


## Neural Networks, Convexity, Kernels and Curses



Yoshua Bengio

Work done with Nicolas Le Roux, Olivier Delalleau and  
Hugo Larochelle  
August 26th 2005

**We're doomed!**

## Perspective

Most common non-parametric approaches based on smoothness prior, which leads to “local” learning algorithms, e.g. kernel-based.

## Perspective

Most common non-parametric approaches based on smoothness prior, which leads to “local” learning algorithms, e.g. kernel-based.

Smoothness may not be the only way to obtain “simple functions” : e.g. According to Kolmogorov complexity, *sinus*( $x$ ) and *parity*( $x$ ) are simple yet they are highly variable (apparently complex) functions.

## Perspective

Most common non-parametric approaches based on smoothness prior, which leads to “local” learning algorithms, e.g. kernel-based.

Smoothness may not be the only way to obtain “simple functions” : e.g. According to Kolmogorov complexity,  $\sin(x)$  and  $\text{parity}(x)$  are simple yet they are highly variable (apparently complex) functions.

We find that what matters is not dimensionality as such but “variability” of the target function (which is easier to obtain in high dimension).

## Perspective

Most common non-parametric approaches based on smoothness prior, which leads to “local” learning algorithms, e.g. kernel-based.

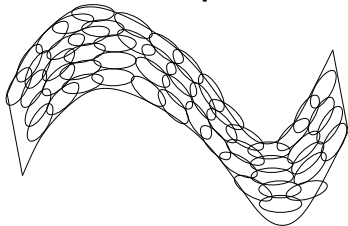
Smoothness may not be the only way to obtain “simple functions” : e.g. According to Kolmogorov complexity,  $\sin(x)$  and  $\text{parity}(x)$  are simple yet they are highly variable (apparently complex) functions.

We find that what matters is not dimensionality as such but “variability” of the target function (which is easier to obtain in high dimension).

Our work suggest that already established results for classical non-parametric learning generalize to modern kernel machines. How about neural networks? Can we have non-local and non-trivial learning?

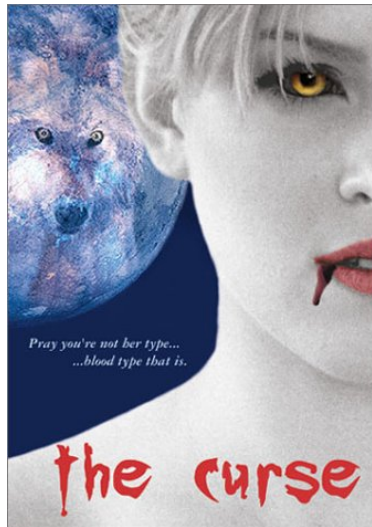
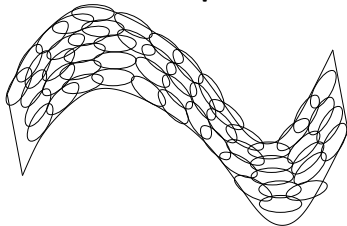
## Geometric Intuition

**If** we have to tile the space or the manifold where the bulk of the distribution is concentrated, then we will need an **exponential number of “patches”** :



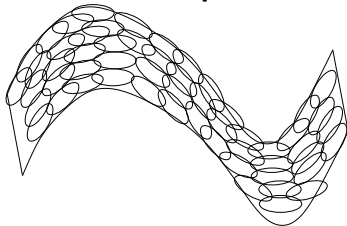
## Geometric Intuition

If we have to tile the space or the manifold where the bulk of the distribution is concentrated, then we will need an **exponential number of “patches”** :

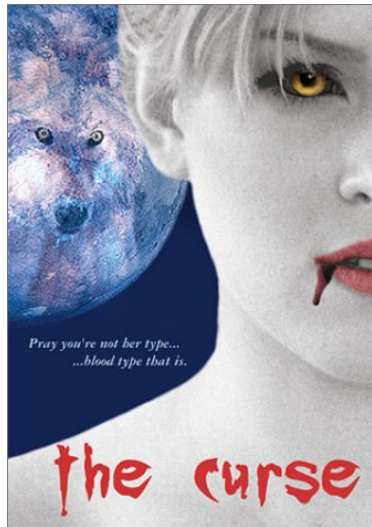


## Geometric Intuition

If we have to tile the space or the manifold where the bulk of the distribution is concentrated, then we will need an **exponential number of “patches”** :



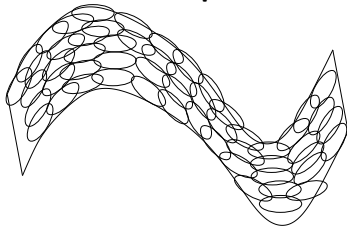
For classification problems no need to cover the whole space/manifold, only decision surface, but still has dim.  $d - 1$ .





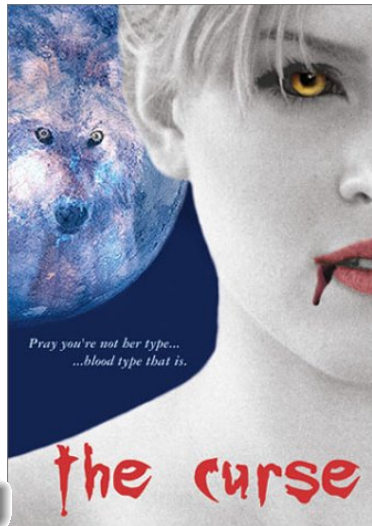
## Geometric Intuition

If we have to tile the space or the manifold where the bulk of the distribution is concentrated, then we will need an **exponential number of “patches”** :



For classification problems no need to cover the whole space/manifold, only decision surface, but still has dim.  $d - 1$ .

Number of required examples  $\propto \text{const}^d$



# Kernel Density Estimation

For a wide class of kernel density estimators (Hardle 2004), the generalization error converges in  $n^{-4/(4+d)}$ .

# Kernel Density Estimation

For a wide class of kernel density estimators (Hardle 2004), the generalization error converges in  $n^{-4/(4+d)}$ .  
 $O(n^{-1/d})$  for k-nearest-neighbors classifiers (Snapp & Venkatesh 1998).

# Kernel Density Estimation

For a wide class of kernel density estimators (Hardle 2004), the generalization error converges in  $n^{-4/(4+d)}$ .  
 $O(n^{-1/d})$  for k-nearest-neighbors classifiers (Snapp & Venkatesh 1998).

