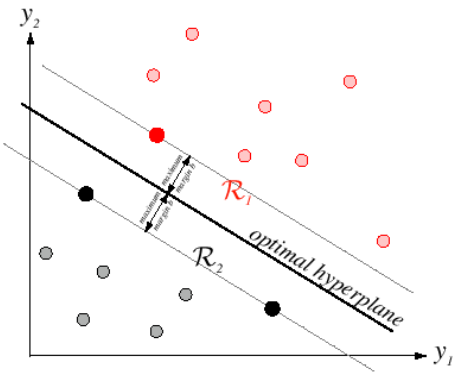
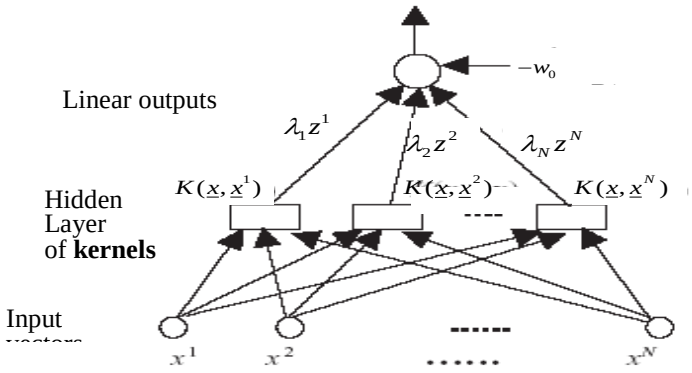
Recall that SVM nonlinearly transforms data into a higher dimensional feature space using Kernels (which require only inner products for their computation) and solve a quadratic programming problem in the dual space to find an optimal hyperplane separating each pair of classes in the new space. They are also called Large margin classifiers because they find a hyperplane with the largest separation (margin) between two classes. Can transform $\underline{x}$ into K($\underline{x}$): Gaussian RBF, Polynomial, MLP, etc. are used as Kernels. Kernels exploit inner products between data points. Pegasos algorithm using sub-gradient method is one of the better known algorithms.





Kernels allow you to transform data for linear separability. Kernels exploit inner product between data points.

$K = [K(\underline{x}^i, \underline{x}^j)]$ *is a Kernel if* $K \geq 0$

Mercer's theorem

Lagrange multipliers with positive values are called the support vectors. They correspond to samples that satisfy the primal inequality constraints as equalities…. Such constraints are called 'active' in nonlinear programming. **Support vectors are critical elements of the training set**. **They lie closest to the decision boundary**!! Typically 30% of data are support vectors, which means you need to store 30% of data….. now you can see why this method may not scale to big data.

The ν-SVM seek to minimize the number of support vectors.

ν=*lower bound on the number of* sup *port vectors*

- ν-SVM: $\min_{\underline{w},w_0,\underline{\alpha}\geq\underline{0},\rho\geq 0} \frac{1}{2}\|\underline{w}\|_2^2 - \nu\rho + \frac{1}{N}\sum_{i=1}^{N}\alpha_i$

  $s.t.\ \ z^i(\underline{w}^T\Phi(\underline{x}^i) - w_0) \geq \rho - \alpha_i$

$Dual: q(\underline{\lambda}) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda_i\lambda_j z^i z^j K(\underline{x}^i,\underline{x}^j)$

$subject\ to: \sum_{i=1}^{N}\lambda_i z^i = 0$ **and** $0 \leq \lambda_i \leq \frac{1}{N}; \sum_{i=1}^{N}\lambda_i \geq \nu$

$\nu \in [0,1] \Rightarrow easy\ to \exp eriment$

$C = \frac{1}{N\rho}$

$K(\underline{x}^i,\underline{x}^j) = \Phi(\underline{x}^i)^T\Phi(\underline{x}^j)$....*inner product*

- Elastic net regression problem can be solved using an SVM algorithm or vice versa.
- Kernel Regression

Recall Ridge-regression

$D = \{(z_1,\underline{x}_1),(z_2,\underline{x}_2),....,(z_N,\underline{x}_N)\}$

subtract mean from z and $\underline{\phi}(\underline{x})$ prior to regression as in LS because intercept $w_0 = \bar{z} - \underline{w}^T\underline{\bar{\phi}}$

$error = z - \bar{z} - y(\underline{x}); y(\underline{x}) = \underline{w}^T(\underline{\phi}(\underline{x}) - \underline{\bar{\phi}}\underline{e})$

