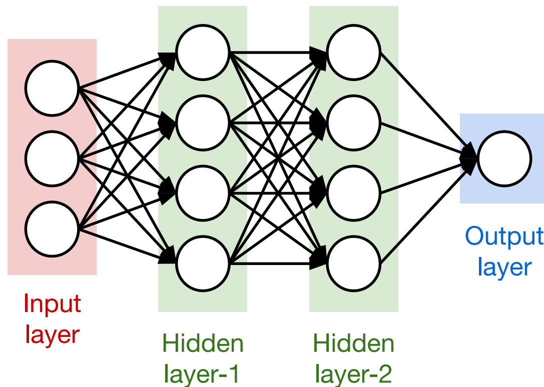
NOTE: This is a draft for [CS 3780/5780] **Lecture 23: Convolutional neural networks**. Do not distribute without explicit permission from the instructors.

Lecture 23: Convolutional neural networks

CS 3780/5780, Sp25

Tushaar Gangavarapu (TG352@cornell.edu)

Over the past few lectures, we discussed fully-connected, feed-forward networks.



Starting with $x^{(0)} = x \in \mathbb{R}^3$ (in the example to the left), and using an element-wise nonlinearity $g(\cdot)$, like ReLU, we have:

$$x_1^{(1)} = g(W_{11}^{(1)} x_1^{(0)} + W_{12}^{(1)} x_2^{(0)} + W_{1d}^{(1)} x_3^{(0)} + b_1^{(1)})$$

$$\vdots$$

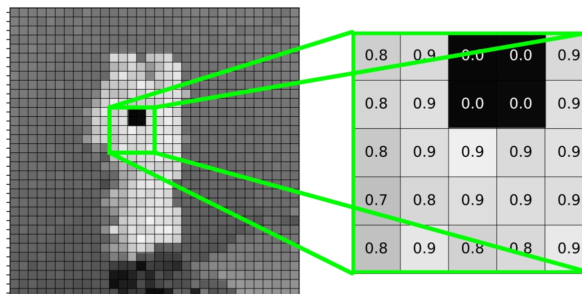
$$x_4^{(1)} = g(W_{41}^{(1)} x_1^{(0)} + W_{42}^{(1)} x_2^{(0)} + W_{4d}^{(1)} x_3^{(0)} + b_4^{(1)})$$

Or, more compactly, in matrix form,

$$x^{(1)} = g(W^{(1)} x^{(0)} + b^{(1)}),$$

where $W^{(1)} \in \mathbb{R}^{4\times 3}$, $b^{(1)} \in \mathbb{R}^4$, and $x^{(1)} \in \mathbb{R}^4$. Sometimes, for notational convenience, the above is also written as $x^{(1)} = g(\text{linear}_4(x^{(0)}))$. Beyond the compactness of using the matrix form, it makes it really convenient to discuss runtime complexities and we have fast hardware-aware linear algebra routines for running matrix multiplications. Two pieces of advice: (1) think in matrices when you can, because we know how to parallelize matrix multiplications, and (2) always check dimensions!—for e.g., $W^{(1)}$ is a 4-by-3 matrix, which is multiplied with a 3-dimensional vector, which results in a 4-dimensional output vector.

Now, we wish to build a "Berry-vs.-llama" detector—given a (32×32 RGB) *image*, identify if it is of Berry (the class mascot) or not. Let's start with a grayscale image and see how we can process it using a fully-connected net:



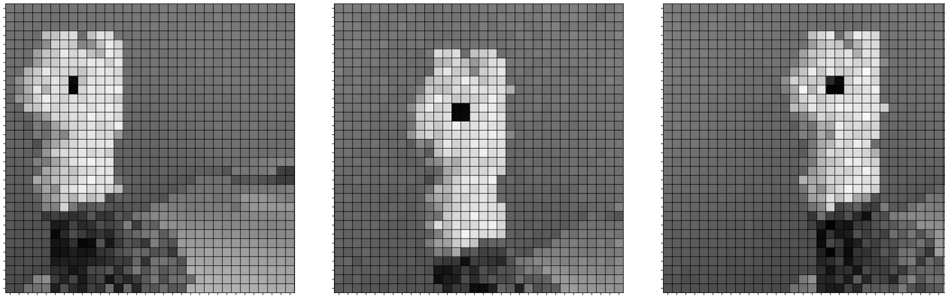
Since each pixel in a grayscale image is a value between 0.0 and 1.0 (e.g., 0.0 is black), we can "flatten" the image by scanning it from left-to-right, top-to-bottom. The image is now a vector of $32 \times 32 = 1,024$ dimensions, which can then be processed by our fully-connected net.

