Scaling Learning Algorithms towards AI

Yoshua Bengio (1) and Yann LeCun (2)
(1) Yoshua.Bengio@umontreal.ca
Département d'Informatique et Recherche Opérationnelle
Université de Montréal,
(2) yann@cs.nyu.edu
The Courant Institute of Mathematical Sciences,
New York University, New York, NY

To appear in "Large-Scale Kernel Machines",
L. Bottou, O. Chapelle, D. DeCoste, J. Weston (eds)
MIT Press, 2007

Abstract

One long-term goal of machine learning research is to produce methods that are applicable to highly complex tasks, such as perception (vision, audition), reasoning, intelligent control, and other artificially intelligent behaviors. We argue that in order to progress toward this goal, the Machine Learning community must endeavor to discover algorithms that can learn highly complex functions, with minimal need for prior knowledge, and with minimal human intervention. We present mathematical and empirical evidence suggesting that many popular approaches to non-parametric learning, particularly kernel methods, are fundamentally limited in their ability to learn complex high-dimensional functions. Our analysis focuses on two problems. First, kernel machines are *shallow architectures*, in which one large layer of *simple template matchers* is followed by a single layer of trainable coefficients. We argue that shallow architectures can be very inefficient in terms of required number of computational elements and examples. Second, we analyze a limitation of kernel machines with a local kernel, linked to the curse of dimensionality, that applies to supervised, unsupervised (manifold learning) and semi-supervised kernel machines. Using empirical results on invariant image recognition tasks, kernel methods are compared with *deep architectures*, in which lower-level features or concepts are progressively combined into more abstract and higher-level representations. We argue that deep architectures have the potential to generalize in non-local ways, i.e., beyond immediate neighbors, and that this is crucial in order to make progress on the kind of complex tasks required for artificial intelligence.

1 Introduction

Statistical machine learning research has yielded a rich set of algorithmic and mathematical tools over the last decades, and has given rise to a number of commercial and scientific applications. However, some of the initial goals of this field of research remain elusive. A long-term goal of machine learning research is to produce methods that will enable artificially intelligent agents capable of learning complex behaviors with minimal human intervention and prior knowledge. Examples of such complex behaviors are found in visual perception, auditory perception, and natural language processing.

The main objective of this chapter is to discuss fundamental limitations of certain classes of learning algorithms, and point towards approaches that overcome these limitations. These limitations arise from two aspects of these algorithms: *shallow architecture*, and *local estimators*.

We would like our learning algorithms to be efficient in three respects:

1. computational: number of computations during training and during recognition,
2. statistical: number of examples required for good generalization, especially labeled data, and
3. human involvement: amount of human labor necessary to tailor the algorithm to a task, i.e., specify the prior knowledge built into the model before training. (explicitly, or implicitly through engineering designs with a human-in-the-loop).

The last quarter century has given us flexible non-parametric learning algorithms that can learn any continuous input-output mapping, *provided* enough computing resources and training data. A crucial question is how efficient are some of the popular learning methods when they are applied to complex perceptual tasks, such a visual pattern recognition with complicated intra-class variability. The chapter mostly focuses on computational and statistical efficiency.

Among flexible learning algorithms, we establish a distinction between *shallow architectures*, and *deep architectures*. Shallow architectures are best exemplified by modern kernel machines [Schölkopf et al., 1999], such as Support Vector Machines (SVMs) [Boser et al., 1992, Cortes and Vapnik, 1995]. They consist of one layer of fixed kernel functions, whose role is to match the incoming pattern with templates extracted from a training set, followed by a linear combination of the matching scores. Since the templates are extracted from the training set, the first layer of a kernel machine can be seen as being trained in a somewhat trivial unsupervised way. The only components subject to supervised training are the coefficients of the linear combination. [1]

Deep architectures are perhaps best exemplified by multi-layer neural networks with several hidden layers. In general terms, deep architectures are composed of multiple layers of parameterized non-linear modules. The parameters of every module are

[1] In SVMs only a subset of the examples are selected as templates (the support vectors), but this is equivalent to choosing which coefficients of the second layer are non-zero.

subject to learning. Deep architectures rarely appear in the machine learning literature; the vast majority of neural network research has focused on shallow architectures with a single hidden layer, because of the difficulty of training networks with more than 2 or 3 layers [Tesauro, 1992]. Notable exceptions include work on convolutional networks [LeCun et al., 1989, LeCun et al., 1998], and recent work on Deep Belief Networks [Hinton et al., 2006].

While shallow architectures have advantages, such as the possibility to use convex loss functions, we show that they also have limitations in the *efficiency* of the representation of certain types of function families. Although a number of theorems show that certain shallow architectures (Gaussian kernel machines, 1-hidden layer neural nets, etc) can approximate any function with arbitrary precision, they make no statements as to the efficiency of the representation. Conversely, deep architectures can, in principle, represent certain families of functions more efficiently (and with better scaling properties) than shallow ones, but the associated loss functions are almost always non convex.

The chapter starts with a short discussion about task-specific versus more general types of learning algorithms. Although the human brain is sometimes cited as an existence proof of a general-purpose learning algorithm, appearances can be deceiving: the so-called no-free-lunch theorems [Wolpert, 1996], as well as Vapnik's necessary and sufficient conditions for consistency [Vapnik, 1998, see], clearly show that there is no such thing as a completely general learning algorithm. All practical learning algorithms are associated with some sort of explicit or implicit prior that favors some functions over others.

Since a quest for a completely general learning method is doomed to failure, one is reduced to searching for learning models that are well suited for a particular type of tasks. For us, high on the list of useful tasks are those that most animals can perform effortlessly, such as perception and control, as well as tasks that higher animals and humans can do such as long-term prediction, reasoning, planning, and language understanding. In short, our aim is to look for learning methods that bring us closer to an artificially intelligent agent. What matters the most in this endeavor is how *efficiently* our model can capture and represent the required knowledge. The efficiency is measured along three main dimensions: the amount of training data required (especially labeled data), the amount of computing resources required to reach a given level of performance, and most importantly, the amount of human effort required to specify the prior knowledge built into the model before training (explicitly, or implicitly) This chapter discusses the scaling properties of various learning models, in particular kernel machines, with respect to those three dimensions, in particular the first two. Kernel machines are *non-parametric learning models*, which make apparently weak assumptions on the form of the function $f()$ to be learned. By non-parametric methods we mean methods which allow the complexity of the solution to increase (e.g., by hyper-parameter selection) when more data are available. This includes classical k-nearest-neighbor algorithms, modern kernel machines, mixture models, and multi-layer neural networks (where the number of hidden units can be selected using the data). Our arguments are centered around two limitations of kernel machines: the first limitation applies more generally to shallow architectures, which include neural networks with a single hidden layer. In Section 3 we consider different types of function classes, i.e., architectures, including different sub-types of shallow architectures. We consider the trade-off between the depth of the architecture and its breadth (number of elements in each layer), thus clarifying the representational limitation of shallow architectures. The second limitation is more specific and concerns kernel machines with a *local kernel*. This limitation is studied first informally in Section 3.3 by thought experiments in the use of template matching for visual perception. Section 4 then focusses more formally on local estimators, i.e., in which the prediction $f(x)$ at point x is dominated by the near neighbors of x taken from the training set. This includes kernel machines in which the kernel is local, like the Gaussian kernel. These algorithms rely on a prior expressed as a distance or similarity function between pairs of examples, and encompass classical statistical algorithms as well as modern kernel machines. This limitation is pervasive, not only in classification, regression, and density estimation, but also in manifold learning and semi-supervised learning, where many modern methods have such locality property, and are often explicitly based on the graph of near neighbors. Using visual pattern recognition as an example, we illustrate how the shallow nature of kernel machines leads to fundamentally inefficient representations.

Finally, deep architectures are proposed as a way to escape from the fundamental limitations above. Section 5 concentrates on the advantages and disadvantages of deep architectures, which involve multiple levels of trainable modules between input and output. They can retain the desired flexibility in the learned functions, and increase the efficiency of the model along all three dimensions of amount of training data, amount of computational resources, and amount of human prior hand-coding. Although a number of learning algorithms for deep architectures have been available for some time, training such architectures is still largely perceived as a difficult challenge. We discuss recent approaches to training such deep networks that foreshadows new breakthroughs in this direction.

The trade-off between convexity and non-convexity has, up until recently, favored research into learning algorithms with convex optimization problems. We have found that non-convex optimization is sometimes more efficient that convex optimization. Non-convex loss functions may be an unavoidable property of learning complex functions from weak prior knowledge.

2 Learning Models Towards AI

The *No-Free-Lunch* theorem for learning algorithms [Wolpert, 1996] states that no completely general-purpose learning algorithm can exist, in the sense that for every learning model there is a data distribution on which it will fare poorly (on both training and test, in the case of finite VC dimension). Every learning model *must* contain implicit or explicit restrictions on the class of functions that it can learn. Among the set of all possible functions, we are particularly interested in a subset that contains all the tasks involved in intelligent behavior. Examples of such tasks include visual perception, auditory perception, planning, control, etc. The set does not just include specific visual perception tasks (e.g human face detection), but the set of all the tasks that an intelligent agent should be able to learn. In the following, we will call this set of functions *the AI-set*. Because we want to achieve AI, we prioritize those tasks that are in the AI-set.

Although we may like to think that the human brain is somewhat general-purpose, it is extremely restricted in its ability to learn high-dimensional functions. The brains of humans and higher animals, with their learning abilities, can potentially implement the AI-set, and constitute a working proof of the feasibility of AI. We advance that the AI-set is a tiny subset of the set of all possible functions, but the specification of this tiny subset may be easier than it appears. To illustrate this point, we will use the example first proposed by [LeCun and Denker, 1992]. The connection between the retina and the visual areas in the brain gets wired up relatively late in embryogenesis. If one makes the apparently reasonable assumption that all possible permutations of the millions of fibers in the optic nerve are equiprobable, there is not enough bits in the genome to encode the correct wiring, and no lifetime long enough to learn it. The flat prior assumption must be rejected: some wiring must be simpler to specify (or more likely) than others. In what seems like an incredibly fortunate coincidence, a particularly good (if not "correct") wiring pattern happens to be one that preserves topology. Coincidentally, this wiring pattern happens to be very simple to describe in almost any language (for example, the biochemical language used by biology can easily specify topology-preserving wiring patterns through concentration gradients of nerve growth factors). How can we be so fortunate that the correct prior be so simple to describe, yet so informative? LeCun and Denker [1992] point out that the brain exists in the very same physical world for which it needs to build internal models. Hence the specification of good priors for modeling the world happen to be simple in that world (the dimensionality and topology of the world is common to both). Because of this, we are allowed to hope that the AI-set, while a tiny subset of all possible functions, may be specified with a relatively small amount of information.

In practice, prior knowledge can be embedded in a learning model by specifying three essential components:

1. The representation of the data: pre-processing, feature extractions, etc.
2. The *architecture* of the machine: the family of functions that the machine can implement and its parameterization.
3. The *loss function and regularizer*: how different functions in the family are rated, given a set of training samples, and which functions are preferred in the absence of training samples (prior or regularizer).

Inspired by [Hinton, To appear. 2007], we classify machine learning research strategies in the pursuit of AI into three categories. One is *defeatism*: "Since no good parameterization of the AI-set is currently available, let's specify a much smaller set for each specific task through careful hand-design of the pre-processing, the architecture, and the regularizer". If task-specific designs must be devised by hand for each new task, achieving AI will require an overwhelming amount of human effort. Nevertheless, this constitutes the most popular approach for applying machine learning to new problems: design a clever pre-processing (or data representation scheme), so that a standard learning model (such as an SVM) will be able to learn the task. A somewhat similar approach is to specify the task-specific prior knowledge in the structure of a *graphical model* by explicitly representing important intermediate features and concepts through latent variables whose functional dependency on observed variables is hard-wired. Much of the research in graphical models [Jordan, 1998] (especially of the parametric type) follows this approach. Both of these approaches, the kernel approach with human-designed kernels or features, and the graphical models approach with human-designed dependency structure and semantics, are very attractive in the short term because they often yield quick results in making progress on a specific task, taking advantage of human ingenuity and implicit or explicit knowledge about the task, and requiring small amounts of labeled data.

