

3 Past work

Convolutional models and DBNs have been in use for years. In [5], the authors trained a convolutional DBN to recognize digits from the MNIST dataset. More directly related to this work, [1] trained a fully-connected two-layer DBN on the tiny images dataset and fine-tuned the model’s features with a logistic regression classifier. The author achieved a 65% accuracy on the CIFAR-10 test set. In [7], the authors train a “mcRBM” to achieve a 71.0% accuracy rate. In [6], the authors achieve 74.5% with a sparse-coding scheme. This is currently the best-published result.

4 Models

4.1 Overview of our methods

Briefly speaking, we train a convolutional DBN on the 1.6 million tiny images dataset, and then use it to initialize a convolutional neural net. We then fine-tune the weights of the convolutional net to classify images from the CIFAR-10 dataset using a logistic regression classifier at the output of the net. [4] is an excellent reference for training Restricted Boltzmann Machines (RBMs) in general.

To learn features from unlabeled color images in an unsupervised manner, we build upon the work of [1], in which the author trained a two-layer Gaussian-Bernoulli DBN on color images. However, the DBN in that work was fully connected to the image pixels, while we instead focus on convolutional models. In addition, we adopt the use of “ReLU”s (REctified Linear Units) in the hidden layer, following [8]. Our tests confirm that these units are in fact superior to Bernoulli units. Given input x , the output of a ReLU unit is

$$y = \max(x, 0).$$

Our ReLU units differ from those of [8] in two respects. First, we cap the units at 6, so our ReLU activation function is

$$y = \min(\max(x, 0), 6).$$

In our tests, this encourages the model to learn sparse features earlier. In the formulation of [8], this is equivalent to imagining that each ReLU unit consists of only 6 replicated bias-shifted Bernoulli units, rather than an infinite amount. We will refer to ReLU units capped at n as ReLU- n units.

Additionally, we employ a different noise model than that of [8]. [8] computes that in order to sample from a ReLU unit, one must add Gaussian noise with standard deviation

$$\frac{1}{1 + e^{-x}}$$

to the unit’s mean-field activity y . We instead add noise with standard deviation

$$\begin{cases} 0 & : \text{if } y = 0 \text{ or } y = 6 \\ 1 & : \text{if } 0 < y < 6 \end{cases}.$$

For values of y approaching 6, the two noise models are identical. But for values of y slightly greater than zero, our model adds significantly more noise. We believe this encourages the model to learn features that are almost always off, as features that are frequently on will incur a high variance (noise) penalty. Both of these modifications to ReLU units are similar in spirit to the various sparseness-inducing tricks commonly employed when training unsupervised models on natural images. They also happen to be extremely cheap computationally.

Although our model’s convolutional filters share weights, they do not share biases. This allows the model to learn filters that do not necessarily have to be useful at every part of the image. Thus, our model is governed by the following energy function:

$$\begin{aligned} E(\mathbf{v}, \mathbf{h}) &= \sum_{i=1}^V \frac{(v_i - b_i^v)^2}{2\sigma_i^2} - \sum_{j=1}^F \sum_{p=1}^P b_j^{f,p} h_j^p \\ &- \sum_{i=1}^V \frac{v_i}{\sigma_i} \sum_{j=1}^F \langle \mathbf{w}^j, \mathbf{h}^j \rangle_i \end{aligned}$$

where \mathbf{v}, \mathbf{h} are the visible and hidden unit activities, respectively, V is the number of visible units, F is the number of convolutional filters, b_i^v is the i th visible bias, σ_i is the standard deviation of the i th visible unit, P is the number of outputs that a filter produces when convolved with an image, $b_j^{f,p}$ is the bias of the j th filter at position p , h_j^p is the activity of the j th filter at position p , \mathbf{w}^j is the j th filter, \mathbf{h}^j is the grid of activities produced when \mathbf{w}^j is convolved with the image, and $\langle \mathbf{w}^j, \mathbf{h}^j \rangle_i$ is the i th output when \mathbf{h}^j is convolved with \mathbf{w}^j rotated by 180 degrees.

We use convolutional filters of size 9×9 , for several reasons. First, 9×9 is big enough to capture the scale of most edges and other salient features in the image. Second, 9×9 filters lead to a convolutional output grid of dimensions 24×24 , which can be easily divided into subsquares of 3×3 , 4×4 , or 6×6 , making local pooling simple. Third, odd-sized filters can lead to nicely centered features.