Even before realizing *if* such image processing is meaningful, one can easily see issues with the scalability of a fully-connected network. A single neuron in first hidden layer would have $32 \times 32 = 1,024$ weights associated with it. More realistically, a 300×300 RGB image would have $300 \times 300 \times 3 = 270,000$ weights! Adding more layers would quickly add up the total number of model parameters. As noted earlier, more parameters means model complexity is higher, meaning possible overfitting.

Now, let us reason if such an approach of flatten-then-use-fully-connected-net is truly meaningful in the context of images (which will roughly apply to text as well). There is a global and

local structure built into images and looking at what a single pixel captures might be less informative (and meaningful) than, say, looking a pixel grid as a whole. For example, in the image above, the green bounding box capture the “presence of an eye in the image,” while looking at any individual pixel wouldn’t give us that information.

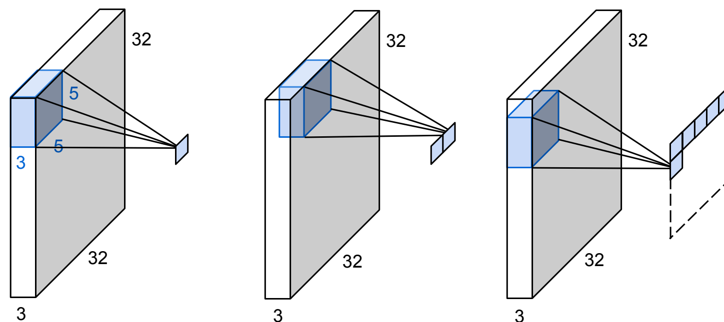
As an extreme (but not impossible) example, if our fully-connected net is trained on images of a centered-in-frame Berry, to spot the llama’s eyes, then the network would fail to identify eyes in any translated (not centered- in-frame) version of the image. Why?—the network learns to associate a black-dot in the center-ish region of the image with the presence of an eye.



All-in-all, we wish to design approaches with more appropriate inductive bias[1] than fully-connected nets for image processing.

1 Preserving spatial structure: Convolution layers

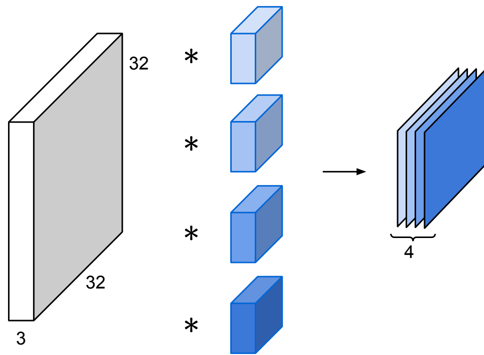
Following the goal of preserving the spatial structure, we will attempt to operate on images in patches. More concretely, we will learn a set of filters that when *convolved* with the image, can extract features from the image. Here, by “convolving,” we mean the convolution operation (denoted by $*$): slide the filter across the width and height of the image, and compute the dot product between the filter and the underlying image patch being slid on. A single convolution operation (read: one plop-the-filter-and-take-dot-product) results in a single number.



Observe that the filter is spatially small (small width and height) but extends to the full depth of the input image. But, more importantly, the $5 \times 5 \times 3$ filter (or, 75 weights and a bias) is *shared* across the entire image, unlike in a fully-connected net.

Intuitively, we hope the the filters are learned such that they are activated when a visual feature is detected. Now, we can learn as many filters as we would like; this is analogous to each filter as being responsible for identifying different features (e.g., eyes, mouth, etc.).

