

# Who is afraid of non-convex loss functions?

Yann LeCun

The Courant Institute of Mathematical Sciences

New York University

# Convex Shmonvex

## ■ The NIPS community has suffered of an acute convexivitis epidemic

- ▶ ML applications seem to have trouble moving beyond logistic regression, SVMs, and exponential-family graphical models.
- ▶ For a new ML model, convexity is viewed as a virtue
- ▶ Convexity is sometimes a virtue
- ▶ But it is often a limitation
  
- ▶ ML theory has essentially never moved beyond convex models
  - the same way control theory has not really moved beyond linear systems
  
- ▶ Often, the price we pay for insisting on convexity is an unbearable increase in the size of the model, or the scaling properties of the optimization algorithm [ $O(n^2)$ ,  $O(n^3)\dots$ ]
- ▶ SDP-based manifold learning, QP-based kernel method, CRF. MMMN, .....

## Convex Shmonvex

- ➊ Other communities aren't as afraid as we are of non-convex optimization

- ▶ handwriting recognition
    - HMMs and Graph-Transformer-Network-based systems are non-convex
  - ▶ speech recognition
    - discriminative HMMs are non convex

- ➋ This is not by choice: non-convex models simply work better

- ▶ have you tried acoustic modeling in speech with a convex loss?

## To solve complicated AI tasks, ML will have to go non-convex

- paraphrasing the deep learning satellite session: Ultimately, complex learning tasks (e.g. vision, speech, language) will be implemented with “deep” hierarchical systems.
  - ▶ To learn hierarchical representations (low-level features, mid-level representations, high-level concepts....), we need “deep architectures”.
  - ▶ These inevitably lead to non-convex loss functions
- But wait! don't we have theorems that “shallow” (and convex) kernel methods can learn anything?
  - ▶ Yes. But that says nothing about the efficiency of the representation.
  - ▶ For example: there is empirical and theoretical evidence that shallow architectures cannot implement invariant visual recognition tasks efficiently
  - ▶ see [Bengio & LeCun 07] “scaling learning algorithms towards AI”

# Best Results on MNIST (from raw images: no preprocessing)

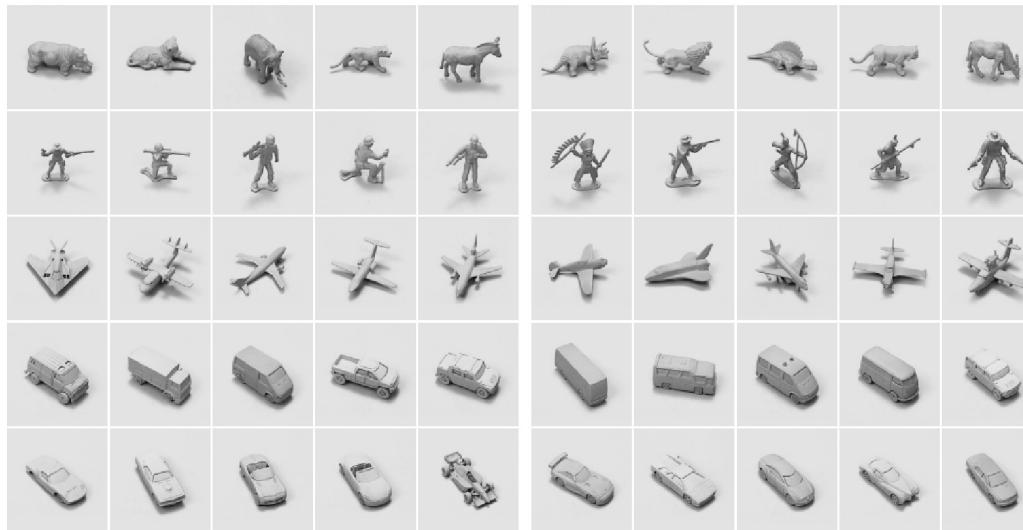
CLASSIFIER	DEFORMATION	ERROR	Reference
<b>Knowledge-free methods</b>			
2-layer NN, 800 HU, CE		1.60	Simard et al., ICDAR 2003
3-layer NN, 500+300 HU, CE, reg		1.53	Hinton, in press, 2005
SVM, Gaussian Kernel		1.40	Cortes 92 + Many others
Unsupervised Stacked RBM + backprop		0.95	Hinton, in press, 2005
<b>Convolutional nets</b>			
Convolutional net LeNet-5,		0.80	LeCun 2005 Unpublished
Convolutional net LeNet-6,		0.70	LeCun 2006 Unpublished
Conv. net LeNet-6- + unsup learning		0.60	LeCun 2006 Unpublished
<b>Training set augmented with Affine Distortions</b>			
2-layer NN, 800 HU, CE	Affine	1.10	Simard et al., ICDAR 2003
Virtual SVM deg-9 poly	Affine	0.80	Scholkopf
Convolutional net, CE	Affine	0.60	Simard et al., ICDAR 2003
<b>Training set augmented with Elastic Distortions</b>			
2-layer NN, 800 HU, CE	Elastic	0.70	Simard et al., ICDAR 2003
Convolutional net, CE	Elastic	0.40	Simard et al., ICDAR 2003
Conv. net LeNet-6- + unsup learning	Elastic	0.38	LeCun 2006 Unpublished

# Convexity is Overrated

- ➊ **Using a suitable architecture (even if it leads to non-convex loss functions) is more important than insisting on convexity (particularly if it restricts us to unsuitable architectures)**
  - ▶ e.g.: Shallow (convex) classifiers versus Deep (non-convex) classifiers
- ➋ **Even for shallow/convex architecture, such as SVM, using non-convex loss functions actually improves the accuracy and speed**
  - ▶ See “trading convexity for efficiency” by Collobert, Bottou, and Weston, ICML 2006 (best paper award)

# Normalized-Uniform Set: Error Rates

- ➊ Linear Classifier on raw stereo images: **30.2% error.**
- ➋ K-Nearest-Neighbors on raw stereo images: **18.4% error.**
- ➌ K-Nearest-Neighbors on PCA-95: **16.6% error.**
- ➍ Pairwise SVM on 96x96 stereo images: **11.6% error**
- ➎ Pairwise SVM on 95 Principal Components: **13.3% error.**
- ➏ Convolutional Net on 96x96 stereo images: **5.8% error.**



Training instances   Test instances

## Normalized-Uniform Set: Learning Times

	SVM	Conv Net				SVM/Conv
test error	11.6%	10.4%	6.2%	5.8%	6.2%	5.9%
train time (min*GHz)	480	64	384	640	3,200	50+
test time per sample (sec*GHz)	0.95			0.03		0.04+
#SV	28%					28%
parameters	$\sigma=2,000$ $C=40$					dim=80 $\sigma=5$ $C=0.01$



SVM: using a parallel implementation by  
Graf, Durdanovic, and Cosatto (NEC Labs)

Chop off the  
last layer of the  
convolutional net  
and train an SVM on it

## Experiment 2: Jittered-Cluttered Dataset



- ➊ **291,600 training samples, 58,320 test samples**
- ➋ **SVM with Gaussian kernel** **43.3% error**
- ⌂ **Convolutional Net with binocular input:** **7.8% error**
- ⌃ **Convolutional Net + SVM on top:** **5.9% error**
- ⌄ **Convolutional Net with monocular input:** **20.8% error**
- ⌅ **Smaller mono net (DEMO):** **26.0% error**
- ⌆ **Dataset available from <http://www.cs.nyu.edu/~yann>**

# Jittered-Cluttered Dataset

	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	$\sigma=10^4$ $C=40$				dim=100 $\sigma=5$ $C=1$

OUCH!

The convex loss, VC bounds  
and representer theorems  
don't seem to help

Chop off the last layer,  
and train an SVM on it  
it works!

# Optimization algorithms for learning

## ■ Neural nets:

- ▶ conjugate gradient, BFGS, LM-BFGS, ..... don't work as well as stochastic gradient

## ■ SVM:

- ▶ "batch" quadratic programming methods don't work as well as SMO. SMO don't work as well as recent on-line methods

## ■ CRF:

- ▶ Iterative scaling (or whatever) doesn't work as well as stochastic gradient (Schraudolph et al ICML 2006)
- ▶ The discriminative learning folks in speech and handwriting recognition have known this for a long time

## ■ Stochastic gradient has no good theoretical guarantees

- ▶ That doesn't mean we shouldn't use them, because the empirical evidence that it works better is overwhelming

## Theoretical Guarantees are overrated

- ➊ When Empirical Evidence suggests a fact for which we don't have theoretical guarantees, it just means the theory is inadequate.
- ➋ When empirical evidence and theory disagree, the theory is wrong.
- ➌ Let's not be afraid of methods for which we have no theoretical guarantee, particularly if they have been shown to work well
- ➍ But, let's aggressively look to those theoretical guarantees.
- ➎ We should use our theoretical understanding to expand our creativity, not to restrict it.

# The visual system is “deep” and learned

- ➊ The primate's visual system is “deep”

- ▶ It has 10-20 layers of neurons from the retina to the infero-temporal cortex (where object categories are encoded).
- ▶ How does it train itself by just looking at the world?.

- ➋ Is there a magic bullet for visual learning?

- ▶ The neo-cortex is pretty much the same all over
- ▶ The “learning algorithm” it implements is not specific to a modality (what works for vision works for audition)
- ▶ There is evidence that everything is learned, down to low-level feature detectors in V1
- ▶ Is there a **universal learning algorithm/architecture** which, given a small amount of appropriate prior structure, can produce an intelligent vision system?
- ▶ Or do we have to keep accumulating a large repertoire of pre-engineered “modules” to solve every specific problem an intelligent vision system must solve?

# Do we really need deep architectures?

- ➊ We can approximate any function as close as we want with shallow architecture. Why would we need deep ones?

$$y = \sum_{i=1}^P \alpha_i K(X, X^i) \quad y = F(W^1 \cdot F(W^0 \cdot X))$$

- ▶ kernel machines and 2-layer neural net are “universal”.

- ➋ Deep learning machines

$$y = F(W^K \cdot F(W^{K-1} \cdot F(\dots F(W^0 \cdot X) \dots)))$$

- ➌ Deep machines are more efficient for representing certain classes of functions, particularly those involved in visual recognition

- ▶ they can represent more complex functions with less “hardware”

- ➍ We need an efficient parameterization of the class of functions that we need to build intelligent machines (the “AI-set”)

# Why are Deep Architectures More Efficient?

## ➊ A deep architecture trades space for time

- ▶ more layers (more sequential computation),
- ▶ but less hardware (less parallel computation).
- ▶ Depth-Breadth tradeoff

## ➋ Example1: N-bit parity

- ▶ requires  $N-1$  XOR gates in a tree of depth  $\log(N)$ .
- ▶ requires an exponential number of gates if we restrict ourselves to 2 layers (DNF formula with exponential number of minterms).

## ➌ Example2: circuit for addition of 2 N-bit binary numbers

- ▶ Requires  $O(N)$  gates, and  $O(N)$  layers using  $N$  one-bit adders with ripple carry propagation.
- ▶ Requires lots of gates (some polynomial in  $N$ ) if we restrict ourselves to two layers (e.g. Disjunctive Normal Form).
- ▶ Bad news: almost all boolean functions have a DNF formula with an exponential number of minterms  $O(2^N)$ .....

