Optimization Methods for Non-Linear/Non-Convex Learning Problems

Yann LeCun
Courant Institute
http://yann.lecun.com

Non-Convex Optimization in Machine Learning

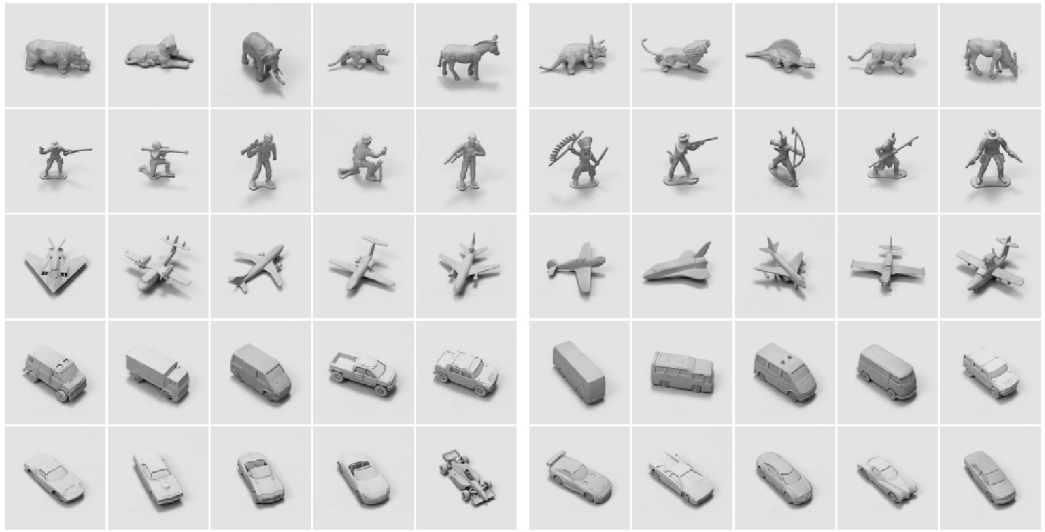
- Generalized linear models have (generally) convex loss functions
- SVMs (including non-linear ones) have convex loss functions, but have inequality constraints that make the problem difficult
  - What samples are support vectors?
- Models with non-convex loss functions:
  - Multi-layer neural nets
  - Discriminative training of mixture models
  - Any model with "products" of parameters
  - Any models with complex non-linearities after parameters
- Non convexity is scary to some, but there are vastly different types of non convexity (some of which are really scary!)

Convexity is Overrated

- 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
- 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

- Linear Classifier on raw stereo images: 30.2% error.
- K-Nearest-Neighbors on raw stereo images: 18.4% error.
- K-Nearest-Neighbors on PCA-95: 16.6% error.
- Pairwise SVM on 96x96 stereo images: 11.6% error
- Pairwise SVM on 95 Principal Components: 13.3% error.
- 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	σ=2,000 C=40					dim=80 σ=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

- 291,600 training samples, 58,320 test samples
- 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	σ=10^4 C=40				dim=100 σ=5 C=1

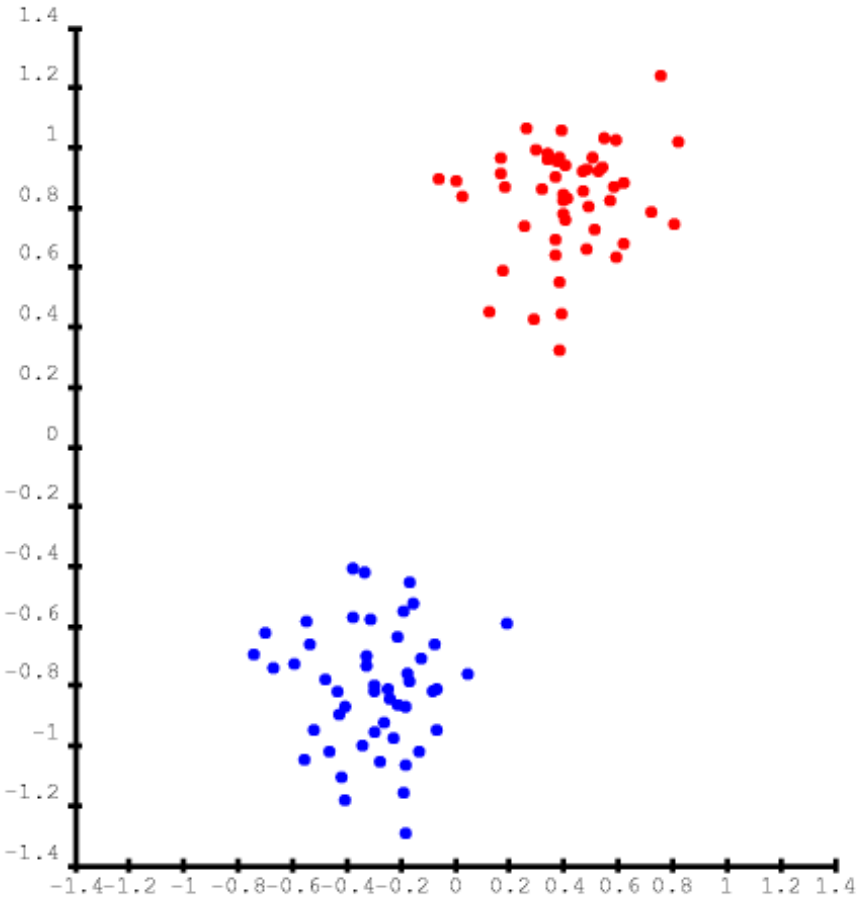
OUCH!

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

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

Example

- Simple 2-class classification. Classes are drawn from Gaussians distributions
  - Class1: mean=[-0.4, -0.8]
  - Class2: mean=[0.4,0.8]
  - Unit covariance matrix
  - 100 samples

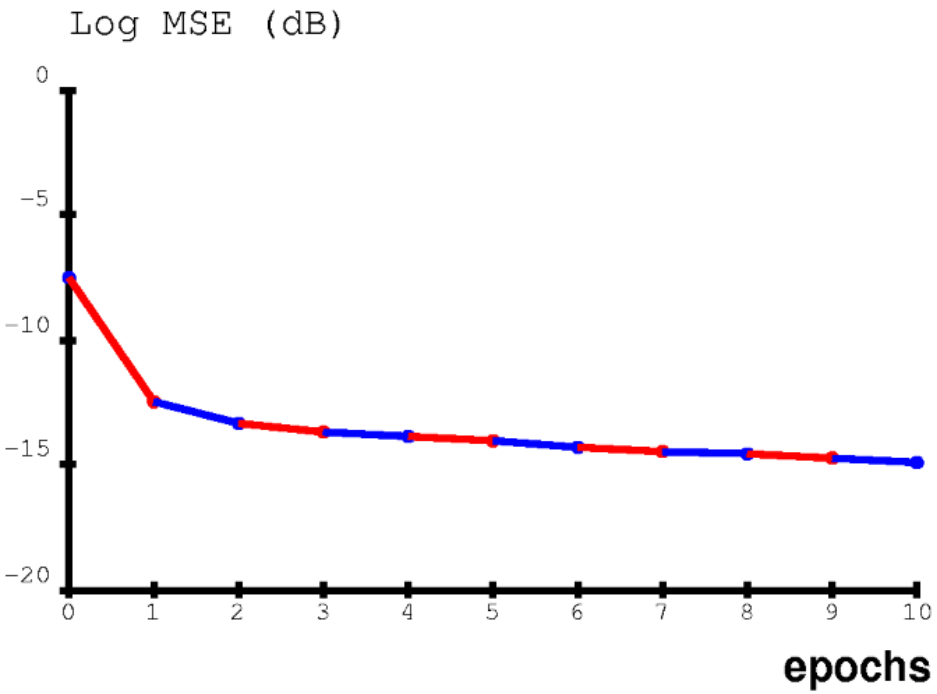


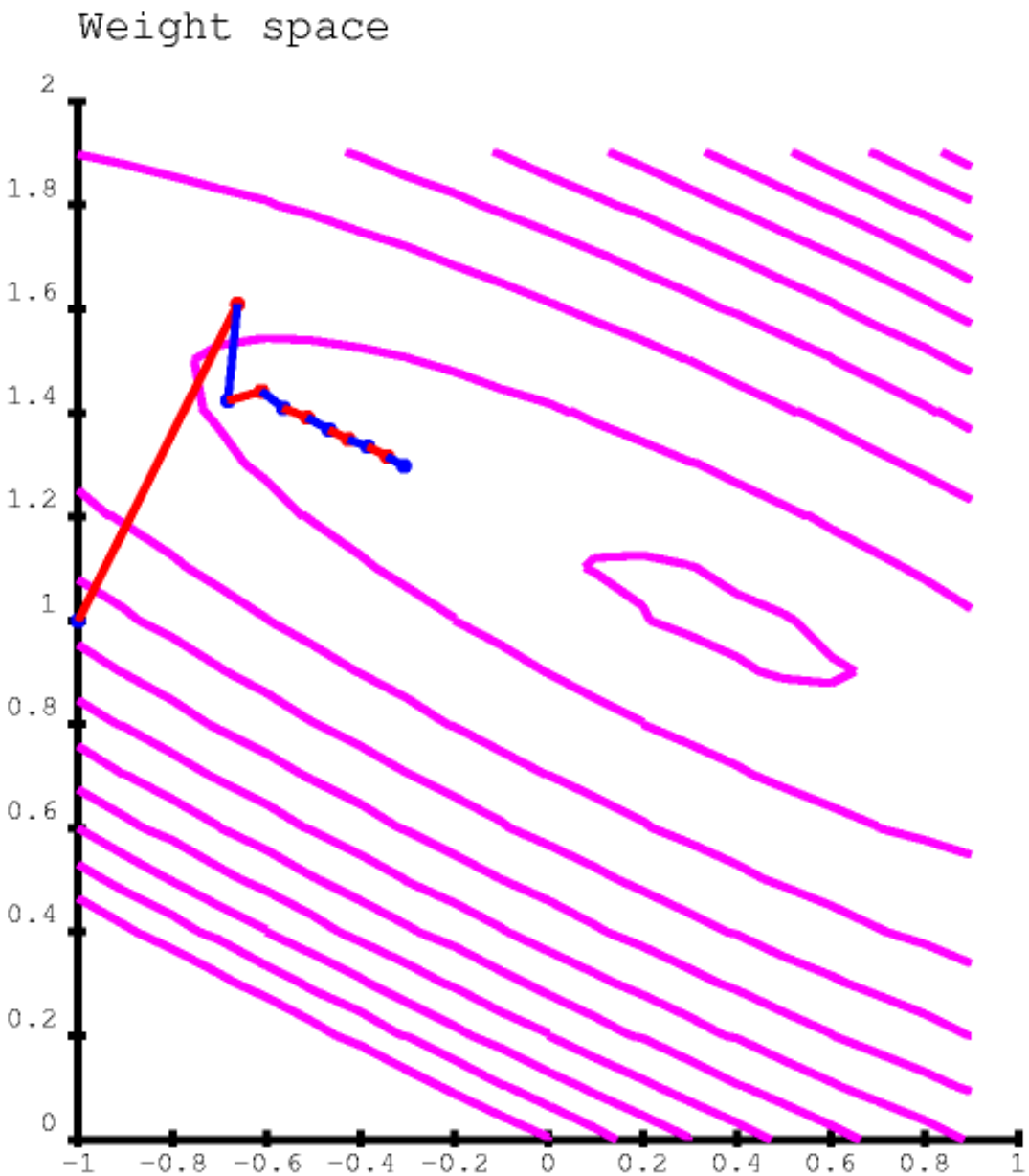
- Least square regression with y in {-1,+1}

$$L(W)=\frac{1}{P}\sum_{i=1}^{P}\frac{1}{2}\|y^i-W'X^i\|^2$$

Example

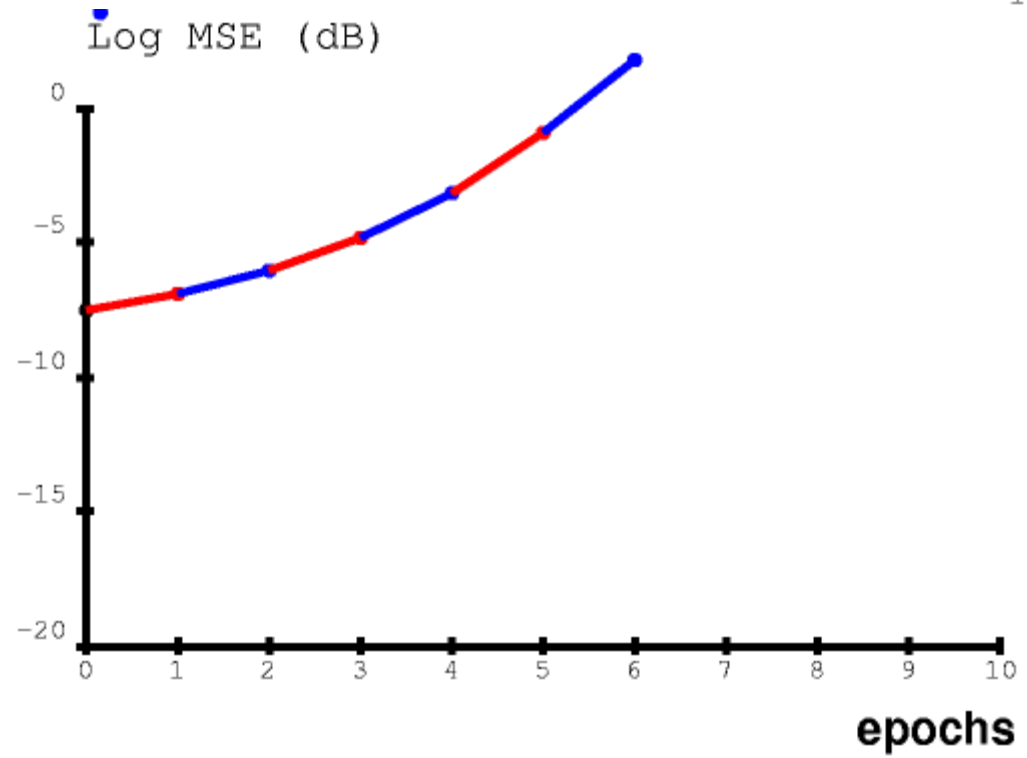
- Batch gradient descent
  - Learning rate = 1.5
  - Divergence for 2.38

