**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.

**IMAGE**

* **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.

**IMAGE**

* 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.

**IMAGE 9.8**

* 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.

**IMAGE 9.9, 9.10**

* 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**
  2. **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.

**IMAGE 9.17**

* 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**
  2. **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**