Berkeley Math Circle - Intermediate 2: Convolutions (after wrapping up Flexagons)

230503 - Chris Overton (lecture 2, handout after class)

- These notes are not meant to be quite self-contained, but may help you remember key points we discussed in more dtail in class
- When you see: <---- SPOILER ALERT----> (abbreviated as <----->) , please challenge yourself to answer before reading on.
- --> At the start of a line indicates a question to think about before reading further

Plan for today:

1. Wrapping up Flexagons

- quick recap of last time
- non-isomorphic 6#6's (hexa-hexaflexagons)

2. Convolutions

   Motivation: "classical" convolutions

- Fourier analysis
- Wavelets
- Convolutions (in Fourier analysis)
- DCT (discrete cosine transforms)

The main goal of today's talk:

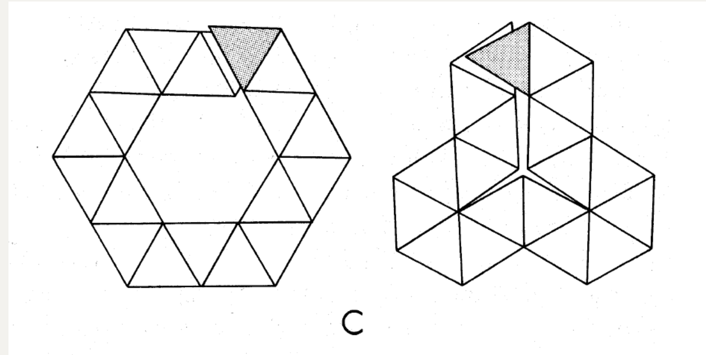
- Convolutions in neural nets

Topic 1: Wrapping up flexagons...

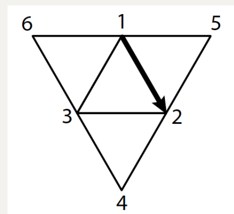
For background, see last week's handout & its references

Are there additional examples of 6#6 besides the one we built?

Hint: consider the appendix to Gardner's haxaflexagon chapter for recipes, and consider whether graphs (or 2-simplexes) are equivalent



Remember the simplicial complex we got for the 6#6 made from a strip of 18 triangles: a triangle made of 4 sub-triangles, each of these contributing an oriented 2-simplex:



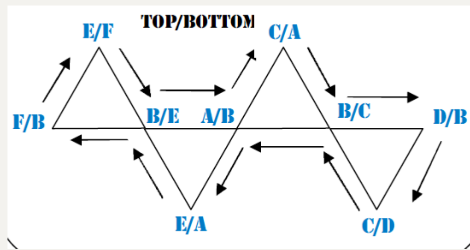
<--->

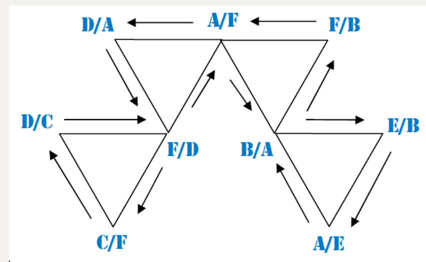
Hints:

It's actually a bit tricky to know how to fold the diagrams above to make 6#6's (e.g. where you can get to each side.) We'll take a shortcut and use diagrams and instructions from ht tps://flexagon.net/

In these diagrams, each vertex is labeled as the sides on top and on bottom of a hexaflexagon. This correspoonds to edges in the "Feynman" diagrams we used last week.

