Stochastic Learning

Léon Bottou
Email: `leon@bottou.org`

Summary

1- **Framework, Examples**
- Framework
- Examples
- Discussion
- Illustration

2- Backprop Cookbook
3- Convergence
4- Optimal Learning

Introduction.

Online learning is not a new idea.

Early Learning Systems:
(Hebb-49, Rosenblatt-57, Widrow-Hoff-60,
Amari-67, Tsypkin-71, Kohonen-82)

Recursive Adaptive Algorithms:
(Robbins-Monro-51, Kushner-Clark-78, Widrow-Stearns-85,
Ljung-Sorderstrom-83, Benveniste-Metivier-Priouret-90.

Expected Risk, Empirical Risk.

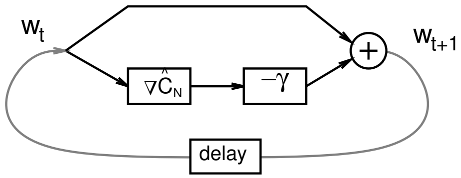
Expected Risk

$$C(w) \triangleq \mathbf{E}_z\, L(z,w) \triangleq \int L(z,w)\; dP(z)$$

Empirical Risk

$$\hat{C}_N(w) \triangleq \frac{1}{N}\sum_{i=1}^{N} L(z_i,w)$$

Batch Gradient Descent.



Repeat:

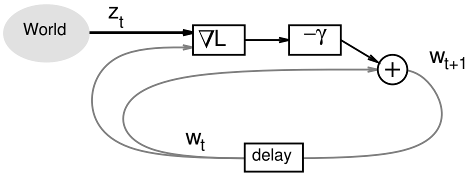
- Compute gradient:

$$\nabla_w \widehat{C}_N(w_t) = \frac{1}{N} \sum_{i=1}^{N} \nabla_w L(z_n, w_t)$$

- Update:

$$w_{t+1} = w_t - \gamma_t \nabla_w \widehat{C}_N(w_t)$$

Stochastic Gradient Descent.



Repeat:

- Pick random example z_t.
- Update:

$$w_{t+1} = w_t - \gamma_t \nabla_w L(z_t, w_t)$$

Remark:

- Optimize $\mathbf{E}_z\, L(z, w)$ (when everything goes right. . .)

Stochastic vs. Online.

Stochastic

Examples drawn from a finite training set.
Often requires several epochs. . .

Online

Keep learning after deployment.
Examples provided on-the-fly.

The same mathematics apply to both cases.
I rarely make the distinction.

Stochastic/Online vs. Generalization.

Stochastic/Online Gradient optimizes $\mathbf{E}_z\, L(z, w)$
for whatever distribution the z_t are drawn from.

Assume the z_t are drawn from an infinite pool:
$\Longrightarrow$ Optimize $C(w)$ instead of $\widehat{C}_N(w)$.

Same mathematics again.

Are we bypassing Vapnik's theorems ? . . .

More General.

To minimize cost $C(w)$:

Repeat:
1- Draw something random: $\mathbf{z}_t$.
2- Update:

$$w_{t+1} \;=\; w_t - \gamma_t J(\mathbf{z}_t, w_t)$$

Requirement:

$$\mathbf{E}_{\mathbf{z}}\, J(\mathbf{z}, w) \;=\; \nabla_w\, C(w)$$

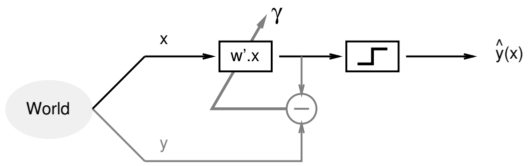
Examples.

Adaline.

Model: Linear + threshold.

Loss:

$$L_{\text{Adaline}}(x,y,w) = (y - wx)^2 \qquad x \in \mathcal{R}^n,\; y \in \{-1,+1\}$$

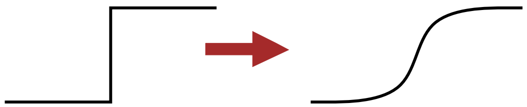


Update Rule:

$$w_{t+1} = w_t + \gamma_t (y - w_t x) x$$

Widrow-Hoff 1960

Multilayer network.



Loss:

$$L_{\text{MLP}}(x,y,w) = (y - f(x,w))^2$$

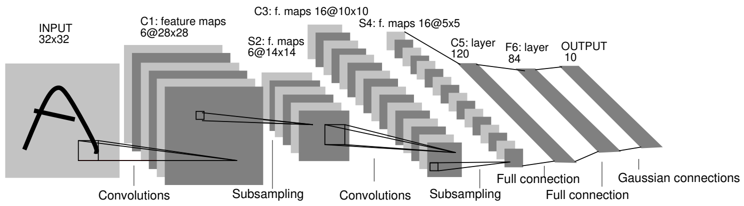
Update Rule:

Use back-propagation to compute $\nabla_w f(x, w_t)$.

$$w_{t+1} = w_t + \gamma_t (y - f(x,w))\, \nabla_w f(x, w_t)$$

Multilayer network.

Example:



- Convolutional layers
- Subsampling layers
- Gaussian layers (i.e. RBFs.)

LeCun et al. 1990

Non Differentiable Loss Functions.

What if $L(z, w)$ has a few non differentiable points?

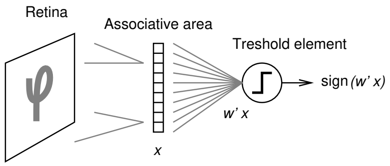
Just pick another (better) example!

$$\nabla_w C(w) = \nabla_w \int L(z, w)\, dp(z) \stackrel{?}{=} \int \nabla_w L(z, w)\, dp(z)$$

Bounded convergence theorem $\Rightarrow$ Sufficient condition.

$$\exists \Phi, \forall z,\ \forall v \in \vartheta(w),\quad |Q(z, v) - Q(z, w)| \ \le\ |w - v|\, \Phi(z, w)$$

Rosenblatt's perceptron.



Loss:

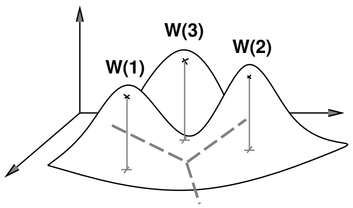
$$L_{\text{Perceptron}}(x, y, w) = (\text{sign}(wx) - y)\, wx \qquad y = \pm 1$$

Update Rule:

$$w_{t+1} = w_t + \begin{cases} 0 & \text{if correct} \\ \gamma_t\, y\, x_t & \text{if mistake} \end{cases}$$

Rosenblatt 1957

K-Means.



$$L_{\text{KMeans}}(x, w) = \min_{k=1}^{K} (x - w(k))^2$$

w^- : closest centroid.

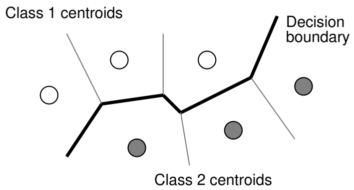
Update:

$$w^-_{t+1} = w^-_t + \gamma_t (x_t - w^-_t)$$

Note: Learning rate $\gamma_t = 1/t$ yields optimal algorithm.

MacQueen 1967

Learning Vector Quantization.



Kohonen 1982, 1988

w^- : closest centroid.
w^+ : closest centroid w/correct class.

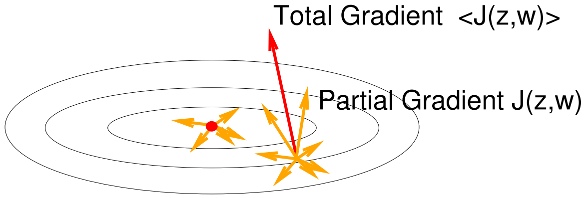
$$L_{\text{LVQ}}(x, y, w) = \begin{cases} 0 & \text{if correct} \\ \frac{(x-w^+)^2 - (x-w^-)^2}{\delta (x-w^-)^2} & \text{if } (x - w^+)^2 < (1+\delta)(x - w^-)^2 \\ 1 & \text{otherwise} \end{cases}$$

Update:

$$\text{if } \begin{cases} x \text{ is misclassified} \\ \text{and } \ (x - w^+)^2 < (1+\delta)(x - w^-)^2 \end{cases}$$

$$\text{then } \begin{cases} w^-_{t+1} = w^-_t - \gamma_t k_1 (x - w^-_t) \\ w^+_{t+1} = w^+_t + \gamma_t k_2 (x - w^+_t) \end{cases}$$

Stochastic noise.



Does stochastic gradient converge? Where?

Residual noise proportional to learning rate.

$$|\theta_t - \theta^*|^2 \equiv \frac{1}{t} \qquad (\ldots \text{at best}\ldots)$$

Is stochastic gradient slow?

Speed Advantage.

Assume training set contains
10 copies of the 100 same examples.

- Batch — Blindly computes redundant gradients.
  1 epoch on large set $\equiv$ 1 epochs on small set.
- Online — Take advantage of redundancy.
  1 epoch on large set $\equiv$ 10 epochs on small set.

In practice, stochastic gradient
can be orders of magnitude faster.

Many examples.

Learning N examples.
Fixed capacity.

	Memory	Time
Stochastic (optimal)	$\mathcal{O}(1)$	$\mathcal{O}(N)$
Batch (superlinear)	$\mathcal{O}(N)$	$\mathcal{O}(N \log\log N)$
SVM (QP)	$\mathcal{O}\left(N^{2?}\right)$	$\mathcal{O}\left(N^{3?}\right)$

Beware the constants...

Other approaches.

- Mistake bounds (e.g. Novikoff, Littlestone)
- Statistical physics (e.g. Solla, Saad, Caticha)
- Bayesian (e.g. Opper)

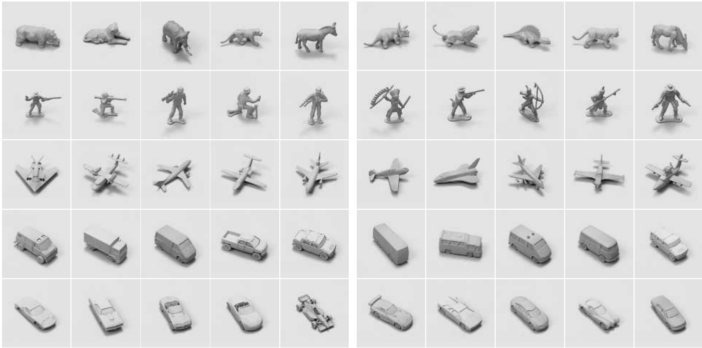
NORB data

- 97200 stereo image pairs
- 50 objects (5 classes, 10 instance per class)
- 9 azimuths
- 36 orientations
- 6 lighting conditions

179 GB of raw data.

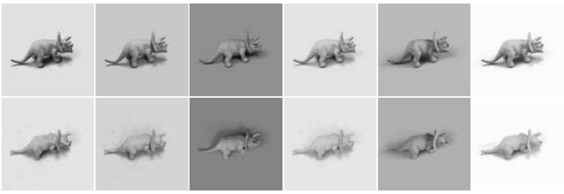
NORB data

Objects:

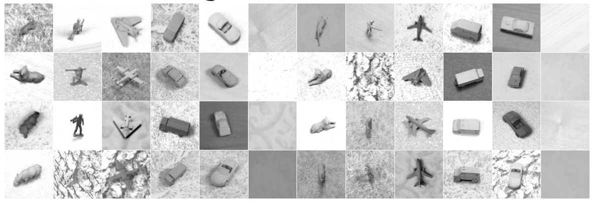


NORB data

Lighting:



Textured Background:



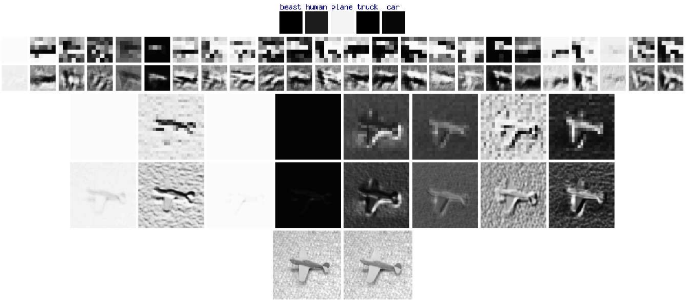
NORB experiments

Norb24K:
- 24300 train + 24300 test
- stereo pairs 2x96x96
- every 20 degrees
- uniform background

Norb243K:
- 243000 train + 24300 test
- random perturbations

Norb291K:
- 291600 train + 29160 test
- random perturbations
- textured background.

