CS 189 Introduction to Machine Learning
Fall 2019 Jennifer Listgarten & Stella Yu Note 27

1 Convolutional Neural Networks

Neural networks have been successfully applied to many real-world problems, such as predicting stock prices and learning robot dynamics in autonomous systems. The most general type of neural network is **multilayer perceptrons (MLPs)**, which we have already studied in detail and employed to classify 28×28 pixel images as digits (MNIST). While MLPs can be used to effectively classify small images (such as those in MNIST), they are impractical for large images. Let's see why. Given a $W \times H \times 3$ image (over 3 channels — red, green, blue), an MLP would take the flattened image as input, pass it through several fully connected (FC) layers and non-linearities, and finally output a vector of probabilities for each of the classes.

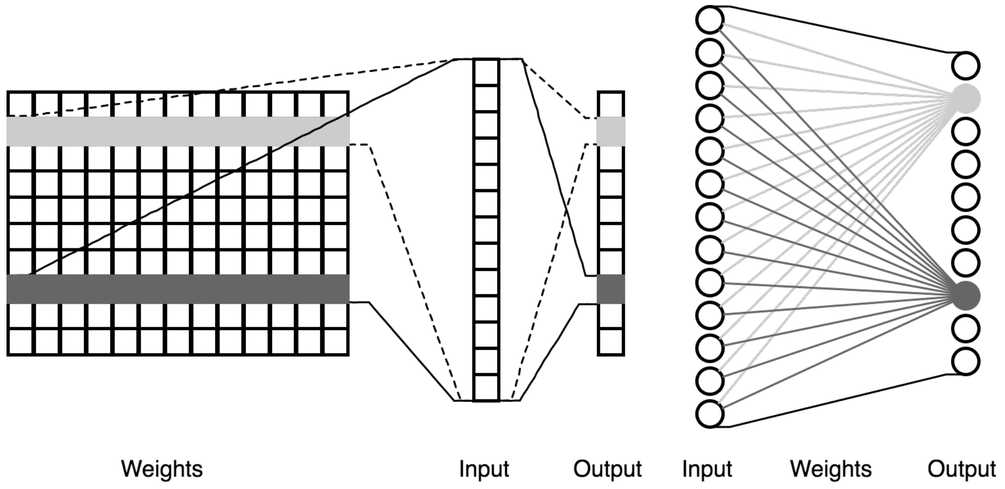


Figure 1: A Fully Connected layer connects every input neuron to every output neuron.

Associated with each FC layer is an $n_i \times n_o$ weight matrix that "connects" each of the n_i input neurons to each of the n_o output neurons, hence the term "fully connected layer". The first FC layer takes an image as input, with $n_i = W \times H \times 3$ input neurons. Assuming that there are $n_o \approx n_i$ output neurons, then there are $n_i \times n_o \approx W^2 \times H^2 \times 3^2$ weights — a prohibitively large number of weights (in the millions)! This analysis extends to all FC layers that have large inputs, not just the first layer. In the framework of image classification, MLPs are generally ineffective — not only are they computationally expensive to train (both in terms of time and memory usage), but they also have high variance due to the large number of weights.

Convolutional neural networks (CNNs, or ConvNets) are a different neural network architecture that significantly reduces the number of weights, and in turn reduces variance. Like MLPs, CNNs use FC layers and non-linearities, but they introduce two new types of layers — convolutional and pooling layers. Let's look at these two layers in detail.

1.1 Convolutional Layers

A **convolutional layer** takes a $W \times H \times D$ dimensional input $\mathbf{I}$ and *convolves* it with a $w \times h \times D$ dimensional **filter (or kernel)** $\mathbf{G}$. The weights of the filter can be hand designed, but in the context of machine learning we tune them automatically, just like we tune the weights of an FC layer. Mathematically, the convolution operator is defined as

$$(\mathbf{I} * \mathbf{G})[x, y] = \sum_{a=0}^{w-1} \sum_{b=0}^{h-1} \sum_{c \in \{1 \cdots D\}} I_c[x + a, y + b] \cdot G_c[a, b]$$

The subscript in I_c indexes into the depth of the image, in this case for depth c. We can view convolution as either:

1. a 2-D operator over the width/height of the image, "broadcast" over the depth
2. a 3-D operator over the weight/height/depth of the image, with the convolution over the depth spanning the whole image with no room to move.

