

VII.2 Convolutional Neural Nets

This section is about networks with a different architecture. Up to now, each layer was fully connected to the next layer. If one layer had n neurons and the next layer had m neurons, then the matrix A connecting those layers is m by n . *There were mn independent weights in A .* The weights from all layers were chosen to give a final output that matched the training data. The derivatives needed in that optimization were computed by backpropagation. Now we might have only 3 or 9 independent weights per layer.

That fully connected net will be extremely inefficient for image recognition. First, the weight matrices A will be huge. If one image has 200 by 300 pixels, then its input layer has 60,000 components. The weight matrix A_1 for the first hidden layer has 60,000 columns. The problem is: We are looking for connections between faraway pixels. Almost always, the important connections in an image are **local**.

Text and music have a 1D local structure: a time series

Images have a 2D local structure: 3 copies for red-green-blue

Video has a 3D local structure: Images in a time series

More than this, the search for structure is essentially the same everywhere in the image. There is normally no reason to process one part of a text or image or video differently from other parts. We can use the same weights in all parts: *Share the weights*. The neural net of local connections between pixels is **shift-invariant**: the same everywhere.

The result is a big reduction in the number of independent weights. Suppose each neuron is connected to only E neurons on the next layer, and those connections are the same for all neurons. Then the matrix A between those layers has only E independent weights x . The optimization of those weights becomes enormously faster. In reality we have time to create several different channels with their own E or E^2 weights. They can look for edges in different directions (horizontal, vertical, and diagonal).

In one dimension, a banded shift-invariant matrix is a **Toeplitz matrix** or a **filter**. Multiplication by that matrix A is a **convolution** $x * v$. The network of connections between all layers is a **Convolutional Neural Net (CNN or ConvNet)**. Here $E = 3$.

$$A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\ 0 & x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} & 0 \\ 0 & 0 & 0 & x_1 & x_0 & x_{-1} \end{bmatrix} \quad \begin{array}{l} \mathbf{v} = (v_0, v_1, v_2, v_3, v_4, v_5) \\ \mathbf{y} = A\mathbf{v} = (y_1, y_2, y_3, y_4) \\ N + 2 \text{ inputs and } N \text{ outputs} \end{array}$$

It is valuable to see A as a combination of shift matrices L, C, R : Left, Center, Right.

$$\text{Each shift has a diagonal of 1's} \quad A = x_1 L + x_0 C + x_{-1} R$$

Then the derivatives of $\mathbf{y} = A\mathbf{v} = x_1 L\mathbf{v} + x_0 C\mathbf{v} + x_{-1} R\mathbf{v}$ are exceptionally simple:

$$\boxed{\frac{\partial \mathbf{y}}{\partial x_1} = L\mathbf{v} \quad \frac{\partial \mathbf{y}}{\partial x_0} = C\mathbf{v} \quad \frac{\partial \mathbf{y}}{\partial x_{-1}} = R\mathbf{v}} \quad (1)$$

Convolutions in Two Dimensions

When the input \mathbf{v} is an image, the convolution with \mathbf{x} becomes two-dimensional. The numbers x_{-1}, x_0, x_1 change to $E^2 = 3^2$ independent weights. The inputs v_{ij} have two indices and \mathbf{v} represents $(N + 2)^2$ pixels. The outputs have only N^2 pixels unless we pad with zeros at the boundary. The 2D convolution $\mathbf{x} * \mathbf{v}$ is a *linear combination of 9 shifts*.

$$\text{Weights} \begin{bmatrix} x_{11} & x_{01} & x_{-11} \\ x_{10} & x_{00} & x_{-10} \\ x_{1-1} & x_{0-1} & x_{-1-1} \end{bmatrix} \begin{array}{l} \text{Input image } v_{ij} \quad i, j \text{ from } (0, 0) \text{ to } (N + 1, N + 1) \\ \text{Output image } y_{ij} \quad i, j \text{ from } (1, 1) \text{ to } (N, N) \\ \text{Shifts } \mathbf{L}, \mathbf{C}, \mathbf{R}, \mathbf{U}, \mathbf{D} = \text{Left, Center, Right, Up, Down} \end{array}$$

$$A = x_{11}LU + x_{01}CU + x_{-11}RU + x_{10}L + x_{00}C + x_{-10}R + x_{1-1}LD + x_{0-1}CD + x_{-1-1}RD$$

