More Optimization Methods

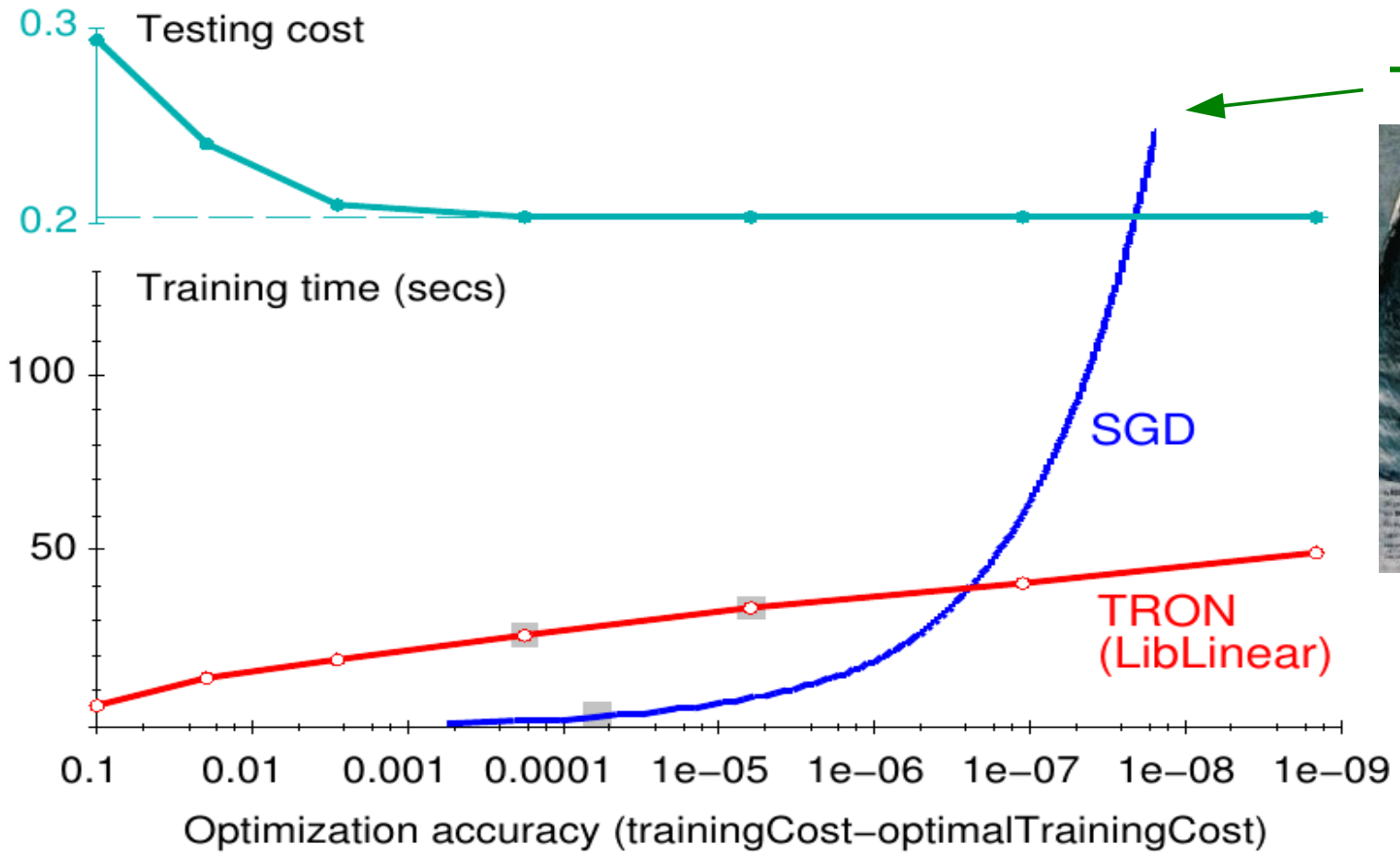
Yann LeCun
Courant Institute
http://yann.lecun.com

(almost) everything you've learned about optimization is...

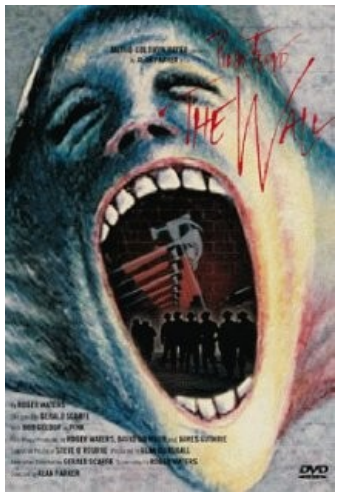
- ... (almost) irrelevant to machine learning
  - particularly to large-scale machine learning.
- OK, that's an exaggeration (with a kernel of truth).
- Much of optimization theory is about asymptotic convergence speed
  - How many additional significant digits of the solution do you get at each iteration when you approach a solution.
  - Steepest descent algorithms (1st order) are "linear": constant number of additional significant digits per iteration
  - Newton-type algorithms (2nd order) are "super-linear".
- But in machine learning, we really don't care about optimization accuracy
  - The loss we are minimizing is not the loss we actually want to minimize
  - The loss we are minimizing is measured on the training set, but we are interested in the performance on the test (or validation) set.
  - If we optimize too well, we overfit!
  - The asymptotic convergence speed is irrelevant

"The Wall" (not by Pink Floyd, but by Léon Bottou)

- SGD is very fast at first, and very slow as we approach a minimum.
- 2nd order methods are (often) slow at first, and fast near a minimum.
- But the loss on the test set saturates long before the crossover
  - http://leon.bottou.org/projects/sgd

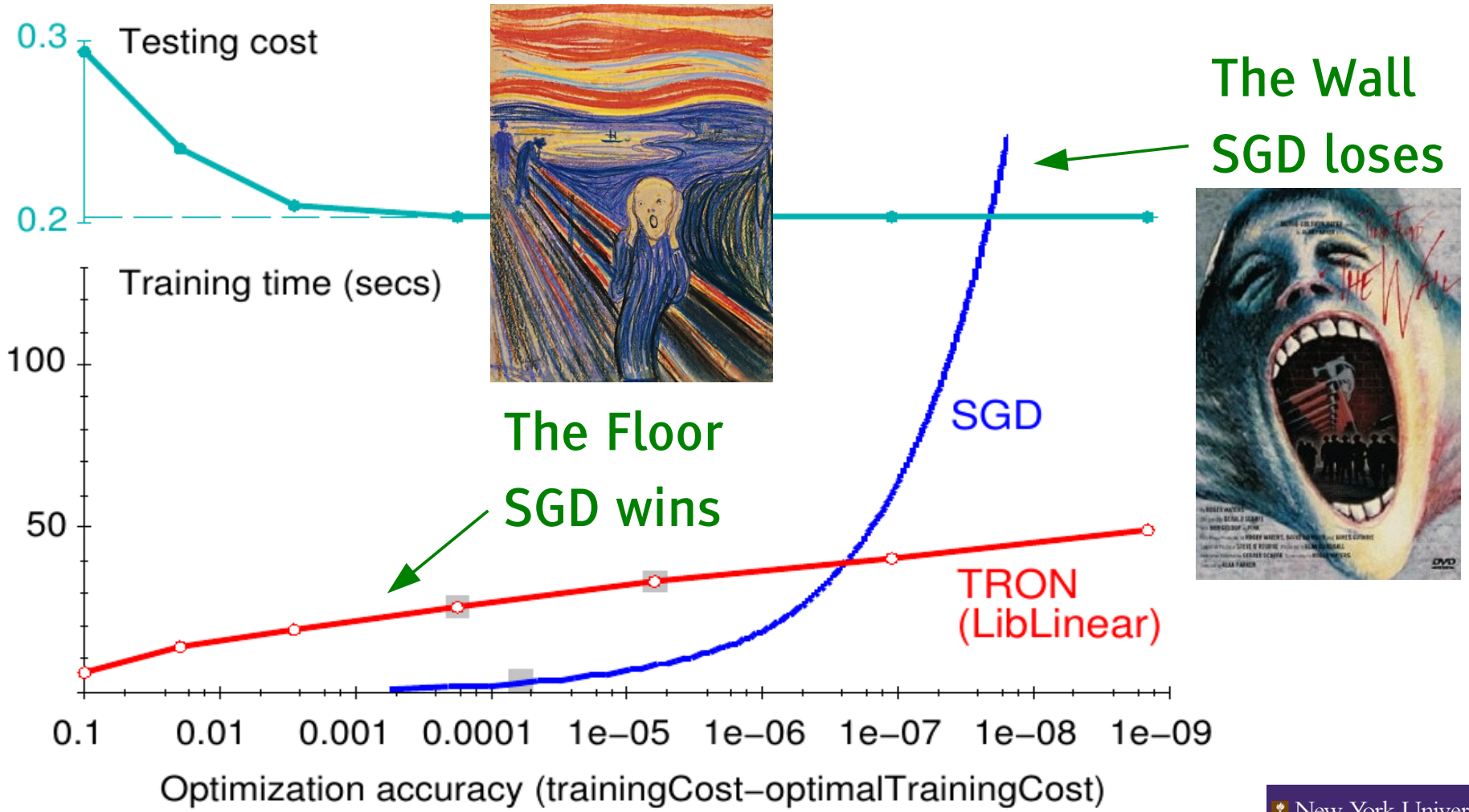


The Wall



"The Wall" (by Léon Bottou, not Pink Floyd)

- SGD is very fast at first, and very slow as we approach a minimum.
- 2nd order methods are (often) slow at first, and fast near a minimum.
- But the loss on the test set saturates long before the crossover

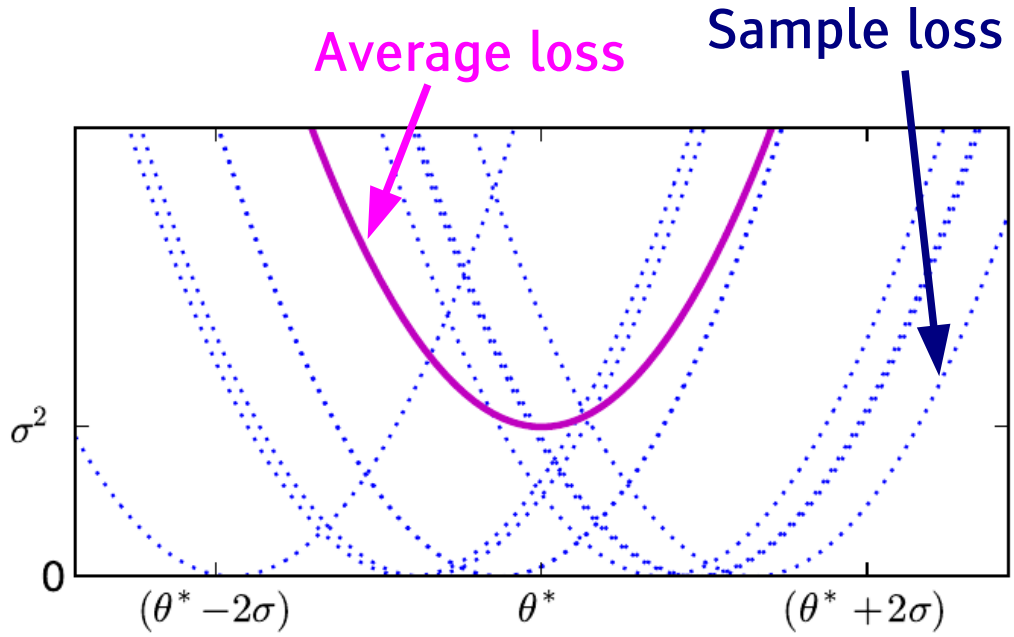


Why is SGD so fast? Why is SGD so slow?