The output $\mathbf{L} = \mathbf{I} * \mathbf{G}$ is an array of $(W - w + 1) \times (H - h + 1) \times 1$ values.

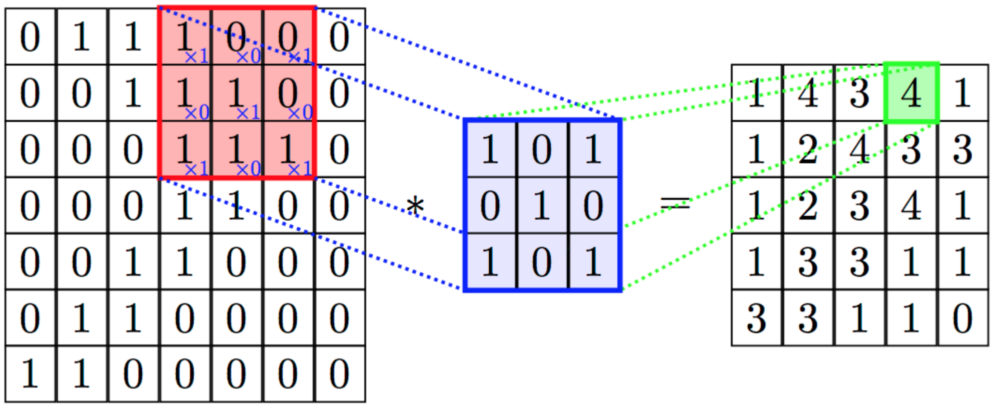


Figure 2: Convolving a filter with an image. In this example, we have $W = H = 7, w = h = 3, D = 1$. We extract $n_o = (W - w + 1) \times (H - h + 1) \times 1 = 25$ output values from $n_i = W \times H \times D = 49$ input values, via a filter with $3 \times 3 = 9$ weights.

What exactly is convolution useful for, and why do we use it in the context of image classification? In simple terms, convolutions help us *extract features*. On a low level, filters can be used to detect all kinds of edges in an image, and at a high level they can detect more complex shapes and objects that are critical to classifying an image. Consider a simple horizontal edge detector filter $[1 \quad -1]$. This filter will produce large negative values for inputs in which the left pixel is bright and the right pixel is dark; conversely, it will produce large positive values for inputs in which the left pixel is dark and the right pixel is bright.

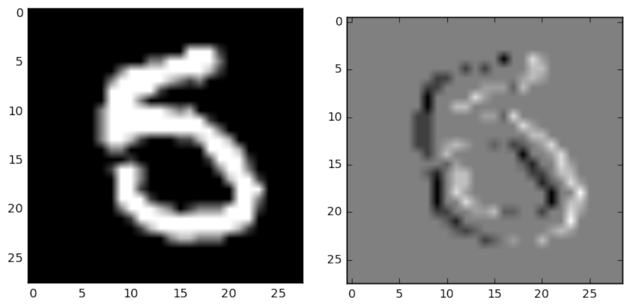


Figure 3: Left: a sample image.
Right: the output of convolving the $[1 \quad -1]$ filter about the image.

In general, a filter will produce large positive values in the areas of the image which appear most similar to it. As another example, here is a filter detects edges at a positive 45-degree angle:

$$\begin{bmatrix} 0.6 & 0.2 & 0 \\ 0.2 & 0 & 0.2 \\ 0 & 0.2 & 0.6 \end{bmatrix}$$

How do conv layers compare to FC layers? Let's revisit the example from figure 2, where the input is $n_i = 49$ units and the output is $n_o = 25$ units. In an FC layer, we would have used $n_i \times n_o = 1225$ weights, but in our conv layer the filter only has 9 weights! conv layers use a significantly smaller number of weights, which reduces the variance of the model significantly while retaining the expressiveness. This is because we make use of *weight sharing*:

1. the same weights are shared among all the pixels of the input

2. the individual units of the output layer are all determined by the same weights

Compare this to the fully-connected architecture where for each output unit, there is a separate weight to learn for each input-ouput weight. We can illustrate the point for a simple 1-D input-output example.

