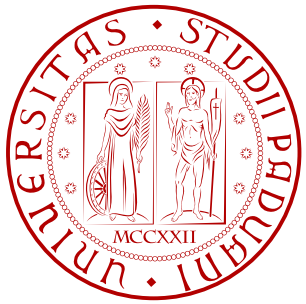
# TOOLS FOR LEARNING: FUNDAMENTALS OF NUMERICAL OPTIMIZATION – PART 2

Michele Rossi

rossi@dei.unipd.it

Dept. of Information Engineering

University of Padova, IT

# Outline

- Advanced methods
  - Gradient descent and quadratic surrogates
  - Backtracking line search
  - Stochastic gradient descent

# Methods for Large Scale Datasets

- We discuss
  - Fixed
  - *Adaptive* step rules

- All these methods have
  - Guaranteed convergence to global minimum

  (of course, assuming that the function is *convex*, if non-convex, convergence is to a stationary point)

# Gradient descent and quadratic surrogates

- Second-order Taylor expression of a cost function g centered at point $\mathbf{w}^0$ (used in *Newton's method*)

$$g(\boldsymbol{w}^0) + \nabla g(\boldsymbol{w}^0)^T(\boldsymbol{w} - \boldsymbol{w}^0) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^0)^T \nabla^2 g(\boldsymbol{w}^0)(\boldsymbol{w} - \boldsymbol{w}^0)$$

- However
  - For large scale problems
  - There are difficulties in storing & even calculating the Hessian
- Still
  - The idea of hopping down a function g(**w**) using a second order approximation is *very* appealing

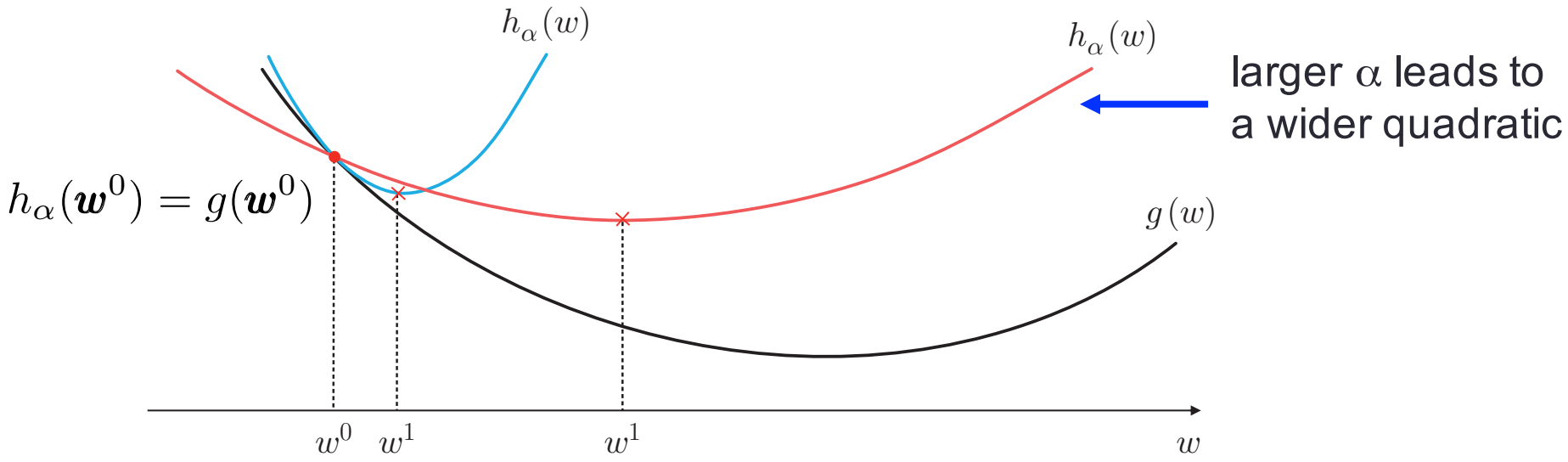
**Question**

Can we replace the Hessian with a simpler quadratic term and still obtain an effective gradient descent algorithm?

# A simple quadratic surrogate (1/2)

$$h_\alpha(\boldsymbol{w}) = g(\boldsymbol{w}^0) + \nabla g(\boldsymbol{w}^0)^T(\boldsymbol{w} - \boldsymbol{w}^0) + \frac{1}{2\alpha}\|\boldsymbol{w} - \boldsymbol{w}^0\|_2^2$$

- With $\alpha$>0
- This is just the previous 2nd order Taylor approx
  - where we replaced the Hessian with the diagonal matrix $(1/\alpha)\boldsymbol{I}_{N\times N}$



# A simple quadratic surrogate (2/2)

- The quadratic surrogate has a single global min (found through first order condition on gradient)

$$\nabla h_\alpha(\boldsymbol{w}^0) = \nabla g(\boldsymbol{w}^0) + \frac{1}{\alpha}(\boldsymbol{w} - \boldsymbol{w}^0) = \mathbf{0}$$

