#h1 Chapter 28 - Convolutional Deep Belief Neural Networks

#h2 Why images are special

#h3 28.1 - Introduction

#pg Convolutional deep belief neural networks (cDBNNs) represent a special class of DBNNs, almost exclusively used in machine vision. As discussed in previous chapters, vision remains the most complex of all sensory systems and the hardest to emulate with machine learning. In addition, images (or video) lend themselves to various visual tasks that can be performed using them as inputs. Categorizing an object in an image was considered an extremely hard, if not impossible problem for AI to achieve accuracies close to human ability. In the last decade, this has changed rapidly, due to advances in DBNNs and cDBBNs. Categorization is no longer the major challenge. cDBNNs of sufficient size (in terms of architecture, number of nodes, etc.) can learn to recognize just about any category of visual objects, provided that adequate training data exists. In addition, transfer learning (the re-use of a neural network that has already been trained on other visual objects, more on that later) reduced the time needed to train such networks, even on conventional hardware. Now, models are trained to perform more complex visual tasks. Moving on beyond categorization, models now also are successful in locating the object in the image, segmenting objects from other objects in the same image, computing the object's orientation in world space, determining the spatial relationships between objects (the pizza is on the plate, which is sitting on the table), and turning entire visual scenes into textual descriptions. They also inspired a whole new class of generative models that can create images from textual descriptions, create photorealistic images from a few user sketches and doodles, and apply artistic styles to regular images (e.g., turning a selfie into a work of art in the style of Van Gogh, Rousseau or even Picasso). And with perhaps some exceptions, most of the networks use a series of so-called convolutional layers. So, what is convolution? And why does convolution significantly improve the quality, accuracy, speed of training, and generalizability of cDBNNs? For that, we need to unpack convolution as a mathematical concept, and see how to apply it. First to one-dimensional timeseries, and then to two (black and white image), three (color image with three channels) or four dimensional (sequence of consecutive images in a video). We will again draw the analogy with biological neural networks, as we know from research that the visual systems of many animals have evolved to be wired as if they were indeed performing convolution. More about that later, let us first unpack convolution as a mathematical operation.

(26.1)

(26.2)

(26.3)

Equations 28.1-3. Equations 28.1 and 28.2 are alternative notations for the same convolution of f(x) and g(x). Equation 28.3 is the notation used for cross-correlation, a mathematical operation very similar to convolution when we apply them to discrete data, rather than continuously specified functions.

#h3 28.2 - Introduction to Convolution

#pg Convolution is a mathematical operation on two function - usually denoted by f(x) and g(x) - that produces a third function expressing how the shape of one is modified by the other. More specifically, the resulting function is created by progressively shifting a reversed (-f) function f (we will get to this odd reversal in a bit) relative to function g, computing the product of the two and taking the integral, creating the resulting function h(x). Notations differ, but typically you will find convolution expressed using one of two symbols (equations 28.1 and 28.2). If we wanted to derive the convolution between two continuous functions, expressed as equations, we would have to do some rather complicated math, which involves integrating the functions symbolically and deriving the function h(x) from the convolution of functions f(x) and g(x).

#im ../../../assets/figures/028/028-01.png 50 256 Figure 28.1 - A convolution of timeseries f(x) and a kernel/IRF g(x) can be seen as a filtering applied to an incoming signal to yield output signal h(x). In this example, a square form waveform is smoothed into an almost sinusoidal output by applying a filter modeled as a gaussian/normal distribution.

#br

#pg However, rather than going down this rather abstract (and out of scope) rabbit hole, we will work through a few examples that use discrete data, actual timeseries to which the rules of arithmetic can be applied. It is important to note that in many real-world applications of convolution, f(x) is a time series of some undetermined length, if we want it to be. In contrast, function g(x) typically takes on a characteristic shape not based on any measurement, but instead reflecting some operation a system applies to a timeseries passing through it, see Fig. 27-1. They are sometimes referred to impulse response functions (IRF), kernels, or filters. This convolution between a signal and kernel has many practical applications. For example, many effects applied to modern music can be seen as applying specific filters to the audio waveform. Delay, echo, reverb, low/high/bandpass filtering. Even bass, treble, and high equalizer bands can be seen as convolutions of a filter g on a timeseries f. Similarly, in photography, we can imagine things like blurring, sharpening and edge detection as well as applying a small 2D filter or kernel g to a much larger image f. In probability theory, the probability distribution of the sum of two independent random variables is the convolution of their distributions. As a final example, the polarization of a neuron in response to a brief impulse of inputs will decay back to baseline in way that can be computed by convolving the impulse (as a timeseries) and the impulse response function of a typical neuron. If the inputs combine into a strong enough polarization, the neuron sets of a chain of reactions, all governed by their own impulse response functions that has the neuron produce an action potential: an impulse send to neurons connecting to it.

