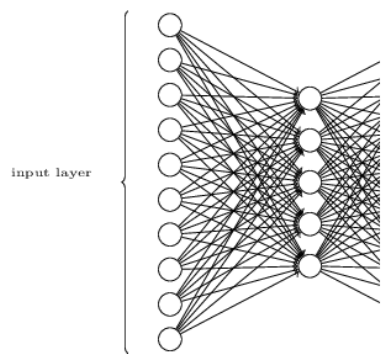
1 Convolutional Neural Nets

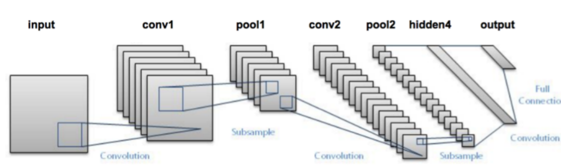
A **convolutional neural net** is just like a regular neural net, except more complicated (convoluted). But in some ways, a convolutional neural net can actually be a simpler model than a fully-connected neural net.

We'll be talking about convolutional neural networks (**ConvNets**) mostly in the framework of image classification. First, let's recall what one layer of a regular neural net looks like:



Remember that each layer consists of units which represent a single number. The values on the units of some layer are determined by the values of the units behind them and the weights on the edges in between the layers. More specifically, the vector of numbers representing layer 2, say y, can be represented by linear combinations Wx of the vector representing layer 1. If we were dealing with images, each unit of the input layer might be the intensity of a single *pixel* of the image.

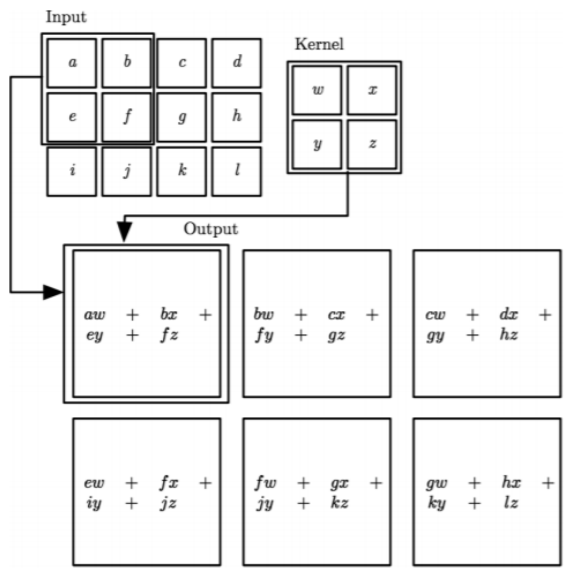
The fully-connected architecture of units has a *lot* of weights to learn. For just a small 32×32 image, there would be 1024 input units and at least as many weights just between the first two layers. Now, let's look at how a ConvNet which deals with images uses its weights differently.



Notice that some of the layers are called "conv" and "pool." These layers are the new ideas that ConvNets introduce to the neural net architecture.

1.1 Convolutional Layers

A **convolutional layer** is determined by **convolving** a **kernel** about the previous layer. The kernel is just a fancy name for an array of weights, and convolving means that we slide the array of weights across the pixels of the previous layer and compute the sum of the elementwise products (kind of like a 2D dot-product). Here is a picture that illustrates this:



In the above picture, we extracted 6 activation values from 12 input values (we would supposedly pass the dot-products through some kind of nonlinearity as well). In a regular fully-connected neural net, we would have used $6 \times 12 = 72$ weights to accomplish this. However, in this convolutional layer, we only used 4 weights. This is because we made use of **weight sharing**, as in:

1. the weights w, x, y, z were shared among all the pixels of the input
2. the individual units of the output layer were all determined by the same weights w, x, y, z

Compare this to the fully-connected architecture where for each output unit, there is a separate weight to learn for each input unit. This kind of strategy decreases the complexity of our model (there are fewer weights), but it makes sense for image processing because there are lots of repeated patterns in images, and if you have one kernel which can detect some kind of phenomenon, then it would make sense to use it elsewhere in the image.

How do kernels detect things, anyways? The short answer is: they will produce large values in the areas of the image which appear most similar to them. Consider a simple kernel $[1 \quad -1]$. This kernel will have produce large values for which the left pixel is bright and the right pixel is dark. Conversely, it will produce small values for which the left pixel is dark and the right pixel is bright.

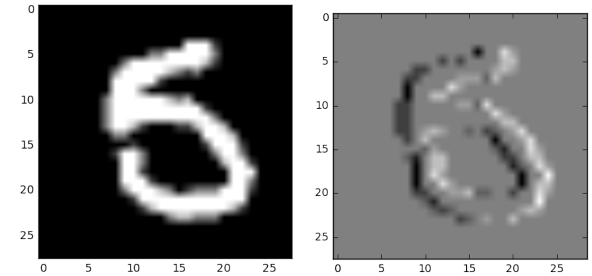


Figure 1: Left: a sample image.
Right: the output of convolving the $[1 \quad -1]$ kernel about the image.

