Session 4 - Building the First Neural Networks;

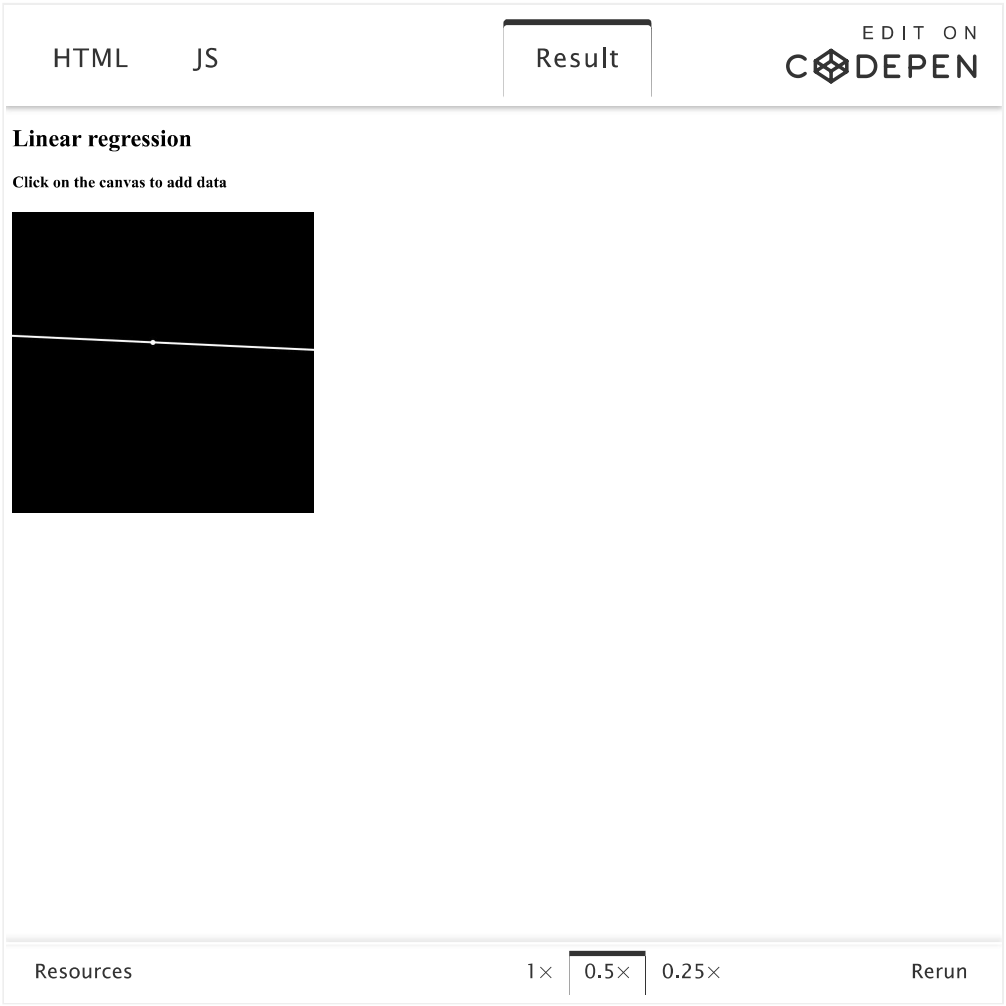
- Due Feb 17 by 9am
- Points 0
- Available after Feb 10 at 12am

Session 4 - Building First Neural Networks

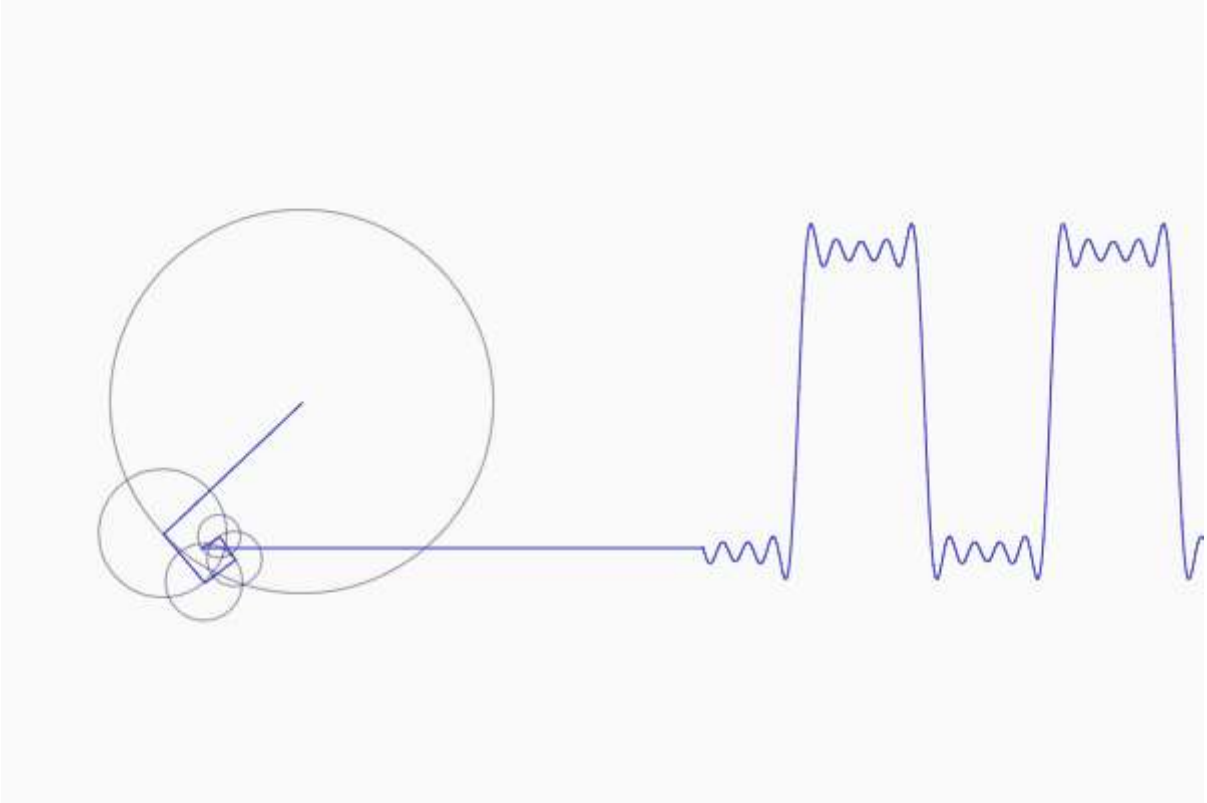
Weights, Kernels, Activations & Layers

Why do neural networks work so well?

Most of the problems in the world are not simple, i.e. real-world "data" cannot be modeled with a single line. Play with this tool below:



Most of the problems are non-linear in nature and we need non-linear tools. Just observe how difficult it is to get a perfect square wave using the Fourier transform in this image below:



Look at this amazing work by Jazzemon on **Fourier** **(https://www.jezzamon.com/fourier/)**

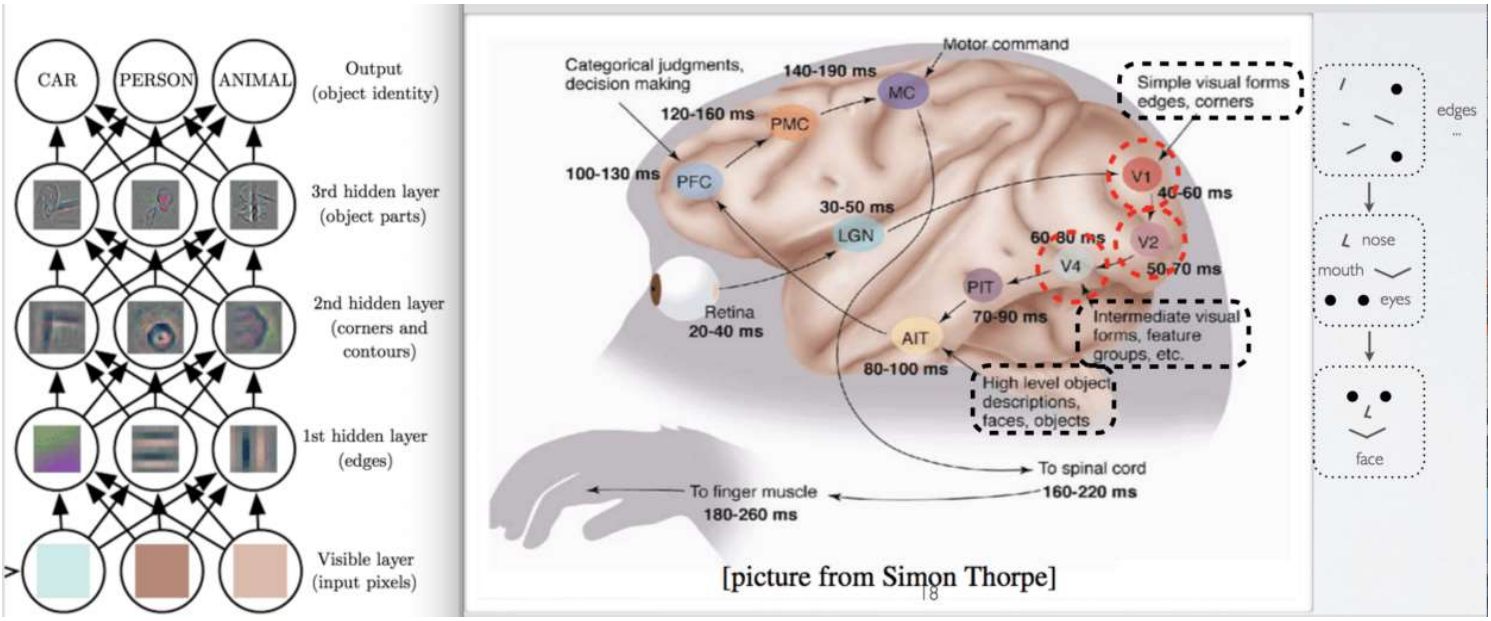
Neural Networks can have a large number of free parameters (or weights between interconnected units) and this gives them the flexibility to fit highly complex data (when trained correctly) that other models are too simple to fit.

This model complexity brings with it the problems of training such a complex network and ensuring the resultant model generalizes to the examples it is trained on. Typically neural networks ***require*** ***(https://stackoverflow.com/questions/38595451/why-do-neural-networks-work-so-well)*** large volumes of training data, that other models don't.

What kind of data are we dealing with?

Below you see a car:

This image is broken down into smaller "features" by the 4 layers in our visual cortex:



Only after an image is broken down into its features, our brain can combine it back into logical pieces:

Edges/Gradients (lines on the skin)

↓

Textures/Patterns (large hatch pattern)

↓

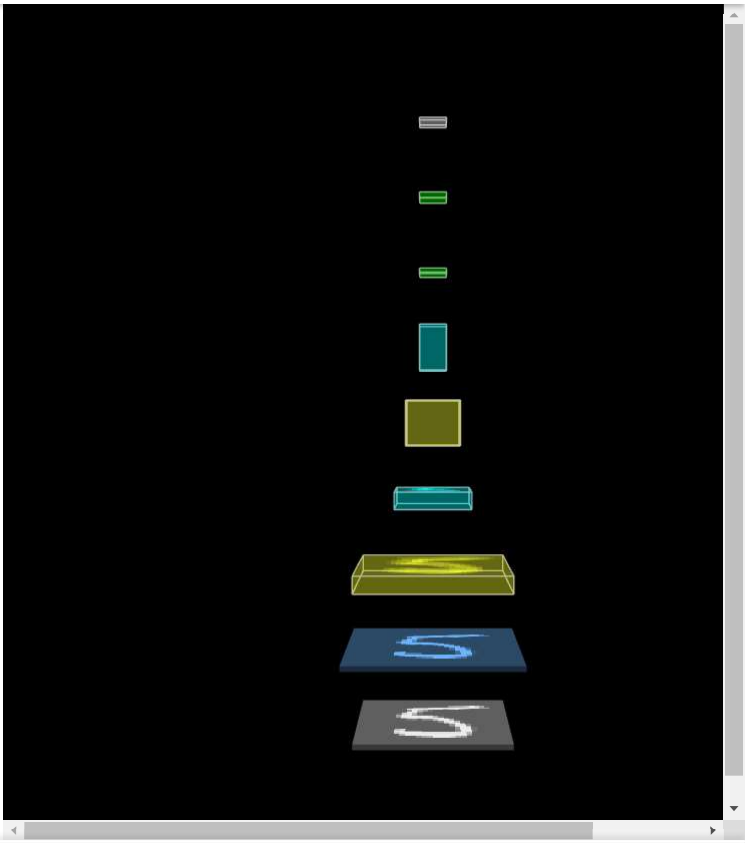
Part of Objects (head similar to crocodile)

↓

Objects (it's a dragon!)