- **SGD is fast on large datasets because it exploits the redundancy in the data**
  - I give you 1,000,000 samples, produced by replicating 10,000 samples 100 times.
  - After 1 epoch, SGD will have done 100 iterations on the 10,000 sample dataset
  - Any batch method will waste 100 times too much computation for each single iteration.
- **SGD is slow near a solution because of gradient noise**
  - Example: least square regression

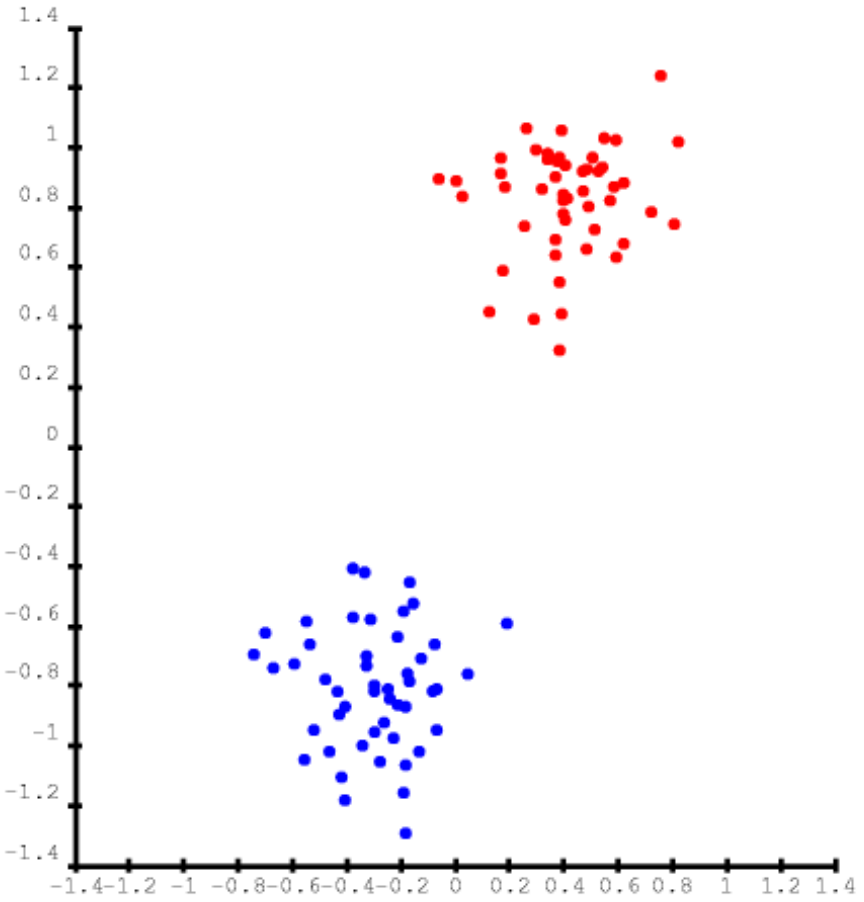
$$L(W)=\frac{1}{P}\sum_{i=1}^{P}\|y^i-W'X^i\|^2$$

  - Samples have different minima
  - Gradient pulls towards each one
  - Learning rate must decrease
  - Very slow near the minimum
  - But we don't care!



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

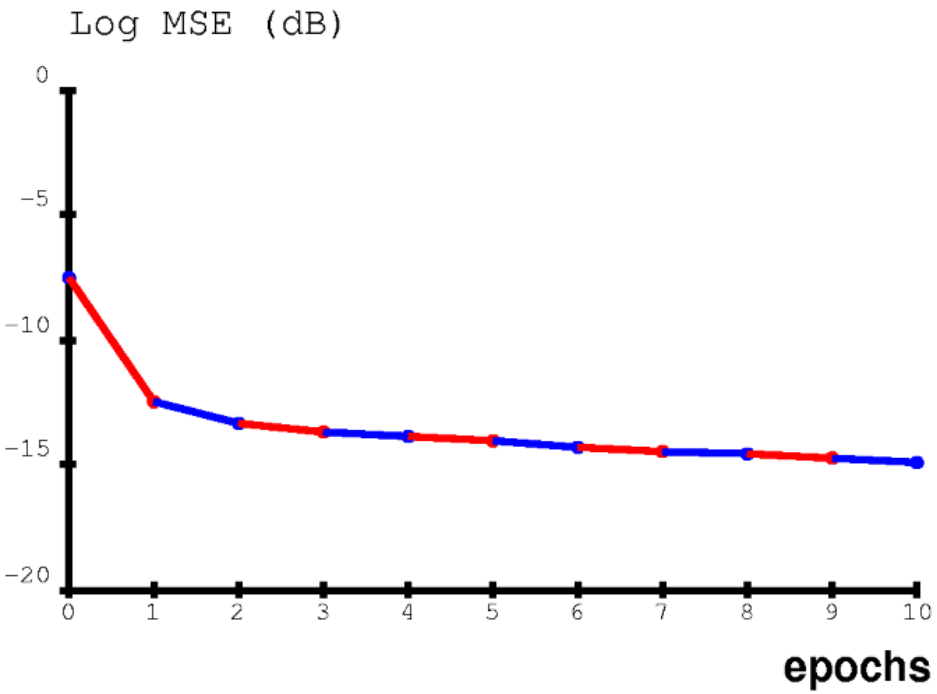


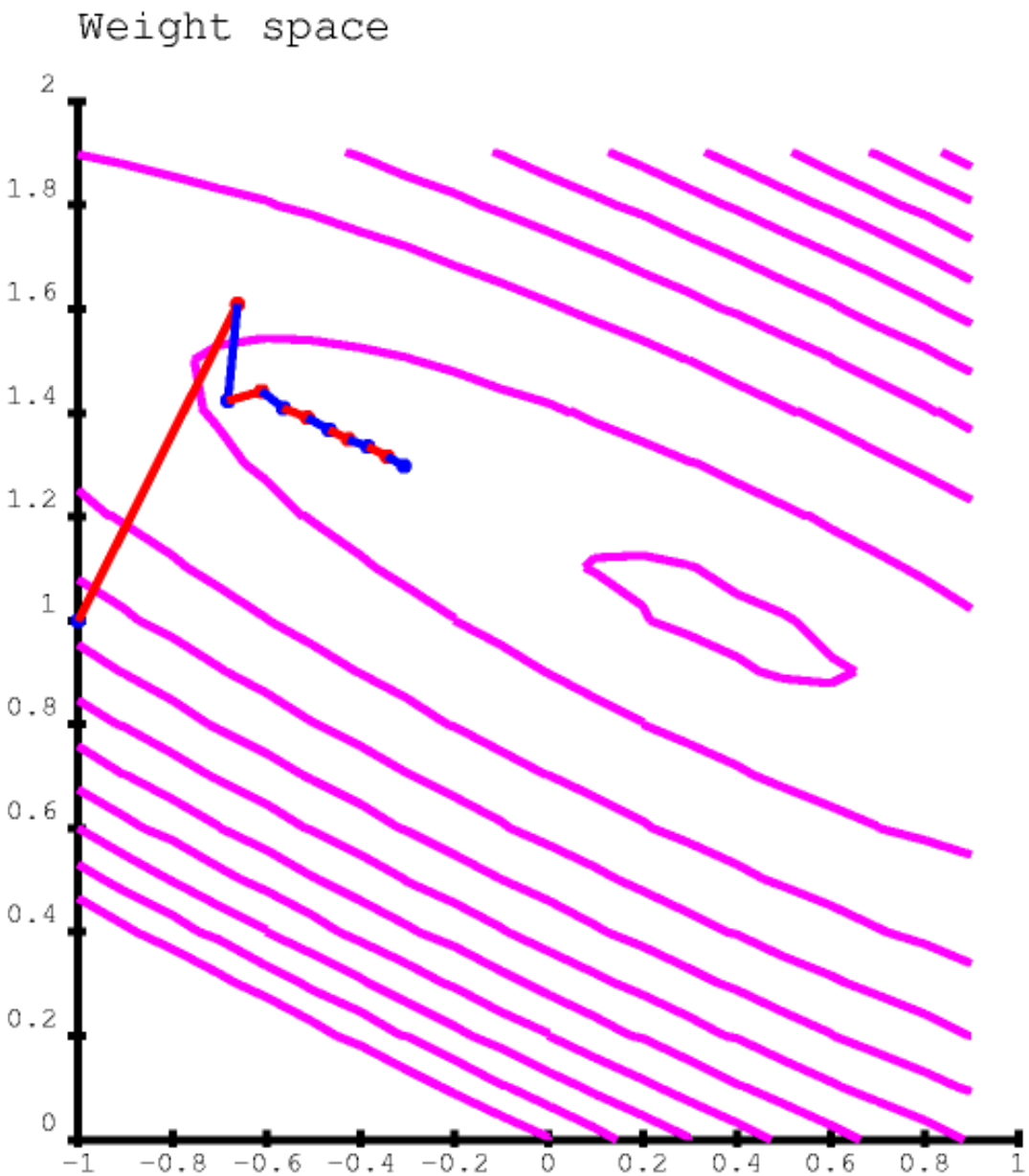
- Least square regression with y in {-1,+1}

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

Example

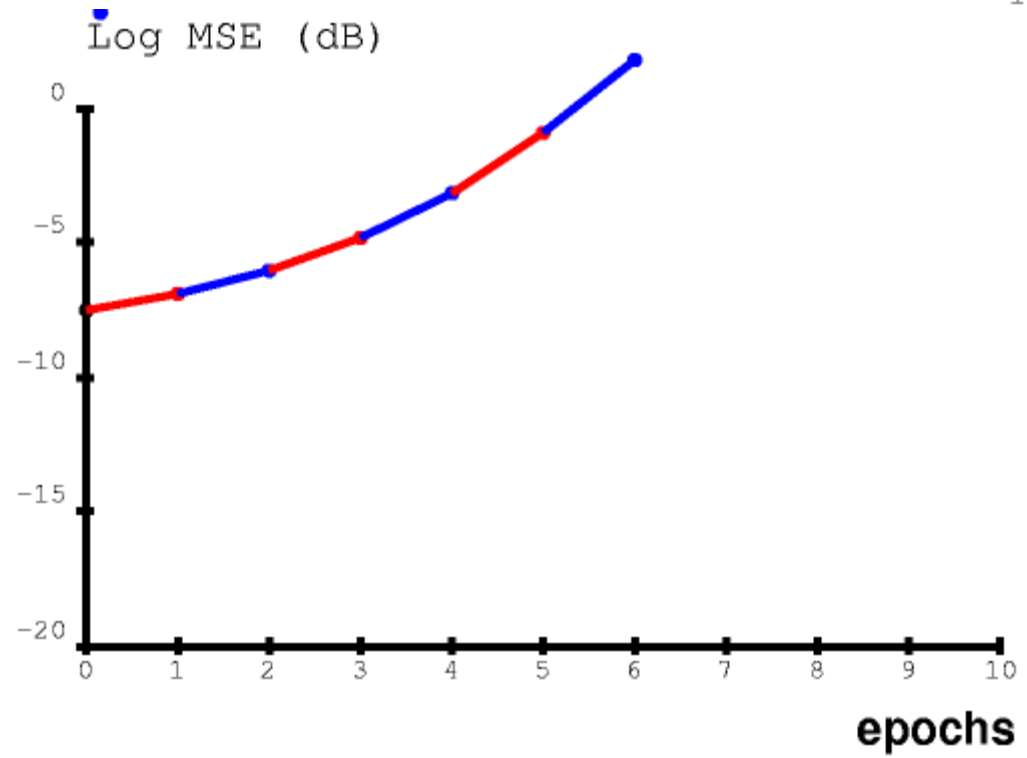
- Batch gradient descent
  - Learning rate = 1.5
  - Divergence for 2.38