This expresses the convolution matrix A as a combination of 9 shifts. The derivatives of the output $\mathbf{y} = A\mathbf{v}$ are again exceptionally simple. We use these nine derivatives to create the gradients ∇F and ∇L that are needed in stochastic gradient descent to improve the weights \mathbf{x}_k . The next iteration $\mathbf{x}_{k+1} = \mathbf{x}_k - s\nabla L_k$ has weights that better match the correct outputs from the training data.

These nine derivatives of $\mathbf{y} = A\mathbf{v}$ are computed inside backpropagation:

$$\frac{\partial \mathbf{y}}{\partial x_{11}} = LU\mathbf{v} \quad \frac{\partial \mathbf{y}}{\partial x_{01}} = CU\mathbf{v} \quad \frac{\partial \mathbf{y}}{\partial x_{-11}} = RU\mathbf{v} \quad \dots \quad \frac{\partial \mathbf{y}}{\partial x_{-1-1}} = RD\mathbf{v} \quad (2)$$

CNN's can readily afford to have **B parallel channels** (and that number B can vary as we go deeper into the net). The count of weights in \mathbf{x} is so much reduced by weight sharing and weight locality, that we don't need and we can't expect one set of $E^2 = 9$ weights to do all the work of a convolutional net.

Let me highlight the operational meaning of convolution. In 1 dimension, the formal algebraic definition $y_j = \sum x_i v_{j-i} = \sum x_{j-k} v_k$ involves a "flip" of the v 's or the x 's. This is a source of confusion that we do not need. We look instead at left-right shifts L and R of the whole signal (in 1D) and also up-down shifts U and D in two dimensions. Each shift is a matrix with a diagonal full of 1's. That saves us from the complication of remembering flipped subscripts.

A convolution is a combination of shift matrices (producing a filter or Toeplitz matrix)

A cyclic convolution is a combination of cyclic shifts (producing a circulant matrix)

A continuous convolution is a continuous combination (an integral) of shifts

In deep learning, the coefficients in the combination will be the "weights" to be learned.

Two-dimensional Convolutional Nets

Now we come to the real success of CNN's: **Image recognition**. ConvNets and deep learning have produced a small revolution in computer vision. The applications are to self-driving cars, drones, medical imaging, security, robotics—there is nowhere to stop. Our interest is in the algebra and geometry and intuition that makes all this possible.

In two dimensions (for images) the matrix A is **block Toeplitz**. Each small block is E by E . This is a familiar structure in computational engineering. The count E^2 of independent weights to be optimized is far smaller than for a fully connected network.

The same weights are used around all pixels (*shift-invariance*). The matrix produces a 2D convolution $\mathbf{x} * \mathbf{v}$. Frequently A is called a **filter**.

To understand an image, look to see where it changes. *Find the edges*. Our eyes look for sharp cutoffs and steep gradients. Our computer can do the same by creating a filter. The dot products between a smooth function and a moving filter window will be smooth. But when an edge in the image lines up with a diagonal wall, *we see a spike*. Those dot products (fixed image) \cdot (moving image) are exactly the “**convolution**” of the two images.

The difficulty with two or more dimensions is that edges can have many directions. We will need horizontal and vertical and diagonal filters for the test images. And filters have many purposes, including smoothing, gradient detection, and edge detection.

1 Smoothing For a 2D function f , the natural smoother is *convolution with a Gaussian*:

$$Gf(x, y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} * f = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} * \frac{1}{\sqrt{2\pi}\sigma} e^{-y^2/2\sigma^2} * f(x, y)$$

This shows G as a product of 1D smoothers. The Gaussian is everywhere positive, so it is *averaging*: Gf cannot have a larger maximum than f . The filter removes noise (at a price in sharp edges). For small variance σ^2 , details become clearer.

