Convolutional Neural Network (CNN)

Images used for Computer Vision problems nowadays are often 224x224 or larger. Imagine building a neural network to process 224x224 color images: including the 3 color channels (RGB) in the image, that comes out to 224 x 224 x 3 = 150,528 input features! A typical hidden layer in such a network might have 1024 nodes, so we'd have to train 150,528 x 1024 = 150+ million weights for the first layer alone. Our network would be huge and nearly impossible to train.

It's not like we need that many weights, either. The nice thing about images is that we know pixels are most useful in the context of their neighbors. Objects in images are made up of small, localized features, like the circular iris of an eye or the square corner of a piece of paper. Doesn't it seem wasteful for every node in the first hidden layer to look at every pixel?

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm that can take an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a CNN is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, CNNs have the ability to learn these filters/ characteristics.

Why CNN is Preferred over Feed-Forward Neural Nets?

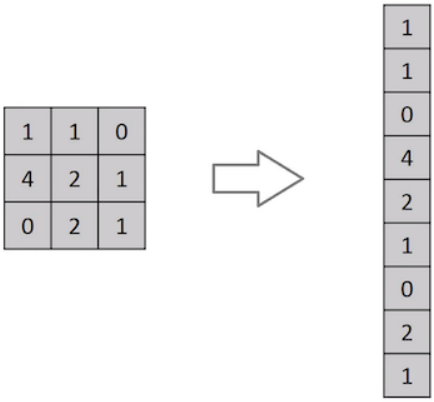
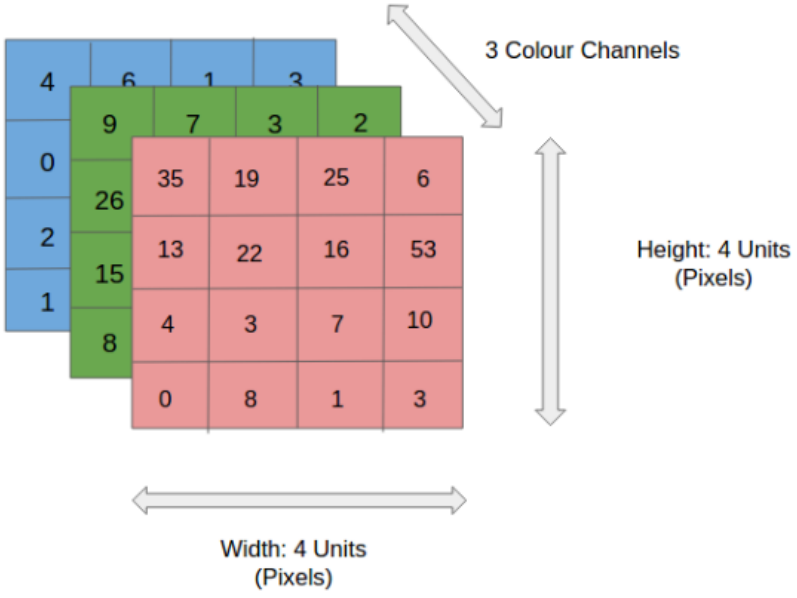


Fig: Flattening of a 3x3 image matrix into a 9x1 vector

An image is nothing but a matrix of pixel values, right? So why not just flatten the image (e.g. 3x3 image matrix into a 9x1 vector) and feed it to a Multi-Level Perceptron for classification purposes?

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

Whereas, a CNN is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. Such architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and the reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

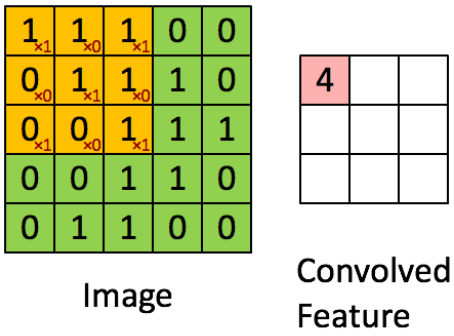


For example, in the above figure, we have an RGB image which has been separated by its three color planes — Red, Green, and Blue.

One can imagine how computationally intensive things would get once the images reach dimensions, say 8K (7680×4320). The role of the CNN is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets. *That is why; CNN is preferred over feed-forward Net.*

The Kernel – Objective of Convolution Operation:

In the below image, the green section resembles our 5x5x1 input image, I. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the Kernel/Filter, K, represented in the color yellow. We have selected K as a 3x3x1 matrix.




```
Kernel/Filter, K = 1  0  1
                   0  1  0
                   1  0  1
```

The Kernel shifts 9 times because of Stride Length = 1 (Non-Strided), every time performing a matrix multiplication operation between K and the portion P of the image over which the kernel is hovering.

