

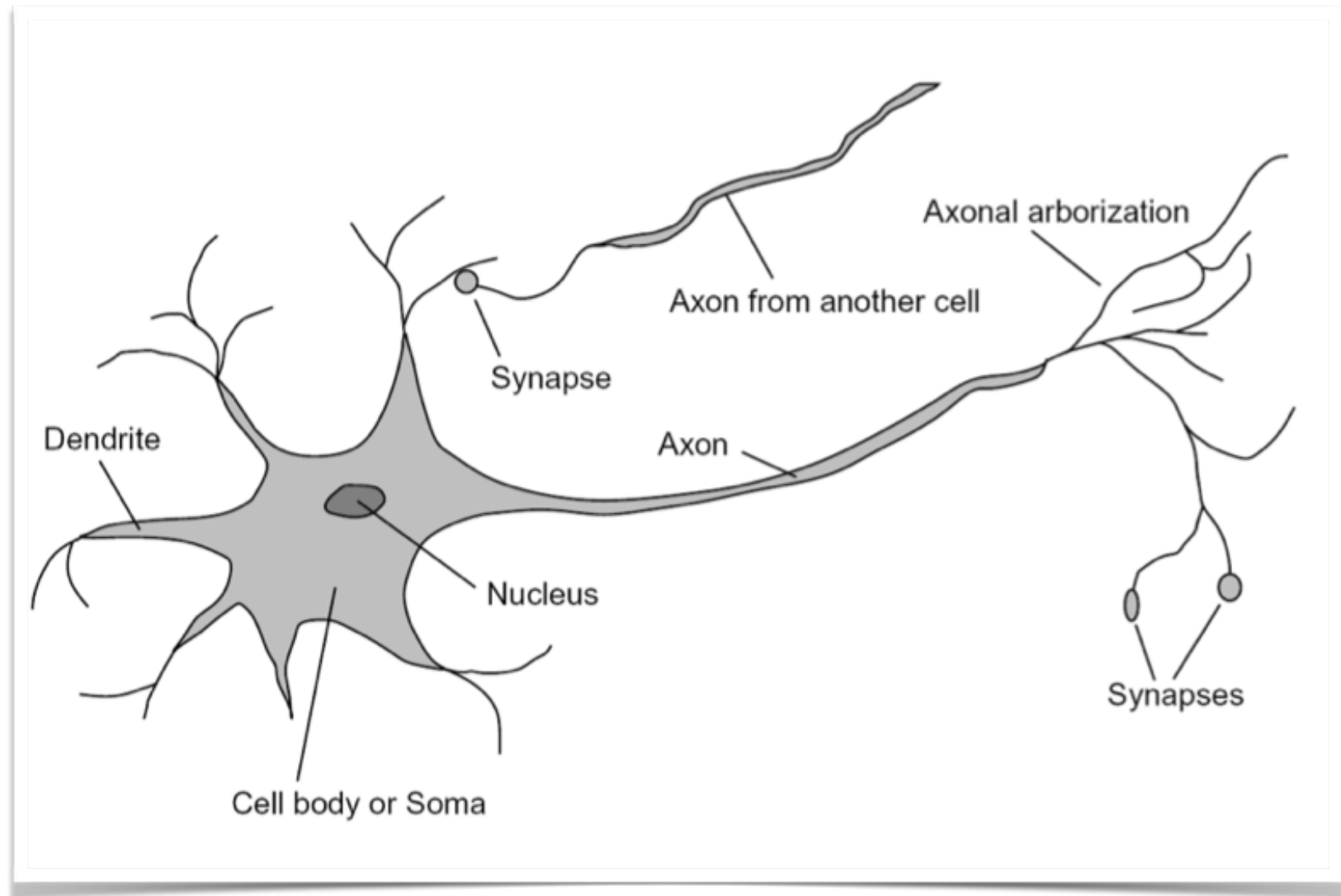
# Linear models

Subhransu Maji

CMPSCI 670: Computer Vision

November 3, 2016

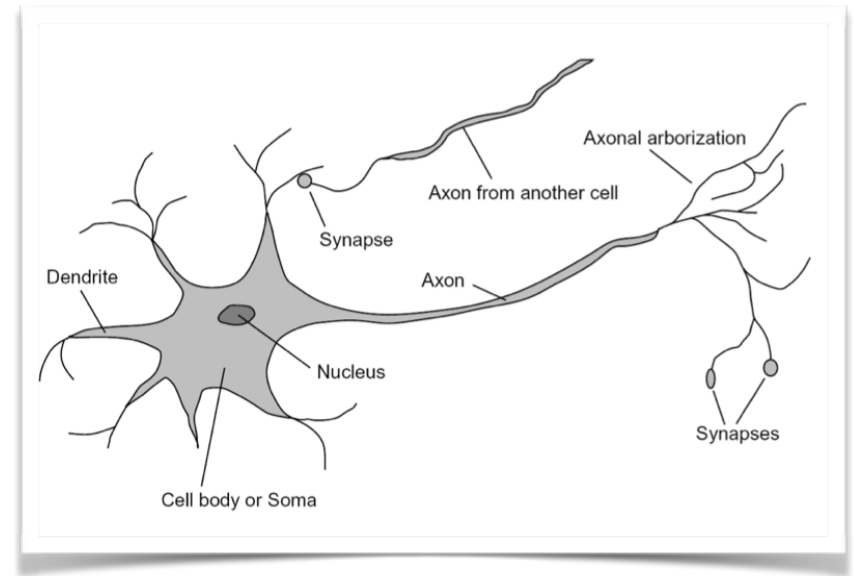
# A neuron (or how our brains work)



Neuroscience 101

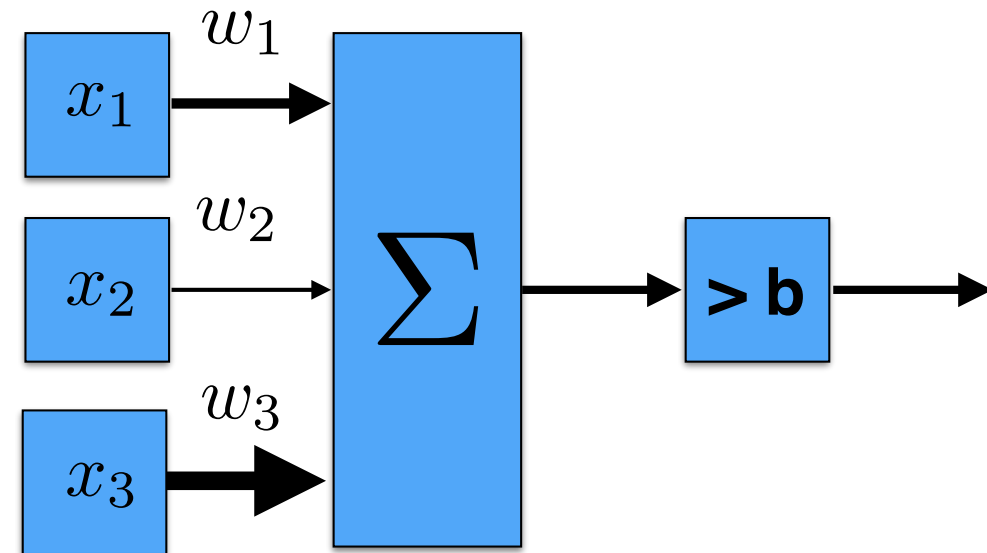
# Perceptron

- ◆ Input are **feature values**
- ◆ Each feature has a **weight**
- ◆ Sum in the **activation**



$$\text{activation}(\mathbf{w}, \mathbf{x}) = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