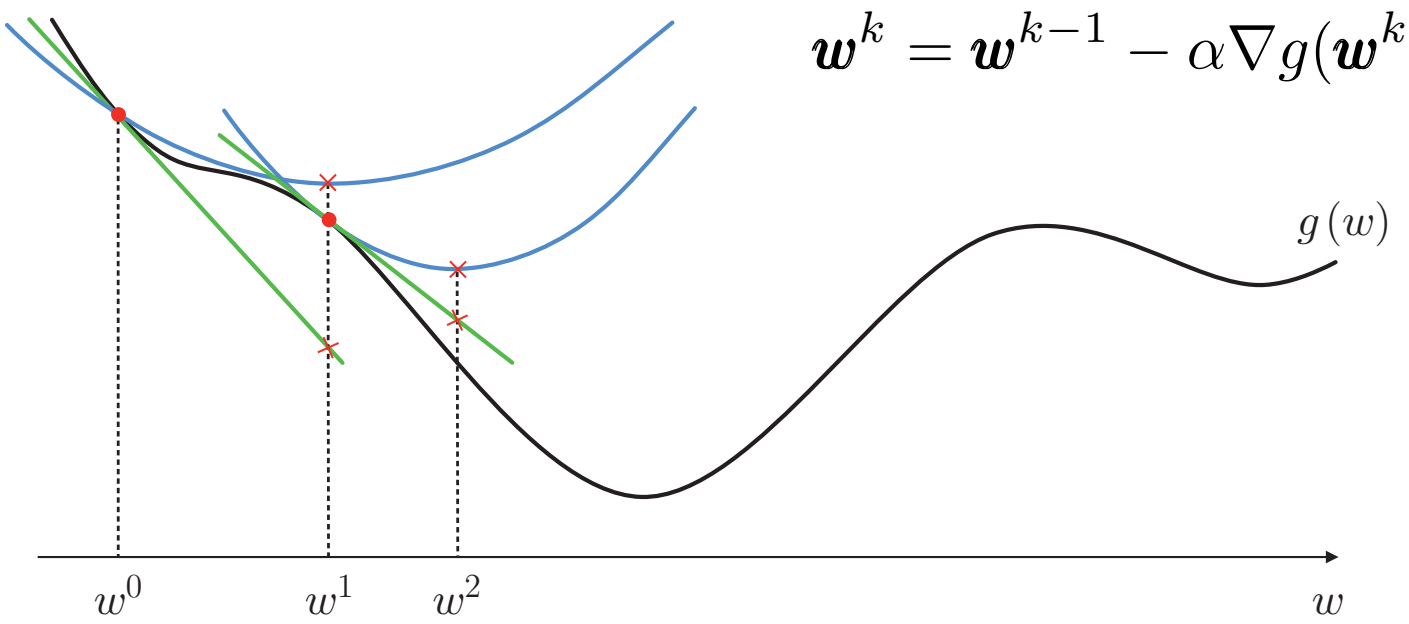
- Solving it, leads to $\mathbf{w}^1$

$$\boldsymbol{w}^1 = \boldsymbol{w}^0 - \alpha \nabla g(\boldsymbol{w}^0)$$

- Note that the minimizer of the surrogate is precisely a gradient descent step at $\mathbf{w}^0$ with step length $\alpha$
- This produces the same result as using a *linear approximation* and taking a step length of $\alpha$

# Quadratic *vs* linear surrogate

- Repeating the procedure for subsequent points…

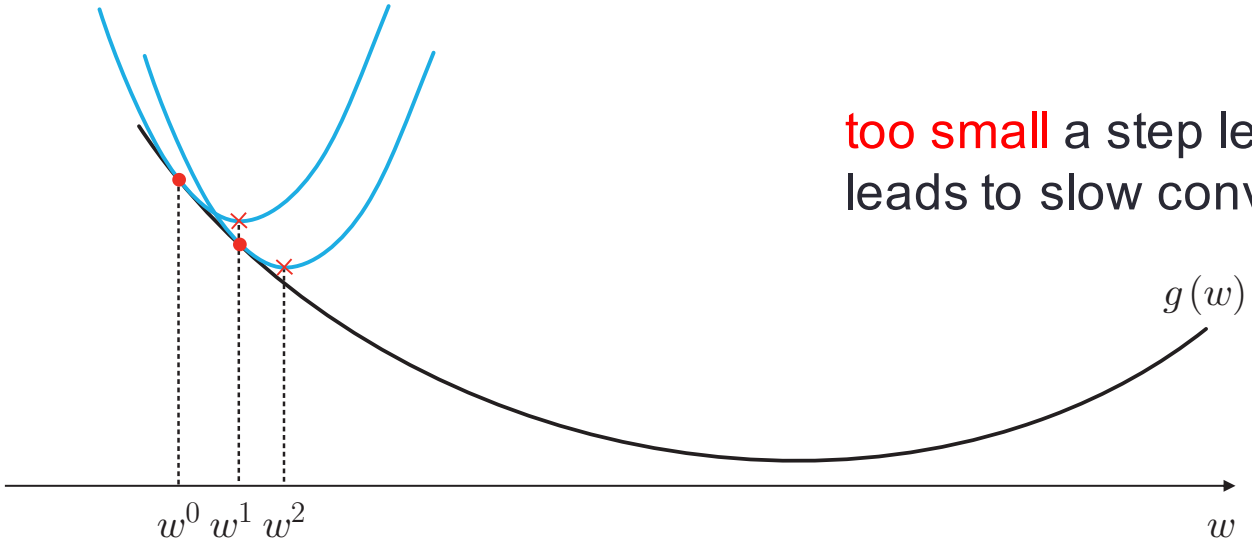



Gradient descent can be viewed simultaneously as using either linear or simple quadratic surrogates to find a stationary point of $g$. At each step the associated step length defines both how far along the linear surrogate we move before hopping back onto the function $g$, and at the same time the width of the simple quadratic surrogate which we minimize to reach the same point on $g$.

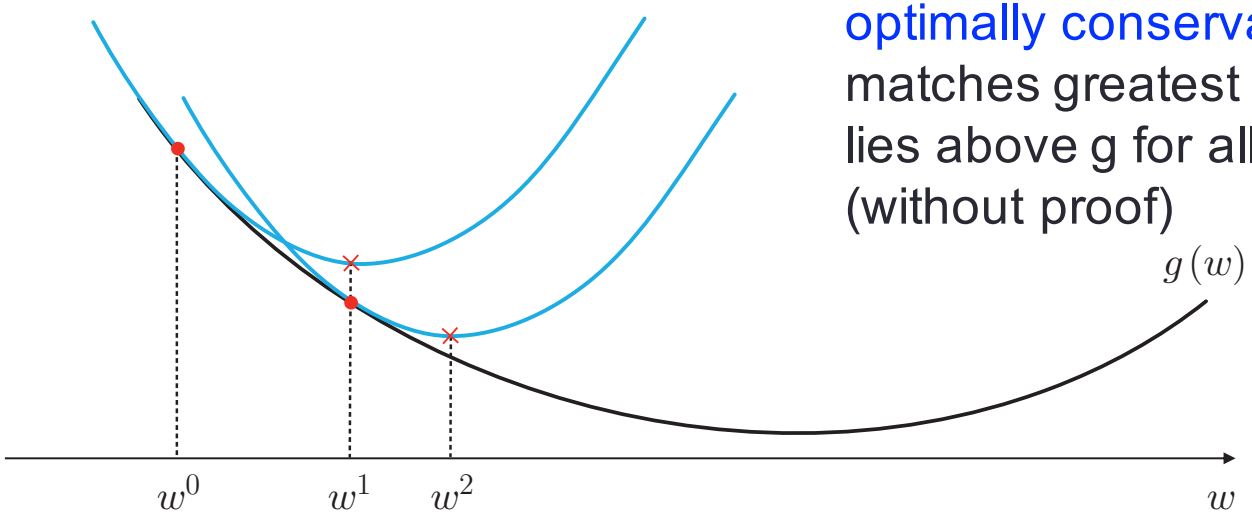
# Choosing the right step length

- We would like to pick a step length such that
  - It matches the greatest curvature of the cost function
  - This leads to the widest surrogate
  - That always lies above the cost function
  - It is called the "optimally conservative" step length
- Advantages
  - This optimal step size can be analytically computed for most of the popular cost functions
  - It is still small (conservative), but it is guaranteed to lead to convergence to global minimum
  - For this reason, it is very useful as a term of comparison against other methods

# Choosing the right step length - example




too small a step length
leads to slow convergence




optimally conservative
matches greatest curvature of g
lies above g for all values of w
(without proof)

# Formally

- If g has *globally bounded curvature*, there must exist a constant L>0 that bounds its second order derivative *above* and *below*

- Scalar function $$-L \leq \frac{\partial^2}{\partial^2 w} g(w) \leq L$$

- Vector input $$-L \preceq \nabla^2 g(\boldsymbol{w}) \preceq L$$

(all eigenvalues of the Hessian must be bounded between –L and L)

In this case, g is said to have "bounded curvature" or equivalently to have "Lipschitz continuous gradient" with Lipschitz constant L>0