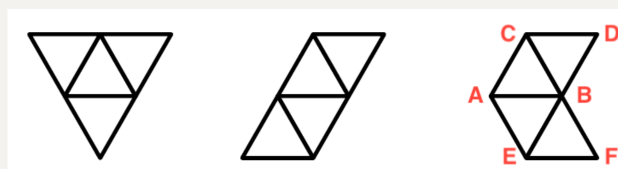
Here are Tuckerman-style diagrams respectively for the two paper layouts above:





Now to generate for example the first Feyman diagram (but rendered as triangles rather than inside regular polygons):

- Notice that vertices B/E -> A/B above corresponds to an edge in a Feymnan diagram doing from B to A, with these in a triangle also including E.
- However, above, you also see C/A -> B/C, which in a Feynman diagram would correspond to and edge between vertices A and B, part of a triangle with other vertex C.
- Continuing like this, we obtain the vertex labeling shown below at right:



- Similarly, the second Tuckerman-style diagram above corresponds to the middle Feynman-style diagram.

--> Prove these are different simplicial complexes that cannot be shifted into each other just by permuting their corners!

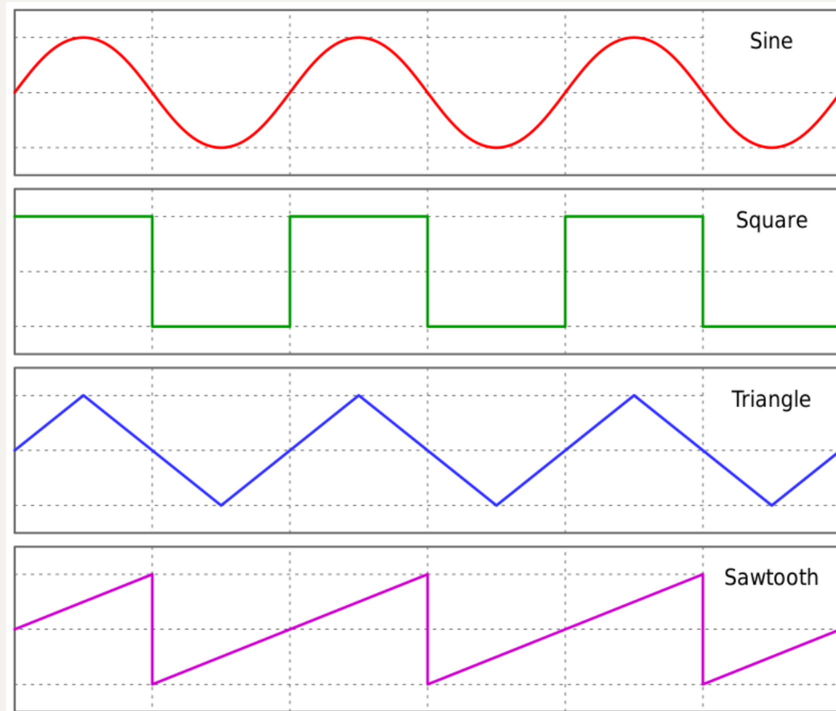
<---->

Proof: for example, the three diagrams differ in how many vertices have four edges!

So we conclude there are at least 3 "different" 6#6's. It turns out these are all of them (but I still haven't convinced myself by reading through a proof!)

Topic 2: Convolutions!! (new topic for today)

How do you tell what frequencies these waves are? Do they have the same frequency?



<---->

We said they each share the same **fundamental** frequency (because of same periodicity), but differ in **overtones** (named after me - just kidding.) Only the first sine wave has no overtones.

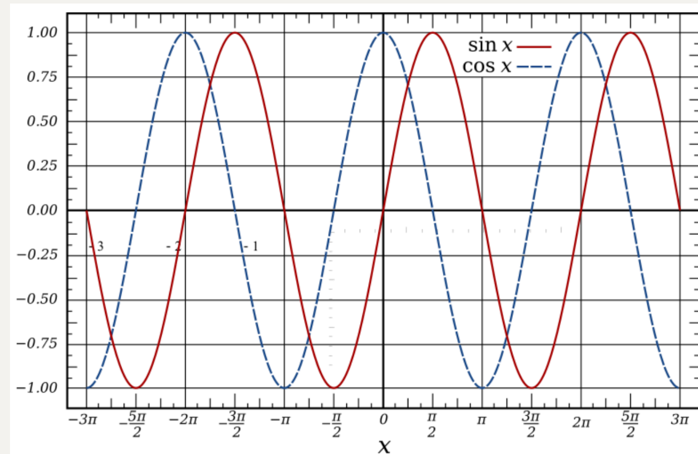
In class, we took a shortcut to detection of a given frequency: multiply a function as above by a sine wave, and **integrate** to add the area underneath a curve:
$\int_0^{2\pi} square(x) * sin(x)dx$