The second strategy is *denial*: "Even with a generic kernel such as the Gaussian kernel, kernel machines can approximate any function, and regularization (with the bounds) guarantee generalization. Why would we need anything else?" This belief contradicts the no free lunch theorem. Although kernel machines can represent any labeling of a particular training set, they can *efficiently represent* a very small and very specific subset of functions, which the following sections of this chapter will attempt to characterize. Whether this small subset covers a large part of the AI-set is very dubious, as we will show. In general, what we think of as generic learning algorithms can only work well with certain types of data representations and not so well with others. They can in fact represent certain types of functions efficiently, and not others. While the clever preprocessing/generic learning algorithm approach may be useful for solving specific problems, it brings about little progress on the road to AI. How can we hope to solve the wide variety of tasks required to achieve AI with this labor-intensive approach? More importantly, how can we ever hope to integrate each of these separately-built, separately-trained, specialized modules into a coherent artificially intelligent system? Even if we could build those modules, we would need another learning paradigm to be able to integrate them into a coherent system.

The third strategy is *optimism*: "let's look for learning models that can be applied to the largest possible subset of the AI-set, while requiring the smallest possible amount of additional hand-specified knowledge for each specific task within the AI-set". The question becomes: is there a parameterization of the AI-set that can be efficiently implemented with computer technology?

Consider for example the problem of object recognition in computer vision: we could be interested in building recognizers for at least several thousand categories of objects. Should we have specialized algorithms for each? Similarly, in natural language processing, the focus of much current research is on devising appropriate features for specific tasks such as recognizing or parsing text of a particular type (such as spam email, job ads, financial news, etc). Are we going to have to do this labor-intensive work for all the possible types of text? our system will not be very smart if we have to manually engineer new patches each time new a type of text or new types of object category must be processed. If there exist more general-purpose learning models, at least general enough to handle most of the tasks that animals and humans can handle, then searching for them may save us a considerable amount of labor in the long run.

As discussed in the next section, a mathematically convenient way to characterize the kind of complex task needed for AI is that they involve learning highly non-linear functions with many variations (i.e., whose derivative changes direction often). This is problematic in conjunction with a prior that smooth functions are more likely, i.e., having few or small variations. We mean f to be smooth when the value of $f(x)$ and of its derivative $f'(x)$ are close to the values of $f(x + \Delta)$ and $f'(x + \Delta)$ respectively when x and $x + \Delta$ are close as defined by a kernel or a distance. This chapter advances several arguments that the smoothness prior alone is insufficient to learn highly-varying functions. This is intimately related to the curse of dimensionality, but as we find throughout our investigation, it is not the number of dimensions so much as the amount of variation that matters. A one-dimensional function could be difficult to learn, and many high-dimensional functions can be approximated well enough with a smooth function, so that non-parametric methods relying only on the smooth prior can still give good results.

We call *strong priors* a type of prior knowledge that gives high probability (or low complexity) to a very small set of functions (generally related to a small set of tasks), and *broad priors* a type of prior knowledge that give moderately high probability to a wider set of relevant functions (which may cover a large subset of tasks within the AI-set). Strong priors are task-specific, while broad priors are more related to the general structure of our world. We could prematurely conjecture that if a function has many local variations (hence is not very smooth), then it is not learnable unless strong prior knowledge is at hand. Fortunately, this is not true. First, there is no reason to believe that smoothness priors should have a special status over other types of priors. Using smoothness priors when we know that the functions we want to learn are non-smooth would seem counter-productive. Other broad priors are possible. A simple way to define a prior is to define a language (e.g., a programming language) with which we express functions, and favor functions that have a low Kolmogorov complexity in that language, i.e. functions whose program is short. Consider using the C programming language (along with standard libraries that come with it) to define our prior, and learning functions such as $g(x) = sin(x)$ (with x a real value) or $g(x) = parity(x)$ (with x a binary vector of fixed dimension). These would be relatively easy to learn with a small number of samples because their description is extremely short in C and they are very probable under the corresponding prior, despite the fact that they are highly non-smooth. We do not advocate the explicit use of Kolmogorov complexity in a conventional programming language to design new learning algorithms, but we use this example to illustrate that it is possible to learn apparently complex functions (in the sense they vary a lot) using broad priors, by using a non-local learning algorithm, corresponding to priors other than the smoothness prior. This thought example and the study of toy problems like the parity problem in the rest of the chapter also shows that the main challenge is to design learning algorithms that can *discover representations of the data that compactly describe regularities in it*. This is in contrast with the approach of enumerating the variations present in the training data, and hoping to rely on local smoothness to correctly fill in the space between the training samples.

As we mentioned earlier, there may exist broad priors, with seemingly simple description, that greatly reduce the space of accessible functions in appropriate ways. In visual systems, an example of such a broad prior, which is inspired by Nature's bias towards retinotopic mappings, is the kind of connectivity used in convolutional networks for visual pattern recognition [LeCun et al., 1989, LeCun et al., 1998]. This will be examined in detail in section 6. Another example of broad prior, which we discuss in section 5, is that the functions to be learned should be expressible as multiple levels of composition of simpler functions, where *different levels of functions can be viewed as different levels of abstraction*. The notion of "concept" and of "abstraction" that we talk about is rather broad and simply means a random quantity strongly dependent of the observed data, and useful in building a representation of its distribution that generalises well. Functions at lower levels of abstraction should be found useful for capturing some simpler aspects of the data distribution, so that it is possible to first learn the simpler functions and then compose them to learn more abstract concepts. Animals and humans do learn in this way, with simpler concepts earlier in life, and higher-level abstractions later, expressed in terms of the previously learned concepts. Not all functions can be decomposed in this way, but humans appear to have such a constraint. If such a hierarchy did not exist, humans would be able to learn new concepts in any order. Hence we can hope that this type of prior may be useful to help cover the AI-set, but yet specific enough to exclude the vast majority of useless functions.

It is a thesis of the present work that learning algorithms that build such deeply layered architectures offer a promising avenue for scaling machine learning towards AI. Another related thesis is that one should not consider the large variety of tasks separately, but as different aspects of a more general problem: that of learning the basic structure of the world, as seen say through the eyes and ears of a growing animal or a young child. This is an instance of multi-task learning where it is clear that the different tasks share a strong commonality. This allows us to hope that after training such a system on a large variety of tasks in the AI-set, the system may generalize to a new task from only a few labeled examples. We hypothesize that many tasks in the AI-set may be built around common *representations*, which can be understood as a set of interrelated concepts.

If our goal is to build a learning machine for the AI-set, our research should concentrate on devising learning models with the following features:

- A highly flexible way to specify prior knowledge, hence a learning algorithm that can function with a large repertoire of architectures.
- A learning algorithm that can deal with deep architectures, in which a decision involves the manipulation of many intermediate concepts, and multiple levels of non-linear steps.
- A learning algorithm that can handle large families of functions, parameterized with millions of individual parameters.
- A learning algorithm that can be trained efficiently even, when the number of training examples becomes very large. This excludes learning algorithms requiring to store and iterate multiple times over the whole training set, or for which the amount of computations per example increases as more examples are seen. This strongly suggest the use of on-line learning.
- A learning algorithm that can discover concepts that can be shared easily among multiple tasks and multiple modalities (multi-task learning), and that can take advantage of large amounts of unlabeled data (semi-supervised learning).

3 Learning Architectures, Shallow and Deep

3.1 Architecture Types

In this section, we define the notions of shallow and deep architectures. An informal discussion of their relative advantages and disadvantage is presented using examples. A more formal discussion of the limitations of shallow architectures with local smoothness (which includes most modern kernel methods) is given in the next section.

Following the tradition of the classic book *Perceptrons* [Minsky and Papert, 1969], it is instructive to categorize different types of learning architectures and to analyze their limitations and advantages. To fix ideas, consider the simple case of classification in which a discrete label is produced by the learning machine $y = f(x, w)$, where x is the input pattern, and w a parameter which indexes the family of functions $\mathcal{F}$ that can be implemented by the architecture $\mathcal{F} = \{f(\cdot, w), w \in \mathcal{W}\}$.

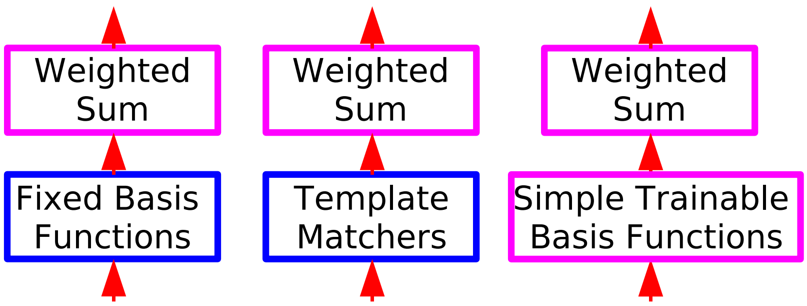

Figure 1: Different types of shallow architectures. (a) Type-1: fixed preprocessing and linear predictor; (b) Type-2: template matchers and linear predictor (kernel machine); (c) Type-3: simple trainable basis functions and linear predictor (neural net with one hidden layer, RBF network).

Traditional Perceptrons, like many currently popular learning models, are *shallow architectures*. Different types of shallow architectures are represented in figure 1. Type-1 architectures have fixed preprocessing in the first layer (e.g., Perceptrons). Type-2 architectures have template matchers in the first layer (e.g., kernel machines). Type-3 architectures have simple trainable basis functions in the first layer (e.g., neural net with one hidden layer, RBF network). All three have a linear transformation in the second layer.

3.1.1 Shallow Architecture Type 1

Fixed pre-processing plus linear predictor, figure 1(a): The simplest shallow architecture is composed of a fixed preprocessing layer (sometimes called features or basis functions), followed by a linear predictor. The type of linear predictor used, and the way it is trained is unspecified (maximum-margin, logistic regression, Perceptron, squared error regression....). The family $\mathcal{F}$ is linearly parameterized in the parameter vector: $f(x) = \sum_{i=1}^{k} w_i \phi_i(x)$. This type of architecture is widely used in practical applications. Since the pre-processing is fixed (and hand-crafted), it is necessarily task-specific in practice. It is possible to imagine a shallow type-1 machine that would parameterize the complete AI-set. For example, we could imagine a machine in which each feature is a member of the AI-set, hence each particular member of the AI-set can be represented with a weight vector containing all zeros, except for a single 1 at the right place. While there probably exist more compact ways to linearly parameterize the entire AI-set, the number of necessary features would surely be prohibitive. More importantly, we do not know explicitly the functions of the AI-set, so this is not practical.

3.1.2 Shallow Architecture Type 2

Template matchers plus linear predictor, figure 1(b): Next on the scale of adaptability is the traditional kernel machine architecture. The preprocessing is a vector of values resulting from the application of a kernel function $K(x, x_i)$ to each training sample $f(x) = b + \sum_{i=1}^{n} \alpha_i K(x, x_i)$, where n is the number of training samples, the parameter w contains all the α_i and the bias b. In effect, the first layer can be seen as a series of template matchers in which the templates are the training samples. Type-2 architectures can be seen as special forms of Type-1 architectures in which the features are data-dependent, which is to say $\phi_i(x) = K(x, x_i)$. This is a simple form of unsupervised learning, for the first layer. Through the famous *kernel trick* (see [Schölkopf et al., 1999]), Type-2 architectures can be seen as a compact way of representing Type-1 architectures, including some that may be too large to be practical. If the kernel function satisfies the Mercer condition it can be expressed as an inner product between feature vectors $K_\phi(x, x_i) =< \phi(x), \phi(x_i) >$, giving us a linear relation between the parameter vectors in both formulations: w for Type-1 architectures is $\sum_i \alpha_i \phi(x_i)$. A very attractive feature of such architectures is that for several common loss functions (e.g., squared error, margin loss) training them involves a convex optimization program. While these properties are largely perceived as the magic behind kernel methods, they should not distract us from the fact that the first layer of a kernel machine is often just a series of template matchers. In most kernel machines, the kernel is used as a kind of template matchers, but other choices are possible. Using task-specific prior knowledge, one can design a kernel that incorporates the right abstractions for the task. This comes at the cost of lower efficiency in terms of human labor. When a kernel acts like a template matcher, we call it *local*: $K(x, x_i)$ discriminates between values of x that are near x_i and those that are not. Some of the mathematical results in this chapter focus on the Gaussian kernel, where nearness corresponds to small Euclidean distance. One could say that one of the main issues with kernel machine with local kernels is that they are *little more than template matchers*. It is possible to use kernels that are non-local yet not task-specific, such as the linear kernels and polynomial kernels. However, most practitioners have been prefering linear kernels or local kernels. Linear kernels are type-1 shallow architectures, with their obvious limitations. Local kernels have been popular because they make intuitive sense (it is easier to insert prior knowledge), while polynomial kernels tend to generalize very poorly when extrapolating (e.g., grossly overshooting). The smoothness prior implicit in local kernels is quite reasonable for a lot of the applications that have been considered, whereas the prior implied by polynomial kernels is less clear. Learning the kernel would move us to Type-3 shallow architectures or deep architectures described below.