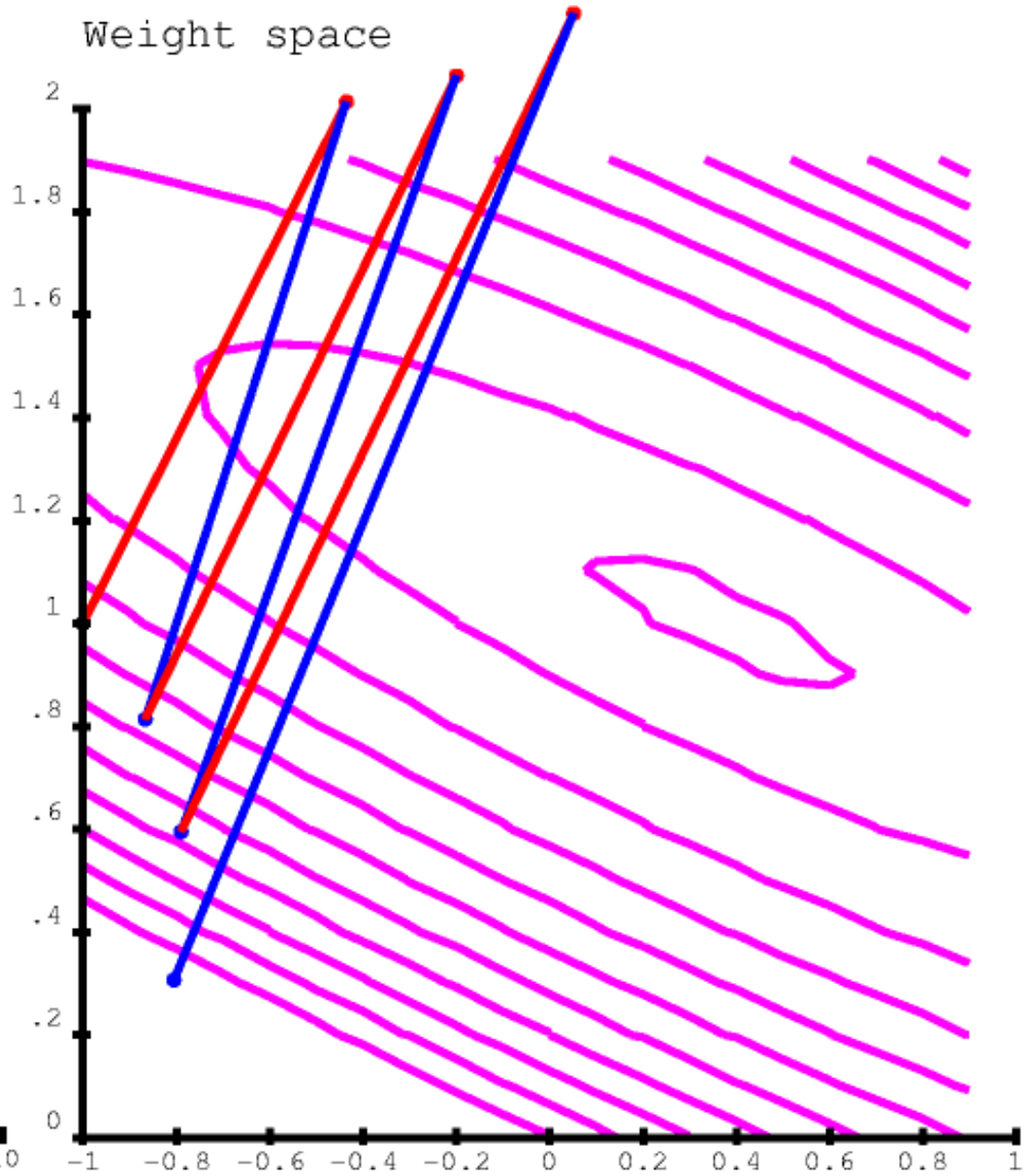




Example

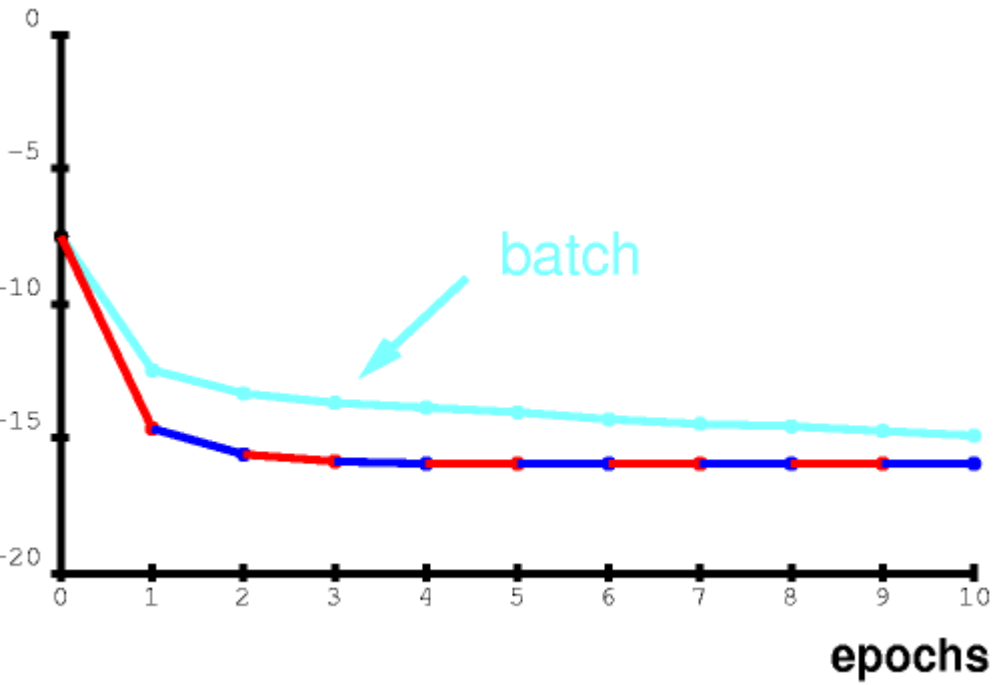
- Batch gradient descent
  - Learning rate = 2.5
  - Divergence for 2.38

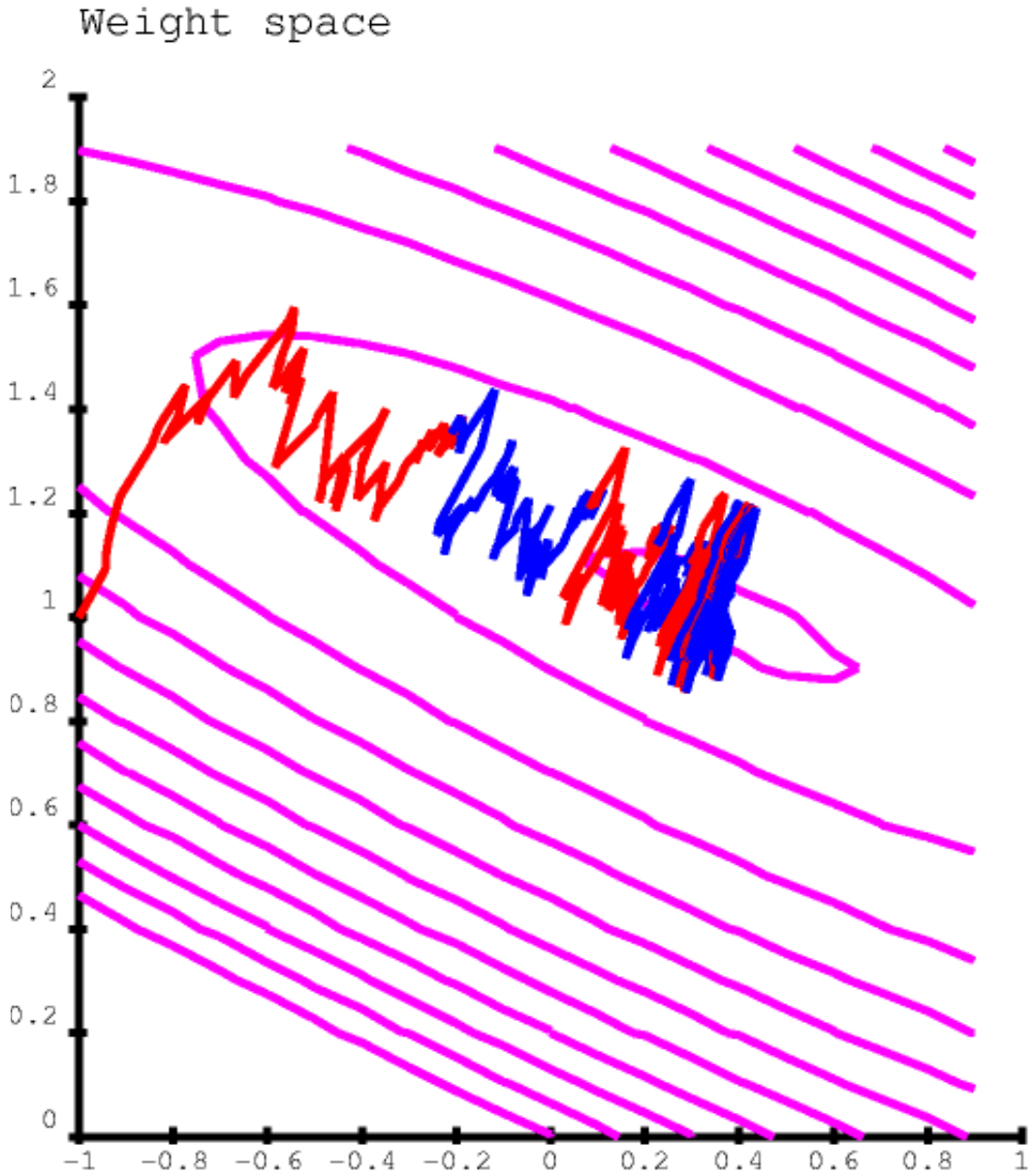




Example

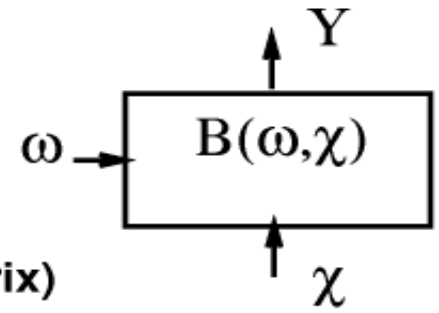
- Batch gradient descent
  - Learning rate = 0.2
  - Equivalent to batch learning rate of 20