(We also showed how to "cheat" to figure out certain areas without doing calculus:
$\int_0^{2\pi} sin(x) * sin(x)dx = \int_0^{2\pi} cos(x) * cos(x)dx$ by horizontal motion along the x-axix, and these add up to the area of the surrounding rectangle with height 1 and width 2π because $sin^2(x) + cos^2(x) = 1$.)

If square(x) were to slide horizontally (change in phase), detected area could wander between positive and negative, depending on positive and negative values of square(x) align with those of sin(x))

Conclusion: if just doing direct multiplication, you need comparison with more than one (real) function of a given frequency

It turns out you just need two:



It turns out sine and cosine are really just (imaginary and real) parts of the same thing:
$e^{i\theta} = cos(\theta) + i * sin(\theta)$

Fourier transform: time domain <--> frequency domain

Need infinitely much of one to completly resolve the other!

The Fourier transform **transforms** a wave (function of **time**) into a function of **frequency**

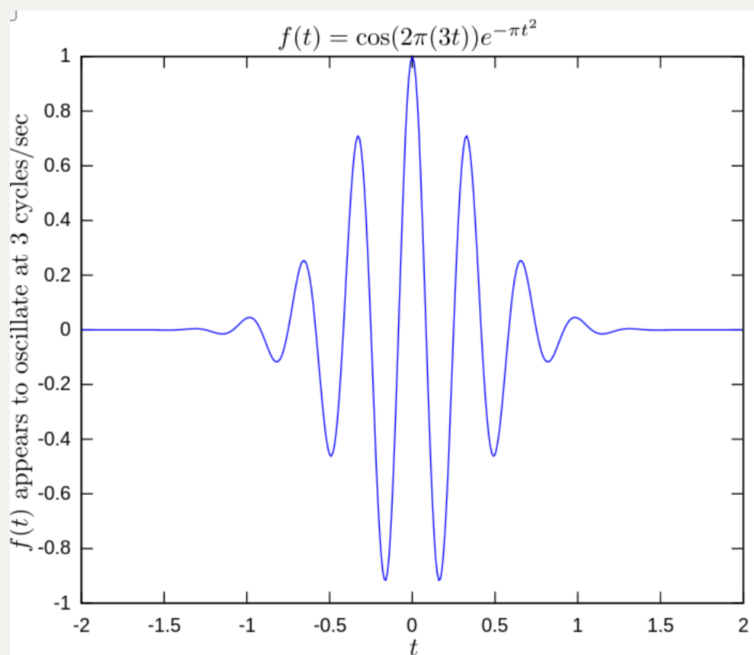
Fourier transform

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-i2\pi\xi x}\, dx.$$

Wavelets

What if you don't have an infinite amount of time?

Then you have to "cut corners" - more precisely: instead of e^{ix}, use a function that decays to zero for small and large x. Example:



Now imagine the kind of stuff you can start "seeing" with these!