One of the first difficulties in training a convolutional RBM is dealing with the boundary pixels. Since these pixels contribute to far fewer hidden unit activities than do interior pixels, the convolutional RBM tends to have difficulty modeling them. Typically, the RBM gets around this by learning filters that are zero everywhere except near one of the four corners. That is, it will attempt to make every filter useful for modeling the boundaries (and especially the corners) of the image, and it will do so by learning corner filters. Since we do not wish our model to focus overtly on the boundaries and corners of images – the interior of the image is much more likely to contain useful hints about the class label – we investigate several methods for dealing (or not dealing) with this boundary problem. These are

1. Not dealing with the boundary in any special way.
2. Adding a border of zeros around the image. We use a padding of width 4 in order to encourage the RBM to learn centered filters and not to focus overtly on the boundary.
3. Using some number of globally-connected units in addition to the convolutional units. Ideally, the global units should learn to focus on the boundaries of the image, freeing the convolutional units from that task.
4. Combining 2 and 3.
5. Holding fixed the pixels of the image that are near the boundary – perhaps those that are half a filter-width away. This is, in a sense, of the converse of adding a border of zeros around the image. It is less computationally expensive than adding a border of zeros, and should also instruct the RBM to learn centered filters and not to focus overtly on the boundary, since the RBM will never have to reconstruct it.
6. Combining 3 and 5.

After training the convolutional RBM, we use it to generate features from images by computing the hidden unit activations for each image. In the case when we do not add any zero padding to the images, each filter produces a 24×24 output grid when applied to an image. We subsample this grid using 4×4 overlapping local-max regions, as illustrated in Figure 2. This reduces the feature dimensionality from $f \times 24 \times 24 = f \times 576$ to $f \times 24 \times 24 / 16 + f \times 20 \times 20 / 16 = f \times 61$, where f is the number of convolutional filters. The case with zero padding is analogous, except that the output grid that we subsample is 32×32 rather than 24×24 .

After subsampling, we obtain a feature vector with values in the range $[0, 6]$ (the range of the ReLU-6 unit activations). We wish to use the weights from the convolutional RBM to initialize a convolutional neural net, which will be trained to classify the images in the CIFAR-10 dataset. It turns out that when placed at the lower levels of the net, units with such a large linear range (0-6) encourage the net to simply memorize the training data rather than generalize to new examples. So we cap the ReLU activations at 1, to obtain a feature vector with values in the range $[0, 1]$. Fortunately, the features that the convolutional RBM learns are very sparse. Few of them ever have activations greater than 1. So we do not lose much resolution by capping the ReLU activations. Figure 3 shows a histogram of ReLU feature activations on a random subset of 100 images (after the cap at 1).

We train an ordinary, fully-connected RBM on these convolutional ReLU-1 features. This RBM has ReLU-1 visible units and ReLU-6 hidden units (the number of hidden units in this RBM is either the same or almost the same as the number of visibles). We opt not to train a convolutional RBM in the second layer because the “images” on which it would be trained are only 6×6 pixels.

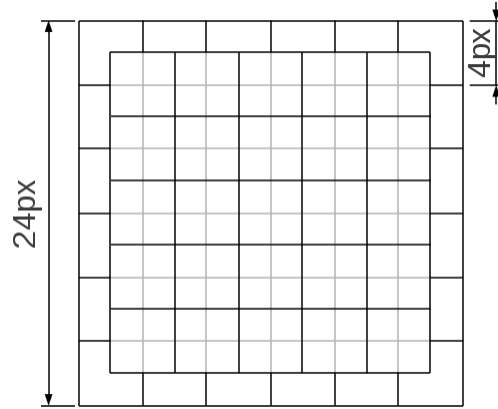


Figure 2: The local-pooling grid that we use. Each pictured 4×4 square is subsampled (using max) to one value.

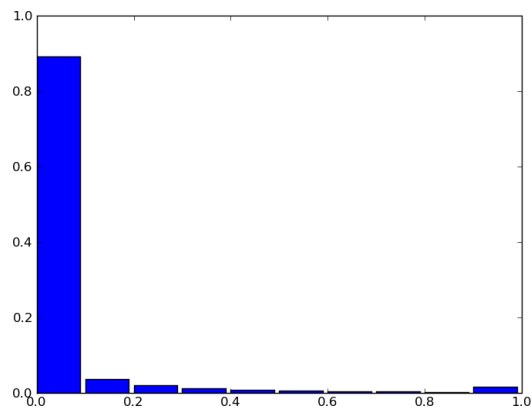


Figure 3: Histogram of mean-field ReLU feature activations on a random subset of 100 images, after capping the activations at 1. Notice how sparse the features are and how rare activations greater than 1 must have been prior to capping. 90% of the activations are in the range $[0, 0.1)$, and 79% of the activations are exactly zero.