*The required number of examples to reach a given error level is exponential in  $d$*

# Kernel Methods

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

Used in regression, classification (KNN, SVM, ...), dimensionality reduction (kernel PCA, LLE, Isomap, Laplacian eigenmaps, ...).  $D$  = data.

CHALLENGES FOR THIS WORK :

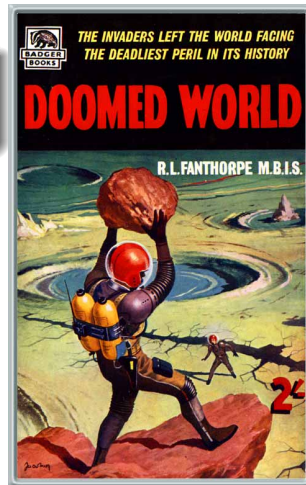
# Kernel Methods

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

Used in regression, classification (KNN, SVM, ...), dimensionality reduction (kernel PCA, LLE, Isomap, Laplacian eigenmaps, ...).  $D$  = data.

CHALLENGES FOR THIS WORK :

- 1 SVM's  $\alpha_i$ 's may depend on  $x_j$  far from  $x_i$



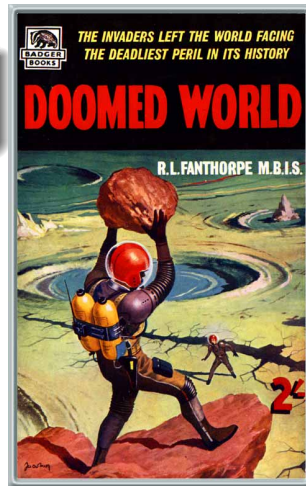
# Kernel Methods

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

Used in regression, classification (KNN, SVM, ...), dimensionality reduction (kernel PCA, LLE, Isomap, Laplacian eigenmaps, ...).  $D$  = data.

CHALLENGES FOR THIS WORK :

- 1 *SVM's  $\alpha_i$ 's may depend on  $x_j$  far from  $x_i$*
- 2 *Kernel may be data-dependent (e.g. tuned hyper-parameters)*



# When a Test Example is Far from Training Examples

If the kernel is **local**, i.e.

$$\lim_{||x-x_i|| \rightarrow \infty} K(x, x_i) \rightarrow c_i$$

then when  $x$  gets farther from the training set

$$f(x) \rightarrow b + \sum_i \alpha_i c_i$$



# When a Test Example is Far from Training Examples

If the kernel is **local**, i.e.

$$\lim_{||x-x_i|| \rightarrow \infty} K(x, x_i) \rightarrow c_i$$

then when  $x$  gets farther from the training set

$$f(x) \rightarrow b + \sum_i \alpha_i c_i$$

*Gaussian kernel : **after becoming approx. linear**, predictor becomes either constant or (approximately) the nearest neighbor predictor.*

# When a Test Example is Far from Training Examples

If the kernel is **local**, i.e.

$$\lim_{||x-x_i|| \rightarrow \infty} K(x, x_i) \rightarrow c_i$$

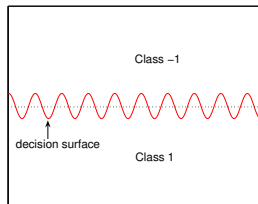
then when  $x$  gets farther from the training set

$$f(x) \rightarrow b + \sum_i \alpha_i c_i$$

*Gaussian kernel : **after becoming approx. linear**, predictor becomes either constant or (approximately) the nearest neighbor predictor.*

In high dimensions, a random test point tends to be **equally far** from most training examples.

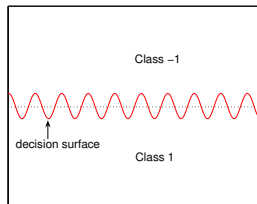
## Simple but Highly Variable Functions : Difficult to Learn



This “complex” sinusoidal decision surface cannot be learned with less than 10 Gaussians. However, in “C” language, it has a high prior.

**\*\*\* KEY RESULT \*\*\***

## Simple but Highly Variable Functions : Difficult to Learn



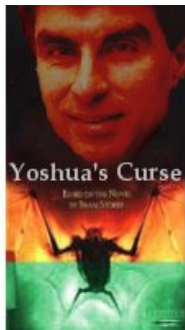
This “complex” sinusoidal decision surface cannot be learned with less than 10 Gaussians. However, in “C” language, it has a high prior.

**\*\*\* KEY RESULT \*\*\***

Corollary of (Schmitt 2002)

If  $\exists$  a line in  $\mathbb{R}^d$  that intersects  $m$  times with the decision surface  $S$  (and is not included in  $S$ ), then one needs at least  $\lceil \frac{m}{2} \rceil$  Gaussians (of same width) to learn  $S$  with a Gaussian kernel classifier.

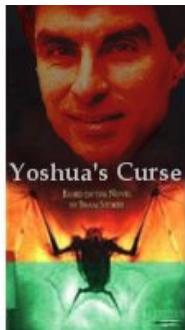
# The Parity Problem



parity :

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

# The Parity Problem



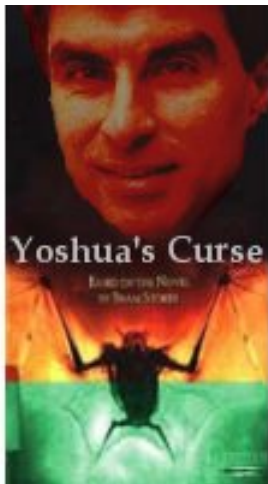
parity :

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

## Theorem

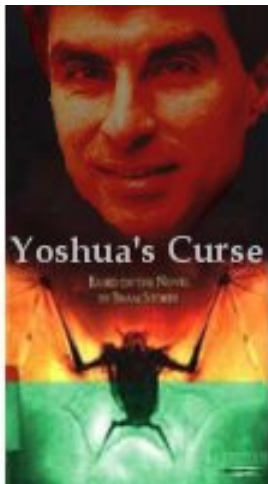
A Gaussian kernel classifier needs at least  $2^{d-1}$  Gaussians (i.e. support vectors) to learn the parity function (when Gaussians have fixed width and are centered on training points).

## Is Parity a Pathological Case?



- Parity thm : proof technique exploits the fact that neighbors of  $x$  have  $y$  with opposite sign.

