Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree

Chen-Yu Lee  
UCSD ECE  
chl260@ucsd.edu

Patrick W. Gallagher  
UCSD Cognitive Science  
patrick.w.gallagher@gmail.com

Zhuowen Tu  
UCSD Cognitive Science  
ztu@ucsd.edu

Abstract
We seek to improve deep neural networks by generalizing the pooling operations that play a central role in current architectures. We pursue a careful exploration of approaches to allow pooling to learn and to adapt to complex and variable patterns. The two primary directions lie in (1) learning a pooling function via (two strategies of) combining of max and average pooling, and (2) learning a pooling function in the form of a tree-structured fusion of pooling filters that are themselves learned. In our experiments every generalized pooling operation we explore improves performance when used in place of average or max pooling. We experimentally demonstrate that the proposed pooling operations provide a boost in invariance properties relative to conventional pooling and set the state of the art on several widely adopted benchmark datasets; they are also easy to implement, and can be applied within various deep neural network architectures. These benefits come with only a light increase in computational overhead during training and a very modest increase in the number of model parameters.

1 Introduction
The recent resurgence of neurally-inspired systems such as deep belief nets (DBN) [10], convolutional neural networks (CNNs) [18], and the sum-and-max infrastructure [32] has derived significant benefit from building more sophisticated network structures [38, 33] and from bringing learning to non-linear activations [6, 24]. The pooling operation has also played a central role, contributing to invariance to data variation and perturbation. However, pooling operations have been little revised beyond the current primary options of average, max, and stochastic pooling [3, 40]; this despite indications that e.g. choosing from more than just one type of pooling operation can benefit performance [31].

In this paper, we desire to bring learning and “responsiveness” (i.e., to characteristics of the region being pooled) into the pooling operation. Various approaches are possible, but here we pursue two in particular. In the first approach, we consider combining typical pooling operations (specifically, max pooling and average pooling); within this approach we further investigate two strategies by which to combine these operations. One of the strategies is “unresponsive”; for reasons discussed later, we call this strategy mixed max-average pooling. The other strategy is “responsive”; we call this strategy gated max-average pooling, where the ability to be responsive is provided by a “gate” in analogy to the usage of gates elsewhere in deep learning.

Another natural generalization of pooling operations is to allow the pooling operations that are being combined to themselves be learned. Hence in the second approach, we learn to combine pooling filters that are themselves learned. Specifically, the learning is performed within a binary tree (with number of levels that is pre-specified rather than “grown” as in traditional decision trees) in which each leaf is associated with a learned pooling filter. As we consider internal nodes of the tree, each parent node is associated with an output value that is the mixture of the child node output values, until we finally reach the root node. The root node corresponds to the overall output produced by the tree. We refer to this strategy as tree pooling. Tree pooling is intended (1) to learn pooling filters directly from the data; (2) to learn how to combine leaf node pooling filters in a differentiable fashion; (3) to bring together these other characteristics within a hierarchical tree structure.

When the mixing of the node outputs is allowed to be “responsive”, the resulting tree pooling operation becomes an integrated method for learning pooling filters and combinations of those filters that are able to display a range of different behaviors depending on the characteristics of the region being pooled.

We pursue experimental validation and find that: In the ar-
chitectures we investigate, replacing standard pooling opera-
tions with any of our proposed generalized pooling meth-
ods boosts performance on each of the standard bench-
mark datasets, as well as on the larger and more com-
plex ImageNet dataset. We attain state-of-the-art results
on MNIST, CIFAR10 (with and without data augmentation),
and SVHN. Our proposed pooling operations can be used
to as drop-in replacements for standard pooling opera-
tions in various current architectures and can be used in
tandem with other performance-boosting approaches such
as learning activation functions, training with data augmenta-
tion, or modifying other aspects of network architecture
— we confirm improvements when used in a DSN-style
architecture, as well as in AlexNet and GoogLeNet. Our
proposed pooling operations are also simple to implement,
computationally undemanding (ranging from 5% to 15%
additional overhead in timing experiments), differentiable,
and use only a modest number of additional parameters.

2 Related Work
In the current deep learning literature, popular pooling
functions include max, average, and stochastic pooling
[3, 2, 40]. A recent effort using more complex pooling op-
erations, spatial pyramid pooling [9], is mainly designed
to deal with images of varying size, rather than delving in
to different pooling functions or incorporating learning.
Learning pooling functions is analogous to receptive field
learning [8, 11, 5, 15]. However methods like [15] lead
to a more difficult learning procedure that in turn leads
to a less competitive result, e.g. an error rate of 16.89% on
unaugmented CIFAR10.