# The most well known cost functions

| Cost function | Form of cost function | Lipschitz constant |
|---|---|---|
| Least Squares regression | $\sum_{p=1}^{P}\left(\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}-y_p\right)^2$ | $L=2\left\|\tilde{\mathbf{X}}\right\|_2^2$ |
| Softmax cost/logistic regression | $\sum_{p=1}^{P}\log\left(1+e^{-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}}\right)$ | $L=\frac{1}{4}\left\|\tilde{\mathbf{X}}\right\|_2^2$ |
| Squared margin/soft-margin SVMs | $\sum_{p=1}^{P}\max^2\left(0,1-y_p\tilde{\mathbf{x}}_p^T\tilde{\mathbf{w}}\right)$ | $L=2\left\|\tilde{\mathbf{X}}\right\|_2^2$ |
| Multiclass softmax | $\sum_{c=1}^{C}\sum_{p\in\Omega_c}\log\left(1+\sum_{\substack{j=1\\j\neq c}}^{C}e^{\tilde{\mathbf{x}}_p^T\left(\tilde{\mathbf{w}}_j-\tilde{\mathbf{w}}_c\right)}\right)$ | $L=\frac{C}{4}\left\|\tilde{\mathbf{X}}\right\|_2^2$ |
| $\ell_2$-regularizer | $\lambda\left\|\mathbf{w}\right\|_2^2$ | $L=2\lambda$ |

$$\tilde{X}=\left[\tilde{\boldsymbol{x}}_1\ \tilde{\boldsymbol{x}}_2\cdots\tilde{\boldsymbol{x}}_P\right]$$

$\left\|\tilde{X}\right\|_2^2$ is the *spectral norm* of $\tilde{X}^T\tilde{X}$ which corresponds to the *largest eigenvalue* of this matrix

# Gradient descent based on Lipschitz constant

---

**Input:** function $g$ with Lipschitz continuous gradient, and initial point $\mathbf{w}^0$

$k = 1$

**Find** the smallest $L$ such that $-L\mathbf{I} \preceq \nabla^2 g(\mathbf{w}) \preceq L\mathbf{I}$ for all $\mathbf{w}$ in the domain of $g$

**Choose** a constant $t \geq 1$

Set $\alpha = t \cdot \frac{1}{L}$

**Repeat until stopping condition is met:**

$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$

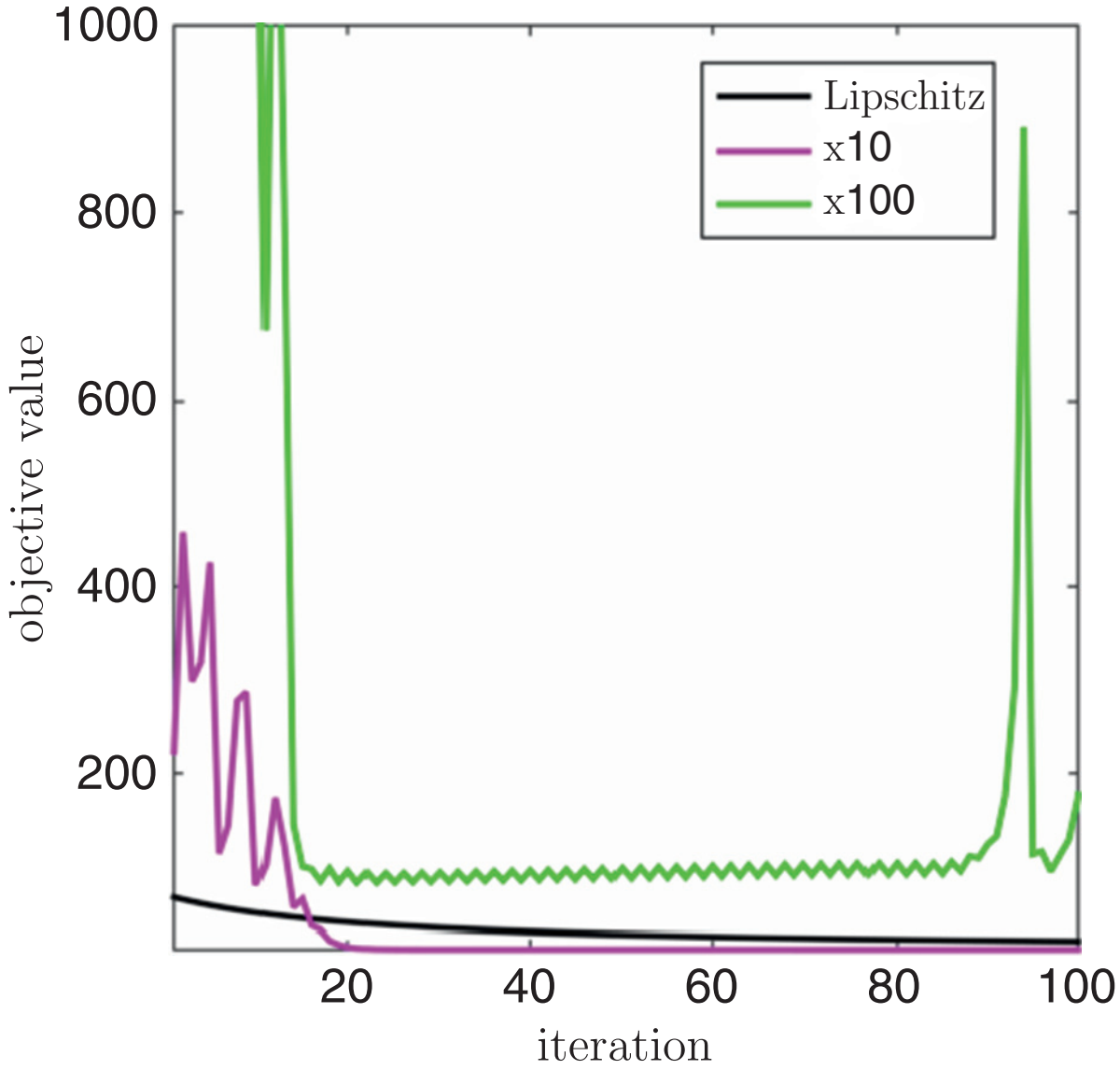
$k \leftarrow k + 1$

---

- L can be multiplied by t to *empirically evaluate* step lengths that are larger than the optimally conservative one
- There are no guarantees of convergence when a larger step is used (t>1)

# Example results

- Minimize softmax cost / logistic regression
- t=10 provides faster convergence
- t=100 leads to instability
- This *depends on the dataset*
- Has to be *empirically evaluated*



# Adaptive step length rule: backtracking line search

- General and widely applicable method
- Dynamically determines the step length at each iteration
- If $\mathbf{w}^{k-1}$ is the current point, the next one is (1) $\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \alpha \nabla g(\boldsymbol{w}^{k-1})$
- If we pick $\alpha$ such that the following inequality holds:

$$g(\boldsymbol{w}^k) \leq h_\alpha(\boldsymbol{w}^k) = g(\boldsymbol{w}^{k-1}) + \nabla g(\boldsymbol{w}^{k-1})^T(\boldsymbol{w}^k - \boldsymbol{w}^{k-1}) + \frac{1}{2\alpha}\|\boldsymbol{w}^k - \boldsymbol{w}^{k-1}\|_2^2$$

using (1): $-\alpha \nabla g(\boldsymbol{w}^{k-1})$ and $\alpha^2 \|\nabla g(\boldsymbol{w}^{k-1})\|_2^2$