[1]Inductive bias means biasing the underlying models to utilize some kind of assumed structure in the data; for e.g., nearby pixels are similar, there are similar features (say, eyes) in different locations, bigger features are made of smaller features.



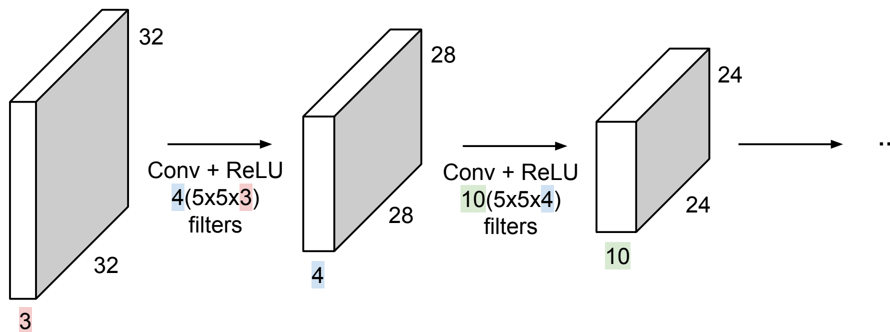
We stack these output feature maps together, which is then passed through an element-wise nonlinearity (e.g., ReLU), similar to that in a fully-connected net.[23]

The output dimension. Before we proceed, one (obvious) question to ask is: what is the dimensionality of the output from a convolution layer. For a $32 \times 32 \times 3$ image, convolved using a $5 \times 5 \times 3$ filter, what is the shape of the resultant feature map? Before answering that, we need to define *how much* we slide the filter each time, a.k.a., "stride"—say, move to the right or bottom by one pixel.

Now assuming movement by one pixel, if we had a $5 \times 6 \times 3$ image, we would get 1×2, a $5 \times 7 \times 3$ image would result in a 1×3, and so on. For a $5 \times n \times 3$, we would get $1 \times (n-5)+1$, and following the same logic along the height, for an $n \times n \times 3$, the resultant output would be $(n-5)+1 \times (n-5)+1$. Thus, for a $32 \times 32 \times 3$, our resultant feature map is 28×28.

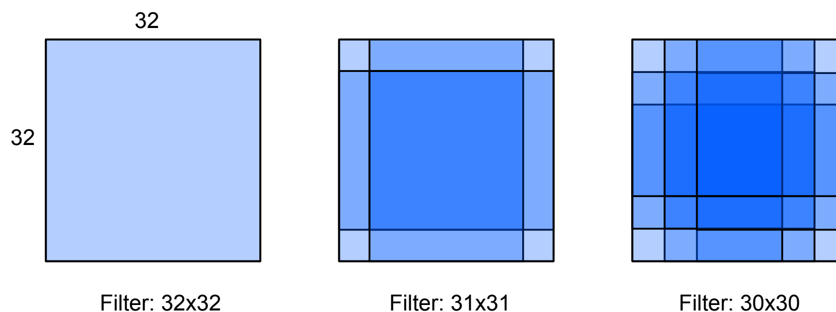
2 Convolutional networks

For completeness, we note here that a convolutional net is a neural network with one or more of convolution-layer-followed-by-activation. A typical convolutional net is as follows:



Same as with fully-connected nets, convolutional nets are trained using a suitable optimizer (such as stochastic gradient descent) and backpropagation. As it turns out, upon inspection of learned filters and feature maps in a network with several convolutional layers, Zeiler and Fergus (2014, Fig. 2) observed that earlier layers tended to learn low-level features such as edges, color blobs, etc., while more deeper layers learned filters to capture more complex patterns.[4]

Padding. Clearly, as the number of convolutional layers increase, the size of the feature map goes down; this is exacerbated with a larger stride.