<whiteborad: simultaneous localization by time and freq>

Note that the shorter window of wavelets results in greater ambiguity in detecting frequency

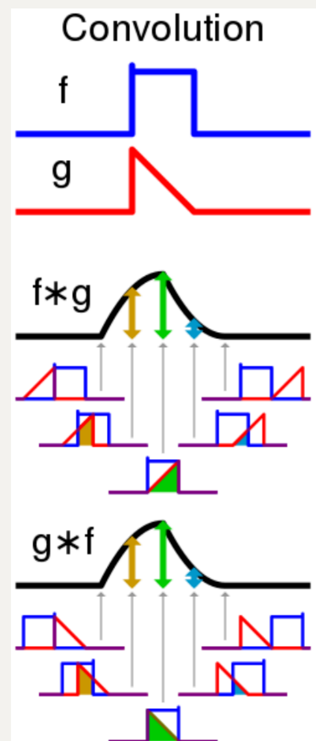
Convolution

How do you use a wavelet or other "kernel" (like the one above) to "see" or "detect" oscillation in a function?

Answer: **convolution**. This is the operation where a kernel function "slides past" another function to reveal a transform. First the formula, then an example.

Note how the second function is flipped horizontally, and note how the diagram shows particular values calculated for specific values of t (tan, green, turquoise colors)

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t - \tau) \, d\tau.$$



Continuous vs. discrete

So far, we had a continuous signal. But in real life, while this may be simulated by analog signals (e.g. volume on a phonograph record), we have now moved to the digital world of samples taken at a fixed frequency (like 44kHz for CD's)

How do you deal with discrete values?

A first attempt is just to change the inputs from continuous x to a value given by multiples of integer n:

$$X_{2\pi}(\omega) = \sum_{n=-\infty}^{\infty} x[n]\, e^{-i\omega n}.$$

But how do you do this given limited time history $|n| < K$?

Enter the **Discrete cosine transform** ("DCT")

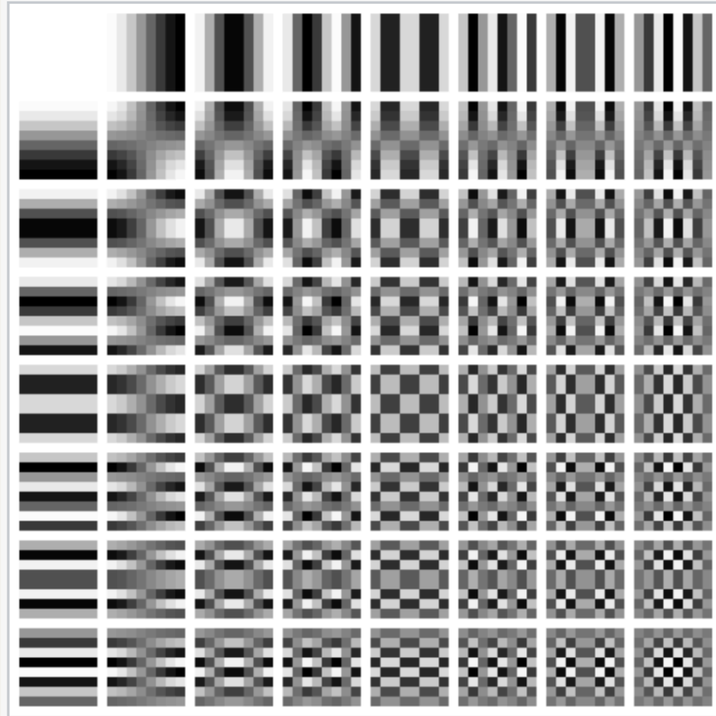
There are several formula versions, but here is the most common one:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right) k\right] \qquad \text{for } k = 0, \ldots N-1$$

Notice that the input is x_n in time domain, and one obtains output X_k of the same size, but in frequency domain.

This can also be done for higher-dimensional x, such as 2d. The following graph shows the "kernels" that one "convolves" with, in 8 by 8 two-dimensional DCT.

As one moves further right (respectively down), kernels include more horizontal (resp. vertical) wiggles:



Efficient computer implementations are called the **Fast Fourier Transform** ("FFT".)
Note: The best known algorithm is called the "Cooley-Tukey" algorithm - the same Tukey we heard of last week on the Flexagon Committee.