- Which is equivalent to:

$$g(\boldsymbol{w}^k) \leq g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2 \quad (2)$$

- Note that, unless we have reached the *minimum* the *gradient term in the RHS is always negative* and we *have a descent*, i.e., $g(\boldsymbol{w}^k) < g(\boldsymbol{w}^{k-1})$

# Quadratic surrogate – review of so far results

$$h_\alpha(\boldsymbol{w}^k) = g(\boldsymbol{w}^{k-1}) + \nabla g(\boldsymbol{w}^{k-1})^T(\boldsymbol{w}^k - \boldsymbol{w}^{k-1}) + \frac{1}{2\alpha}\|\boldsymbol{w}^k - \boldsymbol{w}^{k-1}\|_2^2$$

- The minimum of the surrogate is found taking the derivative:

$$\nabla h_\alpha(\boldsymbol{w}^k) = \nabla g(\boldsymbol{w}^{k-1}) + \frac{1}{\alpha}(\boldsymbol{w}^k - \boldsymbol{w}^{k-1}) = \mathbf{0}$$

- From which we obtain the update rule:

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \alpha\nabla g(\boldsymbol{w}^{k-1})$$

- And using this result back into the surrogate, we obtain the value of the quadratic surrogate at the next point $\mathbf{w}^k$ (which is a minimum of the surrogate):

$$h_\alpha(\boldsymbol{w}^k) = g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2$$

# Backtracking line search

$$g(\boldsymbol{w}^k) \leq g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2 \quad (2)$$

From the previous slides, we have found that

- Computing the minimum of the surrogate
- And using it to compute the next point leads to $\mathbf{w}^k$

$$h_\alpha(\boldsymbol{w}^k) = g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2$$

- Using (2) above, for the given $\alpha$, we know that the surrogate rests about the curve (cost function) g(w):

$$g(\boldsymbol{w}^k) \leq h_\alpha(\boldsymbol{w}^k)$$

# Backtracking line search: IDEA

$$g(\boldsymbol{w}^k) \leq g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2 \quad (2)$$

If we already know the value of $\alpha$ s.t. (2) holds

- We use

$$\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \alpha \nabla g(\boldsymbol{w}^{k-1})$$

$$h_\alpha(\boldsymbol{w}^k) = g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2$$

- And automatically obtain

$$g(\boldsymbol{w}^k) \leq h_\alpha(\boldsymbol{w}^k)$$

- TRICK: if we instead do not know $\alpha$ in advance, we can use (2) as the rule to check whether h($\mathbf{w}^k$) rests about g($\mathbf{w}^k$) or not

# Backtracking Line Search (BLS)

- Common way of performing BLS is:
  - 1) Initialize $\alpha$>0
  - 2) Compute $\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \alpha \nabla g(\boldsymbol{w}^{k-1})$
  - 3) Check whether:

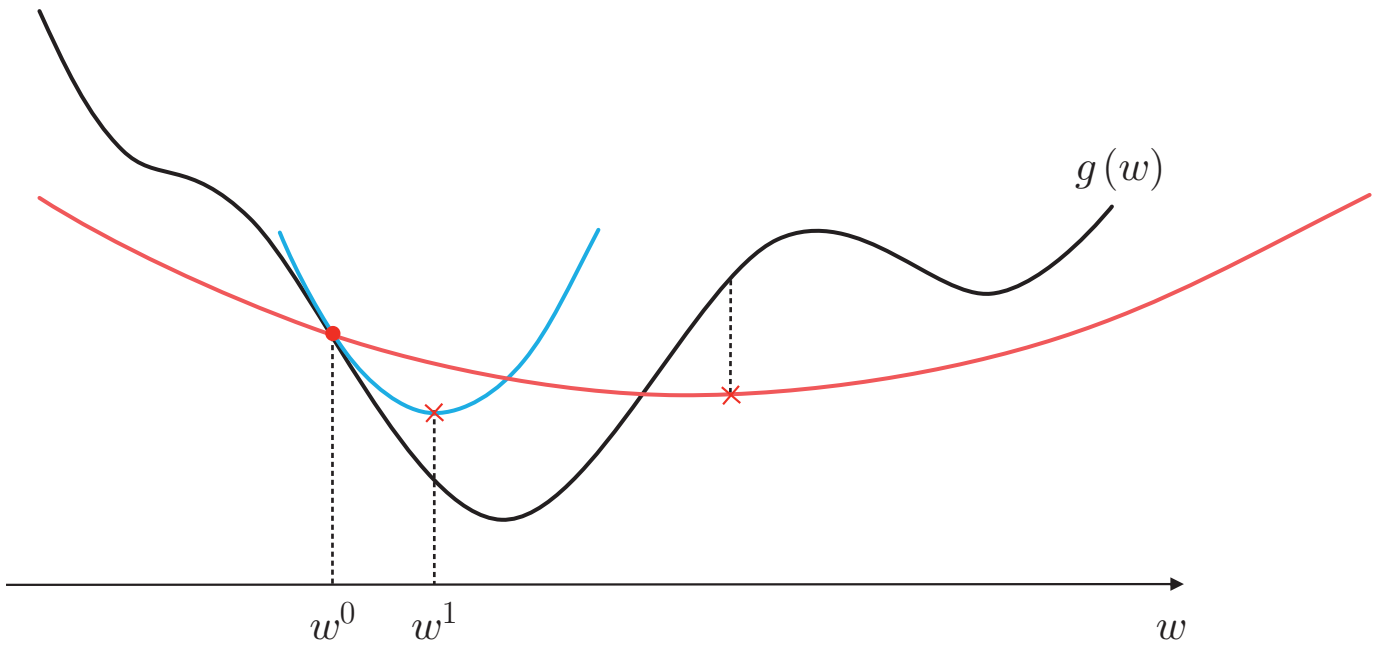
$$g(\boldsymbol{w}^k) \leq g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\nabla g(\boldsymbol{w}^{k-1})\|_2^2$$

  - Which is equivalent to check whether:

$$\underbrace{g(\boldsymbol{w}^{k-1} - \alpha \nabla g(\boldsymbol{w}^{k-1}))}_{\boldsymbol{w}^k} \leq g(\boldsymbol{w}^{k-1}) - \frac{\alpha}{2}\|\boldsymbol{w}^k - \boldsymbol{w}^{k-1}\|_2^2 \qquad (3)$$

  - If this is not verified: try again setting $\alpha \leftarrow \alpha t$ with $t \in (0,1)$
  - 4) Repeat check (3) until verified
  - 5) Use the final value of $\alpha$: $\boldsymbol{w}^k = \boldsymbol{w}^{k-1} - \alpha \nabla g(\boldsymbol{w}^{k-1})$

# Adaptive step length: illustration




A geometric illustration of backtracking line search. We begin with a relatively large initial guess for the step length, which generates the larger red quadratic, whose minimum may not lie above $g$. The guess is then adjusted downwards until the minimum of the associated quadratic (in blue) lies above the function.