Computing the Diagonal Hessian in Complicated Structures

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

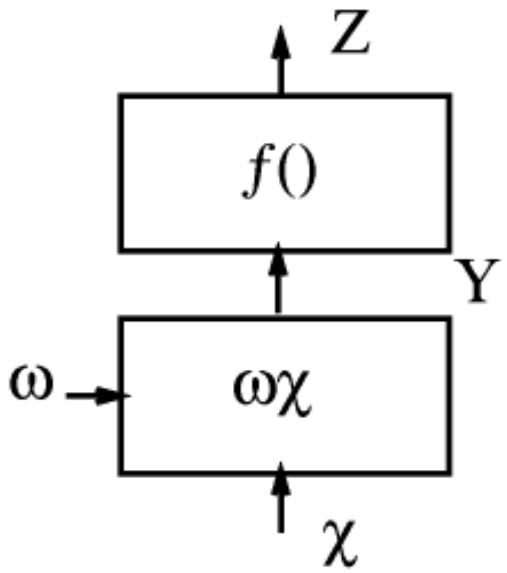
If we are only interested in the diagonal terms, it reduces to:

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

Computing the Diagonal Hessian in Complicated Structures

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

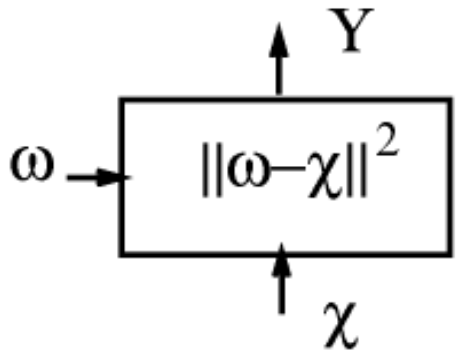
$$\frac{\partial^2 E}{\partial \omega_{ki}^2} = \frac{\partial^2 E}{\partial Y_k^2} \chi_i^2$$

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

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

Stochastic Diagonal Levenberg-Marquardt

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) ω_{ki} has its own learning rate η_{ki} computed as:

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

ε 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)

μ is a "Levenberg–Marquardt" parameter to prevent η_{ki} form blowing up if the 2nd derivative is small

Stochastic Diagonal Levenberg-Marquardt

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"

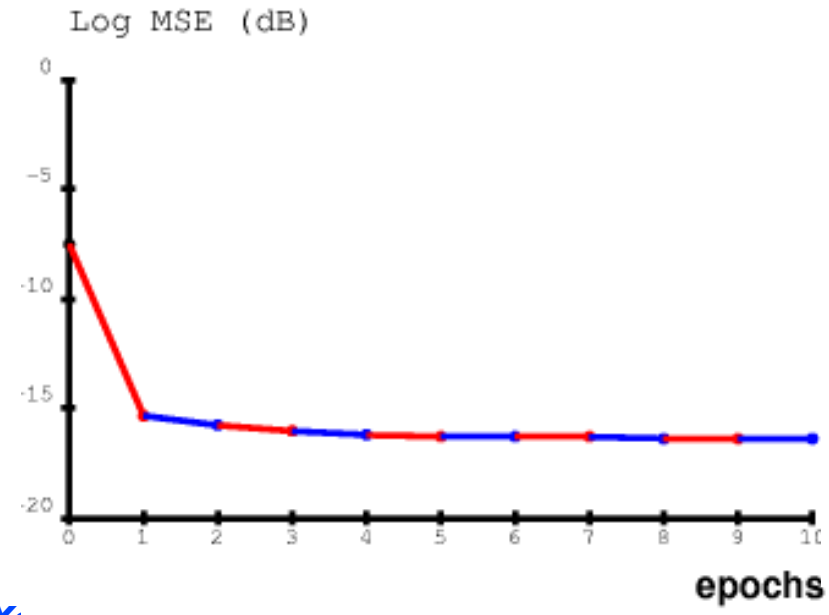
Since the second derivatives evolve slowly, there is no need to reestimate them often.
They can be estimated once at the beginning by sweeping over a few hundred patterns.
Then, they can be reestimated every few epochs.

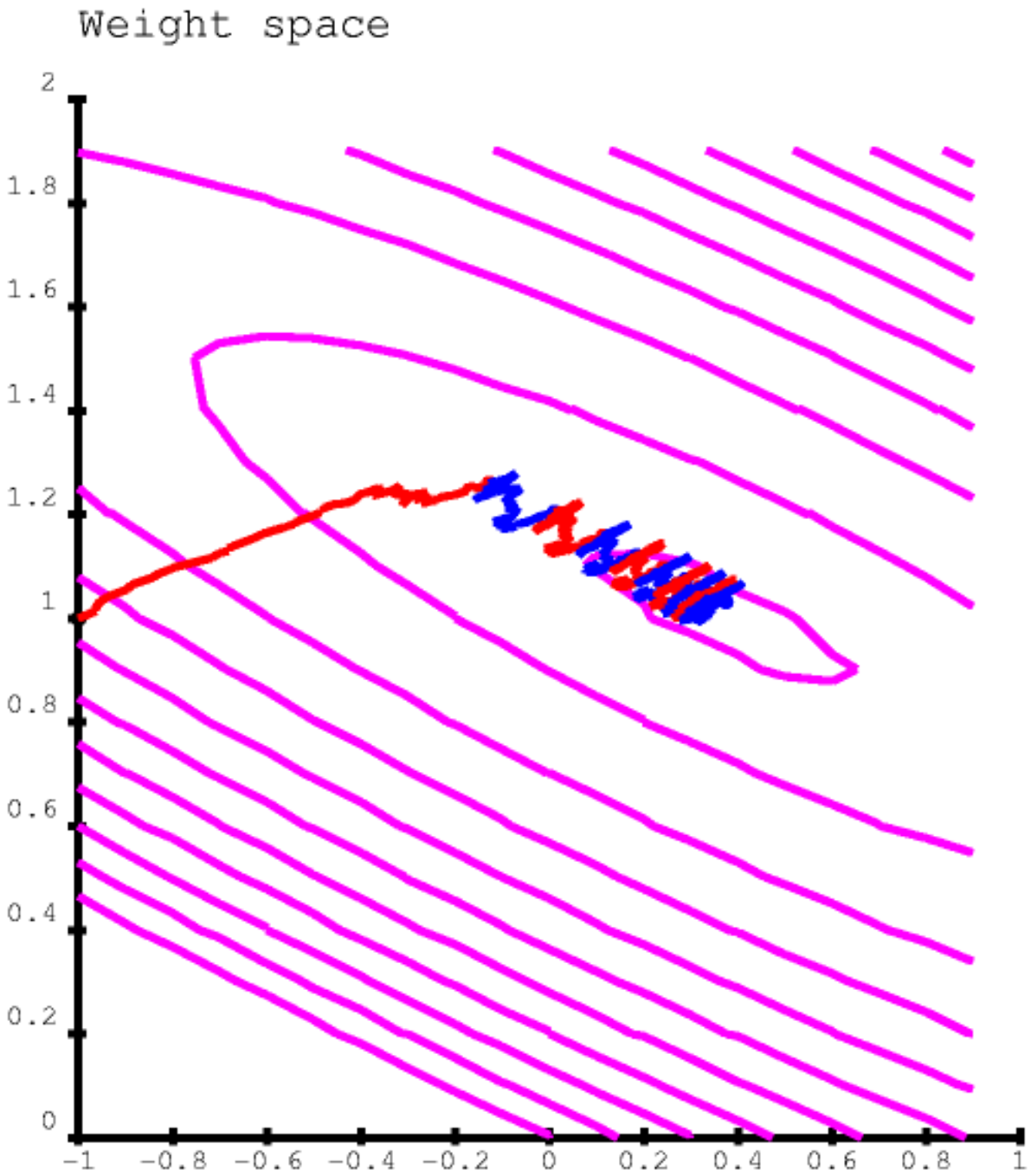
The additional cost over regular backprop is negligible.

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

Example

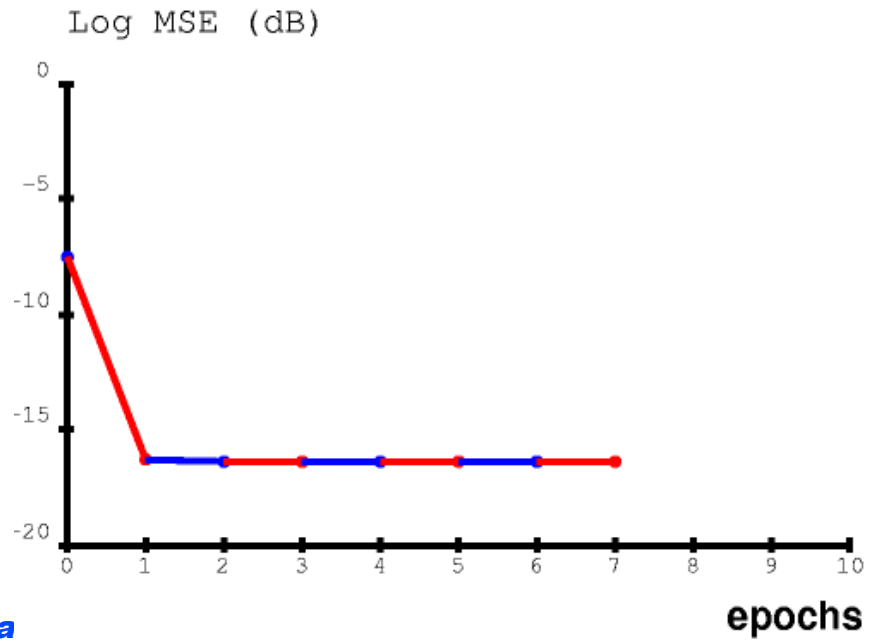
- Stochastic Diagonal Levenberg-Marquardt
  - Learning rates:
  - Eta0 = 0.12
  - Eta1 = 0.03
  - Eta2 = 0.02
  - Lambda_max = 0.84
  - Max batch learning rate: Etamax = 2.38

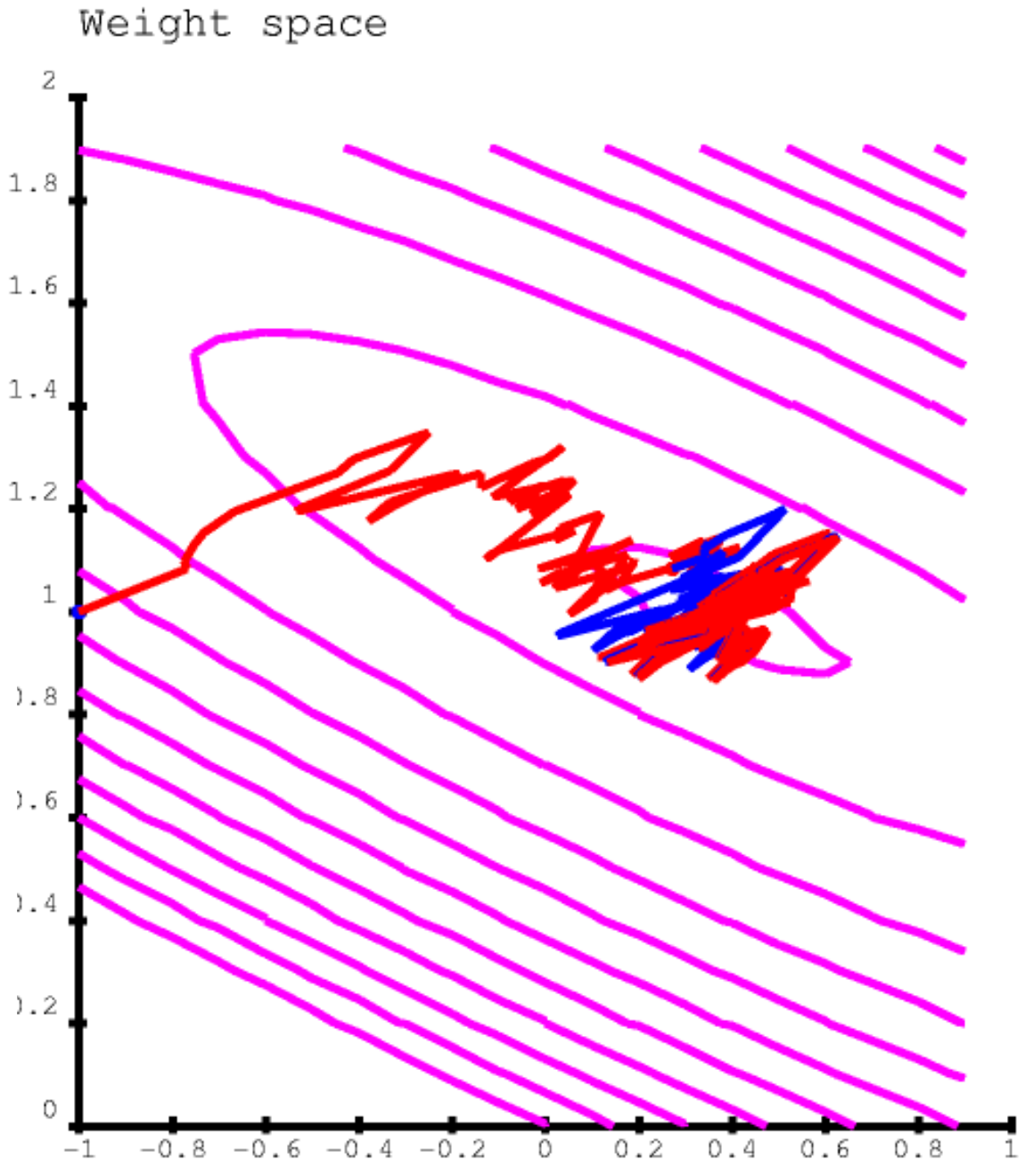




Example

- Stochastic Diagonal Levenberg-Marquardt
  - Learning rates:
  - Eta0 = 0.76
  - Eta1 = 0.18
  - Eta2 = 0.12
  - Lambda_max = 0.84
  - Max batch learning rate: Etamax = 2.38





