Data mining & Machine Learning

CS 373
Purdue University

Dan Goldwasser
dgoldwas@purdue.edu

Today's Lecture

Learning as Optimization

- *Our discussion of learning so far focused on specific algorithms*
- *Today, we'll discuss a new way of thinking about learning – optimizing an objective function, consisting of both a data-fitting term (performance on training data) and generalization term (e.g., margin).*
- *We will start with finding the best margin classifier*
- *Then, we will extend this framework to include more general terms for controlling overfitting*

Perceptron In Practice

- The perceptron algorithm is actually a pretty good practical algorithm.
  - It’s also very simple to implement and modify.

In the next slides we will introduce some design choices and understand their implication on how Perceptron works.

(Realistic) Perceptron

- We learn $\mathbf{f:X} \rightarrow \{-1,+1\}$ represented as $\mathbf{f} = \mathrm{sgn}\{\mathbf{w} \bullet \mathbf{x})$
- Where $\mathbf{X}=\{0,1\}^n$
- Given Labeled examples: $\{(\mathbf{x}_1,\mathbf{y}_1), (\mathbf{x}_2,\mathbf{y}_2), \ldots (\mathbf{x}_m,\mathbf{y}_m)\}$

1. Initialize w=0 $\in \mathbf{R}^n$
2. **For Iter = 0,…,T**
   3. Iterate over all the examples
      a. Predict the label of instance x to be y' = sgn{w•x)
      b. If ***y'≠y***, update the weight vector:
         w = w + r y x (r - a constant, learning rate)
         Otherwise, if **y'=y**, leave weights unchanged.

Regularization: Perceptron with Margin

- *Weights with better margin generalize better*
  - Perceptron finds *any* separating hyperplane
- **Thick Separator (aka as Perceptron with Margin)**
  - **Predict positive**

bias term

$$w x - \theta > \gamma$$

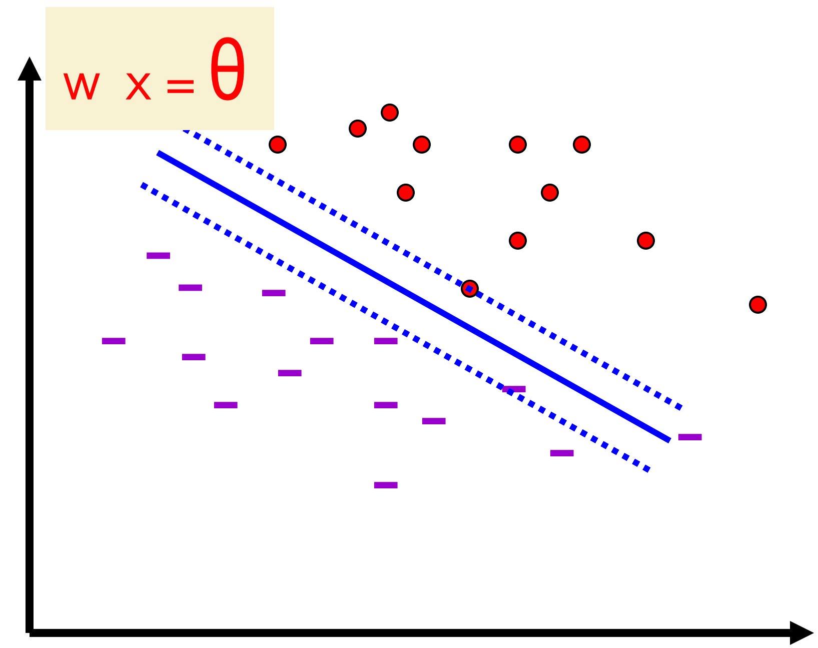
margin

  - **Predict negative**

$$w x - \theta < -\gamma$$

  - **Mistake:**

$$\gamma > (w x - \theta) > -\gamma$$



Regularization: Perceptron with Margin

- **Perceptron margin** : hyperparameter that has to be tuned using the validation set (try different values)
  - *In the future: the data will decide the margin*
- The impact of margin regularization in perceptron becomes smaller as *w* grows

$$w\,x - \theta > \gamma$$

what happens if we multiply the ***weights*** *by 2?* ***by 2000?***

  - ***Can we control the growth of w?***
- In practice, **very effective**

Perceptron: Robust Variation

- *The perceptron algorithm counts later points more than earlier points*

1: (0,1,..,1,0,1)

2: (0,1,..,1,0,1)

…

100: (0,1,..,1,0,1)

…

10000: (0,1,..,1,0,1)

Makes some mistakes, update..
After 100 examples, learner stops making mistakes

We keep going…

BUT then at the 10,000 example the learner makes a mistake!

Is this a problem?

Voted Perceptron

- *Training:*
  - *Learner remembers how long each hypothesis survived (no mistakes on w)*
- Test:
  - Weighted vote of all participating hypotheses

BIG Idea in ML:
reduce variance using classifier ensemble

$$\hat{y} = \operatorname{sign}\left(\sum_{k=1}^{K} c^{(k)} \operatorname{sign}\left(w^{(k)} \cdot \hat{x} + b^{(k)}\right)\right)$$

- Heavy: (1) Computational effort (2) **Storage**

Averaged Perceptron

- **Training**: *Maintain a running weighted average of survived hypotheses*
- **Test**: *Predict according to the averaged weight vector*

Voted Perceptron $\longrightarrow$ $$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \text{sign}\left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right)$$

$$\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c^{(k)} \left(\boldsymbol{w}^{(k)} \cdot \hat{\boldsymbol{x}} + b^{(k)}\right)\right)$$ $\longleftarrow$ Averaged Perceptron

- *An efficient approximation of voted perceptron*
- *Almost always better than regular perceptron!*

Understanding Weighted Perceptron

$D = \{x_i, y_i\}_{i=1,\ldots,n}$
$w_0 = (0,\ldots,0)$
$t=0$
repeat T times
    for (x_i, y_i) in D
    $y' = \text{sign}(w_t x)$ prediction based on the **current** model
    if $(y' != y)$
        $w_{t+1} = w_t + xyr$ Update Rule
    else
        $w_{t+1} = w_t$
    $t = t+1$
return $(w_1 + \ldots + w_{nT})/(nT)$ return the averaged result

key idea: represent the averaged function **explicitly** by keeping in memory all the functions visited by the algorithm.