# Adaptive step length: algorithm

---

**Input**: starting point $\mathbf{w}^0$, damping factor $t \in (0, 1)$, and initial $\alpha > 0$

$k = 1$

**Repeat until stopping condition is met:**

$\alpha_k = \alpha$

**While** $g\left(\mathbf{w}^{k-1} - \alpha_k \nabla g\left(\mathbf{w}^{k-1}\right)\right) > g\left(\mathbf{w}^{k-1}\right) - \frac{\alpha_k}{2}\left\|\nabla g\left(\mathbf{w}^{k-1}\right)\right\|_2^2$

$\alpha_k \longleftarrow t\alpha_k$

**End while**

$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha_k \nabla g\left(\mathbf{w}^{k-1}\right)$

$k \leftarrow k + 1$

---

# Backtracking line search: considerations

- It works out of the box
  - Tends to produce large steps at each iteration
  - However, each step *must be actively computed*
- Due to this
  - difficult to judge which rule (conservative-fixed *vs* adaptive) works best
- The choice of parameter t is also important
  - t close to 1: slow convergence, more computation, finer grained search
  - t close to 0: the other way around

# Stochastic Gradient Descent

- For large datasets gradient descent techniques become
  - computationally demanding
  - difficult if not impossible to store tables into memory
- Stochastic gradient descent solves this
  - completely overcomes the memory issue
  - is often computationally more efficient than Newton's
- For our discussion of it, we consider a dataset of P points

  $\{(\boldsymbol{x}_p, y_p)\}_{p=1}^{P}$ y can be continuous (regression) or discrete (classification)
- For simplicity, we adopt the compact notation:

$$\tilde{\boldsymbol{x}}_p = \begin{bmatrix} 1 \\ \boldsymbol{x}_p \end{bmatrix} \quad \tilde{\boldsymbol{w}} = \begin{bmatrix} b \\ \boldsymbol{w} \end{bmatrix}$$

# Cost function

- For a predictive model the cost function can be written as:

$$g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} h(\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{x}}_p)$$

- Example, Least Squares Regression:

$$g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} \left( \tilde{\boldsymbol{x}}_p^T \tilde{\boldsymbol{w}} - y_p \right)^2$$

- Example, Softmax perceptron cost:

$$g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} \log \left( 1 + e^{-y_p \tilde{\boldsymbol{x}}_p^T \tilde{\boldsymbol{w}}} \right)$$

# Gradient of the cost function $g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} h(\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{x}}_p)$

- With the Softmax cost:

$$g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} \log\left(1 + e^{-y_p \tilde{\boldsymbol{x}}_p^T \tilde{\boldsymbol{w}}}\right)$$

- The gradient is given by:

$$\nabla g(\tilde{\boldsymbol{w}}) = \sum_{p=1}^{P} \nabla h(\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{x}}_p) = -\sum_{p=1}^{P} \sigma(-y_p \tilde{\boldsymbol{x}}_p^T \tilde{\boldsymbol{w}}) y_p \tilde{\boldsymbol{x}}_p$$

Where the *sigmoid* function is: $\sigma(x) = \dfrac{1}{1 + e^{-x}}$

$$\nabla h(\tilde{\boldsymbol{w}}, \tilde{\boldsymbol{x}}_p) = -\sigma(-y_p \tilde{\boldsymbol{x}}_p^T \tilde{\boldsymbol{w}}) y_p \tilde{\boldsymbol{x}}_p$$

# Standard gradient descent

- With any gradient descent-based algorithm
  - We apply an update of the following type:

$$\tilde{\boldsymbol{w}}^k = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla g(\tilde{\boldsymbol{w}}^{k-1}) = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \sum_{p=1}^{P} \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

  - Where $\alpha_k$ is adapted at each step (see later)

- This is referred to as
  - Batch gradient descent: the whole dataset (P data "examples") are used to compute the next update to the parameters
  - Converges well to local minima

# Standard gradient descent

- With any gradient descent-based algorithm
  - We apply an update of the following type:

$$\tilde{\boldsymbol{w}}^k = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla g(\tilde{\boldsymbol{w}}^{k-1}) = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \sum_{p=1}^{P} \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

  - Where $\alpha_k$ is adapted at each step (see later)
- Problems
  - The above equation involves loading into the memory all P data points
  - Computing all the P gradients, summing them
  - And performing a single update of the weight vector from step k-1 to k
  - Recall that if the number of involved variables (dimensions) is large, each update can be difficult (slow) to deal with
  - It is difficult to incorporate new data in an online fashion

# Batch *vs* Stochastic Gradient Descent (1/3)

- Batch gradient descent
  - Gradient computed using entire dataset

$$\tilde{\boldsymbol{w}}^k = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \sum_{p=1}^{P} \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

- Stochastic gradient descent
  - Gradient approximated using a single point at a time

$$\tilde{\boldsymbol{w}}^k = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

# Batch *vs* Stochastic Gradient Descent

$$\tilde{\boldsymbol{w}}^k = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

- Rationale
  - Load one point at a time, sequentially, into the memory
  - Compute the gradient *for this single point p*
  - Apply this point-wise gradient step
  - This entails applying P smaller gradient steps, sequentially
    (one per point, p=1,2,…,P)
  - As opposed to applying a single step

# Batch *vs* Stochastic Gradient Descent

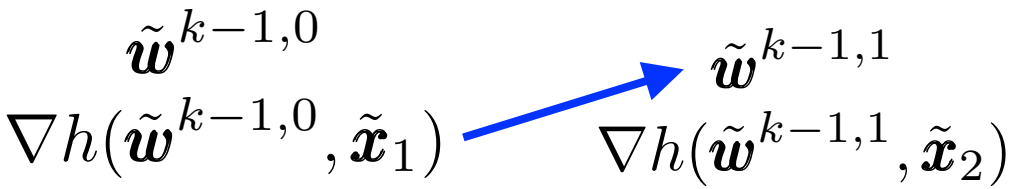

- This is called training epoch

- We go from parameters (weights) at step k-1 to those at step k
  - Through sequential updates
  - Each update involves computing the gradient of a single point

