Image Processing and Recognition

Dr. Călin-Adrian POPA

Lecture 6

November 8th, 2022

3.5 Pooling

- often, as we process images, we want to gradually reduce the spatial resolution of our hidden representations, aggregating information so that the higher up we go in the network, the larger the receptive field (in the input) to which each hidden node is sensitive
- often, our ultimate task asks some global question about the image, e.g., *does it contain a cat*?
- so, typically, the units of our final layer should be sensitive to the entire input
- by gradually aggregating information, yielding coarser and coarser maps, we accomplish this goal of ultimately learning a global representation, while keeping all of the advantages of convolutional layers at the intermediate layers of processing

- moreover, when detecting lower-level features, such as edges, we often want our representations to be somewhat invariant to translation
- for instance, if we take the image $\mathbf{X}$, with a sharp delineation between black and white, and shift the whole image by one pixel to the right, i.e., $[\mathbf{Z}]_{i,j} = [\mathbf{X}]_{i,j+1}$, then the output for the new image $\mathbf{Z}$ might be vastly different
- the edge will have shifted by one pixel; in reality, objects hardly ever occur exactly at the same place
- in fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift everything by a pixel or so (high-end cameras are loaded with special features to address this problem)

3.5 Pooling

- this section introduces *pooling layers*, which serve the dual purposes of mitigating the sensitivity of convolutional layers to location and of spatially downsampling representations
- like convolutional layers, *pooling* operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*)
- however, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer contains no parameters (there is no *kernel*)

- instead, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window
- these operations are called *maximum pooling* (*max-pooling*, for short) and *average pooling*, respectively
- in both cases, as with the cross-correlation operator, we can think of the pooling window as starting from the upper-left of the input tensor, and sliding across the input tensor from left to right and top to bottom
- at each location that the pooling window hits, it computes the maximum or average value of the input subtensor in the window, depending on whether max or average pooling is employed

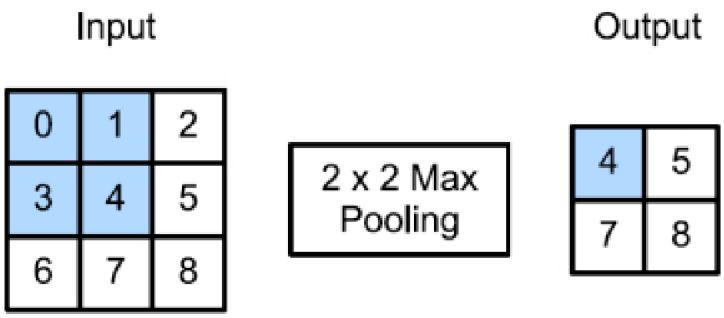

Figure 8: Maximum pooling with a pooling window shape of 2×2. The shaded portions are the first output element, as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

- the output tensor in Figure 8 has a height of 2 and a width of 2
- the four elements are derived from the maximum value in each pooling window:

$$\max(0, 1, 3, 4) = 4,$$
$$\max(1, 2, 4, 5) = 5,$$
$$\max(3, 4, 6, 7) = 7,$$
$$\max(4, 5, 7, 8) = 8.$$

- a pooling layer with a pooling window shape of $p \times q$ is called a $p \times q$ *pooling layer*
- the pooling operation is called $p \times q$ *pooling*

- let us return to the object edge detection example mentioned at the beginning of this section
- now, we will use the output of the convolutional layer as the input for 2×2 maximum pooling
- set the convolutional layer input as $\mathbf{X}$ and the pooling layer output as $\mathbf{Y}$
- whether or not the values of $[\mathbf{X}]_{i,j}$ and $[\mathbf{X}]_{i,j+1}$ are different, or $[\mathbf{X}]_{i,j+1}$ and $[\mathbf{X}]_{i,j+2}$ are different, the pooling layer always outputs $[\mathbf{Y}]_{i,j} = 1$
- that is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height or width

- as with convolutional layers, pooling layers can also change the output shape
- and, as before, we can alter the operation to achieve a desired output shape by padding the input and adjusting the stride
- when processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels, as in a convolutional layer
- this means that the number of output channels for the pooling layer is the same as the number of input channels

3.6 Convolutional neural networks (LeNet)

- we now have all the ingredients required to assemble a fully-functional CNN
- in this section, we will introduce *LeNet*, among the first published CNNs to capture wide attention for its performance on computer vision tasks
- the model was introduced by (and named for) Yann LeCun, then a researcher at AT&T Bell Labs, for the purpose of recognizing handwritten digits in images
- this work represented the culmination of a decade of research developing the technology
- in 1989, LeCun published the first study to successfully train CNNs via backpropagation

