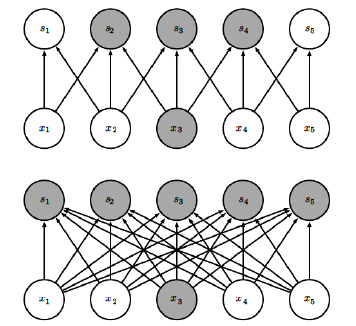
**Chapter 9**

**CONVOLUTIONAL NETWORKS**

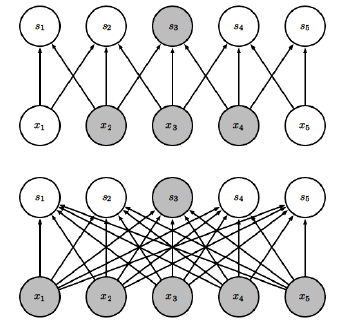
* Convolutional networks (LeCun, 1989), also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels.
* The name “convolutional neural network” indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation.
* Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.
  1. **The Convolution Operation**
* Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output x(t), the position of the spaceship at time t. Both x and t are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.
* Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship’s position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements.
* We can do this with a weighting function w(a), where a is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function providing a smoothed estimate of the position s of the spaceship:
* This operation is called **convolution**. The convolution operation is typically denoted with an asterisk:
* In our example, w needs to be a valid probability density function, or the output is not a weighted average. Also, w needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.
* In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the **input**and the second argument (in this example, the function w) as the **kernel**. The output is sometimes referred to as the **feature map**.
* Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. If we now assume that x and w are defined only on integer t, we can define the discrete convolution:
* In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately we usually assume that these functions are zero everywhere but the finite set of points for which we store the values.
* Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:
* Convolution is commutative, meaning we can equivalently write:
* The commutative property of convolution arises because we have flippedthe kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases.
* Instead, many neural network libraries implement a related function called the **cross-correlation**, which is the same as convolution but without flipping the kernel:
* Many machine learning libraries implement cross-correlation but call it convolution.
* Discrete convolution can be viewed as multiplication by a matrix. However, the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a **Toeplitz matrix**. In two dimensions, a **doubly block circulant matrix**corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image.
  1. **Motivation**
* Convolution leverages three important ideas that can help improve a machine learning system:

1. sparse interactions
2. parameter sharing
3. equivariant representations*.*

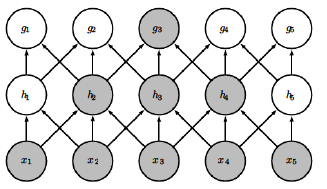
* Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means every output unit interacts with every input unit.
* Convolutional networks, however, typically have **sparse interactions**(also referred to as **sparse connectivity**or **sparse weights**). This is accomplished by making the kernel smaller than the input.
* For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.
* These improvements in efficiency are usually quite large. If there are m inputs and n outputs, then matrix multiplication requires mxn parameters and the algorithms used in practice have O(mxn) runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only k x n parameters and O(k x n) runtime.

****

**Figure 1:** *Sparse connectivity, viewed from below*

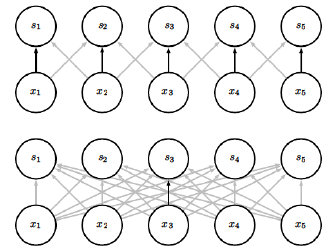


**Figure 2:** *Sparse connectivity, viewed from above*

**

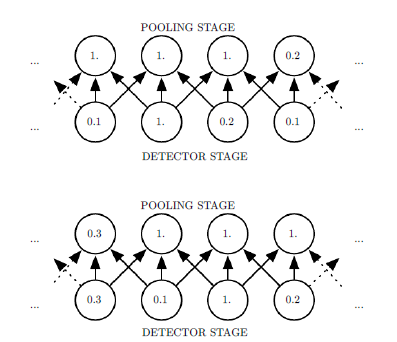
**Figure 3:** *The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers.*

* **Parameter sharing**refers to using the same parameter for more than one function in a model.
* In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has **tied weights**, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere.
* In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary).
* The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still O(kxn)—but it does further reduce the storage requirements of the model to k parameters.