# Batch *vs* Stochastic Gradient Descent

$\tilde{\boldsymbol{w}}^{k-1}$ starting point

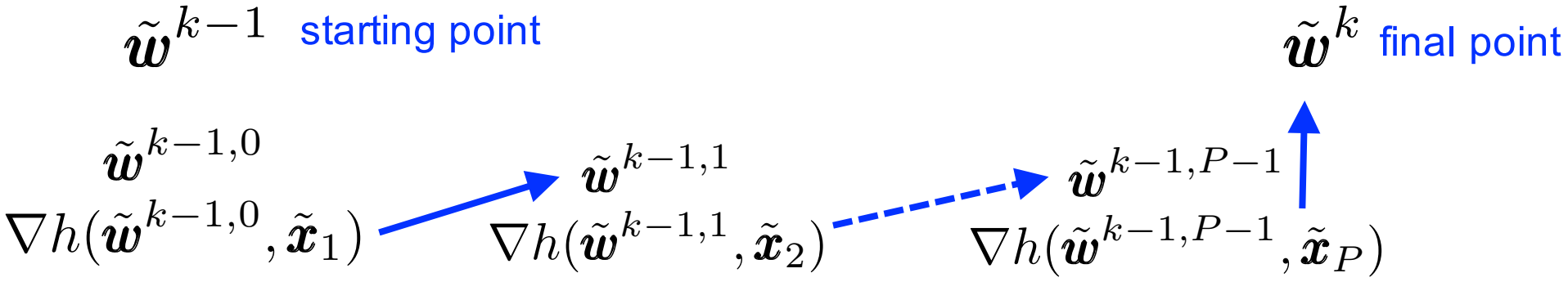
$\tilde{\boldsymbol{w}}^{k}$ final point




$$\tilde{\boldsymbol{w}}^{k,p} = \tilde{\boldsymbol{w}}^{k,p-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k,p-1}, \tilde{\boldsymbol{x}}_p)$$

- We go from parameters (weights) at step k-1 to those at step k
  - Through sequential updates
  - Each update involves computing the gradient of a single point

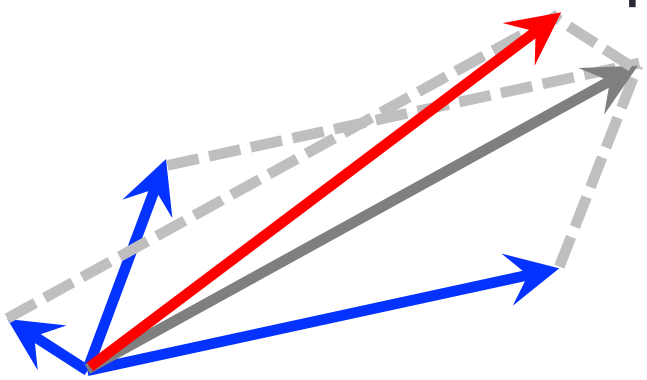
# Batch *vs* Stochastic Gradient Descent




$$\tilde{\boldsymbol{w}}^{k,p} = \tilde{\boldsymbol{w}}^{k,p-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k,p-1}, \tilde{\boldsymbol{x}}_p)$$

- We go from parameters (weights) at step k-1 to those at step k
  - Through sequential updates
  - Each update involves computing the gradient of a single point

# Total vs poitwise-gradients

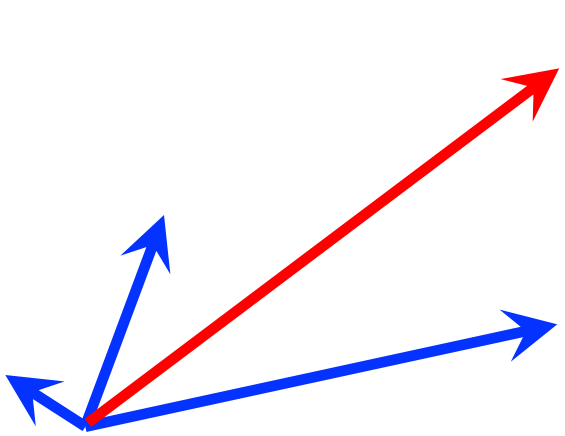


This is the true gradient at time k-1

The tree component (P=3) of the current gradient at time k-1

$$\tilde{\boldsymbol{w}}^{k} = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla g(\tilde{\boldsymbol{w}}^{k-1})$$

# Total vs poit-wise gradients

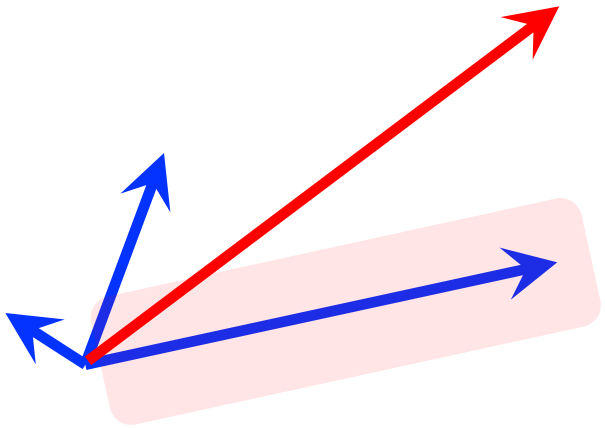


This is the true gradient at time k-1

- The true gradient goes in the right direction (stationary point)
- The point-wise gradients
  - Summing them leads to the right gradient
  - But, point-wise they may steer the search “randomly”

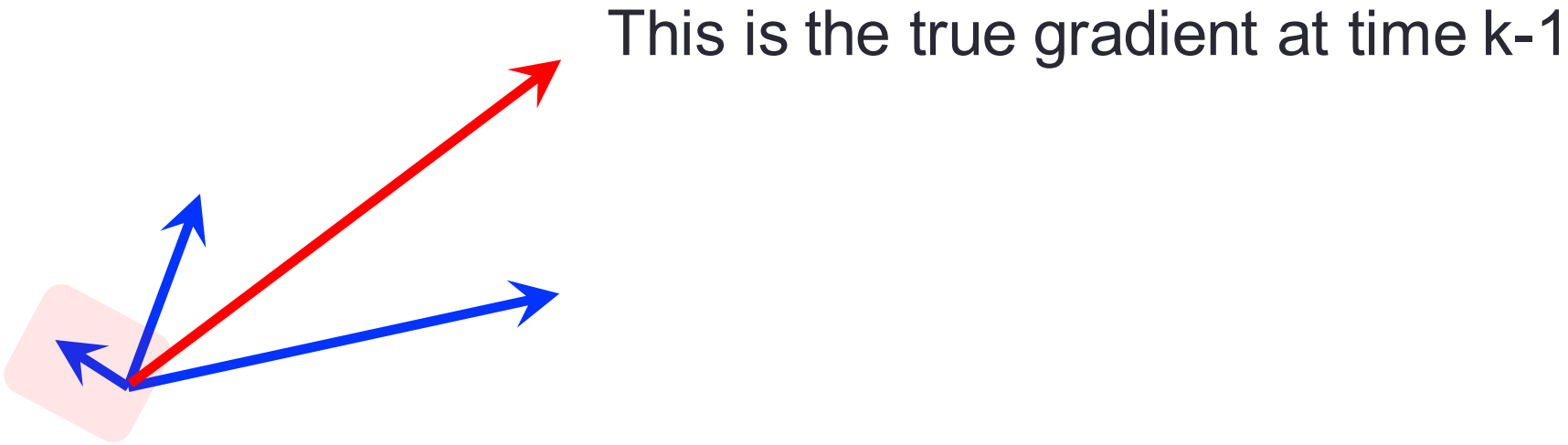
# Total vs poit-wise gradients

This is the true gradient at time k-1



- Point-wise gradients
  - The dominant direction should be the largest

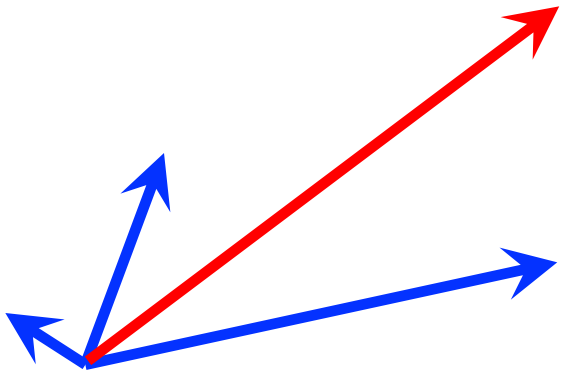
# Total vs poit-wise gradients




- Point-wise gradients
  - But sometimes a local gradient may move the solution away from the stationary point
  - This is often desirable, as it introduces some random behavior in the search