3.6 Convolutional neural networks (LeNet)

- at the time, LeNet achieved outstanding results, matching the performance of support vector machines, then a dominant approach in supervised learning
- LeNet was eventually adapted to recognize digits for processing deposits in ATM machines; to this day, some ATMs still run the code that Yann LeCun and his colleague Leon Bottou wrote in the 1990s
- at a high level, LeNet (LeNet-5) consists of two parts:
  - a convolutional encoder consisting of two convolutional layers
  - a dense block consisting of three fully-connected layers; the architecture is summarized in Figure 9

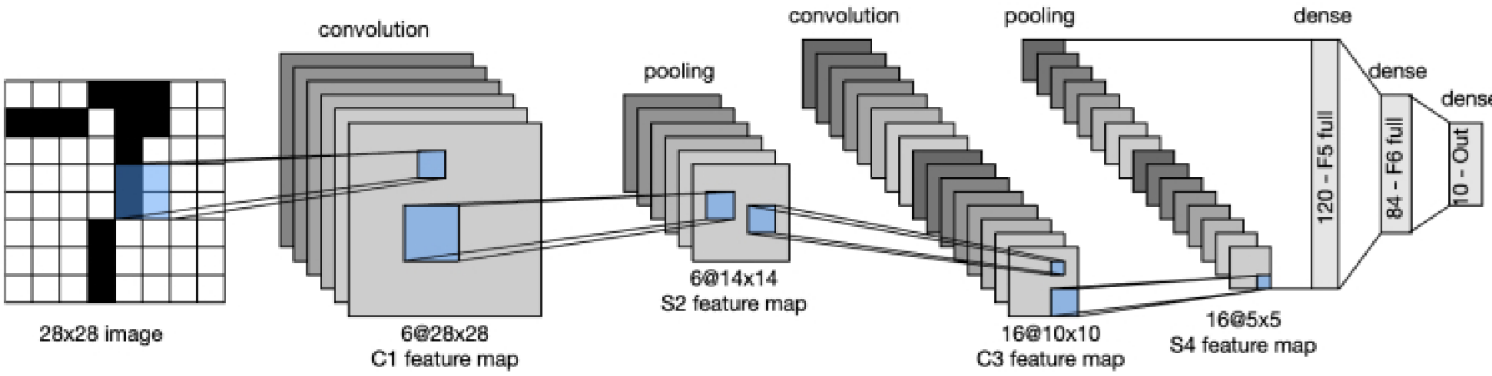

Figure 9: Data flow in LeNet. The input is a handwritten digit and the output is a probability over 10 possible outcomes.

3.6 Convolutional neural networks (LeNet)

- the basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation
- note that, while ReLUs and max-pooling work better, these discoveries had not yet been made in the 1990s
- each convolutional layer uses a 5×5 kernel and a sigmoid activation function
- these layers map spatially arranged inputs to a number of two-dimensional feature maps, typically increasing the number of channels
- the first convolutional layer has 6 output channels, while the second has 16
- each 2×2 pooling operation (stride 2) reduces dimensionality by a factor of 4 via spatial downsampling
- the convolutional block emits an output with shape given by (batch size, number of channels, height, width)

- in order to pass output from the convolutional block to the dense block, we must *flatten* each example in the mini-batch
- in other words, we take this four-dimensional input and transform it into the two-dimensional input expected by fully-connected layers: as a reminder, the two-dimensional representation that we desire uses the first dimension to index examples in the mini-batch and the second to give the flat vector representation of each example
- LeNet's dense block has three fully-connected layers, with 120, 84, and 10 outputs, respectively
- because we are still performing classification, the 10-dimensional output layer corresponds to the number of possible output classes
- by passing a single-channel (black and white) 28×28 image through the network, the model is depicted in Figure 10
- we took a small liberty with the original model, removing the Gaussian activation in the final layer; other than that, this network matches the original LeNet-5 architecture

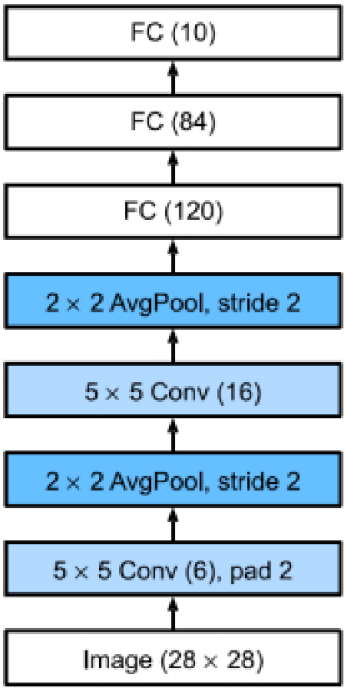

Figure 10: Compressed notation for LeNet-5.