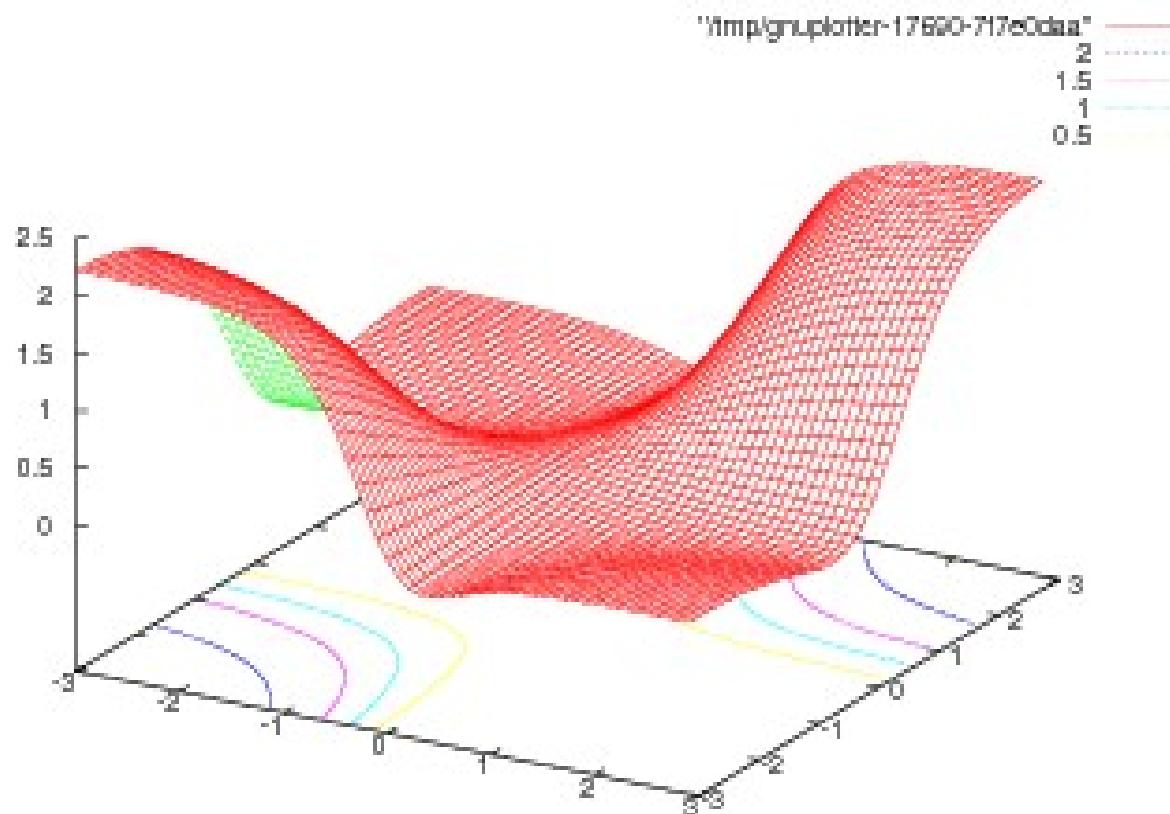
## Strategies (after Hinton 2007)

- ➊ **Defeatism:** since no good parameterization of the AI-set is available, let's parameterize a much smaller set for each specific task through careful engineering (preprocessing, kernel....).
- ➋ **Denial:** kernel machines can approximate anything we want, and the VC-bounds guarantee generalization. Why would we need anything else?
  - ▶ unfortunately, kernel machines with common kernels can only represent a tiny subset of functions efficiently
- ➌ **Optimism:** Let's look for learning models that can be applied to the largest possible subset of the AI-set, while requiring the smallest amount of task-specific knowledge for each task.
  - ▶ There is a parameterization of the AI-set with neurons.
  - ▶ Is there an efficient parameterization of the AI-set with computer technology?
- ➍ Today, the ML community oscillates between defeatism and denial.

# Deep Learning is Hard?

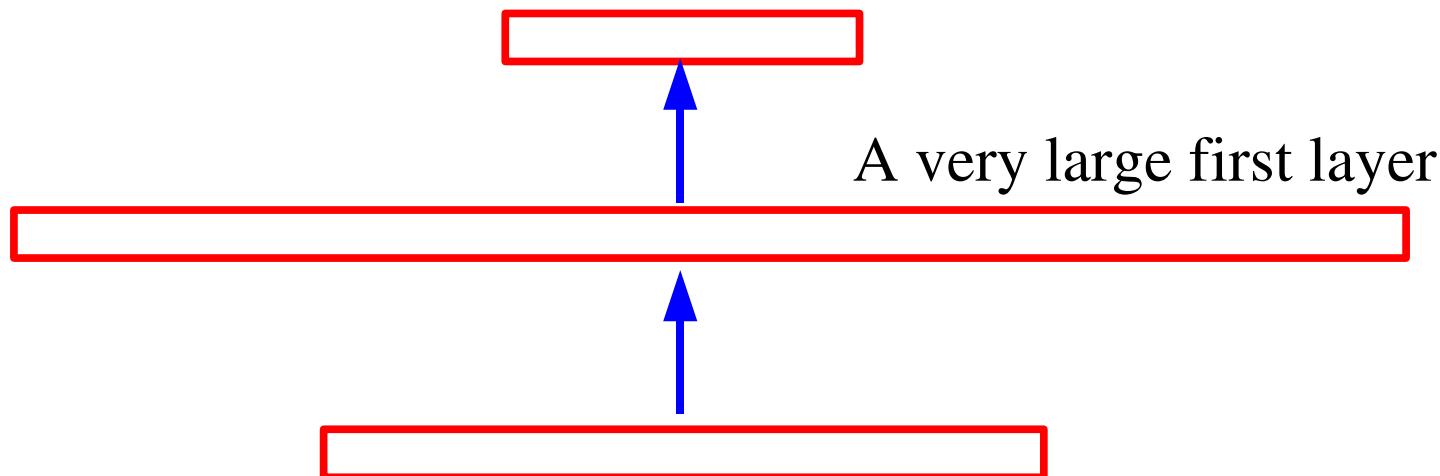
- ➊ Example: what is the loss function for the simplest 2-layer neural net ever
  - ▶ Function: 1-1-1 neural net. Map 0.5 to 0.5 and -0.5 to -0.5 (identity function) with quadratic cost:

$$y = \tanh(W_1 \tanh(W_0.x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0.5))^2$$



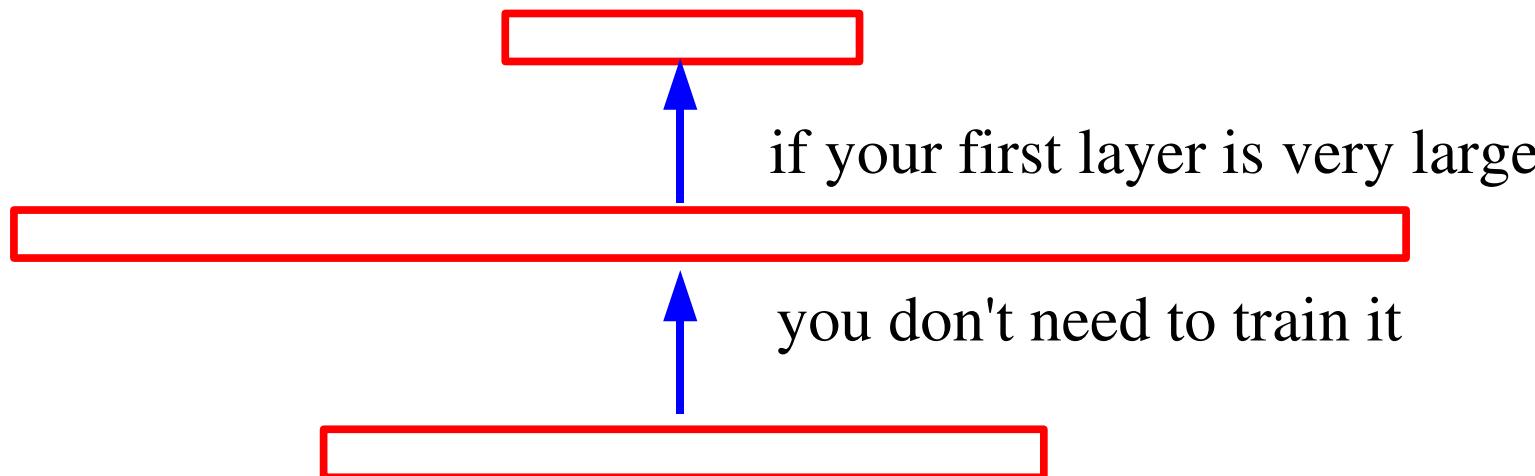
# Deep Learning is Hard?

- ➊ The loss surface is non-convex, ill-conditioned, has saddle points, has flat spots.....
- ➋ For large networks, it will be horrible!
- ➌ It will be horrible if the network is tall and skinny.
- ➍ It won't be too bad if the network is short and fat.



## Shallow models

- ➊ 1957: perceptron: fixed/random first layer. Trainable second layer
- ➋ 1985: backprop: both layers are trained. But many people are afraid of the lack of convergence guarantees
- ➌ 1992: kernel machines: large first layer with one template matcher for each training sample. Trainable second layer
  - ▶ sparsity in the second layer with hinge loss helps with efficiency, but not with accuracy



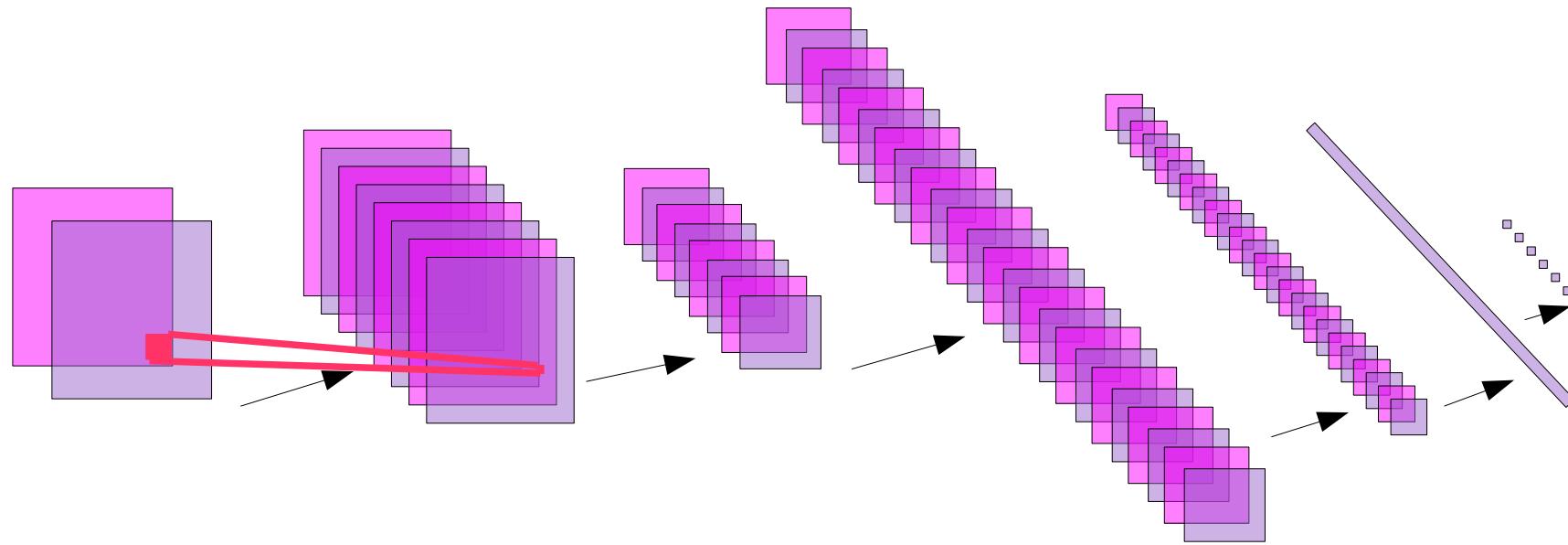
# The Problem with Non-Convex Learning

- ➊ **None of what you read in the optimization literature applies**
  - ▶ (see Leon Bottou's tutorial at NIPS 2007)
- ➋ **You need to use stochastic methods to take advantage of the redundancy in the data**
- ➌ **stochastic methods have horrible asymptotic properties, but that is irrelevant**
  - ▶ they converge very quickly to the “best solution” as measured by the test error
- ➍ **the optimization literature does not talk about stochastic methods**
  - ▶ forget about conjugate gradient, LM-BFGS, BFGS, Quasi-Newton...
- ➎ **the optimization literature deals with problems where  $O(N^2)$  is OK**
  - ▶ when you have  $10^6$  parameters,  $O(N)$  is all you can afford.