- ◆ If the activation is:
  - $> b$ , output *class 1*
  - otherwise, output *class 2*



$$\mathbf{x} \rightarrow (\mathbf{x}, 1)$$

$$\mathbf{w}^T \mathbf{x} + b \rightarrow (\mathbf{w}, b)^T (\mathbf{x}, 1)$$

# Example: Spam

- ◆ Imagine 3 features (spam is “positive” class):
  - ▶ free (number of occurrences of “free”)
  - ▶ money (number of occurrences of “money”)
  - ▶ BIAS (intercept, always has value 1)

email	$\mathbf{x}$	$\mathbf{w}$	$\mathbf{w}^T \mathbf{x}$
“free money”	BIAS : 1	BIAS : -3	(1)(-3) +
	free : 1	free : 4	(1)(4) +
	money : 1	money : 2	(1)(2) +
	...	...	...
			= 3

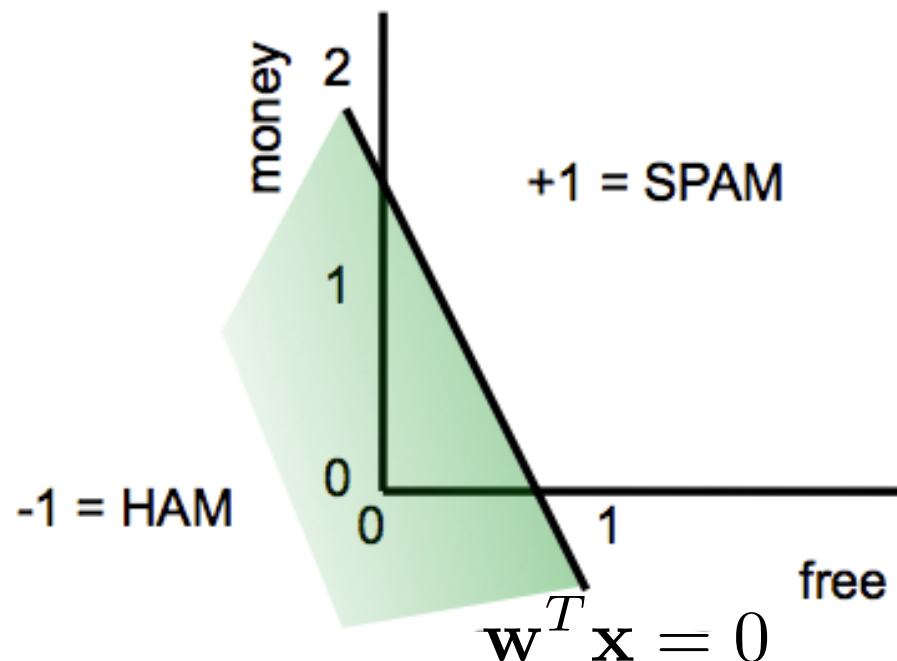
$$\mathbf{w}^T \mathbf{x} > 0 \rightarrow \text{SPAM!!}$$

# Geometry of the perceptron

- ◆ In the space of feature vectors
  - ▶ examples are points (in  $D$  dimensions)
  - ▶ an weight vector is a hyperplane (a  $D-1$  dimensional object)
  - ▶ One side corresponds to  $y=+1$
  - ▶ Other side corresponds to  $y=-1$
- ◆ Perceptrons are also called as linear classifiers

**w**

BIAS	:	-3
free	:	4
money	:	2
...		

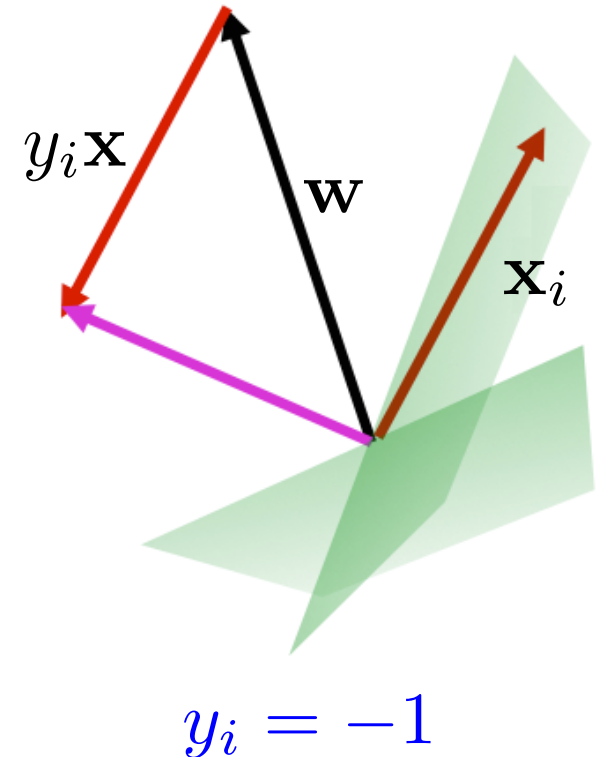


# Learning a perceptron

Input: training data  $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)$

Perceptron training algorithm [Rosenblatt 57]

- ◆ Initialize  $\mathbf{w} \leftarrow [0, \dots, 0]$
- ◆ for iter = 1, ..., T
  - ▶ for i = 1, ..., n
    - predict according to the current model
$$\hat{y}_i = \begin{cases} +1 & \text{if } \mathbf{w}^T \mathbf{x}_i > 0 \\ -1 & \text{if } \mathbf{w}^T \mathbf{x}_i \leq 0 \end{cases}$$
      - if  $y_i = \hat{y}_i$ , no change
      - else,  $\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$