- note that the height and width of the representation at each layer throughout the convolutional block is reduced (compared with the previous layer)
- the first convolutional layer uses 2 pixels of padding to compensate for the reduction in height and width that would otherwise result from using a 5×5 kernel
- in contrast, the second convolutional layer doesn't use padding, and thus the height and width are both reduced by 4 pixels
- as we go up the stack of layers, the number of channels increases layer-over-layer from 1 in the input to 6 after the first convolutional layer and 16 after the second convolutional layer
- however, each pooling layer halves the height and width
- finally, each fully-connected layer reduces dimensionality, finally emitting an output whose dimension matches the number of classes

3.7 Deep convolutional neural networks (AlexNet)

- now that we understand the basics of wiring together CNNs, we will take a tour of modern CNN architectures
- each next section will correspond to a significant CNN architecture that was briefly a dominant architecture and were winners (AlexNet, ResNet) or runners-up (VGG) in the ImageNet competition, which has served as a barometer of progress on supervised learning in computer vision since 2010
- these models include AlexNet, the first large-scale network deployed to beat conventional computer vision methods on a large-scale vision challenge; the VGG network, which makes use of a number of repeating blocks of elements; and residual networks (ResNet), which remain the most popular off-the-shelf architecture in computer vision

3.7 Deep convolutional neural networks (AlexNet)

- while the idea of *deep neural networks* is quite simple (stack together a bunch of layers), performance can vary wildly across architectures and hyperparameter choices
- the neural networks described next are the product of intuition, a few mathematical insights, and a whole lot of trial and error
- we present these models in chronological order, partly to convey a sense of the history, so that we can form our own intuitions about where the field is heading, and perhaps develop our own architectures
- for instance, batch normalization and residual connections, described next, have offered two popular ideas for training and designing deep models

3.7 Deep convolutional neural networks (AlexNet)

- although CNNs were well known in the computer vision and machine learning communities following the introduction of LeNet, they did not immediately dominate the field
- although LeNet achieved good results on early small datasets, the performance and feasibility of training CNNs on larger, more realistic datasets had yet to be established
- in fact, for much of the intervening time between the early 1990s and the breakthrough results of 2012, neural networks were often surpassed by other machine learning methods, such as support vector machines

- for computer vision, this comparison is perhaps not fair
- that is, although the inputs to convolutional networks consist of raw or lightly-processed (e.g., by centering) pixel values, practitioners would never feed raw pixels into traditional models
- instead, typical computer vision pipelines consisted of *manually* engineering *feature extraction* pipelines
- rather than *learn the features*, the features were *crafted*
- most of the progress came from having more clever ideas for features, and the learning algorithm was often very simple

3.7 Deep convolutional neural networks (AlexNet)

- although some neural network accelerators were available in the 1990s, they were not yet sufficiently powerful to make deep multichannel, multilayer CNNs with a large number of parameters
- moreover, datasets were still relatively small
- added to these obstacles, key tricks for training neural networks including parameter initialization heuristics, clever variants of stochastic gradient descent, non-squashing activation functions, and effective regularization techniques were still missing
- thus, rather than training *end-to-end* (pixel to classification) systems, classical pipelines looked more like this:
  1. Obtain an interesting dataset. In early days, these datasets required expensive sensors (at the time, 1 megapixel images were state-of-the-art).
  2. Preprocess the dataset with hand-crafted features based on some knowledge of optics, geometry, other analytic tools, and occasionally on the discoveries of lucky graduate students.
  3. Feed the data through a standard set of feature extractors such as the SIFT (scale-invariant feature transform), the SURF (speeded up robust features), or any number of other hand-tuned pipelines.
  4. Dump the resulting representations into a classifier, likely a linear model or kernel method, to train a classifier.

3.7 Deep convolutional neural networks (AlexNet)

- another group of researchers, including Yann LeCun, Geoffrey Hinton, Yoshua Bengio, Andrew Ng, and Juergen Schmidhuber had different plans: they believed that features themselves should be learned
- moreover, they believed that, to be reasonably complex, the features should be *hierarchically* composed with multiple jointly learned layers, each with learnable parameters
- in the case of an image, the lowest layers might come to detect edges, colors, and textures
- indeed, Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton proposed a new variant of a CNN, *AlexNet*, that achieved excellent performance in the 2012 ImageNet challenge
- AlexNet was named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper

- interestingly, in the lowest layers of the network, the model learned feature extractors that resembled some traditional filters; Figure 11 is reproduced from the AlexNet paper, and describes lower-level image descriptors

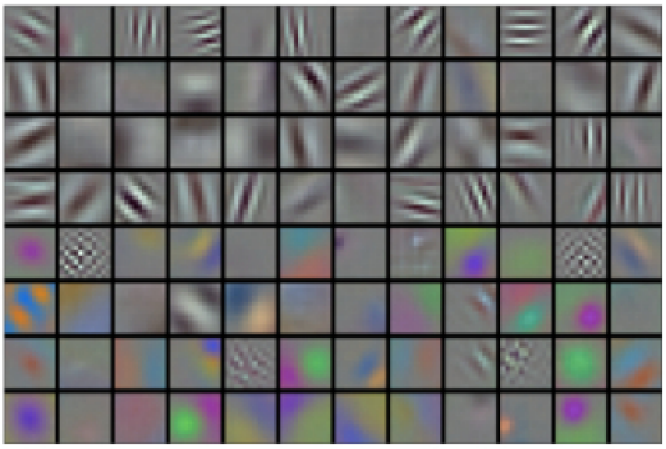

Figure 11: Image filters learned by the first layer of AlexNet.