For a 2D vector (a matrix f_{ij} instead of a function $f(x, y)$) the Gaussian must become discrete. The perfection of radial symmetry will be lost because the matrix G is square. Here is a 5 by 5 discrete Gaussian G ($E = 5$):

$$G = \frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \approx \frac{1}{289} \begin{bmatrix} 1 \\ 4 \\ 7 \\ 4 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \end{bmatrix} \quad (3)$$

We also lost our exact product of 1D filters. To come closer, use a larger matrix $G = \mathbf{x}\mathbf{x}^T$ with $\mathbf{x} = (.006, .061, .242, .383, .242, .061, .006)$ and discard the small outside pixels.

2 Gradient detection Image processing (as distinct from learning by a CNN) needs filters that detect the gradient. They contain specially chosen weights. We mention some simple filters just to indicate how they can find gradients—the first derivatives of f .

$$\mathbf{E} = \mathbf{3} \quad (x_1, x_0, x_{-1}) = \left(-\frac{1}{2}, \mathbf{0}, \frac{1}{2}\right) \quad \left[\left(\frac{1}{2}, 0, -\frac{1}{2}\right) \text{ in convolution form}\right]$$

In this case the components of $A\mathbf{v}$ are centered differences: $(A\mathbf{v})_i = \frac{1}{2}v_{i+1} - \frac{1}{2}v_{i-1}$. When the components of \mathbf{v} are increasing linearly from left to right, as in $\mathbf{v}_i = 3i$, the output from the filter is $\frac{1}{2}3(i+1) - \frac{1}{2}3(i-1) = 3 = \text{correct gradient}$.

The flip to $(\frac{1}{2}, 0, -\frac{1}{2})$ comes from the definition of convolution as $\sum x_{i-k}v_k$.

Two dimensions These 3×3 Sobel operators approximate $\partial/\partial x$ and $\partial/\partial y$:

$$\mathbf{E} = \mathbf{3} \quad \frac{\partial}{\partial x} \approx \frac{1}{2} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \frac{\partial}{\partial y} \approx \frac{1}{2} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad (4)$$

For functions, the gradient vector $\mathbf{g} = \text{grad } f$ has $\|\mathbf{g}\|^2 = |\partial f/\partial x|^2 + |\partial f/\partial y|^2$.

Those weights were created for image processing, to locate the most important features of a typical image: *its edges*. These would be candidates for E by E filters inside a 2D convolutional matrix A . But remember that in deep learning, weights like $\frac{1}{2}$ and $-\frac{1}{2}$ are not chosen by the user. They are created from the training data.

Sections IV.2 and IV.5 of this book studied cyclic convolutions and Toeplitz matrices. Shift-invariance led to the application of discrete Fourier transforms. But in a CNN, ReLU is likely to act on each neuron. The network may include zero-padding—as well as max-pooling layers. So we cannot expect to apply the full power of Fourier analysis.

3 Edge detection After the gradient direction is estimated, we look for edges—the most valuable features to know. “*Canny Edge Detection*” is a highly developed process. Now we don’t want smoothing, which would blur the edge. The good filters become *Laplacians of Gaussians* :

$$E f(x, y) = \nabla^2 [g(x, y) * f(x, y)] = [\nabla^2 g(x, y)] * f(x, y). \quad (5)$$

The Laplacian $\nabla^2 G$ of a Gaussian is $(x^2 + y^2 - 2\sigma^2) e^{-(x^2+y^2)/2\sigma^2} / \pi\sigma^4$.

The Stride of a Convolutional Filter

Important The filters described so far all have a *stride* $S = 1$. For a larger stride, the *moving window takes longer steps* as it moves across the image. Here is the matrix A for a 1-dimensional 3-weight filter with a stride of 2. Notice especially that the length of the output $\mathbf{y} = A\mathbf{v}$ is reduced by that factor of 2 (previously four outputs and now two) :

$$\mathbf{Stride } S = \mathbf{2} \quad A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} \end{bmatrix} \quad (6)$$

Now the nonzero weights like x_1 in L are two columns apart (S columns apart for stride S). In 2D, a stride $S = 2$ reduces each direction by 2 and the whole output by 4.