Since our tree pooling approach involves a tree structure
in its learning, we observe an analogy to “logic-type” ap-
proaches such as decision trees [27] or “logical operators”
[25]. Such approaches have played a central role in artifi-
cial intelligence for applications that require “discrete” rea-
soning, and are often intuitively appealing. Unfortunately,
despite the appeal of such logic-type approaches, there is
a disconnect between the functioning of decision trees and
the functioning of CNNs — the output of a standard de-
cision tree is non-continuous with respect to its input (and
thus nondifferentiable). This means that a standard deci-
sion tree is not able to be used in CNNs, whose learning
process is performed by back propagation using gradients
of differentiable functions. Part of what allows us to pur-
sue our approaches is that we ensure the resulting pooling
operation is differentiable and thus usable within network
backpropagation.

A recent work, referred to as auto-encoder trees [13], also
pays attention to a differentiable use of tree structures in
deep learning, but is distinct from our method as it focuses
on learning encoding and decoding methods (rather than
pooling methods) using a “soft” decision tree for a gener-
ative model. In the supervised setting, [4] incorporates mul-
tilayer perceptrons within decision trees, but simply uses
trained perceptrons as splitting nodes in a decision forest;
not only does this result in training processes that are sep-
arate (and thus more difficult to train than an integrated
training process), this training process does not involve
the learning of any pooling filters.

3 Generalizing Pooling Operations
A typical convolutional neural network is structured as a
series of convolutional layers and pooling layers. Each con-
volutional layer is intended to produce representations (in
the form of activation values) that reflect aspects of local
spatial structures, and to consider multiple channels when
doing so. More specifically, a convolution layer computes
“feature response maps” that involve multiple channels
within some localized spatial region. On the other hand, a
pooling layer is restricted to act within just one channel at a
time, “condensing” the activation values in each spatially-
local region in the currently considered channel. An early
reference related to pooling operations (although not ex-
licitly using the term “pooling”) can be found in [11].
In modern visual recognition systems, pooling operations
play a role in producing “downstream” representations that
are more robust to the effects of variations in data while
still preserving important motifs. The specific choices of
average pooling [18, 19] and max pooling [28] have been
widely used in many CNN-like architectures; [3] includes
a theoretical analysis (albeit one based on assumptions that
do not hold here).

Our goal is to bring learning and “responsiveness” into
the pooling operation. We focus on two approaches in
particular. In the first approach, we begin with the (con-
ventional, non-learned) pooling operations of max pooling
and average pooling and learn to combine them. Within
this approach, we further consider two strategies by which
to combine these fixed pooling operations. One of these
strategies is “unresponsive” to the characteristics of the re-

gen being pooled; the learning process in this strategy will
result in an effective pooling operation that is some spe-
cific, unchanging “mixture” of max and average. To em-
phasize this unchanging mixture, we refer to this strategy
as mixed max-average pooling.

The other strategy is “responsive” to the characteristics
of the region being pooled; the learning process in this strat-

ey results in a “gating mask”. This learned gating mask
is then used to determine a “responsive” mix of max pool-
ing and average pooling; specifically, the value of the inner
product between the gating mask and the current region be-
ing pooled is fed through a sigmoid, the output of which is
used as the mixing proportion between max and average.
To emphasize the role of the gating mask in determining
the “responsive” mixing proportion, we refer to this strat-

ey as gated max-average pooling.

Both the mixed strategy and the gated strategy involve com-
binations of fixed pooling operations; a complementary
generalization to these strategies is to learn the pooling operations themselves. From this, we are in turn led to consider learning pooling operations and also learning to combine those pooling operations. Since these combinations can be considered within the context of a binary tree structure, we refer to this approach as tree pooling. We pursue further details in the following sections.

3.1 Combining max and average pooling functions

3.1.1 “Mixed” max-average pooling

The conventional pooling operation is fixed to be either a simple average \( f_{\text{ave}}(x) = \frac{1}{N} \sum_{i=1}^{N} x_i \), or a maximum operation \( f_{\text{max}}(x) = \max_i x_i \), where the vector \( x \) contains the activation values from a local pooling region of \( N \) pixels (typical pooling region dimensions are \( 2 \times 2 \) or \( 3 \times 3 \)) in an image or a channel.

At present, max pooling is often used as the default in CNNs. We touch on the relative performance of max pooling and, e.g., average pooling as part of a collection of exploratory experiments to test the invariance properties of pooling functions under common image transformations (including rotation, translation, and scaling); see Figure 2. The results indicate that, on the evaluation dataset, there are regimes in which either max pooling or average pooling demonstrates better performance than the other (although we observe that both of these choices are outperformed by our proposed pooling operations). In the light of observation that neither max pooling nor average pooling dominates the other, a first natural generalization is the strategy we call “mixed” max-average pooling, in which we learn specific mixing proportion parameters from the data. When learning such mixing proportion parameters one has several options (listed in order of increasing number of parameters): learning one mixing proportion parameter (a) per net, (b) per layer, (c) per layer/region being pooled (but used for all channels across that region), (d) per layer/channel (but used for all regions in each channel) (e) per layer/region/channel combination.

