# Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition

Marc'Aurelio Ranzato, Fu-Jie Huang, Y-Lan Boureau, Yann LeCun
Courant Institute of Mathematical Sciences, New York University, New York, NY, USA
{ranzato,jhuangfu,ylan,yann}@cs.nyu.edu, http://www.cs.nyu.edu/~yann

## Abstract

*We present an unsupervised method for learning a hierarchy of sparse feature detectors that are invariant to small shifts and distortions. The resulting feature extractor consists of multiple convolution filters, followed by a pointwise sigmoid non-linearity, and a feature-pooling layer that computes the max of each filter output within adjacent windows. A second level of larger and more invariant features is obtained by training the same algorithm on patches of features from the first level. Training a supervised classifier on these features yields 0.64% error on MNIST, and 54% average recognition rate on Caltech 101 with 30 training samples per category. While the resulting architecture is similar to convolutional networks, the layer-wise unsupervised training procedure alleviates the over-parameterization problems that plague purely supervised learning procedures, and yields good performance with very few labeled training samples.*

## 1. Introduction

The use of unsupervised learning methods for building feature extractors has a long and successful history in pattern recognition and computer vision. Classical methods for dimensionality reduction or clustering, such as Principal Component Analysis and K-Means, have been used routinely in numerous vision applications [15, 16].

In the context of object recognition, a particularly interesting and challenging question is whether unsupervised learning can be used to learn *invariant features*. The ability to learn robust invariant representations from a limited amount of labeled data is a crucial step towards building a solution to the object recognition problem. In this paper, we propose an unsupervised learning method for learning *hierarchies of feature extractors that are invariant to small distortions*. Each level in the hierarchy is composed of two layers: (1) a bank of local filters that are convolved with the input, and (2) a pooling/subsampling layer in which each unit computes the maximum value within a small neighborhood of each filter's output map, followed by a pointwise non-linearity (a sigmoid function). When multiple such levels are stacked, the resulting architecture is essentially identical to the Neocognitron [7], the Convolutional Network [13, 10], and the HMAX, or so-called "Standard Model" architecture [20, 17]. All of those models use alternating layers of convolutional feature detectors (reminiscent of Hubel and Wiesel's *simple cells*), and local pooling and subsampling of feature maps using a max or an averaging operation (reminiscent of Hubel and Wiesel's *complex cells*). A final layer trained in supervised mode performs the classification. We will call this general architecture the *multi-stage Hubel-Wiesel architecture*. In the Neocognitron, the feature extractors are learned with a rather ad-hoc unsupervised competitive learning method. In [20, 17], the first layer is hard-wired with Gabor filters, and the second layer is trained by feeding natural images to the first layer, and simply storing its outputs as templates. In Convolutional Networks [13, 10], all the filters are learned with a supervised gradient-based algorithm. This global optimization process can achieve high accuracy on large datasets such as MNIST with a relatively small number of features and filters. However, because of the large number of trainable parameters, Convolutional Networks seem to require a large number of examples per class for training. Training the lower layers with an unsupervised method may help reduce the necessary number of training samples. Several recent works have shown the advantages (in terms of speed and accuracy) of pre-training each layer of a deep network in unsupervised mode, before tuning the whole system with a gradient-based algorithm [9, 3, 19]. The present work is inspired by these methods, but incorporates invariance at its core. Our main motivation is to arrive at a well-principled method for unsupervised training of invariant feature hierarchies. Once high-level invariant features have been trained with unlabeled data, a classifier can use these features to classify images through supervised training on a small number of samples.

Currently, the main way to build invariant representations is to compute local or global histograms (or bags) of sparse, hand-crafted features. These features generally have invariant properties themselves. This includes SIFT [14] features and their many derivatives, such as affine-invariant

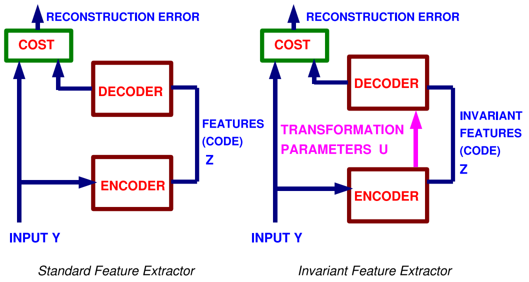



Figure 1. Left: generic architecture of encoder-decoder paradigm for unsupervised feature learning. Right: architecture for shift-invariant unsupervised feature learning. The feature vector $Z$ indicates *what* feature is present in the input, while the *transformation parameters* $U$ indicate *where* each feature is present in the input.

patches [11]. However, *learning* the features may open the door to more robust methods with a wider spectrum of applications. In most existing unsupervised feature learning methods, invariance appears as an afterthought. For example, in [20, 17, 19], the features are learned without regard to invariance. The invariance comes from the feature pooling (complex cell) layer, which is added after the training phase is complete. Here, we propose to integrate the feature pooling within the unsupervised learning architecture.

