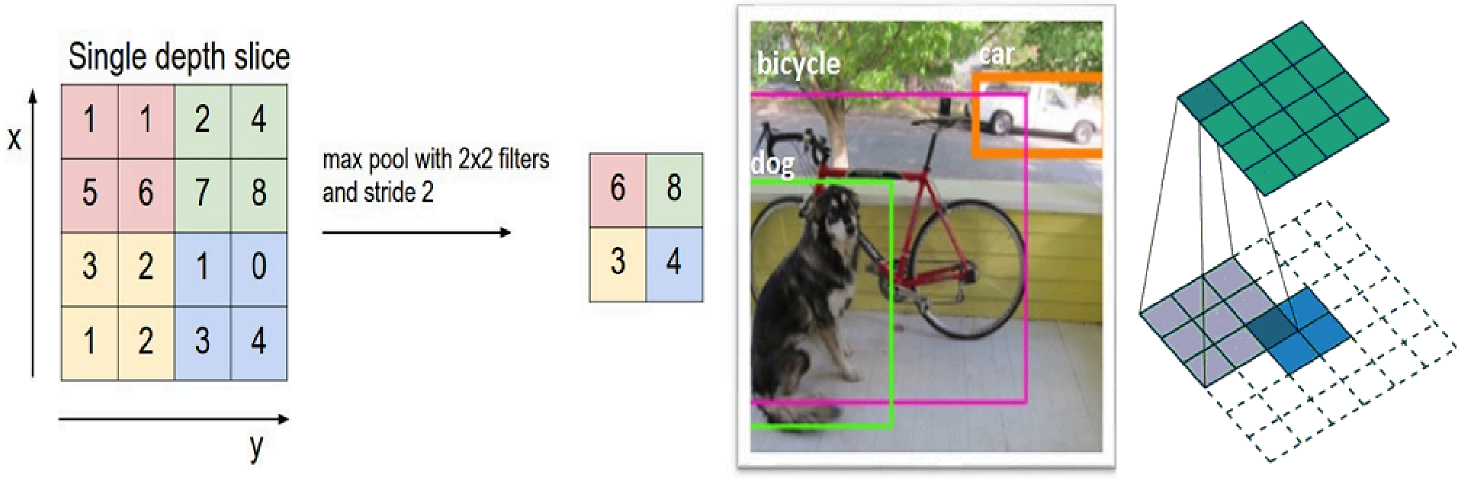
Adit Deshpande (/)

Engineering at Forward I UCLA CS '19

(/)

Blog (/) About (/about) GitHub (https://github.com/adeshpande3) Resume (/resume.pdf)

A Beginner's Guide To Understanding Convolutional Neural Networks Part 2



Introduction

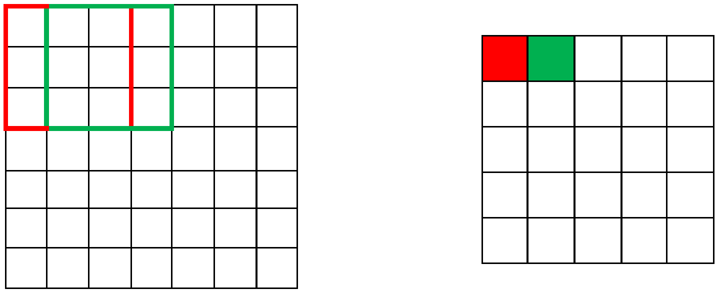
Link to Part 1 (https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/)

In this post, we'll go into a lot more of the specifics of ConvNets. **Disclaimer:** Now, I do realize that some of these topics are quite complex and could be made in whole posts by themselves. In an effort to remain concise yet retain comprehensiveness, I will provide links to research papers where the topic is explained in more detail.

Stride and Padding

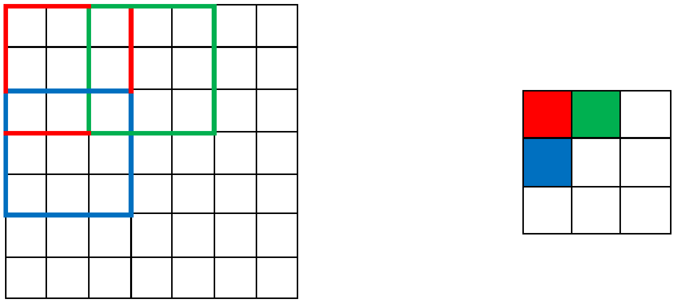
Alright, let's look back at our good old conv layers. Remember the filters, the receptive fields, the convolving? Good. Now, there are 2 main parameters that we can change to modify the behavior of each layer. After we choose the filter size, we also have to choose the **stride** and the **padding**.