The form for each “mixed” pooling operation (written here for the “one per layer” option; the expression for other options differs only in the subscript of the mixing proportion \( a \)) is:

\[
    f_{\text{mix}}(x) = a \cdot f_{\text{max}}(x) + (1 - a) \cdot f_{\text{ave}}(x) \quad (1)
\]

where \( a \in [0, 1] \) is a scalar mixing proportion specifying the specific combination of max and average; the subscript \( \ell \) is used to indicate that this equation is for the “one per layer” option. Once the output loss function \( E \) is defined, we can automatically learn each mixing proportion \( a \) (where we now suppress any subscript specifying which of the options we choose). Vanilla backpropagation for this learning is given by

\[
    \frac{\partial E}{\partial a} = \frac{\partial E}{\partial f_{\text{mix}}(x_i)} \frac{\partial f_{\text{mix}}(x_i)}{\partial a} = \delta (\max_i x_i - \frac{1}{N} \sum_{i=1}^{N} x_i), \quad (2)
\]

where \( \delta = \partial E / \partial f_{\text{mix}}(x_i) \) is the error backpropagated from the following layer. Since pooling operations are typically placed in the midst of a deep neural network, we also need to compute the error signal to be propagated back to the previous layer:

\[
    \frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial f_{\text{mix}}(x_i)} \frac{\partial f_{\text{mix}}(x_i)}{\partial x_i} = \delta \left[ a \cdot 1[x_i = \max x_i] + (1 - a) \cdot \frac{1}{N} \right], \quad (3)
\]

where \( 1[\cdot] \) denotes the 0/1 indicator function. In the experiment section, we report results for the “one parameter per pooling layer” option; the network for this experiment has 2 pooling layers and so has 2 more parameters than a network using standard pooling operations. We found that even this simple option yielded a surprisingly large performance boost. We also obtain results for a simple 50/50 mix of max and average, as well as for the option with the largest number of parameters: one parameter for each combination of layer/channel/region, or \( pc \times ph \times pw \) parameters for each “mixed” pooling layer using this option (where \( pc \) is the number of channels being pooled by the pooling layer, and the number of spatial regions being pooled in each channel is \( ph \times pw \)). We observe that the increase in the number of parameters is not met with a corresponding boost in performance, and so we pursue the “one per layer” option.

3.1.2 “Gated” max-average pooling

In the previous section we considered a strategy that we referred to as “mixed” max-average pooling; in that strat-
Generalizing Pooling Functions in CNNs: Mixed, Gated, and Tree

ey we learned a mixing proportion to be used in combining max pooling and average pooling. As mentioned earlier, once learned, each mixing proportion \( a \) remains fixed — it is “nonresponsive” insofar as it remains the same no matter what characteristics are present in the region being pooled. We now consider a “responsive” strategy that we call “gated” max-average pooling. In this strategy, rather than directly learning a mixing proportion that will be fixed after learning, we instead learn a “gating mask” (with spatial dimensions matching that of the regions being pooled). The scalar result of the inner product between the gating mask and the region being pooled is fed through a sigmoid to produce the value that we use as the mixing proportion. This strategy means that the actual mixing proportion can vary during use depending on characteristics present in the region being pooled. To be more specific, suppose we use \( x \) to denote the values in the region being pooled and \( \omega \) to denote the values in a “gating mask”. The “responsive” mixing proportion is then given by \( \sigma(\omega^T x) \), where \( \sigma(\omega^T x) = 1/(1+\exp(-\omega^T x)) \in [0, 1] \) is a sigmoid function.

Analogously to the strategy of learning mixing proportion parameter, when learning gating masks one has several options (listed in order of increasing number of parameters): learning one gating mask (a) per net, (b) per layer, (c) per layer/region being pooled (but used for all channels across that region), (d) per layer/channel (but used for all regions in each channel) (e) per layer/region/channel combination. We suppress the subscript denoting the specific option, since the equations are otherwise identical for each option.

The resulting pooling operation for this “gated” max-average pooling is:

\[
 f_{\text{gate}}(x) = \sigma(\omega^T x) f_{\text{max}}(x) + (1 - \sigma(\omega^T x)) f_{\text{avg}}(x) \tag{5}
\]

We can compute the gradient with respect to the internal “gating mask” \( \omega \) using the same procedure considered previously, yielding

\[
 \frac{\partial E}{\partial \omega} = \frac{\partial E}{\partial f_{\text{gate}}(x)} \frac{\partial f_{\text{gate}}(x)}{\partial \omega} = \delta \sigma(\omega^T x)(1 - \sigma(\omega^T x)) x \left(\max_i x_i - \frac{1}{N} \sum_{i=1}^{N} x_i\right), \tag{6}
\]

and

\[
 \frac{\partial E}{\partial x_i} = \frac{\partial E}{\partial f_{\text{gate}}(x)} \frac{\partial f_{\text{gate}}(x)}{\partial x_i} \frac{\partial x_i}{\partial x_i} = \delta \left[\sigma(\omega^T x)(1 - \sigma(\omega^T x)) \omega_i \left(\max_i x_i - \frac{1}{N} \sum_{i=1}^{N} x_i\right) \right. \tag{7}
\]

\[
 + \sigma(\omega^T x) \cdot 1[x_i = \max_i x_i] + (1 - \sigma(\omega^T x)) \frac{1}{N} \right]. \tag{8}
\]