3.1.3 Shallow Architecture Type 3

Simple trainable basis functions plus linear predictor, figure 1(c): In Type-3 shallow architectures, the first layer consists of simple basis functions that are *trainable through supervised learning*. This can improve the efficiency of the function representation, by tuning the basis functions to a task. Simple trainable basis functions include linear combinations followed by point-wise non-linearities and Gaussian radial-basis functions (RBF). Traditional neural networks with one hidden layer, and RBF networks belong to that category. Kernel machines in which the kernel function is learned (and simple) also belong to the shallow Type-3 category. Many boosting algorithms belong to this class as well. Unlike with Types 1 and 2, the output is a non-linear function of the parameters to be learned. Hence the loss functions minimized by learning are likely to be non-convex in the parameters. The definition of Type-3 architectures is somewhat fuzzy, since it relies on the ill-defined concept of "simple" parameterized basis function.

We should immediately emphasize that the boundary between the various categories is somewhat fuzzy. For example, training the hidden layer of a one-hidden-layer neural net (a type-3 shallow architecture) is a non-convex problem, but one could imagine constructing a hidden layer so large that all possible hidden unit functions would be present from the start. Only the output layer would need to be trained. More specifically, when the number of hidden units becomes very large, and an L2 regularizer is used on the output weights, such a neural net becomes a kernel machine, whose kernel has a simple form that can be computed analytically [Bengio et al., 2006b]. If we use the margin loss this becomes an SVM with a particular kernel. Although convexity is only achieved in the mathematical limit of an infinite number of hidden units, we conjecture that optimization of single-hidden-layer neural networks becomes easier as the number of hidden units becomes larger. If single-hidden-layer neural nets have any advantage over SVMs, it is that they can, in principle, achieve similar performance with a smaller first layer (since the parameters of the first layer can be optimized for the task).

Note also that our mathematical results on local kernel machines are limited in scope, and most are derived for specific kernels such as the Gaussian kernel, or for local kernels (in the sense of $K(u, v)$ being near zero when $||u - v||$ becomes large). However, the arguments presented below concerning the shallowness of kernel machines are more general.

3.1.4 Deep Architectures

Deep architectures are *compositions of many layers of adaptive non-linear components*, in other words, they are cascades of parameterized non-linear modules that contain trainable parameters at all levels. Deep architectures allow the representation of wide families of functions in a more compact form than shallow architectures, because they can trade space for time (or breadth for depth) while making the time-space product smaller, as discussed below. The outputs of the intermediate layers are akin to intermediate results on the way to computing the final output. Features produced by the lower layers represent lower-level abstractions, that are combined to form high-level features at the next layer, representing higher-level abstractions.

3.2 The Depth-Breadth Tradeoff

Any specific function can be implemented by a suitably designed shallow architecture or by a deep architecture. Similarly, when parameterizing a family of functions, we have the choice between shallow or deep architectures. The important questions are: 1. how large is the corresponding architecture (with how many parameters, how much computation to produce the output); 2. how much manual labor is involved in specializing the architecture to the task.

Using a number of examples, we shall demonstrate that deep architectures are often more efficient (in terms of number of computational components and parameters) for representing common functions. Formal analyses of the computational complexity of shallow circuits can be found in Håstad [1987] or Allender [1996]. They point in the same direction: shallow circuits are much less expressive than deep ones.

Let us first consider the task of adding two N-bit binary numbers. The most natural circuit involves adding the bits pair by pair and propagating the carry. The carry propagation takes $O(N)$ steps, and also $O(N)$ hardware resources. Hence the most natural architecture for binary addition is a deep one, with $O(N)$ layers and $O(N)$ elements. A shallow architecture can implement any boolean formula expressed in disjunctive normal form (DNF), by computing the minterms (AND functions) in the first layer, and the subsequent OR function using a linear classifier (a threshold gate) with a low threshold. Unfortunately, even for simple boolean operations such as binary addition and multiplication, the number of terms can be extremely large (up to $O(2^N)$ for N-bit inputs in the worst case). The computer industry has in fact devoted a considerable amount of effort to optimize the implementation of exponential boolean functions, but the largest it can put on a single chip has only about 32 input bits (a 4-Gbit RAM chip, as of 2006). This is why practical digital circuits, e.g., for adding or multiplying two numbers are built with multiple layers of logic gates: their 2-layer implementation (akin to a lookup table) would be prohibitively expensive. See [Utgoff and Stracuzzi, 2002] for a previous discussion of this question in the context of learning architectures.

Another interesting example is the boolean parity function. The N-bit boolean parity function can be implemented in at least five ways:

(1) with N daisy-chained XOR gates (an N-layer architecture or a recurrent circuit with one XOR gate and N time steps);

(2) with $N-1$ XOR gates arranged in a tree (a $\log_2 N$ layer architecture), for a total of $O(N \log N)$ components;

(3) a DNF formula with $O(2^N)$ minterms (two layers).

Architecture 1 has high depth and low breadth (small amount of computing elements), architecture 2 is a good tradeoff between depth and breadth, and architecture 3 has high breadth and low depth. If one allows the use of multi-input binary threshold gates (linear classifiers) in addition to traditional logic gates, two more architectures are possible [Minsky and Papert, 1969]:

(4) a 3-layer architecture constructed as follows. The first layer has N binary threshold gates (linear classifiers) in which unit i adds the input bits and subtracts i, hence computing the predicate $x_i = (\text{SUM\_OF\_BITS} \geq i)$. The second layer contains $(N-1)/2$ AND gates that compute $(x_i AND(NOTX_{i+1}))$ for all i that are odd. The last layer is a simple OR gate.

(5) a 2-layer architecture in which the first layer is identical to that of the 3-layer architecture above, and the second layer is a linear threshold gate (linear classifier) where the weight for input x_i is equal to $(-2)^i$.

The fourth architecture requires a dynamic range (accuracy) on the weight linear in N, while the last one requires a dynamic range exponential in N. A proof that N-bit parity requires $O(2^N)$ gates to be represented by a depth-2 boolean circuit (with AND, NOT and OR gates) can be found in Ajtai [1983]. In theorem 4 (section 4.1.1) we state a similar result for learning architectures: an exponential number of terms is required with a Gaussian kernel machine in order to represent the parity function. In many instances, space (or breadth) can be traded for time (or depth) with considerable advantage.

These negative results may seem reminiscent of the classic results in Minsky and Papert's book Perceptrons [Minsky and Papert, 1969]. This should come as no surprise: shallow architectures (particularly of type 1 and 2) fall into Minsky and Papert's general definition of a Perceptron and are subject to many of its limitations.

Another interesting example in which adding layers is beneficial is the fast Fourier transform algorithm (FFT). Since the discrete Fourier transform is a linear operation, it can be performed by a matrix multiplication with N^2 complex multiplications, which can all be performed in parallel, followed by $O(N^2)$ additions to collect the sums. However the FFT algorithm can reduce the total cost to $\frac{1}{2}N\log_2 N$, multiplications, with the tradeoff of requiring $\log_2 N$ sequential steps involving $\frac{N}{2}$ multiplications each. This example shows that, even with linear functions, adding layers allows us to take advantage of the intrinsic regularities in the task.

Because each variable can be either absent, present, or negated in a minterm, there are $M = 3^N$ different possible minterms when the circuit has N inputs. The set of all possible DNF formulae with k minterms and N inputs has $C(M,k)$ elements (the number of combinations of k elements from M). Clearly that set (which is associated with the set of functions representable with k minterms) grows very fast with k. Going from $k-1$ to k minterms increases the number of combinations by a factor $(M-k)/k$. When k is not close to M, the size of the set of DNF formulae is exponential in the number of inputs N. These arguments would suggest that only an exponentially (in N) small fraction of all boolean functions require a less than exponential number of minterms.

We claim that most functions that can be represented compactly by deep architectures cannot be represented by a compact shallow architecture. Imagine representing the logical operations over K layers of a logical circuit into a DNF formula. The operations performed by the gates on each of the layers are likely to get combined into a number of minterms that could be exponential in the original number of layers. To see this, consider a K layer logical circuit where every odd layer has AND gates (with the option of negating arguments) and every even layer has OR gates. Every AND-OR consecutive layers corresponds to a sum of products in modulo-2 arithmetic. The whole circuit is the composition of $K/2$ such sums of products, and it is thus a deep *factorization* of a formula. In general, when a factored representation is expanded into a single sum of products, one gets a number of terms that can be exponential in the number of levels. A similar phenomenon explains why most compact DNF formulae require an exponential number of terms when written as a Conjuctive Normal Form (CNF) formula. A survey of more general results in computational complexity of boolean circuits can be found in Allender [1996]. For example, Håstad [1987] show that for all k, there are depth $k + 1$ circuits of linear size that require exponential size to simulate with depth k circuits. This implies that *most functions representable compactly with a deep architecture would require a very large number of components if represented with a shallow one*. Hence restricting ourselves to shallow architectures unduly limits the spectrum of functions that can be represented compactly and learned efficiently (at least in a statistical sense). In particular, highly-variable functions (in the sense of having high frequencies in their Fourier spectrum) are difficult to represent with a circuit of depth 2 [Linial et al., 1993]. The results that we present in section 4 yield a similar conclusion: representing highly-variable functions with a Gaussian kernel machine is very inefficient.

3.3 The Limits of Matching Global Templates

Before diving into the formal analysis of local models, we compare the kernel machines (Type-2 architectures) with deep architectures using examples. One of the fundamental problems in pattern recognition is how to handle intra-class variability. Taking the example of letter recognition, we can picture the set of all the possible images of the letter 'E' on a 20×20 pixel grid as a set of continuous manifolds in the pixel space (e.g., a manifold for lower case and one for cursive). The E's on a manifold can be continuously morphed into each other by following a path on the manifold. The dimensionality of the manifold at one location corresponds to the number of independent distortions that can can be applied to an image while preserving its category. For handwritten letter categories, the manifold has a high dimension: letters can be distorted using affine transforms (6 parameters), distorted using an elastic sheet deformation (high dimension), or modified so as to cover the range of possible writing styles, shapes, and stroke widths. Even for simple character images, the manifold is very non-linear, with high curvature. To convince ourselves of that, consider the shape of the letter 'W'. Any pixel in the lower half of the image will go from white to black and white again four times as the W is shifted horizontally within the image frame from left to right. This is the sign of a highly non-linear surface. Moreover, manifolds for other character categories are closely intertwined. Consider the shape of a capital U and an O at the same location.

They have many pixels in common, many more pixels in fact than with a shifted version of the same U. Hence the distance between the U and O manifolds is smaller than the distance between two U's shifted by a few pixels. Another insight about the high curvature of these manifolds can be obtained from the example in figure 4: the tangent vector of the horizontal translation manifold changes abruptly as we translate the image only one pixel to the right, indicating high curvature. As discussed in section 4.2, many kernel algorithms make an implicit assumption of a locally smooth function (e.g., locally linear in the case of SVMs) *around each training example* x_i. Hence a high curvature implies the necessity of a large number of training examples in order to cover all the desired twists and turns with locally constant or locally linear pieces.

This brings us to what we perceive as the main shortcoming of template-based methods: a very large number of templates may be required in order to cover each manifold with enough templates to avoid misclassifications. Furthermore, the number of necessary templates can grow exponentially with the intrinsic dimension of a class-invariant manifold. The only way to circumvent the problem with a Type-2 architecture is to design similarity measures for matching templates (kernel functions) such that two patterns that are on the same manifold are deemed similar. Unfortunately, devising such similarity measures, even for a problem as basic as digit recognition, has proved difficult, despite almost 50 years of active research. Furthermore, if such a good task-specific kernel were finally designed, it may be inapplicable to other classes of problems.

To further illustrate the situation, consider the problem of detecting and identifying a simple motif (say, of size $S = 5 \times 5$ pixels) that can appear at D different locations in a uniformly white image with N pixels (say 10^6 pixels). To solve this problem, a simple kernel-machine architecture would require one template of the motif for each possible location. This requires $N.D$ elementary operations. An architecture that allows for *spatially local* feature detectors would merely require $S.D$ elementary operations. We should emphasize that this spatial locality (feature detectors that depend on pixels within a limited radius in the image plane) is distinct from the locality of kernel functions (feature detectors that produce large values only for input vectors that are within a limited radius in the input vector space). In fact, spatially local feature detectors have non-local response in the space of input vectors, since their output is independent of the input pixels they are not connected to.

