Deep Convolutional Neural Nets

Carlo Tomasi

February 13, 2019

Neural nets are a class of predictors that have recently been shown empirically to achieve very good performance on tasks whose inputs are images, speech, or audio signals. They have also been applied to inputs of other types, with varied results. The reasons why these predictors work so well are still unclear. What *is* clear is that they are very expressive, in the sense that the hypothesis space that they define is very large. Theorems show that *any* Lipschitz function[1] from a hypercube in $\mathbb{R}^d$ to an interval in $\mathbb{R}$ can be approximated arbitrarily closely (that is, within any pre-specified $\epsilon > 0$) with a neural net.

While expressive power is good, we know that it hides dangers. First, the approximation theorems just mentioned are practically irrelevant, because the computational complexity of a neural net needed to achieve a given accuracy ϵ turns out to grow exponentially with the dimension d of the input space. Thus, neural nets can approximate anything, but at an unrealistically high cost. Second, the fact that an approximator *exists* in a given hypothesis class does not mean that we know how to find it: Training a predictor amounts to minimizing the empirical risk L_T over the training set T, and if L_T is not a convex function, finding a global minimum is generally computationally intractable. Third, and perhaps most importantly for machine learning, a high expressive power leads to overfitting.

The last consideration implies that training a neural net requires large training sets. Indeed, perhaps the greatest hurdle to the widespread use of neural nets is the cost of collecting, and most importantly annotating, data sets with millions of samples. Arguably, the most important reason for the success of neural nets is not so much the nets themselves, but the emergence of the Amazon Mechanical Turk, a crowdsourcing marketplace made available by Amazon. With the Turk, you can post millions of data samples on a web site, and let any user anywhere annotate the data for you, for a reward of a few cents per sample. There would be no successful deep neural nets without the Turk.

At the same time, even the large sizes of current training nets cannot fully explain their ability to generalize: The inputs to neural nets often have dimensionality d in the tens or hundreds of thousands, and no amount of data under the sun can keep up with the exponential growth of X with d. There must be deeper reasons at play, that have to do with (i) the special structure of image space (or audio space, and so forth); (ii) the specialized architectures proposed for neural nets; and (iii) tricks and techniques used to regularize training.

In summary, neural nets are very expressive and data hungry. In spite of their expressiveness, they often generalize better than one would predict. We don't fully know why, although theoreticians are making daily progress towards an answer. Because of the still immature degree of

---

[1] Somewhat loosely speaking, a differentiable function is Lipschitz when its gradient is uniformly bounded by a constant. This notion can be defined more generally without reference to differentiability.

theoretical understanding of neural nets, the treatment in these notes will have to be based on half-baked intuitions and empirical evidence.

This note covers one particular way to build a particular type of deep feed-forward network. Such a network can be used for either classification or regression, and we will focus on classification. More variants and details can be found in many books or articles on neural nets [1], convolutional neural nets [6], and deep learning in general [2, 3].

A later note on training will describe how to determine the parameters (weights) of a deep feed-forward network of a given structure and for a given classification task.

1 Circuits

Suppose that you want to implement a predictor $h : X \to Y$ on a computer. There are various ways to describe the implementation of h. The one we are most familiar with is in terms of an *algorithm*, a sequence of steps to be performed in sequence over time. Another way is to specify a *circuit*, a computational model that mimics how electrical circuits are built. A (computational) circuit is made of a possibly large number of *gates*, each of which implements one of a small set of predefined functions. For example, logical circuits are made mostly out of NAND (not-and) gates, which when combined can produce any Boolean function.

Circuits and algorithms are equivalent to each other: You can build a Boolean function by buying NAND gates at Radio Shack and wiring them together, or you can write a piece of Python code that simulates the circuit. Since the simulation simulates one gate at a time, it may take a long time to simulate a complex circuit. You can also go the other way: Given an algorithm, come up with a circuit that implements the same (Boolean) function. This must be possible: After all, a computer is a large circuit that runs algorithms. You may object that computers compute more than just Boolean functions, including, say, real-valued functions, but they really do not: A number is represented by a finite string of bits in a computer, so the output is still a set of Boolean variables, which are functions of the Boolean variables that represent the inputs.

A neural network is a class of algorithms that are typically described as circuits, and are made by *neurons*. A set of neurons is said to form a *layer* if each neuron in the set receives the same inputs. A *neural network* is a cascade of layers, in which the outputs from one layer are the inputs to the next. The network is *deep* if it has many layers. Every neuron has *parameters*, so a neural network has many parameters. The network is *convolutional* if the parameters of the neurons in each layer are constrained in a special way. The Sections that follow define these concepts. *Training* a neural network amounts to finding the parameters that minimize the training risk. Training is discussed in a later note.