Many unsupervised feature learning methods are based on the encoder-decoder architecture depicted in fig. 1. The input (an image patch) is fed to the encoder which produces a feature vector (a.k.a a code). The decoder module then reconstructs the input from the feature vector, and the reconstruction error is measured. The encoder and decoder are parameterized functions that are trained to minimize the average reconstruction error. In most algorithms, the code vector must satisfy certain constraints. With PCA, the dimension of the code must be smaller than that of the input. With K-means, the code is the index of the closest prototype. With Restricted Boltzmann Machines [9], the code elements are stochastic binary variables. In the method proposed here, the code will be forced to be *sparse*, with only a few components being non-zero at any one time.

The key idea to invariant feature learning is to represent an input patch with two components: The *invariant feature vector*, which represents *what* is in the image, and *the transformation parameters* which encodes *where* each feature appears in the image. They may contain the precise locations (or other instantiation parameters) of the features that compose the input. The invariant feature vector and the transformation parameters are both produced by the encoder. Together, they contain all the information necessary for the decoder to reconstruct the input.

## 2. Architecture for Invariant Feature Learning

We now describe a specific architecture for learning shift-invariant features. Sections 3 and 4 will discuss how the model can be trained to produce features that are not only invariant, but also sparse. An image patch can be modeled as a collection of features placed at particular locations within the patch. A patch can be reconstructed from the list of features that are present in the patch together with their respective locations. In the simplest case, the features are templates (or basis functions) that are combined additively to reconstruct a patch. If we assume that each feature can appear at most once within a patch, then computing a shift-invariant representation can come down to applying each feature detector at all locations in the patch, and recording the location where the response is the largest. Hence the invariant feature vector records the presence or absence of each feature in the patch, while the transformation parameters record the location at which each feature output is the largest. In general, the feature outputs need not be binary.

The overall architecture is shown in fig. 2(d). Before describing the learning algorithm, we show how a trained system operates using a toy example as an illustration. Each input sample is a binary image containing two intersecting bars of equal length, as shown in fig. 2(a). Each bar is 7 pixels long, has 1 of 4 possible orientations, and is placed at one of 25 random locations ($5\times5$) at the center of a $17\times17$ image frame. The input image is passed through 4 convolutional filters of size $7\times7$ pixels. The convolution of each detector with the input produces an $11\times11$ feature map. The *max-pooling* layer finds the largest value in each feature map, recording the position of this value as the *transformation parameter* for that feature map. The invariant feature vector collects these max values, recording the presence or absence of each feature independently of its position. No matter where the two bars appear in the input image, the result of the *max-pooling* operation will be identical for two images containing bars of identical orientations at different locations. The reconstructed patch is computed by placing each code value at the proper location in the decoder feature map, using the transformation parameters obtained in the encoder, and setting all other values in the feature maps to zero. The reconstruction is simply the sum of the decoder basis functions (which are essentially identical to the corresponding filters in the encoder) weighted by the feature map values at all locations.

A solution to this toy experiment is one in which the invariant representation encodes the information about which orientations are present, while the transformation parameters encode where the two bars appear in the image. The oriented bar detector filters shown in the figure are in fact the ones discovered by the learning algorithm described in the next section. In general, this architecture is not limited to binary images, and can be used to compute shift invariant features with any number of components.

## 3. Learning Algorithm

The encoder is given by two functions $Z = \mathrm{Enc}_Z(Y; W_C)$ and $U = \mathrm{Enc}_U(Y; W_C)$ where $Y$ is the