HTML CSS JS Result EDIT ON CODEPEN

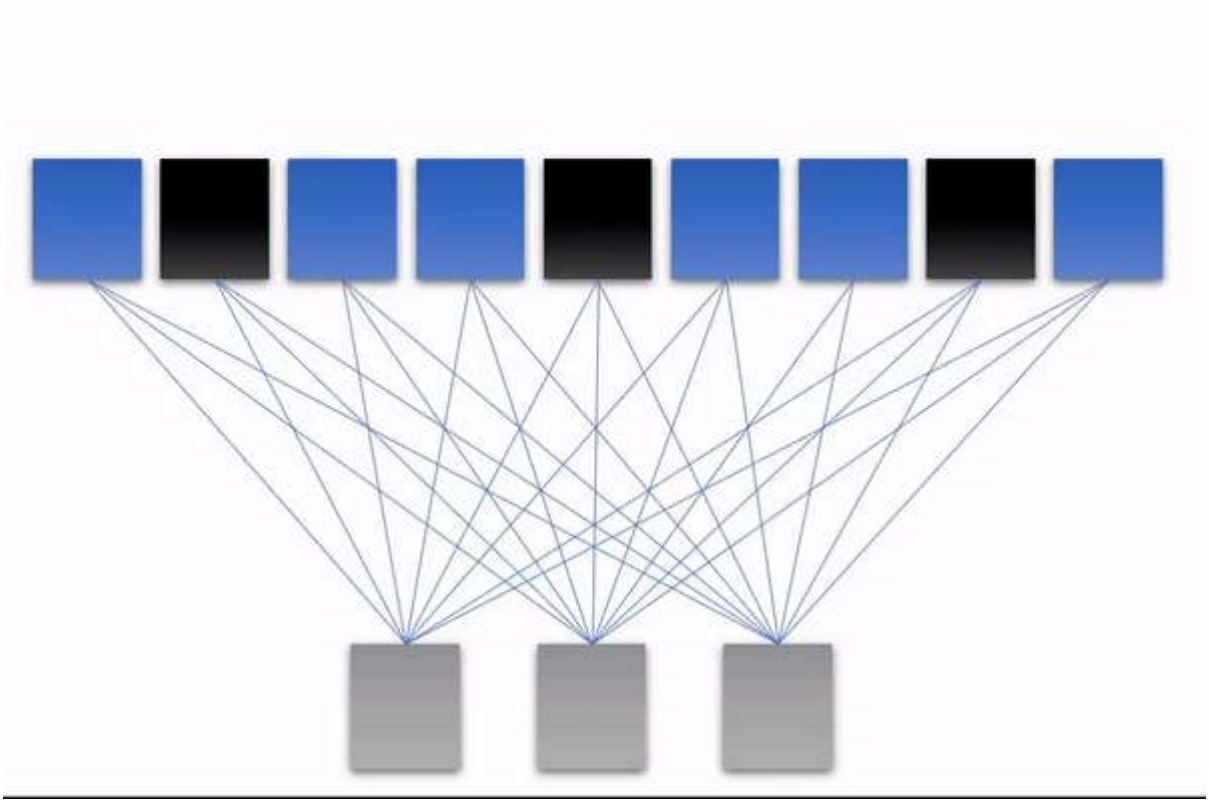
```
let modelContainer =
document.getElementById("container");
let model = new
TSP.models.Sequential(modelContainer);

model.add(new TSP.layers.GreyscaleInput());
model.add(new TSP.layers.Padding2d());
model.add(new TSP.layers.Conv2d());
model.add(new TSP.layers.Pooling2d());
model.add(new TSP.layers.Conv2d());
model.add(new TSP.layers.Pooling2d());
model.add(new TSP.layers.Dense());
model.add(new TSP.layers.Dense());
model.add(new TSP.layers.Output1d({
  outputs: ["0", "1", "2", "3", "4", "5", "6",
"7", "8", "9"]
}));

model.load({
  type: "tfjs",
  url:
"https://tensorspace.org/assets/model/lenet/mnist
```



Resources 1× 0.5× 0.25× Rerun

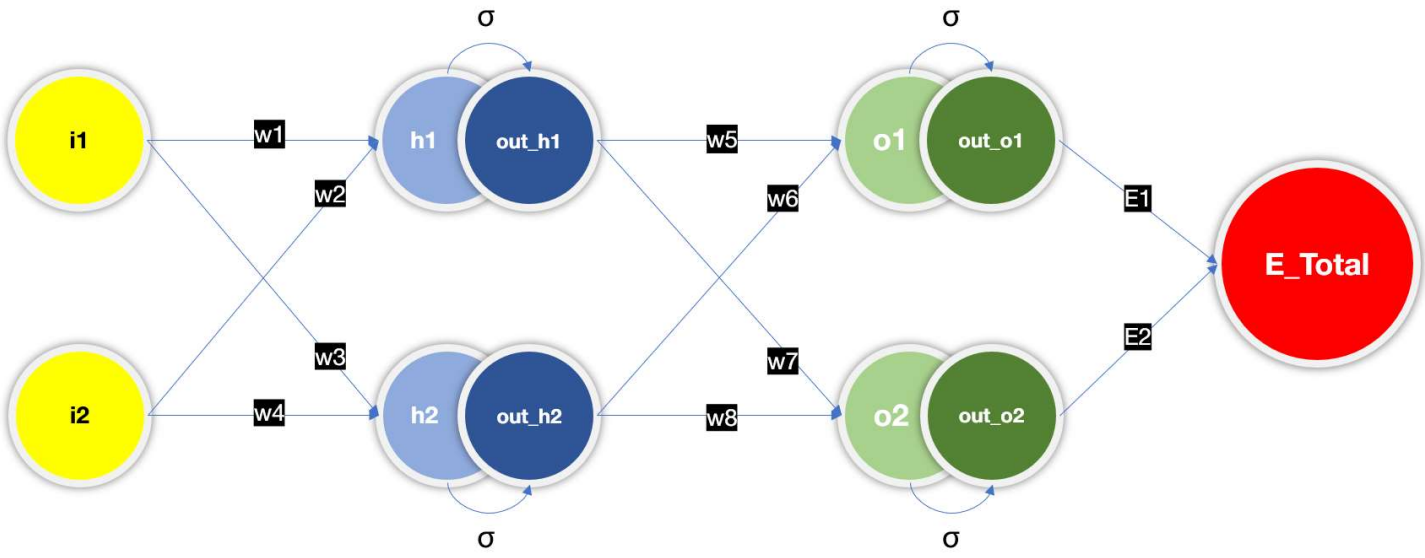
Feed-forward Networks

We discussed in the last session that when we use FC layers, we lose the temporal data. Look at that animation below:



In an FC layer, there is no concept of spatial or temporal information. Neurons are responsible for learning an abstract representation of the data. But what if??

Feedforward networks have the following characteristics

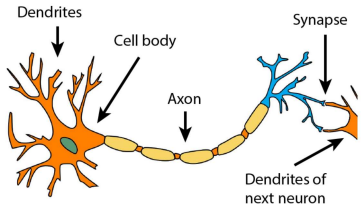


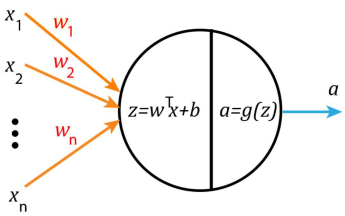
Neurons or perceptrons are arranged in a layer, with the first layer taking in inputs and the last layer producing the output. The middle layers have no connections with the external world and hence are called hidden layers. Each neuron or perceptron in one layer is connected to every perceptron on the next layer. Hence information is constantly "fed forward" from one layer to the next.

There are no connections among the perceptions in the same layer.

Each of the circles that you see is actually a neuron, but **each line** you see is the **weight** that we are training and is of importance to us. Those circles are "temporary" values that will be stored. Once you train the model, *lines are what all matter!*

These lines are denoted by **w**'s where i and j are the neurons it is connecting together.





What are neurons?

A neuron is a fundamental unit of our brain. Neuron, w.r.t. our brain consists of a small "memory storage" or a "signal", as well as a "small computational unit". When we refer to neurons in DNNs/NNs we only consider a small "memory storage" or a "signal" and keep the computation unit outside. This computation unit consists of two elements, a **weight,** and an **activation** function. Each neuron in both cases has input connections. Input connections to a brain neuron are called a dendrite and output connection is called an axon. Both are called just input and output weights in NNs.

Assuming these are **n** connections coming in, the output of the neuron (after using the weights and activation function) can be represented as:

$$a = \tan h\,(z) = \tanh(\sum_{i=1}^{n}(w_i + b_i))$$

where **b** is a bias with which we are stuck for historical reasons. $\tanh$ is playing the role of an activation function here.

In our brain, we have different kinds of neurons doing different things (the brain's activation function) with the information coming in. In the case of NNs, we generally have a single activation like tanh/sigmoid/ReLU, etc.

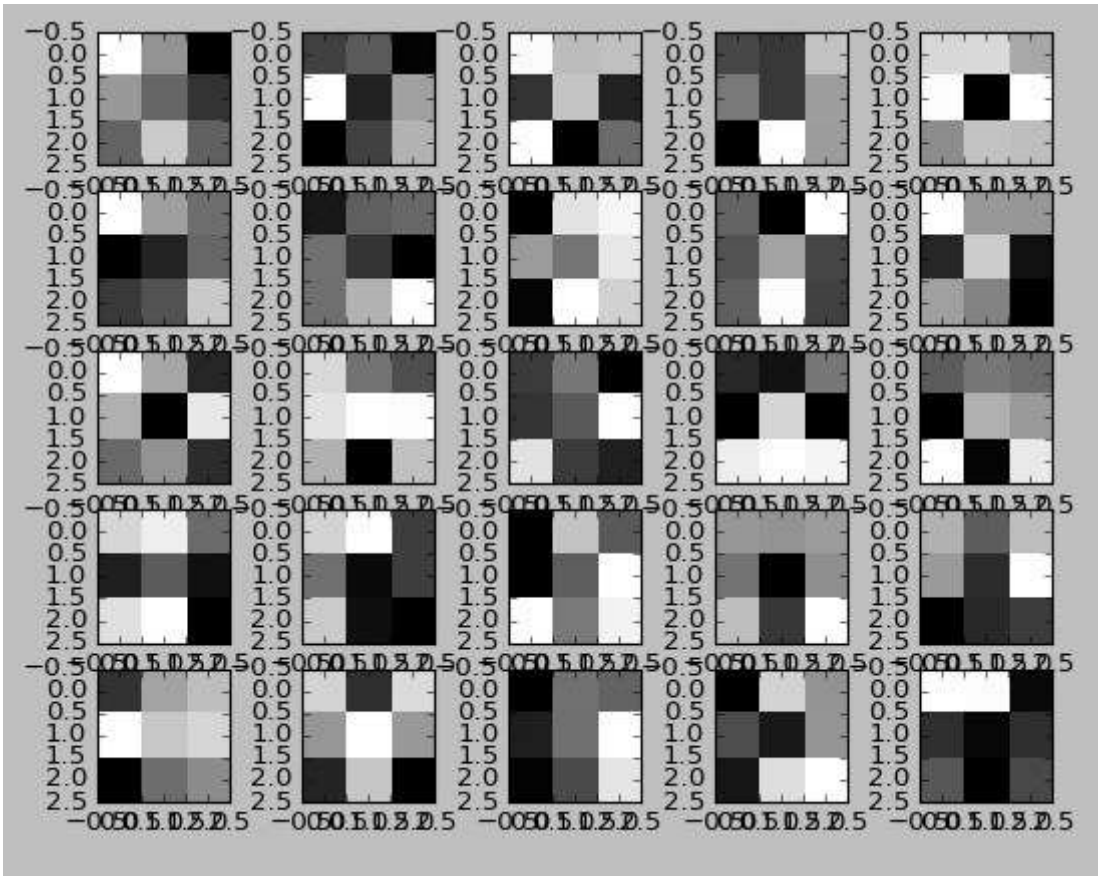
Let's quickly build up an Excel sheet and make a Feedforward NN, that we will be using for performing backpropagation in the next session.

The neuron is a fundamental unit of our brain. Neuron, w.r.t. our brain consists of a small "memory storage" or a "signal", as well as a "small computational unit". When we refer to neurons in DNNs/NNs we only consider a small "memory storage" or a "signal" and keep the computation unit outside.

This computation unit consists of two elements, a **weight,** and an **activation** function. Each neuron in both cases has input connections. Input connections to a brain neuron are called a dendrite and output connection is called an axon. Both are called just input and output weights in NNs.

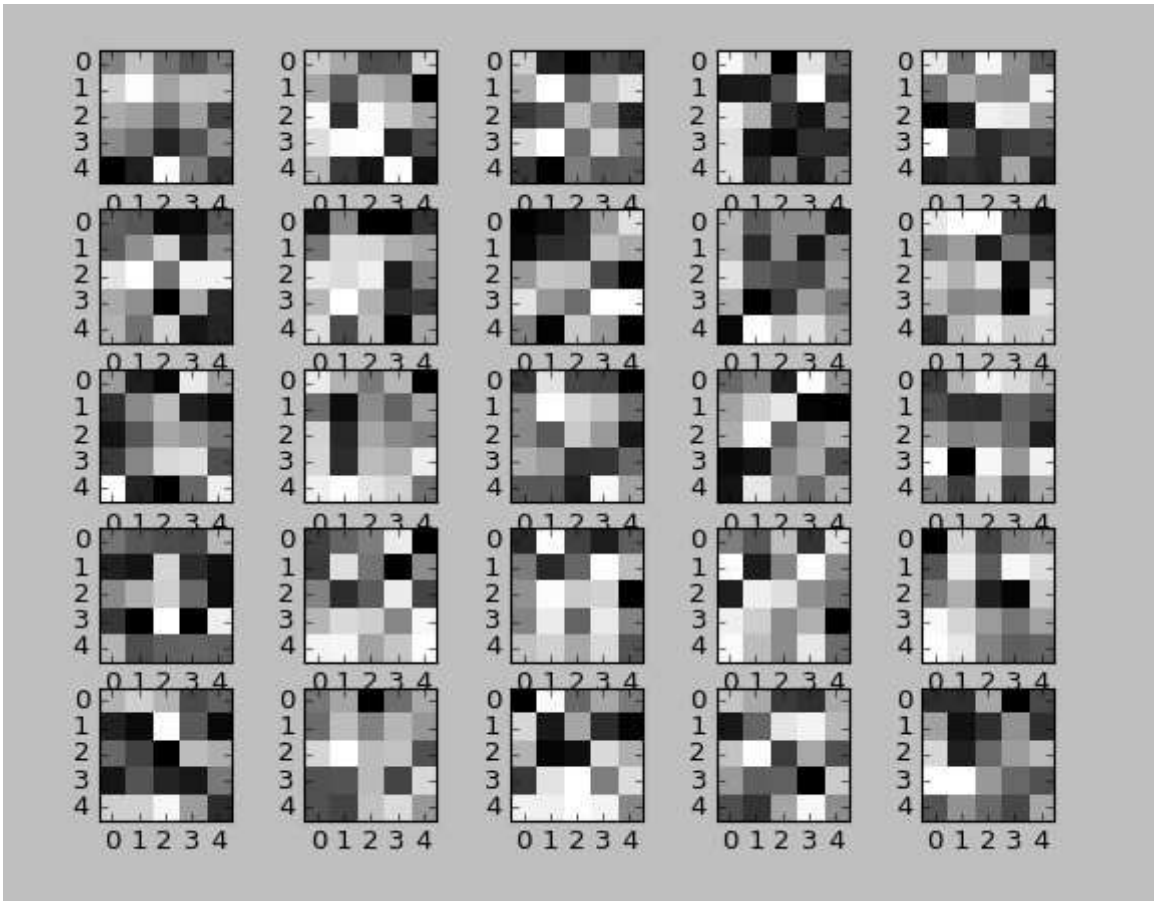
Kernels

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

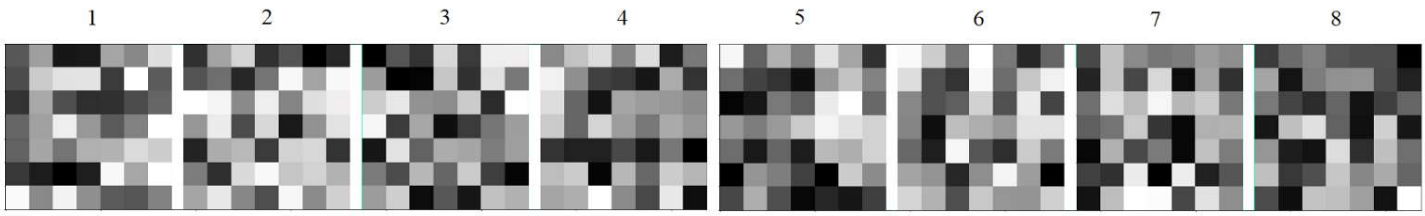


We really cannot find any pattern here. A 3x3 block is a very small area to gather any perceivable information (for human eyes), especially when we are referring to an image of size 400x400.