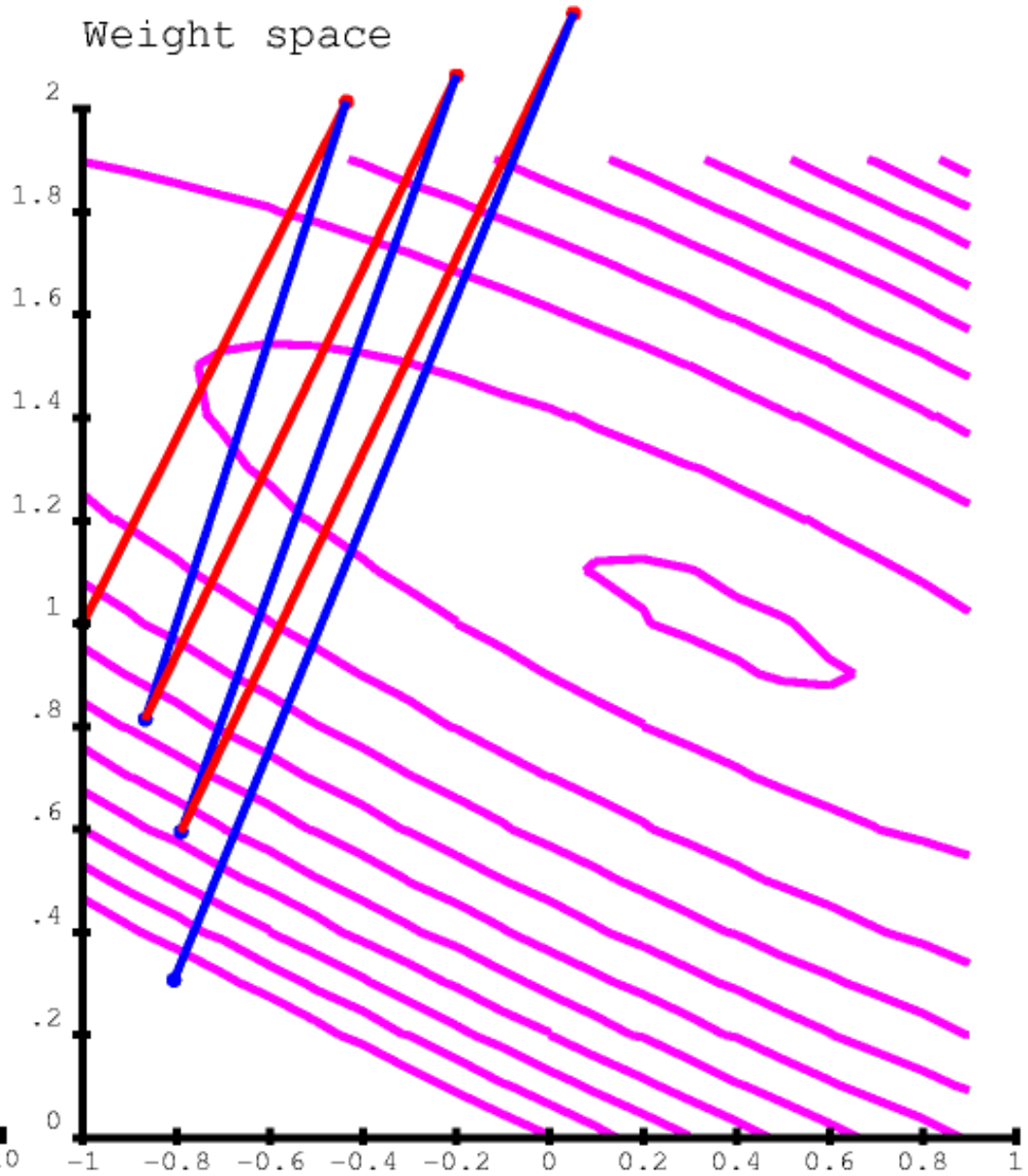




Example

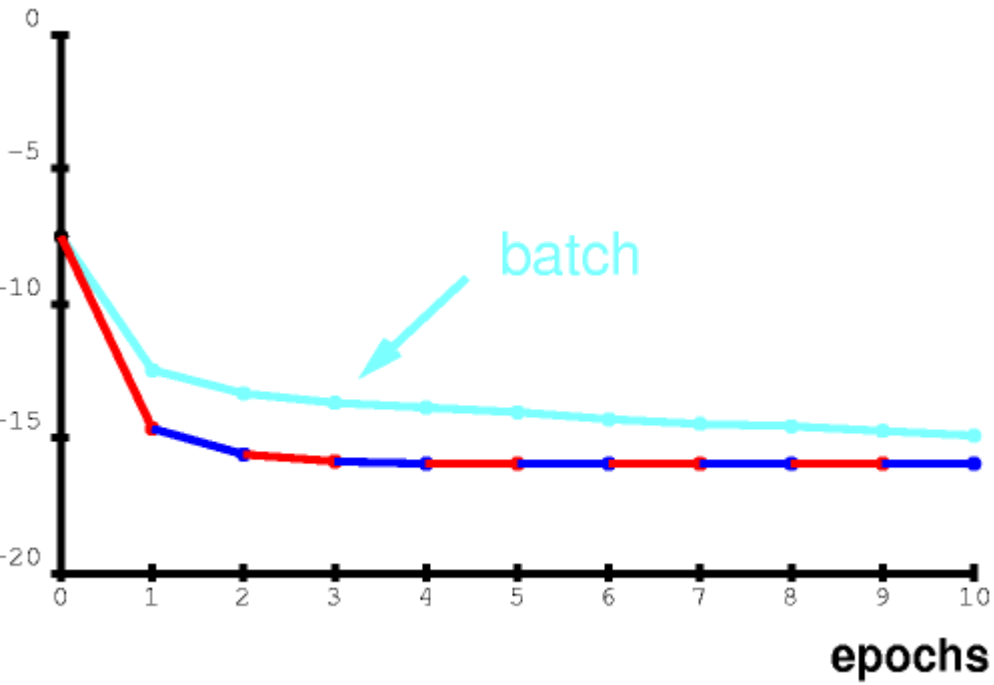
- Batch gradient descent
  - Learning rate = 2.5
  - Divergence for 2.38

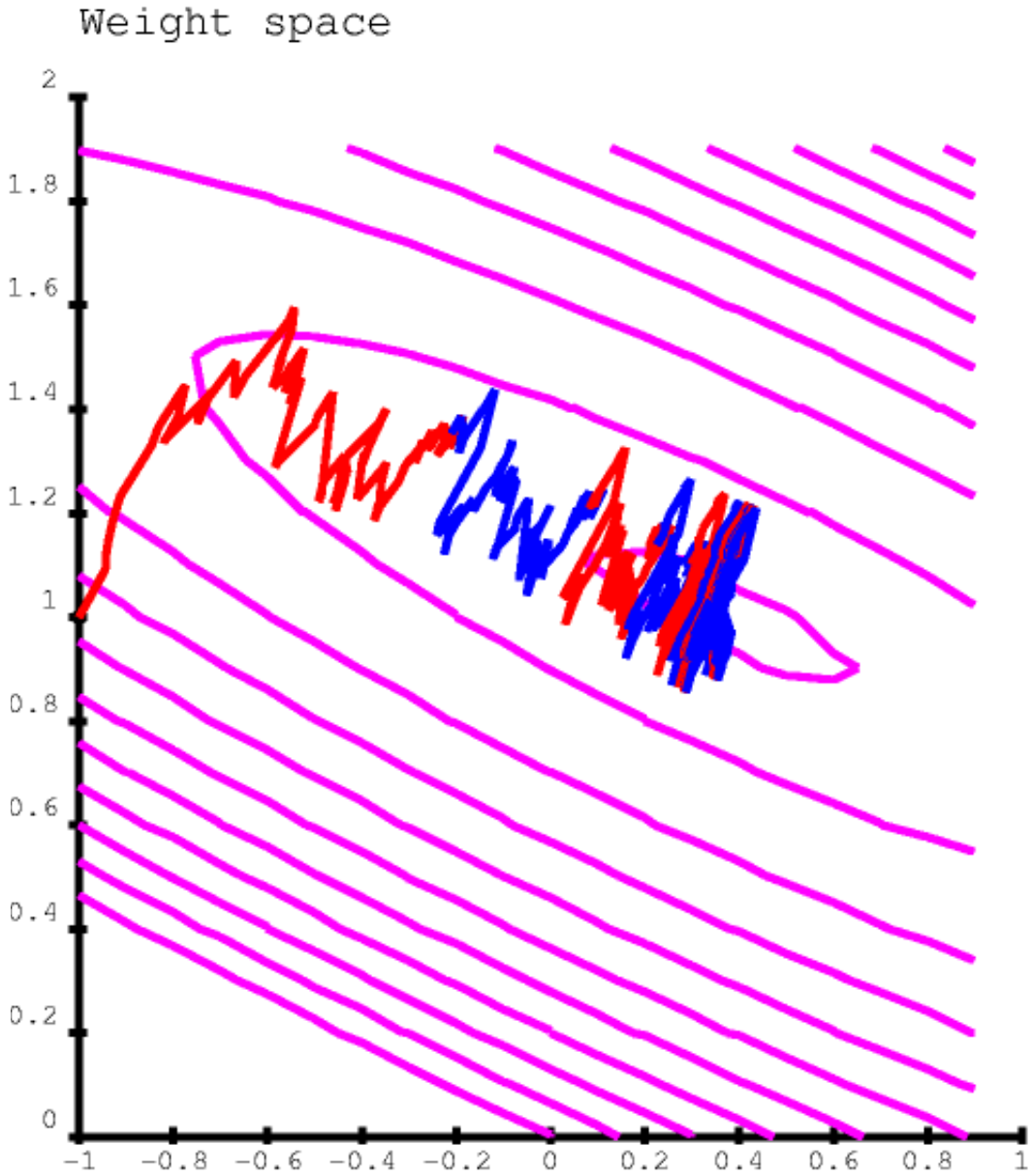




Example

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



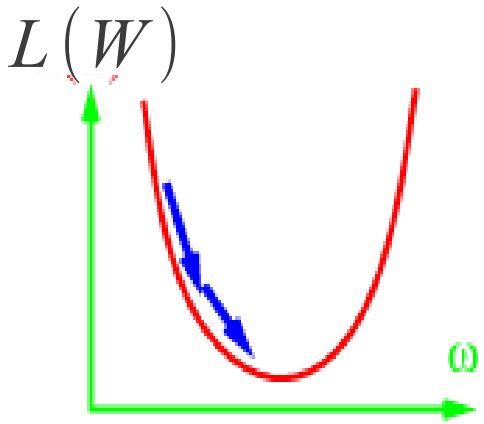


The convergence of gradient descent: 1 dimension

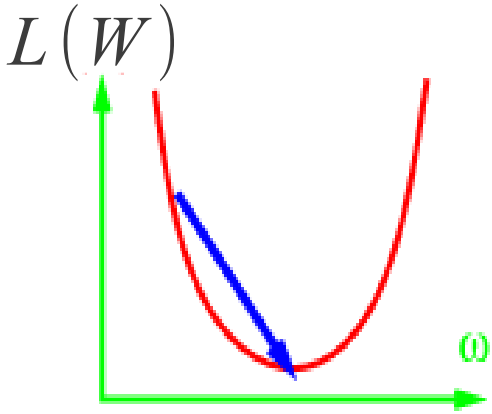
- Update rule:

$$W_{t+1} = W_t - \eta_t \frac{\partial L(W_t)}{\partial W}$$

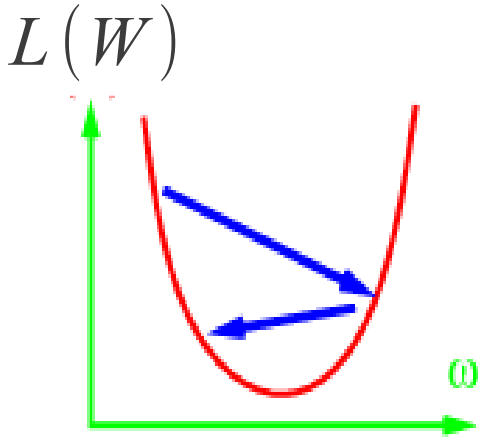
- What is the optimal learning rate?