Newton's Algorithm as a warping of the space

- Newton's algorithm is like GD in a warped space
- Precomputed warp = preconditioning

$$H = \Theta' \Lambda^{\frac{1}{2}} \Lambda^{\frac{1}{2}} \Theta$$

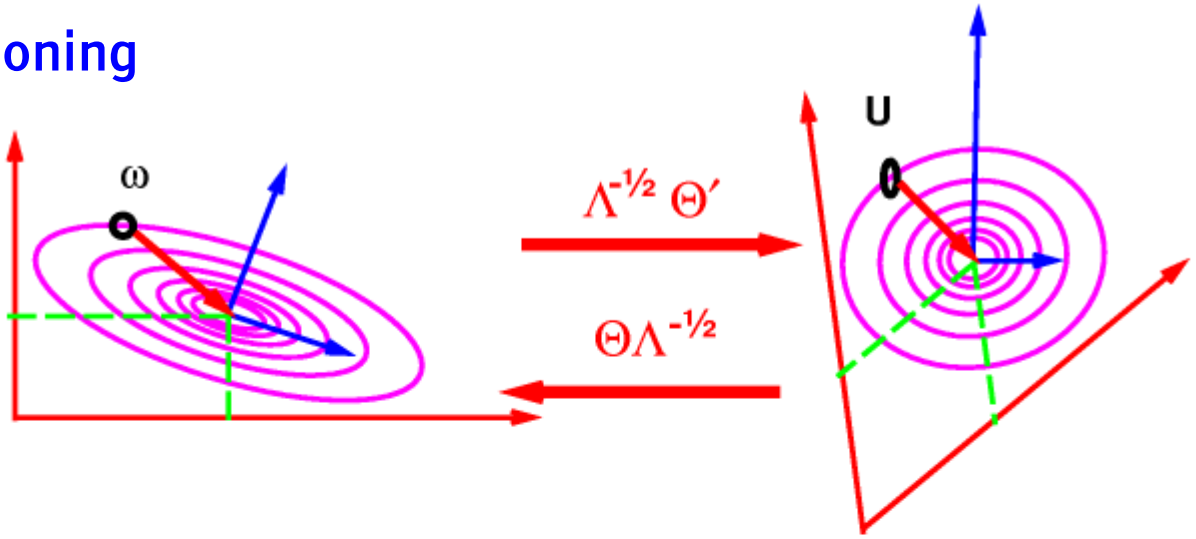
$$H^{-1} = \Theta \Lambda^{-\frac{1}{2}} \Lambda^{-\frac{1}{2}} \Theta'$$

$$W = \Theta \Lambda^{-\frac{1}{2}} U$$

$$\frac{\partial L}{\partial U} = \Lambda^{-\frac{1}{2}} \Theta' \frac{\partial L}{\partial W}$$

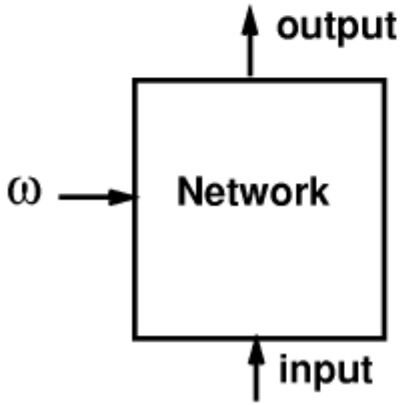
$$\delta U = H^{-\frac{1}{2}} \frac{\partial L}{\partial W}$$

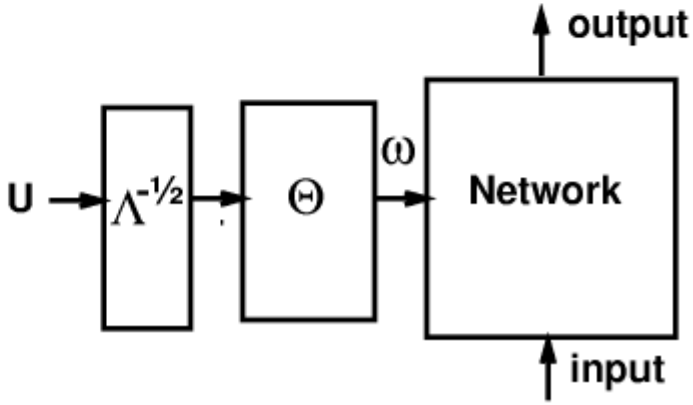
$$\delta W = H^{-1} \frac{\partial L}{\partial W}$$



Newton Algorithm here .....

....is like Gradient Descent there





Computing the Hessian by finite difference (rarely useful)

The k–th line of the Hessian is the derivative of the GRADIENT with respect to the k–th parameter

$$\text{(Line k of H)} = \frac{\partial \left(\nabla E(\omega) \right)}{\partial \omega_\kappa}$$

Finite difference approximation:

$$\text{(Line k of H)} = \frac{\nabla E(\omega + \delta \, \phi_\kappa) - \nabla E(\omega)}{\delta}$$

$$\phi_\kappa = (0,0,0,.....,1,....,0)$$

RECIPE for computing the k–th line of the Hessian

1– compute total gradient (multiple fprop/bprop)
2– add Delta to k–th parameter
3– compute total gradient
4– subtract result of line 1 from line 3, divide by Delta.

due to numerical errors, the resulting Hessian may not be perfectly symetric. It should be symetrized.

Gauss-Newton Approximation of the Hessian

Assume the cost function is the Mean Square Error:

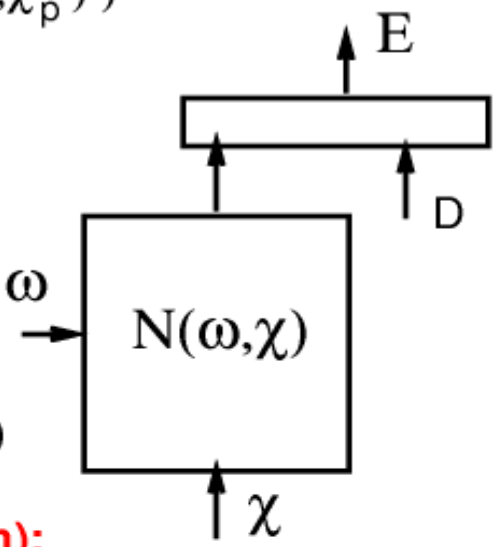
$$E(\omega) = 1/2 \sum_p (D_p - N(\omega,\chi_p))' (D_p - N(\omega,\chi_p))$$

Gradient:

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

Hessian:

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



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;
}
```

What does the Hessian look like in a neural net?

– 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

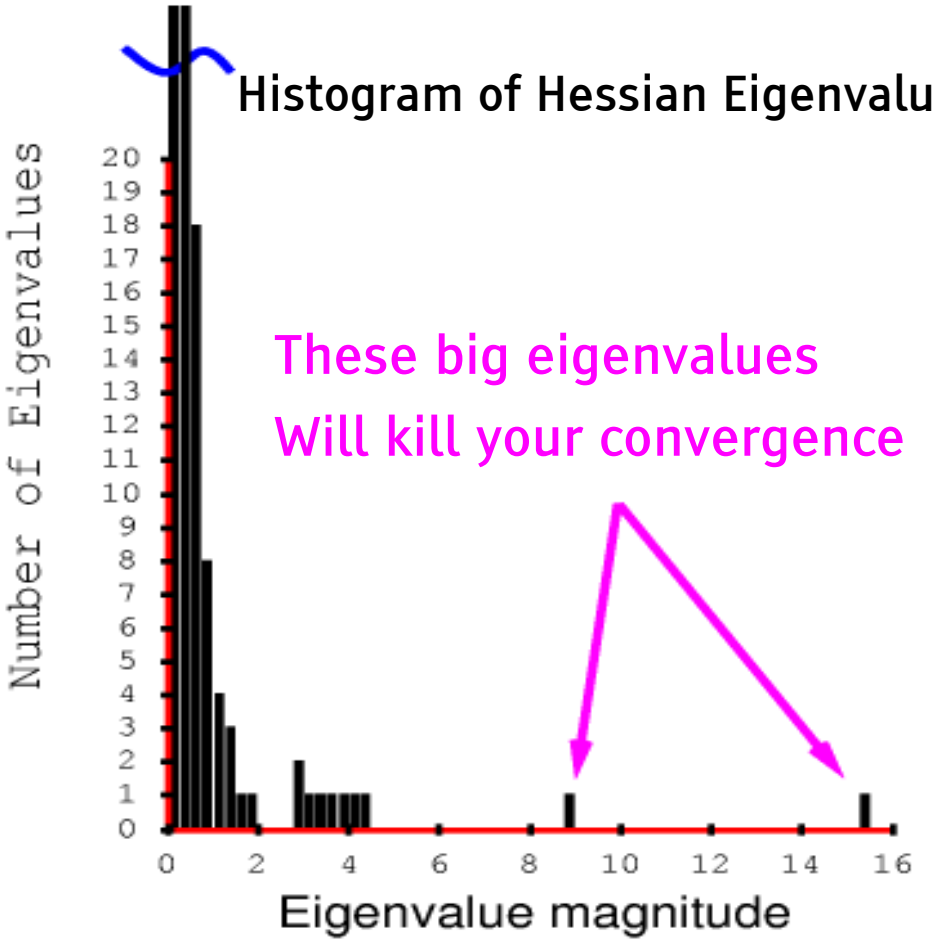
These large ones are the killers

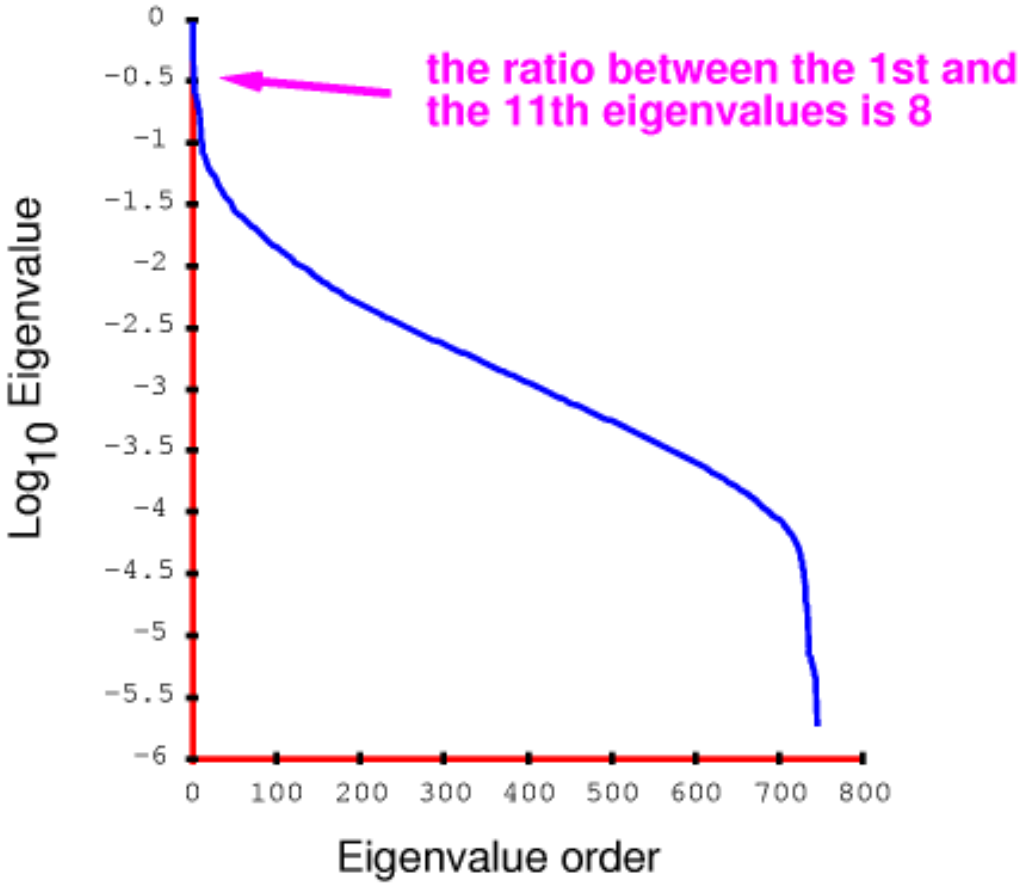
They come from:
– non–zero mean inputs or neuron states
– wide variations second derivatives from layer to layer
– correlations between state variables

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

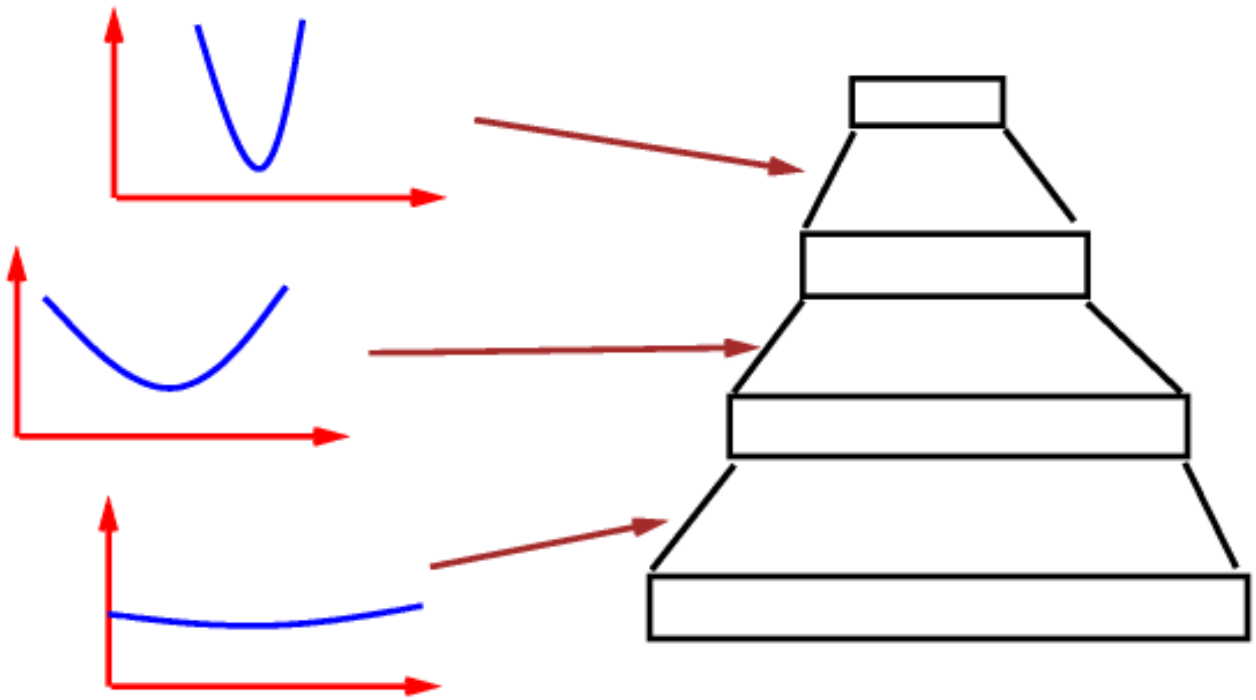
What does the Hessian look like in a neural net?

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





What does the Hessian look like in a neural net?



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

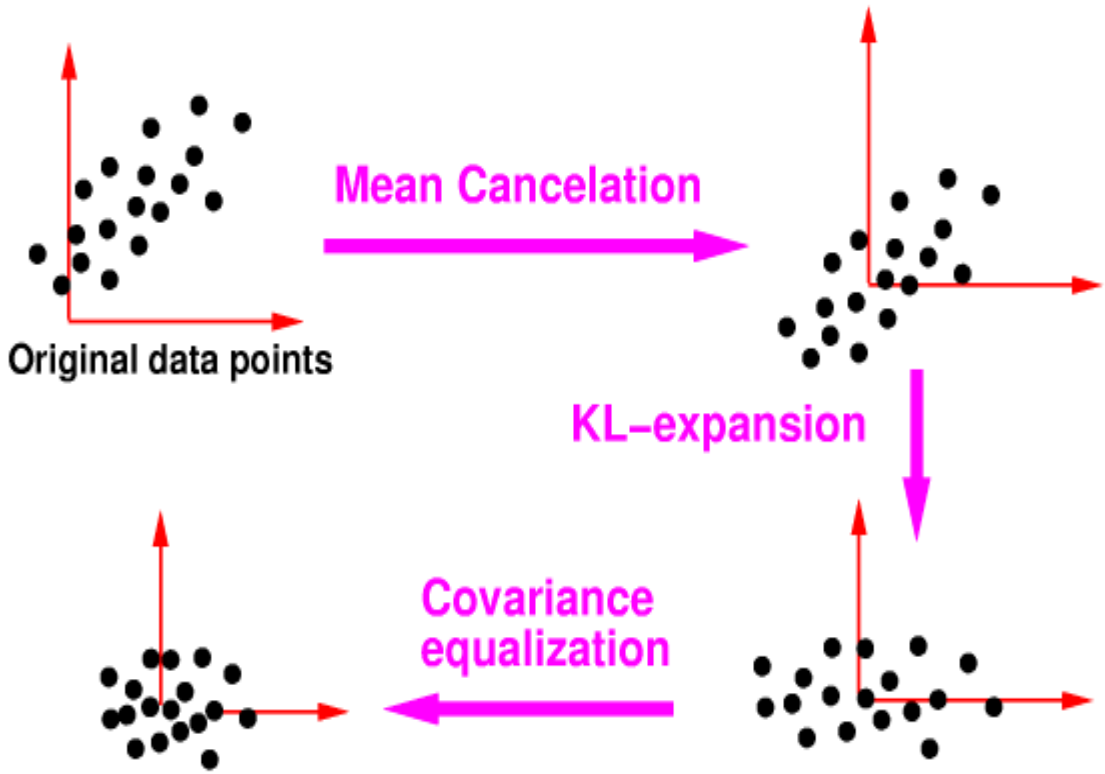
This can be compensated for using the diagonal 2nd derivatives (more on this later)

Making the Hessian better conditioned

- The Hessian is the covariance matrix of the inputs (in a linear system)

$$H = \frac{1}{2P} \sum_{i=1}^{P} X^i X^{i\prime}$$

- Non-zero mean inputs create large eigenvalues!
  - Center the input vars, unless it makes you lose sparsity (see Vowpal Wabbit)
- Decorrelate the input vars
  - Correlations leads to ill conditioning



- Decorrelation can be done with a "Karhunen-Loève transform (like PCA)
  - Rotate to the eigenspace of the covariance matrix
- Normalize the variances of the input variables
  - or use per-weight learning rates / diagonal Hessian

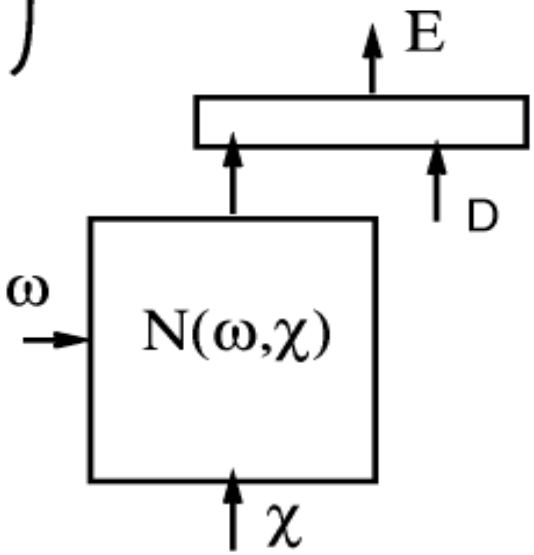
Computing the product of the Hessian by a Vector

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 Ψ 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 α