2 Neurons

A *neuron* (in the computational sense) is a function $\mathbb{R}^d \to \mathbb{R}$ of the form

$$ y = \rho(a(\mathbf{x})) \quad \text{where} \quad a = \mathbf{w}^T \tilde{\mathbf{x}} \quad , \quad \tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} . $$

The entries of the vector $\mathbf{w} \in \mathbb{R}^{d+1}$ are called the *weights*, and the *activation function* is a nonlinear and weakly monotonic function $\mathbb{R} \to \mathbb{R}$. The input $a(\mathbf{x})$ to ρ is called the *activation* of the neuron,

and the particular type of activation function

$$\rho(a) = \max(0, a)$$

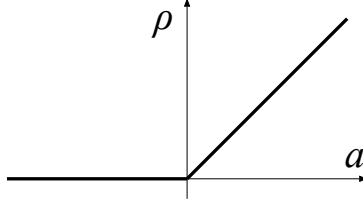is called the *Rectified Linear Unit* (ReLU, Figure 1).

Figure 1: The Rectified Linear Unit (ReLU).

We view the tilde (as in $\tilde{\mathbf{x}}$) as an operator: Given any vector $\mathbf{x}$, this operator appends a 1 at the end of $\mathbf{x}$.

The activation can be rewritten as follows

$$a = \mathbf{v}^T\mathbf{x} + b \quad \text{where} \quad \mathbf{v}^T = [w_1, \ldots, w_d] \quad \text{and} \quad b = w_{d+1} \,,$$

and is an inner product between a *gain*[2] vector $\mathbf{v}$ and the input $\mathbf{x}$ plus a *bias* b. Figure 2 shows a neuron in diagrammatic form.

For different inputs $\mathbf{x}$ of the same magnitude[3], the activation is maximum when $\mathbf{x}$ is parallel to $\mathbf{v}$, and the latter can be viewed as a *pattern* or *template* to which $\mathbf{x}$ is compared. The bias b then raises or lowers the activation before it is passed through the activation function.

The ReLU will *respond* (that is, return a nonzero output) if the inner product $\mathbf{v}^T\mathbf{x}$ is greater than $-b$ (so that a is positive), and the response thereafter increases with the value of a. So the negative of the bias can be viewed as a *threshold* that the inner product between pattern and input must exceed before it is deemed to be significant, and the neuron can be viewed as a *score function* that measures the similarity of the suitably normalized input $\mathbf{x}$ to the pattern $\mathbf{v}$ when the similarity is significant (that is, greater than $-b$). When the similarity is not significant, the neuron does not respond.

A *pattern classifier* would add a stage that decides if the score is large enough to declare the input $\mathbf{x}$ to contain the pattern represented by $\mathbf{v}$. So another way to view a neuron is a pattern classifier without the decision stage.

3 Two-Layer Neural Nets

A neural-net *layer* is a vector of $d^{(1)}$ neurons, that is, a function $\mathbb{R}^d \to \mathbb{R}^{d^{(1)}}$

$$\mathbf{y} = \rho(\mathbf{a}(\mathbf{x})) \quad \text{where} \quad \mathbf{a}(\mathbf{x}) = W\tilde{\mathbf{x}} \,,$$

the weight matrix W is $d^{(1)} \times (d+1)$, and the activation function ρ is applied to each entry of the activation vector $\mathbf{a}(\mathbf{x}) \in \mathbb{R}^{d^{(1)}}$. So a neural-net layer can be viewed as a *bank* of pattern scoring devices, one pattern per neuron. Figure 3 illustrates.

---

[2]Gains are often called weights as well.
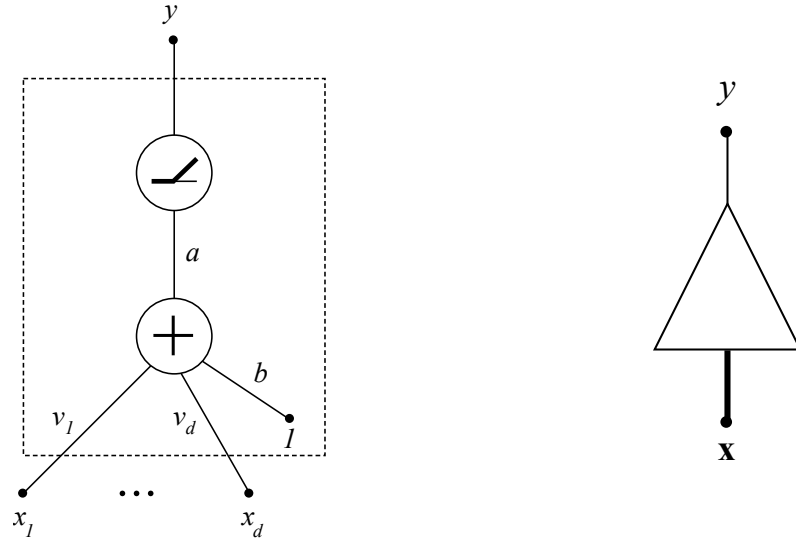[3]As measured by its Euclidean norm $\|\mathbf{x}\|$.

