# Convolutional Neural Networks: The Network

Kevin Fu*

January 2019

## 1 Introduction

Last week, we went in-depth on how a convolution works. In this lecture, we'll apply our knowledge of convolutions to see how convolutional layers are actually implemented in a CNN.

## 2 Convolutional vs. Fully Connected Layers

Recall that the reason we chose convolutional layers over fully-connected ones initially was to take advantage of spatial structuring in an image—the idea that pixels close together are more correlated than pixels far apart. By sliding a *kernel* across an image, we consider only small regions of an image at a time.

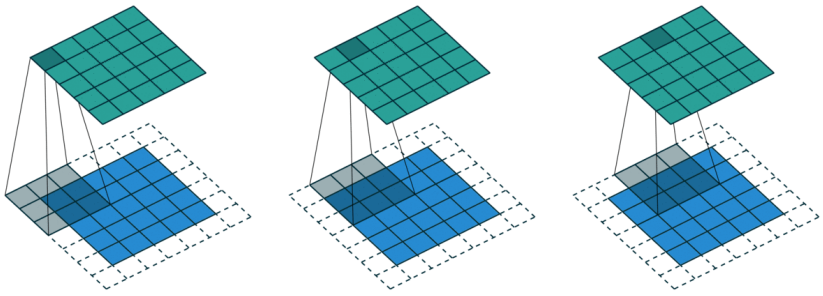


Figure 1: A 3x3 convolution with zero-padding.

By using convolutions, we also address the problem of translation. In a standard fully-connected layer, the network may learn how to detect a specific feature when it appears in the center of an image, but if that feature is in a different position in the image, it won't be detected. A kernel, when applied to an input image, will find every place that matches the feature it's looking for.

*Expansion of Sylesh Suresh and Mihir Patel's lectures of the same name

This property of CNNs is called *translation invariance*; later, we'll introduce a new type of layer that makes CNNs rotation and scale invariant as well.

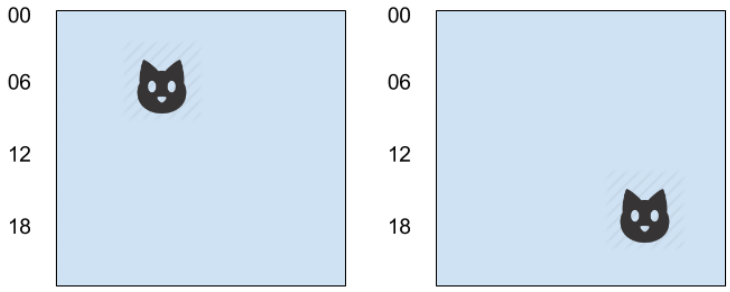



Figure 2: Translation invariance.

Convolutional layers can also handle larger input images. A vanilla neural network was fine for the MNIST dataset, with its 28x28x1 inputs, but moving to a modestly-sized color input image, say, 224x224x3, means we need a input layer of 150,000 neurons. Attempting to add a single hidden layer of half that size would require over 10 billion parameters (weights and biases).

Having the *kernels* in a CNN be what stores weights dramatically reduces the number of weights needed from layer to layer. Moreover, the fact that we store the weights in kernels means we pass the same weights over the whole image. With convolutional layers, the 152-layer deep ResNet architecture, which runs on 224x224x3 input images, has fewer than 20 million parameters *total*.

(To those unfamiliar with computer vision, storing weights in kernels seems like paring the connections between neurons, then sharing weights between them, which is why you'll see this described as "locally-connected neurons sharing weights." The convolution operation explains why and how we locally connect neurons and share weights, however.)

# 3 The Full CNN Structure

Now that we understand how convolutional layers are superior to fully-connected ones for image recognition, the problem becomes how we actually implement one. Recall from last week that a *feature map* is simply the output of a kernel convolved over an input image. It's so named because the output matrix will have high values wherever the input matches the feature the kernel is searching for. (See Figure 3.)

A convolutional layer is simply multiple kernels run over the same input. The number of feature maps produced, and thus the number of features detected, is determined by the number of kernels convolved over the input. Together, the output feature maps can be treated as an output *stack* of feature maps.