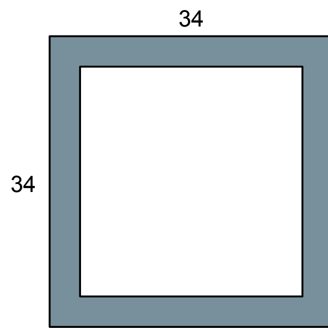


[2]To be explicit, we consistently use *feature* map to indicate $\text{conv}(\text{image})$, i.e., without any activation, and *activation* map to indicate $g(\text{conv}(\text{image}))$.

[3]Without nonlinearity, stacking multiple convolutions achieves the same effect as having a single convolution.

[4]Neuron microscope: https://poloclub.github.io/cnn-explainer/.

This shrinkage is resultant of sampling the edges and corners of an image lesser than the rest of the image. Above, (in an extreme example,) we show the coverage of different filters with varying widths. Observe that the corners and edges are lighter (i.e., sampled less) than the center region. This could result in a loss of information around the corners and edges.

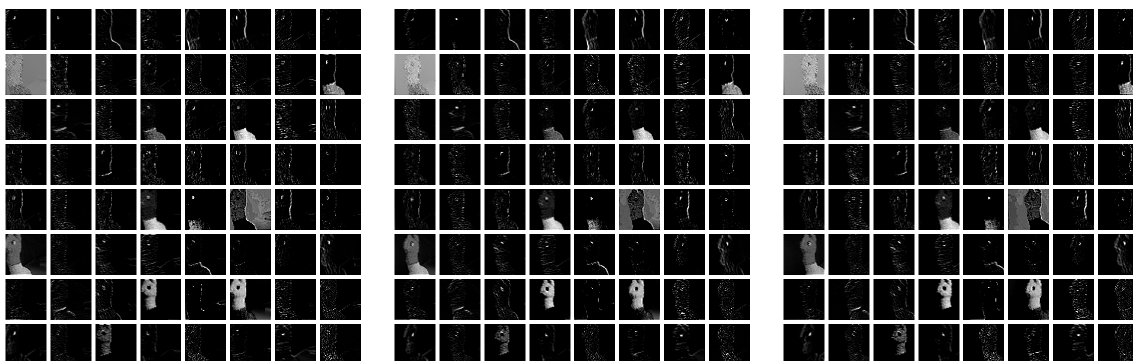


One way to handle this is to zero-pad the perimeter of the image. While other-than-zero padding[5] can be considered in specific circumstances, zero padding is more generic, in that, padding with zeros would not induce any artificial activations around the corner, causing the network to learn spurious patterns.

As a final note: since the number of padding rows (or columns) is more of a hyperparameter, one could pad such that the resultant output feature map is the same size as the input image.[6] With p rows and p columns of padding, the resultant image is $n + 2p \times n + 2p$, meaning the resultant feature map using filter of width k is $((n+2p)-k)+1 \times ((n+2p)-k)+1$. Now, all we need to do is to set p such that $((n + 2p) - k) + 1 = n$.

Translation (equi)variance. Assuming *right* padding, one of the nice(?[7]) properties of a convolution is that it is translation equivariant, i.e., if the image is translated, then the resultant feature map is equivalently translated as well. In other words, the features of images don't depend on their location in the images. Simply put, $\text{translate}(\text{conv}(\text{image}))$ is the same as $\text{conv}(\text{translate}(\text{image}))$. Such equivariance is because the filter being shared across patches of an image, unlike in fully-connected nets.

Below is a grid of 64 feature maps extracted the first convolution layer in AlexNet (Krizhevsky et al., 2012); Berry's position is shifted from left (in the left image) to the right (in the right image):