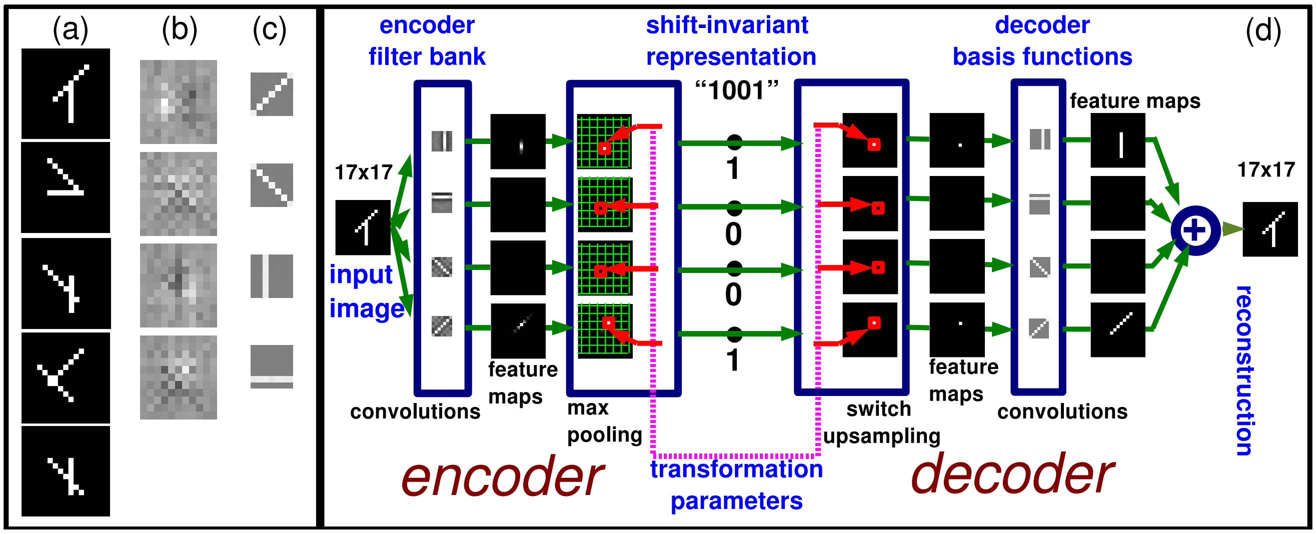



Figure 2. Left Panel: (a) sample images from the "two bars" dataset. Each sample contains two intersecting segments at random orientations and random positions. (b) Non-invariant features learned by an auto-encoder with 4 hidden units. (c) Shift-invariant decoder filters learned by the proposed algorithm. The algorithm finds the most natural solution to the problem. Right Panel (d): architecture of the shift-invariant unsupervised feature extractor applied to the two bars dataset. The encoder convolves the input image with a filter bank and computes the max across each feature map to produce the invariant representation. The decoder produces a reconstruction by taking the invariant feature vector (the "what"), and the transformation parameters (the "where"). Reconstructions is achieved by adding each decoder basis function (identical to encoder filters) at the position indicated by the transformation parameters, and weighted by the corresponding feature component.

input image, $W_C$ is the trainable parameter vector of the encoder (the filters), $Z$ is the invariant feature vector, and $U$ is the transformation parameter vector. Similarly, the decoder is a function $\text{Dec}(Z, U; W_D)$ where $W_D$ is the trainable parameter vector of the decoder (the basis functions). The reconstruction error $E_D$, also called the *decoder energy* measures the Euclidean distance between the input $Y$ and its reconstruction $E_D = ||Y - \text{Dec}(Z, U; W_D)||^2$. The *learning architecture* is slightly different from the ones in figs. 1 and 2(d): the output of the encoder is not directly fed to the decoder, but rather is fed to a cost module that measures the code prediction error, also called the *encoder energy*: $E_C = ||Z - \text{Enc}(Y, U; W_C)||^2$. Learning proceeds in an EM-like fashion in which $Z$ plays the role of auxiliary variable. For each input, we seek the value $Z^*$ that minimizes $E_D + \alpha E_C$ where $\alpha$ is a positive constant. In all the experiments we present in this paper $\alpha$ is set to 1. In other words, we search for a code that minimizes the reconstruction error, while being not too different from the encoder output. We describe an *on-line* learning algorithm to learn $W_C$ and $W_D$ consisting of four main steps:
**1.** propagate the input $Y$ through the encoder to produce the predicted code $Z_0 = \text{Enc}(Y, U; W_C)$ and the transformation parameters $U$ that are then copied into the decoder.
**2.** keeping $U$ fixed, and using $Z_0$ as initial value for the code $Z$, minimize the energy $E_D + \alpha E_c$ with respect to the code $Z$ by gradient descent to produce the optimal code $Z^*$.
**3.** update the weights in the decoder by one step of gradient descent so as to minimize the decoder energy: $\Delta W_D \propto -\partial ||Y - \text{Dec}(Z^*, U; W_D)||^2 / \partial W_D$.
**4.** update the weights in the encoder by one step of gradient descent so as to minimize the encoder energy (using the optimal code $Z^*$ as target value): $\Delta W_C \propto -\partial ||Z^* - \text{Enc}(Y, U; W_C)||^2 / \partial W_C$.

The decoder is trained to produce good reconstructions of input images from optimal codes $Z^*$ and, at the same time, the encoder is trained to give good predictions of these optimal codes. As training proceeds, fewer and fewer iterations are required to get to $Z^*$. After training, a single pass through the encoder gives a good approximation of the optimal code $Z^*$ and minimization in code space is not necessary. Other basis function models [18] that do not have an encoder module are forced to perform an expensive optimization in order to do inference (to find the code) even after learning the parameters. Note that this *general learning algorithm* is suitable for any encoder-decoder architecture, and not specific to a particular kind of feature or architecture choice. Any differentiable module can be used as encoder or decoder. In particular, we can plug in the encoder and decoder described in the previous section and learn filters that produce shift invariant representations.

We tested the proposed architecture and learning algorithm on the "two bars" toy example described in the previous section. In the experiments, both the encoder and the decoder are linear functions of the parameters (linear filters and linear basis functions), However, the algorithm is not restricted to linear encoders and decoders. The input images are $17 \times 17$ binary images containing two bars in different orientations: horizontal, vertical and the two diagonals as shown in fig. 2(a). The encoder contains four $7 \times 7$ linear filters, plus four $11 \times 11$ max-pooling units. The decoder contains four $7 \times 7$ linear basis functions. The parameters are randomly initialized. The learned basis functions are shown in fig. 2(c), and the encoder filters in fig. 2(d). After training on a few thousand images, the filters converge as expected to the oriented bar detectors shown in the figure. The resulting 4-dimensional representation extracted from the input image is translation invariant. These filters and the corresponding representation differ strikingly from what can be achieved by PCA or an auto-encoder neural network. For comparison, an auto-encoder neural network with 4 hidden units was trained on the same data. The filters (weights of the hidden units) are shown in fig. 2(b). There is no appearance of oriented bar detectors, and the resulting codes are not shift invariant.