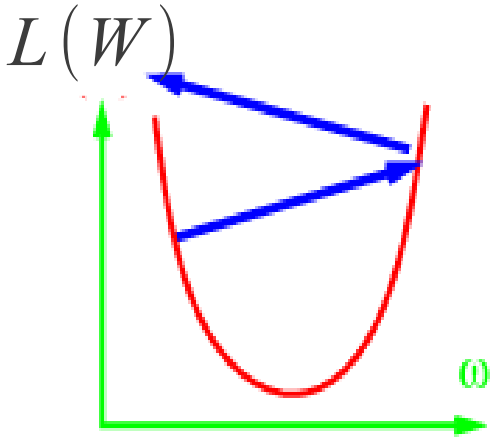
$\eta < \eta_{opt}$



$\eta = \eta_{opt}$



$\eta > \eta_{opt}$



$\eta > 2\,\eta_{opt}$

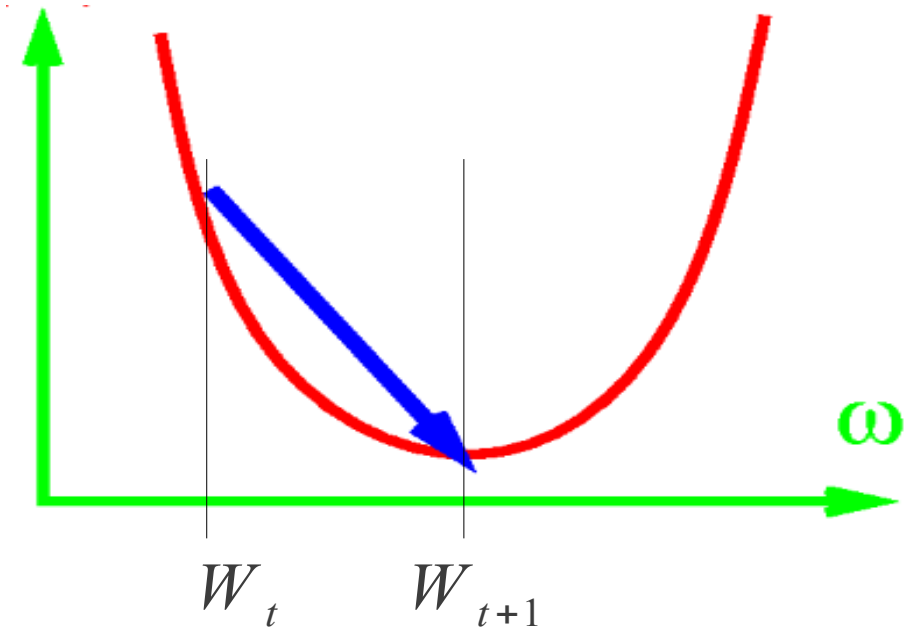
The convergence of gradient descent

- Taylor expansion of the loss around Wt:

$$L(W) \approx \frac{1}{2}[W - W_t]' H(W_t)[W - W_t] + G(W_t)[W - W_t] + L_0$$

- G: gradient (1xN), H: hessian (NxN)

$$G(W_t) = \frac{\partial L(W_t)}{\partial W} \quad ; \quad H(W_t) = \frac{\partial^2 L(W_t)}{\partial W \partial W'}$$



The convergence of gradient descent

- Taylor: $L(W) \approx \frac{1}{2}[W - W_t]' H(W_t)[W - W_t] + G(W_t)[W - W_t] + L_0$

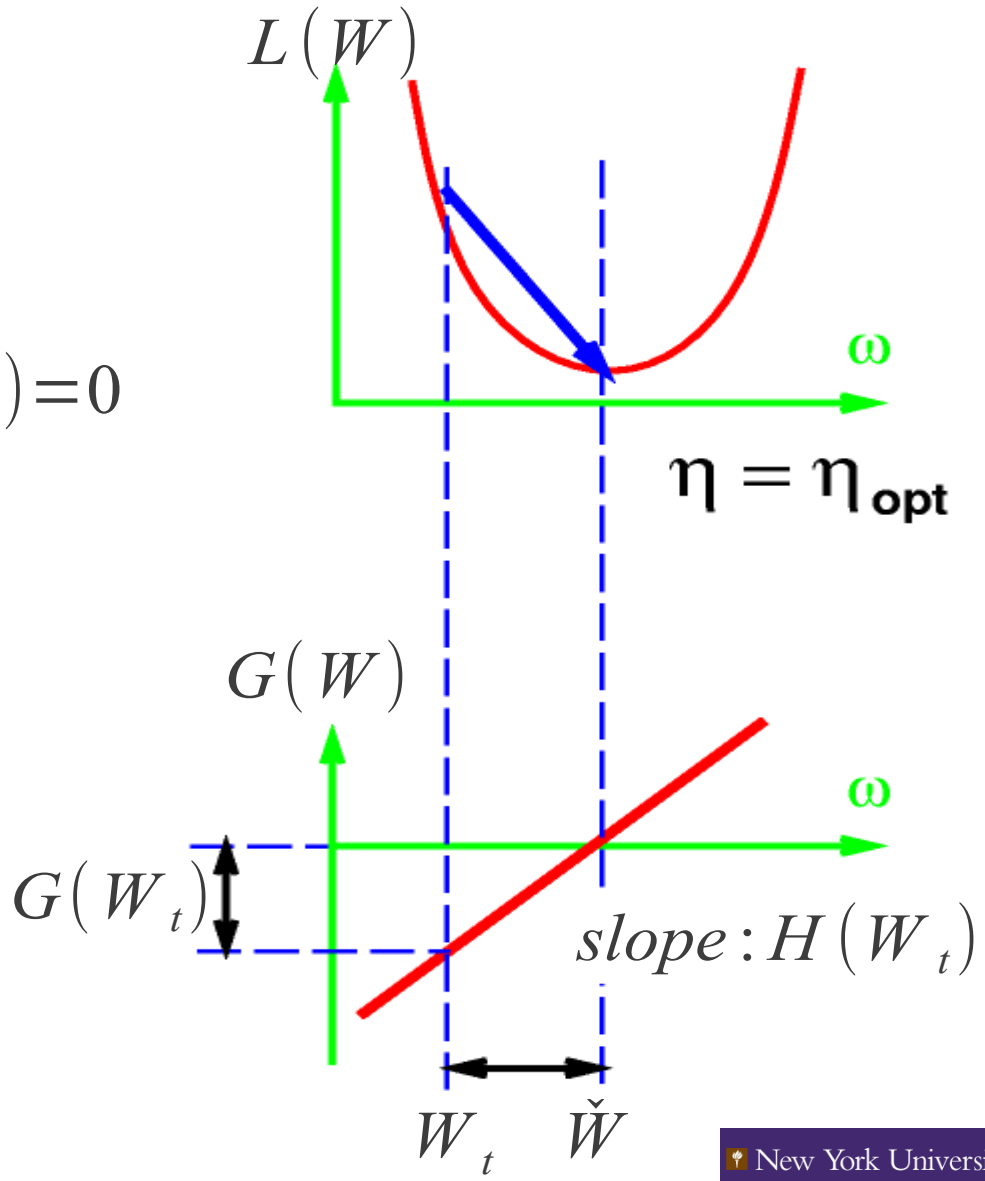
- At the solution: $\frac{\partial L(\check{W})}{\partial W} = 0$

$$\frac{\partial L(\check{W})}{\partial W} = H(W_t)[\check{W} - W_t] - G(W_t) = 0$$

$$\check{W} = W_t - H(W_t)^{-1} G(W_t)$$

$$\check{W} = W_t - \eta_{opt} G(W_t)$$

$$\eta_{opt} = H(W_t)^{-1}$$



Newton's Algorithm

- At each iteration, solve for W{t+1}:

$$H(W_t)[W_{t+1} - W_t] - G(W_t) = 0$$

- Newton's algorithm is impractical for most machine learning tasks
  - It's batch!
  - Each iteration is O(N^3)
  - It only works if the objective is convex and near quadratic
  - If the function is non convex, it can actually go uphill

Newton's Algorithm as a warping of the space

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

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

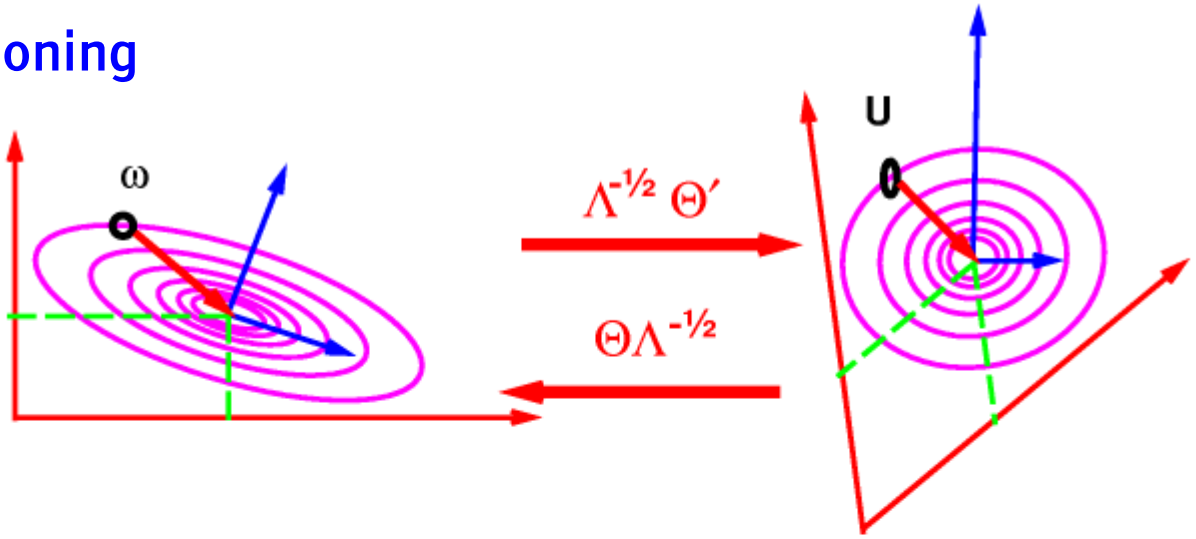
$$H^{-1} = \Theta \Lambda^{-\frac{1}{2}} \Lambda^{-\frac{1}{2}} \Theta'$$

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

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

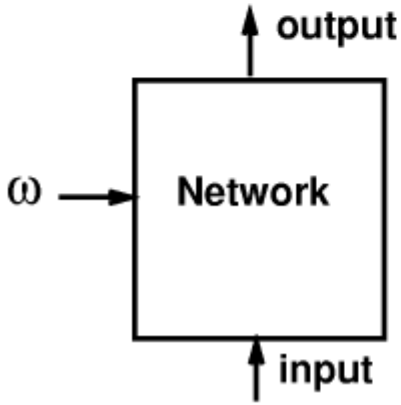
$$\delta U = H^{-\frac{1}{2}} \frac{\partial L}{\partial W}$$

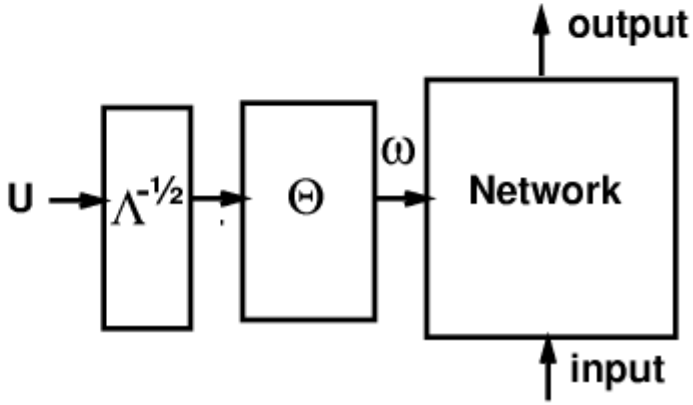
$$\delta W = H^{-1} \frac{\partial L}{\partial W}$$



Newton Algorithm here .....