#h3 28.3 - Convolution versus Cross-Correlation

#pg For discrete timeseries computing the convolution between f and g is very similar to computing a cross correlation, with the only difference being that in convolution, the filter or kernel g is reversed. In other words, the cross-correlation of functions f(x) and g(x) is the convolution of f(x) and g(-x). Despite this and the fact that both operations look different when written out as equations and algorithms, they produce the same result under certain circumstances. Why we need to reverse the kernel g will become clearer below. To makes things even more confusing, the symbol for a cross-correlation operation is the star symbol (Eq. 28.3). In contrast to convolution, cross-correlations are more commonly computed for time series that are both of considerable and similar length. In such cases, neither f nor g holds a special status as the kernel, filter, or impulse function. To compute the cross-correlation between f and g, we take the two time series, and their different correlation shifts one time-series relative to the other. In other words, we compute the correlation between f and g, shift the entire time course g one data point to the right, recompute the correlation, and so on until we have exhausted all possible shifts of g versus f. The resulting series of correlations is cross-correlation. Consider the cross-correlation between the rise and fall of stock prices and the rise and fall of employment. We expect them to be correlated, but not instantaneously in time. A stock market crash will increase unemployment levels at some later date, and an economic boom will decrease unemployment, presumable after a similar delay.

#h3 28.4 - Convolution explained - 1D time series - example 1

#pg For the sake of consistency and clarity, we will refer to g(x) as the filter rather than kernel or impulse response function. All three names are technically correct but let us stick to one. When we have a discrete time series or signal f as well as a discrete filter g, computing the convolution is relatively straightforward.

#im ../../../assets/figures/028/028-02.png 50 256 Figure 28.2 - Top panel: a rectangular/step function time series f(x) with a brief 'on' or 1 state, lasting 200ms, with the remaining 800ms being in the 'off' or 0 state. The gray areas on either side represent the padding of the time series. Bottom panel, normal/gaussian distribution is used as the filter g(x) to be applied to f(x) using convolution.

#br

#pg Imagine that f(x) is a so-called rectangular or step function (Fig. 28.2), zero for most of the time series (we can call that the 'completely off' state'), only to turn into a sequence of ones halfway in for some brief duration (the ''completely on' state) before returning to the off state. This function can represent an alarm clock going off, the sound of a notification on your phone, or a vibration on your wearable. The problem is it is quite abrupt. The suddenness of such a sound or sensation might not be something the user is keen on. So, we want to gradually increase the intensity from completely off to completely on and gradually decrease it back to completely off. So, our filter g(x) must have two features. First, it needs to rise gradually but asymptote at some point, like the sigmoid function we have seen in earlier chapters. Second, when it reaches its asymptotic plateau, it must show the same gradual decay back when the input f(x) returns to 0 (completely 'off'). If this sounds familiar, you are on the right track. As it turns out, the normal distribution has these exact properties. It is symmetrical, gradually rising as it approaches the mean, asymptoting at the mean, before gradually decaying back to zero. The standard deviation can control the rate of growth and decay, with greater standard deviations making the normal distribution wider, thus making the rise and decay more gradual. One important thing to point out: as stated before, in this example, and for many use cases, the time series f(x) is substantially longer than the filter g(x) we which to convolve it with. This makes the implementation of convolution (and cross-correlation) a bit cumbersome. However, it is easily fixed by padding the filter g(x) with zeros on either side so that it matches, in length, the time series f(x). Convolution between the step function f(x) and the filter g(x) can proceed as follows. First, we shift g(x) leftwards until it is completely outside the original extent of f(x) but is in the padding we added as described above. From this starting point, we shift the filter g(x) one data point to the right and compute the element-wise product of f(x) and g(x). If the kernel does not overlap with any non-zero elements in f(x), the result will be zeros throughout. Instead, if g(x) and f(x) do overlap partially or completely, the result of f(x)g(x) will have non-zero values. Summing these together computes the discrete version of taking the integral of a function: computing the area under the products of functions f(x) and g(x). We repeat this process of moving filter g(x) one data point at a time until the kernel g(x) is completely outside f(x) and into the padded region to the right of it. A visual representation of this can be seen in Fig. 28.3.

#im ../../../assets/figures/028/028-03.gif 50 256 Figure 28.3 - Animation of the convolution between time series f and filter g. Top row: time series f. Second row: filter g, iteratively being shifted rightward one data point at a time. Third row: element-wise product of f and g at each shift of g. Bottom row: resulting convolution h.

#br

#pg Of course, we could have defined this smooth function completely beforehand, but one of the nice features of convolution is that it can be done in real-time as the time series comes in. Think of a trending algorithm that monitors specific keywords appearing in tweets on Twitter and displays this on some dashboard in real-time. Without any smoothing over some period (say a few minutes), the presence or absence of a particular keyword behaves erratically. Even if a keyword is trending, it will still only appear in a handful of tweets out of the millions being posted every day (at the time of writing, the average number of tweets posted per second is around 6000). Using convolution, we can smooth and average the presence of keywords over longer time periods, giving us a more interpretable 'trending' time series.

#im ../../../assets/figures/028/028-04.png 50 256 Figure 28.4 - The complex coupling between stimuli (visual images shown to a human subject), the resulting neural activity, and the hemodynamic response it elicits. In a nutshell, when a part of your brain is active, like your visual cortex responding to watching images displayed on a screen, it needs higher amounts of oxygen compared to when it is less or no activity. This shifts the balance between oxygen-rich and de-oxygenated blood. It is this shifting balance that is detected using functional Magnetic Resonance Imaging or fMRI.

#h3 28.5 - Convolution explained - 1D time series - example 2

#pg The previous section used a symmetrical function g. As we already pointed out, the convolution of time series f with a filter g requires us to reverse the filter g. Using an asymmetrical function in a second example will clearly demonstrate why. The asymmetrical function we will use is the so-called hemodynamic response function.