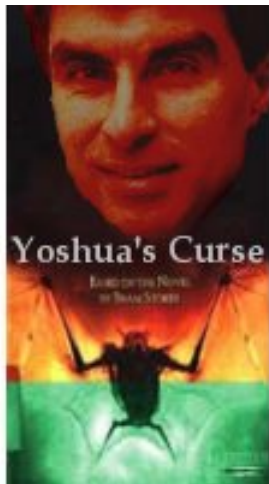
# Is Parity a Pathological Case?



- Parity thm : proof technique exploits the fact that neighbors of  $x$  have  $y$  with opposite sign.
- Argument would apply to a smooth fn on  $[0, 1]^d$  rather than  $\{0, 1\}^d$ , but considering finite training sample in  $\{0, 1\}^d$ .



## Is Parity a Pathological Case?



- Parity thm : proof technique exploits the fact that neighbors of  $x$  have  $y$  with opposite sign.
- Argument would apply to a smooth fn on  $[0, 1]^d$  rather than  $\{0, 1\}^d$ , but considering finite training sample in  $\{0, 1\}^d$ .
- Thm on *highly varying functions along a line* suggests that it is representative of highly varying functions in general : curse of dimensionality (or rather curse of local complexity).

## Local-Derivative Kernels

SVM :  $f(x)$  not local (depends on  $x_i$  far from  $x$ ) through  $\alpha_i$ 's!

The derivative of  $f$  is

$$\frac{\partial f(x)}{\partial x} = \sum_{i=1}^n \alpha_i \frac{\partial K(x, x_i)}{\partial x}$$

# Local-Derivative Kernels

SVM :  $f(x)$  not local (depends on  $x_i$  far from  $x$ ) through  $\alpha_i$ 's!

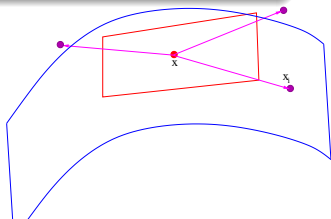
The derivative of  $f$  is

$$\frac{\partial f(x)}{\partial x} = \sum_{i=1}^n \alpha_i \frac{\partial K(x, x_i)}{\partial x}$$

## Local-derivative kernel

When  $\partial f / \partial x$  is (approximately) contained in **the span of the vectors  $(x - x_j)$  with  $x_j$  a neighbor of  $x$**

$$\frac{\partial f(x)}{\partial x} \simeq \sum_{x_j \in \mathcal{N}(x)} \gamma_j (x - x_j)$$



# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial \mathbf{x}$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial \mathbf{x}$  is the **decision surface normal vector**

## Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

## Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

*SVMs with Gaussian kernel are **local-derivative***

# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

*SVMs with Gaussian kernel are **local-derivative***

*LLE is **local-derivative***



# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

*SVMs with Gaussian kernel are **local-derivative***

*LLE is **local-derivative***

*Isomap is **local-derivative***

# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

*SVMs with Gaussian kernel are **local-derivative***

*LLE is **local-derivative***

*Isomap is **local-derivative***

*Kernel PCA with Gaussian kernel is **local-derivative***

# Tangent Planes and Decision Surfaces

**Manifold learning** :  $\partial f / \partial x$  span manifold's **tangent plane**

**Classification** :  $\partial f / \partial x$  is the **decision surface normal vector**

*Constraining  $\partial f / \partial x$  in the span of the neighbors is a very strong constraint, possibly leading to **high-variance** estimators.*

*SVMs with Gaussian kernel are **local-derivative***

*LLE is **local-derivative***

*Isomap is **local-derivative***

*Kernel PCA with Gaussian kernel is **local-derivative***

*Spectral clustering with Gaussian kernel is **local-derivative***

# General Curse of Dimensionality Argument

**Locality.** Show that crucial properties of  $f(x)$  (e.g. tangent plane, decision surface normal vector) depend mostly on examples in ball  $\mathcal{N}(x)$ .

## General Curse of Dimensionality Argument

**Locality.** Show that crucial properties of  $f(x)$  (e.g. tangent plane, decision surface normal vector) depend mostly on examples in ball  $\mathcal{N}(x)$ .

**Smoothness.** Show that within  $\mathcal{N}(x)$ , crucial property of  $f(x)$  must vary slowly ( = smoothness within  $\mathcal{N}(x)$  ).

# General Curse of Dimensionality Argument

**Locality.** Show that crucial properties of  $f(x)$  (e.g. tangent plane, decision surface normal vector) depend mostly on examples in ball  $\mathcal{N}(x)$ .

**Smoothness.** Show that within  $\mathcal{N}(x)$ , crucial property of  $f(x)$  must vary slowly ( = smoothness within  $\mathcal{N}(x)$  ).

**Complexity.** Consider targets that vary sufficiently so that one needs to consider  $O(\text{const}^d)$  different neighborhoods, with significantly different properties in each neighborhood.

N.B. : the issue is not really **DIMENSIONALITY** but rather **NUMBER OF VARIATIONS** (e.g. sign changes).

# Spectral Manifold Learning Algorithms

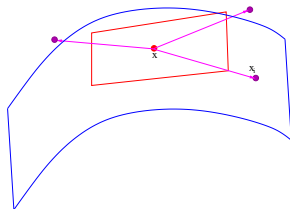
Many manifold learning alg. = kernel machines with data-dependent kernel (LLE, Isomap, kPCA, Laplacian Eigenmaps, charting, etc...).

# Spectral Manifold Learning Algorithms

Many manifold learning alg. = kernel machines with data-dependent kernel (LLE, Isomap, kPCA, Laplacian Eigenmaps, charting, etc...).

## Locality

Shown for the estimated tangent plane.





# Spectral Manifold Learning Algorithms

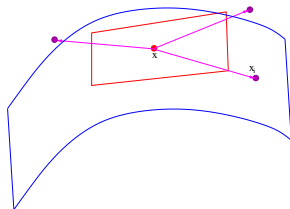
Many manifold learning alg. = kernel machines with data-dependent kernel (LLE, Isomap, kPCA, Laplacian Eigenmaps, charting, etc...).

## Locality

Shown for the estimated tangent plane.

## Smoothness of $f(\cdot)$

Tangent plane varies slowly within  $\mathcal{N}(x)$ , since in span of vectors  $x - x_i$ .



# Spectral Manifold Learning Algorithms

Many manifold learning alg. = kernel machines with data-dependent kernel (LLE, Isomap, kPCA, Laplacian Eigenmaps, charting, etc...).

