HairNet: The Thinned Convolutional Network

Harrison Jansma
hsj180000@utdallas.edu
April 10, 2020

Abstract

In this paper we introduce several methods to the training of deep convolutional neural networks. These methods improve the computational efficiency of the convolutional operation, allow the training of deeper networks, and drastically increase the speed of training.

In the sections that follow, we will introduce depthwise separable convolutions as a factorized alternative to the standard convolutions. Batch normalization which makes normalization of inputs inherent to the network architecture. Lastly, we introduce a new gradient descent optimizer that drastically improves the speed of training.

With all of these design improvements, we propose a novel architecture, titled HairNet, that achieves AlexNet performance on the ImageNet dataset with 4% of the parameters.

1 Introduction

Image recognition is one of the foundational problems of computer vision. In short, it is the task of identifying the dominant object in an image. With the recent promising applications of convolutional neural networks [3] to image recognition we see the potential for major advances in the design and implementation of vision-aware computer systems using deep neural networks.

With the staggering results of Krizhevsky et al. on the already challenging ImageNet dataset (37.5% top-1 error, 17.0% top-5 error), deep convolutional networks (CNNs) are sure to see extensive research and improvement in the near future. This paper seeks to be the first to make significant improvements in the design of CNNs for image recognition. Our new architecture is called HairNet. Titled for its intended use in low-resource hardware, HairNet is light, fast, and efficient.

In this paper we will also introduce several improvements to the design of convolutional networks. The most important of which we call depthwise separable convolutions, global average pooling, and batch normalization. These three improvements are each designed to improve the efficiency and performance of the training process, and together will drastically increase the learning potential of subsequent convolutional network architectures.

2 Related Work

2.1 Design

Convolutions were first popularized by LeCun *et al.* in the late 1980s for the task of digit recognition. [1] Applied to image recognition, these operations utilize 3D kernels with height, width, and channel dimensions to learn patterns in image data.

With the prevalence of large, publicly available image recognition datasets like the ImageNet dataset, [2] interest in applying CNNs has only been limited by the computational requirements of the training process. Krizhevsky et al. utilized high-performance GPUs to enable the training of deep convolutional networks on the ImageNet dataset, winning the ILSVRC-2012 competition.

The proposed CNN architecture of Krizhevsky et al., which we hereafter call AlexNet, utilizes a host of tricks to improve its performance and training time. The ReLU nonlinearity increases neuron resistance to saturation and speeds training time. Max pooling decreases the dimensions of intermediate outputs of convolutional layers, resulting in fewer parameters and computations. Overlapping pooling and local response normalization increase the networks ability to generalize to unseen inputs.

The AlexNet architecture boasts impressive results in image recognition. However, the size and computational requirements of the model restrict its applications to the academic setting.

2.2 Training

Neural networks learn complex patterns by initializing randomized parameters, then applying gradient descent to optimize said parameters over an objective function. Stochastic gradient descent (SGD) optimizes this collection of network parameters Θ to minimize the loss

$$\Theta = argmin_{\Theta} \frac{1}{N} \sum_{i=1}^{N} L(x_i, \Theta)$$

Where $X = [x_1, \dots, x_N]$ is a training dataset. Mini-batch SGD greatly speeds up the training process by only computing the average loss value for a smaller randomized subset of the training data. The small batch size also introduces randomness into the weight update phase that can lead to better model generalization.

While mini-batch SGD is extremely efficient, its main flaw is that achieving good training results require careful tuning of the learning rate. Often leading to multiple iterations of training and retraining a model to find optimal hyperparameter values.

2.3 Implementation

An added benefit of batched backpropagation is the opportunity to exploit parallelism to speed the training process. The application of GPUs to training neural networks has brought with it the possibility of learning from the truly massive datasets we have available to us.

With its focus on parallelism and strong computational ability, GPUs appear to be well matched to the task of iterative training algorithms. The main limitation of this tool is the limited on-device memory contained within most commercial GPUs.

3 Design

The HairNet design is focused on efficiency and size. Before we can describe the model architecture, we must first introduce a few novel methods intended to improve the size and speed of deep convolutional networks.