Efficient Weighted Perceptron

- We can express the learned function as a sum of all its updates:

$$
\begin{aligned}
w_0 &= (0,\ldots,0) \\
w_1 &= w_0 + \Delta_1 = \Delta_1 \\
w_2 &= w_1 + \Delta_2 = \Delta_1 + \Delta_2 \\
w_3 &= w_2 + \Delta_3 = \Delta_1 + \Delta_2 + \Delta_3
\end{aligned}
$$

...

- This allows us to express the average of the classifiers in terms of updates:

$$
\begin{aligned}
&(w_1 + w_2 + w_3)/3 \\
&= (\Delta_1)/3 + (\Delta_1 + \Delta_2)/3 + (\Delta_1 + \Delta_2 + \Delta_3)/3 \\
&= (3/3)\Delta_1 + (2/3)\Delta_2 + (1/3)\Delta_3
\end{aligned}
$$

Efficient Weighted Perceptron

$D = \{x_i, y_i\}_{i=1,...,n}$

$w = (0,...,0)$ **current** function weights

$a = (0,...,0)$ counter of all the **updates** seen so far

step= nT

repeat T times

    for (x_i,y_i) in D

        $y' = \text{sign}(wx)$ prediction based on **current** model

    if $(y' != y)$

        $w = w + xyr$ update Rule

        $a = a + (\text{step}/nT)(xyr)$ update the weight counter

    step = step - 1

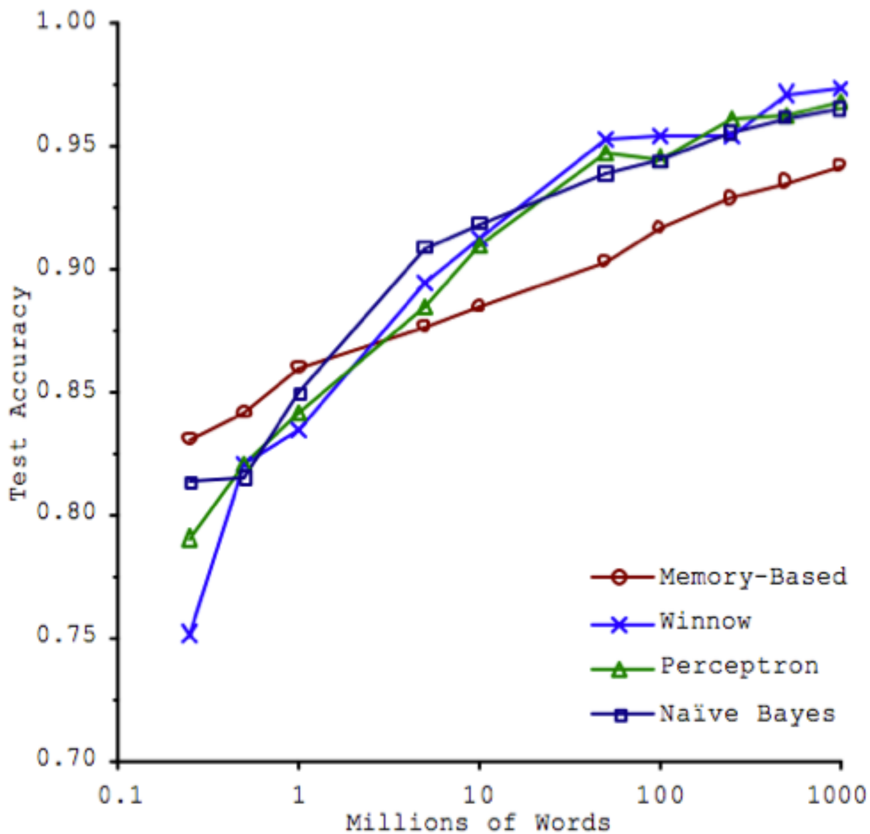
return a return the averaged result

key idea: represent the averaged function as averaged updates, instead of explicitly keeping in memory all the functions visited by the algorithm.

Practical Example

Task: ***context sensitive spelling***

“I didn’t know {weather,whether} to laugh or cry”



Source: Scaling to very very large corpora for natural language disambiguation Michele Banko, Eric Brill. MSR, 2001.

Learning as Optimization

Learning as Optimization

- The perceptron algorithm is a mistake driven algorithm.
  - The model is updated in response to an error
- In that sense, it is minimizing the number of mistake on the training data.
  - Does it provide a globally optimal result?
- In the next couple of lectures we will make the optimization process explicit, and essentially equate learning with an optimization objective.
  - **Key advantage**: "one size fits all": if you can optimize a function, the difference between the algorithms is minimal!

Searching over models/patterns

- Consider a **space** of possible models $M=\{M_1, M_2, ..., M_k\}$ with parameters θ

- Search could be over model structures or parameters, e.g.:

  - **Parameters**: In a linear regression model, what are regression coefficients (**β**) that minimize squared loss on the training data?

  - **Model structure**: In a decision trees, what is the tree structure that minimizes 0/1 loss on the training data?

Optimization

- **Non-smooth** functions:
  - If the function is *discrete*, then traditional optimization methods that rely on smoothness are not applicable. Instead we need to use **combinatorial optimization**

  - ***Example***: *Choosing what features (structure) to add to a decision tree*

  - *You need to define the search space and a search procedure.*
    - ***Formally:*** *State Space (including Initial State, Final States), Transition Function (i.e., actions), Scoring Function.*

Search algorithms for discrete spaces

- **Conduct the search by:**
  - Considering a particular state (*model*)
  - Testing to see if it is the goal state (*model with maximum score*)
  - And if not, expand the current state to generate successor states by applying all possible actions
    (*determine alternative models to consider next*)

Search strategies differ in their choice of how to expand states

Optimization

- **Smooth** functions:
  - If a function is ***smooth***, it is differentiable and the derivatives are continuous, then we can use gradient-based optimization
    - If function is ***convex***, we can solve the minimization problem in closed form: ⍰S(θ) using **convex optimization**
    - If function is smooth but **non-convex**, we can use iterative search over the surface of S to find a local minimum (e.g., hill-climbing)

Convex optimization problems

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in C \end{array}$$

- Where ***f*** is a convex function (*score function*)
  C is a convex set (*constraints on model parameters or structure*)
  x is the optimization variable (*includes data and parameters*)

- For convex optimization problems, all locally optimal points are globally optimal

- **Example algorithms**: Quadratic programming (SVMs), least squares estimation, *maximum likelihood estimation*

Loss functions

- To formalize performance let's define a **loss function**: $loss(y, \hat{y})$
    - Where $\hat{y}$ is the gold label
- The loss function measures the error on a single instance
    - Specific definition depends on the learning task

