An Introduction to Convolutional Neural Networks

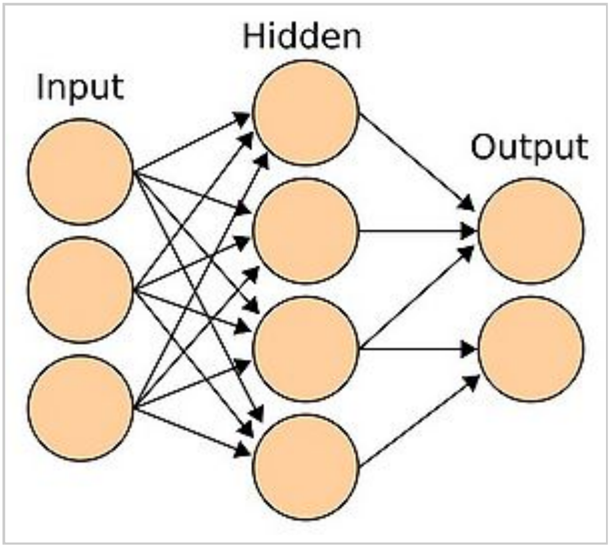
From Teach

Contents

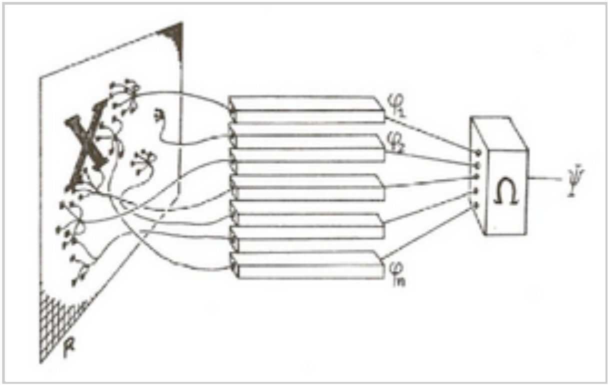
Background

The Early Neural Network Models

Computational models of neural networks have been around for more than half a century, beginning with the simple model that McCulloch and Pitts developed in 1943 [1]. Hebb subsequently contributed a learning algorithm to train such models [2], summed up by the familiar refrain: 'Neuron that fire together, wire together'. Hebb's rule, and a popular variant known as the Delta rule, were crucial for early models of cognition, but they quickly ran into trouble with respect to their computational power. In their extremely influencial book, *Perceptrons*, Minsky and Papert proved that these networks couldn't even learn the boolean XOR function, due to due only being able to learn a single layer of weights[3].Luckily, a more complex learning algorithm, backpropagation, eventually emerged, which could learn across arbitrarily many layers, and was later proven to be capable of approximating any computable function.



A typical feed-forward neural network architecture used in backpropagation.



A single layer neural network from Minsky and Papert's book.

How a Feed-Forward Neural Network Works

The backpropagation algorithm is defined over a multilayer feed-forward neural network, or FFNN. A FFNN can be thought of in terms of neural activation and the strength of the connections between each pair of neurons. Because we are only concerned with feed forward networks, the pools of neurons are connected together in some directed, acyclic way so that the networks activation has a clear starting and stopping place (i.e. an input pool and an output pool). The pools in between these two extremes are known as hidden pools.

The flow of activation in these network is specified through a weighted summation process. Each neuron sends its current activation any unit is connected to, which is then multiplied by the weight of the connection to the receiving neuron and passed through some squashing function, typical a sigmoid, to introduce nonlinearities (if this were a purely linear process, then additional layers wouldn't matter, since adding two linear combinations together produces another linear combination). Since we typically assume each layer to be completed connected to the next layer, these calculations can be done via multiplying the vector of activations by the weight matrix and then passing all of the results through the squashing function.

Learning in these networks occurs through changing the weights so as to minimize some error function, typically specified as the difference between the output pool's activation vector and the desired activation vector. Normally this is accomplished incrementally via the previously mentioned backpropagation algorithm, in which the partial derivative of the error with respect to last layer of weights is calculated (and generally scaled down) and used to update the weights. Then the partial derivatives can be calculated for the second-to-last weight layers and so on, with the process repeating recursively until the weight layer connected to the input pool is updated. For more information about how these derivatives are calculated, visit the PDPHandbook (http://www.stanford.edu/group/pdplab/pdphandbook/) .