****

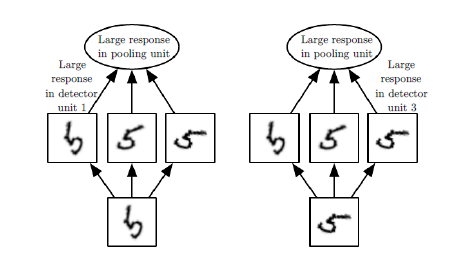
**Figure 4:** *Parameter sharing*

* In the case of convolution, the particular form of parameter sharing causes the layer to have a property called **equivariance**to translation.
* To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function f(x) is equivariant to a function g if f(g(x)) = g(f(x)).
* In the case of convolution, if we let g be any function that translates the input, i.e., shifts it, then the convolution function is equivariant to g.
* For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that I’= g(I ) is the image function with I’(x, y) = I(x − 1, y). This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I’, then applied the transformation g to the output.
* When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time.
* Similarly, with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.
* In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual’s face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.
  1. **Pooling**
* A typical layer of a convolutional network consists of three stages (see Fig. 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the *detector* stage. In the third stage, we use a *pooling* *function* to modify the output of the layer further.
* A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the **max pooling**(Zhou and Chellappa, 1988) operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.
* In all cases, pooling helps to make the representation become approximately **invariant**to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.

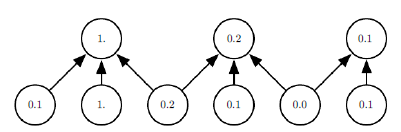
****

**Figure 5:** *Max pooling introduces invariance. (Top) A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. (Bottom) A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.*

* Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced k pixels apart rather than 1 pixel apart.
* This improves the computational efficiency of the network because the next layer has roughly k times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.
* For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of an offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size.



**Figure 6:** *Example of learned invariances: Here we show how a set of three learned filters and a max pooling unit can learn to become invariant to rotation. All three filters are intended to detect a hand-written 5. Each filter attempts to match a slightly different orientation of the 5. When a 5 appears in the input, the corresponding filter will match it and cause a large activation in a detector unit. The max pooling unit then has a large activation regardless of which pooling unit was activated. We show here how the network processes two different inputs, resulting in two different detector units being activated. The effect on the pooling unit is roughly the same either way. This principle is leveraged by maxout networks (Goodfellow et al., 2013a) and other convolutional networks. Max pooling over spatial positions is naturally invariant to translation; this multi-channel approach is only necessary for learning other transformations.*



**Figure 7:** *Pooling with downsampling*. *Here we use max-pooling with a pool width of three and a stride between pools of two. This reduces the representation size by a factor of two, which reduces the computational and statistical burden on the next layer. Note that the rightmost pooling region has a smaller size, but must be included if we do not want to ignore some of the detector units.*

* It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image.
  1. **Convolution and Pooling as an Infinitely Strong Prior**
* **prior probability distribution** This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.
* Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such as a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.
* An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data gives to those values.
* We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit.
* Of course, implementing a convolutional net as a fully connected net with an infinitely strong prior would be extremely computationally wasteful. But thinking of a convolutional net as a fully connected net with an infinitely strong prior can give us some insights into how convolutional nets work. One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error.
* Some convolutional network architectures (Szegedy *et al.*, 2014a) are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect.
* Another key insight from this view is that we should only compare convolutional models to other convolutional models in benchmarks of statistical learning performance. Models that do not use convolution would be able to learn even if we permuted all the pixels in the image. For many image datasets, there are separate benchmarks for models that are **permutation invariant**and must discover the concept of topology via learning, and models that have the knowledge of spatial relationships hard-coded into them by their designer.
  1. **Variants of the Basic Convolution Function**

In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates

of each channel. Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch.

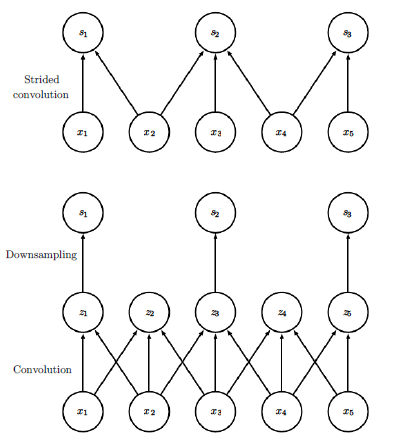
Because convolutional networks usually use multi-channel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel-flipping is used. These multi-channel operations are only commutative if each operation has the same number of output channels as input channels.