Regression

y = w^T^x

$$loss(y, \hat{y}) = (y - \hat{y})^2$$

Binary classification

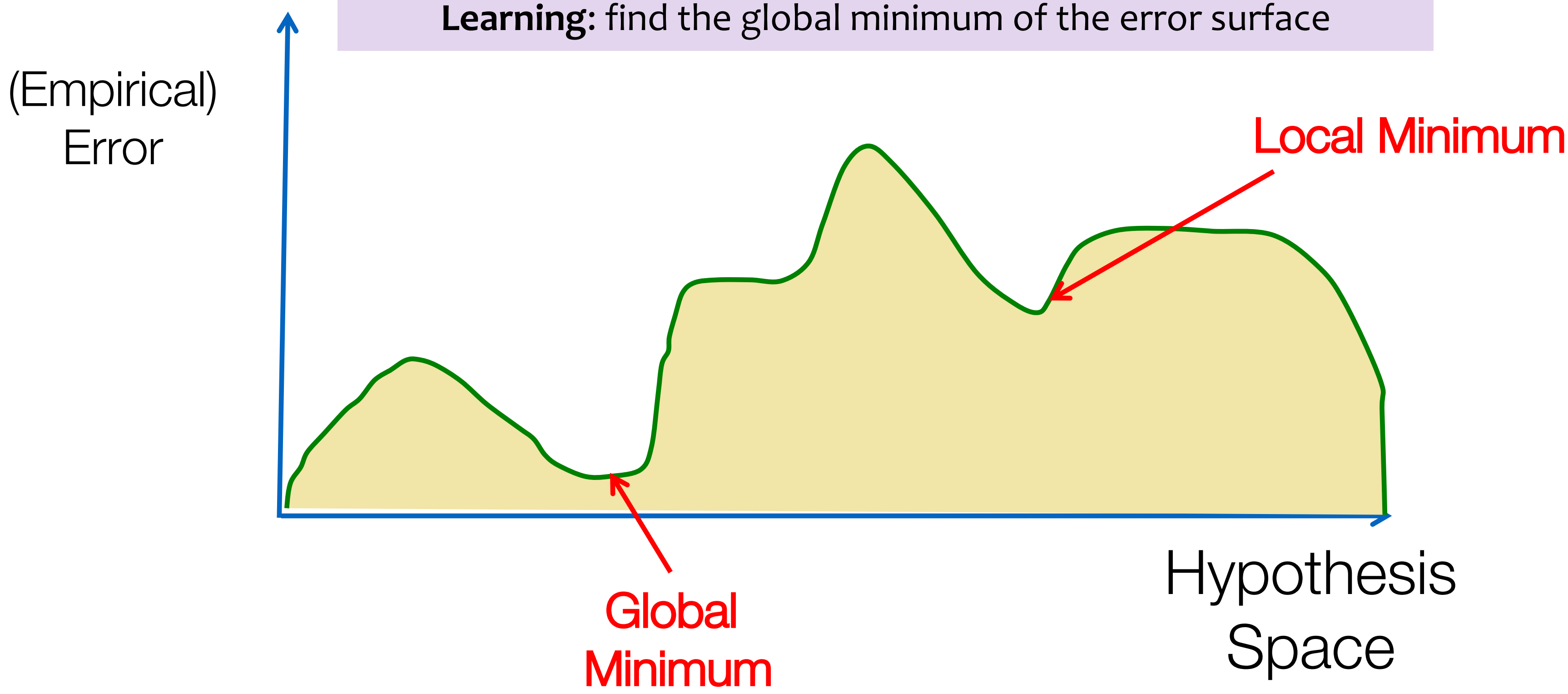
y = sign(w^T^x)

$$loss(y, \hat{y}) = \begin{cases} 0 & y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

Error Surface

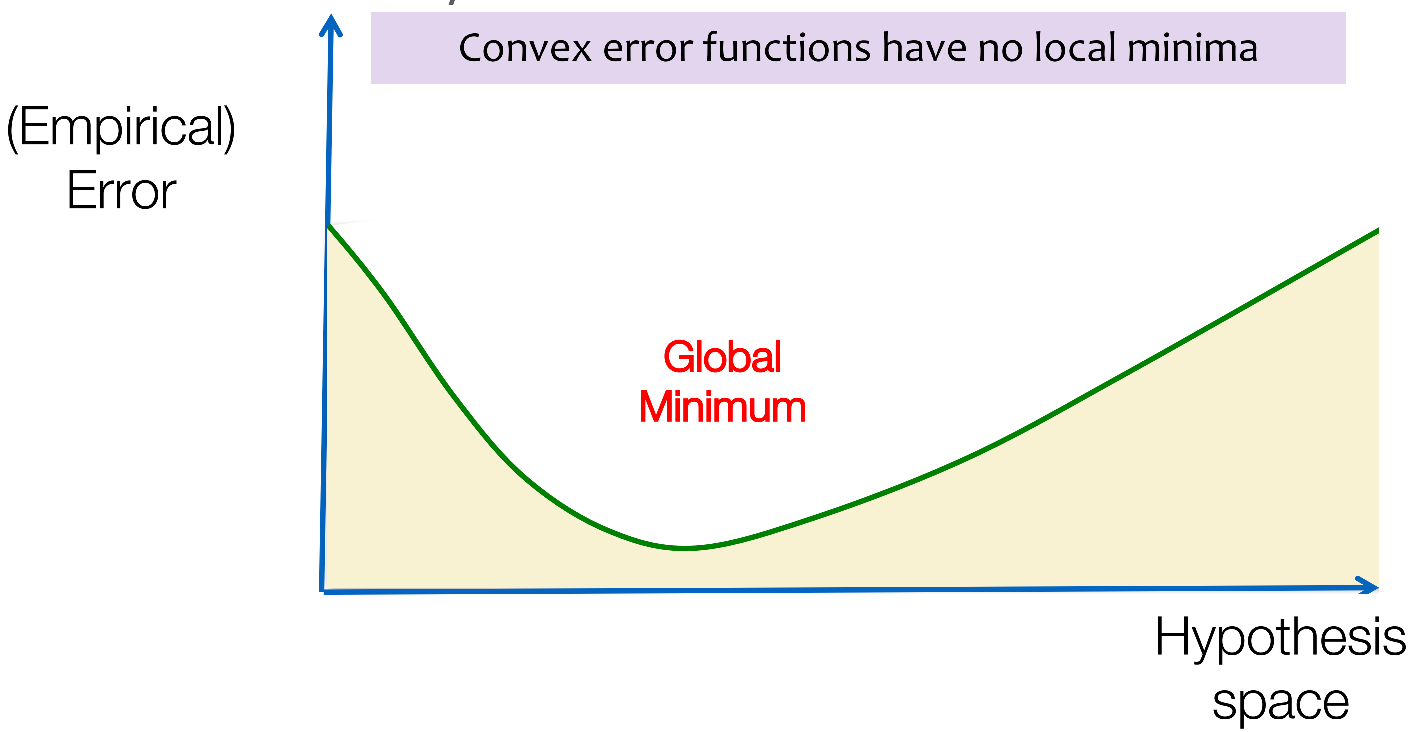
- **Linear classifiers**: hypothesis space parameterized by w
- *Error/Loss/Risk* are all functions of w

