0.1 Intro

Conv Nets 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 takings samples at regular time intervals, and image data which can be thought of as a 2D grid of pixels.

The name convolutional neural network implies that the network employs a mathematical operation called a **convolution**. CNNs use convolution in place of general matrix multiplication in at least one of their layers.

0.2 The Convolution Operation

The convolution is an operation on two functions of a real valued argument. Suppose we are tracking the location of a spaceship with a laser sensor. The laser provides a single output $x(t)$, the position of the space ship at time t. Both x and t are real valued.

Now suppose our laser sensor is noisy. To obtain a less noisy estimate of the ships position, we would like to average together several measurements. More recent measurements are more relevant, so we will want to give a weighted average with more weight to recent measurements. We can do this by applying a weighting function $w(a)$ where a is the age of the measurement.

If we apply such a weighted average operation at every moment, we obtain a new function s providing a smoothed estimate of the position of the spaceship

$$s(t) = \int x(a)w(t-a)da$$

This operation is called a convolution and is typically denoted with an asterisk

$$s(t) = (x * w)(t)$$

In our example, w needs to be a valid probability distribution. In convolutional network terminology, the first argument (x) to the convolution is often referred to as the **input** and the second argument (w) is known as the **kernel**. The output is sometimes referred to as the **feature map**.

For a discrete convolution, we can assume that $t \in N$. Then we can define the discrete convolution as

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In ml applications, the input is generally a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm.

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 also want to use a two dimensional kernel K:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution is commutative, so equivalently we could write

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

While the commutativity of convolutions is useful for writing proofs, in practice neural network libraries often implement a related function called **cross-correlation**, which is the same as convolution but without flipping the kernel:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Discrete convolution can be viewed as multiplication by a matrix. 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, convolution usually cooresponds to a very sparse matrix. This is because the kernel is usually much smaller than the input image. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution without requiring further changes to the network.

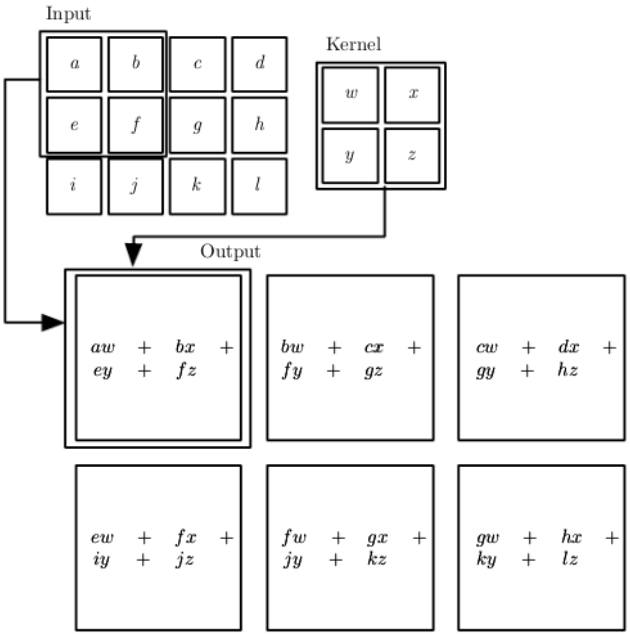


Figure 1: 2-D convolution without kernel flipping

0.3 Motivation

Convolution leverages three important ideas that can help improve a machine learning system:

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

0.3.1 sparse interactions

Traditional neural networks 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 that each input unit interacts with each output unit.

Convolutional neural networks in contrast typically have **sparse interactions** (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. When processing an image, there may be thousands or millions of pixels, but we can detect small meaningful features such as edges with kernels that only occupy tens of pixels. This means we can store fewer parameters, reducing the space and time requirements and improving statistical efficiency.

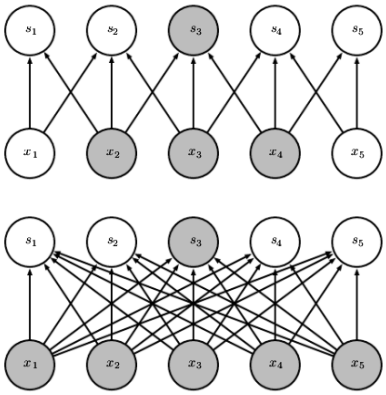


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, s_3, and also highlight the input units in $\boldsymbol{x}$ that affect this unit. These units are known as the **receptive field** of s_3. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three inputs affect s_3. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3.

