Analysis-by-Synthesis by Learning to Invert Generative Black Boxes

Vinod Nair, Josh Susskind, and Geoffrey E. Hinton

Department of Computer Science
University of Toronto, Toronto, Ontario, Canada

Abstract. For learning meaningful representations of data, a rich source of prior knowledge may come in the form of a generative black box, e.g. a graphics program that generates realistic facial images. We consider the problem of learning the *inverse* of a given generative model from data. The problem is non-trivial because it is difficult to create labelled training cases by hand, and the generative mapping is a black box in the sense that there is no analytic expression for its gradient. We describe a way of training a feedforward neural network that starts with just one labelled training example and uses the generative black box to "breed" more training data. As learning proceeds, the training set evolves and the labels that the network assigns to unlabelled training data converge to their correct values. We demonstrate our approach by learning to invert a generative model of eyes and an active appearance model of faces.

1 Introduction

"Analysis-by-synthesis" is the idea of explaining an observed data vector (e.g. an image) in terms of a compact set of hidden causes that generated it. A *generative model* specifies how the underlying causes produce the data vector. A *recognition model* is the inverse mapping – it infers the causes from a given data vector. In coding terms, the recognition and generative models are the encoder and decoder, respectively, and the hidden causes represent a *code vector*. The composite of the two models implements the identity function: recognition of a data vector followed by generation from the resulting code vector should reconstruct the original data vector.

In this paper we consider the following problem: given a training set of data vectors and a generative model for that data, learn the corresponding recognition model. For example, suppose that we have a face dataset and a graphics program that can generate realistic images of faces. This program may have a set of real-valued inputs (e.g. pose, lighting, facial muscle activations) that can be smoothly varied to create any face. The task is to learn a recognition model that infers from a face image the graphics inputs that will accurately reconstruct that image. The inputs to the generative model can be seen as a compact, high-level, generative representation of the data vector.

Note that this is a *new type of problem* that existing learning algorithms are not designed to solve. Here we assume that an arbitrary generative model of

the data is given as part of the problem, and the goal is to learn the inverse of that particular model. In contrast, algorithms such as PCA, factor analysis, and ICA simply assume specific parametric forms for the generative model and fit the parameters to the data. A nonlinear autoencoder learns separate generative and recognition models simultaneously by also assuming the generative model to be of a specific form with an analytic gradient. As explained later, our problem is more difficult than the ones solved by these standard methods. Our main contribution here is a learning algorithm that solves the above problem.

There are two main motivations for this work. First, it provides a general way of incorporating domain knowledge, via a generative model, into the learning of a data representation. Generative models are a natural way of expressing complex prior knowledge. For example, in speech, knowledge about how the vocal system produces sounds can be expressed as an articulatory synthesizer. In modelling face images, knowledge about facial muscles and skin can be expressed as a physics-based graphics model. The knowledge contained in such models gives their inputs (i.e. the code vector) high-level semantics, so inverting these models creates useful high-level representations of the data. Such representations can capture the true degrees of freedom in the data much better than those learned by generic algorithms like PCA or ICA.

Second, our work allows the automatic transfer of research advances in generative modelling to the learning of compact data representations. Progress in graphics and speech synthesis can be used directly to learn better representations of images and speech data. For example, the graphics community is rapidly advancing the realism and detail of physically-based graphics models for faces [1], [2]. Successfully inverting them will result in new facial image representations that can significantly improve applications such as face recognition, facial image coding and expression analysis.

To make our learning algorithm as general as possible, we do not assume any knowledge of the internal details of the generative model. It is a "black box" function that can be evaluated as many times as we like, but no additional information about it – in particular its gradient – is given. So the learning of the recognition model is de-coupled from the internal details of the generative model. Thus, any of the wide variety of generative models proposed in the literature for different types of data can be used without changing the learning algorithm. The alternative to not using the black box assumption is to design a separate learning algorithm for each inversion problem, which is undesirable.

Overview of our approach: Learning to invert a nonlinear[1], deterministic generative black box is difficult. Assuming real-valued code and data vectors, we want to learn a regression mapping from data space to code space. There are three problems: first, only the regression function's inputs (data vectors) are given for learning – the corresponding target outputs are unknown. If they were known, then the problem reduces to standard supervised learning. For a complex

[1] The linear case is easy: elements of the generative matrix can be discovered trivially by evaluating it on the standard basis vectors.

generative model, inferring a code vector that reconstructs a given data vector is nontrivial.

