Session 2 & Assignment

Published | Edit | ⋮

Neural Network Architectures and Receptive Field

Where were we in the Session 1?

We stopped here:

400x400x3 | (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 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".
7. We are again following the same pattern of the increasing number of kernels from 32 to 512 in the second convolution block for the same reason we picked this pattern in the first block.
8. Again 512 kernels in the second convolution block are important for us as we need a large number of textures to be able to define all out images.

We 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.

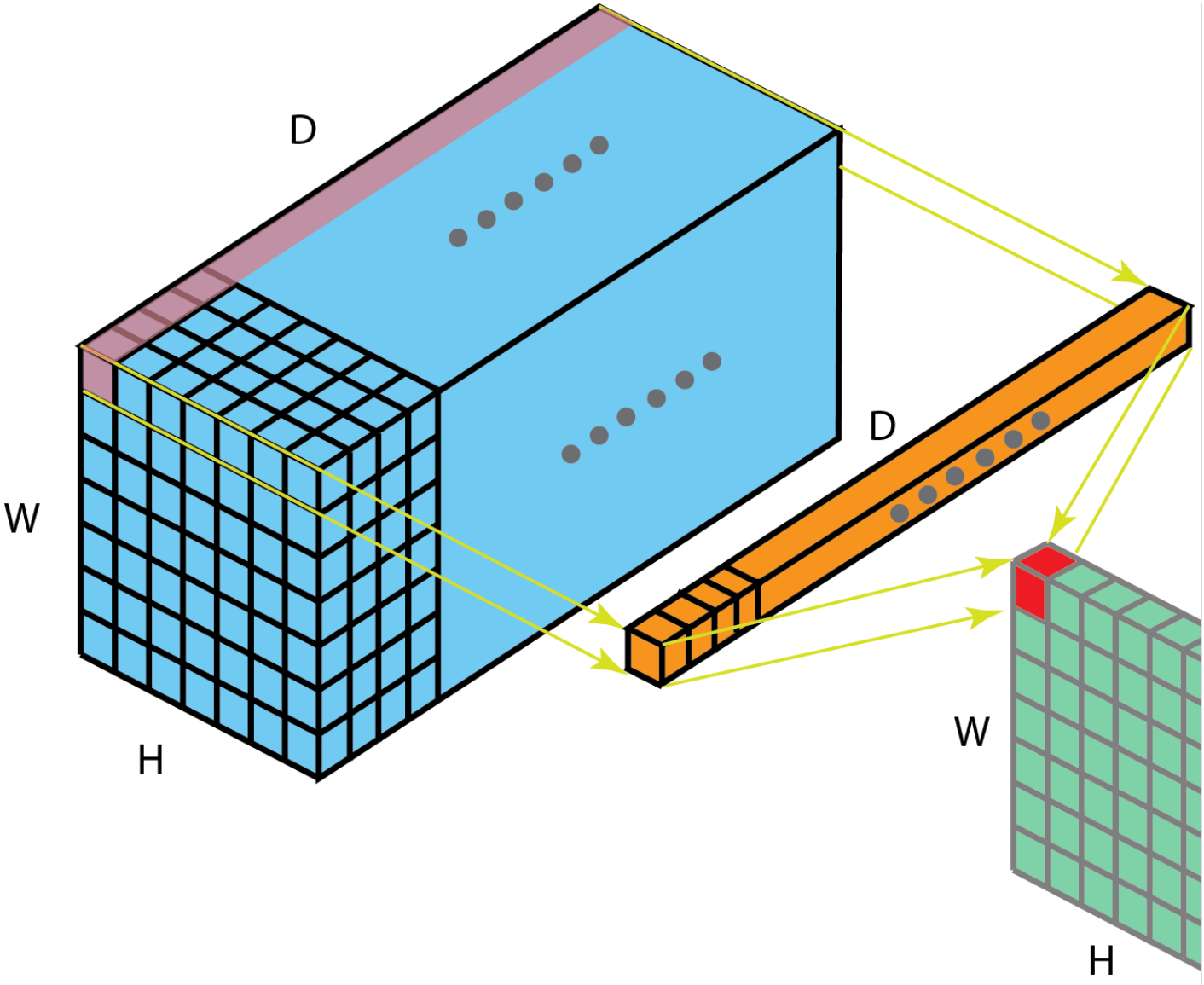
Think about these few points:

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).

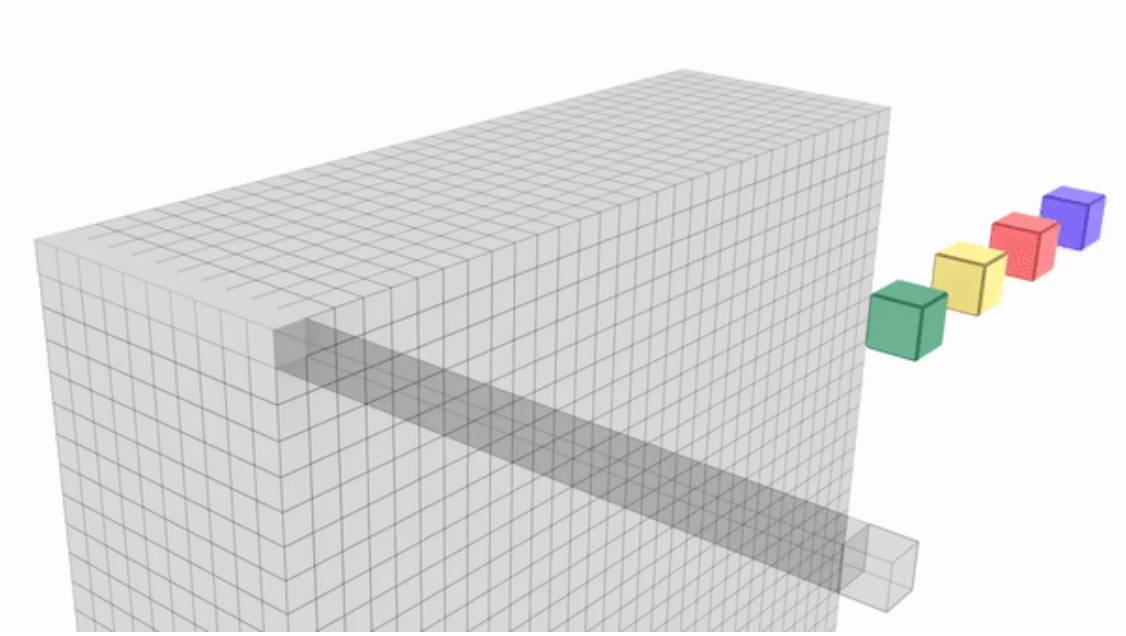
1x1 provides all these features.

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 which 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.

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 this 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:

400x400x3 | (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

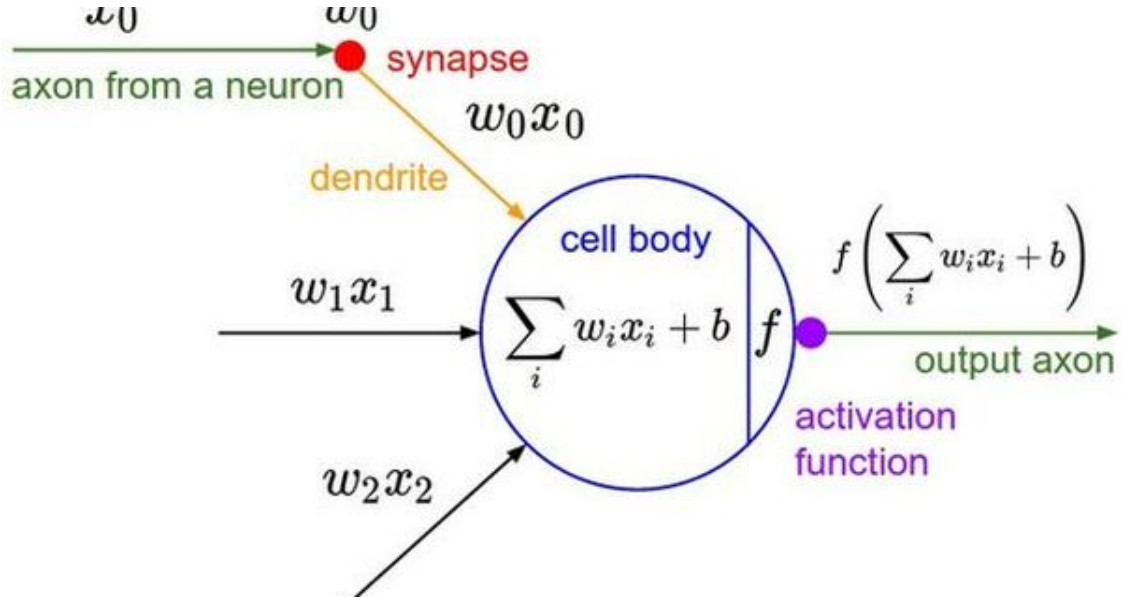
TRANSITION BLOCK 2 BEGINS
MAXPOOLING(2x2)
93x93x512 | **(1x1x512)x32** | 93x93x32 RF of 60x60
TRANSITION BLOCK 2 ENDS

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.

Activation Function

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 AIML community occupied in 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.