For inputs with multiple channels, like color images, we can stack kernels to make a *filter*. A single filter still produces a single output feature map. If we

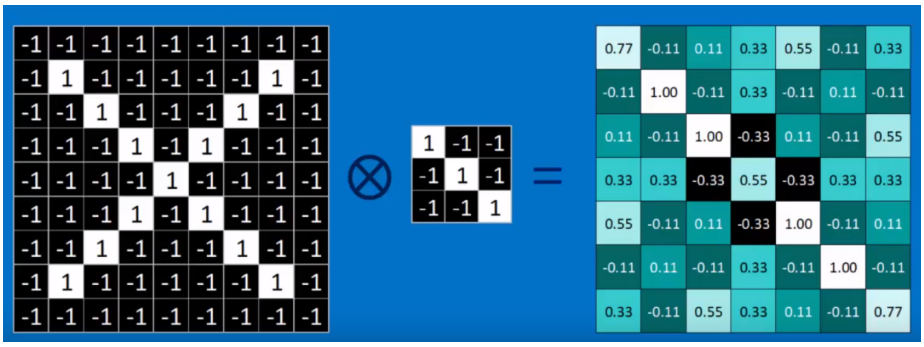



Figure 3: A feature map for the negative-slope diagonal.

append a second convolutional layer to the first, its input will be the output stack of feature maps from the first layer: the output stack of the first layer is treated as a multi-channel input for the second layer.

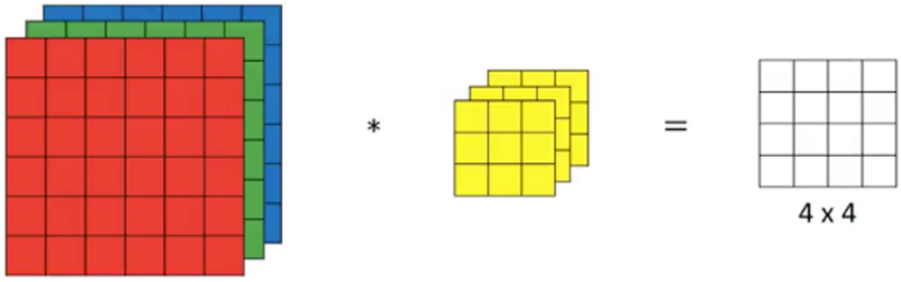



Figure 4: A convolution on a three-channel input.

By feeding successive convolutional layers the inputs of the prior layer, we structure our CNN to be able to abstractly combine features. For example, for a CNN designed to classify if an image contains an X or not, layer one might have three kernels that identify a diagonal with a positive slope, a diagonal with a negative slope, and the intersection of those two. This would produce a three-deep feature map stack, which the next layer would then look at to see if the feature maps from the prior layer activate in the right areas for an image with an X.

## 3.1 Activation Function

Of course, to get these convolutional layers to learn what features to detect, by backpropagating and updating the weights in each kernel, we need to introduce an activation function. We learned in the basic neural network lectures that a standard activation function is the sigmoid function, or:

$$(x) = \frac{1}{1 + e^{-x}}$$

Another standard activation function is the hyperbolic tangent function:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^{x} \quad e^{-x}}{e^{x} + e^{-x}}$$

Tanh has roughly the same shape as the sigmoid function, but has a larger range of -1 to 1 when compared to sigmoid's range of 0 to 1:

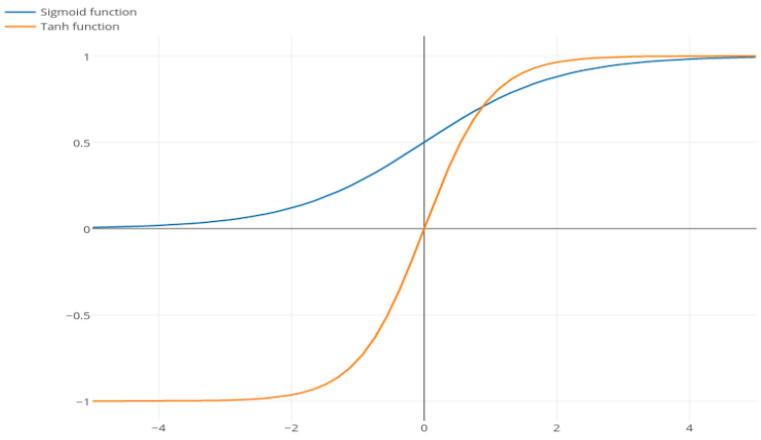



Figure 5: Sigmoid vs tanh function.

Both of these functions suffer from the vanishing gradient problem, where the gradients become negligibly small when the inputs are very large or very small. When developing CNNs, researchers instead chose to go with the Rectified Linear Unit, or ReLU, as their activation function.