Second, the black box's gradient is unknown, so learning cannot be done by propagating the gradient of the data reconstruction error through the black box. Estimating it numerically by finite differencing is too inefficient. If the black box gradient were known, then learning becomes similar to that of an autoencoder, except the generative model is fixed.

Third, codes corresponding to the real data occupy only a small volume in code space. For example, consider a face model that simulates muscles with springs. Humans cannot independently control each facial muscle, so the muscle activations are dependent on each other. Therefore only a small subspace of possible spring states correspond to valid facial configurations. This makes it impractical to naively take *random* samples from code space, generate data vectors from those samples using the black box, and learn the recognition model from the resulting input-output pairs. Such an approach will waste almost all the capacity of the recognition model on "junk" training cases far away from the real data that we are interested in modelling.

Our approach solves all three problems. The basic idea is to use the black box itself to generate synthetic data vectors, but from codes sampled in a more intelligent way than random sampling. To restrict the learning to the small portion of code space that is actually relevant, a single code vector corresponding to a real training data vector is assumed to be given. Starting from this single labelled example, the algorithm computes a set of nearby code vectors from which the synthetic data vectors are generated. The recognition model is trained by standard supervised learning on the resulting input-output pairs. As learning proceeds, the sampling procedure produces code vectors from an increasingly broader distribution. The details are in section 2. As the algorithm "breeds" its own labelled training cases, we refer to it as *breeder learning*.

We use breeder learning to invert two different generative black boxes, one for images of eyes (section 3.1) and the other for faces (section 3.2). In the former case, we got the graphics program from its authors [3] and simply used it as a subroutine in our algorithm without looking at its internal implementation details. This shows the generality of the algorithm, as well as the usefulness of de-coupling the design details of the generative model from the learning.

2 Breeder learning

Figure 1 summarizes the learning algorithm. The algorithm requires a single code vector, chosen by hand, that produces a data vector close to the ones in the training set. We call this code vector the *prototype*. The inputs to the algorithm are the prototype, the generative black box, and the training set X of data vectors. The final output is a recognition model trained to accurately infer from a data vector its corresponding code vector. We choose a feedforward neural network with a single hidden layer of sigmoid units to implement the recognition model. An important advantage of neural networks is that, unlike

Algorithm for training a recognition network R_w parameterized by weight vector w:

Given: Training set X of n data vectors $\{x_1, x_2, ..., x_n\}$, generative black box G, prototype code vector p.

Initialization: Set output biases of R_w using p, and the remaining weights to samples from a zero-mean Gaussian with a small standard deviation.

Weight update computed using the i^{th} (unlabelled) training case x_i:
Let y_i be the code vector inferred from x_i using the current recognition network R_w.

1. $y_i = R_w(x_i)$.
2. Perturb y_i randomly to create y'_i. (Note: The exact perturbation method is specified later.)
3. $x'_i = G(y'_i)$.
4. Supervised learning on (x'_i, y'_i):
   (a) $y''_i = R_w(x'_i)$.
   (b) $E = \|y'_i - y''_i\|^2$.
   (c) $w \leftarrow w - \eta \frac{\partial E}{\partial w}$.

Fig. 1. Summary of the breeder learning algorithm.

batch learning methods such as support vector machines, they can be trained online on a dynamically generated, ever-changing dataset, as is the case here.

The biases of the recognition network's output units are initialized in such a way that the network outputs the prototype in the absence of any input. The remaining weights of the neural network are initialized to small random values, which prevents the input from having much effect on the output initially. So, early on in training, the network (mostly) ignores its input and outputs codes in the vicinity of the prototype.

Once initialized, the weights are updated iteratively. Each iteration has four major steps: 1) The "real" data vectors in X are propagated through the current recognition network to infer their corresponding code vectors. 2) These code vectors are then perturbed with noise. (The exact perturbation method depends on whether the output units are linear or sigmoid. It will be specified later when the applications are discussed.) 3) The generative model is applied to the perturbed codes to produce their corresponding synthetic data vectors. The noisy codes and the data vectors generated from them form a correct set of training pairs for the network. 4) The weights are updated by the negative gradient of the squared error loss (in code space) calculated for the synthetic pairs.

Because of how the network is initialized, it first learns to invert the generative model in a small neighbourhood around the prototype. The early noisy codes will be minor variants of the prototype, so the synthetic training pairs will not be very diverse. At this point the network can correctly infer the codes for only a small subset of X, i.e., those that are near the prototype's data vector. Randomly perturbing the outputs allows the network to discover codes slightly farther away from the prototype. Training on them expands the region in code space that the network can correctly handle.