- higher layers in the network might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and so on
- even higher layers might represent whole objects like people, airplanes, dogs, or frisbees
- ultimately, the final hidden state learns a compact representation of the image that summarizes its contents, such that data belonging to different categories can be easily separated
- while the ultimate breakthrough for many-layered CNNs came in 2012, a core group of researchers had dedicated themselves to this idea, attempting to learn *hierarchical representations* of visual data for many years
- the ultimate breakthrough in 2012 can be attributed to *two key factors*

3.7 Deep convolutional neural networks (AlexNet)

- deep models with many layers require large amounts of *data* in order to enter the regime where they significantly outperform traditional methods based on convex optimizations (e.g., linear and kernel methods)
- however, given the limited storage capacity of computers, the relative expense of sensors, and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets
- numerous papers addressed the UCI collection of datasets, many of which contained only hundreds or (a few) thousands of images captured in unnatural settings with low resolution

3.7 Deep convolutional neural networks (AlexNet)

- in 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, each from one of 1000 distinct categories of objects
- the researchers, led by Fei-Fei Li, who introduced this dataset, leveraged Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category
- this scale was unprecedented
- the associated competition, named the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered

3.7 Deep convolutional neural networks (AlexNet)

- the second factor is *hardware*
- deep learning models are massive consumers of compute cycles
- training can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations
- this is one of the main reasons why, in the 1990s and early 2000s, simple algorithms based on the more-efficiently optimized convex objectives were preferred

3.7 Deep convolutional neural networks (AlexNet)

- *graphical processing units* (GPUs) proved to be a game changer in making deep learning feasible
- these chips had long been developed for accelerating graphics processing to benefit computer games
- in particular, they were optimized for high throughput 4×4 matrix-vector products, which are needed for many computer graphics tasks
- fortunately, this math is strikingly similar to that required to calculate convolutional layers
- around that time, NVIDIA and ATI had begun optimizing GPUs for general computing operations, going as far as to market them as *general-purpose GPUs* (GPGPU)

3.7 Deep convolutional neural networks (AlexNet)

- back to 2012; a major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep CNN that could run on GPU hardware
- they realized that the computational bottlenecks in CNNs, convolutions and matrix multiplications, are all operations that could be parallelized in hardware
- using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions
- the code `cuda-convnet` was good enough that, for several years, it was the industry standard, and powered the first couple years of the deep learning boom

3.7 Deep convolutional neural networks (AlexNet)

- AlexNet, which employed an 8-layer CNN, won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin
- this network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision
- the architectures of AlexNet and LeNet are very similar, as Figure 12 illustrates
- note that we provide a slightly streamlined version of AlexNet, removing some of the design tricks that were needed in 2012 to make the model fit on two small GPUs

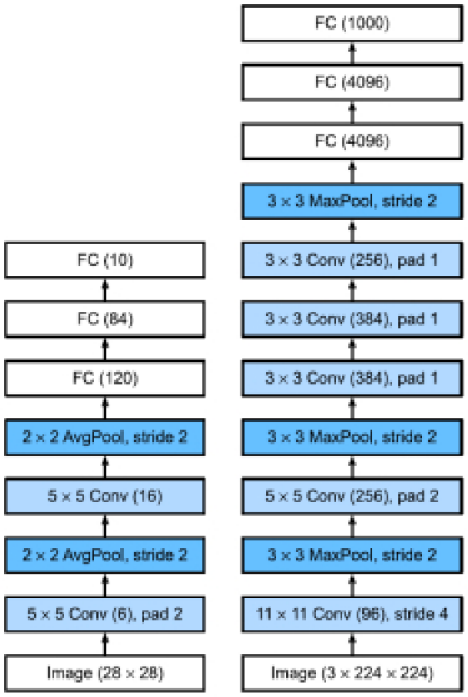

Figure 12: From LeNet (left) to AlexNet (right).

3.7 Deep convolutional neural networks (AlexNet)

- the design philosophies of AlexNet and LeNet are very similar, but there are also significant differences
- first, AlexNet is much deeper than the comparatively small LeNet5
- AlexNet consists of eight layers: five convolutional layers, two fully-connected hidden layers, and one fully-connected output layer
- second, AlexNet used the ReLU instead of the sigmoid as its activation function

3.7 Deep convolutional neural networks (AlexNet)

- in AlexNet's first layer, the convolution window shape is 11×11
- since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet data tend to occupy more pixels
- consequently, a larger convolution window is needed to capture the object
- the convolution window shape in the second layer is reduced to 5×5, followed by 3×3
- in addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2
- moreover, AlexNet has ten times more convolution channels than LeNet