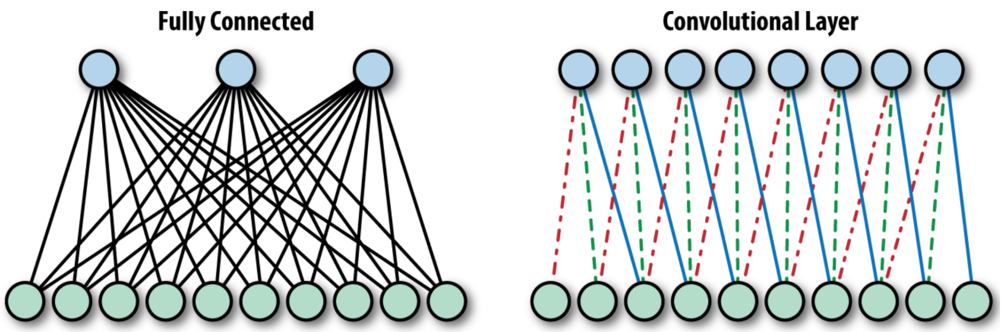


Figure 4: FC vs. conv layer. Conv layers are equivalent to FC layers, except that (1) all weights outside the receptive field are 0, and (2) the weights are shared.

This architecture not only decreases the complexity of our model (there are fewer weights), it is actually reasonable for image processing because there are repeated patterns in images — ie. a filter that can detect some kind of pattern in one area of the image can be used elsewhere in the image to detect the same pattern.

In practice, we can apply several different filters to the image to detect different patterns in the input image. For example, we can use a filter that detects horizontal edges, one that detects vertical edges, and another that detects diagonal edges all at once. Given a $W \times H \times D$ input image and k separate $w \times h \times D$ filters, each filter produces an $(W-w+1) \times (H-h+1) \times 1$ dimensional output. These individual outputs are stacked together for a $(W-w+1) \times (H-h+1) \times k$ combined output.

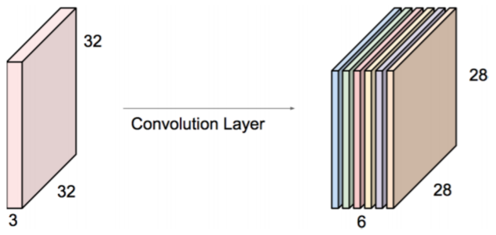
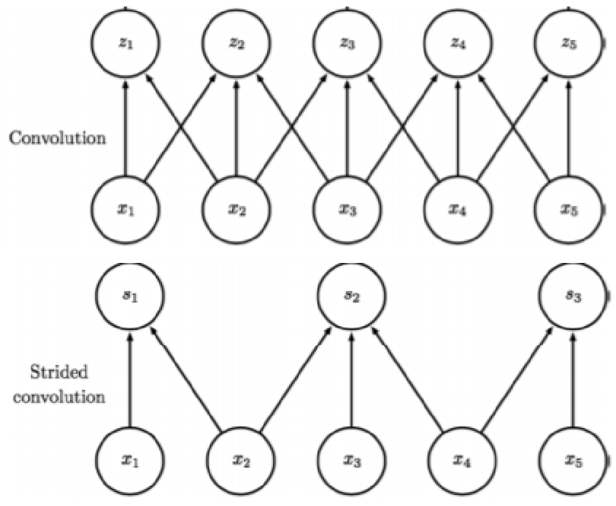


Figure 5: Here, we slid 6 independent $5 \times 5 \times 3$ filters across the original image to produce 6 activation maps in the next convolutional layer.