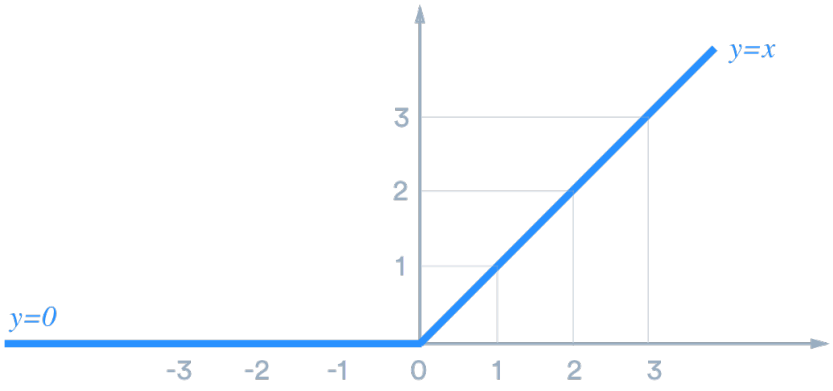



Figure 6: The ReLU function.

Mathematically, this is simply:

$$f(x) = max(0; x)$$

The slope of the ReLU function for any positive inputs, or any values a neuron activates on, is one. This prevents the gradient from vanishing, speeding up training. Also, the function itself and its derivative are incredibly cheap to compute, which also speeds up training.

The ReLU activation is applied directly to the output feature map(s) of a convolutional layer. In practice, this is as simple as replacing all the negatives with zeroes. For example, let's say this is the output feature map for a very strange feature:

$$\begin{bmatrix} 2 & 1 & 1 & 4 & 3 \\ 6 & 2 & 2 & 7 & 7 \\ 6 & 3 & 7 & 1 & 5 \\ 4 & 3 & 5 & 4 & 6 \end{bmatrix}$$

Applying ReLU gives us this matrix:

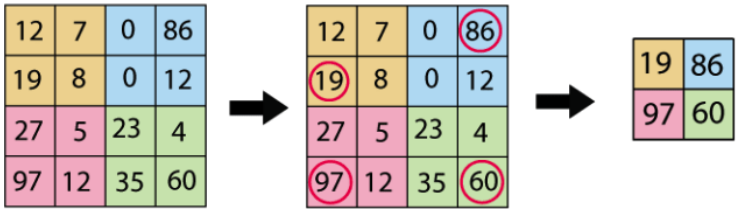
$$\begin{bmatrix} 2 & 1 & 0 & 0 & 3 \\ 6 & 2 & 0 & 7 & 7 \\ 6 & 0 & 0 & 0 & 5 \\ 4 & 3 & 5 & 4 & 0 \end{bmatrix}$$

Conceptually, this can be thought of as the network only taking positive cases into account when training, since the slope is zero for negative x values of ReLU. This mirrors how biological neurons fire only when a certain activation threshold is reached, and do nothing otherwise.

## 3.2 Pooling Layers

However, even with speedy ReLU activations and the inherent advantages of convolutional layers over fully-connected ones, CNNs are still computationally intensive. To further reduce computational cost, we can add a new type of layer to our CNN: pooling layers. As the name implies, these layers pool together values that are spatially close together.

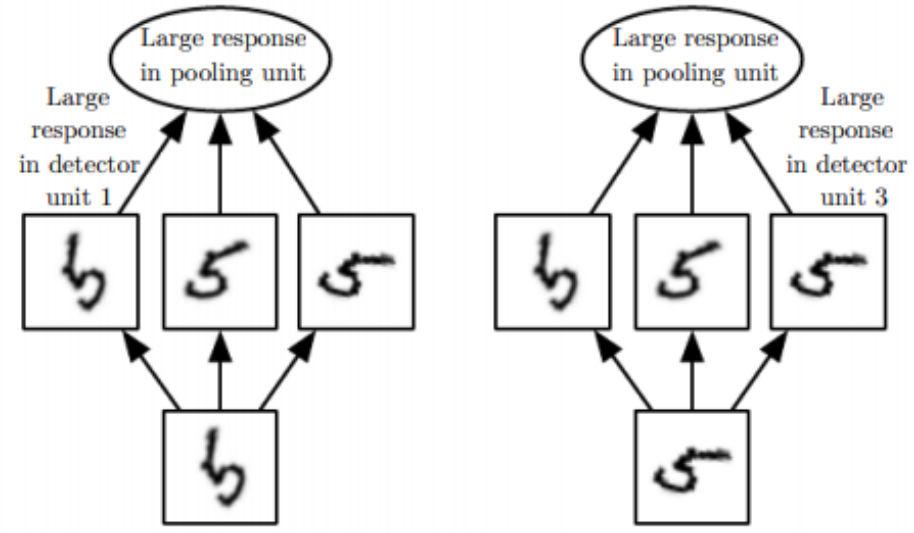
The most common type of pooling is max pooling, where the highest value within a certain region is selected and passed down, reducing the size of the matrix given. In this example, we consider 2x2 regions:



This has a few key benefits. The motivation behind pooling layers was to reduce the size of the input matrix for the next convolutional layer, which saves training time. Taking only the max value of a feature map has a de-noising effect as well, because weak activations are suppressed.