....is like Gradient Descent there





Convergence of Gradient Descent

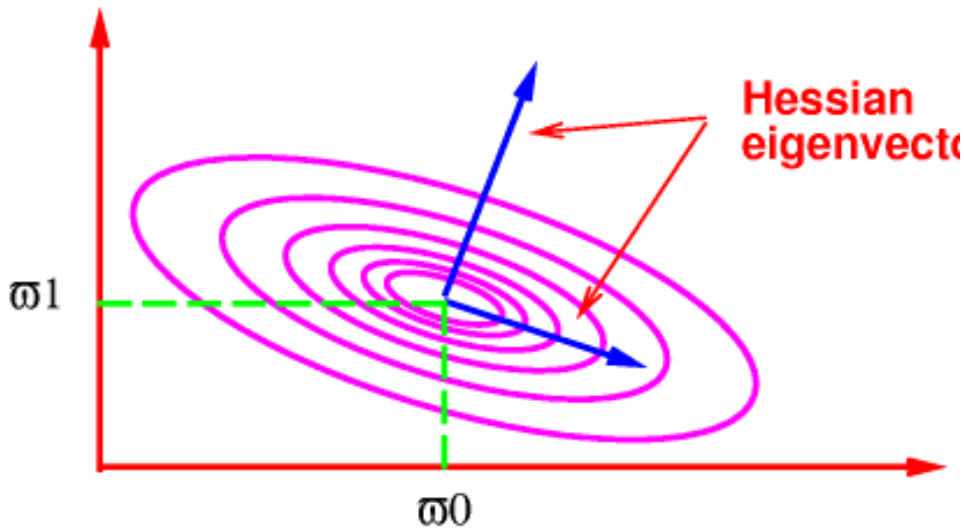
- Taylor expansion around a minimum:

$$L(W) \approx \frac{1}{2}[W - \check{W}]' H(\check{W})[W - \check{W}] + L_0$$

$$W_{t+1} = W_t - \eta \frac{\partial L(W_t)}{\partial W} = W_t - \eta H(W_t - W)$$

$$W_{t+1} - \check{W} = (I - \eta H)(W_t - \check{W})$$

- This needs to be a contracting operator
  - All the eigenvalues of (I-eta H) must be between -1 and 1



Convergence of Gradient Descent

- Lets' diagonalise H: $H = \Theta' \Lambda \Theta \;\; ; \;\; \Lambda = \Theta H \Theta'$

$\Theta' \Theta = I \qquad \Theta \, is \, a \, rotation \, , \, \Lambda \, is \, diagonal$

- transform the space (change of variable):

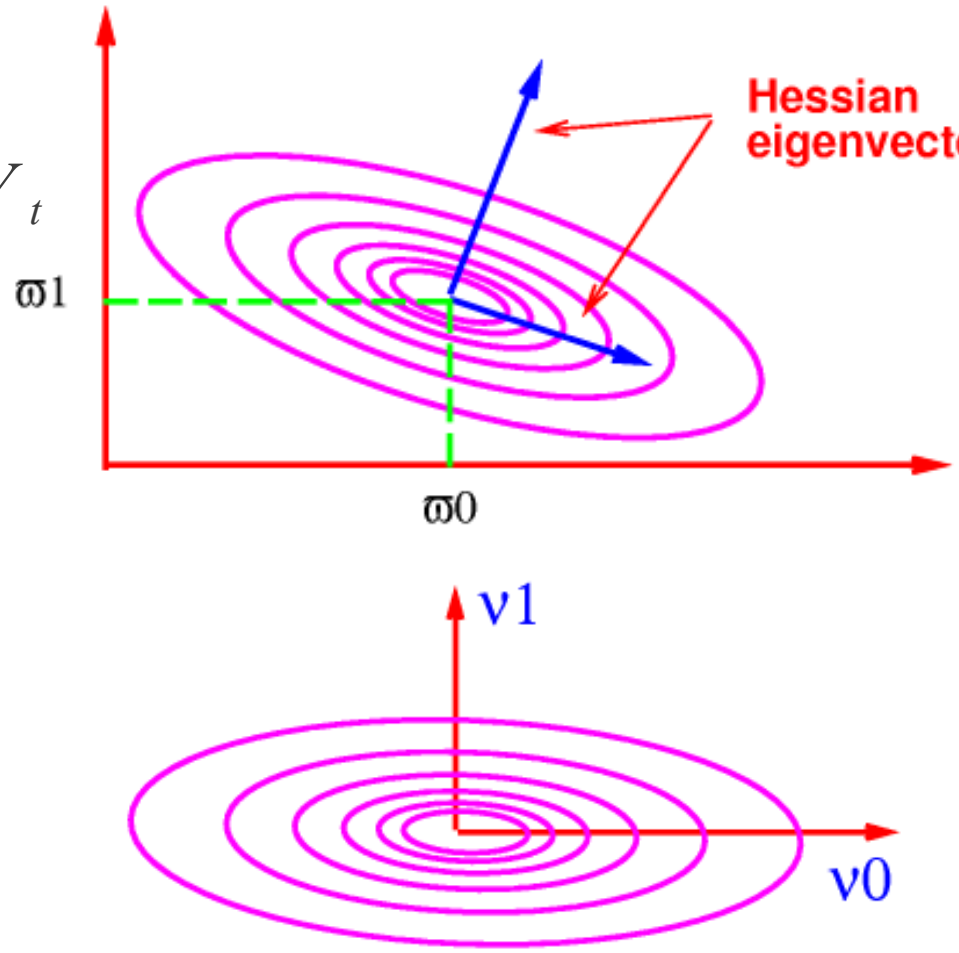
$$V_t = \Theta (W_t - \check{W}) \;\; ; \;\; (W_t - \check{W}) = \Theta' V_t$$

$$W_{t+1} - \check{W} = (I - \eta H)(W_t - \check{W})$$

$$\Theta' V_{t+1} = (I - \eta H) \Theta' V_t$$

$$V_{t+1} = \Theta (I - \eta H) \Theta' V_t$$

$$V_{t+1} = (I - \eta \Lambda) V_t$$



Convergence of Gradient Descent

$$V_{t+1} = (I - \eta \Lambda) V_t$$

- This is N independent update equations of the form:

$$V_{t+1,j} = (1 - \eta \lambda_j) V_{t,j}$$

- These terms are all between -1 and 1 when: $\eta < \frac{2}{\lambda_{max}}$

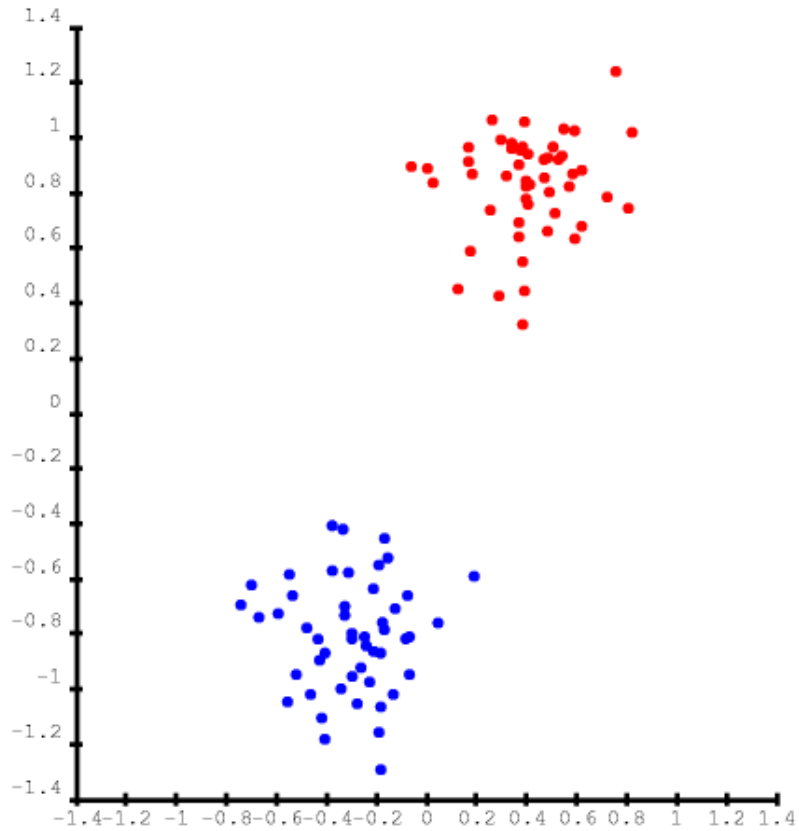
Gradient Descent in N dimensions can be viewed as N independent unidimensional Gradient Descents along the eigenvectors of the Hessian.

Convergence is obtained for $\eta < 2/\lambda_{max}$

where λ_{max} is the largest eigenvalue of the Hessian

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}



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

$$L(W)=\frac{1}{P}\sum_{i=1}^{P}[\frac{1}{2}W'X^iX^{i\prime}W-y^iX^{i\prime}W+(y^i)^2]$$

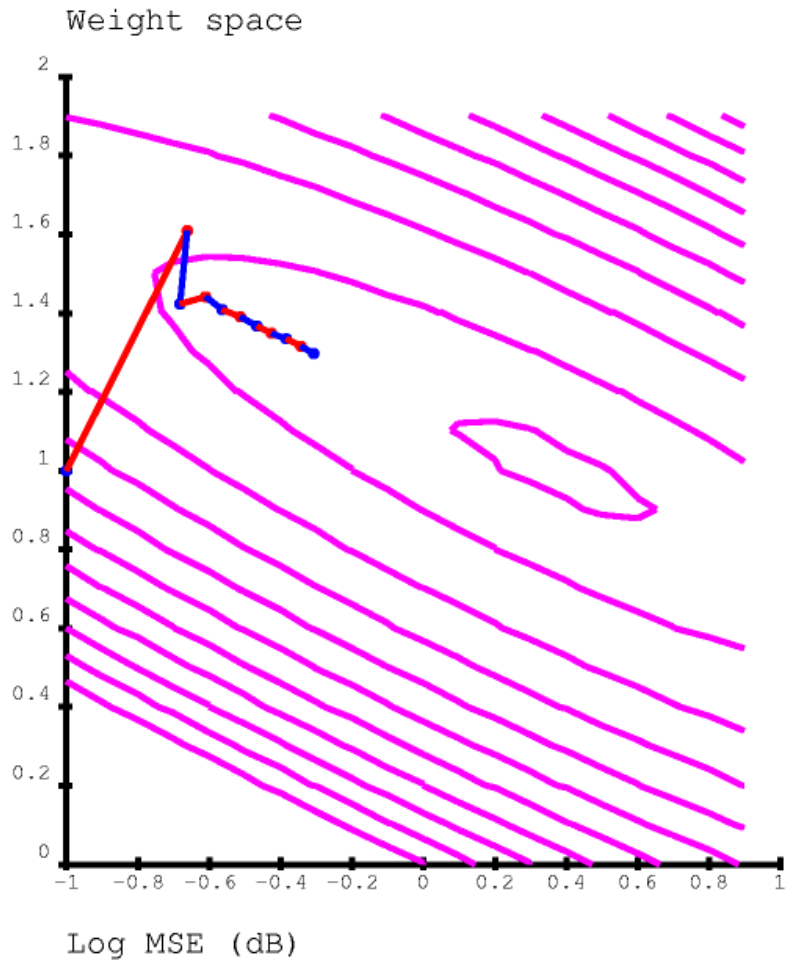
$$L(W)=W'[\frac{1}{2P}\sum_{i=1}^{P}X^iX^{i\prime}]W-[\frac{1}{2P}\sum_{i=1}^{P}y^iX^{i\prime}]W+[\frac{1}{2P}\sum_{i=1}^{P}(y^i)^2]$$

Problem

- Batch gradient descent

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

- Eigenvalues are 0.83 and 0.036



- The largest eigenvalue limits the maximum learning rate (2.36)
- But for this learning rate, convergence is slow in the other dimensions
- Convergence time is proportional to: $\frac{1}{\eta \lambda_{min}}$
- Convergence time for $\eta = \frac{1}{\lambda_{max}}$: $\tau = \frac{\lambda_{max}}{2\lambda_{min}}$