Assume we have a 4-D kernel tensor **K** with element Ki,j,k,l giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. Assume our input consists of observed data **V** with element Vi,j,k giving the value of the input unit within channel i at row j and column k. Assume our output consists of **Z** with the same format as **V**. If **Z** is produced by convolving **K** across **V** without flipping **K**, then

where the summation over l, m and n is over all values for which the tensor indexing operations inside the summation is valid. In linear algebra notation, we index into arrays using a 1 for the first entry. This necessitates the −1 in the above formula.

We may want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. If we want to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

We refer to s as the *stride* of this downsampled convolution. It is also possible to define a separate stride for each direction of motion.



**Figure 8:** *Convolution with a stride. In this example, we use a stride of two. (Top) Convolution with a stride length of two implemented in a single operation. (Bottom) Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling.*

One essential feature of any convolutional network implementation is the ability to implicitly zero-pad the input **V** in order to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called *valid* convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image has width m and the kernel has width k, the output will be of width m− k+ 1. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to 1 × 1, at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero-padding is added to keep the size of the output equal to the size of the input. MATLAB calls this *same* convolution. In this case, the network can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities available to the next layer. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other

extreme case, which MATLAB refers to as *full convolution*, in which enough zeroes are added for every pixel to be visited k times in each direction, resulting in an output image of width m+ k − 1. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between “valid” and “same” convolution.

In some cases, we do not actually want to use convolution, but rather locally connected layers (LeCun, 1986, 1989). In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified by a 6-D tensor **W**. The indices into **W** are respectively: i, the output channel, j, the output row, k, the output column, l, the input channel, m, the row offset within the input, and n, the column offset within the input. The linear part of a

locally connected layer is then given by

This is sometimes also called *unshared convolution*, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations.

Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space. For example, if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image.

It can also be useful to make versions of convolution or locally connected layers in which the connectivity is further restricted, for example to constrain that each output channel i be a function of only a subset of the input channels l. A common way to do this is to make the first m output channels connect to only the first n input channels, the second m output channels connect to only the second n input channels, and so on.

*Tiled convolution* (Gregor and LeCun, 2010a; Le *et al.*, 2010) offers a compromise between a convolutional layer and a locally connected layer. Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space. This means that immediately neighboring locations will have different filters, like in a locally connected layer, but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels, rather than the size of the entire output feature

map.

To define tiled convolution algebraically, let k be a 6-D tensor, where two of the dimensions correspond to different locations in the output map. Rather than having a separate index for each location in the output map, output locations cycle through a set of t different choices of kernel stack in each direction. If t is equal to the output width, this is the same as a locally connected layer.

where % is the modulo operation

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, one must be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to back-propagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed if we wish to reconstruct the visible units from the hidden units (Simard *et al.*, 1992).

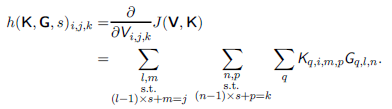
Transpose convolution is necessary to construct convolutional versions of those models. Like the kernel gradient operation, this input gradient operation can be implemented using a convolution in some cases, but in the general case requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero padding policy and stride of the forward propagation operation, as well as the size of the forward propagation’s output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

These three operations—convolution, backprop from output to weights, and backprop from output to inputs—are sufficient to compute all of the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution.

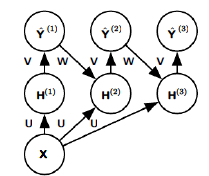
Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack **K** applied to multi-channel image **V** with stride s as defined by c(**K**,**V**, s) Suppose we want to minimize some loss function J(**V**,**K**). During forward propagation, we will need to use c itself to output **Z**, which is then propagated through the rest of the network and used to compute the cost function J. During back-propagation, we will receive a tensor **G** such that

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

If this layer is not the bottom layer of the network, we will need to compute the gradient with respect to **V** in order to back-propagate the error farther down. To do so, we can use a function



* 1. **Structured Outputs**
* Convolutional networks can be used to output a high-dimensional, structured object, rather than just predicting a class label for a classification task or a real value for a regression task. Typically, this object is just a tensor, emitted by a standard convolutional layer. For example, the model might emit a tensor **S**, where Si,j,k is the probability that pixel (j,k) of the input to the network belongs to class i. This allows the model to label every pixel in an image and draw precise masks that follow the outlines of individual objects.
* One strategy for pixel-wise labeling of images is to produce an initial guess of the image labels, then refine this initial guess using the interactions between neighboring pixels. Repeating this refinement step several times corresponds to using the same convolutions at each stage, sharing weights between the last layers of the deep net (Jain *et al.*, 2007). This makes the sequence of computations performed by the successive convolutional layers with weights shared across layers a particular kind of recurrent network (Pinheiro and Collobert, 2014, 2015). Below figure shows the architecture of such a recurrent convolutional network.

