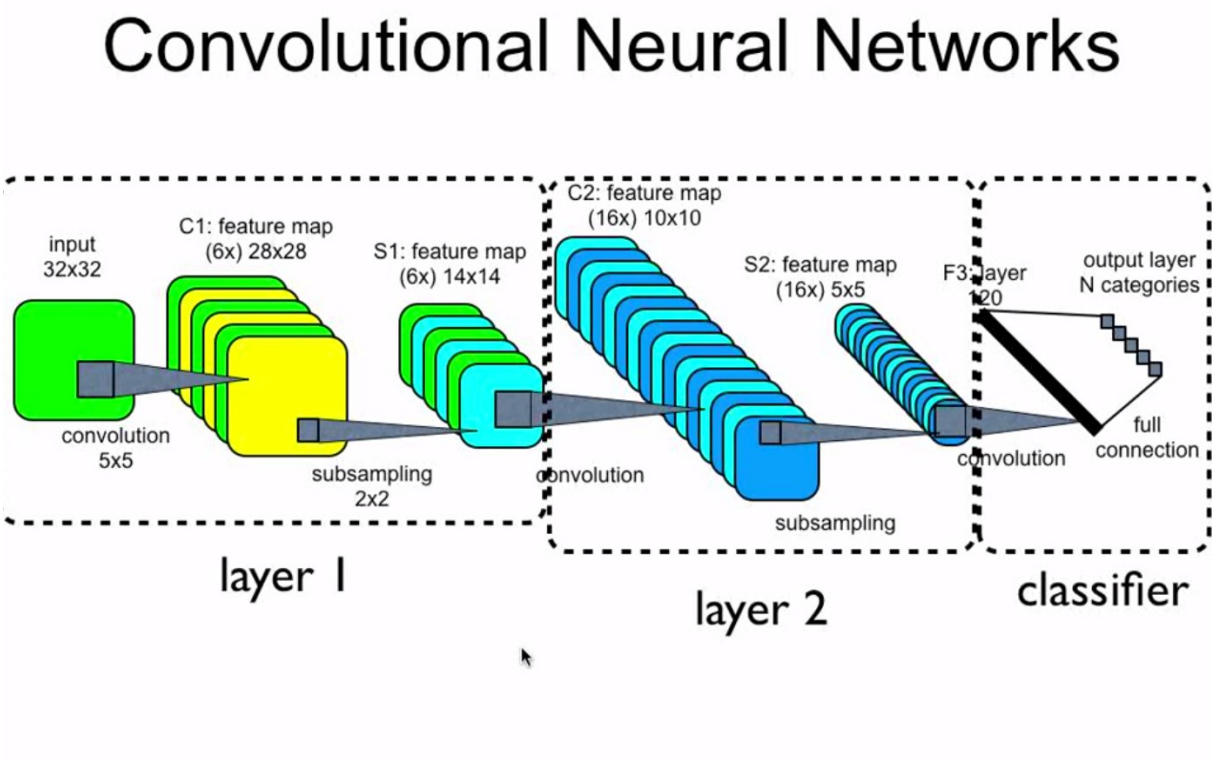
When learning convolutional neural network, I have questions regarding the following figure.

1) C1 in layer 1 has 6 feature maps, does that mean there are six convolutional kernels? Each convolutional kernel is used to generate a feature map based on input.

2) S1 in layer 2 has 6 feature maps, C2 has 16 feature maps. What is the process look like to get these 16 feature maps based on 6 feature maps in S1?



1) C1 in the layer 1 has 6 feature maps, does that mean there are six convolutional kernels? Each convolutional kernel is used to generate a feature map based on input.

There are 6 convolutional kernels and each is used to generate a feature map based on input. Another way to say this is that there are 6 filters or 3D sets of weights which I will just call weights. What this image doesn't show, that it probably should, to make it clearer is that typically images have 3 channels, say red, green, and blue. So the weights that map you from the input to C1 are of shape/dimension 3x5x5 not just 5x5. The same 3 dimensional weights, or kernel, are applied across the entire 3x32x32 image to generate a 2 dimensional feature map in C1. There are 6 kernels (each 3x5x5) in this example so that makes 6 feature maps (each 28x28 since the stride is 1 and padding is zero) in this example, each of which is the result of applying a 3x5x5 kernel across the input.

2) S1 in layer 1 has 6 feature maps, C2 in layer 2 has 16 feature maps. What is the process look like to get these 16 feature maps based on 6 feature maps in S1?

Now do the same thing we did in layer one, but do it for layer 2, except this time the number of channels is not 3 (RGB) but 6, six for the number of feature maps/filters in S1. There are now 16 unique kernels each of shape/dimension 6x5x5. each layer 2 kernel is applied across

all of S1 to generate a 2D feature map in C2. This is done 16 times for each unique kernel in layer 2, all 16, to generate the 16 feature maps in layer 2 (each 10x10 since stride is 1 and padding is zero)

Convolutional Neural Networks (CNNs / ConvNets)

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous chapter: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

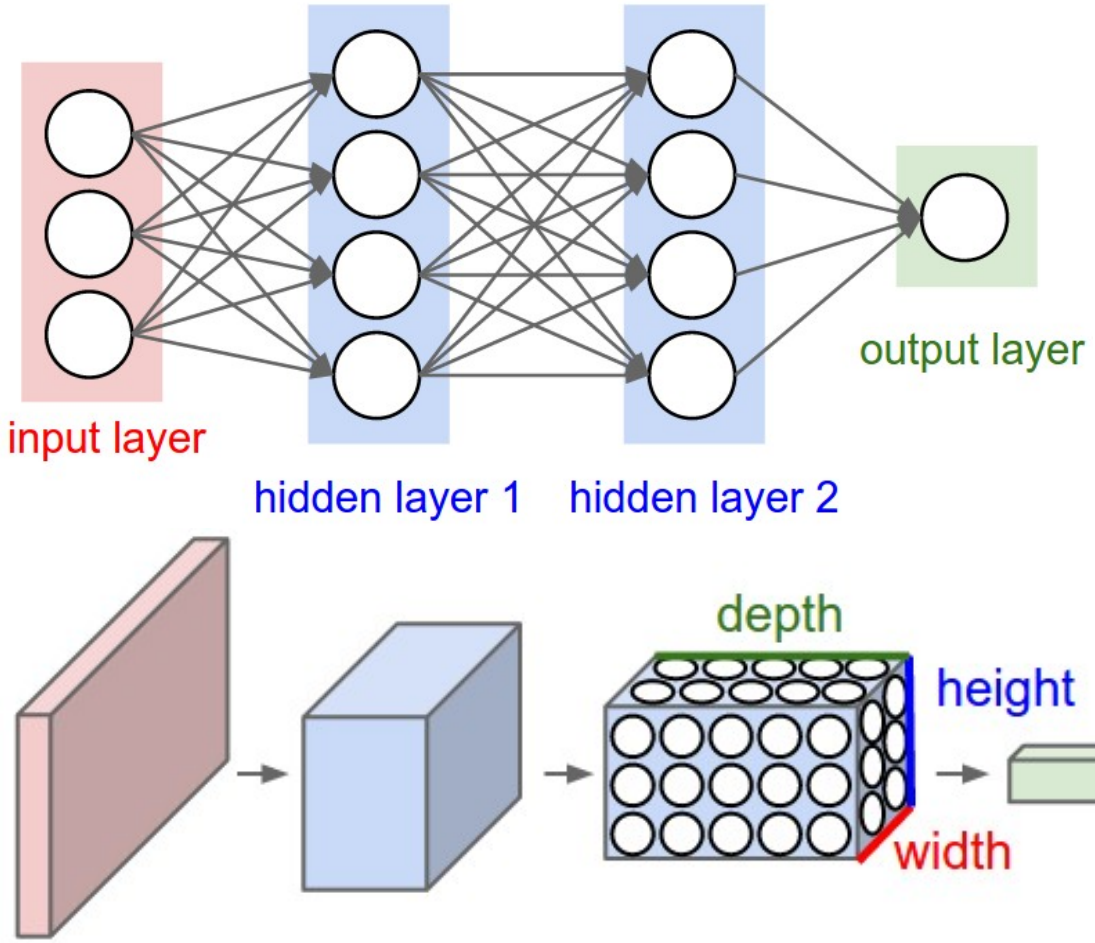
So what changes? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Architecture Overview

Recall: Regular Neural Nets. As we saw in the previous chapter, Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores.

Regular Neural Nets don’t scale well to full images. In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, e.g. 200x200x3, would lead to neurons that have 200*200*3 = 120,000 weights. Moreover, we would almost certainly want to have several such neurons, so the parameters would add up quickly! Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

3D volumes of neurons. Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**. (Note that the word *depth* here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). As we will soon see, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

A ConvNet is made up of Layers. Every Layer has a simple API: It transforms an input 3D volume to an output 3D volume with some differentiable function that may or may not have parameters.

Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

Example Architecture: Overview. We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. In more detail:

- INPUT [32x32x3] will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.

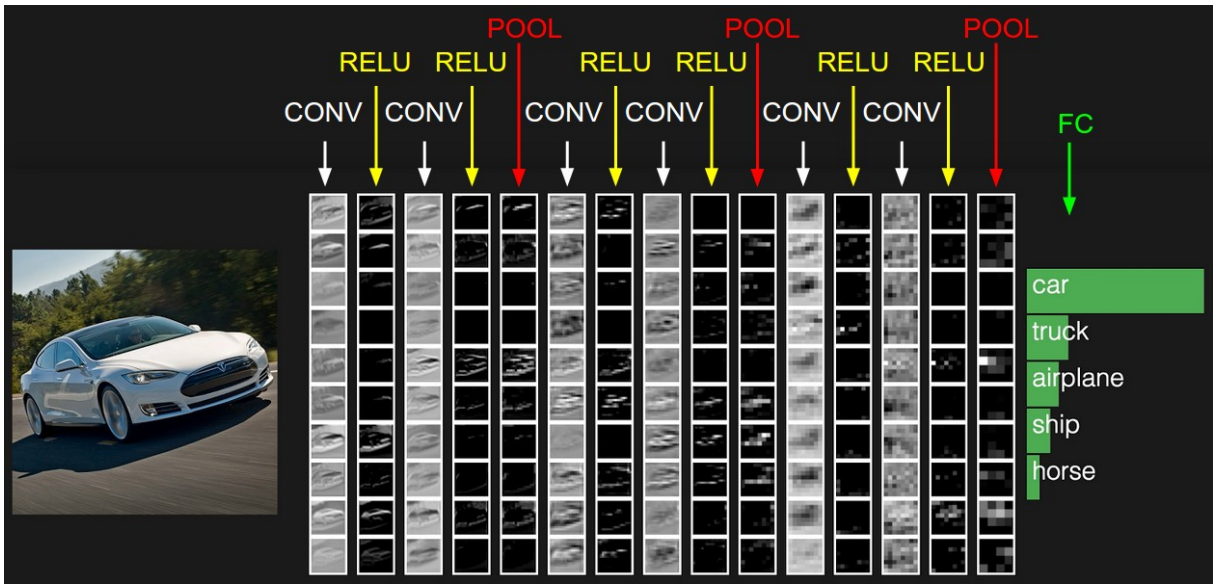
- CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer will apply an elementwise activation function, such as the $max(0,x)$

- thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

In summary:

- A ConvNet architecture is in the simplest case a list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- There are a few distinct types of Layers (e.g. CONV/FC/RELU/POOL are by far the most popular)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters (e.g. CONV/FC do, RELU/POOL don't)
- Each Layer may or may not have additional hyperparameters (e.g. CONV/FC/POOL do, RELU doesn't)



The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one. The full web-based demo is shown in the header of our website. The architecture shown here is a tiny VGG Net, which we will discuss later.

We now describe the individual layers and the details of their hyperparameters and their connectivities.

Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting.

Overview and intuition without brain stuff. Lets first discuss what the CONV layer computes without brain/neuron analogies. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

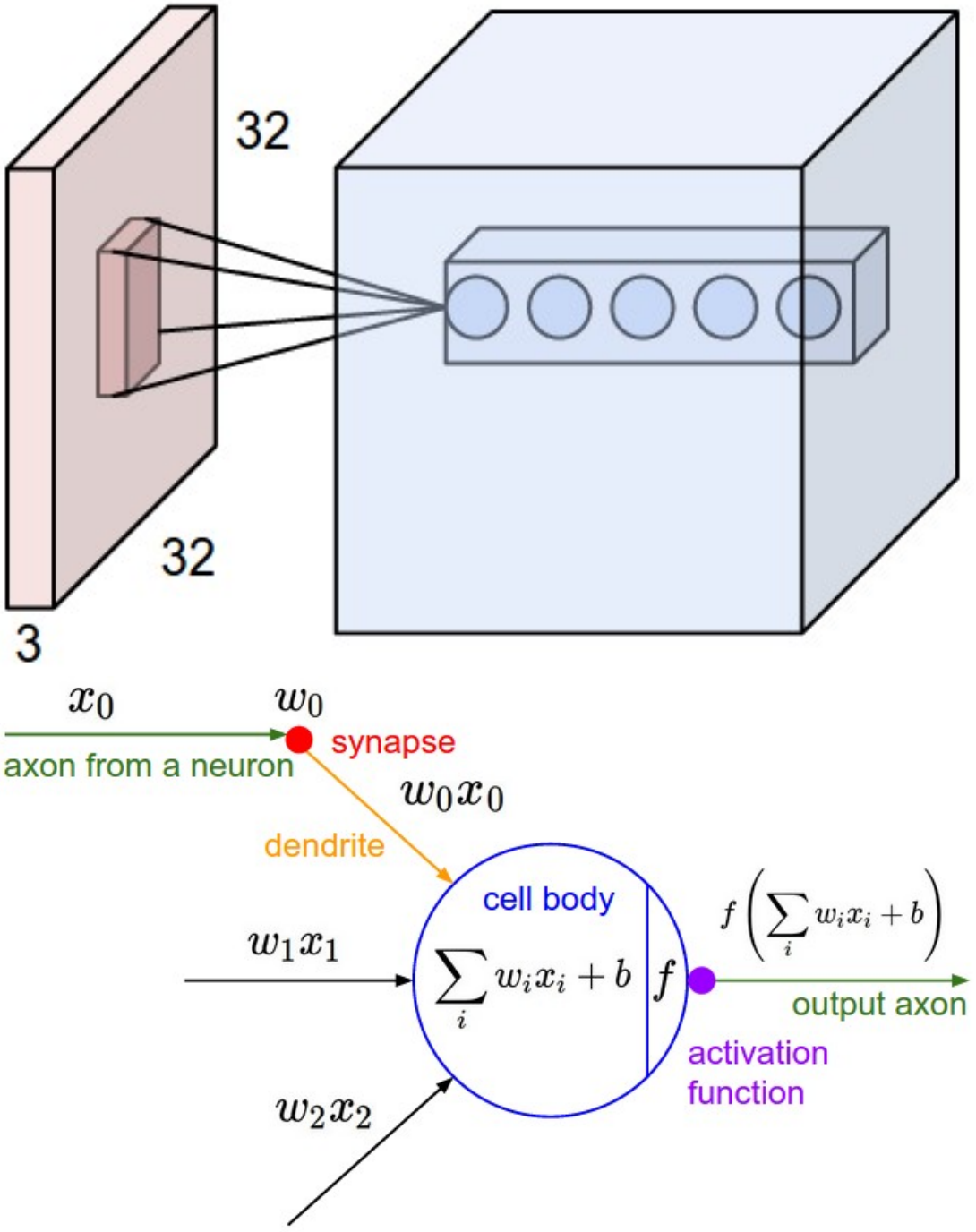
The brain view. If you're a fan of the brain/neuron analogies, every entry in the 3D output volume can also be interpreted as an output of a neuron that looks at only a small region in the

input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter). We now discuss the details of the neuron connectivities, their arrangement in space, and their parameter sharing scheme.

Local Connectivity. When dealing with high-dimensional inputs such as images, as we saw above it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron (equivalently this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. It is important to emphasize again this asymmetry in how we treat the spatial dimensions (width and height) and the depth dimension: The connections are local in space (along width and height), but always full along the entire depth of the input volume.

Example 1. For example, suppose that the input volume has size [32x32x3], (e.g. an RGB CIFAR-10 image). If the receptive field (or the filter size) is 5x5, then each neuron in the Conv Layer will have weights to a [5x5x3] region in the input volume, for a total of 5*5*3 = 75 weights (and +1 bias parameter). Notice that the extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume.

Example 2. Suppose an input volume had size [16x16x20]. Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).



Left: An example input volume in red (e.g. a 32x32x3 CIFAR-10 image), and an example volume of neurons in the first Convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e. all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input - see discussion of depth columns in text below. **Right:** The neurons from the Neural Network chapter remain unchanged: They still compute a dot product of their weights with the input followed by a non-linearity, but their connectivity is now restricted to be local spatially.

Spatial arrangement. We have explained the connectivity of each neuron in the Conv Layer to the input volume, but we haven't yet discussed how many neurons there are in the output

volume or how they are arranged. Three hyperparameters control the size of the output volume: the **depth, stride** and **zero-padding**. We discuss these next:

1. First, the **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
2. Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.
3. As we will soon see, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we'll see soon we will use it to exactly preserve the spatial size of the input volume so the input and output width and height are the same).

We can compute the spatial size of the output volume as a function of the input volume size (W

), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the amount of zero padding used (P) on the border. You can convince yourself that the correct formula for calculating how many neurons "fit" is given by $(W-F+2P)/S+1$

. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output. Lets also see one more graphical example:

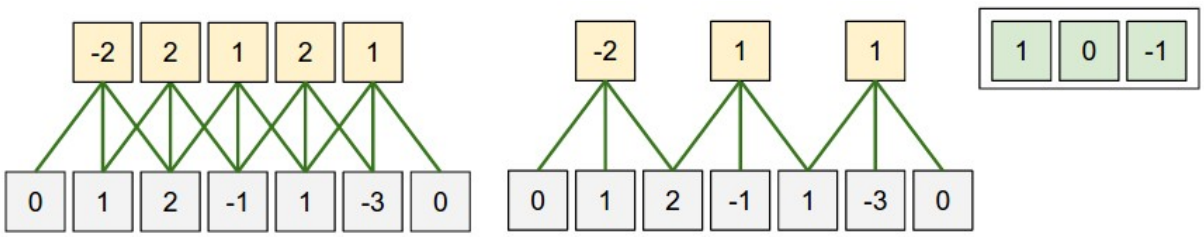


Illustration of spatial arrangement. In this example there is only one spatial dimension (x-axis), one neuron with a receptive field size of F = 3, the input size is W = 5, and there is zero padding of P = 1. **Left:** The neuron strided across the input in stride of S = 1, giving output of size (5 - 3 + 2)/1+1 = 5. **Right:** The neuron uses stride of S = 2, giving output of size (5 - 3 + 2)/2+1 = 3. Notice that stride S = 3 could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since (5 - 3 + 2) = 4 is not divisible by 3.

The neuron weights are in this example [1,0,-1] (shown on very right), and its bias is zero. These weights are shared across all yellow neurons (see parameter sharing below).

Use of zero-padding. In the example above on left, note that the input dimension was 5 and the output dimension was equal: also 5. This worked out so because our receptive fields were 3 and we used zero padding of 1. If there was no zero-padding used, then the output volume would have had spatial dimension of only 3, because that it is how many neurons would have "fit" across the original input. In general, setting zero padding to be $P=(F-1)/2$

when the stride is $S=1$

ensures that the input volume and output volume will have the same size spatially. It is very common to use zero-padding in this way and we will discuss the full reasons when we talk more about ConvNet architectures.

Constraints on strides. Note again that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size $W=10$

, no zero-padding is used $P=0$, and the filter size is $F=3$, then it would be impossible to use stride $S=2$, since $(W-F+2P)/S+1=(10-3+0)/2+1=4.5$

, i.e. not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a ConvNet library could throw an exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. As we will see in the ConvNet architectures section, sizing the ConvNets appropriately so that all the dimensions "work out" can be a real headache, which the use of zero-padding and some design guidelines will significantly alleviate.

Real-world example. The Krizhevsky et al. architecture that won the ImageNet challenge in 2012 accepted images of size [227x227x3]. On the first Convolutional Layer, it used neurons with receptive field size $F=11$

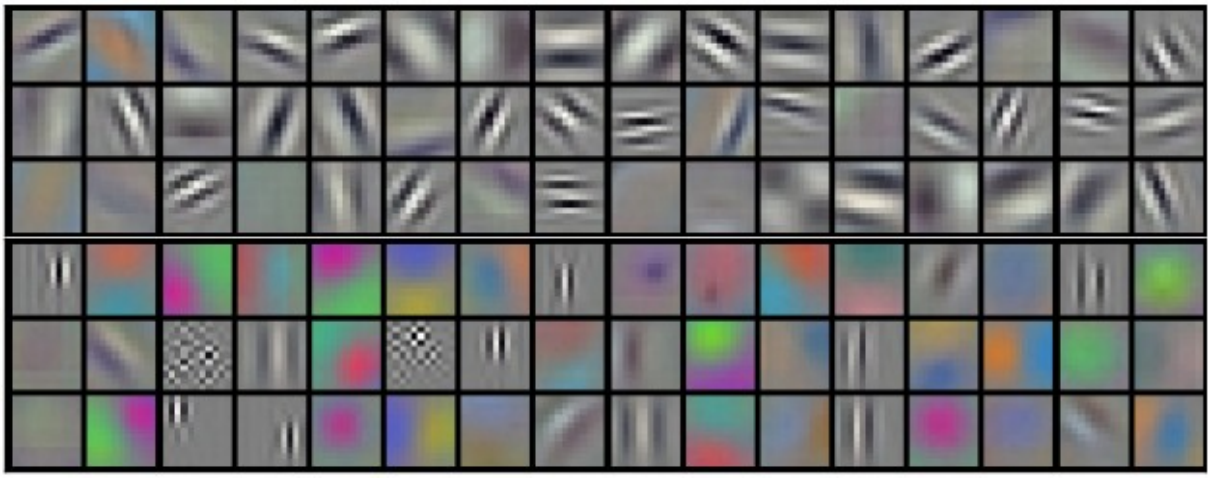
, stride $S=4$ and no zero padding $P=0$. Since (227 - 11)/4 + 1 = 55, and since the Conv layer had a depth of $K=96$

, the Conv layer output volume had size [55x55x96]. Each of the 55*55*96 neurons in this volume was connected to a region of size [11x11x3] in the input volume. Moreover, all 96 neurons in each depth column are connected to the same [11x11x3] region of the input, but of course with different weights. As a fun aside, if you read the actual paper it claims that the input images were 224x224, which is surely incorrect because (224 - 11)/4 + 1 is quite clearly not an integer. This has confused many people in the history of ConvNets and little is known about what happened. My own best guess is that Alex used zero-padding of 3 extra pixels that he does not mention in the paper.