Stride controls how the filter convolves around the input volume. In the example we had in part 1, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. Let's look at an example. Let's imagine a 7 x 7 input volume, a 3 x 3 filter (Disregard the 3rd dimension for simplicity), and a stride of 1. This is the case that we're accustomed to.


7 x 7 Input Volume

5 x 5 Output Volume

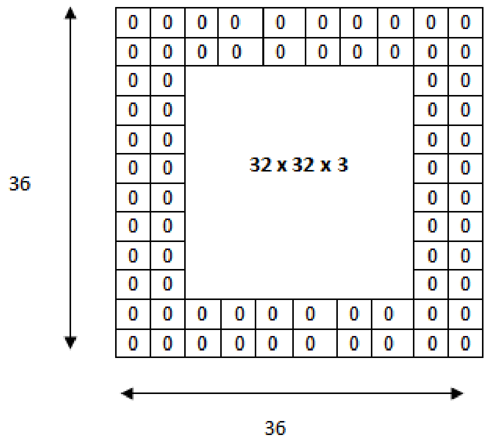
Same old, same old, right? See if you can try to guess what will happen to the output volume as the stride increases to 2.


7 x 7 Input Volume

3 x 3 Output Volume

So, as you can see, the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if we tried to set our stride to 3, then we'd have issues with spacing and making sure the receptive fields fit on the input volume. Normally, programmers will increase the stride if they want receptive fields to overlap less and if they want smaller spatial dimensions.

Now, let's take a look at padding. Before getting into that, let's think about a scenario. What happens when you apply three 5 x 5 x 3 filters to a 32 x 32 x 3 input volume? The output volume would be 28 x 28 x 3. Notice that the spatial dimensions decrease. As we keep applying conv layers, the size of the volume will decrease faster than we would like. In the early layers of our network, we want to preserve as much information about the original input volume so that we can extract those low level features. Let's say we want to apply the same conv layer but we want the output volume to remain 32 x 32 x 3. To do this, we can apply a zero padding of size 2 to that layer. Zero padding pads the input volume with zeros around the border. If we think about a zero padding of two, then this would result in a 36 x 36 x 3 input volume.



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

If you have a stride of 1 and if you set the size of zero padding to

$$Zero\ Padding = \frac{(K-1)}{2}$$

where K is the filter size, then the input and output volume will always have the same spatial dimensions.

The formula for calculating the output size for any given conv layer is

$$O = \frac{(W-K+2P)}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

Choosing Hyperparameters

How do we know how many layers to use, how many conv layers, what are the filter sizes, or the values for stride and padding? These are not trivial questions and there isn't a set standard that is used by all researchers. This is because the network will largely depend on the type of data that you have. Data can vary by size, complexity of the image, type of image processing task, and more. When looking at your dataset, one way to think about how to choose the hyperparameters is to find the right combination that creates abstractions of the image at a proper scale.