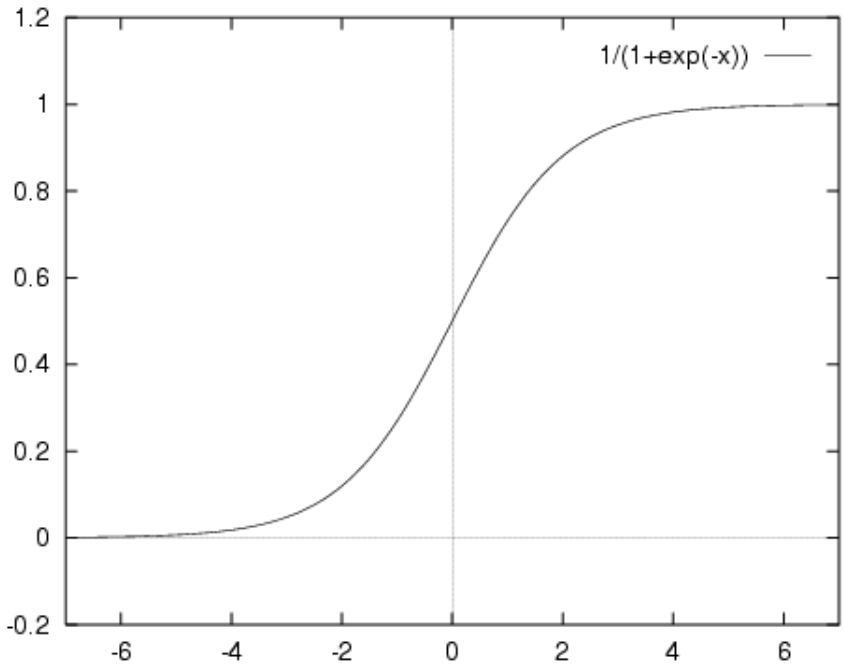
Why do we need an activation function?

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 times. We want our Neural Network to not just learn and compute a linear function but something more complicated than that.

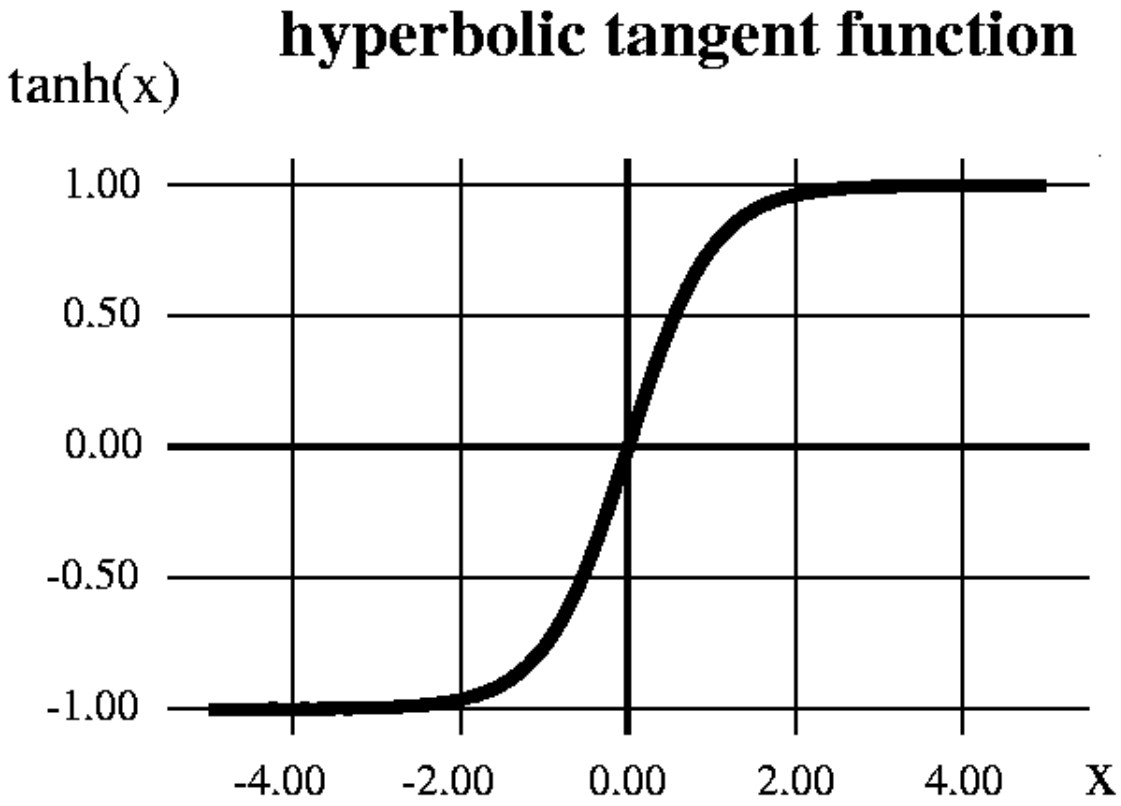
Read More **(https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f)**

Sigmoid

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



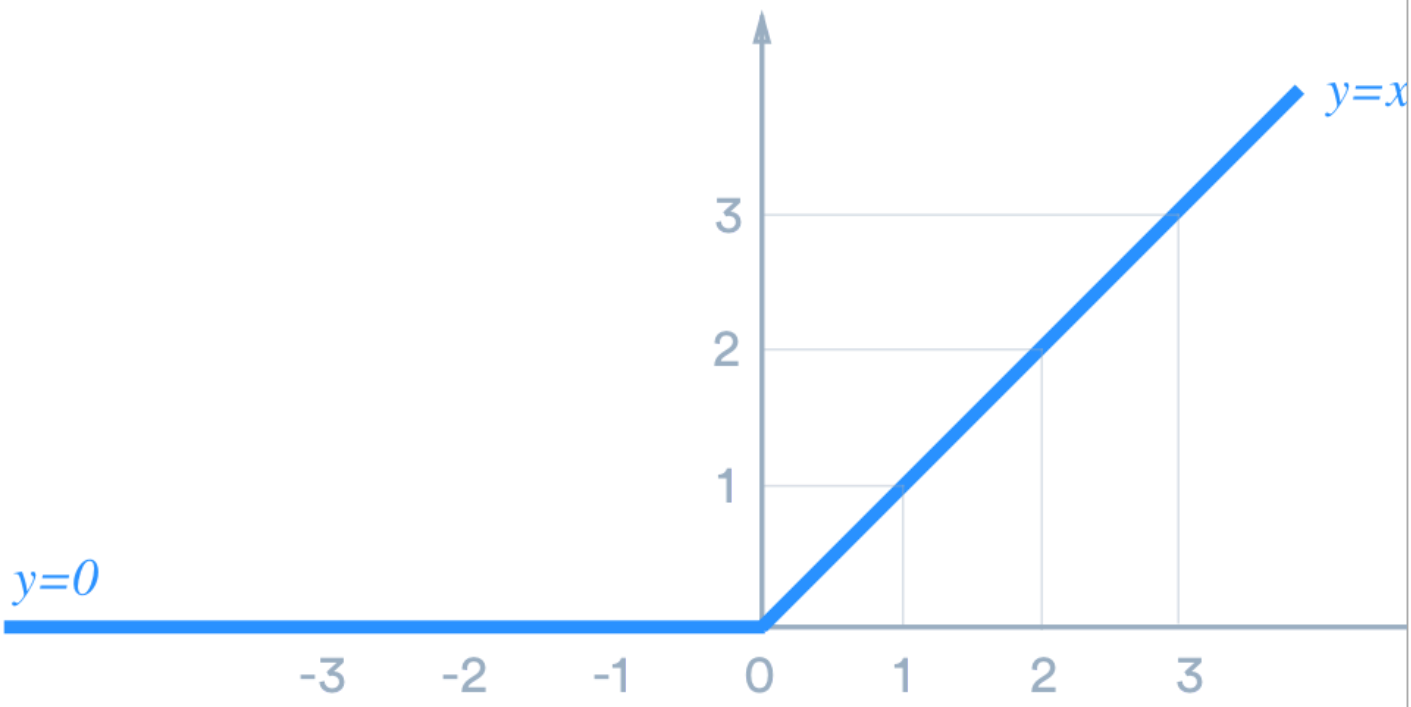
When sigmoids didn't work, we felt maybe another complicated function, called TanH might work:



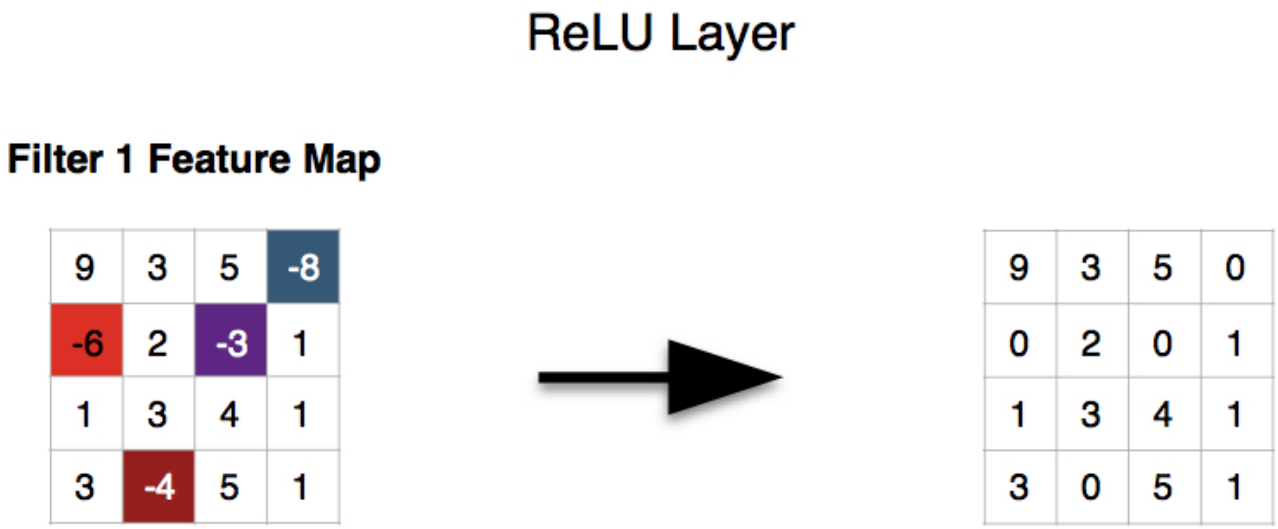
But neither did.

