POLYCNN: LEARNING SEED CONVOLUTIONAL FILTERS

Anonymous authors
Paper under double-blind review

ABSTRACT

In this work, we propose the polynomial convolutional neural network (PolyCNN), as a new design of a weight-learning efficient variant of the traditional CNN. The biggest advantage of the PolyCNN is that at each convolutional layer, only one convolutional filter is needed for learning the weights, which we call the seed filter, and all the other convolutional filters are the polynomial transformations of the seed filter, which is termed as an early fan-out. Alternatively, we can also perform late fan-out on the seed filter response to create the number of response maps needed to be input into the next layer. Both early and late fan-out allow the PolyCNN to learn only one convolutional filter at each layer, which can dramatically reduce the model complexity by saving $10\times$ to $50\times$ parameters during learning. While being efficient during both training and testing, the PolyCNN does not suffer performance due to the non-linear polynomial expansion which translates to richer representational power within the convolutional layers. By allowing direct control over model complexity, PolyCNN provides a flexible trade-off between performance and efficiency. We have verified the on-par performance between the proposed PolyCNN and the standard CNN on several visual datasets, such as MNIST, CIFAR-10, SVHN, and ImageNet.

1 INTRODUCTION

Applications of deep convolutional neural networks (CNNs) have been overwhelmingly successful in all aspect of perception tasks, ranging from computer vision to speech recognition and understanding, from biomedical data analysis to quantum physics. In the past couple of years, we have seen the evolution of many successful CNN architectures such as AlexNet (Krizhevsky et al., 2012), VGG (Simonyan & Zisserman, 2015), Inception (Szegedy et al., 2015), and ResNet (He et al., 2016b;a). However, training these networks end-to-end with fully learnable convolutional filters (as is standard practice) is still very computationally expensive and is prone to over-fitting due to the large number of parameters. To alleviate this issue, we have come to think about this question: can we arrive at a more efficient CNN in terms of learnable parameters, without sacrificing the high CNN performance?

In this paper, we present an alternative approach to reducing the computational complexity of CNNs while performing as well as standard CNNs. We introduce the polynomial convolutional neural networks (PolyCNN). The core idea behind the PolyCNN is that at each convolutional layer, only one convolutional filter is needed for learning the weights, which we call the seed filter, and all the other convolutional filters are the polynomial transformations of the seed filter, which is termed as an early fan-out. Alternatively, we could also perform late fan-out on the seed filter response to create the number of response maps desired to be input into the next layer. Both early and late fan-out allow the PolyCNN to learn only one convolutional filter at each layer, which can dramatically reduce the model complexity. Parameter savings of at least $10\times$, $26\times$, $50\times$, *etc*. can be realized during the learning stage depending on the spatial dimensions of the convolutional filters (3×3, 5×5, 7×7 *etc*. sized filters respectively). While being efficient during both training and testing, the PolyCNN does not suffer performance due to the non-linear polynomial expansion which translates to richer representational power within the convolutional layers. We have verified the on-par performance between the proposed PolyCNN and the standard CNN on several visual datasets, such as MNIST, CIFAR-10, SVHN, and ImageNet.