Notice how the output image has high values (white) in the areas where the original image turned from bright to dark (like the right hand side of the figure), and it has low values (black) in the areas where the original image turned from dark to bright (like the left hand side of the figure). This kernel can be thought of as a simple edge detector! As another example, consider the kernel:

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

If this was convolved about an image, it would detect edges at a positive 45-degree angle.

Just a few more things on convolutional layers:

1. You can stack the outputs of multiple kernels together to form a convolutional layer.

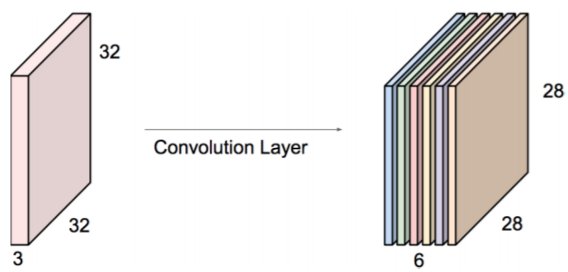
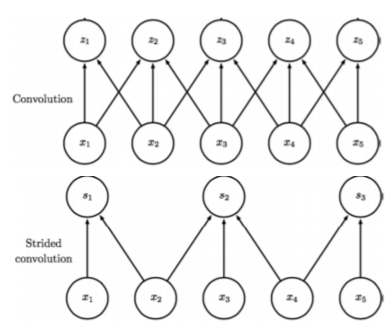


Figure 2: Here, we slid 6 separate $5 \times 5 \times 3$ kernels across the original image to produce 6 activation maps in the next convolutional layer. Each activation map can also be referred to as a **channel**.

2. To save memory, you can have your kernel stride across the image by multiple pixels.



3. Zero-padding is sometimes used to control the exact dimensions of the convolutional layer.

4. As you add more convolutional layers, the effective **receptive field** of each successive layer increases. That is to say, as you 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 you use two successive layers of 3×3 kernels, 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.

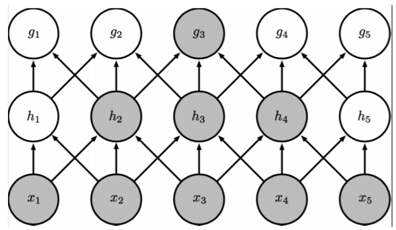


Figure 3: The highlighted unit in the downstream layer uses information from all the highlighted units in the input layer.

5. You can think of convolutional layers as a *complexity-regularized version* of fully-connected layers.

1.2 Pooling Layers

A **pooling layer** does not involve more weights. Instead, it is a layer whose purpose is solely to downsample AKA pool AKA gather AKA consolidate the previous layer. It does this by sliding a window of some size across the units of a layer of the ConvNet and choosing (somehow) one value to effectively "represent" all the units captured by the window. You can tweak the nature of a pooling layer in a few orthogonal ways.

1. *How to pool?* In max-pooling, the representative value just becomes the largest of all the units in the window. In average-pooling, the representative value is the average of all the units in the window.

2. *In which direction to pool?*

   (a) Spatial pooling pools values within the same channel. This has the capability of introducing translational invariance to your model.

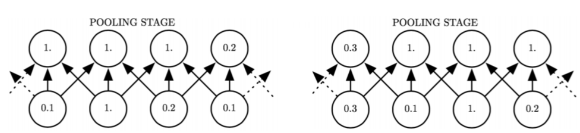


Figure 4: Here, the input layer of the right image is a translated version of the input layer of the left image, but because of max-pooling, the next layer looks more or less the same.

(b) Cross-channel pooling pools values across different channels. This has the capability of introducing transformational invariance to your model.

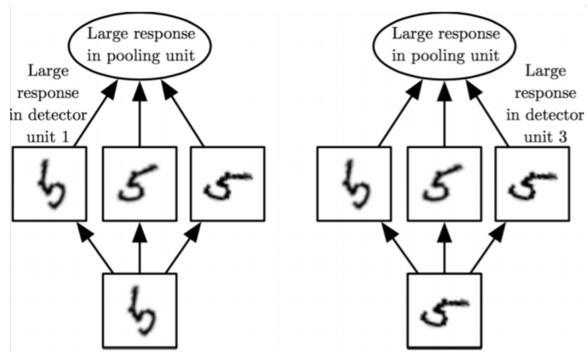


Figure 5: Here, we have an example where our convolutional layer is represented by 3 kernels. Suppose they were each good for detecting the number 5 in some degree of rotation. If we pooled across the three channels determined by these kernels, then no matter what orientation of the number "5" we got as input to our ConvNet, the pooling layer would have a large response!