# Total vs poit-wise gradients

This is the true gradient at time k-1



- Note that
  - There are no guarantees that sequential local updates will lead, at the end, to the true gradient (red arrow)
  - This is why when we compute a point-wise update, we also modify the weight vector, and the next point-wise gradient is obtained from the new weight coordinates

# Stochastic Gradient Descent algorithm

$$\tilde{\boldsymbol{w}}^{k} = \tilde{\boldsymbol{w}}^{k-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k-1}, \tilde{\boldsymbol{x}}_p)$$

- The SGD algorithm (initialize with p=1):
  - 1) Load point with index p into the memory $\tilde{\boldsymbol{x}}_p$
  - 2) Compute the gradient for the current point (index p): $\nabla h(\tilde{\boldsymbol{w}}^{k,p-1}, \tilde{\boldsymbol{x}}_p)$
  - 3) Apply the update for this single point, using its gradient:

$$\tilde{\boldsymbol{w}}^{k,p} = \tilde{\boldsymbol{w}}^{k,p-1} - \alpha_k \nabla h(\tilde{\boldsymbol{w}}^{k,p-1}, \tilde{\boldsymbol{x}}_p)$$

  - 4) Discard the current point from memory, set p = p+1
  - 5) If p=P+1 stop, the new weight (output) is $\tilde{\boldsymbol{w}}^{k,P} = \tilde{\boldsymbol{w}}^{k+1,0}$

    ELSE If p<P+1 **go to 1)**

**Each step k → k+1 involves P sequential (smaller) updates**

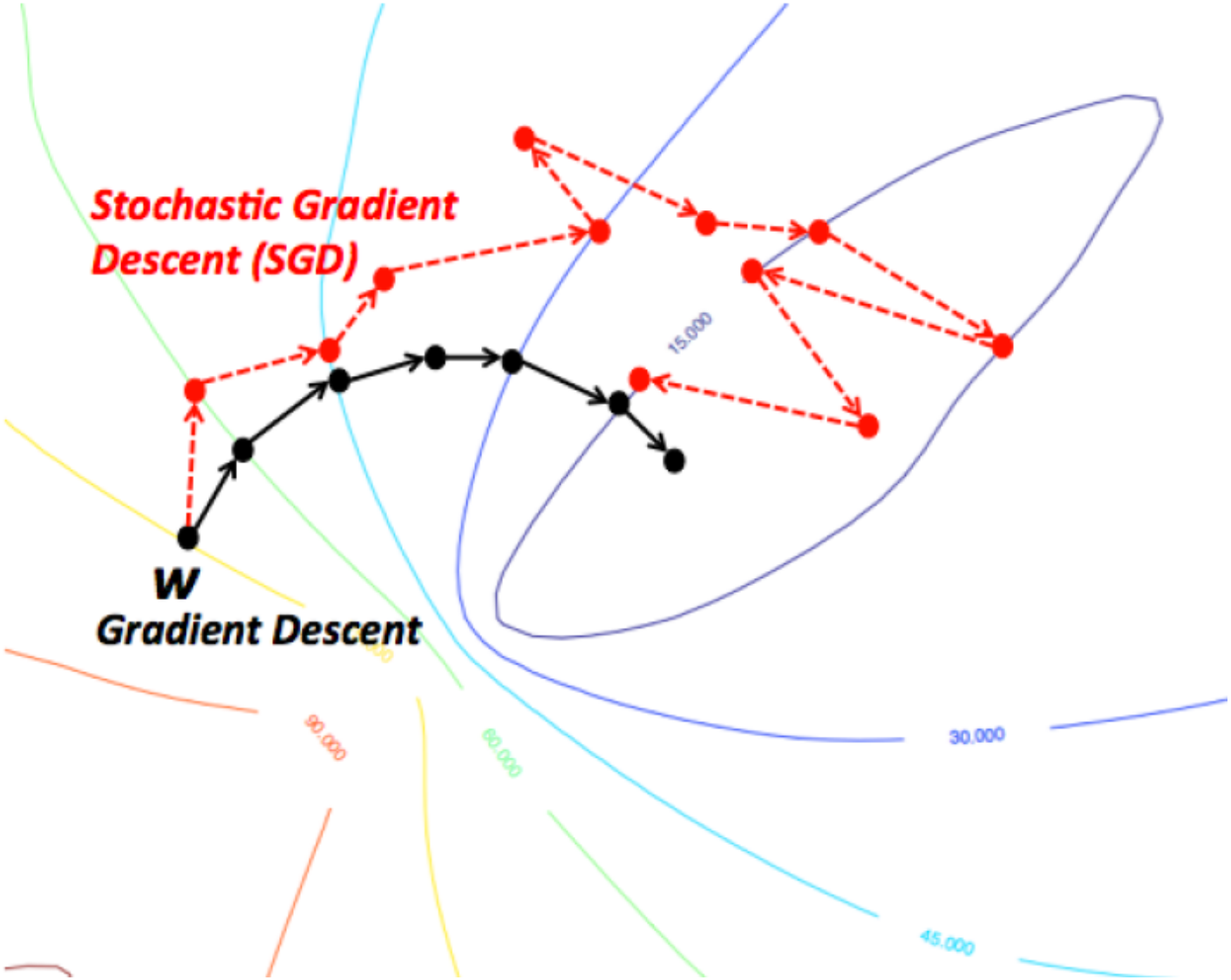
# SGD: observations

- The learning rate $\alpha$ is typically smaller than that in the standard (batch) algorithm
  - As with SGD there is much more variance in the update
- The order in which we pick the points for the sequential updates is important → will lead to different search paths
- In practice
  - A not too small learning rate is used at the beginning
  - The learning rate is set to a smaller value as convergence slows down
    - Probably, we are getting closer to a stationary point
    - Reducing the learning rate decreases the variance
  - Not all points are used for the training epochs

# SGD: selection of points and order

- Specific sequences of points
  - May lead to poor performance
  - Moving the solution away from the minimum
  - Picking the right (static) schedule is fairly costly (search)
- To prevent this: prior to each training epoch
  - Points are randomly shuffled
  - This generally leads to good convergence