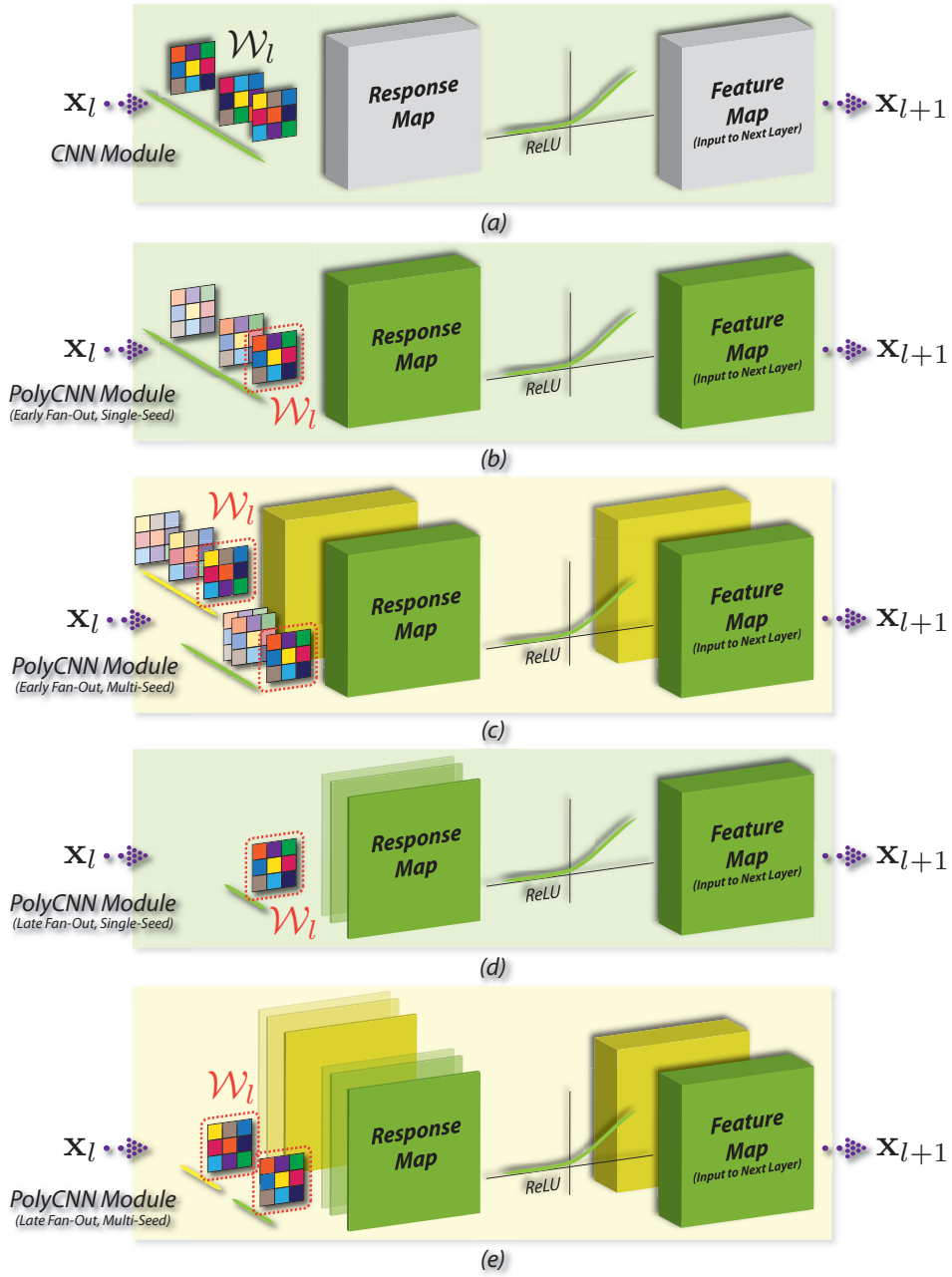

Figure 1: Basic module in (a) CNN and (b-e) PolyCNN (both early fan-out and late fan-out). $\mathcal{W}_l$ and $\mathcal{W}_l$ (encircled in red dashed line) are the learnable weights for CNN and PolyCNN respectively. (b-c) Early fan-out PolyCNN, single-seed and multi-seed cases. (d-e) Late fan-out PolyCNN, single-seed and multi-seed cases.

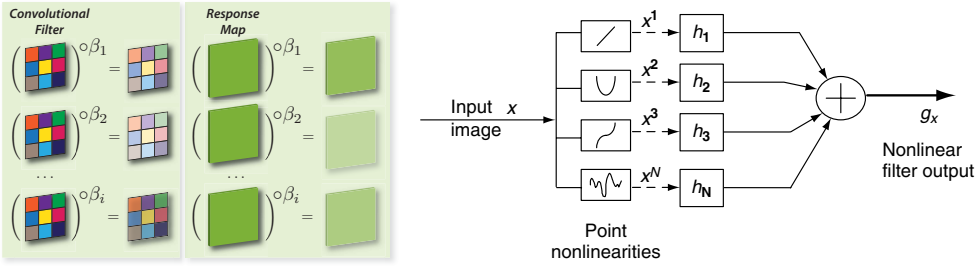

Figure 2: (L) Polynomial expansion of the seed filter as well as the seed filter response map, with exponent parameters $\beta_1, \ldots, \beta_i$, (R) Image taken from Vijaya Kumar et al. (2005): polynomial correlation filter.

2 PROPOSED METHOD

2.1 POLYNOMIAL CONVOLUTIONAL NEURAL NETWORKS

Two decades ago, Mahalanobis & Vijaya Kumar (1997) generalized the traditional correlation filter and created the polynomial correlation filter (PCF), whose fundamental difference is that the correlation output from a PCF is a nonlinear function of the input. As shown in Figure 2(R), the input image $\boldsymbol{x}$ undergoes a set of point-wise nonlinear transformation (polynomial) for augmenting the input channels. Based on some pre-defined objective function, usually in terms of simultaneously maximizing average correlation peak and minimizing some correlation filter performance criterion such as average similarity measure (ASM) (Mahalanobis et al., 1994), output noise variance (ONV) (Vijaya Kumar et al., 2005), the average correlation energy (ACE) (Vijaya Kumar et al., 2005), or any combination thereof, the filters $\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_N$ can be solved in closed-form (Mahalanobis & Vijaya Kumar, 1997; Vijaya Kumar et al., 2005; Alkanhal & Vijaya Kumar, 2003).

We draw inspiration from the design principles of the polynomial correlation filter and propose the polynomial convolutional neural network (PolyCNN) as a weight-learning efficient variant of the traditional convolutional neural networks. The core idea of PolyCNN is that at each convolutional layer, only one convolutional filter (seed filter) needs to be learned, and we can augment other filters by taking point-wise polynomials of the seed filter. The weights of these augmented filters need not to be updated during the network training. When convolved with the input data, the learnable seed filter and k non-learnable augmented filters result in $(k+1)$ response maps. We call this procedure: early fan-out. Similarly, one can instead fan-out the response map from the seed filter to create $(k+1)$ response maps for the subsequent layers. We call this procedure: late fan-out. The details of both early and late fan-out are shown in the following sections. The PolyCNN pipelines are depicted in Figure 1 with distinctions between early and late fan-out, as well as single-seed vs. multi-seed cases. Figure 2(L) shows the polynomial expansion of seed filters as well as seed filter response maps.

2.2 EARLY FAN-OUT: FILTER WEIGHTS

At any given layer, given the seed weights $\boldsymbol{w}_i$ for that layer, we generate many new filter weights. The weights are generated via a non-linear transformation $\boldsymbol{v} = f(\boldsymbol{w}_i)$ of the weights. The convolutional outputs are computed as follows (1-D signals for simplicity):

$$\boldsymbol{y} = \sum_{j=1}^{C} f\left(\boldsymbol{w}_i^j\right) * \boldsymbol{x}^j \implies y[\ell] = \sum_{j=1}^{C} \sum_{k} x^j[\ell - k] f\left(w_i^j[k]\right) \tag{1}$$

where $\boldsymbol{x}^j$ is the j^{th} channel of the input image and $\boldsymbol{w}_i^j$ is the j^{th} channel of the i^{th} filter. During the forward pass weights are generated from the seed convolutional kernel and are then convolved with the inputs *i.e.*,

$$z[i] = f(w[i]) = \text{sign}(w[i])|w[i]|^{\alpha} \tag{2}$$

$$v[i] = \frac{z[i] - \frac{1}{n}\sum_i z[i]}{\left(\sum_i \left(z[i] - \frac{1}{n}\sum_i z[i]\right)^2\right)^{\frac{1}{2}}} \tag{3}$$

where we normalize the response maps to prevent the responses from vanishing or exploding and the normalized response map is now called $\boldsymbol{v}$. $f(\cdot)$ is a non-linear function that operates on each element of $\boldsymbol{w}$. Backpropagating through this filter transformation necessitates the computation of $\frac{\partial l}{\partial \boldsymbol{w}}$ and $\frac{\partial l}{\partial \boldsymbol{x}}$.