3.1 Depthwise Separable Convolutions [4][5]

The body of HairNet is based on depthwise separable convolutions which factorizes the traditional convolution operation into a depthwise convolution, that applies a single $K_h \times K_w \times 1$ kernel to each channel, and a pointwise convolution, which applies a $1 \times 1 \times channel$ kernel to the outputs of the depthwise convolution. This results in an initial filtering of information along the height/width dimensions, followed by a feature recombination along the channel dimension, resulting in new features.

Factorization of the traditional convolution greatly improves the efficiency of the operation. Applied to a feature map of size $D_h \times D_w \times M$, standard convolutions require a kernel of size $K \times K \times M \times N$ where K represents the height and width of a square convolutional kernel, D_h and D_w represent the height and width of the input feature map, M is the number of input channels, and N is the number of output channels. Standard convolutions require a computational cost of:

$$K^2 \cdot M \cdot N \cdot D_h \cdot D_w$$

Depthwise separable convolutions apply a depthwise convolution follows by a pointwise convolution. Depthwise convolutions apply a single $K \times K \times 1$ filter to each channel of the input, resulting in a feature map of size $R_h \times \mathrm{R_w} \times \mathrm{M}$. Thus, depthwise convolutions have a computational cost of:

$$K^2 \cdot M \cdot D_h \cdot D_w$$

Depthwise convolutions mix information across the height and width dimension, but do not combine this information across channels to create new features. For this, pointwise convolution applies a kernel of size $1 \times 1 \times M \times N$ to the output of the depthwise convolution. Together depthwise convolutions and pointwise convolution have a computational cost of:

$$K^2 \cdot M \cdot D_h \cdot D_w + M \cdot N \cdot D_h \cdot D_w$$

Resulting in a reduction of computation of:

$$\frac{K^2 \cdot M \cdot D_h \cdot D_w + M \cdot N \cdot D_h \cdot D_w}{K^2 \cdot M \cdot N \cdot D_h \cdot D_w} = \frac{1}{N} + \frac{1}{K^2}$$

Throughout the body of the HairNet network, we utilize depthwise separable convolutions in place of traditional convolutions. This results in a significantly smaller and more efficient convolutional network design.

3.2 Batch Normalization [6]

Despite the introduction of GPU accelerated training, deep convolutional networks remain a significant challenge to train. Along with the issues of divergence and neuron saturation, the training process is also complicated by the changing distribution of each layer's inputs as parameters of earlier layers change.

Change of input in a learning system is called *covariate shift*. [] If we consider each layer in a deep neural network as an independent learner, we see that as weights are updated, layers deeper in the network can be said to experience covariate shift. As a result, deeper networks require careful parameter initialization and lower learning rates to generate good results.

To mitigate the effects of covariate shift, the inputs to a learning system must be normalized to force consistent input distribution. For a layer within a neural network, this means applying a parameterized normalization function that fixes the distribution of layer inputs.

To accomplish this, we normalize each scalar feature independently. For a layer with *d*-dimensional input, we normalize each input dimension with variance and expectation statistics computed from the training dataset.

$$\widehat{x^k} = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

Note that simply normalizing the input can constrain the representational power of a layer. If normalization is applied to a layer with sigmoid nonlinearity, layer activations would be confined to the near-linear portion of the sigmoid function.

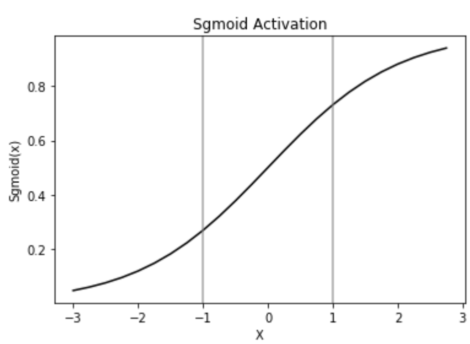

***Figure 1**: Normalized inputs can reduce the representational power of networks with the sigmoid function.*

To address this problem, the normalizing transformation must be capable of representing the identity transformation. For each dimension of the input x^k, two parameters γ^k, β^k are introduced to scale and shift the normalized value.

$$y^k = \gamma^k \widehat{\mathrm{x^k}} + \beta^k$$

These are trainable parameters to be optimized through gradient descent. Thus the model can recapture the original activations if this would result in better model performance.