Convolutional Network.



Stochastic gradient.
Diagonal Levenberg-Marquardt.

Norb24K Results

Classifier	Training Set	Test Set	Input	Test Error
Linear Classifer	Norb24K	Norb24K	raw 2x96x96	30.2%
K-NN (K=1)	Norb24K	Norb24K	raw 2x96x96	18.4%
K-NN (K=1)	Norb24K	Norb24K	PCA 95	16.6%
SVM Gaussian	Norb24K	Norb24K	raw 2x96x96	...
SVM Gaussian	Norb24K	Norb24K	raw 1x48x48	13.9%
SVM Gaussian	Norb24K	Norb24K	PCA 95	13.3%
Convo. Net 80	Norb24K	Norb24K	raw 2x96x96	6.6%
SVM Gaussian	Norb24K	Norb243K	raw 1x48x48	34.4%
Convo. Net 80	Norb24K	Norb243K	raw 2x96x96	15.7%

Norb243K and Norb291K Results

Classifier	Training Set	Test Set	Input	Test Error
Linear Classifer	Norb243K	Norb243K	raw 2x96x96	30.6%
k-NN (K=1)	Norb243K	Norb243K	raw 2x96x96	...
SVM Gaussian	Norb243K	Norb243K	raw 2x96x96	...
SVM Gaussian	Norb243K	Norb243K	raw 1x32x32	...
Convo. Net 100	Norb243K	Norb243K	raw 2x96x96	7.1%
Convo. Net 100	Norb291K	Norb291K	raw 2x96x96	10.6%

Links

Léon Bottou (1998) **Online Algorithms and Stochastic Approximations**, In Saad, D., editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK.

Leon Bottou, Noboru Murata (2002) **Stochastic approximations and efficient learning**. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks, second edition*. The MIT Press, Cambridge, MA, 2002.

Stochastic Learning

Léon Bottou
Email: leon@bottou.org

Summary

1- Framework, Examples
2- **Backprop Cookbook**
3- Convergence
4- Optimal Learning.

Introduction:

Old slides (1994), but good review of the practical difficulty of running large scale stochastic learning algorithms.

Link:

Yann Le Cun, Léon Bottou, Genevieve Orr, Klaus-Robert Müller: **Efficient Backprop**, In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science 1524. Springer Verlag.

OVERVIEW

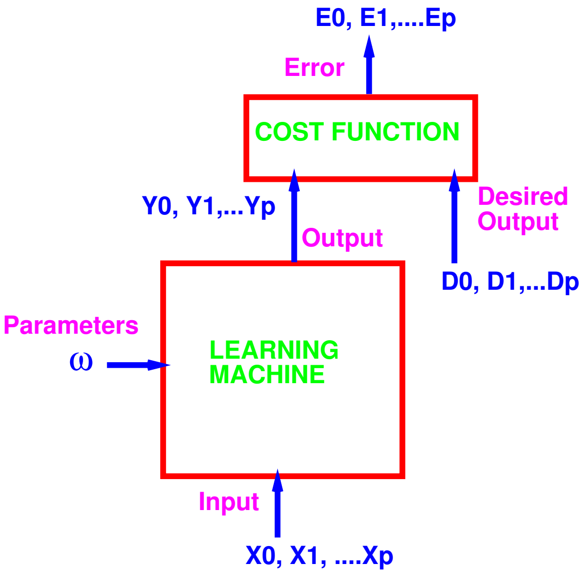
- 1 – Plain Backprop: how to make it work
  - – Basic concepts, terminology and notation
  - – Intuitive analysis and practical tricks
- 2– The convergence of gradient descent
  - – A little theory
    - – Quadratic forms, Hessians, and Eigenvalues
    - – maximum learning rate, minimum learning time
  - – How GD works in simple cases
    - – a single 2–input neuron
    - – stochastic vs batch update
  - – Transformation laws
    - – shifting, scaling, and rotating the input
    - – the non–invariance of GD
  - – The minimal multilayer network
- 3 – Second order methods
  - – Newton's algorithm, and why it does not work.
    - – parameter space transformations
  - – computing the second derivative information
    - – diagonal terms, quasi–linear hessians, partial hessians
  - – analysis of multilayer net Hessians
  - – Classical optimization methods
    - – Conjugate Gradient methods
    - – Quasi Newton methods: BFGS et al.
    - – Levenberg–Marquardt methods
- 4 – Applying second order methods to multilayer nets
  - – (non)applicability of 2nd order techniques to backprop
  - – Mini batch methods
  - – An on–line diagonal Levenberg–Marquardt method
  - – computing the maximum learning rate and the principal eigenvalues

1

PLAIN BACKPROP:

HOW TO MAKE IT WORK

BASIC CONCEPTS, TERMINOLOGY, NOTATIONS

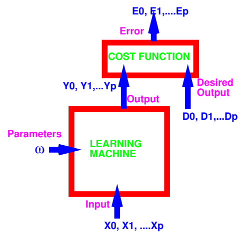


Average Error: $E = \frac{1}{p} \sum E\kappa$

GRADIENT DESCENT LEARNING

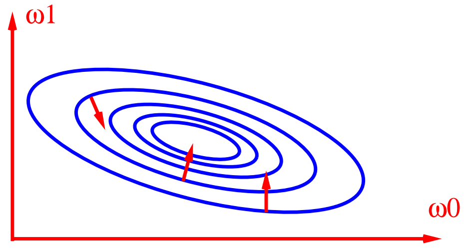
Average Error:

$$E(\omega) = \frac{1}{p} \sum E\kappa(\omega)$$

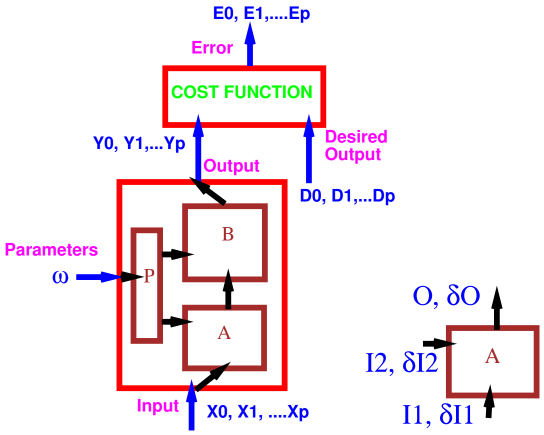


Gradient Descent:

$$\omega(\tau+1) = \omega(\tau) - \eta \frac{\partial E}{\partial \omega}$$



COMPUTING THE GRADIENT WITH BACKPROPAGATION



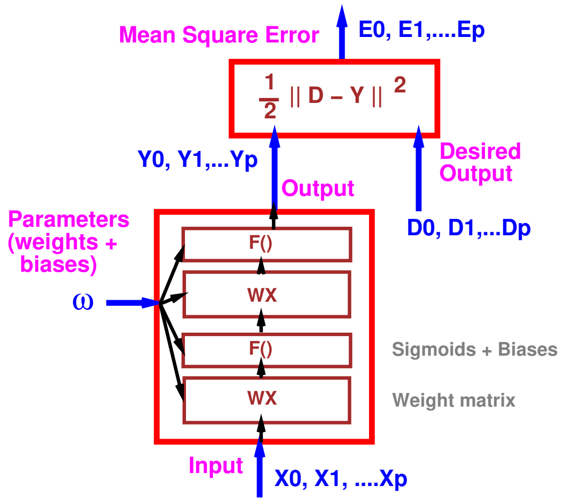
– The learning machine is composed of modules (e.g. layers)
– Each module can do two things:
  1– compute its outputs from its inputs (FPROP)

$$O = A(I1, I2)$$

  2– compute gradient vectors at its inputs from gradient vectors at its outputs (BPROP)

$$\delta I1 = \frac{\partial A}{\partial I1}\ \delta O \qquad \delta I2 = \frac{\partial A}{\partial I2}\ \delta O$$

AN INTERESTING SPECIAL CASE: MULTILAYER NETWORKS



Weight matrices:

FPROP $O = \omega\, I$

BPROP $\delta I = \omega'\, \delta O\ ;\quad \delta\omega = \delta O'\, I$

Sigmoids + Bias:

FPROP $O = f(I+B)$

BPROP $\delta I = f'(I+B)\, \delta O\ ;\quad \delta B = \delta I$

STOCHASTIC UPDATE
BATCH UPDATE

STOCHASTIC GRADIENT

$$\omega(\tau+1) = \omega(\tau) - \eta \frac{\partial E\tau}{\partial \omega}$$

Repeat {
 for all examples in training set {
  forward prop // compute output
  backward prop // compute gradients
  update parameters }}

The parameters are updated after each presentation of an example

FULL GRADIENT

$$\omega(\rho+1) = \omega(\rho) - \eta \frac{\partial E}{\partial \omega}$$

Repeat {
 for all examples in training set {
  forward prop // compute output
  backward prop // compute gradients
  accumulate gradient }
 update parameters }

The gradients are accumulated over the whole training set before a parameter update is performed

A FEW PRACTICAL TRICKS

BackProp is a simple algorithm, but convergence can take ages if it is not used properly.

The error surface of a multilayer network is non quadratic and non−convex, and has often many dimensions.
THERE IS NO MIRACLE TECHNIQUE for finding the minimum. Heuristics (tricks) must be used.

Depending on the details of the implementation, the convergence time can vary by orders of magnitude, especially on small problems.

On large, real−world problems, the convergence time is usually much better than one would expect from extrapolating the results on small problems.

Here is a list of some common traps, and some ideas about how to avoid them.

The theoretical justifications for many of these tricks will be given later in the talk.

STOCHASTIC vs BATCH UPDATE

Stochastic update is usually MUCH faster than Batch update. Especially on large, redundant data sets.
Here is why:

Imagine you are given a training set with 1000 examples.
Imagine this training set is in fact composed of 10 copies of a set of 100 patterns

- BATCH: the computation for one update will be 10 times larger then necessary.
- STOCHASTIC: the redundancy in the training set will be taken advantage of. One epoch on the large set will be like 10 epochs on the smaller set.

Batch will be AT LEAST 10 times slower than Stochastic

In real life, repetitions rarely occur, but very often the training examples are highly redundant (many patterns are similar to one another), which has the same effect.

In practice speed differences of orders of magnitude between Batch and Stochastic are not uncommon

small batches can be used without penalty, provided the patterns in a minibatch are not too similar.

STOCHASTIC vs BATCH UPDATE (continued)

STOCHASTIC:

ADVANTAGES:
- much faster convergence on large redundant datasets
- stochastic trajectory allows escaping from local minima

DISADVANTAGES:
- keeps bouncing around unless the learning rate is reduced
- theoretical conditions for convergence are not as clear as for batch
- convergence proofs are probabilistic
- most nice acceleration tricks or second−order methods do not work with stochastic gradient
- it is harder to parallelize than batch

BATCH:

ADVANTAGES:
- guaranteed convergence to a local minimum under simple conditions.
- lots of tricks and second order methods to accelerate it
- easy convergence proofs

DISADVANTAGES:
- painfully slow on large problems

Despite the long list of disadvantages for STOCHASTIC, that is what most people use (and rightfully so, at least on large problems).

SHUFFLING THE EXAMPLES

RULE: at any time, chose the training example with the maximum information content.

For example:
- the one with the largest error
- the one that is maximally different from its predecessors

A SIMPLE TRICK:

(applicable to stochastic gradient on classification tasks)

Shuffle the training set so that successive examples never (or rarely) belong to the same class.