$$\frac{\partial l}{\partial w[i]} = \sum_j \frac{\partial l}{\partial y[j]} \frac{\partial y[j]}{\partial w[i]} = \sum_j \frac{\partial l}{\partial y[j]} \frac{\partial y[j]}{\partial f(w[i])} \frac{\partial f(w[i])}{\partial w[i]} = \sum_j \frac{\partial l}{\partial y[j]} x[j-i] f'(w[i]) \tag{4}$$

$$\frac{\partial l}{\partial \boldsymbol{w}} = \left(\frac{\partial l}{\partial \boldsymbol{y}} * \boldsymbol{x}\right) \odot f'(\boldsymbol{w}) \tag{5}$$

Plug in the normalized response map, we have:

$$\frac{\partial v[i]}{\partial w[i]} = \frac{\partial v[i]}{\partial z[i]} \frac{\partial z[i]}{\partial w[i]} \tag{6}$$

$$\frac{\partial v[i]}{\partial z[i]} = \frac{1 - \frac{1}{n}}{\left(\sum_i \left(z[i] - \frac{1}{n}\sum_i z[i]\right)^2\right)^{\frac{1}{2}}} - \frac{\left(1 - \frac{1}{n}\right)\left(z[i] - \frac{1}{n}\sum_j z[j]\right)}{\left(\sum_i \left(z[i] - \frac{1}{n}\sum_i z[i]\right)^2\right)^{\frac{3}{2}}} \tag{7}$$

$$\frac{\partial z[i]}{\partial w[i]} = \text{sign}\left(w[i]\right) j w[i]^{j-1} \tag{8}$$

Similarly, we can compute the gradient with respect to input $\boldsymbol{x}$ as follows:

$$\frac{\partial l}{\partial x[i]} = \sum_j \frac{\partial l}{\partial y[j]} \frac{\partial y[j]}{\partial x[i]} = \sum_j \frac{\partial l}{\partial y[j]} f(w[j-i]) \tag{9}$$

$$\frac{\partial l}{\partial \boldsymbol{x}} = \frac{\partial l}{\partial \boldsymbol{y}} * f(\boldsymbol{w}) \tag{10}$$

2.3 Late Fan-Out: Response Maps

At any given layer, we compute the new feature maps from the seed feature maps via non-linear transformations of the feature maps. The forward pass for this layer involves the application of the following non-linear function $s[i] = f(x[i])$,

$$s[i] = f(x[i]) = \text{sign}(x[i])|x[i]|^{\alpha} \tag{11}$$

$$t[i] = \frac{s[i] - \frac{1}{n}\sum_i s[i]}{\left(\sum_i \left(s[i] - \frac{1}{n}\sum_i s[i]\right)^2\right)^{\frac{1}{2}}} \tag{12}$$

where we normalize the response maps to prevent the responses from vanishing or exploding and the normalized response map is now called $\boldsymbol{t}$. Backpropagating through such a transformation of the response maps requires the computation of $\frac{\partial l}{\partial \boldsymbol{x}}$.

$$\frac{\partial l}{\partial x[i]} = \frac{\partial l}{\partial t[i]} \frac{\partial t[i]}{\partial s[i]} \frac{\partial s[i]}{\partial x[i]} \tag{13}$$

$$\frac{\partial t[i]}{\partial s[i]} = \frac{1 - \frac{1}{n}}{\left(\sum_i \left(s[i] - \frac{1}{n}\sum_i s[i]\right)^2\right)^{\frac{1}{2}}} - \frac{\left(1 - \frac{1}{n}\right)\left(s[i] - \frac{1}{n}\sum_j s[j]\right)}{\left(\sum_i \left(s[i] - \frac{1}{n}\sum_i s[i]\right)^2\right)^{\frac{3}{2}}} \tag{14}$$

$$\frac{\partial s[i]}{\partial x[i]} = \text{sign}\left(x[i]\right) j x[i]^{j-1} \tag{15}$$

2.4 Design of the Basic PolyCNN Module

The core idea of the PolyCNN[1] is to restrict the network to learn only one (or a few) convolutional filter at each layer, and through polynomial transformations we can augment the convolutional filters, or the response maps. The gist is that the augmented filters do not need to be updated or learned during the network back-propagation. As shown in Figure 1, the basic module of PolyCNN (early fan-out, single-seed) starts with just one learnable convolutional filter $\mathcal{W}_l$, which we call the seed filter. If we desire m filters in total for one layer, the remaining $m-1$ filters are non-learnable and are the polynomial transformation of the seed filter $\mathcal{W}_l$. The input image $\boldsymbol{x}_l$ is filtered by these convolutional filters and becomes m response maps, which are then passed through a non-linear activation gate, such as ReLU, and become m feature maps. Optionally, these m feature maps can be further lineally combined using m learnable weights, which is essentially another convolution operation with filters of size 1×1.

[1] In this paper we assume convolutional filters do not have bias terms.