# Intuitively…



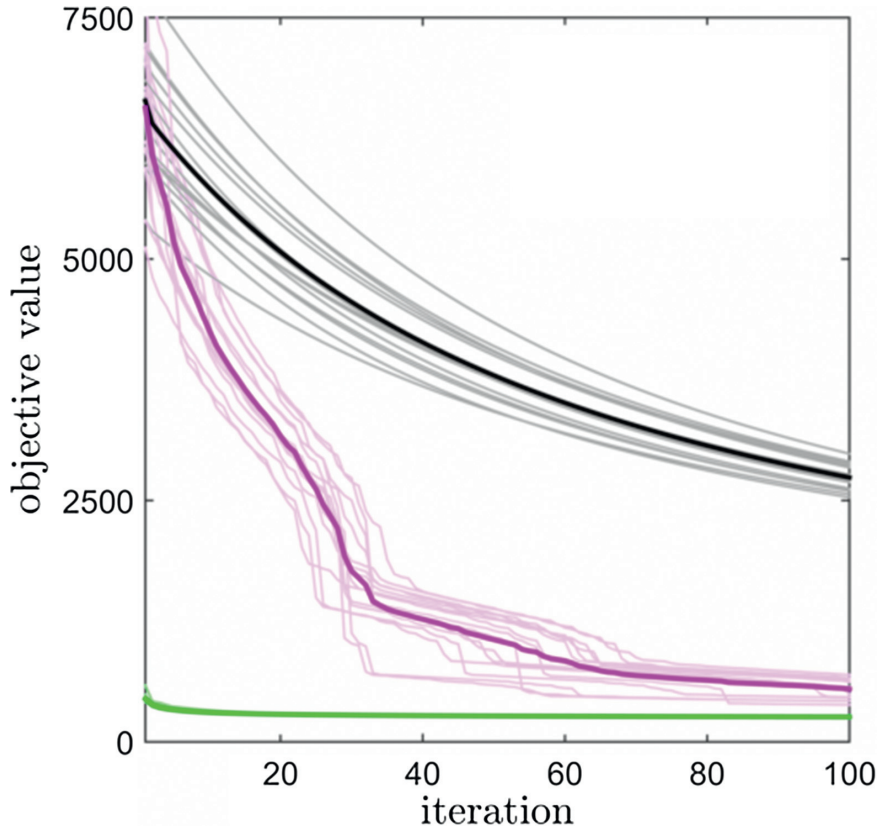
# Classification Example (1/2)

- Large dataset with P=10,000
- Two class classification problem $\{(\boldsymbol{x}_p, y_p)\}_{p=1}^{P}$

- Softmax cost used to perform logistic regression
- Data generated for the task of face detection
  - 3,000 facial images
  - 7,000 non-face images
  - For SGD we use decreasing step lengths $\alpha_k = \frac{1}{k}$

- Labels are $y_p \in \{-1, +1\}$
  - +1: the image contains a face
  - -1: the image does not contain a face

# Classification Example (2/2)




The objective value of the first 100 iterations of stochastic gradient descent (shown in green), compared with standard gradient descent with conservatively optimal fixed step length (black) and adaptively chosen step length (magenta). In this instance a real two class classification dataset is used consisting of $P = 10\,000$ points (see text for further details). 15 runs of each method are shown as the lighter colored curves, with their averages shown in bold. Here the stochastic gradient descent scheme is massively superior to both standard gradient methods in terms of the rapidity of its convergence.

# On the step length for SGD (1/2)

- A step rule satisfying the following two requirements is guaranteed to make SGD converge to a stationary point

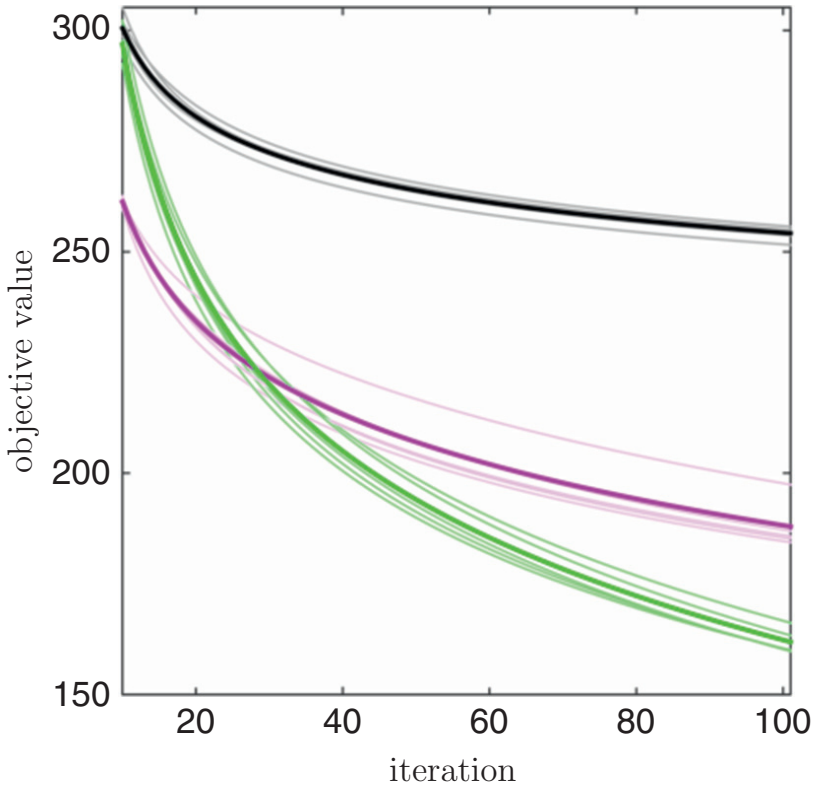
(1) The step size must diminish as the number of iterations increases:
$\alpha_k \longrightarrow 0$ as $k \longrightarrow \infty$.

(2) The sum of the step sizes is not finite: i.e., $\sum_{k=1}^{\infty} \alpha_k = \infty$.

- Common choices are: $\alpha_k = \frac{1}{k}$ $\alpha_k = \frac{1}{\sqrt{k}}$

- Observation: while a decreasing step length ensures convergence, in practice a constant step length can also be used (convergence is no longer ensured in this case) and in many cases this leads to faster learning. It has to be verified experimentally, on the given dataset, however

# On the step length for SGD (2/2)




The objective value of the first $k = 1$–$100$ iterations of three versions of stochastic gradient descent with diminishing step-size rules, $\alpha_k = \frac{1}{k}$ and $\alpha_k = \frac{1}{\sqrt{k}}$, and a constant step size $\alpha_k = \frac{1}{3}$ for all $k$ (shown in light black, magenta, and green respectively, with the average of the runs shown in bold of each color). The two-class classification dataset used here consists of $P = 10\,000$ data points (see text for further details). In this instance the constant step size-runs tend to outperform those governed by the provably convergent diminishing step-size rules.

# References

Reference book:

[1] J. Watt, R. Borhani, A. K. Katsaggelos, “Machine Learning Refined: Foundations, Algorithms and Applications,” Cambridge University Press 2016.

Chapter 8 “Advanced gradient descent”

# TOOLS FOR LEARNING: FUNDAMENTALS OF NUMERICAL OPTIMIZATION – PART 2

Michele Rossi

rossi@dei.unipd.it

Dept. of Information Engineering

University of Padova, IT