error driven, online, activations increase for +, randomize

# Properties of perceptrons

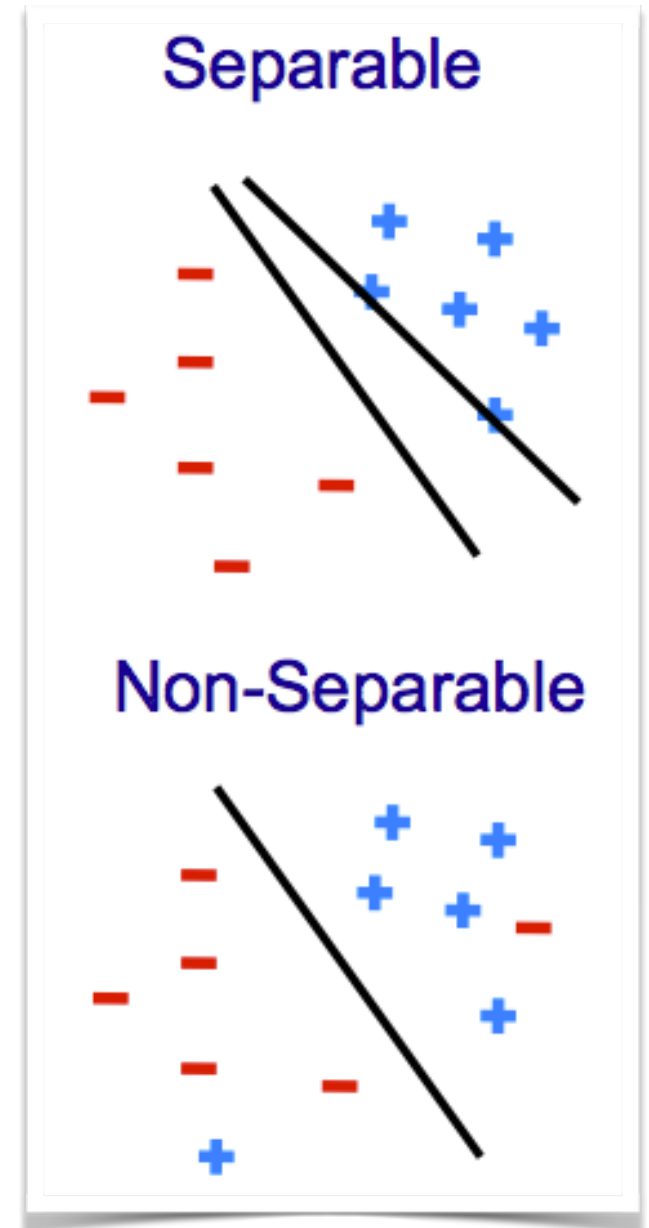
- ◆ **Separability:** some parameters will classify the training data perfectly
- ◆ **Convergence:** if the training data is separable then the perceptron training will eventually converge [Block 62, Novikoff 62]
- ◆ **Mistake bound:** the maximum number of mistakes is related to the margin

assuming,  $\|\mathbf{x}_i\| \leq 1$

$$\# \text{mistakes} < \frac{1}{\delta^2}$$

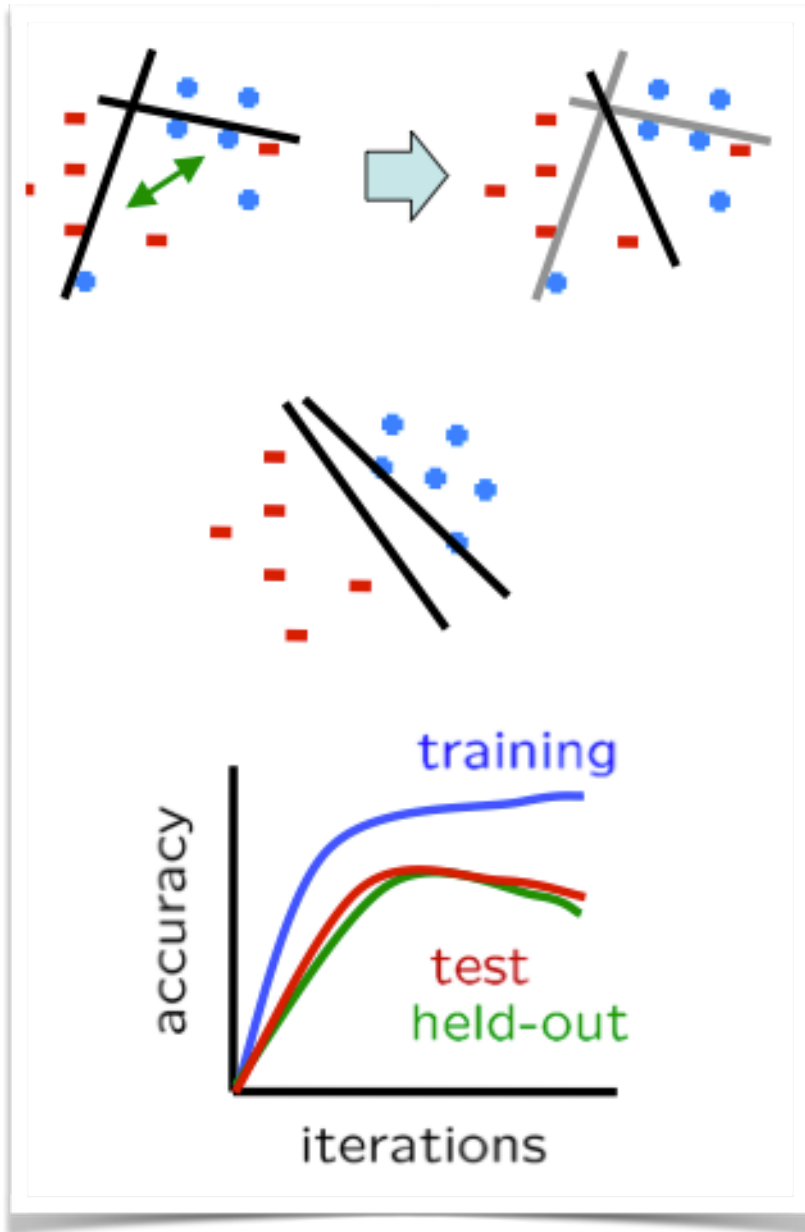
$$\delta = \max_{\mathbf{w}} \min_{(\mathbf{x}_i, y_i)} [y_i \mathbf{w}^T \mathbf{x}_i]$$

such that,  $\|\mathbf{w}\| = 1$



# Limitations of perceptrons

- ◆ **Convergence:** if the data isn't separable, the training algorithm may not terminate
  - noise can cause this
  - some simple functions are not separable (xor)
- ◆ **Mediocre generation:** the algorithm finds a solution that “barely” separates the data
- ◆ **Overtraining:** test/validation accuracy rises and then falls
  - Overtraining is a kind of overfitting





# Overview

- ◆ Linear models
  - ▶ Perceptron: model and learning algorithm combined as one
  - ▶ Is there a better way to learn linear models?
- ◆ We will separate models and learning algorithms
  - ▶ Learning as optimization
  - ▶ Surrogate loss function
  - ▶ Regularization

} model design

  - ▶ Gradient descent
  - ▶ Batch and online gradients
  - ▶ Subgradient descent
  - ▶ Support vector machines

} optimization

# Learning as optimization

$$\min_{\mathbf{w}} \sum_n \mathbf{1}[y_n \mathbf{w}^T \mathbf{x}_n < 0]$$

↑  
fewest mistakes

- ◆ The **perceptron algorithm** will find an optimal  $\mathbf{w}$  if the data is **separable**
  - efficiency depends on the **margin** and **norm** of the data
- ◆ However, if the data is not separable, optimizing this is **NP-hard**
  - i.e., there is no efficient way to minimize this unless **P=NP**