Compared to the CNN module under the same structure (with 1×1 convolutions), the number of learnable parameters is significantly smaller in PolyCNN. Let us assume that the number of input and output channels are p and q. Therefore, the size of each 3D filter in both CNN and PolyCNN is $p \cdot h \cdot w$, where h and w are the spatial dimensions of the filter, and there are m such filters. The 1×1 convolutions act on the m filters and create the q-channel output. For standard CNN, the number of learnable weights is $p \cdot h \cdot w \cdot m + m \cdot q$. For PolyCNN, the number of learnable weights is $p \cdot h \cdot w \cdot 1 + m \cdot q$. For simplicity let us assume $p = q$, which is usually the case for multi-layer CNN architecture. Then we have the parameter saving ratio:

$$\tau = \frac{\text{\# parameters in CNN}}{\text{\# parameters in PolyCNN}} = \frac{p \cdot h \cdot w \cdot m + m \cdot q}{p \cdot h \cdot w \cdot 1 + m \cdot q} = \frac{h \cdot w \cdot m + m}{h \cdot w + m} \tag{16}$$

and when the spatial filter size $h = w = 3$ and the number of convolutional filters desired for each layer $m \gg 3^2$, we have the parameter saving ratio $\tau = \frac{10m}{m+9} \approx 10$. Similarly for spatial filter size $h = w = 5$ and $m \gg 5^2$, the parameter saving ratio $\tau = \frac{26m}{m+25} \approx 26$. For spatial filter size $h = w = 7$ and $m \gg 7^2$, the parameter saving ratio $\tau = \frac{50m}{m+49} \approx 50$.

If we do not include the 1×1 convolutions for both standard CNN and PolyCNN, and thus make $m = q = p$, readers can verify that the parameter saving ratio τ becomes m. Numerically, PolyCNN saves around $10\times$, $26\times$, and $50\times$ parameters during learning for 3×3, 5×5, and 7×7 convolutional filters respectively. The aforementioned calculation also applies to late fan-out of the PolyCNN.

2.5 Training of the PolyCNN

The training of the PolyCNN is quite straightforward, where the back-propagation is the same for the learnable weights and the augmented weights that do not update. Gradients get propagated through the polynomial augmented filters just like they would with learnable filters. This is similar to propagating gradients through layers without learnable parameters *e.g.*, ReLU, Max Pooling *etc.*). However, we do not compute the gradient with respect to the fixed filters nor update them during the training process.

The non-learnable filter banks (tensor) of size $p \times h \times w \times (m-1)$ (assuming a total of m filters in each layer) in the PolyCNN can be generated by taking polynomial transformations from the seed filter, by raising to some exponents, which can either be integer exponents, or fractional exponents that are randomly sampled from a distribution. Strictly speaking, to qualify for polynomials, only non-negative integer powers are allowed. In this work, without violating the forward and backward pass derivation, we allow the exponents to take negative numbers (relating to Laurent series), and even fractional numbers (relating to Puiseux series).

3 Towards a General Convolutional Layer

In this section, we will first analyze the PolyCNN layer and how the early fan-out and late fan-out can very well approximate the standard convolutional layer. Then, we will extend the formulation to a generalized convolutional layer representation.

3.1 Understanding PolyCNN Layer

At layer l, let $\boldsymbol{x}_\pi \in \mathbb{R}^{(p \cdot h \cdot w) \times 1}$ be a vectorized single patch from the p-channel input maps at location π, where h and w are the spatial sizes of the convolutional filter. Let $\boldsymbol{w} \in \mathbb{R}^{(p \cdot h \cdot w) \times 1}$ be a vectorized single convolution filter from the convolutional filter tensor $\boldsymbol{W} \in \mathbb{R}^{p \times h \times w \times m}$ which contains a total of m fan-out convolutional filters at layer l. We drop the layer subscription l for brevity.

In a standard CNN, this patch $\boldsymbol{x}_\pi$ is taken dot-product with (projected onto) the filter $\boldsymbol{w}$, followed by the non-linear activation resulting in a single output feature value d_π, at the corresponding location π on the feature map. Similarly, each value of the output feature map is a direct result of convolving the entire input map $\boldsymbol{x}$ with a convolutional filter $\boldsymbol{w}$. This microscopic process can be expressed as:

$$d_\pi = \sigma_{\text{relu}}(\boldsymbol{w}^\top \boldsymbol{x}_\pi) \tag{17}$$

Without loss of generality, we assume single-seed PolyCNN case for the following analysis. For an early fan-out PolyCNN layer, a single seed filter $\boldsymbol{w}_S$ is expanded into a set of m convolutional filters

$\boldsymbol{W} \in \mathbb{R}^{m \times p \times h \times w}$ where $\boldsymbol{w}_i = \boldsymbol{w}_{\mathcal{S}}^{\circ \beta_i}$, and the power terms $\beta_1, \beta_2, \ldots, \beta_m$ are pre-defined and are not updated during training. The $\circ$ is again the Hadamard power.

The corresponding output feature map value d_π^{early} for early fan-out PolyCNN layer is a linear combination of multiple elements from the intermediate response maps (implemented as 1×1 convolution). Each slice of this response map is obtained by convolving the input map $\boldsymbol{x}$ with $\boldsymbol{W}$, followed by a non-linear activation. The corresponding output feature map value d_π^{early} is thus obtained by linearly combining the m response maps via 1×1 convolution with parameters $\alpha_1, \alpha_2, \ldots, \alpha_m$. This entire process can be expressed as:

$$d_\pi^{\text{early}} = \sigma_{\text{relu}}(\underbrace{\boldsymbol{W}\boldsymbol{x}_\pi)^\top}_{1 \times m} \underbrace{\boldsymbol{\alpha}}_{m \times 1} = \boldsymbol{c}_{\text{relu}}^\top \boldsymbol{\alpha} \tag{18}$$

where $\boldsymbol{W}$ is now a 2D matrix of size $m \times (p \cdot h \cdot w)$ with m filters $\text{vec}(\boldsymbol{w}_i)$ stacked as rows, with a slight abuse of notation. $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_m]^\top \in \mathbb{R}^{m \times 1}$. Comparing Equation 17 and 18, we consider the following two cases (i) $d_\pi = 0$: since $\boldsymbol{c}_{\text{relu}} = \sigma_{\text{relu}}(\boldsymbol{W}\boldsymbol{x}_\pi) \geq 0$, a vector $\boldsymbol{\alpha} \in \mathbb{R}^{m \times 1}$ always exists such that $d_\pi^{\text{early}} = d_\pi$. However, when (ii) $d_\pi > 0$: it is obvious that the approximation does not hold when $\boldsymbol{c}_{\text{relu}} = \boldsymbol{0}$. Therefore, under the mild assumption that $\boldsymbol{c}_{\text{relu}}$ is not an all-zero vector, the approximation $d_\pi^{\text{early}} \approx d_\pi$ will hold.