3.7 Deep convolutional neural networks (AlexNet)

- after the last convolutional layer there are two fully-connected layers with 4096 outputs
- these two huge fully-connected layers produce model parameters of nearly 1 GB
- due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that each of their two GPUs could be responsible for storing and computing only its half of the model
- fortunately, GPU memory is comparatively abundant now, so we rarely need to break up models across GPUs these days (our version of the AlexNet model deviates from the original paper in this aspect)

- besides, AlexNet changed the sigmoid activation function to a simpler ReLU activation function
- on one hand, the computation of the ReLU activation function is simpler; for example, it does not have the exponentiation operation found in the sigmoid activation function
- on the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods
- this is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that backpropagation cannot continue to update some of the model parameters
- in contrast, the gradient of the ReLU activation function in the positive interval is always 1
- therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained

3.7 Deep convolutional neural networks (AlexNet)

- AlexNet controls the model complexity of the fully-connected layer by dropout (Section 2.5), while LeNet only uses weight decay
- to augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes
- this makes the model more robust, and the larger sample size effectively reduces overfitting
- we will discuss data augmentation in greater detail in Chapter 6

3.8 Networks using blocks (VGG)

- while AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks
- in the following sections, we will introduce several heuristic concepts commonly used to design deep networks
- progress in this field mirrors that in chip design, where engineers went from placing transistors to logical elements to logic blocks
- similarly, the design of neural network architectures had grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers
- the idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, which gave the name of their *VGG* network
- it is easy to implement these repeated structures in code with any modern deep learning framework, by using loops and subroutines

3.8 Networks using blocks (VGG)

- the basic building block of classic CNNs is a sequence of the following:
  - a convolutional layer with padding to maintain the resolution
  - a nonlinearity such as a ReLU
  - a pooling layer such as a maximum pooling layer
- one VGG block consists of a sequence of convolutional layers, followed by a maximum pooling layer for spatial downsampling
- in the original VGG paper, the authors employed convolutions with 3×3 kernels with padding of 1 (keeping height and width) and 2×2 maximum pooling with stride of 2 (halving the resolution after each block)
- like AlexNet and LeNet, the VGG Network can be partitioned into two parts: the first consisting mostly of convolutional and pooling layers, and the second consisting of fully-connected layers; this is depicted in Figure 13

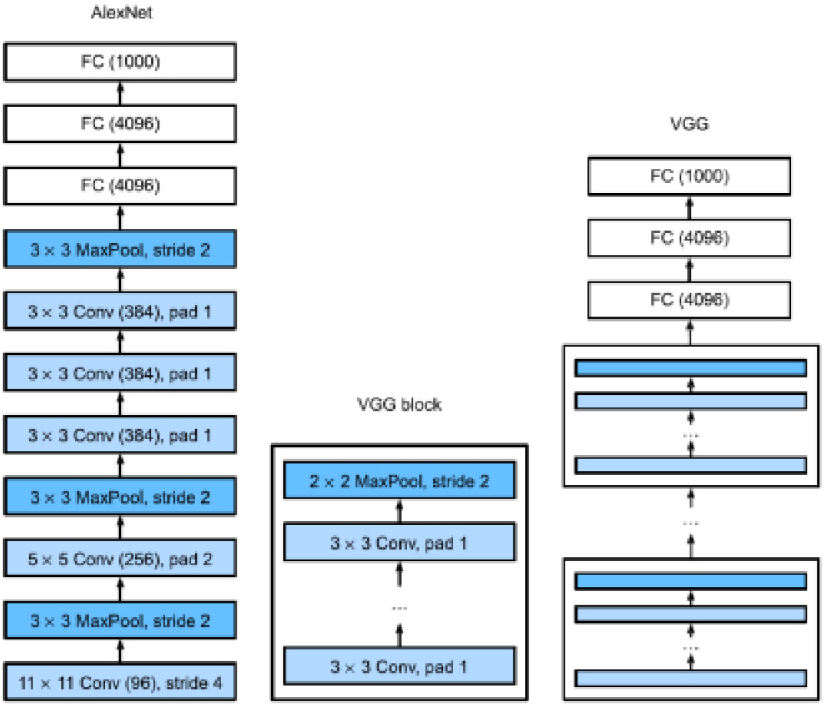

Figure 13: From AlexNet to VGG, which is designed from building blocks.