# Learning as optimization

$$\min_{\mathbf{w}} \sum_n \mathbf{1}[y_n \mathbf{w}^T \mathbf{x}_n < 0] + \lambda R(\mathbf{w})$$

hyperparameter

fewest mistakes

simpler model

- ◆ In addition to minimizing **training error**, we want a **simpler model**
  - Remember our goal is to minimize **generalization error**
  - Recall the **bias** and **variance** tradeoff for learners
- ◆ We can add a **regularization** term  $R(\mathbf{w})$  that prefers simpler models
  - For example we may prefer decision trees of shallow depth
- ◆ Here  $\lambda$  is a **hyperparameter** of optimization problem

# Learning as optimization

$$\min_{\mathbf{w}} \sum_n \mathbf{1}[y_n \mathbf{w}^T \mathbf{x}_n < 0] + \lambda R(\mathbf{w})$$

hyperparameter

fewest mistakes

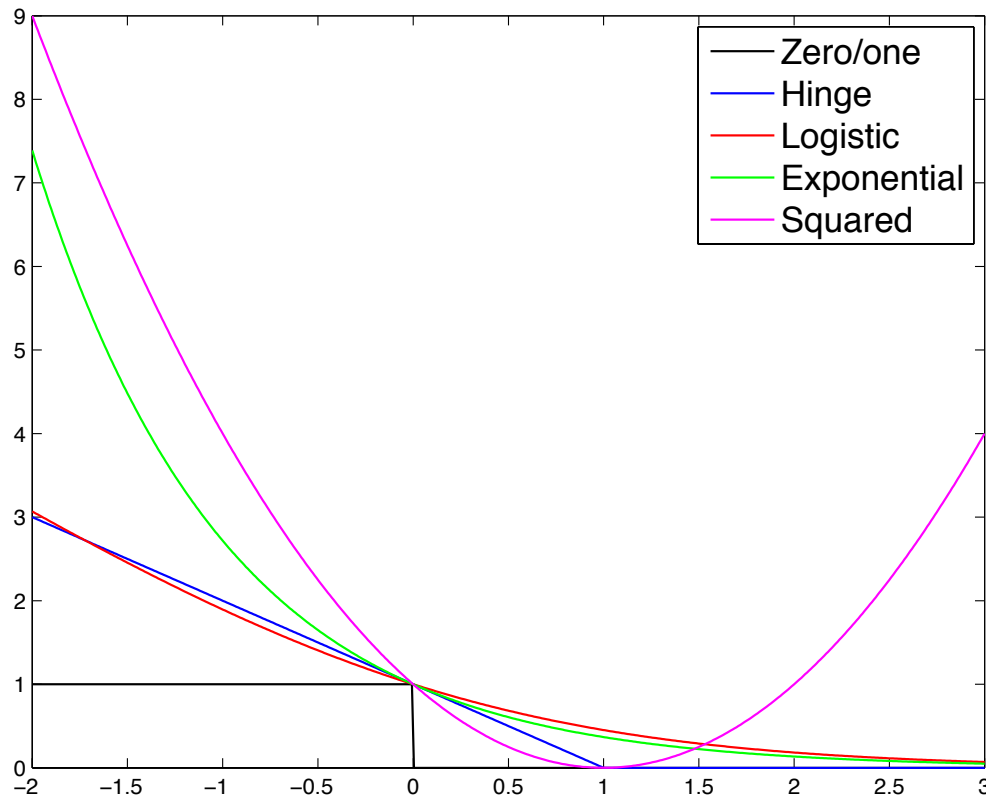
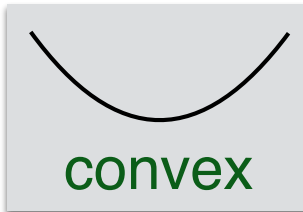
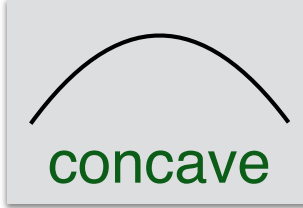
simpler model

◆ The questions that remain are:

- ▶ What are good ways to **adjust the optimization problem** so that there are efficient algorithms for solving it?
- ▶ What are good **regularizations**  $R(\mathbf{w})$  for hyperplanes?
- ▶ Assuming that the optimization problem can be adjusted appropriately, what **algorithms** exist for solving the regularized optimization problem?

# Convex surrogate loss functions

- ◆ **Zero/one loss** is hard to optimize
  - Small changes in  $\mathbf{w}$  can cause large changes in the **loss**
- ◆ **Surrogate loss**: replace **Zero/one loss** by a smooth function
  - Easier to optimize if the **surrogate loss** is **convex**
- ◆ **Examples:**



$$y = +1 \quad \hat{y} \leftarrow \mathbf{w}^T \mathbf{x}$$

Zero/one:  $\ell^{(0/1)}(y, \hat{y}) = \mathbf{1}[y\hat{y} \leq 0]$

Hinge:  $\ell^{(\text{hin})}(y, \hat{y}) = \max\{0, 1 - y\hat{y}\}$

Logistic:  $\ell^{(\text{log})}(y, \hat{y}) = \frac{1}{\log 2} \log(1 + \exp[-y\hat{y}])$

Exponential:  $\ell^{(\text{exp})}(y, \hat{y}) = \exp[-y\hat{y}]$

Squared:  $\ell^{(\text{sqr})}(y, \hat{y}) = (y - \hat{y})^2$

# Weight regularization

- ◆ What are good **regularization** functions  $R(\mathbf{w})$  for hyperplanes?
- ◆ We would like the weights —
  - ▶ To be **small** —
    - ➔ Change in the features cause small change to the score
    - ➔ Robustness to noise
  - ▶ To be **sparse** —
    - ➔ Use as few features as possible
    - ➔ Similar to controlling the depth of a decision tree
- ◆ This is a form of **inductive bias**

# Weight regularization

- ◆ Just like the **surrogate loss function**, we would like  $R(\mathbf{w})$  to be **convex**
- ◆ **Small weights** regularization

$$R^{(\text{norm})}(\mathbf{w}) = \sqrt{\sum_d w_d^2}$$

$$R^{(\text{sqr})}(\mathbf{w}) = \sum_d w_d^2$$

- ◆ **Sparsity** regularization

$$R^{(\text{count})}(\mathbf{w}) = \sum_d \mathbf{1}[|w_d| > 0]$$

**not convex**

- ◆ Family of **“p-norm”** regularization

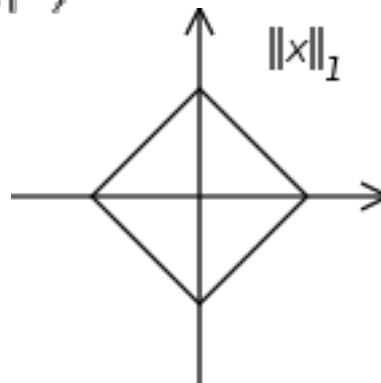
$$R^{(\text{p-norm})}(\mathbf{w}) = \left( \sum_d |w_d|^p \right)^{1/p}$$