In a head-to-head parameter count, every single mixing proportion parameter \( a \) in the “mixed” max-average pooling strategy corresponds to a gating mask \( \omega \) in the “gated” strategy (assuming they use the same parameter count option). To take a specific example, suppose that we consider a network with 2 pooling layers and pooling regions that are \( 3 \times 3 \). If we use the “mixed” strategy and the per-layer option, we would have a total of \( 2 \times 2 \times 1 \) extra parameters relative to standard pooling. If we use the “gated” strategy and the per-layer option, we would have a total of \( 18 \times 2 \times 9 \) extra parameters, where \( 9 \) is the number of parameters in each gating mask. The “mixed” strategy detailed immediately above uses fewer parameters and is “nonresponsive”; the “gated” strategy involves more parameters and is “responsive”. In our experiments, we find that “mixed” (with one mix per pooling layer) is outperformed by “gated” with one gate per pooling layer. Interestingly, an 18 parameter “gated” network with only one gate per pooling layer also outperforms a “mixed” option with far more parameters (40,960 with one mix per layer/channel/region) — except on the relatively large SVHN dataset. We touch on this below; Section 5 contains details.

### 3.1.3 Quick comparison: mixed and gated pooling

The results in Table 1 indicate the benefit of learning pooling operations over not learning. Within learned pooling operations, we see that when the number of parameters in the mixed strategy is increased, performance improves; however, parameter count is not the entire story. We see that the “responsive” gated max-avg strategy consistently yields better performance (using 18 extra parameters) than is achieved with the >40k extra parameters in the 1 per layer/rg/ch “non-responsive” mixed max-avg strategy. The relatively larger SVHN dataset provides the sole exception (SVHN has \( \approx600k \) training images versus \( \approx50k \) for MNIST, CIFAR10, and CIFAR100) — we found baseline 1.91%, 50/50 mix 1.84%, mixed (1 per lyr) 1.76%, mixed (1 per lyr/ch/rg) 1.64%, and gated (1 per lyr) 1.74%.

**Table 1:** Classification error (in %) comparison between baseline model (trained with conventional max pooling) and corresponding networks in which max pooling is replaced by the pooling operation listed. A superscripted \( + \) indicates the standard data augmentation as in [24, 21, 34]. We report means and standard deviations over 3 separate trials without model averaging.

<table>
<thead>
<tr>
<th>Method</th>
<th>MNIST</th>
<th>CIFAR10</th>
<th>CIFAR10(^+)</th>
<th>CIFAR100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>0.39</td>
<td>9.10</td>
<td>7.32</td>
<td>34.21</td>
</tr>
<tr>
<td>w/ Stochastic no learning</td>
<td>0.38±0.04</td>
<td>8.50±0.05</td>
<td>7.30±0.07</td>
<td>33.48±0.07</td>
</tr>
<tr>
<td>w/ 50/50 mix no learning</td>
<td>0.34±0.02</td>
<td>8.11±0.10</td>
<td>6.78±0.17</td>
<td>33.53±0.16</td>
</tr>
<tr>
<td>w/ Mixed 1 per pool layer</td>
<td>0.33±0.018</td>
<td>8.09±0.19</td>
<td>6.62±0.21</td>
<td>33.51±0.11</td>
</tr>
<tr>
<td>2 extra params</td>
<td>0.30±0.012</td>
<td>8.05±0.16</td>
<td>6.58±0.30</td>
<td>33.35±0.19</td>
</tr>
<tr>
<td>w/ Gated 1 per pool layer</td>
<td>0.29±0.016</td>
<td>7.90±0.07</td>
<td>6.36±0.28</td>
<td>33.22±0.16</td>
</tr>
<tr>
<td>18 extra params</td>
<td>0.29±0.016</td>
<td>7.90±0.07</td>
<td>6.36±0.28</td>
<td>33.22±0.16</td>
</tr>
</tbody>
</table>