Learning: find the global minimum of the error surface



Convex Error Surfaces

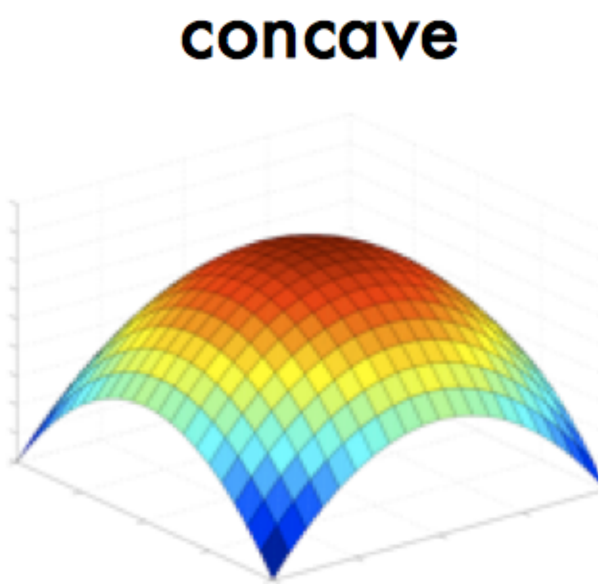
- **Convex functions** have a single minimum point
  - Local minimum = global minimum
  - *Easier to optimize*

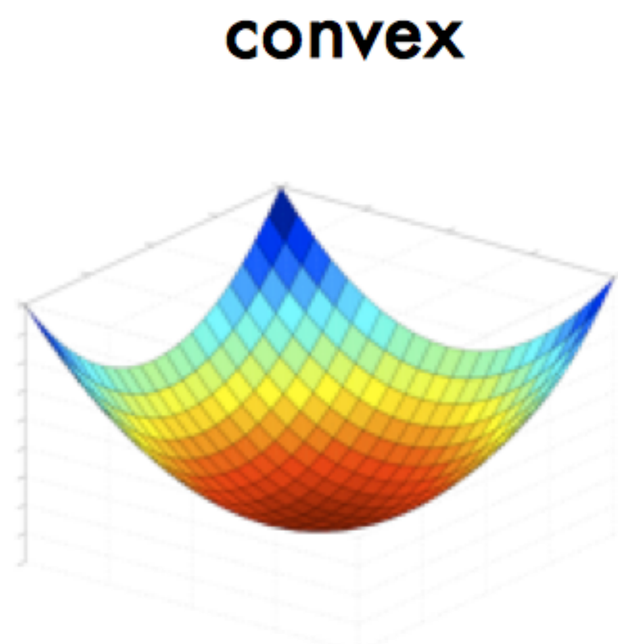


Convex optimization

Concave vs convex

- Maximizing a concave function is equivalent to minimizing a convex function





Score function: Likelihood

- Let $D = \{x(1), ..., x(n)\}$
- Assume the data *D* are independently sampled from the same distribution: $p(X|\theta)$
- The likelihood function represents the probability of the data as a function of the model parameters:

$$
\begin{aligned}
L(\theta|D) &= L(\theta|x(1), ..., x(n)) \\
&= p(x(1), ..., x(n)|\theta) \\
&= \prod_{i=1}^{n} p(x(i)|\theta)
\end{aligned}
$$

If instances are independent, likelihood is product of probs

Maximum likelihood estimation

- Most widely used method of parameter estimation
- "Learn" the best parameters by finding the values of θ that maximizes likelihood:

$$\hat{\theta}_{MLE} = \arg\max_{\theta} L(\theta)$$

- Often easier to work with loglikelihood:

$$\begin{aligned} l(\theta|D) &= log\, L(\theta|D) \\ &= log \prod_{i=1}^{n} p(x(i)|\theta) \\ &= \sum_{i=1}^{n} logp(x(i)|\theta) \end{aligned}$$

Maximum likelihood estimation

- Define likelihood, take derivative, set to 0, and solve
- **Example**
    - Toss a weighted coin 100 times, observe 30 heads
    - What is the MLE estimate for the *p* parameter of the Binomial distribution that generated the data?
    - First define likelihood

$$\begin{aligned} L(p|H=30, n=100) &= P(H=30|n=100, p) \\ &= \binom{100}{30} p^{30}(1-p)^{70} \end{aligned}$$

Take derivative, set to 0, solve

$$\begin{aligned}
0 &= \frac{d}{dp}(\binom{100}{30}p^{30}(1-p)^{70}) \\
&\propto 30p^{29}(1-p)^{70} - 70p^{30}(1-p)^{69} \\
&= p^{29}(1-p)^{69}[30(1-p) - 70p] \\
&= p^{29}(1-p)^{69}[30 - 100p] \\
p &= 0, 1, \frac{30}{100}
\end{aligned}$$

Gradient descent

- For some convex functions, we may be able to take the derivative, but it may be difficult to directly solve for parameter values
- **Solution**:
  - Start at some value of the parameters
  - Take derivative and use it to move the parameters in the direction of the solution
  - Repeat