Problems with Backpropagation

Despite being a universal function approximator in theory, FFNNs weren't good at dealing with many sorts of problems in practice. The example relevant to the current discussion is the FFNN's poor ability to recognize objects presented visually. Since every unit in a pool was connected to every unit in the next pool, the number of weights grew very rapidly with the dimensionality of the input, which led to slow learning for the typically high dimensional domain of vision. Even more disconcerting was the spatial ignorance of the FFNN. Since every pair of neurons between two pools had their own weight, learning to recognize a object in one location wouldn't transfer to the same object presented in a different part of the visual field; separate weights would be involved in that calculation. What was needed was an architecture that exploited the two dimensional spacial constraints imposed by its input modality whilst reducing the amount of parameters involved in training. Convolutional neural networks are the architecture.

Convolutional Neural Networks

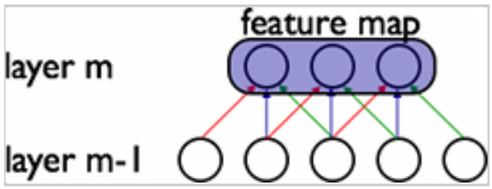
The solution to FFNNs' problems with image processing took inspiration from neurobiology, Yann LeCun and Toshua Bengio tried to capture the organization of neurons in the visual cortex of the cat, which at that time was known to consist of maps of local receptive fields that decreased in granularity as the cortex moved anteriorly. There are several different theory about how to precisely define such a model, but all of the various implementations can be loosely described as involving the following process:

1. Convolve several small filters on the input image
2. Subsample this space of filter activations
3. Repeat steps 1 and 2 until your left with sufficiently high level features.
4. Use a standard a standard FFNN to solve a particular task, using the results features as input.

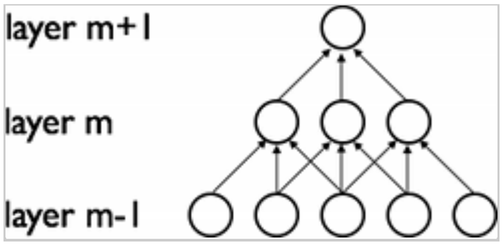
The LeCun Formulation

There are several different ways one might formalize the high level process described above, but the most common is LeCun's implementation, the LeNet.

Convolution



Color coding which weights are shared between neuronal connections.



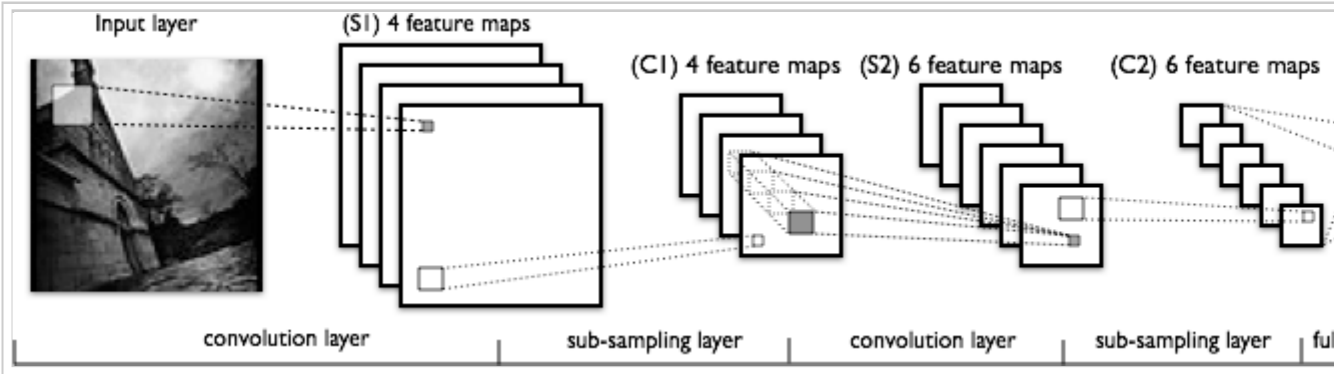
The sparse connectivity responsible for the receptive field behavior.