Stacking filters can incur high computational costs, and in order to mitigate this issue, we can stride our filter across the image by multiple pixels instead:



In conjunction to striding, zero-padding the borders of the image is sometimes used to control the exact dimensions of the convolutional layer.

So far, we have introduced convolutional layers as an intuitive and effective approach to extracting features from images, but one potential inspection a disadvantage is that they can only detect "local" features, which is not sufficient to capture complex, global patterns in images. This is not actually the case, because as we stack more convolutional layers, the effective **receptive field** of each successive layer increases. That is, as we go downstream (of the layers), the value of any single unit is informed by an increasingly large patch of the original image. For example, if we use two successive layers of 3×3 filters, any one unit in the first convolutional layer is informed by 9 separate image pixels. Any one unit in the second convolutional layer is informed by 9 separate units of the first convolutional layer, which could informed by up to $9 \times 9 = 81$ original pixels. The increasing receptive field of the successive layers means that the filters in the first few layers extract local low level features, and the later layers extract global high level features.

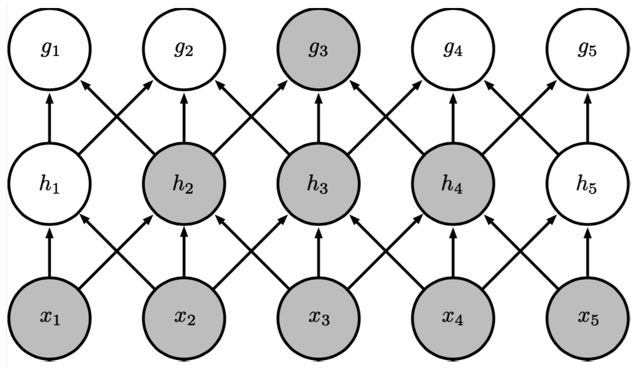


Figure 6: The highlighted unit in the downstream layer uses information from all the highlighted units in the input layer.

1.2 Pooling Layers

In line with convolutional layers reducing the number of weights in neural networks to reduce variance, **pooling layers** directly reduce the number of neurons in neural networks. The sole purpose of a pooling layer is to *downsample* (also known as *pool*, *gather*, *consolidate*) the previous layer, by sliding a fixed window across a layer and choosing one value that effectively "represents" all of the units captured by the window. There are two common implementations of pooling. In max-pooling, the representative value just becomes the largest of all the units in the window, while in average-pooling, the representative value is the average of all the units in the window. In practice, we stride pooling layers across the image with the stride equal to the size of the pooling layer. None of these properties actually involve any weights, unlike fully connected and convolutional layers.

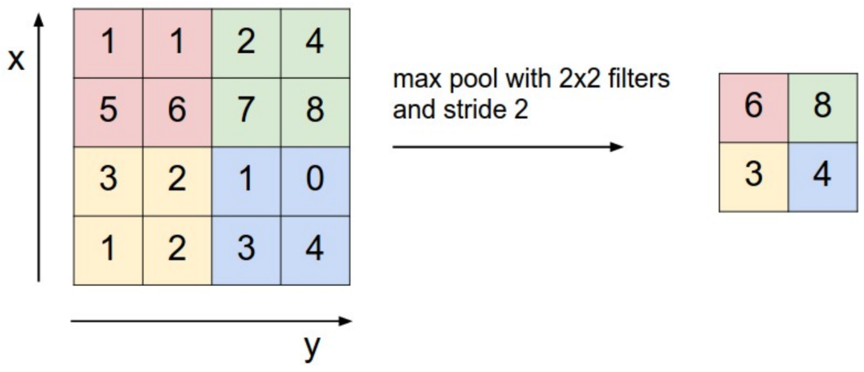


Figure 7: Max-pooling layer

Orthogonal to the choice between max and average pooling is option between spatial and cross-channel pooling. Spatial pooling pools values within the same channel, which induces translational invariance in our model and adding generalization capabilities. In the following figure, we can see that even though the input layer of the right image is a translated version of the input layer of the left image, due to spacial pooling the next layer looks more or less the same.