A slightly more complicated example is the task of detecting and recognizing a pattern composed of two different motifs. Each motif occupies S pixels, and can appear at D different locations independently of each other. A kernel machine would need a separate template for each possible occurrence of the two motifs, i.e., $N.D^2$ computing elements. By contrast, a properly designed Type-3 architecture would merely require a set of local feature detectors for all the positions of the first motifs, and a similar set for the second motif. The total amount of elementary operations is a mere $2.S.D$. We do not know of any kernel that would allow to efficiently handle compositional structures.

An even more dire situation occurs if the background is not uniformly white, but can contain random clutter. A kernel machine would probably need many different templates containing the desired motifs on top of many different backgrounds. By contrast, the locally-connected deep architecture described in the previous paragraph will handle this situation just fine. We have verified this type of behavior experimentally (see examples in section 6).

These thought experiments illustrate the limitations of kernel machines due to the fact that their first layer is restricted to matching the incoming patterns with global templates. By contrast, the Type-3 architecture that uses spatially local feature detectors handles the position jitter and the clutter easily and efficiently. Both architectures are shallow, but while each kernel function is activated in a small area of the input space, the spatially local feature detectors are activated by a huge $(N - S)$-dimensional subspace of the input space (since they only look at S pixels). Deep architectures with spatially-local feature detectors are even more efficient (see Section 6). Hence the limitations of kernel machines are not just due to their shallowness, but also to the *local* character of their response function (local in input space, not in the space of image coordinates).

4 Fundamental Limitation of Local Learning

A large fraction of the recent work in statistical machine learning has focused on non-parametric learning algorithms which rely solely, explicitly or implicitly, on a *smoothness prior*. A smoothness prior favors functions f such that when $x \approx x'$, $f(x) \approx f(x')$. Additional prior knowledge is expressed by choosing the space of the data and the particular notion of similarity between examples (typically expressed as a kernel function). This class of learning algorithms includes most instances of the kernel machine algorithms [Schölkopf et al., 1999], such as Support Vector Machines (SVMs) [Boser et al., 1992, Cortes and Vapnik, 1995] or Gaussian processes [Williams and Rasmussen, 1996], but also unsupervised learning algorithms that attempt to capture the manifold structure of the data, such as Locally Linear Embedding [Roweis and Saul, 2000], Isomap [Tenenbaum et al., 2000], kernel PCA [Schölkopf et al., 1998], Laplacian Eigenmaps [Belkin and Niyogi, 2003], Manifold Charting [Brand, 2003], and *spectral clustering* algorithms (see Weiss [1999] for a review). More recently, there has also been much interest in non-parametric *semi-supervised learning algorithms*, such as Zhu et al. [2003], Zhou et al. [2004], Belkin et al. [2004], Delalleau et al. [2005], which also fall in this category, and share many ideas with manifold learning algorithms.

Since this is a large class of algorithms and one that continues to attract attention, it is worthwhile to investigate its limitations. Since these methods share many characteristics with classical non-parametric statistical learning algorithms – such as the k-nearest neighbors and the Parzen windows regression and density estimation algorithms [Duda and Hart, 1973] – which have been shown to suffer from the so-called *curse of dimensionality*, it is logical to investigate the following question: to what extent do these modern kernel methods suffer from a similar problem? See [Härdle et al., 2004] for a recent and easily accessible exposition of the curse of dimensionality for classical non-parametric methods.

To explore this question, we focus on algorithms in which the learned function is expressed in terms of a linear combination of kernel functions applied on the training examples:

$$f(x) = b + \sum_{i=1}^{n} \alpha_i K_D(x, x_i) \tag{1}$$

where we have included an optional bias term b. The set $D = \{z_1, \ldots, z_n\}$ contains training examples $z_i = x_i$ for unsupervised learning, $z_i = (x_i, y_i)$ for supervised learning. Target value y_i can take a special missing value for semi-supervised learning. The α_i's are scalars chosen by the learning algorithm using D, and $K_D(\cdot, \cdot)$ is the kernel function, a symmetric function (sometimes expected to be positive semi-definite), which may be chosen by taking into account all the x_i's. A typical kernel function is the Gaussian kernel,

$$K_\sigma(u, v) = e^{-\frac{1}{\sigma^2}||u-v||^2}, \tag{2}$$

with the width σ controlling how local the kernel is. See Bengio et al. [2004] to see that LLE, Isomap, Laplacian eigenmaps and other spectral manifold learning algorithms such as spectral clustering can be generalized and written in the form of eq. 1 for a test point x, but with a different kernel (that is data-dependent, generally performing a kind of normalization of a data-independent kernel).

One obtains the consistency of classical non-parametric estimators by appropriately varying the hyper-parameter that controls the locality of the estimator as n increases. Basically, the kernel should be allowed to become more and more local, so that statistical bias goes to zero, but the effective number of examples involved in the estimator at x (equal to k for the k-nearest neighbor estimator) should increase as n increases, so that statistical variance is also driven to 0. For a wide class of kernel regression estimators, the unconditional variance and squared bias can be shown to be written as follows [Härdle et al., 2004]:

$$\text{expected error} = \frac{C_1}{n\sigma^d} + C_2\sigma^4,$$

with C_1 and C_2 not depending on n nor on the dimension d. Hence an optimal bandwidth is chosen proportional to $n^{\frac{-1}{4+d}}$, and the resulting generalization error (not counting the noise) converges in $n^{-4/(4+d)}$, which becomes very slow for large d. Consider for example the increase in number of examples required to get the same level of error, in 1 dimension versus d dimensions. If n_1 is the number of examples required to get a particular level of error, to get the same level of error in d dimensions requires on the order of $n_1^{(4+d)/5}$ examples, i.e., the *required number of examples is exponential in d*. For the k-nearest neighbor classifier, a similar result is obtained [Snapp and Venkatesh, 1998]:

$$\text{expected error} = E_\infty + \sum_{j=2}^{\infty} c_j n^{-j/d}$$

where E_∞ is the asymptotic error, d is the dimension and n the number of examples.

Note however that, if the data distribution is concentrated on a lower dimensional manifold, it is the *manifold dimension* that matters. For example, when data lies on a smooth lower-dimensional manifold, the only dimensionality that matters to a k-nearest neighbor classifier is the dimensionality of the manifold, since it only uses the Euclidean distances between the near neighbors. Many unsupervised and semi-supervised learning algorithms rely on a graph with one node per example, in which nearby examples are connected with an edge weighted by the Euclidean distance between them. If data lie on a low-dimensional manifold then geodesic distances in this graph approach geodesic distances on the manifold [Tenenbaum et al., 2000], as the number of examples increases. However, convergence can be exponentially slower for higher-dimensional manifolds.

4.1 Minimum Number of Bases Required

In this section we present results showing the number of required bases (hence of training examples) of a kernel machine with Gaussian kernel may grow linearly with the number of variations of the target function that must be captured in order to achieve a given error level.

4.1.1 Result for Supervised Learning

The following theorem highlights the number of sign changes that a Gaussian kernel machine can achieve, when it has k bases (i.e., k support vectors, or at least k training examples).

Theorem 1 (Theorem 2 of Schmitt [2002])**.** *Let $f : \mathbb{R} \to \mathbb{R}$ computed by a Gaussian kernel machine (eq. 1) with k bases (non-zero α_i's). Then f has at most $2k$ zeros.*

We would like to say something about kernel machines in $\mathbb{R}^d$, and we can do this simply by considering a straight line in $\mathbb{R}^d$ and the number of sign changes that the solution function f can achieve along that line.

Corollary 2. *Suppose that the learning problem is such that in order to achieve a given error level for samples from a distribution P with a Gaussian kernel machine (eq. 1), then f must change sign at least $2k$ times along some straight line (i.e., in the case of a classifier, the decision surface must be crossed at least $2k$ times by that straight line). Then the kernel machine must have at least k bases (non-zero α_i's).*

A proof can be found in Bengio et al. [2006a].

Example 3. *Consider the decision surface shown in figure 2, which is a sinusoidal function. One may take advantage of the global regularity to learn it with few parameters (thus requiring few examples), but with an affine combination of Gaussians, corollary 2 implies one would need at least $\lceil \frac{m}{2} \rceil = 10$ Gaussians. For more complex tasks in higher dimension, the complexity of the decision surface could quickly make learning impractical when using such a local kernel method.*

Of course, one only seeks to approximate the decision surface S, and does not necessarily need to learn it perfectly: corollary 2 says nothing about the existence of an easier-to-learn decision surface approximating S. For instance, in the example of

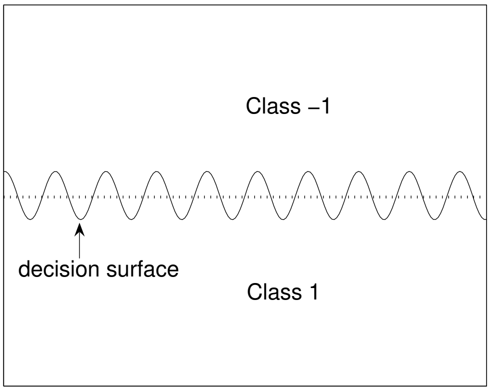

Figure 2: The dotted line crosses the decision surface 19 times: one thus needs at least 10 Gaussians to learn it with an affine combination of Gaussians with same width.

figure 2, the dotted line could turn out to be a good enough estimated decision surface if most samples were far from the true decision surface, and this line can be obtained with only two Gaussians.

The above theorem tells us that in order to represent a function that locally varies a lot, in the sense that its sign along a straight line changes many times, a Gaussian kernel machine requires many training examples and many computational elements. Note that it says nothing about the dimensionality of the input space, but we might expect to have to learn functions that vary more when the data is high-dimensional. The next theorem confirms this suspicion in the special case of the d-bits parity function:

$$\text{parity}: \ (b_1, \ldots, b_d) \in \{0,1\}^d \mapsto \begin{cases} 1 \text{ if } \sum_{i=1}^d b_i \text{ is even} \\ -1 \text{ otherwise.} \end{cases}$$

Learning this apparently simple function with Gaussians centered on points in $\{0,1\}^d$ is actually difficult, in the sense that it requires a number of Gaussians exponential in d (for a fixed Gaussian width). Note that our corollary 2 does not apply to the d-bits parity function, so it represents another type of local variation (not along a line). However, it is also possible to prove a very strong result for parity.

Theorem 4. *Let $f(x) = b + \sum_{i=1}^{2^d} \alpha_i K_\sigma(x_i, x)$ be an affine combination of Gaussians with same width σ centered on points $x_i \in X_d$. If f solves the parity problem, then there are at least 2^{d-1} non-zero coefficients α_i.*

A proof can be found in Bengio et al. [2006a].

The bound in theorem 4 is tight, since it is possible to solve the parity problem with exactly 2^{d-1} Gaussians and a bias, for instance by using a negative bias and putting a positive weight on each example satisfying $\mathrm{parity}(x_i) = 1$. When trained to learn the parity function, a SVM may learn a function that looks like the opposite of the parity on test points (while still performing optimally on training points), but it is an artifact of the specific geometry of the problem, and only occurs when the training set size is appropriate compared to $|X_d| = 2^d$ (see Bengio et al. [2005] for details). Note that if the centers of the Gaussians are not restricted anymore to be points in the training set (i.e., a Type-3 shallow architecture), it is possible to solve the parity problem with only $d + 1$ Gaussians and no bias [Bengio et al., 2005].

One may argue that parity is a simple discrete toy problem of little interest. But even if we have to restrict the analysis to discrete samples in $\{0, 1\}^d$ for mathematical reasons, the parity function can be extended to a smooth function on the $[0, 1]^d$ hypercube depending only on the continuous sum $b_1 + \ldots + b_d$. Theorem 4 is thus a basis to argue that the number of Gaussians needed to learn a function with many variations in a continuous space may scale linearly with the number of these variations, and thus possibly exponentially in the dimension.

4.1.2 Results for Semi-Supervised Learning

In this section we focus on algorithms of the type described in recent papers [Zhu et al., 2003, Zhou et al., 2004, Belkin et al., 2004, Delalleau et al., 2005], which are graph-based, non-parametric, semi-supervised learning algorithms. Note that transductive SVMs [Joachims, 1999], which are another class of semi-supervised algorithms, are already subject to the limitations of corollary 2. The graph-based algorithms we consider here can be seen as minimizing the following cost function, as shown in Delalleau et al. [2005]:

$$C(\hat{Y}) = \|\hat{Y}_l - Y_l\|^2 + \mu \hat{Y}^\top L \hat{Y} + \mu\epsilon \|\hat{Y}\|^2 \tag{3}$$

with $\hat{Y} = (\hat{y}_1, \ldots, \hat{y}_n)$ the estimated labels on both labeled and unlabeled data, and L the (un-normalized) graph Laplacian matrix, derived through $L = D^{-1/2} W D^{-1/2}$ from a kernel function K between points such that the Gram matrix W, with $W_{ij} = K(x_i, x_j)$, corresponds to the weights of the edges in the graph, and D is a diagonal matrix containing in-degree: $D_{ii} = \sum_j W_{ij}$. Here, $\hat{Y}_l = (\hat{y}_1, \ldots, \hat{y}_l)$ is the vector of estimated labels on the l labeled examples, whose known labels are given by $Y_l = (y_1, \ldots, y_l)$, and one may constrain $\hat{Y}_l = Y_l$ as in Zhu et al. [2003] by letting $\mu \to 0$. We define a region with constant label as a connected subset of the graph where all nodes x_i have the same estimated label (sign of $\hat{y}_i$), and such that no other node can be added while keeping these properties.