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's 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.



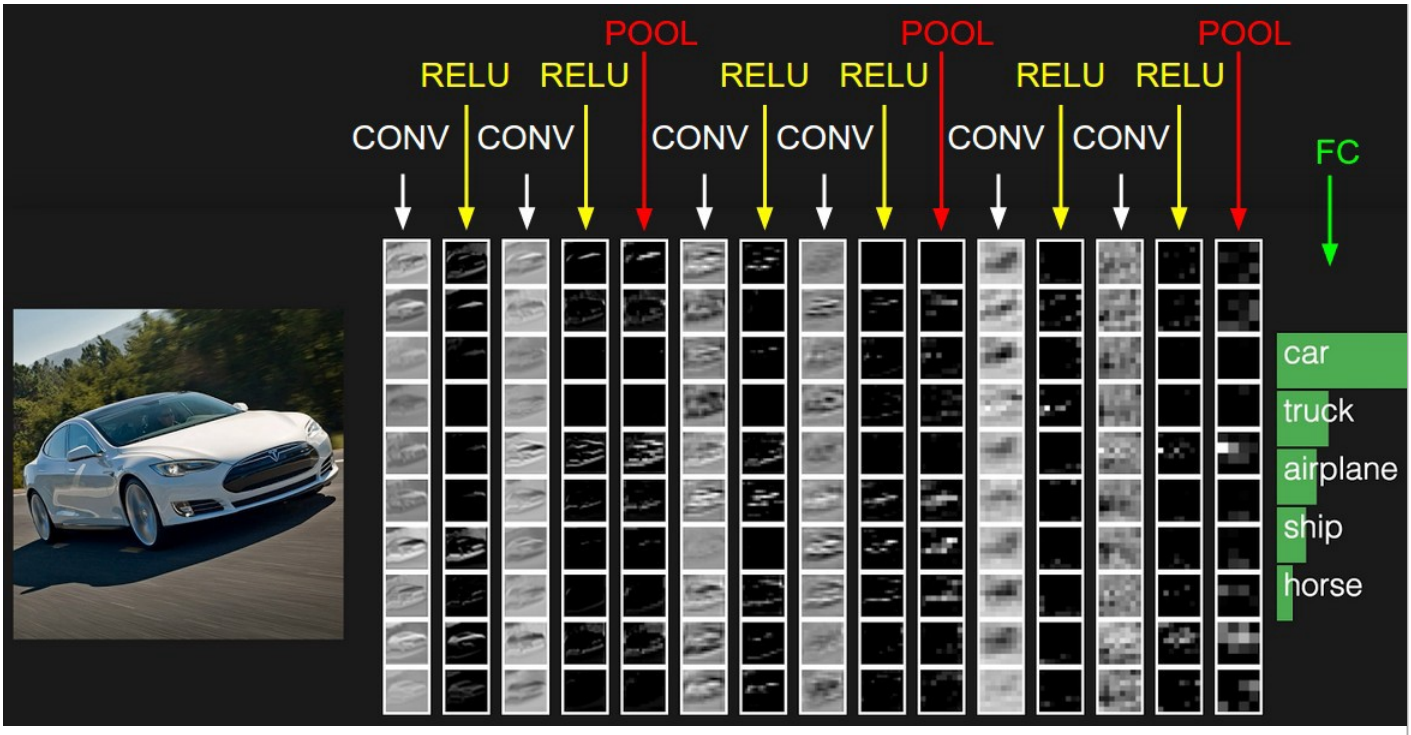
ReLU since has worked wonders for us. It is fast, simple and efficient.

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) = max(0,x) i.e if x < 0 , R(x) = 0 and if x >= 0 , R(x) = x.*

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

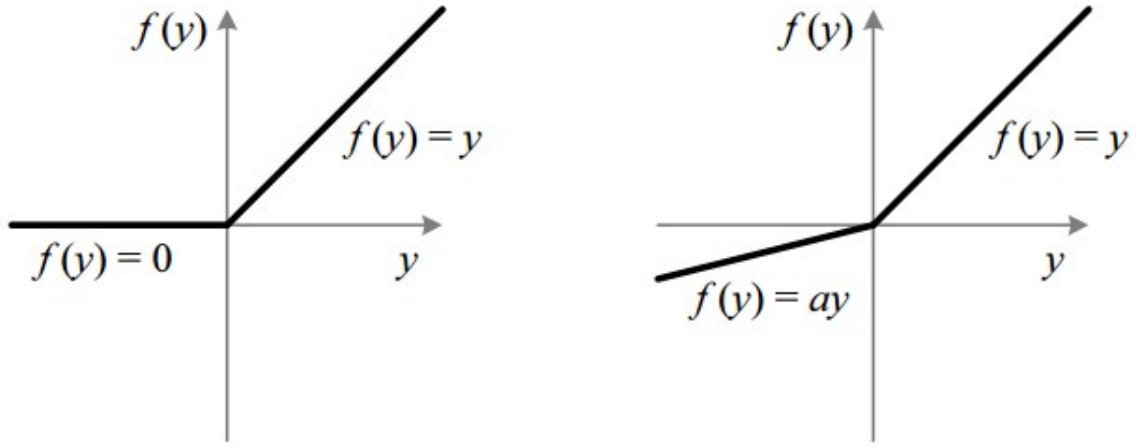
- It's cheap to compute as there is no complicated math. The model can therefore take less time to train or run.
- It converges faster. Linearity means that the slope doesn't plateau, or "saturate," when *x* gets large. It doesn't have the vanishing gradient problem suffered by other activation functions 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:

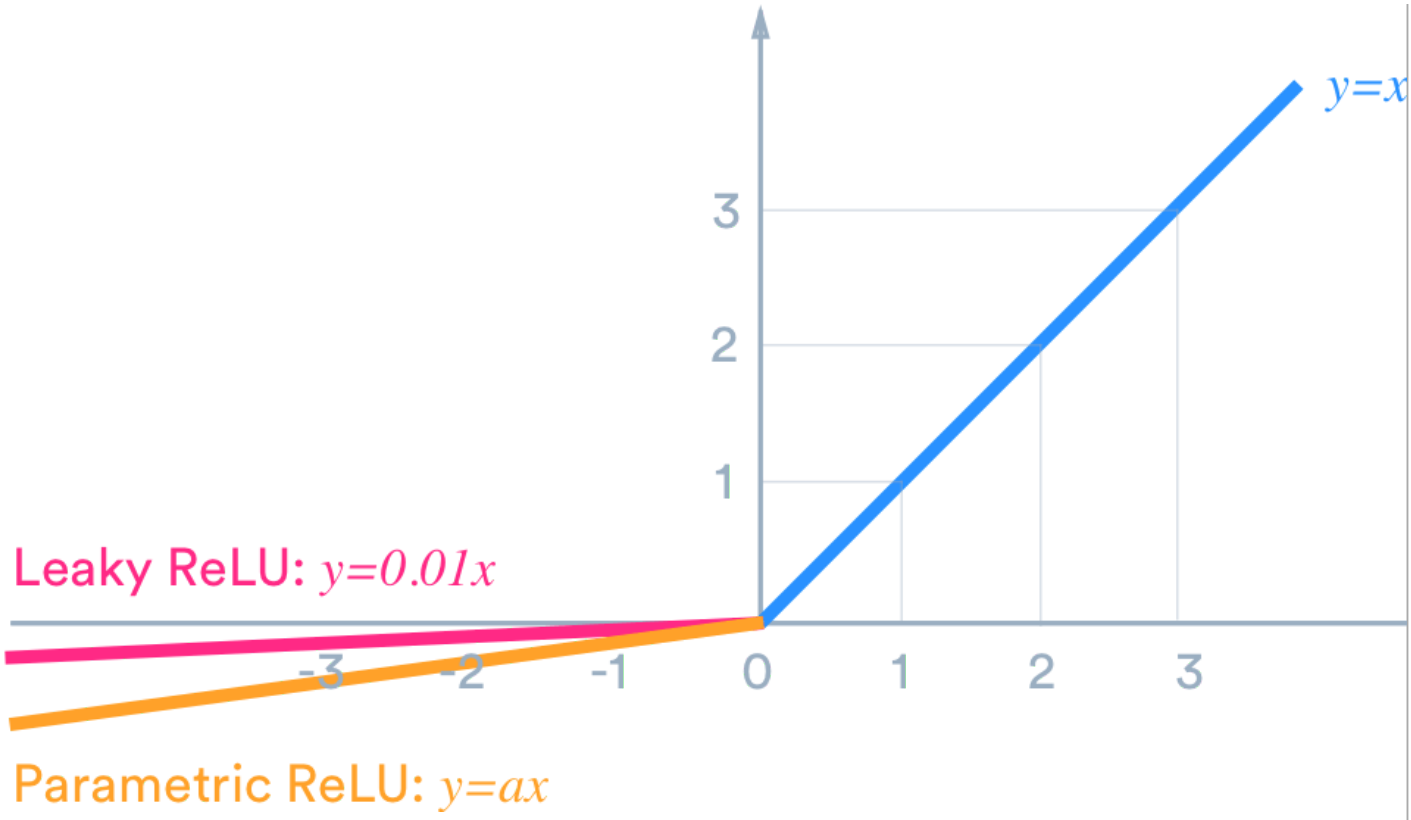


But aren't humans known to complicate things? One might wonder, why such partiality to all the negative numbers? What if we allow a negative number to *leak* a bit of their values. Hence came **Leaky ReLU**