3. *"Lossiness" of pooling.* This is determined by the stride of the pooling window. If you stride by a large amount, you potentially lose more information, but you conserve memory.

If you now look back at the picture of the ConvNet near the beginning of this note, you should have a better idea of what each layer is doing. The ConvNet in that picture is Yann Lecun's **LeNet**, which is used to classify handwritten alphanumeric characters!

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 AlexNet (Krizhevsky et al, 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.

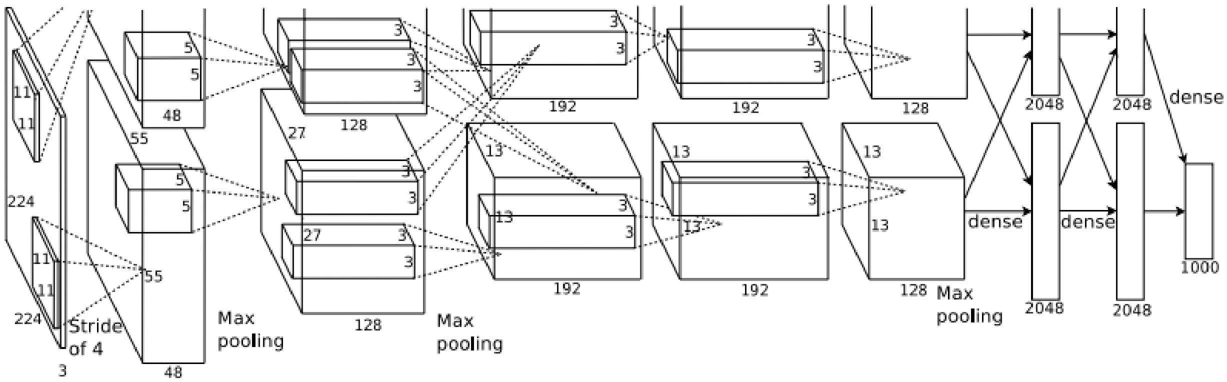


Figure 6: AlexNet architecture. Reference: "ImageNet Classification with Deep Convolutional Neural Networks," NIPS 2012.

- Trained over two GPU's - the top and bottom divisions in Figure **??** 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.

2.2 VGGNet (Simonyan and Zisserman, 2014)

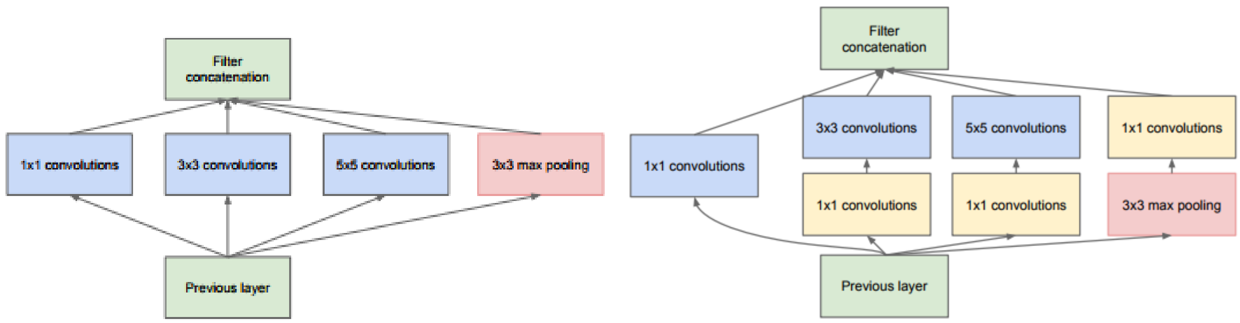
Reference paper: "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR 2015.[1] Key characteristics:

- 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.

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

2.3 GoogLeNet (Szegedy et al, 2014)

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



(a) Inception module, naïve version (b) Inception module with dimensionality reduction

Figure 7: 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.

2.4 ResNet (He et al, 2015)

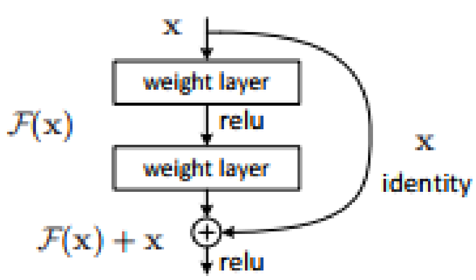


Figure 8: Building block for the ResNet from "Deep Residual Learning for Image Recognition," CVPR 2016. If the desired function to be learned is $\mathscr{H}(x)$, we instead learn the residual $\mathscr{F}(x) := \mathscr{H}(x) - x$, so the output of the network is $\mathscr{F}(x) + x = \mathscr{H}(x)$.

Key characteristics:

[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.

- Very deep (152 layers). Residual blocks (Figure **??**) 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 Towards an Understanding of Convolutional Nets

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?)

See Stella's slides for images of these techniques in practice.