Minimization of the cost criterion of eq. 3 can also be seen as a *label propagation* algorithm, i.e., labels are spread around labeled examples, with nearness being defined by the structure of the graph, i.e., by the kernel. An intuitive view of label propagation suggests that a region of the manifold near a labeled (e.g., positive) example will be entirely labeled positively, as the example spreads its influence by propagation on the graph representing the underlying manifold. Thus, the number of regions with constant label should be on the same order as (or less than) the number of labeled examples. This is easy to see in the case of a sparse Gram matrix W. We define a region with constant label as a connected subset of the graph where all nodes x_i have the same estimated label (sign of $\hat{y}_i$), and such that no other node can be added while keeping these properties. The following proposition then holds (note that it is also true, but trivial, when W defines a fully connected graph).

Proposition 5. *After running a label propagation algorithm minimizing the cost of eq. 3, the number of regions with constant estimated label is less than (or equal to) the number of labeled examples.*

A proof can be found in Bengio et al. [2006a]. The consequence is that we will need at least as many labeled examples as there are variations in the class, as one moves by small steps in the neighborhood graph from one contiguous region of same label to another. Again we see the same type of non-parametric learning algorithms with a local kernel, here in the case of semi-supervised learning: we may need about as many labeled examples as there are variations, even though an arbitrarily large number of these variations could have been characterized more efficiently than by their enumeration.

4.2 Smoothness versus Locality: Curse of Dimensionality

Consider a Gaussian SVM and how that estimator changes as one varies σ, the hyper-parameter of the Gaussian kernel. For large σ one would expect the estimated function to be very smooth, whereas for small σ one would expect the estimated function to be very local, in the sense discussed earlier: the near neighbors of x have dominating influence in the shape of the predictor at x.

The following proposition tells us what happens when σ is large, or when we consider what a ball whose radius is small compared to σ.

Proposition 6. *For the Gaussian kernel classifier, as σ increases and becomes large compared with the diameter of the data, within the smallest sphere containing the data the decision surface becomes linear if $\sum_i \alpha_i = 0$ (e.g., for SVMs), or else the normal vector of the decision surface becomes a linear combination of two sphere surface normal vectors, with each sphere centered on a weighted average of the examples of the corresponding class.*

A proof can be found in Bengio et al. [2006a].

Note that with this proposition we see clearly that when σ becomes large, a kernel classifier becomes non-local (it approaches a linear classifier). However, this non-locality is at the price of constraining the decision surface to be very smooth, making it difficult to model highly varying decision surfaces. This is the essence of the trade-off between smoothness and locality in many similar non-parametric models (including the classical ones such as k-nearest-neighbor and Parzen windows algorithms).

Now consider in what senses a Gaussian kernel machine is local (thinking about σ small). Consider a test point x that is near the decision surface. We claim that the orientation of the decision surface is dominated by the neighbors x_i of x in the training set, making the predictor *local in its derivative*. If we consider the α_i fixed (i.e., ignoring their dependence on the training x_i's), then it is obvious that the prediction $f(x)$ is dominated by the near neighbors x_i of x, since $K(x, x_i) \to 0$ quickly when $||x - x_i||/\sigma$ becomes large. However, the α_i can be influenced by all the x_j's. The following proposition skirts that issue by looking at the first derivative of f.

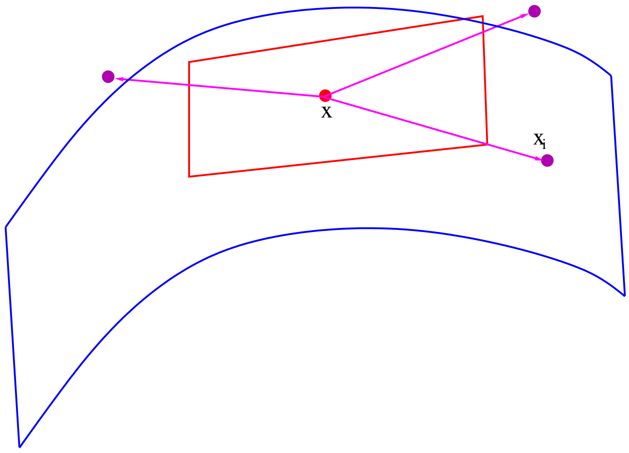

Figure 3: For local manifold learning algorithms such as LLE, Isomap and kernel PCA, the manifold tangent plane at x is in the span of the difference vectors between test point x and its neighbors x_i in the training set. This makes these algorithms sensitive to the curse of dimensionality, when the manifold is high-dimensional and not very flat.

Proposition 7. *For the Gaussian kernel classifier, the normal of the tangent of the decision surface at x is constrained to approximately lie in the span of the vectors $(x - x_i)$ with $||x - x_i||$ not large compared to σ and x_i in the training set.*

Sketch of the Proof

The estimator is $f(x) = \sum_i \alpha_i K(x, x_i)$. The normal vector of the tangent plane at a point x of the decision surface is

$$\frac{\partial f(x)}{\partial x} = \sum_i \alpha_i \frac{(x_i - x)}{\sigma^2} K(x, x_i).$$

Each term is a vector proportional to the difference vector $x_i - x$. This sum is dominated by the terms with $||x - x_i||$ not large compared to σ. We are thus left with $\frac{\partial f(x)}{\partial x}$ approximately in the span of the difference vectors $x - x_i$ with x_i a near neighbor of x. The α_i being only scalars, they only influence the weight of each neighbor x_i in that linear combination. Hence although $f(x)$ can be influenced by x_i far from x, the decision surface near x has a normal vector that is constrained to approximately lie in the span of the vectors $x - x_i$ with x_i near x. Q.E.D.

The constraint of $\frac{\partial f(x)}{\partial x}$ being in the span of the vectors $x - x_i$ for neighbors x_i of x is not strong if the manifold of interest (e.g., the region of the decision surface with high density) has low dimensionality. Indeed if that dimensionality is smaller or equal to the number of dominating neighbors, then there is no constraint at all. However, when modeling complex dependencies involving many factors of variation, the region of interest may have very high dimension (e.g., consider the effect of variations that have arbitrarily large dimension, such as changes in clutter, background , etc. in images). For such a complex highly-varying target function, we also need a very local predictor (σ small) in order to accurately represent all the desired variations. With a small σ, the number of dominating neighbors will be small compared to the dimension of the manifold of interest, making this locality in the derivative a strong constraint, and allowing the following curse of dimensionality argument.

This notion of locality in the sense of the derivative allows us to define a ball around each test point x, containing neighbors that have a dominating influence on $\frac{\partial f(x)}{\partial x}$. Smoothness within that ball constrains the decision surface to be approximately either linear (case of SVMs) or a particular quadratic form (the decision surface normal vector is a linear combination of two vectors defined by the center of mass of examples of each class). Let N be the number of such balls necessary to cover the region Ω where the value of the estimator is desired (e.g., near the target decision surface, in the case of classification problems). Let k be the smallest number such that one needs at least k examples in each ball to reach error level ϵ. The number of examples thus required is kN. To see that N can be exponential in some dimension, consider the maximum radius r of all these balls and the radius R of Ω. If Ω has intrinsic dimension d, then N could be as large as the number of radius-r balls that can tile a d-dimensional manifold of radius R, which is on the order of $\left(\frac{R}{r}\right)^d$.

In Bengio et al. [2005] we present similar results that apply to unsupervised learning algorithms such as non-parametric manifold learning algorithms [Roweis and Saul, 2000, Tenenbaum et al., 2000, Schölkopf et al., 1998, Belkin and Niyogi, 2003]. We find that when the underlying manifold varies a lot in the sense of having high curvature in many places, then a large number of examples is required. Note that the tangent plane of the manifold is defined by the derivatives of the kernel machine function f, for such algorithms. The core result is that the manifold tangent plane at x is dominated by terms associated with the near neighbors of x in the training set (more precisely it is constrained to be in the span of the vectors $x - x_i$, with x_i a neighbor of x). This idea is illustrated in figure 3. In the case of graph-based manifold learning algorithms such as LLE and Isomap, the domination of near examples is perfect (i.e., the derivative is strictly in the span of the difference vectors with the neighbors), because the kernel implicit in these algorithms takes value 0 for the non-neighbors. With such local manifold learning algorithms, one needs to cover the manifold with small enough linear patches with at least $d+1$ examples per patch (where d is the dimension of the manifold). This argument was previously introduced in Bengio and Monperrus [2005] to describe the limitations of neighborhood-based manifold learning algorithms.

An example that illustrates that many interesting manifolds can have high curvature is that of translation of high-contrast images, shown in figure 4. The same argument applies to the other geometric invariances of images of objects.

5 Deep Architectures

The analyzes in the previous sections point to the difficulty of learning *highly-varying functions*. These are functions with a large number of *variations* (twists and turns) in the domain of interest, e.g., they would require a large number of pieces to be well-represented by a piecewise-linear approximation. Since the number of pieces can be

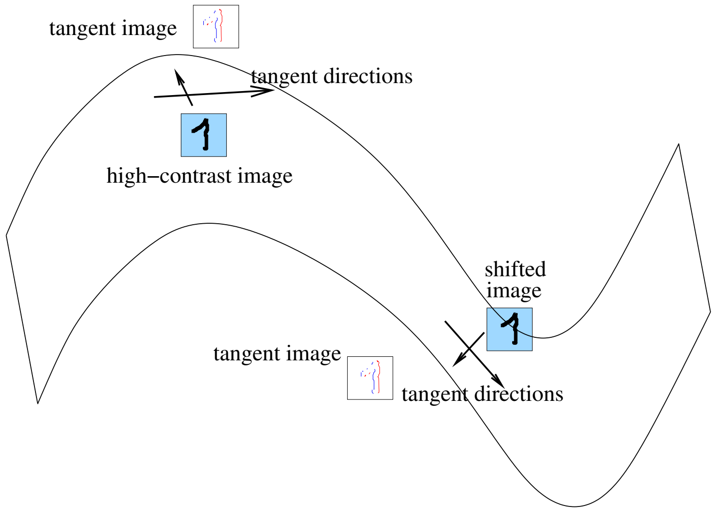

Figure 4: The manifold of translations of a high-contrast image has high curvature. A smooth manifold is obtained by considering that an image is a sample on a discrete grid of an intensity function over a two-dimensional space. The tangent vector for translation is thus a *tangent image*, and it has high values only on the edges of the ink. The tangent plane for an image translated by only one pixel looks similar but changes abruptly since the edges are also shifted by one pixel. Hence the two tangent planes are almost orthogonal, and the manifold has high curvature, which is bad for local learning methods, which must cover the manifold with many small linear patches to correctly capture its shape.

made to grow exponentially with the number of input variables, this problem is directly connected with the well-known curse of dimensionality for classical non-parametric learning algorithms (for regression, classification and density estimation). If the shapes of all these pieces are unrelated, one needs enough examples for each piece in order to generalize properly. However, if these shapes are related and can be predicted from each other, *non-local learning algorithms* have the potential to generalize to pieces not covered by the training set. Such ability would seem necessary for learning in complex domains such as in the AI-set.

One way to represent a highly-varying function compactly (with few parameters) is through the composition of many non-linearities. Such multiple composition of non-linearities appear to grant non-local properties to the estimator, in the sense that the value of $f(x)$ or $f'(x)$ can be strongly dependent on training examples far from x_i while at the same time allowing to capture a large number of variations. We have already discussed parity and other examples (section 3.2) that strongly suggest that the learning of more abstract functions is much more efficient when it is done sequentially, by composing previously learned concepts. When the representation of a concept requires an exponential number of elements, (e.g., with a shallow circuit), the number of training examples required to learn the concept may also be impractical.