## Locality

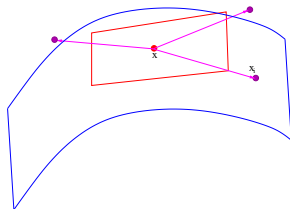
Shown for the estimated tangent plane.

## Smoothness of $f(\cdot)$

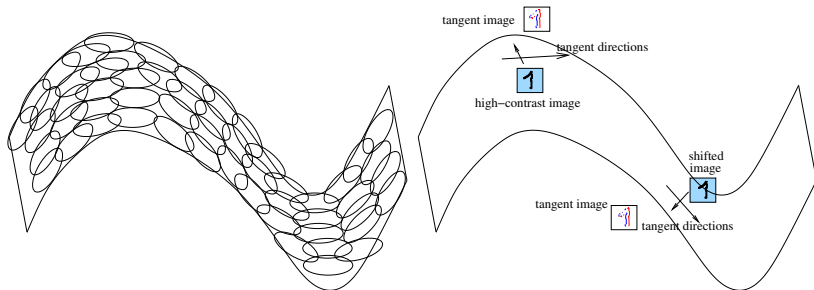
Tangent plane varies slowly within  $\mathcal{N}(x)$ , since in span of vectors  $x - x_i$ .

## Non-Smoothness of Target

High curvature underlying manifold  $\rightarrow$  we are doomed...



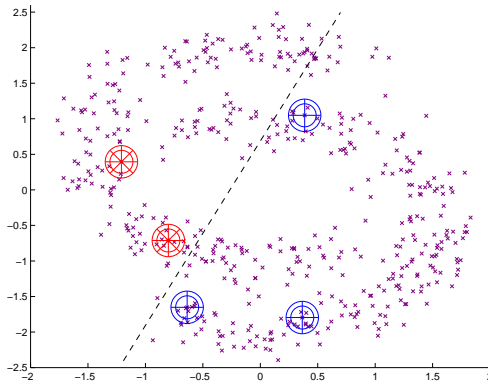
## Ex : Translation of a High Contrast Image



N.B.  $\exists$  examples of **non-local learning** with **no domain-specific prior knowledge** which worked on learning such manifolds (rotations and translations), (Bengio, Monperrus 2005), generalizing far from training examples.

# Graph-Based Semi-Supervised Learning

Semi-supervised learning : few labeled examples, many unlabeled.



Graph-based methods : implicitly or explicitly propagate labels along the near-neighbor graph. Very similar to spectral manifold learning.

## Curse on Graph-Based Semi-Supervised Methods

Large number of graph-based semi-supervised methods are exactly or approximately quadratic cost to trade-off between smoothness (near examples should get a close label) and fitting the labeled examples (Zhu et al 2003, Zhou et al 2004, Belkin et al 2004, Delalleau et al 2005).

## Curse on Graph-Based Semi-Supervised Methods

Large number of graph-based semi-supervised methods are exactly or approximately quadratic cost to trade-off between smoothness (near examples should get a close label) and fitting the labeled examples (Zhu et al 2003, Zhou et al 2004, Belkin et al 2004, Delalleau et al 2005).

### Theorem

After running a label propagation algorithm minimizing quadratic cost, number of regions with constant estimated label  $\leq$  number of labeled examples.

# The 1-Norm Soft Margin SVM with Gaussian Kernel

## Locality

As shown in (Keerthi et al 2003), the SVM becomes constant when  $\sigma \rightarrow 0$  or  $\sigma \rightarrow \infty \Rightarrow$  notion of locality w.r.t  $\sigma$ .

**Local-derivative** : Locality of normal vector of decision surface.

# The 1-Norm Soft Margin SVM with Gaussian Kernel

## Locality

As shown in (Keerthi et al 2003), the SVM becomes constant when  $\sigma \rightarrow 0$  or  $\sigma \rightarrow \infty \Rightarrow$  notion of locality w.r.t  $\sigma$ .

**Local-derivative** : Locality of normal vector of decision surface.

## Smoothness of $f(\cdot)$

When there are training examples at a distance of the order of  $\sigma$ , the normal vector is almost constant in a ball whose radius is small with respect to  $\sigma$ .



## Then What ?

- Local Kernel machines won't scale to highly variable functions in high manifold dimension.

Good news : SVMs interpolate between very local and very smooth (vary  $\sigma$ ). Bad news : if target function structured but not smooth...

## Then What ?

- Local Kernel machines won't scale to highly variable functions in high manifold dimension.

Good news : SVMs interpolate between very local and very smooth (vary  $\sigma$ ). Bad news : if target function structured but not smooth...

- The no-free-lunch thm : no universal recipe without appropriate prior.

## Then What ?

- Local Kernel machines won't scale to highly variable functions in high manifold dimension.

Good news : SVMs interpolate between very local and very smooth (vary  $\sigma$ ). Bad news : if target function structured but not smooth...

- The no-free-lunch thm : no universal recipe without appropriate prior.
- Is there hope ?

## Then What ?

- Local Kernel machines won't scale to highly variable functions in high manifold dimension.  
Good news : SVMs interpolate between very local and very smooth (vary  $\sigma$ ). Bad news : if target function structured but not smooth...
- The no-free-lunch thm : no universal recipe without appropriate prior.
- Is there hope ?
- Humans seem to learn such functions !
- There might be loose enough priors on general classes of functions that allow non-local learning algorithms to learn them.

## Then What ?

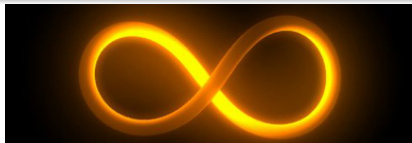
- Local Kernel machines won't scale to highly variable functions in high manifold dimension.  
Good news : SVMs interpolate between very local and very smooth (vary  $\sigma$ ). Bad news : if target function structured but not smooth...
  - The no-free-lunch thm : no universal recipe without appropriate prior.
  - Is there hope ?
  - Humans seem to learn such functions !
  - There might be loose enough priors on general classes of functions that allow non-local learning algorithms to learn them.
- What about good old neural networks ?

# Convex Neural Networks

Ordinary neural networks :  $f(x) = \sum_i w_i h(x, z_i)$

## Changing the Parametrization

- Many interesting loss functions (MSE, hinge, cross-entropy) are convex in  $w$
- Regularization penalty can also be chosen convex in  $w$
- Imagine a huge neural network will **all the possible**  $z_i$ , i.e. enumerate all the possible hidden units  $\Rightarrow$  the only free parameters are the  $w$ 's
- If  $L_1$  regularization,  $w$  is sparse  $\Rightarrow$  same final form.