****

**Figure 9:** *An example of a recurrent convolutional network for pixel labeling. The input is an image tensor* ***X****, with axes corresponding to image rows, image columns, and channels (red, green, blue). The goal is to output a tensor of labels , with a probability distribution over labels for each pixel. Rather than outputting in a single shot, the recurrent network iteratively refines its estimate by using a previous estimate of as input for creating a new estimate. The same parameters are used for each updated estimate, and the estimate can be refined as many times as we wish. The tensor of convolution kernels* ***U*** *is used on each step to compute the hidden representation given the input image. The kernel tensor* ***V*** *is used to produce an estimate of the labels given the hidden values. On all but the first step, the kernels* ***W*** *are convolved over to provide input to the hidden layer. On the first-time step, this term is replaced by zero.*

* Once a prediction for each pixel is made, various methods can be used to further process these predictions in order to obtain a segmentation of the image into regions (Briggman *et al.*, 2009; Turaga *et al.*, 2010; Farabet *et al.*, 2013).
* The general idea is to assume that large groups of contiguous pixels tend to be associated with the same label. Graphical models can describe the probabilistic relationships between neighboring pixels.
  1. **Data Types**

One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues.

For example, consider a collection of images, where each image has a different width and height. It is unclear how to model such inputs with a weight matrix of fixed size. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly.

Sometimes the output of the network is allowed to have variable size as well as the input, for example if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example if we want to assign a single class label to the entire image. In this case we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, in order to maintain a fixed number of pooled outputs.

Note that the use of convolution for processing variable sized inputs only makes sense for inputs that have variable size because they contain varying amounts of observation of the same kind of thing—different lengths of recordings over time, different widths of observations over space, etc. Convolution does not make sense if the input has variable size because it can optionally include different kinds of observations. For example, if we are processing college applications, and our features consist of both grades and standardized test scores, but not every applicant took the standardized test, then it does not make sense to convolve the same weights over both the features corresponding to the grades and the features corresponding to the test scores.

|  |  |  |
| --- | --- | --- |
|  | Single channel | Multi-channel |
| 1-D | Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step. | Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a “skeleton” over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character’s skeleton. Each channel in the data we feed to the convolutional  model represents the angle about one axis of one joint. |
| 2-D | Audio data that has been preprocessed with a Fourier transform: We can transform the audio waveform into a 2D tensor with different rows corresponding to different  frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network’s output. | Color image data: One channel contains the red pixels, one the green pixels, and one the blue  pixels. The convolution kernel moves over both the horizontal and vertical axes of the image, conferring translation equivariance in both directions. |
| 3-D | Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans. | Color video data: One axis corresponds to time, one to the height of the video frame, and one to  the width of the video frame. |