Another benefit is our CNN is more translation invariant, and now rotation and scale invariant as well. To demonstrate, imagine our CNN, by piecing together lower-level features from earlier layers, has a layer that identifies the number five in three different rotated positions:

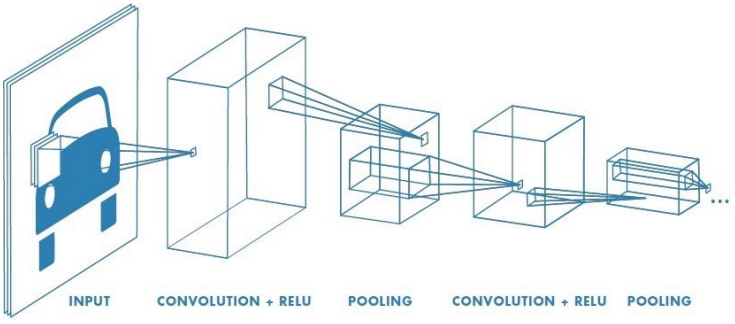
Of course, these are all fives, and we want the next layer of our CNN to activate if any of these three versions activate. By introducing a max pooling layer, our CNN activates if any of the three rotations matches, even if the other two don't:




The arguments for translation and scale invariance are similar. Generally, we put max pooling layers after every ReLU activation, to gain this invariance and computational speedup for every convolutional layer.

### 3.3 Fully-Connected Tail

We understand now that convolutional layers produce stacks of feature maps, which, after some additional processing with ReLU and max pooling layers, are fed as inputs to successive convolutional layers. Graphically, that looks like:




How do we turn a stack of feature maps into a prediction, though? CNNs are designed to classify images well, but the current output—a stack of feature maps—doesn't contain any classification of what's actually in the image.

The solution is to turn back to the fully-connected layer. After successive convolution and pooling layers, the CNN should have an high-level sense of which features exist in the image, regardless of position, scale, or rotation. It should also have reduced the dimensions of the input image to a manageable size. This means our main problems with using fully-connected layers for image recognition have been solved.

By flattening the final 2D output feature maps from a series of convolutional layers, turning them into a 1D array, image classification turns into a standard classification problem. With many arbitrary numerical inputs and a known desired output, we can plug in a fully-connected layer at the end to classify images. Our full CNN structure looks like this:

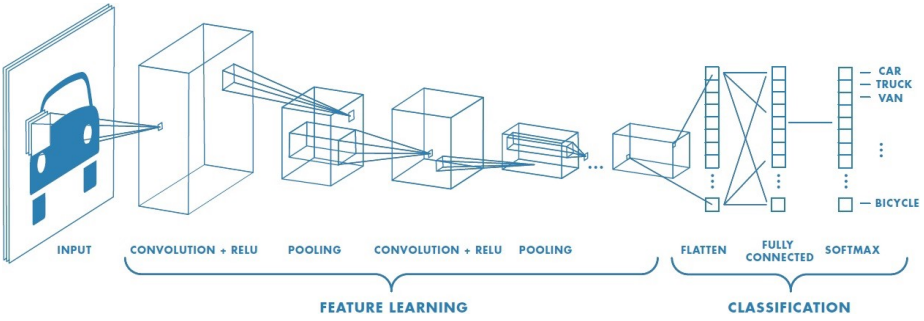



Figure 7: The full CNN structure.