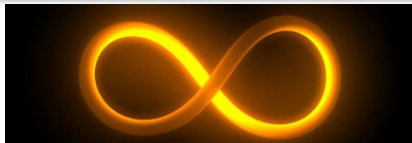


# Convex Neural Networks

Ordinary neural networks :  $f(x) = \sum_i w_i h(x, z_i)$

## Changing the Parametrization

- Many interesting loss functions (MSE, hinge, cross-entropy) are convex in  $w$
- Regularization penalty can also be chosen convex in  $w$
- Imagine a huge neural network will **all the possible**  $z_i$ , i.e. enumerate all the possible hidden units  $\Rightarrow$  the only free parameters are the  $w$ 's
- If  $L_1$  regularization,  $w$  is sparse  $\Rightarrow$  same final form.

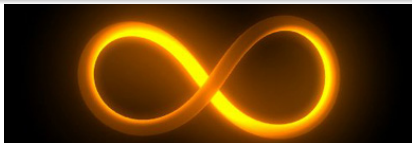


# Convex Neural Networks

Ordinary neural networks :  $f(x) = \sum_i w_i h(x, z_i)$

## Changing the Parametrization

- Many interesting loss functions (MSE, hinge, cross-entropy) are convex in  $w$
- Regularization penalty can also be chosen convex in  $w$
- Imagine a huge neural network will **all the possible**  $z_i$ , i.e. enumerate all the possible hidden units  $\Rightarrow$  the only free parameters are the  $w$ 's
- If  $L_1$  regularization,  $w$  is sparse  $\Rightarrow$  same final form.



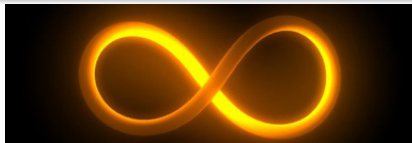


# Convex Neural Networks

Ordinary neural networks :  $f(x) = \sum_i w_i h(x, z_i)$

## Changing the Parametrization

- Many interesting loss functions (MSE, hinge, cross-entropy) are convex in  $w$
- Regularization penalty can also be chosen convex in  $w$
- Imagine a huge neural network will **all the possible**  $z_i$ , i.e. enumerate all the possible hidden units  $\Rightarrow$  the only free parameters are the  $w$ 's
- If  $L_1$  regularization,  $w$  is sparse  $\Rightarrow$  same final form.



## How to optimize the output weights ?

Training criterion, e.g.  $L(\hat{y}, y)$  convex, e.g.  $= \max(0, 1 - y\hat{y})$  :

$$C = \lambda \|w\|_1 + \sum_t L(f(x_t), y_t)$$

Iterative procedure : Column Generation (Chvatal 83)

Initialize with  $w = 0$ .

- **sub-problem** : select and insert a new hidden neuron with weights  $z$  that would reduce  $C$ .
- Re-optimize the non-zero output weights (using LASSO regression).
- Optionally remove neurons whose output weight has been set to 0.
- Stop when no additional neuron may decrease the cost.

Similar to Gradient Boosting (Friedman 2001, Mason et al 2000).

## How to optimize the output weights ?

Training criterion, e.g.  $L(\hat{y}, y)$  convex, e.g.  $= \max(0, 1 - y\hat{y})$  :

$$C = \lambda \|w\|_1 + \sum_t L(f(x_t), y_t)$$

Iterative procedure : Column Generation (Chvatal 83)

Initialize with  $w = 0$ .

- **sub-problem** : select and insert a new hidden neuron with weights  $z$  that would reduce  $C$ .
- Re-optimize the non-zero output weights (using LASSO regression).
- Optionally remove neurons whose output weight has been set to 0.
- Stop when no additional neuron may decrease the cost.

Similar to Gradient Boosting (Friedman 2001, Mason et al 2000).

## How to optimize the output weights ?

Training criterion, e.g.  $L(\hat{y}, y)$  convex, e.g.  $= \max(0, 1 - y\hat{y})$  :

$$C = \lambda \|w\|_1 + \sum_t L(f(x_t), y_t)$$

### Iterative procedure : Column Generation (Chvatal 83)

Initialize with  $w = 0$ .

- **sub-problem** : select and insert a new hidden neuron with weights  $z$  that would reduce  $C$ .
- Re-optimize the non-zero output weights (using LASSO regression).
- Optionally remove neurons whose output weight has been set to 0.
- Stop when no additional neuron may decrease the cost.

Similar to Gradient Boosting (Friedman 2001, Mason et al 2000).

## How to optimize the output weights ?

Training criterion, e.g.  $L(\hat{y}, y)$  convex, e.g.  $= \max(0, 1 - y\hat{y})$  :

$$C = \lambda \|w\|_1 + \sum_t L(f(x_t), y_t)$$

### Iterative procedure : Column Generation (Chvatal 83)

Initialize with  $w = 0$ .

- **sub-problem** : select and insert a new hidden neuron with weights  $z$  that would reduce  $C$ .
- Re-optimize the non-zero output weights (using LASSO regression).
- **Optionally remove neurons whose output weight has been set to 0.**
- Stop when no additional neuron may decrease the cost.

Similar to Gradient Boosting (Friedman 2001, Mason et al 2000).

## How to optimize the output weights ?

Training criterion, e.g.  $L(\hat{y}, y)$  convex, e.g.  $= \max(0, 1 - y\hat{y})$  :

$$C = \lambda \|w\|_1 + \sum_t L(f(x_t), y_t)$$

### Iterative procedure : Column Generation (Chvatal 83)

Initialize with  $w = 0$ .

- **sub-problem** : select and insert a new hidden neuron with weights  $z$  that would reduce  $C$ .
- Re-optimize the non-zero output weights (using LASSO regression).
- Optionally remove neurons whose output weight has been set to 0.
- **Stop when no additional neuron may decrease the cost.**

Similar to Gradient Boosting (Friedman 2001, Mason et al 2000).

## Which neuron to add / remove ?

**Convergence to global optimum iff can find a descent direction (new hidden unit) when there is one.**

## Which neuron to add / remove?

**Convergence to global optimum iff can find a descent direction (new hidden unit) when there is one.**

This condition is equivalent to finding  $z$  s.t.

$$WCA(z) = |\sum_t L'_t h(x_t, z)| > \lambda$$

where  $L'_t = \frac{\partial L(f(x_t), y_t)}{\partial f(x_t)}$ .