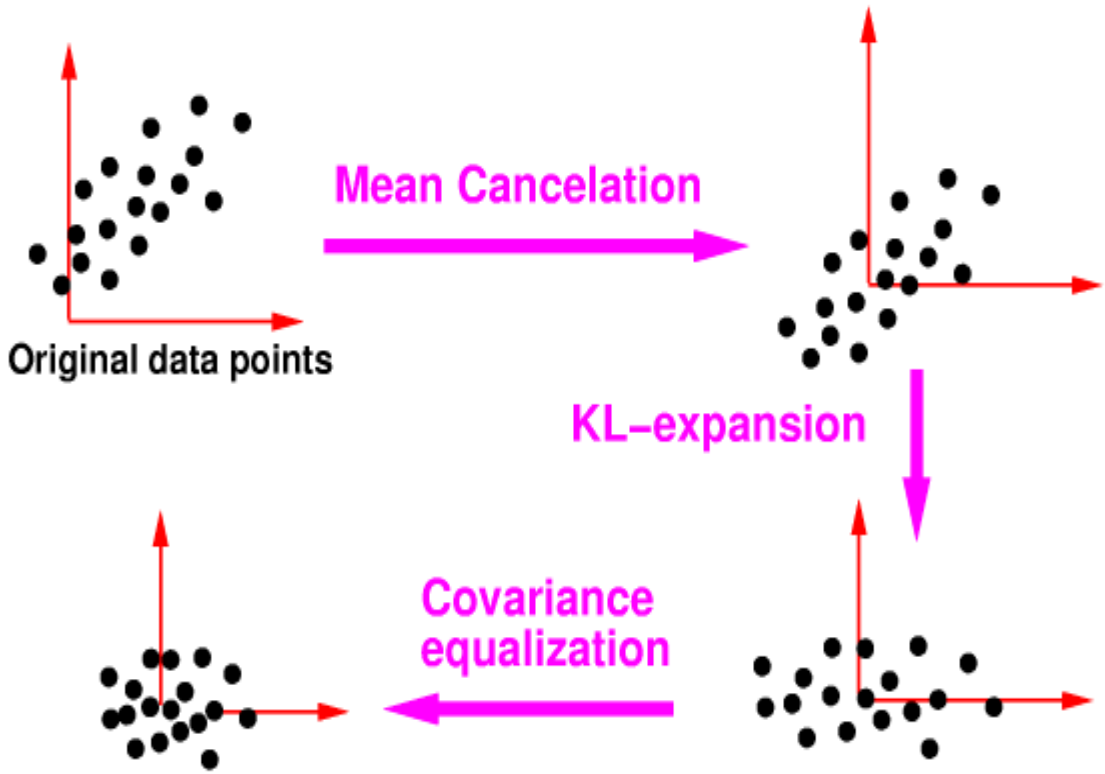
- Convergence time is proportional to condition number

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

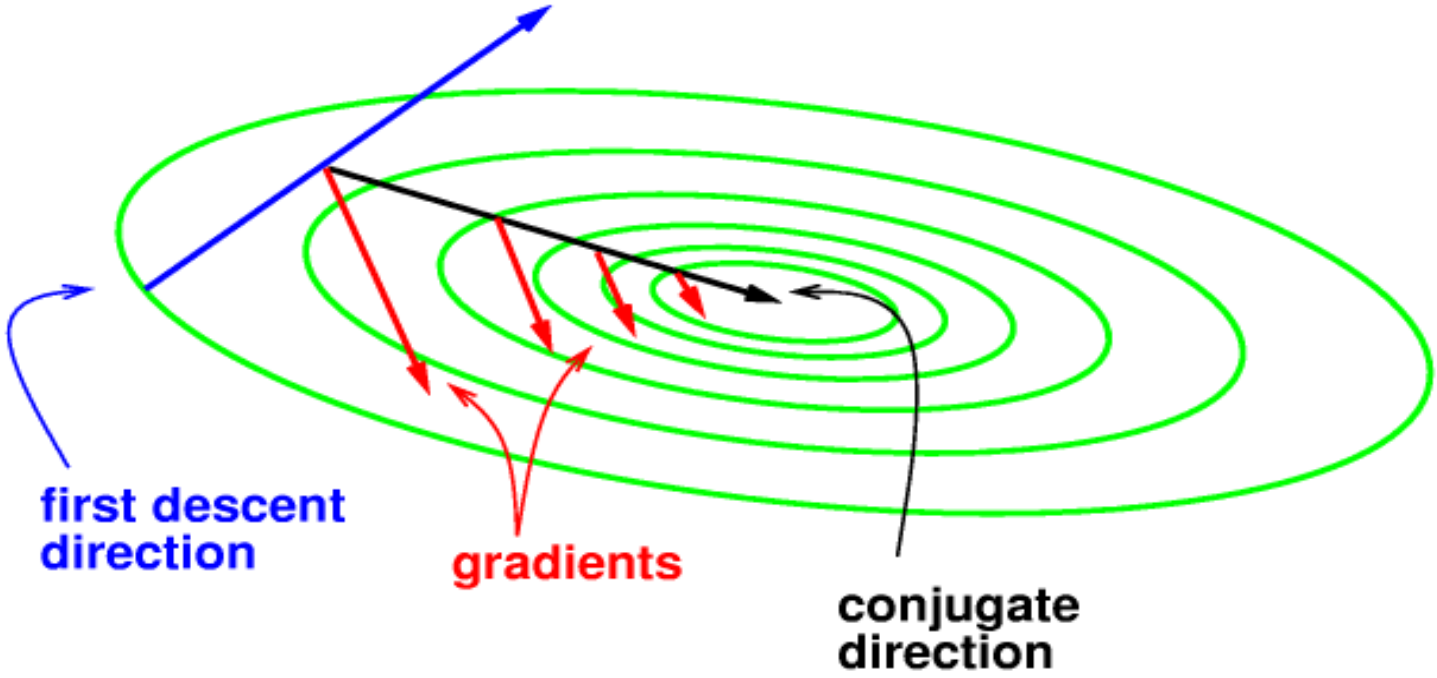
Classical (non stochastic) Optimization Methods

- They are occasionally useful, but rarely
  - Linear system with a finite training set
  - "stochastic" training with mini-batch on certain machines
  - Search for hyper-parameters
  - Optimization over latent variables
- Conjugate gradient
  - Iteration is O(N)
  - Works with non-quadratic functions
  - Requires a line search
  - Can't work in stochastic mode. Requires mini-batches
- Levenberg-Marquardt
  - Iteration is O(N^2) (except for diagonal approximation)
  - Full version only practical for small N
  - Works with non-quadratic functions

Conjugate Gradient

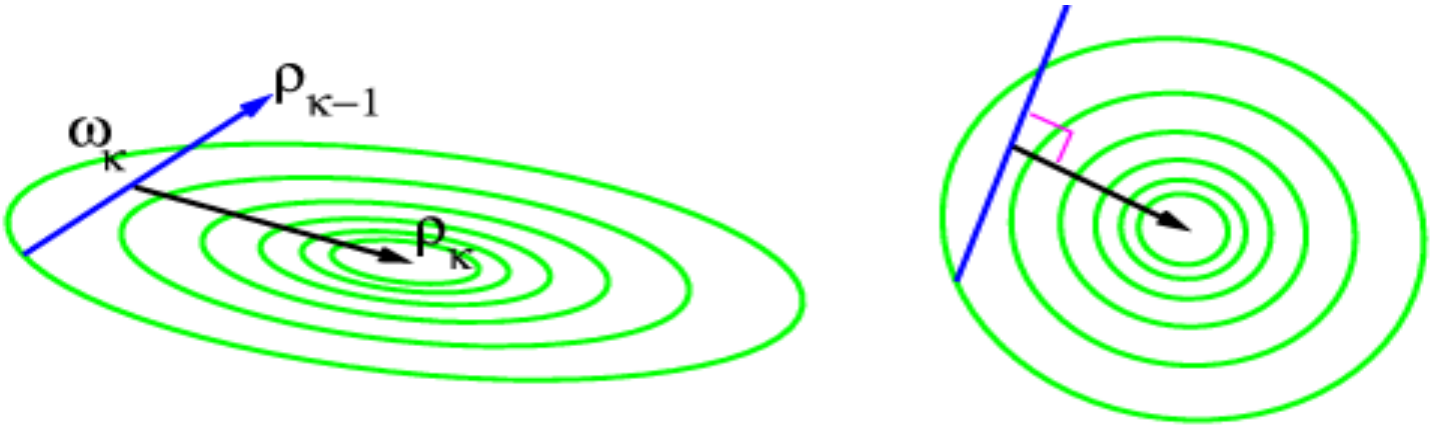
MAIN IDEA: to find a descent direction which does not spoil the result of the previous iterations

Pick a descent direction (say the gradient), find the minimum along that direction (line search). Now, find a direction along which the gradient does not change its direction, but merely its length (conjugate direction). Moving along that direction will not spoil the result of the previous iteration



There are two slightly different formulae: Fletcher–Reeves & Polak–Ribiere

Conjugate Gradient



Conjugate directions are like ORTHOGONAL directions in the space where the Hessian is the identity matrix

p and q are conjugate <===> p'Hq = 0

ρ_κ: **descent direction at iteration k**

$$\rho_\kappa = -\nabla E(\omega_\kappa) + \beta_\kappa \rho_{\kappa-1}$$

$$\beta_\kappa = \frac{\nabla E(\omega_\kappa)' \nabla E(\omega_\kappa)}{\nabla E(\omega_{\kappa-1})' \nabla E(\omega_{\kappa-1})}$$ **(Fletcher–Reeves)**

$$\beta_\kappa = \frac{(\nabla E(\omega_\kappa) - \nabla E(\omega_{\kappa-1}))' \nabla E(\omega_\kappa)}{\nabla E(\omega_{\kappa-1})' \nabla E(\omega_{\kappa-1})}$$ **(Polak–Ribiere)**

- **A good line search must be done along each descent direction (works only in batch)**
- **Convergence in N iterations is guaranteed for a quadratic function with N variables**

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:

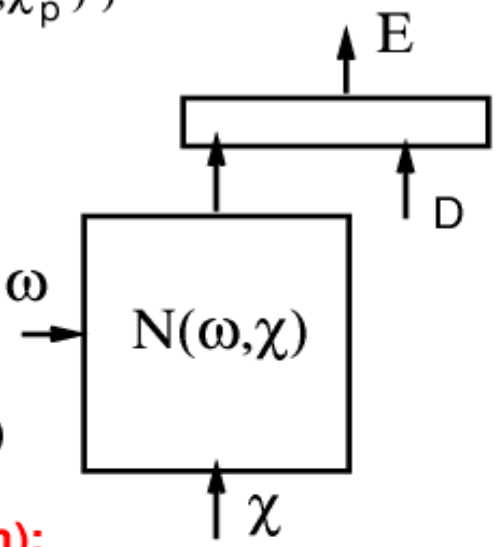
$$E(\omega) = 1/2 \sum_p (D_p - N(\omega,\chi_p))' (D_p - N(\omega,\chi_p))$$

