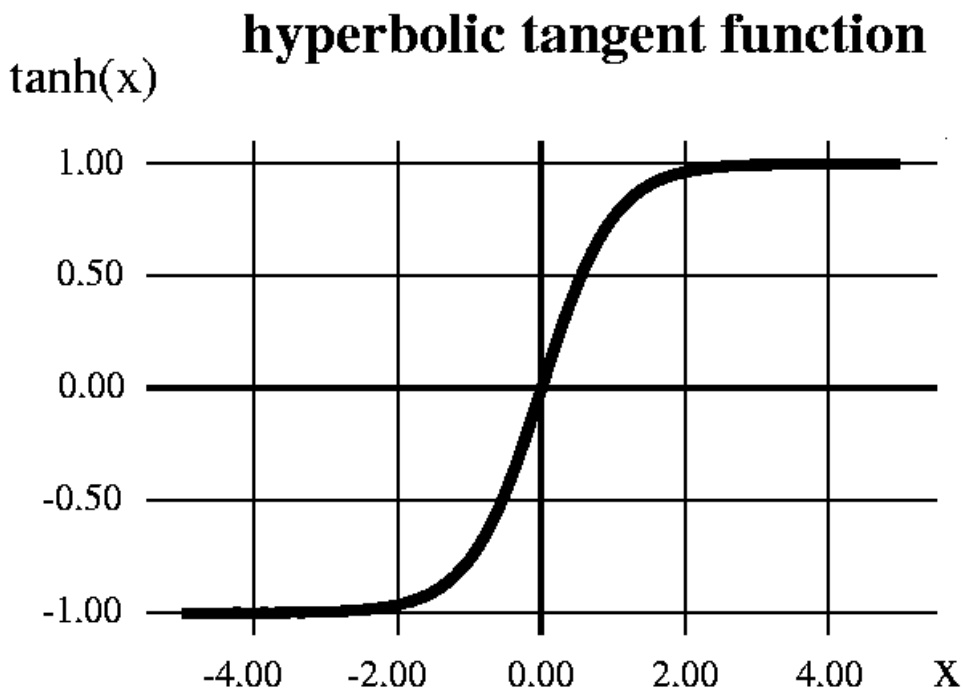
**Read More** **(https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f)**

## Sigmoid

We used Sigmoid function for decades and faced with something called gradient descent or gradient explosion problems. This is how Sigmoid works:



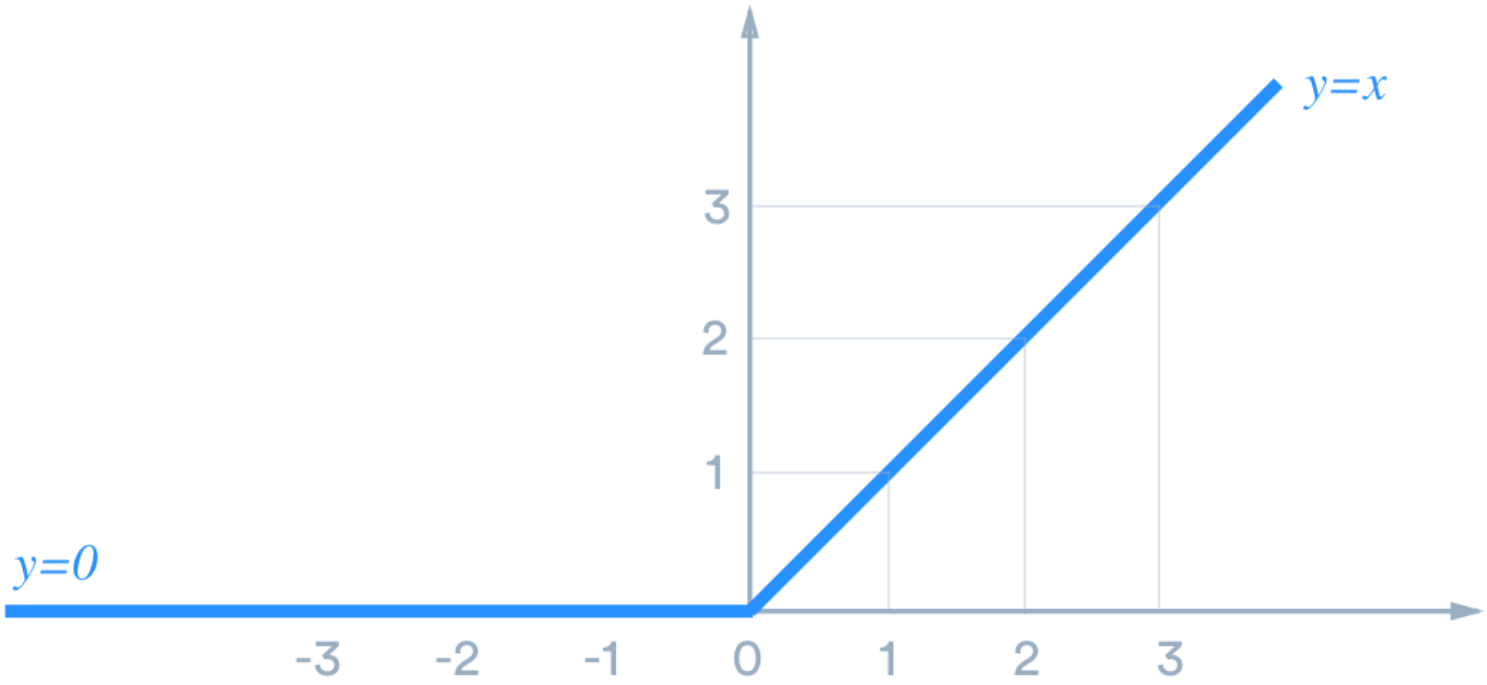

When sigmoids didn't work, we felt maybe another complicated function, called **TanH** might work:



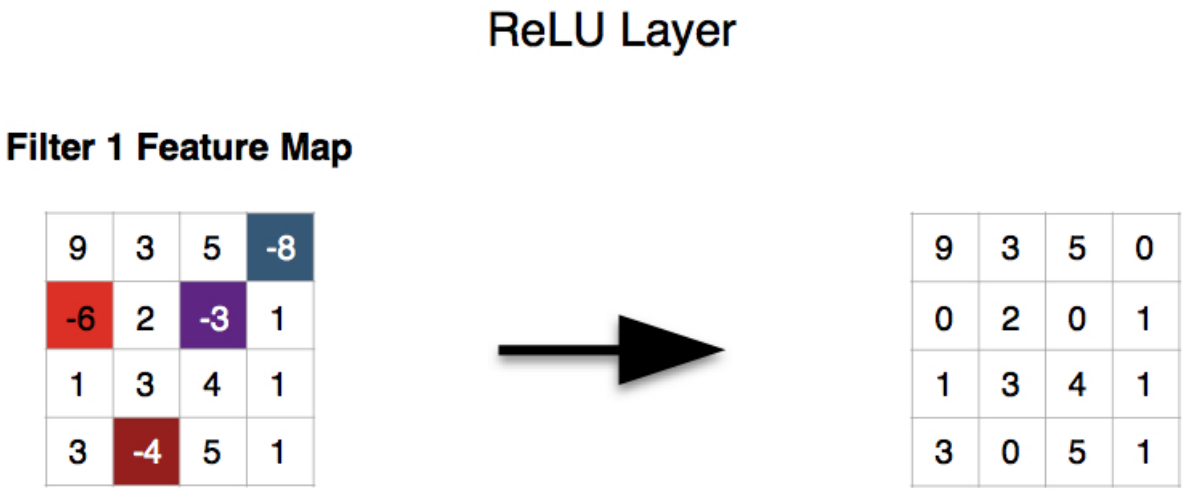

But neither did.

It turns out that the process of making a decision is not so complicated after all.

## ReLU




This is a very simple function. It allows all the positive numbers to pass on, as it is, and converts all the negatives to zero. It is a message to backpropagation or kernels is simple. If you want some data to be passed on to the next layers, please make sure the values are positive. Negative values would be filtered out. This also means that if some value should not be passed on to the next layers, just convert them to negatives.




ReLU since has worked wonders for us. It is fast, simple and efficient.

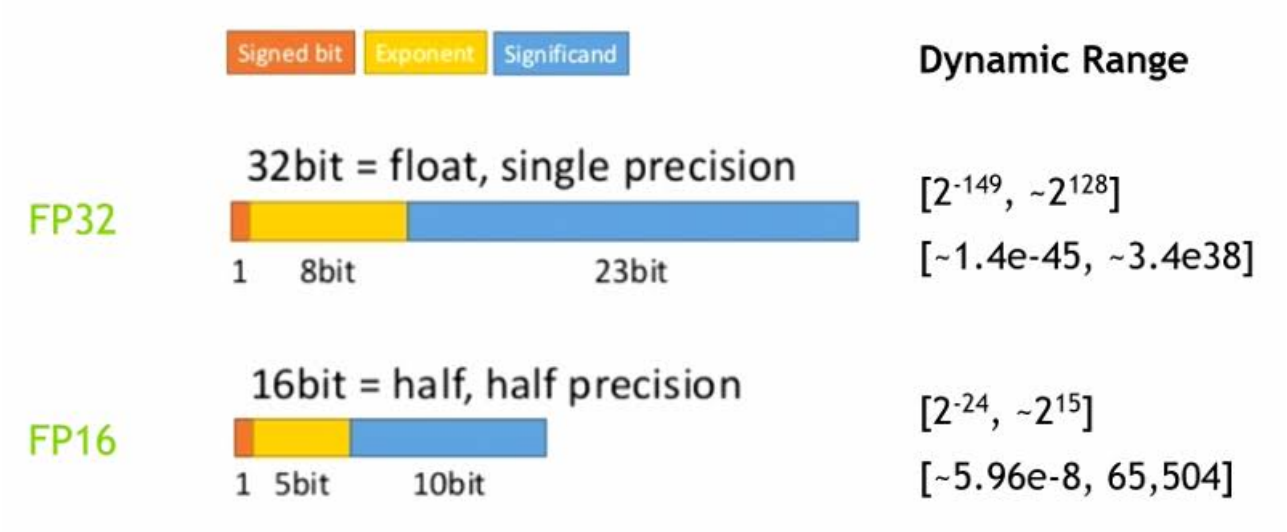
**ReLu- Rectified Linear units**: It has become very popular in the past couple of years. It was recently proved that it had *6 times improvement in convergence* from Tanh function.