<u>Gradient Descent Rule:</u>

$$\underline{\mathbf{w}}_{\mathbf{new}} = \underline{\mathbf{w}}_{\mathbf{old}} - \eta \; \Delta\,(\underline{\mathbf{w}})$$

where

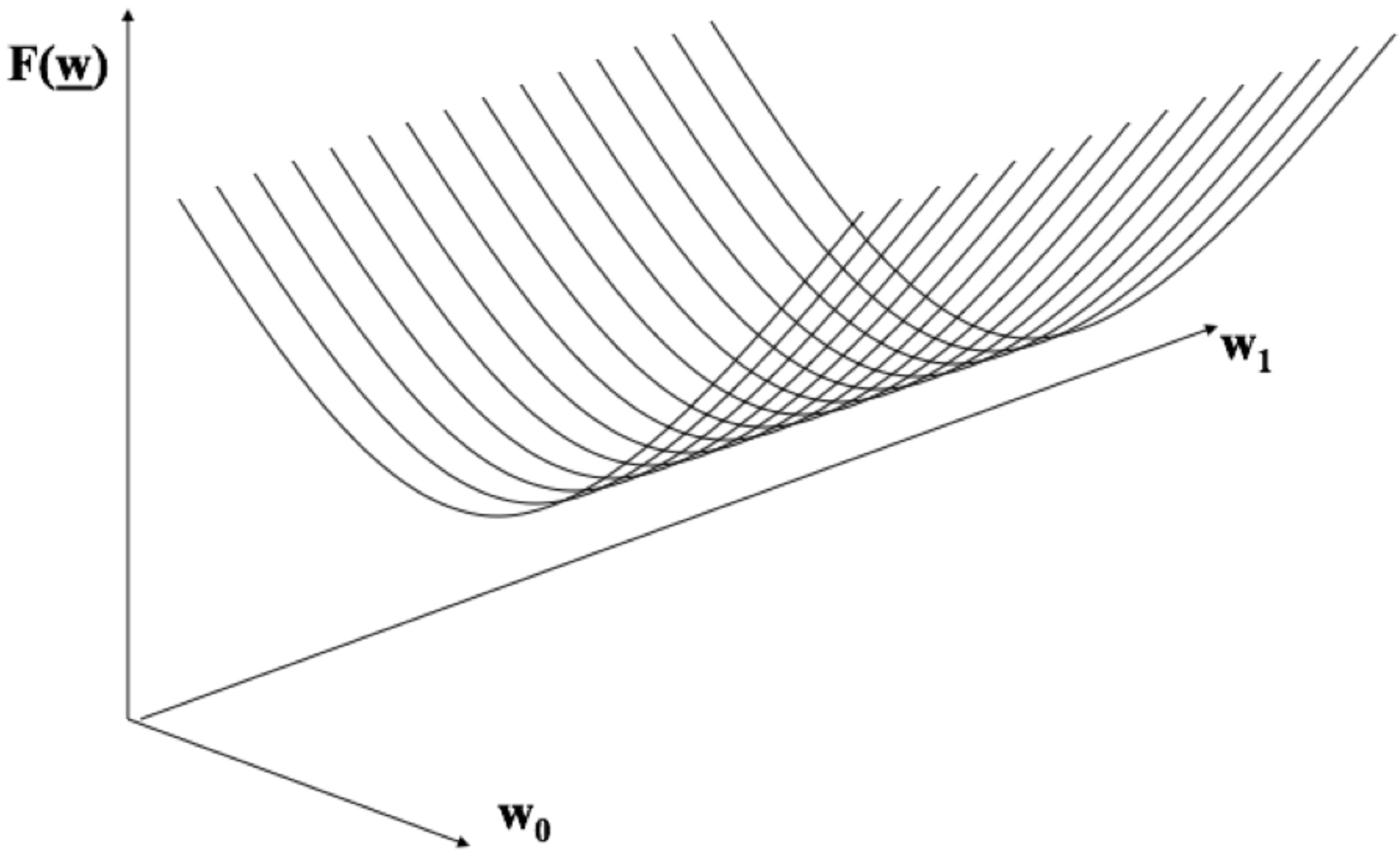
$\Delta\,(\underline{w})$ is the gradient and

η is the learning rate (small, positive)

Notes:

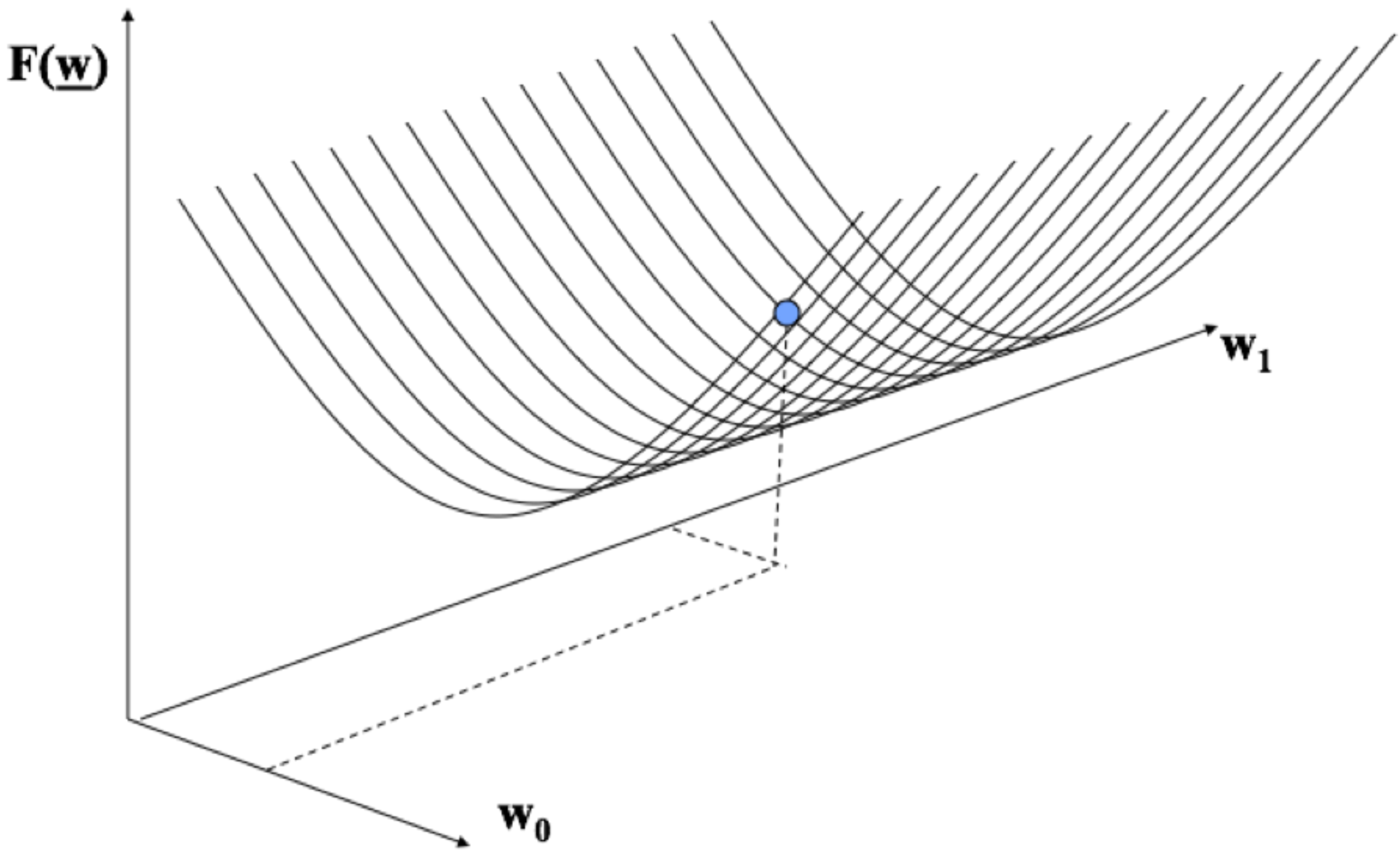
1. This moves us downhill in direction $\Delta\,(\underline{\mathbf{w}})$ (steepest downhill direction)
2. How far we go is determined by the value of η