## 4. Sparse, Invariant Features

There are well-known advantages to using sparse, overcomplete features in vision: robustness to noise, good tiling of the joint space of frequency and location, and good class separation for subsequent classification [5, 18, 19]. More importantly, when the dimension of the code in an encoder-decoder architecture is larger than the input, it is necessary to limit the amount of information carried by the code, lest the encoder-decoder may simply learn the identity function in a trivial way and produce uninteresting features. One way to limit the information content of an overcomplete code is to make it sparse. Following [19], the code is made sparse by inserting a *sparsifying logistic* non-linearity between the encoder and the decoder. The learning algorithm is left unchanged. The sparsifying logistic module transforms the input code vector into a sparse code vector with positive components between $[0, 1]$. It is a sigmoid function with a large adaptive threshold which is automatically adjusted so that each code unit is only turned on for a small proportion of the training samples. Let us consider the $k$-th training sample and the $i$-th component of the code, $z_i(k)$ with $i \in [1..m]$ where $m$ is the number of components in the code vector. Let $\bar{z}_i(k)$ be its corresponding output after the sparsifying logistic. Given two parameters $\eta \in [0, 1]$ and $\beta > 0$, the transformation performed by this non-linearity is given by:

$$\bar{z}_i(k) = \frac{e^{\beta z_i(k)}}{\zeta_i(k)}, \text{ with } \zeta_i(k) = e^{\beta z_i(k)} + \frac{(1-\eta)}{\eta}\zeta_i(k-1) \quad (1)$$

This can be seem as a kind of weighted "softmax" function over past values of the code unit. By unrolling the recursive expression of the denominator in eq. (1), we can express it as a sum of past values of $e^{\beta z_i(n)}$ with exponentially decaying weights. This adaptive logistic can output a large value, i.e. a value close to 1, only if the unit has undergone a long enough quiescent period. The parameter $\eta$ controls the sparseness of the code by determining the length of the time window over which samples are summed up. $\beta$ controls the gain of the logistic function, with large values yielding quasi-binary outputs. After training is complete, the running averages $\zeta_i(k)$ are kept constant, and set to the average of its last 1,000 values during training. With a fixed $\zeta_i(k)$, the non-linearity turns into a logistic function with a large threshold equal to $\log(\zeta_i(k-1)(1-\eta)/\eta)$.

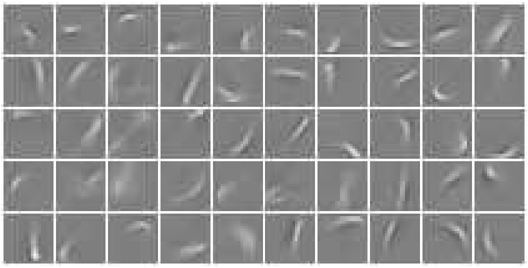


Figure 3. Fifty $20 \times 20$ filters learned in the decoder by the sparse and shift invariant learning algorithm after training on the MNIST dataset of $28 \times 28$ digits. A digit is reconstructed as linear combination of a small subset of these features positioned at one of 81 possible locations ($9 \times 9$), as determined by the transformation parameters produced by the encoder.

A sparse and shift-invariant feature extractor using the sparsifying logistic above is composed of: **(1.)** an encoder which convolves the input image with a filter bank and selects the largest value in each feature map, **(2.)** a decoder which first transforms the code vector into a sparse and positive code vector by means of the sparsifying logistic, and then computes a reconstruction from the sparse code using an additive linear combination of its basis functions and the information given by the transformation parameters.

Learning the filters in both encoder and decoder is achieved by the iterative algorithm described in sec. 3. In fig. 3 we show an example of sparse and shift invariant features. The model and the learning algorithm were applied to the handwritten digits from the MNIST dataset [1], which of consist of quasi-binary of size $28 \times 28$. We considered a set of fifty $20 \times 20$ filters in both encoder and decoder that are applied to the input at 81 locations ($9 \times 9$ grid), over which the max-pooling is performed. Hence image features can move over those 81 positions while leaving the invariant feature vector unchanged. The sparsifying logistic parameters settings $\eta = 0.015$ and $\beta = 1.5$ yielded sparse feature vectors. Because they must be sparse, the learned features (shown in fig. 3) look like part detectors. Each digit can be expressed as a linear combination of a small number of these 50 parts, placed at one of 81 locations in the image frame. Unlike with the non-invariant method described in [19], no two filters are shifted versions of each other.