#### 3.2 Tree pooling

The strategies described above each involve combinations of fixed pooling operations; another natural generalization
of pooling operations is to allow the pooling operations that are being combined to themselves be learned. These pooling layers remain distinct from convolution layers since pooling is performed separately within each channel; this channel isolation also means that even the option that introduces the largest number of parameters still introduces far fewer parameters than a convolution layer would introduce. The most basic version of this approach would not involve combining learned pooling operations, but simply learning pooling operations in the form of the values in pooling filters. One step further brings us to what we refer to as tree pooling, in which we learn pooling filters and also learn to responsively combine those learned filters. Both aspects of this learning are performed within a binary tree (with number of levels that is pre-specified rather than “grown” as in traditional decision trees) in which each leaf is associated with a pooling filter learned during training. As we consider internal nodes of the tree, each parent node is associated with an output value that is the mixture of the child node output values, until we finally reach the root node. The root node corresponds to the overall output produced by the tree and each of the mixtures (by which child outputs are “fused” into a parent output) is responsively learned. Tree pooling is intended (1) to learn pooling filters directly from the data; (2) to learn how to “mix” leaf node pooling filters in a differentiable fashion; (3) to bring together these other characteristics within a hierarchical tree structure. Each leaf node in our tree is associated with a “pooling filter” that will be learned; for a node with index \( m \), we denote the pooling filter by \( \mathbf{v}_m \in \mathbb{R}^N \). If we had a “degenerate tree” consisting of only a single (leaf) node, pooling a region \( x \in \mathbb{R}^N \) would result in the scalar value \( \mathbf{v}_1^T x \). For (internal) nodes at which two child values are combined into a single parent value, we proceed in a fashion analogous to the case of gated max-average pooling, with learned “gating masks” denoted (for an internal node \( m \)) by \( \omega_m \in \mathbb{R}^N \). The “pooling result” at any arbitrary node \( m \) is thus

\[
f_m(x) = \begin{cases} \mathbf{v}_{m, \text{out}}^T x & \text{if leaf node} \\ \sigma(\omega_m^T x) f_{m, \text{in}}(x) + (1 - \sigma(\omega_m^T x)) f_{m, \text{diff}}(x) & \text{if internal node} \end{cases}
\]

The overall pooling operation would thus be the result of evaluating \( f_{\text{root node}}(x) \). The appeal of this tree pooling approach would be limited if one could not train the proposed layer in a fashion that was integrated within the network as a whole. This would be the case if we attempted to directly use a traditional decision tree, since its output presents points of discontinuity with respect to its inputs. The reason for the discontinuity (with respect to input) of traditional decision tree output is that a decision tree makes “hard” decisions; in the terminology we have used above, a “hard” decision node corresponds to a mixing proportion that can only take on the value 0 or 1. The consequence is that this type of “hard” function is not differentiable (nor even continuous with respect to its inputs), and this in turn interferes with any ability to use it in iterative parameter updates during backpropagation. This motivates us to instead use the internal node sigmoid “gate” function \( \sigma(\omega_m^T x) \in [0, 1] \) so that the tree pooling function as a whole will be differentiable with respect to its parameters and its inputs. For the specific case of a “2 level” tree (with leaf nodes “1” and “2” and internal node “3”) pooling function \( f_{\text{tree}}(x) = \sigma(\omega_3^T x) \mathbf{v}_1^T x + (1 - \sigma(\omega_3^T x)) \mathbf{v}_2^T x \), we can use the chain rule to compute the gradients with respect to the leaf node pooling filters \( \mathbf{v}_1, \mathbf{v}_2 \) and the internal node internal gating mask \( \omega_3 \):

\[
\frac{\partial E}{\partial v_1} = \frac{\partial E}{\partial f_{\text{tree}}(x)} \frac{\partial f_{\text{tree}}(x)}{\partial v_1} = \delta(\omega_3^T x)x 
\]

\[
\frac{\partial E}{\partial v_2} = \frac{\partial E}{\partial f_{\text{tree}}(x)} \frac{\partial f_{\text{tree}}(x)}{\partial v_2} = \delta(1 - \sigma(\omega_3^T x))x 
\]

\[
\frac{\partial E}{\partial \omega_3} = \frac{\partial E}{\partial f_{\text{tree}}(x)} \frac{\partial f_{\text{tree}}(x)}{\partial \omega_3} = \delta \sigma(\omega_3^T x)(1 - \sigma(\omega_3^T x))x(\mathbf{v}_1^T - \mathbf{v}_2^T) x 
\]

The error signal to be propagated back to the previous layer is

\[
\frac{\partial E}{\partial x} = \frac{\partial E}{\partial f_{\text{tree}}(x)} \frac{\partial f_{\text{tree}}(x)}{\partial x} = \delta \sigma(\omega_3^T x)(1 - \sigma(\omega_3^T x)) \times (\mathbf{v}_1^T - \mathbf{v}_2^T) x 
\]

3.2.1 Quick comparison: tree pooling

Table 2 collects results related to tree pooling. We observe that on all datasets but the comparatively simple MNIST, adding a level to the tree pooling operation improves performance. However, even further benefit is obtained from the use of tree pooling in the first pooling layer and gated max-avg in the second.

Table 2: Classification error (in %) comparison between our baseline model (trained with conventional max pooling) and proposed methods involving tree pooling. A superscripted + indicates the standard data augmentation as in [24, 21, 34].