It's just:

$$R(x) = \begin{cases} x \ if \ x > 0 \\ 0 \ if \ x \leq 0 \end{cases}$$

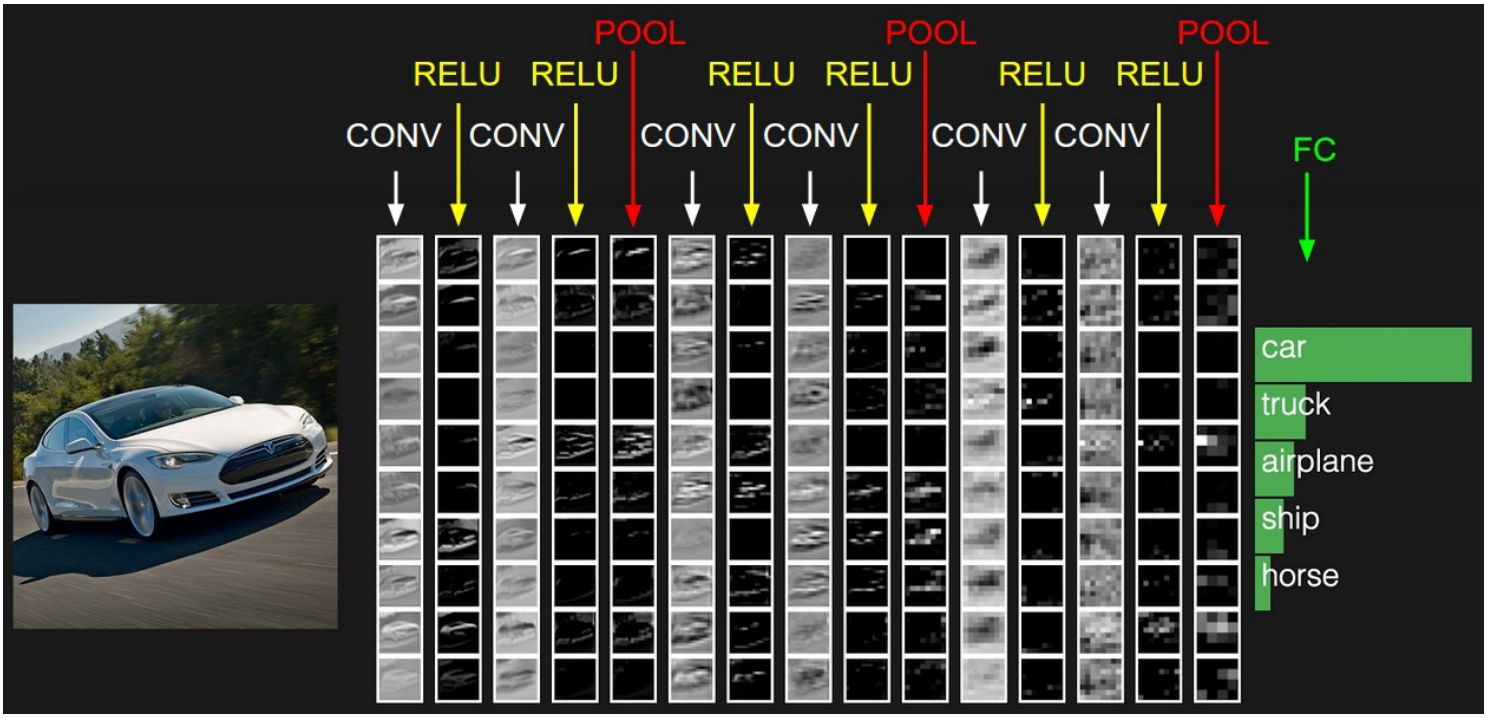
ReLU is linear (identity) for all positive values, and zero for all negative values. This means that:

- It's cheap to compute as there is no complicated math. The model can, therefore, take less time to train or run.




- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when *x* gets large. It doesn't have the vanishing gradient problem suffered by other activation functions like sigmoid or tanh.
- It's sparsely activated. Since ReLU is zero for all negative inputs, it's likely for any given unit to not activate at all.

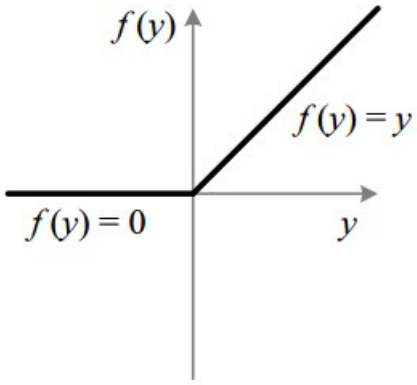
Zoom on this image to see what ReLU does ("filtering stuff out"):

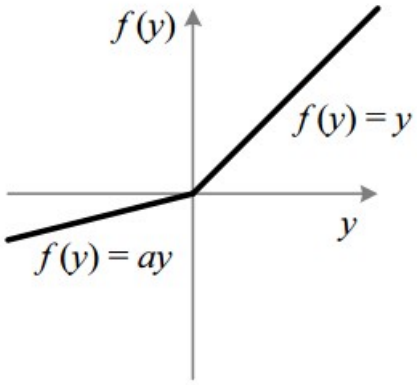



But aren't humans known to complicate things? One might wonder, why such partiality to all the negative numbers? What if we allow a negative number to *leak* a few their values.