## 5. Learning Feature Hierarchies

Once trained, the filters produced by the above algorithm can be applied to large images (of size $p \times q$). The max pooling operation is then performed over $M \times M$ neighborhoods. Assuming that these pooling windows do not overlap, the output is a set of feature maps of size $p/M \times q/M$. This output is invariant to shifts within the $M \times M$ max pooling windows. We can extract local patches from these locally-invariant multidimensional feature maps and feed them to another instance of the same unsupervised learning algorithm. This second level in the feature hierarchy will generate representations that are even more shift and distortion invariant because a max-pooling over $N \times N$ windows at the second level corresponds to an invariance over an $NM \times NM$ window in the input space. The second-level features will combine several first-level feature maps into each output feature map according to a predefined connectivity table. The invariant representations produced by the second level will contain more complex features than the first level.

Each level is trained in sequence, starting from the bottom. This layer-by-layer training is similar to the one proposed by Hinton et al. [9] for training deep belief nets. Their motivation was to improve the performance of deep multi-layer network trained in supervised mode by pre-training each layer unsupervised.

Our experiments also suggest that training the bottom layers unsupervised significantly improves the performance of the multi-layer classifier when few labeled examples are available. Unsupervised training can make use of large amount of unlabeled data and help the system extract informative features that can be more easily classified. Training the parameters of a deep network with supervised gradient descent starting from random initial values by does not work well with small training datasets because the system tends to overfit.

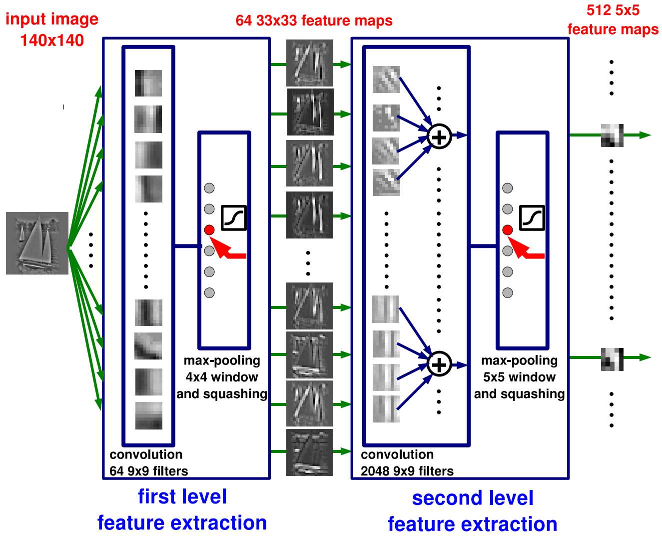



Figure 4. Example of the computational steps involved in the generation of two 5×5 shift-invariant feature maps from a pre-processed image in the Caltech101 dataset. Filters and feature maps are those actually produced by our algorithm.

## 6. Experiments

We used the proposed algorithm to learn two-level hierarchies of local features from two different datasets of images: the MNIST set of handwritten digits and the Caltech-101 set of object categories [6]. In order to test the representational power of the second-level features, we used them as input to two classifiers: a two-layer fully connected neural network, and a Gaussian-kernel SVM. In both cases, the feature extractor after training is composed of two stacked modules, each with a convolutional layer followed by a max-pooling layer. It would be possible to stack as many such modules as needed in order to get higher-level representations. Fig. 4 shows the steps involved in the computation of two output feature maps from an image taken from the Caltech101 dataset. The filters shown were among those learned, and the feature maps were computed by feed-forward propagation of the image through the feature extractor.

The layer-by-layer unsupervised training is conducted as follows. First, we learn the filters in the convolutional layer with the sparsifying encoder-decoder model described in sec. 3 trained on patches randomly extracted from training images. Once training is complete, the encoder and decoder filters are frozen, and the sparsifying logistic is replaced by a tanh sigmoid function with a trainable bias and a gain coefficient. The bias and the gain are trained with a few iterations of back-propagation through the encoder-decoder system. The rationale for relaxing the sparsity constraint is to produce representation with a richer information content. While the the sparsifying logistic drives the system to produce good filters, the quasi-binary codes it produces does not carry enough information for classification purpose. This substitution is similar to the one advocated in [9] in which the stochastic binary units used during the unsupervised training phase are replaced by continuous sigmoid units after the filters are learned. After this second unsupervised training, the encoder filters are placed in the corresponding feed-forward convolution/pooling layer pair, and are followed by the tanh sigmoid with the trained bias and gain (see fig. 4). Training images are run through this level to generate patches for the next level in the hierarchy. We emphasize that in the second level feature extractor each feature combines multiple feature maps from the previous level.