#im ../../../assets/figures/028/028-05.png 50 256 Figure 28.5 - Hemodynamic response function. A brief (<1s) burst of brain activity following the presentation of a stimulus (e.g., showing a subject an image) is followed by a rather sluggish and lengthy change in the balance of oxygenated and de-oxygenated blood, lasting many seconds.

#br

#pg In functional Magnetic Resonance Imaging or fMRI, local neural activity is measured as the ratio between oxygenated and de-oxygenated blood. The underlying mechanism is not completely understood, but the consensus is that neural activity is a metabolic process like all other bodily functions and therefore requires oxygen as energy. Local neural activity increased the demand for oxygen and the ratio between oxygenated and de-oxygenated blood shifts. However, this is not instantaneous. When a brain area is activated, for example, the visual cortex, by showing subjects in the scanner images instead of absolute darkness, it will take some time for the ratio in oxygen to shift. This sluggish response is captured by the HRF. For a brief presentation of an image (say <1 second), the rise and fall of the ratio can take up to 25 seconds to return to its previous baseline. In addition, there is a characteristic undershoot, where for some time, the ratio reverses before returning to baseline. Like the first example, we will compute the convolution between a rectangular function f and the hemodynamic response function HRF g. First, we add padding to f on both sides, with the padding being equal to the length of the HRF. We then create g by first reversing the HRF so that the last element is now first, the second to last element is now second, and so on. Second, we pad g with zeros to make it equal in length to f by padding the right-hand side with zeros.

#im ../../../assets/figures/028/028-06.png 50 256 Figure 28.6 - Top panel: a rectangular/step function time series f with two short 'on' states of different amplitudes (0.5 and 1), with the remaining elements of f being in the 'off' or 0 state. Bottom panel, the HRF function reversed and positioned in its entirety outside of f but instead inside the padding we introduced on both sides of f.

#br

#pg Now it should become apparent why we needed to reverse filter g prior to the convolution operation. As you can see in the animation in Fig. 28.7, by sliding the function g iteratively from left to right, the first non-zero point of time series f is met by the most rightward part of the HRF. Since we are explicitly computing the convolution of a timeseries f with filter g, the reversal of g ensures that the time ordering is correct: the response h rise quickly when the HRF in g starts to overlap with the non-zero elements in f, and slowly decays and undershoots before returning to a baseline response.

#im ../../../assets/figures/028/028-07.gif 50 256 Figure 28.7 - Animation of the convolution between time series f and filter g. Top row: timeseries f. Second row: filter g, a reversed hemodynamic response function, iteratively being shifted rightward one data point at a time. Third row: element-wise product of f and g at each shift of g. Bottom row: resulting convolution h.

#h3 28.6 - 2D Convolution on images - Sobel Filter

#pg Convolution can be applied to any N-dimensional array of data, and images defined in 2D (or 3D if we include color channels, like RGB or CYMK) are no different. During the first wave of computer vision, a lot of effort was put into designed filters that allowed for certain features to be detected within images.

(26.4)

(26.5)

(26.6)

(26.7)

Equations 28.4-7. Sobel Operators to compute the vertical (Gx) and horizontal (Gy) local gradients in an image. Combine the two allows us to estimate both the magnitude (Eq. 28.6) and orientation Eq. 28.7 of the local gradient.

#br

#pg The best known of these prefab filters are probably the edge detectors, such as the Sobel operator. For good reason, edge detecting was seen as a fundamental first step in detecting, locating, and classifying visual objects in images and video. The Sobel operator uses two 3x3 filters, which in machine vision are more commonly called kernels that are both applied to an input image. Mathematically, one filter computes the gradient (a change in pixel intensity) in the horizontal direction, the other computes the gradient in the vertical direction. Individually, the output of the convolution between image and each filter yields an output where non-zero elements indicate the presence of horizontal edges and vertical edges respectively. However, taking a weighted average allows us to compute the local gradient in any direction, by taking a pixel-by-pixel ratio between the output of horizontal and vertical filter at each location.

#im ../../../assets/figures/028/028-08.png 50 256 Figure 28.8 - Using the Sobel operator, we apply two filters Gx and Gy to an input image, see Equations 26.4 and 26.5. These filters compute the input image's local horizontal and vertical gradients, which are visualized in (B) and (C). Combined, they yield the amplitude of the gradients by means of equation 26.6 (D) and the dominant orientation (E) of the gradient by means of equation 26.7.

#h3 28.7 - 2D Convolution on images - Orientation Tuning

#pg In the previous chapter we discussed the response properties of neurons in biological visual systems. When visual information reaches the first stage of cortical processing, we find cells that are predominantly responsive to edges, much like the Sobel filters above. A typical neuron will favor these edges to be 1) at some part of the visual field, as the neuron only received inputs from that location, 2) oriented in a certain way, which we call the orientation tuning of the neuron and 3) the absence of other overlapping orientations that it isn't tuned to and 4) the spatial frequency of the edge.

#im ../../../assets/figures/028/028-09.png 50 256 Figure 28.9 - The first neurons in the mammalian visual system to receive inputs from the retina can be characterized by their so-called receptive field and orientation tuning. The receptive field indicates what part of the visual field (or image) elicits a response of the neuron. The orientation tuning reflects the neuron having a preferred orientation of the edges within its receptive field to which is responds maximally. We used to idealize receptive fields with preferred orientations of 30 and -60 degrees respectively. Like the Sobel operator, the neurons in the visual system can be seen as filters or kernels that detect local light-dark gradients in our visual field. Similarly, the nodes in the early layers of a cDBNNs trained for a machine vision will learn weights that when visualized spatially, look remarkably like those discovered in biological systems.