# Back-prop learning is not as bad as it seems

- ➊ gradient descent is unreliable when the network is small, particularly when the network has just the right size to learn the problem
- ➋ the solution is to make the network much larger than necessary and regularize it (SVM taught us that).
- ➌ Although there are lots of local minima, many of them are equivalent
  - ▶ it doesn't matter which one you fall into
  - ▶ with large nets, back-prop yields very consistent results
- ➍ Many local minima are due to symmetries in the system
- ➎ Breaking the symmetry in the architecture solves many problems
  - ▶ this may be why convolutional nets work so well

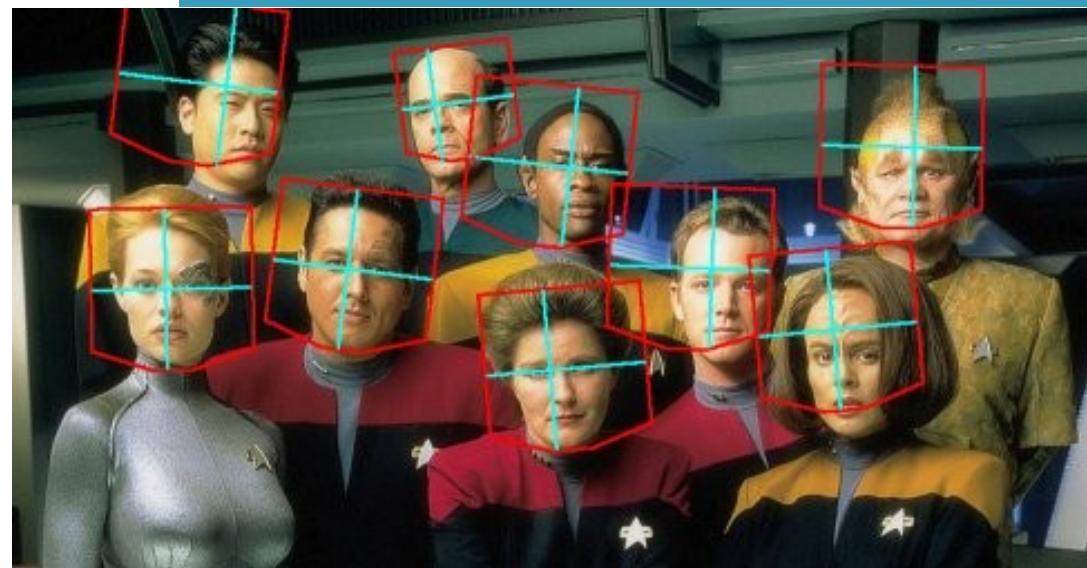
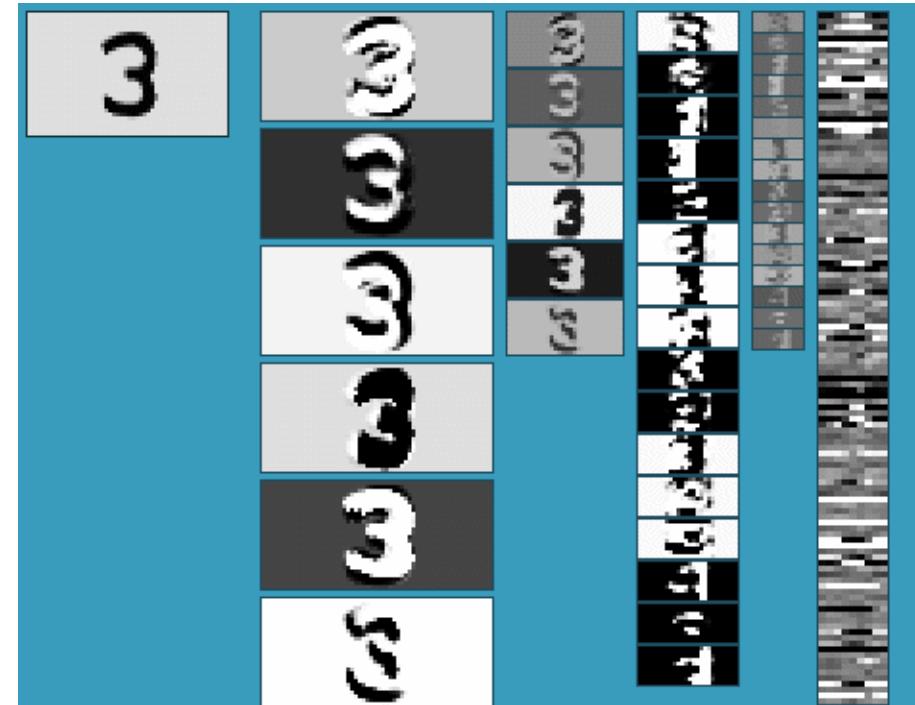
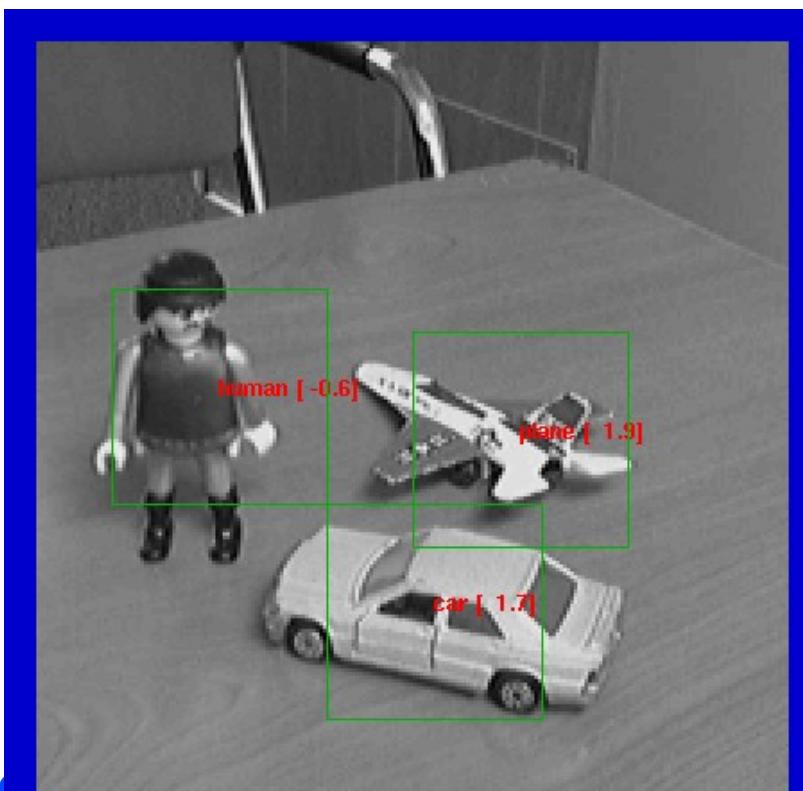
# Convolutional Networks



# Deep Supervised Learning works well with lots of data

- Supervised Convolutional nets work very well for:

- ▶ handwriting recognition(winner on MNIST)
- ▶ face detection
- ▶ object recognition with few classes and lots of training samples



**“Only Yann can do it” (NOT!)**

➊ What about Mike O'Neill?

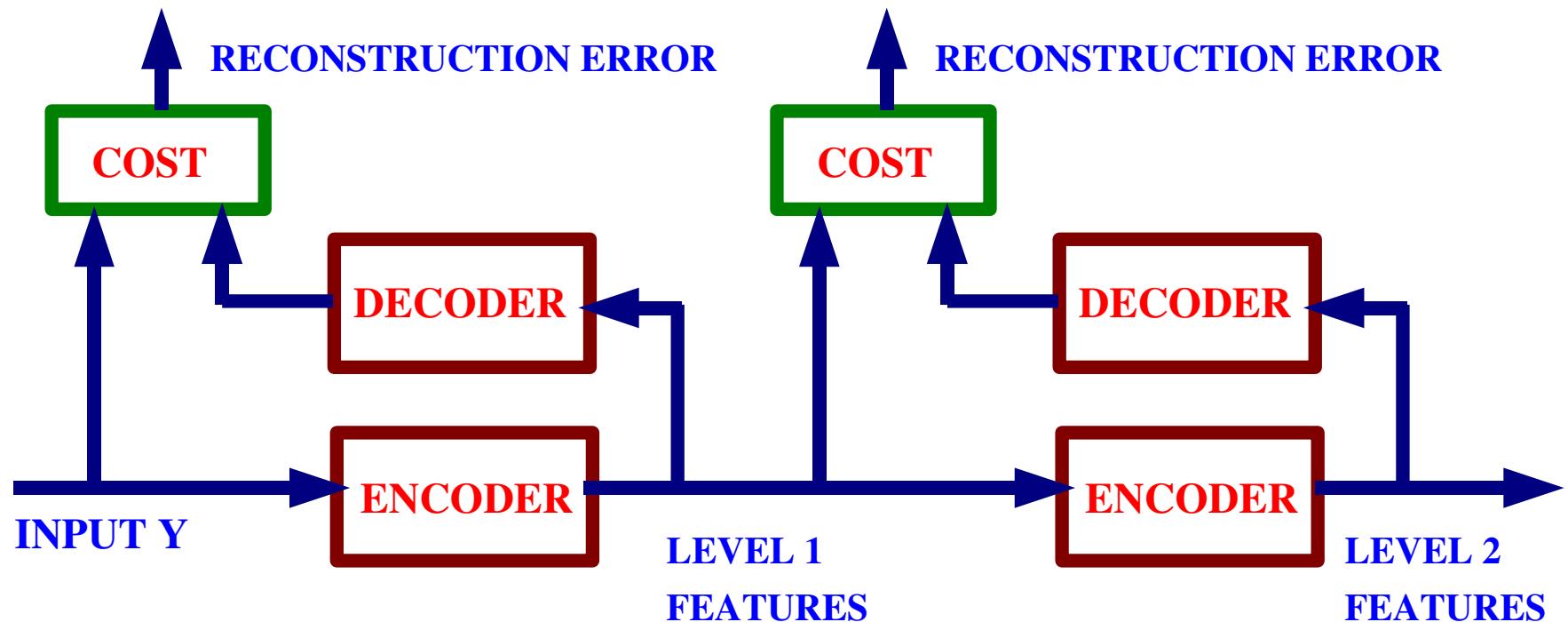
# The Basic Idea for Training Deep Feature Hierarchies

- Each stage is composed of

- an encoder that produces a feature vector from the input
- a decoder that reconstructs the input from the feature vector
  - (RBM is a special case)

