Symmetric neural networks

YKY

May 6, 2019

1 Convolutions have translational equivariance

The following is quoted from Wikipedia:

> The convolution commutes with translations, meaning that
>
> $$\tau_x(f * g) = (\tau_x f) * g = f * (\tau_x g) \tag{1}$$
>
> where $\tau_x f$ is the translation of the function f by x defined by:
>
> $$(\tau_x f)(y) = f(y - x). \tag{2}$$
>
> If f is a **Schwartz function** [a] , then $\tau_x f$ is the convolution with a translated Dirac delta function $\tau_x f = f * \tau_x \delta$. So translation invariance of the convolution of Schwartz functions is a consequence of the associativity of convolution.
> Furthermore, under certain conditions, convolution is the most general translation invariant operation. Informally speaking, the following holds:
>
> > Suppose that S is a bounded linear operator acting on functions which commutes with translations: $S(\tau_x f) = \tau_x(Sf)$ for all x. Then S is given as convolution with a function (or distribution) gS; that is $Sf = gS * f$.
>
> Thus some translation invariant operations can be represented as convolution. Convolutions play an important role in the study of time-invariant systems, and especially LTI system theory. The representing function gS is the impulse response of the transformation S.
> A more precise version of the theorem quoted above requires specifying the class of functions on which the convolution is defined, and also requires assuming in addition that S must be a continuous linear operator with respect to the appropriate topology. It is known, for instance, that every continuous translation invariant continuous linear operator on $L1$ is the convolution with a finite Borel measure. More generally, every continuous translation invariant continuous linear operator on L_p for $1 \leq p < \infty$ is the convolution with a tempered distribution whose Fourier transform is bounded. To wit, they are all given by bounded Fourier multipliers.
>
> [a] Schwartz space is the function space of all functions whose derivatives are rapidly decreasing. This space has the important property that the Fourier transform is an automorphism on this space.

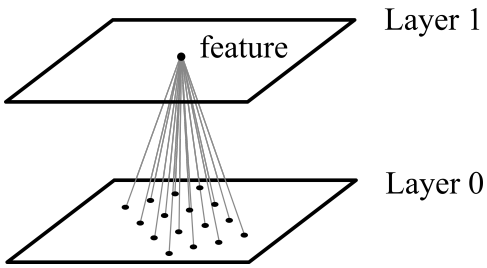
In 1989, Yann LeCun invented **convolutional** neural networks (ConvNets) which established the importance of this type of neural networks for **computer vision** (it seems to remain the dominant approach to this day).

For this he won the Turing Award recently.

In computer vision, translational invariance is obviously desirable. The structure of ConvNets has translational invariance built-in, so it does not need to be learned, thus making learning more efficient. This enabled the breakthrough in machine vision to surpass human level, around 2011.

2 Structure of a standard neural network

Recall that a neural network is constructed from **features**:



(3)

A **feature** = **neuron**, is a **filter** applied to a *local* part of the **input signal**, usually implemented as a **dot product** followed by a **non-linearity** $\int$:

$$\boxed{\text{neuron}} = \boxed{\text{feature}} = \int \langle \boldsymbol{v}, \boldsymbol{w} \rangle. \tag{4}$$

Such a function is also called a **ridge function**. $\boldsymbol{v}$ is the **input** vector, $\boldsymbol{w}$ is the **weight** vector.

Last time I forgot to mention that the non-linearity $\int$ being a **sigmoid** function is not essential to a neural network. In fact, any non-linearity will do, as long as it is *continuous* and has first-order derivatives. That even includes **piecewise-linear** functions.

The "universal" approximation ability of neural networks comes from the **Weierstrass approximation theorem** (1885) that says that every continuous function can be uniformly approximated by polynomials. This is later generalized to the **Stone-Weierstrass** theorem (1937). For neural networks, the universal approximation property can be proven with just a single layer — which means it does not depend on the "deep" property.

If we have a **pool** of features or neurons, they form a **layer**:

$$\boxed{\text{pool of neurons}} = \boxed{\text{layer}} = \int(W\boldsymbol{v}) \tag{5}$$

where W is a **matrix** of weights. $\int$ is applied *component-wise* to the resulting vector.

A deep network is just the **composition** of many layers:

$$\boxed{\text{deep network}} = [\int W]^L \ \boldsymbol{v}. \tag{6}$$

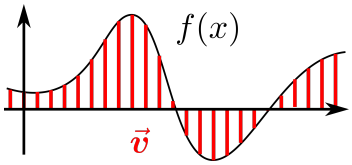
3 Structure of a ConvNet

In the convolutional network, each **feature** is replaced by a convolutional feature:

$$\boxed{\text{convolutional feature}} = \mathcal{O}(\ f * g\ ) \tag{7}$$

where f, g are functions, f is a **filter** applied to the **input** g.

We can regard the input signal as a **function** or as a discretized **vector** of its graph:



(8)

so there is no big difference between a function and a vector, in this context.