<table>
<thead>
<tr>
<th>Method</th>
<th>MNIST</th>
<th>CIFAR10</th>
<th>CIFAR10+</th>
<th>CIFAR100</th>
<th>SVHN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our baseline</td>
<td>0.39</td>
<td>9.10</td>
<td>7.32</td>
<td>34.21</td>
<td>1.91</td>
</tr>
<tr>
<td>Tree</td>
<td>0.35</td>
<td>8.25</td>
<td>6.88</td>
<td>33.53</td>
<td>1.80</td>
</tr>
<tr>
<td>Tree+Max-Avg</td>
<td>0.37</td>
<td>8.22</td>
<td>6.67</td>
<td>33.13</td>
<td>1.70</td>
</tr>
<tr>
<td>Tree+Max-Avg</td>
<td>0.31</td>
<td>7.62</td>
<td>6.05</td>
<td>32.37</td>
<td>1.69</td>
</tr>
</tbody>
</table>

Comparison with making the network deeper using conv layers To further investigate whether simply adding depth to our baseline network gives a performance boost comparable to that observed for our proposed pooling operations, we report in Table 3 below some additional experiments on CIFAR10 (error rate in percent; no data augmentation). If we count depth by counting any layer with learned parameters as an extra layer of depth (even if there is only 1 parameter), the number of parameter layers in a baseline network with 2 additional standard convolution layers matches the number of parameter layers in our best performing net (although the convolution layers contain many more parameters).
Our method requires only 72 extra parameters and obtains state-of-the-art 7.62% error. On the other hand, making networks deeper with conv layers adds many more parameters but yields test error that does not drop below 9.08% in the configuration explored. Since we follow each additional conv layer with a ReLU, these networks correspond to increasing nonlinearity as well as adding depth and adding (many) parameters. These experiments indicate that the performance of our proposed pooling is not accounted for as a simple effect of the addition of depth/parameters/nonlinearity.

Table 3: Classification error (%) on CIFAR10 (without data augmentation) comparison between networks made deeper with standard convolution layers and proposed Tree+(gated) Max-Avg pooling.

<table>
<thead>
<tr>
<th>Method</th>
<th>% Error</th>
<th>Extra parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>9.10</td>
<td>0</td>
</tr>
<tr>
<td>w/ 1 extra conv layer (+ReLU)</td>
<td>9.08</td>
<td>0.6M</td>
</tr>
<tr>
<td>w/ 2 extra conv layers (+ReLU)</td>
<td>9.17</td>
<td>1.2M</td>
</tr>
<tr>
<td>w/ Tree+(gated) Max-Avg</td>
<td>7.62</td>
<td>72</td>
</tr>
</tbody>
</table>

Comparison with alternative pooling layers To see whether we might find similar performance boosts by replacing the max pooling in the baseline network configuration with alternative pooling operations such as stochastic pooling, “pooling” using a stride 2 convolution layer as pooling (cf All-CNN), or a simple fixed 50/50 proportion in max-avg pooling, we performed another set of experiments on unaugmented CIFAR10. From the baseline error rate of 9.10%, replacing each of the 2 max pooling layers with stacked stride 2 conv:ReLU (as in [34]) lowers the error to 8.77%, but adds 0.5M extra parameters. Using stochastic pooling [40] adds computational overhead but no parameters and results in 8.50% error. A simple 50/50 mix of max and average is computationally light and yields 8.07% error with no additional parameters. Finally, our tree+gated max-avg configuration adds 72 parameters and achieves a state-of-the-art 7.62% error.

4 Quick Performance Overview

For ease of discussion, we collect here observations from subsequent experiments with a view to highlighting aspects that shed light on the performance characteristics of our proposed pooling functions.

First, as seen in the experiment shown in Figure 2 replacing standard pooling operations with either gated max-avg or (2 level) tree pooling (each using the “one per layer” option) yielded a boost (relative to max or avg pooling) in CIFAR10 test accuracy as the test images underwent three different kinds of transformations. This boost was observed across the entire range of transformation amounts for each of the transformations (with the exception of extreme downscaling). We already observe improved robustness in this initial experiment and intend to investigate more instances of our proposed pooling operations as time permits.

Second, the performance that we attain in the experiments reported in Figure 2, Table 1, Table 2, Table 4, and Table 5 is achieved with very modest additional numbers of parameters — e.g. on CIFAR10, our best performance (obtained with the tree+gated max-avg configuration) only uses an additional 72 parameters (above the 1.8M of our baseline network) and yet reduces test error from 9.10% to 7.62%; see the CIFAR10 Section for details. In our AlexNet experiment, replacing the maxpool layers with our proposed pooling operations gave a 6\% relative reduction in test error (top-5, single-view) with only 45 additional parameters (above the >50M of standard AlexNet); see the ImageNet 2012 Section for details. We also investigate the additional time incurred when using our proposed pooling operations; in the experiments reported in the Timing section, this overhead ranges from 5\% to 15\%.

Testing invariance properties Before going to the overall classification results, we investigate the invariance properties of networks utilizing either standard pooling operations (max and average) or two instances of our proposed pooling operations (gated max-avg and 2 level tree, each using
in [21]. The learning rate is decreased whenever the validation error stops decreasing; we use the schedule \{0.025, 0.0125, 0.0001\} for all experiments. The momentum of 0.9 and weight decay of 0.0005 are fixed for all datasets as another regularizer besides dropout. All the initial pooling filters and pooling masks have values sampled from a Gaussian distribution with zero mean and standard deviation 0.5. We use these hyperparameter settings for all experiments reported in Tables 1, 2, and 3. No model averaging is done at test time.