### 6.1. MNIST

We constructed a deep network and trained it on subsets of various sizes from the MNIST dataset, with three different learning procedures. In all cases the feature extraction

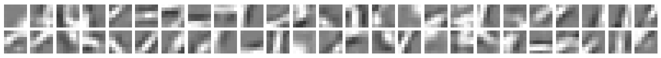
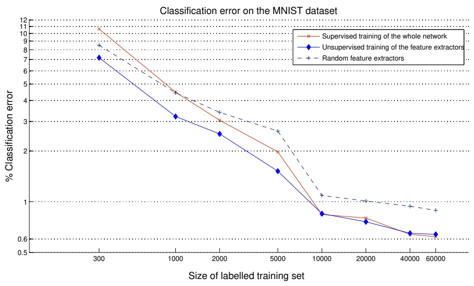


Figure 5. Fifty 7×7 sparse shift-invariant features learned by the unsupervised learning algorithm on the MNIST dataset. These filters are used in the first convolutional layer of the feature extractor.




| Labeled training samples | Unsupervised training for bottom layers, supervised training for top layers | Supervised training from random initial conditions | Random bottom layers, supervised training for top layers |
|---|---|---|---|
| 60,000 | 0.64 | 0.62 | 0.89 |
| 40,000 | 0.65 | 0.64 | 0.94 |
| 20,000 | 0.76 | 0.80 | 1.01 |
| 10,000 | 0.85 | 0.84 | 1.09 |
| 5,000 | 1.52 | 1.98 | 2.63 |
| 2,000 | 2.53 | 3.05 | 3.40 |
| 1,000 | 3.21 | 4.48 | 4.44 |
| 300 | 7.18 | 10.63 | 8.51 |

Figure 6. Classification error on the MNIST test set (%) when training on various size subsets of the labeled training set. With large labeled sets, the error rate is the same whether the bottom layers are learned unsupervised or supervised. The network with random filters at bottom levels performs surprisingly well (under 1% classification error with 40K and 60K training samples). With smaller labeled sets, the error rate is lower when the bottom layers have been trained unsupervised, while pure supervised learning of the whole network is plagued by over-parameterization; however, despite the large size of the network the effect of over-fitting is surprisingly limited.

is performed by the four bottom layers (two levels of convolution/pooling). The input is a 34×34 image obtained by evenly padding the 28×28 original image with zeros. The first layer is a convolutional layer with fifty 7×7 filters, which produces 50 feature maps of size 28×28. The second layer performs a max-pooling over 2×2 neighborhoods and outputs 50 feature maps of size 14×14 (hence the unsupervised training is performed on $8 \times 8$ input patches with $2 \times 2$ pooling). The third layer is a convolutional layer with 1,280 filters of size 5×5, that connect the subsets of the 50 layer-two feature maps to the 128 layer-three maps of size 10×10. Each layer-three feature map is connected to 10 layer-two feature maps according to a fixed, randomized connectivity table. The fourth layer performs a max-pooling over 2×2 neighborhoods and outputs 128 feature maps of size 5×5. The layer-four representation has $128 \times 5 \times 5 = 3,200$ components that are fed to a two-layer neural net with 200 hidden units, and 10 input units (one per class). There is a total of about $10^5$ trainable parameters in this network.

The **first training procedure** trains the four bottom layers of the network unsupervised over the whole MNIST dataset, following the method presented in the previous sections. In particular the first level module was learned using 100,000 8×8 patches extracted from the whole training dataset (see fig.5), while the second level module was trained on 100,000 50×6×6 patches produced by the first level extractor. The second-level features are receptive fields of size 18×18 when backprojected on the input. In both cases, these are the smallest patches that can be reconstructed from the convolutional and max-pooling layers. Nothing prevents us from using larger patches if so desired. The top two layers are then trained supervised with features extracted from the labeled training subset. The **second training procedure** initializes the whole network randomly, and trains supervised the parameters in all layers using the labeled samples in the subset. The **third training procedure** randomly initializes the parameters in both levels of the feature extractor, and only trains (in supervised mode) the top two layers on the samples in the current labeled subset, using the features generated by the feature extractor *with random filters*.

For the supervised portion of the training, we used labeled subsets of various sizes, from 300 up to 60,000. Learning was stopped after 50 iterations for datasets of size bigger than 40,000, 100 iterations for datasets of size 10,000 to 40,000, and 150 iterations for datasets of size less than 5,000.

The results are presented in fig.6. For larger datasets ($>$ 10,000 samples) there is no difference between training the bottom layer unsupervised or supervised. However for smaller datasets, networks with bottom layers trained unsupervised perform consistently better than networks trained entirely supervised. Keeping the bottom layers random yields surprisingly good results (less than 1% classification error on large datasets), and outperforms supervised training of the whole network on very small datasets ($<$ 1,000 samples). This counterintuitive result shows that it might be better to freeze parameters at random initial values when the paucity of labeled data makes the system widely over-parameterized. Conversely, the good performance with random features hints that the lower-layer weights in fully supervised back-propagation do not need to change much to provide good enough features for the top layers. This might explain why overparameterization does not lead to a more dramatic collapse of performance when the whole network is trained supervised on just 30 samples per category. For comparison, the best published testing error rate when training on 300 samples is 3% [2], and the best error rate when training on the whole set is 0.60% [19].

## 6.2. Caltech 101

The Caltech 101 dataset has images of 101 different object categories, plus a background category. It has various numbers of samples per category (from 31 up to 800), with a total of 9,144 samples of size roughly 300 × 300 pixels.