Gaussian processes, SVMs, log-linear models, graph-based manifold learning and graph-based semi-supervised learning algorithms can all be seen as shallow architectures. Although multi-layer neural networks with many layers can represent deep circuits, training deep networks has always been seen as somewhat of a challenge. Until very recently, empirical studies often found that deep networks generally performed no better, and often worse, than neural networks with one or two hidden layers [Tesauro, 1992]. A notable exception to this is the convolutional neural network architecture [LeCun et al., 1989, LeCun et al., 1998] discussed in the next section, that has a sparse connectivity from layer to layer. Despite its importance, the topic of deep network training has been somewhat neglected by the research community. However, a promising new method recently proposed by Hinton et al. [2006] is causing a resurgence of interest in the subject.

A common explanation for the difficulty of deep network learning is the presence of local minima or plateaus in the loss function. Gradient-based optimization methods that start from random initial conditions appear to often get trapped in poor local minima or plateaus. The problem seems particularly dire for narrow networks (with few hidden units or with a bottleneck) and for networks with many symmetries (i.e., fully-connected networks in which hidden units are exchangeable). The solution recently introduced by Hinton et al. [2006] for training deep layered networks is based on a greedy, layer-wise unsupervised learning phase . The unsupervised learning phase provides an initial configuration of the parameters with which a gradient-based supervised learning phase is initialized. The main idea of the unsupervised phase is to pair each feed-forward layer with a feed-back layer that attempts to reconstruct the input of the layer from its output. This reconstruction criterion guarantees that most of the information contained in the input is preserved in the output of the layer. The resulting architecture is a so-called Deep Belief Networks (DBN). After the initial unsupervised training of each feed-forward/feed-back pair, the feed-forward half of the network is refined using a gradient-descent based supervised method (back-propagation). This training strategy *holds great promise as a principle to break through the problem of training deep networks*. Upper layers of a DBN are supposed to represent more abstract concepts that explain the input observation x, whereas lower layers extract low-level features from x. Lower layers learn simpler concepts first, and higher layers build on them to learn more abstract concepts. This strategy has not yet been much exploited in machine learning, but it is at the basis of the greedy layer-wise constructive learning algorithm for DBNs. More precisely, each layer is trained in an unsupervised way so as to capture the main features of the distribution it sees as input. It produces an internal representation for its input that can be used as input for the next layer. In a DBN, each layer is trained as a Restricted Boltzmann Machine [Teh and Hinton, 2001] using the Contrastive Divergence [Hinton, 2002] approximation of the log-likelihood gradient. The outputs of each layer (i.e., hidden units) constitute a factored and distributed representation that estimates causes for the input of the layer. After the layers have been thus initialized, a final output layer is added on top of the network (e.g., predicting the class probabilities), and the whole deep network is fine-tuned by a gradient-based optimization of the prediction error. The only difference with an ordinary multi-layer neural network resides in the initialization of the parameters, which is not random, but is performed through unsupervised training of each layer in a sequential fashion.

Experiments have been performed on the MNIST and other datasets to try to understand why the Deep Belief Networks are doing much better than either shallow networks or deep networks with random initialization. These results are reported and discussed in [Bengio et al., 2007]. Several conclusions can be drawn from these experiments, among which the following, of particular interest here:

1. Similar results can be obtained by training each layer as an auto-associator instead of a Restricted Boltzmann Machine, suggesting that a rather general principle has been discovered.

2. Test classification error is significantly improved with such greedy layer-wise unsupervised initialization over either a shallow network or a deep network with the same architecture but with random initialization. In all cases many possible hidden layer sizes were tried, and selected based on validation error.

3. When using a greedy layer-wise strategy that is *supervised* instead of unsupervised, the results are not as good, probably because it is *too greedy*: unsupervised feature learning extracts more information than strictly necessary for the prediction task, whereas greedy supervised feature learning (greedy because it does not take into account that there will be more layers later) extracts less information than necessary, which prematurely scuttles efforts to improve by adding layers.

4. The greedy layer-wise unsupervised strategy helps generalization mostly because it helps the supervised optimization to get started near a better solution.

6 Experiments with Visual Pattern Recognition

One essential question when designing a learning architecture is how to represent invariance. While invariance properties are crucial to any learning task, it is particularly apparent in visual pattern recognition. In this section we consider several experiments in handwriting recognition and object recognition to illustrate the relative advantages and disadvantages of kernel methods, shallow architectures, and deep architectures.

6.1 Representing Invariance

The example of figure 4 shows that the manifold containing all translated versions of a character image has high curvature. Because the manifold is highly varying, a classifier that is invariant to translations (i.e., that produces a constant output when the input moves on the manifold, but changes when the input moves to another class manifold) needs to compute a highly varying function. As we showed in the previous section, template-based methods are inefficient at representing highly-varying functions. The number of such variations may increase exponentially with the dimensionality of the manifolds where the input density concentrates. That dimensionality is the number of dimensions along which samples within a category can vary.

We will now describe two sets of results with visual pattern recognition. The first part is a survey of results obtained with shallow and deep architectures on the MNIST dataset, which contains isolated handwritten digits. The second part analyzes results of experiments with the NORB dataset, which contains objects from five different generic categories, placed on uniform or cluttered backgrounds.

For visual pattern recognition, Type-2 architectures have trouble handling the wide variability of appearance in pixel images that result from variations in pose, illumination, and clutter, unless an impracticably large number of templates (e.g., support vectors) are used. Ad-hoc preprocessing and feature extraction can, of course, be used to mitigate the problem, but at the expense of human labor. Here, we will concentrate on methods that deal with raw pixel data and that integrate feature extraction as part of the learning process.

6.2 Convolutional Networks

Convolutional nets are multi-layer architectures in which the successive layers are designed to learn progressively higher-level features, until the last layer which represents categories. All the layers are trained simultaneously to minimize an overall loss function. Unlike with most other models of classification and pattern recognition, there is no distinct feature extractor and classifier in a convolutional network. All the layers are similar in nature and trained from data in an integrated fashion.

The basic module of a convolutional net is composed of a *feature detection layer* followed by a *feature pooling layer*. A typical convolutional net is composed of one, two or three such detection/pooling modules in series, followed by a classification module. The input state (and output state) of each layer can be seen as a series of two-dimensional retinotopic arrays called feature maps. At layer i, the value c_{ijxy} produced by the j-th feature detection layer at position (x, y) in the j-th feature map is computed by applying a series of convolution kernels w_{ijk} to feature maps in the previous layer (with index $i-1$), and passing the result through a hyperbolic tangent sigmoid function:

$$c_{ijxy} = \tanh\left(b_{ij} + \sum_k \sum_{p=0}^{P_i-1} \sum_{q=0}^{Q_i-1} w_{ijkpq} c_{(i-1),k,(x+p),(y+q)}\right) \tag{4}$$

where P_i and Q_i are the width and height of the convolution kernel. The convolution kernel parameters w_{ijkpq} and the bias b_{ij} are subject to learning. A feature detection layer can be seen as a bank of convolutional filters followed by a point-wise non-linearity. Each filter detects a particular feature at every location on the input. Hence spatially translating the input of a feature detection layer will translate the output but leave it otherwise unchanged. Translation invariance is normally built-in by constraining $w_{ijkpq} = w_{ijkp'q'}$ for all p, p', q, q', i.e., the same parameters are used at different locations.

A feature pooling layer has the same number of features in the map as the feature detection layer that precedes it. Each value in a subsampling map is the average (or the max) of the values in a local neighborhood in the corresponding feature map in the previous layer. That average or max is added to a trainable bias, multiplied by a trainable coefficient, and the result is passed through a non-linearity (e.g., the $\tanh$ function). The windows are stepped without overlap. Therefore the maps of a feature

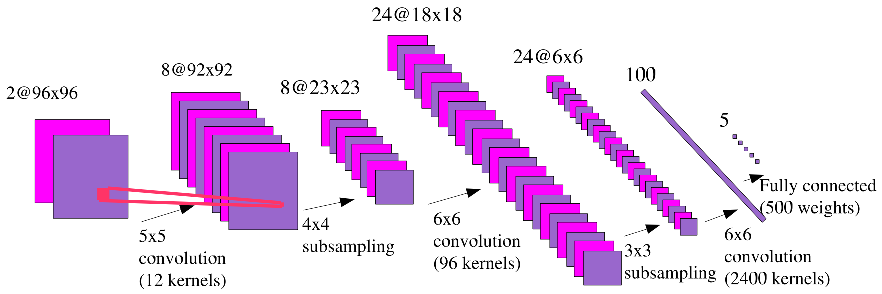

Figure 5: The architecture of the convolutional net used for the NORB experiments. The input is an image pair, the system extracts 8 feature maps of size 92×92, 8 maps of 23×23, 24 maps of 18×18, 24 maps of 6×6, and 100 dimensional feature vector. The feature vector is then transformed into a 5-dimensional vector in the last layer to compute the distance with target vectors.

pooling layer are less than the resolution of the maps in the previous layer. The role of the pooling layer is build a representation that is invariant to small variations of the positions of features in the input. Alternated layers of feature detection and feature pooling can extract features from increasingly large receptive fields, with increasing robustness to irrelevant variabilities of the inputs. The last module of a convolutional network is generally a one- or two-layer neural net.

Training a convolutional net can be performed with stochastic (on-line) gradient descent, computing the gradients with a variant of the back-propagation method. While convolutional nets are deep (generally 5 to 7 layers of non-linear functions), they do not seem to suffer from the convergence problems that plague deep fully-connected neural nets. While there is no definitive explanation for this, we suspect that this phenomenon is linked to the heavily constrained parameterization, as well as to the asymmetry of the architecture.

Convolutional nets are being used commercially in several widely-deployed systems for reading bank check [LeCun et al., 1998], recognizing handwriting for tablet-PC, and for detecting faces, people, and objects in videos in real time.

6.3 The lessons from MNIST

MNIST is a dataset of handwritten digits with 60,000 training samples and 10,000 test samples. Digit images have been size-normalized so as to fit within a 20×20 pixel window, and centered by center of mass in a 28×28 field. With this procedure, the position of the characters vary slightly from one sample to another. Numerous authors have reported results on MNIST, allowing precise comparisons among methods. A small subset of relevant results is listed in table 1. Not all good results on MNIST are listed in the table. In particular, results obtained with deslanted images or with hand-designed feature extractors were left out.

Results are reported with three convolutional net architectures: LeNet-5, LeNet-6, and the subsampling convolutional net of [Simard et al., 2003]. The input field is a 32×32 pixel map in which the 28×28 images are centered. In LeNet-5 [LeCun et al., 1998], the first feature detection layer produces 6 feature maps of size 28×28 using 5×5 convolution kernels. The first feature pooling layer produces 6 14×14 feature maps through a 2×2 subsampling ratio and 2×2 receptive fields. The second feature detection layer produces 16 feature maps of size 10×10 using 5×5 convolution kernels, and is followed by a pooling layer with 2×2 subsampling. The next layer produces 100 feature maps of size 1×1 using 5×5 convolution kernels. The last layer produces 10 feature maps (one per output category). LeNet-6 has a very similar architecture, but the number of feature maps at each level are much larger: 50 feature maps in the first layer, 50 in the third layer, and 200 feature maps in the penultimate layer.

The convolutional net in [Simard et al., 2003] is somewhat similar to the original one in [LeCun et al., 1989] in that there is no separate convolution and subsampling layers. Each layer computes a convolution with a subsampled result (there is no feature pooling operation). Their simple convolutional network has 6 features at the first layer, with 5 by 5 kernels and 2 by 2 subsampling, 60 features at the second layer, also with 5 by 5 kernels and 2 by 2 subsampling, 100 features at the third layer with 5 by 5 kernels, and 10 output units.

The MNIST samples are highly variable because of writing style, but have little variation due to position and scale. Hence, it is a dataset that is particularly favorable for template-based methods. Yet, the error rate yielded by Support Vector Machines with Gaussian kernel (1.4% error) is only marginally better than that of a considerably smaller neural net with a single hidden layer of 800 hidden units (1.6% as reported by [Simard et al., 2003]), and similar to the results obtained with a 3-layer neural net as reported in [Hinton et al., 2006] (1.53% error). The best results on the original MNIST set with a knowledge free method was reported in [Hinton et al., 2006] (0.95% error), using a Deep Belief NetworkBy knowledge-free method, we mean a method that has no prior knowledge of the pictorial nature of the signal. Those methods would produce exactly the same result if the input pixels were scrambled with a fixed permutation.

Convolutional nets use the pictorial nature of the data, and the invariance of categories to small geometric distortions. It is a broad (low complexity) prior, which can be specified compactly (with a short piece of code). Yet it brings about a considerable reduction of the ensemble of functions that can be learned. The best convolutional net on the unmodified MNIST set is LeNet-6, which yields a record 0.60%. As with Hinton's results, this result was obtained by initializing the filters in the first layer using an unsupervised algorithm, prior to training with back-propagation [Ranzato et al., 2006]. The same LeNet-6 trained purely supervised from random initialization yields 0.70% error. A smaller convolutional net, LeNet-5 yields 0.80%. The same network was reported to yield 0.95% in [LeCun et al., 1998] with a smaller number of training iterations.