The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.

The objective of the Convolution Operation is to extract the high-level features such as edges, from the input image. CNNs need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, color, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network which has the wholesome understanding of images in the dataset, similar to how we would.

Pooling Layer:

The Pooling layer can be seen between Convolution layers in a CNN architecture. This layer basically reduces the amount of parameters and computation in the network, controlling overfitting by progressively reducing the spatial size of the network.

Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to decrease the computational power required to process the data through dimensionality reduction. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model.

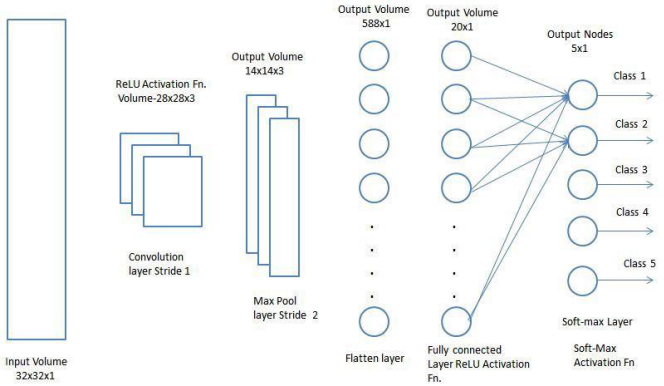
There are two types of Pooling: Max Pooling and Average Pooling. Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values from the portion of the image covered by the Kernel.

Max Pooling also performs as a Noise Suppressant. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction. On the other hand, Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that Max Pooling performs a lot better than Average Pooling.

Classification — Fully Connected Layer (FC Layer):

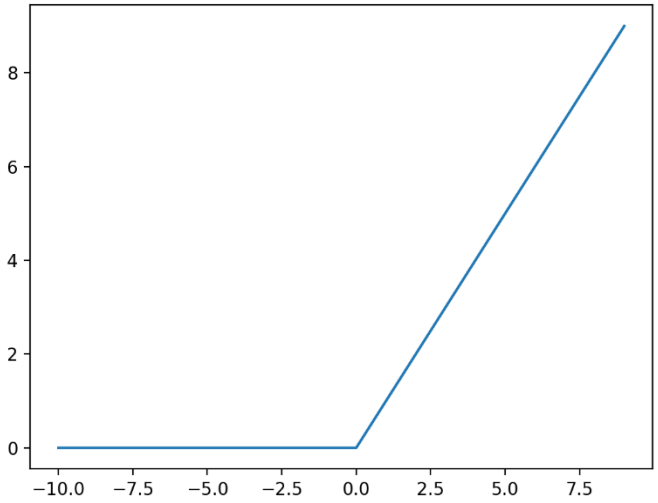
Adding a Fully-Connected layer is a (usually) cheap way of learning non-linear combinations of the high-level features as represented by the output of the convolutional layer. The Fully-Connected layer is learning a possibly non-linear function in that space.

Now that we have converted our input image into a suitable form for our Multi-Level Perceptron, we shall flatten the image into a column vector. The flattened output is fed to a feed-forward neural network and backpropagation applied to every iteration of training. Over a series of epochs, the model is able to distinguish between dominating and certain low-level features in images and classify them using the Softmax Classification technique.



Why ReLU is popular compared to the other activation function?

The rectified linear unit (ReLU) function is the most widely used activation function in neural networks today. One of the greatest advantages ReLU has over other activation functions is that it does not activate all neurons at the same time. From the image for ReLU function above, we'll notice that it converts all negative inputs to zero and the neuron does not get activated. This makes it very computational efficient as few neurons are activated per time. It does not saturate at the positive region. In practice, ReLU converges six times faster than *tanh* and sigmoid activation functions.



Some disadvantage ReLU presents is that it is saturated at the negative region, meaning that the gradient at that region is zero. With the gradient equal to zero, during back propagation all the weights will not be updated, to fix this, we use Leaky ReLU. Also, ReLU functions are not zero-centered. This means that for it to get to its optimal point, it will have to use a zig-zag path which may be longer.

Understanding and Calculating the Number of Parameters in Convolution Neural Networks (CNNs):

If you've been working with CNN's it is common to encounter a summary of parameters as seen in the below image. We all know it is easy to calculate the activation size, considering it's merely the product of width, height and the number of channels in that layer.