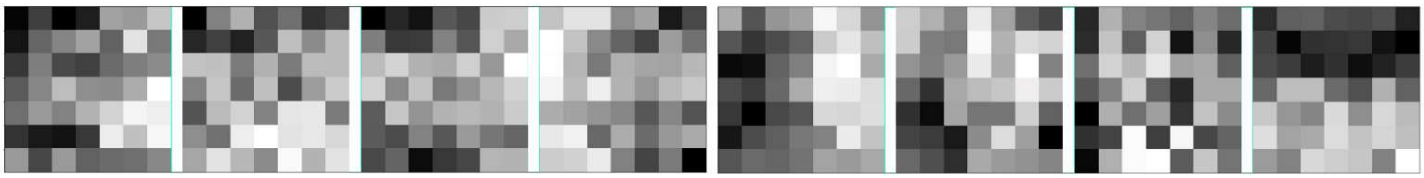
How about 5x5 kernels?



Even at 5x5 (from large image sizes) we cannot make out the patterns being extracted. In fact, at 7x7 we might also fail (look below):



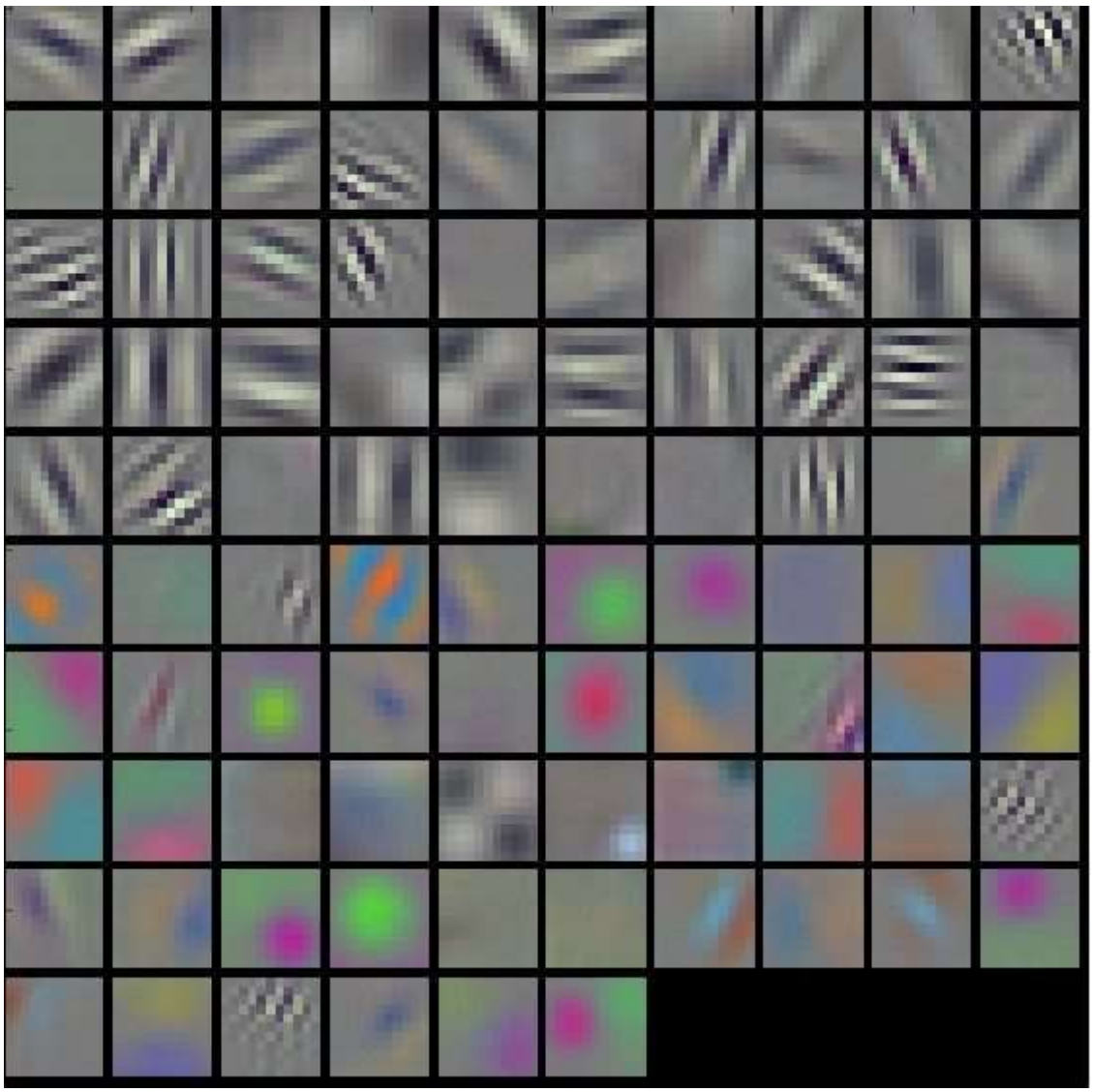
7x7 Kernels before training



7x7 Kernels after training

11x11 - the turning points for +200 images

It is only at the receptive fields of around 11x11 when we would be able to make out patterns being extracted:

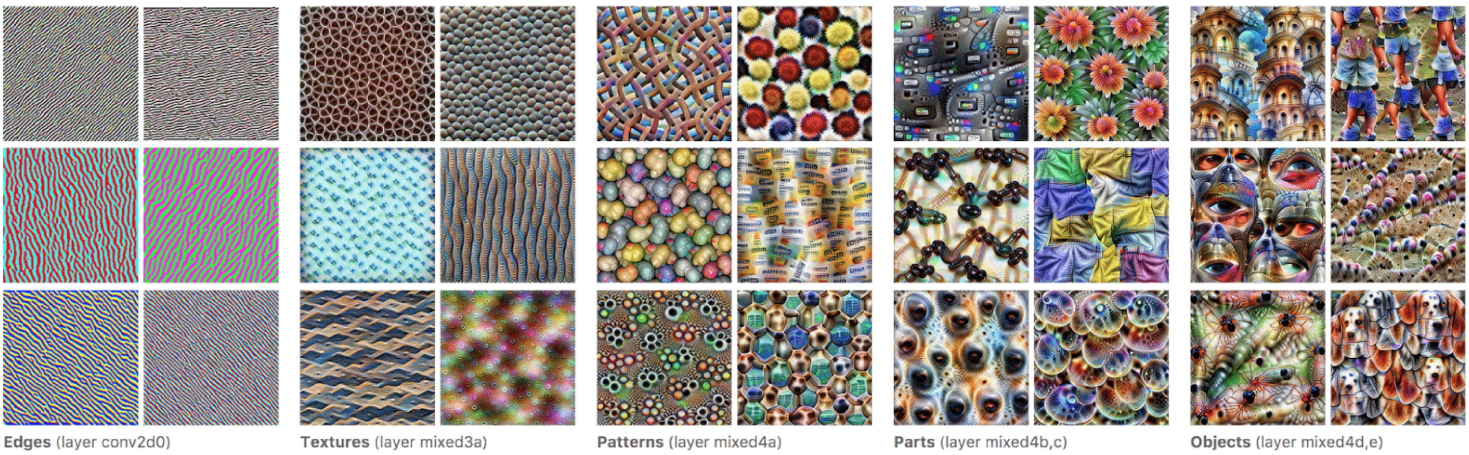


Let's visualize how these things work: **here (https://poloclub.github.io/cnn-explainer/)**, and **here. (https://adamharley.com/nn_vis/)**

4 Blocks - Edges/Gradients, Textures/Patterns, Part of Objects, and Objects.

All this while we have been discussing that we extract edges and gradients, textures, patterns, parts of objects, and then objects. But is it true?

It is indeed true! Look at the visualization below where we can see what our kernels are extracting:



Some of the very important weblinks that we need to take a look at:

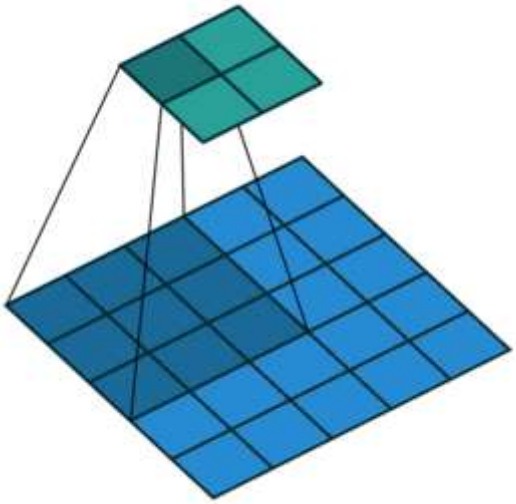
CNN Explorer (https://poloclub.github.io/cnn-explainer/)

Feature Visualization (https://distill.pub/2017/feature-visualization/)

Receptive Fields (https://distill.pub/2019/computing-receptive-fields/)

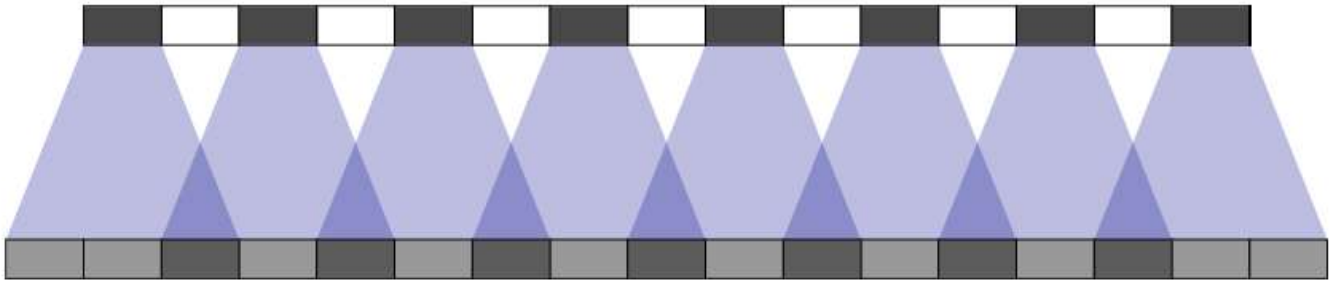
Convolution with Strides

Let us first discuss the consequences of using strides:

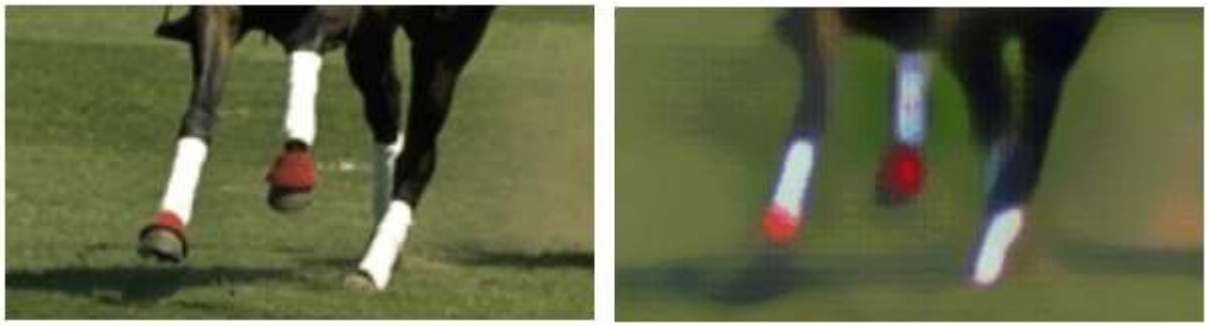


We learned in the last lecture that when we convolve with the standard way (stride of 1), we cover most of the pixels 9 times (we covered only 1 pixel 9 times on a channel of size 5x5, but as the channel size increases, we cover more and more pixels 9 times, e.g. on a channel of size 400x400, 396x396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



Let's check out **distill** (https://distill.pub/2016/deconv-checkerboard/) again.

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

Perceptual Losses for Real-Time Style Transfer and Super-Resolution

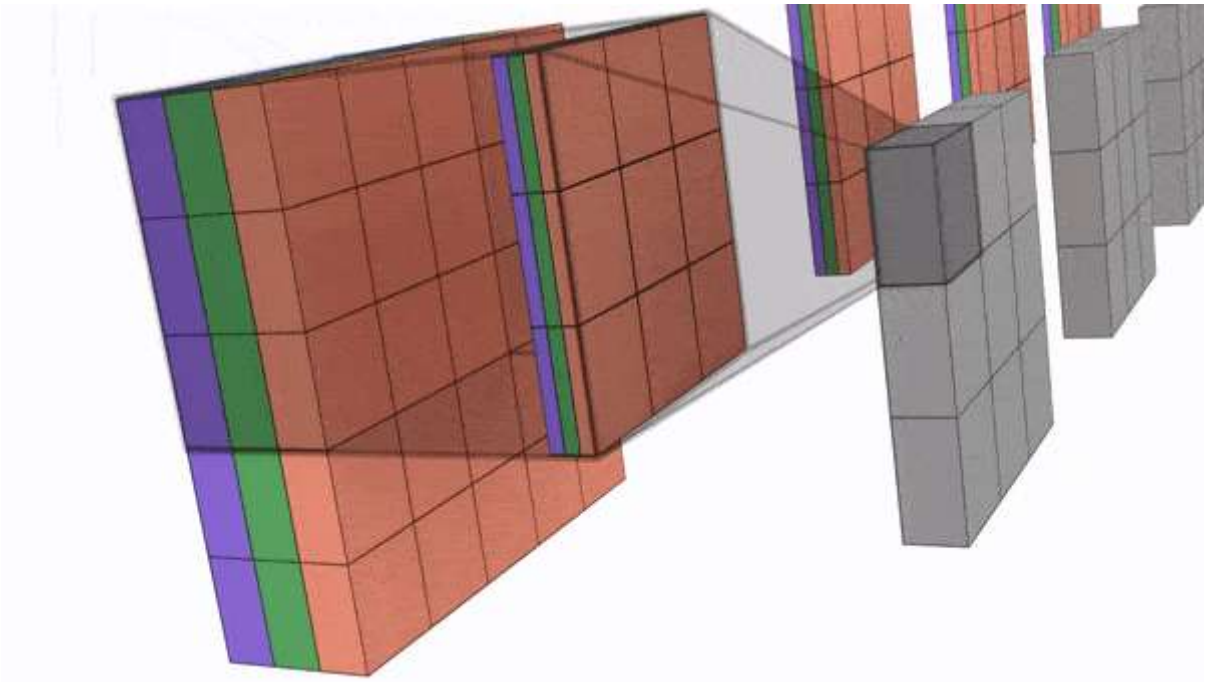
But then why do we use Convolutions with strides?

1. it reduces the amount of computation required by the network
2. it reduces the size of the output, hence reducing the overall number of layers to achieve the target RF
3. it increases the RF drastically
4. it adds additional invariant feature learning capabilities to the network
5. it improves the speed of the inferencing.

Listing two major disadvantages as well:

1. it causes checkerboard issues in the channels/images
2. it reduces the spatial resolution of the output, which means it makes it more difficult for further layers to learn detailed, fine-grained features.

Multi-Channel Convolution



It should be clear that we are using a 3x3x**3**x4 kernel above.

Let's build the first block and see why we are still missing a very strong Avenger from our team, Antman!