#br

#pg And the presence of such oriented edge detectors isn't unique to biological system. As we will see later, the convolutional DBNNs that have been so successful in machine vision object detecting and recognition have nodes in their early layers that have remarkably similar properties to those found in biological systems.

#h3 28.8 - Convolutional Layers in cDBNNs

#pg In a traditional feedforward DBNNs the nodes of the input layer typically project to all the nodes in the next hidden layer. In a convolutional DBNN, this is not the case. Assuming our inputs are images, like photographs, stills, or produced by medical imaging techniques like MRI or CAT, a convolutional layer consists of nodes that only receive inputs from a particular subregion of the entire input image. Dividing the image into a collection of partially overlapping squares, each of these squares connects to a particular subset of nodes. The weights of each of these nodes can be seen as operating as representing filters, as each pixel of the square subregion feeds into the node with a particular weight. Multiple nodes see the same square subregion, with the implicit aim for these nodes to learn, by training the model, the different types of filters we encountered before when discussing 2D convolution. Again, there is a very strong analogy here with biological visual systems, such as our own. As we discussed in chapter 25, in the early stages of visual processing, neurons in the visual cortex 'see' only a bit of the entire visual field, and each neuron is tuned, meaning that is effectively implements a filter. For example, a neuron might be tuned to a dark/white pattern oriented at 45 degrees - an edge. Many neurons see that same bit of the visual field, and all of them have slightly different tunings. Together, there activity essentially encodes that bit of the visual field. Similarly, in cDBNNs, the nodes in the convolutional layers too encode certain filters, and each of them a slightly different one if the model is trained correctly.

#h3 28.9 - Layers upon Layers

#pg In cDBNNs you will typically find a sequence of convolutional layers. This is another idea that follow biological neural networks closely. In the human visual system, the early stages of visual processing are hyperlocal - the neurons only see a tiny bit of the entire visual field. However, neurons that are nearby then all project to one or more nodes in the next layer, which effectively makes the bit of visual field those nodes see a bit larger. At this scale, the visual system uses more complex filters, rather than just simple line detectors. The same is true for cDBNNs. Deeper convolutional layers are harder to grasp as to what their function is, but they combine information over larger areas of the image than earlier layers. What exactly these filters have learned to detect is the subject of the next chapter, where we will visualize the models, we train using the notebooks accompanying the current chapter.

#h3 28.10 - Why convolution is such a powerful concept

#pg So why does this strategy, that of layers of local filters of increasing size and complexity works, so well in both artificial and biological neural networks when processing images? You might wonder why we went through such great lengths of explaining convolution. Why use convolution at all? Are the DBNNs we explored in the last chapter not already powerful enough? Can we just add a few nodes or layers, and not bother with this extra bit of complexity? There are a few good reasons for this that are all interrelated, but equally true for both our own brains and those we implement on our computers.

#h3 28.11 - Statistics of Natural Images

#pg The images that both artificial and biological neural networks take as input are typically natural images. A natural image is what we expect the world to look like: containing objects, lit by light from some source, casting shadows, being partially occluded, etc. Even if the objects are not recognized, if we were on an alien world for example, we would still comprehend the composition of these objects and their color, shape, relative positioning, etc. Natural images are a subset of all possible images with statistics associated with them. Take a high-resolution image, which at the time of writing, we can say would probably be around 4K. A 4K image is typically 3840 by 2160 pixels, and with 3 channel s(red, green, and blue) at a bit depth of 8, which allows us to construct 3840 times 2160 times 3 times 256 or 6,370,099,200 uniquely different images. Out of the 6.37 billion images, only a small subset of those are natural images. Most of these images will appear as noise. And other than obviously depicting a scene with objects of different shapes and colors, these images have a unique statistic: the correlation between the intensity of two pixels is a function of their distance. Nearby pixels correlate strongly, far away pixels, in opposite corner of the image, do not. Therefore, both in biological and artificial neural networks it makes sense to have the early nodes only receive inputs from a small region of the image. If I am a node (or neuron) in charge of detecting horizontal lines in the top left corner of the image (or visual field) I do not need to know about anything happening in the bottom right corner.

#im ../../../assets/figures/028/028-10.png 50 256 Figure 28.10 - Statistics of Natural Images. Left panel shows a distribution of Euclidean distances between the RGB values of two pixels as a function of the distance between the pixels in the image.

#br

#pg Visual information at the lowest level is strictly local. Objects are continuous in space and occupy an isolated region within the image, rather than its features appearing everywhere in the image with any coherence. The fact that nearby pixels are more similar than pixels separated by large distances in the image can easily be tested. Figure 28.10 shows the result of a quick calculation on many natural images. In short, we took the RGB vector of each pixel and computed the Euclidean distances as well as the cosine dissimilarity metric. Cosine dissimilarity measures the relative angle between two vectors (in our case given by the RGB vectors of each pixel considered) but not difference in length. This removes differences in intensity between pixels. A pixel with an RGB vector of [1,0,0] (bright red) points in the exact same direction as a RGB vector of [0.5,0.,0] (dark red), whereas Euclidean distance still considers the intensity (0.5 vs 1.0). Regardless, both measures increase as interpixel distance is increased, indicating that the greater the distance between two pixels, the less similar their RGB values are. Interestingly, at a certain point the similarity between pixels starts to increase again. This is an artifact at large distances between pixels. To be far apart, pixels will find themselves close the edges of the image, which is also typically where we see most of the background included in the image. Backgrounds generally are rather homogenous, like skies, walls, forests, cityscapes, etc. The homogeneity on the outskirts of the images drives the eventual increase in similarity between far away pixels.