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

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

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 been proposed [Pearlmutter 93]

Using the Power Method to Compute the Largest Eigenvalue

1 – Choose a vector Ψ at random

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

OLD ESTIMATE OF EIGENVECTOR

NEW ESTIMATE OF EIGENVECTOR

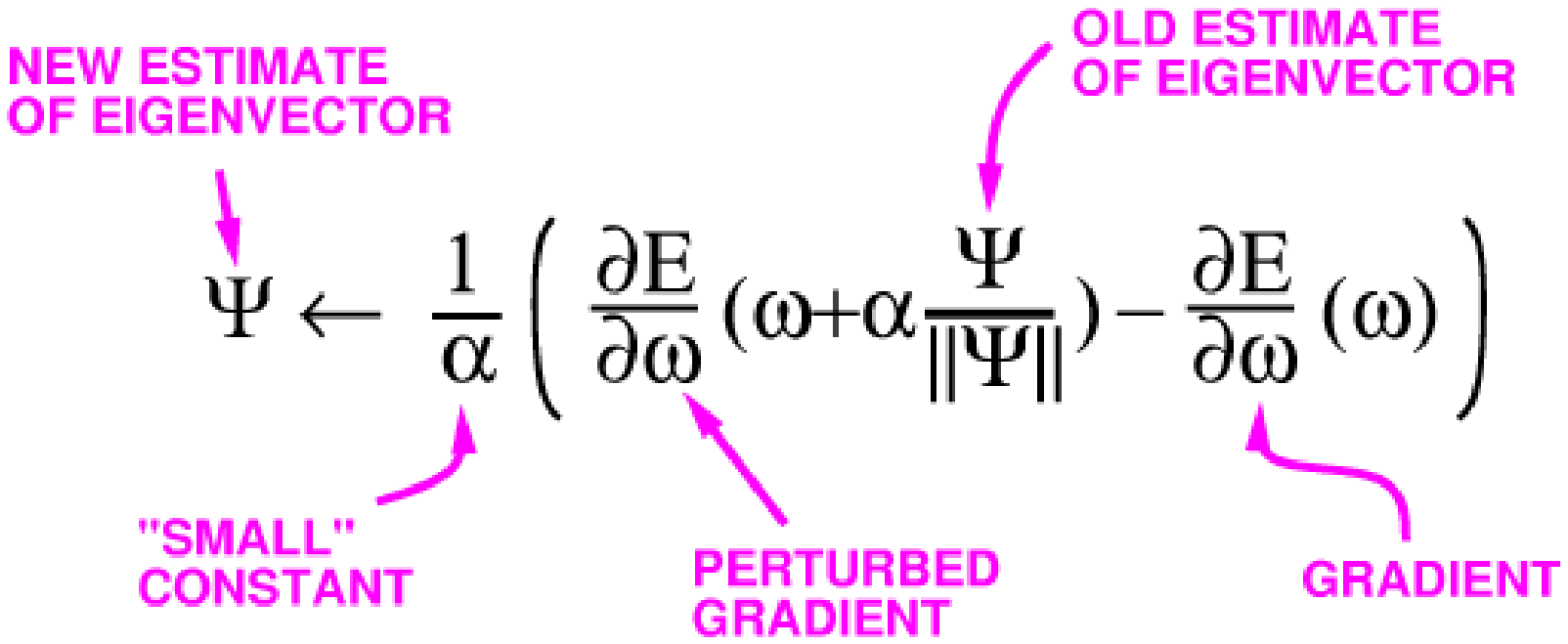
HESSIAN

ESTIMATE OF EIGENVALUE

Ψ will converge to the principal eigenvector (or a vector in the principal eigenspace)

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

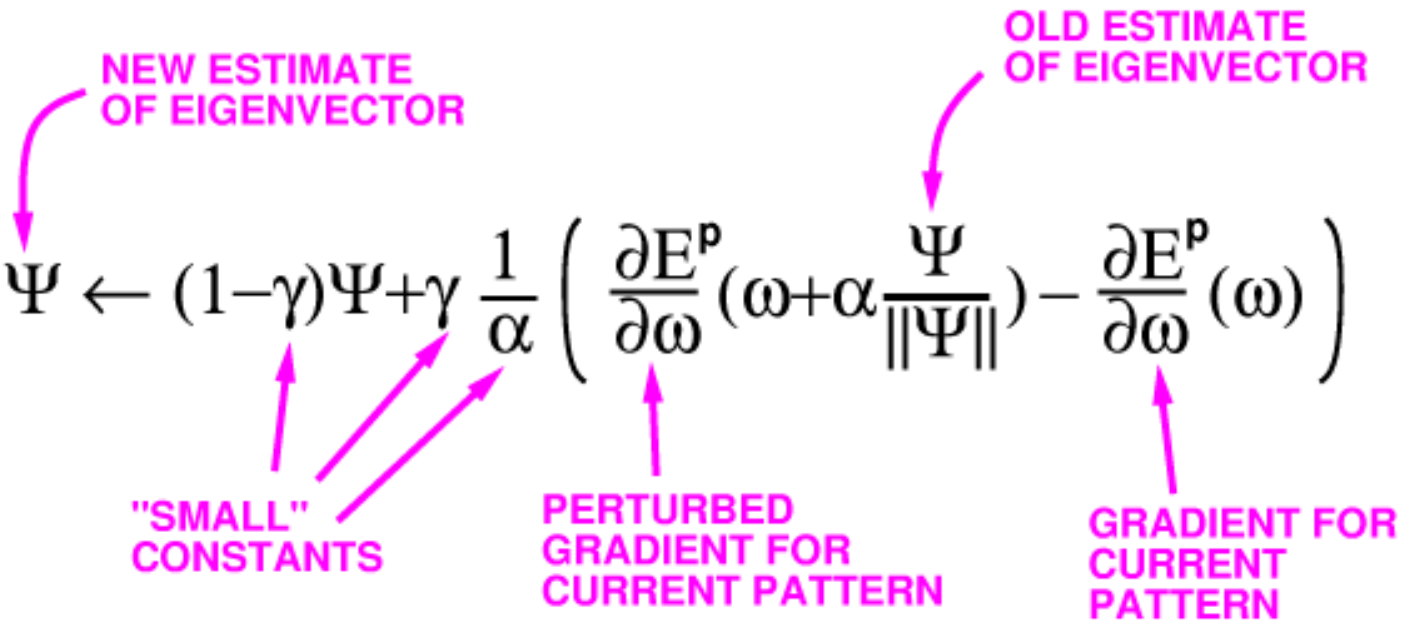
Using the Power Method to Compute the Largest Eigenvalue



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

Stochastic version of the principal eigenvalue computation



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 to compute the maximum learning rate for SGD

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

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

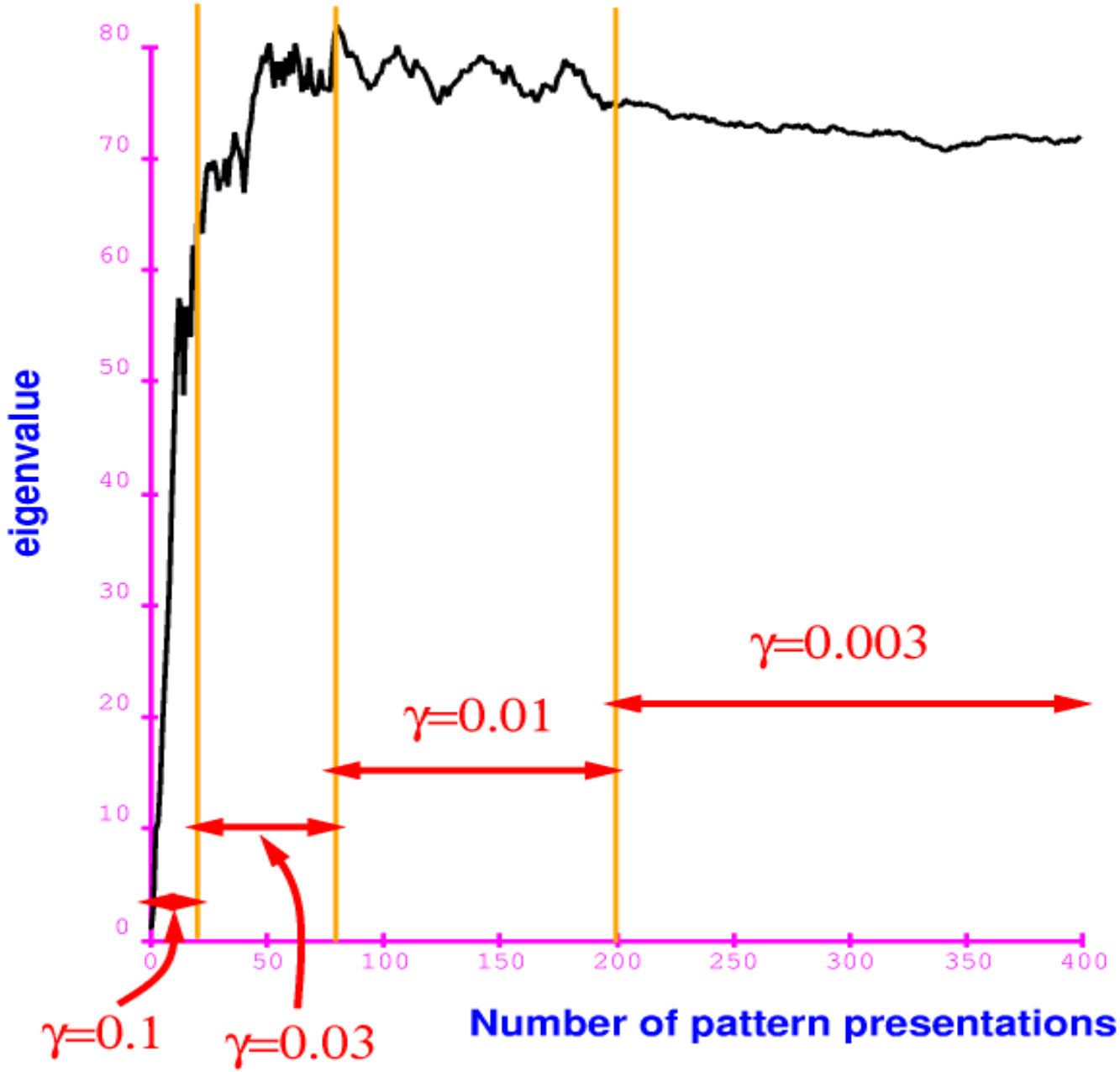
4 – perform forward prop and backward prop with
perturbed weight vector. Save gradient vector G'(w)

5 – compute difference G'(w)–G(w). and divide by α
update running average of eigenvector
with the result

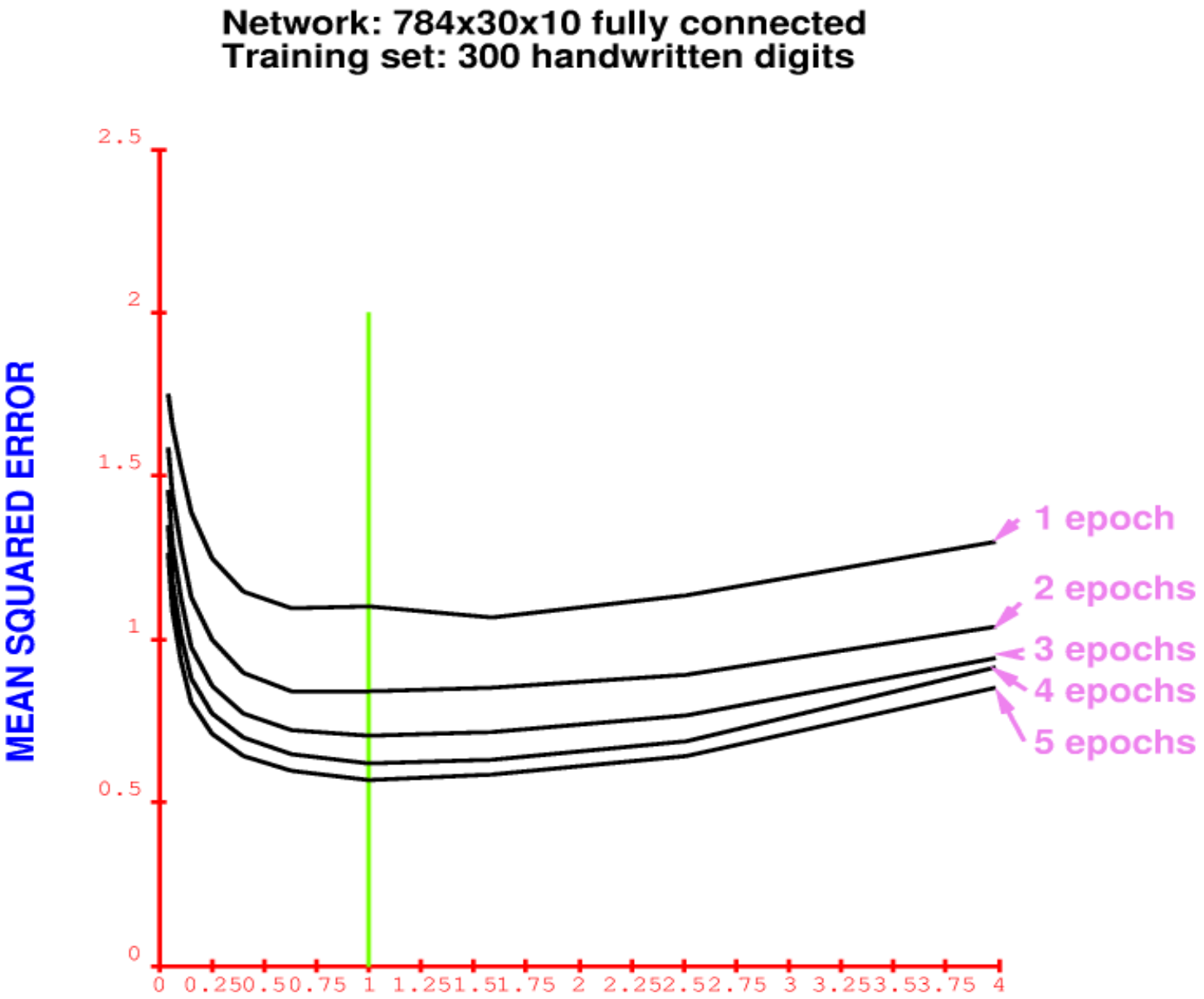
6 – goto 2 unless a reasonably stable result is obtained

7 – the optimal learning rate is $\frac{1}{\|\Psi\|}$

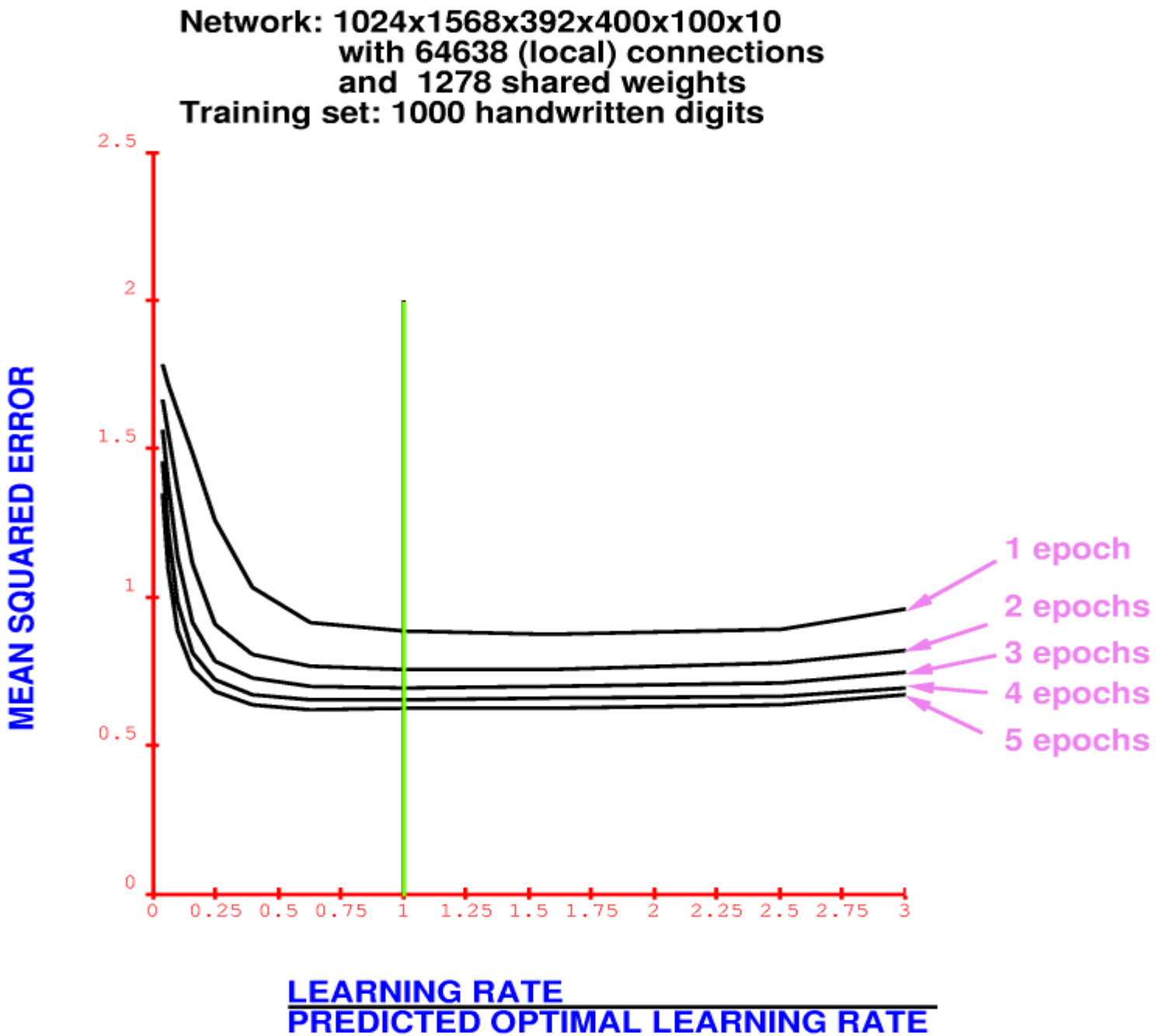
Recipe to compute the maximum learning rate for SGD



Recipe to compute the maximum learning rate for SGD



Recipe to compute the maximum learning rate for SGD