3.8 Networks using blocks (VGG)

- the convolutional part of the network connects several VGG blocks from Figure 13, in succession
- the fully-connected part of the VGG network is identical to that covered in AlexNet
- the original VGG network had 5 convolutional blocks, among which the first two have one convolutional layer each, and the latter three contain two convolutional layers each
- the first block has 64 output channels, and each subsequent block doubles the number of output channels, until that number reaches 512
- since this network uses 8 convolutional layers and 3 fully-connected layers, it is often called *VGG-11*

3.9 Batch normalization

- training deep neural networks is difficult, and getting them to converge in a reasonable amount of time can be tricky
- in this section, we describe *batch normalization*, a popular and effective technique that consistently accelerates the convergence of deep networks
- together with residual blocks – covered later, in Section 3.10 – batch normalization has made it possible to routinely train networks with over 100 layers
- to motivate batch normalization, let us review a few practical challenges that arise when training machine learning models, and neural networks in particular

3.9 Batch normalization

- first, choices regarding data preprocessing often make an enormous difference in the final results
- our first step when working with data is to standardize our input features to each have a mean of zero and variance of one
- intuitively, this standardization plays nicely with our optimizers, because it puts the parameters *a priori* at a similar scale
- second, for a typical MLP or CNN, as we train, the variables (e.g., affine transformation outputs in MLP) in intermediate layers may take values with widely varying magnitudes: both along the layers from the input to the output, across units in the same layer, and over time due to our updates to the model parameters

3.9 Batch normalization

- the inventors of batch normalization postulated informally that this drift in the distribution of such variables could prevent the convergence of the network
- intuitively, we might conjecture that, if one layer has variable values that are 100 times that of another layer, this might need compensatory adjustments in the learning rates
- third, deeper networks are complex and easily capable of overfitting; this means that regularization becomes more critical

3.9 Batch normalization

- batch normalization is applied to individual layers (optionally, to all of them) and works as follows: in each training iteration, we first normalize the inputs (of batch normalization) by subtracting their mean and dividing by their standard deviation, where both are estimated based on the statistics of the current mini-batch
- next, we apply a scale coefficient and a scale offset
- it is precisely due to this *normalization* based on *batch* statistics that *batch normalization* derives its name

3.9 Batch normalization

- note that, if we tried to apply batch normalization with mini-batches of size 1, we would not be able to learn anything
- that is because, after subtracting the means, each hidden unit would take value 0
- as we might guess, since we are devoting a whole section to batch normalization, with large enough mini-batches, the approach proves effective and stable
- one takeaway here is that, when applying batch normalization, the choice of batch size may be even more significant than without batch normalization

- formally, denoting by $\mathbf{x} \in \mathcal{B}$ an input to batch normalization (BN) that is from a mini-batch $\mathcal{B}$, batch normalization transforms $\mathbf{x}$ according to the following expression:

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}_{\mathcal{B}}}{\hat{\sigma}_{\mathcal{B}}} + \beta. \tag{4}$$

- in (4), $\hat{\mu}_{\mathcal{B}}$ is the sample mean and $\hat{\sigma}_{\mathcal{B}}$ is the sample standard deviation of the mini-batch $\mathcal{B}$
- after applying standardization, the resulting mini-batch has zero mean and unit variance
- because the choice of unit variance (vs. some other number) is an arbitrary choice, we commonly include element-wise *scale parameter* γ and *shift parameter* β, which have the same shape as $\mathbf{x}$
- note that γ and β are parameters that need to be learned jointly with the other model parameters

- consequently, the variable magnitudes for intermediate layers cannot diverge during training, because batch normalization actively centers and rescales them back to a given mean and size (via $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$)
- one piece of practitioner's intuition or wisdom is that batch normalization seems to allow for more aggressive learning rates
- formally, we calculate $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$ in (4) as follows:

$$\begin{aligned} \hat{\mu}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \\ \hat{\sigma}^2_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\mu}_{\mathcal{B}})^2 + \epsilon. \end{aligned}$$

- note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never attempt division by zero, even in cases where the empirical variance estimate might vanish
- the estimates $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$ counteract the scaling issue by using noisy estimates of mean and variance
- we might think that this noisiness should be a problem; as it turns out, this is actually beneficial

3.9 Batch normalization

- fixing a trained model, we might think that we would prefer using the entire dataset to estimate the mean and variance
- once training is complete, why would we want the same image to be classified differently, depending on the batch in which it happens to reside?
- during training, such exact calculation is infeasible, because the intermediate variables for all data examples change every time we update our model
- however, once the model is trained, we can calculate the means and variances of each layer's variables based on the entire dataset
- indeed, this is standard practice for models employing batch normalization, and thus batch normalization layers function differently in *training mode* (normalizing by mini-batch statistics) and in *prediction mode* (normalizing by dataset statistics)

3.9 Batch normalization

- we are now ready to take a look at how batch normalization works in practice
- batch normalization implementations for fully-connected layers and convolutional layers are slightly different; we discuss both cases
- recall that one key difference between batch normalization and other layers is that, because batch normalization operates on a full mini-batch at a time, we cannot just ignore the batch dimension, as we did before, when introducing other layers

3.9 Batch normalization

- when applying batch normalization to fully-connected layers, the original paper inserts batch normalization after the affine transformation and before the nonlinear activation function (later applications may insert batch normalization right after activation functions)
- denoting the input to the fully-connected layer by $\mathbf{x}$, the affine transformation by $\mathbf{Wx} + \mathbf{b}$ (with the weight parameter $\mathbf{W}$ and the bias parameter $\mathbf{b}$), and the activation function by ϕ, we can express the computation of a batch-normalization-enabled, fully-connected layer output $\mathbf{h}$ as follows:

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{Wx} + \mathbf{b})).$$

- recall that mean and variance are computed on the *same* mini-batch on which the transformation is applied

3.9 Batch normalization

- similarly, with convolutional layers, we can apply batch normalization after the convolution and before the nonlinear activation function
- when the convolution has multiple output channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has its own scale and shift parameters, both of which are scalars
- assume that our mini-batches contain m examples, and that, for each channel, the output of the convolution has height p and width q
- for convolutional layers, we carry out each batch normalization over the $m \cdot p \cdot q$ elements per output channel, simultaneously
- thus, we collect the values over all spatial locations when computing the mean and variance, and consequently apply the same mean and variance within a given channel to normalize the value at each spatial location

3.9 Batch normalization

- as we mentioned earlier, batch normalization typically behaves differently in training mode and prediction mode
- first, the noise in the sample mean and the sample variance arising from estimating each on mini-batches are no longer desirable, once we have trained the model
- second, we might not have the luxury of computing per-batch normalization statistics
- for example, we might need to apply our model to make one prediction at a time
- typically, after training, we use the entire dataset to compute stable estimates of the variable statistics, and then fix them at prediction time
- consequently, batch normalization behaves differently during training and at test time; recall that dropout also exhibits this characteristic

3.10 Residual networks (ResNet)

- as we design increasingly deeper networks, it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network
- even more important is the ability to design networks where adding layers makes networks strictly more expressive, rather than just different
- this is the question that He et al. considered when working on very deep computer vision models
- at the heart of their proposed *residual network* (*ResNet*) is the idea that every additional layer should more easily contain the *identity function* as one of its elements
- these considerations are rather profound, but they led to a surprisingly simple solution, the *residual block*
- with it, ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015
- the design had a profound influence on how to build deep neural networks

3.10 Residual networks (ResNet)

- let us focus on a local part of a neural network, as depicted in Figure 14
- denote the input by $\mathbf{x}$, and assume that the desired underlying mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function on the top
- on the left of Figure 14, the portion within the dotted-line box must directly learn the mapping $f(\mathbf{x})$
- on the right, the portion within the dotted-line box needs to learn the *residual mapping* $f(\mathbf{x}) - \mathbf{x}$, which is how the residual block derives its name
- if the identity mapping $f(\mathbf{x}) = \mathbf{x}$ is the desired underlying mapping, the residual mapping is easier to learn: we only need to push the weights and biases of the upper weight layer (e.g., fully-connected layer and convolutional layer) within the dotted-line box to zero
- the right figure in Figure 14 illustrates the *residual block* of ResNet, where the solid line carrying the layer input $\mathbf{x}$ to the addition operator is called a *residual connection* (or *shortcut connection*)
- with residual blocks, inputs can forward propagate faster through the residual connections across layers

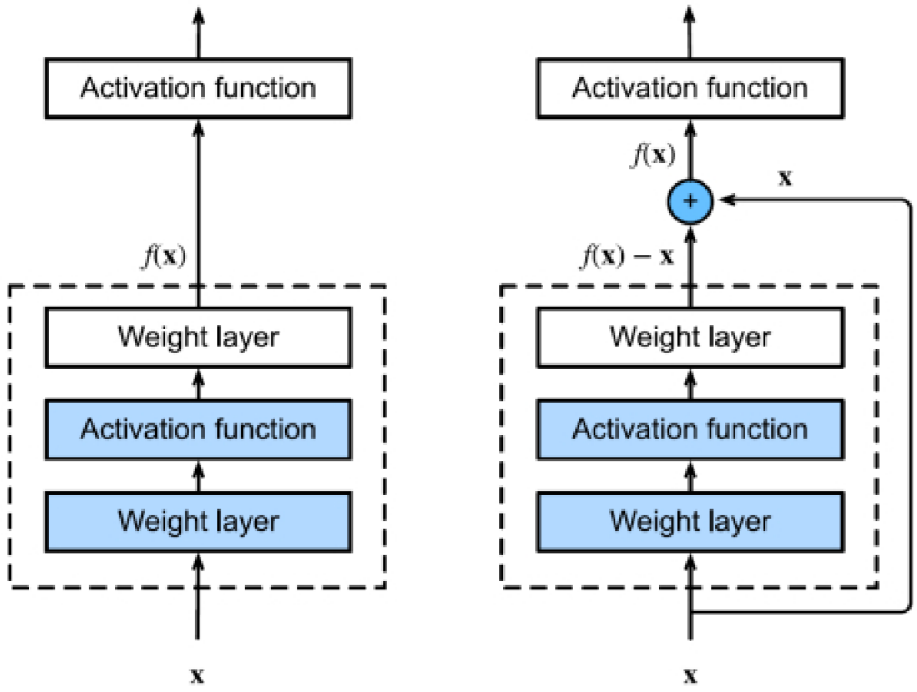

Figure 14: A regular block (left) and a residual block (right).