# Contours of p-norms

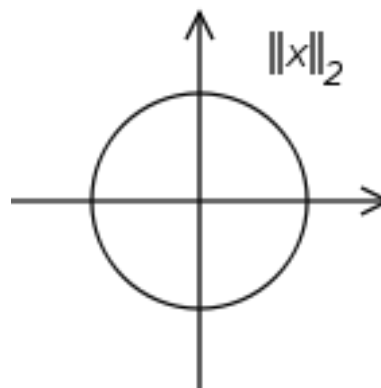
$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$$

convex for  $p \geq 1$

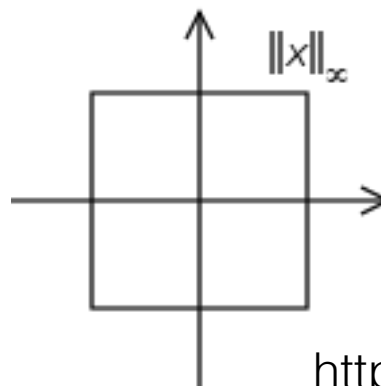
$$\|x\|_1 = \sum_{i=1}^n |x_i|$$



$$\|x\|_2 = \sqrt{\sum_{i=1}^n |x_i|^2}$$



$$\|x\|_\infty = \max_{i=1,\dots,n} |x_i|$$



[http://en.wikipedia.org/wiki/Lp\\_space](http://en.wikipedia.org/wiki/Lp_space)

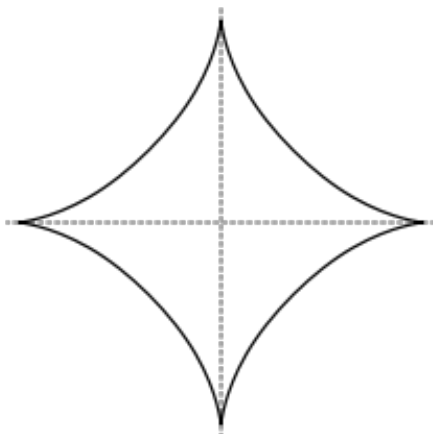


# Contours of p-norms

$$\|x\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}}$$

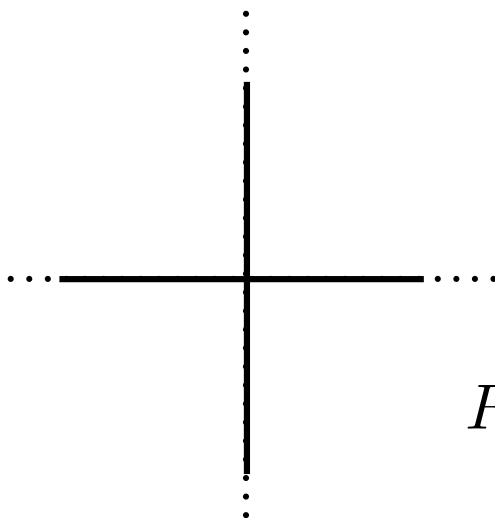
not convex for  $0 \leq p < 1$

$$p = \frac{2}{3}$$



Counting non-zeros:

$$p = 0$$



$$R^{(\text{count})}(\mathbf{w}) = \sum_d \mathbf{1}[|w_d| > 0]$$

[http://en.wikipedia.org/wiki/Lp\\_space](http://en.wikipedia.org/wiki/Lp_space)

# General optimization framework

$$\min_{\mathbf{w}} \sum_n \ell(y_n, \mathbf{w}^T \mathbf{x}_n) + \lambda R(\mathbf{w})$$

Diagram illustrating the general optimization framework:

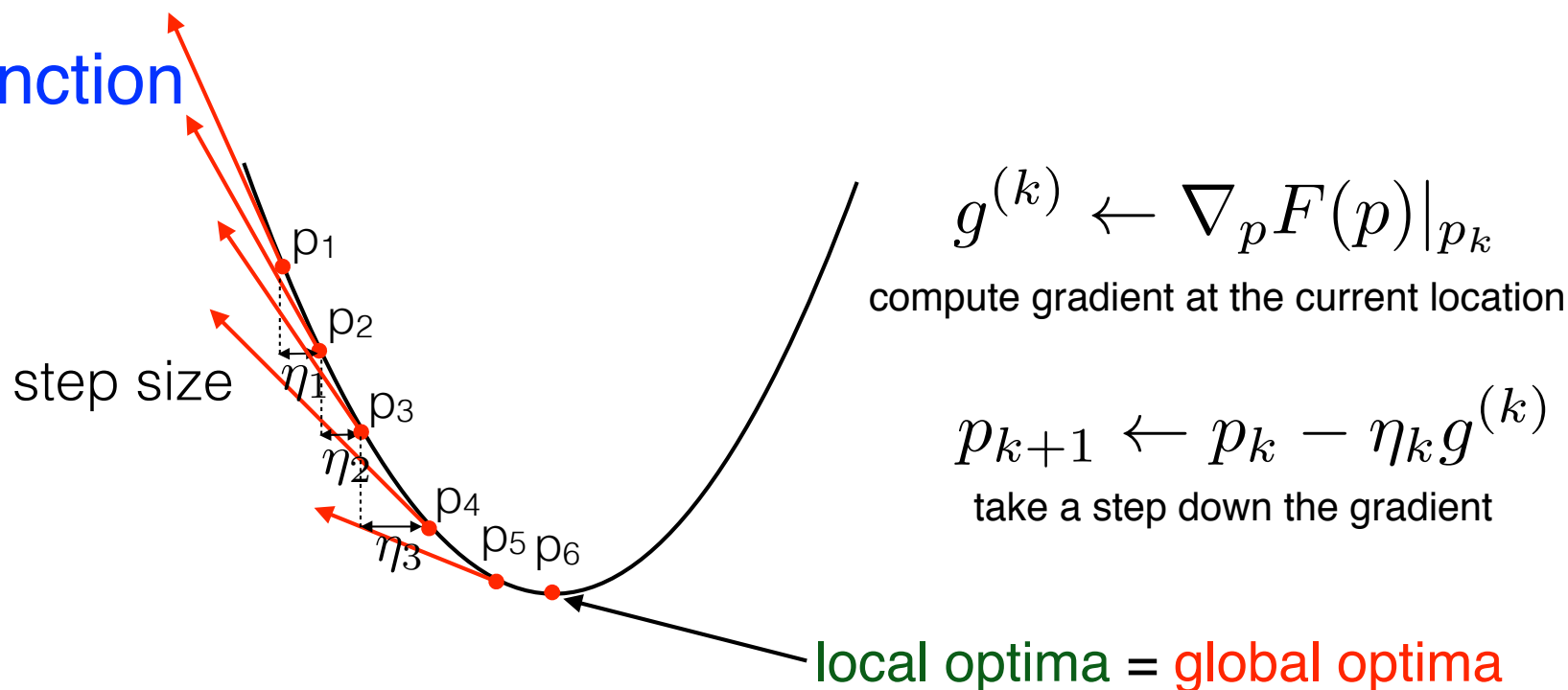
- hyperparameter** (blue text) points to  $\lambda$ .
- surrogate loss** (blue text) points to  $\ell(y_n, \mathbf{w}^T \mathbf{x}_n)$ .
- regularization** (blue text) points to  $R(\mathbf{w})$ .