For example, as shown in the below image, the input layer's shape is (32, 32, 3), the activation size of that layer is 32 * 32 * 3 = 3072. The same holds good if you want to calculate the activation shape of any other layer. Say, we want to calculate the activation size for CONV2. All we have to do is just multiply (10,10,16) , i.e 10*10*16 = 1600, and you're a done calculating the activation size.

However, what sometimes may get tricky, is the approach to calculate the number of parameters in a given layer. With that said, here are some simple ideas to keep in mind to do the same.

Neural network example

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Andrew Ng

Some Context (How does a CNN Learn?):

This goes back to the idea of understanding what we are doing with a convolution neural net, which is basically trying to learn the values of filter(s) using back propagation. In other words, if a layer has weight matrices, that is a "learnable" layer.

Basically, the number of parameters in a given layer is the count of "learnable" elements for a filter aka parameters for the filter for that layer.

Suppose you have 5, 4*4 filters in a layer, how many parameters can you learn in that layer?
You have 4*4 parameters for each filter and you have 5 such filters. Thus, total parameters you could learn would be 4*4*5 = 80 parameters, for that entire layer.

Now that you know what "parameters" are, let's dive into calculating the number of parameters in the sample image we saw above.

1. Input layer: Input layer has nothing to learn, at it's core, what it does is just provide the input image's shape. So no learnable parameters here. Thus number of parameters = 0.
2. CONV layer: This is where CNN learns, so certainly we'll have weight matrices. To calculate the learnable parameters here, all we have to do is just multiply the by the shape of width m, height n and account for all such filters k. Don't forget the bias term for each of the filter. Number of parameters in a CONV layer would be : ((m * n)+1)*k), added 1 because of the bias term for each filter. The same expression can be written as follows: ((shape of width of the filter*shape of height of the filter+1)*number of filters).

3. POOL layer: This has got no learnable parameters because all it does is calculate a specific number, no backprop learning involved! Thus number of parameters = 0.
4. Fully Connected Layer (FC): This certainly has learnable parameters, matter of fact, in comparison to the other layers, this category of layers has the highest number of parameters, why? because, every neuron is connected to every other neuron! So, how to calculate the number of parameters here? You probably know, take the product of the number of neurons in the current layer and the number of neurons on the previous layer. Thus number of parameters here : ((current layer n*previous layer n)+1). As always, do not forget the bias term!

Explanation:

1. The first input layer has no parameters. You know why.
2. The second CONV1(filter shape =5*5, stride=1) layer has how many parameters? Let's do the math according to the formula: it is ((shape of width of filter*shape of height filter+1)*number of filters) = (((5*5)+1)*8) = 208.
3. The third POOL1 layer has no parameters. You know why.
4. The fourth CONV2(filter shape =5*5, stride=1) layer has how many parameters? Let's do the math according to the formula: it is ((shape of width of filter*shape of height filter+1)*number of filters) = (((5*5)+1)*16) = 416.
5. The fifth POOL2 layer has no parameters. You know why.
6. The Sixth FC3 layer has ((current layer n*previous layer n)+1) parameters = 120*400+1 = 48,001.
7. The Seventh FC4 layer has ((current layer n*previous layer n)+1) parameters = 84*120+1 = 10,081.
8. The Eighth Softmax layer has ((current layer n*previous layer n)+1) parameters = 10*84+1 = 841.

FYI:

1. I've used the term "layer" very loosely to explain the separation. Ideally, CONV + Pooling is termed as a layer.

2. Just because there are no parameters in the pooling layer, it does not imply that pooling has no role in backprop. Pooling layer is responsible for passing on the values to the next and previous layers during forward and backward propagation respectively.

A Simple Explanation of the Softmax Function:

Softmax turns arbitrary real values into probabilities, which are often useful in Machine Learning. The math behind it is pretty simple: given some numbers,

1. Raise e (the mathematical constant) to the power of each of those numbers.
2. Sum up all the exponentials (powers of e). This result is the *denominator*.
3. Use each number's exponential as its *numerator*.
4. $\text{Probability} = \frac{\text{Numerator}}{\text{Denominator}}$.

Written more fancily, Softmax performs the following transform on n numbers $x_1 \ldots x_n$:

$$s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The outputs of the Softmax transform are always in the range $[0, 1]$ and add up to 1. Hence, they form a **probability distribution**.