A MORE REFINED TRICK:

use an "emphasizing" scheme:
show difficult patterns more often than easy patterns

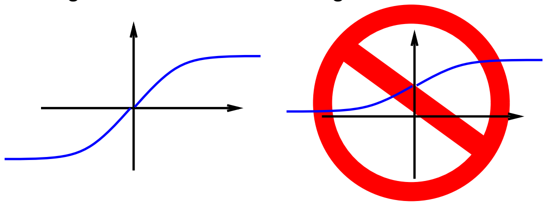
[whether a pattern is easy of hard can be determined with the error it produced during the previous iterations]

Problem with emphasizing techniques:

- they perturb the distribution of inputs
- the presence of outliers or of mislabeled examples can be catastrophic.

THE SIGMOID

Symetric sigmoids (like tanh) often yield faster convergence than the standard logistic function.



MORE GENERALLY: the mean of each input to a neuron should be small compared to its standard deviation [more on this later]

Symetric sigmoids are more likely to produce "small mean" signals than are positive sigmoids.

Sigmoids (and their derivatives) can be efficiently computed as ratios of polynomials

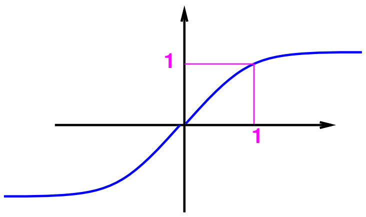
Problems with symetric sigmoids:

- The error surface is VERY FLAT around the origin. (the origin is a saddle point which is attractive in almost all directions)
- Avoid small weights

THE SIGMOID (continued)

The one I use: a rational approximation to

$$f(x) = 1.7159 \tanh\left(\frac{2}{3}x\right)$$



Properties:
- $f(1) = 1$, $f(-1) = -1$
- 2nd derivative is maximum at $x=1$
- the effective gain is close to 1

The precise choice of the sigmoid is almost irrevelant, but some choices are more convenient than others

It is sometimes helpful to add a small linear term to avoid flat spots, e.g.

$$f(x) = \tanh(x) + ax$$

NORMALIZING THE INPUTS

Each input input variable should be sifted so that its mean (averaged over the training set) is close to 0 (or is small compared to its standard deviation).

Here is why:

Consider the extreme case where the input variables are always positive.

The weights of neuron in the first hidden layer can only increase together or decrease together (for a given input pattern the gradients all have the same sign).

This means that if the weight vector has to change its direction, it will have to do it by zigzaging (read: SLOW).

Shifts of the input variables to a neuron introduce a preferred direction for weight changes, which slows down the learning.

This is also why we prefer symetric sigmoids: what is true for input units is also true for other units in the network.

NORMALIZING THE INPUTS (continued)

The speed at which the output of a particular weight varies with gradient descent is proportional to the COVARIANCE of its input. [more on this later]

covariance: $\frac{1}{P}\sum_{\kappa}\chi_{\kappa}^{2}$

To equalize the learning speeds of input weights, the input variables should be scaled to have approximately equal covariances.

To equalize the learning speeds of these weights with that of the weights in the next layers, this covariance should be comparable to the expected covariances of the hidden units states (around 1 with the type of sigmoid proposed earlier).

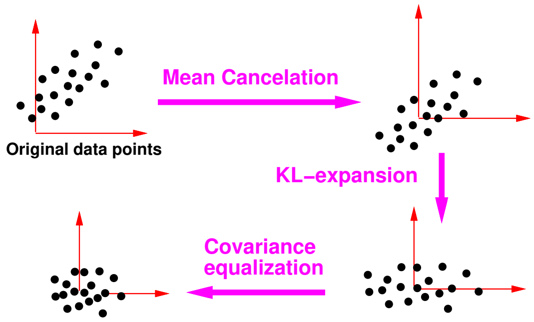
An exception to this rule is when some inputs are known to be of lesser significance than others. Scaling them down makes them less "visible" to the learning process.

NORMALIZING THE INPUTS (continued)

Input variables should be UNCORRELATED if possible

Correlations between input variables also introduce "preferred directions for weight changes"

Decorrelation can be performed by a Principal Component Analysis (Karhunen–Loeve expansion).



Sometimes the input have a particular meaning that would be destroyed by the KL–expansion (e.g.: if the input variables are the pixels of an image and the network uses local connections)

CHOOSING THE TARGET VALUES

Avoid saturating the output units. Choose target values within the range of the sigmoid

In classification applications, the desired outputs are often binary.

Common sense would suggest to set the target values on the asymptotes of the sigmoid.

However this has several adverse effects:

1 – this tends to drive the output weights to infinity (and to saturate the hidden units as well). When a training example happens not to saturate the outputs (say an outlier), it will produce ENORMOUS gradients (due to the large weights).

2 – outputs will tend to be binary EVEN WHEN THEY ARE WRONG. This means that a mistake will be difficult to correct, and that the output levels cannot be used as reliable confidence factors.

Saturating the units erases the differences between typical and non–typical examples.

Setting the target values at the point of maximum second derivative on the sigmoid (–1 and +1 for the sigmoid proposed earlier) is the best way to take advantage of the non linearity without saturating the units

INITIALIZING THE WEIGHTS

Initialize the weights so that the expected standard deviation of the weighted sums is at the transition between the linear part and the saturated part of the sigmoid function

Large initial weights saturate the units, leading to small gradients and slow learning. Small weights correspond to a very flat area of the error surface (especially with symetric sigmoids)

Assuming the sigmoid proposed earlier is used, the expected standard deviation of the inputs to a unit is around 1, and the desired standard deviation of its weighted sum is also around 1.

Assuming the inputs are independent, the expected standard deviation of the weighted sum is

$$\sigma = \left(\sum_i \omega_i^2\right)^{1/2} = \phi^{1/2}\varpi$$

where ϕ is the number of input to the unit, and ϖ is the standard deviation of its incoming weights.

To ensure that σ is close to 1, the weights to a unit can be drawn from a distribution with standard deviation

$$\varpi = \phi^{-1/2}$$

CHOOSING LEARNING RATES

Equalize the learning speeds.

Each weight (or parameter) should have its own learning rate. η

Some weights may require a small learning rate to avoid divergence, while others may require a large learning rate to converge at reasonable speed.

Because of possible correlations between input variables, the learning rate of a unit should be inversely proportional to the square root of the number of inputs to the unit.

If shared weights are used (as in TDNNs and convolutional networks), the learning rate of a weight should be inversely proportional to the square root of the number of connection sharing that weight.

Learning rates in the lower layers should generally be larger than that in the higher layers.

The rationale for many of these rules of thumb will become clearer later.
Several techniques are available to reduce "learning rate fiddling".

NEURONS AND WEIGHTS

Although most systems use neurons based on dot products and sigmoids, many other types of units (or layers) can be used.

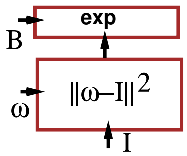
A particularly interesting example is when the dot product of the input by the weight vector is replaced by a Euclidean distance, and the sigmoid by an exponential (Gaussian units of RBF).

These units can replace (or coexist with) standard units, and they can be trained with gradient descent:

exponential + Bias:

FPROP $O = \exp(I+B)$

BPROP $\delta I = O\, \delta O\ ;\ \ \delta B = \delta I$



weight vector:

FPROP $O = (\omega - I)'(\omega - I)$

BPROP $\delta I = 2\, \delta O'(I - \omega)\ ;$
$\delta\omega = -\delta I$

PROS and CONS

- Locality: each unit is only affected by a small part of the input space. This can be good (for learning speed) and bad (a lot of RBF are needed to cover high dimensional spaces)
- gradient descent learning may fail if the RBFs are not properly initialized (using clustering techniques e.g. K–means). There are LOTS of local minima.
- RBFs are more apropriate in higher layers, and sigmoids in lower layers (higher dimension).

MORE STANDARD TRICKS

- Momentum
  - Increases speed in batch mode. seems marginally useful but not indispensable in stochastic mode.

- Adaptive learning rates:
  - a separate learning rate for each weight is increased if the gradient is steady, decreased if the gradient changes sign often [Jacobs 88]. This only works with BATCH.
  - a global learning rate is adjusted using line searches. Again, this only works for BATCH.

2

THE CONVERGENCE OF GRADIENT DESCENT

A LITTLE THEORY

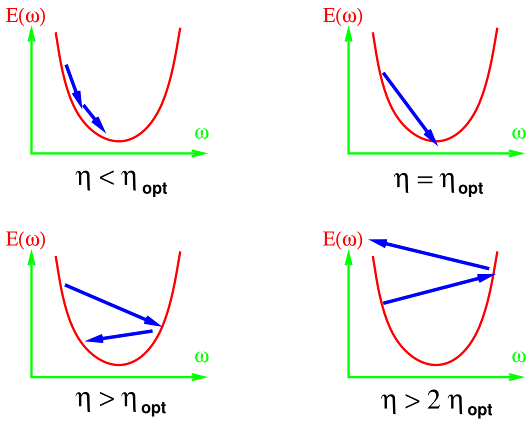
Gradient Descent in one dimension

$$\omega \leftarrow \omega - \eta \frac{\partial E}{\partial \omega}$$

gradient of objective function

weight vector

learning rate



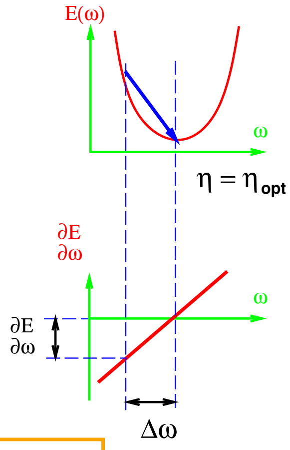
OPTIMAL LEARNING RATE IN 1D

Weight change:

$$\Delta\omega = \eta \frac{\partial E}{\partial \omega}$$

Assuming E is quadratic:

$$\frac{\partial^2 E}{\partial \omega^2}\, \Delta\omega = \frac{\partial E}{\partial \omega}$$



Optimal Learning Rate

$$\boxed{\eta_{opt} = \left(\frac{\partial^2 E}{\partial \omega^2}\right)^{-1}}$$

Maximum Learning Rate

$$\boxed{\eta_{max} = 2\,\eta_{opt}}$$

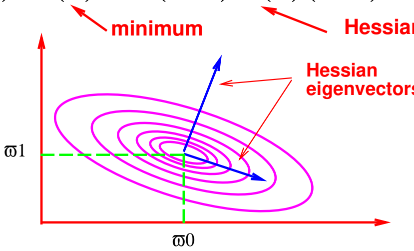
CONVERGENCE OF GRADIENT DESCENT

Local quadratic approximation of the cost function around a minimum:

$$E(\omega) \approx E(\varpi) + 1/2(\omega - \varpi)'\, H(\varpi)\, (\omega - \varpi)$$

minimum

Hessian



HESSIAN
Second derivative matrix

$$H_{ij} = \frac{\partial^2 E}{\partial \omega_i \partial \omega_j}$$

The Hessian is a symetric NxN matrix

Gradient Descent weight update:

$$\omega_{\kappa+1} = \omega_\kappa - \eta \frac{\partial E}{\partial \omega} = \omega_\kappa - \eta\, H(\omega_\kappa)\, (\omega_\kappa - \varpi)$$

$$(\omega_{\kappa+1} - \varpi) = (I - \eta H(\omega_\kappa))(\omega_\kappa - \varpi)$$

Convergence <====> **if the prefactor of the right handside shrinks any vector**

CONVERGENCE OF GRADIENT DESCENT (continued)

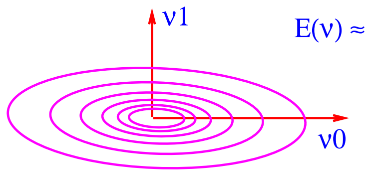
Let Θ be the rotation matrix that make H diagonal:

$$\Theta H \Theta' = \Lambda\,; \qquad \Theta'\Theta = I$$

$$E(\omega) \approx E(\varpi) + 1/2[(\omega - \varpi)'\Theta']\,[\Theta H(\varpi)\, \Theta']\,[\Theta(\omega - \varpi)]$$

Now denote: $\nu = \Theta\,(\omega - \varpi)$

$$E(\nu) \approx E(0) + 1/2\, \nu' \Lambda\, \nu$$



The eigenvectors of a diagonal matrix are the coordinate axes

Gradient update in the transformed space:

$$\nu_{\kappa+1} = (I - \eta\, \Lambda)\, \nu_\kappa$$

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

CONVERGENCE SPEED OF GRADIENT DESCENT

The maximum learning rate to ensure convergence is

$$\eta_{max} = 2/\lambda_{max}$$

The one that yields the fastest convergence in the direction of highest curvature is

$$\eta_{opt} = 1/\lambda_{max}$$

With this choice, the convergence time will be determined by the directions of SMALL eigenvalues (they will be the slowest to converge).