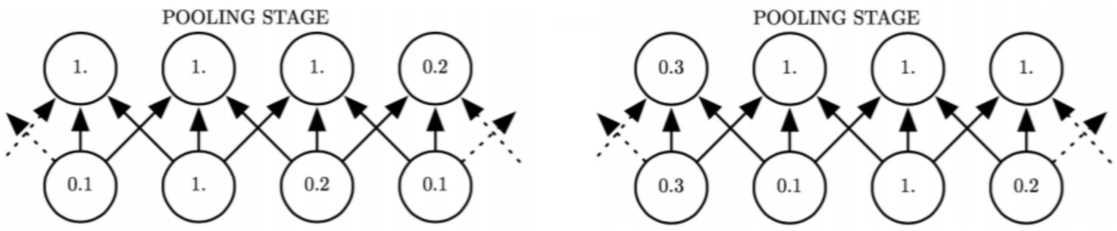


Figure 8: Spatial pooling

Cross-channel pooling pools values across different channels, which induces transformational invariance in our model, again adding generalization capabilities. To illustrate the point, consider an example with a convolutional layer represented by 3 filters. Suppose each can detect the number 5 in some degree of rotation. If we pooled across the three channels determined by these filters, then no matter what orientation of the number "5" we got as input to our CNN, the pooling layer would have a large response!

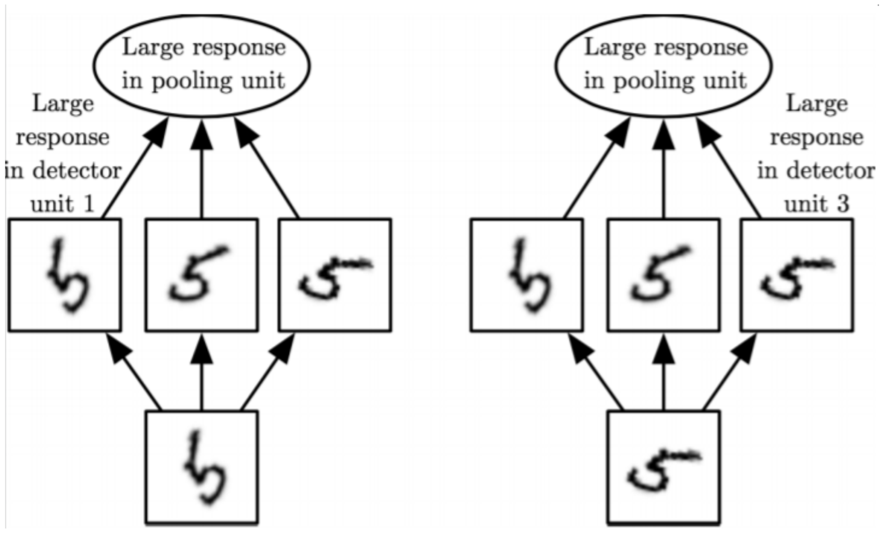

Large response in pooling unit
Large response in detector unit 1
Large response in pooling unit
Large response in detector unit 3

Figure 9: Cross-channel pooling

1.3 Backpropagation for CNNs

Just like MLPs, we can use the Backpropagation algorithm to train CNNs as well. We simply have to compute partial derivatives for conv and pool layers, just as we did for FC layers and non-linearities.

1.3.1 Derivatives for conv layers

Let's denote the error function as f. In classification tasks, this is typically cross entropy loss. The forward pass will gives us the input I to the CNN. The backward pass will compute the partial derivatives of the the error f with respect the output of the layer L, $\frac{\partial f}{\partial L}$. Without additional knowledge about the error function after L, we can compute the derivatives with respect to elements in the input I and filter G using the chain rule. Specifically, we have

$$\frac{\partial f}{\partial G_c[x,y]} = \frac{\partial f}{\partial L}\frac{\partial L}{\partial G_c[x,y]}$$

and

$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L}\frac{\partial L}{\partial I_c[x,y]}$$

where $G_c[x,y]$ denotes the entry in the filter for color c at position (x,y) and similarly for $I_c[x,y]$.

