We'll describe how to build a convolutional neural network. We need two new layers: **conv2d** and **flatten**. Pooling layers are useful but not technically necessary.

1 conv2d

The general idea: conv2d wants to take an incoming image and transform it into an alternate more useful representation of the image by representing important features / not including useless noise from the input image, in the output image.

- The structure of how conv2d works helps converge on images faster, because we've already done part of the work for the model with the kernel structure — dense nets would need to learn how to emulate this behavior on their own to achieve the same results.

So we accept input X with shape $[N, C_{in}, H, W]$, and output A with shape $[N, C_{out}, H', W']$.

- N: batch index
- C_{in}: number of incoming channels (for example, for raw image data, 3 channels might come in for RGB)
- C_{out} number of output channels. (basically never interpretable to us)
- H, W: height and width.
- H', W': output height and width. ($H' = H-k+1$, $W' = W-k+1$ — assume stride 1 and padding 0.)

The "kernels" are what conv2d uses to create the output images. Imagine sliding a 3D $C_{in} \times k \times k$ window across every possible spot on a single $[C_{in}, H, W]$ image: this single kernel dots against every region it overlays to create a new singular pixel in its channel in its output image.

- This single kernel contributes to **one** channel in the output image — we use C_{out} kernels to create our full output images.

We could just kernel-dot our way through — if you've implemented dense layers, you likely have hadamard and sum as an autograd primitive. But in practice, implementations like `im2col` are used because we want matmul optimizations.

im2col converts an image of shape $[C_{in}, H, W]$ to $[H' \cdot W', C_{in} \cdot k^2]$ (for kernel size $k \times k$).

- After `im2col`, axis 0 gives us the index of the pixel in the output image, and axis 1 gives us an index into the $k \times k$ patch for each input channel, that yields that output image pixel.
- To carry out this op, do the following:
  - **Flatten input**: X.reshape($[C_{in} \cdot H \cdot W]$). The image is now a row which is composed of C_{in} sequential segments, each segment containing H segments of length-W pixel rows.
  - **Precompute an index tensor**: I is of shape $[H' \cdot W', C_{in} \cdot k^2]$. This tensor indicates that we want to create a new tensor X' where $X'[i, j] = X[I[i, j]]$.
  - **Gather**: Use our index tensor I: $X' = X$.gather(I)
- Gather may be a new autograd primitive depending on your autograd progress — the backprop for gather is simple, though. Think: for each index tuple i, what is $\dfrac{\partial J}{\partial X[i]}$? When we change $X[i]$, then all $X'[j]$ change where $I[j] = i$, and we already know how J changes with $X'[j]$ (autograd rule). So $\boxed{\dfrac{\partial J}{\partial X[i]} = \sum_{j:I[j]=i} \dfrac{\partial J}{\partial X'[j]}}$.

For forward: our weights W are shaped $[C_{in} \cdot k^2, C_{out}]$ — each column in W which is an individual kernel dots with each row in $X' = \text{im2col}(X)$ (im2col works independently across batch examples) which is an individual patch. It does this for each patch corresponding to each output pixel, as there are $H' \cdot W'$ patch rows in X', each row corresponding to an output pixel.

- Now we have $X'W$ which has shape $[H' \cdot W', C_{out}]$ — columns are each a single output channel, with channel data represented as H' segments of W'-length runs of pixels down the column.
- Add a bias b of the shape $[C_{out}]$ so output image expressibility is complete.
- So we have an intermediate $X'W + b$ of shape $[N, H' \cdot W', C_{out}]$ — permute to get $[N, C_{out}, H' \cdot W']$, then reshape to get $[N, C_{out}, H', W']$.
- conv2d's output is therefore: $\boxed{(\text{im2col}(X)W + b).\text{permute}([0, 2, 1]).\text{reshape}([N, C_{out}, H', W'])}$.

2 flatten

Just take in X with shape $[N, C_{in}, H, W]$ and output $A = X$.reshape($N, C_{in} \cdot H \cdot W$). No parameters.