The convergence time is proportional to:

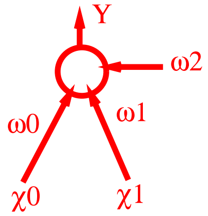
$$\frac{1}{\eta\,\lambda_{min}} > \frac{\lambda_{max}}{2\lambda_{min}}$$

where λ_{min} is the smallest "non-negligible" eigenvalue

The convergence time is proportional to the ratio of the largest eigenvalue to smallest "non-negligible" eigenvalue of the Hessian

CONVERGENCE OF GRADIENT DESCENT A SIMPLE EXAMPLE

A single 2-input linear neuron:



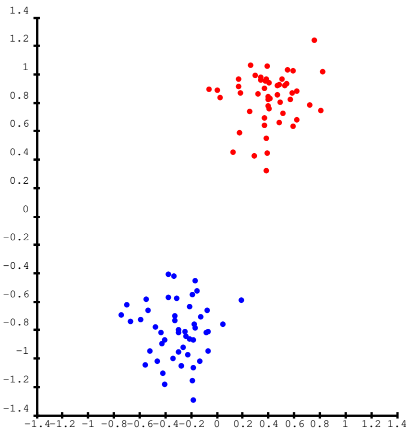
$$E(\omega) = \frac{1}{2P}\sum_p \| d_p - y_p(\omega) \|^2 = \frac{1}{2P}\sum_p \| d_p - \omega'\chi_p \|^2$$

$$E(\omega) = \frac{1}{2P}\left[\sum_p d^2 - 2\left(\sum_p d_p \chi_p\right)'\omega + \omega'\left(\sum_p \chi_p \chi'_p\right)\omega \right]$$

$$H = \frac{1}{P}\sum_p \chi_p \chi'_p$$

The Hessian of a single linear neuron is the covariance matrix of the inputs

Dataset #1



Examples of each class are drawn from a Gaussian distribution centered at (−0.4, −0.8), and (0.4, 0.8).

Eigenvalues of covariance matrix: 0.83 and 0.036

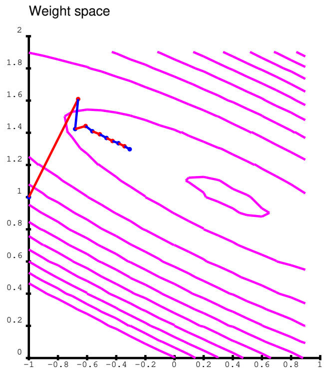
Batch gradient descent

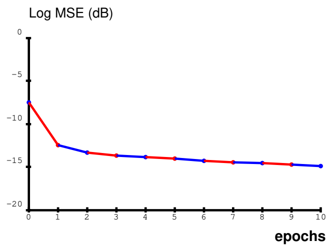
data set: set−1 (100 examples, 2 gaussians)
network: 1 linear unit, 2 inputs, 1 output. 2 weights, 1 bias.

Learning rate: $\eta = 1.5$

Hessian largest eigenvalue: $\lambda_{max} = 0.84$

Maximum admissible Learning rate: $\eta_{max} = 2.38$





Batch gradient descent

data set: set–1 (100 examples, 2 gaussians)
network: 1 linear unit, 2 inputs, 1 output.
2 weights, 1 bias.

Learning rate:

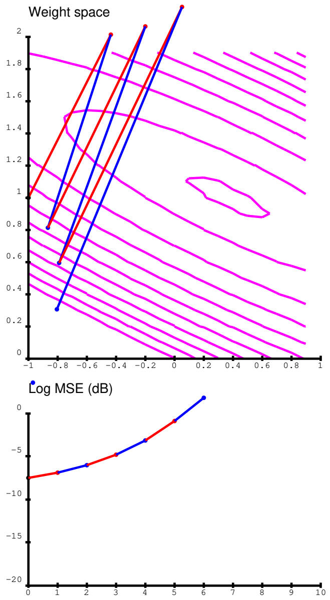
$\eta = 2.5$

Hessian largest eigenvalue:

$\lambda_{max} = 0.84$

Maximum admissible Learning rate:

$\eta_{max} = 2.38$



Stochastic gradient descent

data set: set–1 (100 examples, 2 gaussians)
network: 1 linear unit, 2 inputs, 1 output.
2 weights, 1 bias.

Learning rate:

$\eta = 0.2$

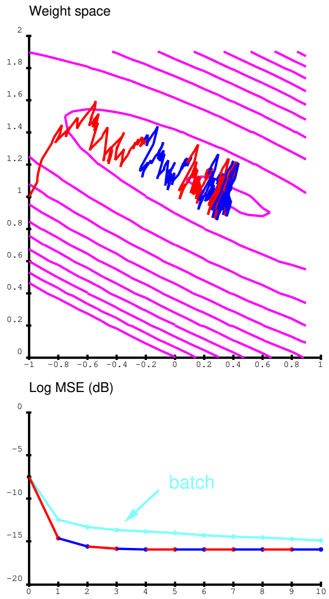
(equivalent to a batch learning rate of 20)

Hessian largest eigenvalue:

$\lambda_{max} = 0.84$

Maximum admissible Learning rate (for batch):

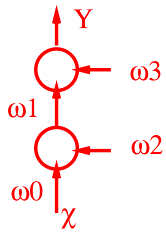
$\eta_{max} = 2.38$



CONVERGENCE OF GRADIENT DESCENT: MINIMAL MULTILAYER NETWORK

1 input
1 hidden unit
1 output

2 weights
2 biases



Sigmoid: 1.71 tanh(2/3 x)

Targets: –1 for class 1, +1 for class 2

TRAINING SET: 20 examples.

Class1:
10 examples drawn from a Gaussian distribution with mean –1, and standard deviation 0.4

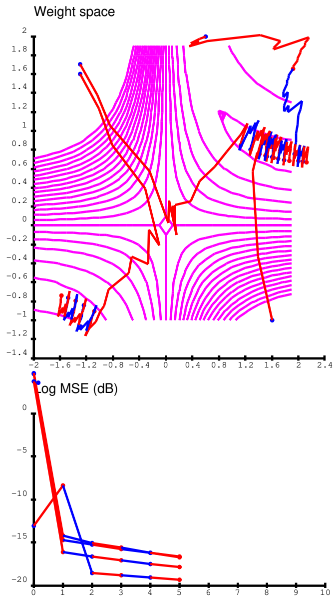
Class2:
10 examples drawn from a Gaussian distribution with mean +1, and standard deviation 0.4

Stochastic gradient: 1–1–1 network

data set: 20 examples, 2 1D–gaussians)
network: 1 input, 1 hidden, 1output
2 weights 2biases

Learning rate:

$\eta = 0.4$



INPUT TRANSFORMATIONS
ERROR SURFACE TRANSFORMATIONS

A shift (non zero mean) in the input variables creates a VERY LARGE eigenvalue which leads to eccentric paraboloids, and to slow convergence

Subtract the means from the input variables

For a single linear neuron, If the input variables have zero means, the eigenvectors of the Hessian are the principal axes of the cloud of training vectors

widely spread variances for the input variables lead to widely spread Hessian eigenvalues

Normalize the variances of the input variables

Correlations between input variables lead to eccentric paraboloid with ROTATED axes

Decorrelate the input variables

The gradient is NOT the best descent direction

Use a spearate learning rate for each weight.
No use spending time and effort to compute an accurate gradient estimate

>>> filename

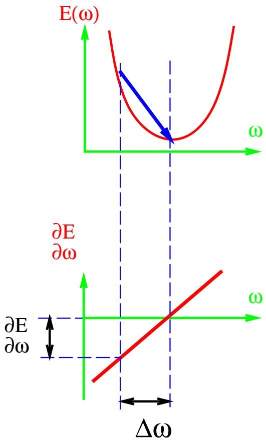
3

SECOND ORDER METHODS

NEWTON ALGORITHM

Newton Algorithm in one dimension

Optimal weight change:



Assuming E is quadratic:

$$\frac{\partial^2 E}{\partial \omega^2}\,\Delta\omega = \frac{\partial E}{\partial \omega}$$

$$\boxed{\Delta\omega = \left(\frac{\partial^2 E}{\partial \omega^2}\right)^{-1} \frac{\partial E}{\partial \omega}}$$

If E is not perfectly quadratic : $\Delta\omega = \eta \left(\frac{\partial^2 E}{\partial \omega^2}\right)^{-1} \frac{\partial E}{\partial \omega} \qquad 0<\eta<1$

NEWTON ALGORITHM

Local quadratic approximation of the cost function around the current point:

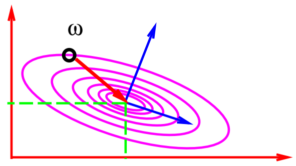
$$E(\omega + \Delta\omega) \approx E(\omega) + \nabla E(\omega)\,\Delta\omega + 1/2\,\Delta\omega' H(\omega)\,\Delta\omega$$

weight change **gradient** **Hessian**

IDEA: find the weight change that minimizes the above.
(i.e.: find the weight change for which the gradient is 0)

Solve: $\nabla E(\omega) + H(\omega)\,\Delta\omega = 0$ **for** $\Delta\omega$

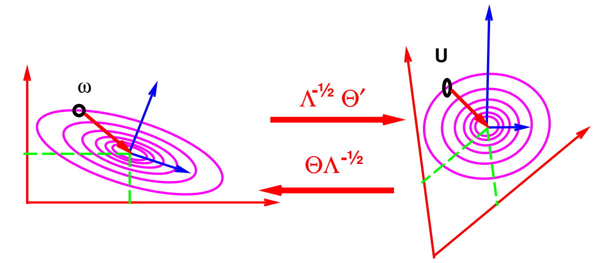
$$\Delta\omega = H(\omega)^{-1}\,\nabla E(\omega)$$



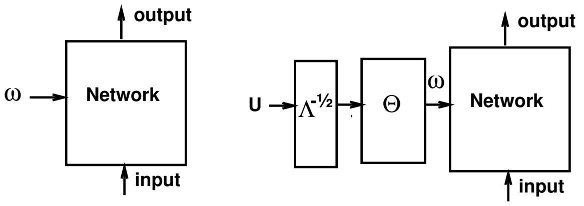
NEWTON ALGORITHM AND PARAMETER SPACE TRANSFORMS

Diagonalized Hessian

$$H(\omega) = \Theta' \Lambda^{1/2} \Lambda^{1/2} \Theta \qquad H^{-1}(\omega) = \Theta \Lambda^{-1/2} \Lambda^{-1/2} \Theta'$$



Newton Algorithm here ....is like Gradient Descent there



NEWTON ALGORITHM

– it converges in 1 iteration if the error is quadratic

– unlike Gradient Descent, it is invariant with respect to linear transformations of the input vectors, i.e. the convergence time is not affected by shifts, scaling and rotations of the input vectors.

BUT:

– it requires computing, storing and inverting the NxN Hessian (or solving an NxN linear system). The complexity is O(N^3), which is impractical with more than a few variables)

– there is ABSOLUTELY NO GUARANTEE of convergence when the error is non–quadratic

– in fact, it diverges if the Hessian has some null or negative eigenvalues (the error surface is flat or curved downward in some directions). The Hessian MUST be Positive Definite. (this is obviously not the case in multilayer nets)

The Newton Algorithm in its original form is unusable for Neural Net learning.

but its basic idea is useful for understanding more sophisticated algorithms

COMPUTING THE HESSIAN INFORMATION IN MULTILAYER NETWORKS

There are many techniques to compute the full Hessian, or parts of it, or approximations to it, in multilayer networks.