## Which neuron to add / remove?

**Convergence to global optimum iff can find a descent direction (new hidden unit) when there is one.**

This condition is equivalent to finding  $z$  s.t.

$$WCA(z) = |\sum_t L'_t h(x_t, z)| > \lambda$$

where  $L'_t = \frac{\partial L(f(x_t), y_t)}{\partial f(x_t)}$ .

This is a weighted classification accuracy when  $h$  is binary, e.g.  $h(x, z) = \text{sign}(x \cdot z)$ ,  $L(\hat{y}, y) = (\hat{y} - y)^2$  gives

$$WCA(z) = \sum_t |f(x_t) - y_t| \mathbf{1}_{\text{sign}(z \cdot x_t) = \text{sign}(f(x_t) - y_t)}$$

## Which neuron to add / remove?

**Convergence to global optimum iff can find a descent direction (new hidden unit) when there is one.**

This condition is equivalent to finding  $z$  s.t.

$$WCA(z) = |\sum_t L'_t h(x_t, z)| > \lambda$$

where  $L'_t = \frac{\partial L(f(x_t), y_t)}{\partial f(x_t)}$ .

This is a weighted classification accuracy when  $h$  is binary, e.g.  $h(x, z) = \text{sign}(x \cdot z)$ ,  $L(\hat{y}, y) = (\hat{y} - y)^2$  gives

$$WCA(z) = \sum_t |f(x_t) - y_t| \mathbf{1}_{\text{sign}(z \cdot x_t) = \text{sign}(f(x_t) - y_t)}$$

### Stopping criterion

If there is no  $z$  such that  $WCA(z) > \lambda$ , the algorithm stops.

# Theoretical Results

## Theorems

- **Trivial** :  $C$  is convex in  $w$ .
- Great news : exact algorithm converges to **global optimum**.
- Reassuring : nb resulting  $w_i \neq 0$ 's  $\leq$  nb examples. (proof uses Hettich 93, similar to Raetsch 2001 for regression boosting).
- Nb final hidden units monotone in penalty  $\lambda$
- No free lunch : finding optimal linear classifier :  $O(\log(n)n^d)$

**KEY : TRAINING LINEAR CLASSIFIER  
WITH WEIGHTED COST**

# Theoretical Results

## Theorems

- Trivial :  $C$  is convex in  $w$ .
- **Great news : exact algorithm converges to global optimum.**
- Reassuring : nb resulting  $w_i \neq 0$ 's  $\leq$  nb examples. (proof uses Hettich 93, similar to Raetsch 2001 for regression boosting).
- Nb final hidden units monotone in penalty  $\lambda$
- No free lunch : finding optimal linear classifier :  $O(\log(n)n^d)$

**KEY : TRAINING LINEAR CLASSIFIER  
WITH WEIGHTED COST**

# Theoretical Results

## Theorems

- Trivial :  $C$  is convex in  $w$ .
- Great news : exact algorithm converges to **global optimum**.
- Reassuring : nb resulting  $w_i \neq 0$ 's  $\leq$  nb examples. (proof uses Hettich 93, similar to Raetsch 2001 for regression boosting).
- Nb final hidden units monotone in penalty  $\lambda$
- No free lunch : finding optimal linear classifier :  $O(\log(n)n^d)$

**KEY : TRAINING LINEAR CLASSIFIER  
WITH WEIGHTED COST**

# Theoretical Results

## Theorems

- Trivial :  $C$  is convex in  $w$ .
- Great news : exact algorithm converges to **global optimum**.
- Reassuring : nb resulting  $w_i \neq 0$ 's  $\leq$  nb examples. (proof uses Hettich 93, similar to Raetsch 2001 for regression boosting).
- Nb final hidden units monotone in penalty  $\lambda$
- No free lunch : finding optimal linear classifier :  $O(\log(n)n^d)$

**KEY : TRAINING LINEAR CLASSIFIER  
WITH WEIGHTED COST**

# Theoretical Results

## Theorems

- Trivial :  $C$  is convex in  $w$ .
- Great news : exact algorithm converges to **global optimum**.
- Reassuring : nb resulting  $w_i \neq 0$ 's  $\leq$  nb examples. (proof uses Hettich 93, similar to Raetsch 2001 for regression boosting).
- Nb final hidden units monotone in penalty  $\lambda$
- **No free lunch : finding optimal linear classifier :  $O(\log(n)n^d)$**

**KEY : TRAINING LINEAR CLASSIFIER  
WITH WEIGHTED COST**

## Experimental Results

Compare exact solution ( $d = 2$ ) with **approximate solution** (logistic regression or linear SVM or ratchet Perceptron) to find weighted cost linear classifier.

### With approximate optimization

- allow to optimize both input weights and output weights of existing neurons (a few iterations of gradient descent),
- lost guarantee to find global optimum.
- can't rely on stopping criterion : use early stopping.
- comparable test errors (on toy 2-D problems, 2 moons) due to less overfitting.



# Continuum of Hidden Units

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x)$$

In the convex neural network formulation we considered  $n \rightarrow \infty$ .

# Continuum of Hidden Units

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x)$$

In the convex neural network formulation we considered  $n \rightarrow \infty$ .

## Low Dimensional Continuous Index

Why not replace discrete index  $i$  by a continuous index  $t$ ?

$$f_c(x) = \int w(t) g(z(t), x) dt$$

# Continuum of Hidden Units

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x)$$

In the convex neural network formulation we considered  $n \rightarrow \infty$ .

## Low Dimensional Continuous Index

Why not replace discrete index  $i$  by a continuous index  $t$ ?

$$f_c(x) = \int w(t) g(z(t), x) dt$$

## Link between the two formulations

Choose  $w(\cdot)$  a sum of diracs, **or**  $z(\cdot)$  and  $w(\cdot)$  piecewise constant :  
recover ordinary neural net.

## Previous Work

- Neural fields (Amari 77, Amari 83) introduced a long time ago... (one or two-dimensional)
- Bayesian neural networks (Neal 94) : countably infinite number of neurons with L2 weight decay on both output and input weights. Integrating over weight values to obtain posterior gives rise to similar integrals (not solved analytically, but using MCMC) : Gaussian Processes.
- In practice Gaussian Processes have been used with a Gaussian kernel.