Extending the Signal

Instead of losing neurons at the edges of the image when A is not square, we can *extend the input layer*. We are “inventing” components beyond the image boundary. Then the output $\mathbf{y} = A\mathbf{v}$ fits the image block: equal dimensions for input and output.

The simplest and most popular approach is **zero-padding**: Choose all additional components to be *zeros*. The extra columns on the left and right of A multiply those zeros. In between, we have a square Toeplitz matrix as in Section IV.5. It is still determined by a much smaller set of weights than the number of entries in A .

For periodic signals, zero-padding is replaced by *wraparound*. The Toeplitz matrix becomes a circulant (Section IV.2). The Discrete Fourier Transform tells its eigenvalues. The eigenvectors are always the columns of the Fourier matrix. The multiplication $A\mathbf{v}$ is a *cyclic convolution* and the Convolution Rule applies.

A more accurate choice is to go beyond the boundary by *reflection*. If the last component of the signal is v_N , and the matrix is asking for v_{N+1} and v_{N+2} , we can reuse v_N and v_{N-1} (or else v_{N-1} and v_{N-2}). Whatever the length of \mathbf{v} and the size of A , all the matrix entries in A come from the same E weights x_{-1} to x_1 or x_{-2} to x_2 (and E^2 weights in 2D).

Note Another idea. We might accept the original dimension (128 in our example) and use the reduction to 64 as a way to apply **two filters C_1 and C_2** . Each filter output is downsampled from 128 to 64. The total sample count remains 128. If the filters are suitably independent, no information is lost and the original 128 values can be recovered.

This process is linear. Two 64 by 128 matrices are combined into 128 by 128: square. If that matrix is invertible, as we intend, the filter bank is *lossless*.

This is what CNN's usually do: Add more channels of weight matrices A in order to capture more features of the training sample. The neural net has a bank of B filters.

Filter Banks and Wavelets

The idea in those last paragraphs produces a **filter bank**. This is just a set of B different filters (convolutions). In signal processing, an important case combines a lowpass filter C_1 with a highpass filter C_2 . The output of $C_1\mathbf{v}$ is a smoothed signal (dominated by low frequencies). The output $C_2\mathbf{v}$ is dominated by high frequencies. A perfect cutoff by ideal filters cannot be achieved by finite matrices C_1 and C_2 .

From two filters we have a total of 256 output components. Then both outputs are subsampled. The result is 128 components, separated approximately into *averages and differences*—low frequencies and high frequencies. The matrix is 128 by 128.

Wavelets The wavelet idea is to repeat the same steps on the 64 components of the lowpass output ($\downarrow 2$) $C_1\mathbf{x}$. Then ($\downarrow 2$) C_1 ($\downarrow 2$) $C_1\mathbf{x}$ is an average of averages. Its frequencies are concentrated in the lowest quarter ($|\omega| \leq \pi/4$) of all frequencies. The mid-frequency output ($\downarrow 2$) C_2 ($\downarrow 2$) $C_1\mathbf{x}$ with 32 components will not be subdivided. Then $128 = 64 + 32 + 16 + 16$.

In the limit of infinite subdivision, *wavelets* enter. This low-high frequency separation is an important theme in signal processing. It has not been so important for deep learning. But with multiple channels in a CNN, frequency separation could be effective.

Counting the Number of Inputs and Outputs

In a one-dimensional problem, suppose a layer has N neurons. We apply a convolutional matrix with E nonzero weights. The stride is S , and we pad the input signal by P zeros at each end. How many outputs (M numbers) does this filter produce?

$$\boxed{\text{Karpathy's formula} \quad M = \frac{N - E + 2P}{S} + 1} \quad (7)$$

In a 2D or 3D problem, this 1D formula applies in each direction.

Suppose $E = 3$ and the stride is $S = 1$. If we add one zero ($P = 1$) at each end, then

$$M = N - 3 + 2 + 1 = N \quad (\text{input length} = \text{output length})$$

This case $2P = E - 1$ with stride $S = 1$ is the most common architecture for CNN's.