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;
}
```

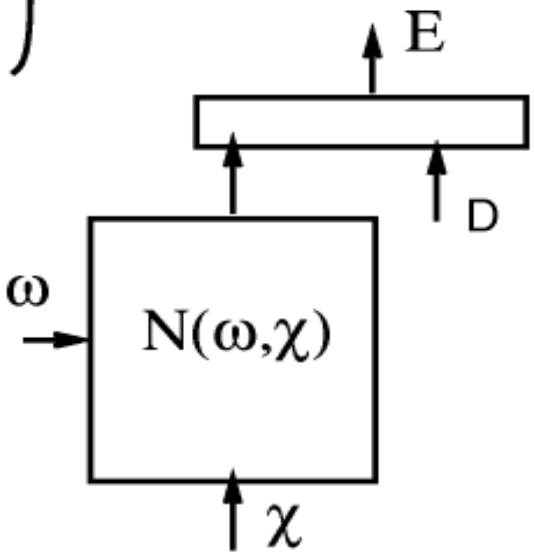
Computing the product of the Hessian by a Vector

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

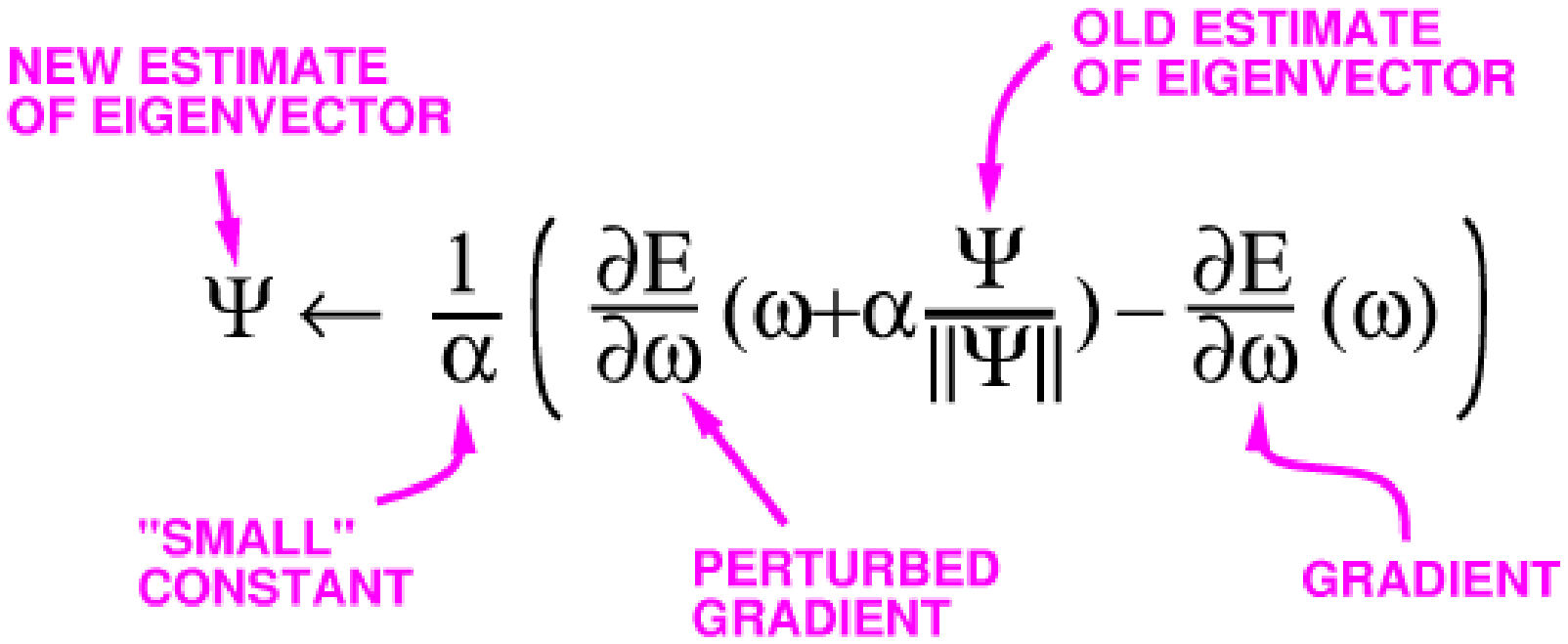
HESSIAN

ESTIMATE OF EIGENVALUE

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

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

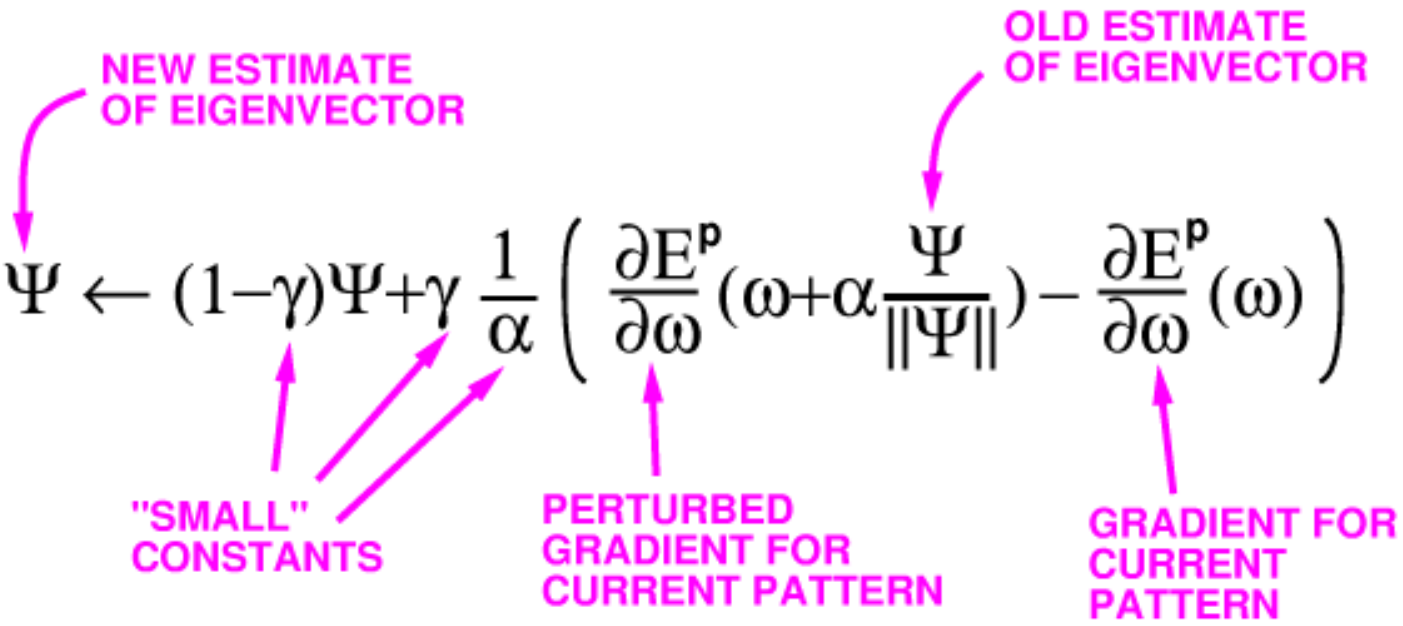
Using the Power Method to Compute the Largest Eigenvalue



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

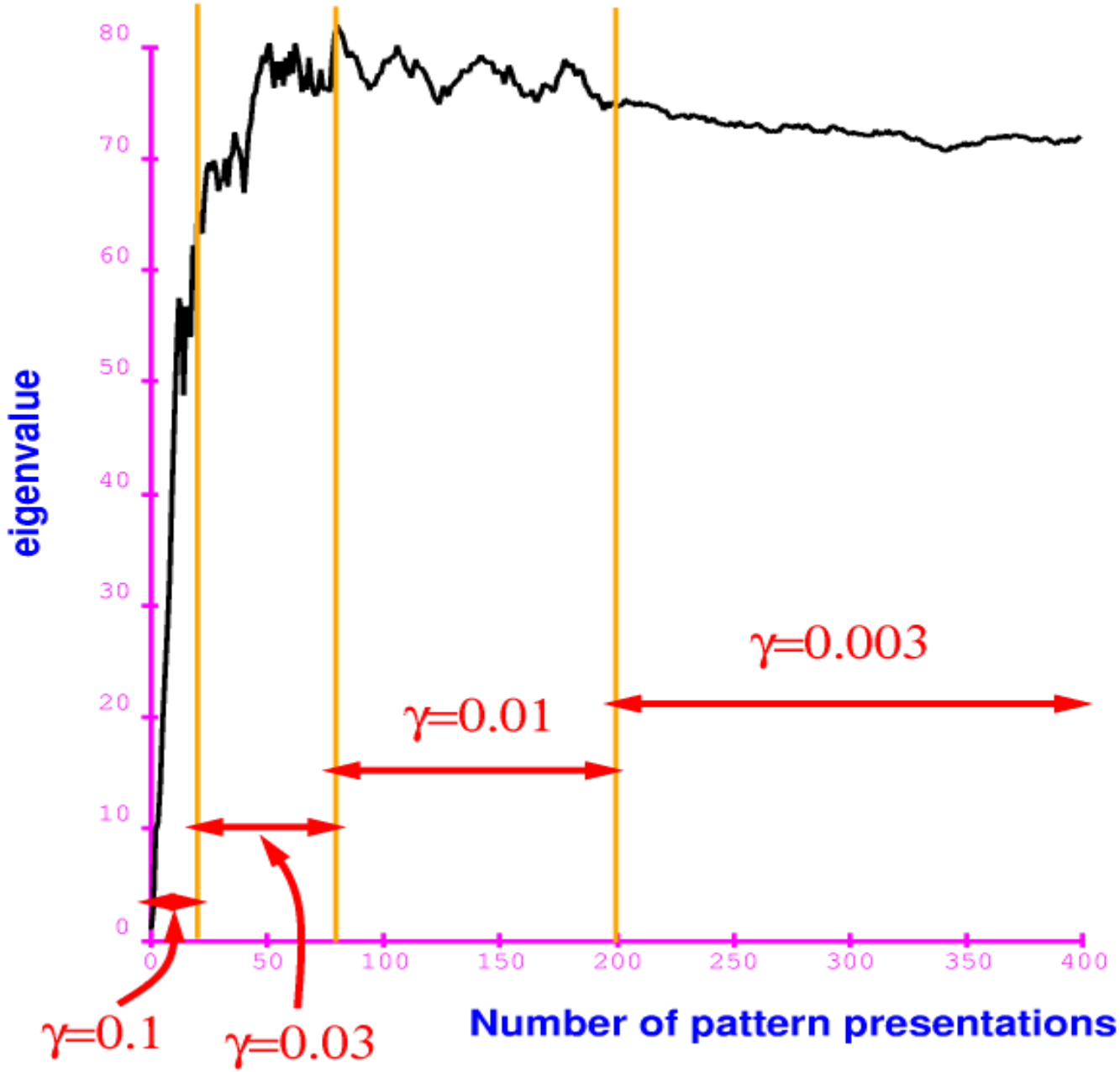
4 – perform forward prop and backward prop with
perturbed weight vector. Save gradient vector G'(w)

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

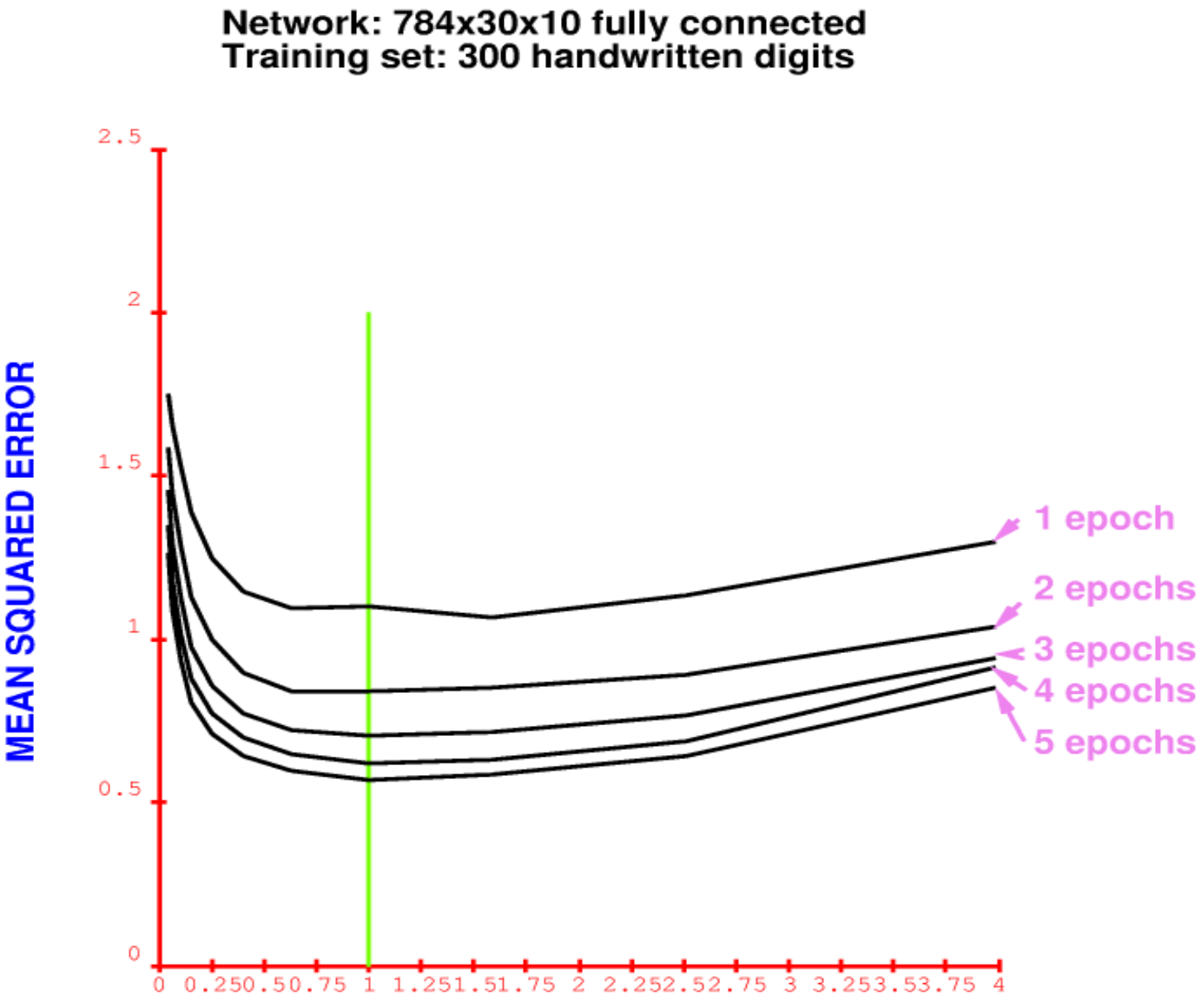
6 – goto 2 unless a reasonably stable result is obtained

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

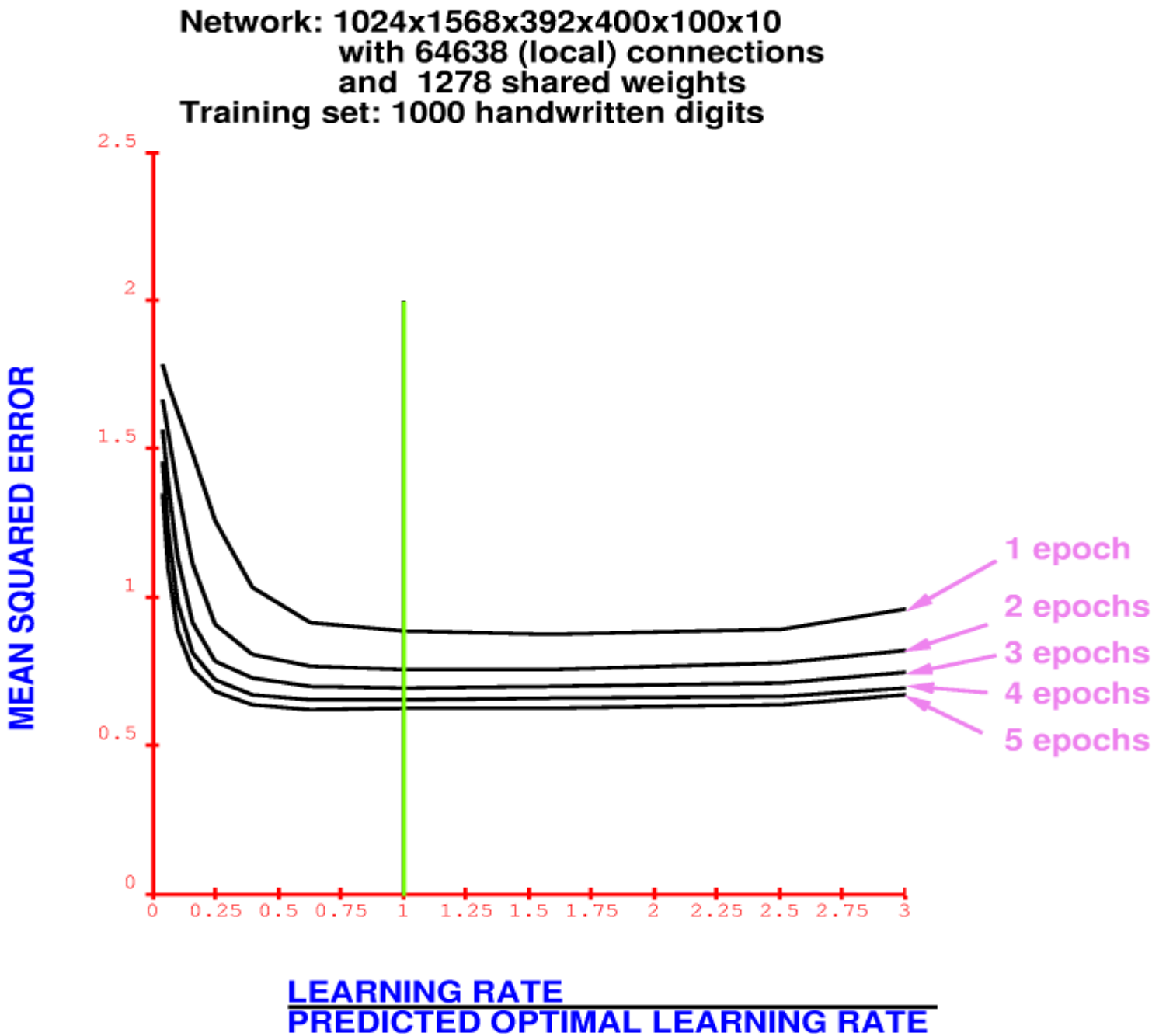
Recipe to compute the maximum learning rate for SGD



Recipe to compute the maximum learning rate for SGD



Recipe to compute the maximum learning rate for SGD



Tricks to Make Stochastic Gradient Work