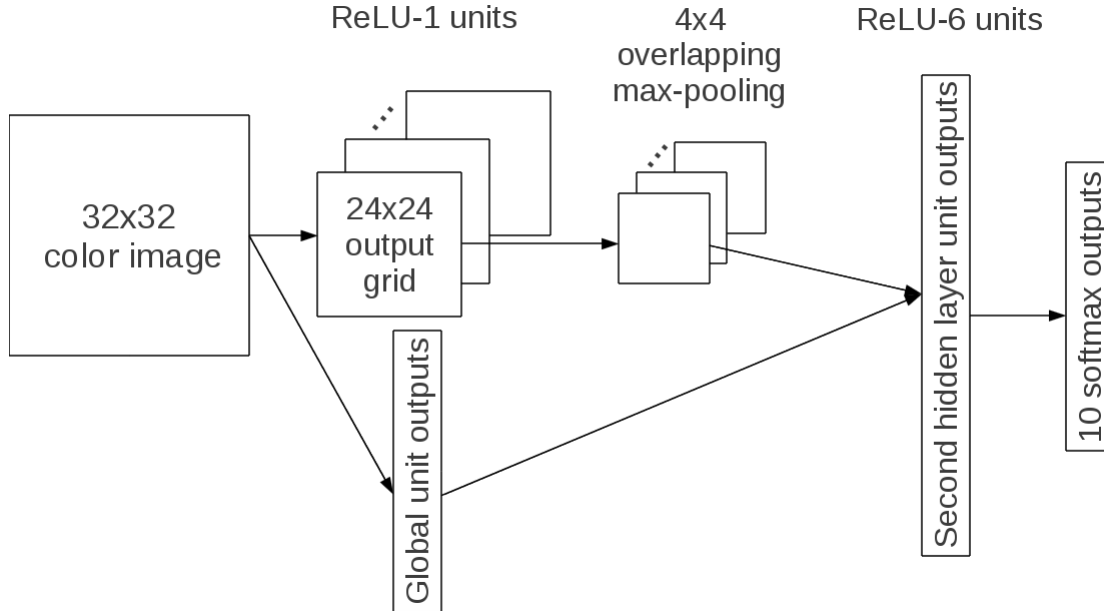


Figure 4: The architecture of our convolutional net. The case with globally-connected units is shown here. The overlapping max-pooling operation is as illustrated in Figure 2. The classifier at the output of the net is multinomial logistic regression.

After training the second-layer RBM, we initialize a two-layer convolutional net with the weights from the two RBMs and fine-tune it to classify the images in the CIFAR-10 dataset, using multinomial logistic regression at the output. Figure 4 illustrates the architecture of the net in the case when we also have globally-connected units.

The most computationally-intensive networks that we describe here take 45 hours to pre-train and 36 hours to fine-tune on an Nvidia GTX 280 GPU. By far, most of the time is spent squeezing the last few fractions of a percent from the nets.

4.2 Details of learning

4.2.1 Dataset

We pre-process the 1.6 million tiny images dataset by deleting the 1000 least-significant principal components. The pre-processed images look identical to the original images to the naked eye. We also subtract the mean and set the average standard deviation over all dimensions to 1.

4.2.2 Pre-training

We initialize all weights in both layers from a normal distribution with mean zero and standard deviation 10^{-3} , and we initialize all biases at zero. We train both RBMs using CD-1 [3]; the first for 45 epochs² and the second for 30. We do not add any noise when reconstructing the pixels from the hidden activities in either RBM. We use a mini-batch size of 128³. We also use a momentum of 0.9 on the weights and biases for both RBMs. Thus, the weight update for mini-batch i is the learning rate⁴ times the average gradient for mini-batch i plus 0.9 times the previous weight update.

For the first-layer convolutional RBM, we use a learning rate of $5 \cdot 10^{-6}$ on the visible-convolutional weights, $5 \cdot 10^{-5}$ on the visible and hidden unit biases, 10^{-3} on the visible-global weights (if any), and 10^{-2}

²An epoch is one cycle through the training data.

³128 happens to be a good number to use on Nvidia GPUs because it leads to matrices whose dimensions are divisible by 64. Multiplication of such matrices is about 50% faster.

⁴The learning rate is the multiplier of the *average* gradient for a mini-batch of images.