Parameter Sharing. Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. Using the real-world example above, we see that there are 55*55*96 = 290,400 neurons in the first Conv Layer, and each has 11*11*3 = 363 weights and 1 bias. Together, this adds up to 290400 * 364 = 105,705,600 parameters on the first layer of the ConvNet alone. Clearly, this number is very high.

It turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position (x,y), then it should also be useful to compute at a different position (x2,y2). In other words, denoting a single 2-dimensional slice of depth as a **depth slice** (e.g. a volume of size [55x55x96] has 96 depth slices, each of size [55x55]), we are going to constrain the neurons in each depth slice to use the same weights and bias. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique set of weights (one for each depth slice), for a total of 96*11*11*3 = 34,848 unique weights, or 34,944 parameters (+96 biases). Alternatively, all 55*55 neurons in each depth slice will now be using the same parameters. In practice during backpropagation, every neuron in the volume will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a **convolution** of the neuron's weights with the input volume (Hence the name: Convolutional Layer). This is why it is common to refer to the sets of weights as a **filter** (or a **kernel**), that is convolved with the input.



Example filters learned by Krizhevsky et al. Each of the 96 filters shown here is of size [11x11x3], and each one is shared by the 55*55 neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable: If detecting a horizontal edge is important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the 55*55 distinct locations in the Conv layer output volume.

Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than another. One practical example is when the input are faces that have been

centered in the image. You might expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations. In that case it is common to relax the parameter sharing scheme, and instead simply call the layer a **Locally-Connected Layer**.

Numpy examples. To make the discussion above more concrete, lets express the same ideas but in code and with a specific example. Suppose that the input volume is a numpy array `X`. Then:

- A *depth column* (or a *fibre*) at position `(x,y)` would be the activations `X[x,y,:]`.
- A *depth slice*, or equivalently an *activation map* at depth `d` would be the activations `X[:,:,d]`.

Conv Layer Example. Suppose that the input volume `X` has shape `X.shape: (11,11,4)`. Suppose further that we use no zero padding ($P=0$

), that the filter size is $F=5$, and that the stride is $S=2$

. The output volume would therefore have spatial size (11-5)/2+1 = 4, giving a volume with width and height of 4. The activation map in the output volume (call it `V`), would then look as follows (only some of the elements are computed in this example):

- `V[0,0,0] = np.sum(X[:5,:5,:] * W0) + b0 # dot prod. across all dim !`
- `V[1,0,0] = np.sum(X[2:7,:5,:] * W0) + b0`
- `V[2,0,0] = np.sum(X[4:9,:5,:] * W0) + b0`
- `V[3,0,0] = np.sum(X[6:11,:5,:] * W0) + b0`

Remember that in numpy, the operation `*` above denotes elementwise multiplication between the arrays. Notice also that the weight vector `W0` is the weight vector of that neuron and `b0` is the bias. Here, `W0` is assumed to be of shape `W0.shape: (5,5,4)`, since the filter size is 5 and the depth of the input volume is 4. Notice that at each point, we are computing the dot product as seen before in ordinary neural networks. Also, we see that we are using the same weight and bias (due to parameter sharing), and where the dimensions along the width are increasing in steps of 2 (i.e. the stride). To construct a second activation map in the output volume, we would have:

- `V[0,0,1] = np.sum(X[:5,:5,:] * W1) + b1`
- `V[1,0,1] = np.sum(X[2:7,:5,:] * W1) + b1`
- `V[2,0,1] = np.sum(X[4:9,:5,:] * W1) + b1`
- `V[3,0,1] = np.sum(X[6:11,:5,:] * W1) + b1`
- `V[0,1,1] = np.sum(X[:5,2:7,:] * W1) + b1` (example of going along y)
- `V[2,3,1] = np.sum(X[4:9,6:11,:] * W1) + b1` (or along both)

where we see that we are indexing into the second depth dimension in `V` (at index 1) because we are computing the second activation map, and that a different set of parameters (`W1`) is now used. In the example above, we are for brevity leaving out some of the other operations the Conv Layer would perform to fill the other parts of the output array `V`. Additionally, recall that these activation maps are often followed elementwise through an activation function such as ReLU, but this is not shown here.

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$

¬ ¬ Requires four hyperparameters:

- Number of filters K

¬ ,
¬ their spatial extent F
¬ ,
¬ the stride S
¬ ,
¬ the amount of zero padding P

- ¬ .

¬ Produces a volume of size $W_2 \times H_2 \times D_2$ where:

- $W_2 = (W_1 - F + 2P)/S + 1$

¬ ¬ $H_2 = (H_1 - F + 2P)/S + 1$
¬ (i.e. width and height are computed equally by symmetry)
¬ $D_2 = K$

- ¬

¬ With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K
¬ biases.
¬ In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d

- -th bias.

A common setting of the hyperparameters is $F = 3, S = 1, P = 1$

. However, there are common conventions and rules of thumb that motivate these hyperparameters. See the ConvNet architectures section below.

Convolution Demo. Below is a running demo of a CONV layer. Since 3D volumes are hard to visualize, all the volumes (the input volume (in blue), the weight volumes (in red), the

output volume (in green)) are visualized with each depth slice stacked in rows. The input volume is of size $W_1=5, H_1=5, D_1=3$

, and the CONV layer parameters are $K=2, F=3, S=2, P=1$. That is, we have two filters of size 3×3, and they are applied with a stride of 2. Therefore, the output volume size has spatial size (5 - 3 + 2)/2 + 1 = 3. Moreover, notice that a padding of $P=1$

is applied to the input volume, making the outer border of the input volume zero. The visualization below iterates over the output activations (green), and shows that each element is computed by elementwise multiplying the highlighted input (blue) with the filter (red), summing it up, and then offsetting the result by the bias.

Implementation as Matrix Multiplication. Note that the convolution operation essentially performs dot products between the filters and local regions of the input. A common implementation pattern of the CONV layer is to take advantage of this fact and formulate the forward pass of a convolutional layer as one big matrix multiply as follows:

1. The local regions in the input image are stretched out into columns in an operation commonly called **im2col**. For example, if the input is [227x227x3] and it is to be convolved with 11x11x3 filters at stride 4, then we would take [11x11x3] blocks of pixels in the input and stretch each block into a column vector of size 11*11*3 = 363. Iterating this process in the input at stride of 4 gives (227-11)/4+1 = 55 locations along both width and height, leading to an output matrix `X_col` of *im2col* of size [363 x 3025], where every column is a stretched out receptive field and there are 55*55 = 3025 of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.
2. The weights of the CONV layer are similarly stretched out into rows. For example, if there are 96 filters of size [11x11x3] this would give a matrix `W_row` of size [96 x 363].
3. The result of a convolution is now equivalent to performing one large matrix multiply `np.dot(W_row, X_col)`, which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be [96 x 3025], giving the output of the dot product of each filter at each location.
4. The result must finally be reshaped back to its proper output dimension [55x55x96].

This approach has the downside that it can use a lot of memory, since some values in the input volume are replicated multiple times in `X_col`. However, the benefit is that there are many very efficient implementations of Matrix Multiplication that we can take advantage of (for example, in the commonly used BLAS API). Moreover, the same *im2col* idea can be reused to perform the pooling operation, which we discuss next.

Backpropagation. The backward pass for a convolution operation (for both the data and the weights) is also a convolution (but with spatially-flipped filters). This is easy to derive in the 1-dimensional case with a toy example (not expanded on for now).

1x1 convolution. As an aside, several papers use 1x1 convolutions, as first investigated by Network in Network. Some people are at first confused to see 1x1 convolutions especially when they come from signal processing background. Normally signals are 2-dimensional so

1x1 convolutions do not make sense (it's just pointwise scaling). However, in ConvNets this is not the case because one must remember that we operate over 3-dimensional volumes, and that the filters always extend through the full depth of the input volume. For example, if the input is [32x32x3] then doing 1x1 convolutions would effectively be doing 3-dimensional dot products (since the input depth is 3 channels).

Dilated convolutions. A recent development (e.g. see paper by Fisher Yu and Vladlen Koltun) is to introduce one more hyperparameter to the CONV layer called the *dilation*. So far we've only discussed CONV filters that are contiguous. However, it's possible to have filters that have spaces between each cell, called dilation. As an example, in one dimension a filter `w` of size 3 would compute over input `x` the following: `w[0]*x[0] + w[1]*x[1] + w[2]*x[2]`. This is dilation of 0. For dilation 1 the filter would instead compute `w[0]*x[0] + w[1]*x[2] + w[2]*x[4]`; In other words there is a gap of 1 between the applications. This can be very useful in some settings to use in conjunction with 0-dilated filters because it allows you to merge spatial information across the inputs much more agressively with fewer layers. For example, if you stack two 3x3 CONV layers on top of each other then you can convince yourself that the neurons on the 2nd layer are a function of a 5x5 patch of the input (we would say that the *effective receptive field* of these neurons is 5x5). If we use dilated convolutions then this effective receptive field would grow much quicker.

Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged. More generally, the pooling layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$

¬ ¬ Requires two hyperparameters:

- their spatial extent F

¬ ,
¬ the stride S

- ¬ ,

¬ Produces a volume of size $W_2 \times H_2 \times D_2$ where:

- $W_2 = (W_1 - F)/S + 1$

¬ ¬ $H_2 = (H_1 - F)/S + 1$

¬ ¬ $D_2 = D_1$

-
  -

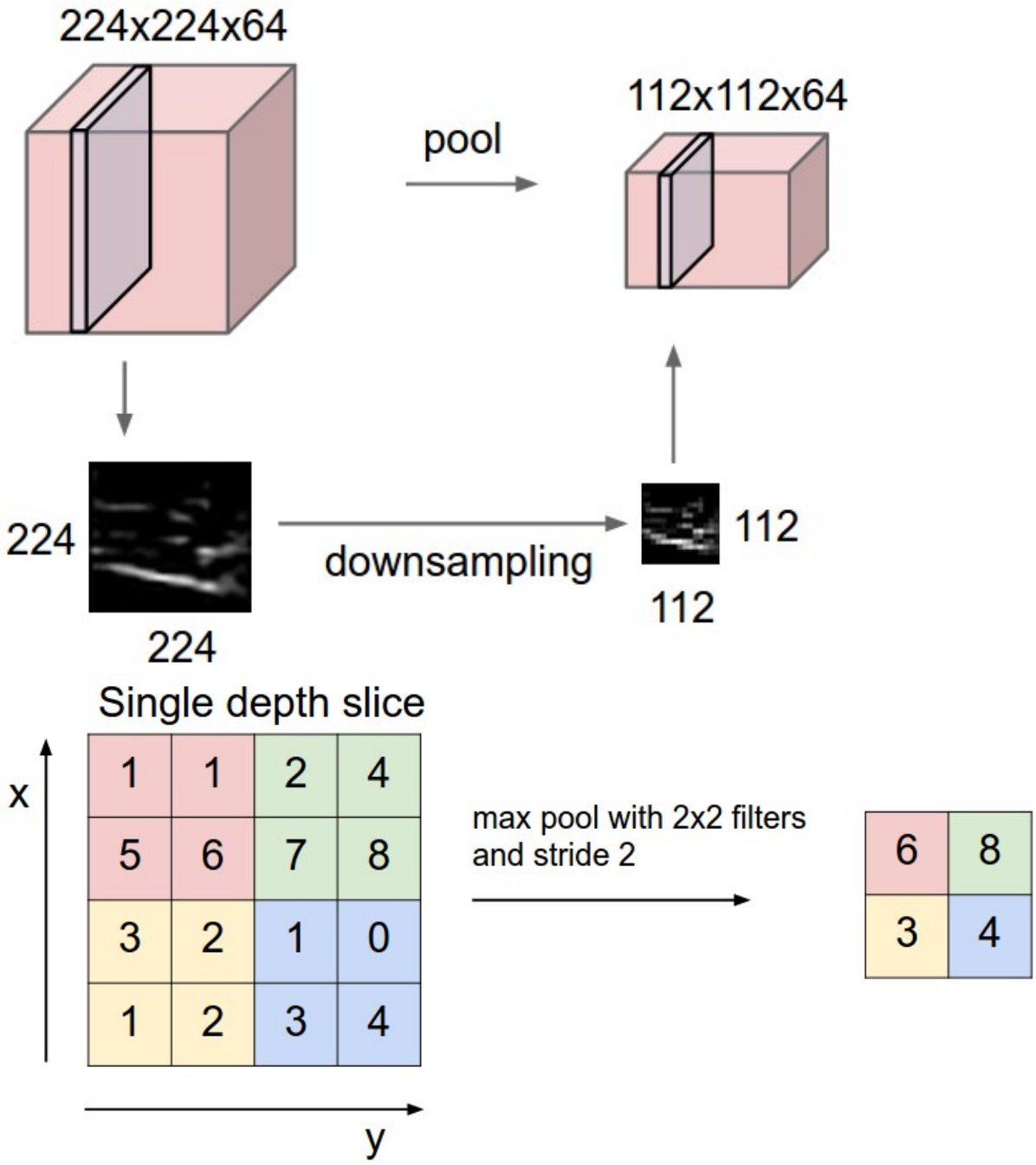
- Introduces zero parameters since it computes a fixed function of the input
- For Pooling layers, it is not common to pad the input using zero-padding.

It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with $F = 3, S = 2$

(also called overlapping pooling), and more commonly $F = 2, S = 2$

. Pooling sizes with larger receptive fields are too destructive.

General pooling. In addition to max pooling, the pooling units can also perform other functions, such as *average pooling* or even *L2-norm pooling*. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Backpropagation. Recall from the backpropagation chapter that the backward pass for a max(x, y) operation has a simple interpretation as only routing the gradient to the input that had the highest value in the forward pass. Hence, during the forward pass of a pooling layer it is common to keep track of the index of the max activation (sometimes also called *the switches*) so that gradient routing is efficient during backpropagation.

Getting rid of pooling. Many people dislike the pooling operation and think that we can get away without it. For example, Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs). It seems likely that future architectures will feature very few to no pooling layers.

Normalization Layer

Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have since fallen out of favor because in practice their contribution has been shown to be minimal, if any. For various types of normalizations, see the discussion in Alex Krizhevsky's cuda-convnet library API.

Fully-connected layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See the *Neural Network* section of the notes for more information.

Converting FC layers to CONV layers

It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical. Therefore, it turns out that it's possible to convert between FC and CONV layers:

- For any CONV layer there is an FC layer that implements the same forward function. The weight matrix would be a large matrix that is mostly zero except for at certain blocks (due to local connectivity) where the weights in many of the blocks are equal (due to parameter sharing).
- Conversely, any FC layer can be converted to a CONV layer. For example, an FC layer with $K=4096$

that is looking at some input volume of size $7\times7\times512$ can be equivalently expressed as a CONV layer with $F=7,P=0,S=1,K=4096$. In other words, we are setting the filter size to be exactly the size of the input volume, and hence the output will simply be $1\times1\times4096$

- since only a single depth column "fits" across the input volume, giving identical result as the initial FC layer.

FC->CONV conversion. Of these two conversions, the ability to convert an FC layer to a CONV layer is particularly useful in practice. Consider a ConvNet architecture that takes a 224x224x3 image, and then uses a series of CONV layers and POOL layers to reduce the image to an activations volume of size 7x7x512 (in an *AlexNet* architecture that we'll see later, this is done by use of 5 pooling layers that downsample the input spatially by a factor of two each time, making the final spatial size 224/2/2/2/2/2 = 7). From there, an AlexNet uses two FC layers of size 4096 and finally the last FC layers with 1000 neurons that compute the class scores. We can convert each of these three FC layers to CONV layers as described above:

- Replace the first FC layer that looks at [7x7x512] volume with a CONV layer that uses filter size $F=7$

¬ , giving output volume [1x1x4096].
¬ Replace the second FC layer with a CONV layer that uses filter size $F=1$
¬ , giving output volume [1x1x4096]
¬ Replace the last FC layer similarly, with $F=1$

- , giving final output [1x1x1000]

Each of these conversions could in practice involve manipulating (e.g. reshaping) the weight matrix W

in each FC layer into CONV layer filters. It turns out that this conversion allows us to "slide" the original ConvNet very efficiently across many spatial positions in a larger image, in a single forward pass.

For example, if 224x224 image gives a volume of size [7x7x512] - i.e. a reduction by 32, then forwarding an image of size 384x384 through the converted architecture would give the equivalent volume in size [12x12x512], since 384/32 = 12. Following through with the next 3 CONV layers that we just converted from FC layers would now give the final volume of size [6x6x1000], since (12 - 7)/1 + 1 = 6. Note that instead of a single vector of class scores of size [1x1x1000], we're now getting an entire 6x6 array of class scores across the 384x384 image.

Evaluating the original ConvNet (with FC layers) independently across 224x224 crops of the 384x384 image in strides of 32 pixels gives an identical result to forwarding the converted ConvNet one time.

Naturally, forwarding the converted ConvNet a single time is much more efficient than iterating the original ConvNet over all those 36 locations, since the 36 evaluations share computation. This trick is often used in practice to get better performance, where for example, it is common to resize an image to make it bigger, use a converted ConvNet to evaluate the class scores at many spatial positions and then average the class scores.

Lastly, what if we wanted to efficiently apply the original ConvNet over the image but at a stride smaller than 32 pixels? We could achieve this with multiple forward passes. For example, note that if we wanted to use a stride of 16 pixels we could do so by combining the volumes received by forwarding the converted ConvNet twice: First over the original image

and second over the image but with the image shifted spatially by 16 pixels along both width and height.

- An IPython Notebook on Net Surgery shows how to perform the conversion in practice, in code (using Caffe)

ConvNet Architectures

We have seen that Convolutional Networks are commonly made up of only three layer types: CONV, POOL (we assume Max pool unless stated otherwise) and FC (short for fully-connected). We will also explicitly write the RELU activation function as a layer, which applies elementwise non-linearity. In this section we discuss how these are commonly stacked together to form entire ConvNets.

Layer Patterns

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

```
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC
```

where the `*` indicates repetition, and the `POOL?` indicates an optional pooling layer. Moreover, `N >= 0` (and usually `N <= 3`), `M >= 0`, `K >= 0` (and usually `K < 3`). For example, here are some common ConvNet architectures you may see that follow this pattern:

- `INPUT -> FC`, implements a linear classifier. Here `N = M = K = 0`.
- `INPUT -> CONV -> RELU -> FC`
- `INPUT -> [CONV -> RELU -> POOL]*2 -> FC -> RELU -> FC`. Here we see that there is a single CONV layer between every POOL layer.
- `INPUT -> [CONV -> RELU -> CONV -> RELU -> POOL]*3 -> [FC -> RELU]*2 -> FC` Here we see two CONV layers stacked before every POOL layer. This is generally a good idea for larger and deeper networks, because multiple stacked CONV layers can develop more complex features of the input volume before the destructive pooling operation.

Prefer a stack of small filter CONV to one large receptive field CONV layer. Suppose that you stack three 3x3 CONV layers on top of each other (with non-linearities in between, of course). In this arrangement, each neuron on the first CONV layer has a 3x3 view of the input volume. A neuron on the second CONV layer has a 3x3 view of the first CONV layer, and hence by extension a 5x5 view of the input volume. Similarly, a neuron on the third CONV layer has a 3x3 view of the 2nd CONV layer, and hence a 7x7 view of the input volume. Suppose that instead of these three layers of 3x3 CONV, we only wanted to use a single CONV layer with 7x7 receptive fields. These neurons would have a receptive field size of the input volume that is identical in spatial extent (7x7), but with several disadvantages. First, the neurons would be computing a linear function over the input, while the three stacks of CONV layers contain

non-linearities that make their features more expressive. Second, if we suppose that all the volumes have C

channels, then it can be seen that the single 7x7 CONV layer would contain $C \times (7 \times 7 \times C) = 49C^2$ parameters, while the three 3x3 CONV layers would only contain $3 \times (C \times (3 \times 3 \times C)) = 27C^2$

parameters. Intuitively, stacking CONV layers with tiny filters as opposed to having one CONV layer with big filters allows us to express more powerful features of the input, and with fewer parameters. As a practical disadvantage, we might need more memory to hold all the intermediate CONV layer results if we plan to do backpropagation.

Recent departures. It should be noted that the conventional paradigm of a linear list of layers has recently been challenged, in Google's Inception architectures and also in current (state of the art) Residual Networks from Microsoft Research Asia. Both of these (see details below in case studies section) feature more intricate and different connectivity structures.

In practice: use whatever works best on ImageNet. If you're feeling a bit of a fatigue in thinking about the architectural decisions, you'll be pleased to know that in 90% or more of applications you should not have to worry about these. I like to summarize this point as "*don't be a hero*": Instead of rolling your own architecture for a problem, you should look at whatever architecture currently works best on ImageNet, download a pretrained model and finetune it on your data. You should rarely ever have to train a ConvNet from scratch or design one from scratch. I also made this point at the Deep Learning school.

