Expressiviy of Overlapping Weights in CNNs Notes

Piyush Patil

September 22, 2017

1 Introduction and Background

The central idea behind this paper is to show that in convolutional layers, making use of overlapping weights between neighboring neurons is an exponential force multiplier in the expressivity of the network. Loosely speaking, expressivity is a term referring to some quantitative metric for the ability and ease with which a neural net is able to approximate a family of functions, and the richness, diversity, and generality of the largest such family.

Before moving on to considering overlapping fields and their effect on expressivity, we need t ofirst formally define expressivity. Below, a *neural network* is a function (usually between matrices and vectors) which can be expressed as the composition of several *layers*, each of which is a differentiable, parameterized function (usually between tensors). The *architecture* of a neural net refers a set of characterizing properties about the network, such as (1) the class of functions from which its layers are drawn from, (2) the length of the composition sequence, the dimensin

> **(Definition) Expressive P-Efficiency**: *Let $\mathcal{H}_1, \mathcal{H}_2$ be sets of neural nets, representing two different network architectures, where $\mathcal{H}_1$ has property P but $\mathcal{H}_2$ doesn't. Relative to some norm $|\cdot|$ defined on $\mathcal{H}_1, \mathcal{H}_2$, we say that $\mathcal{H}_1$ is **expressively P-efficient** with respect to $\mathcal{H}_2$ if the following conditions hold:*
>
> 1. $\forall f \in \mathcal{H}_2 : \exists f^* \in \mathcal{H}_1$ s.t. $|f^*| \in O(|f|)$ *and f^* approximates f to arbitrary precision.*
> 2. $\exists f \in \mathcal{H}_1$ s.t. *if $f^* \in \mathcal{H}_2$ arbitrarily approximates f then $|f^*| \in \Omega(\sigma(|f|))$ for superlinear σ. If this is true for every $f \in \mathcal{H}_1$, we say $\mathcal{H}_1$ is **completely more P-efficient** than $\mathcal{H}_2$.*

By *arbitrary approximation* we mean for any $\epsilon \in \mathbb{R}^+$ we can choose a function which differs in supremum norm from the target by less than ϵ. Essentially, the above definition simply states that if one architecture has a property P and another doesn't, we can use functions from the former to arbitrarily approximate functions from the latter, using only approximation functions that are linear in the size of the target; to go the other direction, and sacrifice the property P and try to approximate functions from the former space using functions from the latter, we can only succeed if the size of the functions needed grows superlinearly in the size of the target. Hence, any function that $\mathcal{H}_2$ can realize can also be realized with $\mathcal{H}_2$, using only a linear multiple of the number of parameters, and further the converse does not hold. Thus, it appears that having property P is crucial to the success of $\mathcal{H}_1$, which is able to easily realize functions that require superlinearly many parameters for $\mathcal{H}_2$ to realize.

The paper focuses on convolutional neural architectures, which are characterized by layers consisting of a set of n small (relative to the input dimensionality), square *kernel matrices*, whose behavior is to (1) convolve the input with each of its kernels, (2) stack the result in an order n tensor, (3) pass the tensor through a point-wise activation function, and (4) pass the activation through a downsampling function, which, similar to a convolution, slides over square patches of the input and combines the elements of the patch into a single element to keep dimensionality from getting out of hand as we continuously stack tensors. The *stride* of the convolution operator is a simple generalization of the convolution operator, and controls how much of the input is "skipped" as the kernel slides over (so the conventional discrete convolution has unit stride). In particular, if the stride is smaller than the kernel matrix dimensionality, adjacent entries in the output matrix will depend on some (but not all) of the same input entries; hence, the output entries depend on overlapping input entries. For this reason, convolutional network architectures whose stride is smaller than the kernel dimension are said to be of *overlapping type*. The objective of the paper is to prove that architectures of the overlapping type are expressively more efficient than those of the non-overlapping type. To do so, they prove a lower bound on the complete expressive capacity of overlapping architectures.

2 Convolutional Arithmetic Circuits

The architecture in use, over which overlapping and non-overlapping variants are considered, is the *convolutional arithmetic circuit* (ConvAC) architecture. In practice, convolutional layers consisting of rectified linear units (ReLU) as activation functions and a simple maximum for the downsampling function (relative to some patch size over which the maximum is

computed), known as max-pooling. ConvACs expect an input of the form $X = (x_1, \cdots, x_N)$ where each $x_i \in \mathbb{R}^s$; the idea here is that the input image X is viewed as a "flattened" vector of its patches with patch size p, so that each x_i corresponds to a "flattened" patch in s-dimensional space, so that $s = p^2 \cdot c$ where c is the number of channels (e.g. 3 for RGB images).

The first layer of the network is called the *representation layer*, and consists of M feature maps, $f_{\theta_d} : \mathbb{R}^s \to \mathbb{R}, 1 \leq d \leq M$. Every patch x_i is put through every feature map, which we conceptualize as "collapsing" an order-3 tensor (i.e. it takes in a 2-D patch of the image, extended along the channel dimension) into a value. The output of the entire layer, then, is also an order-3 tensor, but with depth M instead of c, with the same dimensions as the input image but scaled down by a factor of p. Notice that in the special case $f_{\theta_d}(x) = \sigma(w_d^\intercal x + b_d)$ the representation layer reduces t oa standard convolutional layer with parameters w_d, b_d.

The output of the representation layer is then put through L hidden layers, where layer l its input through a 1×1 *conv* operator, i.e. a standard convolutional layer with r_l channels and a unit sized receptive field, mapping an input with r_{l-1} channels to an output with r_l channels. However, the conv layer modifies the standard convolutional layer by discarding weight sharing, giving each patch induced by the receptive field (in this case, corresponding to single entries in the input) its own corresponding weight. This is simply an element-wise product.

> *Aside*: Convolutional layers with zero padding and 1×1 sized filters are commonly used for dimensionality reduction, as they allow us to transform a tensor with shape (M, H, W) to one with shape (N_F, H, W), where N_F 1×1 filters (i.e. kernel matrices) are used. Indeed, in practice a problem that crops up with CNNs is that even modest kernel sizes, such as 5×5, can explode the dimensionality so high as to make computing layers with large numbers of filters infeasible. Thus, putting inputs through a 1×1 convolution before using a larger sized kernel in a subsequent convolutional layer allows us to first reduce the dimensionality to a controlled, manageable size before passing through the expensive layer with higher kernel sizes and many filters. Seeing as 1×1 convolutions correspond to simple matrix multiplication over the entire image, they allow a simple way to add depth to the network without overly modifying its topology.

In our case, the representation layer outputs a tensor with depth M, and passing through 1×1 conv layers reduces this depth. Each conv layer is then put through a spatial pooling operator, which is a form of product pooling, i.e. so that each (non-overlapping) window put through the pooling function is mapped to the product of the entries. The last layer uses *global pooling*, where the window size is the entire spatial dimension and hence the output if a vector in $\mathbb{R}^{r_L}$.