ReLU vs LeakyReLU

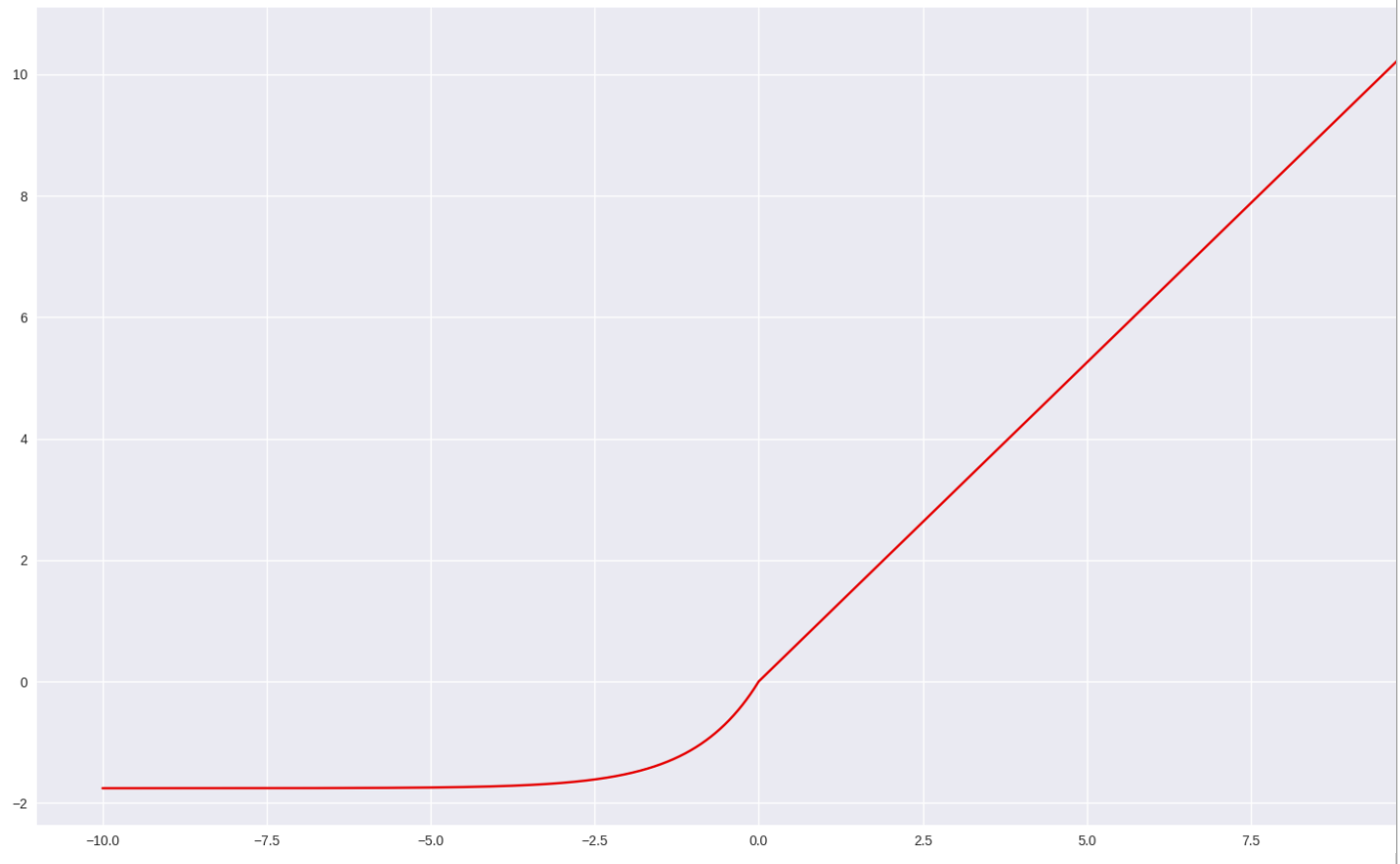


And then why stop there. You can complicate this further. Now instead of a being a constant we can get DNN to figure out what it should be:

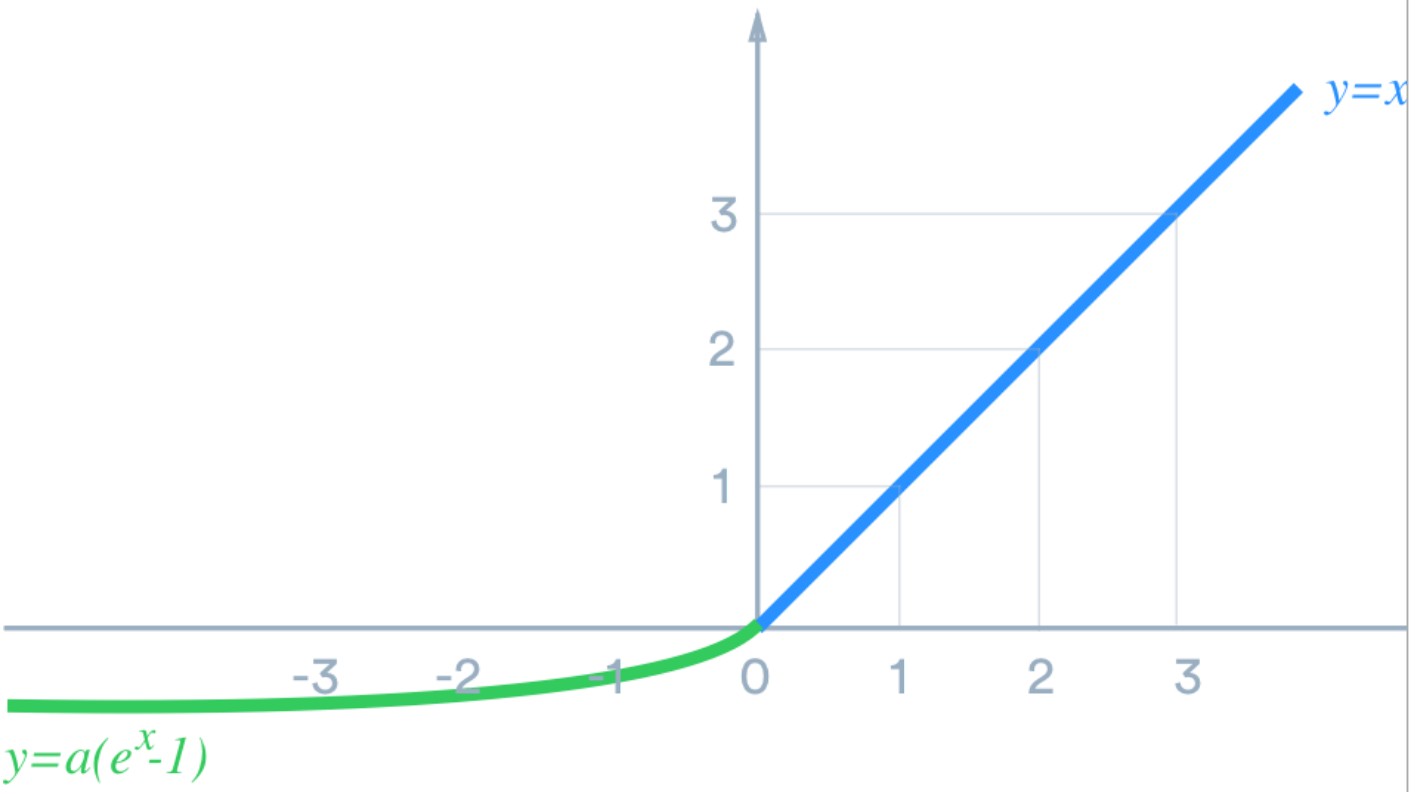


And then came a sequence of papers, each just adding a character before ELU and creating new activation functions.