- Each stage is trained one after the other

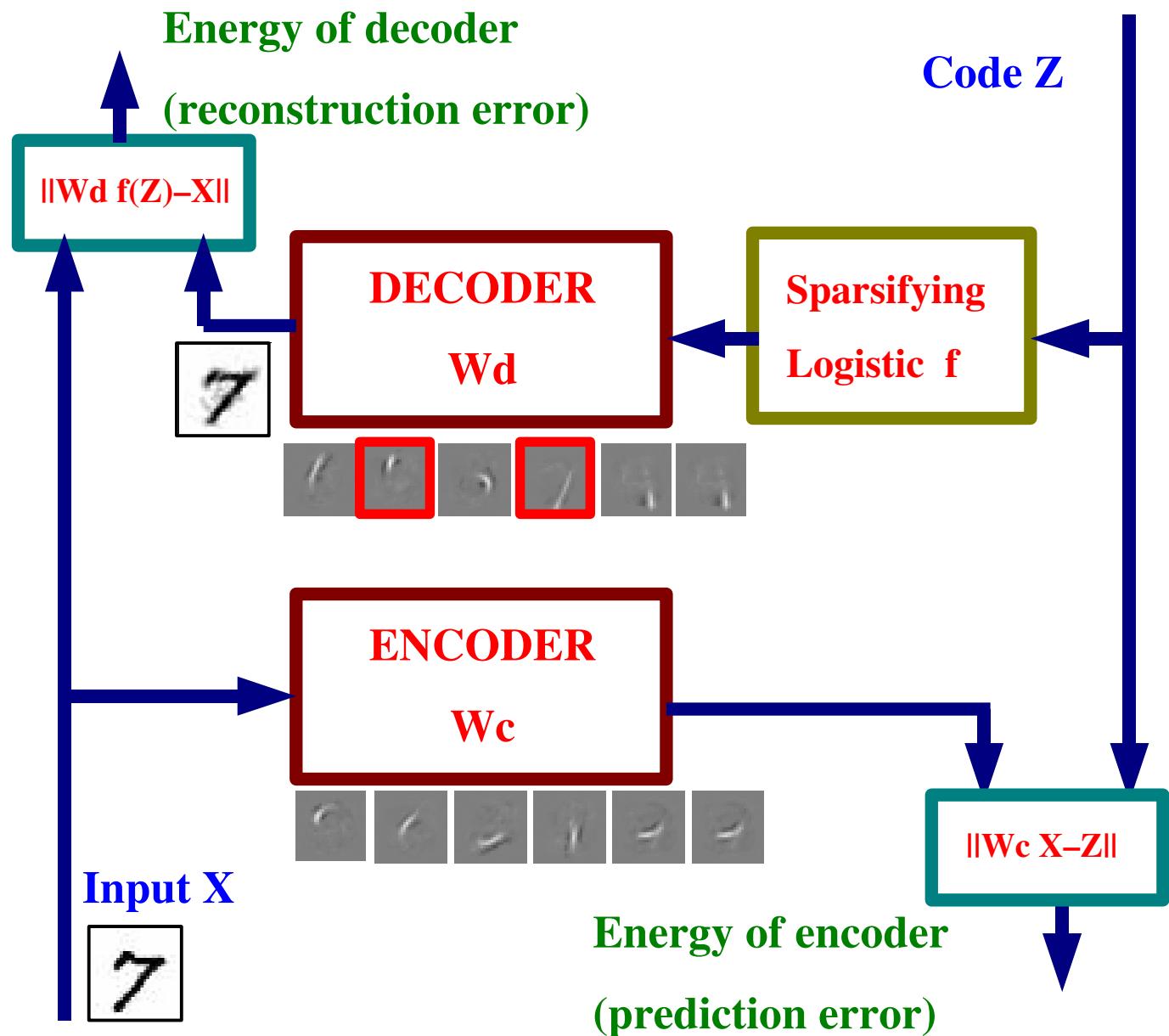
- the input to stage  $i+1$  is the feature vector of stage  $i$ .



# Sparsifying with a high-threshold logistic function

## Algorithm:

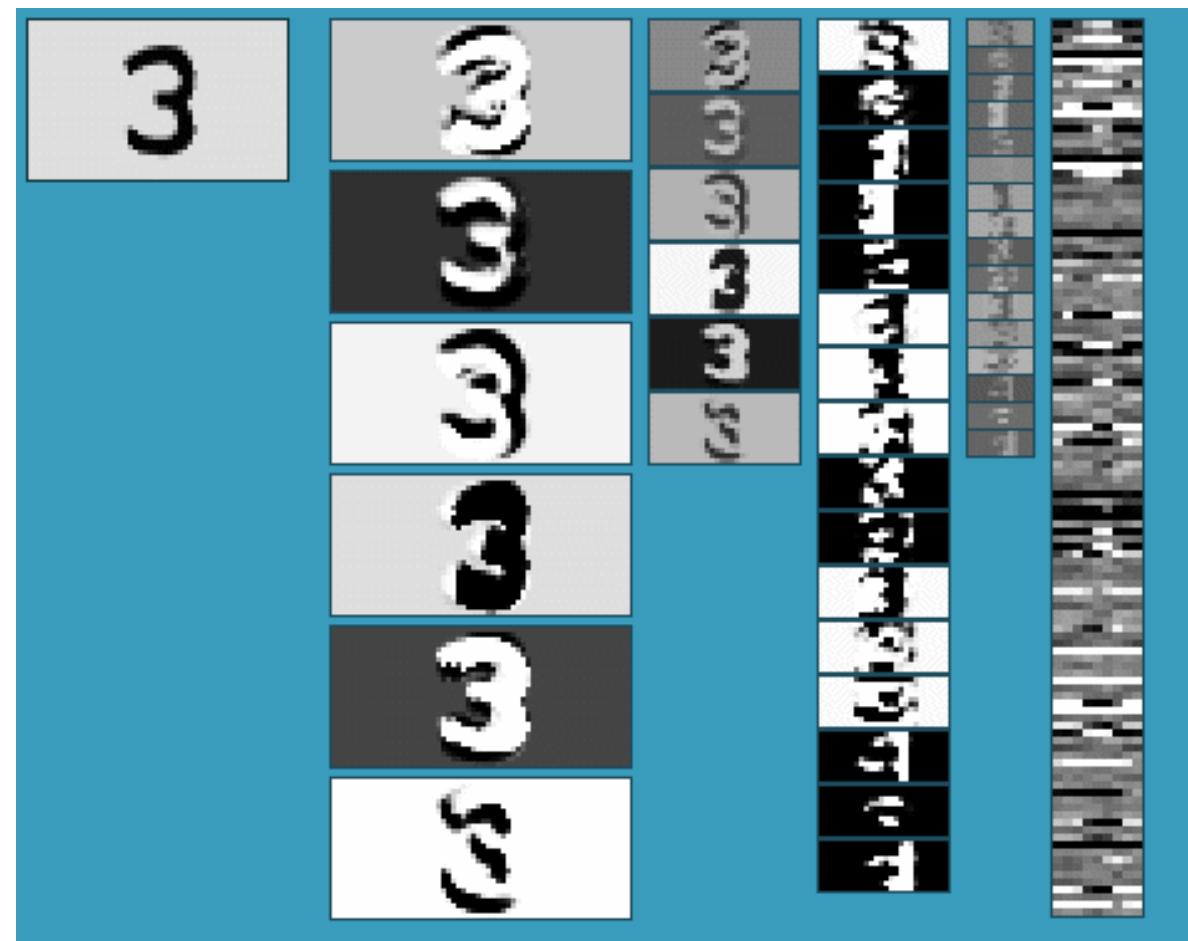
- 1. find the code  $Z$  that minimizes the reconstruction error AND is close to the encoder output
- 2. Update the weights of the decoder to decrease the reconstruction error
- 3. Update the weights of the encoder to decrease the prediction error



# The Multistage Hubel-Wiesel Architecture

## Building a complete artificial vision system:

- ▶ Stack multiple stages of simple cells / complex cells layers
- ▶ Higher stages compute more global, more invariant features
- ▶ Stick a classification layer on top
- ▶ [Fukushima 1971-1982]
  - neocognitron
- ▶ [LeCun 1988-2007]
  - convolutional net
- ▶ [Poggio 2002-2006]
  - HMAX
- ▶ [Ullman 2002-2006]
  - fragment hierarchy
- ▶ [Lowe 2006]
  - HMAX



# Unsupervised Training of Convolutional Filters

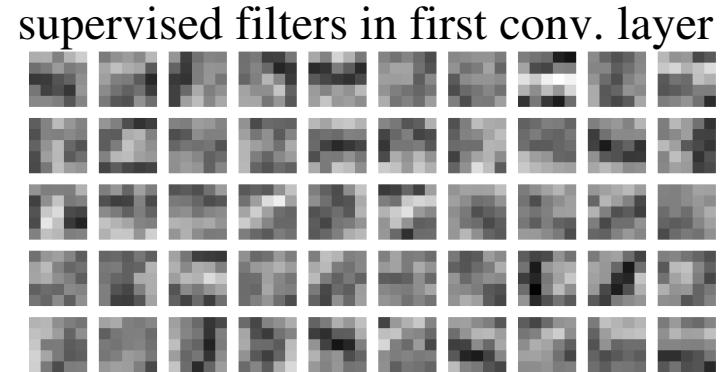
## CLASSIFICATION EXPERIMENTS

**IDEA:** improving supervised learning by pre-training  
with the unsupervised method (\*)

*sparse representations & lenet6 (1->50->50->200->10)*

- The **baseline**: *lenet6* initialized randomly

Test error rate: **0.70%**. Training error rate: 0.01%.

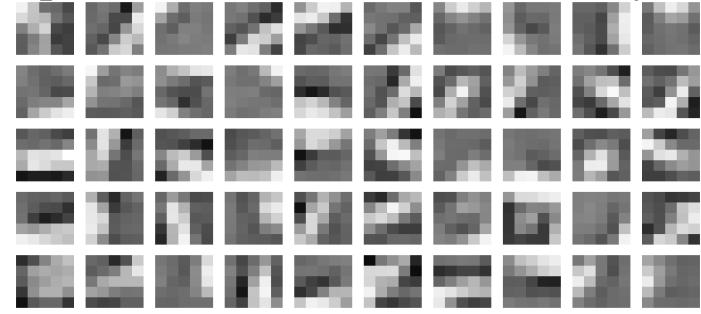


- *Experiment 1*

- Train on 5x5 patches to find 50 features
- Use the scaled filters in the encoder to initialize the kernels in the first convolutional layer

Test error rate: **0.60%**. Training error rate: 0.00%.

unsupervised filters in first conv. layer



- *Experiment 2*

- Same as experiment 1, but training set augmented by elastically distorted digits (random initialization gives test error rate equal to **0.49%**).

Test error rate: **0.39%**. Training error rate: 0.23%.

# Best Results on MNIST (from raw images: no preprocessing)

CLASSIFIER	DEFORMATION	ERROR	Reference
<b>Knowledge-free methods</b>			
2-layer NN, 800 HU, CE		1.60	Simard et al., ICDAR 2003
3-layer NN, 500+300 HU, CE, reg		1.53	Hinton, in press, 2005
SVM, Gaussian Kernel		1.40	Cortes 92 + Many others
Unsupervised Stacked RBM + backprop		0.95	Hinton, Neur Comp 2006
<b>Convolutional nets</b>			
Convolutional net LeNet-5,		0.80	Ranzato et al. NIPS 2006
Convolutional net LeNet-6,		0.70	Ranzato et al. NIPS 2006
Conv. net LeNet-6- + unsup learning		0.60	Ranzato et al. NIPS 2006
<b>Training set augmented with Affine Distortions</b>			
2-layer NN, 800 HU, CE	Affine	1.10	Simard et al., ICDAR 2003
Virtual SVM deg-9 poly	Affine	0.80	Scholkopf
Convolutional net, CE	Affine	0.60	Simard et al., ICDAR 2003
<b>Training set augmented with Elastic Distortions</b>			
2-layer NN, 800 HU, CE	Elastic	0.70	Simard et al., ICDAR 2003
Convolutional net, CE	Elastic	0.40	Simard et al., ICDAR 2003
Conv. net LeNet-6- + unsup learning	Elastic	0.39	Ranzato et al. NIPS 2006

# Learning Invariant Filters in a Convolutional Net

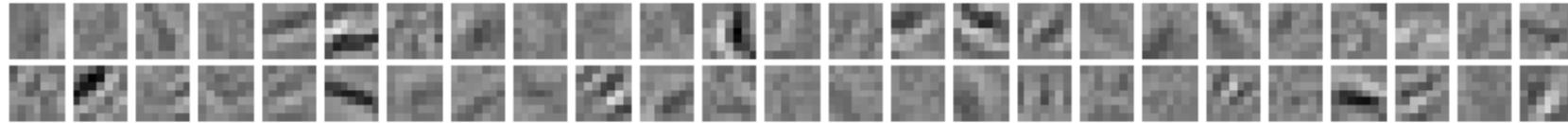


Figure 1: 50 7x7 filters in the first convolutional layer that were learned by the network trained supervised from *random* initial conditions with 600K digits.