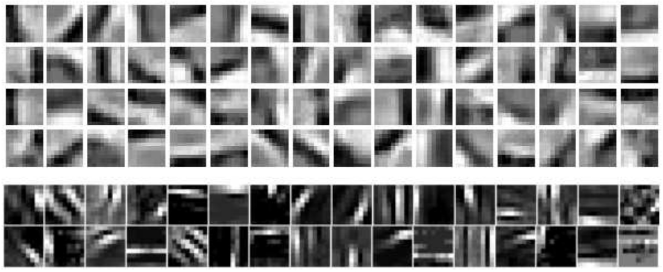


Figure 7. Caltech 101 feature extraction. Top Panel: the 64 convolutional filters of size $9 \times 9$ learned by the first level of the invariant feature extraction. Bottom Panel: a selection of 32 (out of 2048) randomly chosen filters learned in the second level of invariant feature extraction.

The common experiment protocol adopted in the literature is to take 30 images from each category for training, use the rest for testing, and measure the recognition rate for each class, and report the average.

This dataset is particularly challenging for learning-based systems, because the number of training sample per category is exceedingly small. An end-to-end supervised classifier such as a convolutional network would need a much larger number of training samples per category, lest over-fitting would occur. In the following experiment, we demonstrate that extracting features with the proposed unsupervised method leads to considerably higher accuracy than pure supervised training.

Before extracting features, the input images are preprocessed. They are converted to gray-scale, resized so that the longer edge is 140 pixels while maintaining the aspect ratio, high-pass filtered to remove the global lighting variations, and evenly zero-padded to a $140 \times 140$ image frame.

The feature extractor has the following architecture. In the first level feature extractor (layer 1 and 2) there are 64 filters of size $9 \times 9$ that output 64 feature maps of size $132 \times 132$. The next max-pooling layer takes non overlapping $4 \times 4$ windows and outputs 64 feature maps of size $33 \times 33$. Unsupervised training was performed on 100,000 patches randomly sampled from the subset of the Caltech-256 dataset [8] that does not overlap with the Caltech 101 dataset (the C-101 categories were removed). The first level was trained on such patches of size $12 \times 12$. The second level of feature extraction (layer 3 and 4) has a convolutional layer which outputs 512 feature maps and has 2048 filters. Each feature map in layer 3 combines 4 of the 64 layer-2 feature maps. These 4 feature maps are picked at random. Layer 4 is a max-pooling layer with $5 \times 5$ windows. The output of layer 4 has 512 feature maps of size $5 \times 5$. This second level was trained unsupervised on 20,000 samples of size $64 \times 13 \times 13$ produced by the first level feature extractor. Example of learned filters are shown in fig. 7.

After the feature extractor is trained, it is used to extract features on a randomly picked Caltech-101 training set with 30 samples per category (see fig. 4). To test how a baseline classifier fares on these $512 \times 5 \times 5$ features, we applied a $k$-*nearest neighbor* classifier which yielded about 20% overall average recognition rate for $k = 5$.

Next, we trained an SVM with Gaussian kernels in the one-versus-others fashion for multi-class classification. The two parameters of the SVM's, the Gaussian kernel width $\gamma^{-1}$ and the softness $C$, are tuned with cross validation, with 10 out of 30 samples per category used as the validation set. The parameters with the best validation performance, $\gamma = 5.6 \cdot 10^{-7}$, $C = 2.1 \cdot 10^{3}$, were used to train the SVM's. More than 90% of the training samples are retained as support vectors of the trained SVM's. This is an indication of the complexity of the classification task due to the small number of training samples and the large number of categories. We report the average result over 8 independent runs, in each of which 30 images of each category were randomly selected for training and the rest were used for testing. The average recognition rate over all 102 categories is **54%**($\pm$ 1%).

For comparison, we trained an essentially identical architecture in supervised mode using back-propagation (except the penultimate layer was a traditional dot-product and sigmoid layer with 200 units instead of a layer of Gaussian kernels). Supervised training from a random initial condition over the whole net achieves 100% accuracy on the training dataset (30 samples per category), but only 20% average recognition rate on the test set. This is only marginally better than the simplest baseline systems [6, 4], and considerably worse than the above result.

In our experiment, the categories that have the lowest recognition rates are the background class and some of the animal categories (wild cat, cougar, beaver, crocodile), consistent with the results reported in [12] (their experiment did not include the background class).

Our performance is similar to that of similar multi-stage Hubel-Wiesel type architectures composed of alternated layers of filters and max pooling layers. Serre et al. [20] achieved an average accuracy of 42%, while Mutch and Lowe [17] improved it to 56%. Our system is smaller than those models, and does not include feature pooling over scale. It would be reasonable to expect an improvement in accuracy if pooling over scale were used. More importantly, our model has several advantages. First, our model uses no prior knowledge about the specific dataset. Because the features are learned, it applies equally well to natural images and to digit images (and possibly other types). This is quite unlike the systems in [20, 17] which use fixed Gabor filters at the first layer. Second, using trainable filters at the second layer allows us to get away with only 512 feature maps. This is to be compared to Serre et al's 15,000 and Mutch et al's 1,500.