To improve the performance of the normalization transformation, we introduce one further simplification. We use mean and variance of each mini-batch as estimates of expectation and variance of the training set.

We name this transformation Batch Normalization, as it is a normalization that will occur on the batch-level during gradient descent.

Input: Values of x over mini batch $\boldsymbol{B = x_1, \dots, x_m}$
Parameters to be learned $\boldsymbol{\gamma, \beta}$
Output: $\boldsymbol{y_i = BN_{\gamma,\beta}(x_i)}$

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

$$\widehat{x_i} \leftarrow \frac{(x_i - \mu_B)^2}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x_i} + \beta$$

***Algorithm 1**: Batch Normalization algorithm (ϵ is a small positive value, added for numerical stability).*

The resulting values of the batch normalization transformation are normalized by mini-batch statistics. The distribution of $\hat{x}$ has an expected value of 0 and a variance of 1. The output of $BN_{\gamma,\beta}(x_i)$ can be viewed as a linear transformation with normalized inputs. By introducing normalization to the inputs of neural network layer, we accelerate the training of each subsequent layer, as well as the network as a whole.

The batch normalization transformation is differentiable with respect to the loss function. Ensuring that the network can continue training on inputs that exhibit less variation in distribution.

$$\frac{\partial l}{\partial \widehat{x_i}} = \frac{\partial l}{\partial y_i} \cdot \gamma$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \widehat{x_i}} \cdot \frac{1}{\sqrt{\sigma_\beta^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_\beta^2} \cdot \frac{2(x_i - \mu_\beta)}{m} + \frac{\partial l}{\partial \mu_\beta} \cdot \frac{1}{m}$$

$$\frac{\partial l}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \cdot \widehat{x_i}$$

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}$$

$$\frac{\partial l}{\partial \sigma_\beta^2} = \sum_{i=1}^{m} \frac{\partial l}{\partial \widehat{x_i}} \cdot (x_i - \mu_\beta) \cdot \frac{-1}{2} \left(\sigma_\beta^2 + \epsilon\right)^{-3/2}$$

$$\frac{\partial l}{\partial \mu_\beta} = \left(\sum_{i=1}^{m} \frac{\partial l}{\partial \widehat{x_i}} \cdot \frac{-1}{\sqrt{\sigma_\beta^2 + \epsilon}} \right) + \frac{\partial l}{\partial \sigma_\beta^2} \cdot \frac{\Sigma_{i=1}^{m} - 1\left(x_i - \mu_\beta\right)}{m}$$

Batch normalization can be inserted after the activation of any layer within a neural network. The resulting batch normalized network can be trained with stochastic gradient descent with a mini batch $m > 1$.

During inference, the dependence of the batch normalization transformation on batch data can lead to results that are non-deterministic. This is a characteristic that is unfavorable for model inference. To correct this, during inference we use statistics computed over the entire training set, rather than mini-batch statistics.

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}}$$

We utilize the unbiased estimate $Var[x] = \frac{m}{m-1} \cdot E_\beta\left[\sigma_\beta^2\right]$, with expectation over training mini-batches with sample variances σ_β^2. Thus, inference batch normalization can be seen as a linear transformation with fixed parameters $\gamma, \beta, Var[x]$, and $E[x]$.

Input: Network N with trainable parameters Θ;
Subset of activations $\{x^k\}_{k=1}^{K}$
Output: Batch Normalized network for inference, N_{BN}^{inf}

1. $\mathrm{N_{BN}^{tr}} \leftarrow \mathrm{N}$ //Training BN Network
2. **for** $k = 1 \ldots K$ **do**
3. Add transformation $y^k = BN_{\gamma^k, \beta^k}(x^k)$ to N_{BN}^{tr} (Alg.1)
4. Modify each layer in N_{BN}^{tr} with input x^k to take y^k instead
5. **end for**
6. Train N_{BN}^{tr} to optimize parameters $\Theta \cup [\gamma^k, \beta^k]_{k=1}^{K}$
7. $N_{BN}^{inf} \leftarrow N_{BN}^{tr}$
8. **for** $k = 1, \ldots, K$ **do**
9. Average over multiple training mini-batches β of size m
$$E[x] \leftarrow E_\beta\left[\mu_\beta\right]$$
$$Var[x] \leftarrow \frac{m}{m-1} E_\beta[\sigma_\mathrm{B}^2]$$
10. In N_{BN}^{inf}, replace the transform $y = BN_{\gamma,\beta}(x)$ With
$$y = \frac{\gamma}{\sqrt{Var[x] + \epsilon}} \cdot x + \left(\beta - \frac{\gamma E[x]}{\sqrt{Var[x] + \epsilon}}\right\}\}$$
11. **end for**