The AntMan!

We stopped here in the last session.

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)x512 | 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

.. 3x3x256x512 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 that 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 we aimed to reach the 11x11 receptive field. For some other datasets we might have reached the 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, and 512 kernels, but there are other possible patterns. We are choosing this specific one as this is expressive enough to build the 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" from textures.
6. The 5 Convolution Layers we see above form the "**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 that could extract multiple features that come together?
2. 3x3 is an expensive kernel, and we should be able to figure out something lighter, and less computationally expensive method.
3. Since we are merging 512 kernels into 32 complex ones, it would be great if we do not pick those features that are not required by the network to predict our images (like backgrounds).

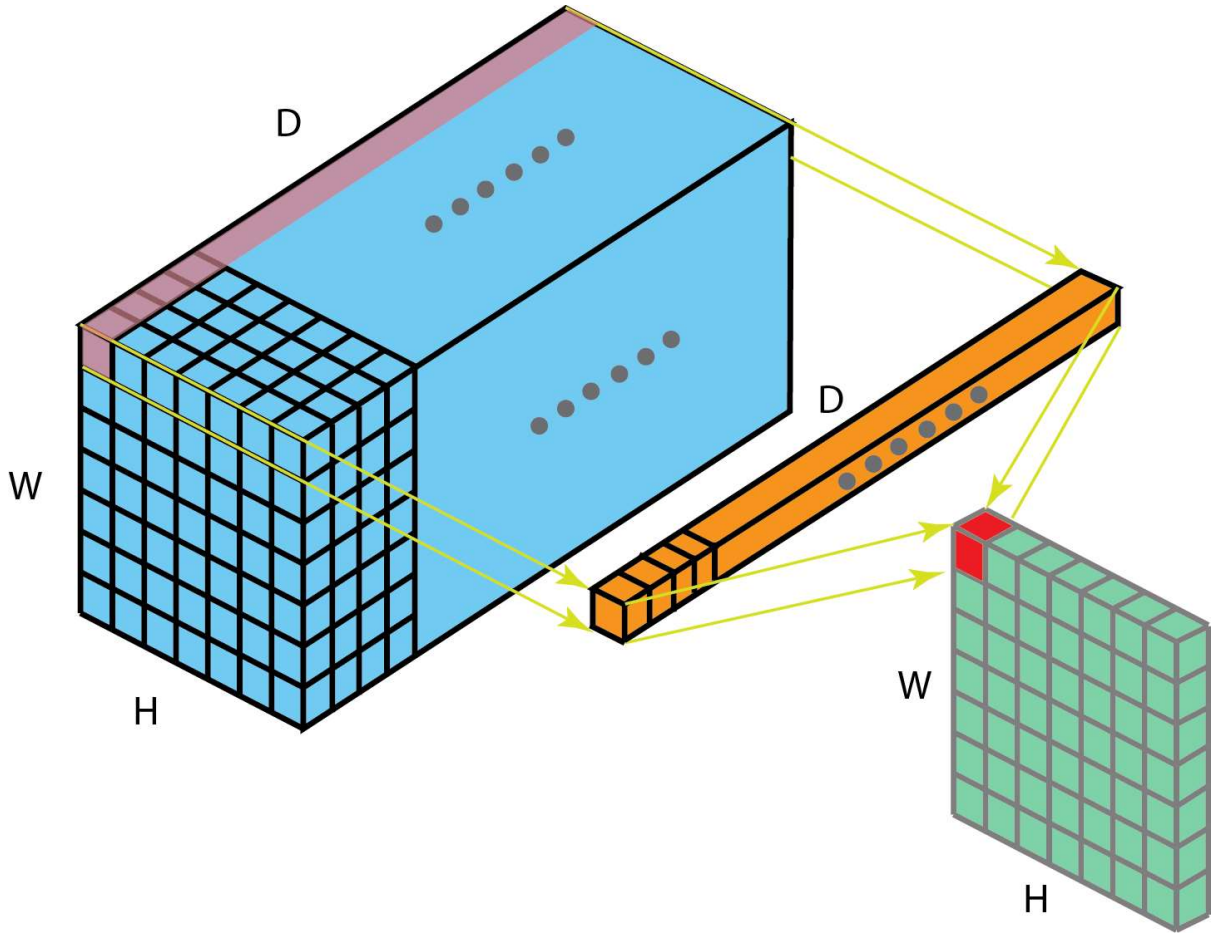
1x1 provides all these features.

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 that make up an eye)
4. 1x1 is performing a weighted sum of the channels, so it may decide 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. Imagine 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.

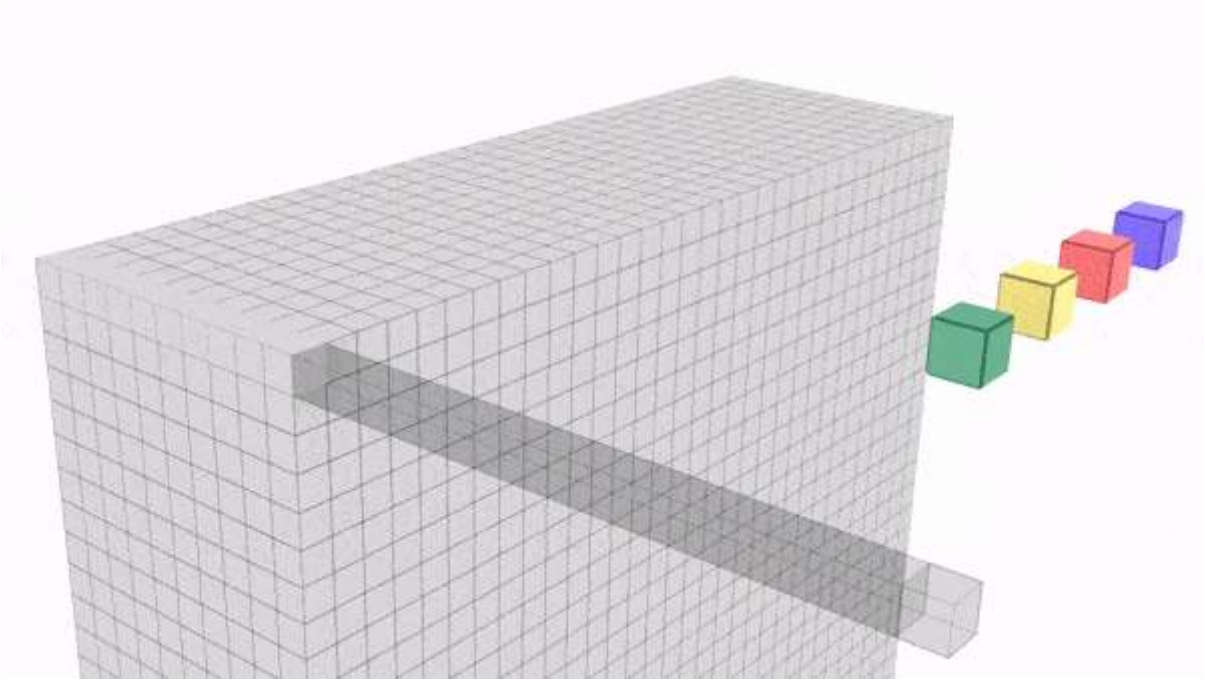
1x1 Convolutions

This is what 1x1 convolutions look 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.

Animated 1x1:



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.

32x32**x10** | 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

...

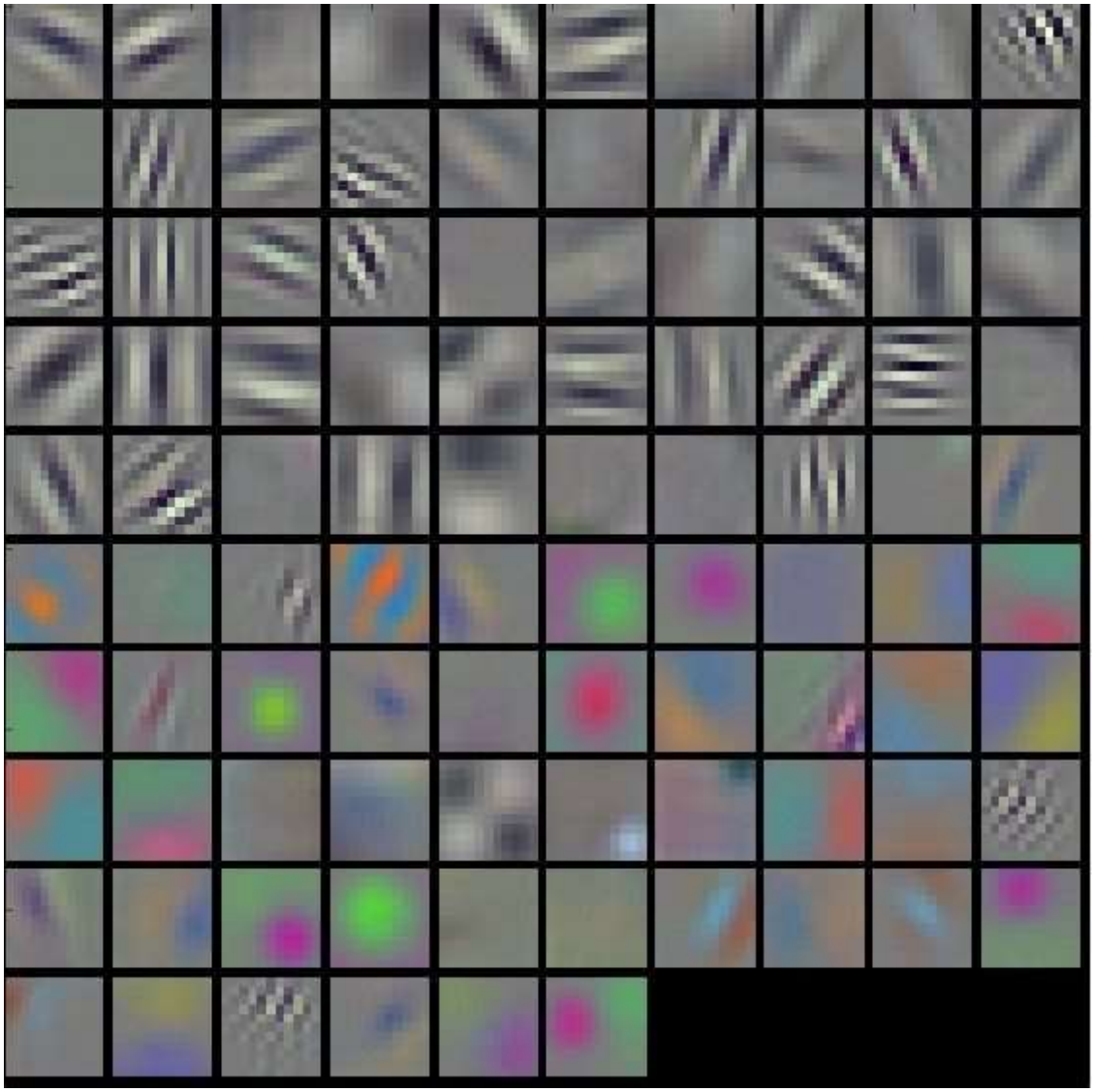
Notice, that we have kept the first convolution outside of our convolution block, as now we can create a functional block receiving 32 channels and then perform 4 convolutions, giving finally 512 channels, which can then be fed to the transition block (hoping to receive 512 channels) which finally reduces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

Let's practice all of these concepts right now!
(https://colab.research.google.com/drive/1oVt7T6tb90Y1EXvFalWgm72emqZZPXi4?usp=sharing)

Embeddings

What are embeddings?

Let's look at this image:



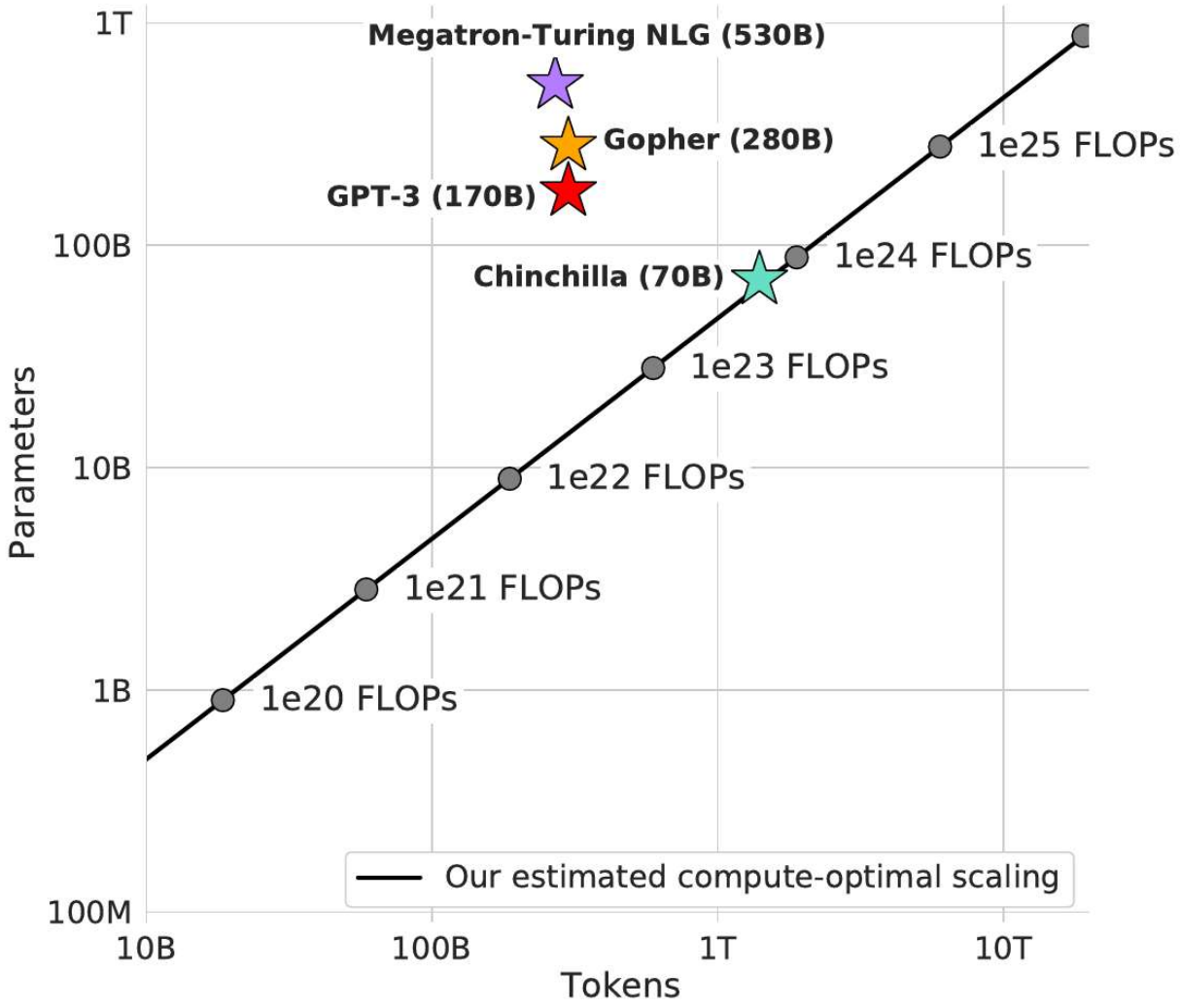
We know what this is, and we know how we get these visualizations, by training the model.