0.3.2 parameter sharing

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional network, we have **tied weights**. This means that the value of a weight applied to one input is tied to the value of a weight applied elsewhere.

In a conv net, each member of the kernel is used at every position of the input (except perhaps the boundary pixels, depending on the design decisions). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we can learn only one set. Convolution is thus dramatically more efficient than dense matrix multiplicationin terms of the memory requirements and statistical efficiency.

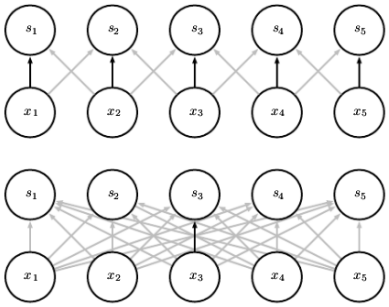


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. *(Top)*The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Bottom)*The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

0.3.3 equivariance

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.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

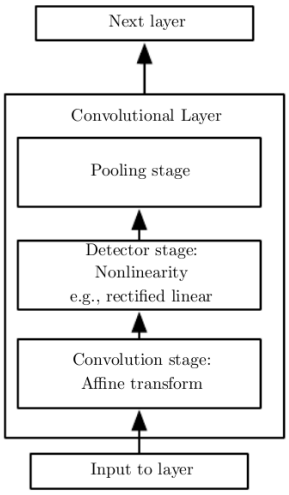
0.4 Pooling

A typical layer of a convolutional neural network consists of three stages:

1. The layer performs several convolutions in parallel to produce a set of linear activations. 2. Each linear activation is run through a nonlinear activation function, such as ReLU 3. This stage is sometimes called the **detector** 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, max pooling returns the maximum output within a rectangular neighborhood. Other

popular pooling functions include the average, the L^2 norm, 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 local translation can be a very useful property if we care more about whether some feature is present than exactly where it is. The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the efficiency of the network.

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.

Some theoretical work gives guidance on which kinds of pooling to use in various situations (Boureau et al, 2010):

> A Theoretical Analysis of Feature Pooling in Visual Recognition
>
> Many modern visual recognition algorithms incorporate a step of spatial 'pooling', where the outputs of several nearby feature detectors are combined into a local or global 'bag of features',

in a way that preserves task-related information while removing irrelevant details. Pooling is used to achieve invariance to image transformations, more compact representations, and better robustness to noise and clutter. Several papers have shown that the details of the pooling operation can greatly influence the performance, but studies have so far been purely empirical. In this paper, we show that the reasons underlying the performance of various pooling methods are obscured by several confounding factors, such as the link between the sample cardinality in a spatial pool and the resolution at which low-level features have been extracted. We provide a detailed theoretical analysis of max pooling and average pooling, and give extensive empirical comparisons for object recognition tasks.

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.

Ask the locals: multi-way local pooling for image recognition

Invariant representations in object recognition systems are generally obtained by pooling feature vectors over spatially local neighborhoods. But pooling is not local in the feature vector space, so that widely dissimilar features may be pooled together if they are in nearby locations. Recent approaches rely on sophisticated encoding methods and more specialized codebooks (or dictionaries), e.g., learned on subsets of descriptors which are close in feature space, to circumvent this problem. In this work, we argue that a common trait found in much recent work in image recognition or retrieval is that it leverages locality in feature space on top of purely spatial locality. We propose to apply this idea in its simplest form to an object recognition system based on the spatial pyramid framework, to increase the performance of small dictionaries with very little added engineering. Stateof-the-art results on several object recognition benchmarks show the promise of this approach.

Another approach is to learn a single pooling structure that is then applied to all images (Jia et al, 2012)

Beyond Spartial Pyramids: Receptive Field Learning for Pooled Image Features

In this paper we examine the effect of receptive field designs on classification accuracy in the commonly adopted pipeline of image classification. While existing algorithms usually use manually defined spatial regions for pooling, we show that learning more adaptive receptive fields increases performance even with a significantly smaller codebook size at the coding layer. To learn the optimal pooling parameters, we adopt the idea of over-completeness by starting with a large number of receptive field candidates, and train a classifier with structured sparsity to only use a sparse subset of all the features. An efficient algorithm based on incremental feature selection and retraining is proposed for fast learning. With this method, we achieve the best published performance on the CIFAR-10 dataset, using a much lower dimensional feature space than previous methods.