We will review the following simple methods:

– finite difference

– square Jacobian approximation (for the Gauss–Newton and Levenberg–Marquardt algorithms)

– computing the diagonal term (or block diagonal terms) by backpropagation

– computing the product of the Hessian by a vector without computing the Hessian

There exist more complex techniques to compute semi–analytically the full Hessian [Bishop 92, Buntine&Weigend 93, others.....] but they are REALLY complicated, and require many forwardprop/backprop passes.

FINITE DIFFERENCE

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

$$\text{(Line k of H)} = \frac{\partial(\nabla E(\omega))}{\partial \omega_K}$$

Finite difference approximation:

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

$$\phi_K = (0,0,0,\ldots\ldots,1,\ldots,0)$$

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.

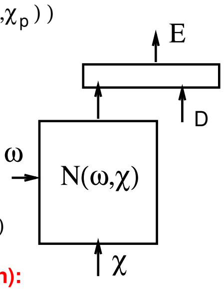
SQUARE JACOBIAN APPROXIMATION FOR GAUSS–NEWTON AND LEVENBERG–MARQUARDT ALGOS.

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:

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



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):

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

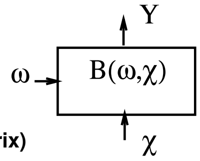
Jacobian: NxO matrix
(O: number of outputs)

– 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:

$$\underbrace{\frac{\partial^2 E}{\partial \omega^2}}_{NxN} = \underbrace{\frac{\partial Y}{\partial \omega}'}_{NxO} \underbrace{\frac{\partial^2 E}{\partial Y^2}}_{OxO} \underbrace{\frac{\partial Y}{\partial \omega}}_{OxN} + \boxed{\underbrace{\frac{\partial E}{\partial Y}}_{1xO} \underbrace{\frac{\partial^2 Y}{\partial \omega^2}}_{OxNxN}}$$

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:

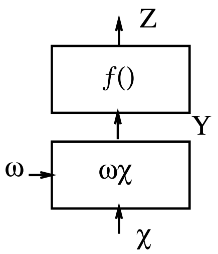
$$\frac{\partial^2 E}{\partial \omega_{ii}^2} = \sum_k \frac{\partial^2 E}{\partial Y_{kk}^2} \left(\frac{\partial Y_{kk}}{\partial \omega_{ii}}\right)^2$$ (and same with χ instead of ω)

BACKPROPAGATING THE DIAGONAL HESSIAN IN NEURAL NETS

(with the square Jacobian approximation)
[LeCun 87, Becker&LeCun 88, LeCun 89]

Sigmoids (and other scalar functions)

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

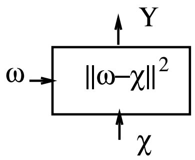


Weighted sums

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

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

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

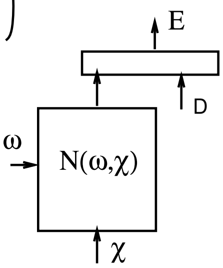
the "OBD" network pruning techniques uses this procedure [LeCun,Denker&Solla 90]

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 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 $\Psi \leftarrow H\Psi / ||\Psi||$

Y

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

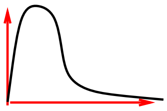
A more accurate method which does not use finite differences (and has the same complexity) has recently been proposed [Pearlmutter 93]

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



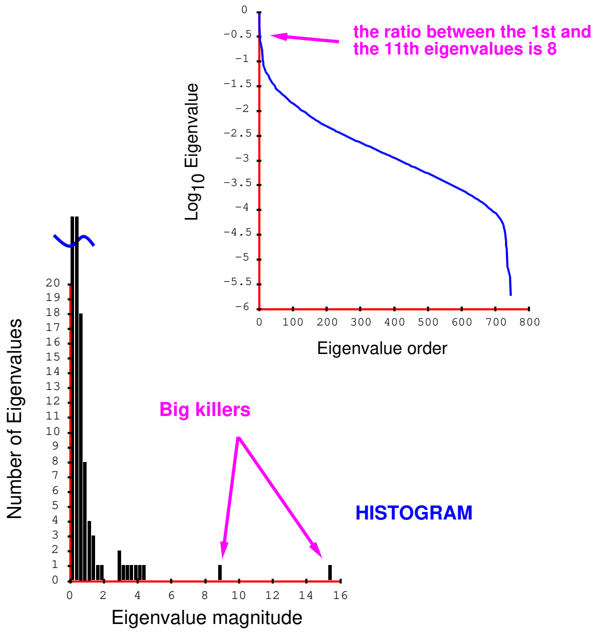
These large ones are the killers

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

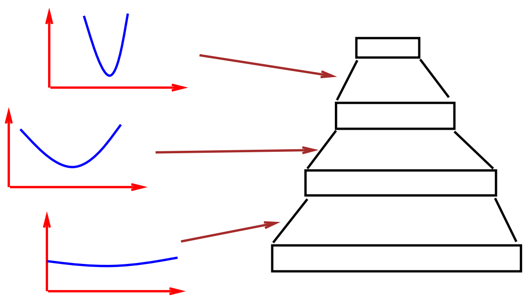
for more details see [LeCun, Simard&Pearlmutter 93] [LeCun, Kanter&Solla 91]

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)

CLASSICAL 2ND ORDER OPTIMIZATION METHODS

– Conjugate Gradient methods
  – O(N) methods
  – do not use the Hessian directly
  – attempts to find descent directions that minimally disturb the result of the previous iterations
  – uses line search. Works only in BATCH.

– Gauss–Newton and Levenberg–Marquardt methods
  – use the square Jacobian approximation
  – works only for mean–square error
  – mainly designed for BATCH
  – O(N^3)

– Quasi–Newton methods (BFGS)
  – iteratively computes an estimate of the inverse Hessian.
  – requires a line search. Works only in BATCH.
  – O(N^2)

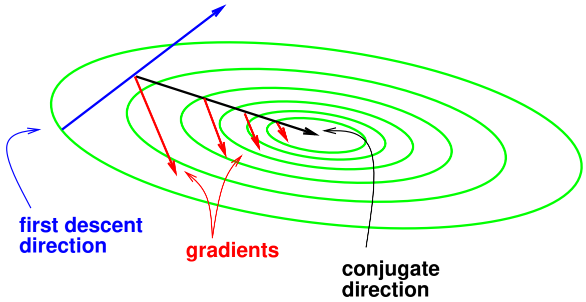
[Dennis&Schnabel 83], [Fletcher 87], [Press et al. 88] [Battiti 92]

CONJUGATE GRADIENT

[Hestenes&Stiefel 52], [Fletcher&Reeves 64]
[Polak 71] (see [Fletcher 87])

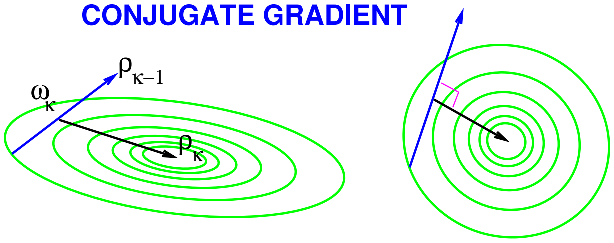
MAIN IDEA: to find a descent direction which does not spoil the result of the previous iterations

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

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

$$\beta_\kappa = \frac{\nabla E(\omega_\kappa)' \nabla E(\omega_\kappa)}{\nabla E(\omega_{\kappa-1})' \nabla E(\omega_{\kappa-1})} \quad \text{(Fletcher−Reeves)}$$

$$\beta_\kappa = \frac{(\nabla E(\omega_\kappa) - \nabla E(\omega_{\kappa-1}))' \nabla E(\omega_\kappa)}{\nabla E(\omega_{\kappa-1})' \nabla E(\omega_{\kappa-1})} \quad \text{(Polak−Ribiere)}$$

– 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

CONJUGATE GRADIENT

– Conjugate gradient is simple and effective. the Polak−Ribiere formula seems to be more robust for non−quadratic functions.

– The conjugate gradient formulae can be viewed as "smart ways" to choose the momentum.

– Conjugate gradient has been used with success in the context of multilayer network training [Kramer&Sangiovani−Vincentelli 88, Barnard&Cole 88, Bengio&Moore 89, Møller 92, Hinton's group in Toronto.....]

– It seems particularly apropriate for moderate size problems with relatively low redundancy in the data.
Typical applications include function approximation, robotic control, times−series prediction, and other real−valued problems (especially if a high accuracy solution is sought).
On large classification problems, stochastic backprop is faster.

– The main drawback of CG is that it is a BATCH method, partly due to its requirement for an accurate line search.
There have been attempts to solve that problem using "mini−batches" [Møller 92].

BFGS and Quasi−Newton methods

[Fletcher 87],[Dennis&Schnabel 83],
[Watrous 88],[Battiti 92].

Quasi−Newton (or secant) methods attempt to keep a positive definite estimate of the INVERSE HESSIAN directly, without requiring matrix inversion, and by only resorting to the gradient information.

They work as follows:

1− pick a positive definite matrix M (say M=I)
2− set search direction $\rho = M \nabla E(\omega)$
3− line search along ρ giving $\omega \leftarrow \omega - \eta\rho$
4− update estimate of inverse Hessian M

Compared to Newton method, Quasi−Newton methods only require the first derivative, they use positive definite approximations to the inverse Hessian (which means they go downhill), and they require O(N^2) operations per iteration.

All of the quasi−Newton methods are BATCH methods

There exist several Quasi−Newton methods, but the most successful is the Broyden−Fletcher−Goldfarb−Shanno (BFGS) method.

BFGS

past parameter vector: $\omega_{\kappa-1}$
present parameter vector: ω_{κ}
past parameter increment: $\delta = \omega_{\kappa} - \omega_{\kappa-1}$
past gradient: $\nabla E(\omega_{\kappa-1})$
present gradient: $\nabla E(\omega_{\kappa})$
past gradient increment: $\varphi = \nabla E(\omega\ ) - \nabla E(\omega_{\kappa-1})$
present inverse Hessian: $M_{\kappa-1}$
future inverse Hessian M_{κ}

update inverse Hessian estimate

$$M_{\kappa} = M_{\kappa-1} + \left(1 + \frac{\varphi' M \varphi}{\delta' \varphi}\right) \frac{\delta \delta'}{\delta' \varphi} - \left(\frac{\delta \varphi' M + M \varphi \delta}{\delta \varphi}\right)$$

compute descent direction $\rho_{\kappa+1} = M_{\kappa} \nabla E(\omega_{\kappa})$

line search $\omega_{\kappa+1} = \omega_{\kappa} - \eta_{\kappa+1} \rho_{\kappa+1}$

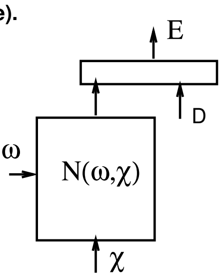
- it is an O(N^2) algorithm BUT
- it requires storing an NxN matrix
- it is a BATCH algorithm (requires a line search)

Only practical for VERY SMALL networks with non–redundant training sets.

Several variations exist that attempt to reduce the storage requirements:
- limited storage BFGS [Nocedal 80]
- memoryless BFGS, OSS [Battiti 92]

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)

4

APPLYING SECOND ORDER METHODS TO MULTILAYER NETS

(NON)APPLICABILITY OF 2nd ORDER METHODS TO NEURAL–NET LEARNING

BAD NEWS:

- Full Hessian techniques (GN, LM, BFGS) can only apply to small networks. But small networks are not the ones we need to speed up most.
- Most 2nd order techniques (CG, BFGS....) require a line search, and therefore are not directly usable in stochastic mode
- Many heuristic tricks (adaptive learning rates....) also apply to batch only.

On large classification problems, a carfully tuned stochastic gradient is hard to beat.