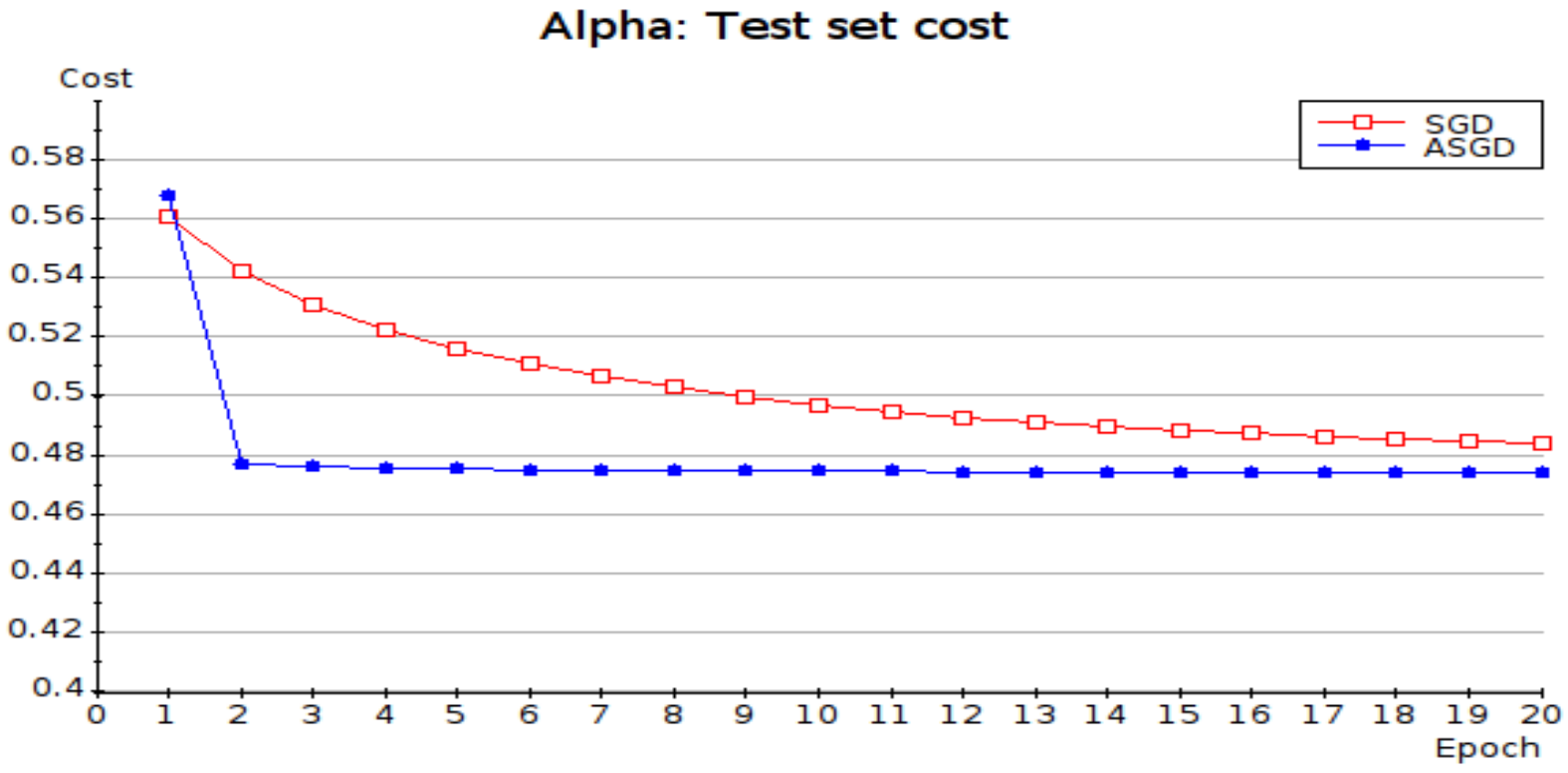
- **Best reads:**
  - Yann LeCun, Léon Bottou, Genevieve B. Orr and Klaus-Robert Müller: Efficient Backprop, Neural Networks, Tricks of the Trade, Lecture Notes in Computer Science LNCS 1524, Springer Verlag, 1998. http://yann.lecun.com/exdb/publis/index.html#lecun-98b
  - Léon Bottou: Stochastic Gradient 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)

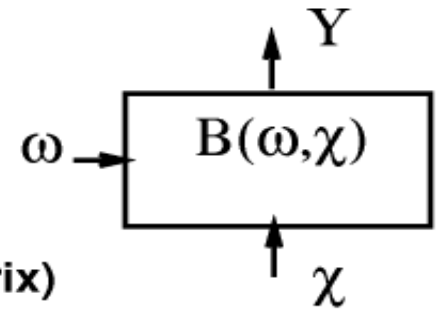
$$\eta_t = \eta_0 / \left(1 + \lambda \eta_0 t\right)^{0.75}$$

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



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

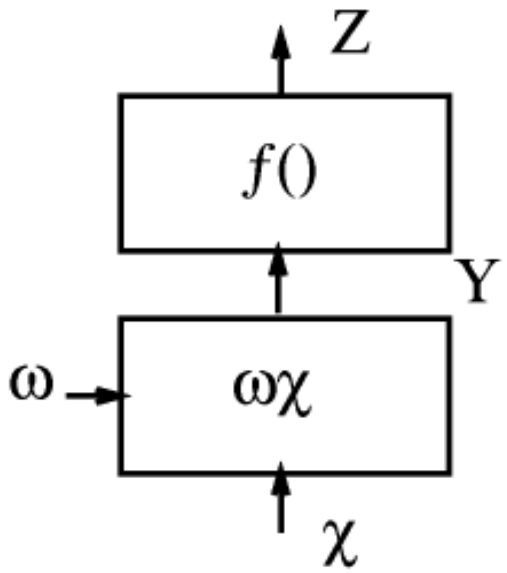
If we are only interested in the diagonal terms, it reduces to:

$$\frac{\partial^2 E}{\partial \omega_{ii}^2} = \sum_k \frac{\partial^2 E}{\partial Y_{kk}^2}\left(\frac{\partial Y_{kk}}{\partial \omega_{ii}}\right)^2$$ (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

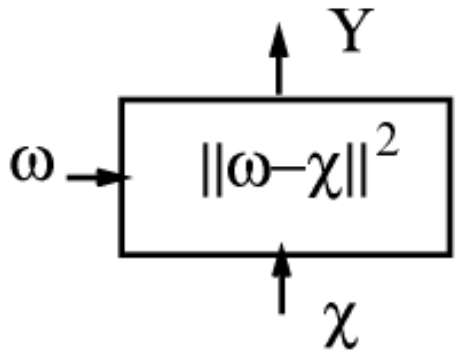
$$\frac{\partial^2 E}{\partial \omega_{ki}^2} = \frac{\partial^2 E}{\partial Y_k^2} \chi_i^2$$

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

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