Inverse transform & uses

Both continuous and discrete transforms can be run backwards: from frequency to time. This can reconstruct the original function or sequence, except perhaps you get a more continuous version.

If you leave out higher-frequency components, you get a "less wiggly" time domain version. This can be used for **data compression**.

===================

These convolutions motivate the material we want to get to: **neural nets**! ("**NN**")
(sometimes with the annoying brand name "deep learning.")

If you have been paying attention, you have noticed that NN's are doing all kinds of unprecedented new things, such as, identifying objects in images. NN's that do this use the technique they are named after:

Convolutional neural nets ("CNN")

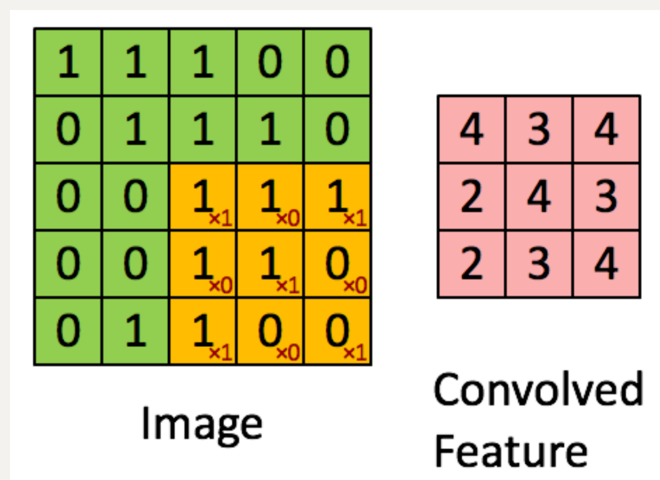
Around 1989, Yann LeCun and others demonstrated CNN's as a new way to succeed at the problem of recognizing handwritten digits (industry-standard "MNIST".)
Progress was being made with statistical learning techniques (classification algorithms running on cleverly pre-computed features), but CNN's quickly surpassed prior approaches in their accuracy and generalizability.

CNN's have now become one of the two main threads of NN, featuring prominently for example in self-driving cars. The other thread is natural language processing ("NLP"), including chatbots, and large language models "LLM".

At the most basic level, a **convolution layer** in a neural net has a **kernel** (called a "**filter**", or described as resulting in a "**feature**") that slides over an input. At each position, the value returned is a "dot product", but may be higher-dimensional than the usual vector dot product.

You can think of an image as a matrix of pixel brightness values.

Here is an example of convolution with the 2-d filter $\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$



--> Convince yourself the other numbers are correct!

--> What kind of image "feature" can this filter detect?

<---->

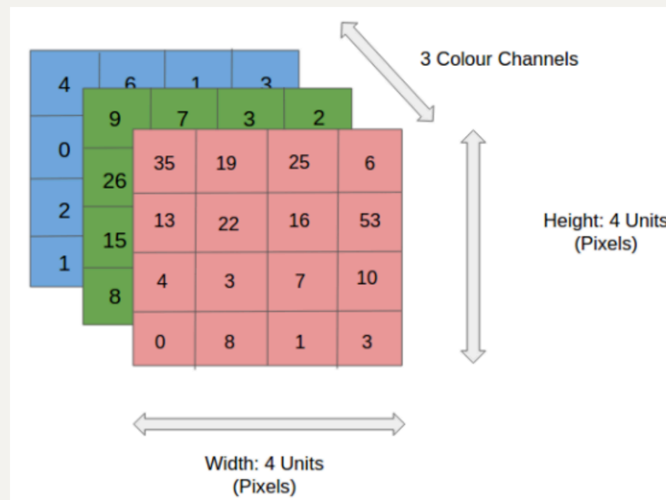
(Sort of: the centers of diagonal X shapes, but this filter cannot distinguish between these and all 1's)

This operation resembles the kind of discrete convolution we discussed earlier (e.g. for DCT.) That similarity led to calling these NN's convolutional.