#h3 28.12 - Efficiency

#pg With this locality principle comes another great perk. Since the convolutional layers are only sparsely connected, with each node only receiving inputs from a small region of the image, the number of connections is dramatically reduced relative to a fully connected network. And the reducing in connections means a reduction in the number of weights that need to be optimized. This makes training cDBNNs faster and more robust, and less prone to overfitting.

#h3 28.13 - Transfer Learning

#pg The final benefit of the convolutional approach is again rooted in the idea of natural images. A convolutional neural network trained on images will have learned a hierarchy of visual features of increasing complexity. Transfer learning is the re-purposing of the first sequence of layers to re-train the entire model to recognize new visual object categories. The core idea is that all visual objects can be decomposed into low level features. Both a picture of a cat and a picture of the Large Hadron Collider in Switzerland share certain visual features: gradients, edges, line segments, patterns, texture, colors, etc. Therefore, a neural network trained to detect Large Hadron Colliders can be repurposed to also detect cats. What we allow to be retrained (what weights are updated during training) is only the top layers where object specific features start to matter, while at the same time all the low-level features that describe both cats and supercolliders are not retrained. This dramatically reduced the time needed to train a network to recognize a novel visual object category as the early layers of a convolutional neural network have already learned the low-level features that generalize to recognizing all possible visual object categories.

#im ../../../assets/figures/028/028-11.png 50 320 Figure 28.11 - LeNet-5 architecture, as described in LeCunn, et al, 1998.

#h3 28.14 - Our first convolutional neural network

#pg Our first network, for which the code can be found in notebooks 28.04 and 28.05 will use the convolutional network architecture that is considered the common ancestor of all convolutional neural networks, LeNet (LeCunn et al., 1989). Combining newly developed back-propagation algorithms with a novel convolutional neural network architecture, LeCunn et al. became pioneers of image classification using deep learning. Although it still had a set of traditional fully connected or dense layers, the inputs first passed through a sequence of 3 convolution layers. In the LeNet architecture, the first two convolutional layers consist of three distinct stages: convolution, application of tanh activation function and subsampling by means of maximum pooling (MaxPooling for short).

#im ../../../assets/figures/028/028-12.png 50 320 Figure 28.12 - Non-linear activation functions. (A) Sigmoidal activation, (B) Tanh activation function, (C) Rectified Linear Unit (ReLU).

#br

#pg At this point it is worth pointing out that numerous implementations of the LeNet architecture can be found online, that oddly enough, they do seem to differ a bit in certain details. In the original paper, a sigmoid activation function is applied to the output of the MaxPooling layers, in addition to the tanh activation applied to the output of the convolution layers. However, many of the online tutorials and demos seem to omit the application of the sigmoid. As such, we have created two models per dataset we used to train our own version of LeNet on. One notebook will have the architecture described in the original paper, the other will omit the sigmoid. We have seen the tanh and sigmoid activation functions in previous chapters, as well as the ReLu activation function, all three can be seen in Fig. 28.12.

#im ../../../assets/figures/028/028-13.gif 50 256 Figure 28.13 - In the first convolution layer, the input images divided into 5x5 grids. Scanning over the input image of 32x32 pixels with a stride of 1 (each grid overlaps almost completely with its neighbors yield 28 x 28 of 5x5 pixel grids which are each convolved with a feature map of size 6. Please note that we only show the animation looping over the center grid, to preserve some bandwidth needed for the animation to load and play.

#h3 28.15 - Convolution Stage - Layer 1

#pg In the LeNet architecture, the input image is divided into a set of overlapping smaller grids of 5 by 5 pixels. Each of these squares is convolved with 6 filters. The total set of filters is called a feature map. Each of these squares is convolved with 6 filters. At the beginning of training the network, these filters are initialized to small random numbers, with the expectation that during training, they will gradually take on structure. For example, for the MNIST data, we expect these filters to resemble the line and edge detectors we have seen previously in this chapter, both in biological neural networks, or mathematically constructed, like the Canny and Sobel filters. The overlap between the 5x5 grids is specified by the stride. In the LeNet architecture, the stride for the convolution layers is 1. This means that each grid is shifted 1 pixel relative to the previous grid. In other words, the grids almost completely overlap. Slide a 5x5 grid with a stride of 1 across a 32x32 pixel input image yields 28 \* 28 of such grids. After computing the output of the convolution of each grid with each of its 6 associated filters, a Tanh activation function is applied to it, creating a useful nonlinearity that can be exploited to learn more complex filters.