Illustration of gradient descent



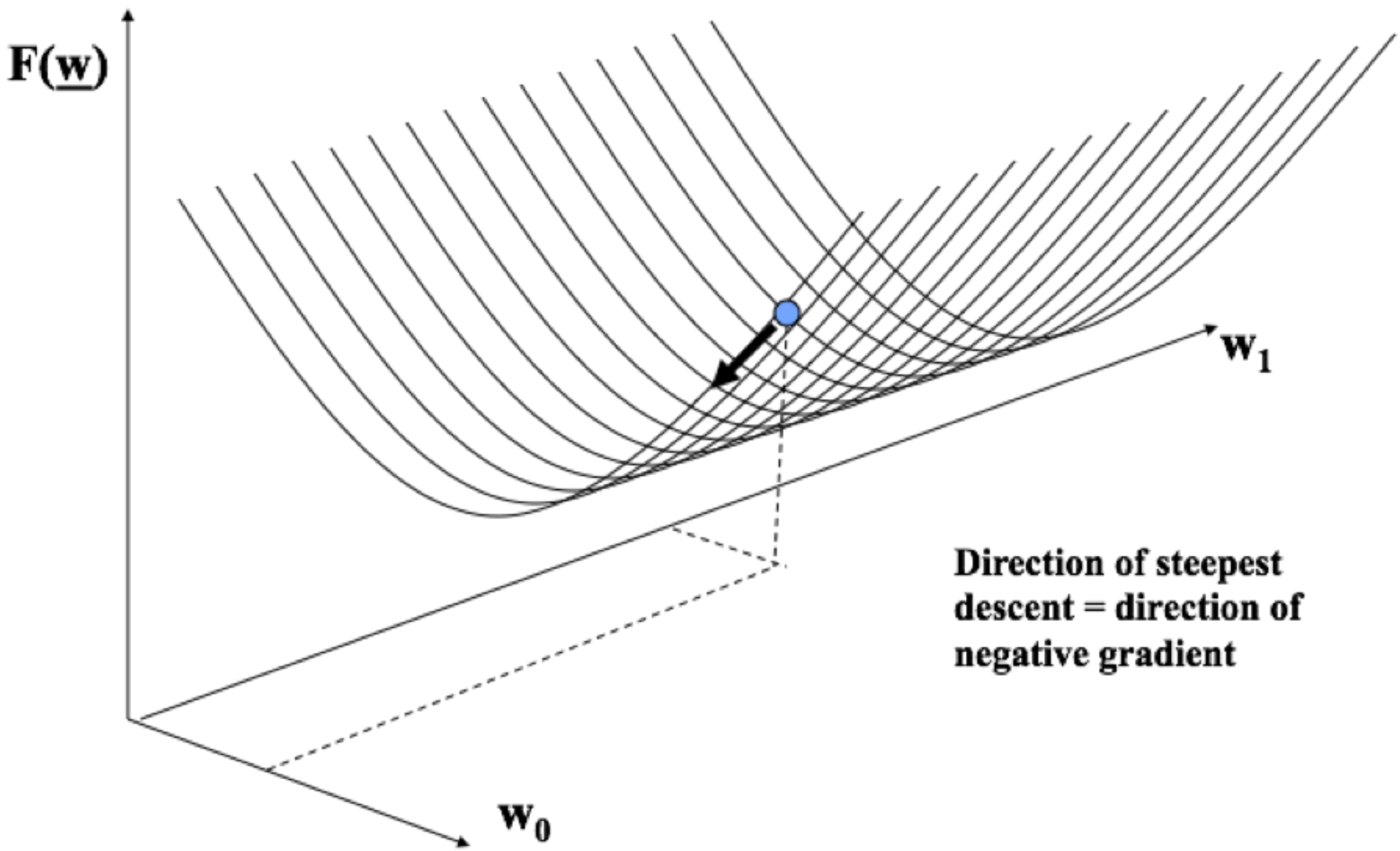
Slides adapted from CS175, UCIrvine, Padhraic Smyth