In subsequent iterations, the network will correctly infer the codes for a few more real data vectors. Perturbing their codes generates new ones that are even farther away from the prototype. As learning progresses, the synthetic training pairs become increasingly diverse, as the codes come from a larger region in code space. The network eventually learns to handle the entire region of code space corresponding to the real data vectors.

Some notes about the algorithm:

• The underlying assumption is that Euclidean distance in code space is a more semantically meaningful way of assessing similarity than any generic distance metric in data space. Therefore small random perturbations in code space should produce semantically similar data vectors that may nevertheless have a large Euclidean distance between them.

• The algorithm is not doing a naive random search in code space. Instead, it uses the current recognition network itself to produce new codes to learn on. So the network's ability to correctly generalize to previously unseen data vectors allows the algorithm to discover codes that correspond to real data vectors much more efficiently than a random search.

• The training set of code-data pairs is generated on the fly, and interestingly, it depends on the trained model itself. It starts off as a mostly homogeneous set and becomes more diverse as learning progresses.

• There is no attempt to filter the synthetic code-data pairs by removing those data vectors that are highly dissimilar from the real data vectors. Generic similarity metrics in data space can be highly misleading, and such filtering is likely to make the learning worse. Without filtering the network will occasionally learn on "junk" training cases. But such cases are unlikely to be exactly re-generated many times, so they are quickly forgotten.

• The use of a single prototype code vector amounts to assuming that there are no well-separated modes in code space corresponding to real data. If this assumption is false, it is possible to formulate a mixture version of the algorithm that can handle far-apart modes. It would require creating one prototype per mode.

Random-code learning: A simpler alternative to breeder learning is to 1) sample an isotropic Gaussian centred on the prototype code vector and with a fixed variance, 2) generate synthetic data vectors from these samples, and 3) train the recognition network on the resulting pairs. In our experiments (section 3) this alternative consistently performs worse than breeder learning. If the Gaussian's variance is too large, many of the sampled codes will correspond to junk data vectors. If it is too small, it will almost never see valid codes that happen to be far away from the prototype. So the particular way in which breeder learning creates new codes is crucial for its success and cannot be replaced by a naive random search.

3 Results

The rest of the paper describes two applications of breeder learning: inverting a generative model for eye images [3], and inverting an active appearance model

for faces [4]. In both cases we learn a generative representation for real images by taking a generative model from the literature and simply "plugging it in" as the black box into breeder learning. These applications show 1) the generality of our algorithm, and 2) its usefulness for exploiting an existing generative model to learn a compact, high-level representation of the data.

3.1 Inverting a 2D model of eye images

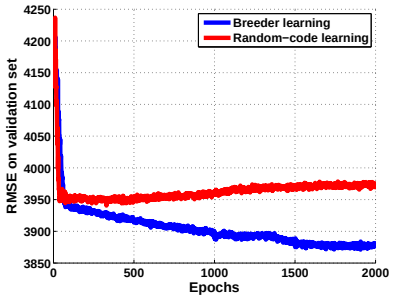
The generative black box is a 2D model of eye images proposed by Moriyama et al. [3]. They use knowledge about the eye's anatomy to define a high-level generative representation of eye images. Since breeder learning does not need to know the model's internal details, we explain them only briefly here. See [3] for a full description.

The inputs to the black box are parameters that specify high-level properties of the eye, such as gaze direction and how open the eyelid is. The model uses the inputs to first compute a set of polygonal curves that represent the 2D shape of the sclera, iris, upper eyelid, lower eyelid, and the corners of the eye. Once the shape is computed, we use a simple texture model to generate a 32×64 grayscale image from it (see examples in the even columns of figure 3). In total there are eight inputs to the black box, all normalized to be in the range $[0, 1]$. (These inputs affect only the shape; the texture model is fixed.) Given this black box and a training set of real eye images, we use breeder learning to learn the corresponding recognition model.

Dataset and training details: We use 1272 eye images collected from faces of people acting out different expressions. We normalize all images to be 32×64, and apply histogram equalization to remove lighting variations. See the odd-numbered columns of figure 3 for example images. Since the eye images come from faces with many different expressions and ethnicities, they contain a wide variety of shapes and represent a difficult shape modelling task. We select the prototype code to be the vector with all components set to 0.5, which is the midpoint of each code dimension's range of possible values. From the set of 1272 images, 872 are used for training, 200 for validation and 200 for testing.