What if we could store them and reuse them later?

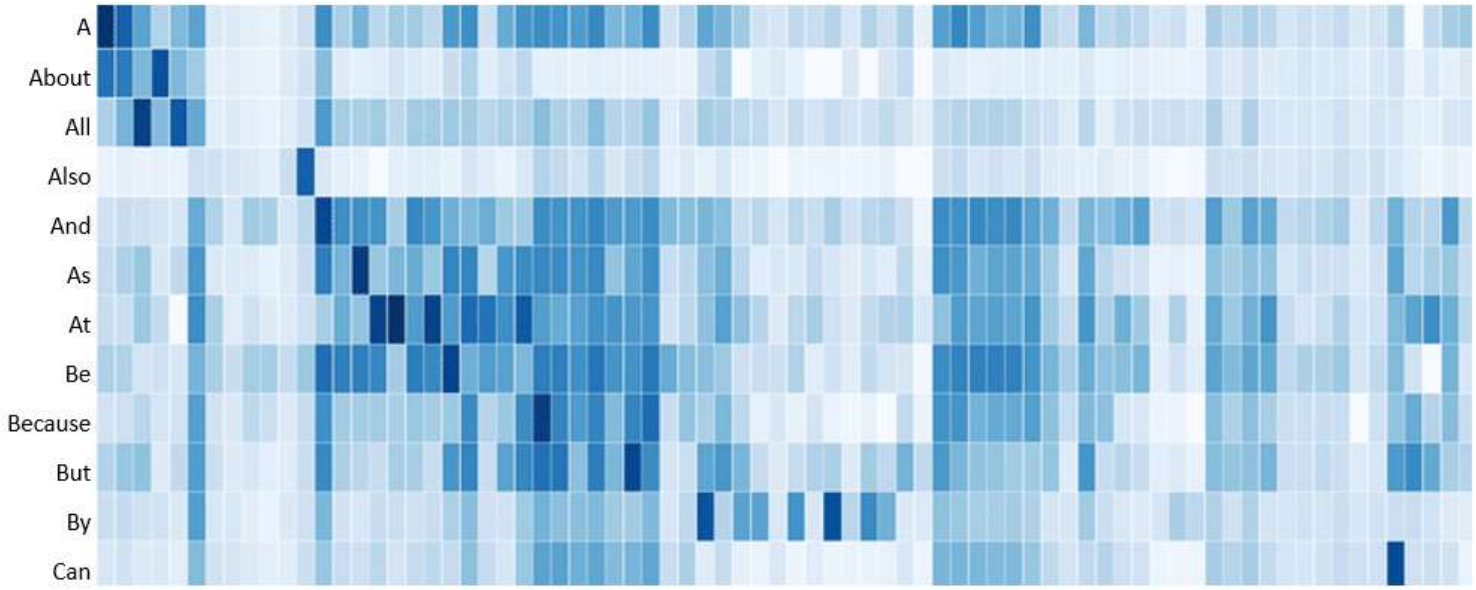
For images we don't do it at this level (block 1), but certainly at block 3/4 (transfer learning). But for images, the process isn't still standardized. For NLP, however, it is! Let's look at the same image again but with words overlapped to understand contextually what we mean.

Here I am asking you to imagine that we **can** come up with the exact representation value for each word. But something like this is tricky for the images (as there are *types* of images like natural images, astronomical, microscopic, X-ray/medical, etc). (I would also argue, that theoretically it is possible to do this for images, but for some reason, this hasn't been done)

But words, well, we can get a big enough dictionary (called tokens) where we can represent every word in the world.



Also since we represent NLP data in 1D, it would look like this:

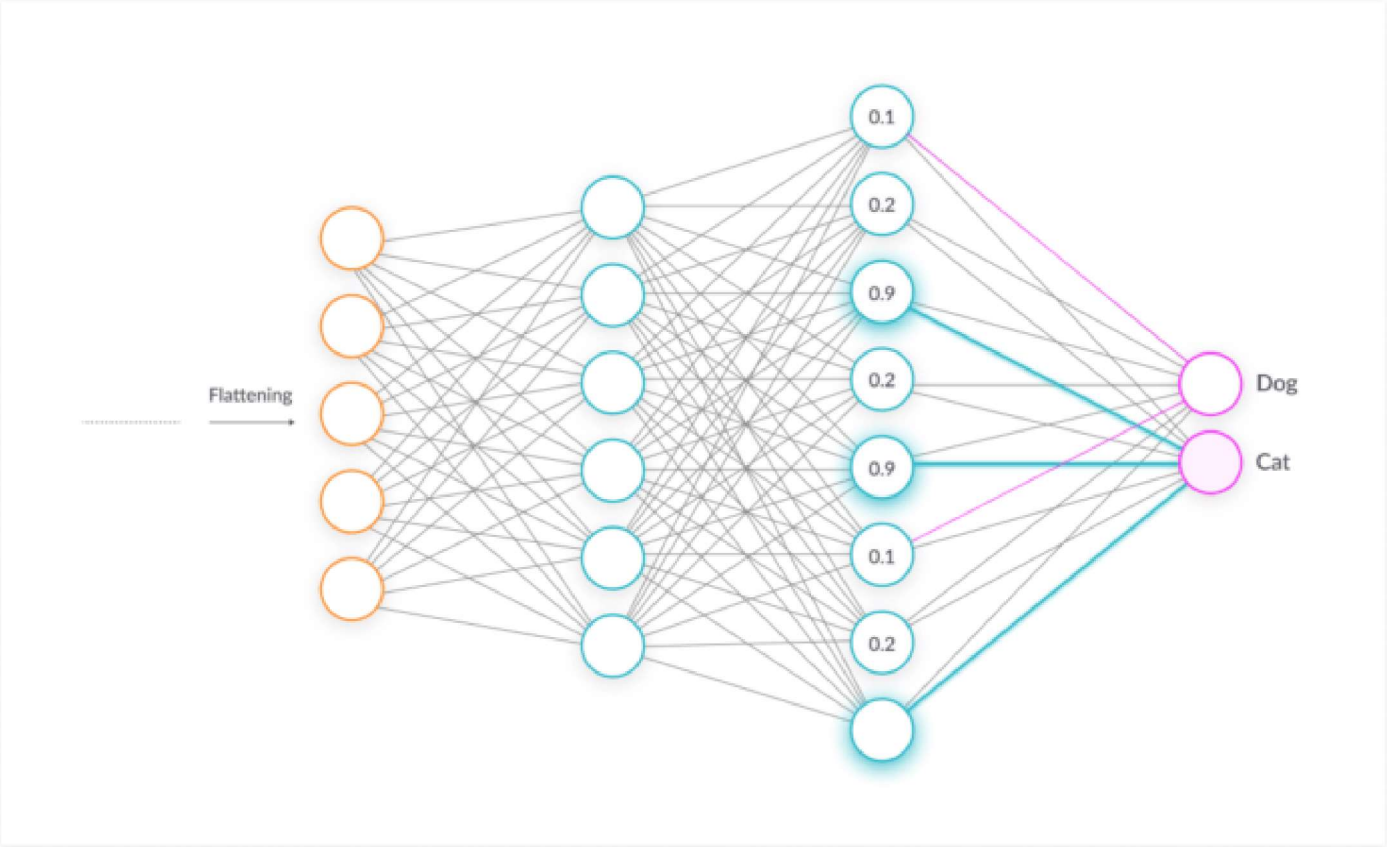


Positional Embeddings

What are positional embeddings?

We learned earlier that when we convert 2D data into 1D, we lose position data.

Well when we use 1D data with FC Layer, we lose position data there as well!

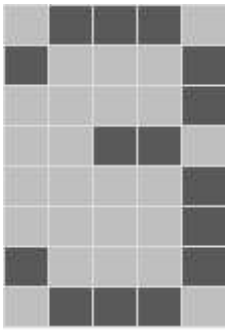


Remember: "These circles are "temporary" values that will be stored. Once you train the model, *lines are what all matters*"

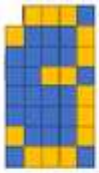
But since it is easier to understand with images, let's stick to images. What digit do you think this pattern represents?

Exactly, that's the point!

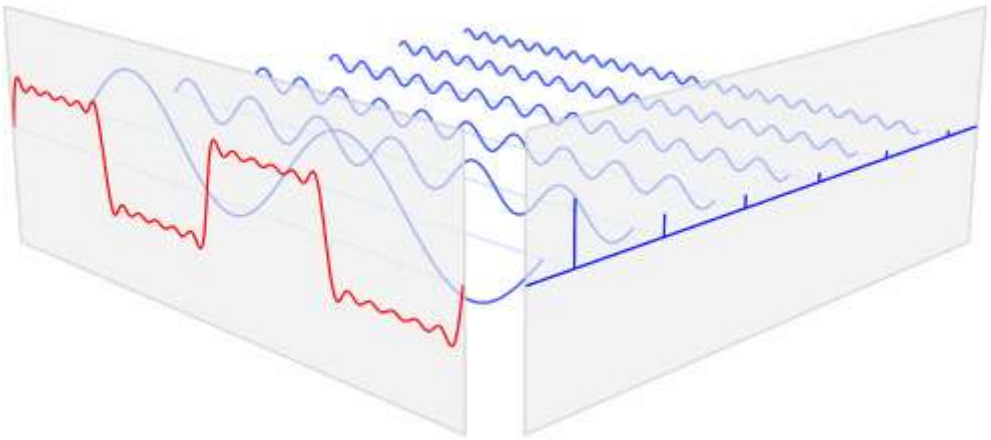
This is actually a:



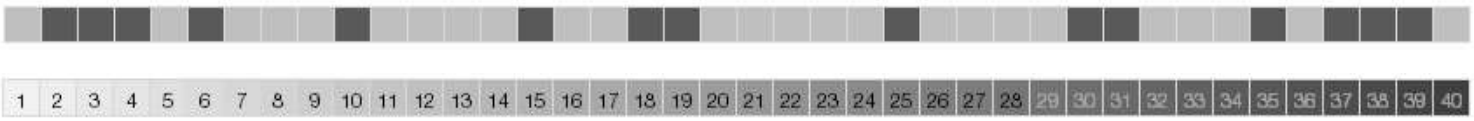
The process of conversion looks something like this, as you already know:



And as you can see we lost positional information here. How do we add the positional information back?



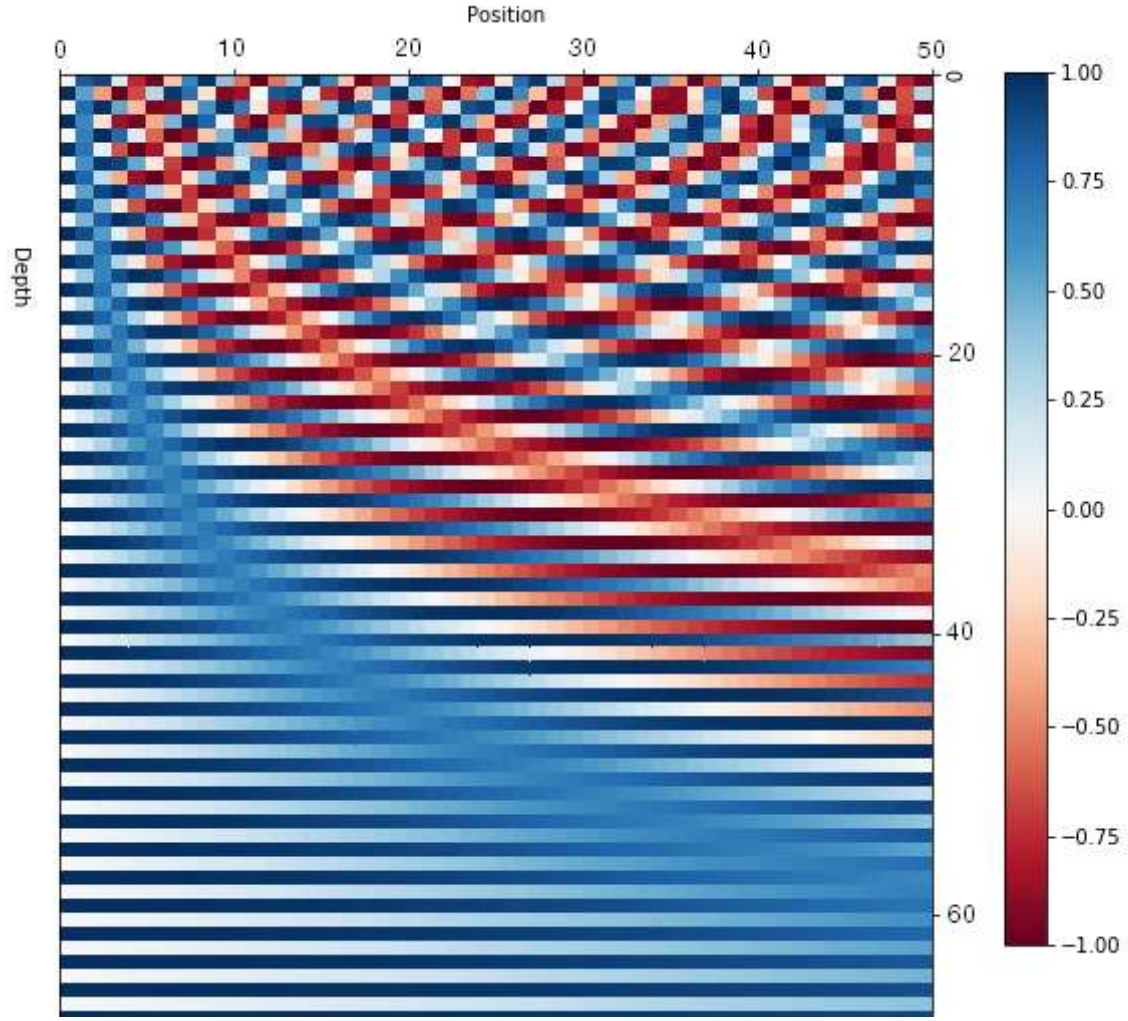
DNNs are surprisingly amazing at extracting the added signals to original data, just like Fourier transformers. This means, that we can add data to our original image/source and DNNs will be able to extract that information back. The convolution integral is, in fact, directly related to the Fourier transform and relies on a mathematical property of it.



buy **ADDING** the positional-related data back to each element.

But you can already see, adding 40 to one element and 1 to another element would screw up their scales (and neither this is something that FT can extract). (This is a topic for the future).