#h3 28.16 - MaxPooling Stage - Layer 1

#pg After the convolution stage and the application of the activation function, the information passes through a second stage, still considered be some to be part of the same layer (opinions are somewhat divided on this), that of MaxPooling. MaxPooling aims to subsample the image, by sliding a 2x2 window across the 28x28 output of the convolution layer with a stride of 2, within each window, the maximum of the 4 (2x2) values is retained. This thus yields a new matrix for each feature map, size 14 by 14. A stride of two for a window with a size of 2x2 means that there is no overlap between any of the smaller grids created by the MaxPooling operation. Again, the animation below probably does a better job in conveying the exact nature of the MaxPooling operation, more so than we can express in words alone. The output of the MaxPooling operation too is passed through an activation function. In the original LeNet implementation, this is now a sigmoid function, rather than tanh function we used after our convolution operation.

#im ../../../assets/figures/028/028-14.gif 50 256 Figure 28.14 - In the first MaxPooling layer, the input images divided into 2x2 grids. Scanning over the input image of 28x28 pixels with a stride of 0 (grids do not overlaps at all) yield 14 x 14 of 2x2 pixel grids. The 2x2 grids are subsampled by only retraining the maximum of the 4 values in each 2x2 grid.

#h3 28.17 - Convolution Stage - Layer 2

#pg After the MaxPooling operation in the first convolutional layer, we now have as input to the second convolutional layer feature map activations of size 14 by 14, with a depth of 6. Whereas height and width denote the dimensions of either the image or the feature maps, depth indicates the number of features maps at each grid location. In the second layer, these features maps are convolved with a new set of filters, again 5 x 5 in width and height. However, now the depth is increased to 16, meaning that each grid position has 16 filters. Sticking to a stride of 1, a 14 by 14 input image yields 10 x 10 smaller grids, at a filter size of 5 x 5. And like the first convolutional layer, a Tanh activation function is applied to output of the filters.

#im ../../../assets/figures/028/028-15.gif 50 256 Figure 28.15 - In the second convolution layer, the input from the previous layer (MaxPooling layer 1) is again into 5x5 grids. Scanning over the input image of 14x14 pixels with a stride of 1 (each grid overlaps almost completely with its neighbors yield 10 x 10 of 5x5 pixel grids which are each convolved with a feature map of size 16.

#h3 28.18 - MaxPooling Stage - Layer 2

#pg Like Layer 1, we apply MaxPooling to the output of the convolutional layer, again with a stride of 2 and a size of 2 by 2. The 10 x 10 input thus is down sampled again, this time to a size of 5 x 5. And again, a sigmoid activation function is applied to the output of the MaxPooling operation.

#im ../../../assets/figures/028/028-16.gif 50 256 Figure 28.16 - In the second MaxPooling layer, the input from the previous convolution layers is again divided into 2x2 grids. Scanning over the input image of 10x10 pixels with a stride of 0 (grids do not overlaps at all) yield 5 x 5 of 2x2 pixel grids. And like the first MaxPooling layer, the 2x2 grids are subsampled by only retraining the maximum of the 4 values in each 2x2 grid.

#h3 28.19 - Convolution Stage - Layer 3

#pg The final convolution layer only applies convolution. Sticking to a filter size of 5x5, since the image has been down samples through the feature maps to a size of 5 x 5, we only have one grid, encompassing the entire input. This time around, though, we increase the number of filters at each grid position (which now happens to be just one) to 120. No activation function is applied at this stage, instead the output (a vector of 120 is fed into the next layer which we detail in the next paragraph

#im ../../../assets/figures/028/028-17.png 50 256 Figure 28.17 - MNIST and CIFAR10 example images. The famous MNIST dataset has been the benchmark for computer vision for a long time, but for the modern generation of machine vision cDBNNs, it no longer poses any challenges. Instead, algorithms are now benchmarked on much larger data sets consisting of photographs of much higher resolution than the MNIST dataset. The CIFAR10 dataset falls somewhere in between these two extremes, with a small number of categories and low-resolution images.

#h3 28.20 - Dense Layers 4 and 5

#pg After the three convolutional layers, LeNet has two more additional layers. These are dense, fully connected layers, as opposed to the sparse layers that implement the convolutions and MaxPooling operations. First, the data from convolutional layer 3 is flattened, meaning it is made 1-dimensional. Since there were 120 filters in the layers in the last convolutional layer, we thus have a vector of length 120 that serves as input to the first dense layer. This dense layer itself has 84 nodes, meaning that a total of 121 \* 84 weights must be trained between layers 3 and 4. Why 121 and 120? We need one more node to serve as a bias node, which activation is always 1. This bias node has a weight on each of the 84 nodes in layer 4, so the total number of weights is 120 \* 84 + 84 = 121 \* 84 = 10164 weights. Finally, a fifth SoftMax layer (see chapter 24) reduces the 84 nodes of layer 4 to 10, the number of classes in the MNIST and CIFAR 10 data. Therefore, the output of each node, after training, represents the probability of the input being one of the 10 digits.

#im ../../../assets/figures/028/028-18.png 50 256 Figure 28.18 - Performance of LeNet (top row), compared to a conventional DBNN with a similar number of parameters trained on the MNIST dataset. For this relatively simple dataset in terms of input (image) complexity and diversity, both convolutional and conventional DBNNs achieve high accuracies. Left column shows the gradual increase in accuracy for each epoch of training, the middle column the associated decrease in loss. The right column shows the confusion matrix: the relationship between the predicted and actual digit in the images.