Layer Sizing Patterns

Until now we've omitted mentions of common hyperparameters used in each of the layers in a ConvNet. We will first state the common rules of thumb for sizing the architectures and then follow the rules with a discussion of the notation:

The **input layer** (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.

The **conv layers** should be using small filters (e.g. 3x3 or at most 5x5), using a stride of $S = 1$

, and crucially, padding the input volume with zeros in such way that the conv layer does not alter the spatial dimensions of the input. That is, when $F = 3$, then using $P = 1$ will retain the original size of the input. When $F = 5$, $P = 2$. For a general F, it can be seen that $P = (F - 1)/2$

preserves the input size. If you must use bigger filter sizes (such as 7x7 or so), it is only common to see this on the very first conv layer that is looking at the input image.

The **pool layers** are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2x2 receptive fields (i.e. $F=2$

), and with a stride of 2 (i.e. $S=2$

). Note that this discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height). Another slightly less common setting is to use 3x3 receptive fields with a stride of 2, but this makes. It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

Reducing sizing headaches. The scheme presented above is pleasing because all the CONV layers preserve the spatial size of their input, while the POOL layers alone are in charge of down-sampling the volumes spatially. In an alternative scheme where we use strides greater than 1 or don't zero-pad the input in CONV layers, we would have to very carefully keep track of the input volumes throughout the CNN architecture and make sure that all strides and filters "work out", and that the ConvNet architecture is nicely and symmetrically wired.

Why use stride of 1 in CONV? Smaller strides work better in practice. Additionally, as already mentioned stride 1 allows us to leave all spatial down-sampling to the POOL layers, with the CONV layers only transforming the input volume depth-wise.

Why use padding? In addition to the aforementioned benefit of keeping the spatial sizes constant after CONV, doing this actually improves performance. If the CONV layers were to not zero-pad the inputs and only perform valid convolutions, then the size of the volumes would reduce by a small amount after each CONV, and the information at the borders would be "washed away" too quickly.

Compromising based on memory constraints. In some cases (especially early in the ConvNet architectures), the amount of memory can build up very quickly with the rules of thumb presented above. For example, filtering a 224x224x3 image with three 3x3 CONV layers with 64 filters each and padding 1 would create three activation volumes of size [224x224x64]. This amounts to a total of about 10 million activations, or 72MB of memory (per image, for both activations and gradients). Since GPUs are often bottlenecked by memory, it may be necessary to compromise. In practice, people prefer to make the compromise at only the first CONV layer of the network. For example, one compromise might be to use a first CONV layer with filter sizes of 7x7 and stride of 2 (as seen in a ZF net). As another example, an AlexNet uses filter sizes of 11x11 and stride of 4.

Case studies

There are several architectures in the field of Convolutional Networks that have a name. The most common are:

- **LeNet**. The first successful applications of Convolutional Networks were developed by Yann LeCun in 1990's. Of these, the best known is the LeNet architecture that was used to read zip codes, digits, etc.
- **AlexNet**. The first work that popularized Convolutional Networks in Computer Vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton. The

AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to runner-up with 26% error). The Network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

- **ZF Net**. The ILSVRC 2013 winner was a Convolutional Network from Matthew Zeiler and Rob Fergus. It became known as the ZFNet (short for Zeiler & Fergus Net). It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.
- **GoogLeNet**. The ILSVRC 2014 winner was a Convolutional Network from Szegedy et al. from Google. Its main contribution was the development of an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M). Additionally, this paper uses Average Pooling instead of Fully Connected layers at the top of the ConvNet, eliminating a large amount of parameters that do not seem to matter much. There are also several followup versions to the GoogLeNet, most recently Inception-v4.
- **VGGNet**. The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. Their pretrained model is available for plug and play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it was since found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.
- **ResNet**. Residual Network developed by Kaiming He et al. was the winner of ILSVRC 2015. It features special *skip connections* and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. The reader is also referred to Kaiming's presentation (video, slides), and some recent experiments that reproduce these networks in Torch. ResNets are currently by far state of the art Convolutional Neural Network models and are the default choice for using ConvNets in practice (as of May 10, 2016). In particular, also see more recent developments that tweak the original architecture from Kaiming He et al. Identity Mappings in Deep Residual Networks (published March 2016).

VGGNet in detail. Lets break down the VGGNet in more detail as a case study. The whole VGGNet is composed of CONV layers that perform 3x3 convolutions with stride 1 and pad 1, and of POOL layers that perform 2x2 max pooling with stride 2 (and no padding). We can write out the size of the representation at each step of the processing and keep track of both the representation size and the total number of weights:

```
INPUT: [224x224x3]        memory:  224*224*3=150K   weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*3)*64 =
1,728
```

```
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M   weights: (3*3*64)*64 =
36,864
POOL2: [112x112x64]  memory:  112*112*64=800K   weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights: (3*3*64)*128
= 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=1.6M   weights:
(3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K   weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*128)*256 =
294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 =
589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K   weights: (3*3*256)*256 =
589,824
POOL2: [28x28x256]  memory:  28*28*256=200K   weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*256)*512 =
1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 =
2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K   weights: (3*3*512)*512 =
2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K   weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 =
2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 =
2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K   weights: (3*3*512)*512 =
2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

As is common with Convolutional Networks, notice that most of the memory (and also compute time) is used in the early CONV layers, and that most of the parameters are in the last FC layers. In this particular case, the first FC layer contains 100M weights, out of a total of 140M.

Computational Considerations

The largest bottleneck to be aware of when constructing ConvNet architectures is the memory bottleneck. Many modern GPUs have a limit of 3/4/6GB memory, with the best GPUs having about 12GB of memory. There are three major sources of memory to keep track of:

- From the intermediate volume sizes: These are the raw number of **activations** at every layer of the ConvNet, and also their gradients (of equal size). Usually, most of the activations are on the earlier layers of a ConvNet (i.e. first Conv Layers). These are kept around because they are needed for backpropagation, but a clever implementation that runs a ConvNet only at test time could in principle reduce this by a huge amount, by only storing the current activations at any layer and discarding the previous activations on layers below.
- From the parameter sizes: These are the numbers that hold the network **parameters**, their gradients during backpropagation, and commonly also a step cache if the

optimization is using momentum, Adagrad, or RMSProp. Therefore, the memory to store the parameter vector alone must usually be multiplied by a factor of at least 3 or so.

- Every ConvNet implementation has to maintain **miscellaneous** memory, such as the image data batches, perhaps their augmented versions, etc.

Once you have a rough estimate of the total number of values (for activations, gradients, and misc), the number should be converted to size in GB. Take the number of values, multiply by 4 to get the raw number of bytes (since every floating point is 4 bytes, or maybe by 8 for double precision), and then divide by 1024 multiple times to get the amount of memory in KB, MB, and finally GB. If your network doesn't fit, a common heuristic to "make it fit" is to decrease the batch size, since most of the memory is usually consumed by the activations.

Convolutional Neural Networks

Overview

Convolutional neural networks (CNNs) are a class of artificial neural networks. CNNs are widely used in image recognition and classification. Like regular neural networks, a CNN consists of multiple layers and a number of neurons. CNNs are designed to take image data as input. This assumption allows CNNs to use structural information so that they are more computationally efficient and use less memory.

Layers in CNNs

A typical CNN consists of a number of different layers. Each type of layer has its own specific properties and functionalities. The main layer types are as follows:

Input Layer

The input layer stores the raw pixel values of the image to be classified. For example, each image in the CIFAR-10 data set has a height of 32, width of 32, and a depth of three color channels (blue, green, and red).

Convolution Layer

Convolution layers apply convolution operations to the input. They also compute the output of neurons that are connected to local regions in the input. The convolution in CNNs mirrors the response of an individual neuron to visual stimuli. Convolution layers do the heavy work of CNNs and are computationally expensive components.

Convolution layers use filters to measure how closely paths of input resemble features. A filter is a vector of weights that are learned during training. Intuitively, a CNN learns filters that are proficient at detecting certain types of visual features, such as a straight line, a semi-circle, and so on. Filters usually have small width and height. Filters also share the same depth as the input. For example, given a filter of size 5 x 5 x 3, the filter width is 5 pixels and the filter height is 5 pixels. The filter depth is 3, for the 3 color channels of blue, green, and red.

When building a CNN using SAS Deep Learning actions, the user does not need to specify padding for the convolution (or pooling) layer. Padding is automatically calculated so that, when using a stride of 1, the image size does not change: the size after convolution is exactly the size that it was before convolution. For a stride of 2, padding is calculated so that the size of the output image size is one half the size of the input image. For a stride of 3, padding adjusts the size of the output image to one third of the size of the input image, and so on.

The convolution layer computes the output of neurons that are connected to local regions in the input. Each computes a dot product between their weights and a small region where they are connected to the input volume. If you decide to use 12 filters, a volume might result such as 32 x 32 x 12.

Pooling Layer

The pooling layer simplifies the output from a convolution layer by progressively reducing the spatial dimensions (and hence, computational parameters) of the image representation. Pooling layers are usually placed between successive convolution layers. A pooling layer conducts a downsampling operation along the height and width dimensions to reduce the size of the image representation. Pooling significantly reduces the number of parameters to be analyzed, which in turn reduces the amount of computation. Pooling layers also help alleviate overfitting, a common challenge with CNNs.

Max pooling is the most popular pooling scheme. Max pooling applies a moving window across a two-dimensional space to choose the maximum value. For example, given a 4 x 4 input matrix, the resulting matrix becomes 2 x 2 after applying a 2 x 2 filter with a stride of 2.

Pooling layers can perform other functions besides max pooling. Other pooling schemes include average pooling and L-2 norm pooling.

Output Layer

The output layer is a fully connected layer that is associated with a particular loss function, such as categorical cross-entropy. The loss function computes the error in prediction. Neurons in the output layer connect to all of the activations in the previous layer, just like a fully connected layer in a regular neural network.

CNNs transform original images to the final class scores layer by layer. The convolution and output layers contain parameters, but the pooling layers do not. Optimization techniques such as stochastic gradient descent or L-BFGS can be used to train and compute the parameters in convolution and fully connected layers.

Batch Normalization Layer

Batch Normalization is an optimization method that often results in faster convergence (in terms of number of epochs required) and better generalization of deep neural networks. The February 2015 paper, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, by Sergey Ioffe and Christian

Szegedy describes the rationale and implementation of batch normalization in deep neural networks.

In the SAS Deep Learning actions, batch normalization is implemented as a separate layer type. The batch normalization layer is typically inserted after a convolution or pooling layer. But batch normalization layers can be placed anywhere after the input layer and prior to the output layer. For more information, see Batch Normalization.

SAS Deep Learning supports all of the layer types listed above, as well as the following:

Residual Layer

Deep CNNs developed for image classification exploit the ability to enrich features by increasing the number of stacked layers. However, adding more layers is not always a better approach: beyond some levels, deeper networks can exhibit radical growth or disappearance of gradients. Normalization techniques such as batch normalization have been successful in damping the wild gradient swings, largely addressing the convergence problems. However, network accuracy degradation often becomes an issue as network depths increase: accuracies can saturate with increasing depth, until a rapid degradation of accuracies occurs.

The degradations are not due to overfitting. Beyond some point, adding layers to a sufficiently deep model leads to degradation of training accuracy and higher training error. This degradation can be approached by the use of a deep residual learning layer. The deep residual learning layer uses stacked nonlinear layers to fit an optimized residual mapping. It uses feedforward neural networks that use shortcut connections that skip one or more layers to perform identity mapping.

Concatenation (Concat) Layer

The concatenation layer combines multiple inputs by concatenating them along an axis.

Keypoints Layer

The keypoints layer performs keypoints detection in deep learning networks. The SAS Deep Learning keypoints layer is designed to train a regression model with multiple interval targets. For more information, see Keypoints Layer.

Object Detection Layer

The object detection layer uses YOLO1 and YOLOv2 algorithms to detect the bounding boxes and to classify the objects within each box. For more information, see Object Detection .

Projection Layer

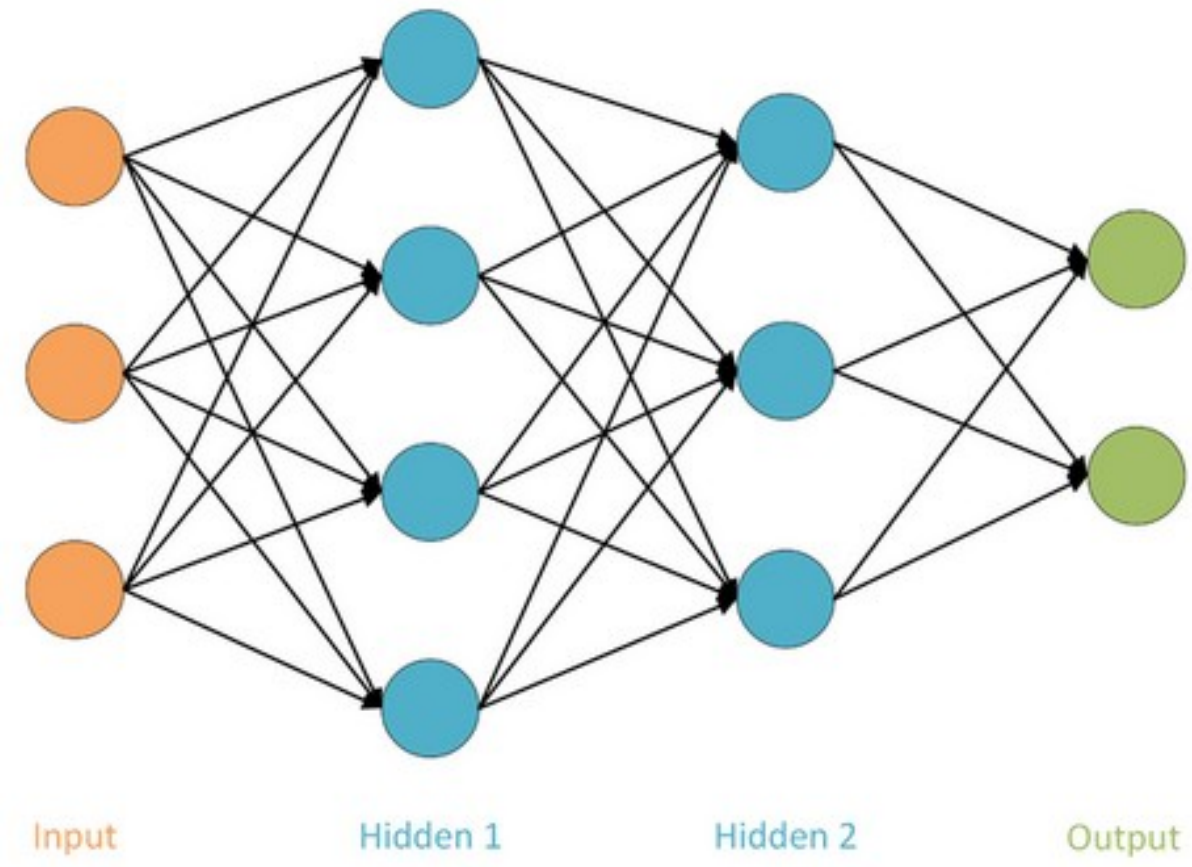
The projection layer is a useful deep learning tool for customizing semantic word embeddings in text analytics tasks. For more information, see Projection Layer.

Neurons in CNNs

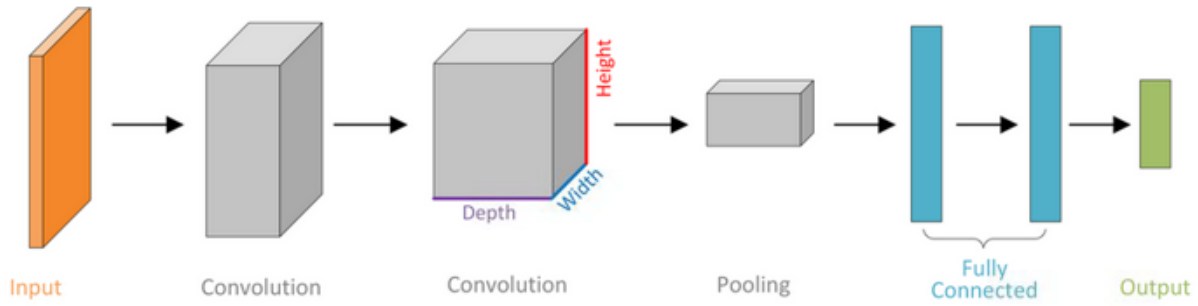
For a fully connected regular neural network, the number of neurons can be large. Using fully connected architecture, the number of weights can become enormous, and require exceptional amounts of memory and computing resources in order to train the network. CNNs can minimize the number of neurons by taking advantage of the architecture of image inputs. In contrast to fully connected networks, CNNs connect each neuron only to a local region of the input. The dimensions of this local region are defined by the filter size. Using a smaller local region drastically reduces the number of neuron connections and weights when compared to fully connected networks.

For example, consider the input images in Krizhevsky, Nair, and Hinton's CIFAR-10 data set. Each image has an input volume of 32 x 32 x 3 (width in pixels, height in pixels, and depth in channels, respectively). If a CNN has a filter with dimensions 5 x 5 x 3, then each neuron in the convolution layer has only (5*5*3) = 75 weights. It follows that the final output layer for the CIFAR-10 data using CNN would have dimensions of 1 x 1 x 10, because at the end of the CNN architecture, the full image is reduced into a single vector of class scores.

Here is a visual representation of a regular 3-layer neural network:



Here is a visual representation of a CNN model:



Overview of Convolution, Filter, and Stride

Convolution is a tool that neural networks use to determine features that can be used to identify subjects within images for classification. A series of convolution layers detects low-level features (such as curves and edges) and accumulates them into more abstract concepts that can be classified. Convolution is the process of analyzing an input image, represented as an array of pixel values, by iteratively sliding (convolving) a smaller matrix (*filter*) of the same depth across the input image, while performing elementwise calculations to compute an output matrix of image features. The *stride* metric quantifies the distance (for example, number of pixels) that the filter traverses across the input matrix between each convolution stride. The larger the stride value used during convolution, the smaller the size of the resulting output image feature matrix. In practice, most convolutional layers in a neural network use a step size of 1, in order to produce an output (feature) matrix that is as close as possible to the same size as the input matrix. It is generally good for convolution layers to use input and output matrices that are the same size (when possible).

As the filter convolves around the input space, elementwise calculations determine the dot product between the filter and the input image during each step. Each successive calculation between the convolving filter and the input matrix becomes a new feature value in the growing output matrix. After the filter convolves across the entire input matrix, the completed output matrix is known as a *feature map*. Feature maps are also called activation maps, because an element of the input volume is said to activate the filter when an elementwise calculation results in an affirmative result. In other words, as the filter for a layer convolves around the input image, the filter activates (by computing high values) when the feature that the filter is looking for appears in the input volume.

Convolution, Stride, and Padding on CPU

The following example, which uses a 1-dimensional input space, is simple, but establishes basic convolution mechanics that also apply to 2- and 3-dimensional input spaces and filters. The simplified input space is useful for understanding concepts of stride and padding.

The stride is the distance metric for the convolving filter matrix, in pixel units. Most convolutional layers use a stride of 1. A stride of 1 has high information density and is most likely to produce an output matrix close in size to the input matrix.

As the size of the convolving stride increases, the proportions of the output matrix decrease inversely. A stride of 1 produces an output matrix that is near in size to the input matrix. And a stride of 2 produces output matrices approximately half the size of the input space; a stride of 3 produces output matrices approximately a third the size of the input matrix; and so on.

Strides of 2 are commonly used in pooling layers, where the smaller matrices condense information in the pooling process. It is easy to see how, for a given input matrix, the size of the stride directly affects the geometry of the output matrix.

Suppose you have an input space that is defined in one dimension. An example is (1 x 7 x 1). The seven columns in the input space represent seven pixel values, as follows: `[2, 4, 6, 1, 5, 3, 2]`.

Now, suppose that you want to convolve this input space using a (1 x 3) filter to perform some sort of feature detection. Suppose the filter has the following values: `[2, -1, 0.1]`.

Where should the convolving filter be positioned at the start? You can choose to start the convolution by aligning the left-most elements of the input space and the filter, as shown.

```
[2,  4,  6,  1, 5, 3, 2]
[2, -1, 0.1]
```

In this position, the dot product of the filter and the matrix is: `[2, -1, 0.1] * [2, 4, 6] = [(2*2), (-1*4), (0.1*6)] = [4, -4, 0.6]`. The result matrix `[4, -4, 0.6]` represents the values for the first pixel in the output array.

Now, you slide the filter to the right, by a stride equal to 1 pixel.