3 Receptive fields

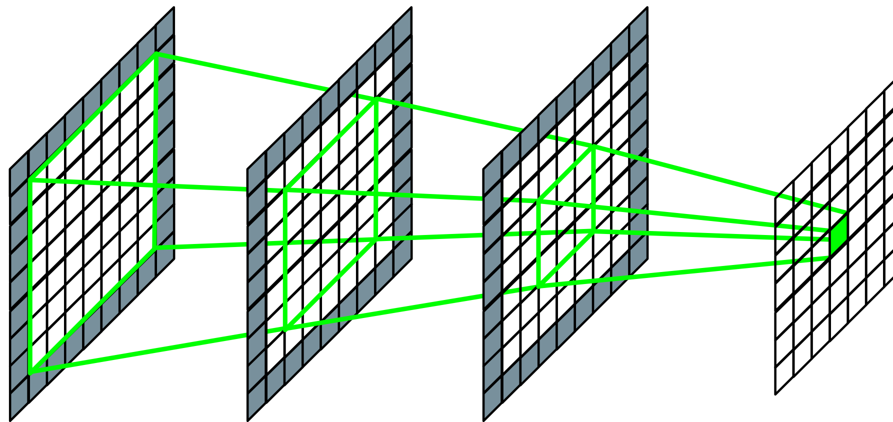
A useful question to ask is: how many convolutional layers do we need to be able to have at least one neuron in the feature map that captures the global structure of the image as a whole.

[5]Clever choices here include: use the nearest neighbor value to the border pixels, or some form of circular padding, where you copy values from the left border to the right pad pixels, etc.

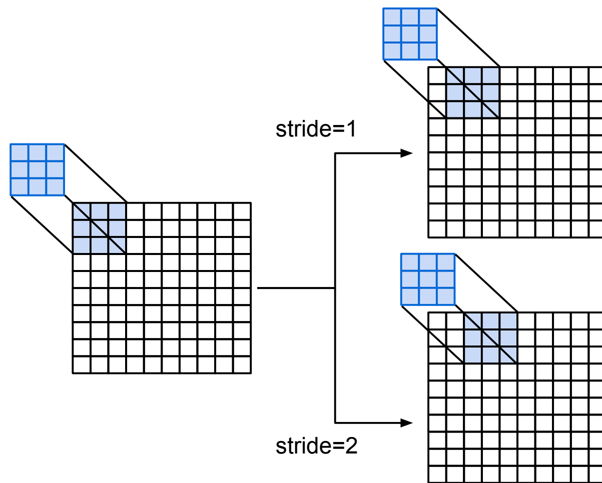
[6]In PyTorch, this is achieved by setting `padding="same"`.

[7]Equivariance is desirable sometimes (e.g., from the perspective of extracting features) and not so much, other times (e.g., learning to classify based on the position of your rook in chess). As a self-exercise, try to convolve a $5 \times 5 \times 1$ with 1.0 in the top-left corner vs. 1.0 in the bottom-right corner (0.0 elsewhere), with a 3×3 filter with a 1.0 in the bottom-right corner (and 0.0 elsewhere).

A width-k filter is such that each element in the output depends on a $k \times k$ *receptive field* in the input. As an example, consider a 7×7 image, with a 3×3 kernel:



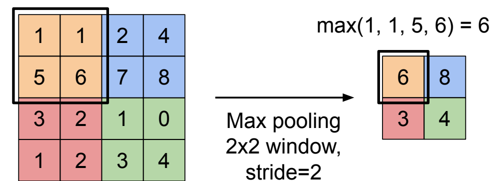
From above, we need 3 convolutional layers with 3×3 kernels to "see" the whole 7×7 image. Since each convolution adds $k - 1$ to the receptive field width, with l layers, the receptive field width is $1 + l(k - 1)$. Hence, for 1024×1024, high-resolution images, we need over 500 convolutional layers! So, can we do better?



Increasing the stride. Let us revisit the notion of *how many* pixels to move per timestep (or, the stride). Say, if we use a stride of 2 on a $7 \times 7 \times 3$ image with a 3×3 filter, the output feature map is 3×3, which would have otherwise been 5×5. Hence, increasing the stride effectively increases the receptive field width.

In general, an input image of size $w_{\text{in}} \times w_{\text{in}} \times c_{\text{in}}$, padded with p rows and p columns, convolved with a $k \times k$ filter using a stride s, results in an output of $w_{\text{out}} \times w_{\text{out}}$, where $w_{\text{out}} = ((w_{\text{in}} + 2p - k)/s) + 1$. You should work out why this is true; follow the same inductive thinking as with $s = 1$.