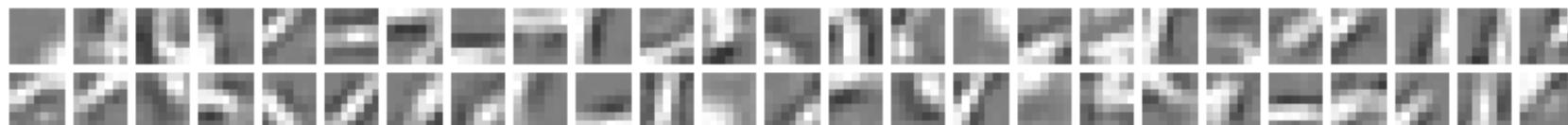


Figure 2: 50 7x7 filters that were learned by the unsupervised method (on 60K digits), and that are used to initialize the first convolutional layer of the network.

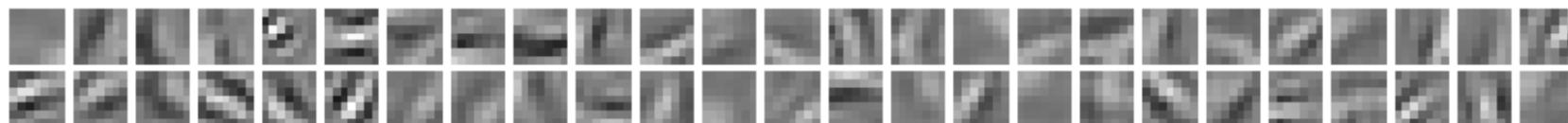
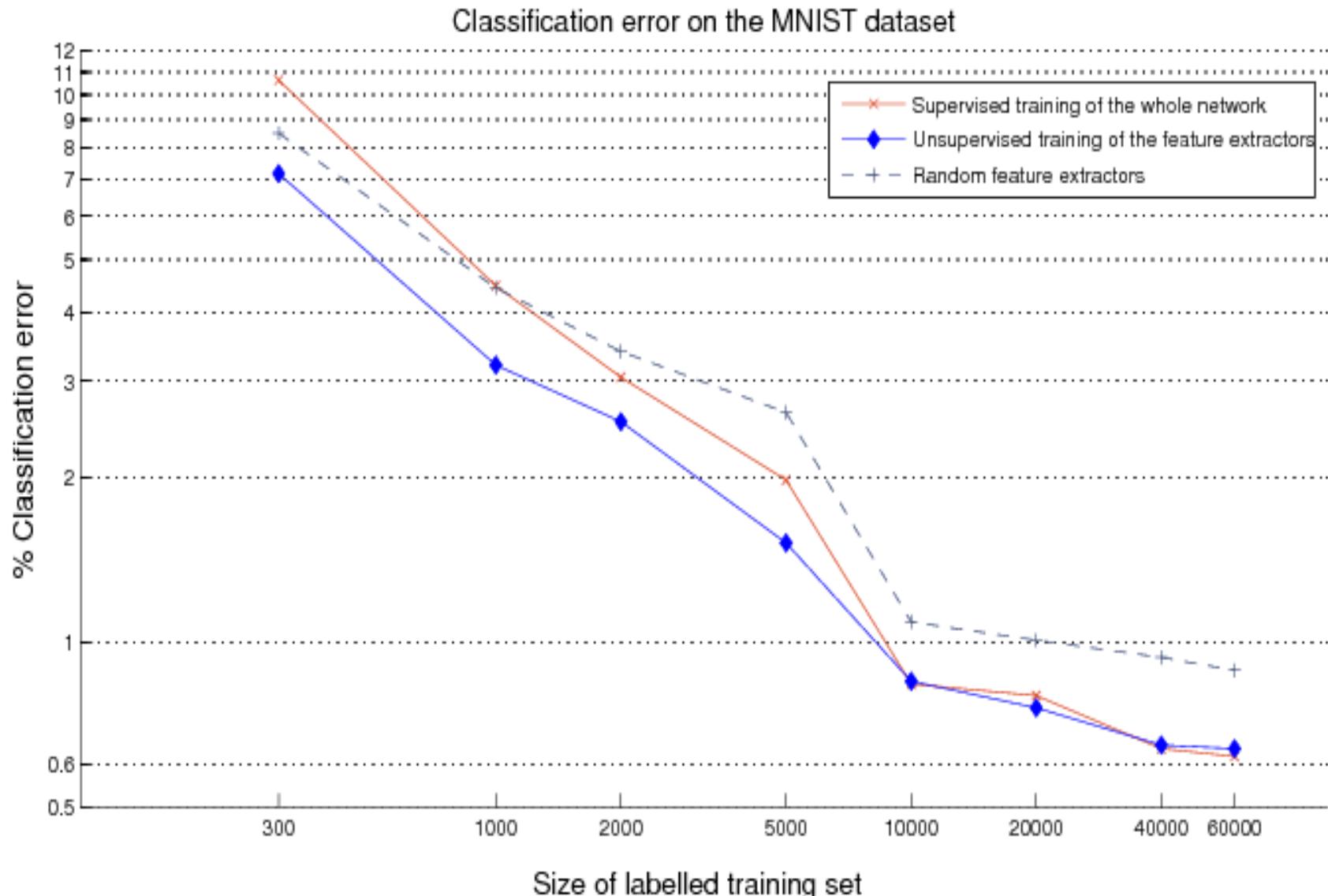


Figure 3: 50 7x7 filters in the first convolutional layer that were learned by the network trained supervised from the initial conditions given by the *unsupervised method* (see fig.2) with 600K digits.

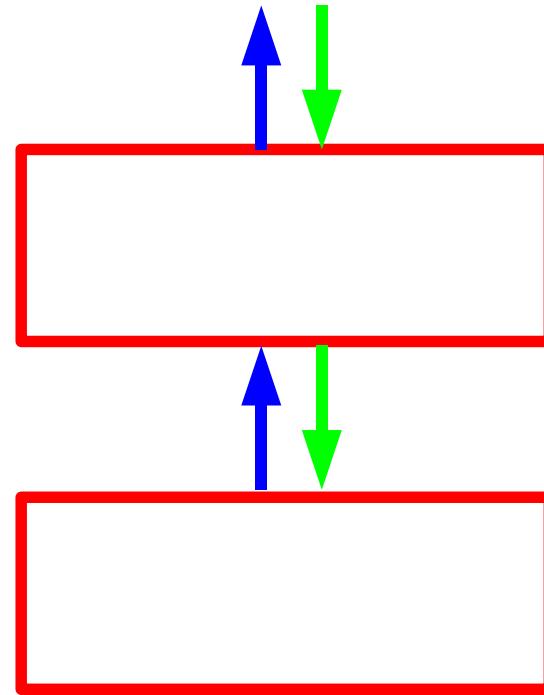
# Influence of Number of Training Samples



# The right tools: Automatic Differentiation

## Object-Oriented, module-based AD

- ▶ [Bottou & Gallinari 1991]
- ▶ Implemented in Lush [Bottou & LeCun]
- ▶ Implemented in Torch [Collobert]
- ▶ Implemented in Monte Python [Memisevic]



`module.fprop(input,output)`  
`module.bprop(input,output)`  
`module.bbprop(input,output)`

# SQUARE JACOBIAN APPROXIMATION FOR GAUSS-NEWTON AND LEVENBERG-MARQUARDT ALGOS.

**Assume the cost function is the Mean Square Error:**

$$E(\omega) = \frac{1}{2} \sum_p (D_p - N(\omega, \chi_p))' (D_p - N(\omega, \chi_p))$$

**Gradient:**

$$\frac{\partial E(\omega)}{\partial \omega} = - \sum_p (D_p - N(\omega, \chi_p))' \frac{\partial N(\omega, \chi_p)}{\partial \omega}$$

**Hessian:**

$$H(\omega) = \sum_p \frac{\partial N(\omega, \chi_p)}{\partial \omega}' \frac{\partial N(\omega, \chi_p)}{\partial \omega} + \sum_p (D_p - N(\omega, \chi_p))' \frac{\partial^2 N(\omega, \chi_p)}{\partial \omega \partial \omega}$$

**Simplified Hessian (square of the Jacobian):**

$$H(\omega) = \sum_p \frac{\partial N(\omega, \chi_p)}{\partial \omega}' \frac{\partial N(\omega, \chi_p)}{\partial \omega}$$

Jacobian: NxO matrix  
(O: number of outputs)

- the resulting approximate Hessian is positive semi-definite
- dropping the second term is equivalent to assuming that the network is a linear function of the parameters

**RECIPE for computing the k-th column of the Jacobian:**  
for all training patterns {

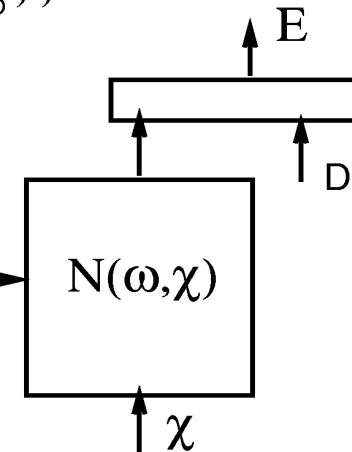
    forward prop

    set gradients of output units to 0;

    set gradient of k-th output unit to 1;

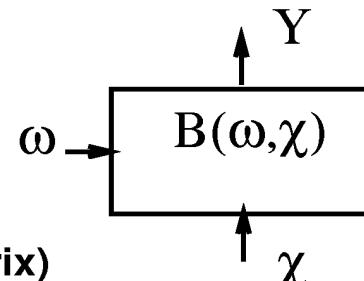
    back propagate; accumulate gradient;

}



## BACKPROPAGATING SECOND DERIVATIVES

A multilayer system composed of functional blocs. Consider one of the blocs with I inputs, O outputs, and N parameters



Assuming we know  $\frac{\partial^2 E}{\partial Y^2}$  (OxO matrix)

what are  $\frac{\partial^2 E}{\partial \omega^2}$  (NxN matrix) and  $\frac{\partial^2 E}{\partial \chi^2}$  (IxI matrix)

Chain rule for 2nd derivatives:

$$\frac{\partial^2 E}{\partial \omega^2} = \frac{\partial Y}{\partial \omega} \frac{\partial^2 E}{\partial Y^2} \frac{\partial Y}{\partial \omega} + \frac{\partial E}{\partial Y} \frac{\partial^2 Y}{\partial \omega^2}$$

↑ NxN    ↑ NxO    ↑ OxO    ↑ OxN  
↓ 1xO    ↓ OxNxN

ignore this!