#h3 28.21 - Training the LeNet architecture on the MNIST and CIFAR10 Data

#pg The notebooks are a guide for implementing the LeNet architecture and applying it to both the classic MNIST handwritten digit data, as well as the CIFAR10 images, which contain small images across ten classes of visual objects. To highlight the benefit of using convolutional layers, we will also train a conventional feedforward DBNN to predict the MNIST and CIFAR10. For these CBNNs, we keep the dense layers 4 and 5 from the LeNet architecture but replace the convolutional and MaxPooling layers with 5 dense layers, calibrating the number of nodes in each of these 5 layers such that the total number of trainable parameters is roughly equal to the LeNet-5 architecture.

#im ../../../assets/figures/028/028-19.png 50 256 Figure 28.19 - Performance of LeNet, compared to a conventional DBNN with a similar number of parameters trained on the CIFAR10 dataset. Due to the higher complexity and diversity of images within each visual object category, we see the conventional DBNN fall behind in terms of accuracy by a substantial margin. Please do also note that the CIFAR10 data might also be too complex for the LeNet architecture itself, yielding only modest accuracies at best.

@br

@pg This should allow for at least a reasonable comparison between conventional feedforward and convolutional neural networks when it comes to image classification. After training, performance of the LeNet cDBNN on the MNIST data set is near perfect, close to human error rate. A conventional DBNN achieves a similar level of performance, lagging only slightly behind the convolutional LeNet architecture in terms of accuracy, see Fig. 28.18. For the CIFAR10 dataset however, the convolutional DBNN, although nowhere near 100% accuracy itself, still greatly outperforms a conventional DBNN with a roughly equal number of trainable parameters (weights), see Fig. 28.19.

#h3 28.22 - AlexNet Architecture

#pg LeNet marked the beginning of a long line of convolutional DBNNs of ever-increasing complexity, number of layers and number of visual object categories they were trained on. The biggest step forward, undoubtedly, is the AlexNet architecture and its accuracy when applied to the ImageNet competition data. ImageNet is an image database organized according to the WordNet hierarchy for nouns, in which each node of the hierarchy is depicted by hundreds and thousands of images. The project has been instrumental advancing computer vision and deep learning research and the images are available for free to researchers for non-commercial use. Since 2010, a yearly ImageNet challenge is held, where different algorithms compete for highest accuracy in correctly classifying visual objects. AlexNet, named after its creator Alex Krizhevsky and in collaboration with Ilya Sutskever and Geoffrey Hinton, was a big step forward, compared to LeNet. First, it had 5 convolutional layers, compared to the three in LeNet. Second, it used a novel non-linear activation function, the Rectified Linear Unit or ReLu. This activation function has several advantages over more traditional ones, like the sigmoid and tanh activation functions. The main benefit is that its use creates sparse representations of different visual objects within the network. For example, in a randomly initialized network, only about 50% of hidden units are activated (have a non-zero output). Sparse representations emphasize the difference between classes of visual objects, rather than their similarities. Which, in turn, allows the neural network to discriminate between similar looking, but different object (e.g., two similar looking but distinct breeds of dogs). The ReLu also yields more stable backpropagation as well allowing for a highly efficient computations: only addition and multiplications are needed.

#im ../../../assets/figures/028/028-20.png 50 256 Figure 28.20 - AlexNet Architecture. The AlexNet architecture has a sequence of 5 convolutional layers, 3 of which are combined with a MaxPooling operation (Conv1, Conv2 and Conv5), and 2 of which are combined with a Batch Normalization operation (Conv1 and Conv2). The convolution stages are followed by three fully connected dense layers of size 2048, 2048 and 1000 nodes respectively. The final of these layers represents the output of the network, applying a SoftMax operation to the activation such that the 1000 element vector of outputs represents the probability of an image containing on of the 1000 visual object classes in the ImageNet data it was trained on.

But what is perhaps the biggest step forward is that by using GPUs to train the model, the authors were able to scale up their neural network so realistically sized images could be ingested. Whereas the MNIST and CIFAR images are black and white images of 28x28 and 32x32 pixels respectively, in contrast, AlexNet allowed for input images of size 227x227x3 - reasonable sized full color images. Two more innovations to neural networks in general and convolutional neural networks specifically were included in the AlexNet architecture: batch normalization and dropout. We discussed Dropout in the previous chapter as an additional technique to improve weight optimization (i.e., learning). In short, dropout, or silencing a particular random subset of nodes each training epoch, prevents the nodes to become overly specialized to a single and highly specific feature. Batch normalization was used to prevent drift in the weights introduced by the fact that nodes are continuously being updated during training. Between each layer, the outputs of the sending layers in response to a batch of training data are normalized to a mean 0 and standard deviation of 1. This simple trick proved to be highly effective in stabilizing and speeding up training the network.