```
[2,  4,  6,  1,  5, 3, 2]
    [2, -1, 0.1]
```

Following the stride, the next dot product of the convolution is calculated: `[2, -1, 0.1] * [4, 6, 1] = [(2*4), (-1*6), (0.1*1)] = [8, -6, 0.1]`. The result `[8, -6, 0.1]` represents the values of the second pixel in the output matrix.

It follows that convolving the filter from the far left edge of the input space to the far right edge of the input space yields the following array of five output pixel values:

- `[2, -1, 0.1] * [2, 4, 6] = [4, -4, 0.6]` output pixel 1
- `[2, -1, 0.1] * [4, 6, 1] = [8, -6, 0.1]` output pixel 2
- `[2, -1, 0.1] * [6, 1, 5] = [12, -1, 0.5]` output pixel 3
- `[2, -1, 0.1] * [1, 5, 3] = [2, -5, 0.3]` output pixel 4
- `[2, -1, 0.1] * [5, 3, 2] = [10, -3, 0.2]` output pixel 5

Recall that the input space is a 1 x 7 image. If you convolve a 1 x 3 filter across a 1 x 7 input space, the matrix operations result in a 1 x 5 output space. Remember that it is desirable for the input and output images in a convolution layer to be the same size whenever possible. Differently sized input and output images can be problematic for some algorithms. You need a way to generate output arrays that are the same size as the input arrays, given the size of the input space, the filter, and the stride.

You can use padding to meet these requirements. If you add a zero-valued pixel to both the left and right sides of the 1 x 7 input image matrix, you create an augmented, or "padded" 1 x 9 matrix. The padded 1 x 9 matrix still retains the original input image pixel values and geometry, but when convolved using a 1 x 3 filter, results in a 1 x 7 feature matrix that matches the dimensions of the original input matrix. The zero-value pixels facilitate the matrix computations that result in an output array with the desired dimensions, without disturbing the original input matrix values.

Now the augmented input matrix has nine pixel values as follows:

`[0, 2, 4, 6, 1, 5, 3, 2, 0]`.

Convolving the 1 x 3 filter from left to right across the 1 x 9 input matrix results in the following output:

- `[2, -1, 0.1] * [0, 2, 4] = [0, -2, 0.4]` output pixel 1
- `[2, -1, 0.1] * [2, 4, 6] = [4, -4, 0.6]` output pixel 2
- `[2, -1, 0.1] * [4, 6, 1] = [8, -6, 0.1]` output pixel 3
- `[2, -1, 0.1] * [6, 1, 5] = [12, -1, 0.5]` output pixel 4
- `[2, -1, 0.1] * [1, 5, 3] = [2, -5, 0.3]` output pixel 5
- `[2, -1, 0.1] * [5, 3, 2] = [10, -3, 0.2]` output pixel 6
- `[2, -1, 0.1] * [3, 2, 0] = [6, -2, 0]` output pixel 7

Now we have input and output images of the same size.

However, suppose that you must use a new filter of dimensions (1 x 4) `[2, -1, 0.1, 5]`, instead of (1 x 3) `[2, -1, 0.1]`?

If you want to produce input and output matrices that are still the same size, you are going to need to add a third padding pixel to the input space. However, when your padding requires an odd number of elements, applying them to the matrix in a symmetrical manner becomes problematic.

Three padding pixels cannot be evenly divided between left and right sides of a matrix. One side of the matrix must have two padding pixels, while the opposing side will have only one. Suppose that you choose to pad the input matrix with one pixel on the left side, and two pixels on the right side, yielding an input image with ten pixels, as follows: `[0, 2, 4, 6, 1, 5, 3, 2, 0, 0]`

Convolving the 1 x 3 filter from left to right across the 1 x 10 input matrix results in the following output:

- `[2, -1, 0.1, 5] * [0, 2, 4, 6] = [0, -2, 0.4, 30]` output pixel 1

- `[2, -1, 0.1, 5] * [2, 4, 6, 1] = [4, -4, 0.6, 5]` output pixel 2
- `[2, -1, 0.1, 5] * [4, 6, 1, 5] = [8, -6, 0.1, 25]` output pixel 3
- `[2, -1, 0.1, 5] * [6, 1, 5, 3] = [12, -1, 0.5, 15]` output pixel 4
- `[2, -1, 0.1, 5] * [1, 5, 3, 2] = [2, -5, 0.3, 10]` output pixel 5
- `[2, -1, 0.1, 5] * [5, 3, 2, 0] = [10, -3, 0.2, 0]` output pixel 6
- `[2, -1, 0.1, 5] * [3, 2, 0, 0] = [6, -2, 0, 0]` output pixel 7

What is the result if you have to apply padding asymmetrically, or unequally to an input matrix in a convolutional network layer? The answer depends on whether you have configured your network session to be processed solely by CPU, or if you have configured GPU resources to share the processing tasks.

If you have configured your SAS Deep Learning session to use only CPU processing, using unequal padding in your input space is a viable approach to generating an output matrix of a specific size, given the dimensions of the input and filter matrices.

On the other hand, if your session implements GPU to share processing loads with the CPU, you will need to make further adjustments. The following section, Convolution, Stride, and Padding on GPU discusses how to make these adjustments.

Convolution, Stride, and Padding on GPU

Ideally, neural networking algorithms should be able to operate in the same manner, whether being processed by CPU or GPU. SAS Deep Learning researchers created and tested the algorithm code for CNNs, and created a CPU architecture where convolutional layers in neural networks can perform matrix operations on arrays that have uneven or asymmetric padding.

In order to implement GPU optimization of computing operations, SAS licenses the cuDNN GPU processing library from NVIDIA Corporation. NVIDIA's cuDNN library is not an open-source entity, and SAS cannot optimize, enhance, or modify the contents. The NVIDIA cuDNN library currently does not support all of the convolution and pooling layer options that exist in SAS Deep Learning tools. If you add cuDNN GPU support to a networking task that was formerly performed exclusively on CPU, you might find small differences in the analytical results.

For example, the SAS code that performs convolutional and pooling layer calculations on the CPU is not identical to the NVIDIA cuDNN GPU library code. Padding is processed differently on the SAS CPU and NVIDIA GPU platforms. The cuDNN library does not support asymmetrical padding that is added to input matrices. If any of the arrays are padded, the GPU calculations require symmetric, equal padding proportions, or an error results.

For cases where GPU processing is desired and unequal amounts of padding are required to match input and output matrix dimensions, SAS Deep Learning created a binary GPU configuration setting called forceEqualPadding, which can be used with the NVIDIA cuDNN

library. The forceEqualPadding setting is active only when GPU processing is enabled. The default (and undeclared) state of forceEqualPadding is False. When you set forceEqualPadding=True, SAS Deep Learning tools scan input matrix padding geometries to ensure that uneven padding distributions do not exist. If uneven padding is found, the forceEqualPadding setting automatically adds padding to the unbalanced side of the matrix until the symmetry required by the NVIDIA GPU processing library is achieved.

Note: As a best practice, SAS recommends that filters for convolutional layers should have only *odd* numbers of elements. Filter matrices that have an even number of elements have not analytically been shown to introduce any additional information value, but they can result in additional processing difficulties. Whenever possible, the filters that you use to convolve input matrices should have an odd number of elements, instead of an even number.

How much does the additional padding that is inserted by the forceEqualPadding setting interfere with the network's analytic qualitative results? It is a compromise. In these cases, the GPU-based cuDNN cannot generate an output image the same size as the input image because it does not support uneven padding , but the CPU-based Deep Learning algorithm can (because it does support uneven padding).

Consider a scenario where you have an odd image size, an even filter width, and a stride of 1:

```
Input = [1, 2, 3, 4, 5]
Filter = [1, 2]
Step = 1
```

Convolving a 1 x 5 input matrix with a 1 x 2 filter produces a 1 x 4 output matrix. You can use the corrective values `padLeft=0` and `padRight=1` to create the augmented (padded) input matrix `[1, 2, 3, 4, 5, 0]`.

The result of convolving the padded matrix with the 1 x 2 filter using CPU is as follows:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 5] = [4, 10]` output pixel 4
- `[1, 2] * [5, 0] = [5, 0]` output pixel 5

Now, look at the same scenario, but using GPU via the cuDNN library. The cuDNN library requires equal padding. If you use `padLeft=0` and `padRight=0`, then cuDNN generates only a 1 x 4 output matrix:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2

- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 5] = [4, 10]` output pixel 4

To satisfy the requirements of cuDNN, if you specify `padLeft = padRight = 1`, then the input matrix becomes 1 x 7, as `[0, 1, 2, 3, 4, 5, 0]`. A 1 x 7 input matrix convolved by a 1 x 2 filter generates a 1 x 6 output matrix as follows:

- `[1, 2] * [0, 1] = [0, 2]` output pixel 1
- `[1, 2] * [1, 2] = [1, 4]` output pixel 2
- `[1, 2] * [2, 3] = [2, 6]` output pixel 3
- `[1, 2] * [3, 4] = [3, 8]` output pixel 4
- `[1, 2] * [4, 5] = [4, 10]` output pixel 5
- `[1, 2] * [5, 0] = [5, 0]` output pixel 6

Likewise, there is a similar difference for cases where you have an even image size and an even filter size.

Consider a scenario where you have an image size of 1 x 4, a filter size of 1 x 2, and a stride of 1:

```
Input = [1, 2, 3, 4]
Filter = [1, 2]
Step = 1
```

Convolving a 1 x 4 input matrix with a 1 x 2 filter produces a 1 x 3 output matrix. The corrective values `padLeft=0` and `padRight=1` are used to create the augmented (padded) input matrix `[1, 2, 3, 4, 0]`.

The result of convolving the padded 1 x 5 matrix with the 1 x 2 filter (using CPU) generates a 1 x 4 output matrix, which is the same size as the original input matrix:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 0] = [4, 0]` output pixel 4

If you want to use GPU, and tell cuDNN that `padLeft = padRight = 0`, the output matrix is only 1 x 3:

- `[1, 2] * [1, 2] = [1, 4]` output pixel 1

- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3

On the other hand, if you want to use GPU, and tell cuDNN that `padLeft = padRight = 1`, the augmented input matrix is 1 x 6 `[0, 1, 2, 3, 4, 0]`, and the resulting output matrix is 1 x 5, which does not match the original input matrix dimensions:

- `[1, 2] * [0, 1] = [0, 2]` output pixel 1
- `[1, 2] * [1, 2] = [1, 4]` output pixel 1
- `[1, 2] * [2, 3] = [2, 6]` output pixel 2
- `[1, 2] * [3, 4] = [3, 8]` output pixel 3
- `[1, 2] * [4, 0] = [4, 0]` output pixel 4

Of course, if you use the CPU setting for forceEqualPadding=True in cases like these, then there will be no difference between the results for CPU processing and GPU processing.

If this is the case, then why not always set forceEqualPadding=True? The answer is that forceEqualPadding=True requires a compromise. As shown by the examples above, the downside of setting forceEqualPadding=True is that for either CPU-enabled or GPU-enabled processing, the resulting output image matrices are not the same size as the input image matrix. The output image will always be one dimension larger than the input image.

Overview of Pooling, Window, and Stride

Pooling layers are a tool that work together with convolution layers in a deep neural network to perform visual object recognition tasks. Convolution uses a sliding filter to map a region of an input image to a corresponding feature map. Successive network convolution layers seek to detect higher-level features from the low-level building blocks. In deep neural models, pooling layers are generally located between successive convolution layers. Pooling layers perform downsampling operations that reduce the spatial dimensions (but not the depth) of the parameter space. Pooling also helps control model overfitting, while removing sources of translational invariance (such as skew or lateral shift) that can interfere with higher-level feature detection. If computational details (such as high activation values) indicate that a specific feature exists in the original input space, pooling seeks to determine the location of individual detected features relative to the other features, rather than seeking to determine the exact location of the feature within the input space.

Pooling layers use various criteria to perform different pooling functions. The most popular pooling layer type is a maxpooling layer. A maxpooling layer uses a small matrix called a *window* (typically sized 2 x 2) with a stride of 2 to convolve around the pooling input volume. In each iteration, the window performs elementwise calculations and generates the maximum value for every subregion that it convolves around.

User-Defined Padding for the Convolution and Pooling Layers

Based on the filter size and stride for convolution layers, and on the window size and stride for pooling layers, you might need zero padding around the input to these layers in order to make these operations mathematically consistent. By default, the SAS Deep Learning toolkit calculates padding automatically. If the stride is 1, the output size of the layer will be the same size as the input. If the stride is 2, the output size of the layer will be approximately one half the size of the input, and so on. For strides greater than 1, different frameworks might generate output sizes that are not equal to 1, depending on how the padding is calculated. If you are trying to closely match results from an imported model, you want the convolution and pooling layers in your SAS Deep Learning model to have the same output size as the corresponding layers in the imported model.

The SAS Deep Learning toolkit enables you to override default padding calculations for convolution and pooling layers, and explicitly specify you own padding for convolution and pooling layers. When you use the `AddLayer` action to define a convolution or pooling layer, you can control padding by specifying your own parameter values. Instead of the default automatic padding calculations, you use the parameters `padding=`, `paddingHeight=` and `paddingWidth=` to configure padding for mathematically consistent operations. If you want to specify the same padding value for both horizontal and vertical dimensions, use the `padding=` keyword. If you want to specify different padding values for horizontal and vertical dimensions, then you use the `paddingWidth=` and `paddingHeight=` keywords. If you use the `padding=` keyword, you cannot also use the `paddingWidth=` or `paddingHeight=` keywords. You can specify just `paddingWidth=` or just `paddingHeight=`, and SAS Deep Learning tools will calculate the normal default value for the remaining dimension.

Convolution Layer Padding Calculations

When specifying individual padding dimensions for a convolution layer, the output sizes of the layer are as follows:

If you are specifying the `paddingWidth=` for a convolution layer, the output width of the layer will be as follows:

If you are specifying the `paddingHeight=` for a convolution layer, the output height of the layer will be as follows:

Note: The padding value specified by these keywords represents the amount of padding on one side of the input. An equal amount will be applied to the opposite side. Therefore, the total padding for each dimension is twice the specified padding. This is why you see the constant of 2 in the formulas above.

Pooling Layer Padding Calculations

When specifying individual padding dimensions for a pooling layer, the output sizes of the layer are as follows:

If you are specifying the `paddingWidth=` for a pooling layer, the output width of the layer will be as follows:

When specifying `paddingHeight=` for a pooling layer, the output height of the layer will be as follows:

Note: The padding value specified by these keywords represents the amount of padding on one side of the input. An equal amount will be applied to the opposite side. Therefore, the total padding for each dimension is twice the specified padding. This is why you see the constant of 2 in the formulas above.

Example: Padding for a Pooling Layer in a LeNet Model

The following SAS CASL code fragment uses the `AddLayer` command to add a pooling layer to a LeNet model. In the code, the `paddingWidth=` keyword overrides default padding width calculations, and explicitly sets the padding width value to 1.

```
AddLayer/model='LeNet'
                name="pool1"
                layer={type='pooling'
                       width=2
                       height=2
                       stride=2
                       pool='max'
                       paddingWidth=1
                       }
                srcLayers={"conv1"}
                replace=1
                ;
```

Differences between Pooling and Convolution Padding

Convolution layers and pooling layers use different approaches to padding. Pooling, in essence, does not calculate padding.

Suppose you have a pooling layer that has an input space that is defined in one dimension. An example is (1 x 7 x 1). The seven columns in the input space represent seven pixel values. Suppose that the 1 x 7 pooling layer has the same input matrix values that was used in a previous example: `[2, 4, 6, 1, 5, 3, 2]`.

Now, suppose that you want to convolve this input space using a (1 x 3) window and a stride of 1 to perform maxpooling. The left edge of the pooling window always starts at the left edge of the input matrix.