- ◆ Select a suitable:
  - **convex surrogate loss**
  - **convex regularization**
- ◆ Select the **hyperparameter**  $\lambda$
- ◆ Minimize the regularized objective with respect to  $\mathbf{w}$
- ◆ This framework for optimization is called **Tikhonov regularization** or generally **Structural Risk Minimization (SRM)**

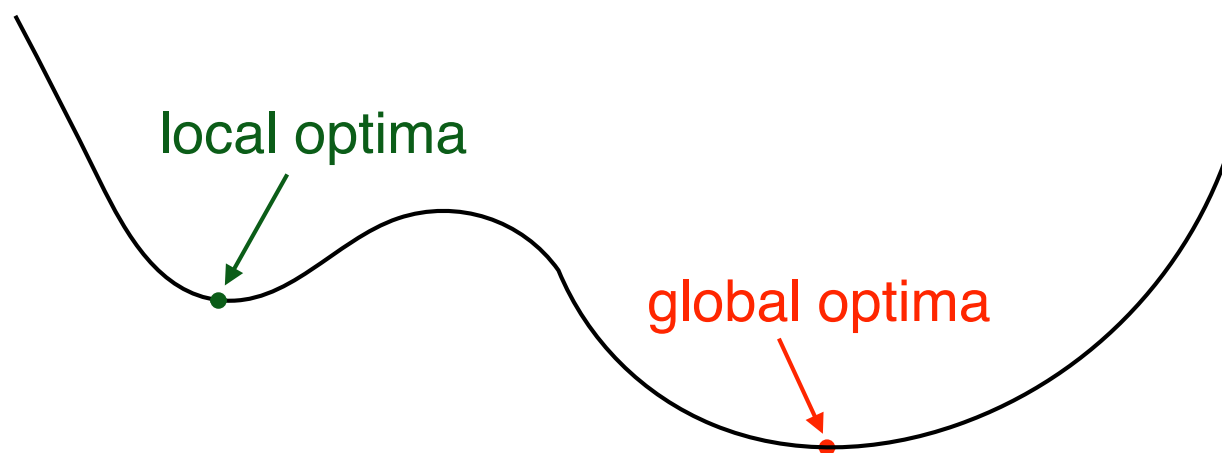
[http://en.wikipedia.org/wiki/Tikhonov\\_regularization](http://en.wikipedia.org/wiki/Tikhonov_regularization)

# Optimization by gradient descent

## Convex function



## Non-convex function

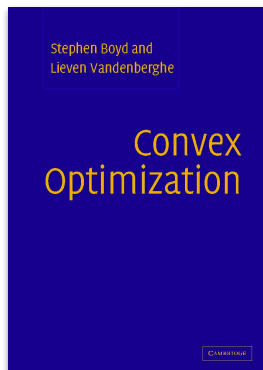


# Choice of step size

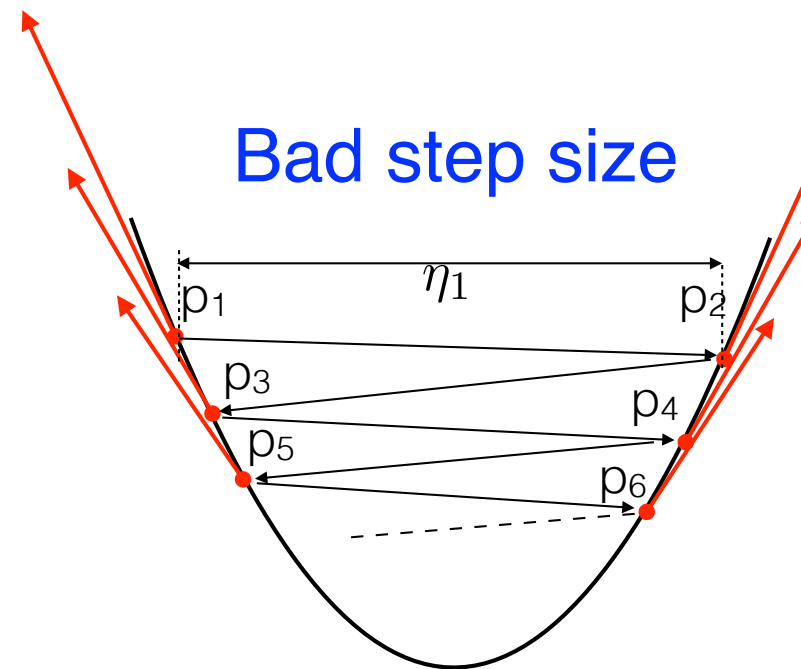
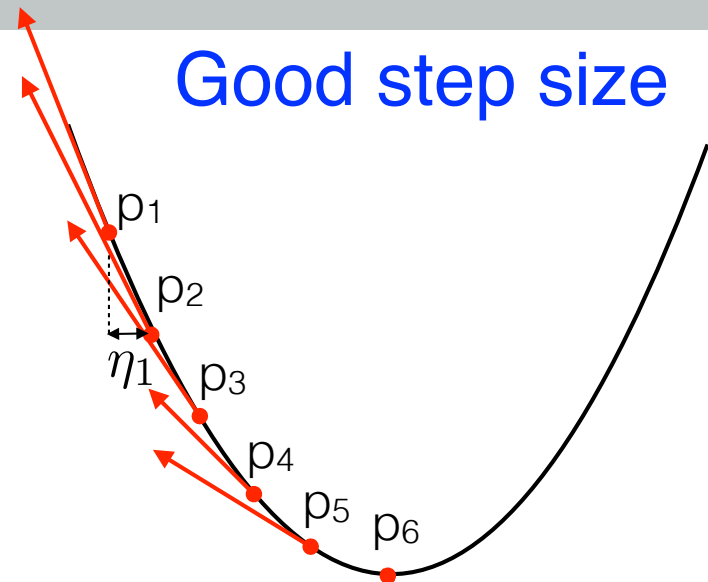
- ◆ The step size is important —
  - ▶ **too small**: slow convergence
  - ▶ **too large**: no convergence
- ◆ A strategy is to use large step sizes initially and small step sizes later:

$$\eta_t \leftarrow \eta_0 / (t_0 + t)$$

- ◆ There are methods that converge faster by adapting step size to the **curvature** of the function
  - ▶ Field of **convex optimization**



<http://stanford.edu/~boyd/cvxbook/>



# Example: Exponential loss

$$\mathcal{L}(\mathbf{w}) = \sum_n \exp(-y_n \mathbf{w}^T \mathbf{x}_n) + \frac{\lambda}{2} \|\mathbf{w}\|^2 \quad \text{objective}$$

$$\frac{d\mathcal{L}}{d\mathbf{w}} = \sum_n -y_n \mathbf{x}_n \exp(-y_n \mathbf{w}^T \mathbf{x}_n) + \lambda \mathbf{w} \quad \text{gradient}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left( \sum_n -y_n \mathbf{x}_n \exp(-y_n \mathbf{w}^T \mathbf{x}_n) + \lambda \mathbf{w} \right) \quad \text{update}$$

loss term

$$\mathbf{w} \leftarrow \mathbf{w} + c y_n \mathbf{x}_n$$

↑

high for misclassified points

similar to the [perceptron update rule](#)!