Algorithm 2: Training a Batch Normalized Network

In convolutional networks, each layer consists of an affine transformation followed by a pointwise nonlinearity:

$$z = g(Wx + b)$$

To apply Batch Normalization, we add the BN transformation before the pointwise nonlinearity normalizing the result of the affine transformation. We choose to normalize this way, as opposed to normalizing x (which is likely the output of a nonlinear transformation), because the affine transformation is more likely to take a Gaussian distribution, and therefore be optimal for normalization.

Note that the bias b can be ignored, since its effect will be countered during mean subtraction. First look at the batchwise statistic μ_β.

$$\mu_\beta = b + \frac{1}{m} \sum_{i=1}^{m} W x_i$$

Applying this within the Batch Normalization transformation:

$$BN(Wx + b)$$
$$= \gamma \cdot \frac{Wx + b - b + \frac{1}{m} \Sigma_{i=1}^{m} W x_i}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$
$$= BN(Wx)$$

The effect of the bias is maintained by the parameter β.

Applied to convolutional layers, batch normalization must transform all activations in a feature map the same way. To accomplish this, Batch Normalization calculates mean and variance mini-batch statistics for every activation in the height/width domain. We apply normalization to each feature map, and for each maintain a pair of parameters γ_k, β_k.

3.3 Fully Convolutional Architecture [7][8]

Much of the computation in AlexNet resides in the final three fully connected layer. Of its 62 million parameters, roughly 58.63 million are contained with in the fully connected head. To effectively apply deep convolutional networks in an application setting these fully connected layers need to be limited or removed all together.

AlexNet performs vectorization on the outputs of the final convolution, followed by three fully connected layers that feed into a softmax layer. To replace the computationally expensive vectorization and fully connected network head, we introduce *global average pooling.* The transformation maps each feature map to a single value that expresses the strength of a kernel's activation across the entire height/width domain of the image. This is achieved by simply averaging the activations within each feature map.

By generating one feature map per class in the final convolution of a network, we can apply global average pooling to create a vector of size $1 \times C$, where C is the number of classes. By feeding this vector through a softmax layer, we can estimate class probability mass functions without a single fully connected layer.

The design of a fully convolutional architecture not only reduces the size and computational requirements of the proposed network, it also allows forces feature maps to estimate class confidence. Furthermore, it decouples network requirements for a given input image size, allowing for flexibility of aspect ratios in regard to mobile applications

In our experiments, we found that adding a single dense layer to the end of our convolutional network achieves better performance than a fully convolutional architecture. It is left to the discretion of the reader whether this increased accuracy is worth the reduced efficiency in an application setting. Adding a dense layer increased the number of parameters by 1.3x.

3.4 Network Architecture

HairNet utilizes an architecture that favors simplicity. We design a single subcomponent "block" that is repeated several times throughout the architecture. You can see an example of a depthwise separable convolutional block with batch normalization below.

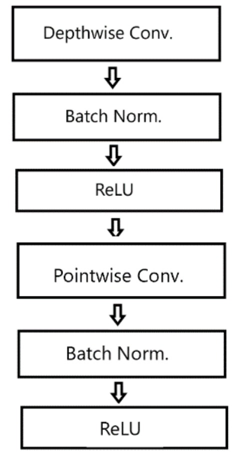

Figure 2: Depthwise separable convolution with Batch Normalization

By repeating these blocks multiple times, we significantly reduce the complexity of architecture design. Our model architecture for the ImageNet dataset is detailed below.