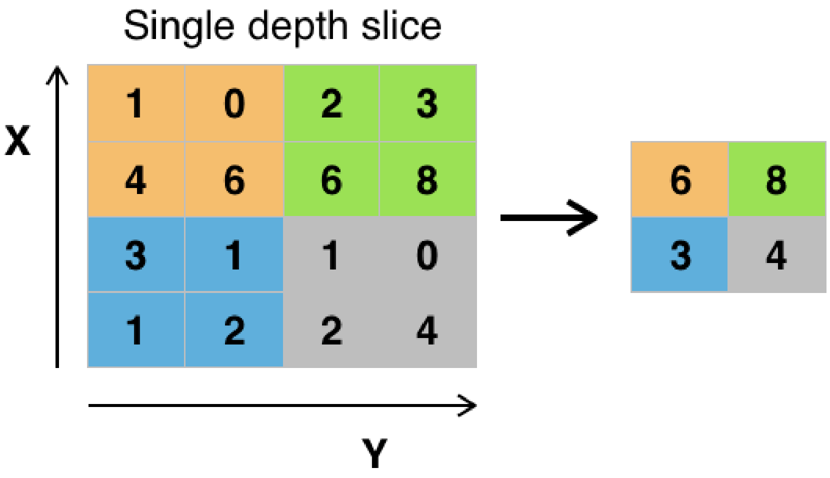
ReLU (Rectified Linear Units) Layers

After each conv layer, it is convention to apply a nonlinear layer (or **activation layer**) immediately afterward.The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that **ReLU layers** work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers (Explaining this might be out of the scope of this post, but see here (https://en.wikipedia.org/wiki/Vanishing_gradient_problem) and here (https://www.quora.com/What-is-the-vanishing-gradient-problem) for good descriptions). The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0.This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.

Paper (http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf) by the great Geoffrey Hinton (aka the father of deep learning).

Pooling Layers

After some ReLU layers, programmers may choose to apply a **pooling layer**. It is also referred to as a downsampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every subregion that the filter convolves around.



Example of Maxpool with a 2x2 filter and a stride of 2

Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the amount of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control **overfitting**. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

Dropout Layers

Now, **dropout layers** have a very specific function in neural networks. In the last section, we discussed the problem of overfitting, where after training, the weights of the network are so tuned to the training examples they are given that the network doesn't perform well when given new examples. The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that I mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviate the overfitting problem. An important note is that this layer is only used during training, and not during test time.

Paper (https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf) by Geoffrey Hinton.

Network in Network Layers

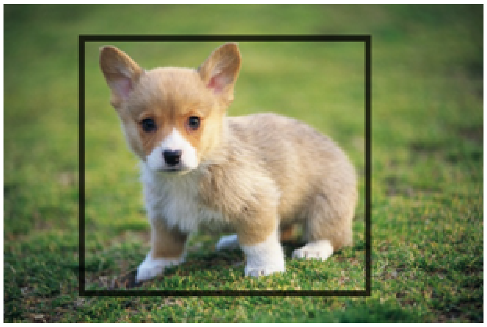
A **network in network** layer refers to a conv layer where a 1 x 1 size filter is used. Now, at first look, you might wonder why this type of layer would even be helpful since receptive fields are normally larger than the space they map to. However, we must remember that these 1x1 convolutions span a certain depth, so we can think of it as a 1 x 1 x N convolution where N is the number of filters applied in the layer. Effectively, this layer is performing a N-D element-wise multiplication where N is the depth of the input volume into the layer.

Paper (https://arxiv.org/pdf/1312.4400v3.pdf) by Min Lin.

Classification, Localization, Detection, Segmentation

In the example we used in Part 1 of this series, we looked at the task of **image classification**. This is the process of taking an input image and outputting a class number out of a set of categories. However, when we take a task like **object localization**, our job is not only to produce a class label but also a bounding box that describes where the object is in the picture.

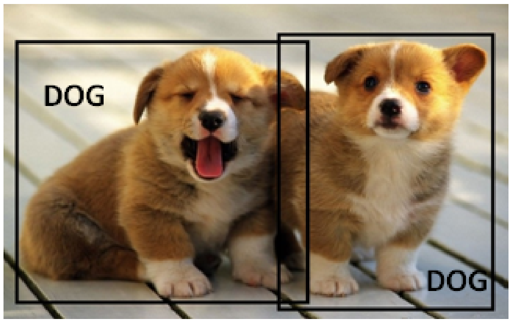
Object Classification is the task of identifying that picture is a dog



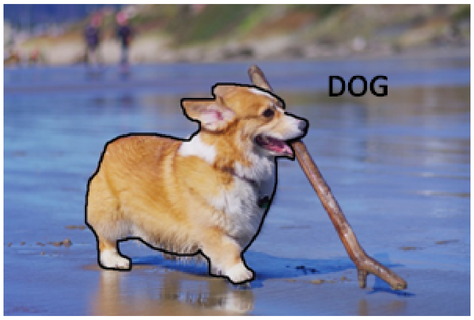
Object Localization involves the class label as well as a bounding box to show where the object is located.

We also have the task of **object detection**, where localization needs to be done on all of the objects in the image. Therefore, you will have multiple bounding boxes and multiple class labels.

Finally, we also have **object segmentation** where the task is to output a class label as well as an outline of every object in the input image.



Object Detection involves localization of multiple objects (doesn't have to be the same class).