When the training set is augmented with elastically distorted versions of the training samples, the test error rate (on the original, non-distorted test set) drops significantly. A conventional 2-layer neural network with 800 hidden units yields 0.70% error [Simard

Classifier	Deformations	Error %	Reference
Knowledge-free methods			
2-layer NN, 800 hid. units		1.60	Simard et al. 2003
3-layer NN, 500+300 units		1.53	Hinton et al. 2006
SVM, Gaussian kernel		1.40	Cortes et al. 1992
Unsupervised stacked RBM + backprop		0.95	Hinton et al. 2006
Convolutional networks			
Convolutional network LeNet-5		0.80	Ranzato et al. 2006
Convolutional network LeNet-6		0.70	Ranzato et al. 2006
Conv. net. LeNet-6 + unsup. learning		0.60	Ranzato et al. 2006
Training set augmented with affine distortions			
2-layer NN, 800 hid. units	Affine	1.10	Simard et al. 2003
Virtual SVM, deg. 9 poly	Affine	0.80	DeCoste et al. 2002
Convolutional network,	Affine	0.60	Simard et al. 2003
Training set augmented with elastic distortions			
2-layer NN, 800 hid. units	Elastic	0.70	Simard et al. 2003
SVM Gaussian Ker. + on-line training	Elastic	0.67	this volume, chapter 13
Shape context features + elastic K-NN	Elastic	0.63	Belongie et al. 2002
Convolutional network	Elastic	0.40	Simard et al. 2003
Conv. net. LeNet-6	Elastic	0.49	Ranzato et al. 2006
Conv. net. LeNet-6 + unsup. learning	Elastic	0.39	Ranzato et al. 2006

Table 1: Test error rates of various learning models on the MNIST dataset. Many results obtained with deslanted images or hand-designed feature extractors were left out.

et al., 2003]. While SVMs slightly outperform 2-layer neural nets on the undistorted set, the advantage all but disappears on the distorted set. In this volume, Loosli et al. report 0.67% error with a Gaussian SVM and a sample selection procedure. The number of support vectors in the resulting SVM is considerably larger than 800.

Convolutional nets applied to the elastically distorted set achieve between 0.39% and 0.49% error, depending on the architecture, the loss function, and the number of training epochs. Simard et al. [2003] reports 0.40% with a subsampling convolutional net. Ranzato et al. [2006] report 0.49% using LeNet-6 with random initialization, and 0.39% using LeNet-6 with unsupervised pre-training of the first layer. This is the best error rate ever reported on the original MNIST test set.

Hence a deep network, with small dose of prior knowledge embedded in the architecture, combined with a learning algorithm that can deal with millions of examples, goes a long way towards improving performance. Not only do deep networks yield lower error rates, they are faster to run and faster to train on large datasets than the best kernel methods.

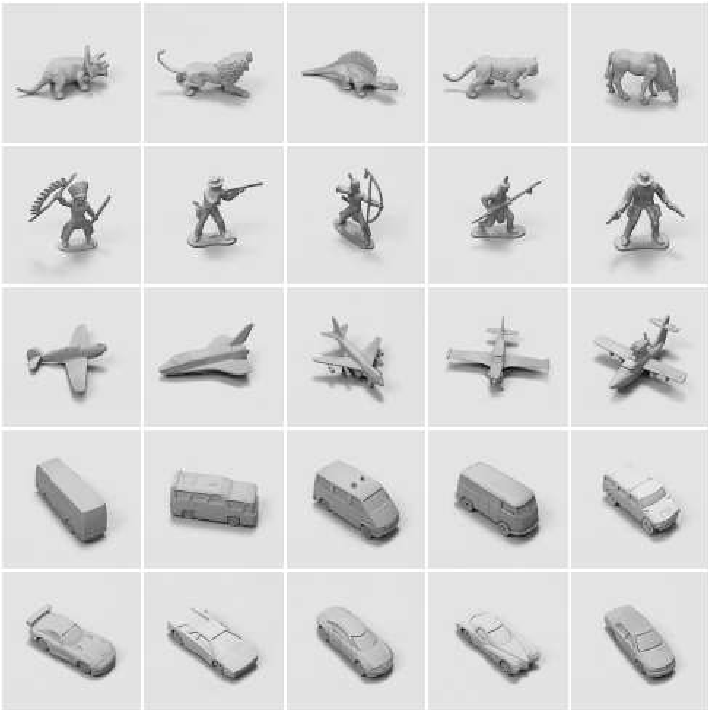

Figure 6: The 25 testing objects in the *normalized-uniform* NORB set. The testing objects are unseen by the trained system.

6.4 The lessons from NORB

While MNIST is a useful benchmark, its images are simple enough to allow a global template matching scheme to perform well. Natural images of 3D objects with background clutter are considerably more challenging. NORB [LeCun et al., 2004] is a publicly available dataset of object images from 5 generic categories. It contains images of 50 different toys, with 10 toys in each of the 5 generic categories: four-legged animals, human figures, airplanes, trucks, and cars. The 50 objects are split into a training set with 25 objects, and a test set with the remaining 25 object (see examples in Figure 6).

Each object is captured by a stereo camera pair in 162 different views (9 elevations, 18 azimuths) under 6 different illuminations. Two datasets derived from NORB are used. The first dataset, called the *normalized-uniform* set, are images of a single object with a normalized size placed at the center of images with uniform background. The training set has 24,300 stereo image pairs of size 96×96, and another 24,300 for testing (from different object instances).

The second set, the *jittered-cluttered* set, contains objects with randomly perturbed positions, scales, in-plane rotation, brightness, and contrast. The objects are placed on highly cluttered backgrounds and other NORB objects placed on the periphery. A 6-th category of images is included: background images containing no objects. Some examples images of this set are shown in figure 7. Each image in the jittered-cluttered set is randomly perturbed so that the objects are at different positions ([-3, +3] pixels horizontally and vertically), scales (ratio in [0.8, 1.1]), image-plane angles ($[-5°, 5°]$), brightness ([-20, 20] shifts of gray scale), and contrasts ([0.8, 1.3] gain). The central object could be occluded by the randomly placed distractor. To generate the training set, each image was perturbed with 10 different configurations of the above parameters, which makes up 291,600 image pairs of size 108×108. The testing set has 2 drawings of perturbations per image, and contains 58,320 pairs.

In the NORB datasets, the only useful and reliable clue is the shape of the object, while all the other parameters that affect the appearance are subject to variation, or are designed to contain no useful clue. Parameters that are subject to variation are: viewing angles (pose), lighting conditions. Potential clues whose impact was eliminated include: color (all images are grayscale), and object texture. For specific object recognition tasks, the color and texture information may be helpful, but for generic recognition tasks the color and texture information are distractions rather than useful clues. By preserving natural variabilities and eliminating irrelevant clues and systematic biases, NORB can serve as a benchmark dataset in which no hidden regularity that would unfairly advantage some methods over others can be used.

A six-layer net dubbed LeNet-7, shown in figure 5, was used in the experiments with the NORB dataset reported here. The architecture is essentially identical to that of LeNet-5 and LeNet-6, except of the sizes of the feature maps. The input is a pair of 96×96 gray scale images. The first feature detection layer uses twelve 5×5 convolution kernels to generate 8 feature maps of size 92×92. The first 2 maps take input from the left image, the next two from the right image, and the last 4 from both. There are 308 trainable parameters in this layer. The first feature pooling layer uses a 4×4 subsampling, to produce 8 feature maps of size 23×23. The second feature detection layer uses 96 convolution kernels of size 6×6 to output 24 feature maps of size 18×18. Each map takes input from 2 monocular maps and 2 binocular maps, each with a different combination, as shown in figure 8. This configuration is used to combine features from the stereo image pairs. This layer contains 3,480 trainable parameters. The next pooling layer uses a 3×3 subsampling which outputs 24 feature maps of size 6×6. The next layer has 6×6 convolution kernels to produce 100 feature maps of size 1×1, and the last layer has 5 units. In the experiments, we also report results using a hybrid method, which consists in training the convolutional network in the conventional way, chopping off the last layer, and training a Gaussian kernel SVM on the output of the penultimate layer. Many of the results in this section were previously reported in [Huang and LeCun, 2006].

6.5 Results on the *normalized-uniform* set

Table 2 shows the results on the smaller NORB dataset with uniform background. This dataset simulates a scenario in which objects can be perfectly segmented from the background, and is therefore rather unrealistic.

	SVM	Conv Net			SVM/Conv
test error	11.6%	10.4%	6.0%	6.2%	5.9%
train time (min*GHz)	480	64	448	3,200	50+
test time per sample (sec*GHz)	0.95	0.03			0.04+
fraction of S.V.	28%				28%
parameters	σ=2,000 C=40	step size = $2{\times}10^{-5}$ - $2{\times}10^{-7}$			dim=80 σ=5 C=0.01

Table 2: Testing error rates and training/testing timings on the *normalized-uniform* dataset of different methods. The timing is normalized to hypothetical 1GHz single CPU. The convolutional nets have multiple results with different training passes due to iterative training.

The SVM is composed of five binary SVMs that are trained to classify one object category against all other categories. The convolutional net trained on this set has a smaller penultimate layer with 80 outputs. The input features to the SVM of the hybrid system are accordingly 80-dimensional vectors.

The timing figures in Table 2 represent the CPU time on a fictitious 1GHz CPU. The results of the convolutional net trained after 2, 14, 100 passes are listed in the table. The network is slightly over-trained with more than 30 passes (no regularization was used in the experiment). The SVM in the hybrid system is trained over the features extracted from the network trained with 100 passes. The improvement of the combination is marginal over the convolutional net alone.

Despite the relative simplicity of the task (no position variation, uniform backgrounds, only 6 types of illuminations), the SVM performs rather poorly. Interestingly, it require a very large amount of CPU time for training and testing. The convolutional net reaches the same error rate as the SVM with 8 times less training time. Further training halves the error rate. It is interesting that despite its deep architecture, its non-convex loss, the total absence of explicit regularization, and a lack of tight generalization bounds, the convolutional net is both better and faster than an SVM.

6.6 Results on the *jittered-cluttered* set

The results on this set are shown in table 3. To classify the 6 categories, 6 binary ("one vs. others") SVM sub-classifiers are trained independently, each with the full set of 291,600 samples. The training samples are raw 108×108 pixel image pairs turned into a 23,328-dimensional input vector, with values between 0 to 255.

SVMs have relatively few free parameters to tune prior to learning. In the case of Gaussian kernels, one can choose σ (Gaussian kernel sizes) and C (penalty coefficient) that yield best results by grid tuning. A rather disappointing test error rate of 43.3% is obtained on this set, as shown in the first column of table 3. The training time depends

	SVM	Conv Net			SVM/Conv
test error	43.3%	16.38%	7.5%	7.2%	5.9%
train time (min*GHz)	10,944	420	2,100	5,880	330+
test time per sample (sec*GHz)	2.2	0.04			0.06+
#SV	5%				2%
parameters	σ=10^4 C=40	step size = 2×10^{-5} - 1×10^{-6}			dim=100 σ=5 C=1

Table 3: Testing error rates and training/testing timings on the *jittered-cluttered* dataset of different methods. The timing is normalized to hypothetical 1GHz single CPU. The convolutional nets have multiple results with different training passes due to its iterative training.

heavily on the value of σ for Gaussian kernel SVMs. The experiments are run on a 64-CPU (1.5GHz) cluster, and the timing information is normalized into a hypothetical 1GHz single CPU to make the measurement meaningful.

For the convolutional net LeNet-7, we listed results after different number of passes (1, 5, 14) and their timing information. The test error rate flattens out at 7.2% after about 10 passes. No significant over-training was observed, and no early stopping was performed. One parameter controlling the training procedure must be heuristically chosen: the global step size of the stochastic gradient procedure. Best results are obtained by adopting a schedule in which this step size is progressively decreased.

A full propagation of one data sample through the network requires about 4 million multiply-add operations. Parallelizing the convolutional net is relatively simple since multiple convolutions can be performed simultaneously, and each convolution can be performed independently on sub-regions of the layers. The convolutional nets are computationally very efficient. The training time scales sublinearly with dataset size in practice, and the testing can be done in real-time at a rate of a few frames per second.