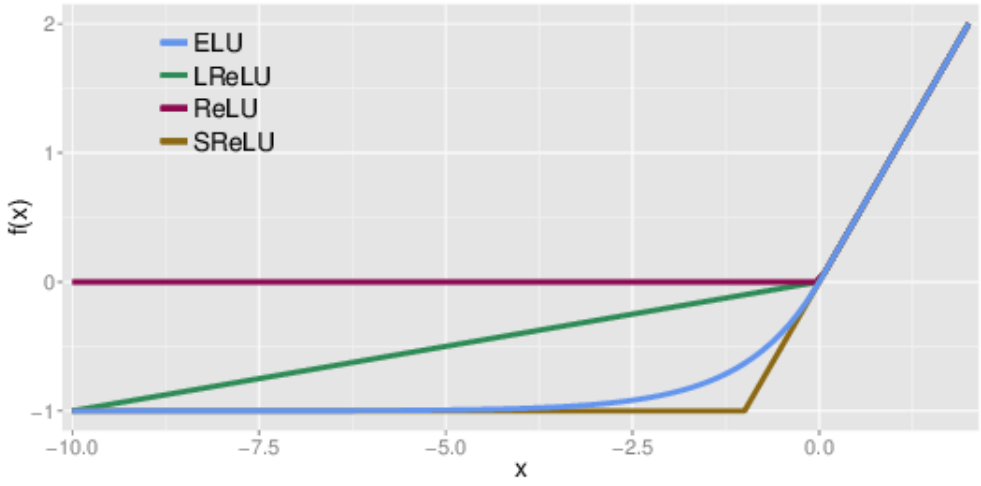
SELU



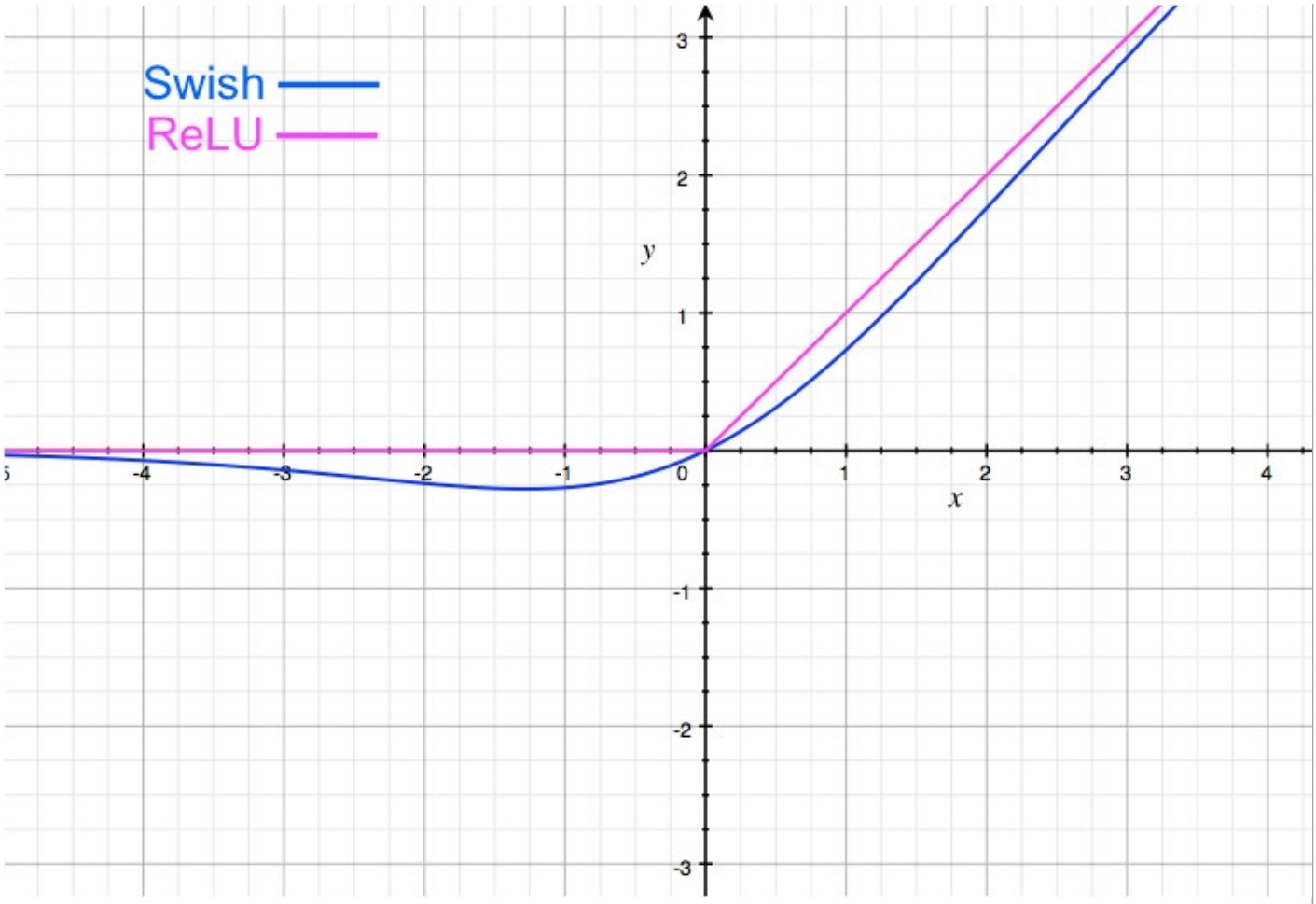
ELU



SReLU



Swish



So which activation function to use?

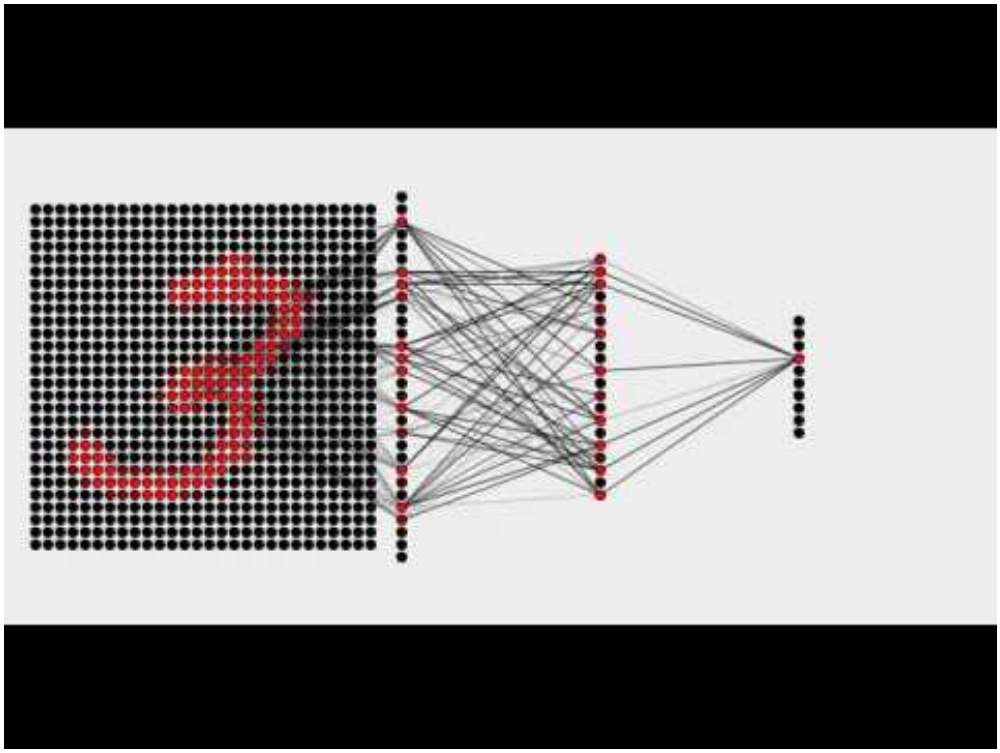
ReLU

Most of the times 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.

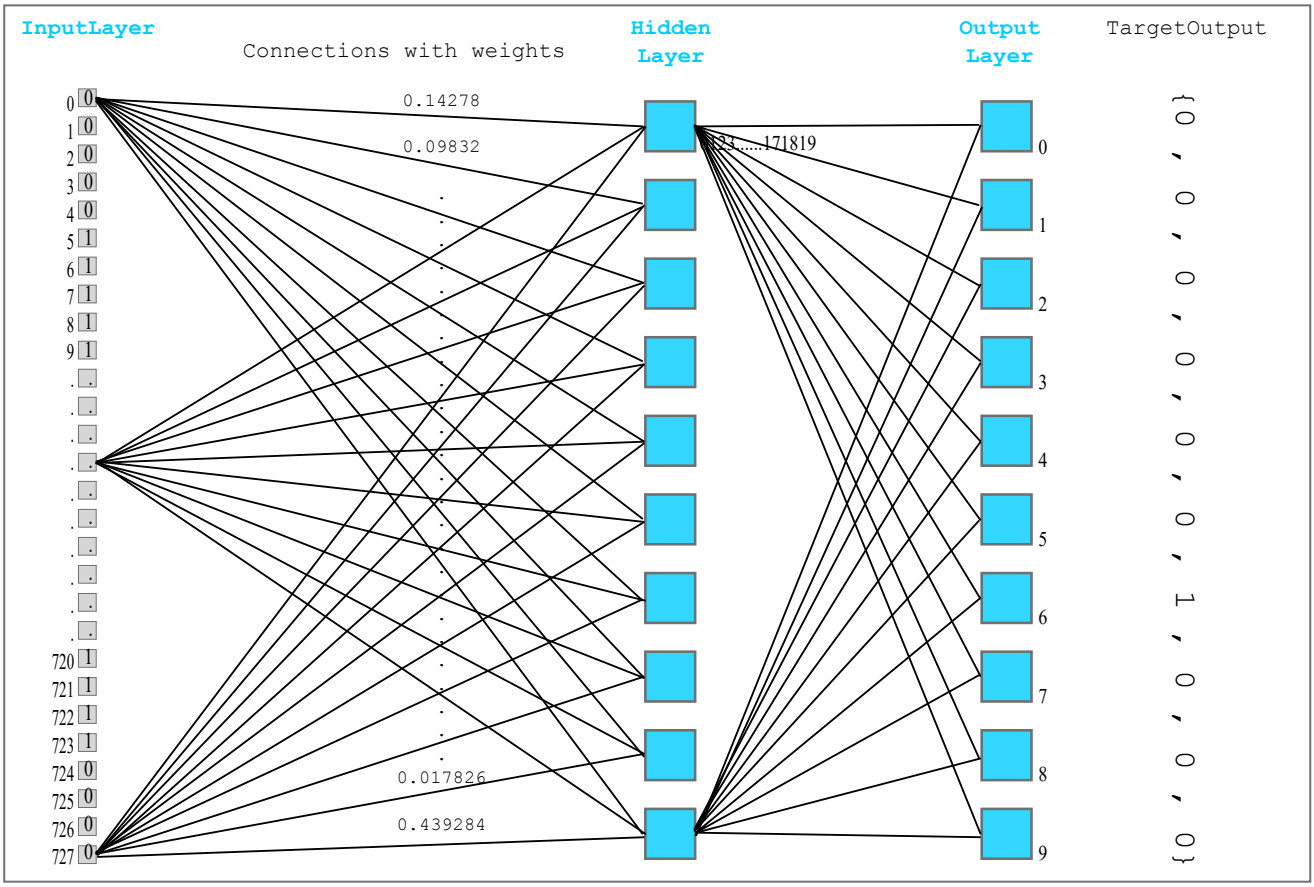
ReLU is simple, efficient and fast, and even if any one of the above is better, we are talking about a marginal benefit, with an increase in computation. We'll stick to ReLU in our course. Also NVIDIA has acceleration for ReLU activation, so that helps too.