Figure 2: The internal structure of a neuron (left) and a neuron as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.
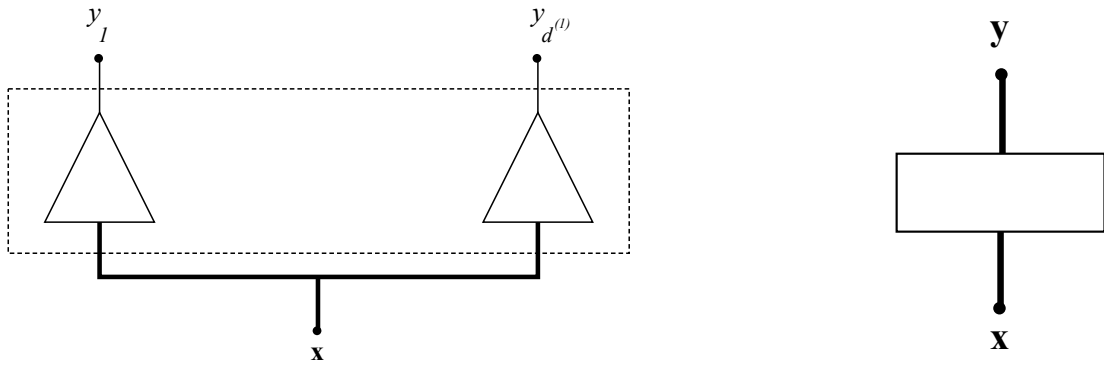

Figure 3: The internal structure of a layer (left) and a layer as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.

To compute the output of a layer from its input one needs to perform $d^{(1)}(d+1)$ multiplications and almost as many additions to compute the activation vector, and then compute the activation function $d^{(1)}$ times. So if $d^{(1)}$ is of the same order of magnitude as d, the cost of this computation is quadratic in the size d of the input $\mathbf{x}$. Even more importantly, there are $O(d^2)$ parameters (the entries of W) that need to be determined when the layer is trained.

A *two-layer neural net* is a cascade of two layers,

$$
\begin{aligned}
\mathbf{x}^{(1)} &= \rho(W^{(1)}\tilde{\mathbf{x}}) \\
\mathbf{y} &= \rho_y(W^{(2)}\tilde{\mathbf{x}}^{(1)})
\end{aligned}
$$

where the activation function ρ_y is in general different from ρ and $W^{(2)}$ is $d^{(2)} \times (d^{(1)} + 1)$.

It can be proven [7] that any mapping $\mathbb{R}^d \to \mathbb{R}^{d^{(2)}}$ that is Lebesgue-integrable and has Lebesgue-integrable Fourier transform[4] can be approximated to any finite degree of accuracy over a hypercube in $\mathbb{R}^d$ with a two-layer neural net where ρ_y is the identity function and ρ is the ReLU. This result, along with similar ones for other activation functions [4], shows that two-layer neural nets are *universal approximators*.

However, a two-layer approximator to a given function f may be very expensive to implement. For instance, many triangle functions may be needed in the construction above, especially for functions whose domain is multidimensional (large d). Deep neural nets are introduced in the hope that they lead to more efficient approximations, as discussed in the next two sections.

4   Convolutional Layers

A neuron matches the input $\mathbf{x}$ to a pattern $\mathbf{v}$. What should the patterns in an image recognition system be? One could make $\mathbf{x}$ be the entire image, with its pixels strung into a vector, and then $\mathbf{v}$ could be an image (in vector form as well) of the object to be recognized—say, your grandmother's face. This net would not work well, as your grandmother's face could show up in images that look very different from $\mathbf{v}$ because of viewpoint, lighting, facial expression, other objects or people in the image, and other causes of discrepancy.