The above can be used to compute a bloc diagonal subset of the Hessian

If the term in the red square is dropped, the resulting Hessian estimate will be positive semi-definite

If we are only interested in the diagonal terms, it reduces to:

$$\frac{\partial^2 E}{\partial \omega_{ii}^2} = \sum_k \frac{\partial^2 E}{\partial Y_{kk}^2} \left( \frac{\partial Y_{kk}}{\partial \omega_{ii}} \right)^2 \quad (\text{and same with } \chi \text{ instead of } \omega)$$

# BACKPROPAGATING THE DIAGONAL HESSIAN IN NEURAL NETS

(with the square Jacobian approximation)

[LeCun 87, Becker&LeCun 88, LeCun 89]

Sigmoids (and other scalar functions)

$$\frac{\partial^2 E}{\partial Y_k^2} = \frac{\partial^2 E}{\partial Z_k^2} (f'(Y_k))^2$$

Weighted sums

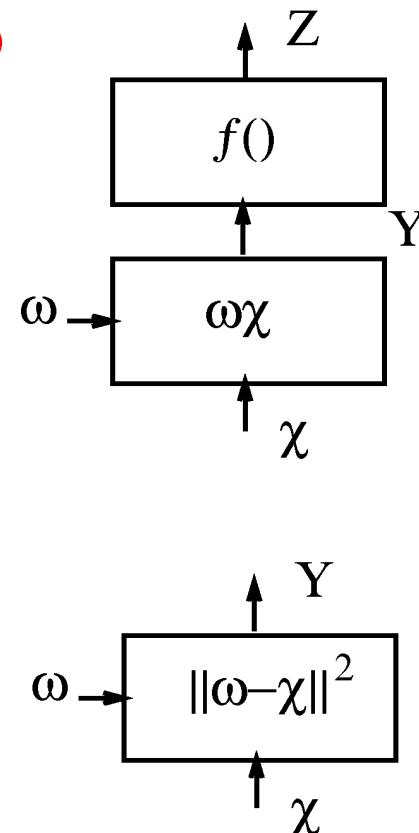
$$\frac{\partial^2 E}{\partial \omega_{ki}^2} = \frac{\partial^2 E}{\partial Y_k^2} \chi_i^2$$

$$\frac{\partial^2 E}{\partial \chi_i^2} = \sum_k \frac{\partial^2 E}{\partial Y_k^2} \omega_{ki}^2$$

RBFs

$$\frac{\partial^2 E}{\partial \omega_{ki}^2} = \frac{\partial^2 E}{\partial Y_k^2} (\chi_i - \omega_{ki})^2$$

$$\frac{\partial^2 E}{\partial \chi_i^2} = \sum_k \frac{\partial^2 E}{\partial Y_k^2} (\chi_i - \omega_{ki})^2$$



(the 2nd derivatives with respect to the weights should be averaged over the training set)

SAME COST AS REGULAR BACKPROP

the "OBD" network pruning techniques uses this procedure [LeCun,Denker&Solla 90]

# COMPUTING THE PRODUCT OF THE HESSIAN BY A VECTOR

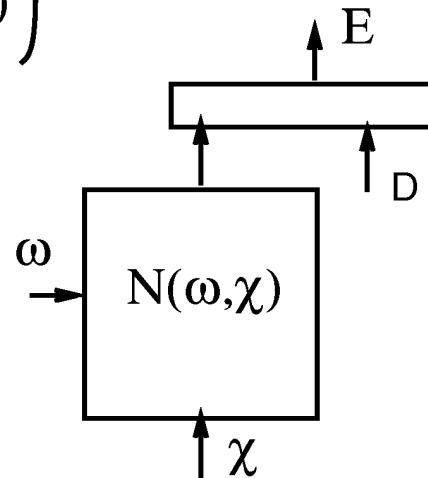
(without computing the Hessian itself)

Finite difference:

$$H\Psi \approx \frac{1}{\alpha} \left( \frac{\partial E}{\partial \omega} (\omega + \alpha \Psi) - \frac{\partial E}{\partial \omega} (\omega) \right)$$

RECIPE for computing the product of a vector  $\Psi$  by the Hessian:

- 1- compute gradient
- 2- add  $\alpha \Psi$  to the parameter vector
- 3- compute gradient with perturbed parameters
- 4- subtract result of 1 from 3, divide by  $\alpha$



This method can be used to compute the principal eigenvector and eigenvalue of  $H$  by the power method.

By iterating     $\Psi \leftarrow H\Psi / \|\Psi\|$

$Y$

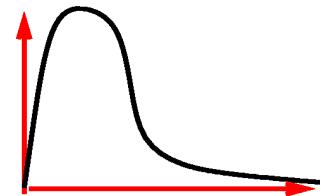
will converge to the principal eigenvector of  $H$  and  $\|\Psi\|$  to the corresponding eigenvalue  
[LeCun, Simard&Pearlmutter 93]

A more accurate method which does not use finite differences (and has the same complexity) has recently been proposed [Pearlmutter 93]

## ANALYSIS OF THE HESSIAN IN MULTILAYER NETWORKS

- What does the Hessian of a multilayer network look like?
- How does it change with the architecture and the details of the implementation?
- Typically, the distribution of eigenvalues of a multilayer network looks like this:

a few small eigenvalues, a large number of medium ones, and a small number of very large ones



These large ones are the killers

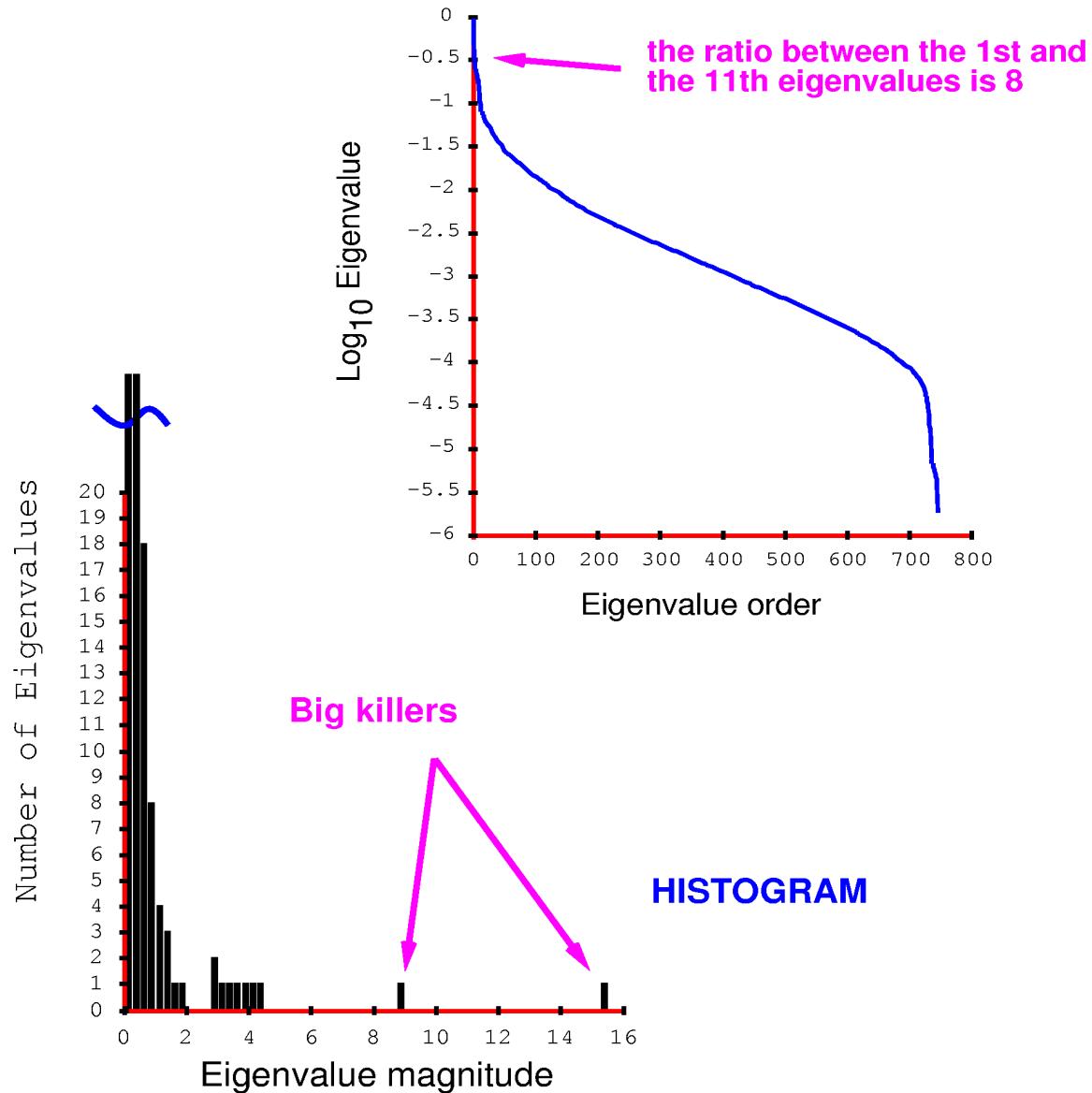
They come from:

- non-zero mean inputs or neuron states
- wide variations second derivatives from layer to layer
- correlations between state variables

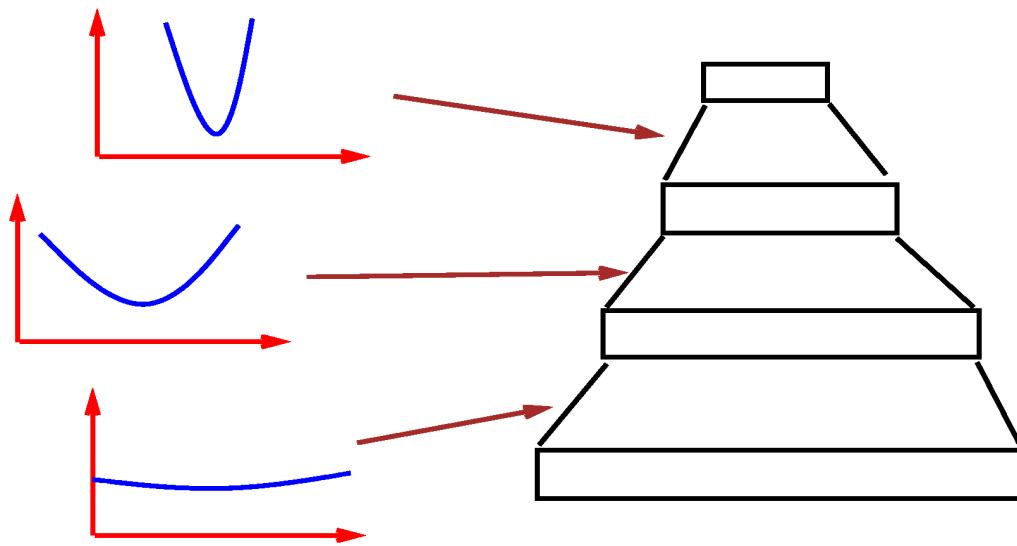
for more details see [LeCun, Simard&Pearlmutter 93]  
[LeCun, Kanter&Solla 91]

## EIGENVALUE SPECTRUM

Network: 256–128–64–10 with local connections and shared weights (around 750 parameters)  
Data set: 320 handwritten digits



## MULTILAYER NETWORKS HESSIAN



**The second derivative is often smaller in lower layers. The first layer weights learn very slowly, while the last layer weights change very quickly.**

**This can be compensated for using the diagonal 2nd derivatives (more on this later)**

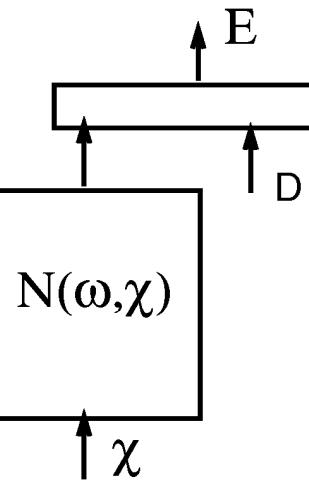
## GAUSS-NEWTON AND LEVENBERG-MARQUARDT METHODS

These methods only apply to Mean-Square Error objective functions (non-linear least square).