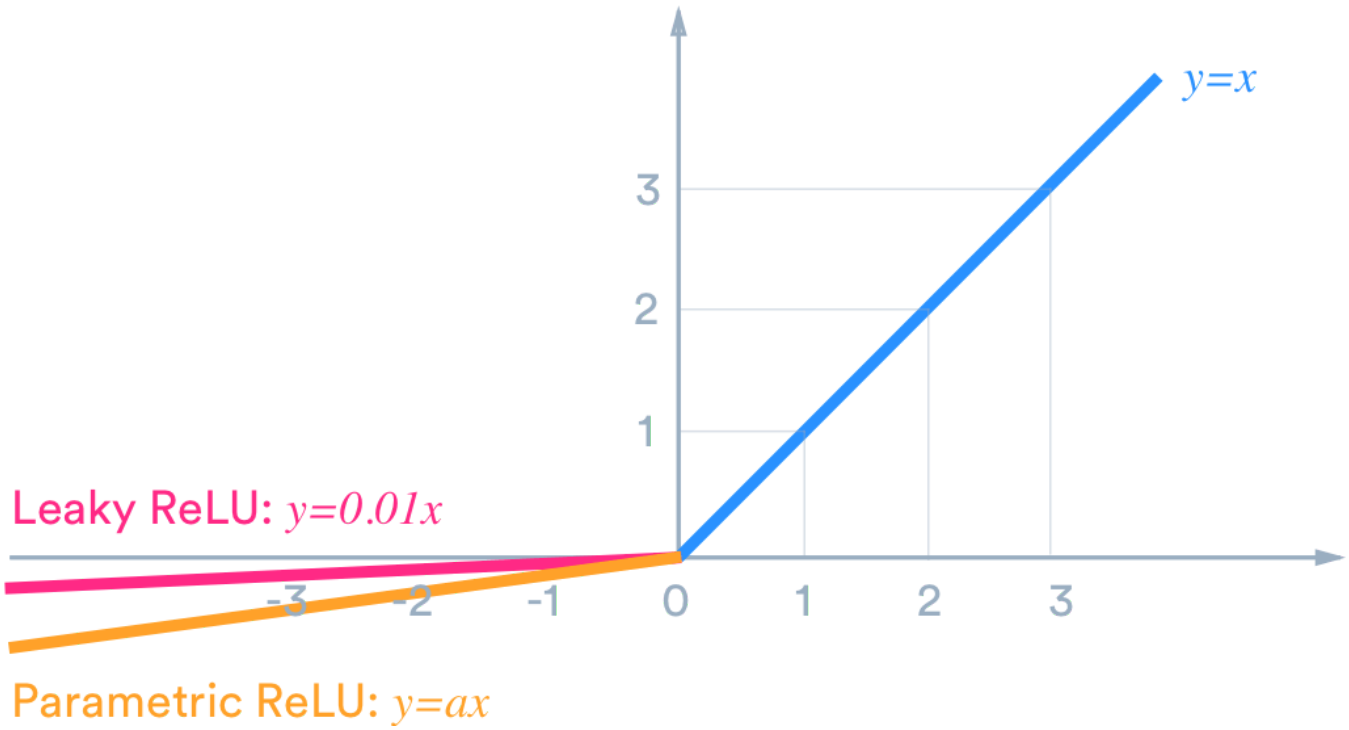
Hence came **Leaky ReLU**

## ReLU vs LeakyReLU



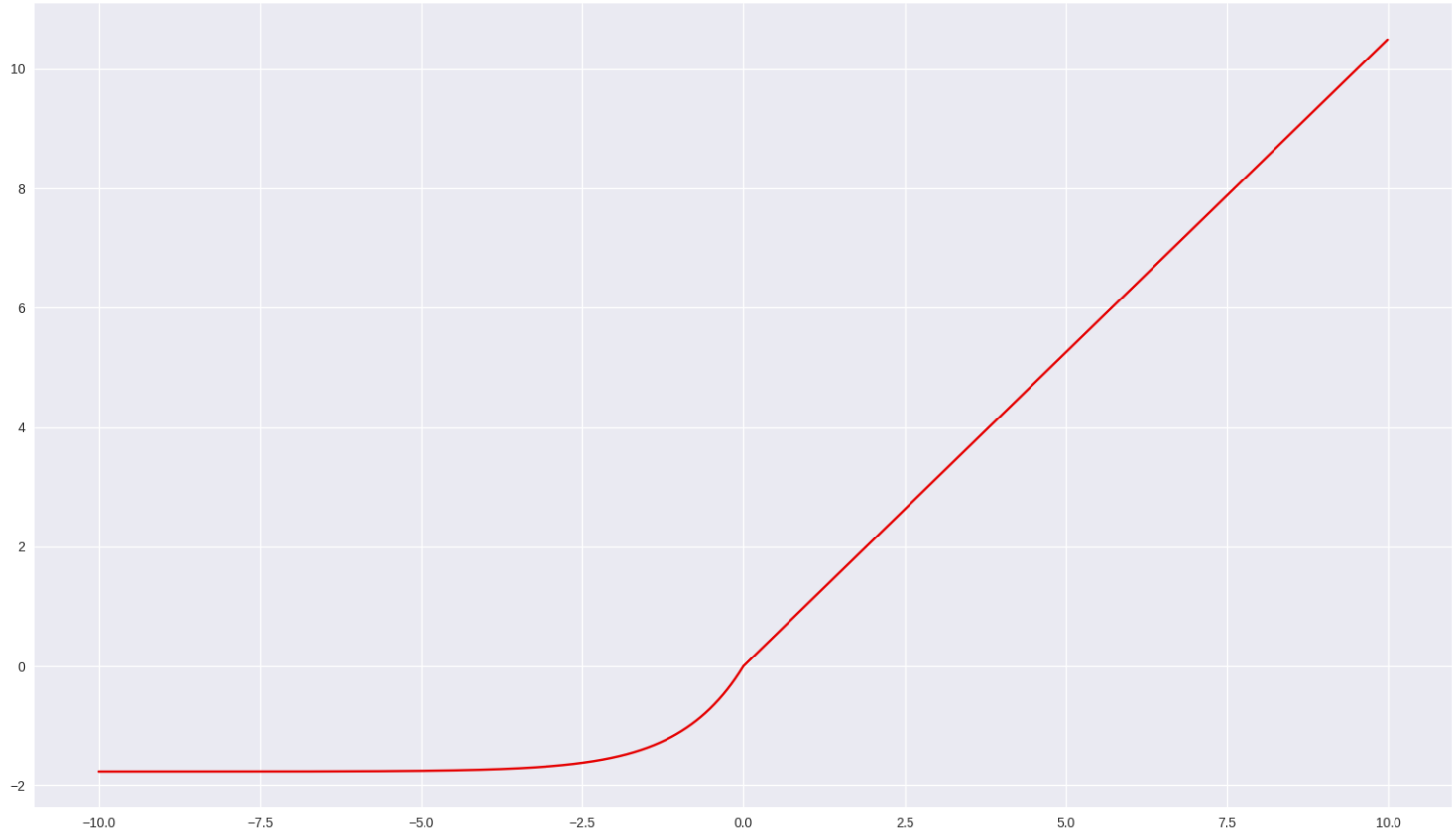




And then why stop there. You can complicate this further. Now instead of a being a constant, we can get DNN to figure out what it should be:

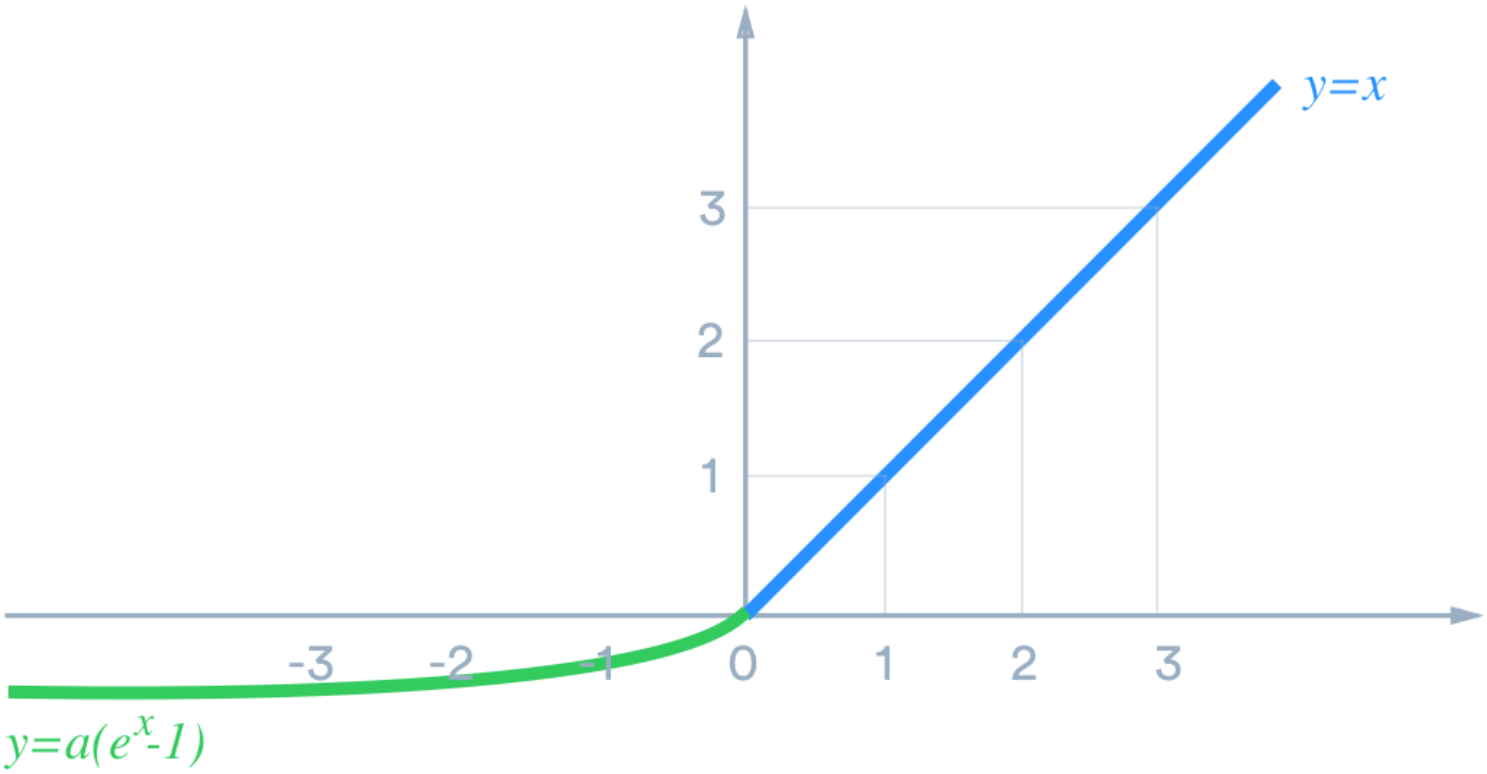



And then came a sequence of papers, each just adding a character before ELU and creating new activation functions.