Tensors

Specifying a color requires three numbers. Color movies add yet another dimension, say for 30 frames per second. The data structures used to hold these are called **tensors**: respectively 3-4 and 4-d for color pictures and color movies.

Here is a 4 by 4 by 3 color image 3-d tensor:



This explains the name "tensorflow", a leading neural net software platform that was launched by Google.

I call images like the ones above **content tensors** to distinguish them from filter tensors.

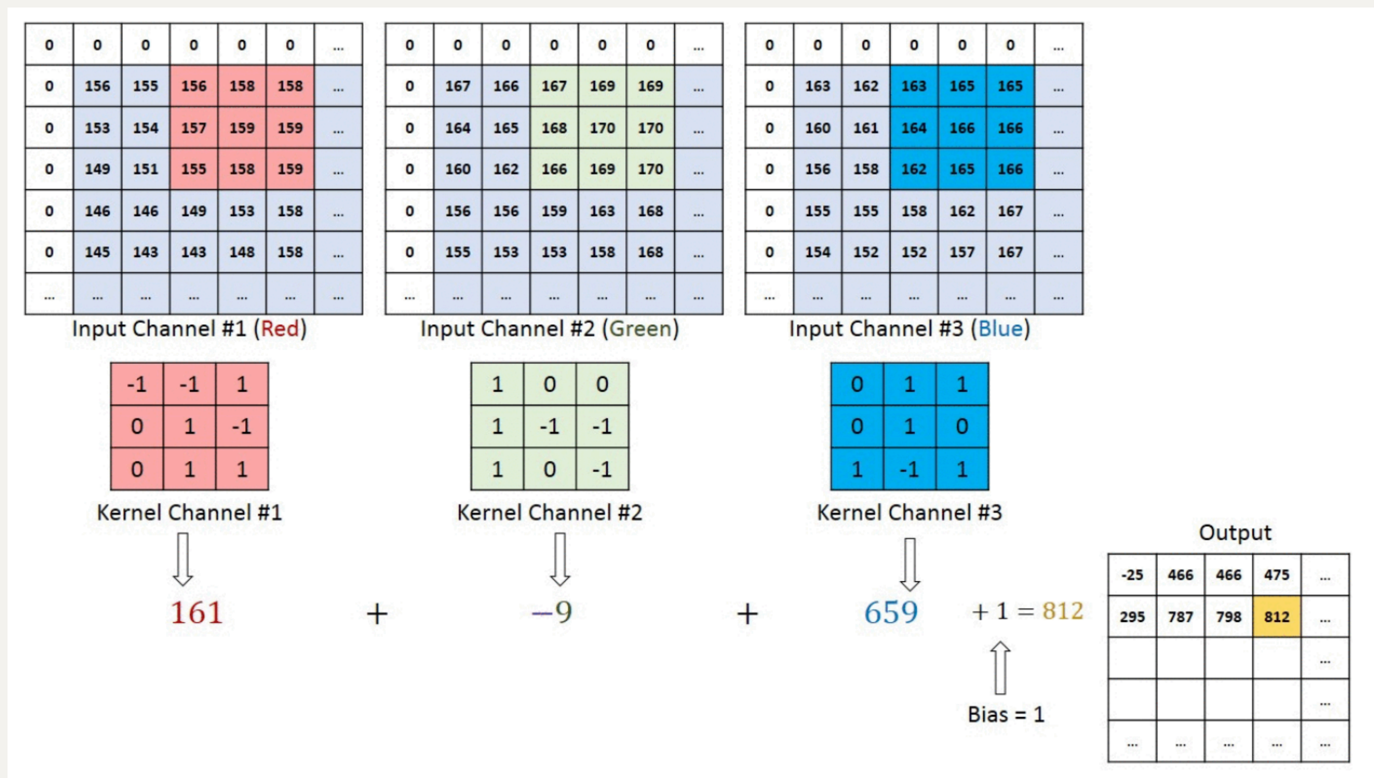
Tensor operations are like a generalization of linear algebra to higher dimensions.