The third column shows the result of a hybrid system in which the last layer of the convolutional net was replaced by a Gaussian SVM after training. The training and testing features are extracted with the convolutional net trained after 14 passes. The penultimate layer of the network has 100 outputs, therefore the features are 100-dimensional. The SVMs applied on features extracted from the convolutional net yield an error rate of 5.9%, a significant improvement over either method alone. By incorporating a learned feature extractor into the kernel function, the SVM was indeed able to leverage both the ability to use low-level spatially local features and at the same time keep all the advantages of a large margin classifier.

The poor performance of SVM with Gaussian kernels on raw pixels is not unexpected. As we pointed out in previous sections, a Gaussian kernel SVM merely computes matching scores (based on Euclidean distance) between the incoming pattern and templates from the training set. This global template matching is very sensitive to variations in registration, pose, and illumination. More importantly, most of the pixels in a NORB image are actually on the background clutter, rather than on the object to be recognized. Hence the template matching scores are dominated by irrelevant variabilities of the background. This points to a crucial deficiency of standard kernel methods: their inability to select relevant input features, and ignore irrelevant ones.

SVMs have presumed advantages provided by generalization bounds, capacity control through margin maximization, a convex loss function, and universal approximation properties. By contrast, convolutional nets have no generalization bounds (beyond the most general VC bounds), no explicit regularization, a highly non-convex loss function, and no claim to universality. Yet the experimental results with NORB show that convolutional nets are more accurate than Gaussian SVMs by a factor of 6, faster to train by a large factor (2 to 20), and faster to run by a factor of 50.

7 Conclusion

This work was motivated by our requirements for learning algorithms that could address the challenge of AI, which include statistical scalability, computational scalability and human-labor scalability. Because the set of tasks involved in AI is widely diverse, engineering a separate solution for each task seems impractical. We have explored many limitations of *kernel machines* and other *shallow architectures*. Such architectures are inefficient for representing complex, highly-varying functions, which we believe are necessary for AI-related tasks such as invariant perception.

One limitation was based on the well-known depth-breadth tradeoff in circuits design Håstad [1987]. This suggests that many functions can be much more efficiently represented with deeper architectures, often with a modest number of levels (e.g., logarithmic in the number of inputs).

The second limitation regards mathematical consequences of the curse of dimensionality. It applies to local kernels such as the Gaussian kernel, in which $K(x, x_i)$ can be seen as a template matcher. It tells us that architectures relying on local kernels can be very inefficient at representing functions that have many variations, i.e., functions that are not globally smooth (but may still be locally smooth). Indeed, it could be argued that *kernel machines are little more than souped-up template matchers*.

A third limitation pertains to the computational cost of learning. In theory, the convex optimization associated with kernel machine learning yields efficient optimization and reproducible results. Unfortunately, most current algorithms are (at least) quadratic in the number of examples. This essentially precludes their application to very large-scale datasets for which linear- or sublinear-time algorithms are required (particularly for on-line learning). This problem is somewhat mitigated by recent progress with on-line algorithms for kernel machines (e.g., see [Bordes et al., 2005]), but there remains the question of the increase in the number of support vectors as the number of examples increases.

A fourth and most serious limitation, which follows from the first (shallowness) and second (locality) pertains to inefficiency in *representation*. Shallow architectures and local estimators are simply too inefficient (in terms of required number of examples and adaptable components) to represent many abstract functions of interest. Ultimately, this makes them unaffordable if our goal is to learn the AI-set. We do not mean to suggest that kernel machines have no place in AI. For example, our results suggest that combining a deep architecture with a kernel machine that takes the higher-level learned representation as input can be quite powerful. Learning the transformation from pixels to high-level features before applying an SVM is in fact a way to learn the kernel. We do suggest that machine learning researchers aiming at the AI problem should investigate architectures that do not have the representational limitations of kernel machines, and deep architectures are by definition not shallow and usually not local as well.

Until recently, many believed that training deep architectures was too difficult an optimization problem. However, at least two different approaches have worked well in training such architectures: simple gradient descent applied to convolutional networks [LeCun et al., 1989, LeCun et al., 1998] (for signals and images), and more recently, layer-by-layer unsupervised learning followed by gradient descent [Hinton et al., 2006, Bengio et al., 2007, Ranzato et al., 2006]. Research on deep architectures is in its infancy, and better learning algorithms for deep architectures remain to be discovered. Taking a larger perspective on the objective of discovering learning principles that can lead to AI has been a guiding perspective of this work. We hope to have helped inspire others to seek a solution to the problem of scaling machine learning towards AI.

Acknowledgments

We thank Geoff Hinton for our numerous discussions with him and the Neural Computation and Adaptive Perception program of the Canadian Institute of Advanced Research for making them possible. We wish to thank Fu-Jie Huang for conducting much of the experiments in section 6, and Hans-Peter Graf and Eric Cosatto and their collaborators for letting us use their parallel implementation of SVM. We thank Leon Bottou for his patience and for helpful comments. We thank Sumit Chopra, Olivier Delalleau, Raia Hadsell, Hugo Larochelle, Nicolas Le Roux, Marc'Aurelio Ranzato, for helping us to make progress towards the ideas presented here. This project was supported by NSF Grants No. 0535166 and No. 0325463, by NSERC, the Canada Research Chairs, and the MITACS NCE.

References

Miklos Ajtai. $\sum_1^1$-formulae on finite structures. *Annals of Pure and Applied Logic*, 24 (1):48, 1983.

Eric Allender. Circuit complexity before the dawn of the new millennium. In *16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 1–18. Lecture Notes in Computer Science 1180, 1996.

Mikhail Belkin and Partha Niyogi. Using manifold structure for partially labeled classification. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, Cambridge, MA, 2003. MIT Press.

Mikhail Belkin, Irina Matveeva, and Partha Niyogi. Regularization and semi-supervised learning on large graphs. In John Shawe-Taylor and Yoram Singer, editors, *COLT'2004*. Springer, 2004.

Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, April 2002.

Yoshua Bengio and Martin Monperrus. Non-local manifold tangent learning. In L.K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*. MIT Press, 2005.

Yoshua Bengio, Olivier Delalleau, Nicolas Le Roux, Jean-François Paiement, Pascal Vincent, and Marie Ouimet. Learning eigenfunctions links spectral embedding and kernel PCA. *Neural Computation*, 16(10):2197–2219, 2004.

Yoshua Bengio, Olivier Delalleau, and Nicolas Le Roux. The curse of dimensionality for local kernel machines. Technical Report 1258, Département d'informatique et recherche opérationnelle, Université de Montréal, 2005.

Yoshua Bengio, Olivier Delalleau, and Nicolas Le Roux. The curse of highly variable functions for local kernel machines. In *Advances in Neural Information Processing Systems 18*. MIT Press, 2006a.

Yoshua Bengio, Nicolas Le Roux, Pascal Vincent, Olivier Delalleau, and P. Marcotte. Convex neural networks. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 123–130. MIT Press, 2006b.

Yoshua Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 19*. MIT Press, 2007.

Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579–1619, September 2005.

Bernhard Boser, Isabelle Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifiers. In *Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, Pittsburgh, 1992.

Matthew Brand. Charting a manifold. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*. MIT Press, 2003.

Corinna Cortes and Vladimir N. Vapnik. Support vector networks. *Machine Learning*, 20:273–297, 1995.

Dennis DeCoste and Bernhard Schölkopf. Training invariant support vector machines. *Machine Learning*, 46:161–190, 2002.

Olivier Delalleau, Yoshua Bengio, and Nicolas Le Roux. Efficient non-parametric function induction in semi-supervised learning. In R.G. Cowell and Z. Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, Jan 6-8, 2005, Savannah Hotel, Barbados*, pages 96–103. Society for Artificial Intelligence and Statistics, 2005.

Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

Wolfgang Härdle, Stefan Sperlich, Marlene Müller, and Axel Werwatz. *Nonparametric and Semiparametric Models*. Springer, 2004.

Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

Geoffrey E. Hinton. To recognize shapes, first learn to generate images. In P. Cisek, T. Drew, and J. Kalaska, editors, *Computational Neuroscience: Theoretical insights into brain function*. Elsevier, To appear. 2007.

Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 2006.

Johan T. Håstad. *Computational Limitations for Small Depth Circuits*. MIT Press, Cambridge, MA, 1987.

Fu-Jie Huang and Yann LeCun. Large-scale learning with svm and convolutional nets for generic object categorization. In *Proc. Computer Vision and Pattern Recognition Conference (CVPR'06)*. IEEE Press, 2006.

Thorsten Joachims. Transductive inference for text classification using support vector machines. In Ivan Bratko and Saso Dzeroski, editors, *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages 200–209, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US.

Michael I. Jordan. *Learning in Graphical Models*. Kluwer, Dordrecht, Netherlands, 1998.

Yann LeCun and John S. Denker. Natural versus universal probability complexity, and entropy. In *IEEE Workshop on the Physics of Computation*, pages 122–127. IEEE, 1992.

Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

Yann LeCun, Fu-Jie Huang, and Léon Bottou. Learning methods for generic object recognition with invariance to pose and lighting. In *Proceedings of CVPR'04*. IEEE Press, 2004.

Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, Fourier transform, and learnability. *J. ACM*, 40(3):607–620, 1993.

Marvin L. Minsky and Seymour A. Papert. *Perceptrons*. MIT Press, Cambridge, 1969.

Marc'Aurelio Ranzato, Christopher Poultney, Sumit Chopra, and Yann LeCun. Efficient learning of sparse representations with an energy-based model. In J. Platt et al., editor, *Advances in Neural Information Processing Systems (NIPS 2006)*. MIT Press, 2006.

Sam Roweis and Lawrence Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, Dec. 2000.

Michael Schmitt. Descartes' rule of signs for radial basis function neural networks. *Neural Computation*, 14(12):2997–3011, 2002.

Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural Computation*, 10:1299–1319, 1998.

Bernhard Schölkopf, Christopher J. C. Burges, and Alexander J. Smola. *Advances in Kernel Methods — Support Vector Learning*. MIT Press, Cambridge, MA, 1999.

Patrice Simard, Dave Steinkraus, and John C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *Proceedings of ICDAR 2003*, pages 958–962, 2003.

Robert R. Snapp and Santosh S. Venkatesh. Asymptotic derivation of the finite-sample risk of the k nearest neighbor classifier. Technical Report UVM-CS-1998-0101, Department of Computer Science, University of Vermont, 1998.

Yee Whye Teh and Geoffrey E. Hinton. Rate-coded restricted boltzmann machines for face recognition. In T.K. Leen, T.G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*. MIT Press, 2001.

Josh B. Tenenbaum, Vin de Silva, and John C. L. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, Dec. 2000.

Gerry Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8: 257–277, 1992.

Paul E. Utgoff and David J. Stracuzzi. Many-layered learning. *Neural Computation*, 14:2497–2539, 2002.

Vladimir N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

Yair Weiss. Segmentation using eigenvectors: a unifying view. In *Proceedings IEEE International Conference on Computer Vision*, pages 975–982, 1999.

Christopher K. I. Williams and Carl E. Rasmussen. Gaussian processes for regression. In D.S. Touretzky, M.C. Mozer, and M.E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 514–520. MIT Press, Cambridge, MA, 1996.

David H. Wolpert. The lack of a priori distinction between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.

Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems 16*, Cambridge, MA, 2004. MIT Press.

Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. In *ICML'2003*, 2003.

Figure 7: Some of the 291,600 examples from the *jittered-cluttered* training set (left camera images). Each column shows images from one category. A 6-th background category is added

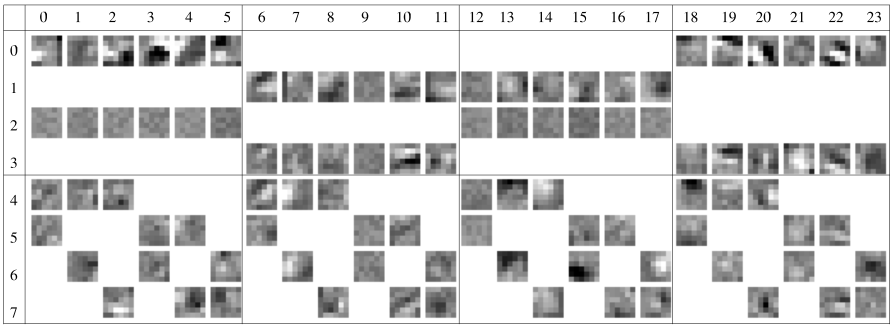

Figure 8: The learned convolution kernels of the C3 layer. The columns correspond to the 24 feature maps output by C3, and the rows correspond to the 8 feature maps output by the S2 layer. Each feature map draw from 2 monocular maps and 2 binocular maps of S2. 96 convolution kernels are use in total.