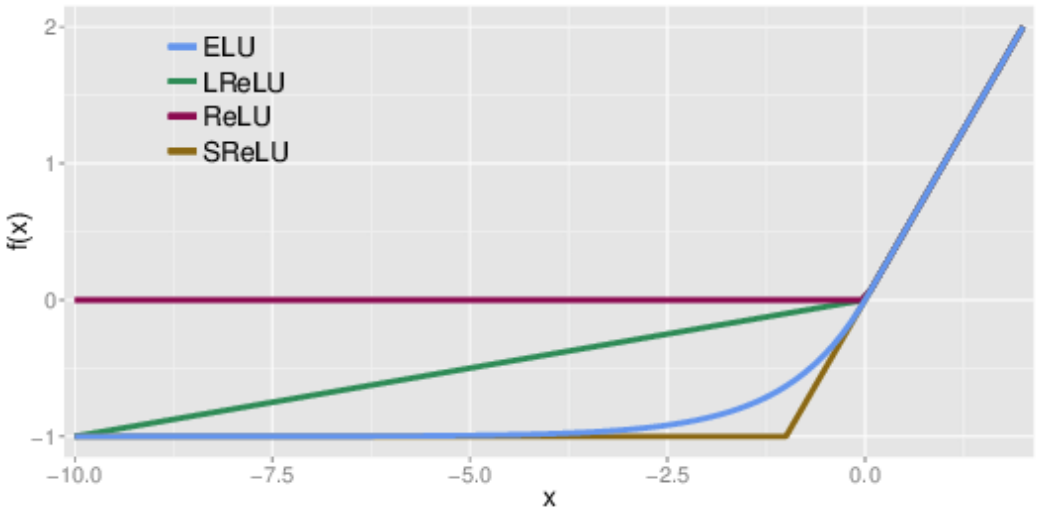
SELU



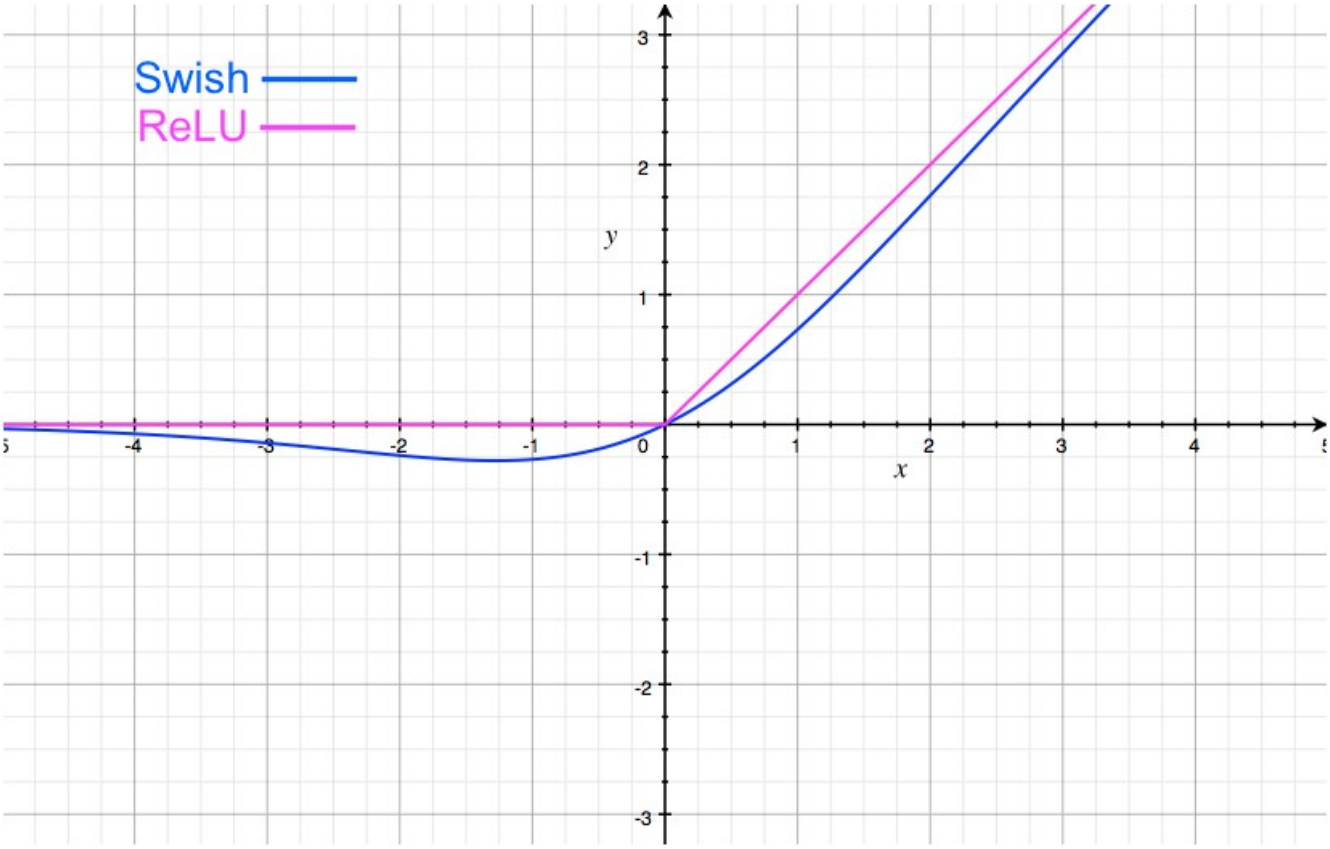

ELU


y=x
3
2
1
-3
-2
-1
0
1
2
3
y=a(e^x-1)