When implemented on a computer, the discrete, **full convolution** is defined as *:

$$\boxed{\text{full convolution}} \quad \boldsymbol{x} * \boldsymbol{k} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \boldsymbol{x}_{ij}\, \boldsymbol{k}_{u-i,v-j}. \tag{9}$$

A **valid** convolution is a restriction of the full convolution to a **window**:

$$\boxed{\text{valid convolution}} \quad \boldsymbol{x} * \boldsymbol{k} = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \boldsymbol{x}_{i+u,j+v}\, \boldsymbol{k}_{\text{rot}\ i,j}\ \chi(i,j) \tag{10}$$

$$\chi(i,j) = \begin{cases} 1, & 0 \le i,j \le n \\ 0, & \text{otherwise} \end{cases}$$

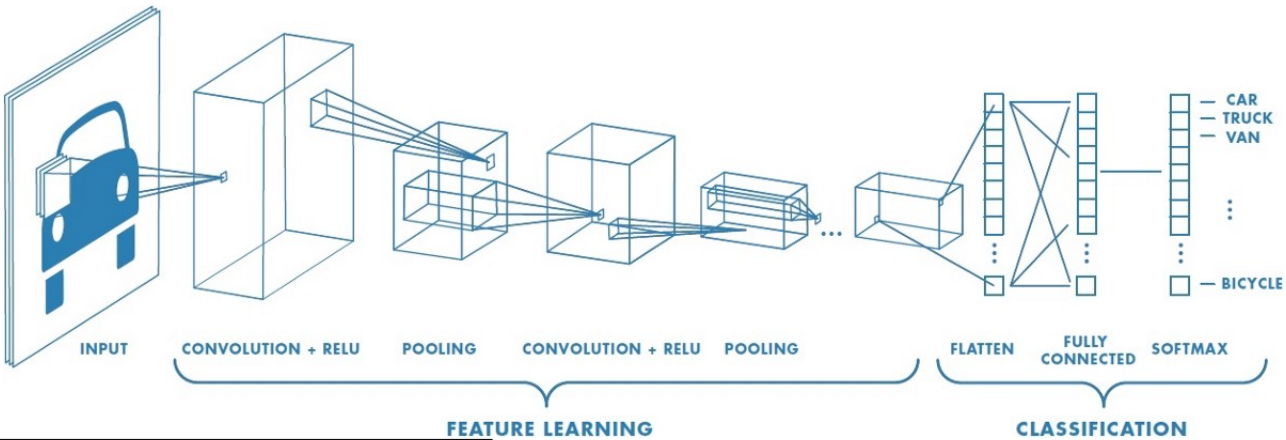
where $\boldsymbol{k}_{\text{rot}}$ is $\boldsymbol{k}$ rotated by 180°.

The pool of local features has a lot of **redundancy**, which can be reduced by **sub-sampling** them:

$$\boxed{\text{sub-sampling}} \quad \boldsymbol{x}_{i,j}^{(\ell+1)} = \mathcal{O}(\ \beta \sum_{u=ir}^{(i+1)r-1} \sum_{v=jr}^{(j+1)r-1} \boldsymbol{x}_{u,v}^{(\ell)}\ ) \tag{11}$$

where β is a scalar weight. The output is a matrix of *reduced* size: $\left(\dfrac{m-n-1}{r} \times \dfrac{m-n+1}{r}\right)$.

Repeating this structure gives rise to the following typical architecture (which I won't explain in detail):



(12)

*Thanks to at (Zhihu.com) for providing these formulas.

4 Symmetric neural networks

For **vision**, we need invariance under translations as well as rotations, scaling, etc. In short, **affine transformations**. ConvNet merely provides translational invariance, but the boost in learning speed is sufficient to kick start a revolution in computer vision.

Now we turn to the topic of symbolic **logic**, my research focus. Suppose the input is a list of vectors $(\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_K)$. We want neural networks that are invariant under **permutations** of the vectors $\boldsymbol{v}_i$; in other words, invariant under the action of the symmetric group $\mathfrak{S}_K$.

Such a requirement would impose some constraints on the weights of the neural network. For a long time, I thought it would be impossible to realize such constraints in a neural network, but I wasn't aware that the other researchers have investigated the problem and gotten significant results. For example Domingos and Gens's 2014 idea [†].

Note that Pedro Domingos also has a research interest in logic-based learning (with neural networks), and he is the author of the book *The Master Algorithm: How the Quest for the Ultimate Learning Machine Will Remake Our World*.

Their idea is simply to replace the convolution operator in (7) with other **filters** that have desired invariance properties (*eg* affine invariance). The resulting neural network is training with back-propagation as usual.

So perhaps we can use the following kind of features for the logical purpose:

$$\boxed{\text{commutative feature}} = P_{\text{sym}}(\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n) \tag{13}$$

where P_{sym} is a **symmetric polynomial** (which is already non-linear so we don't need the $\int$ function).

So we have the following correspondence:

levels:	**neurons** = features	**pool** = layer	**network**
neural networks	dot product $\int \langle \boldsymbol{v}, \boldsymbol{w} \rangle$	matrix	composition of layers
ConvNets	convolution $\int(f * g)$	pool	composition of layers
SymNets	symmetric polynomials $P_{\text{sym}}(\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_n)$	pool	composition of layers

(14)

[†]Robert Gens and Pedro Domingos 2014, *Deep symmetry networks*, NIPS'14, proceedings of the 27th international conference on Neural Information Processing Systems, v2, 2537-2545.

Unfortunately, if P_{1234} is symmetric over $(\boldsymbol{v}_1, ..., \boldsymbol{v}_4)$, and P_{5678} symmetric over $(\boldsymbol{v}_5, ..., \boldsymbol{v}_8)$, then $P_{1234} + P_{5678}$ may not be invariant if we swap $\boldsymbol{v}_1$ with $\boldsymbol{v}_5$, say.

First consider the simple case where the vectors $\boldsymbol{v}_i$ are 1-dimensional.