Fully Connected Layers



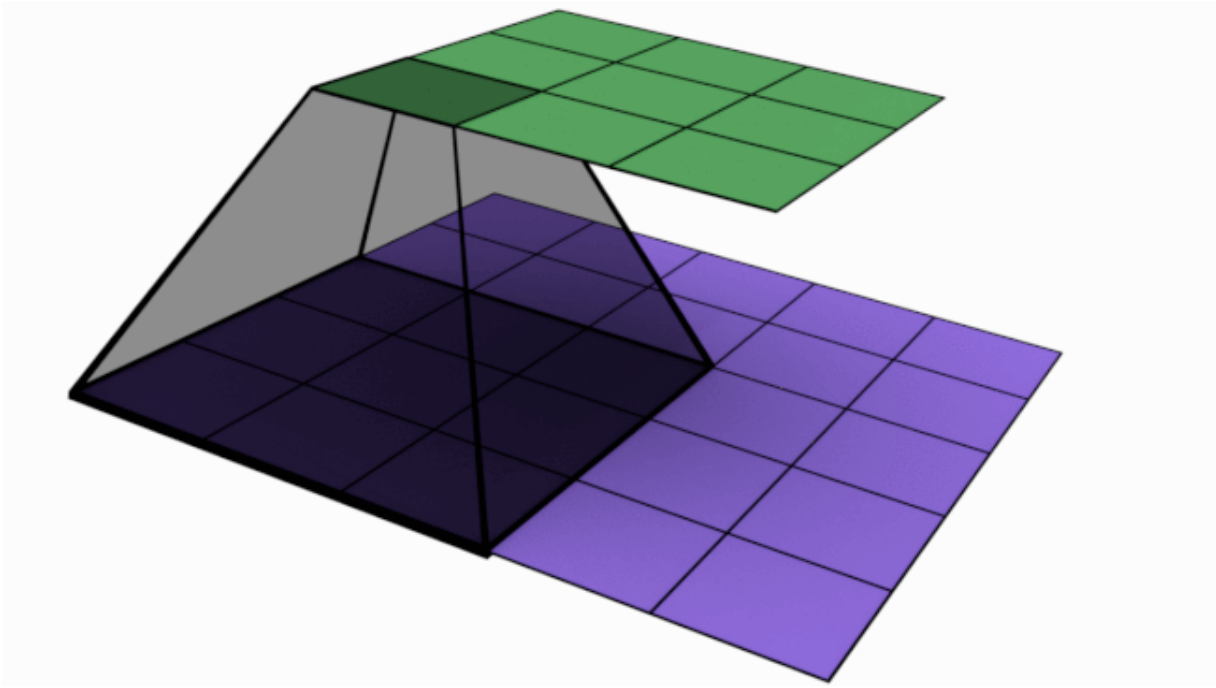
As you can see we lose the spatial information when we use fully connected layers.

Let's talk about this network below:

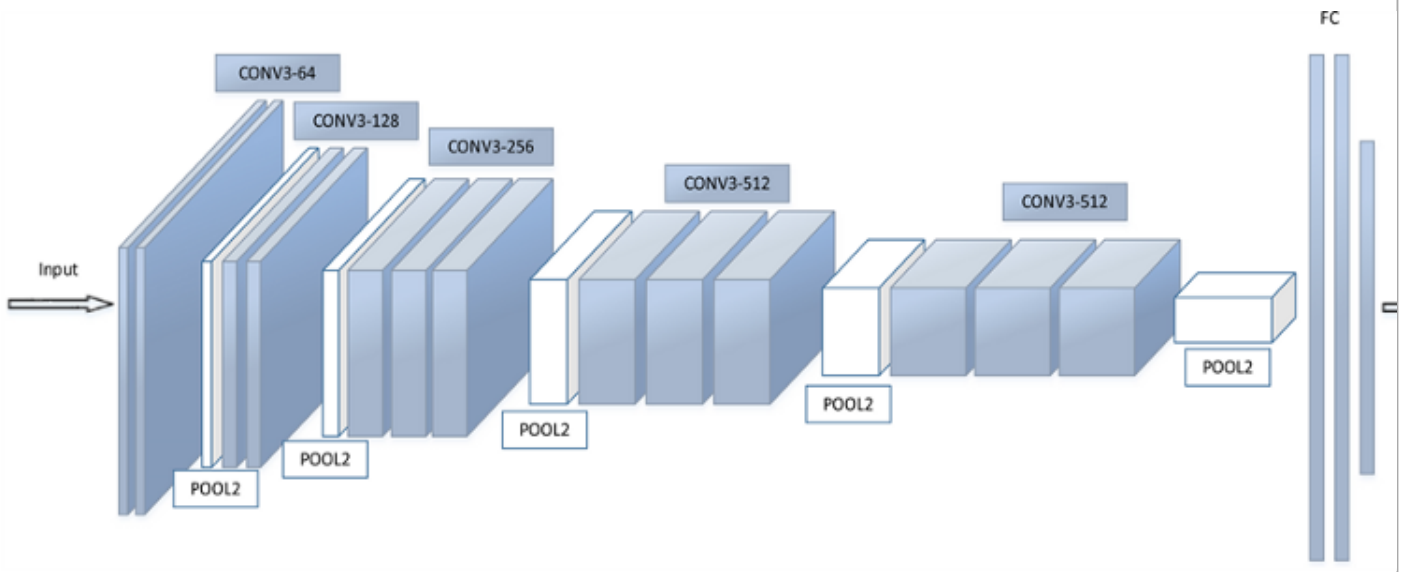


This is a simple three layer network, input neuron layer, hidden neuron layer and output neuron layers.

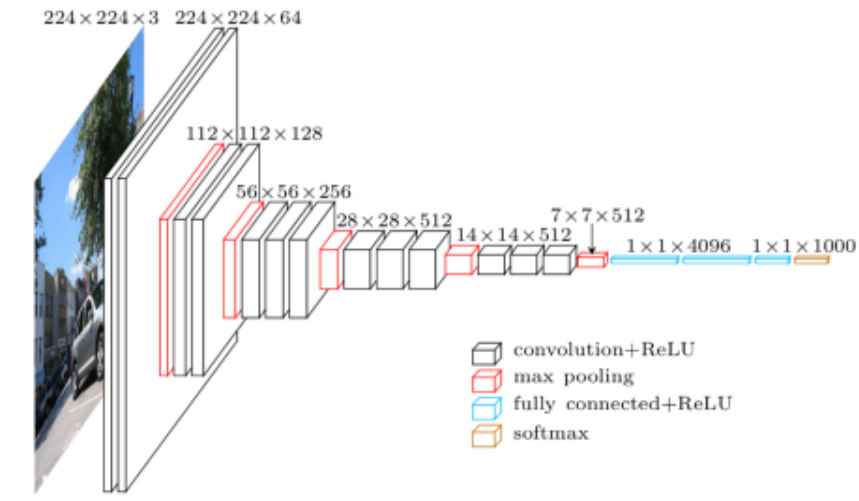
Getting old, and new kid on the block



Before 2014



In the background



Let's look at the parameters

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 224, 224, 3)	0	
block1_conv1 (Convolution2D)	(None, 224, 224, 64)	1792	input_1[0][0]
block1_conv2 (Convolution2D)	(None, 224, 224, 64)	36928	block1_conv1[0][0]
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0	block1_conv2[0][0]
block2_conv1 (Convolution2D)	(None, 112, 112, 128)	73856	block1_pool[0][0]
block2_conv2 (Convolution2D)	(None, 112, 112, 128)	147584	block2_conv1[0][0]
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0	block2_conv2[0][0]
block3_conv1 (Convolution2D)	(None, 56, 56, 256)	295168	block2_pool[0][0]
block3_conv2 (Convolution2D)	(None, 56, 56, 256)	590080	block3_conv1[0][0]
block3_conv3 (Convolution2D)	(None, 56, 56, 256)	590080	block3_conv2[0][0]
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0	block3_conv3[0][0]
block4_conv1 (Convolution2D)	(None, 28, 28, 512)	1180160	block3_pool[0][0]
block4_conv2 (Convolution2D)	(None, 28, 28, 512)	2359808	block4_conv1[0][0]
block4_conv3 (Convolution2D)	(None, 28, 28, 512)	2359808	block4_conv2[0][0]
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0	block4_conv3[0][0]
block5_conv1 (Convolution2D)	(None, 14, 14, 512)	2359808	block4_pool[0][0]
block5_conv2 (Convolution2D)	(None, 14, 14, 512)	2359808	block5_conv1[0][0]
block5_conv3 (Convolution2D)	(None, 14, 14, 512)	2359808	block5_conv2[0][0]
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0	block5_conv3[0][0]
flatten (Flatten)	(None, 25088)	0	block5_pool[0][0]
fc1 (Dense)	(None, 4096)	102764544	flatten[0][0]
fc2 (Dense)	(None, 4096)	16781312	fc1[0][0]
predictions (Dense)	(None, 1000)	4097000	fc2[0][0]

Total params: 138357544

Softmax:

This is how our code looked like after the last session:

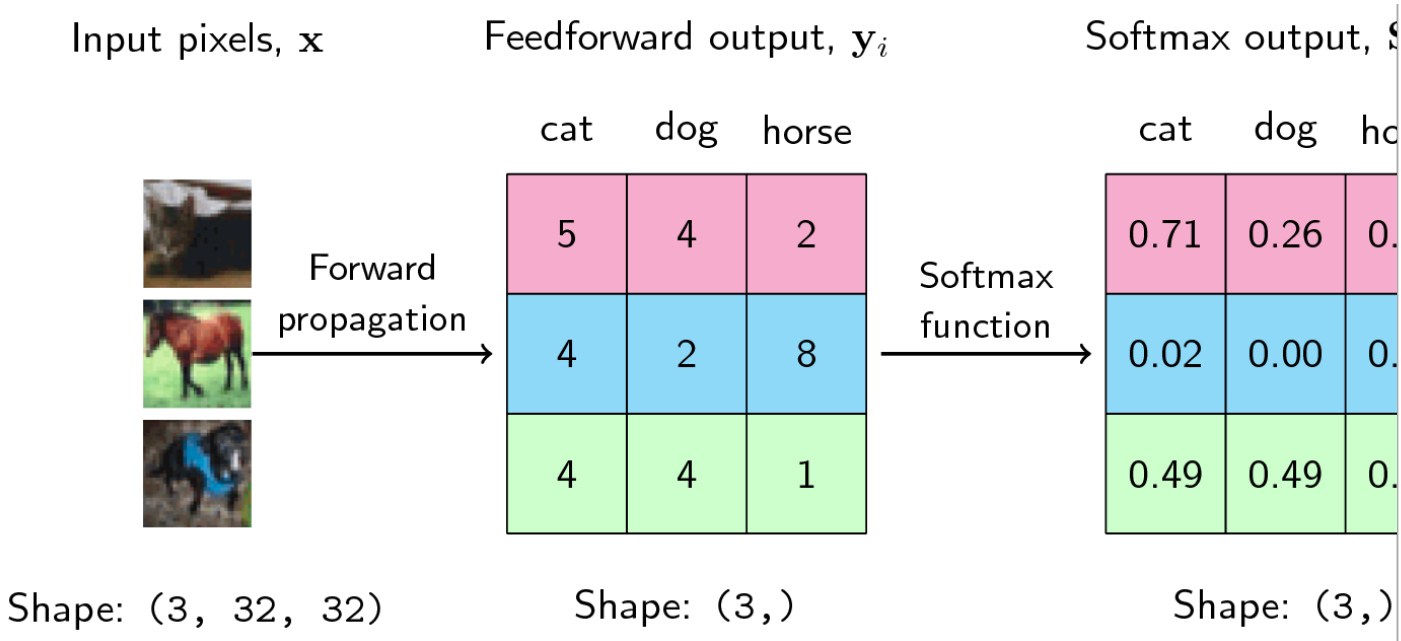
```
model.add(Convolution2D(32, 3, 3, activation='relu', input_shape=(28,28,1)))
model.add(Convolution2D(10, 1, activation='relu'))
model.add(Convolution2D(10, 26)) [1x1x10]
model.add(Flatten()) [10]
model.add(Activation('softmax'))
```

What is softmax?

The **softmax** function is often used in the final layer of a neural network-based classifier.

```
keras.layers.Softmax()
```

Let's look at what it actually does:



Why Softmax is not probability, but probability like!

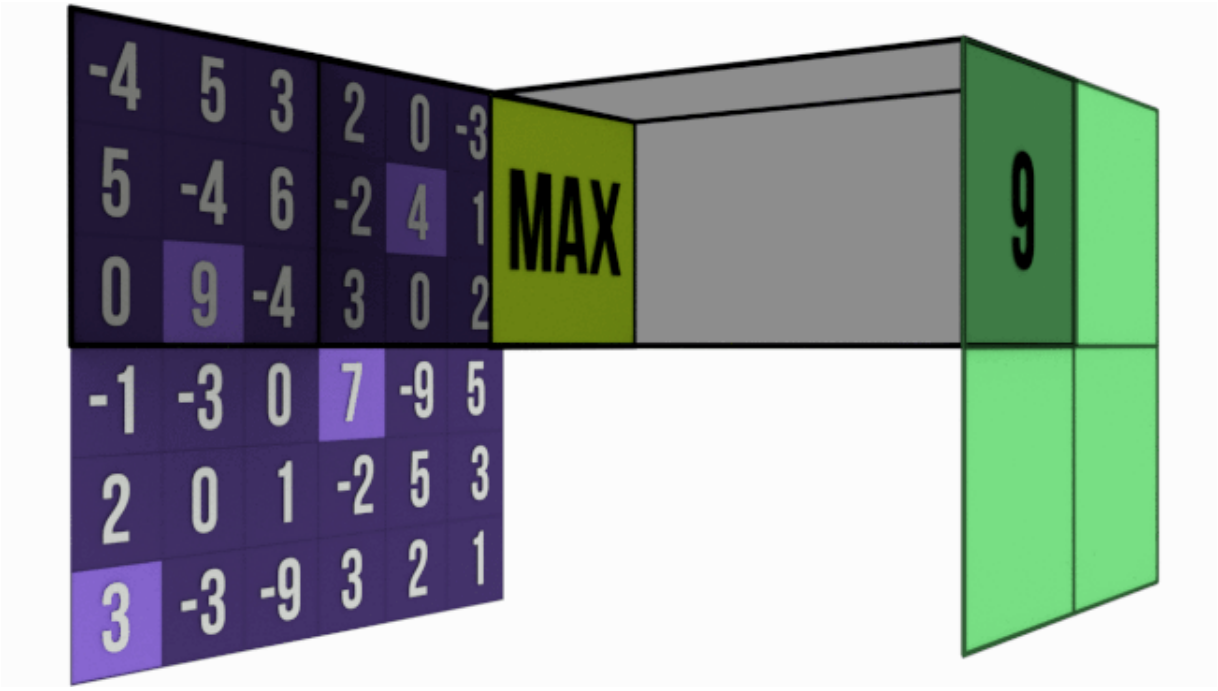
Let's Code:

First DNN: https://tinyurl.com/yyhddxw5 (https://tinyurl.com/yyhddxw5)

(https://tinyurl.com/yyhddxw5) 68k Params
98.28% Vacc

Second DNN: https://tinyurl.com/y248vr36 (https://tinyurl.com/y248vr36)

195k Params
99.18 Vacc

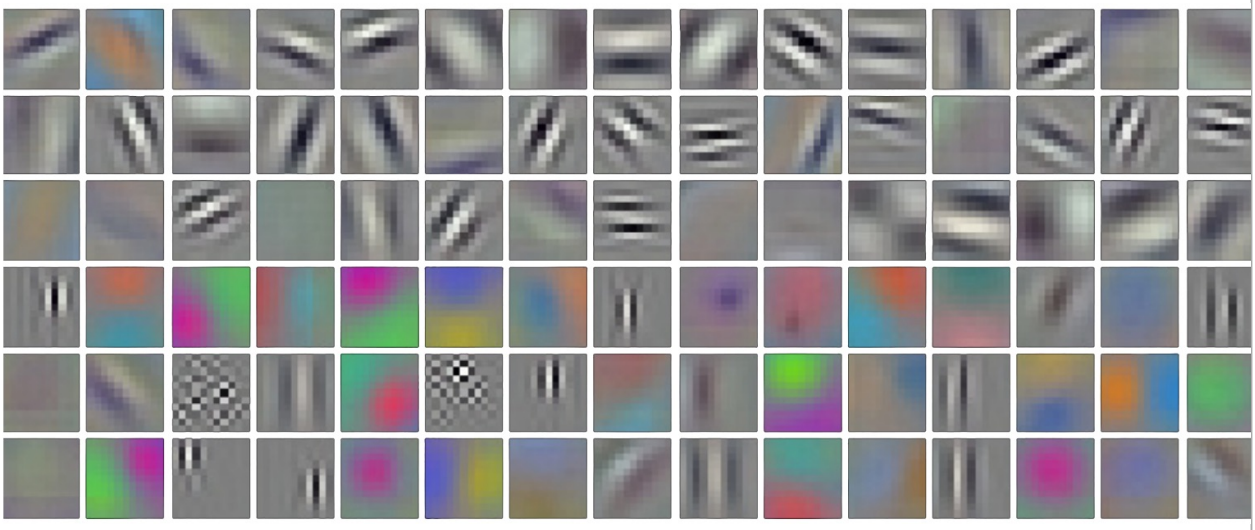


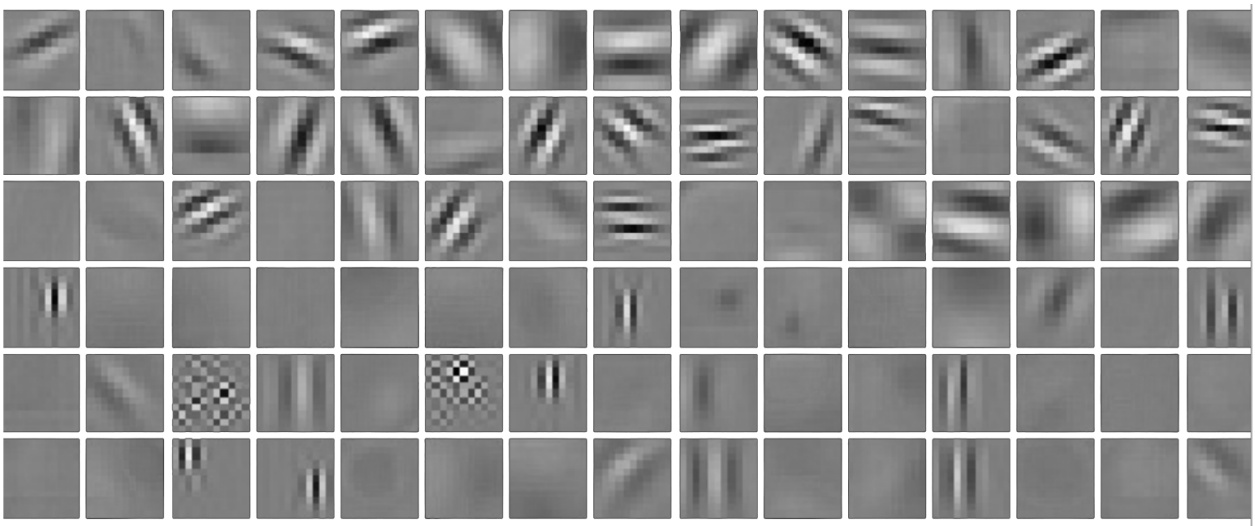
Third DNN: https://tinyurl.com/yyv4md9y (https://tinyurl.com/yyv4md9y)