For reference, the best reported performance of 66.2% on this dataset was reported by Zhang et al. [21], who used a geometric blur local descriptor on interest points, and matching distance for a combined nearest neighbor and SVM. Lazebnik et al. [12] report 64.6% by matching multi-resolution histogram pyramids on SIFT. While such carefully engineered methods have an advantage with very small training set sizes, we can expect this advantage to be reduced or disappear as larger training sets become available. As evidence for this, the error rate reported by Zhang et al. on MNIST with 10,000 training samples is over 1.6%, twice our 0.84% on the same, and considerably more than our 0.64% with the full training set.

Our method is very time efficient in recognition. The feature extraction is a feed-forward computation with about $2 \cdot 10^8$ multiply-add operations for a $140 \times 140$ image and $10^9$ for $320 \times 240$. Classifying a feature vector with the Caltech-101 SVM takes another $4 \cdot 10^7$ operations. An optimized implementation of our system could be run on a modern PC at several frames per second.

## 7. Discussion and Future Work

We have presented an unsupervised method for learning sparse hierarchical features that are locally shift invariant. A simple learning algorithm was proposed to learn the parameters, level by level. We applied this method to extract features for a multi-stage Hubel-Wiesel type architecture. The model was trained on two different recognition tasks. State-of-art accuracy was achieved on handwritten digits from the MNIST dataset, and near state-of-the-art accuracy was obtained on Caltech 101. Our system is in its first generation, and we expect its accuracy on Caltech-101 to improve significantly as we gain experience with the method. Improvements could be obtained through pooling over scale, and through using position-dependent filters instead of convolutional filters. More importantly, as new datasets with more training samples will become available, we expect our learning-based methodology to improve in comparison to other methods that rely less on learning.

The contribution of this work lies in the definition of a principled method for learning the parameters of an invariant feature extractor. It is widely applicable to situations where purely supervised learning would over-fit for lack of labeled training data. The ability to *Learn the features* allows the system to adapt to the task, the lack of which limits the applicability of hand-crafted systems.

The quest for invariance under a richer set of transformations than just translations provides ample avenues for future work. Another promising avenue is to devise an extension of the unsupervised learning procedure that could train multiple levels of feature extractors in an integrated fashion rather than one at a time. A further extension would seamlessly integrate unsupervised and supervised learning.

## Acknowledgements

We thank Sebastian Seung, Geoffrey Hinton, and Yoshua Bengio for helpful discussions, and the Neural Computation and Adaptive Perception program of the Canadian Institute of Advanced Research for making them possible. This work was supported in part by NSF Grants No. 0535166 and No. 0325463.

## References

[1] http://yann.lecun.com/exdb/mnist/. 4

[2] A. Amit and A. Trouve. Pop: Patchwork of parts models for object recognition. Technical report, The Univ. of Chicago, 2005. 6

[3] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In *NIPS*. MIT Press, 2007. 1

[4] A. C. Berg, T. L. Berg, and J. Malik. Shape matching and object recognition using low distortion correspondences. In *CVPR*, 2005. 7

[5] E. Doi, D. C. Balcan, and M. S. Lewicki. A theoretical analysis of robust coding over noisy overcomplete channels. In *NIPS*. MIT Press, 2006. 4

[6] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *CVPR Workshop*, 2004. 5, 7

[7] K. Fukushima and S. Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 1982. 1

[8] G. Griffin, A. Holub, and P. Perona. The caltech 256. Technical report, Caltech, 2006. 7

[9] G. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554, 2006. 1, 2, 5

[10] F.-J. Huang and Y. LeCun. Large-scale learning with svm and convolutional nets for generic object categorization. In *CVPR*. IEEE Press, 2006. 1

[11] S. Lazebnik, C. Schmid, and J. Ponce. Semi-local affine parts for object recognition. In *BMVC*, 2004. 2

[12] S. Lazebnik, C. Schmid, and J. Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *CVPR*, 2006. 7, 8

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. 1

[14] D. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 2004. 1

[15] B. Moghaddam and A. Pentland. Probabilistic visual learning for object detection. In *ICCV*. IEEE, June 1995. 1

[16] K. Murphy, A. Torralba, D. Eaton, and W. Freeman. Object detection and localization using local and global features. *Towards Category-Level Object Recognition*, 2005. 1

[17] J. Mutch and D. Lowe. Multiclass object recognition with sparse, localized features. In *CVPR*, 2006. 1, 2, 7

[18] B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: a strategy employed by v1? *Vision Research*, 37:3311–3325, 1997. 3, 4

[19] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun. Efficient learning of sparse representations with an energy-based model. In *NIPS*. MIT Press, 2006. 1, 2, 4, 6

[20] T. Serre, L. Wolf, and T. Poggio. Object recognition with features inspired by visual cortex. In *CVPR*, 2005. 1, 2, 7

[21] H. Zhang, A. C. Berg, M. Maire, and J. Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *CVPR*, 2006. 7