The flattening process is exactly what it sounds like. For example, this matrix:

$$\begin{matrix} 2 & & & 3 \\ & 1 & 2 & 3 \\ 4 & 4 & 5 & 6 & 5 \\ & 7 & 8 & 9 \end{matrix}$$

flattens to:

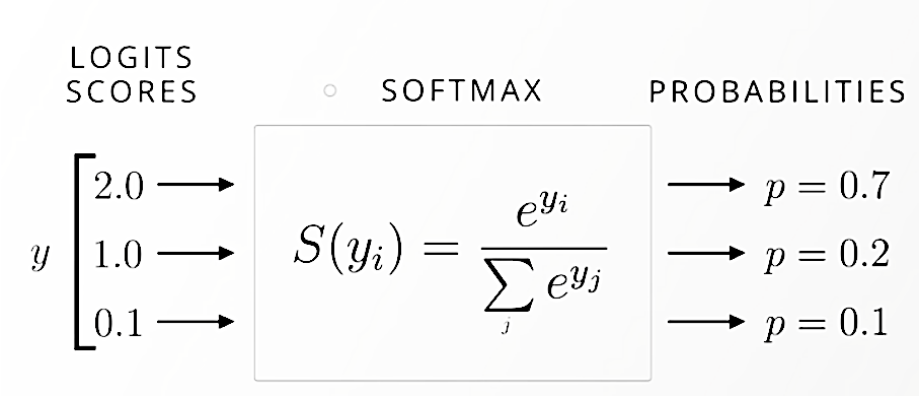
$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$$

(For a 3D stack of matrices, the process is identical, with the layers further back simply tacked onto the end of the flat 1D matrix.)

In theory, you could add multiple fully-connected layers at the end of any CNN. In practice, however, since we want the convolutional layers to do most of the image recognition, there is usually only one fully-connected layer added for classification.

### 3.4 Softmax

Finally, you'll see the last layer of the diagram in Figure 7 is labelled "softmax." Like ReLU, this is another new activation function. Luckily, it's easy to understand. Softmax simply converts the numerical outputs of a CNN into probabilities, with higher numbers translating into higher probabilities.




The use of softmax layers at the end of CNNs is to make them easier to train. This way, to turn a categorical label like "car" or "cat" into the desired training output of the CNN, we just have to create an array with as many entries as we have classes, and set the "probability" of the right class to 1. This is called one-hot encoding.

## 4 CNN Hyperparameters

Like a vanilla neural network, some basic hyperparameters to tune in a CNN are layer count, learning rate, and batch size. There are a few CNN-specific hyperparameters though, mostly for the convolutional layers.

## 4.1 Kernel Size

Changing the kernel size impacts the computational cost in forward and backward propagation and the scale of the features learned. Smaller kernels learn smaller patterns whereas larger kernels are more difficult to train but can extract more spacial information. Typically, odd numbers are used for kernel sizes so that at each step the center is a specific pixel and not the center of a pixel. Common sizes are 3x3, 5x5, and 1x1 (which adjusts the number of channels).

## 4.2 Depth

Again, each kernel convolved will produce an output feature map, so the depth of the output stack will be equivalent to the number of kernels in a layer. In papers, the notation for a convolutional layer is kernel size, then depth. [3x3, 64] is a layer with 64 3x3 kernels.

## 4.3 Stride

In the examples above and in last week's lecture, we slid a kernel over every possible pixel of the input. This means our kernel had a stride length of 1, as at every step we move the kernel by 1 in the right direction. However, it can be the case that two kernel locations may have high enough overlap that calculating both is repetitive. This is especially true for larger kernels. To account for this, we increase the stride length.

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 6 & 4 \\ 10 & 24 & 20 \\ 8 & 18 & 10 \end{bmatrix}$$

In the above equation, we apply a 2 x 2 kernel to a 4 x 4 image to obtain a 3 x 3 result. However, if we increase the stride length to 2:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 8 & 10 \end{bmatrix}$$

This reduces the dimensions of the output matrix. The max pooling layer described above can be thought of as having a stride of 2.

## 4.4 Zero Padding

A kernel cannot be centered around border pixels, which may lead to it missing certain features:

$$\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = 24$$

To fix this, we can add zeros to the border, a technique called zero-padding:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 6 & 4 \\ 10 & 24 & 20 \\ 8 & 18 & 10 \end{bmatrix}$$

This allows us to apply the kernel to the edges. The reason we commonly use kernels with odd dimensions is because the center of an even-dimensioned matrix is ambiguous, as in the above case, where a 2 x 2 kernel is used.

# 5 References

## 5.1 Sources

See list of references in the first CNN lecture ("Convolutional Neural Networks: The Convolution").

## 5.2 Images


https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7

https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function

https://stats.stackexchange.com/questions/207195/translational-variance-in-convolutional-neural-networks

https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

https://principlesofdeeplearning.com/index.php/2018/08/27/is-pooling-dead-in-convolutional-networks/

http://egrcc.github.io/docs/dl/deeplearningbook-convnets.pdf

https://medium.com/data-science-bootcamp/understand-the-softmax-function-in-minutes-f3a59641e86d


Some images from the first CNN lecture ("Convolutional Neural Networks: The Convolution").