On smaller problems requiring accurate real–valued outputs (function approximation, control...), conjugate gradient (with Polak–Ribiere) offers the best combination of speed, reliability and simplicity.

This section is devoted to 2nd order techniques specifically designed for large neural–net training

MINI BATCH METHODS

Attempts at applying Conjugate Gradient to large and redundant problems have been made [Kramer&Sangiovani–Vincentelli 88], [Møller 92]

They use "mini batches": subsets of increasing sizes are used.

Møller proposes a systematic way of choosing the size of the mini batch.

He also uses a variant of CG which he calls "scaled CG". Essentially, the line search is replaced by a 1D Levenberg–Marquardt–like algorithm.

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

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

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

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:

$$\underbrace{\frac{\partial^2 E}{\partial \omega_{ki}^2}}_{\text{new estimate of 2nd der.}} \leftarrow (1-\gamma)\underbrace{\frac{\partial^2 E}{\partial \omega_{ki}^2}}_{\text{previous estimate}} + \underbrace{\gamma}_{\text{small constant}} \underbrace{\frac{\partial^2 E^p}{\partial \omega_{ki}^2}}_{\text{instantaneous 2nd der. for pattern p}}$$

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.

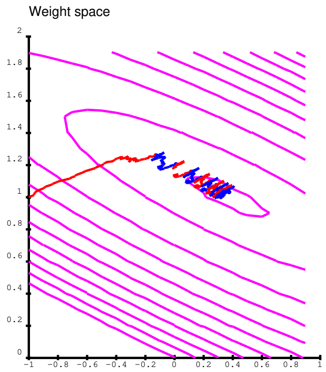
Is usually about 3 times faster than carefully tuned stochastic gradient.

Stochastic Diagonal Levenberg–Marquardt

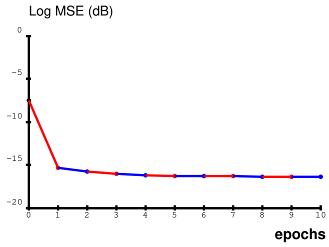
data set: set–1 (100 examples, 2 gaussians)
network: 1 linear unit, 2 inputs, 1 output.
2 weights, 1 bias.

Learning rates:
$\eta 0 = 0.12$
$\eta 1 = 0.03$
$\eta 2 = 0.02$

Hessian largest eigenvalue:
$\lambda_{max} = 0.84$



Maximum admissible Learning rate (batch):
$\eta_{max} = 2.38$



Stochastic Diagonal Levenberg–Marquardt

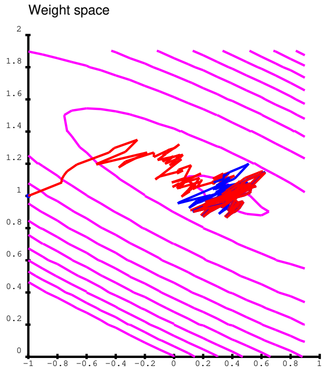
data set: set–1 (100 examples, 2 gaussians)
network: 1 linear unit, 2 inputs, 1 output.
2 weights, 1 bias.

Learning rates:

$\eta 0 = 0.76$
$\eta 1 = 0.18$
$\eta 2 = 0.12$

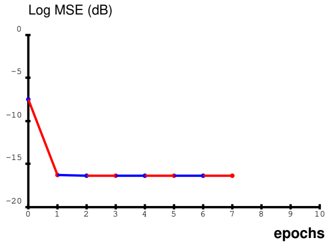
Hessian largest eigenvalue:

$\lambda_{max} = 0.84$



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 Ψ at random

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

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

COMPUTING THE PRODUCT $H\Psi$

IDEA #2 (Taylor expansion):

NEW ESTIMATE OF EIGENVECTOR

OLD ESTIMATE OF EIGENVECTOR

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

"SMALL" CONSTANT

PERTURBED GRADIENT

GRADIENT

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 Ψ

IDEA #3 (running average):

NEW ESTIMATE OF EIGENVECTOR

OLD ESTIMATE OF EIGENVECTOR

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

"SMALL" CONSTANTS

PERTURBED GRADIENT FOR CURRENT PATTERN

GRADIENT FOR CURRENT PATTERN

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

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

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)

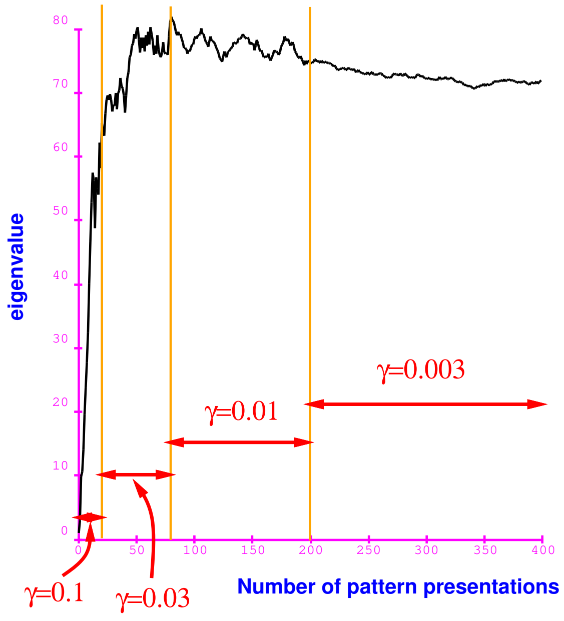
3 – add $\alpha\frac{\Psi}{\|\Psi\|}$ to current weight vector

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

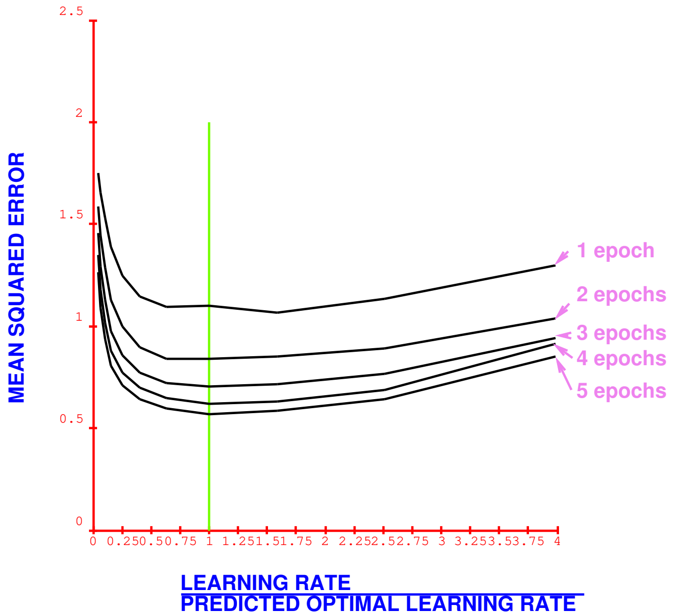
5 – compute difference G'(w)–G(w). and divide by α update running average of eigenvector with the result

6 – goto 2 unless a reasonably stable result is obtained

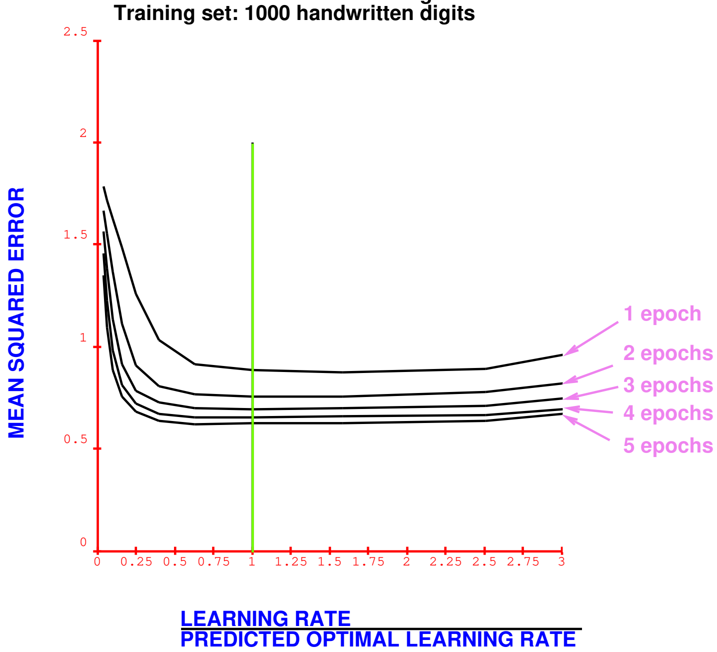
7 – the optimal learning rate is $\frac{1}{\|\Psi\|}$



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



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



Stochastic Learning

Léon Bottou
Email: leon@bottou.org

Summary

Notations.

Expected Risk

$$C(w) \triangleq \mathbf{E}_z\, L(z,w) \triangleq \int L(z,w)\; dP(z)$$

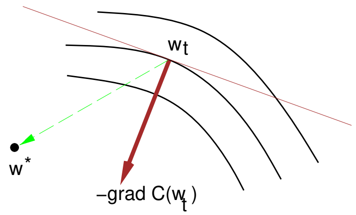
Stochastic Gradient Update

$$w_{t+1} = w_t - \gamma_t J(\mathbf{z}_t, w_t)$$
$$\mathbf{E_z}\, J(\mathbf{z}, w) = \nabla_w\, C(w)$$

Hessian

$$H(w) = \frac{\partial^2}{\partial w^2} C(w)$$

General Convexity.



General convexity assumption.

$$\forall \varepsilon > 0, \quad \inf_{(w-w^*)^2>\varepsilon} (w - w^*)\, \nabla_w C(w) > 0$$

- Gradient point towards the right direction.
- Gradient does not vanish (no plateaus).

Learning rates.

Requirement:

$$\gamma_t \, J(\mathbf{z}_t, w_t) \longrightarrow 0$$

Two possibilities:
- Decreasing learning rates: $\gamma_t \to 0$
- Decreasing gradients: $J(\mathbf{z}_t, w_t) \to 0$.

Note:

$$\mathbf{E}_z\left(J(\mathbf{z}_t, w_t)^2\right) \;\approx\; \mathbf{E}_z\left(J(\mathbf{z}_t, w^*)^2\right) + (w_t - w^*)' H(w^*)(w_t - w^*)$$
$$\mathbf{E}_z\left(J(\mathbf{z}_t, w_t)^2\right) \;\leq\; A + B(w_t - w^*)^2$$

In general $A \neq 0$. Therefore learning rates must decrease.

Three convergence proofs.

I will present three proofs for the following cases.
- Continuous gradient,
- Batch gradient,
- Stochastic gradient

All share the same three-step structure,
but use increasingly sophisticated tools.

Reference: Metivier (1981).

Continuous Gradient: step A.

Differential equation defines $w(t)$:

$$\frac{\mathrm{d}w}{\mathrm{d}t} = -\nabla_w C(w)$$

Step A: define Lyapunov function:

$$h(t) \;\stackrel{\triangle}{=}\; (w(t) - w^*)^2$$

Continuous Gradient: step B.

Step B: Lyapunov function converges

$$\frac{\mathrm{d}h}{\mathrm{d}t} = 2(w - w^*)\frac{\mathrm{d}w}{\mathrm{d}t} = -2(w - w^*)\nabla_w C(w) \leq 0$$

Function $h(t)$ is positive, decreasing $\Longrightarrow$ converges.

Continuous Gradient: step C.

Step C: Continuous gradient converges

Since $h(t)$ converges,

$$\frac{\mathrm{d}h}{\mathrm{d}t} = -2(w-w^*)\nabla_w C(w) \to 0$$

General convexity $\Longrightarrow$ $w(t) \to w^*$

Batch Gradient: step A.