**Table 1***: Examples of different formats of data that can be used with convolutional networks.*

* 1. **Efficient Convolution Algorithms**
* Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naïve implementation of discrete convolution.
* When a d-dimensional kernel can be expressed as the outer product of d vectors, one vector per dimension, the kernel is called *separable*. When the kernel is separable, naive convolution is inefficient. It is equivalent to compose d one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one d-dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors.
* If the kernel is w elements wide in each dimension, then naive multidimensional convolution requires O(wd) runtime and parameter storage space, while separable convolution requires O(wxd) runtime and parameter storage space. Of course, not every convolution can be represented in this way.
  1. **Random or Unsupervised Features**
* Typically, the most expensive part of convolutional network training is learning the features. The output layer is usually relatively inexpensive due to the small number of features provided as input to this layer after passing through several layers of pooling. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation
* through the entire network. One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.
* There are three basic strategies for obtaining convolution kernels without supervised training. One is to simply initialize them randomly. Another is to design them by hand, for example by setting each kernel to detect edges at a certain orientation or scale. Finally, one can learn the kernels with an unsupervised criterion. For example, Coates *et al.* (2011) apply k-means clustering to small image patches, then use each learned centroid as a convolution kernel.
* One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer. Learning the last layer is then typically a convex optimization problem, assuming the last layer is something like logistic regression or an SVM.
* Saxe *et al.* (2011) showed that layers consisting of convolution following by pooling naturallybecome frequency selective and translation invariant when assigned random weights.They argue that this provides an inexpensive way to choose the architecture ofa convolutional network: first evaluate the performance of several convolutionalnetwork architectures by training only the last layer, then take the best of thesearchitectures and train the entire architecture using a more expensive approach.
* An intermediate approach is to learn the features, but using methods that do not require full forward and back-propagation at every gradient step. As with multilayer perceptrons, we use greedy layer-wise pretraining, to train the first layer in isolation, then extract all features from the first layer only once, then train the second layer in isolation given those features, and so on.
* Convolutional networks offer us the opportunity to take the pretraining strategy one step further than is possible with multilayer perceptrons. Instead of training an entire convolutional layer at a time, we can train a model of a small patch, as Coates *et al.* (2011) do with k-means. We can then use the parameters from this patch-based model to define the kernels of a convolutional layer. This means that it is possible to use unsupervised learning to train a convolutional network **without ever using convolution during the training process**. Using this approach, we can train very large models and incur a high computational cost only at inference time (Ranzato *et al.*, 2007b; Jarrett *et al.*, 2009; Kavukcuoglu *et al.*, 2010; Coates *et al.*, 2013).
  1. **The Neuroscientific Basis for Convolutional Networks**
* Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many
* of the most basic facts about how the mammalian vision system works (Hubel and Wiesel, 1959, 1962, 1968). Their accomplishments were eventually recognized with a Nobel prize.
* They observed how neurons in the cat’s brain responded to images projected in precise locations on a screen in front of the cat. Their great discovery was that neurons in the early visual system responded most strongly to very specific patterns of light, such as precisely oriented bars, but responded hardly at all to other patterns.
* We focus on a part of the brain called *V1*, also known as the *primary visual cortex*. V1 is the first area of the brain that begins to perform significantly advanced processing of visual input.
* The neurons in the retina perform some simple preprocessing of the image but do not substantially alter the way it is represented. The image then passes through the optic nerve and a brain region called the lateral geniculate nucleus. The main role, as far as we are concerned here, of both of these anatomical regions is primarily just to carry the signal from the eye to V1, which is located at the back of the head.
* A convolutional network layer is designed to capture three properties of V1:

1. V1 is arranged in a spatial map. It actually has a two-dimensional structure mirroring the structure of the image in the retina. For example, light arriving at the lower half of the retina affects only the corresponding half of V1.
2. V1 contains many *simple cells*. A simple cell’s activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field.
3. V1 also contains many *complex cells*. These cells respond to features that are similar to those detected by simple cells, but complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks. Complex cells are also invariant to some changes in lighting that cannot be captured simply by pooling over spatial locations. These invariances have inspired some of the cross-channel pooling strategies in convolutional networks, such as maxout units (Goodfellow *et al.*, 2013a).

* The basic strategy of detection followed by pooling is repeatedly applied as we move deeper into the brain. As we pass through multiple anatomical layers of the brain, we eventually find cells that respond to some specific concept and are invariant to many transformations of the input. These cells have been nicknamed “grandmother cells”—the idea is that a person could have a neuron that activates when seeing an image of their grandmother, regardless of whether she appears in the left or right side of the image, whether the image is a close-up of her face or zoomed out shot of her entire body, whether she is brightly lit, or in shadow, etc.
* Researchers tested whether individual neurons would respond to photos of famous individuals. They found what has come to be called the “Halle Berry neuron”: an individual neuron that is activated by the concept of Halle Berry. This neuron fires when a person sees a photo of Halle Berry, a drawing of Halle Berry, or even text containing the words “Halle Berry.” Of course, this has nothing to do with Halle Berry herself; other neurons responded to the presence of Bill Clinton, Jennifer Aniston, etc.
* The closest analog to a convolutional network’s last layer of features is a brain area called the inferotemporal cortex (IT). When viewing an object, information flows from the retina, through the LGN, to V1, then onward to V2, then V4, then IT. This happens within the first 100ms of glimpsing an object. If a person is allowed to continue looking at the object for more time, then information will begin to flow backwards as the brain uses top-down feedback to update the activations in the lower level brain areas.
* However, if we interrupt the person’s gaze, and observe only the firing rates that result from the first 100ms of mostly feedforward activation, then IT proves to be very similar to a convolutional network. Convolutional networks can predict IT firing rates, and also perform very similarly to (time limited) humans on object recognition tasks (DiCarlo, 2013).
* That being said, there are many differences between convolutional networks and the mammalian vision system. Some of these differences are not yet known, because many basic questions about how the mammalian vision system works remain unanswered. As a brief list:

1. The human eye is mostly very low resolution, except for a tiny patch called the *fovea*. The fovea only observes an area about the size of a thumbnail held at arm’s length. Though we feel as if we can see an entire scene in high resolution, this is an illusion created by the subconscious part of our brain, as it stitches together several glimpses of small areas. Most convolutional networks actually receive large full resolution photographs as input.
2. The human visual system is integrated with many other senses, such as hearing, and factors like our moods and thoughts. Convolutional networks so far are purely visual.
3. The human visual system does much more than just recognize objects. It is able to understand entire scenes including many objects and relationships between objects. Convolutional networks have been applied to some of these problems but these applications are in their infancy.
4. Even simple brain areas like V1 are heavily impacted by feedback from higher levels. Feedback has been explored extensively in neural network models but has not yet been shown to offer a compelling improvement.
5. While feedforward IT firing rates capture much of the same information as convolutional network features, it is not clear how similar the intermediate computations are. The brain probably uses very different activation and pooling functions.

* In a deep, nonlinear network, it can be difficult to understand the function of individual cells. Simple cells in the first layer are easier to analyze, because their responses are driven by a linear function. In an artificial neural network, we can just display an image of the convolution kernel to see what the corresponding channel of a convolutional layer responds to. In a biological neural network, we do not have access to the weights themselves. Instead, we put an electrode in the neuron itself, display several samples of white noise images in front of the animal’s retina, and record how each of these samples causes the neuron to activate. We can then fit a linear model to these responses in order to obtain an approximation of the neuron’s weights. This approach is known as *reverse correlation* (Ringach and Shapley, 2004).
* Reverse correlation shows us that most V1 cells have weights that are described by *Gabor functions*. The Gabor function describes the weight at a 2-D point in the image. We can think of an image as being a function of 2-D coordinates, I (x, y). Likewise, we can think of a simple cell as sampling the image at a set of locations, defined by a set of x coordinates X and a set of y coordinates, Y, and applying weights that are also a function of the location, w(x, y). From this point of view, the response of a simple cell to an image is given by

Specifically, w(x, y) takes the form of a Gabor function:

where

and

Here, α, β x, βy , f, φ, x0, y0, and τ are parameters that control the properties of the Gabor function. Above equation shows some examples of Gabor functions with different settings of these parameters.

* The parameters x0, y0 , and τ define a coordinate system. We translate and rotate x and y to form x’and y’. Specifically, the simple cell will respond to image features centered at the point (x0, y0), and it will respond to changes in brightness as we move along a line rotated τ radians from the horizontal.
* Viewed as a function of x’ and y’, the function w then responds to changes in brightness as we move along the x’ axis. It has two important factors: one is a Gaussian function and the other is a cosine function.
* The Gaussian factor can be seen as a gating term that ensures the simple cell will only respond to values near where x’ and y’ are both zero, in other words, near the center of the cell’s receptive field. The scaling factor α adjusts the total magnitude of the simple cell’s response, while βx and βy control how quickly its receptive field falls off.
* The cosine factor cos (f x’ + φ ) controls how the simple cell responds to changing brightness along the x’ axis. The parameter f controls the frequency of the cosine and φ controls its phase offset.
* Altogether, this cartoon view of simple cells means that a simple cell responds to a specific spatial frequency of brightness in a specific direction at a specific location. Simple cells are most excited when the wave of brightness in the image has the same phase as the weights. This occurs when the image is bright where the weights are positive and dark where the weights are negative.
* The cartoon view of a complex cell is that it computes the L2 norm of the 2-D vector containing two simple cells’ responses: c(I) =. An important special case occurs when s1 has all of the same parameters as s0 except for φ , and φ is set such that s1 is one quarter cycle out of phase with s0 . In this case, s0 and s1 form a quadrature pair. A complex cell defined in this way responds when the Gaussian reweighted image contains a high amplitude sinusoidal wave with frequency f in direction τ near (x0 , y0), **regardless of the phase offset of this wave**. In other words, the complex cell is invariant to small translations of the image in direction τ , or to negating the image (replacing black with white and vice versa).