Instead, observe that faces typically have eyes, noses, ears, hair, and wrinkles—especially for an older person. These features can be analyzed in turn in terms of image edges, corners, curved segments, small dark regions, and so forth. This suggest building a *hierarchy* of patterns, where higher-level ones are made of lower level ones, and only the lowest-level patterns are made directly out of pixels from the input image. At each level, each pattern should then take only a relatively small number of inputs in consideration, from a relatively small and compact part of the image: each neuron should have a *local receptive field*.

In addition, many of the lower-level features appear multiple times in images and across objects, and this suggests that the same neuron could score patterns of its own type no matter where they appear in an image: the same detector could be *reused* over its domain.

Finally, if higher-level patterns are somehow made somewhat insensitive to exactly where in the image the relevant lower-level patterns occur, then the overall system would be able to recognize your grandmother's face even in the presence of at least some amount of variation. A hierarchy with many levels may be able to achieve this even more easily, since a small amount of *resilience*

---

[4]Just think of these as mild requirements on the mapping. It is not important for our purposes to know what they mean.

to spatial variation in each layer might result in more significant resilience once it is compounded across layers.

These notions of locality, reuse, and resilience to spatial variations suggest the following structure for a neural-net layer [6, 5].

- Think of the input $\mathbf{x}$ as a two-dimensional array, one entry per pixel, rather than a vector, so that the notion of locality is more readily expressed.

- Group the activations a_i (the entries of $\mathbf{a}$) into m *maps*: each map takes care of one type of pattern through a separate correlation kernel with a small *receptive field* $k_1 \times k_2$. A pattern kernel with a small receptive field is also called a *feature*, so the m maps are called *feature maps*. The (common) activation function ρ is then applied to each feature map, entry by entry.

- Reduce the size of each feature map by *max-pooling*. Specifically, square receptive fields are defined in each feature map in turn. A new, smaller feature map is then computed whose values are each the maximum value in its receptive field.

  In addition to reducing the size of the feature maps in the output from the layer, max-pooling makes the output of the layer somewhat less sensitive to the exact location of the features in the image. For instance, with a 3×3 receptive field for max-pooling and a stride of 3 (no overlap between pools), the output of the maximum is oblivious to which of those 3^2 activations produced the final output from the layer. In other words, max-pooling achieves some degree of *translation-invariance*. If the same is done in every layer of a deep network, the amounts of invariance add up.

Stride: In a standard convolution or max-pooling operation, the kernel or receptive field is slid over the entire input. However, if the kernel or receptive field is not too small, any given output entry is not very different from neighboring output entries, because images often vary slowly as a function of image coordinates. To reduce the redundancy in the output, the values of convolutions or other local operators are often computed with a *stride s* greater than one. That is, only one value every s is computed in each dimension, thereby reducing the size of the output by a factor of about s in every dimension.

The *convolutional organization* described above for a layer is illustrated in figure 4. As a result of this structure, the number of distinct parameters in W drops dramatically. If the layer were *fully connected*, that is, if every entry in W were nonzero and had its own separate value, then there would be about d^4 parameters if the input image were a $d \times d$ array (that is, a square, black-and-white image).

With m feature maps, the number of parameters is about $m(k^2+1)$ if all the kernels are $k \times k$, and if biases are counted as well. For color images, the count drops from about $9n^4$ to about $3m(k^2+1)$, if each color band gets a separate set of feature maps.