If we don't pad the input with zeros, then $P = 0$ and $M = N - 2$ (as in the 4 by 6 matrix A at the start of this section). In 2 dimensions this becomes $M^2 = (N - 2)^2$. We lose neurons this way, but we avoid zero-padding.

Now suppose the stride is $S = 2$. Then $N - E$ must be an even number. Otherwise the formula (4) produces a fraction. Here are two examples of success for stride $S = 2$, with $N - E = 5 - 3$ and padding $P = 0$ or $P = 1$ at both ends of the five inputs:

$$\text{Stride } \mathbf{2} \quad \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix} \quad \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix}$$

Again, our counts apply in each direction to an image in 2D or a tensor.

A Deep Convolutional Network

Recognizing images is a major application of deep learning (and a major success). The success came with the creation of AlexNet and the development of convolutional nets. This page will describe a deep network of local convolutional matrices for image recognition. We follow the prize-winning paper of Simonyan and Zisserman from ICLR 2015. That paper recommends a deep architecture of $L = 16$ – 19 layers with small (3×3) filters. The network has a breadth of B parallel channels (B images on each layer).

If the breadth B were to stay the same at all layers, and all filters had E by E local weights, a straightforward formula would estimate the number W of weights in the net:

$$\boxed{W \approx LBE^2 \quad L \text{ layers, } B \text{ channels, } E \text{ by } E \text{ local convolutions}} \quad (8)$$

Notice that W does not depend on the count of neurons on each layer. This is because A has E^2 weights, whatever its size. Pooling will change that size without changing E^2 .

But the count of B channels can change—and it is very common to end a CNN with fully-connected layers. This will radically change the weight count W !

It is valuable to discuss the decisions taken by Simonyan and Zisserman, together with other options. Their choices led to $W \approx 135,000,000$ weights. The computations were on four NVIDIA GPU's, and training one net took 2-3 weeks. The reader may have less computing power (and smaller problems). So the network hyperparameters L and B will be reduced. We believe that the important principles remain the same.

A key point here is the recommendation to reduce the size E of the local convolutions. 5 by 5 and 7 by 7 filters were rejected. In fact a 1 by 1 convolutional layer can be a way to introduce an extra bank of ReLU's—as in the ResNets coming next.

The authors compare three convolution layers, each with 3 by 3 filters, to a single layer of less local 7 by 7 convolutions. They are comparing 27 weights with 49 weights, and three nonlinear layers with one. In both cases the influence of a single data point spreads to three neighbors vertically and horizontally in the image or the RGB images ($B = 3$). Preference goes to the 3 by 3 filters with extra nonlinearities from more neurons per layer.

Softmax Outputs for Multiclass Networks

In recognizing digits, we have 10 possible outputs. For letters and other symbols, 26 or more. With multiple output classes, we need an appropriate way to decide the very last layer (the output layer w in the neural net that started with v). “Softmax” replaces the two-output case of logistic regression. **We are turning n numbers into probabilities.**

The outputs w_1, \dots, w_n are converted to probabilities p_1, \dots, p_n that add to 1:

$$\text{Softmax} \quad p_j = \frac{1}{S} e^{w_j} \quad \text{where} \quad S = \sum_{k=1}^n e^{w_k} \quad (9)$$

Certainly softmax assigns the largest probability p_j to the largest output w_j . But e^w is a nonlinear function of w . So the softmax assignment is not invariant to scale: If we double all the outputs w_j , softmax will produce different probabilities p_j . For small w 's softmax actually deemphasizes the largest number w_{\max} .

In the CNN example of **teachyourmachine.com** to recognize digits, you will see how softmax produces the probabilities displayed in a pie chart—an excellent visual aid.

CNN We need a lot of weights to fit the data, and we are proud that we can compute them (with the help of gradient descent). But there is no justification for the number of weights to be uselessly large—if *weights can be reused*. For long signals in 1D and especially images in 2D, we may have no reason to change the weights from pixel to pixel.