Parameter	Learning rate	Momentum	Weight decay coefficient
Visible-convolutional weights	10^{-4}	0.9	0
Visible-global weights	$2.5 \cdot 10^{-6}$	0	0
Second-layer weights	$2 \cdot 10^{-4}$	0.9	0
Output weights	10^{-3}	0.9	$8 \cdot 10^{-3}$
Convolutional unit biases	10^{-4}	0.9	0
Global unit biases	$2 \cdot 10^{-4}$	0.9	0
Second-layer unit biases	$2 \cdot 10^{-4}$	0.9	0
Output unit biases	10^{-3}	0.9	0

Table 1: The learning rates, momenta, and weight decay coefficients that we use when training our two-layer convolutional nets.

on the global unit biases (if any). We fix the standard deviations of the Gaussian visible units at $\sigma = 1$.

For the second-layer RBM, we use a learning rate of 10^{-3} on the weights and 10^{-2} on the visible and hidden biases.

4.2.3 Classification

Table 1 lists the learning rates, weight decay coefficients⁵, and momenta that we use when training our two-layer convolutional nets. Most of the precise values of these of these hyperparameters are not terribly important, but we found these to work well on a validation set. The one important value is the visible-global unit learning rate. We found that setting this value too high caused the model to simply memorize the training data quickly. This probably happens because it is easier for the model to fine-tune an un-shared weight to memorize training cases than it is to do so for a weight that must be shared over 24×24 image regions.

We found it to be detrimental to use weight decay on the weights that were learned by RBMs, so we only use it on the output weights, which we initialize from a normal distribution with zero mean and standard deviation 10^{-3} .

When using a logistic regression classifier, it is often the case that the accuracy on test data continues to improve even after the log probability of the test data has peaked. So a choice has to be made as to when to stop training. Our stopping criterion is this: we find the log prob x of the training data when the classification accuracy on the validation set peaks. We then train our logistic regression classifier on the entire training set and stop when the training log prob reaches x .

5 Experiments

5.1 Pre-training

Figure 5 shows the 9×9 local and 32×32 global filters learned by the models that we trained. Several obvious patterns can be extracted from the pictures.

- Broadly speaking, the models with global units (models 3, 4, and 6) produce local filters that are more separated (“edge-like”) and have fewer artifacts.
- Adding a zero-padding around the images (models 2, 4, 5, and 6) produces local filters with centered, rather than cornered, receptive fields.
- When using global units and reconstructing the entire image (that is, when keeping no boundary pixels fixed), the global units tend to learn low-frequency (or *global*) filters as well as high-frequency filters near the image edges (models 3 and 4).

The global units of model 6, which holds fixed a five-pixel border of the image, appear to learn merely to recognize the boundary between pixels that are kept fixed and those that are not.

⁵When using weight decay, we add to the average gradient the value $\epsilon w v$, where ϵ is the learning rate, w is the weight decay coefficient, and v is the current value of the weight that we are updating.


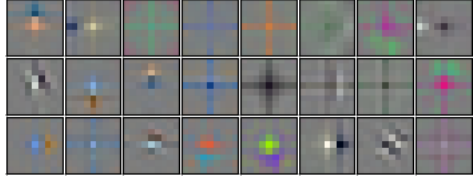
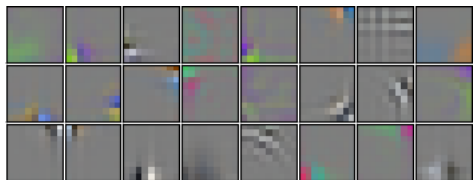
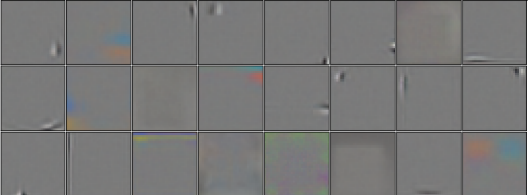
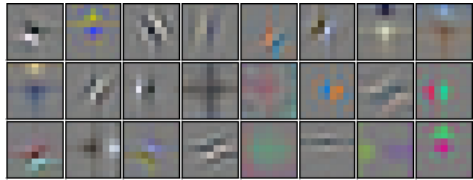
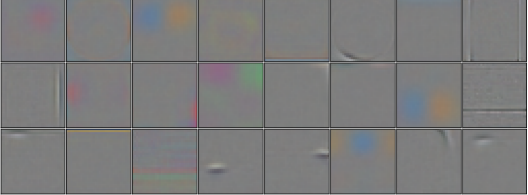
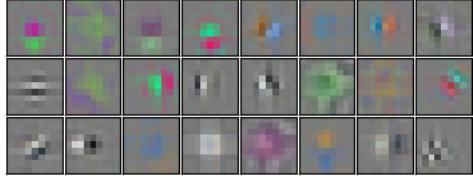
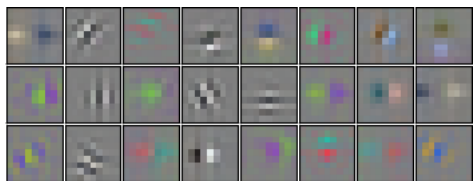