You have to get comfortable with all the index gymnastics, but there are just a few common tensor operations. Software platforms are now making it easier to glue together tensors of the correct dimensions.

Additional settings for convolution ("conv") layers

- Which dimensions of the filter sweep over what positions in the content tensor?
- You might have **padding**: extra rows/columns of zeroes added around image edges of the content tensor. The can prevent the output tensor from shrinking

Here is the beginning of a conv of an $M \times N \times 3$ image by a $3 \times 3 \times 3$ filter, with one row & column of padding. This one also adds a "**bias**" term, which is a general NN tool that might be added to get better default behavior.



- You can also have a **stride** > 1. This says how many units over a filter moves for each dot product.

--> Putting these together, we get a little algebra problem: along a chosen dimension, given content size C, padding width P, filter lengh F, and stride S, how big is the output along this dimension?

<----->

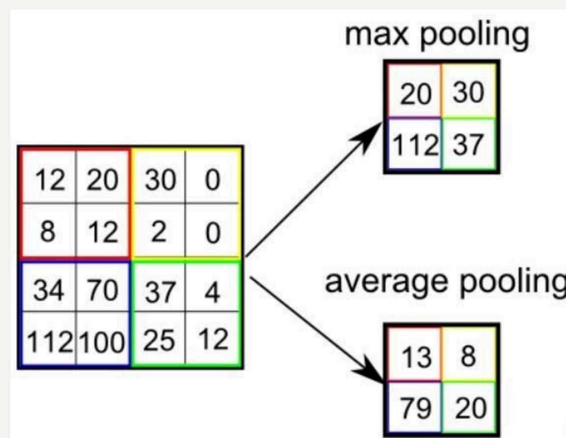
Positioning the filter at the beginning of the content + padding (say in the x-dimension), the number of positions "left over" is $C + 2P - F$. Counting this original position plus the number of possible stride steps gives:
$1 + \lfloor (C + 2P - F)/S \rfloor$

The activation function

In neural net architecture diagrams (like below), you might see weird names like "Relu" and "Softmax." These are just functions acting on individual values that return a tensor of the same size. They add the nonlinearity that is critical to learn non-linear patterns.

Pooling

These operations could be thought of as a matrix operation in 2d, but conventional notation is not that helpful. It should be clear what the following two operations max and average pooling do:



--> Which of these would do a better job preserving features of a grayscale image?

<---->

You would think that average pooling is more mathematically "honest", but for example in the case of pixel brightness, this could average between say a bright bird back and a dark background to produce colors not seen before. For this kind of reason and more generally to pass through the strongest "signals", one generally prefers max pooling.

You can see that this 2×2 pooling shrinks each of the first two dimensions of a tensor by 2 (leaving other dimensions intact.)

Upsampling

We don't illustrate its use here, but when you layer several pieces with pooling, a later NN layer might have much smaller data dimensions than at the start (called a "**bottleneck**.") Several important NN's try to re-create original data from such a bottleneck. These are called **autoencoders**.

Even though information has been thrown out by a pooling layer, data of the original size can be generated by **upsampling**: repeating the same value to fill up a larger tensor. Upsampling layers specify the same number of dimensions as a conv layer, but they tend to increase size rather than reduce it.