Table 1: HairNet Architecture

Type x Repetitions	Number Filters	Input Size	Total Parameters
Conv-BN-ReLU x 1	32	224 x 224 x 3	928
Max Pool x 1		224 x 224 x 32	
Conv DW block x2	64	112 x 112 x 32	7456
Max Pool x 1		112 x 112 x 64	
Conv DW block x3	128	56 x 56 x 64	44736
Max Pool x 1		56 x 56 x 128	
Conv DW block x3	256	28 x 28 x 128	171392
Max Pool x 1		28 x 28 x 256	
Conv DW block x3	512	14 x 14 x 256	670464
Max Pool x 1		14 x 14 x 512	
Conv DW block x2	1000	7 x 7 x 512	1593856
Global Avg Pool		7 x 7 x 1000	
SoftMax		1 x 1 x 1000	

Note that, at 2.4M parameters, the HairNet architecture is roughly 4% of the size of the AlexNet architecture.

4 Training

4.1 Adam [9]

To optimize the training of our convolutional network, we introduce a new first-order gradient-based optimization algorithm called *Adam.*

Stochastic gradient descent (SGD) proved an advantageous optimization algorithm for deep learning techniques

Input: α: stepsize; $\beta_1, \beta_2 \in [0,1)$: Exp. Decay rates for moment estimates; $L(\theta)$: objective function w/ parameters θ; θ_0 initial parameter vector
Output: θ_t: resulting parameters

$$m_0 \leftarrow 0$$
$$v_0 \leftarrow 0$$
$$t \leftarrow 0$$

While $\boldsymbol{\theta_t}$ not converged **do:**

$$\boldsymbol{t \leftarrow t+1}$$
$$\boldsymbol{g_t \leftarrow \nabla_\theta L_t(\theta_{t-1})}$$
$$\boldsymbol{m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t}$$
$$\boldsymbol{v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot {g^2}_t}$$
$$\boldsymbol{\widehat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}}$$
$$\boldsymbol{\widehat{v}_t \leftarrow \frac{v_t}{(1-\beta 2^t)}}$$
$$\boldsymbol{\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / \left(\sqrt{\widehat{v}_t} + \epsilon\right)}$$

end while
return $\boldsymbol{\theta_t}$

Algorithm 3: Adam for stochastic optimization. g_t^2 *represents the elementwise Hadamard square of* g_t. *Good defaults are* $\alpha = 0.001$, $\beta_1 = 0.9, \beta_2 = 0.999$ *and* $\epsilon = 10^{-8}$. *All operations on vectors are elementwise.*

Adam is derived from adaptive moment estimation. The algorithm updates exponential moving averages of the gradient (m_t) and the squared gradient (v_t). Both estimate the first moment (mean) and second moment (uncentered variance) of the gradients. These are computed with the functions

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$$
$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot {g^2}_t$$

Both m_t and v_t ar initialized as vectors of 0's which are biased towards zero in earlier time steps. To counter this, we calculate bias corrected estimates with

$$\widehat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$$
$$\widehat{v}_t \leftarrow \frac{v_t}{(1-\beta 2^t)}$$

4.2 Experimentation

Batch normalization allowed us to utilize higher learning rates during training. Trained with stochastic gradient descent and equivalent learning rate, HairNet with batch normalization converged while the same network without batch normalization did not.

We also compared a fully convolutional HairNet architecture to an alternate variant that adds a single dense layer after global average pooling. The dense unit has units equal to the number of classes and leads into a softmax layer. We found that adding a single dense layer to the design improved the speed and final accuracy of the network.

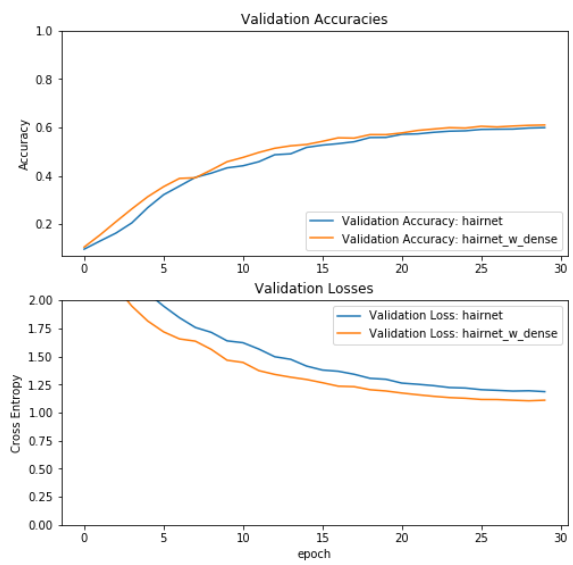

Figure 3: *Fully convolutional network vs. the same architecture with a single terminal dense layer.*

Furthermore, comparing Adam to stochastic gradient descent showed even more staggering results. Training the above network on the CIFAR-10 dataset with both SGD and Adam, we achieve the following results.