Model	Local units	Global units
1		N/A
2		N/A
3		
4		
5		N/A
6		

Figure 5: Random samples of 9×9 local and 32×32 global filters (if applicable) learned by the models described in Section 4.1. All models had $64 \ 9 \times 9$ local units and either zero or 1024 global units.

Model	Two-layer performance
3	77.46%
4	78.90%
5	77.56%
6	77.27%

Number of layers	Model 4 performance
1	70.67%
1 + pooling	75.12%
2	78.90%

Table 2: Classification accuracy on the CIFAR-10 test set achieved by some of the models described in Section 4.1.

(a) the two-layer classification accuracy achieved by models 3, 4, 5, and 6.

(b) the advantage of adding layers to the best model, model 4.

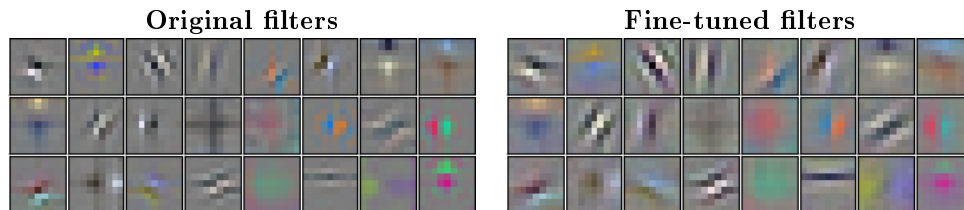


Figure 6: A random sample of 24 of the $64 \ 9 \times 9$ filters learned by model 4, our best. On the left are the filters learned by the convolutional RBM, reprinted from Figure 5. On the right are same filters after being fine-tuned by the two-layer net as described in Section 4.2.3.

5.2 Classification

Using the intuition that the filters of models 3, 4, 5, and 6 appear most promising, we train a second-layer fully-connected RBM over the features extracted by these models, as described in the previous section. Table 2 (a) summarizes the two-layer classification performance of these models. Table 2 (b) demonstrates the necessity of using a deep architecture in order to achieve the best performance. Our best model is model 4, which deals with the edge pixels by using global units and padding the images with a four-pixel border of zeros. This model classifies the CIFAR-10 test set with 78.9% accuracy. It has $64 \ 9 \times 9$ local units and 1024 global units in the first layer. After the pooling phase, the model produces 8256 hidden activities, which are connected to 9024 hidden units in the layer above.

Figure 6 compares the 9×9 local filters of model 4 before and after being fine-tuned to classify the CIFAR-10 dataset. Notice how the fine-tuning phase enhances the appearance of the filters.

6 Acknowledgements

I'd like to thank Geoffrey Hinton for helpful suggestions.

References

- [1] A. Krizhevsky, 2009, "Learning multiple layers of features from tiny images", Master's Thesis, Department of Computer Science, University of Toronto
- [2] A. Torralba and R. Fergus and W. T. Freeman, "80 Million Tiny Images: a Large Database for Non-Parametric Object and Scene Recognition", IEEE PAMI, 2008
- [3] G. E. Hinton, "Training products of experts by minimizing contrastive divergence", Neural Computation, 2002, Vol. 14, pages 1771-1800
- [4] G. E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines", Version 1, UTML TR 2010-003, Department of Computer Science, University of Toronto, August 2010

- [5] H. Lee, R. Grosse, R. Ranganath, A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representation", Proceedings of the Twentieth-Sixth International Conference on Machine Learning (ICML), 2009
- [6] K. Yu, T. Zhang, "Improved Local Coordinate Coding using Local Tangents", Proceedings of the Twentieth-Seventh International Conference on Machine Learning (ICML), 2010
- [7] M. Ranzato, G. E. Hinton, "Modeling Pixel Means and Covariances Using Factorized Third-Order Boltzmann Machines", Proceedings of Computer Vision and Pattern Recognition Conference (CVPR), 2010
- [8] V. Nair, G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines", Proceedings of the Twentieth-Seventh International Conference on Machine Learning (ICML), 2010