Illustration of gradient descent



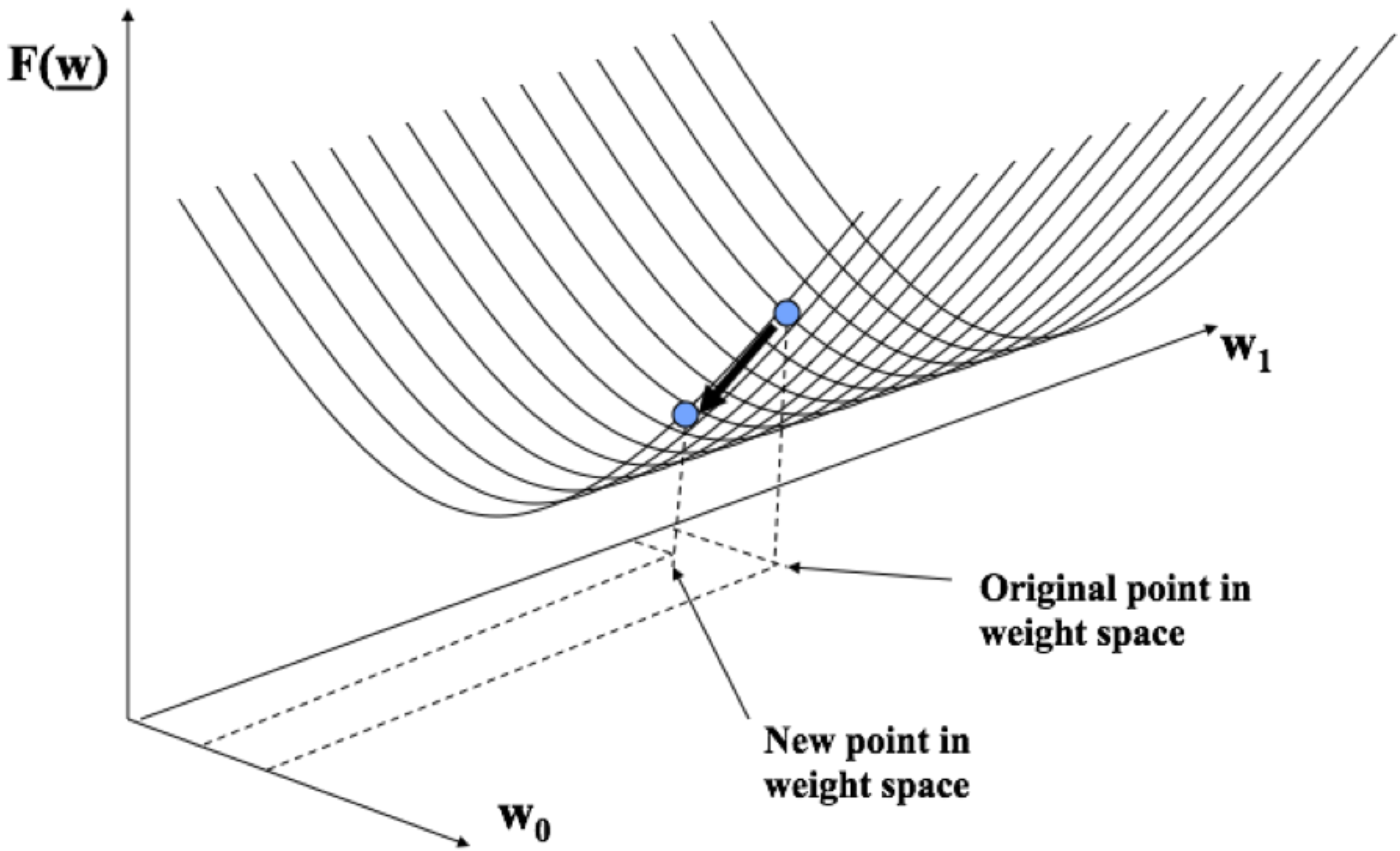
Slides adapted from CS175, UCIrvine, Padhraic Smyth

Illustration of gradient descent



Slides adapted from CS175, UCIrvine, Padhraic Smyth

Illustration of gradient descent

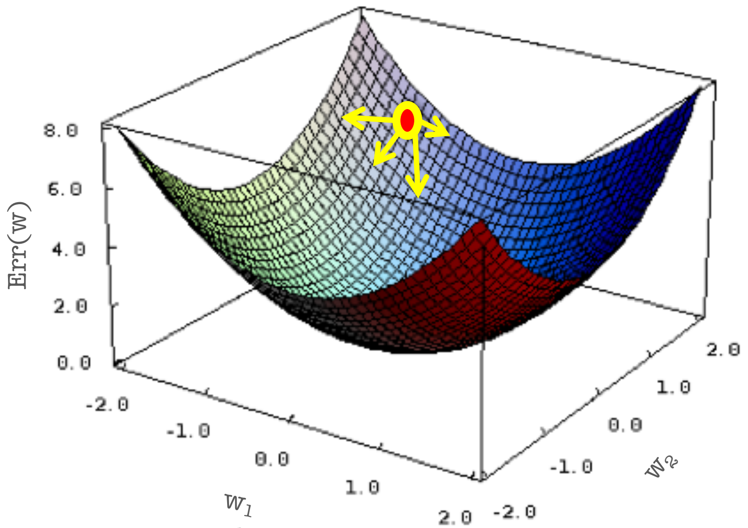


Error Surface for Squared Loss

We add the ½ for convenience

$$Err(\mathrm{w}) = \frac{1}{2}\sum_{d \in D}(\hat{y}_d - y_d)^2$$

$$y = \mathrm{w}^T x$$



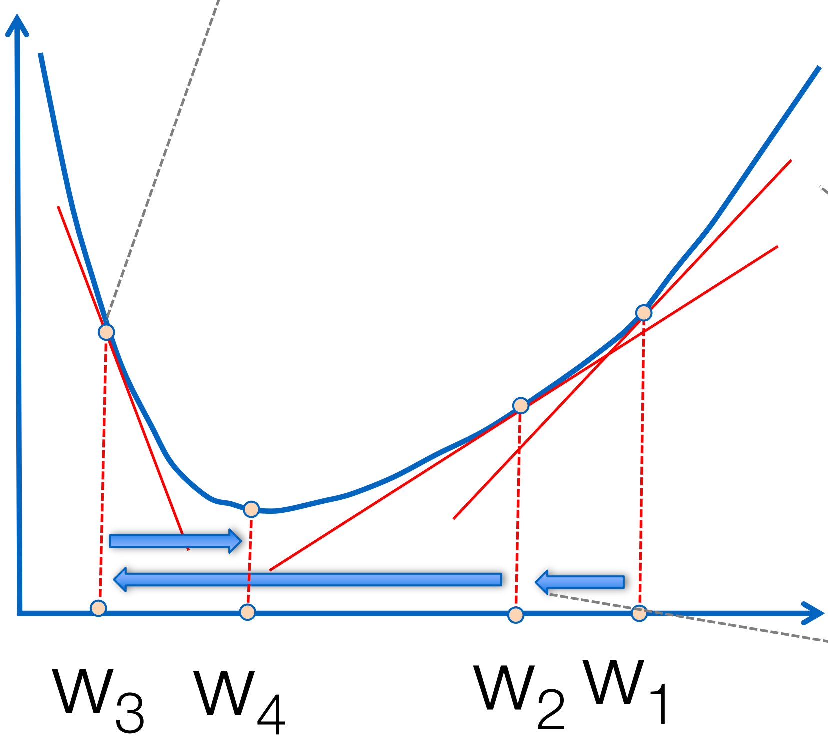
Since $\hat{y}$ is a constant (for a given dataset), the Error function is a *quadratic function* of W (paraboloid)

➔ Squared Loss function is convex!

How can we find the global minimum?

Gradient Descent Intuition

(3) We also need to determine the step size (aka learning rate).
What happens if we overshoot?