Convolution is a mathematical term, defined as applying a function repeatedly across the output of another function. In this context it means to apply a 'filter' over an image at all possible offsets. A filter consists of a layer of connection weights, with the input being the size of a small 2D image patch, and the output being a single unit. Since this filter is applying repeatedly, the resulting connectivity looks like a series of overlapping receptive fields, as shown in the 'sparse connectivity' image, which map to a matrix of the filter outputs (or several such matrices in the common case of using a bank of several filters). An important subtlety here is that why there are still a good deal of connections between the input layer and the filter output layer, the weights are tied together (as shown in the colored diagram). This means that during backpropagation, you only have to adjust a number of parameters equal to a *single* instance of the filter -- a drastic reduction from the typical FFNN architecture. Another nuance is that we could sensibly apply such filter to any input that's spatially organized, not just a picture. This means that we could add another bank of filters directly on top of our first filter bank's output. However, since the dimensionality of applying a filter is equal to the input dimensionality, we wouldn't be gaining any translation invariance with these additional filters, we'd be stuck doing pixel-wise analysis on increasingly abstract features. In order to solve this problem, we must introduce a new sort of layer: a subsampling layer.

Subsampling

Subsampling, or down-sampling, refers to reducing the overall size of a signal. In many cases, such as audio compression for music files, subsampling is done simply for the size reduction. But in the domain of 2D filter outputs, subsampling can also be thought of as increasing the position invariance of the filters. The specific subsampling method used in LeNets is known as 'max pooling'. This involves splitting up the matrix of filter outputs into small non-overlapping grids (the larger the grid, the greater the signal reduction), and taking the maximum value in each grid as the value in the reduced matrix. Semantically, this corresponds to changing the question answered by the convolution layer from "how well does this filter apply right here" to "how well does this filter apply to this area". Now, by applying such a max pooling layer in between convolutional layers, we can increase spatial abstractness as we increase feature abstractness.

Putting it all together



The complete implementation of the LeNet.

Even with the convolution and subsampling layers specified, there are still many free hyper-parameters. Namely, how many filters per convolutional layer? How big should the filters and subsamples be? and how many overall layers should there be? None of these questions be answered definitively, as the effectiveness of each hyper-parameter setting depends on the task setting, but LeCun attempted to provide some plausible values for roughing simulating human performance on natural image classification tasks. As the above figure shows, there are five functional layers in the total architecture, corresponding to 2 sets of convolution-max-pooling pairs and a FFNN for solving the actually classification problem. While training takes quite some time, this network learns much faster than a standard FFNN and performances quite well as a pure piece of computer vision software, which is rather incredible considering it wasn't developed solely for machine learning purposes.

Open Questions

Differing Implementations

While the LeNet is impressive, others have proposed slightly different architectures that can account for different psychological data and outperform it in a computer vision context [4]. However, due to the vast number of ways one might tweak such an architecture, its unclear how these various model might be evaluated with respect to their neural validity. More explicit predictions about human behavior and neural anatomy must be made before this has a hope of being resolved.

The Issue of Feedback

All of the current convolutional networks share the common problem of being strictly feed-forward. While its unclear if this limits their performance as machine learning tools, there is clear evidence from neuroimaging that feed-back connections do exist in the relevant cortical structures.

Feature Depth & Universality

While current model can be trained up to around 8 functional layers, many more might be needed to scale up to the full complexity of the human ventral stream. Unfortunately, the prospects for such scaling seem bleak, since backpropagation is known to be subject to "disappearing gradiants", error signals that rapidly approach zero as there passed down the layers. Luckily, there are several ways in which researchers are fighting back against this problem. The first comes from neuroscience, where some (e.g. Serre), have mapped their architectures directly to neural structure, and the mapping suggests that maybe to only really have 8 or so layers. Another approach is to evoke evolution. Perhaps some convolution layers were learnt on an evolutionary timescale and are task invariant. Such universal features aren't all that implausible for the early layers, as things such as Gabor filters tend to be learnt across most task settings. The last, and most promising, approach is to look to the emerging "deep learning" community for insight. So called deep architectures can have arbitrarily many layers since they are trained in a greedy, layer-by-layer procedure. While the best greedy learning algorithm for convolutional architectures is currently unclear (most deep learning involves unsupervised error signals), inroads are being made and the future for convolutional networks remains bright.

References

1. McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics 5 (4): 115–133.
2. Hebb, Donald (1949). The Organization of Behavior. New York: Wiley.
3. Minsky, M. and Papert, S. (1969). Perceptrons: An Introduction to Computational Geometry. MIT Press, Cambridge, MA.
4. Serre, T., Wolf, L., Bileschi, S., and Riesenhuber, M. (2007). Robust object recog- nition with cortex-like mechanisms. IEEE Trans. Pattern Anal. Mach. Intell., 29(3), 411–426. Member-Poggio, Tomaso.