The recognition network has 2048 input units ($32 \times 64 = 2048$ pixels), 100 sigmoid units in the hidden layer, and 8 sigmoid units in the output layer. A code vector is randomly perturbed during learning by adding zero-mean Gaussian noise with a standard deviation of 0.25 to the total input of each code unit. Training is stopped when the root mean squared error (RMSE) of the validation images is minimized. The recognition network trained by breeder learning achieves its best performance on the validation set after about 1900 epochs.

Figure 2 shows the RMSE achieved by breeder learning on the validation set as training proceeds. Random-code learning is unable to improve the RMSE beyond a certain value and starts overfitting because it only sees training cases from a limited region around the prototype. We tried various values for the variance of random-code learning, and the results shown are for the one that gave the best performance on the validation set.



Algorithm	Test RMSE
Breeder learning	3902.56
Random-code learning	3977.37

Fig. 2. Validation set RMSE (left graph) during training, and test set RMSE (above table) after training, for breeder and random-code learning algorithms on the eye dataset.

Figure 3 shows examples of test images reconstructed by the recognition network trained with breeder learning. The inferred boundaries of the sclera and iris regions are superimposed on the real image. Notice that the network is able to correctly infer the codes for eyes with significantly different shapes. This is despite using a very limited texture model in the black box.

3.2 Inverting an active appearance model of faces

We now consider inverting an active appearance model (AAM) of face images [4]. The AAM is a popular nonlinear generative model that incorporates high-level knowledge about facial shape and texture to learn a low-dimensional representation of faces. Unlike the eye model, here the black box itself is learned from data, but this difference is irrelevant from the point of view of breeder learning.

Our implementation of the AAM follows Cootes et al. [4]. Again, we only give a brief overview of it here. It consists of separate PCA models for facial shape and texture, whose outputs are combined via a nonlinear warp to generate the face image. As in [4], we apply PCA again to the shape and texture representations of the training images to produce a higher-level "appearance" model of faces. The warp makes the AAM's output a nonlinear function of its input.

We first train the AAM using a set of face images, and then use it as a *fixed* generative black box for breeder learning. The face images are of size 30×30, and the AAM's appearance representation (i.e., the code vector) is chosen to be 60-dimensional. So for the purposes of breeder learning, we treat the AAM as a black box that takes 60 real-valued inputs and produces a 30×30 face image as

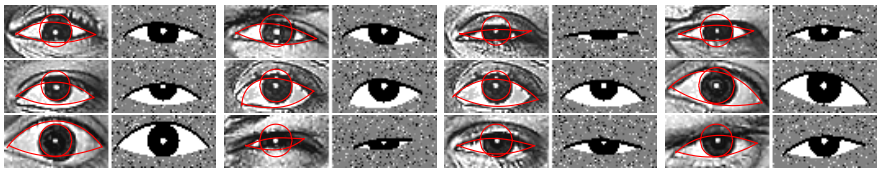
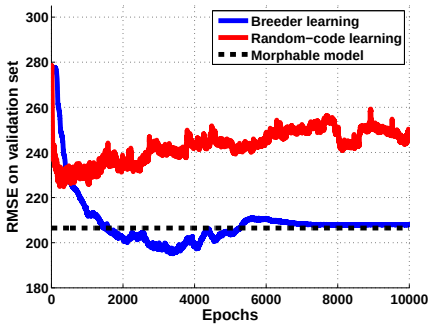


Fig. 3. Test image reconstructions computed by the recognition network trained with breeder learning.



Algorithm	Test RMSE
Breeder learning	200.17
Random-code learning	229.31

Fig. 4. Validation set RMSE (left graph) during training, and test set RMSE (above table) after training, for breeder and random-code learning algorithms on the face dataset.

output. (Note that the AAM learning procedure itself computes the codes for its *training* images as part of learning, so they are known, but we do not use them when learning the recognition network. The correct codes for the *test* images are truly unknown.)

Dataset and training details: We use 400 frontal faces (histogram-equalized) containing different expressions and identities. The dataset is split into 300 training images, 50 validation images, and 50 test images. None of the identities in the test set appear in the training and validation sets, so at test time, the recognition network has to generalize correctly to unseen identities, which is a difficult task. Note that only the 300 training and 50 validation images are used in the learning of the AAM itself.

The recognition network has 900 input units, a hidden layer of 100 sigmoid units, and 60 linear output units. We select the origin of the code space, corresponding to the face with the mean shape and mean texture, as the prototype code. Since the network's output units are linear, the code vectors are perturbed during learning by adding zero-mean Gaussian noise (with 0.1 standard deviation) directly to the outputs. The recognition network trained by breeder learning achieves its best performance on the validation set after slightly fewer than 3400 epochs.