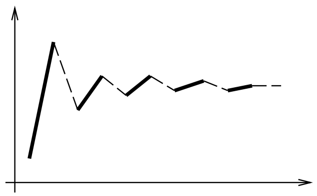
Update rule defines w_t:

$$w_{t+1} = w_t - \gamma_t \nabla_w C(w)$$

Step A: define Lyapunov sequence:

$$h_t \stackrel{\triangle}{=} (w_t - w^*)^2$$

Batch Gradient: lemma for step B.



$$S_t^+ = \sum_{i=1}^{t-1} \left[u_{t+1} - u_t \right]_+ \quad \text{with } [x]_+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Bounded positive variations convergence theorem:

$$\left.\begin{array}{r} \forall t,\ u_t > 0 \\ S_t^+ \text{ converges} \end{array}\right\} \Longrightarrow u_t \text{ converges.}$$

Batch Gradient: step B.

Step B: Convergence of Lyapunov sequence:

⚠ Additional assumptions.

$$\left.\begin{array}{r} \sum \gamma_t^2 \text{ converges} \\ (\nabla_w C(w))^2 < A + B(w-w^*)^2 \end{array}\right\} \Longrightarrow h_t \text{ converges.}$$

Proof (assuming $B=0$):

$$\begin{aligned} \left[h_{t+1} - h_t\right]_+ &= \left[-2\gamma_t (w_t - w^*)\nabla_w C(w_t) + \gamma_t^2 (\nabla_w C(w_t))^2\right]_+ \\ &\leq \gamma_t^2 (\nabla_w C(w_t))^2 \leq \gamma_t^2 A \end{aligned}$$

and apply the bounded positive variations convergence theorem.

Proof sketch (assuming $B \neq 0$):

Define $\mu_t = \prod_{i=1}^t (1 - \gamma_i^2 B)$.
Show that $h'_t = h_t/\mu_t$ converges.
Then show that h_t converges as well.

Batch Gradient: step C.

Step C: Batch gradient converges.

⚠ Additional assumption:

$$\sum \gamma_t = \infty \quad \Longrightarrow \quad w_t \to w^*$$

Proof:

$$h_{t+1} - h_t \;=\; -2\gamma_t\,(w_t - w^*)\nabla_w C(w_t) \;+\; \gamma_t^2\,(\nabla_w C(w_t))^2$$

Both blue terms have convergent sums.

Therefore $\sum \gamma_i (w_i - w^*)\nabla_w C(w_i)$ converges.

The assumption then means that $(w_t - w^*)\nabla_w C(w) \to 0$.

The general convexity hypothesis then implies the convergence.

Stochastic Gradient: step A.

Stochastic update rule defines w_t:

$$w_{t+1} = w_t - \gamma_t J(\mathbf{z}_t, w_t)$$

Step A: define Lyapunov process:

$$h_t \;\triangleq\; (w_t - w^*)^2$$

Stochastic Gradient: Quasi-martingales.

Let u_t be a stochastic process.
Let $\mathcal{P}_t$ represent what is known on time t, ($u_0, \ldots, u_t$.)

Define indicator of expected positive differences.

$$\delta_t = \begin{cases} 1 & \text{if } \mathbf{E}(u_{t+1} - u_t | \mathcal{P}_t) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

and the sum

$$S_t^+ = \sum_{i=1}^{t} \mathbf{E}(\delta_t\,(u_{t+1} - u_t))$$

Positive quasi-martingale convergence theorem:

$$\left.\begin{array}{r} \forall t,\; u_t > 0 \\ S_t^+ \text{ converges} \end{array}\right\} \Longrightarrow \; u_t \text{ converges almost surely.}$$

Metivier: *Semi-martingales*, 1983, Theorem 9.4, Proposition 9.5

Stochastic Gradient: step B.

Step B: Convergence of Lyapunov process:

⚠ Additional assumptions.

$$\left.\begin{array}{r} \sum \gamma_t^2 \text{ converges} \\ \mathbf{E}\left(J(\mathbf{z}, w)^2\right) < A + B(w - w^*)^2 \end{array}\right\} \Longrightarrow \quad h_t \text{ converges a.s.}$$

Proof (assuming $B = 0$):

$$\begin{aligned} \mathbf{E}\,(h_{t+1} - h_t \,|\, \mathcal{P}_t) &= -2\,\gamma_t (w_t - w^*)\,\mathbf{E}\,(J(\mathbf{z}_t, w_t) \,|\, \mathcal{P}_t) + \gamma_t^2\,\mathbf{E}\left(J(\mathbf{z}_t, w_t)^2 \,|\, \mathcal{P}_t\right) \\ &= -2\,\gamma_t (w_t - w^*)\nabla_w C(w_t) + \gamma_t^2\,\mathbf{E}_{\mathbf{z}}(J(\mathbf{z}_t, w_t)^2) \\ &\le \gamma_t^2\,A \end{aligned}$$

$$\mathbf{E}(\delta_t (h_{t+1} - h_t)) \;=\; \mathbf{E}(\delta_t \mathbf{E}\,(h_{t+1} - h_t \,|\, \mathcal{P}_t)) \;\le\; \mathbf{E}\,(\mathbf{E}\,(h_{t+1} - h_t \,|\, \mathcal{P}_t)) \;\le\; \gamma_t^2\,A$$

and apply the quasi-martingale convergence theorem.

The case $B \neq 0$ can be treated as suggested for batch gradient.

Stochastic Gradient: step C.

Step B: Stochastic gradient converges almost surely.

⚠ Additional assumption:

$$\sum \gamma_t = \infty \quad \Longrightarrow \quad w_t \xrightarrow{a.s.} w^*$$

Proof:

$$\mathbf{E}\left(h_{t+1} - h_t\right) \;=\; -2\gamma_t\, \mathbf{E}\left((w_t - w^*)\nabla_w C(w_t)\right) + \gamma_t^2\, \mathbf{E}_{\mathbf{z}}(J(\mathbf{z}_t, w_t)^2)$$

Both blue terms have convergent sums.

Therefore $\sum \gamma_i \mathbf{E}\left((w_i - w^*)\nabla_w C(w_i)\right)$ converges.

The assumption then means that $\liminf (w_t - w^*)\nabla_w C(w) = 0$ (a.s.)

The general convexity hypothesis then implies the convergence.

General case.

- Non convex cost functions.
- Several local minima.
- Possibly several global minima.
- Critical subspaces.
- Saddle points.

Local confinement.

- Partition space into attraction basins.
- Make sure w_t is confined to a specific basin.
- Define suitable Lyapunov functions on the basin

Reference: Krasovskii (1963)

Batch gradient: Confinement works well.
Stochastic gradient: Confinement works poorly.

Stochastic noise can always get w_t out of any attraction basin with small but non zero probability.

Standard assumptions

$$C(w) > 0$$

$C(w)$ has continuous second derivatives.

$$\sum_i \gamma_i = \infty$$

$\sum_i \gamma_i$ converges.

$$\mathbf{E}_{\mathbf{z}}\left(J(\mathbf{z}, w)^2\right) \leq A + Bw^2$$

Global confinement.

⚠ Additional Assumptions

$$\left.\begin{array}{ll} \exists D > 0, & \inf\limits_{w^2>D} \; w \, \nabla_w C(w) > 0 \\ \exists E > D, & \sup\limits_{w^2<E} \; J(\mathbf{z}, w)^2 < K \end{array}\right\} \Longrightarrow \mathbf{P}\left\{\exists t_0, \forall t > t_0,\; w_t^2 < 2E\right\} = 1$$

Proof sketch:

Define Lyapunov process $h_t = \max(E, w_t^2)$.

Performs steps a, b, and c. Conclude.

Benefits:

We are now working in a compact set $\{w_t^2 < E\}$.

Continuity implies that all continous quantities are bounded a.s.

Convergence to Extremal Points.

$$\ldots \Longrightarrow \left\{\begin{array}{l} C(w_t) \text{ converges to some value } C_\infty. \\ \nabla_w C(w_t) \xrightarrow{a.s.} 0 \end{array}\right.$$

Proof sketch:

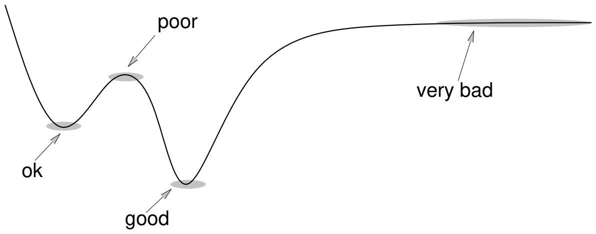
a- Define Lyapunov process $h_t = C(w_t)$.
b- Lyapunov process converges.
c- Define secondary Lyapunov process $g_t = (\nabla_w C(w_t))^2$.
Perform steps a, b, and c again.

Note:

We must either use global confinement

or assume suitable bounds on $||\nabla\nabla C(w)||$.

Convergence to Extremal Points.



- Local maxima and saddle points are unstable.
  But one can construct cases whithout noise.
- Infinite plateaus are ruled out by global confinement.
  Otherwise they can spoil the day.

Speed issues.

Final convergence speed

- Replace learning rate γ_t by a matrix Φ_t.
- Estimate $\mathbf{E}((w_t - w^*)^2)$.

Speed during the search phase

- Information geometry.
- Speedup via regularisation.

$\Longrightarrow$ Optimal Learning Algorithms.

Conclusion.

These convergence results apply to:
- adalines
- multilayer networks
- perceptrons
- kmeans
- lvq2
- . . .

Stochastic Learning

Léon Bottou
Email: `leon@bottou.org`

Summary

1- Framework, Examples
2- Backprop Cookbook
3- Convergence
4- **Optimal Online Algorithms**
   - Theory
   - Discussion
   - Experiments
   - Open questions

Chronology

1970 Tsypkin & others
   Optimal learning systems.
1995 Saad, Solla, Caticha & others
   Optimal online algorithms in statistical physics.
   (e.g. teacher network / student network, overlaps).
1997 Murata, Amari
   Statistical Analysis of Learning.
   Natural Gradient achieves Cramer-Rao bound.
2003 Bottou, LeCun
   to appear...

Question

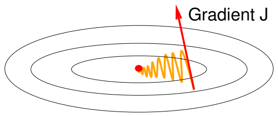
Very large number of examples.
Limited time.

1. Run well designed online algorithm.
   Never present the same example twice.
   Process examples until time runs out.

2. Run superlinear batch algorithm.
   Process the largest set of example we can afford.

Answer: ?

Gradient Descent (batch, constant rate)

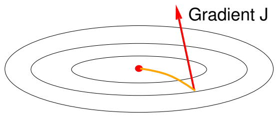


$$J(w) = \frac{\partial}{\partial w} C(w) \qquad w_t = w_{t-1} - \eta \, J(w_{t-1})$$

When $\lambda_{\text{min}} << \lambda_{\text{max}}$ and $\eta < 2/\lambda_{\text{max}}$,

$$|w_t - w^*|^2 \equiv \frac{1}{\exp(2\,\eta\,\lambda_{\text{min}}\,t\,)}$$

Gradient Descent (batch, superlinear)

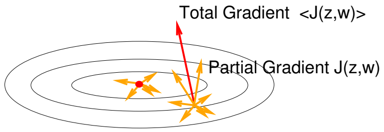


$$J(w) = \frac{\partial}{\partial w} C(w) \qquad w_t = w_{t-1} - \Phi_t \, J(w_{t-1})$$

Proper choice of the scaling matrix Φ_t yields:

$$|w_t - w^*|^2 \equiv \frac{1}{\exp(\ldots\exp(\ldots t))}$$

Online Gradient Descent



$$C(w) \;=\; \mathbf{E}_z(L(z,w))$$

$$J(z,w) \;=\; \frac{\partial}{\partial w} L(z,w) \qquad w_t = w_{t-1} - \frac{1}{t}\,\Phi_t \, J(z, w_{t-1})$$

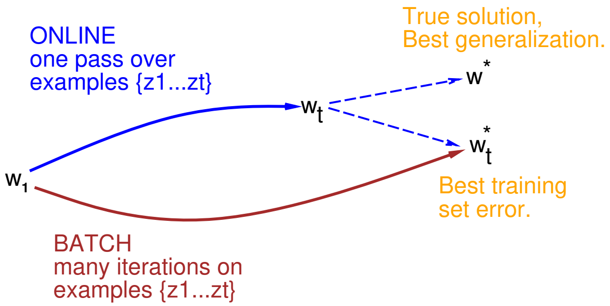
Residual noise proportional to learning rate $1/t$.

$$|w_t - w^*|^2 \equiv \frac{1}{t} \quad (\ldots\text{at best.})$$

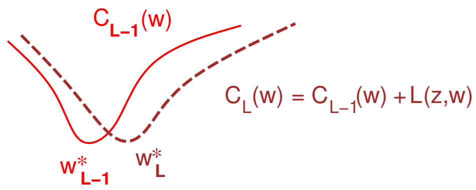
Learning vs. Optimization

- Batch learning optimizes the training error, $< L(z,w) >$ averaged on the training set.
- Online learning slowly optimizes . . . $< L(z,w) >$ averaged on whatever distribution the online examples are drawn from. What distribution exactly?