From the equation for discrete convolution, we can compute the derivatives for each entry (i,j) in L as

$$\begin{aligned}\frac{\partial L[i,j]}{\partial G_c[x,y]} &= \frac{\partial}{\partial G_c[x,y]}\sum_{a=1}^{w}\sum_{b=1}^{h}\sum_{c\in\{r,g,b\}} I_c[i+a,j+b]\cdot G_c[a,b] \\ &= I_c[i+x,j+y]\end{aligned}$$

For the input image, we similarly compute the derivative as

$$\begin{aligned}\frac{\partial L[i,j]}{\partial I_c[x,y]} &= \frac{\partial}{\partial I_c[x,y]}\sum_{a=1}^{w}\sum_{b=1}^{h}\sum_{c\in\{r,g,b\}} I_c[i+a,j+b]\cdot G_c[a,b] \\ &= G_c[x-i,y-j]\end{aligned}$$

where we have $i+a=x$ and $j+b=y$. When $x-i$ or $y-j$ go outside the boundary of the filter, we can treat the derivative as zero.

We can collect the derivatives of the filter parameter for all $L[i,j]$ into a vector and multiply it by the derivatives we computed to get

$$\frac{\partial f}{\partial G_c[x,y]} = \frac{\partial f}{\partial L}\cdot\frac{\partial L}{\partial G_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}\frac{\partial L[i,j]}{\partial G_c[x,y]}] = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}I_c[i+x,j+y]$$

and for the image

$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L}\cdot\frac{\partial L}{\partial I_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}\frac{\partial L[i,j]}{\partial I_c[x,y]} = \sum_{i,j}\frac{\partial f}{\partial L[i,j]}G_c[x-i,y-j]$$

1.3.2 Derivatives for pool layers

Since pooling layers do not involve any weights, we only need to calculate partial derivatives with respect to the input:

$$\frac{\partial f}{\partial I_c[x,y]}$$

Through the chain rule, we have that

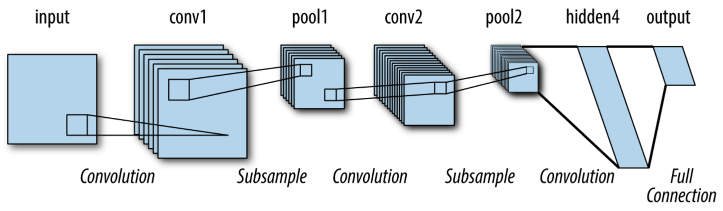
$$\frac{\partial f}{\partial I_c[x,y]} = \frac{\partial f}{\partial L} \cdot \frac{\partial L}{\partial I_c[x,y]}$$

and now the problem entails finding $\frac{\partial L}{\partial I_c[x,y]}$. Computing this derivative depends on the stride, orientation, and nature of the pooling, but in the case of max-pooling the output is simply a maximum of inputs: $L = \max(I_1, I_2, ..., I_n)$ and in this case, we have $\frac{\partial L}{\partial I_j} = \mathbb{1}(I_j = \max(I_1, I_2, ..., I_n))$.

2 CNN Architectures

Convolutional Neural Networks were first applied successfully to the ImageNet challenge in 2012 and continue to outperform computer vision techniques that do not use neural networks. Here are a few of the architectures that have been developed over the years.

2.1 LeNet (LeCun et al, 1998)



Key characteristics:

- Used to classify handwritten alphanumeric characters

2.2 AlexNet (Krizhevsky et al, 2012)

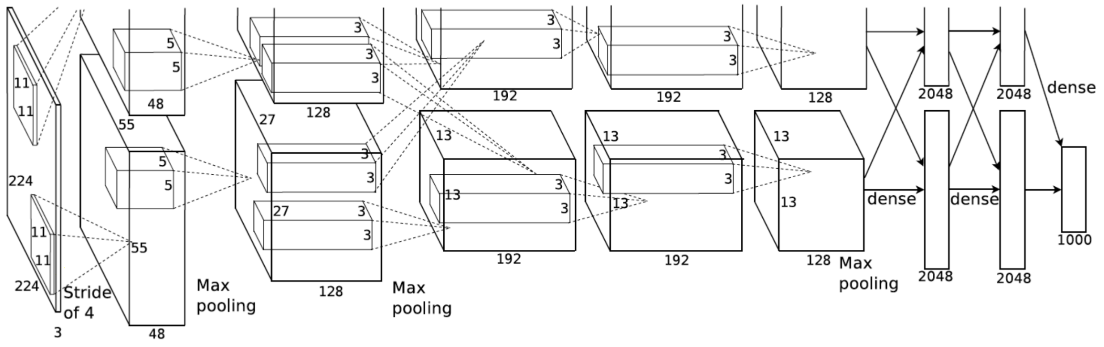


Figure 10: AlexNet architecture. Reference: "ImageNet Classification with Deep Convolutional Neural Networks," NIPS 2012.

Key characteristics:

- Conv filters of varying sizes - for example, the first layer has 11×11 conv filters
- First use of ReLU, which fixed the problem of saturating gradients in the predominant tanh activation.
- Several layers of convolution, max pooling, some normalization. Three fully connected layers at the end of the network (these comprise the majority of the weights in the network).
- Around 60 million weights, over half of which are in the first fully connected layer following the last convolution.
- Trained over two GPU's — the top and bottom divisions in Figure 10 were due to the need to separate training onto two GPU's. There was limited communication between the GPU's, as illustrated by the arrows that go between the top and bottom.
- Dropout in first two FC layers — prevents overfitting
- Heavy data augmentation. One form is image translation and reflection: for example, an elephant facing the left is the same class as an elephant facing the right. The second form is altering the intensity of RGB color channels: different cameras can have different lighting on the same objects, so it is necessary to account for this.

2.3 VGGNet (Simonyan and Zisserman, 2014)

Reference paper: "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR 2015.[1] Key characteristics:

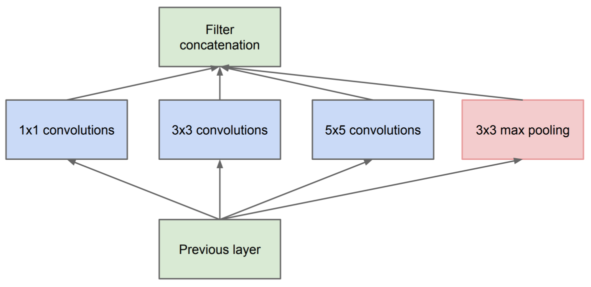
- Only uses 3×3 convolutional filters. Blocks of conv-conv-conv-pool layers are stacked together, followed by fully connected layers at the end (the number of convolutional layers between pooling layers can vary). Note that a stack of 3 3×3 conv filters has the same effective receptive field as one 7×7 conv filter. To see this, imagine sliding a 3×3 filter over a 7×7 image - the result is a 5×5 image. Do this twice more and the result is a 1×1 cell - sliding one 7×7 filter over the original image would also result in a 1×1 cell. The computational cost of the 3×3 filters is lower - a stack of 3 such filters over C channels requires $3 * (3^2 C)$ weights (not including bias weights), while one 7×7 filter would incur a higher cost of $7^2 C$ learned weights. Deeper, more narrow networks can introduce more non-linearities than shallower, wider networks due to the repeated composition of activation functions.

2.4 GoogLeNet (Szegedy et al, 2014)

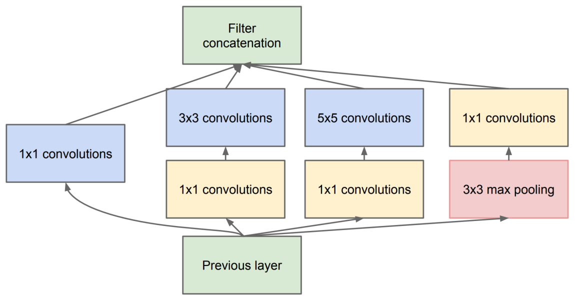
Also codenamed as "Inception."[2] Published in CVPR 2015 as "Going Deeper with Convolutions." Key characteristics:

[1] VGG stands for the "Visual Geometry Group" at Oxford where this was developed.