Pooling layer. Another way of spatially downsampling, i.e., increasing the receptive field, without adding convolution layers is pool across image patches. We can think of this as being similar to our plot-then-compute-dot-product, but now, we plop-and-pool! More specifically, we move a fixed-size window (also known as the pooling window) across the image, according to some stride and perform a pooling operation (e.g., max value of all pixels, average of all pixels, etc.). Pooling layers downsample the input without any learnable parameters (i.e., a pooling operation, such as max or average, has no learnable weights).



Pooling offers additional benefits beyond *just* downsampling. With max pooling, we achieve invariance to small shifts (since we are only going to take the max in a window) that may have been introduced from the camera vibrations (say). With average pooling, we are essentially smoothing the image to achieve a better signal-to-noise ratio. While other pooling variants are sometimes used (e.g., min pooling), max pooling is often the default choice, since it preserves

large activations (say, corresponding to edges or corners), which are often more importation in image recognition tasks.

4 Conclusion

In this lecture, with the goal of realizing architectures with inductive bias towards images, we saw convolutional neural nets, which often stack convolution, pooling (and fully-connected) layers. The generic form of a convolutional network is as follows:

```
input -> [[conv+relu]*N -> pool]*M -> [linear+relu]*K -> linear
```

One of the seminal works in computer vision was the development of AlexNet (Krizhevsky et al., 2012), which was essentially a standard convolutional network.[8] It is hard to overstate how influential this paper has been:

Paper	Citation count (04/24/2025)
Darwin, "The Origin of The Species by Means of Natural Selection" (1859)	$65,778$
Shannon, "A Mathematical Theory of Communication" (1948)	$66,335$
Watson and Crick, "A Structure for Deoxyribose Nucleic Acid" (1953)	$19,909$
The ATLAS Collaboration, "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC" (2012)	$12,071$
Krizhevsky, Sutskever, and Hinton, "ImageNet Classification with Deep Convolutional Neural Networks" (2012) [AlexNet]	$142,510$

While the recent trend in computer vision has been to use attention (in Transformers; which will be covered in the next lecture) or diffusion (covered in the lecture after Transformers), convolution layers are used, even today, in these all-purpose language models. For instance, the Mamba model (Gu and Dao, 2023) (think: faster ChatGPT) captures the local context, through a 1D convolution operating over a fixed window of 4 tokens, which is then passed to more advanced layers.

[8] `227x227 input -> [[conv+relu]*1 -> pool]*2 -> [[conv+relu]*3 -> pool] -> [linear+relu]*2 -> linear.`

A Notation

$x^{(l)}$	The input to the l-th layer of the neural network (and not the l-th example)
d	The hidden dimension; for e.g., $x^{(0)} \in \mathbb{R}^d$ means the input to layer-0 is d-dimensional
$W^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$	The weight matrix associated with layer-l; W_{ij} connects neuron-j in layer $l-1$ to neuron-i in layer-l
$b^{(l)} \in \mathbb{R}^{d_l}$	The bias associated with layer-l; one per neuron in layer-l
$\text{linear}_{d_l}(x^{(l-1)})$	Applies a linear transformation on $x^{(l-1)}$, i.e., computes $W^{(l)}x^{(l-1)} + b^{(l)}$
$g(\cdot)$	Element-wise nonlinearity (e.g., ReLU)
$n \times n \times c_{\text{in}}$ or $w_{\text{in}} \times w_{\text{in}} \times c_{\text{in}}$	The input image with width n (or w_{in}), height n (or w_{in}), and number of channels c_{in} (e.g., 3 for RGB)
k or $k \times k$	The convolution filter (or pooling window) of size $k \times k$
p	The number of padding rows (or columns)

References

A. Gu and T. Dao. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. *arXiv preprint arXiv:2312.00752*, 2023.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10590-1.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into deep learning*. Cambridge University Press, 2023. See Chapter 7: https://d2l.ai/chapter_convolutional-neural-networks/index.html.

(Last compiled: 5/6/2025, 2.53am ET.)