For instance [5], for a $224 \times 224 \times 3$-pixel color image and 96 maps each with an 11×11 receptive field, the drop is from about 22.7 billion to a mere $96 \times 11^2 = 11{,}616$ parameters. More specifically, in that example [5], illustrated in Figure 4, the first layer of a convolutional net has a 224×224 color image as input, so that the input dimensionality is $d = 224^2 \times 3 = 150{,}528$. There are 96 kernels, feature maps, and response maps, 32 per color. The convolution kernels have receptive fields of size 11×11 pixels, so each output pixel in each color channel is computed from $11^2 = 121$ input
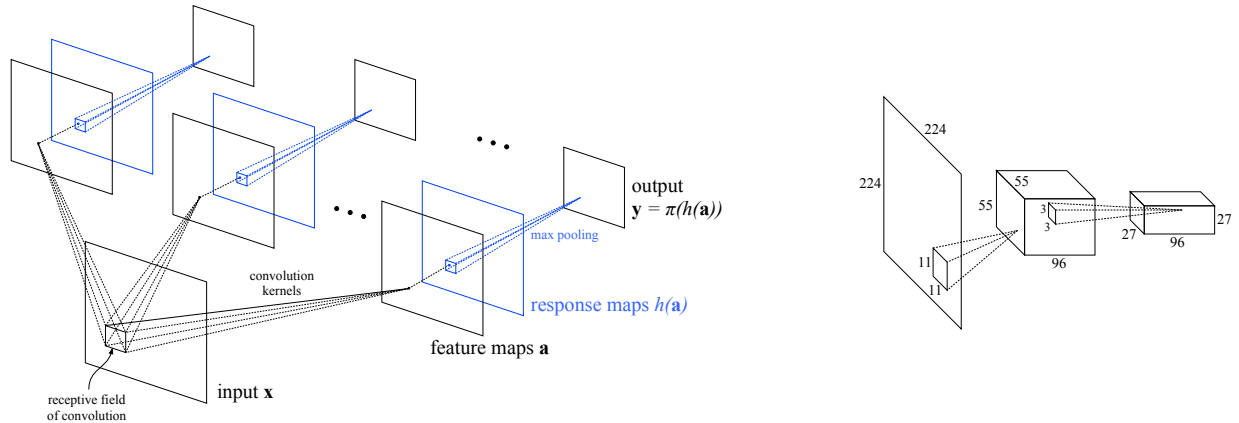
Figure 4: (Left) The structure of a convolutional neural-net layer. (Right) In the literature, neural nets with many layers are drawn with each layer shown in a more compact way than on the left, although there is no standard format. Typically, the maps are stacked in a block, as shown here, rather than drawn side-to-side. Sometimes, max-pooling is only mentioned and not shown explicitly.

pixels which are combined through 122 weights, including a bias value. The stride for computing activations is 4 pixels in each direction, so the activation maps and feature maps are 55×55 pixels each. Max-pooling uses 3×3-pixel receptive fields and a stride of 2 pixels, and produces output maps of size 27×27 pixels.

The set of activation maps computed by the 96 kernels is a $55 \times 55 \times 96$ block, rather than a single image, and max pooling is applied to each of the 96 slices in this block. If a subsequent layer applies convolution to the resulting $27 \times 27 \times 96$ output block from this layer, that convolution kernel is in general three-dimensional, $m \times n \times p$, although p can be equal to 1 for an effectively two-dimensional kernel.

For the layer in the Figure, the output dimensionality is $d^{(1)} = 27^2 \times 96 = 69{,}984$, a bit less than half of the input dimensionality $d = 224^2 \times 3 = 150{,}528$. On the other hand, the map *resolution* decreases more than eightfold, from 224 to 27 pixels on each side. The representation of the image has become more abstract, changing from a pure pixel-by-pixel list of its colors to a coarser map of how much each of 32 features is present at each location in the image and in each color channel.

5 Deep Convolutional Neural Nets

The architecture of a neural-net layer embodies the principles of feature reuse, locality, and translation-invariance. *Deep* Convolutional Neural Nets (CNNs) are CNNs with many layers, and reflect the principle of hierarchy. After several convolutional layers, deep CNNs typically add one or a few fully-connected layers, that is, layers where the weight matrix W is dense. The reasons for doing so are somewhat mixed and not entirely compelling, but are nonetheless plausible: Far away from the input, spatial location is both partially lost and relatively irrelevant to, say, recognition, so the local receptive fields of CNNs are no longer useful. In addition, signals in late stages of a deep net have relatively low dimensionality, and one can then better afford the greater representational flexibility that a fully-connected layer carries.

The output from a deep CNN is fed to a computation that depends on the purpose of the net.

For regression, for instance, the outputs may be used as they are. For classification, one could use the outputs as inputs to a support vector machine or random forest. More commonly, the output stage is a *softmax* function,

$$\mathbf{z} = \sigma(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\mathbf{1}^T \exp(\mathbf{y})}$$

where $\mathbf{1}$ is a column vector of ones. As we saw in an erlier note, the exponential makes all quantities positive, and normalization makes sure the entries of $\mathbf{z}$ add up to 1. In this way, the entries of the softmax output can be viewed as *scores* for each of the categories, and the result of classification is then class

$$h(\mathbf{x}) = \arg\max_i z_i \ .$$

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] L. Deng and D. Yu. *Deep Learning: Methods and Applications*, volume 7(3-4) of *Foundations and Trends in Signal Processing*. Now Publishers, 2014.

[3] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, Cambridge, MA, 2016.

[4] V. Y. Kreinovic. Arbitrary nonlinearity is sufficient to represent all functions by neural networks: a theorem. *Neural Networks*, 4:381–383, 1991.

[5] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1106–1114, 2012.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[7] S. Sonoda and N. Murata. Neural network with unbounded activations is universal approximator. Technical Report 1505.3654 [cs.NE], arXiv, 2015.