Like a word can have an embedding (say each word is represented by a 512D vector), our positions can also have embeddings (512D vector instead of just 1 number). In future sessions, it would look something like this:

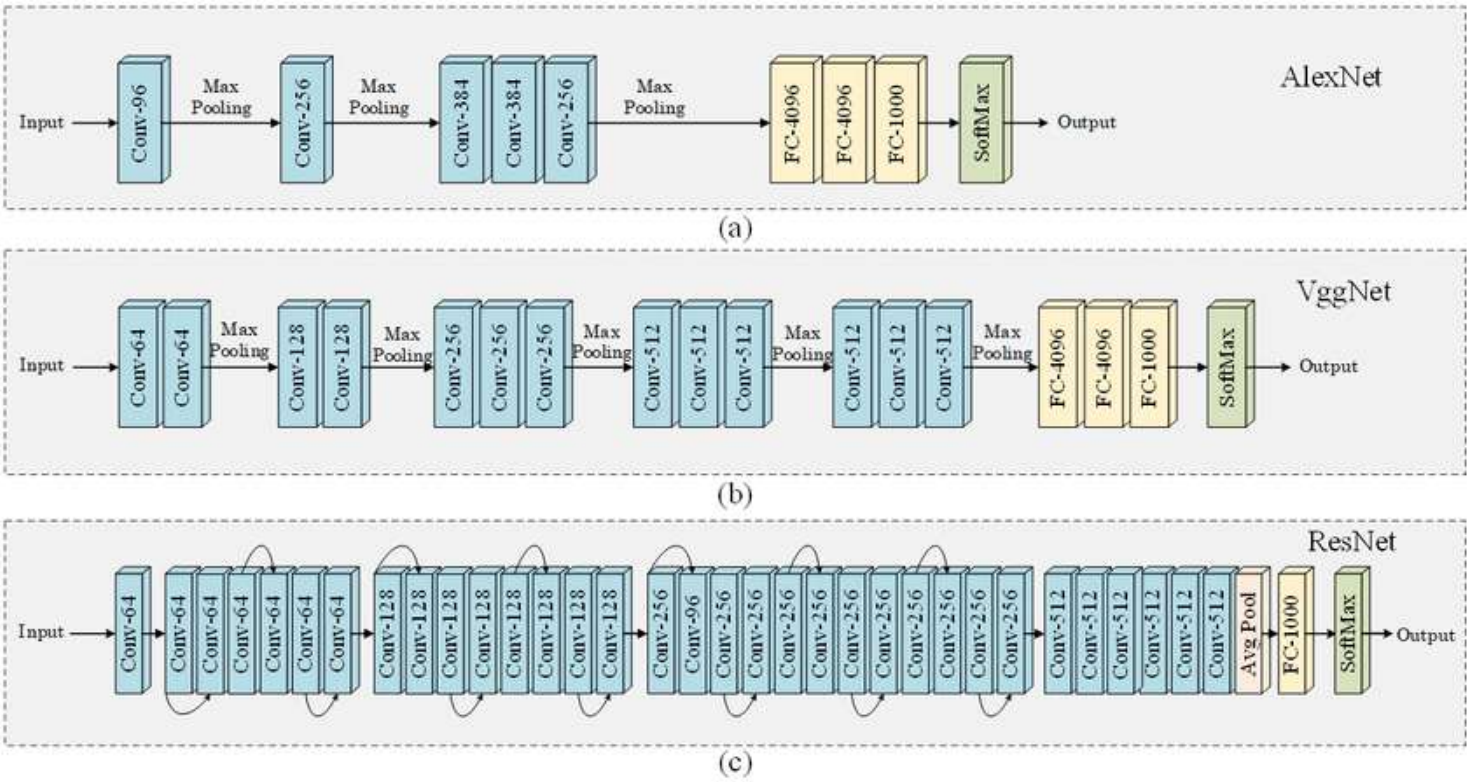


The Modern Architecture

Squeeze and Expand vs Pyramidical

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

Most modern architecture, however, follows Pyramidical architecture:

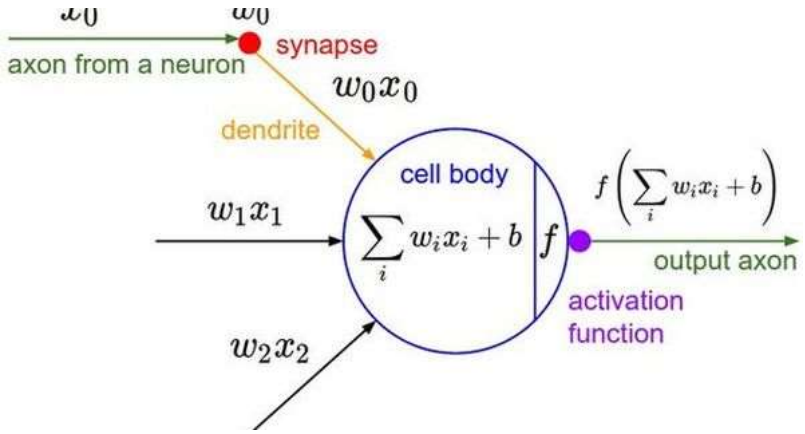


Major Points:

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 to 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. Every network starts from 56x56 resolution!

Activation Functions and which one to use.

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



For ages, this activation function has kept the AIML community occupied in the wrong direction.

After the convolution is done, we need to take a call about 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?

Here is what ChatGPT thinks:

1. Activation functions allow neurons to respond in a non-linear way to the input data, which is important for modeling many real-world problems.
2. Activation functions introduce non-linearity into the network, which improves its ability to learn complex patterns and relationships in the data.
3. Without activation functions, a neural network would be limited to learning only linear relationships, which would make it much less effective at solving many real-world problems.
4. Activation functions are an essential component of neural networks because they enable the network to learn complex patterns and relationships in the data, and they allow the network to be more powerful and versatile.
5. Activation functions are necessary for the proper functioning of neural networks, and they play a crucial role in determining the performance of the network on real-world tasks.

Let ***us*** understand why we need an activation function.

Let's start by asking, what is that **σ** doing there?

A linear activation function (or none) has two major problems:

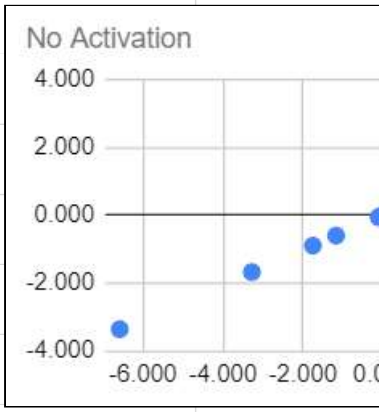
It is not possible to use backpropagation (gradient descent) to train the model—the derivative of the function is a constant and has no relation to the input, X. So it’s not possible to go back and understand which weights in the input neurons can provide a better prediction

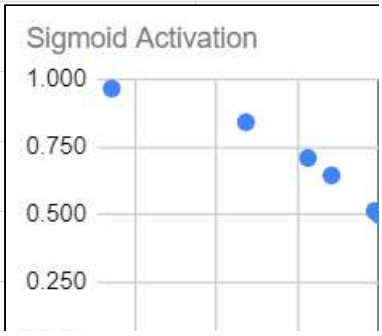
. All layers of the neural network collapse into one—with linear activation functions, no matter how many layers are in the neural network, the last layer will be a linear function of the first layer (because a linear combination of linear functions is still a linear function). So a linear activation function turns the neural network into just one layer

That is why we need activation functions (other than just linear ones)

Activations

Input	Weight	Tem	Ou	No Act	ReLU	Sigmoic	Tanh
0.000	0.511	0.000		0.000	0.000	0.500	0.000
-3.277		-1.674		-1.674	0.000	0.842	-0.932
-6.579		-3.361		-3.361	0.000	0.966	-0.998
-1.173		-0.599		-0.599	0.000	0.646	-0.537
-0.111		-0.057		-0.057	0.000	0.514	-0.057
2.814		1.438		1.438	1.438	0.192	0.893
0.000		0.000		0.000	0.000	0.500	0.000
2.683		1.371		1.371	1.371	0.203	0.879
0.833		0.425		0.425	0.425	0.395	0.401
5.645		2.884		2.884	2.884	0.053	0.994
1.752		0.895		0.895	0.000	0.710	0.714





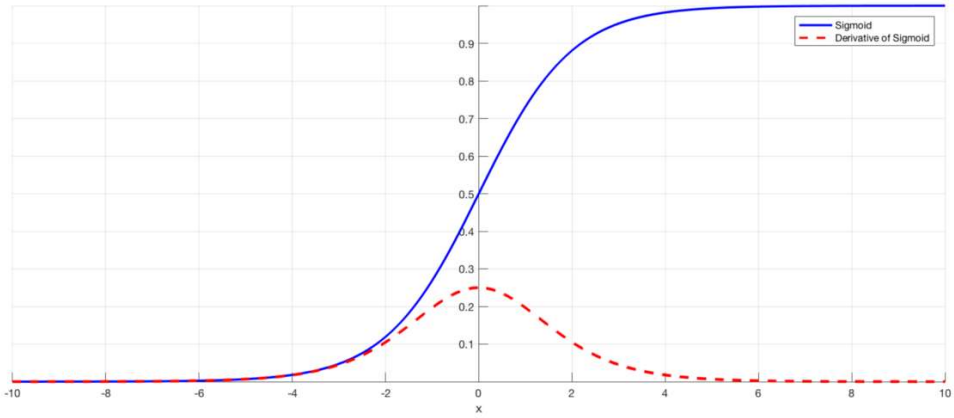
Sheet1 Forward Backward Next Forward

LINK (https://docs.google.com/spreadsheets/d/1TNFhfQBTdiHWR7x9zJ1UDzLcVnkOJV9JhJdp6Nas6h8/edit?usp=sharing)

Activation Function Tyoes

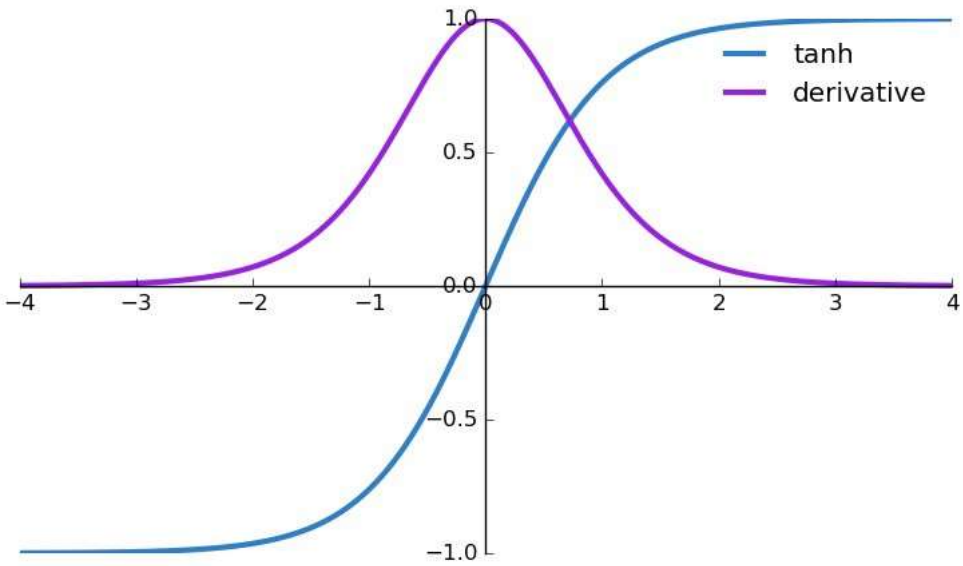
Sigmoid

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



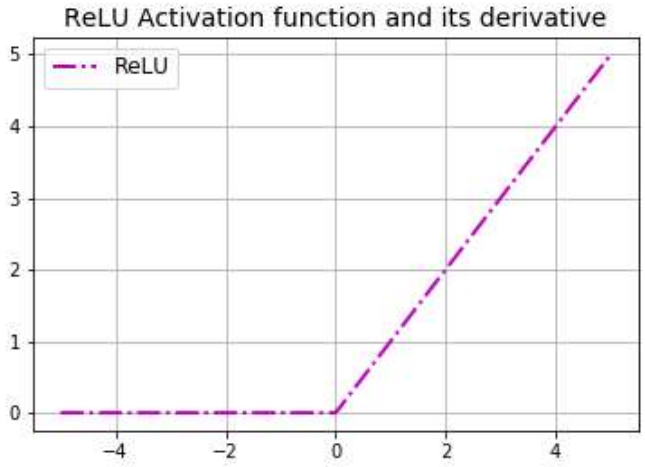
tanh

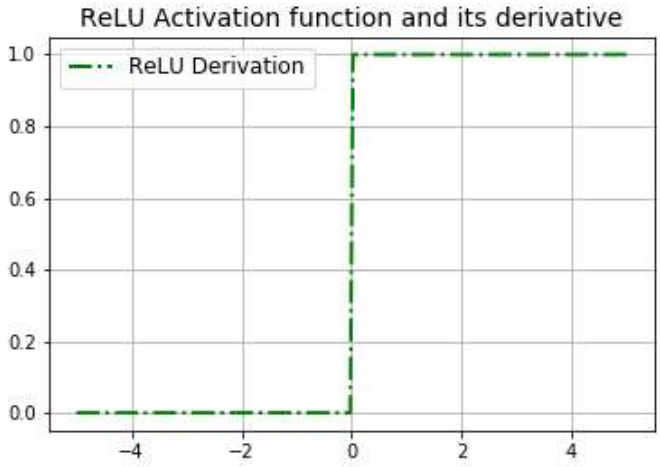
When sigmoids didn't work, we felt maybe another complicated function, called **TanH** might work:



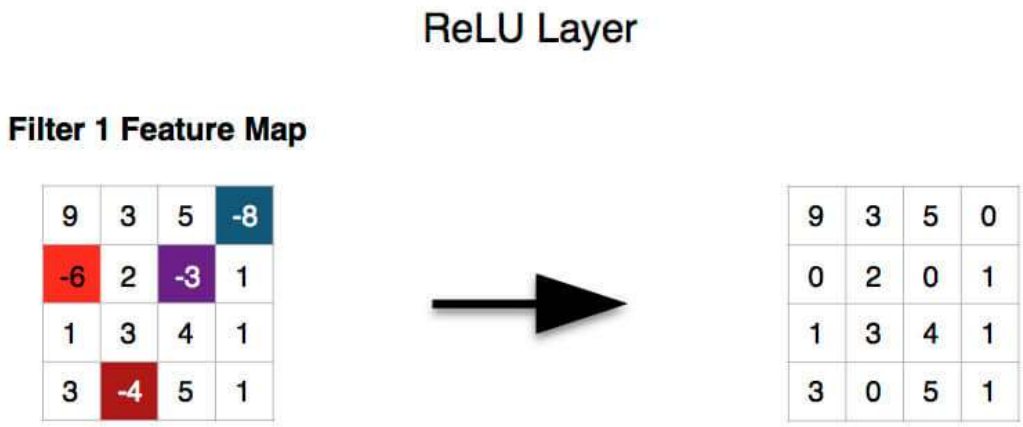
ReLU

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 simple message to backpropagation or kernels: "*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 Layer filters out negative numbers:



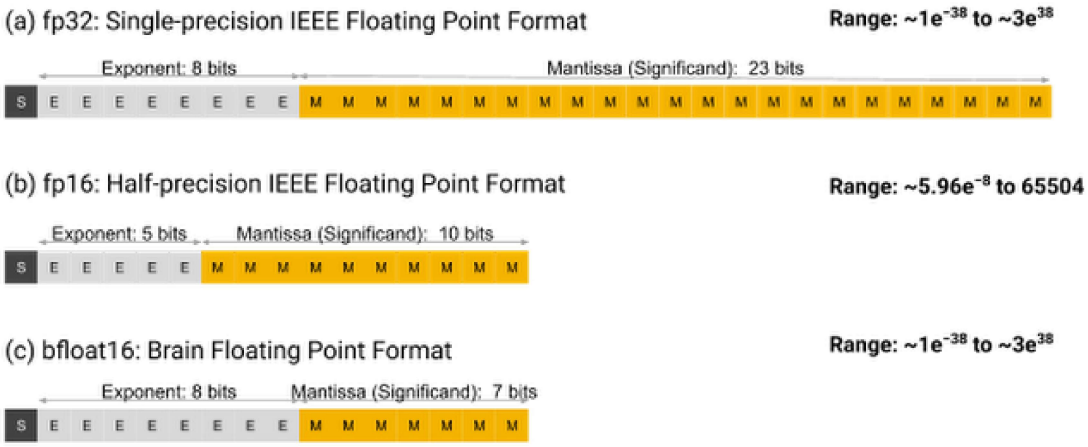
ReLU since has worked wonders for us. It is fast, simple, and efficient.