regularization term

$$\mathbf{w} \leftarrow (1 - \eta \lambda) \mathbf{w}$$

shrinks weights towards zero

# Batch and online gradients

$$\mathcal{L}(\mathbf{w}) = \sum_n \mathcal{L}_n(\mathbf{w}) \quad \text{objective}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{d\mathcal{L}}{d\mathbf{w}} \quad \text{gradient descent}$$

## batch gradient

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left( \sum_n \frac{d\mathcal{L}_n}{d\mathbf{w}} \right)$$

sum of n gradients

update weight after you see all points

## online gradient

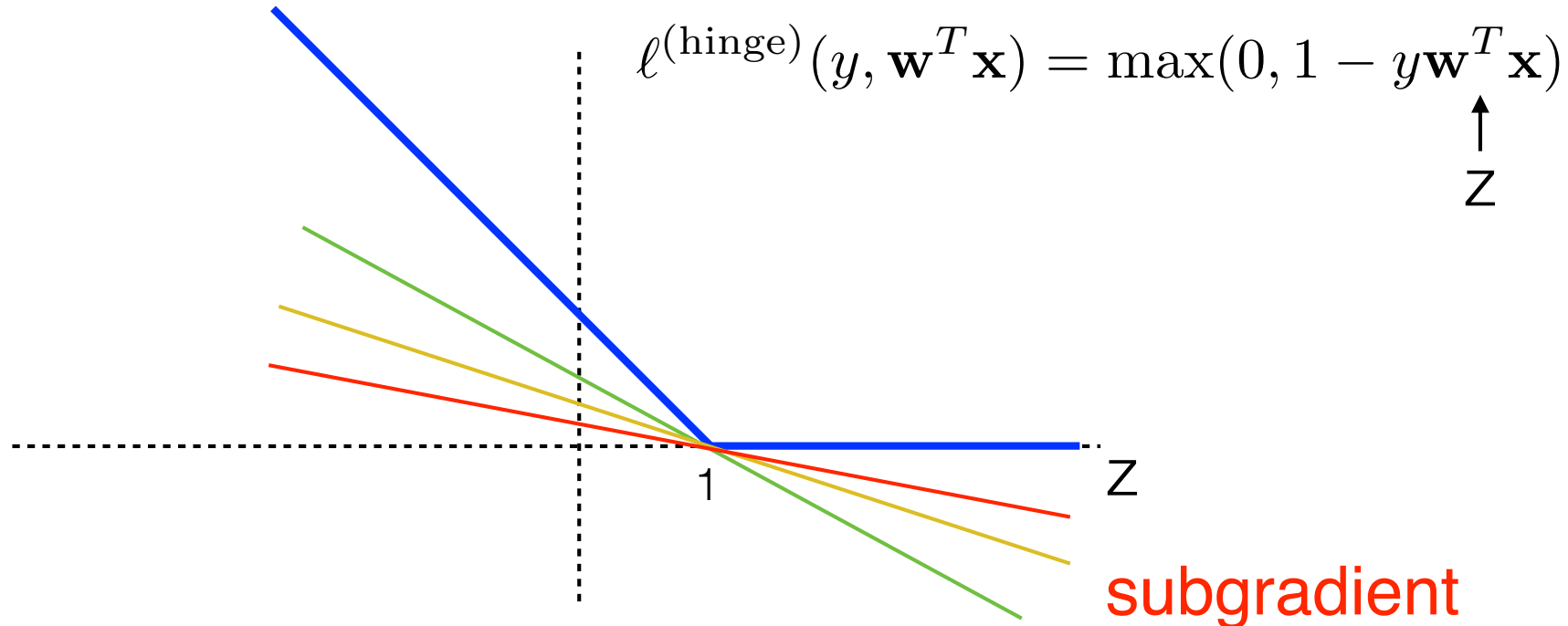
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left( \frac{d\mathcal{L}_n}{d\mathbf{w}} \right)$$

gradient at n<sup>th</sup> point

update weights after you see each point

Online gradients are the default method for multi-layer perceptrons

# Subgradient



- ◆ The **hinge loss** is not differentiable at  $z=1$
- ◆ **Subgradient** is any direction that is **below** the function
- ◆ For the **hinge loss** a possible **subgradient** is:

$$\frac{d\ell^{\text{hinge}}}{d\mathbf{w}} = \begin{cases} 0 & \text{if } y\mathbf{w}^T \mathbf{x} > 1 \\ -y\mathbf{x} & \text{otherwise} \end{cases}$$

# Example: Hinge loss

$$\mathcal{L}(\mathbf{w}) = \sum_n \max(0, 1 - y_n \mathbf{w}^T \mathbf{x}_n) + \frac{\lambda}{2} \|\mathbf{w}\|^2 \quad \text{objective}$$

$$\frac{d\mathcal{L}}{d\mathbf{w}} = \sum_n -\mathbf{1}[y_n \mathbf{w}^T \mathbf{x}_n \leq 1] y_n \mathbf{x}_n + \lambda \mathbf{w} \quad \text{subgradient}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left( \sum_n -\mathbf{1}[y_n \mathbf{w}^T \mathbf{x}_n \leq 1] y_n \mathbf{x}_n + \lambda \mathbf{w} \right) \quad \text{update}$$

loss term

$$\mathbf{w} \leftarrow \mathbf{w} + \eta y_n \mathbf{x}_n$$



only for points  $y_n \mathbf{w}^T \mathbf{x}_n \leq 1$

perceptron update  $y_n \mathbf{w}^T \mathbf{x}_n \leq 0$

regularization term

$$\mathbf{w} \leftarrow (1 - \eta\lambda) \mathbf{w}$$

shrinks weights towards zero



# Example: Squared loss

$$\mathcal{L}(\mathbf{w}) = \sum_n (y_n - \mathbf{w}^T \mathbf{x}_n)^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2 \quad \text{objective}$$

matrix notation

$$\underbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,D} \\ x_{2,1} & x_{2,2} & \dots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,D} \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{bmatrix}}_{\mathbf{w}} = \underbrace{\begin{bmatrix} \sum_d x_{1,d} w_d \\ \sum_d x_{2,d} w_d \\ \vdots \\ \sum_d x_{N,d} w_d \end{bmatrix}}_{\hat{\mathbf{Y}}} \approx \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\mathbf{Y}}$$

equivalent loss

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{2} \|\mathbf{X}\mathbf{w} - \mathbf{Y}\|^2 + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

# Example: Squared loss

$$\min_w \mathcal{L}(w) = \frac{1}{2} ||\mathbf{X}w - \mathbf{Y}||^2 + \frac{\lambda}{2} ||w||^2$$

objective

$$\begin{aligned}\nabla_w \mathcal{L}(w) &= \mathbf{X}^\top (\mathbf{X}w - \mathbf{Y}) + \lambda w \\ &= \mathbf{X}^\top \mathbf{X}w - \mathbf{X}^\top \mathbf{Y} + \lambda w \\ &= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) w - \mathbf{X}^\top \mathbf{Y}\end{aligned}$$

gradient

At optima the gradient=0

$$\begin{aligned}& (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) w - \mathbf{X}^\top \mathbf{Y} = 0 \\ \iff & (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D) w = \mathbf{X}^\top \mathbf{Y} \\ \iff & w = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}_D)^{-1} \mathbf{X}^\top \mathbf{Y}\end{aligned}$$

exact  
closed-form  
solution

# Matrix inversion vs. gradient descent

- ◆ Assume, we have D **features** and N **points**

- ◆ Overall time via **matrix inversion**

- ▶ The closed form solution involves computing:

$$\mathbf{w} = \left( \mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}_D \right)^{-1} \mathbf{X}^T \mathbf{Y}$$