```
[2,  4,  6,  1, 5, 3, 2]
[          ]
```

Instead of containing values like a convolutional filter matrix, the pooling window matrix elements are unpopulated placeholders. As the pooling window convolves across the input space, instead of calculating the dot product of the filter and the input matrix, the array of empty placeholders in the pooling window assume the values of the corresponding input matrix elements. Thus populated, the maxpool function selects the largest value in the pooling window array as the output matrix value for each stride.

Performing maxpooling on the input matrix above by convolving a 1 x 3 pooling window would produce the following results:

- `Max{[2, 4, 6]} = 6` output pixel 1
- `Max{[4, 6, 1]} = 6` output pixel 2
- `Max{[6, 1, 5]} = 6` output pixel 3
- `Max{[1, 5, 3]} = 5` output pixel 4
- `Max{[5, 3, 2]} = 5` output pixel 5
- `Max{[3, 2]} = 3` output pixel 6
- `Max{[2]} = 2` output pixel 7

As the example demonstrates, any padding-like accommodations for pooling layer computations occur on the right side of the input matrix only. The accommodations are not exactly padding because there are never any zero pixel-sized elements created in the input matrix to accommodate the pooling window matrix. Instead, the pooling window effectively becomes smaller. (See output pixels 6 and 7 in the example above.)

The pooling calculations seem to operate using the same methods, whether a session is optimized for GPU using the NVidia cuDNN library, or using CPU processing only.

As mentioned before, pooling layers use various criteria to perform different pooling functions. For example, instead of using the maxpool criteria to select the maximum value in the pooling window as the output matrix value, you could use the mean pool criteria to calculate the mean value of the elements in the pooling window as the output matrix value. In the example above, the first five output matrix values would be the sum of the window values in each stride divided by three. For the sixth output matrix value, the mean would be calculated as the sum of the window values divided by two. Similarly, the mean for the seventh output matrix value is simply equal to the solitary window element value.

These computation rules produce the same pooling layer results, whether computed by CPU or optimized for GPU. For pooling strides of 1, you always get an output matrix with the same dimensions as the input matrix. When using other strides, for pooling (or convolution), you should strive for output images that are as close as possible to the input image size and stride.

Depending on whether the input image size is odd or even, and whether the stride is odd or even, you might have to compromise by generating a results matrix that is off by 1 pixel.

Common CNN Architectures

Here are some of the most commonly referred to convolutional neural networks that have named architectures:

LeNet

> The first successful applications of convolutional networks were developed by Yann LeCun in the 1990s. Of these, the best known is the LeNet architecture. The LeNet architecture was used to read ZIP codes, digits, and so on.

AlexNet

> The AlexNet was the first work that popularized convolutional networks in computer vision, developed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. The AlexNet was submitted to the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012 and significantly outperformed the second runner-up. For example, AlexNet had a Top 5 error of 16%, whereas the runner-up had a Top 5 error of 26%. AlexNet has a very similar architecture to LeNet, but is deeper, bigger, and features convolutional layers stacked on top of each other. This is significant because beforehand, it was common for architectures to have only a single convolutional layer, which was always immediately followed by a pooling layer.

ZF Net

> Matthew Zeiler and Rob Fergus won the 2013 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) using a convolutional network architecture called the ZF Net. It improved performance over AlexNet by tuning the architecture hyperparameters, by expanding the size of the middle convolutional layers and, by using a smaller stride and filter size on the first layer.

GoogLeNet

> The 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winner was a convolutional network from Szegedy et al. from Google. Its main contribution was the development of an inception module that dramatically reduced the number of parameters in the network (4M, compared to 60M in AlexNet). This architecture uses average pooling instead of fully connected layers at the top of the convolutional network. This improvement eliminates a large number of parameters that do not seem to matter much.

VGGNet

The 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) runner-up was the network from Karen Simonyan and Andrew Zisserman. Their network was called VGGNet. VGGNet showed that the depth of a convolutional network is a critical component for good performance. Their final best network contains 16 convolutional, fully-connected layers with a homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. A pretrained VGGNet model is available for plug-and-play use in Caffe. A downside of the VGGNet is that it is more expensive to evaluate and uses a lot more memory and parameters (140M). Most of these parameters are in the first fully connected layer, and it has since been found that these fully connected layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

ResNet

ResNet is a residual network developed by Kaiming He et al. ResNet was the winner of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2015. ResNet features special skip connections and a strong use of batch normalization. The architecture is also notable because it does not have any fully connected layers at the end of the network. ResNets are considered by many (in 2016) to be state-of-the-art convolutional neural network models.

Batch Normalization

Overview of Batch Normalization

Batch Normalization is an optimization method that often results in faster convergence (in terms of number of epochs required) and better generalization of deep neural networks. The February 2015 paper, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," by Sergey Ioffe and Christian Szegedy describes the rationale and implementation of batch normalization in deep neural networks.

In the SAS Deep Learning actions, batch normalization is implemented as a separate layer type. The batch normalization layer is typically inserted after a convolution or pooling layer, but batch normalization layers can generally be placed after any layer except the input layer and prior to the output layer.

The layer preceding the batch normalization layer typically has `noBias=True` and `act='identity'`, and the activation function that would otherwise be used for the preceding layer is specified as the activation function for the batch normalization layer.

Batch Normalization Formula

The batch normalization operation normalizes a set of input data by performing a standardizing calculation to each piece of input data. The standardizing calculation subtracts the mean of the data, and then divides by the standard deviation. It then follows this calculation by multiplying the data by the value of a learned constant, and then adding the value of another learned constant.

Thus, the normalization formula is as follows:

Gamma and beta are learnable parameters. When training, the mean and standard deviation are computed over the current mini-batch of the training data set. When scoring, the mean and standard deviation used are those computed over the entire training data set. In addition, if the data from the source layer to the batch normalization layer is structured as feature maps (as in the convolution layer), then a separate mean, standard deviation, gamma, and beta is computed for each feature map. (That is, the mean of a feature map is the average value of ALL pixels in the feature map for each feature map in the mini-batch.) If the data from the source layer is not structured as feature maps (as in the fully connected layer), then a separate mean, standard deviation, gamma, and beta are computed for each neuron.

Source Layer Parameter Settings

In order for the batch normalization computations to conform to those described in Sergey Ioffe and Christian Szegedy's Batch Normalization research, the source layer should have settings of `act=identity` and `includeBias=False`. The activation function that would normally have been specified in the source layer should instead be specified on the batch normalization layer. If you do not configure your model to follow these option settings, the computation will still work, but it will not match the computation as described by Ioffe and Szegedy.

Computing Related Statistics When Scoring

The statistics used in the batch normalization formula (that is, the mean and standard deviation for each feature map or neuron) are computed during training, and then re-computed over each mini-batch. When scoring, however, it is more appropriate to use mean and standard deviation statistics that are computed over the entire training data set. In order to perform this task without making an extra pass through the data at the end of training, the SAS Deep Learning actions accumulate the sum(x) and sum(x^2) for each feature map or neuron over an entire epoch. Then those sums are used to compute the mean and variance over the entire training data set, using the standard mean and variance computing formulas.

Of course, these are not quite the true mean and standard deviation over the entire training data set. This is because the weights are adjusted after each mini-batch, which means that the sums are computed using different weights. However, in the last epoch of most training runs, those weights often do not change much when the neural network is converged well, or if the learning rate is small in SGD.Experiments have shown that if the training has converged significantly (to the point where the weights are no longer changing much), the resulting final means and standard deviation sums are reasonably close to the actual means and standard deviation calculated over the entire training data set using a fixed set of weights. Experiments have also shown that the accuracy of scoring is not affected by small variations in mean and standard deviation measurements.

Storing Trained Batch Normalization Parameters

When using SAS Deep Learning actions, the batch normalization parameters are saved as part of the weight table. The learnable batch normalization layer parameters, gamma and beta, for each feature map or neuron, are saved in the bias weight portion of each batch normalization

layer's weights. That is so that they will not be part of the L1 or L2 regularization computations. The batch normalization statistics (mean and standard deviation for each feature map or neuron) for each batch normalization layer are stored at the end of the weight table.

Batch Normalization and Source Layer Feature Maps

Historically, convolutional networks are used to perform image classification and related tasks, such as object detection. These tasks typically use two-dimensional images composed of discrete pixels as input observations. The outputs of each convolution layer are also two dimensional, but the output isn't quite an image anymore, because the source images have been transformed in non-linear ways. As a result, the outputs of convolutional layers are not referred to as images: they are called feature maps. The feature map nomenclature indicates that the convolution layer generates a collection of features extracted from the input image. For example, if a convolution layer contains 10 filters, the output would contain 10 feature maps.

In contrast, the output for a fully connected layer is simply a vector of numbers that is equal to the number of hidden neurons in the fully connected layer. The dimensions or "shape" of the input data don't really matter. The input for a fully connected layer is flattened: any sense of two dimensions at the input is lost. The loss of dimensional output means that a fully connected layer cannot extract the dimensional features that compose a feature map. Therefore, the fundamental difference: the output of a convolution layer does have feature maps, but the output of a fully connected layer does not. Even if the input dimensions for a convolution layer are one-dimensional (for example, 1 x 10), the convolution layer output is still considered to be a feature map.

Distinguishing whether the output from a layer type is a feature map or not becomes important when the layer in question is a source layer to a batch normalization layer. In cases like this, the formula for the batch normalization layer is different, depending on whether the source layer output contains feature maps.

In the case where the source layer to a batch normalization layer contains feature maps, the batch normalization layer computes statistics based on all of the pixels in each feature map, over all of the observations in a mini-batch. For example, suppose that your network is configured for a mini-batch size of 3, and the input to the batch normalization layer consists of two 5 x 5 feature maps. In this case, the batch normalization layer computes two means and two standard deviations. The first mean would be the mean of all the pixels in the first feature map for the first observation, the first feature map of the second observation, and the first feature map of the third observation. The second mean would be the mean of all of the pixels in the second feature map of the first observation, the second feature map of the second observation, and the second feature map of the third observation, and so on. Numerically, each mean would be the mean of (3 x 5 x 5) = 75 values.

In the case where the source layer to a batch normalization layer does not contain feature maps (for example, a fully connected layer), then the batch normalization layer computes statistics for each neuron in the input, rather than for each feature map in the input. For example, suppose that your network has a mini-batch size of 3, and the input to the batch normalization layer contains 50 neurons. In this case, the batch normalization layer would compute 50 means and 50 standard deviations. The first mean would be the mean of the first

neuron of the first observation, the first neuron of the second observation, and the first neuron of the third observation. The second mean would be the mean of the second neuron of the first observation, the second neuron of the second observation, and the second neuron of the third observation, and so on. Numerically, each mean would be the mean of 3 values. NVIDIA refers to this calculation as “per activation” mode.

In a SAS Deep Learning network, each layer type makes a decision to determine whether the source layer to a batch normalization layer contains feature maps. Most layers, such as pooling layers, just inherit what its source layer had. Convolution layers always have feature maps, and fully connected layers never have feature maps.

Batch Normalization with Multiple GPUs

When using multiple GPUs, efficient calculation of the batch normalization transform requires a modification to the original algorithm specified by Ioffe and Szegedy in Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. The algorithm specifies that during training, you must calculate the mean and standard deviation of the pixel values in each feature map, over all of the observations in a mini-batch.

When using multiple GPUs, however, the observations in the mini-batch are distributed over the GPUs. It would be very inefficient to try to synchronize each GPU’s batch normalization calculations for each batch normalization layer. Instead, each GPU calculates the required statistics using a subset of available observations, and uses those statistics to perform the transformation on those observations.

Research communities are still debating whether small or large minibatch sizes yield better performance, but when a minibatch of observations are distributed across multiple GPUs, and the model contains batch normalization layers, SAS Deep Learning recommends that you use reasonably large-sized mini-batches on each GPU, so the statistics will be stable.

In addition to calculating feature map statistics on each mini-batch, the batch normalization algorithm also needs to calculate statistics over the entire training data set before saving the training weights. These statistics are the ones used for scoring (whereas the mini-batch statistics are used for training). Rather than perform an extra epoch at the end of training, the statistics from each mini-batch are averaged over the course of the last training epoch to create the epoch statistics.

The statistics computed in this way are a close approximation to the more complicated computation that uses an extra epoch with fixed weights (as long as the weights in the last epoch do not change much) after each mini-batch of the epoch. (This is usually the case for the last training epoch.) When using multiple GPUs, this calculation is performed exactly the same way as when using a single GPU: the statistics for each mini-batch on each GPU are averaged after each mini-batch to compute the final epoch statistics for scoring.

References

- He, Kaiming, et al. 2015. "Deep Residual Learning for Image Recognition." https://arxiv.org/abs/1512.03385.

- Ioffe, Sergey, and Christian Szegedy. 2015. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” *International Conference on Machine Learning*, pp. 448–456, https://arxiv.org/abs/1502.03167.

- Karpathy, Andrej. 2017. “CS231n: Convolutional Neural Networks for Visual Recognition.” Stanford University, Stanford, CA. http://cs231n.github.io/convolutional-networks/.

- Karpathy, Andrej. 2017. Images ”Fully Connected Neural Network” and “Convolutional Neural Network” from CS231n: Convolutional Neural Networks for Visual Recognition.” Stanford University, Stanford, CA. http://cs231n.github.io/convolutional-networks/.

- Krizhevsky, A., I. Sutskever, and G. E. Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." Neural Information Processing Systems (NIPS). http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

- LeCun, Yann, Leon Bouttou, Yoshua Bengio, and Patrick Haffern. 1988. “Gradient-Based Learning Applied to Document Recognition.” *Procedures of the IEEE*, November 1988. http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf.

- Szegedy, Christian. 2014. “Going Deeper with Convolutions.” Google, https://arxiv.org/abs/1409.4842.

- Visual Geometry Group. 2014. "Very Deep Convolutional Networks for Large Scale Visual Recognition.” Visual Geometry Group, University of Oxford, Oxford England. http://www.robots.ox.ac.uk/~vgg/research/very_deep/.

- Zeiler, M. 2014. “Visualizing and Understanding Convolutional Networks.” https://arxiv.org/abs/1311.2901.

Convolutional Neural Networks (LeNet)

Note

This section assumes the reader has already read through Classifying MNIST digits using Logistic Regression and Multilayer Perceptron. Additionally, it uses the following new Theano functions and concepts: T.tanh, shared variables, basic arithmetic ops, T.grad, floatX, pool , conv2d, dimshuffle. If you intend to run the code on GPU also read GPU.

To run this example on a GPU, you need a good GPU. It needs at least 1GB of GPU RAM. More may be required if your monitor is connected to the GPU.

When the GPU is connected to the monitor, there is a limit of a few seconds for each GPU function call. This is needed as current GPUs can’t be used for the monitor while doing computation. Without this limit, the screen would freeze for too long and make it look as if the computer froze. This example hits this limit with medium-quality GPUs. When the GPU isn’t connected to a monitor, there is no time limit. You can lower the batch size to fix the time out problem.

Note

The code for this section is available for download here and the 3wolfmoon image

Motivation

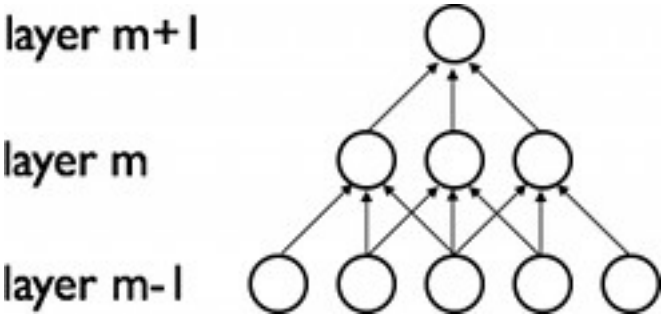
Convolutional Neural Networks (CNN) are biologically-inspired variants of MLPs. From Hubel and Wiesel's early work on the cat's visual cortex [Hubel68], we know the visual cortex contains a complex arrangement of cells. These cells are sensitive to small sub-regions of the visual field, called a *receptive field*. The sub-regions are tiled to cover the entire visual field. These cells act as local filters over the input space and are well-suited to exploit the strong spatially local correlation present in natural images.

Additionally, two basic cell types have been identified: Simple cells respond maximally to specific edge-like patterns within their receptive field. Complex cells have larger receptive fields and are locally invariant to the exact position of the pattern.

The animal visual cortex being the most powerful visual processing system in existence, it seems natural to emulate its behavior. Hence, many neurally-inspired models can be found in the literature. To name a few: the NeoCognitron [Fukushima], HMAX [Serre07] and LeNet-5 [LeCun98], which will be the focus of this tutorial.

Sparse Connectivity

CNNs exploit spatially-local correlation by enforcing a local connectivity pattern between neurons of adjacent layers. In other words, the inputs of hidden units in layer **m** are from a subset of units in layer **m-1**, units that have spatially contiguous receptive fields. We can illustrate this graphically as follows:



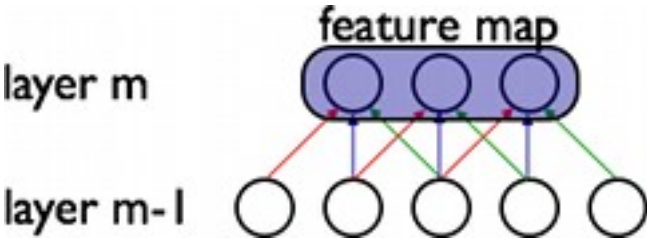
Imagine that layer **m-1** is the input retina. In the above figure, units in layer **m** have receptive fields of width 3 in the input retina and are thus only connected to 3 adjacent neurons in the retina layer. Units in layer **m+1** have a similar connectivity with the layer below. We say that their receptive field with respect to the layer below is also 3, but their receptive field with respect to the input is larger (5). Each unit is unresponsive to variations outside of its receptive field with respect to the retina. The architecture thus ensures that the learnt "filters" produce the strongest response to a spatially local input pattern.

However, as shown above, stacking many such layers leads to (non-linear) "filters" that become increasingly "global" (i.e. responsive to a larger region of pixel space). For example,

the unit in hidden layer **m+1** can encode a non-linear feature of width 5 (in terms of pixel space).

Shared Weights

In addition, in CNNs, each filter h_i is replicated across the entire visual field. These replicated units share the same parameterization (weight vector and bias) and form a *feature map*.



In the above figure, we show 3 hidden units belonging to the same feature map. Weights of the same color are shared—constrained to be identical. Gradient descent can still be used to learn such shared parameters, with only a small change to the original algorithm. The gradient of a shared weight is simply the sum of the gradients of the parameters being shared.

Replicating units in this way allows for features to be detected *regardless of their position in the visual field.* Additionally, weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt. The constraints on the model enable CNNs to achieve better generalization on vision problems.

Details and Notation

A feature map is obtained by repeated application of a function across sub-regions of the entire image, in other words, by *convolution* of the input image with a linear filter, adding a bias term and then applying a non-linear function. If we denote the k-th feature map at a given layer as h^k, whose filters are determined by the weights W^k and bias b_k, then the feature map h^k is obtained as follows (for $tanh$ non-linearities):

$$h_{ij}^k = \tanh((W^k * x)_{ij} + b_k).$$

Note

Recall the following definition of convolution for a 1D signal.
$o[n] = f[n] * g[n] = \sum_{u=-\infty}^{\infty} f[u]g[n-u] = \sum_{u=-\infty}^{\infty} f[n-u]g[u]$
.

This can be extended to 2D as follows:
$o[m,n] = f[m,n] * g[m,n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u,v]g[m-u,n-v]$
.

To form a richer representation of the data, each hidden layer is composed of *multiple* feature maps, $\{h^{(k)}, k = 0..K\}$. The weights W of a hidden layer can be represented in a 4D tensor containing elements for every combination of destination feature map, source feature map, source vertical position, and source horizontal position. The biases b can be represented as a vector containing one element for every destination feature map. We illustrate this graphically as follows:

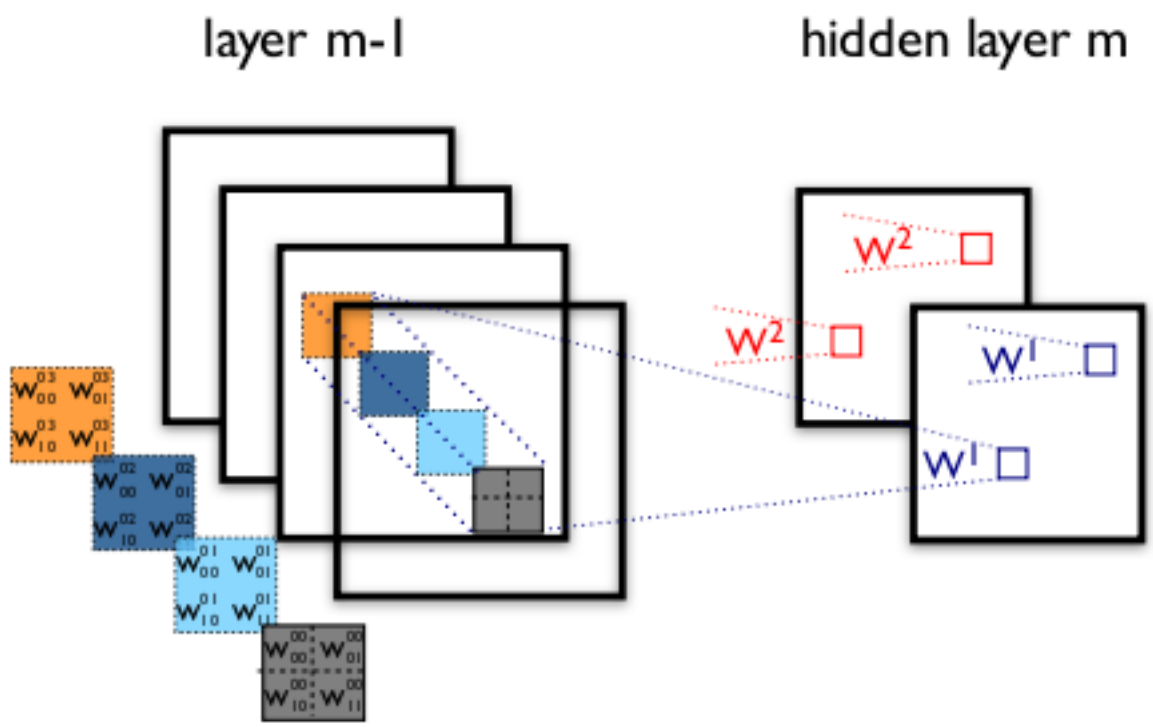


Figure 1: example of a convolutional layer

The figure shows two layers of a CNN. **Layer m-1** contains four feature maps. **Hidden layer m** contains two feature maps (h^0 and h^1). Pixels (neuron outputs) in h^0 and h^1 (outlined as blue and red squares) are computed from pixels of layer (m-1) which fall within their 2x2 receptive field in the layer below (shown as colored rectangles). Notice how the receptive field spans all four input feature maps. The weights W^0 and W^1 of h^0 and h^1 are thus 3D weight tensors. The leading dimension indexes the input feature maps, while the other two refer to the pixel coordinates.

Putting it all together, W_{ij}^{kl} denotes the weight connecting each pixel of the k-th feature map at layer m, with the pixel at coordinates (i,j) of the l-th feature map of layer (m-1).

The Convolution Operator

ConvOp is the main workhorse for implementing a convolutional layer in Theano. ConvOp is used by `theano.tensor.signal.conv2d`, which takes two symbolic inputs:

- a 4D tensor corresponding to a mini-batch of input images. The shape of the tensor is as follows: [mini-batch size, number of input feature maps, image height, image width].
- a 4D tensor corresponding to the weight matrix W. The shape of the tensor is: [number of feature maps at layer m, number of feature maps at layer m-1, filter height, filter width]

Below is the Theano code for implementing a convolutional layer similar to the one of Figure 1. The input consists of 3 features maps (an RGB color image) of size 120x160. We use two convolutional filters with 9x9 receptive fields.

```
import theano
from theano import tensor as T
from theano.tensor.nnet import conv2d

import numpy

rng = numpy.random.RandomState(23455)

# instantiate 4D tensor for input
input = T.tensor4(name='input')

# initialize shared variable for weights.
w_shp = (2, 3, 9, 9)
w_bound = numpy.sqrt(3 * 9 * 9)
W = theano.shared( numpy.asarray(
            rng.uniform(
                low=-1.0 / w_bound,
                high=1.0 / w_bound,
                size=w_shp),
            dtype=input.dtype), name ='W')

# initialize shared variable for bias (1D tensor) with random values
# IMPORTANT: biases are usually initialized to zero. However in this
# particular application, we simply apply the convolutional layer to
# an image without learning the parameters. We therefore initialize
# them to random values to "simulate" learning.
b_shp = (2,)
b = theano.shared(numpy.asarray(
            rng.uniform(low=-.5, high=.5, size=b_shp),
            dtype=input.dtype), name ='b')

# build symbolic expression that computes the convolution of input with
filters in w
conv_out = conv2d(input, W)

# build symbolic expression to add bias and apply activation function, i.e.
produce neural net layer output
# A few words on ``dimshuffle`` :
#   ``dimshuffle`` is a powerful tool in reshaping a tensor;
#   what it allows you to do is to shuffle dimension around
#   but also to insert new ones along which the tensor will be
#   broadcastable;
#   dimshuffle('x', 2, 'x', 0, 1)
#   This will work on 3d tensors with no broadcastable
#   dimensions. The first dimension will be broadcastable,
#   then we will have the third dimension of the input tensor as
#   the second of the resulting tensor, etc. If the tensor has
```

```
#   shape (20, 30, 40), the resulting tensor will have dimensions
#   (1, 40, 1, 20, 30). (AxBxC tensor is mapped to 1xCx1xAxB tensor)
#   More examples:
#    dimshuffle('x') -> make a 0d (scalar) into a 1d vector
#    dimshuffle(0, 1) -> identity
#    dimshuffle(1, 0) -> inverts the first and second dimensions
#    dimshuffle('x', 0) -> make a row out of a 1d vector (N to 1xN)
#    dimshuffle(0, 'x') -> make a column out of a 1d vector (N to Nx1)
#    dimshuffle(2, 0, 1) -> AxBxC to CxAxB
#    dimshuffle(0, 'x', 1) -> AxB to Ax1xB
#    dimshuffle(1, 'x', 0) -> AxB to Bx1xA
output = T.nnet.sigmoid(conv_out + b.dimshuffle('x', 0, 'x', 'x'))

# create theano function to compute filtered images
f = theano.function([input], output)
```

Let’s have a little bit of fun with this...

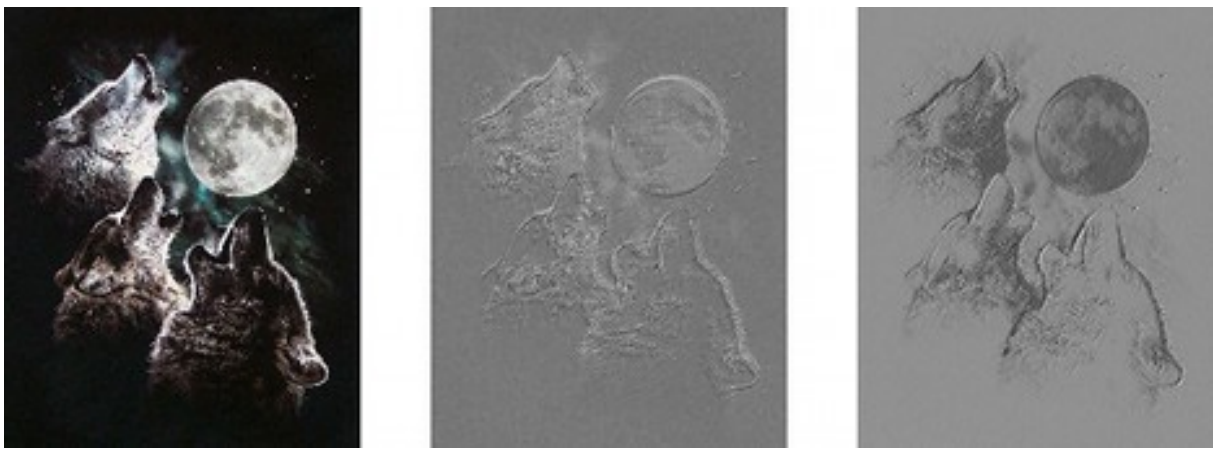
```
import numpy
import pylab
from PIL import Image

# open random image of dimensions 639x516
img = Image.open(open('doc/images/3wolfmoon.jpg'))
# dimensions are (height, width, channel)
img = numpy.asarray(img, dtype='float64') / 256.