(1) The derivative of the function at w_1 is the slope of the tangent line
→ Positive slope (increasing)
Which direction should we move to decrease the value of Err(w) ?

(2) The gradient determines the direction of steepest increase of Err(w) *(go in the opposite direction)*

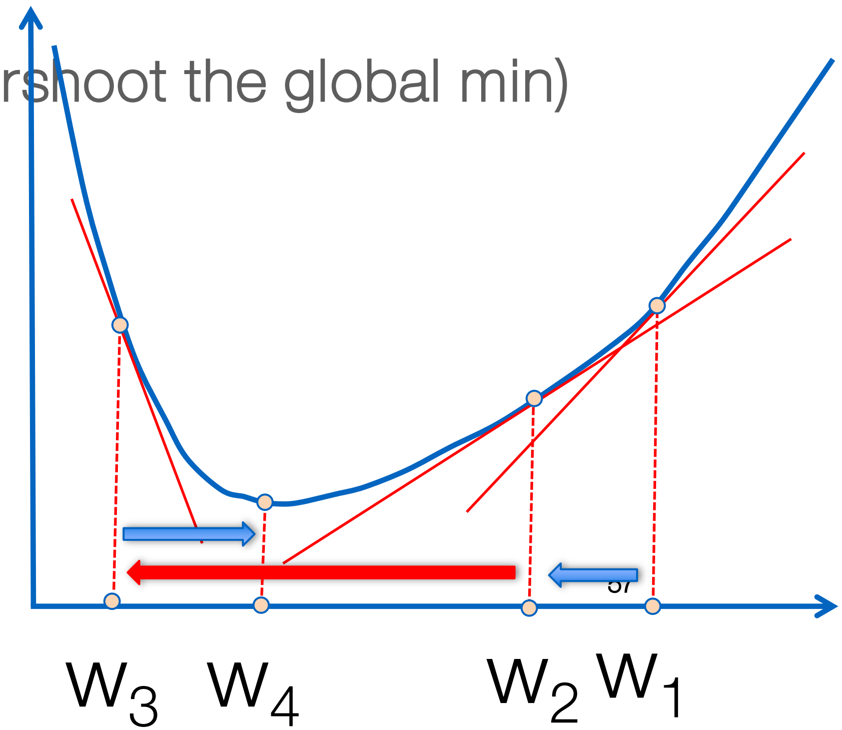
What is the gradient of Error(w) at this point?

Note about GD step size

- Setting the step size to a very small value
    - Slow convergence rate
- Setting the step size to a large value
    - May oscillate (consistently overshoot the global min)
- Tune experimentally
    - More sophisticated algorithm

      set the value automatically

      (conjugate gradient)



The Gradient of Error(w)

The gradient is a generalization of the derivative

$$\nabla Err(\mathrm{w}) = \left(\frac{\partial Err(w)}{\partial w_0}, \frac{\partial Err(w)}{\partial w_1}, ..., \frac{\partial Err(w)}{\partial w_n} \right)$$

The gradient is a vector of partial derivatives.

It Indicates the *direction of steepest increase* in Err(w), for each one of w's coordinates

Gradient Descent Updates

- **Compute** the gradient of the training error at each iteration
  - Batch mode: compute the gradient over all training examples

$$\nabla Err(\mathrm{w}) = \left(\frac{\partial Err(w)}{\partial w_0}, \frac{\partial Err(w)}{\partial w_1}, \ldots, \frac{\partial Err(w)}{\partial w_n}\right)$$

- **Update w:**

Learning rate (>0)

$$\mathrm{w}^{i+1} = \mathrm{w}^i - \alpha \nabla Err(\mathrm{w}^i)$$

Computing $\nabla \mathrm{Err}(w^i)$ for Squared Loss

$$Err(w) = \frac{1}{2}\sum_{d \in D}(y_d - f(x_d))^2$$

$$\frac{\partial \mathrm{Err}(\mathbf{w})}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d \in D}(y_d - f(\mathbf{x}_d))^2$$

$$= \frac{1}{2}\frac{\partial}{\partial w_i}\sum_{d \in D}(y_d - f(\mathbf{x}_d))^2$$

$$= \frac{1}{2}\sum_{d \in D} 2(y_d - f(\mathbf{x}_d))\frac{\partial}{\partial w_i}(y_d - \mathbf{w} \cdot \mathbf{x}_d)$$

$$= -\sum_{d \in D}(y_d - f(\mathbf{x}_d))x_{di}$$

Slide adapted from J Hockenmaier slides

Batch Update Rule for Each w_i

- **Implementing gradient descent***:*
  As you go through the training data, accumulate the change in each w_i of W

$$\Delta w_i = \alpha \sum_{d=1}^{D} (y_d - \mathbf{w}^i \cdot \mathbf{x}_d) x_{di}$$

Gradient Descent for Squared Loss

Initialize $\mathbf{w}^{0}$ randomly
for i = 0...T:
$\quad \Delta\mathbf{w} = (0, ..., 0)$
$\quad$ for every training item d = 1...D:
$\quad\quad f(\mathbf{x_d}) = \mathbf{w^i} \cdot \mathbf{x_d}$
$\quad\quad$ for every component of $\mathbf{w}$ $\quad$ j = 0...N:
$\quad\quad\quad \Delta w_j \mathrel{+}= \alpha(y_d - f(\mathbf{x_d})) \cdot x_{dj}$
$\quad \mathbf{w}^{i+1} = \mathbf{w}^{i} + \Delta\mathbf{w}$
$\quad$ return $\mathbf{w}^{i+1}$ when it has converged

Slide adapted from J Hockenmaier slides

Batch vs. Online Learning

- The Gradient Descent algorithm makes updates after going over the entire data set
    - Data set can be huge
    - Streaming mode (we cannot assume we saw all the data)
    - Online learning allows "adapting" to changes in the target function
- Stochastic Gradient Descent
    - Similar to GD, updates after each example
    - Can we make the same convergence assumptions as in GD?
- Variations: update after a subset of examples (mini-batch)

Stochastic Gradient Descent

Initialize $\mathbf{w^o}$ randomly

for m = 0...M:

$$f(\mathbf{x_m}) = \mathbf{w^i} \cdot \mathbf{x_m}$$

$$\Delta w_j = \alpha(y_d - f(\mathbf{x_m})) \cdot x_{mj}$$

$$\mathbf{w^{i+1}} = \mathbf{w^i} + \Delta\mathbf{w}$$

return $\mathbf{w^{i+1}}$ when it has converged

Slide adapted from J Hockenmaier slides