5 Experiments

We evaluate the proposed max-average pooling and tree pooling approaches on five standard benchmark datasets: MNIST [20], CIFAR10 [16], CIFAR100 [16], SVHN [26] and ImageNet [30]. To control for the effect of differences in data or data preparation, we match our data and data preparation to that used in [21]. Please refer to [21] for the detailed description.

We now describe the basic network architecture and then will specify the various hyperparameter choices. The basic experiment architecture contains six $3 \times 3$ standard convolutional layers (named conv1 to conv6) and three mlpconv layers (named mlpconv1 to mlpconv3) [24], placed after conv2, conv4, and conv6, respectively. We chose the number of channels at each layer to be analogous to the choices in [24, 21]: the specific numbers are provided in the sections for each dataset. We follow every one of these conv-type layers with ReLU activation functions. One final mlpconv layer (mlpconv4) is used to reduce the dimension of the last layer to match the total number of classes for each different dataset, as in [24]. The overall model has parameter count analogous to [24, 21]. The proposed max-average pooling and tree pooling layers with $3 \times 3$ pooling regions are used after mlpconv1 and mlpconv2 layers\(^1\). We provide a detailed listing of the network configurations in Table A1 in the Supplementary Materials.

Moving on to the hyperparameter settings, dropout with rate 0.5 is used after each pooling layer. We also use hidden layer supervision to ease the training process as

\(^1\)There is one exception: on the very small images of the MNIST dataset, the second pooling layer uses $2 \times 2$ pooling regions.
Table 4: Classification error (in %) reported by recent comparable publications on four benchmark datasets with a single model and no data augmentation, unless otherwise indicated. A superscripted † indicates the standard data augmentation as in [24, 21, 34]. A “...” indicates that the cited work did not report results for that dataset. A fixed network configuration using the proposed tree+max-avg pooling (1 per pool layer option) yields state-of-the-art performance on all datasets (with the exception of CIFAR100).

<table>
<thead>
<tr>
<th>Method</th>
<th>MNIST</th>
<th>CIFAR10</th>
<th>CIFAR10†</th>
<th>CIFAR100</th>
<th>SVHN</th>
<th>GoogLeNet w/ ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN [14]</td>
<td>0.53</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Stoch. Pooling [40]</td>
<td>0.47</td>
<td>15.13</td>
<td>-</td>
<td>42.51</td>
<td>2.80</td>
<td>-</td>
</tr>
<tr>
<td>Maxout Networks [6]</td>
<td>0.45</td>
<td>11.68</td>
<td>9.38</td>
<td>38.57</td>
<td>2.47</td>
<td>-</td>
</tr>
<tr>
<td>Tree Priors [36]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>36.85</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DropConnect [22]</td>
<td>0.57</td>
<td>9.41</td>
<td>9.32</td>
<td>-</td>
<td>1.94</td>
<td>-</td>
</tr>
<tr>
<td>FitNet [29]</td>
<td>0.51</td>
<td>-</td>
<td>8.39</td>
<td>35.04</td>
<td>2.42</td>
<td>-</td>
</tr>
<tr>
<td>NiN [24]</td>
<td>0.47</td>
<td>10.41</td>
<td>8.81</td>
<td>35.68</td>
<td>2.35</td>
<td>-</td>
</tr>
<tr>
<td>DSN [21]</td>
<td>0.39</td>
<td>9.69</td>
<td>7.97</td>
<td>34.57</td>
<td>1.92</td>
<td>-</td>
</tr>
<tr>
<td>NiN + LA units [1]</td>
<td>-</td>
<td>9.59</td>
<td>7.51</td>
<td>34.40</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dssNet [37]</td>
<td>-</td>
<td>9.22</td>
<td>-</td>
<td>33.78</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>All-CNN [34]</td>
<td>-</td>
<td>9.08</td>
<td>7.25</td>
<td>33.71</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>R-CNN [23]</td>
<td>0.31</td>
<td>8.69</td>
<td>7.09</td>
<td>31.75</td>
<td>1.77</td>
<td>-</td>
</tr>
<tr>
<td>Our baseline</td>
<td>0.39</td>
<td>9.10</td>
<td>7.32</td>
<td>32.21</td>
<td>1.91</td>
<td>-</td>
</tr>
<tr>
<td>Our TreesMax-Avg</td>
<td>0.31</td>
<td>7.62</td>
<td>6.05</td>
<td>32.37</td>
<td>1.69</td>
<td>-</td>
</tr>
</tbody>
</table>

figure use $7 \times 9 = 63$ parameters for the (first, 3 level) tree-pooling layer — 4 leaf nodes and 3 internal nodes — and 9 parameters in the gating mask used for the (second) gated max-average pooling layer, while the best result in [34] contains a total of nearly 500,000 parameters in layers performing “pooling like” operations; the relative CIFAR10 accuracies are 7.62% (ours) and 9.08% (All-CNN).