For the late fan-out PolyCNN layer, a single response map is a direct result of convolving the input map $\boldsymbol{x}$ with the seed convolutional filter $\boldsymbol{w}_{\mathcal{S}}$. Then, we obtain a set of m response maps by expanding the response map with Hadamard power coefficients $\beta_1, \beta_2, \ldots, \beta_m$ which are pre-defined and not updated during training, just like in the early fan-out case. The corresponding output feature map value d_π^{late} is also a linear combination of the corresponding elements from the m response maps via 1×1 convolution with parameters $\alpha_1, \alpha_2, \ldots, \alpha_m$. This process follows:

$$d_\pi^{\text{late}} = \sigma_{\text{relu}} \underbrace{\begin{bmatrix} (\boldsymbol{w}_{\mathcal{S}}^\top \boldsymbol{x}_{\boldsymbol{\pi}})^{\circ \beta_1} \\ (\boldsymbol{w}_{\mathcal{S}}^\top \boldsymbol{x}_{\boldsymbol{\pi}})^{\circ \beta_2} \\ \cdots \\ (\boldsymbol{w}_{\mathcal{S}}^\top \boldsymbol{x}_{\boldsymbol{\pi}})^{\circ \beta_m} \end{bmatrix}^\top}_{1 \times m} \underbrace{\boldsymbol{\alpha}}_{m \times 1} = \sigma_{\text{relu}} \begin{bmatrix} \phi_{\beta_1}(\boldsymbol{w}_{\mathcal{S}})^\top \phi_{\beta_1}(\boldsymbol{x}_{\boldsymbol{\pi}}) \\ \phi_{\beta_2}(\boldsymbol{w}_{\mathcal{S}})^\top \phi_{\beta_2}(\boldsymbol{x}_{\boldsymbol{\pi}}) \\ \cdots \\ \phi_{\beta_m}(\boldsymbol{w}_{\mathcal{S}})^\top \phi_{\beta_m}(\boldsymbol{x}_{\boldsymbol{\pi}}) \end{bmatrix}^\top \boldsymbol{\alpha} = \boldsymbol{c}_{\text{relu}}^\top \boldsymbol{\alpha} \tag{19}$$

where $\phi_{\beta_i}(\cdot)$ is the point-wise polynomial expansion with Hadamard power β_i. With similar reasoning and the mild assumption that $\boldsymbol{c}_{\text{relu}}$ is not an all-zero vector, the approximation $d_\pi^{\text{late}} \approx d_\pi$ will hold.

3.2 General Convolutional Layer

The formulation of PolyCNN discussed in the previous sections has facilitated the forming of a general convolution layer. To simplify the notation, we use 1D convolution as an example. The idea can be extended to 2D and higher dimensional convolution as well. Here is a description of a general convolutional layer:

$$\boldsymbol{y} = \sum_{k=1}^{K} \sum_{l=1}^{L} \alpha_{kl} \sigma \left(\phi_k(\boldsymbol{w}) * \phi_l^{'}(\boldsymbol{x}) \right) \tag{20}$$

where $\phi_k(\cdot)$ and $\phi_l(\cdot)$ are kernel functions, $\sigma(\cdot)$ is a non-linearity and α_{kl} are linear weights. If both $\phi_k(\cdot)$ and $\phi_l(\cdot)$ are linear functions then the expression reduces to a module consisting of convolutional layer, non-linear activation and 1×1 convolutions. Under this general setting, the learnable parameters are $\boldsymbol{\alpha}$ and $\boldsymbol{w}$. Setting the parameters $\boldsymbol{w}$ to fixed sparse binary values would allow us to arrive a general version of local binary CNN (Juefei-Xu et al., 2017). Now we consider some special cases. If $\phi_k(\cdot)$ is a linear function, then the expression reduces to:

$$\boldsymbol{y} = \sum_{l=1}^{L} \alpha_l \sigma \left(\boldsymbol{w} * \phi_l^{'}(\boldsymbol{x}) \right) \tag{21}$$

Similarly if $\phi_l(\cdot)$ is a linear function, then the expression reduces to:

$$\boldsymbol{y} = \sum_{k=1}^{K} \alpha_k \sigma \left(\phi_k(\boldsymbol{w}) * \boldsymbol{x} \right) \tag{22}$$

At location π of the input $\boldsymbol{x}$, the convolutions can be reduced to:

$$y[\pi] = \sum_{k=1}^{K}\sum_{l=1}^{L} \alpha_{kl}\sigma\left(\phi_k(\boldsymbol{w})^\top \phi_l^{'}(\boldsymbol{x}_\pi)\right) = \sum_{k=1}^{K}\sum_{l=1}^{L} \alpha_{kl}\sigma\left(\mathcal{K}_{kl}(\boldsymbol{w}, \boldsymbol{x}_\pi)\right) \tag{23}$$

where $\mathcal{K}_{kl}$ is a base kernel function defined between $\boldsymbol{w}$ and image patch $\boldsymbol{x}_\pi$ at the π location. The base kernel can take many forms, including polynomials, random Fourier features, Gaussian radial basis functions, *etc*., that adhere to the Mercer's theorem (Schölkopf et al., 2002). The weights α_{kl} can be learned via 1×1 convolutions. In general $\alpha_{kl} > 0$ must hold for a valid overall kernel function, but perhaps this can be relaxed or imposed during the learning process. We can also think of Equation 23 as a generalized learnable activation function if we get rid of $\sigma(\cdot)$. So $\phi(\cdot)'$ and α together can approximate any desired activation function. There are several related work on the combination of kernels and convolutional neural networks such as Mairal et al. (2014); Mairal (2016); Zhang et al. (2016b; 2017). As can be seen, the early fan-out PolyCNN layer can be related to Equation 22 and the late fan-out PolyCNN layer can be related to Equation 21 and 23.