Gauss-Newton algorithm:

like Newton but the Hessian is approximated by the square of the jacobian (which is always positive semidefinite)

$$\Delta\omega = \left( \sum_p \frac{\partial N(\omega, \chi_p)}{\partial \omega} \frac{\partial N(\omega, \chi_p)}{\partial \omega} \right)^{-1} \nabla E(\omega)$$



Levenberg-Marquardt algorithm:

like Gauss-Newton, but has a safeguard parameter to prevent it from blowing up if some eigenvalues are small

$$\Delta\omega = \left( \sum_p \frac{\partial N(\omega, \chi_p)}{\partial \omega} \frac{\partial N(\omega, \chi_p)}{\partial \omega} + \mu I \right)^{-1} \nabla E(\omega)$$

- Both are  $O(N^3)$  algorithms
- they are widely used in statistics for regression
- they are only practical for small numbers of parameters.
- they do not require a line search, so in principle they can be used in stochastic mode (although that has not been tested)

# A STOCHASTIC DIAGONAL LEVENBERG–MARQUARDT METHOD

[LeCun 87, Becker&LeCun 88, LeCun 89]

## THE MAIN IDEAS:

- use formulae for the backpropagation of the diagonal Hessian (shown earlier) to keep a running estimate of the second derivative of the error with respect to each parameter.
- use these term in a "Levenberg–Marquardt" formula to scale each parameter's learning rate

Each parameter (weight)  $\omega_{ki}$  has its own learning rate  $\eta_{ki}$  computed as:

$$\eta_{ki} = \frac{\varepsilon}{\frac{\partial^2 E}{\partial \omega_{ki}^2} + \mu}$$

$\varepsilon$  is a global "learning rate"  
 $\frac{\partial^2 E}{\partial \omega_{ki}^2}$  is an estimate of the diagonal second derivative with respect to weight (ki)

$\mu$  is a "Levenberg–Marquardt" parameter to prevent  $\eta_{ki}$  from blowing up if the 2nd derivative is small

## A STOCHASTIC DIAGONAL LEVENBERG-MARQUARDT METHOD

The second derivatives  $\frac{\partial^2 E}{\partial \omega_{ki}^2}$  can be computed using a running average formula over a subset of the training set prior to training:

$$\frac{\partial^2 E}{\partial \omega_{ki}^2} \leftarrow (1-\gamma) \frac{\partial^2 E}{\partial \omega_{ki}^2} + \gamma \frac{\partial^2 E^p}{\partial \omega_{ki}^2}$$

new estimate  
of 2nd der.    previous  
estimate    small  
constant    instantaneous  
2nd der. for  
pattern p

The instantaneous second derivatives are computed using the formula in the slide entitled:  
**"BACKPROPAGATING THE DIAGONAL HESSIAN IN NEURAL NETS"**

Since the second derivatives evolve slowly, there is no need to reestimate them often.  
They can be estimated once at the beginning by sweeping over a few hundred patterns.  
Then, they can be reestimated every few epochs.

The additional cost over regular backprop is negligible.

Is usually about 3 times faster than carefully tuned stochastic gradient.

# Stochastic Diagonal Levenberg–Marquardt

data set: set-1 (100 examples, 2 gaussians)

network: 1 linear unit, 2 inputs, 1 output.  
2 weights, 1 bias.

Learning  
rates:

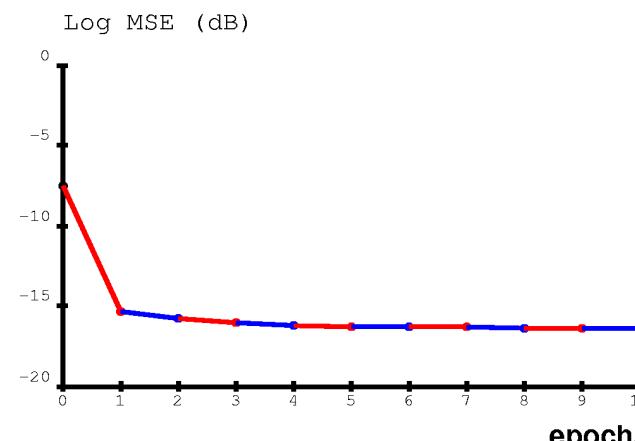
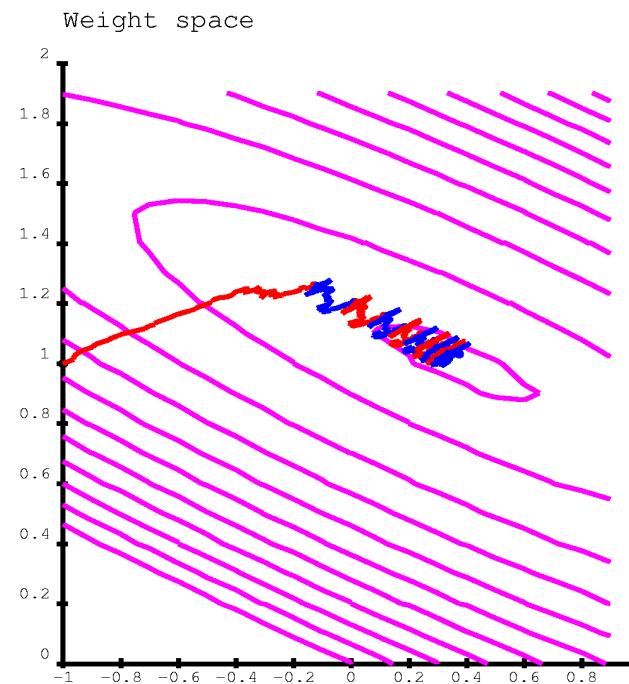
$$\begin{aligned}\eta_0 &= 0.12 \\ \eta_1 &= 0.03 \\ \eta_2 &= 0.02\end{aligned}$$

Hessian  
largest  
eigenvalue:

$$\lambda_{\max} = 0.84$$

Maximum  
admissible  
Learning  
rate (batch):

$$\eta_{\max} = 2.38$$



# Stochastic Diagonal Levenberg–Marquardt

data set: set-1 (100 examples, 2 gaussians)

network: 1 linear unit, 2 inputs, 1 output.  
2 weights, 1 bias.

Learning  
rates:

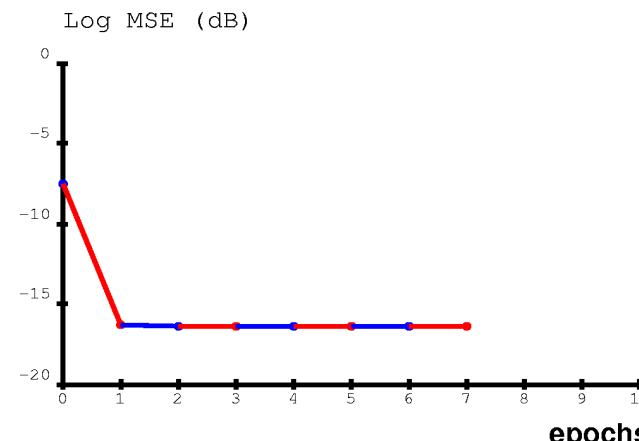
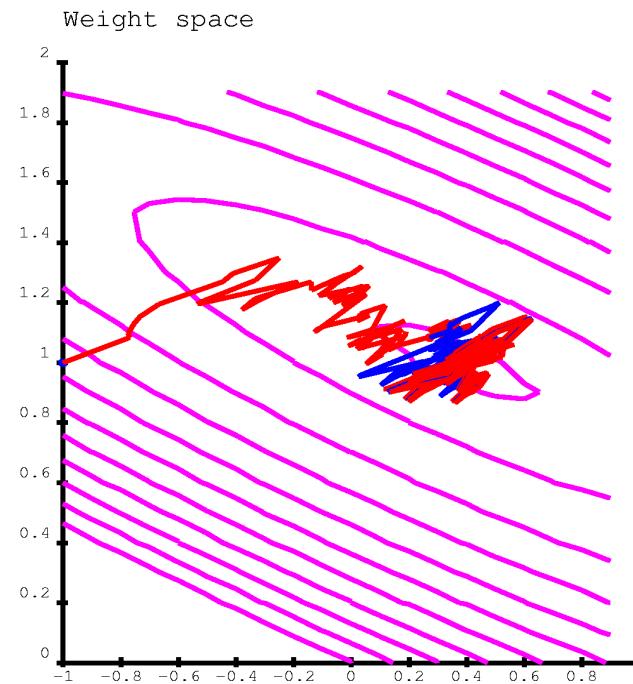
$$\begin{aligned}\eta_0 &= 0.76 \\ \eta_1 &= 0.18 \\ \eta_2 &= 0.12\end{aligned}$$

Hessian  
largest  
eigenvalue:

$$\lambda_{\max} = 0.84$$

Maximum  
admissible  
Learning  
rate (batch):

$$\eta_{\max} = 2.38$$



## COMPUTING THE PRINCIPAL EIGENVALUE/VECTOR OF THE HESSIAN

without computing the Hessian

**IDEA #1 (the power method):**

1 – Choose a vector  $\Psi$  at random

2 – iterate:  $\Psi \leftarrow H \frac{\Psi}{\|\Psi\|}$

OLD ESTIMATE OF EIGENVECTOR  
HESSIAN  
ESTIMATE OF EIGENVALUE

$\Psi$  will converge to the principal eigenvector (or a vector in the principal eigenspace)

$\|\Psi\|$  will converge to the corresponding eigenvalue

## COMPUTING THE PRODUCT $H \Psi$

**IDEA #2 (Taylor expansion):**

$$\Psi \leftarrow \frac{1}{\alpha} \left( \frac{\partial E}{\partial \omega} (\omega + \alpha \frac{\Psi}{\|\Psi\|}) - \frac{\partial E}{\partial \omega} (\omega) \right)$$

Annotations:

- NEW ESTIMATE OF EIGENVECTOR**: Points to  $\Psi$ .
- "SMALL" CONSTANT**: Points to  $\alpha$ .
- PERTURBED GRADIENT**: Points to  $\frac{\partial E}{\partial \omega} (\omega + \alpha \frac{\Psi}{\|\Psi\|})$ .
- OLD ESTIMATE OF EIGENVECTOR**: Points to  $\omega$ .
- GRADIENT**: Points to  $\frac{\partial E}{\partial \omega} (\omega)$ .

**One iteration of this procedure requires 2 forward props and 2 backward props for each pattern in the training set.**

**This converges very quickly to a good estimate of the largest eigenvalue of  $H$**

## ON-LINE COMPUTATION OF $\Psi$

**IDEA #3 (running average):**

$$\Psi \leftarrow (1-\gamma)\Psi + \gamma \frac{1}{\alpha} \left( \frac{\partial E^p}{\partial \omega} (\omega + \alpha \frac{\Psi}{\|\Psi\|}) - \frac{\partial E^p}{\partial \omega} (\omega) \right)$$

Diagram illustrating the components of the update equation:

- NEW ESTIMATE OF EIGENVECTOR**: The result of the update, indicated by a curved arrow.
- "SMALL" CONSTANTS**: The scalar factors  $(1-\gamma)$  and  $\gamma$ .
- OLD ESTIMATE OF EIGENVECTOR**: The previous estimate of the eigenvector, indicated by a curved arrow.
- PERTURBED GRADIENT FOR CURRENT PATTERN**: The term  $\frac{\partial E^p}{\partial \omega} (\omega + \alpha \frac{\Psi}{\|\Psi\|})$ .
- GRADIENT FOR CURRENT PATTERN**: The term  $\frac{\partial E^p}{\partial \omega} (\omega)$ .

This procedure converges **VERY** quickly to the largest eigenvalue of the **AVERAGE** Hessian.

The properties of the average Hessian determine the behavior of ON-LINE gradient descent (stochastic, or per-sample update).

**EXPERIMENT:** A shared-weight network with 5 layers of weights, 64638 connections and 1278 free parameters. Training set: 1000 handwritten digits.