# put image in 4D tensor of shape (1, 3, height, width)
img_ = img.transpose(2, 0, 1).reshape(1, 3, 639, 516)
filtered_img = f(img_)

# plot original image and first and second components of output
pylab.subplot(1, 3, 1); pylab.axis('off'); pylab.imshow(img)
pylab.gray();
# recall that the convOp output (filtered image) is actually a "minibatch",
# of size 1 here, so we take index 0 in the first dimension:
pylab.subplot(1, 3, 2); pylab.axis('off'); pylab.imshow(filtered_img[0, 0,
:, :])
pylab.subplot(1, 3, 3); pylab.axis('off'); pylab.imshow(filtered_img[0, 1,
:, :])
pylab.show()
```

This should generate the following output.



Notice that a randomly initialized filter acts very much like an edge detector!

Note that we use the same weight initialization formula as with the MLP. Weights are sampled randomly from a uniform distribution in the range [-1/fan-in, 1/fan-in], where fan-in is the number of inputs to a hidden unit. For MLPs, this was the number of units in the layer below. For CNNs however, we have to take into account the number of input feature maps and the size of the receptive fields.

MaxPooling

Another important concept of CNNs is *max-pooling*, which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value.

Max-pooling is useful in vision for two reasons:

1. By eliminating non-maximal values, it reduces computation for upper layers.

2. It provides a form of translation invariance. Imagine cascading a max-pooling layer with a convolutional layer. There are 8 directions in which one can translate the input image by a single pixel. If max-pooling is done over a 2x2 region, 3 out of these 8 possible configurations will produce exactly the same output at the convolutional layer. For max-pooling over a 3x3 window, this jumps to 5/8.

   Since it provides additional robustness to position, max-pooling is a “smart” way of reducing the dimensionality of intermediate representations.

Max-pooling is done in Theano by way of `theano.tensor.signal.pool.pool_2d`. This function takes as input an N dimensional tensor (where N >= 2) and a downscaling factor and performs max-pooling over the 2 trailing dimensions of the tensor.

An example is worth a thousand words:

```
from theano.tensor.signal import pool

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = pool.pool_2d(input, maxpool_shape, ignore_border=True)
f = theano.function([input],pool_out)

invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)
print 'With ignore_border set to True:'
print 'invals[0, 0, :, :] =\n', invals[0, 0, :, :]
print 'output[0, 0, :, :] =\n', f(invals)[0, 0, :, :]

pool_out = pool.pool_2d(input, maxpool_shape, ignore_border=False)
f = theano.function([input],pool_out)
print 'With ignore_border set to False:'
print 'invals[1, 0, :, :] =\n ', invals[1, 0, :, :]
print 'output[1, 0, :, :] =\n ', f(invals)[1, 0, :, :]
```

This should generate the following output:

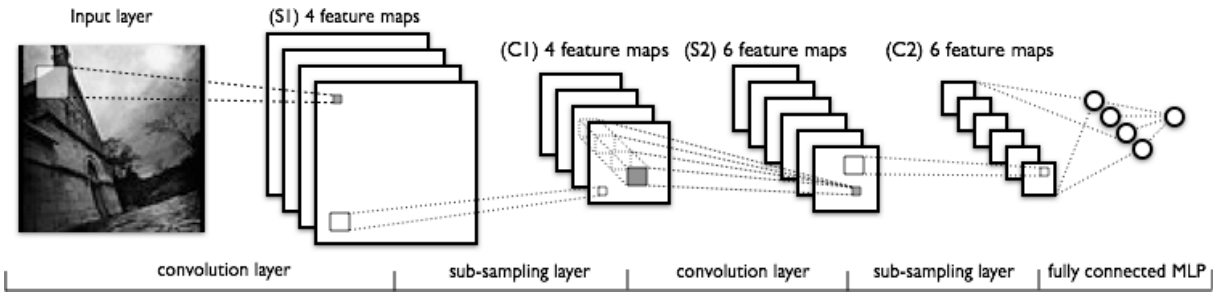
```
With ignore_border set to True:
    invals[0, 0, :, :] =
    [[  4.17022005e-01   7.20324493e-01   1.14374817e-04   3.02332573e-01
1.46755891e-01]
     [  9.23385948e-02   1.86260211e-01   3.45560727e-01   3.96767474e-01
5.38816734e-01]
     [  4.19194514e-01   6.85219500e-01   2.04452250e-01   8.78117436e-01
2.73875932e-02]
     [  6.70467510e-01   4.17304802e-01   5.58689828e-01   1.40386939e-01
1.98101489e-01]
     [  8.00744569e-01   9.68261576e-01   3.13424178e-01   6.92322616e-01
8.76389152e-01]]
    output[0, 0, :, :] =
    [[ 0.72032449  0.39676747]
     [ 0.6852195   0.87811744]]

With ignore_border set to False:
    invals[1, 0, :, :] =
    [[ 0.01936696  0.67883553  0.21162812  0.26554666  0.49157316]
     [ 0.05336255  0.57411761  0.14672857  0.58930554  0.69975836]
     [ 0.10233443  0.41405599  0.69440016  0.41417927  0.04995346]
     [ 0.53589641  0.66379465  0.51488911  0.94459476  0.58655504]
     [ 0.90340192  0.1374747   0.13927635  0.80739129  0.39767684]]
    output[1, 0, :, :] =
    [[ 0.67883553  0.58930554  0.69975836]
     [ 0.66379465  0.94459476  0.58655504]
     [ 0.90340192  0.80739129  0.39767684]]
```

Note that compared to most Theano code, the `max_pool_2d` operation is a little *special*. It requires the downscaling factor `ds` (tuple of length 2 containing downscaling factors for image width and height) to be known at graph build time. This may change in the near future.

The Full Model: LeNet

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family of models. While the exact details of the model will vary greatly, the figure below shows a graphical depiction of a LeNet model.



The lower-layers are composed to alternating convolution and max-pooling layers. The upper-layers however are fully-connected and correspond to a traditional MLP (hidden layer + logistic regression). The input to the first fully-connected layer is the set of all features maps at the layer below.

From an implementation point of view, this means lower-layers operate on 4D tensors. These are then flattened to a 2D matrix of rasterized feature maps, to be compatible with our previous MLP implementation.

Note

Note that the term “convolution” could corresponds to different mathematical operations:

1. theano.tensor.nnet.conv2d, which is the most common one in almost all of the recent published convolutional models. In this operation, each output feature map is connected to each input feature map by a different 2D filter, and its value is the sum of the individual convolution of all inputs through the corresponding filter.
2. The convolution used in the original LeNet model: In this work, each output feature map is only connected to a subset of input feature maps.
3. The convolution used in signal processing: theano.tensor.signal.conv.conv2d, which works only on single channel inputs.

Here, we use the first operation, so this models differ slightly from the original LeNet paper. One reason to use 2. would be to reduce the amount of computation needed, but modern hardware makes it as fast to have the full connection pattern. Another reason would be to slightly reduce the number of free parameters, but we have other regularization techniques at our disposal.

Putting it All Together

We now have all we need to implement a LeNet model in Theano. We start with the LeNetConvPoolLayer class, which implements a {convolution + max-pooling} layer.

```
class LeNetConvPoolLayer(object):
    """Pool Layer of a convolutional network """

    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2,
2)):
        """
        Allocate a LeNetConvPoolLayer with shared variable internal
parameters.

        :type rng: numpy.random.RandomState
        :param rng: a random number generator used to initialize weights

        :type input: theano.tensor.dtensor4
        :param input: symbolic image tensor, of shape image_shape

        :type filter_shape: tuple or list of length 4
        :param filter_shape: (number of filters, num input feature maps,
                              filter height, filter width)

        :type image_shape: tuple or list of length 4
        :param image_shape: (batch size, num input feature maps,
                             image height, image width)

        :type poolsize: tuple or list of length 2
        :param poolsize: the downsampling (pooling) factor (#rows, #cols)
        """

        assert image_shape[1] == filter_shape[1]
        self.input = input

        # there are "num input feature maps * filter height * filter width"
```

```
        # inputs to each hidden unit
        fan_in = numpy.prod(filter_shape[1:])
        # each unit in the lower layer receives a gradient from:
        # "num output feature maps * filter height * filter width" /
        #   pooling size
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:]) //
                   numpy.prod(poolsize))
        # initialize weights with random weights
        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound, size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
        )

        # the bias is a 1D tensor -- one bias per output feature map
        b_values = numpy.zeros((filter_shape[0],),
dtype=theano.config.floatX)
        self.b = theano.shared(value=b_values, borrow=True)

        # convolve input feature maps with filters
        conv_out = conv2d(
            input=input,
            filters=self.W,
            filter_shape=filter_shape,
            input_shape=image_shape
        )

        # pool each feature map individually, using maxpooling
        pooled_out = pool.pool_2d(
            input=conv_out,
            ds=poolsize,
            ignore_border=True
        )

        # add the bias term. Since the bias is a vector (1D array), we
first
        # reshape it to a tensor of shape (1, n_filters, 1, 1). Each bias
will
        # thus be broadcasted across mini-batches and feature map
        # width & height
        self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x',
'x'))

        # store parameters of this layer
        self.params = [self.W, self.b]

        # keep track of model input
        self.input = input
```

Notice that when initializing the weight values, the fan-in is determined by the size of the receptive fields and the number of input feature maps.

Finally, using the LogisticRegression class defined in Classifying MNIST digits using Logistic Regression and the HiddenLayer class defined in Multilayer Perceptron , we can instantiate the network as follows.

```
x = T.matrix('x')   # the data is presented as rasterized images
y = T.ivector('y')  # the labels are presented as 1D vector of
                    # [int] labels

######################
# BUILD ACTUAL MODEL #
######################
print('... building the model')

# Reshape matrix of rasterized images of shape (batch_size, 28 * 28)
# to a 4D tensor, compatible with our LeNetConvPoolLayer
# (28, 28) is the size of MNIST images.
layer0_input = x.reshape((batch_size, 1, 28, 28))

# Construct the first convolutional pooling layer:
# filtering reduces the image size to (28-5+1 , 28-5+1) = (24, 24)
# maxpooling reduces this further to (24/2, 24/2) = (12, 12)
# 4D output tensor is thus of shape (batch_size, nkerns[0], 12, 12)
layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 28, 28),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)

# Construct the second convolutional pooling layer
# filtering reduces the image size to (12-5+1, 12-5+1) = (8, 8)
# maxpooling reduces this further to (8/2, 8/2) = (4, 4)
# 4D output tensor is thus of shape (batch_size, nkerns[1], 4, 4)
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 12, 12),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)

# the HiddenLayer being fully-connected, it operates on 2D matrices of
# shape (batch_size, num_pixels) (i.e matrix of rasterized images).
# This will generate a matrix of shape (batch_size, nkerns[1] * 4 * 4),
# or (500, 50 * 4 * 4) = (500, 800) with the default values.
layer2_input = layer1.output.flatten(2)

# construct a fully-connected sigmoidal layer
layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 4 * 4,
    n_out=500,
    activation=T.tanh
)

# classify the values of the fully-connected sigmoidal layer
layer3 = LogisticRegression(input=layer2.output, n_in=500, n_out=10)

# the cost we minimize during training is the NLL of the model
cost = layer3.negative_log_likelihood(y)

# create a function to compute the mistakes that are made by the model
test_model = theano.function(
```

```
        [index],
        layer3.errors(y),
        givens={
            x: test_set_x[index * batch_size: (index + 1) * batch_size],
            y: test_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    validate_model = theano.function(
        [index],
        layer3.errors(y),
        givens={
            x: valid_set_x[index * batch_size: (index + 1) * batch_size],
            y: valid_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )

    # create a list of all model parameters to be fit by gradient descent
    params = layer3.params + layer2.params + layer1.params + layer0.params

    # create a list of gradients for all model parameters
    grads = T.grad(cost, params)

    # train_model is a function that updates the model parameters by
    # SGD Since this model has many parameters, it would be tedious to
    # manually create an update rule for each model parameter. We thus
    # create the updates list by automatically looping over all
    # (params[i], grads[i]) pairs.
    updates = [
        (param_i, param_i - learning_rate * grad_i)
        for param_i, grad_i in zip(params, grads)
    ]

    train_model = theano.function(
        [index],
        cost,
        updates=updates,
        givens={
            x: train_set_x[index * batch_size: (index + 1) * batch_size],
            y: train_set_y[index * batch_size: (index + 1) * batch_size]
        }
    )
```

We leave out the code that performs the actual training and early-stopping, since it is exactly the same as with an MLP. The interested reader can nevertheless access the code in the ‘code’ folder of DeepLearningTutorials.

Running the Code

The user can then run the code by calling:

```
python code/convolutional_mlp.py
```

The following output was obtained with the default parameters on a Core i7-2600K CPU clocked at 3.40GHz and using flags ‘floatX=float32’:

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 17800,with test
```

```
performance 0.920000 %
The code for file convolutional_mlp.py ran for 380.28m
```

Using a GeForce GTX 285, we obtained the following:

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 15500,with test
performance 0.930000 %
The code for file convolutional_mlp.py ran for 46.76m
```

And similarly on a GeForce GTX 480:

```
Optimization complete.
Best validation score of 0.910000 % obtained at iteration 16400,with test
performance 0.930000 %
The code for file convolutional_mlp.py ran for 32.52m
```

Note that the discrepancies in validation and test error (as well as iteration count) are due to different implementations of the rounding mechanism in hardware. They can be safely ignored.

Tips and Tricks

Choosing Hyperparameters

CNNs are especially tricky to train, as they add even more hyper-parameters than a standard MLP. While the usual rules of thumb for learning rates and regularization constants still apply, the following should be kept in mind when optimizing CNNs.

Number of filters

When choosing the number of filters per layer, keep in mind that computing the activations of a single convolutional filter is much more expensive than with traditional MLPs !

Assume layer $(l-1)$ contains K^{l-1} feature maps and $M \times N$ pixel positions (i.e., number of positions times number of feature maps), and there are K^l filters at layer l of shape $m \times n$. Then computing a feature map (applying an $m \times n$ filter at all $(M-m) \times (N-n)$ pixel positions where the filter can be applied) costs $(M-m) \times (N-n) \times m \times n \times K^{l-1}$. The total cost is K^l times that. Things may be more complicated if not all features at one level are connected to all features at the previous one.

For a standard MLP, the cost would only be $K^l \times K^{l-1}$ where there are K^l different neurons at level l. As such, the number of filters used in CNNs is typically much smaller than the number of hidden units in MLPs and depends on the size of the feature maps (itself a function of input image size and filter shapes).

Since feature map size decreases with depth, layers near the input layer will tend to have fewer filters while layers higher up can have much more. In fact, to equalize computation at

each layer, the product of the number of features and the number of pixel positions is typically picked to be roughly constant across layers. To preserve the information about the input would require keeping the total number of activations (number of feature maps times number of pixel positions) to be non-decreasing from one layer to the next (of course we could hope to get away with less when we are doing supervised learning). The number of feature maps directly controls capacity and so that depends on the number of available examples and the complexity of the task.

Filter Shape

Common filter shapes found in the literature vary greatly, usually based on the dataset. Best results on MNIST-sized images (28x28) are usually in the 5x5 range on the first layer, while natural image datasets (often with hundreds of pixels in each dimension) tend to use larger first-layer filters of shape 12x12 or 15x15.

The trick is thus to find the right level of "granularity" (i.e. filter shapes) in order to create abstractions at the proper scale, given a particular dataset.

Max Pooling Shape

Typical values are 2x2 or no max-pooling. Very large input images may warrant 4x4 pooling in the lower-layers. Keep in mind however, that this will reduce the dimension of the signal by a factor of 16, and may result in throwing away too much information.

Footnotes

[1] For clarity, we use the word "unit" or "neuron" to refer to the artificial neuron and "cell" to refer to the biological neuron.

Tips

If you want to try this model on a new dataset, here are a few tips that can help you get better results:

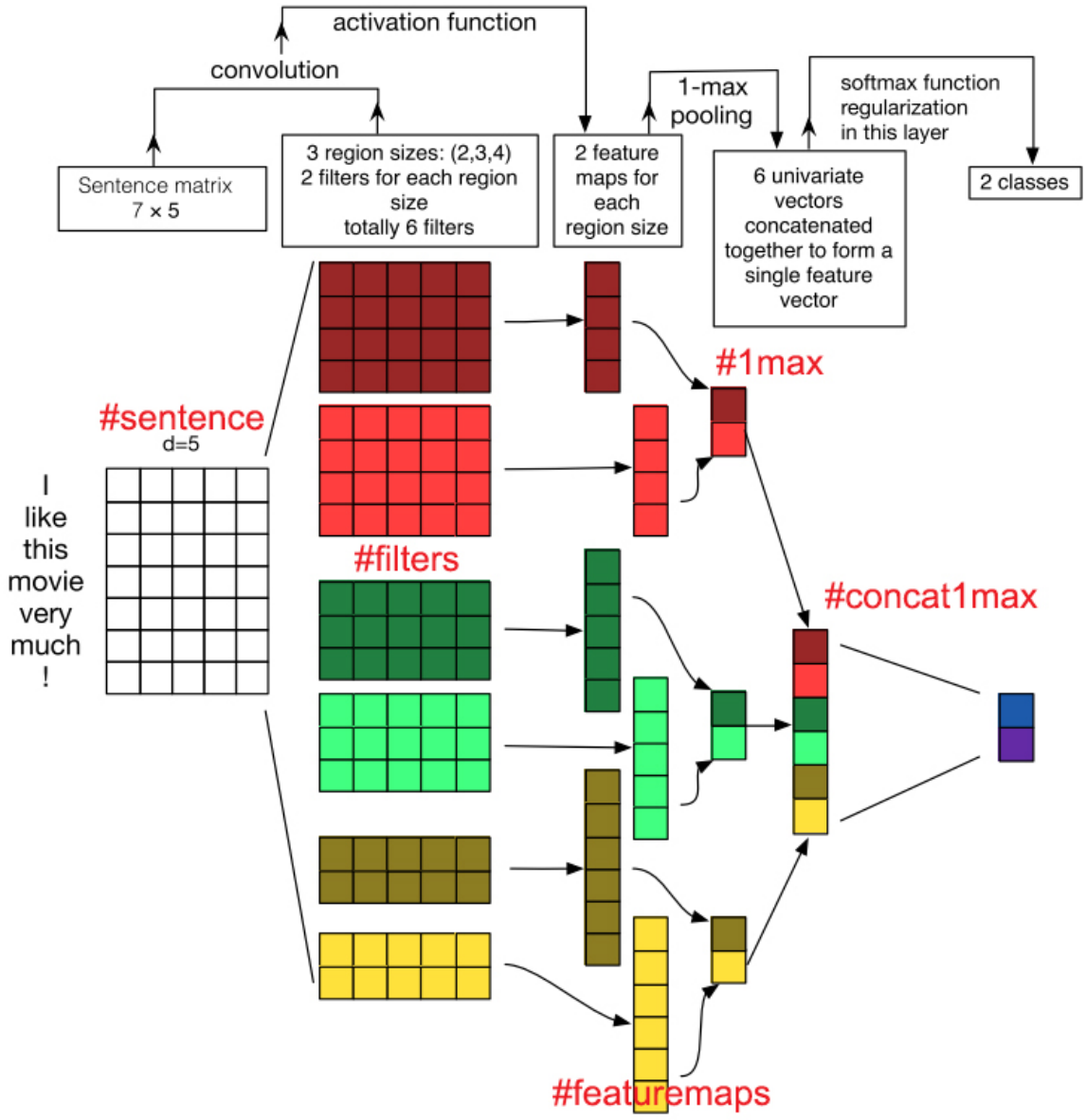
-

Understanding how Convolutional Neural Network (CNN) perform text classification with word embeddings

CNN has been successful in various text classification tasks. In [1], the author showed that a simple CNN with little hyperparameter tuning and static vectors achieves excellent results on multiple benchmarks – improving upon the state of the art on 4 out of 7 tasks.

However, when learning to apply CNN on word embeddings, keeping track of the dimensions of the matrices can be confusing. The aim of this short post is to simply to keep track of these dimensions and understand how CNN works for text classification. We would use a one-layer CNN on a 7-word sentence, with word embeddings of dimension 5 – a toy example to aid the understanding of CNN. All examples are from [2].

Setup



Above figure is from [2], with *#hash-tags* added to aid discussion. Quoting the original caption here, to be discussed later. *"Figure 1: Illustration of a CNN architecture for sentence classification. We depict three filter region sizes: 2,3,4, each of which has 2 filters. Filters perform convolutions on the sentence matrix and generate (variable-length) feature maps; 1-max pooling is performed over each map, i.e., the largest number from each feature map is recorded. Thus, a univariate feature vector is generated from all six maps, and these 6 features are concatenated to form a feature vector for the penultimate layer. The final softmax later then receives this feature vector as input and uses it to classify the sentence; here we assume binary classification and hence depict two possible output states."*

#sentence

The example is "I like this movie very much!", there are 6 words here and the exclamation mark is treated like a word – some researchers do this differently and disregard the exclamation mark – in total there are 7 words in the sentence. The authors chose 5 to be the dimension of the word vectors. We let s denote the length of sentence and d denote the

dimension of the word vector, hence we now have a sentence matrix of the shape *s* x *d*, or 7 x 5.

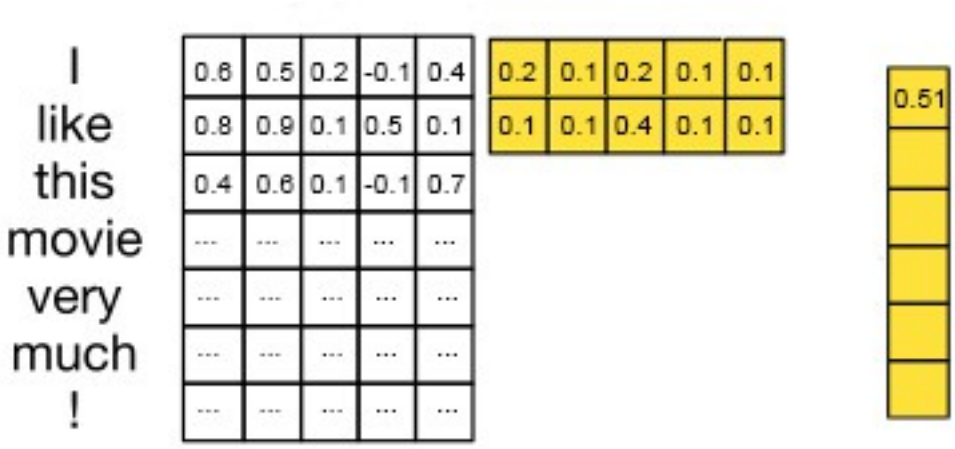
#filters

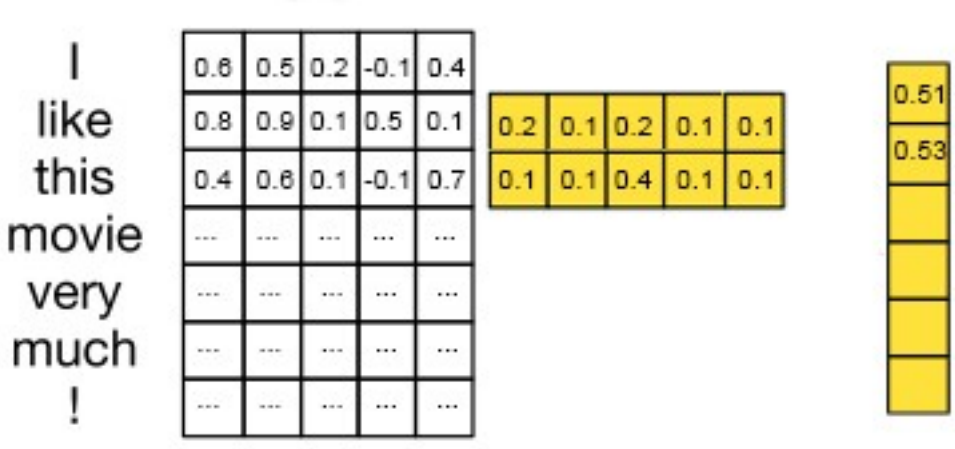
One of the desirable properties of CNN is that it preserves 2D spatial orientation in computer vision. Texts, like pictures, have an orientation. Instead of 2-dimensional, texts have a one-dimensional structure where words sequence matter. We also recall that all words in the example are each replaced by a 5-dimensional word vector, hence we fix one dimension of the filter to match the word vectors (5) and vary the region size, *h*. Region size refers to the number of rows – representing word – of the sentence matrix that would be filtered.

In the figure, #filters are the illustrations of the filters, not what has been filtered out from the sentence matrix by the filter, the next paragraph would make this distinction clearer. Here, the authors chose to use 6 filters – 2 complementary filters to consider (2,3,4) words.

#featuremaps

For this section, we step-through on how CNN perform convolutions / filtering. I have filled in some numbers in the sentence matrix and the filter matrix for clarity.





The above illustrates the action of the 2-word filter on the sentence matrix. First, the two-word filter, represented by the 2 x 5 yellow matrix **w**, overlays across the word vectors of "I" and "like". Next, it performs an element-wise product for all its 2 x 5 elements, and then sum them up and obtain one number (0.6 x 0.2 + 0.5 x 0.1 + … + 0.1 x 0.1 = 0.51). 0.51 is recorded as the first element of the output sequence, **o**, for this filter. Then, the filter moves down 1 word and overlays across the word vectors of 'like' and 'this' and perform the same operation to get 0.53. Therefore, **o** will have the shape of (s–h+1 x 1), in this case (7-2+1 x 1)

To obtain the feature map, **c**, we add a bias term (a scalar, i.e., shape 1×1) and apply an activation function (e.g. ReLU). This gives us **c,** with the same shape as **o** (s–h+1 x 1).

#1max

Notice that the dimensionality of **c** is dependent both s and h, in other words, it will vary across sentences of different lengths and filters of different region sizes. To tackle this problem, the authors employ the 1-max pooling function and extract the largest number from each **c** vector.

#concat1max

After 1-max pooling, we are certain to have a fixed-length vector of 6 elements (= number of filters = number of filters per region size (2) x number of region size considered (3)). This fixed length vector can then be fed into a softmax (fully-connected) layer to perform the classification. The error from the classification is then back-propagated back into the following parameters as part of learning:

- The **w** matrices that produced **o**
- The bias term that is added to **o** to produce **c**
- Word vectors (optional, use validation performance to decide)

Conclusion

This short post clarifies the workings of the CNN on word embeddings by focussing on the dimensionality of matrices in each intermediate step.

References

1. Kim Y. Convolutional Neural Networks for Sentence Classification. 2014;
2. Zhang Y, Wallace B. A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification. arXiv preprint arXiv:151003820. 2015; PMID: 463165

DeepLearning series: Convolutional Neural Networks

In this blog, I will explain the details of Convolutional Neural Networks (CNNs or ConvNets), which have proven to be very effective in areas such as image recognition and classification.

Finally, I will move on to Residual and Inception networks, which help overcome issues related to training very deep networks.

CONVOLUTIONAL NEURAL NETWORK:

Computer vision is an exciting field, which has evolved quickly thanks to deep learning. Researchers in this area have been experimenting many neural-network architectures and algorithms, which have influenced other fields as well.

In computer vision, images are the training data of a network, and the input features are the pixels of an image. These features can get really big. For example, when dealing with a 1megapixel image, the total number of features in that picture is 3 million (=1,000 x 1,000 x 3 color channels). Then imagine passing this through a neural network with just 1,000 hidden units, and we end up with some weights of 3 billion parameters!

These numbers are too big to be managed, but, luckily, we have the perfect solution: Convolutional neural networks (ConvNets).

There are 3 types of layers in a convolutional network:

- Convolution (CONV)
- Pooling (POOL)
- Fully connected (FC)

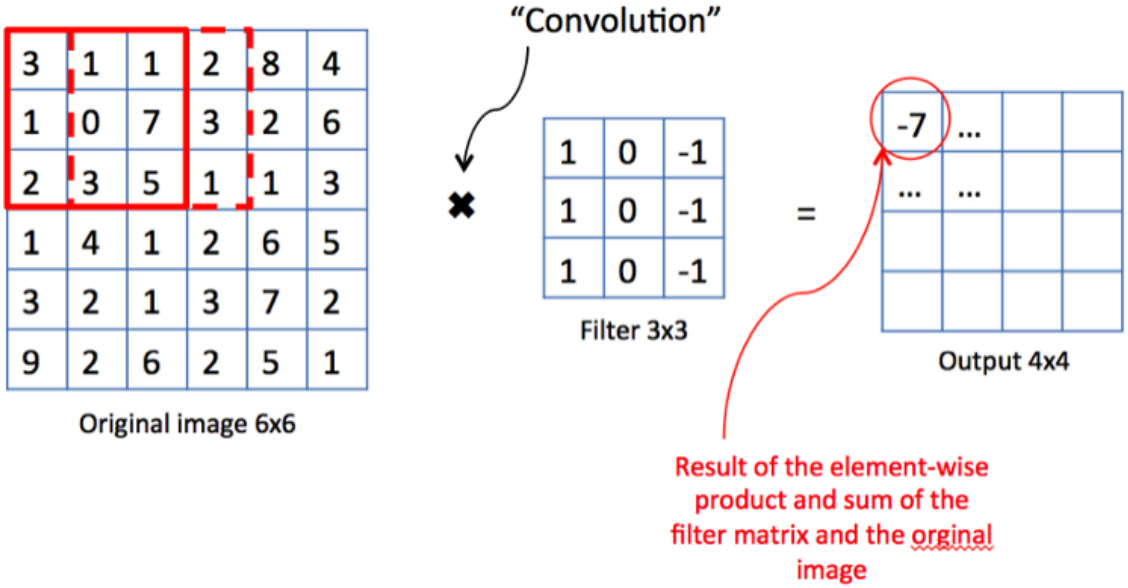
CONVOLUTION LAYER

A "*convolution*" is one of the building blocks of the Convolutional network. The primary purpose of a "convolution" in the case of a ConvNet is to extract features from the input image.

Every image can be represented as a matrix of pixel values. An image from a standard digital camera will have three channels—red, green and blue. You can imagine those as three 2d-matrices stacked over each other (one for each color), each having pixel values in the range 0 to 255.

Applying a convolution to an image is like running a filter of a certain dimension and sliding it on top of the image. That operation is translated into an element-wise multiplication between the two matrices and finally an addition of the multiplication outputs. The final integer of this computation forms a single element of the output matrix.

Let's review this via an example, where we want to apply a filter (kernel) to detect vertical edges from a 2D original image.



The value 1 on the kernel allows filtering brightness, while -1 highlights the darkness and 0 the grey from the original image when the filter slides on top.

In the above example, I used a value of a stride equal to 1, meaning the filter moves horizontally and vertically by one pixel.

In this example the values of the filter were already decided in the convolution. The goal of a convolutional neural network is to learn the number of filters. We treat them as parameters, which the network learns using backpropagation.

You might be wondering how to calculate the output size, based on the filter dimensions and the way we slide it though the image. I will get to the formula, but first I want to introduce a bit of terminology.

You saw in the earlier example how the filter moved with a stride of 1 and covered the whole image from edge to edge. This is what it's called a "valid" convolution since the filter stays within the borders of the image. However, one problem quickly arises. When moving the filter this way we see that the pixels on the edges are "touched" less by the filter than the pixels within the image. That means we are throwing away some information related to those positions. Furthermore, the output image is shrinking on every convolution, which could be intentional, but if the input image is small, we quickly shrink it too fast.

A solution to those setbacks is the use of "padding". Before we apply a convolution, we pad the image with zeros all around its border to allow the filter to slide on top and maintain the output size equal to the input. The result of padding in the previous example will be:

0	0	0	0	0	0	0	0
0	3	1	1	2	8	4	0
0	1	0	7	3	2	6	0
0	2	3	5	1	1	3	0
0	1	4	1	2	6	5	0
0	3	2	1	3	7	2	0
0	9	2	6	2	5	1	0
0	0	0	0	0	0	0	0

Padding will result in a "**same**" convolution.

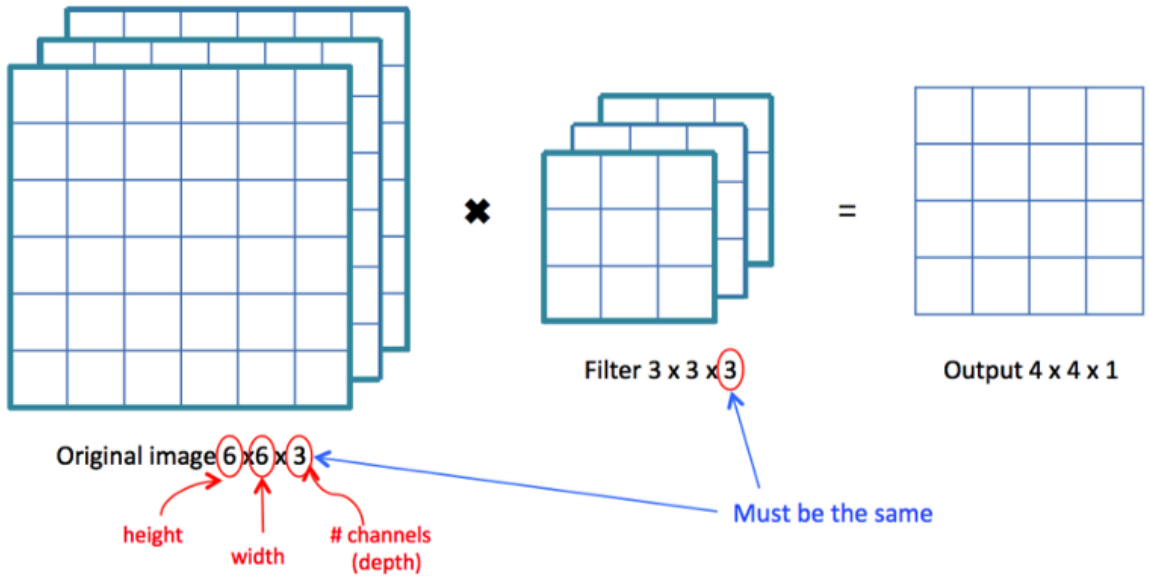
I talked about "stride", which is essentially how many pixels the filter shifts over the original image. Great, so now I can introduce the formula to quickly calculate the output size, knowing the filter size (f), stride (s), pad (p), and input size (n):

$$Output\ size = \left(\frac{n + 2p - f}{s} + 1\right) x \left(\frac{n + 2p - f}{s} + 1\right)$$

Keep in mind that the filter size is usually an odd value, and if the fraction above is not an integer you should round it down.

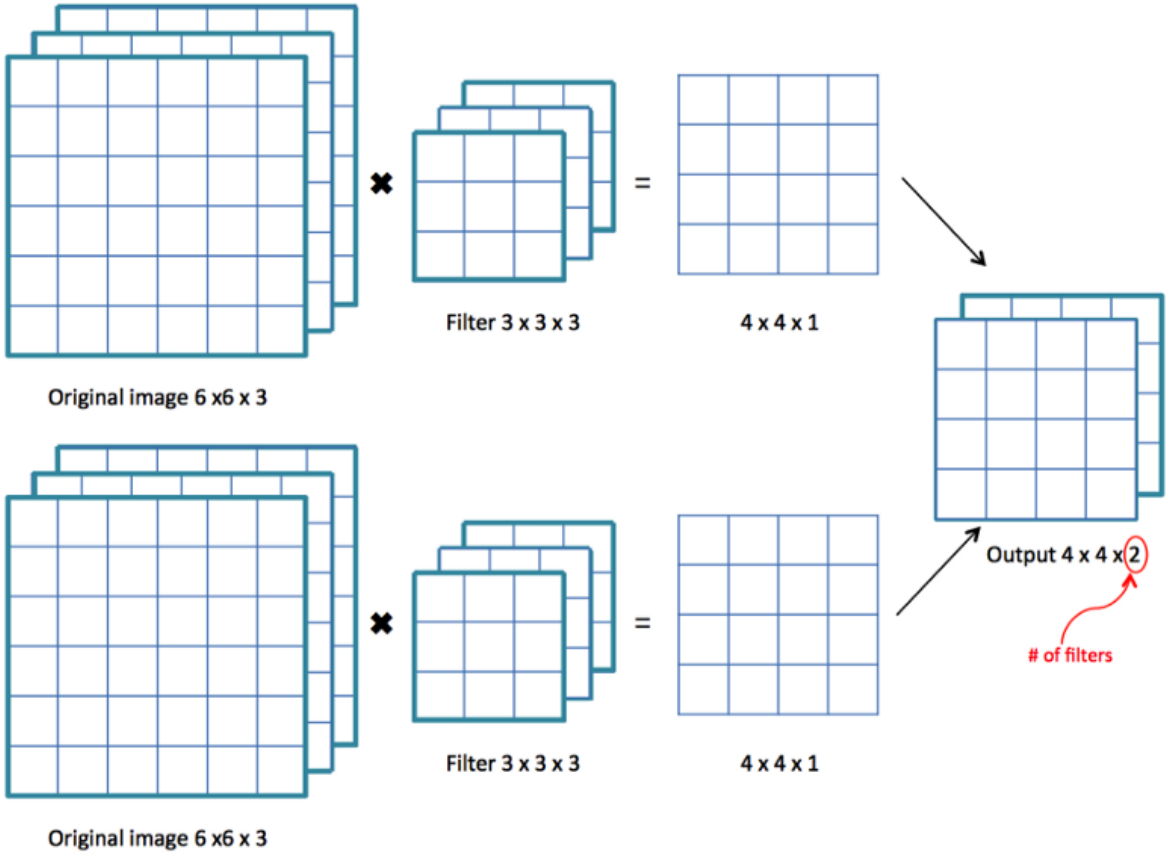
The previous example was on a 2D matrix, but I mentioned earlier that images are composed of three channels (R-red, G-green, B-blue). Therefore the input is a volume, a stack of three matrices, which forms a depth identified by the number of channels.

If we apply only one filter the result would be:



where the cube filter of 27 parameters now slides on top of the cube of the input image.

So far we have only applied one filter at a time, but we can apply multiple filters to detect several different features. This is what brings us to the crucial concept for building convolutional neural networks. Now each filter brings us its own output We can stack them all together and create an output volume, such as:



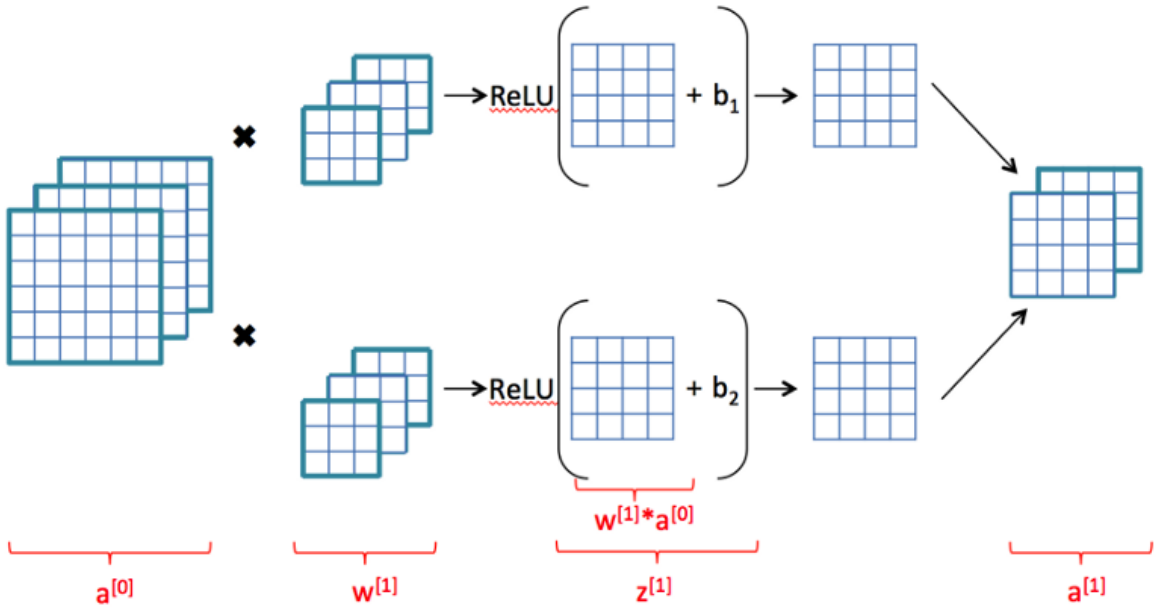
Therefore, in general terms we have:

Input:	Filter:	Output:
$(n \times n \times n_c)$	$(f \times f \times n_c)$	$\left(\left[\frac{n+2p-f}{s}+1\right] \times \left[\frac{n+2p-f}{s}+1\right] \times n_c'\right)$

(with nc' as the number of filters, which are detecting different features)

One-layer of a convolutional neural network

The final step that takes us to a convolutional neural layer is to add the bias and a non-linear function.



Remember that the parameters involved in one layer are independent of the input size image.

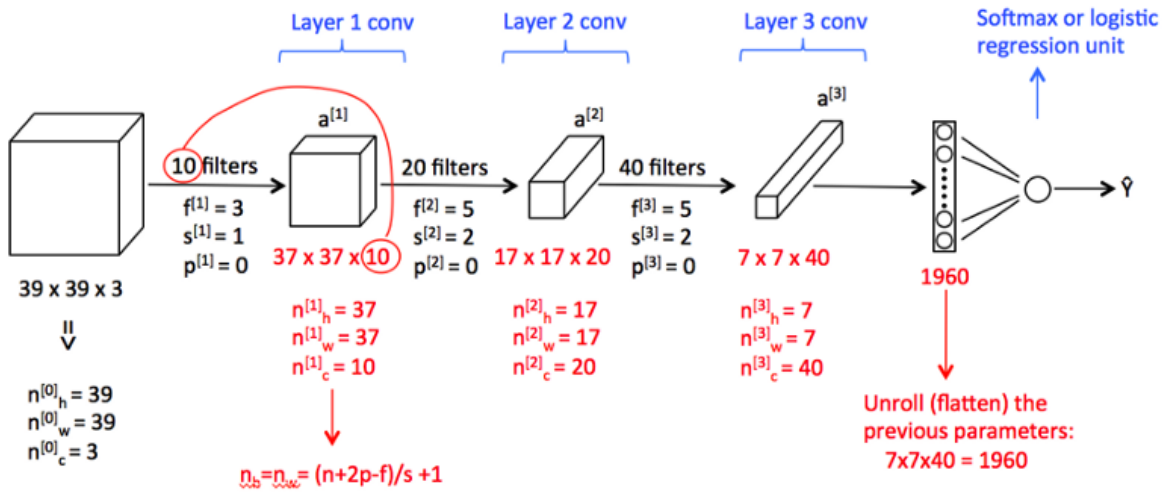
So let's consider, for example, that we have 10 filters that are of size 3x3x3 in one layer of a neural network. Each filter has 27 (3x3x3) + 1 bias => 28 parameters.

Therefore, the total amount of parameters in the layer is 280 (10x28).

Deep Convolutional Network

We are now ready to build a complete deep convolutional neural network.

The following architecture depicts a simple example of that:



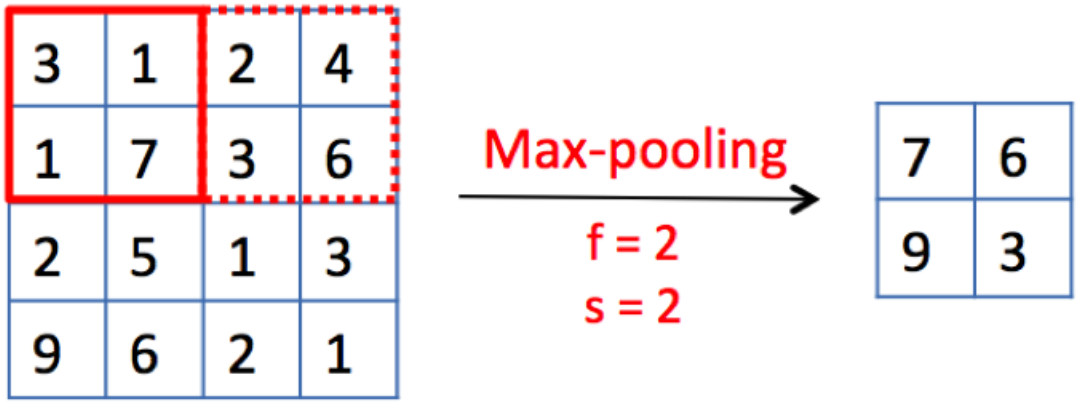
– – – – – – –

POOLING LAYERS

There are two types of pooling layers: max and average pooling.

Max pooling

We define a spatial neighborhood (a filter), and as we slide it through the input, we take the largest element within the region covered by the filter.



Average pooling

As the name suggests, it retains the average of the values encountered within the filter.

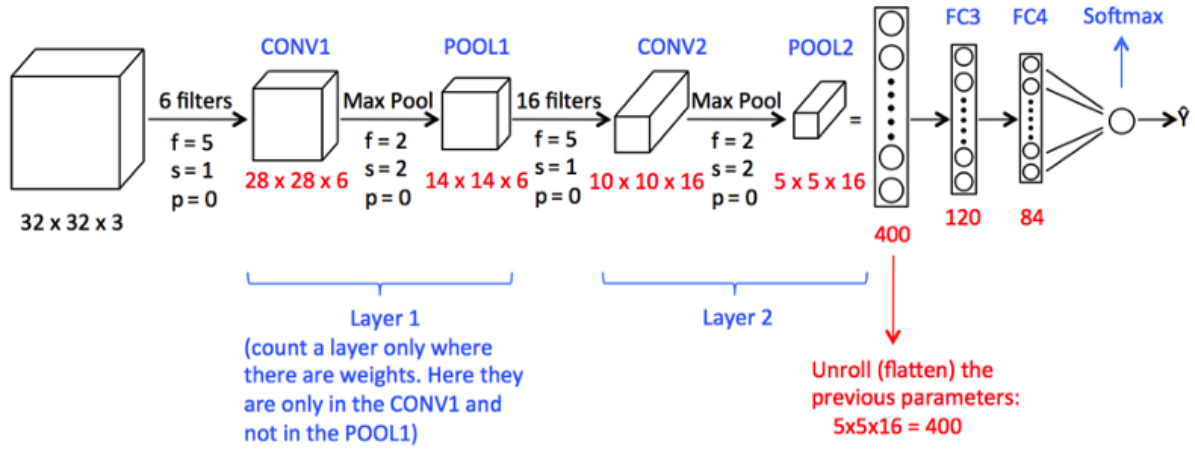
One thing worth noting is the fact that a pooling layer does not have any parameters to learn. Of course, we have hyper-parameters to select, the filter size and the stride (it's common not to use any padding).

– – – – – – –

FULLY CONNECTED LAYER

A fully connected layer acts like a "standard" single neural network layer, where you have a weight matrix W and bias b.

We can see its application in the following example of a Convolutional Neural Network. This network is inspired by the LeNet-5 network:



It's common that, as we go deeper into the network, the sizes (nh, nw) decrease, while the number of channels (nc) increases.

Another common pattern you can see in neural networks is to have CONV layers, one or more, followed by a POOL layer, and then again one or more CONV layers followed by a POOL layer and, at the end, a few FC layers followed by a Softmax.

When choosing the right hyper-parameters (f, s, p, ..), look at the literature and choose an architecture that was successfully used and that can apply to your application. There are several "classic" networks, such as LeNet, AlexNet, VGG, …

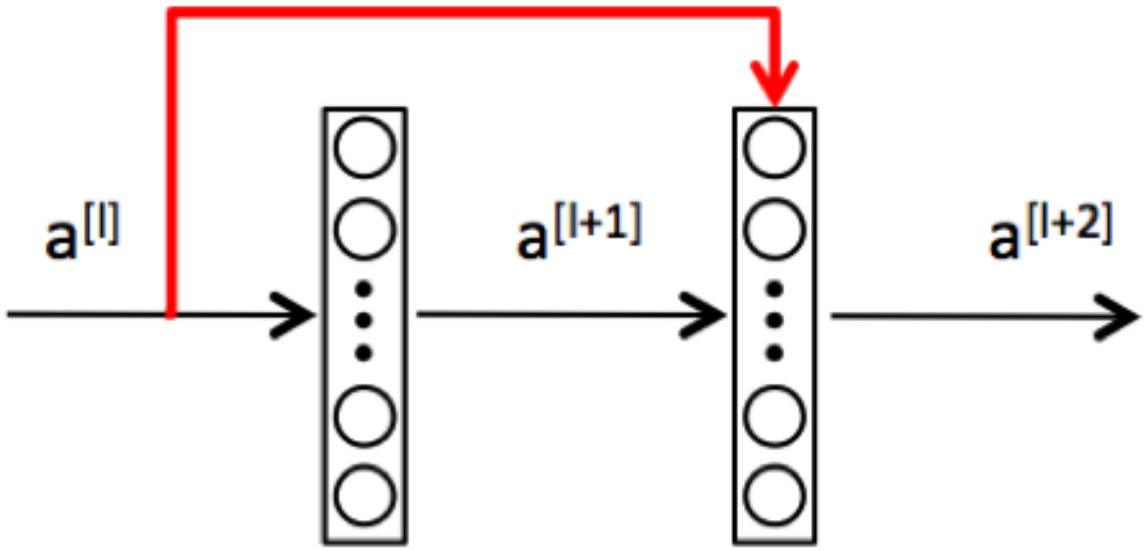
I won't go into details for each one, but you can easily find them online.

These networks are normally used in transfer learning, where we can use the weights coming from the existing trained network and then replace the output unit, since training such a big network from scratch would require a long time otherwise.

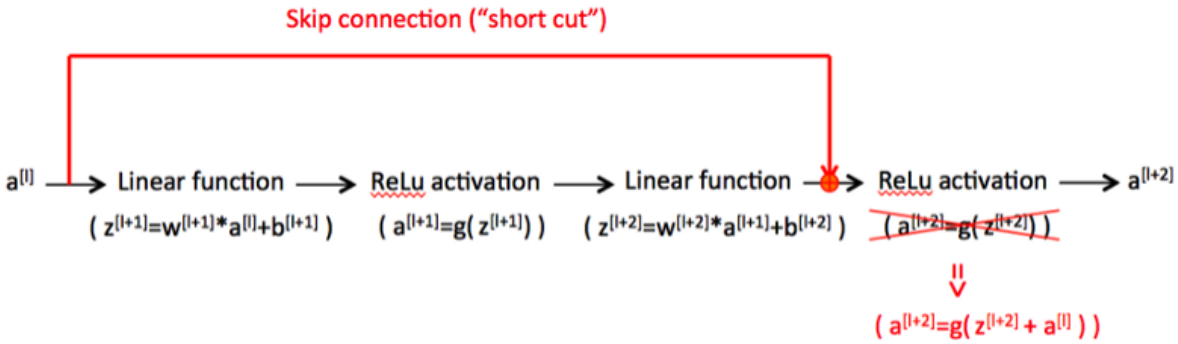
RESIDUAL NETWORKS (RESNETS)

Very deep neural networks are difficult to train because of vanishing and exploding gradients. A solution to that is the use of "skip connections", which allow you to take the activation function from one layer and feed it to another layer even much deeper in the network. Skip connections are the building blocks of a Residual Network (ResNet).

A residual block is sketched as:

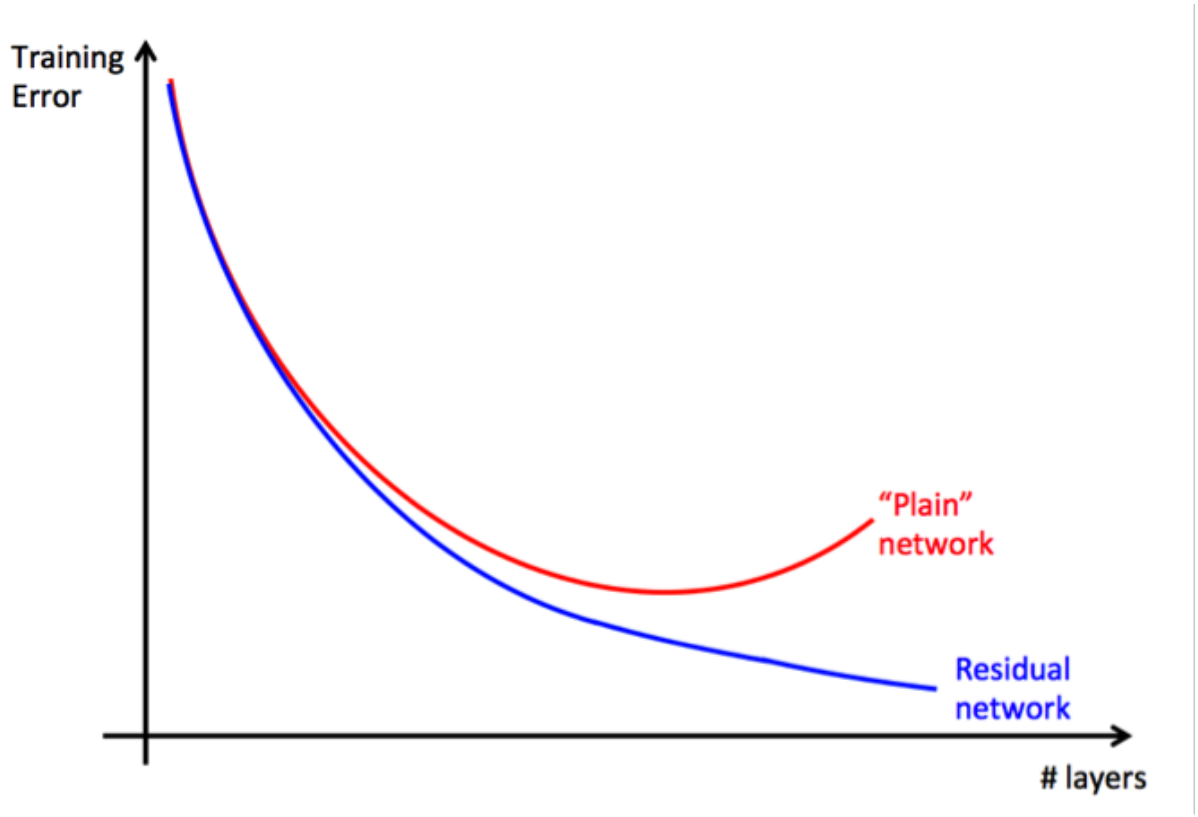


which is computed as:



Therefore, with a residual block, instead of taking the regular, "main" path, we take a[l] and add it to a later layer before applying the non-linearity ReLu.

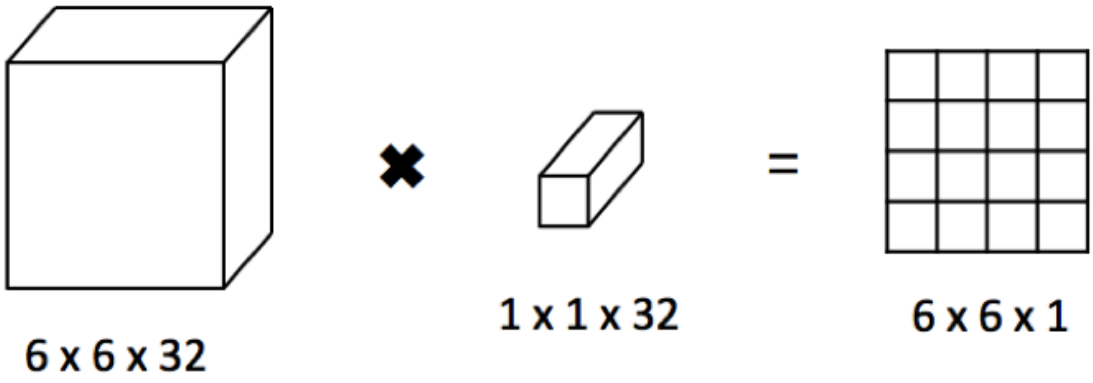
A Residual Network is composed of these residual blocks stacked together to form a deep network. In reality, a "plain", regular network has a harder time to train, as the network gets deeper, so the training error increases. The advantage of using a Residual Network, instead, is to train a very deep network and keep decreasing the training error.



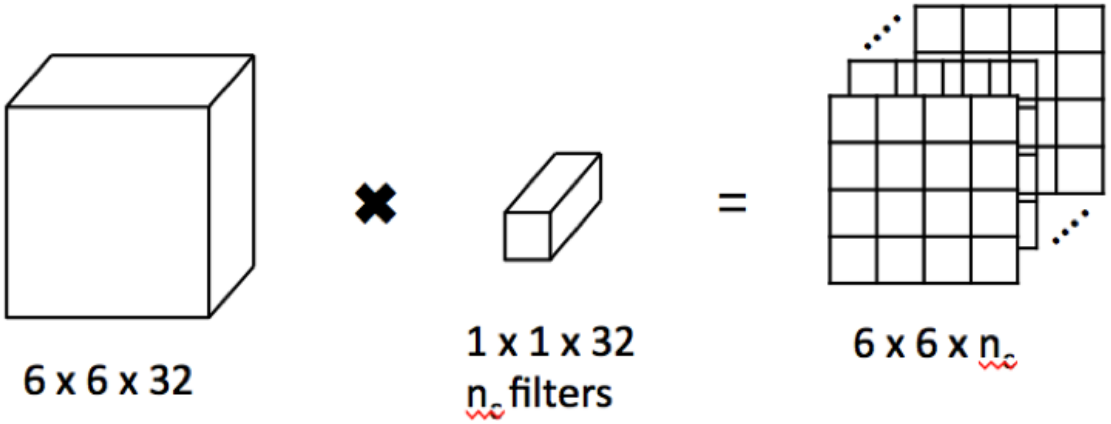
1x1 CONVOLUTIONS (Network-in-Network)

We saw on the previous example of convolutional neural networks that applying a pooling layer we essentially shrink the dimension nh and nw. By applying, instead, a 1x1 convolution, we can shrink the number of channels, therefore save on computation. Furthermore, it adds non-linearity to the network.

Let's see the following example:



Here, the 1x1 convolution looks at each 36 (6x6) different positions, and takes the element-wise product between 32 numbers on the left and 32 number in the filter, and then it applies a ReLU non-linearity to it. If we apply “nc“ number of 1x1 convolutional filters, then the output will be, in our example, 6x6xnc.

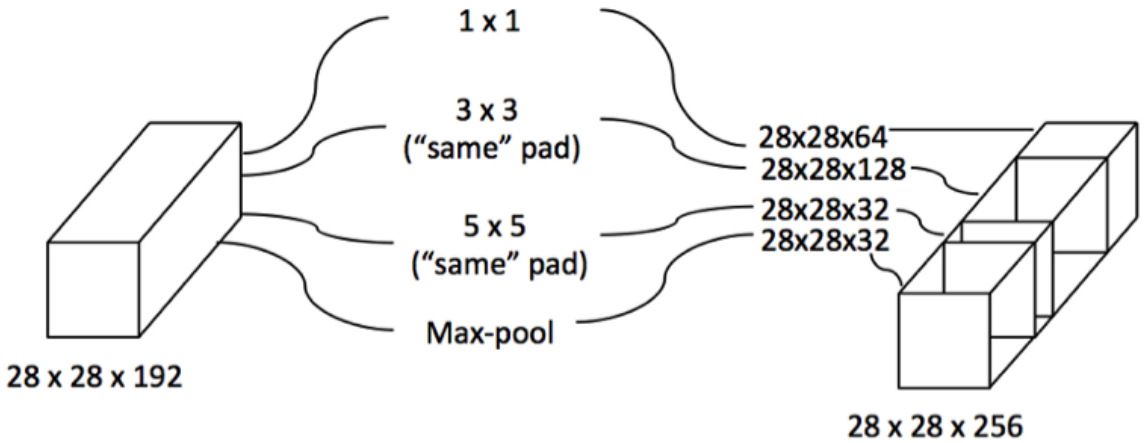


INCEPTION NETWORK

When designing a layer for a ConvNet we need to pick many things: the number of filters, the type of layer (pooling, conv, etc..),… what if we didn’t have to choose, but get them all?

Yes, it sounds pretty greedy!

That’s what the Inception network does. It uses all these options and stacks them up:



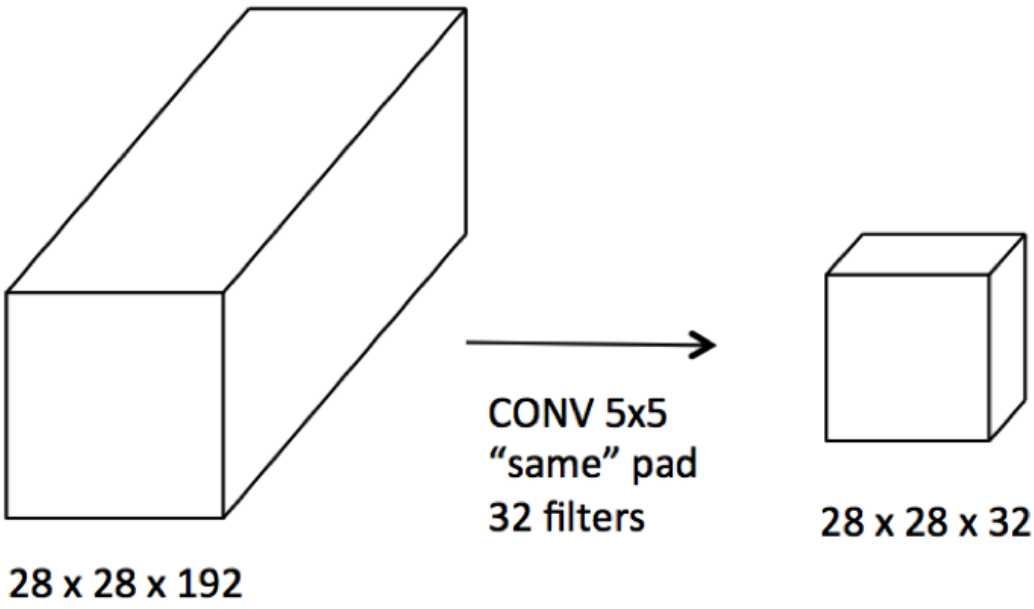
The problem of applying the above architecture is the computational cost.

For example, looking at the computational cost of the 5x5 filter we have:

32 filters and each filter is going to be 5x5x192. The output size is 28x28x32.

So, the total number of multiplications it computes is:

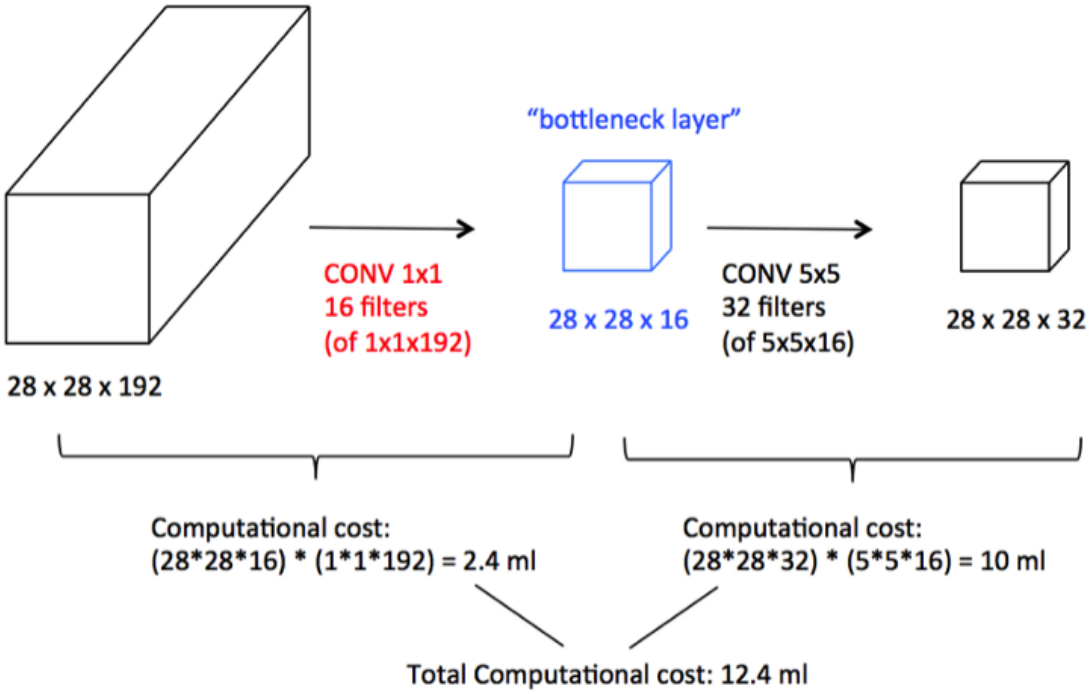
(28*28*32) * (5*5*192) = 120 million!



Fortunately, there is a way to reduce the above number of computations. That comes with some little help from our friend described above, the 1x1 convolution.

The interposed 1x1 convolution reduces by 10 times the total computational cost since we have:

(28*28*16) * (1*1*192) + (28*28*32) * (5*5*16) = 12.4 million



Therefore, an inception network is built interposing 1x1 convolutions on the filters and stacking these modules together.