For the data augmentation experiment, we followed the standard data augmentation procedure [24, 21, 34]. When training with augmented data, we observe the same trends seen in the “no data augmentation” experiments. We note that [7] reports a 4.5% error rate with extensive data augmentation (including translations, rotations, reflections, stretching, and shearing operations) in a much wider and deeper 50 million parameter network — 28 times more than are in our networks.

CIFAR100 Our CIFAR100 model has 192 channels for all convolutional layers and \{96, 192, 192\} channels for mlpoolconv1 to mlpoolconv3, respectively.

Street view house numbers Our SVHN model has \{128, 128, 320, 320, 384, 384\} channels for conv1 to conv6 and \{96, 256, 256\} channels for mlpoolconv1 to mlpoolconv3, respectively. In terms of amount of data, SVHN has a larger training data set (>600k versus the ≈50k of most of the other benchmark datasets). The much larger amount of training data motivated us to explore what performance we might observe if we pursued the one per layer/channel/region option, which even for the simple mixed max-avg strategy results in a huge increase in total the number of parameters to learn in our proposed pooling layers: specifically, from a total of 2 in the mixed max-avg strategy, 1 parameter per pooling layer option, we increase to 40,960.

Using this one per layer/channel/region option for the mixed max-avg strategy, we observe test error (in %) of 0.30 on MNIST, 8.02 on CIFAR10, 6.61 on CIFAR10†, 33.27 on CIFAR100, and 1.64 on SVHN. Interestingly, for MNIST, CIFAR10†, and CIFAR100 this mixed max-avg (1 per layer/channel/region) performance is between mixed max-avg (1 per layer) and gated max-avg (1 per layer); on CIFAR10 mixed max-avg (1 per layer/channel/region) is worse than either of the 1 per layer max-avg strategies. The SVHN result using mixed max-avg (1 per layer/channel/region) sets a new state of the art.

ImageNet 2012 In this experiment we do not directly compete with the best performing result in the challenge (since the winning methods [38] involve many additional aspects beyond pooling operations), but rather to provide an illustrative comparison of the relative benefit of the proposed pooling methods versus conventional max pooling on this dataset. We use the same network structure and parameter setup as in [17] (no hidden layer supervision) but simply replace the first max pooling with the proposed 2 level) tree pooling (2 leaf nodes and 1 internal node for $27 = 3 \times 9$ parameters) and replace the second and third max pooling with gated max-average pooling (2 gating masks for $18 = 2 \times 9$ parameters). Relative to the original AlexNet, this adds 45 more parameters (over the >50M in the original) and achieves relative error reduction of 6% (for top-5, single-view) and 5% (for top-5, multi-view).

Our GoogLeNet configuration uses 4 gated max-avg pooling layers, for a total of 36 extra parameters over the 6.8 million in standard GoogLeNet. Table 5 shows a direct comparison (in each case we use single net predictions rather than ensemble).

Table 5: ImageNet 2012 test error (in %). BN denotes Batch Normalization [12].

<table>
<thead>
<tr>
<th>Method</th>
<th>top-1 s-view</th>
<th>top-5 s-view</th>
<th>top-1 m-view</th>
<th>top-5 m-view</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet [17]</td>
<td>43.1</td>
<td>19.9</td>
<td>40.7</td>
<td>18.2</td>
</tr>
<tr>
<td>AlexNet w/ ours</td>
<td>41.4</td>
<td>18.7</td>
<td>39.3</td>
<td>17.3</td>
</tr>
<tr>
<td>GoogLeNet [38]</td>
<td>55.0</td>
<td>27.6</td>
<td>55.0</td>
<td>27.6</td>
</tr>
<tr>
<td>GoogLeNet w/ BN</td>
<td>28.68</td>
<td>9.53</td>
<td>27.81</td>
<td>9.09</td>
</tr>
<tr>
<td>GoogLeNet w/ BN + ours</td>
<td>28.02</td>
<td>9.16</td>
<td>27.60</td>
<td>8.93</td>
</tr>
</tbody>
</table>

6 Observations from Experiments
In each experiment, using any of our proposed pooling operations boosted performance. A fixed network configuration using the proposed tree+max-avg pooling (1 per pool layer option) yields state-of-the-art performance on MNIST, CIFAR10 (with and without data augmentation), and SVHN. We observed boosts in tandem with data augmentation, multi-view predictions, batch normalization, and several different architectures — NiN-style, DSN-style, the >50M parameter AlexNet, and the 22-layer GoogLeNet.
Acknowledgment This work is supported by NSF awards IIS-1216528 (IIS-1360566) and IIS-0844566(IIS-1360568), and a Northrop Grumman Contextual Robotics grant. We are grateful for the generous donation of the GPUs by NVIDIA.

References