#h3 28.23 - AlexNet - Small Scale

#pg It is possible to train AlexNet on the full ImageNet Challenge dataset (also commonly referred to as ILSVRC-2012) (>1.28 million images over 1000 categories), even without GPU support. However due to its size, expect training to last hours, if not days, on a regular MacBook Pro or Desktop PC. In addition, the sheer number of images creates an I/O bottleneck, because we can't load all the images into memory prior to training. We will discuss how solve this using TensorFlow code in the next section. For now, we will create and train a more modest version of AlexNet, one that uses images of 128 x 128 pixels, taken from only a handful of ImageNet categories

#h3 28.24 - AlexNet - Full Scale - CPU vs GPU

#pg If you do happen to have the necessary GPU hardware, you can use the code in the notebooks to also train AlexNet (or any other network architecture can imagine or have found online) on the full ImageNet data. If you have a desktop or laptop computer that uses one or more NVidia graphics cards, please visit nvidia.com to find instructions on how to enable GPU support in TensorFlow. NVidia has specific drivers and tools (CUDA and CUDnn) that TensorFlow (and other deep learning frameworks) use to run the computations directly on the GPUs. Alternatively, you can spin up a GPU based EC2 machine on AWS that has all the appropriate drivers, tools and software preinstalled and configured (such as AWS Sagemaker). Keep in mind that GPU are still costly and check your usage often to check that you are not amassing a giant bill just by training one moderately sized neural network (yes, at the time of writing, AlexNet is already best described as moderately sized, newer convolutional neural networks like Inception, VGG and MobileNet have many more layers and many more trainable parameters). Whereas GPUs can greatly decrease the time needed for all the computations, it doesn't solve the issue of large I/O overhead. With the latter we mean that we can't really load the entire ImageNet dataset into memory prior to training. Unpacked, the training data of ImageNet is about 130 Gigabytes. While our hard drives have no issue storing that much data, it does far exceed available RAM of most commercially available computer hardware. Therefore, the code in the notebook that allows you to train AlexNet on all the ImageNet data also introduces a new way of fetching and reading large amount of training data on each training epoch. TensorFlow has several tools that allow data to be prefetched or otherwise handled and read from disk in an optimal way as to reduce the I/O bottleneck during training.

#im ../../../assets/figures/028/028-21.png 50 256 Figure 28.21 - Transfer Learning - Jet fighters and commercial airplanes

#h3 28.25 - Transfer Learning

#pg As we discussed briefly above, convolutional networks are particularly well suited for an approach called transfer learning. Again. Natural images are defined by the presence of edges, gradients, line segments, patterns, textures, colors, boundaries between colors and these are the fundamental building blocks of things we see and things we take pictures of. A convolutional neural network typically consists of a cascade of convolutional layers, followed by a few more traditional fully connected layers. That means that once a convolutional neural network is trained to categorize some general set of images, like the ImageNet data set which contains 1000 different visual objects, we can repurpose those early convolutional layers. Whereas the later dense layers have learned to recognize the 1000 original objects in the ImageNet data, the early layers have learned something more general - the filters that detect the building blocks of natural images, beyond the 1000 categories we trained on. Transfer learning is the method of taking such a pretrained network and retraining it on a new category, say LEGO blocks. But instead of retraining the whole network, we only optimize the weights in the fully connected layers. After all, the convolutional layers are already trained to detect certain lower-level visual elements. Elements that images of LEGO blocks surely also contain. Transfer learning is highly efficient, and retraining a model takes only a fraction of the time. In addition, it requires fewer data points, as most of the work has already been done during the initial training on the large and general dataset of images. The notebook that accompanies this paragraph uses images of jet fighters and commercial airliners as two new visual object categories, but we encourage the reader to explore their own set of visual objects they might have an interest in.

#h3 26.26 - Discussion

#pg Convolutional Neural Networks and their ability to do so well on visual detection, location and categorization tasks has made them into the poster child of machine vision and deep belief neural networks in general. Whereas we focused mainly on object categorization (only briefly touching on segmentation) there are many more complex and interesting visual tasks our cDBNNs have shown to be able to perform. The latest generation of cDBNNs taking in images and or video can now verbally explain the content of the image (this is an image of a person sitting on a chair in front of a desk, on the desk there is an abacus and a pencil and a notebook). Perhaps more exciting (and equally concerning if not outright freighting) is the ability of these neural networks to generate visual content, rather than just categorizing or naming it. cDBNNs are now capable of producing photorealistic looking images, generating environments, landscapes, objects, and even human faces of people who do not actually exist. We will discuss such neural nets in chapter 29.

#h3 28.27 - Notebooks and Demos

#pg This chapter comes with several notebooks. The first few notebooks follow our discussion on convolution as a mathematical operator, in the context of both 1d timeseries and 2D images. Subsequent notebooks will have demo implementations of LeNet, trained on two different datasets (CIFAR10 and MNIST), as well as both a full and scaled-down version of AlexNet. The two final notebooks work through an example of transfer learning and image segmentation using an autoencoder respectively.

#h3 28.28 - References

#bs

#be

#bp LeCunn Y., Bottou L., Bengio Y., and Haffner P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86-11, 2278-2324.

#bp Simonyan, K. and Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. https://arxiv.org/pdf/1409.1556.pdf

#bp Krizhevsky A., Sutskever, I., and Hinton, G. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM. 60-6, 84–90.

#bp ImageNet Large Scale Visual Recognition Competition ILSVRC2012 (2012). image-net.org.

#bp Bengio, Y., Goodfellow, I., and Courville, A. (2016). Deep Learning. United Kingdom: MIT Press.

#bp Widder, DV., and Hirschman, I. (2012). The Convolution Transform. United States: Dover Publications.

#bp Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. Biological Cybernetics, 20-3, 121-136.

#bp Nair, V. and Hinton, G. (2010). Rectified linear units improve restricted Boltzmann machines. ICML 2010, 807-814.

#be