0.5 Convolution and Pooling as an Infinitely Strong Prior

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a distribution with high entropy, such as a Gaussian 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 with low variance. Such a prior plays an active role in determining where the parameters end up. An infinitely strong prior places zero probability on some parameters and says that these prarameter values are completely forbidden, regardless of any support from the data.

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. This says that the function the layer should learn contains only local interactions and is equivariant to small translations.

Implementing a fully connected convnet would be ridiculously wasteful, but considering a convolution as an infinitely strong prior can give us in-

sight into how convolutional networks work. A prior is only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving the precise spatial information, then using pooling on all features can increase the training error. When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

Some convolutional network architectures (Szegedy et al, 2014) are designed to use pooling on some channels, but not on other channels in order to get highly invariant features and features that will not underfit when the translation invariance prior in incorrect.

> Going Deeper with Convolutions
>
> We propose a deep convolutional neural network architecture codenamed Inception, which was responsible for setting the new state of the art for classification and detection in the ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14). The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant. To optimize quality, the architectural decisions were based on the Hebbian principle and the intuition of multi-scale processing. One particular incarnation used in our submission for ILSVRC14 is called GoogLeNet, a 22 layers deep network, the quality of which is assessed in the context of classification and detection.

0.6 Variants of the Basic Convolution Function

Assume we have a 4D kernel tensor K with element $K_{i,j,k,l}$ 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 $V_{i,j,k}$ 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

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1} K_{i,l,m,n}$$

We may wish to skip over some of the positions of the kernel in order to reduce the computational cost (at expense). We can think of this as downsampling the output of the full convolution function. If we wish to sample only every s pixels in each direction in the output, then we can define a downsampled convolution function c such that

$$Z_{i,j,k} = c(K, V, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}]$$

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

One essential feature of any convolutional network is its ability to implicitly zero pad the input V in order to make it wider. Without this, the width of the representation would shrink with each layer. Without padding we would need to choose between shrinking the spatial extent of the network rapidly and using small kernels, both of which limit the expressive power of the network.

Three special cases of zero-padding settings are worth mentioning :

1. No zero-padding whatsoever. This is called the **valid** convolution. In it the convolutional kernel is only allowed to visit positions where the entire kernel is contained entirely within the image.

2. Just enough zero-padding to keep the size of the output equal to the size of the input. This is called the **same** convolution. Since the operation of convolution doesn't change the size of the input, these can handle as many layers as the hardware can. 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.

3. Enough zeroes are added for every pixel to be visited k times in each direction, resulting in an output image of width $m + k + 1$. This is called the **full** convolution. In this case, the output pixels near the border are a function of fewer pixels than the outputs near the center. This makes it difficult to learn a single kernel that performs well at all positions in the convolutional feature map.

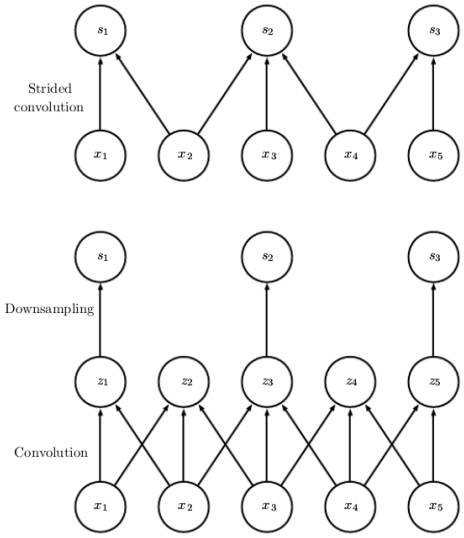


Figure 9.12: Convolution with a stride. In this example, we use a stride of two. *(Top)*Convolution with a stride length of two implemented in a single operation. *(Bottom)*Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by downsampling. Obviously, the two-step approach involving downsampling is computationally wasteful, because it computes many values that are then discarded.

Usually the optimal amount of zero-padding 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 6D 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

$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+n-1} w_{i,j,k,l,m,n}]$$

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