SReLU

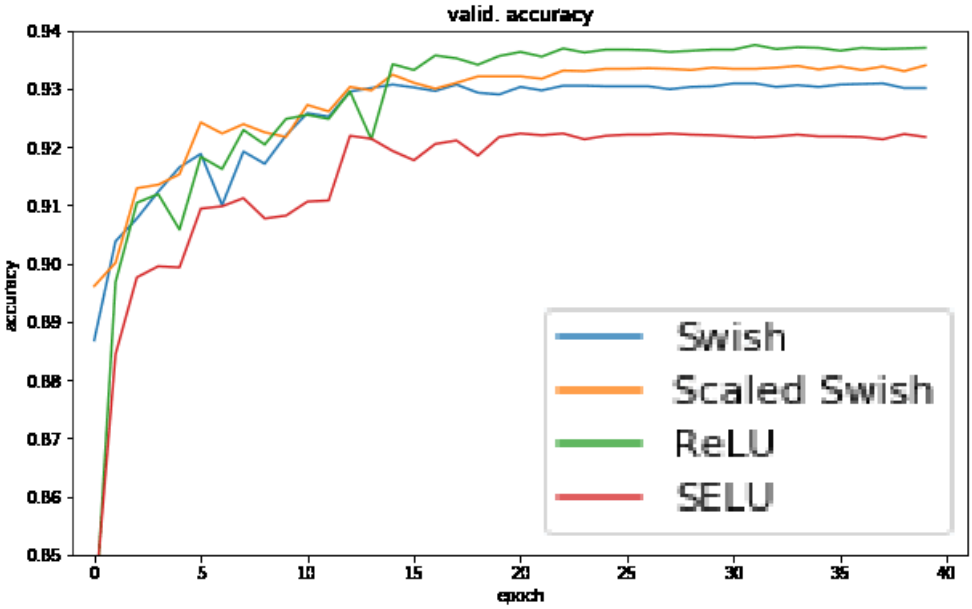

ELU
LReLU
ReLU
SReLU
f(x)
2
1
0
−1
−10.0
−7.5
−5.0
−2.5
0.0
x

## Swish




## So which activation function to use?

x




ReLU

Most of the time the innovators of these new Activation functions are using ReLU in later papers, that encouraging. There is no clear proof of which one is better. The innovators claim their's is better, but then later peer group would release their studies which claim otherwise.

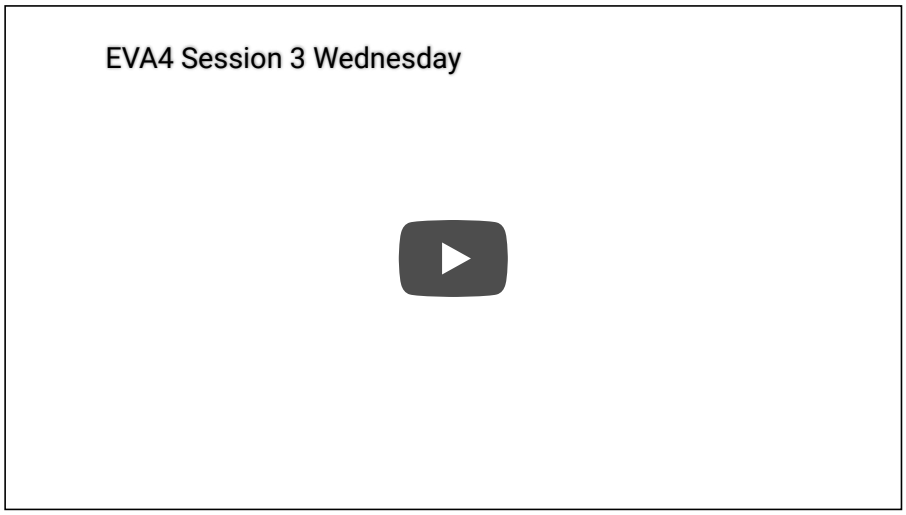
ReLU is simple, efficient and fast, and even if anyone of the above is better, we are talking about a marginal benefit, with an increase in computation. We'll stick to ReLU in our course. Also, NVIDIA has acceleration for ReLU activation, so that helps too.

# Assignment

1. Open this **LINK** (http://bit.ly/2IBqQJD) in your browser

2. Copy the file to your collaboratory
3. You need to:
    1. write comments for all the cells
    2. define a new network such that:
        1. it has less than 20000 parameters
        2. it achieves validation accuracy of more than 99.4% (basically print(score) should be more than 0.994
4. Once done, upload the link to your Github Project to LMS.

**Video for Session 3 - Coming Soon.**




Quiz - Assignment

- Refer to this **Colab File** **(https://colab.research.google.com/drive/1WVQW6ziXMQdAlzOKsB-_-v4nQ7X-65EO)**. You need to go through this file, perform tons of experiments and then proceed to answer the questions in Quiz Q3. Please note that you only have 20 minutes to take the quiz, so you won't have time to answer the question by performing the experiments while the quiz is in progress. So work on the Colab file first, understand the code, try and fix bugs, if any, and then proceed to answer the quiz questions.