3.10 Residual networks (ResNet)

- ResNet follows VGG's full 3×3 convolutional layer design
- the residual block has two 3×3 convolutional layers with the same number of output channels
- each convolutional layer is followed by a batch normalization layer and a ReLU activation function
- then, we skip these two convolution operations and add the input directly before the final ReLU activation function
- this kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together
- if we want to change the number of channels or the resolution, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation
- thus, we have two types of blocks: one where we add the input to the output before applying the ReLU nonlinearity, and one where we adjust channels and resolution by means of a 1×1 convolution before adding; Figure 15 illustrates this

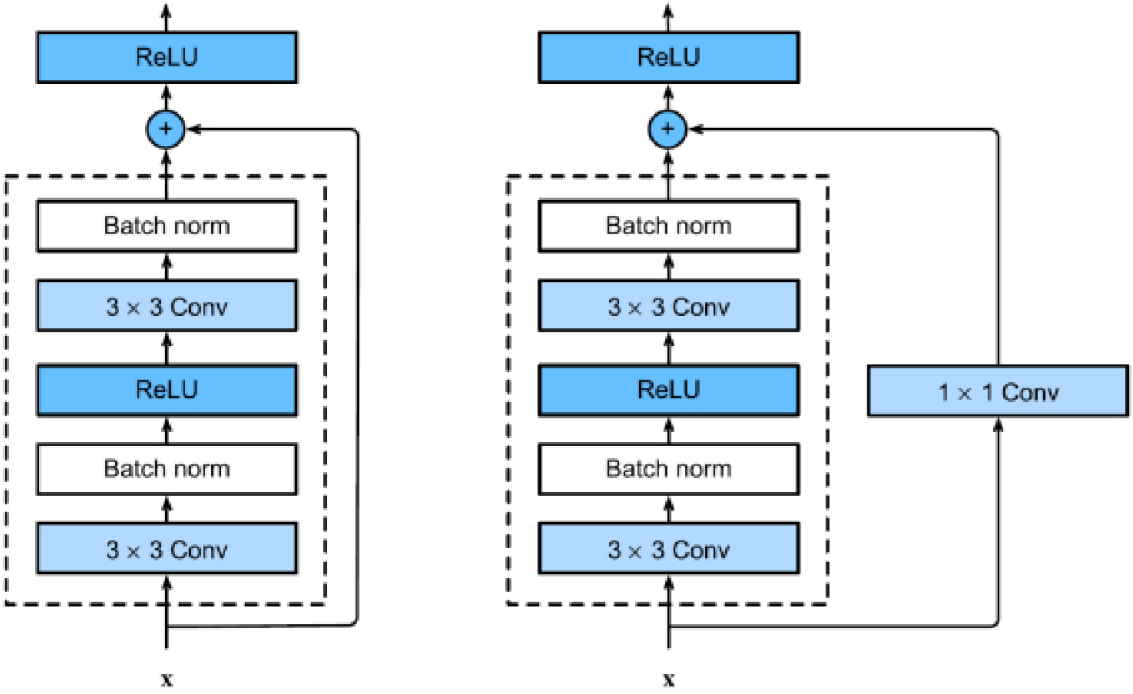

Figure 15: ResNet block with and without 1 × 1 convolution.

3.10 Residual networks (ResNet)

- the first two layers of ResNet are a 7×7 convolutional layer with 64 output channels and a stride of 2, which is followed by the 3×3 maximum pooling layer with a stride of 2
- a batch normalization layer is added after each convolutional layer in ResNet
- ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels
- the number of channels in the first module is the same as the number of input channels
- since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width
- in the first residual block, for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved

- finally, we add a global average pooling layer, followed by the fully-connected layer output
- there are 4 convolutional layers in each module (excluding the 1×1 convolutional layer)
- together with the first 7×7 convolutional layer and the final fully-connected layer, there are 18 layers in total
- therefore, this model is commonly known as *ResNet-18*
- by configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer *ResNet-152*
- the main architecture of ResNet is simple and easy to modify, which has resulted in the rapid and widespread use of ResNet; Figure 16 depicts the full ResNet-18

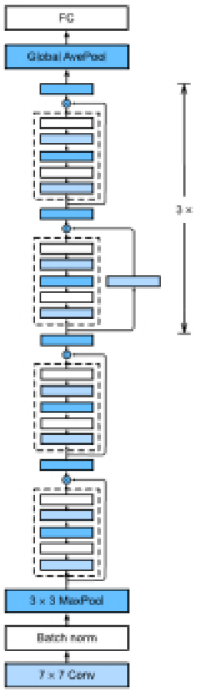

Figure 16: The ResNet-18 architecture.

Thank you!