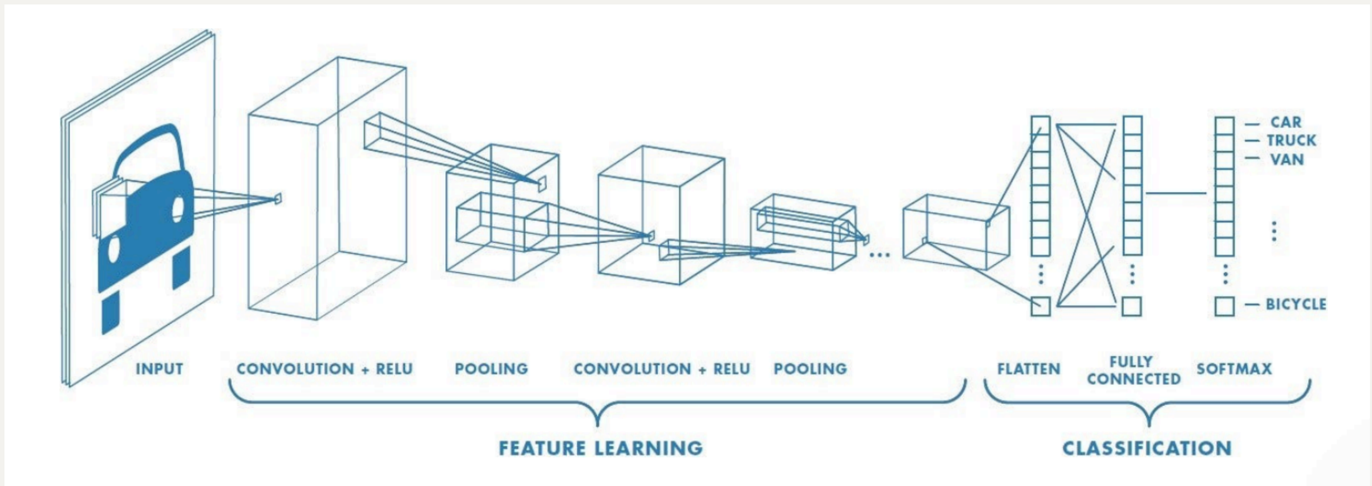
The architecture of conv nets

Here is a relatively simple architecture for image recognition. This is a "**feedforward**" NN, namely in "**production**", data flows in one direction, here from left to right.

In order to make the network learn what it is supposed to, this *modeled* output is compared to known, *correct* output in a **training set**, and during **training**, the gradient of each weight in the network is used to adjust the weight, so as to reduce its contribution to **loss** - namely the penalty for how modeled output differs from correct output.

This correction goes "backwards" (in this picture from right to left.) This motivates the term **backpropagation** - the main "secret sauce" of NN.

There can be billions of weights, and training can take millions of steps, maybe run over thousands of processors.



The boxes in the portion labeled "feature learning" represent multiple conv and pooling layers.

The "depth" (left to right dimension) in conv layers indicate that that there are multiple filters, each operating independently. However, pooling layers pool over only two dimensions, so they shrink only these dimensions and leave the depth dimension intact.

--> Understand the "input" and "output" boxes shown for each operation!

---> Assuming the first two conv layers are 2×2 and pooling layers are 2×2, can you think of filters that could be good at detecting vertical lines of width 1 pixel? How about widths of 3 pixels?

<----->

One idea: $\begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$ for width 1 pixel, and maybe to detect a width of 3 pixels, the same filter in the second conv layer would approximate...(after size has shrunken by a factor of 2)

You might think that you hire high-paid specialists to dream up all the right filters to use, analogously to what we saw earlier today as in wavelets.

The amazing thing about NN is that you usually just set these to be random, and let NN training take care of discovering what works!

However, in real life, more complex NN might build in custom engineered features made by specialists ("**feature engineering**.")

Conclusion

We have motivated convolutions in NN's by their older use in subjects like Fourier analysis.

CNN's have been tremendously and surprisingly successful, and are a key component in a large portion of NN's.

At this point, you need much less math to understand, train, and run CNN's than was anticipated by the inventors of earlier convolutions.

But though it is fairly standard, the kind of tensor math you need is quite critical to get right.

One NN luminary said recently that math is the single most valuable skill set needed to do well with NN's. This is particularly important in choosing architectures that produce good results with cost-effective computation.