ReLu- Rectified Linear Units: It has become very popular in the past 7-8 years. It was recently (2018) proved that it had *6 times improvement in convergence* from the Tanh function. Mathematically it is just:

svg.svg

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.

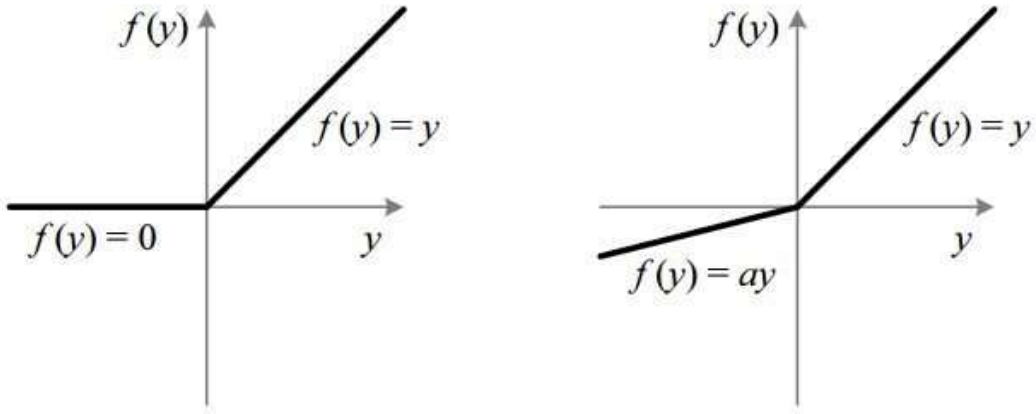
But aren't humans known to complicate things?

One might wonder, why such partiality to all the negative numbers. What if we allow a negative number to *leak* a few of their values?

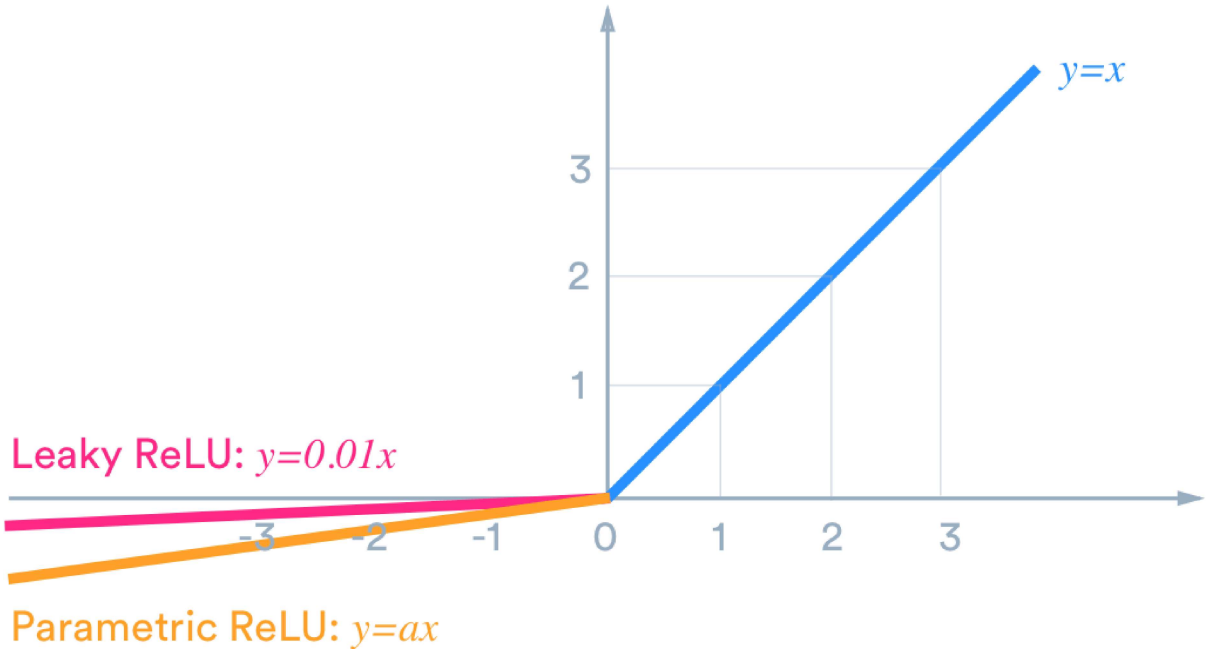
Thence came

LeakyReLU

ReLU vs LeakyReLU

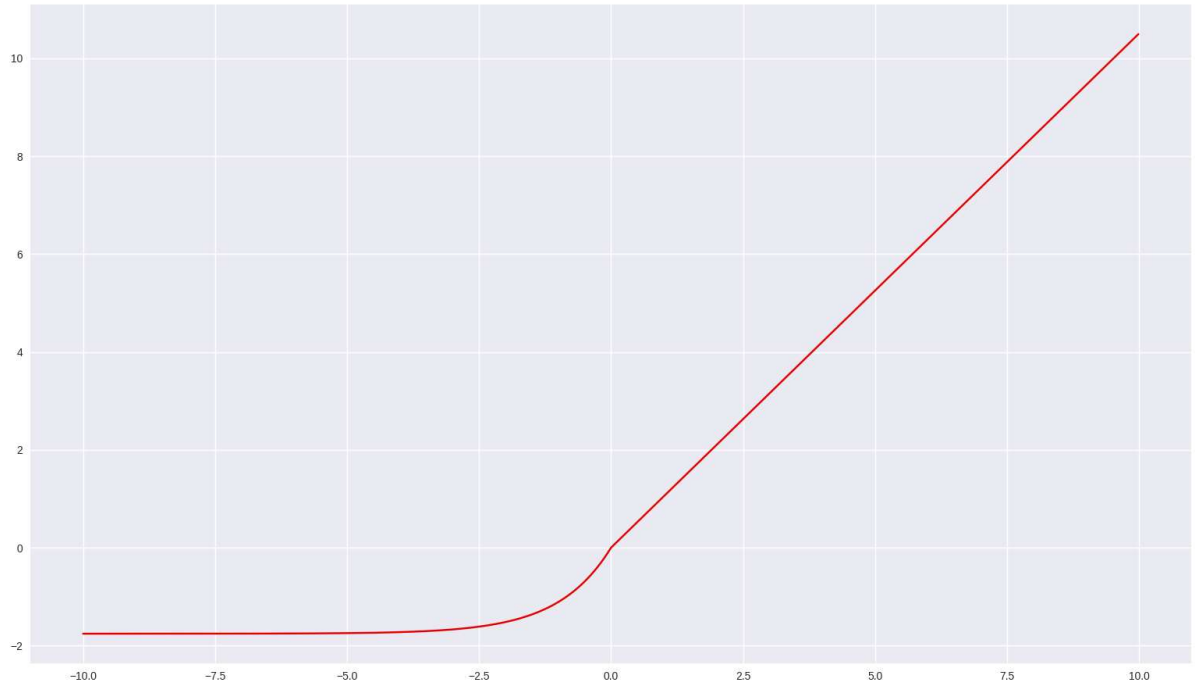


And then why stop there? You can complicate this further. Now instead of being a Constance, we can get DNN to figure out what α should be.

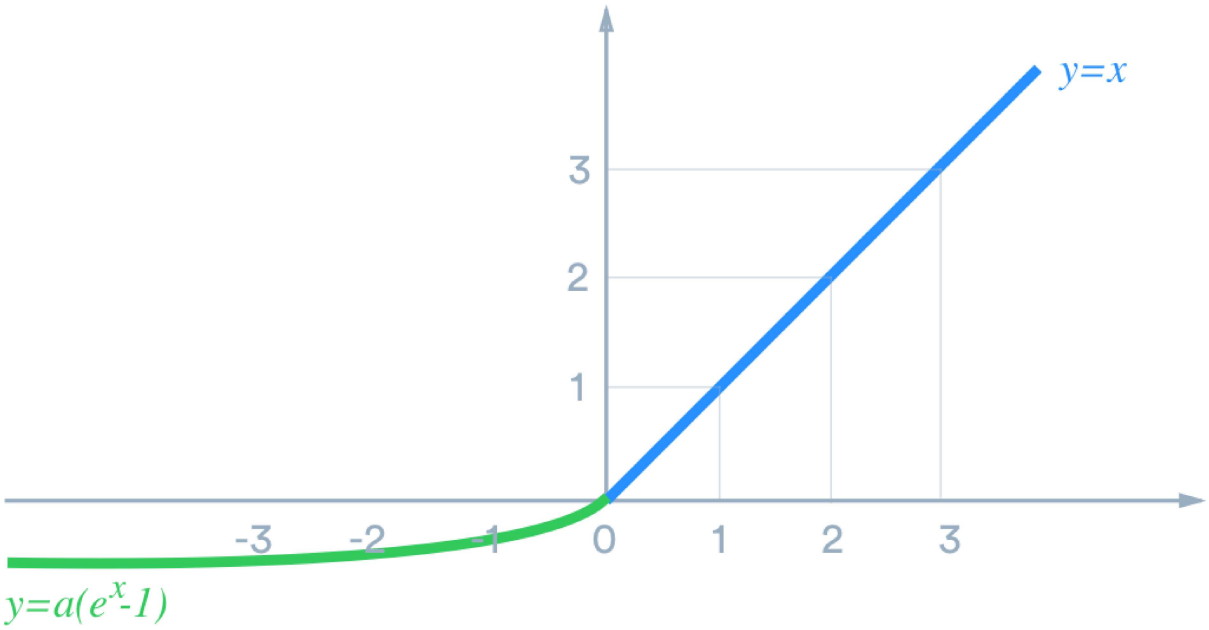


And then came a sequence of papers, each just adding a character before ELU and creating new activation functions.