- ▶ Total time is  $O(D^2N + D^3 + DN)$ , assuming  $O(D^3)$  matrix inversion
- ▶ If  $N > D$ , then total time is  $O(D^2N)$

- ◆ Overall time via **gradient descent**

- ▶ Gradient: 
$$\frac{d\mathcal{L}}{d\mathbf{w}} = \sum_n -2(y_n - \mathbf{w}^T \mathbf{x}_n) \mathbf{x}_n + \lambda \mathbf{w}$$

- ▶ Each iteration:  $O(ND)$ ; T iterations:  $O(TND)$

- ◆ Which one is faster?

- ▶ **Small problems**  $D < 100$ : probably faster to run **matrix inversion**
- ▶ **Large problems**  $D > 10,000$ : probably faster to run **gradient descent**

# Optimization for linear models

- ◆ Under suitable conditions\*, provided you pick the step sizes appropriately, the convergence rate of gradient descent is  $O(1/N)$ 
  - ▶ i.e., if you want a solution within 0.0001 of the optimal you have to run the gradient descent for  $N=1000$  iterations.
- ◆ For linear models ([hinge/logistic/exponential loss](#)) and [squared-norm regularization](#) there are off-the-shelf solvers that are fast in practice: [SVM<sup>perf</sup>](#), [LIBLINEAR](#), [PEGASOS](#)
  - ▶ [SVM<sup>perf</sup>](#), [LIBLINEAR](#) use a different optimization method

\* the function is strongly convex:  $f(y) \geq f(x) + \nabla f(x)^T(y - x) + \frac{m}{2}\|y - x\|_2^2$

# Feature normalization

- ◆ Even if a feature is useful some normalization may be good
- ◆ Per-feature normalization

- ▶ Centering  $x_{n,d} \leftarrow x_{n,d} - \mu_d$

- ▶ Variance scaling  $x_{n,d} \leftarrow x_{n,d} / \sigma_d$

- ▶ Absolute scaling  $x_{n,d} \leftarrow x_{n,d} / r_d$

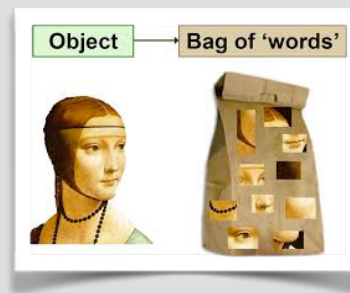
- ▶ Non-linear transformation
  - ➔ square-root

$$x_{n,d} \leftarrow \sqrt{x_{n,d}}$$

(corrects for burstiness)

$$\mu_d = \frac{1}{N} \sum_n x_{n,d}$$
$$\sigma_d = \sqrt{\frac{1}{N} \sum_n (x_{n,d} - \mu_d)^2}$$
$$r_d = \max_n |x_{n,d}|$$

## Caltech-101 image classification



41.6% linear  
63.8% square-root

- ◆ Per-example normalization

- ▶ fixed norm for each example  $\|\mathbf{x}\| = 1$

# Slides credit

- ◆ Figures of various “p-norms” are from Wikipedia
  - ▶ [http://en.wikipedia.org/wiki/Lp\\_space](http://en.wikipedia.org/wiki/Lp_space)
- ◆ Some of the slides are based on CIML book by Hal Daume III

# Appendix: code for surrogateLoss

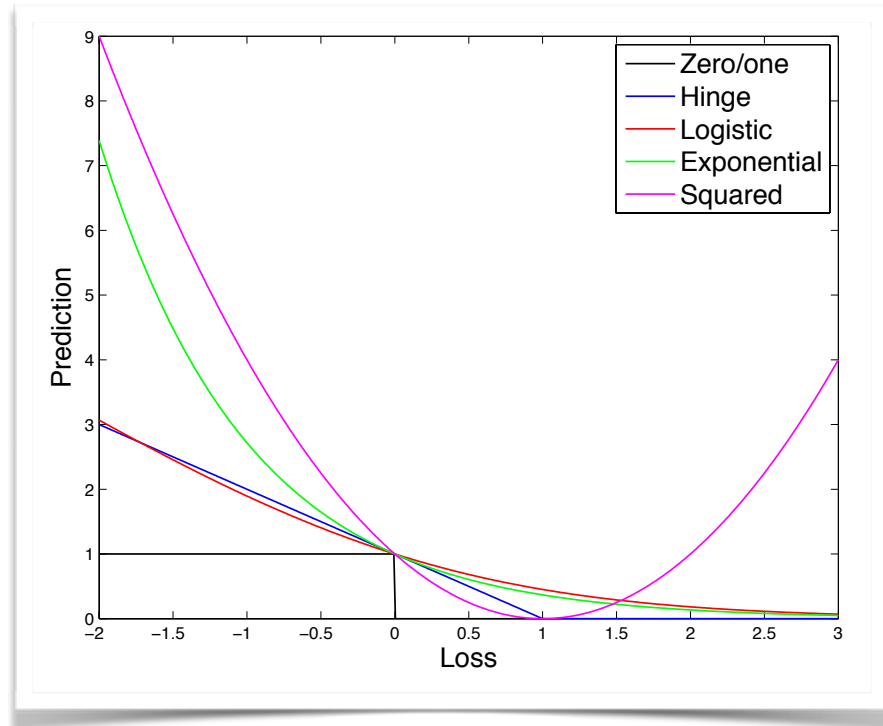
Output  $\longrightarrow$

```
% Code to plot various loss functions
```

```
y1=1;  
y2=linSPACE(-2,3,500);  
zeroOneLoss = y1*y2 <=0;  
hingeLoss = max(0, 1-y1*y2);  
logisticLoss = log(1+exp(-y1*y2))/log(2);  
expLoss = exp(-y1*y2);  
squaredLoss = (y1-y2).^2;
```

```
% Plot them
```

```
figure(1); clf; hold on;  
plot(y2, zeroOneLoss, 'k-', 'LineWidth', 1);  
plot(y2, hingeLoss, 'b-', 'LineWidth', 1);  
plot(y2, logisticLoss, 'r-', 'LineWidth', 1);  
plot(y2, expLoss, 'g-', 'LineWidth', 1);  
plot(y2, squaredLoss, 'm-', 'LineWidth', 1);  
ylabel('Prediction', 'FontSize', 16);  
xlabel('Loss', 'FontSize', 16);  
legend({'Zero/one', 'Hinge', 'Logistic', 'Exponential', 'Squared'}, 'Location', 'NorthEast', 'FontSize', 16);  
box on;
```



$\longleftarrow$  Matlab code