4 Experimental Results

4.1 Datasets

We have experimented with 4 publicly available visual datasets, MNIST (LeCun et al., 1998), SVHN (Netzer et al., 2011), CIFAR-10 (Krizhevsky & Hinton, 2009), and ImageNet ILSVRC-2012 classification dataset (Russakovsky et al., 2015). The **MNIST** dataset contains a training set of 60K and a testing set of 10K 32×32 gray-scale images showing hand-written digits from 0 to 9. **SVHN** is also a widely used dataset for classifying digits, house number digits from street view images in this case. It contains a training set of 604K and a testing set of 26K 32×32 color images showing house number digits. **CIFAR-10** is an image classification dataset containing a training set of 50K and a testing set of 10K 32×32 color images, which are across the following 10 classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks. The **ImageNet** ILSVRC-2012 classification dataset consists of 1000 classes, with 1.28 million images in the training set and 50K images in the validation set, where we use for testing as commonly practiced. For faster roll-out, we first randomly select 100 classes with the largest number of images (1300 training images in each class, with a total of 130K training images and 5K testing images.), and report top-1 accuracy on this subset. Full ImageNet experimental results are also reported in the subsequent section.

4.2 Implementation Details

Conceptually PolyCNN can be easily implemented in any existing deep learning framework. Since the convolutional weights are fixed, we do not have to compute the gradients nor update the weights. This leads to savings both from a computational point of view and memory as well. We have used a custom implementation of backpropagation through the PolyCNN layers that is 3x-5x more efficient than autograd-based back propagation in PyTorch.

We base the model architectures we evaluate in this paper on ResNet (He et al., 2016a), with default 3×3 filter size. Our basic module is the PolyCNN module shown in Figure 1 along with an identity connection as in ResNet. We experiment with different numbers of PolyCNN layers, 10, 20, 50, and 75, which is equivalent to 20, 40, 100, and 150 convolutional layers (1×1 convolution counted).

For PolyCNN, the convolutional weights are generated following the procedure described in Section 2.5. We use 511 randomly sampled fractional exponents for creating the polynomial filter weights (512 convolutional filters in total at each layer), for all of our MNIST, SVHN, and CIFAR-10 experiments. Spatial average pooling is adopted after the convolution layers to reduce the spatial dimensions of the image to 6×6. We use a learning rate of 1e-3 and following the learning rate decay schedule from He et al. (2016a). We use ReLU nonlinear activation and batch normalization (Ioffe & Szegedy, 2015) after PolyCNN convolutional module.

For our experiments with ImageNet-1k, we experiment with *ad hoc* CNN architectures such as the AlexNet and the native ResNet family, including ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152. Standard CNN layers are thus replaced with the proposed PolyCNN layers.

Table 1: Classification accuracy (%). PolyCNN rows only show the best performing (single-seed) model and the Baseline row shows the particular CNN counterpart.

	MNIST	SVHN	CIFAR-10
PolyCNN (early fan-out)	99.37	93.29	90.56
PolyCNN (late fan-out)	98.77	90.11	85.98
Baseline	99.48	95.21	92.95
BC (Courbariaux et al., 2015)	98.99	97.85	91.73
BNN (Courbariaux & Bengio, 2016)	98.60	97.49	89.85
ResNet (He et al., 2016b)	/	/	93.57
Maxout (Goodfellow et al., 2013)	99.55	97.53	90.65
NIN (Lin et al., 2014)	99.53	97.65	91.19

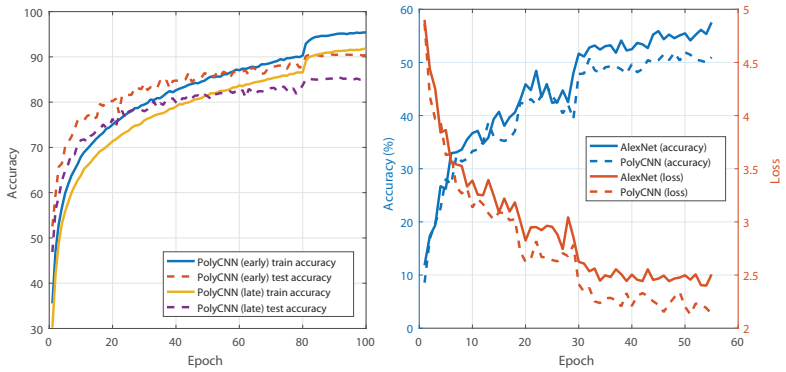

Figure 3: (L) Accuracy of the best performing single-seed PolyCNN (early fan-out) and single-seed PolyCNN (late fan-out) on CIFAR-10. (R) Accuracy and loss on full ImageNet classification.

4.3 Results on MNIST, SVHN, and CIFAR-10

For a fair comparison and to quantify the exact difference between our PolyCNN approach and traditional CNN, we compare ours against the exact corresponding network architecture with dense and learnable convolutional weights. We also use the exact same data and hyper-parameters in terms of the number of convolutional weights, initial learning rate and the learning rate schedule. In this sense, PolyCNN enjoys $10\times$, $26\times$, $50\times$, *etc*. savings in the number of learnable parameters because the baseline CNNs also have the 1×1 convolutional layer. The best performing single-seed PolyCNN models in terms of early fan-out are:

- For MNIST: 75 PolyCNN layers, $m = 512$, $q = 256$, 128 hidden units in the fc layer.
- For SVHN: 50 PolyCNN layers, $m = 512$, $q = 256$, 512 hidden units in the fc layer.
- For CIFAR-10: 50 PolyCNN layers, $m = 512$, $q = 384$, 512 hidden units in the fc layer.