# Kernel Formulation

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x) + b$$

# Kernel Formulation

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x) + b$$

## Continuous formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z) g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

# Kernel Formulation

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x) + b$$

## Continuous formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z) g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

## Link between the two formulations

If  $w = \sum_{i=1}^n w_i \delta(x - z_i)$ ,  $f_c(x) = f_d(x)$

# Kernel Formulation

## Usual formulation

$$f_d(x) = \sum_{i=1}^n w_i g(z_i, x) + b$$

## Continuous formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z) g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

## Link between the two formulations

If  $w = \sum_{i=1}^n w_i \delta(x - z_i)$ ,  $f_c(x) = f_d(x)$

Still fits in the convex neural network formulation.



# Optimization

## Formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z)g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

# Optimization

## Formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z)g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

## What do we optimize ?

The only thing to optimize is  $w$  (and  $b$ ), which corresponds to the output weights.

# Optimization

## Formulation

$$f_c(x) = \int_{\mathbb{R}^{d+1}} w(z)g(z, x) dz + b = \langle w, g_x \rangle_{\mathbb{R}^{d+1}} + b$$

## What do we optimize ?

The only thing to optimize is  $w$  (and  $b$ ), which corresponds to the output weights.

## How do we optimize it ?

This seems extremely difficult in the general case. However, some specific cases are feasible.

## Quadratic cost with $L^2$ penalty

$$f(x) = \langle w, g_x \rangle$$

### Cost function

$$C(w, b) = \|G'w + b - Y\|^2 + \lambda \|w\|^2$$

with  $i$ -th col. of  $G$  the fn  $g_{x_i} : x \mapsto G_i(x) = g(x, x_i)$ .

Gram matrix  $S = G'G, S_{ij} = \langle g_{x_i}, g_{x_j} \rangle$ . Kernel  $K(x, x') = \langle g_x, g_{x'} \rangle$ .

## Quadratic cost with $L^2$ penalty

$$f(x) = \langle w, g_x \rangle$$

### Cost function

$$C(w, b) = \|G'w + b - Y\|^2 + \lambda \|w\|^2$$

with  $i$ -th col. of  $G$  the fn  $g_{x_i} : x \mapsto G_i(x) = g(x, x_i)$ .

Gram matrix  $S = G'G, S_{ij} = \langle g_{x_i}, g_{x_j} \rangle$ . Kernel  $K(x, x') = \langle g_x, g_{x'} \rangle$ .

### Representer Theorems

When  $L$  is quadratic loss + quadratic penalty, **or hinge loss + quadratic penalty** (same as SVMs), then  $w = \sum_{i=1}^n \alpha_i g_{x_i}$ .

$$\implies f_c(x) = \sum_{i=1}^n \alpha_i K(x, x_i).$$

## Computing the Kernel

Let  $z = (v, c)$  with  $v \in \mathbb{R}^d$ , the weights, and  $c \in \mathbb{R}$  the bias.  
Consider  $g(x, z) = \text{sign}(\tilde{x} \cdot z)$ . N.B.  $g(x, z)$  independent of  $\|z\|$ .

# Computing the Kernel

Let  $z = (v, c)$  with  $v \in \mathbb{R}^d$ , the weights, and  $c \in \mathbb{R}$  the bias.  
Consider  $g(x, z) = \text{sign}(\tilde{x} \cdot z)$ . N.B.  $g(x, z)$  independent of  $\|z\|$ .

## Theorem

If  $\langle, \rangle$  integrates over  $z$  s.t.  $\|v\| = 1$  and  $c \in [-\delta\|w\|, \delta\|w\|]$ , then for any given value of  $\|w\|$ ,

$$K(x, y) \propto \delta \frac{\sqrt{d}}{2} - \|x - y\|$$

and  $\delta$  can be chosen arbitrarily large to include the support of the distribution, only changing the result by a constant.

## Locality ? Curse of Dimensionality ?

The above kernel is “kind of local” but does not fit the definition of locality we used to prove our curse of dimensionality results (does not become constant when  $\|x - y\|$  increases).



## Locality ? Curse of Dimensionality ?

The above kernel is “kind of local” but does not fit the definition of locality we used to prove our curse of dimensionality results (does not become constant when  $\|x - y\|$  increases).

## RBF neurons = Gaussian kernel

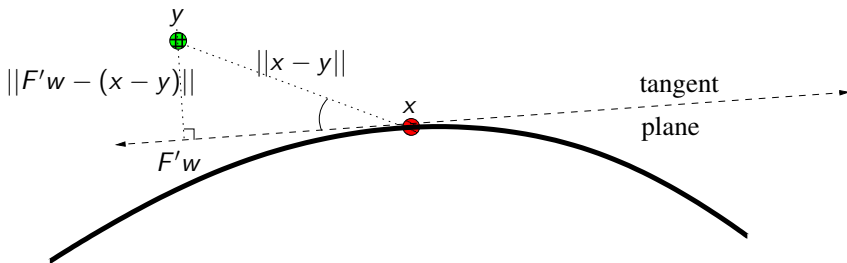
With RBF activation functions for hidden neurons, we obtain the Gaussian kernel predictor !

## Stepping Back

- Local kernels do not seem enough to learn many “interesting” functions, e.g. to deal with complex AI problems.
- How to transform a local kernel into a non-local one?
- How to generalize from what is learned around  $x$  to be able to say something around  $x'$  far from  $x$ ?
- Key idea : “re-use” = sharing of information

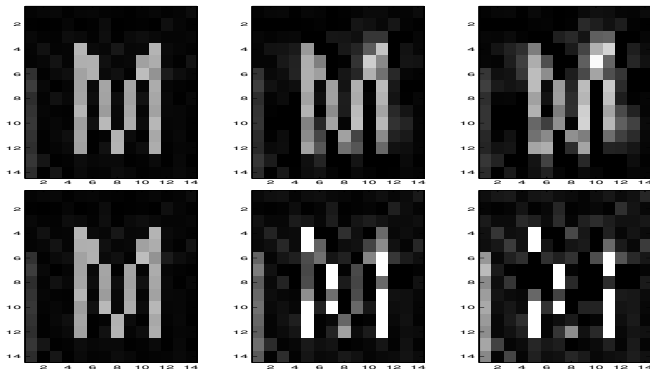
# Non-Local Tangent Manifold Learning

- Tangent plane basis vectors = **learned function** of  $x$  and **global** parameters  $\theta$ , with flexibly parametrized matrix-valued  $d \times n$  function  $F(x)$ . Example algorithm :  $F(x)$  is an MLP.
- Train  $d \times n$  function  $F(x)$  to approximately span the differences between  $x$  and its neighbors  $y$ .