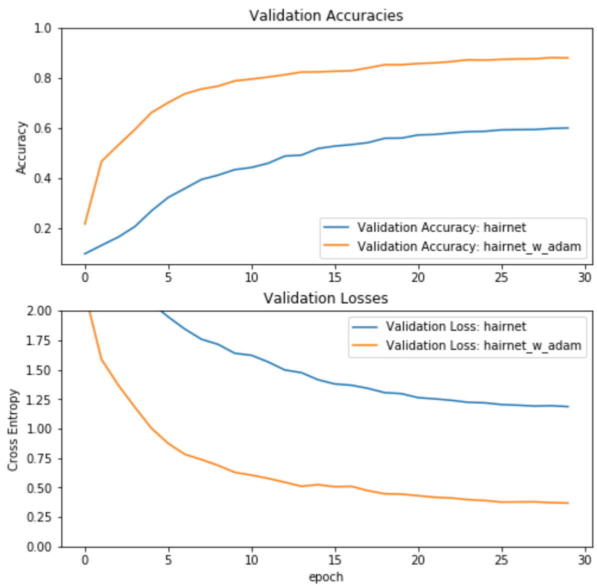

***Figure 4**: comparative results of Adam vs SGD on the CIFAR-10 dataset.*

5 Implementation

We have attached to the Appendix a chart detailing predicted performance for the HairNet architecture. Note that these predictions do not consider data movement/compute concurrency. This unoptimized estimation predicts that the HairNet architecture will achieve inference speeds of 0.177 secs / input image in an unbatched data stream.

We found that the slowest operations tended to be batch norm updates in the earlier layers of the network. This comes mainly from the computational complexity of normalizing large output feature maps.

Our predictions are based on an architecture that consists of an infinite external memory, 1 GB/s DDR bus for data movement between external memory and internal memory, 1 MB of internal memory, 1 TFLOPS of matrix compute and 10 GFLOPS of generic host compute

6 Conclusion

This paper is dedicated to improving the computational efficiency and performance of convolutional networks for the task of image recognition. We have introduced several design improvements that drastically decrease and increase speed of CNNs. Among these, we introduce depthwise separable convolutions as a factorized and efficient alternative to standard convolution. We also introduce batch normalization as a method to enable the training of deeper network. Global average pooling is an alternative to parameter dense, fully-connected layers at the end of a network. Lastly, we introduce Adam as a novel first order gradient descent optimizer that drastically speeds the training of deep neural networks.

Our proposed network, HairNet, achieves excellent results on the ImageNet dataset while maintaining 4% of the size of AlexNet. Further study should be done on how to efficiently apply these methods to other areas of computer vision.

References

Note: As stated above, this paper is a work of fiction. The following are the actual inventors of the ideas described in this paper.

LENET
[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel.
Backpropagation applied to handwritten zip code recognition. Neural Comput., 1(4):541–551,
December 1989.
IMAGENET
[2] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proc. CVPR, 2009.
AlexNet
[3] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106–1114, 2012.
Depthwise Separable Convolutions
[4] F. Chollet. Xception: Deep learning with depthwise separable convolutions. arXiv preprint arXiv:1610.02357v2, 2016.
[5] A. Howard et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision
Applications
arXiv preprint arXiv: 1704.04861, 2017
Batch Normalization
[6] S. Ioffe and C. Szegedy. Batch normalization: Accelerating
deep network training by reducing internal covariate shift.
arXiv preprint arXiv:1502.03167, 2015.

GAP

[7] M. Lin, Q. Chen, and S. Yan. Network in network. arXiv preprint arXiv: 1603.05027, 2014.

[8] B. Zhou, A. Khosla, L. A., A. Oliva, A. Torralba, Learning Deep Features for Discriminative Localization., CVPR.

Adam

[9] D. Kingma, and J. Ba Adam: A Method for Stochastic Optimization
arXiv preprint arXiv: 1412.6980, 2014.