1. cs231n.github.io/convolutional-networks/ (karpathy@cs.stanford.edu)
2. K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, ICLR (2015), arXiv: 1409.1556v6, 10 Apr 2015.
3. A. Krizhevsky, I. Sutskever, and G. Hinton, *ImageNet classification with deep convolutional neural networks*, NIPS (2012) 1106–1114.
4. Y. LeCun and Y. Bengio, *Convolutional networks for images, speech, and time-series*, *Handbook of Brain Theory and Neural Networks*, MIT Press (1998).

Support Vector Machine in the Last Layer

For CNN's in computer vision, the final layer often has a special form. If the previous layers used ReLU and max-pooling (both piecewise linear), the last step can become a difference-of-convex program, and eventually a multiclass Support Vector Machine (SVM). Then optimization of the weights in a piecewise linear CNN can be one layer at a time.

L. Berrada, A. Zisserman, and P. Kumar, *Trusting SVM for piecewise linear CNNs*, arXiv: 1611.02185, 6 Mar 2017.

The World Championship at the Game of Go

A dramatic achievement by a deep convolutional network was to defeat the (human) world champion at Go. This is a difficult game played on a 19 by 19 board. In turn, two players put down “stones” in attempting to surround those of the opponent. When a group of one color has no open space beside it (left, right, up, or down), those stones are removed from the board. Wikipedia has an animated game.

AlphaGo defeated the leading player Lee Sedol by 4 games to 1 in 2016. It had trained on thousands of human games. This was a convincing victory, but not overwhelming. Then the neural network was deepened and improved. Google's new version AlphaGo Zero learned to play without any human intervention—simply by playing against itself. Now it defeated its former self AlphaGo by 100 to 0.

The key point about the new and better version is that **the machine learned by itself**. It was told the rules and nothing more. The first version had been fed earlier games, aiming to discover why winners had won and losers had lost. The outcome from the new approach was parallel to the machine translation of languages. To master a language, special cases from grammar seemed essential. How else to learn all those exceptions? The translation team at Google was telling the system what it needed to know.

Meanwhile another small team was taking a different approach: Let the machine figure it out. In both cases, playing Go and translating languages, success came with a deeper neural net and more games and no coaching.

It is the depth and the architecture of AlphaGo Zero that interest us here. The hyperparameters will come in Section VII.4: the fateful decisions. The parallel history of Google Translate must wait until VII.5 because Recurrent Neural Networks (RNN's) are needed—to capture the sequential structure of text.

It is interesting that the machine often makes opening moves that have seldom or never been chosen by humans. The input to the network is a board position and its history. The output vector gives the probability of selecting each move—and also a scalar that estimates the probability of winning from that position. Every step communicates with a Monte Carlo tree search, to produce *reinforcement learning*.

Residual Networks (ResNets)

Networks are becoming seriously deeper with more and more hidden layers. Mostly these are convolutional layers with a moderate number of independent weights. But depth brings dangers. Information can jam up and never reach the output. The problem of “vanishing gradients” can be serious: so many multiplications in propagating so far, with the result that computed gradients are exponentially small. When it is well designed, depth is a good thing—**but you must create paths for learning to move forward.**

The remarkable thing is that those fast paths can be very simple: “*skip connections*” that go directly to the next layer—bypassing the usual step $v_n = (A_n v_{n-1} + b_n)_+$. An efficient proposal of Veit, Wilber, and Belongie is to allow either a *skip* or a normal convolution, with a ReLU step every time. If the net has L layers, there will be 2^L possible routes—fast or normal from each layer to the next.

One result is that entire layers can be removed without significant impact. The n th layer is reached by 2^{n-1} possible paths. Many paths have length well below n , not counting the skips.

It is hard to predict whether deep ConvNets will be replaced by ResNets.

K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, arXiv: 1512.03385, 10 Dec 2015. This paper works with extremely deep neural nets by adding shortcuts that skip layers, with weights $A = I$. Otherwise depth can degrade performance.