Table 1 consolidates the images classification accuracies from our experiments. The best performing PolyCNNs are compared to their particular baselines, as well as the state-of-the-art methods such as BinaryConnect (Courbariaux et al., 2015), Binarized Neural Networks (BNN) (Courbariaux & Bengio, 2016), ResNet (He et al., 2016b), Maxout Network (Goodfellow et al., 2013), Network in Network (NIN) (Lin et al., 2014). The network structure for the late fan-out follows that of the early fan-out. As can be seen, performance from late fan-out is slightly inferior, but early fan-out reaches on-par performance while enjoying huge parameter savings.

4.3.1 Early Fan-Out vs. Late Fan-Out

Table 2 compares the accuracy on CIFAR-10 achieved by various single-seed PolyCNN architectures (both early and late fan-out) as well as their standard CNN counterparts. We can see that for a fixed number of convolution layers and filters, the more output channels q leads to higher performance. Also, PolyCNN (early fan-out) is on par with the CNN counterpart, while saves $10\times$ parameters. As can be seen from Table 2 and Figure 3(L), the early fan-out version of the PolyCNN is quite comparable to the standard CNN, and is better than its late fan-out counterpart.

Table 2: Classification accuracy $(\%)$ on CIFAR-10 with 20 convolution layers and 512 filters in each layer.

q	32	64	128	192	256	384
Baseline	86.30	88.77	90.86	91.69	92.15	**92.93**
PolyCNN (early fan-out)	83.49	86.11	88.60	89.47	90.01	**90.06**
PolyCNN (late fan-out)	79.23	81.77	84.01	85.36	85.44	**85.50**

Table 3: Classification accuracy $(\%)$ on CIFAR-10 with 20 convolution layers and 512 filters in each layer.

# Seed Filters	1	2	4	8	16	32	64	128	256	512
PolyCNN (early fan-out)	87.24	88.06	88.76	88.98	89.35	90.02	90.78	91.89	92.28	92.48
PolyCNN (late fan-out)	81.73	83.42	85.50	86.95	88.91	90.34	91.48	92.09	92.21	92.33

4.3.2 VARYING THE NUMBER OF SEED FILTERS

Here we report CIFAR-10 accuracy by varying the number of seed filters in Table 3. The network has 20 PolyCNN layers, and the total number of filters per layer is set to 512. We now vary the number of seed filters from 1 to 512, by a factor of 2. So when the number of seed filters is approaching 512, PolyCNN reduces to standard CNN. As can be seen, as we increase the number of seed filters, we are essentially increase the model complexity and the performance is rising monotonically. This experiment will provide insight into trading-off between performance and model complexity.

4.4 RESULTS ON 100-CLASS IMAGENET SUBSET

We report the top-1 accuracy on 100-Class subset of ImageNet 2012 classification challenge dataset in Table 4. The input images of ImageNet is much larger than those of MNIST, SVHN, and CIFAR-10, which allows us to experiments with various convolutional filter sizes. Both the PolyCNN and our baseline share the same architecture: 20 PolyCNN layers, 512 convolutional filters, 512 output channels, 4096 hidden units in the fully connected layer. For this experiment, we omit the late fan-out and only use the better performing early fan-out version of the PolyCNN.

4.5 RESULTS ON FULL IMAGENET

The first *ad hoc* network architecture we experiment with is the AlexNet (Krizhevsky et al., 2012). We train a PolyCNN version of the AlexNet to take on the full ImageNet classification task. The AlexNet architecture is comprised of five consecutive convolutional layers, and two fully connected layers, mapping from the image $(224 \times 224 \times 3)$ to the 1000-dimension feature for the classification purposes in the forward pass. The number of convolutional filters used and their spatial sizes are tabulated in Table 5. For this experiment, we create a single-seed PolyCNN (early fan-out) counterpart following the AlexNet architecture. For each convolutional layer in AlexNet, we keep the same input and output channels. Replacing the traditional convolution module with PolyCNN, we are allowed to specify another hyper-parameter, the fan-out channel m. Table 5 shows the comparison of the number of learnable parameters in convolutional layers in both AlexNet and its PolyCNN counterpart, by setting fan-out channel $m = 256$. As can be seen, PolyCNN saves about $6.4873\times$ learnable parameters in the convolutional layers. What's important is that, by doing so, PolyCNN does not suffer the performance as can be seen in Figure 3(R) and Table 6. We have plotted accuracy curves and loss curves after 55 epochs for both the AlexNet and its PolyCNN counterpart.

The second *ad hoc* network architecture we experiment with is the native ResNet family. We create a single-seed PolyCNN (early fan-out) counterpart following the ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152 architectures, with the same number of input and output channels. The number of convolutional filters in each layer is equivalent for both models. The two baselines are the CNN ResNet implemented by ourselves and by Facebook (Facebook, 2016). Table 7 shows the top-1 accuracy on the two baselines as well as the PolyCNN. Since ResNet is primarily composed of 3×3

Table 4: Classification accuracy $(\%)$ on 100-class ImageNet with varying convolutional filter sizes.

Filter Size	3×3	5×5	7×7	9×9	11×11	13×13
Baseline	65.74	64.90	66.53	65.91	65.22	64.94
PolyCNN	60.47	60.21	60.76	61.16	60.98	60.32

Table 5: Comparison of the number of learnable parameters in convolutional layers in AlexNet and AlexNet with PolyCNN modules. The proposed method saves $6.4873\times$ learnable parameters in the convolutional layers.