Online vs. Batch



Dynamics of w^*_L



Simple expansion shows

$$w^*_L = w^*_{L-1} - \Psi_L \, \frac{1}{L} \, J(z_L, w^*_{L-1}) + \mathcal{O}\left(\frac{1}{L^2}\right)$$

Matrix Ψ_L converges to the inverse of the Hessian matrix

$$\Psi_L \to H^{-1} \qquad\qquad H = \frac{\partial^2}{\partial w^2} C(w^*)$$

Compare. . .

- Movement of the training set solution

  when adding one additional training example z_L.

  $$w^*_L = w^*_{L-1} - \Psi_L \, \frac{1}{L} \, J(z_L, w^*_{L-1}) + \mathcal{O}\left(\frac{1}{L^2}\right)$$

- Parameter update rule for online gradient.

  $$w_t = w_{t-1} - \Phi_t \, \frac{1}{t} \, J(z_t, w_{t-1})$$

Same speed?

Theorem

Consider the stochastic process

$$w_t = w_{t-1} - \Phi_t \, \frac{1}{t} \, J(z_t, w_{t-1}) + \mathcal{O}\left(\frac{1}{t^2}\right)$$

Assume $\Phi_t \to \Phi^* = H^{-1}$

THEN: $\mathbf{E}\left((w_t - w^*)^2\right) = \frac{K}{t} + \mathrm{o}\left(\frac{1}{t}\right)$

AND: $K = \mathbf{trace}\left(\Phi^* \; \mathbf{E}\left[J(z, w^*) J'(z, w^*)\right] \; \Phi^* \right)$

Theorem

Proof sketch:

1- Lemma: Consider a positive sequence u_t satisfying

$$u_t = \left(1 - \frac{\alpha}{t} + o\left(\frac{1}{t}\right)\right) u_{t-1} + \frac{\beta}{t^2} + o\left(\frac{1}{t^2}\right)$$

Prove that $t\,u_t \longrightarrow \frac{\beta}{\alpha-1}$ if $\alpha > 1$ and $\beta > 0$.

2- Write $\mathbf{trace}\left(\mathbf{E}\left(\theta_t\theta_t' \mid \mathcal{P}_t\right)\right)$ as a function of θ_{t-1}.

3- Take the unconditional expectation and apply the lemma.

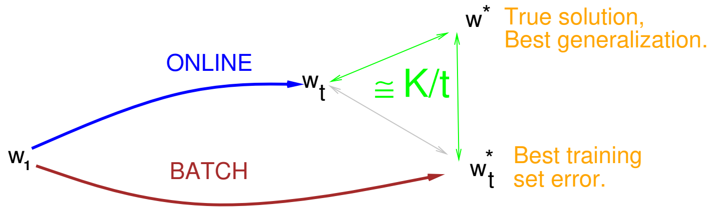
Remark:

The same $\frac{1}{t}$ behavior holds if $\Phi_t \to \Phi^* > \frac{1}{2}H^{-1}$.

The constant K is worse by a factor $\approx ||H\Phi^*||$.

Consequence

When $\Phi_t \to \Phi^* = H^{-1}$



$$\mathbf{E}\left[(w_t - w^*)^2\right] \approx \mathbf{E}\left[(w_t^* - w^*)^2\right] \approx \frac{K}{t}$$

Same K.

Special Case

Murata & Amari, 1998:
"Natural Gradient achieves Cramer-Rao bound."

- Maximum likelihood loss $L(z,w) = -\log \phi_w(z)$
- Hessian H equals Fisher information matrix $\mathcal{G}(w^*)$.
- Natural Gradient matrix $\to H^{-1}$.
- $K = \mathbf{trace}\left(\mathcal{G}(w^*)^{-1}\right)$

where we recognize the Cramer-Rao bound.

Consequence I

Learning is mainly limited by the fact that some informative examples have not yet been seen rather than by the fact that the examples already seen have not been fully exploited by the minimization process.

Consequence II

Very large number of examples.
Limited time.

1. Run well designed online algorithm.
   Never present the same example twice.
   Process examples until time runs out.

2. Run superlinear batch algorithm.
   Process the largest set of example we can afford.

Answer: 1 — Option 1 wins by a factor $\log \log T$.

Consequence II

Proof:

Batch learning on N examples:

$$(w_N^* - w^*)^2 \sim \frac{1}{N}$$

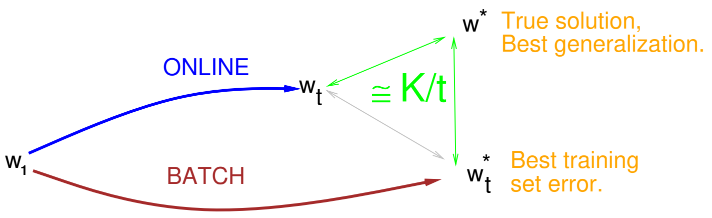
Batch must perform $\sim \log \log N$ iterations to this accuracy.

During the same time, online can process $\sim N \log \log N$ examples.

$$(w_t - w^*)^2 \sim \frac{1}{N \log \log N} \ll \frac{1}{N} \sim (w_N^* - w^*)^2$$

No need to use the constant K!

Consequence III



When $\Phi_t \to \Phi^* = H^{-1}$

One epoch is (almost) enough.

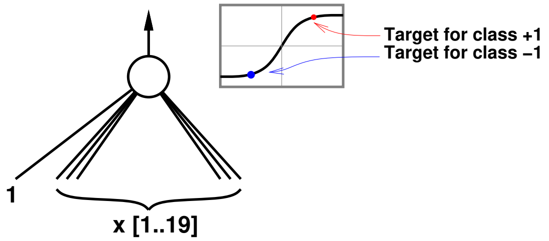
Experiments: data.

Simple validation experiment.

- Dimension 19
- Two classes $+1$ and -1.
- One gaussian per class.
- Eigenvalues range from 1 to 20.
- Optimal classification error $\approx 10\%$.

Experiments: model.

One sigmoid unit trained with least squares.



Akin to a Generalized Linear Model (GLIM).
– Represents well the final convergence phase.
– Significant applications exist.

Batch-Newton algorithm.

Repeat:

- Compute total gradient $g(w)$ and hessian $H(w)$.
- Update:

$$w := w - H^{-1}(w)g(w)$$

Quadratic convergence.
Similar to GLIM's traditional algorithm (IRLS).

Online-Kalman algorithm.

One sequential sweep over the training set.

$$w := w - \frac{1}{\tau}\Phi_t \frac{\partial L}{\partial w}(x_t, y_t, w)$$

– Leaky approximation of the Hessian $\widehat{H}_t$.
– Gauss Newton trick eliminates negative curvature.
– Scaling matrix $\Phi_t = \widehat{H}_t^{-1}$

Note: $\tau = \max(20, t-40)$

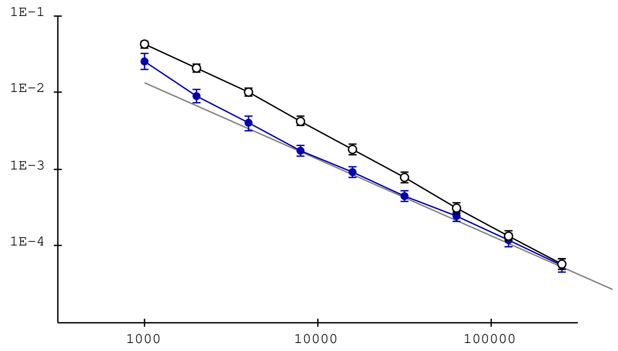
Online-Kalman algorithm.

$$\Phi_t = \left(\left(1 - \frac{2}{\tau}\right) \Phi_{t-1}^{-1} + \left(\frac{2}{\tau}\right) \left(f'(\theta_{t-1} x_t)\right)^2 x_t x_t^\top \right)^{-1}$$

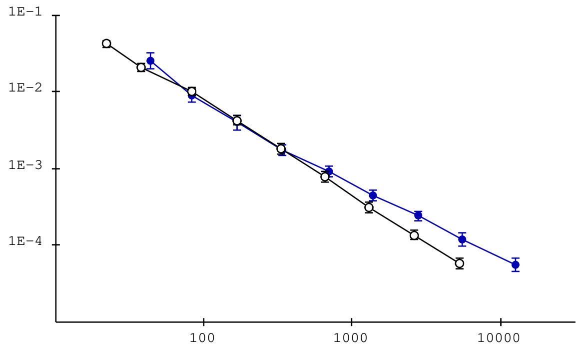
No matrix inversion required.
(Matrix inversion lemma)

$$\left(\alpha A^{-1} + \beta u u^\top\right)^{-1} = \frac{1}{\alpha}\left(A - \frac{(Au)(Au)^\top}{\alpha/\beta + u^\top A u}\right)$$

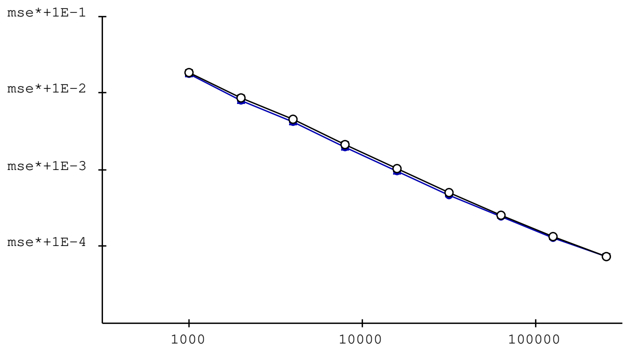
$w - w^*$ vs. number of examples.



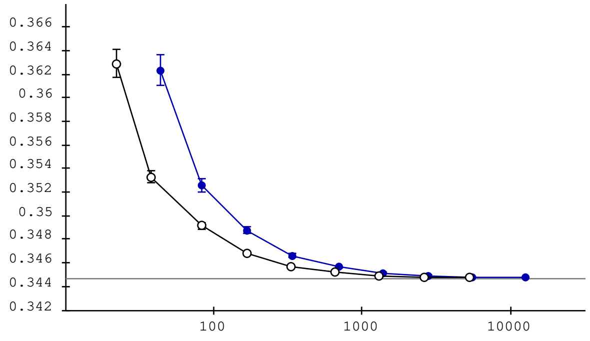
$w - w^*$ vs. time.



Test MSE vs. number of examples.



Test MSE vs. time.



Discussion.

It works!

Immediate conclusion
– Can train GLIM faster than Newton (or IRLS).
– Only one pass over the training data.

Perspectives
Theory applies to arbitrary gradient descent.
– Non linear networks (convolutional, transducers, etc)
– One pass? (after how many examples?)

Open issues

Final convergence vs. Search phase.
– Asymptotic result addresses the final convergence phase.
– Search phase impacts how fast we reach this regime.

Online learning vs. Vapnik's theory.
– Both address generalization questions.
– Links?

Simplified scaling matrices.
– Full rank scaling matrices are too expensive.
– Limited storage approximations might be good enough.

Final words.

Hardware facts:

Tremendous improvements in $\left\{ \begin{array}{l} \text{mass storage.} \\ \text{network bandwidth.} \end{array} \right.$

Faster than Moore's law on processing power.

Consequences:

Large data sets are available for machine learning.
Computing power is getting relatively scarce.

Stochastic algorithms become attractive (again).