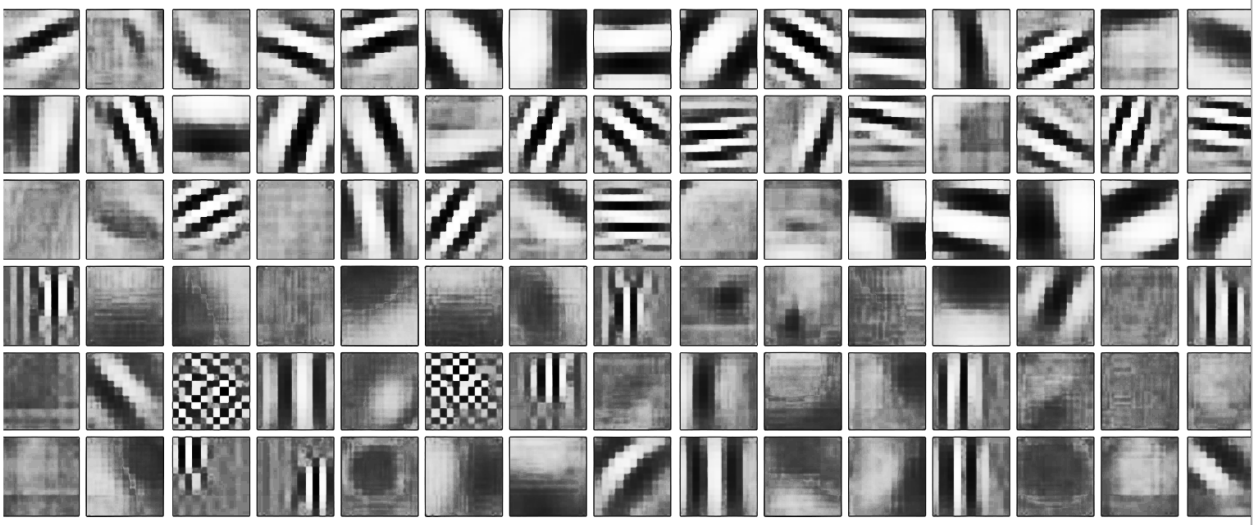
10k Params
98.84 Vacc

Fourth DNN: https://tinyurl.com/y65qk93c (https://tinyurl.com/y65qk93c)

11k Params
98.97 Vacc (10 Epoch)
99.08 Vacc (30 Epoch)







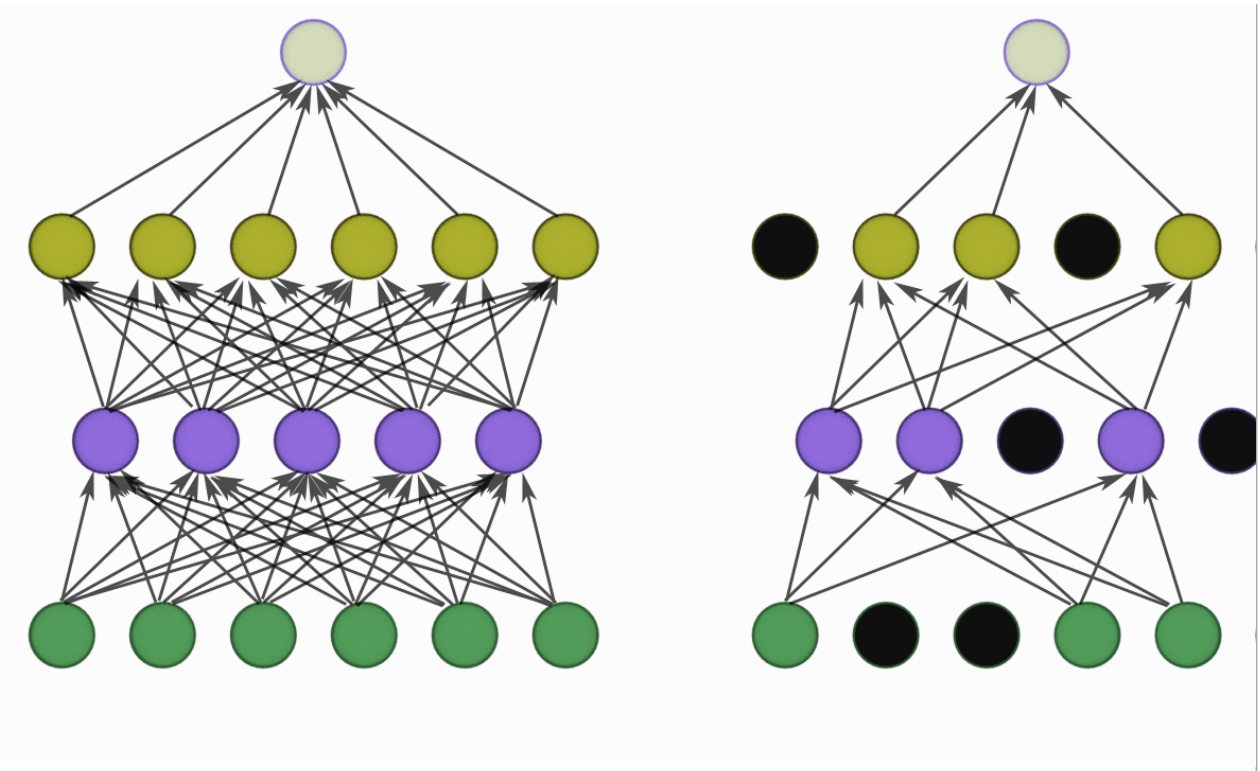
Fifth DNN: https://tinyurl.com/y2qacq7u (https://tinyurl.com/y2qacq7u)

18.4 Params
99.05 Vacc (20 Epoch)

Sixth DNN: https://tinyurl.com/yyfxyozm (https://tinyurl.com/yyfxyozm)

18.4 Params
99.4 Vacc (21 Epoch)

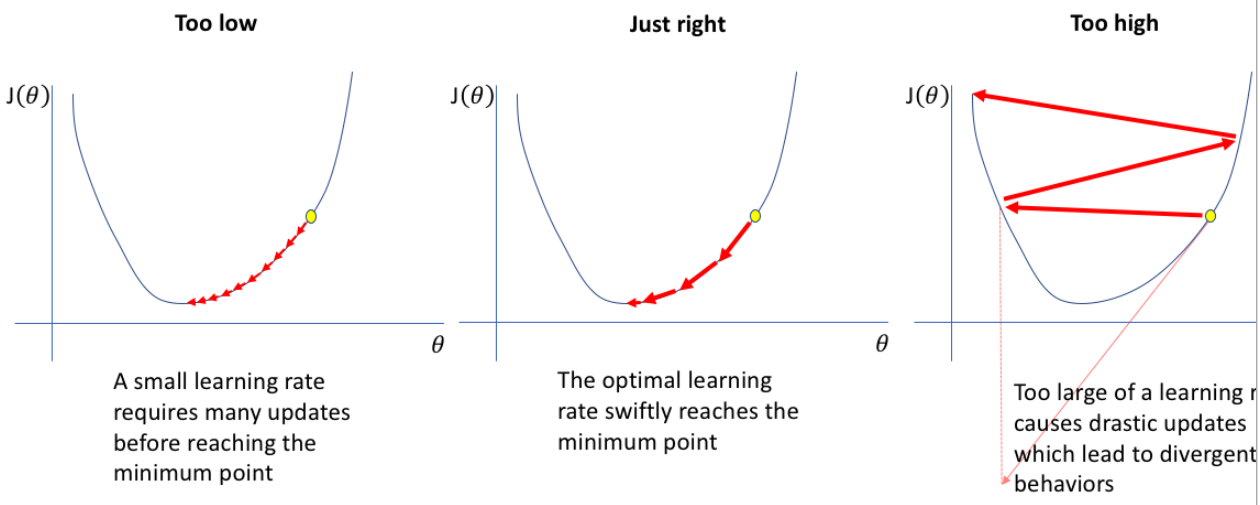
Drop Out



Seventh DNN: https://tinyurl.com/y2omp9s3

18.4 Params
99.42 Vacc (14th Epoch)

Learning Rate



EIGHTH: https://tinyurl.com/y494cv85 (https://tinyurl.com/y494cv85)

16.6k Params
99.46 VAcc (11th Epoch)

Assignment:

1. Write the CODE9 file to achieve:
    1. 99.4% accuracy
    2. in 15k parameters
    3. within 20 Epochs
2. You are not allowed to use:
    1. Fully connected layer to convert any 2D channel to 1D channel
    2. Use biases
3. Submit your GitHub link with a ReadMe.md file
4. In the ReadMe file:
    1. copy and paste your Logs for 20 epochs
    2. copy and paste the result of your model.evaluate (on test data)
    3. strategy you have taken to achieve the said results
5. Deadline is 1 hour before your next session starts.

Session 2 Video:

VIDEO IS NOT AVAILABLE AT THE MOMENT. YOU CAN REFER TO THIS VIDEO WHICH COVER NEARLY THE SAME CONTENT. THESE VIDEOS ARE FROM EVA COURSE

https://youtu.be/ul2-1KEHQsY (https://youtu.be/ul2-1KEHQsY)

(https://youtu.be/ul2-1KEHQsY)

Points 1,000

Submitting a website url

Due	For	Available from	Until
Nov 22 at 5:30am	F6	Nov 15 at 6:30am	Nov 23 at 5:30am
Nov 23 at 1:30pm	S12	Nov 16 at 12:30pm	Nov 23 at 1:30pm
Nov 25 at 5:30am	M6	Nov 18 at 6:30am	Nov 25 at 5:30am
Nov 27 at 5:30am	W6	Nov 20 at 6:30am	Nov 27 at 5:30am

+ **Rubric**