[2] "In this paper, we will focus on an efficient deep neural network architecture for computer vision, codenamed Inception, which derives its name from the Network in network paper by Lin et al [12] in conjunction with the famous we need to go deeper internet meme [1]." The authors seem to be meme-friendly.



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Figure 11: Inception Module

- Deeper than previous networks (22 layers), but more computationally efficient (5 million parameters - no fully connected layers).
- Network is composed of stacked sub-networks called "Inception modules." The naive Inception module (a) runs convolutional layers in parallel and concatenates the filters together. However, this can be computationally inefficient. The dimensionality reduction Inception module (b) performs 1×1 convolutions that act as dimensionality reduction. This lowers the computational cost and makes it tractable to stack many Inception modules together.

2.5 ResNet (He et al, 2015)

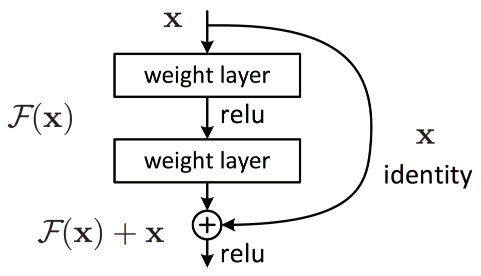


Figure 12: Building block for the ResNet from "Deep Residual Learning for Image Recognition," CVPR 2016. If the desired function to be learned is $\mathcal{H}(x)$, we instead learn the residual $\mathcal{F}(x) := \mathcal{H}(x) - x$, so the output of the network is $\mathcal{F}(x) + x = \mathcal{H}(x)$.

Key characteristics:

- Very deep (152 layers). Residual blocks (Figure 12) are stacked together - each individual weight layer in the residual block is implemented as a 3×3 convolution. There are no FC layers until the final layer.
- Residual blocks solve the "vanishing gradient" problem: the gradient signal diminishes in layers that are farther away from the end of the network. Let L be the loss, Y be the output at a layer, x be the input. Regular neural networks have gradients that look like

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial x}$$

but the derivative of Y with respect to x can be small. If we use a residual block where $Y = F(x) + x$, we have

$$\frac{\partial Y}{\partial x} = \frac{\partial F(x)}{\partial x} + 1$$

The $+x$ term in the residual block always provides some default gradient signal so the signal is still backpropagated to the front of the network. This allows the network to be very deep.

To conclude this section, we note that the winning ImageNet architectures have all increased in depth over the years. While both shallow and deep neural networks are known to be universal function approximators, there is growing empirical and theoretical evidence that deep neural networks can require fewer (even exponentially fewer) parameters than shallow nets to achieve the same approximation performance. There is also evidence that deep neural networks possess better generalization capabilities than their shallow counterparts. The performance, generalization, and optimization benefits of adding more layers is an ongoing component of theoretical research.

3 Visualizing and Understanding CNNs

We know that a convolutional net learns features, but these may not be directly useful to visualize. There are several methods available that enable us to better understand what convolutional nets actually learn. These include:

- Visualizing filters - can give an idea of what types of features the network learns, such as edge detectors. This only works in the first layer. Visualizing activations - can see sparsity in the responses as the depth increases. One can also visualize the feature map before a fully connected layer by conducting a nearest neighbor search in feature space. This helps to determine if the features learned by the CNN are useful - for example, in pixel space, an elephant on the left side of the image would not be a neighbor of an elephant on the right side of the image, but in a translation-invariant feature space these pictures might be neighbors.
- Reconstruction by deconvolution - isolate an activation and reconstruct the original image based on that activation alone to determine its effect.
- Activation maximization - Hubel and Wiesel's experiment, but computationally
- Saliency maps - find what locations in the image make a neuron fire
- Code inversion - given a feature representation, determine the original image
- Semantic interpretation - interpret the activations semantically (for example, is the CNN determining whether or not an object is shiny when it is trying to classify?)