## Experiments : Generalize FAR

Train on rotated **DIGITS** and generalize on **LETTERS** ! Learn rotation manifold.



Top : using neural network **DOES** rotate.

Bottom : using local predictor **DOES NOT** rotate.

# Local Manifold Parzen Windows

Regularized Gaussian mixture, centers near examples  $x_i$ , covariance matrices flat in “principal directions” :

$$p(y) = \frac{1}{n} \sum_{i=1}^n \text{Normal}(y; x_i + \mu(x_i), \text{Cov}(x_i))$$

where  $x_i + \mu(x_i)$  = Gaussian center and  $\text{Cov}(x_i)$  = covariance :

$$\text{Cov}(x_i) = \sigma_{\text{noise}}^2(x_i)I + \sum_{j=1}^k s_j(x_i) v_j(x_i) v_j(x_i)'$$

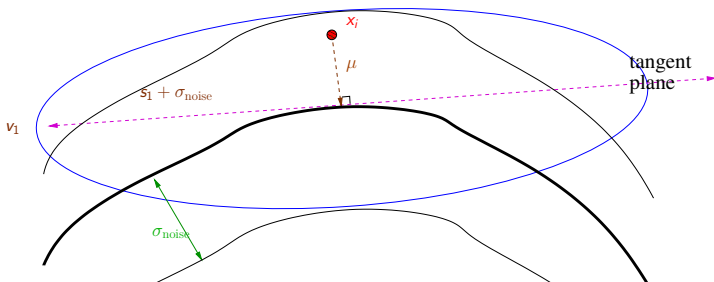
In (?),  $\mu(x_i) = 0$ , and  $\sigma_{\text{noise}}^2(x_i) = \sigma^2$  is a global hyper-parameter, while  $(\lambda_j(x_i), v_j(x_i)) = (s_j(x_i) + \sigma_{\text{noise}}^2(x_i), v_j(x_i))$  are **leading (eigenvalue, eigenvector) of local empirical covariance**.

# Non-Local Manifold Parzen

Consider  $\mu(x_i)$  and  $\text{Cov}(x_i)$  as **functions** of  $x_i$  with *global parameters* : **share information about the density across different regions of space.**

Neural network with  $x_i$  in input to predict  $\mu(x_i)$ ,  $\sigma_{\text{noise}}^2(x_i)$ , and the  $s_j(x_i)$  and  $v_j(x_i)$ . The  $v_j(x_i)$  do not need to be orthonormal.

- Gaussian near  $x_i$  tells where to expect neighbors of  $x_i$ .
- “Principal” vectors  $v_j(x_i)$  span tangent of manifold near  $x_i$ .



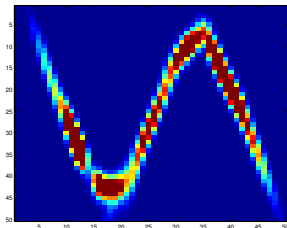
# Experimental Results

On two toy datasets : sinus and spiral, and a high-dimensional dataset : images of USPS digits. Training on all except 1, testing on 1.

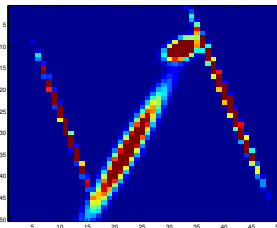
Hyper-parameters chosen by cross-validation

Out-of-sample Negative Log-Likelihood (NLL) :

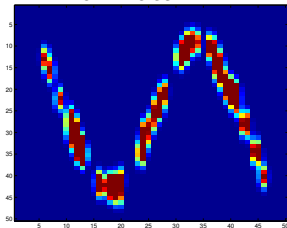
Algorithm	sinus	spiral	digits
<b>Non-Local Manifold Parzen</b>	<b>1.144</b>	<b>-1.346</b>	<b>-76.03</b>
Manifold Parzen	1.345	-0.914	58.33
Gaussian Mixture	1.567	-0.857	
Parzen Windows	1.841	-0.487	65.94



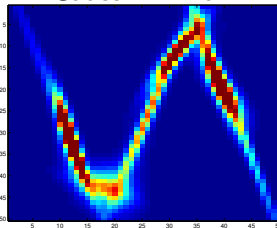
Non Local MP



Gauss Mix Full

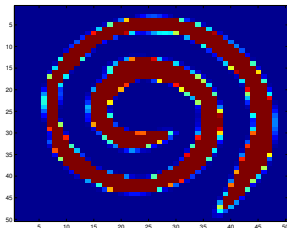


Parzen Window

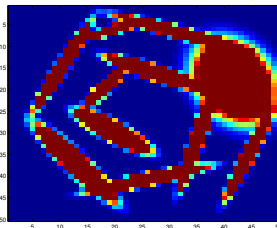


Manifold Parzen

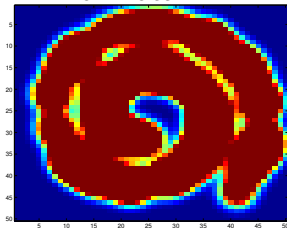




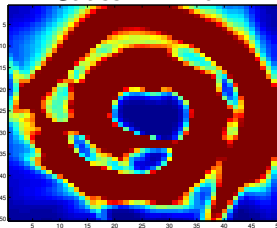
Non-Local MP



Gauss Mix Full



Parzen Window



Manifold Parzen

## Conclusions and Outlook

- Strong indications that **smoothness** prior alone == “**local kernels**”, such as the Gaussian and compact support kernels, yield to a form of **curse of dimensionality** for SVMs, spectral manifold learning, graph-based semi-supervised learning.
- **Neural nets** re-parametrized  $\Rightarrow$  **convex program**, using L1 penalty on output weights.  $O(n^d)$  algorithm finds **global optimum**. Approximations seem to work well.
- **Continuum of hidden units** yield to *computable* kernel formulations, radial but not compact. **With RBF activation, get Gaussian kernel ! (i.e. local).**
- L2 penalty  $\rightarrow$  kernel machines. With L1  $\rightarrow$  boosting
- Conjecture : **highly variable** functions that can be compactly represented can still be learned, using *broad priors* = non-local learning. KEY : RE-USE.