Appendix

layer_name	in_map_(Mb)	out_map_(Mb)	param_(Mb)	ops	in_map_loc	out_map_loc	param_loc	transfer_time	compute_time	total_time
conv2D	0.602112	6.422528	0.003456	86704128	internal	external	external	0.006426	8.67E-05	0.006513
BN_RELU	6.422528	6.422528	0.000256	8028160	external	external	external	0.012845	0.008028	0.020873
max_pool	6.422528	1.605632	0	1605632	external	external	external	0.008028	0.001606	0.009634
dwise_conv0	1.605632	1.605632	0.001152	2.31E+08	external	external	external	0.003212	0.000231	0.003444
BN_RELU0	1.605632	1.605632	0.000256	2007040	external	external	external	0.003212	0.002007	0.005219
pointwise_conv0	1.605632	3.211264	0.008192	51380224	external	external	external	0.004825	5.14E-05	0.004876
BN_RELU0_2	3.211264	3.211264	0.000512	4014080	external	external	external	0.006423	0.004014	0.010437
dwise_conv1	3.211264	3.211264	0.002304	9.25E+08	external	external	external	0.006425	0.000925	0.00735
BN_RELU1	3.211264	3.211264	0.000512	4014080	external	external	external	0.006423	0.004014	0.010437
pointwise_conv1	3.211264	3.211264	0.016384	1.03E+08	external	external	external	0.006439	0.000103	0.006542
BN_RELU1_2	3.211264	3.211264	0.000512	4014080	external	external	external	0.006423	0.004014	0.010437
max_pool	3.211264	0.802816	0	802816	external	internal	external	0.003211	0.000803	0.004014
max_pool	3.211264	0.802816	0	802816	external	internal	external	0.003211	0.000803	0.004014
dwise_conv0	0.802816	0.802816	0.002304	2.31E+08	internal	internal	external	2.30E-06	0.000231	0.000234
BN_RELU0	0.802816	0.802816	0.000512	1003520	internal	internal	external	5.12E-07	0.001004	0.001004
pointwise_conv0	0.802816	1.605632	0.032768	51380224	internal	external	external	0.001638	5.14E-05	0.00169
BN_RELU0_2	1.605632	1.605632	0.001024	2007040	external	external	external	0.003212	0.002007	0.005219
dwise_conv1	1.605632	1.605632	0.004608	9.25E+08	external	external	external	0.003216	0.000925	0.004141
BN_RELU1	1.605632	1.605632	0.001024	2007040	external	external	external	0.003212	0.002007	0.005219
pointwise_conv1	1.605632	1.605632	0.065536	1.03E+08	external	external	external	0.003277	0.000103	0.00338
BN_RELU1_2	1.605632	1.605632	0.001024	2007040	external	external	external	0.003212	0.002007	0.005219
dwise_conv2	1.605632	1.605632	0.004608	9.25E+08	external	external	external	0.003216	0.000925	0.004141
BN_RELU2	1.605632	1.605632	0.001024	2007040	external	external	external	0.003212	0.002007	0.005219