Layers	AlexNet (Krizhevsky et al., 2012)	**PolyCNN (AlexNet)**
Layer 1	96*(11*11*3)=34,848	(11*11*3)+96*256=24,576
Layer 2	256*(5*5*48)=307,200	(5*5*48)+256*256=65,536
Layer 3	384*(3*3*256)=884,736	(3*3*256)+384*256=98,304
Layer 4	384*(3*3*192)=663,552	(3*3*192)+384*256=98,304
Layer 5	256*(3*3*192)=442,368	(3*3*192)+256*256=65,536
Total	$2,332,704$ ($\sim 2.333M$)	$359,579$ ($\sim 0.3596M$)

Table 6: Top-1 classification accuracy $(\%)$ on full ImageNet with AlexNet.

	PolyCNN	AlexNet (ours)	AlexNet (BLVC)
ImageNet	51.9008	56.7821	56.9

convolutional filter, the PolyCNN enjoys around 10x parameters savings while achieving competitive performance.

4.6 Discussions

We have shown the effectiveness of the proposed PolyCNN. Not only can it achieve on-par performance with the state-of-the-art, but also enjoy a significant utility savings. The PyTorch implementation of the PolyCNN will be made publicly available.

5 Related Work

Given the proliferation and success of deep convolutional neural networks, there is growing interest in improving the efficiency of such models both in terms computational and memory requirements. Multiple approaches have been proposed to compress existing models as well as to directly train efficient neural networks. Approaches include pruning unnecessary weights in exiting models, sharing of parameters, binarization and more generally quantization of model parameters, transferring the knowledge of high-performance networks into a smaller more more compact network by learning a student network to mimic a teacher network.

The weights of existing networks can be pruned away using the magnitude of weights (Pratt, 1989), or the Hessian of the loss function (Hassibi et al., 1993; LeCun et al., 1989). Ba & Caruana (2014) showed that it is possible to train a shallow but wider student network to mimic a teacher network, performing almost as well as the teacher. Similarly Hinton et al. (2015) proposed Knowledge Distillation to train a student network to mimic a teacher network. Among recent approaches for training high-performance CNNs, PolyNet (Zhang et al., 2016a) shares similar names to our proposed PolyCNN. PolyNet considers higher-order compositions of learned residual functions while PolyCNN considers higher-order polynomials of the weights and response maps.

6 Conclusions

Inspired by the polynomial correlation filter, in this paper, we have proposed the PolyCNN as an alternative to the standard convolutional neural networks. The PolyCNN module enjoys significant savings in the number of parameters to be learned at training, at least $10\times$ to $50\times$. PolyCNN have much lower model complexity compared to traditional CNN with standard convolutional layers. The proposed PolyCNN demonstrates performance on par with the state-of-the-art architectures on several image recognition datasets.

Table 7: Top-1 classification accuracy $(\%)$ on full ImageNet with native ResNet family.

	ResNet-18	ResNet-34	ResNet-50	ResNet-101	ResNet-152
Baseline (Facebook, 2016)	69.57	73.27	75.99	77.56	77.84
Baseline (ours)	67.69	71.38	74.02	75.43	75.48
PolyCNN	62.26	65.46	67.98	69.49	69.69

REFERENCES

Mohamed Alkanhal and B.V.K. Vijaya Kumar. Polynomial distance classifier correlation filter for pattern recognition. *Applied optics*, 42(23):4688–4708, 2003.

Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pp. 2654–2662, 2014.

Matthieu Courbariaux and Yoshua Bengio. BinaryNet: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pp. 3105–3113, 2015.

Facebook. Resnet training in torch, 2016. `https://github.com/facebook/fb.resnet.torch`.

Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*, pp. 293–299. IEEE, 1993.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. *arXiv preprint arXiv:1603.05027*, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *IEEE International Conference on Computer Vision and Pattern Recognition*, 2016b.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

F. Juefei-Xu, V. N. Boddeti, and M. Savvides. Local Binary Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 19–28. IEEE, July 2017.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *CIFAR Dataset*, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPs*, volume 2, pp. 598–605, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *ICLR*, 2014.

Abhijit Mahalanobis and B.V.K. Vijaya Kumar. Polynomial filters for higher order correlation and multi-input information fusion. In *Optoelectronic Information Processing: Invited Contributions from a Workshop Held 2-5 June 1997, Barcelona, Spain*, pp. 1221. SPIE Press, 1997.

Abhijit Mahalanobis, BVK Vijaya Kumar, Sewoong Song, SRF Sims, and JF Epperson. Unconstrained correlation filters. *Applied Optics*, 33(17):3751–3759, 1994.

Julien Mairal. End-to-end kernel learning with supervised convolutional kernel networks. In *Advances in neural information processing systems*, pp. 1399–1407, 2016.

Julien Mairal, Piotr Koniusz, Zaid Harchaoui, and Cordelia Schmid. Convolutional kernel networks. In *Advances in neural information processing systems*, pp. 2627–2635, 2014.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, 2011.

Lorien Y Pratt. *Comparing biases for minimal network construction with back-propagation*, volume 1. Morgan Kaufmann Pub, 1989.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

Bernhard Schölkopf, Alexander J Smola, et al. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations (ICLR)*, 2015.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

B.V.K. Vijaya Kumar, Abhijit Mahalanobis, and Richard D Juday. *Correlation pattern recognition*. Cambridge University Press, 2005.

Shuai Zhang, Jianxin Li, Pengtao Xie, Yingchun Zhang, Minglai Shao, Haoyi Zhou, and Mengyi Yan. Stacked kernel network. *arXiv preprint arXiv:1711.09219*, 2017.

Xingcheng Zhang, Zhizhong Li, Chen Change Loy, and Dahua Lin. Polynet: A pursuit of structural diversity in very deep networks. *arXiv preprint arXiv:1611.05725*, 2016a.

Yuchen Zhang, Percy Liang, and Martin J Wainwright. Convexified convolutional neural networks. *arXiv preprint arXiv:1609.01000*, 2016b.