A Simple Example

Say we have the numbers -1, 0, 3, and 5. First, we calculate the denominator:

$$\begin{aligned} \text{Denominator} &= e^{-1} + e^{0} + e^{3} + e^{5} \\ &= \boxed{169.87} \end{aligned}$$

Then, we can calculate the numerators and probabilities:

x	Numerator (e^x)	Probability ($\frac{e^x}{169.87}$)
-1	0.368	0.002
0	1	0.006
3	20.09	0.118
5	148.41	0.874

The bigger the x, the higher its probability. Also, notice that the probabilities all add up to 1, as mentioned before.

Why is Softmax useful?

Imagine building a Neural Network to answer the question: *Is this picture of a dog or a cat?*

A common design for this neural network would have it output 2 real numbers, one representing *dog* and the other *cat*, and apply Softmax on these values. For example, let's say the network outputs $[-1, 2]$:

Animal	x	e^x	Probability
Dog	-1	0.368	0.047
Cat	2	7.39	0.953

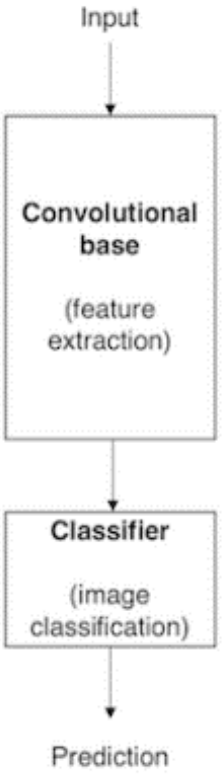
This means our network is **95.3% confident** that the picture is of a cat. Softmax lets us **answer classification questions with probabilities**, which are more useful than simpler answers (e.g. binary yes/no).

Dropout Layers:

Dropout layers have a very specific function in neural networks. Sometimes 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.

Data Augmentation Techniques:

By now, we're all probably numb to the importance of data in CNNs, 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.



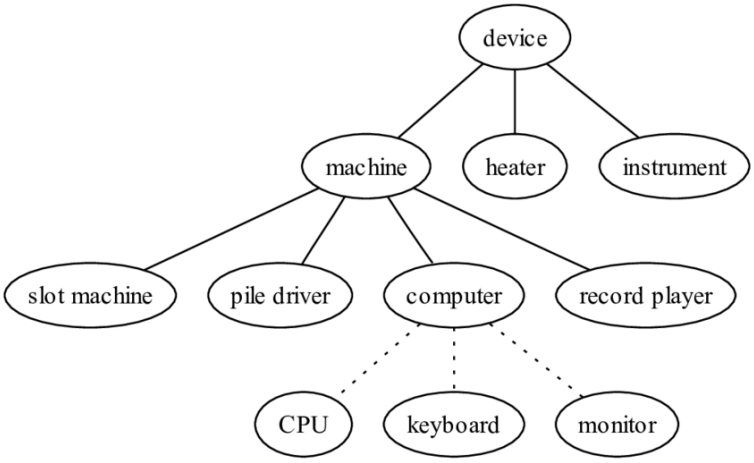
ImageNet:

ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently we have an **average of over five hundred images per node**. We hope ImageNet will become a useful resource for researchers, educators, students and all of you who share our passion for pictures.

Models for image classification with weights trained on **ImageNet**:

- Xception
- VGG16
- VGG19
- ResNet, ResNetV2, ResNeXt
- InceptionV3
- InceptionResNetV2
- MobileNet
- MobileNetV2
- DenseNet
- NASNet

Large Scale Visual Recognition Challenge 2017



The ImageNet project is a large visual database designed for use in visual object recognition software research. More than 14 million[1][2] images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided.[3] ImageNet contains more than 20,000 categories[2] with a typical category, such as "balloon" or "strawberry", consisting of several hundred images.[4] The database of annotations of third-party image URLs is freely available directly from ImageNet, though the actual images are not owned by ImageNet.[5] Since 2010, the ImageNet project runs an annual software contest, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where software programs compete to correctly classify and detect objects and scenes. The challenge uses a "trimmed" list of one thousand non-overlapping classes.

Miscellaneous:

Common to see convolution layers with stride of 1, filters of size K, and zero padding with $\frac{K-1}{2}$ to preserve size

Note that ,you can have different strides horizontally and vertically. You can use the following equations to calculate the exact size of the convolution output for an input with the size of (width = W, height = H) and a Filter with the size of (width = F_w, height = F_h):

$$\textbf{output width} = \frac{W - F_w + 2P}{S_w} + 1$$

$$\textbf{output height} = \frac{H - F_h + 2P}{S_h} + 1$$