Figure 4 shows the RMSE results. Interestingly, the best reconstruction error achieved by breeder learning on the validation set is below that of the AAM itself (dashed line in the graph). This means that the net is able to find codes that are better in the squared pixel error sense than the ones found by the AAM learning. Example reconstructions of test faces are shown in figure 5. The recognition task here is quite challenging as the network has to generalize to new identities. In most cases, the network reconstructs the face with approximately the correct expression and identity. In contrast, the reconstructions computed by the network learned with random-code learning are visually much worse and most of them resemble the face corresponding to the prototype code.

3.3 Iterative refinement of reconstructions with a generative network

So far recognition has been treated as a purely *feedforward*, *bottom-up* computation. A key property of analysis-by-synthesis is the use of top-down knowledge in the generative model to improve inference via a feedback loop that minimizes an error measure in data space itself. In our case implementing such a feedback loop requires knowing the gradient of the generative black box. But once a fully-trained recognition network is available, an alternative approach becomes possible.

The idea is to approximate the function implemented by the black box with a *generative neural network* [5]. This network emulates the black box: it takes a code vector as input and computes the corresponding data vector as output. Once such a generative network is trained, it can be used to compute the gradient of the data reconstruction error with respect to the code. As a result, inference now becomes a gradient-based iterative optimization problem that minimizes reconstruction error in data space.

Training the generative network is possible only because a fully-trained recognition model is already available. It provides the generative network approximately correct target codes for the training images. Given these code-image pairs, training reduces to a standard supervised learning task. The recognition network restricts the learning to the small part of data space that contains the real data, thus making it practical.

Figure 6 summarizes the closed-loop version of the recognition procedure. The initial code is computed by a bottom-up pass through the recognition network as before. But unlike in open-loop recognition, this initial estimate is subsequently refined by gradient descent on the squared error between the data vector and its reconstruction. The iterations continue until the squared error stops improving.

We learned a generative network to emulate the AAM and then used it to refine the reconstructions of faces. The average improvement in squared pixel error for the validation and test sets are 6.28% and 5.41%, respectively. It should be emphasized that the closed-loop recognition algorithm is used only as a way of fine-tuning the initial open-loop code estimate, which is already a very good solution. This side-steps the issue of whether a generic distance metric such as

Fig. 5. Test image reconstructions computed by the recognition network trained with breeder learning.

Algorithm for closed-loop recognition of data vector x:

Given: Generative black box G, recognition network R_w, generative network $G_{w'}$.

Initialization: $y = R_w(x)$.

For each refinement iteration:

1. $x' = G(y)$.
2. $E = \|x - x'\|^2$.
3. Compute $\frac{\partial E}{\partial y}$ by backpropagation through $G_{w'}$.
4. $y \leftarrow y - \eta \frac{\partial E}{\partial y}$.

Fig. 6. The closed-loop recognition algorithm using a generative neural network.

Euclidean distance can be used to correctly measure similarity in data space. Here we use Euclidean distance to measure *local* similarity only, i.e. to decide how an already good solution can be made a little bit better. So minimizing Euclidean distance can be sensible for fine-tuning, even if it is prone to get stuck in shallow local minima when starting from a random solution.

4 Conclusions

Breeder learning is a new tool for engineers building recognition models. By taking advantage of the rich domain knowledge in a generative model, it can learn much higher-level representations of data than those learned by standard methods such as PCA or ICA. Inverting complex physically-based generative models is one of the most promising future applications of breeder learning. Successfully inverting them will result in improved data representations for applications such as classification and compression.

5 Acknowledgements

This work was supported by NSERC, CIFAR, and FQRNT.

References

1. Lee, Y., Terzopoulos, D., Waters, K.: Realistic modeling for facial animation. In: SIGGRAPH. (1995)
2. Sifakis, E., Neverov, I., Fedkiw, R.: Automatic determination of facial muscle activations from sparse motion capture marker data. In: SIGGRAPH. (2005)
3. Moriyama, T., Kanade, T., Xiao, J., Cohn, J.F.: Meticulously detailed eye region model and its application to analysis of facial images. IEEE Transactions on Pattern Analysis and Machine Intelligence **28**(5) (2006) 738–752
4. Cootes, T.F., Edwards, G.J., Taylor, C.J.: Active appearance models. IEEE Transactions on Pattern Analysis and Machine Intelligence **23**(6) (2001) 681–685
5. Jordan, M., Rumelhart, D.: Forward models: Supervised learning with a distal teacher. Cognitive Science **16** (1992) 307–354