Object Segmentation involves the class label as well as an outline of the object in interest.

More detail on how these are implemented to come in Part 3, but for those who can't wait…

Detection/ Localization: RCNN (https://arxiv.org/pdf/1311.2524v5.pdf), Fast RCNN (https://arxiv.org/pdf/1504.08083.pdf), Faster RCNN (http://arxiv.org/pdf/1506.01497v3.pdf), MultiBox (http://www.cv-foundation.org/openaccess/content_cvpr_2014/papers/Erhan_Scalable_Object_Detection_2014_CVPR_paper.pdf), Bayesian Optimization (http://web.eecs.umich.edu/~honglak/cvpr15-cnn-detection.pdf), Multi-region (http://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Gidaris_Object_Detection_via_ICCV_2015_paper.pdf), RCNN Minus R (http://www.robots.ox.ac.uk/~vedaldi/assets/pubs/lenc15rcnn.pdf), Image Windows (http://calvin.inf.ed.ac.uk/wp-content/uploads/Publications/alexe12pami.pdf)
Segmentation: Semantic Seg (http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Long_Fully_Convolutional_Networks_2015_CVPR_paper.pdf), Unconstrained Video (http://calvin.inf.ed.ac.uk/wp-content/uploads/Publications/papazoglouICCV2013-camera-

ready.pdf), Shape Guided (https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/Borenstein06.pdf), Object Regions (http://crcv.ucf.edu/papers/cvpr2013/VideoObjectSegmentation.pdf), Shape Sharing (http://www.cs.utexas.edu/~grauman/papers/shape-sharing-ECCV2012.pdf)

Yeah, there's a lot more.

Transfer Learning

Now, a common misconception in the DL community is that without a Google-esque amount of data, you can't possibly hope to create effective deep learning models. While data is a critical part of creating the network, the idea of transfer learning has helped to lessen the data demands. **Transfer learning** is the process of taking a pre-trained model (the weights and parameters of a network that has been trained on a large dataset by somebody else) and "fine-tuning" the model with your own dataset. The idea is that this pre-trained model will act as a feature extractor. You will remove the last layer of the network and replace it with your own classifier (depending on what your problem space is). You then freeze the weights of all the other layers and train the network normally (Freezing the layers means not changing the weights during gradient descent/optimization).

Let's investigate why this works. Let's say the pre-trained model that we're talking about was trained on ImageNet (For those that aren't familiar, ImageNet is a dataset that contains 14 million images with over 1,000 classes). When we think about the lower layers of the network, we know that they will detect features like edges and curves. Now, unless you have a very unique problem space and dataset, your network is going to need to detect curves and edges as well. Rather than training the whole network through a random initialization of weights, we can use the weights of the pre-trained model (and freeze them) and focus on the more important layers (ones that are higher up) for training. If your dataset is quite different than something like ImageNet, then you'd want to train more of your layers and freeze only a couple of the low layers.

Paper (https://arxiv.org/pdf/1411.1792v1.pdf) by Yoshua Bengio (another deep learning pioneer).
Paper (http://arxiv.org/pdf/1403.6382.pdf) by Ali Sharif Razavian.
Paper (https://arxiv.org/pdf/1310.1531.pdf) by Jeff Donahue.
Paper (https://arxiv.org/pdf/1705.07706.pdf) and subsequent paper (https://arxiv.org/pdf/1707.09872.pdf) by Dario Garcia-Gasulla.

Data Augmentation Techniques

By now, we're all probably numb to the importance of data in ConvNets, so let's talk about ways that you can make your existing dataset even larger, just with a couple easy transformations. Like we've mentioned before, when a computer takes an image as an input, it will take in an array of pixel values. Let's say that the whole image is shifted left by 1 pixel. To you and me, this change is imperceptible. However, to a computer, this shift can be fairly significant as the classification or label of the image doesn't change, while the array does. Approaches that alter the training data in ways that change the array representation while keeping the label the same are known as **data augmentation** techniques. They are a way to artificially expand your dataset. Some popular augmentations people use are grayscales, horizontal flips, vertical flips, random crops, color jitters, translations, rotations, and much more. By applying just a couple of these transformations to your training data, you can easily double or triple the number of training examples.

Link to Part 3 (https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html)

Dueces.

Sources (/assets/Sources2.txt)