Tricks to Make Stochastic Gradient Work

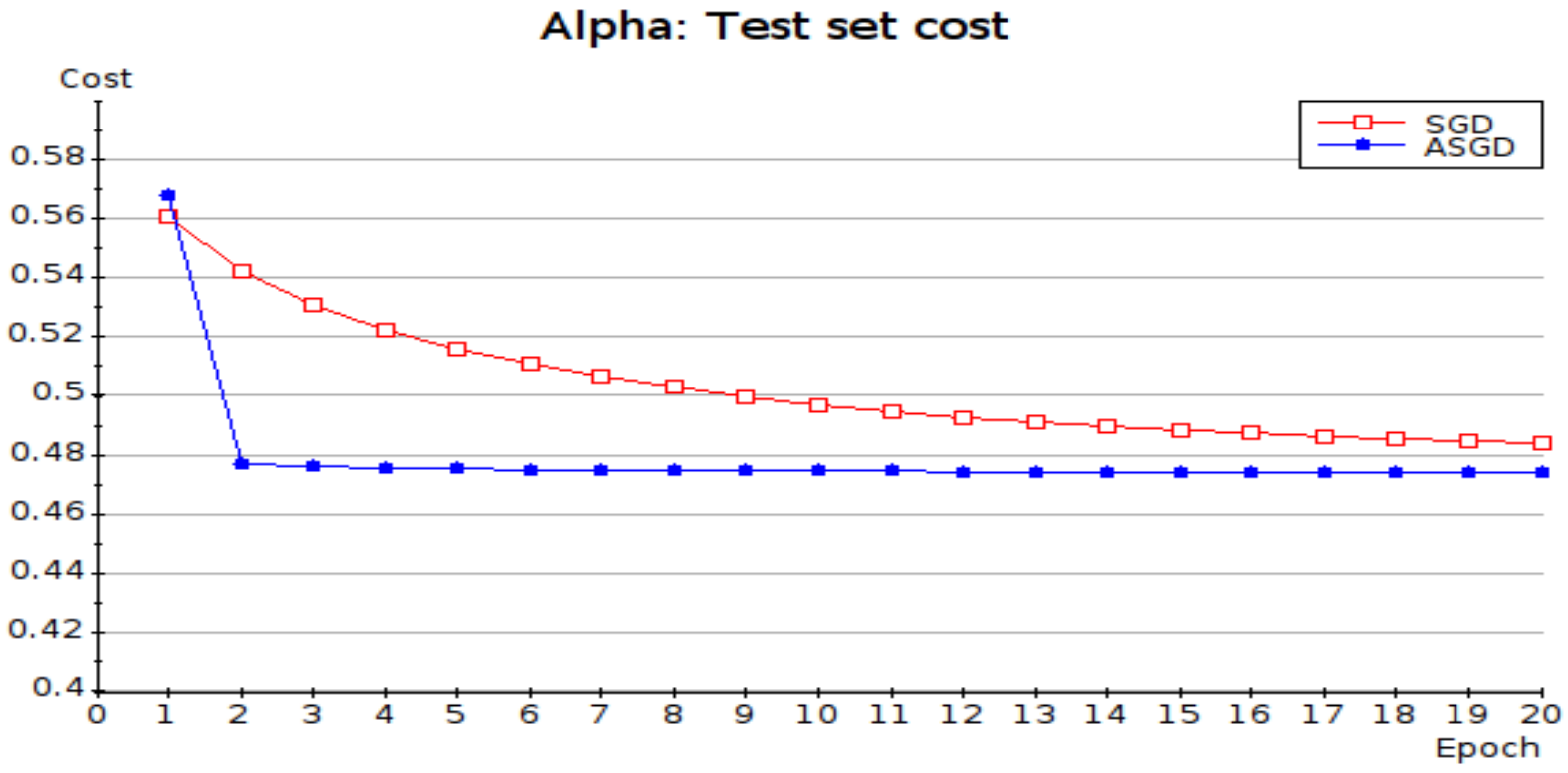
- **Best reads:**
  - Yann LeCun, Léon Bottou, Genevieve B. Orr and Klaus-Robert Müller: Efficient Backprop, Neural Networks, Tricks of the Trade, Lecture Notes in Computer Science LNCS 1524, Springer Verlag, 1998. http://yann.lecun.com/exdb/publis/index.html#lecun-98b
  - Léon Bottou: Stochastic Gradient Tricks, Neural Networks, Tricks of the Trade, Reloaded, 430–445, Edited by Grégoire Montavon, Genevieve B. Orr and Klaus-Robert Müller, Lecture Notes in Computer Science (LNCS 7700), Springer, 2012. http://leon.bottou.org/papers/bottou-tricks-2012
- **Tricks**
  - Average SGD: http://leon.bottou.org/projects/sgd
  - Normalization of inputs
  - All the tricks in Vowpal Wabbit

Learning Rate Schedule + Average SGD

- Learning rate schedule (after Wei Xu & Leon Bottou)

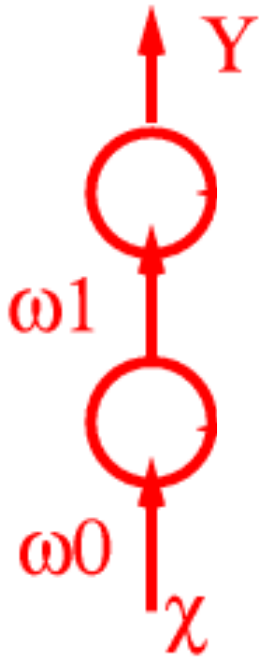
$$\eta_t = \eta_0 / \left(1 + \lambda \eta_0 t\right)^{0.75}$$

- Average SGD: time averaging (Polyak & Juditsky 1992)
  - Report the average of the last several weight vectors
  - e.g. by using a running average



the loss landscape for a 2-layer neural net

- The simplest 2-layer neural net:
  - 1 input, 1 output, 1 hidden unit
  - 2 weights, 2 biases
- Non-linearity: tanh(X)
- Targets: -0.5 for class 1, +0.5 for class 2
- Training set: 2 samples
  - 0.5 → 0.5
  - -0.5 → -0.5
- Task: "identity function".

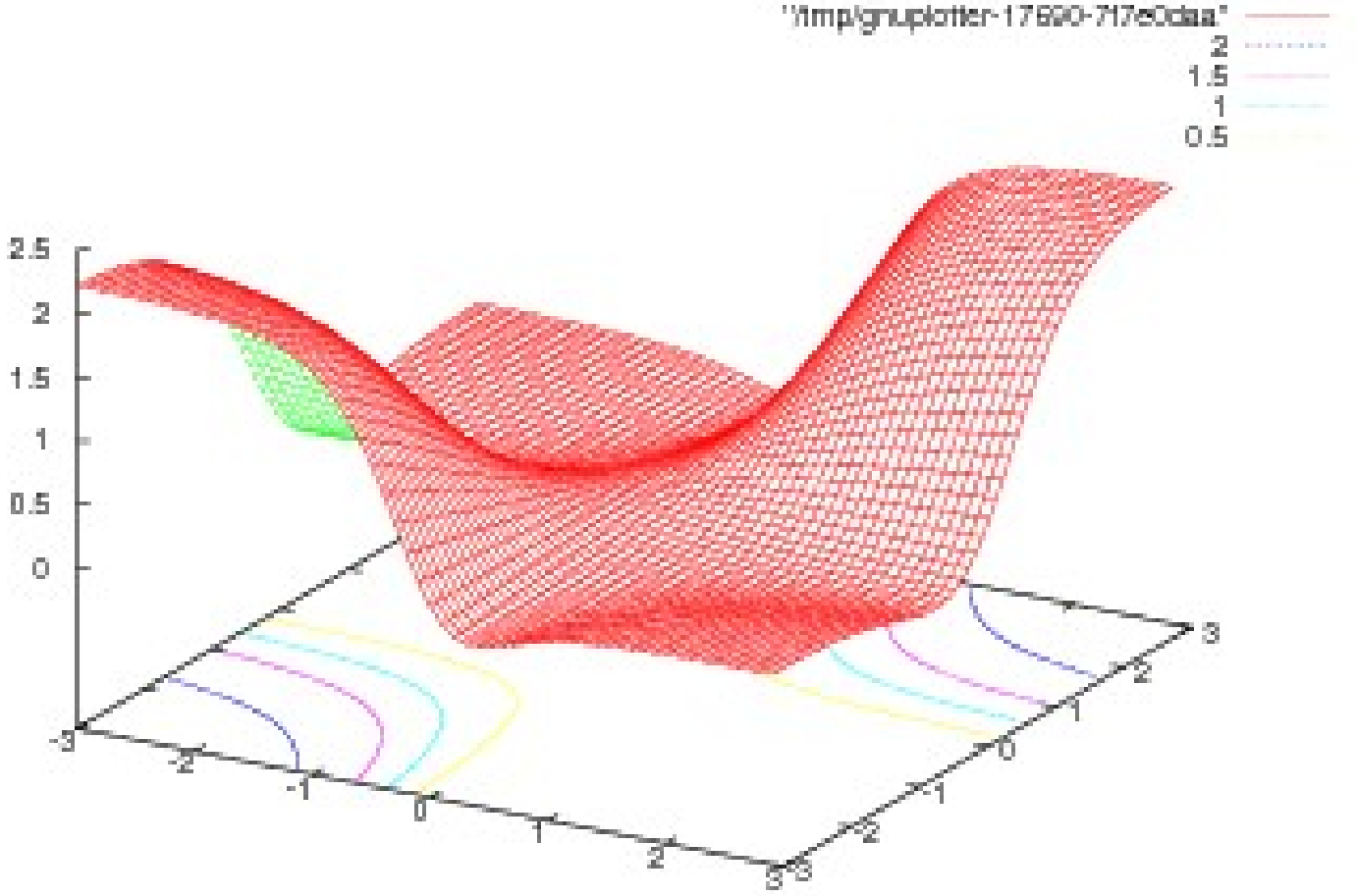


$$y = \tanh(W_1 \tanh(W_0.x)) \quad L = (0.5 - \tanh(W_1 \tanh(W_0 0.5)^2$$

Deep Learning is Hard?

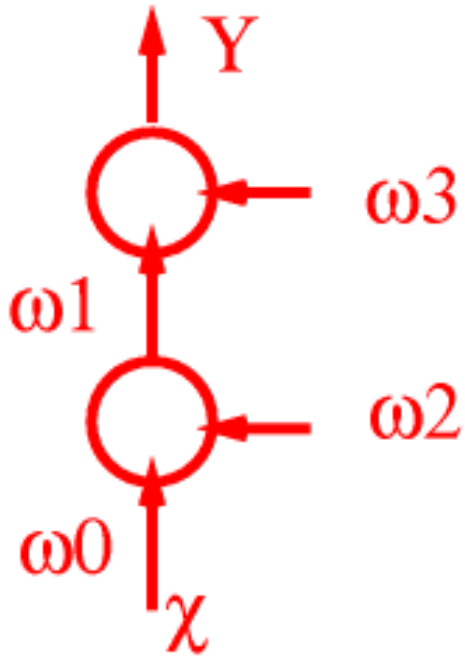
- **Example: what is the loss function for the simplest 2-layer neural net ever**