K. He, X. Zhang, S. Ren, and J. Sun, *Identity mappings in deep residual networks*, arXiv: 1603.05027, 25 Jul 2016.

A. Veit, M. Wilber, and S. Belongie, *Residual networks behave like ensembles of relatively shallow networks*, arXiv: 1605.06431, 27 Oct 2016.

A Simple CNN: Learning to Read Letters

One of the class projects at MIT was a convolutional net. The user begins by drawing multiple copies (not many) of A and B . On this training set, the correct classification is part of the input from the user. Then comes the mysterious step of *learning this data*—creating a continuous piecewise linear function $F(v)$ that gives high probability to the correct answer (the letter that was intended).

For learning to read digits, 10 probabilities appear in a pie chart. You quickly discover that too small a training set leads to frequent errors. If the examples had centered numbers or letters, and the test images are not centered, the user understands why those errors appear.

One purpose of **teachyourmachine.com** is education in machine learning at all levels (schools included). It is accessible to every reader.

These final references apply highly original ideas from signal processing to CNN’s:

R. Balestriero and R. Baraniuk, *Mad Max: Affine spline insights into deep learning*, arXiv: 1805.06576.

S. Mallat, *Understanding deep convolutional networks*, Phil. Trans. Roy. Soc. **374** (2016); arXiv: 1601.04920.

C.-C. J. Kuo, *The CNN as a guided multilayer RECOs transform*, IEEE Signal Proc. Mag. **34** (2017) 81-89; arXiv: 1701.08481.

Problem Set VII.2

- 1 Wikipedia proposes a 5×5 matrix (different from equation (3)) to approximate a Gaussian. Compare the two filters acting on a horizontal edge (all 1's above all 0's) and a diagonal edge (lower triangle of 1's, upper triangle of 0's).
- 2 What matrix—corresponding to the Sobel matrices in equation (4)—would you use to find gradients in the 45° diagonal direction?
- 3 (Recommended) For image recognition, remember that the input sample v is a matrix (say 3 by 3). Pad it with zeros on all sides to be 5 by 5. Now apply a convolution as in the text (before equation (2)) to produce a 3 by 3 output Av . What are the 1, 1 and 2, 2 entries of Av ?
- 4 Here are two matrix approximations L to the Laplacian $\partial^2 u / \partial x^2 + \partial^2 u / \partial y^2 = \nabla^2 u$:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}.$$

What are the responses LV and LD to a vertical or diagonal step edge?

$$V = \begin{bmatrix} 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \\ 2 & 2 & 2 & 6 & 6 & 6 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- 5 *Could a convolutional net learn calculus?* Start with the derivatives of fourth degree polynomials $p(x)$. The inputs could be graphs of $p = a_0 + a_1x + \dots + a_4x^4$ for $0 \leq x \leq 1$ and a training set of a 's. The correct outputs would be the coefficients $0, a_1, 2a_2, 3a_3, 4a_4$ from dp/dx . Using softmax with 5 classes, could you design and create a CNN to learn differential calculus?
- 6 Would it be easier or harder to learn integral calculus? With the same inputs, the six outputs would be $0, a_0, \frac{1}{2}a_1, \frac{1}{3}a_2, \frac{1}{4}a_3, \frac{1}{5}a_4$.
- 7 How difficult is addition of polynomials, with two graphs as inputs? The training set outputs would be the correct sums $a_0 + b_0, \dots, a_4 + b_4$ of the coefficients. Is multiplication of polynomials difficult with 9 outputs $a_0b_0, a_0b_1 + a_1b_0, \dots, a_4b_4$?

The inputs in 5-7 are pictures of the graphs. Cleve Moler reported on experiments:

<https://blogs.mathworks.com/cleve/2018/08/06/teaching-calculus-to-a-deep-learner>

Also .. [2018/10/22/teaching-a-newcomer-about-teaching-calculus-to-a-deep-learner](https://blogs.mathworks.com/cleve/2018/10/22/teaching-a-newcomer-about-teaching-calculus-to-a-deep-learner)

A theory of deep learning for hidden physics is emerging: for example see arXiv: 1808.04327.