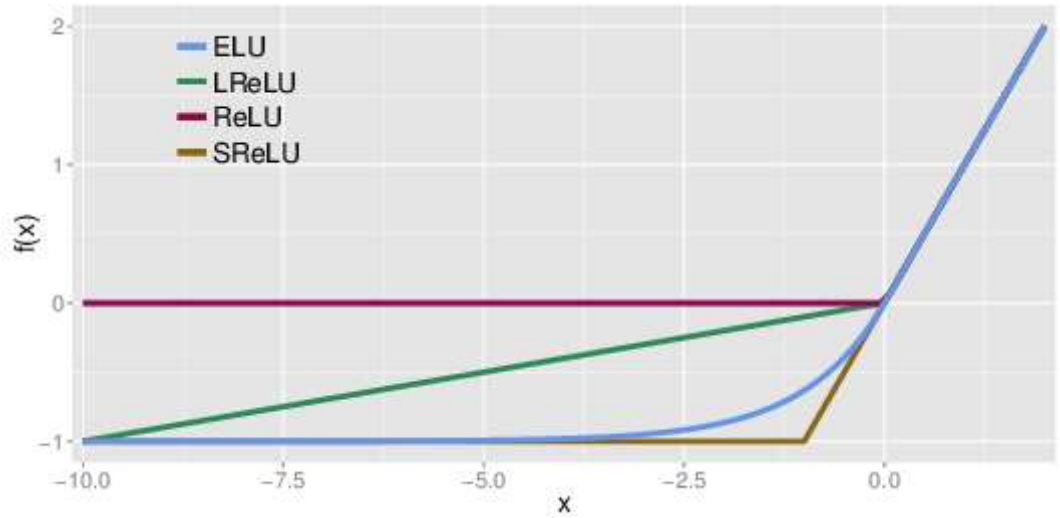
SELU



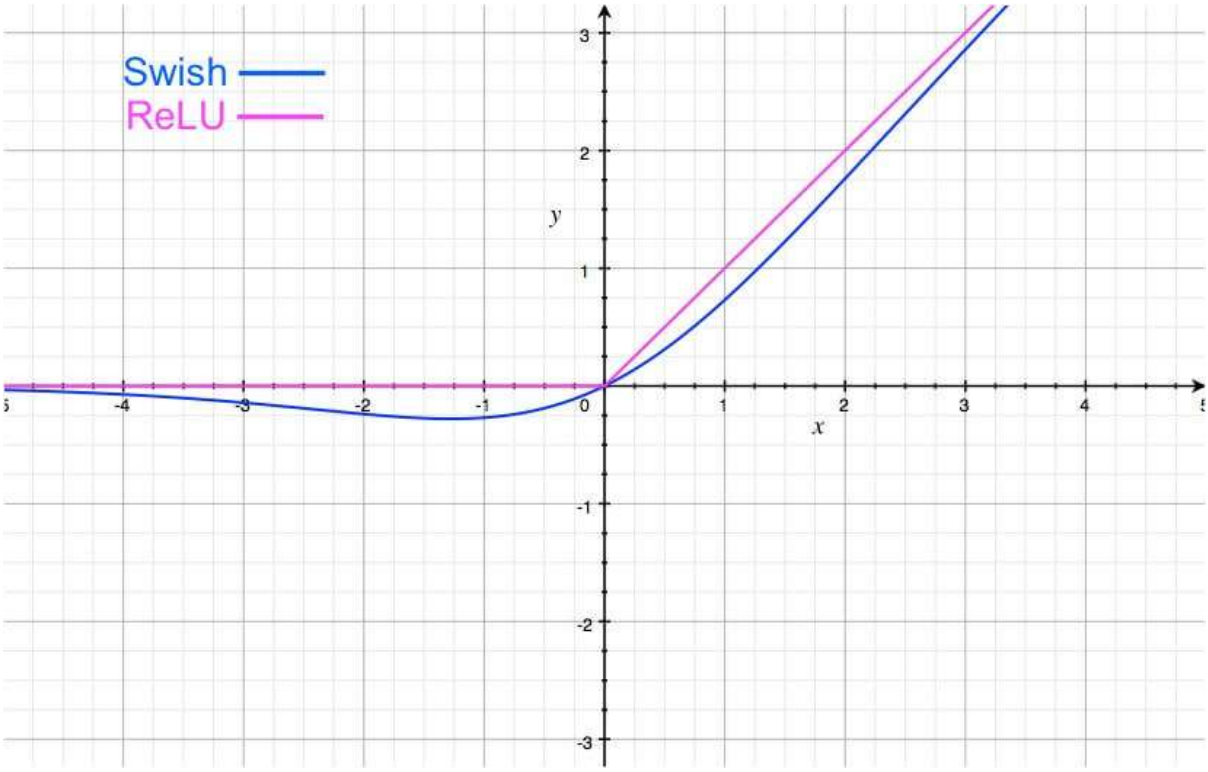
ELU


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

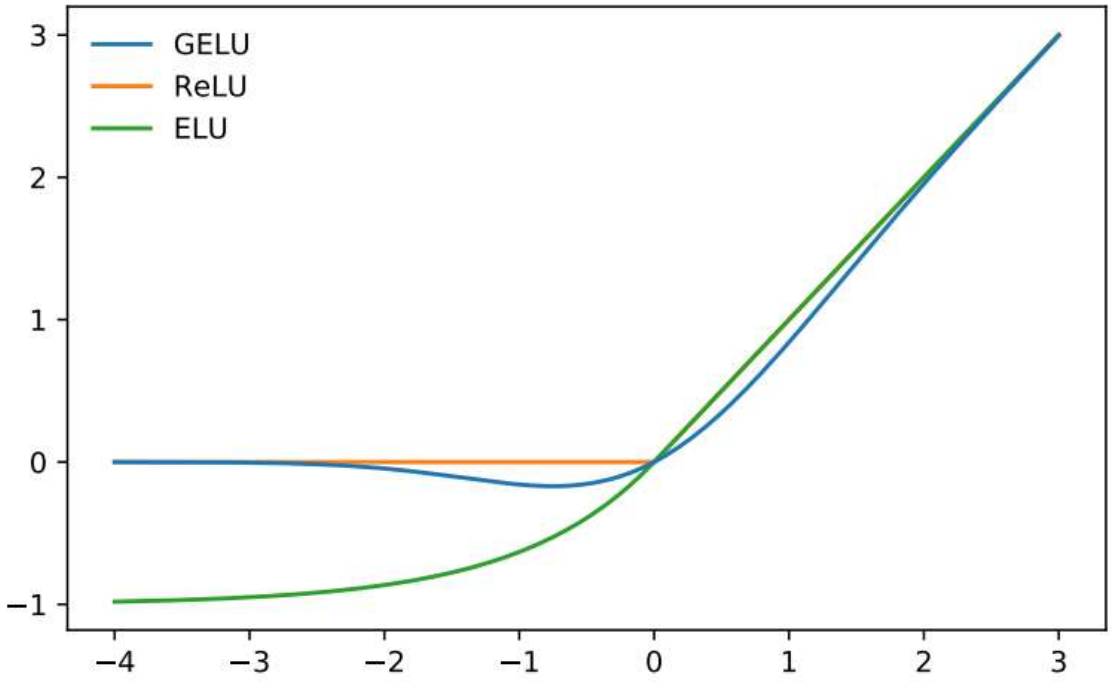
SReLU


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

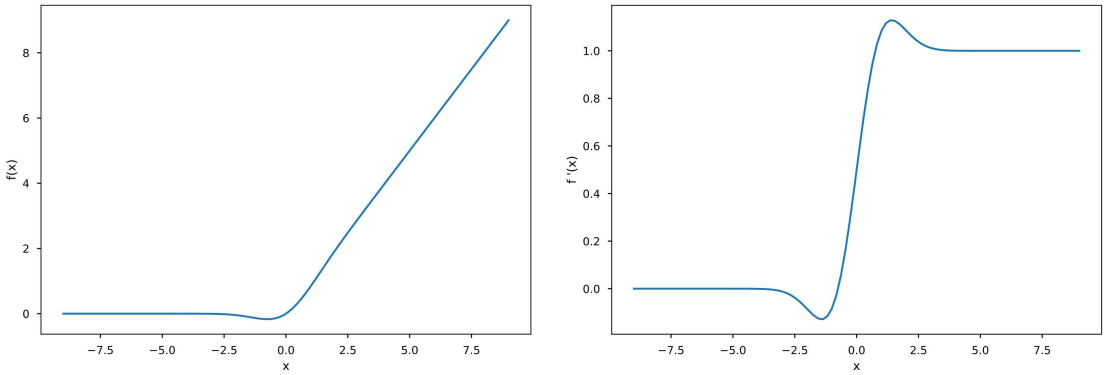
Swish



GELU



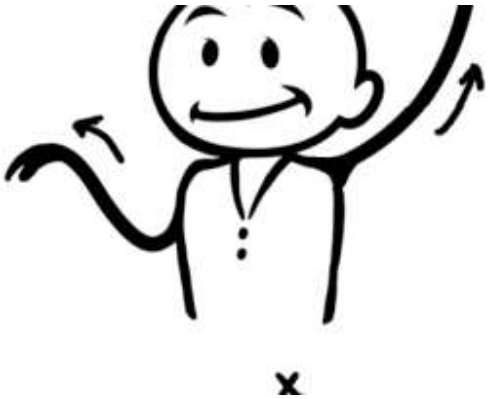
GELU function and it's Derivative



$$GELU$$

$$f(x) = xP(X \leq x) = x\Phi(x)$$
$$= 0.5x\left(1 + tanh\left[\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right]\right)$$

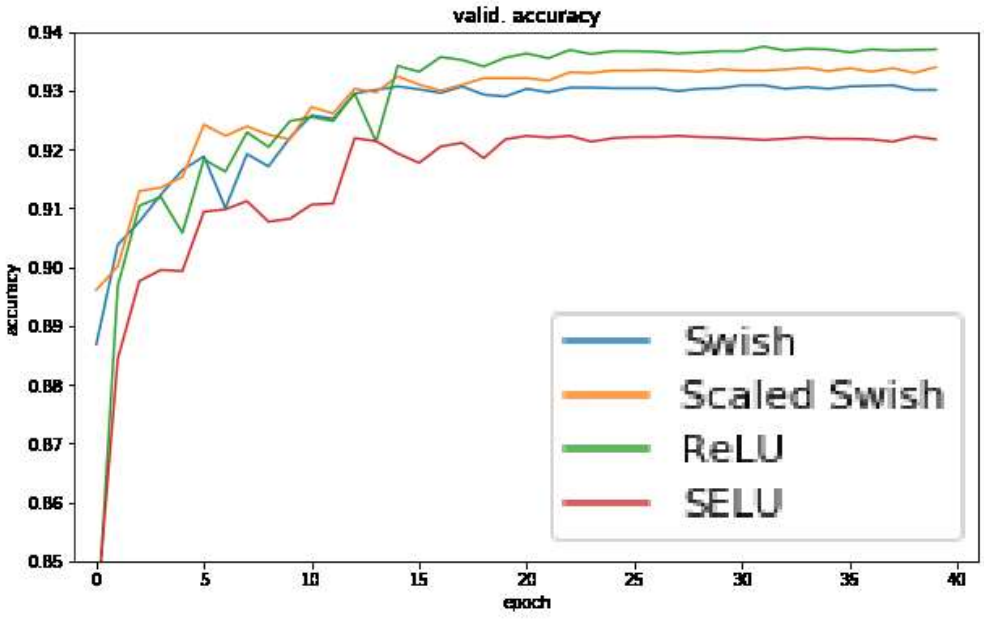
So which activation function to use?



Accuracy	CNN Models					
Activations	MobileNet	VGG16	GoogleNet	ResNet50	SENet18	DenseNet121
Sigmoid	88.60 ± 0.17	87.69 ± 0.49	87.33 ± 2.48	80.13 ± 3.33	90.29 ± 0.29	89.92 ± 1.96
Tanh	87.21 ± 0.24	90.49 ± 0.11	90.16 ± 1.86	89.09 ± 1.47	90.44 ± 0.09	91.80 ± 0.69
Elliott [46]	88.48 ± 0.18	87.94 ± 0.49	89.84 ± 3.43	81.60 ± 3.91	90.25 ± 0.25	91.53 ± 1.04
ReLU [82]	90.10 ± 0.22	*92.84* ± 0.19	*93.43* ± 0.48	93.74 ± 0.34	93.70 ± 0.16	**93.96** ± 0.51
LReLU [103]	90.10 ± 0.19	91.09 ± 0.09	89.28 ± 0.82	*93.83* ± 0.42	93.66 ± 0.19	93.85 ± 0.48
PReLU [58]	90.43 ± 0.18	92.19 ± 0.08	92.85 ± 0.55	92.99 ± 0.62	92.76 ± 0.26	92.82 ± 0.63
ELU [29]	90.92 ± 0.25	88.55 ± 1.17	92.47 ± 0.76	93.53 ± 0.66	93.39 ± 0.20	92.89 ± 0.62
SELU [77]	90.11 ± 0.32	92.25 ± 0.28	91.87 ± 0.84	93.53 ± 0.52	89.96 ± 0.31	92.71 ± 0.73
GELU [61]	90.71 ± 0.20	92.42 ± 0.09	93.16 ± 0.61	93.81 ± 0.46	**93.72** ± 0.18	*93.90* ± 0.41
CELU [14]	*91.04* ± 0.17	88.11 ± 0.14	92.60 ± 0.60	**94.09** ± 0.17	91.63 ± 0.22	93.46 ± 0.35
Softplus [158]	**91.05** ± 0.22	92.69 ± 0.20	92.66 ± 0.66	93.34 ± 0.65	93.25 ± 0.11	93.07 ± 0.70
Swish [118]	90.66 ± 0.34	92.32 ± 0.20	92.68 ± 0.53	93.02 ± 0.85	93.24 ± 0.19	93.16 ± 0.51
ABReLU [34]	88.97 ± 0.47	92.36 ± 0.15	93.34 ± 0.23	93.29 ± 0.52	93.35 ± 0.14	93.26 ± 0.55
LiSHT [119]	86.53 ± 0.49	89.83 ± 0.28	90.27 ± 0.80	90.89 ± 0.66	90.25 ± 0.84	87.91 ± 0.93
SRS [159]	89.43 ± 0.81	92.06 ± 0.26	91.36 ± 1.19	92.28 ± 0.48	78.05 ± 1.37	90.64 ± 1.93
Mish [100]	90.82 ± 0.15	**92.85** ± 0.25	93.29 ± 0.61	93.69 ± 0.63	93.66 ± 0.12	93.62 ± 0.62
PAU [101]	90.67 ± 0.17	92.00 ± 0.26	92.80 ± 0.65	93.67 ± 0.52	93.08 ± 0.20	93.05 ± 0.53
PDELU [27]	90.18 ± 0.19	92.80 ± 0.13	**93.49** ± 0.30	93.42 ± 0.71	*93.71* ± 0.07	**93.96** ± 0.59

Training Time	CNN Models					
Activations	MobileNet	VGG16	GoogleNet	ResNet50	SENet18	DenseNet121
Sigmoid	00:33:15	00:49:16	04:55:54	03:36:03	01:13:14	04:12:24
Tanh	00:33:18	00:49:55	04:58:02	03:33:03	01:13:18	04:09:24
Elliott [46]	00:49:52	00:59:13	06:53:55	05:38:49	01:41:38	07:46:55
ReLU [82]	00:31:22	00:47:19	04:55:10	03:32:30	01:15:33	04:15:06
LReLU [95]	00:31:48	00:49:03	05:01:30	03:33:00	01:18:38	04:14:09
PReLU [58]	00:44:24	00:49:01	05:42:18	03:55:57	01:27:05	04:55:47
ELU [29]	00:31:05	00:47:38	04:57:37	03:36:47	01:13:25	04:08:39
SELU [77]	00:29:40	00:47:31	04:54:57	03:33:47	01:13:27	04:09:17
GELU [61]	00:29:43	00:47:22	04:55:53	03:32:32	01:13:32	04:11:26
CELU [14]	00:29:36	00:46:47	05:00:44	03:31:40	01:14:08	04:18:11
Softplus [158]	00:29:44	00:47:06	04:58:55	03:32:03	01:14:02	04:12:08
Swish [118]	00:43:13	00:55:37	06:18:38	04:58:38	01:32:15	06:41:14
ABReLU [34]	00:38:51	00:53:49	05:43:59	04:27:02	01:25:30	05:42:53
LiSHT [119]	00:37:01	00:54:10	05:40:00	04:25:57	01:23:59	05:38:15
SRS [159]	01:06:38	01:11:36	08:43:09	07:35:35	02:05:33	11:10:27
Mish [100]	00:40:19	00:54:23	05:59:48	04:46:45	01:28:53	06:10:27
PAU [101]	00:41:59	00:54:10	05:54:22	04:12:31	01:25:37	05:39:57
PDELU [27]	05:23:38	04:01:55	34:22:00	36:48:48	08:32:40	50:23:00

Want to keep things simple? **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 theirs is better, but then later peer group would release their studies that 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.

Want amazing things at cost? **GELU**

Test Error Comparison

Dataset	GELU	ReLU	ELU
CIFAR-10 (vision)	7.89%	8.16%	8.41%
CIFAR-100 (vision)	20.74%	21.77%	22.98%
TIMIT (audio)	29.3%	29.5%	29.66%
Twitter POS tagging (nlp)	12.57%	12.67%	12.91%

Median Test Error Rate after 5 training runs.

Read more **here (https://towardsai.net/p/l/is-gelu-the-relu-successor)**

References

1. (https://www.v7labs.com/blog/neural-networks-activation-functions) Interactive Introduction to Fourier Transforms (https://www.jezzamon.com/fourier/)
2. Why do neural networks work so well - Quora (https://stackoverflow.com/questions/38595451/why-do-neural-networks-work-so-well)
3. What is a Convolutional Neural Network (https://poloclub.github.io/cnn-explainer/)

4. Feature Visualization - How Neural networks build up their understanding of images (https://distill.pub/2017/feature-visualization/) .
5. Computing Receptive Fields (https://distill.pub/2019/computing-receptive-fields/)
6. Deconvolution and Checkerboard artifacts (https://distill.pub/2016/deconv-checkerboard/)
7. Activation functions in Neural Networks (https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6)
8. Is GELU, the ReLU successor? (https://towardsai.net/p/l/is-gelu-the-relu-successor)
9. Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark (https://arxiv.org/pdf/2109.14545.pdf)

The Assignment

Refer to this **Colab File** (https://colab.research.google.com/drive/1oVt7T6tb90Y1EXvFaIWgm72emqZZPXi4?usp=sharing) You need to go through this file, perform tons of experiments, and then proceed to answer the questions in S4 - Assignment QnA.

Please note that you only have 90 minutes to answer the assignment, 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.

Videos

Studio

Google Meet