  - Function: 1-1-1 neural net. Map 0.5 to 0.5 and -0.5 to -0.5 (identity function) with quadratic cost:

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



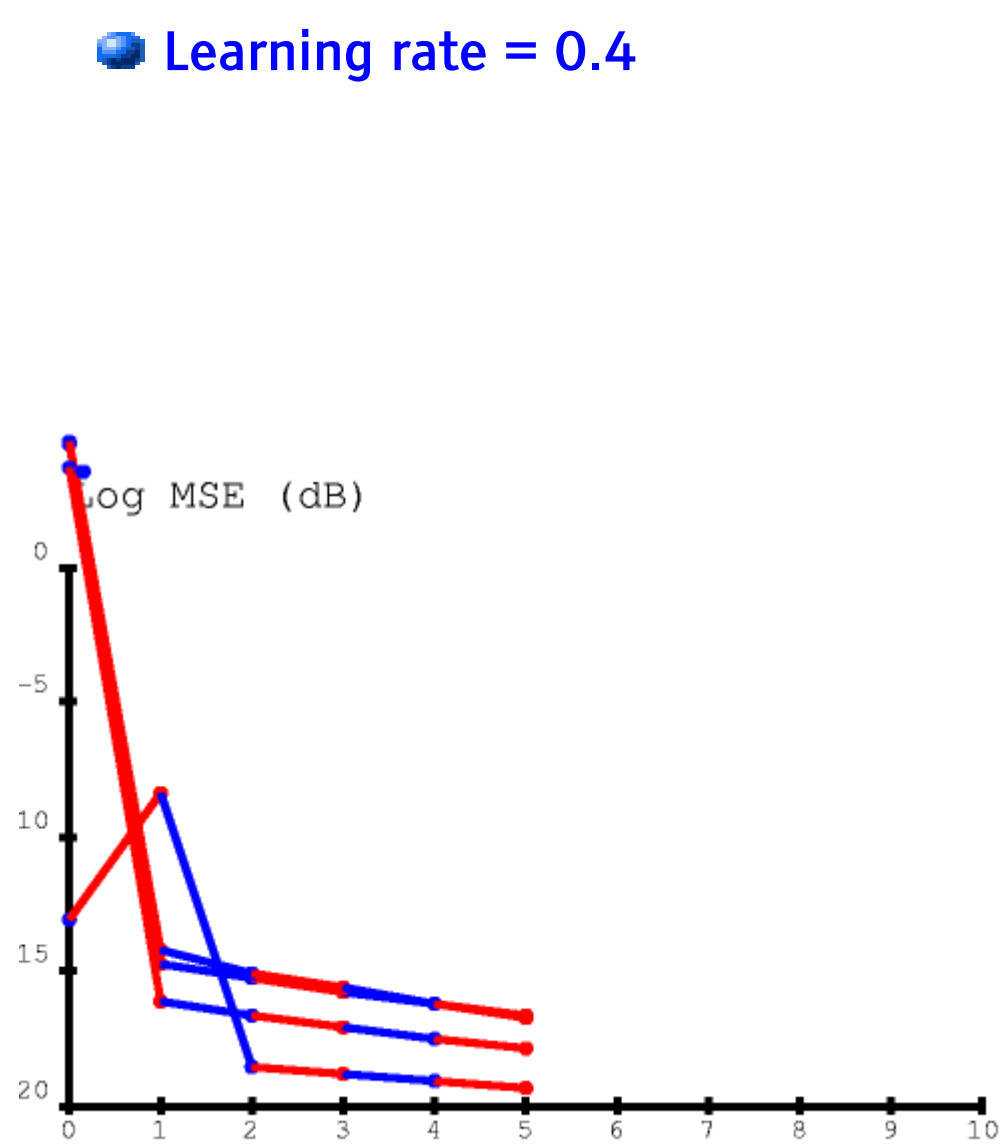
the loss landscape for a 2-layer neural net

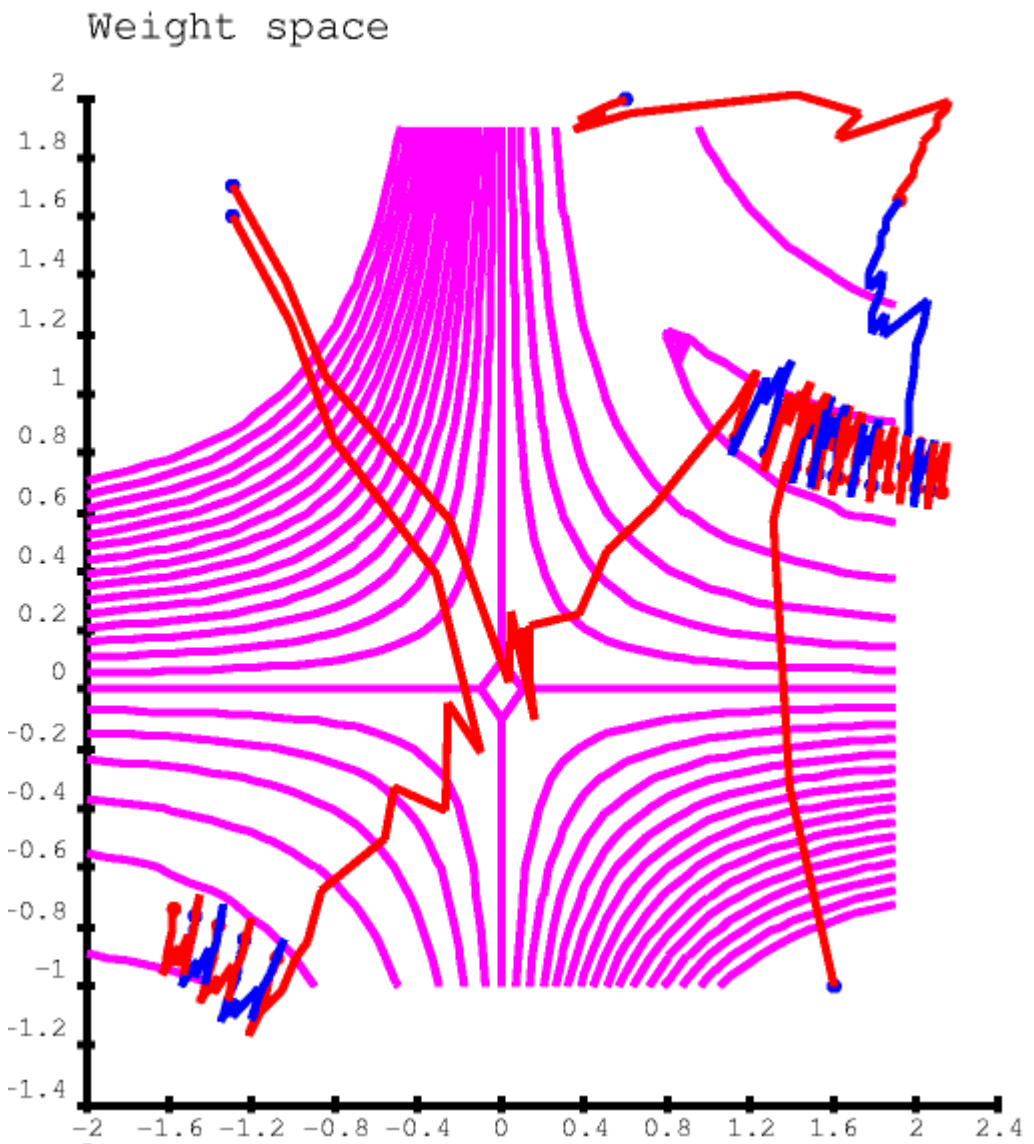
- The simplest 2-layer neural net:
  - 1 input, 1 output, 1 hidden unit
  - 2 weights, 2 biases
- Non-linearity: 1.71.tanh(2/3 X)
- Targets: -1 for class 1, +1 for class 2
- Training set: 20 samples
  - Class 1: 10 samples drawn from a Gaussian with mean -1 and standard deviation 0.4
  - Class 2: 10 samples drawn from a Gaussian with mean +1 and standard deviation 0.4
- Task: "noisy identity function".



the loss landscape for a 2-layer neural net

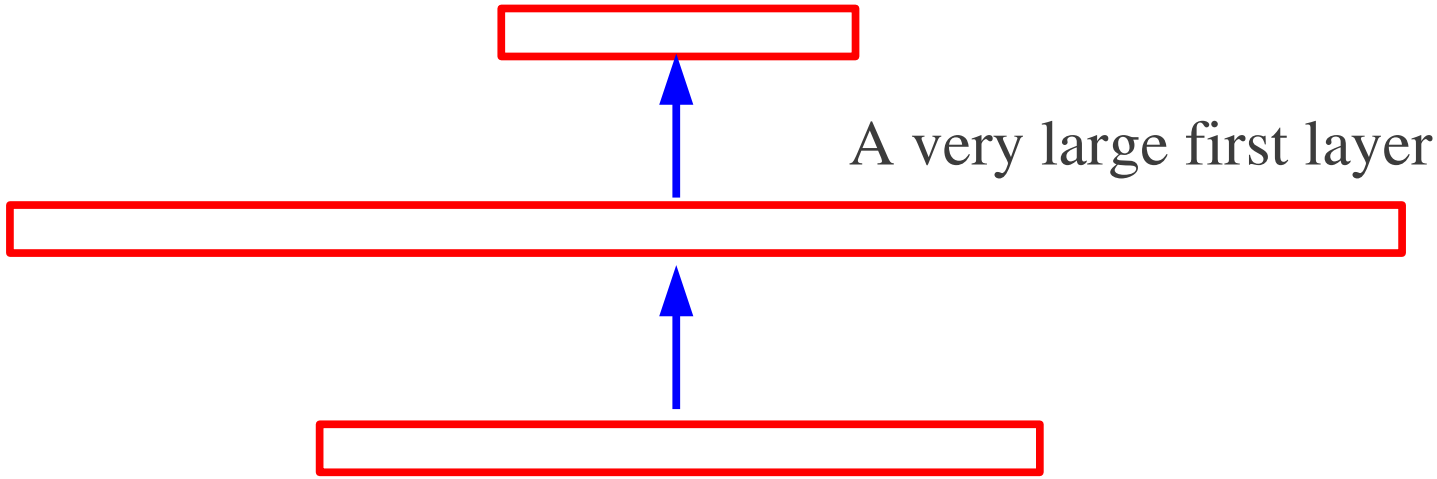
- Learning rate = 0.4





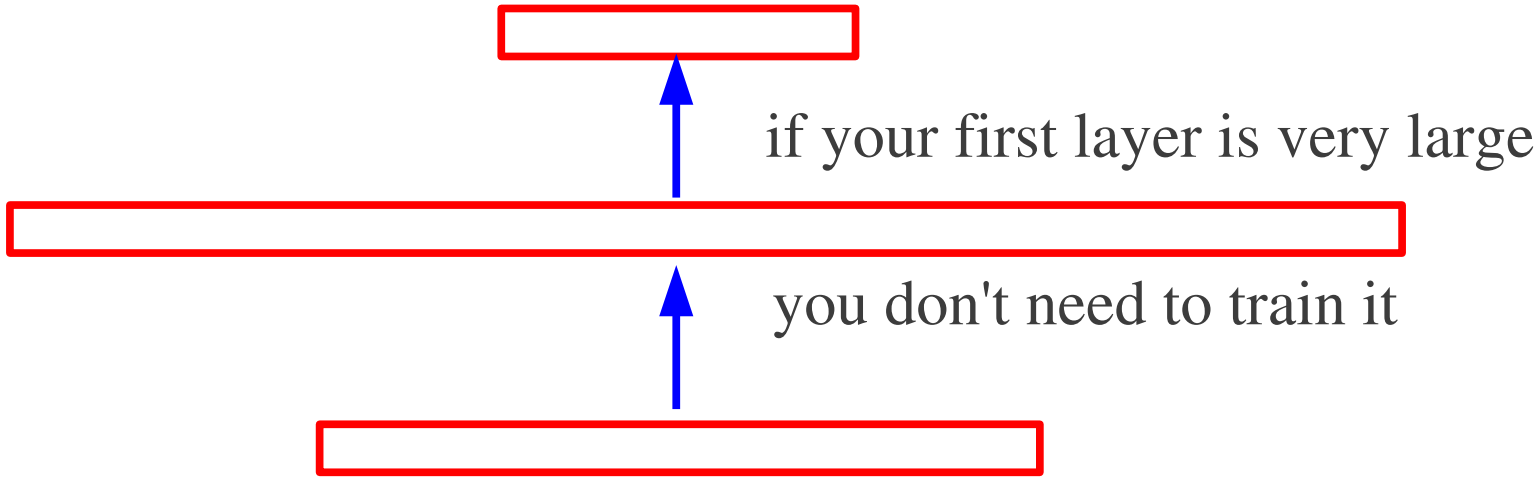
Deep Learning is Hard?

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



Shallow models

- 1957: perceptron: fixed/random first layer. Trainable second layer
- 1985: backprop: both layers are trained. But many people are afraid of the lack of convergence guarantees
- 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



Back-prop learning is not as bad as it seems

- gradient descent is unreliable when the network is small, particularly when the network has just the right size to learn the problem
- the solution is to make the network much larger than necessary and regularize it (SVM taught us that).
- 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
- Many local minima are due to symmetries in the system
- Breaking the symmetry in the architecture solves many problems
  - this may be why convolutional nets work so well