pointwise_conv2	1.605632	1.605632	0.065536	1.03E+08	external	external	external	0.003277	0.000103	0.00338
BN_RELU2_2	1.605632	1.605632	0.001024	2007040	external	external	external	0.003212	0.002007	0.005219
max_pool	1.605632	0.401408	0	401408	external	internal	external	0.001606	0.000401	0.002007
max_pool	1.605632	0.401408	0	401408	external	internal	external	0.001606	0.000401	0.002007
dwise_conv0	0.401408	0.401408	0.004608	2.31E+08	internal	internal	external	4.61E-06	0.000231	0.000236
BN_RELU0	0.401408	0.401408	0.001024	501760	internal	internal	external	1.02E-06	0.000502	0.000503
pointwise_conv0	0.401408	0.802816	0.131072	51380224	internal	internal	external	0.000131	5.14E-05	0.000182
BN_RELU0_2	0.802816	0.802816	0.002048	1003520	internal	internal	external	2.05E-06	0.001004	0.001006
dwise_conv1	0.802816	0.802816	0.009216	9.25E+08	internal	internal	external	9.22E-06	0.000925	0.000934
BN_RELU1	0.802816	0.802816	0.002048	1003520	internal	internal	external	2.05E-06	0.001004	0.001006
pointwise_conv1	0.802816	0.802816	0.262144	1.03E+08	internal	internal	external	0.000262	0.000103	0.000365
BN_RELU1_2	0.802816	0.802816	0.002048	1003520	internal	internal	external	2.05E-06	0.001004	0.001006
dwise_conv2	0.802816	0.802816	0.009216	9.25E+08	internal	internal	external	9.22E-06	0.000925	0.000934
BN_RELU2	0.802816	0.802816	0.002048	1003520	internal	internal	external	2.05E-06	0.001004	0.001006
pointwise_conv2	0.802816	0.802816	0.262144	1.03E+08	internal	internal	external	0.000262	0.000103	0.000365
BN_RELU2_2	0.802816	0.802816	0.002048	1003520	internal	internal	external	2.05E-06	0.001004	0.001006
max_pool	0.802816	0.200704	0	200704	internal	internal	external	0	0.000201	0.000201
max_pool	0.802816	0.200704	0	200704	internal	internal	external	0	0.000201	0.000201
dwise_conv0	0.200704	0.200704	0.009216	2.31E+08	internal	internal	external	9.22E-06	0.000231	0.00024
BN_RELU0	0.200704	0.200704	0.002048	250880	internal	internal	external	2.05E-06	0.000251	0.000253
pointwise_conv0	0.200704	0.401408	0.524288	51380224	internal	internal	external	0.000524	5.14E-05	0.000576
BN_RELU0_2	0.401408	0.401408	0.004096	501760	internal	internal	external	4.10E-06	0.000502	0.000506
dwise_conv1	0.401408	0.401408	0.018432	9.25E+08	internal	internal	external	1.84E-05	0.000925	0.000943
BN_RELU1	0.401408	0.401408	0.004096	501760	internal	internal	external	4.10E-06	0.000502	0.000506
pointwise_conv1	0.401408	0.401408	1.048576	1.03E+08	internal	internal	external	0.001049	0.000103	0.001151

BN_RELU1_2	0.401408	0.401408	0.004096	501760	internal	internal	external	4.10E-06	0.000502	0.000506
dwise_conv2	0.401408	0.401408	0.018432	9.25E+08	internal	internal	external	1.84E-05	0.000925	0.000943
BN_RELU2	0.401408	0.401408	0.004096	501760	internal	internal	external	4.10E-06	0.000502	0.000506
pointwise_conv2	0.401408	0.401408	1.048576	1.03E+08	internal	internal	external	0.001049	0.000103	0.001151
BN_RELU2_2	0.401408	0.401408	0.004096	501760	internal	internal	external	4.10E-06	0.000502	0.000506
max_pool	0.401408	0.100352	0	100352	internal	internal	external	0	0.0001	0.0001
max_pool	0.401408	0.100352	0	100352	internal	internal	external	0	0.0001	0.0001
dwise_conv0	0.100352	0.100352	0.018432	2.31E+08	internal	internal	external	1.84E-05	0.000231	0.00025
BN_RELU0	0.100352	0.100352	0.004096	125440	internal	internal	external	4.10E-06	0.000125	0.00013
pointwise_conv0	0.100352	0.196	2.048	50176000	internal	internal	external	0.002048	5.02E-05	0.002098
BN_RELU0_2	0.196	0.196	0.008	245000	internal	internal	external	8.00E-06	0.000245	0.000253
dwise_conv1	0.196	0.196	0.036	8.82E+08	internal	internal	external	3.60E-05	0.000882	0.000918
BN_RELU1	0.196	0.196	0.008	245000	internal	internal	external	8.00E-06	0.000245	0.000253
pointwise_conv1	0.196	0.196	4	98000000	internal	internal	external	0.004	9.80E-05	0.004098
BN_RELU1_2	0.196	0.196	0.008	245000	internal	internal	external	8.00E-06	0.000245	0.000253
max_pool	0.196	0.049	0	49000	internal	internal	external	0	4.90E-05	4.90E-05
GAP	0.196	0.004	0	49000	internal	internal	external	0	4.90E-05	4.90E-05
SoftMax	0.004	0.004	0	3000	internal	internal	external	0	3.00E-06	3.00E-06