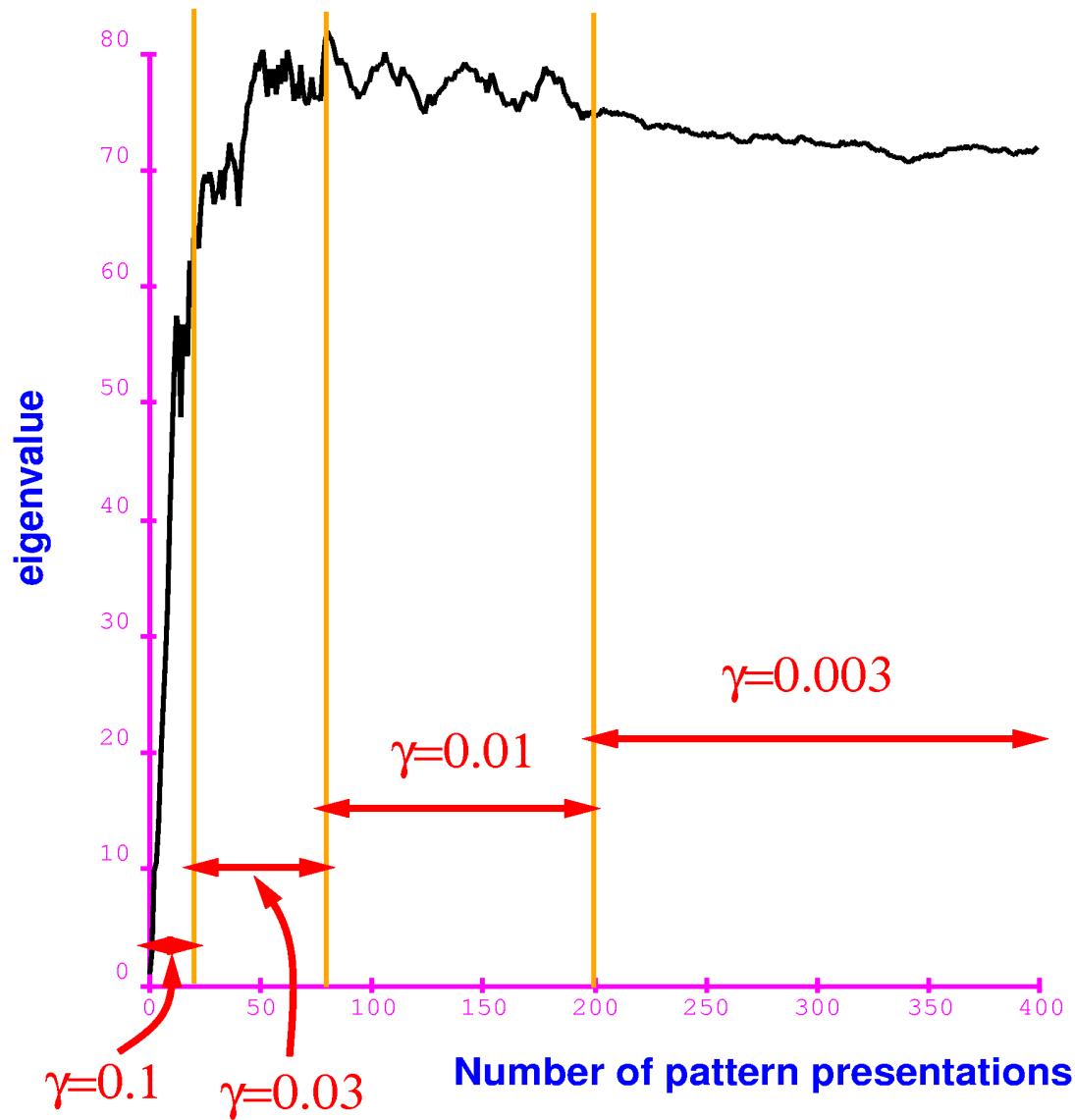
Correct order of magnitude is obtained in less than 100 pattern presentations (10% of training set size)

The fluctuations of the average Hessian over the training set are small.

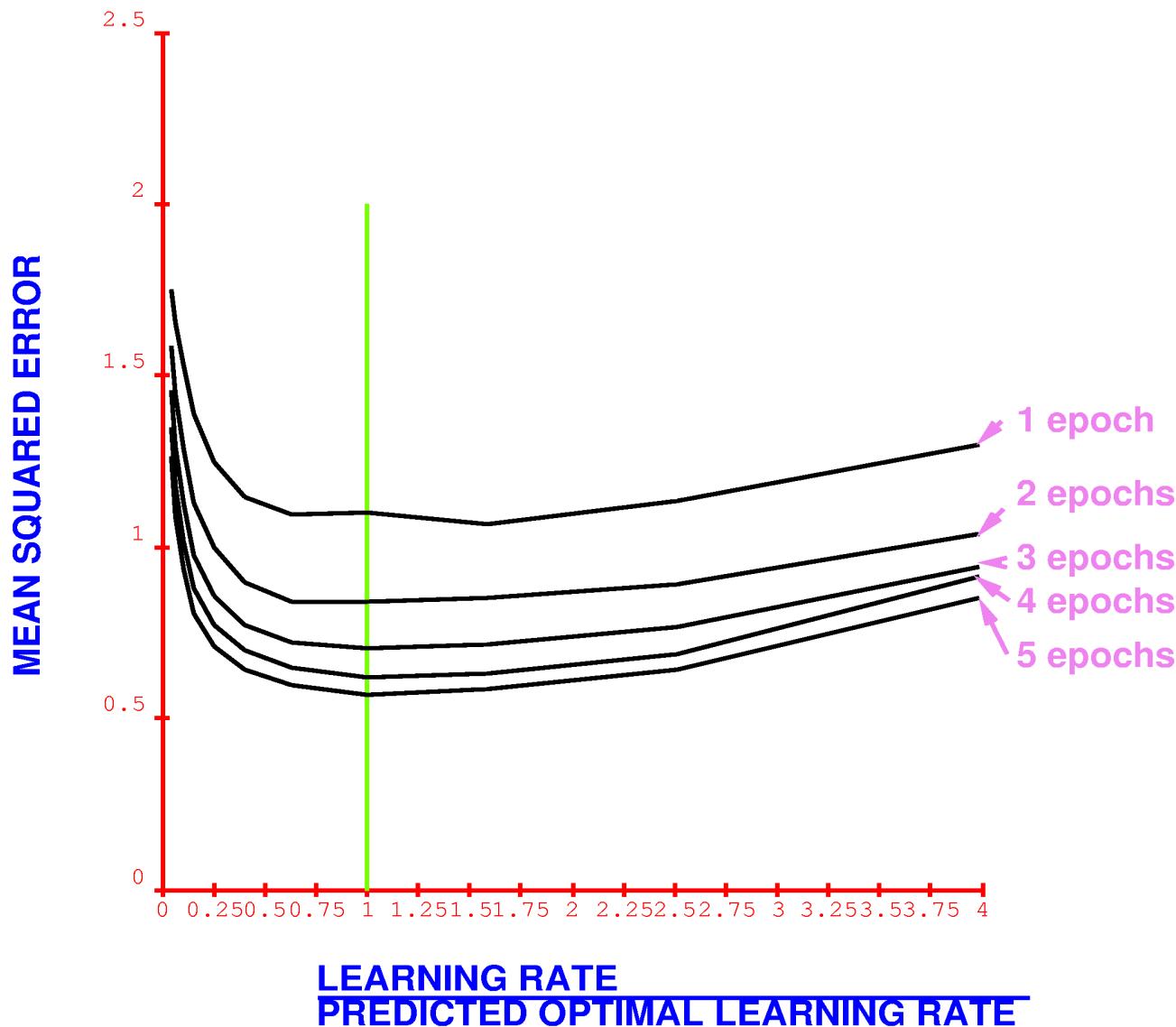
## RECIPE

$$\Psi \leftarrow (1-\gamma)\Psi + \gamma \frac{1}{\alpha} \left( \frac{\partial E^p}{\partial \omega} (\omega + \alpha \frac{\Psi}{\|\Psi\|}) - \frac{\partial E^p}{\partial \omega} (\omega) \right)$$

- 1 – Pick initial eigenvector estimate at random
- 2 – present input pattern, and desired output.  
perform forward prop and backward prop.  
Save gradient vector  $G(w)$
- 3 – add  $\alpha \frac{\Psi}{\|\Psi\|}$  to current weight vector
- 4 – perform forward prop and backward prop with  
perturbed weight vector. Save gradient vector  $G'(w)$
- 5 – compute difference  $G'(w) - G(w)$ , and divide by  $\alpha$   
update running average of eigenvector  
with the result
- 6 – goto 2 unless a reasonably stable result is obtained
- 7 – the optimal learning rate is  $\frac{1}{\|\Psi\|}$

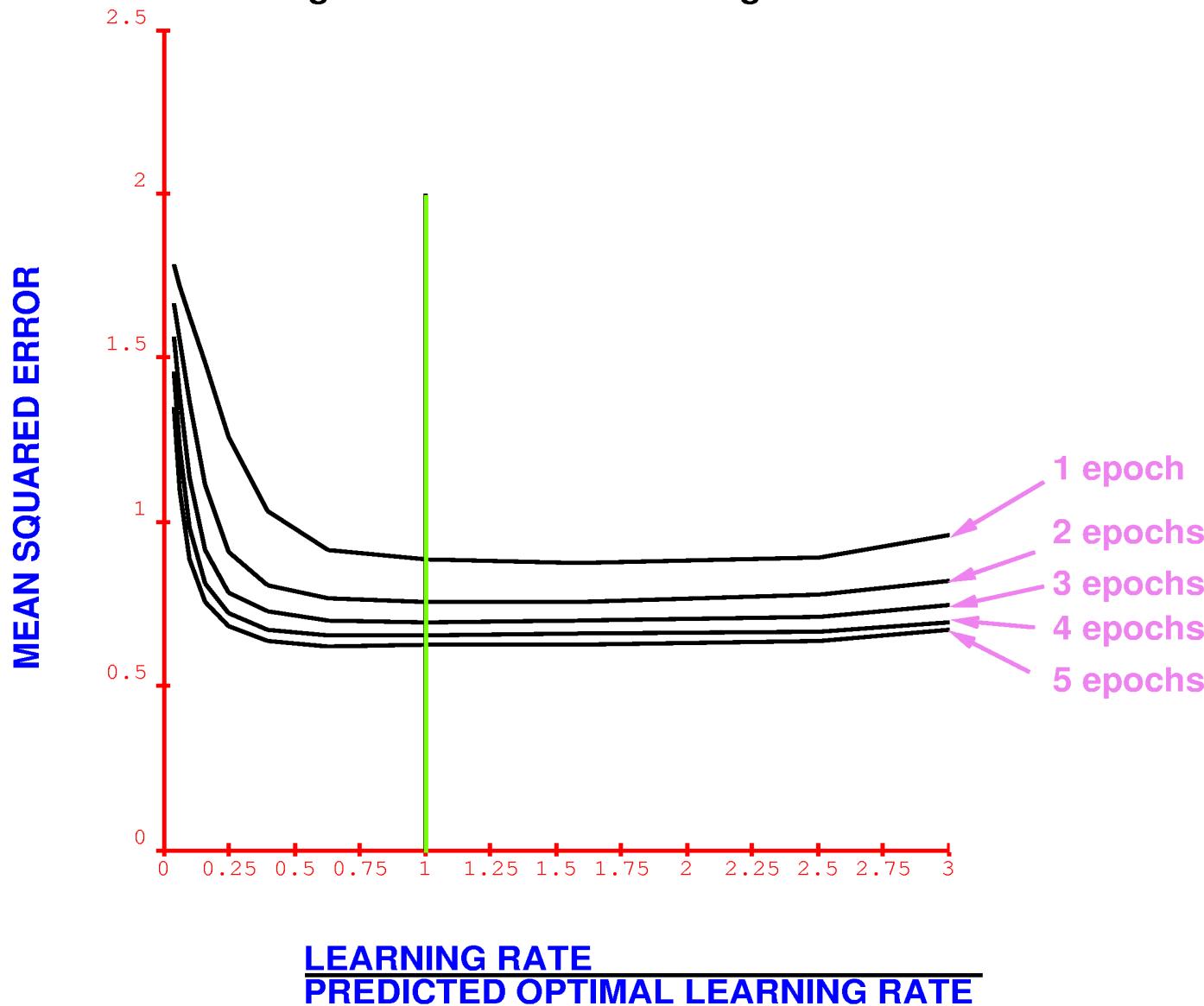


**Network: 784x30x10 fully connected**  
**Training set: 300 handwritten digits**



**LEARNING RATE**  
**PREDICTED OPTIMAL LEARNING RATE**

**Network: 1024x1568x392x400x100x10**  
**with 64638 (local) connections**  
**and 1278 shared weights**  
**Training set: 1000 handwritten digits**



**LEARNING RATE**  
**PREDICTED OPTIMAL LEARNING RATE**



The End