$X^T = \left[\underline{\phi}(\underline{x}_1) - \underline{\bar{\phi}} \quad \underline{\phi}(\underline{x}_2) - \underline{\bar{\phi}} \quad . \quad \underline{\phi}(\underline{x}_N) - \underline{\bar{\phi}}\right] = \Phi;\ p\ by\ N\ matrix$

$\text{Re}call\ \hat{\underline{w}} = (X^TX + \mu I_p)^{-1}X^T\underline{z} = \left[\frac{1}{\mu}X^T - \frac{1}{\mu^2}X^T(I_N + \frac{XX^T}{\mu})^{-1}XX^T\right](\underline{z} - \bar{z})$

$= \left[\frac{1}{\mu}X^T - \frac{1}{\mu}X^T(\mu I_N + XX^T)^{-1}(XX^T + \mu I_N - \mu I_N)\right](\underline{z} - \bar{z}\underline{e})$

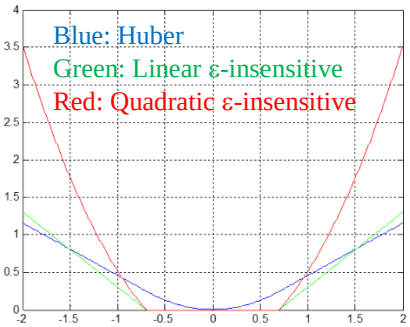
$= X^T\underbrace{(\mu I_N + XX^T)^{-1}(\underline{z} - \bar{z}\underline{e})}_{\underline{\alpha}} = \Phi\underline{\alpha}$

$Linear\ estimate: \underline{w} = X^T\underline{\alpha} = \sum_{n=1}^{N}\alpha_n(\underline{\phi}(\underline{x}_n) - \underline{\bar{\phi}}\underline{e}) = \Phi\underline{\alpha}$

$XX^T = \Phi^T\Phi = [K(\underline{x}_i,\underline{x}_j)] = [(\underline{\phi}(\underline{x}_i) - \underline{\bar{\phi}}\underline{e})^T(\underline{\phi}(\underline{x}_j) - \underline{\bar{\phi}}\underline{e})]$ *NxN symmetric Kernel* (*Grammian*)....*inner products*

At a test point $\underline{x}$: $\hat{z}(\underline{x}) = \hat{\underline{w}}^T(\underline{\phi}(\underline{x}) - \underline{\bar{\phi}}\underline{e}) + \bar{z} = \underline{\alpha}^T\Phi^T(\underline{\phi}(\underline{x}) - \underline{\bar{\phi}}\underline{e}) = \sum_{n=1}^{N}\alpha_n(\underline{\phi}(\underline{x}_n) - \underline{\bar{\phi}}\underline{e})^T(\underline{\phi}(\underline{x}) - \underline{\bar{\phi}}\underline{e})$....*inner products*

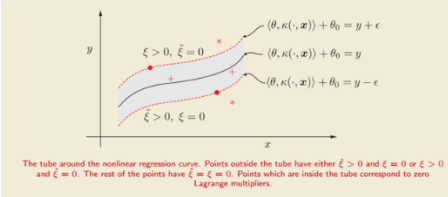
- SVM Regression



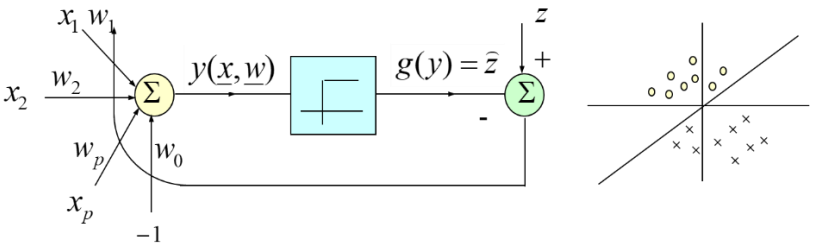
$$e = z - y(\underline{x})$$
$$y(\underline{x}) = \underline{w}^T \underline{\Phi}(\underline{x}) + w_0$$
$$Huber:$$
$$L_\varepsilon(e) = \begin{cases} \varepsilon\,|e| - \dfrac{\varepsilon^2}{2}; |e| > \varepsilon \\ \dfrac{e^2}{2}; |e| \le \varepsilon \end{cases}$$

$$Linear\ \varepsilon - insensitive:$$
$$L_\varepsilon(e) = \begin{cases} |e| - \varepsilon; |e| > \varepsilon \\ 0; |e| \le \varepsilon \end{cases}$$

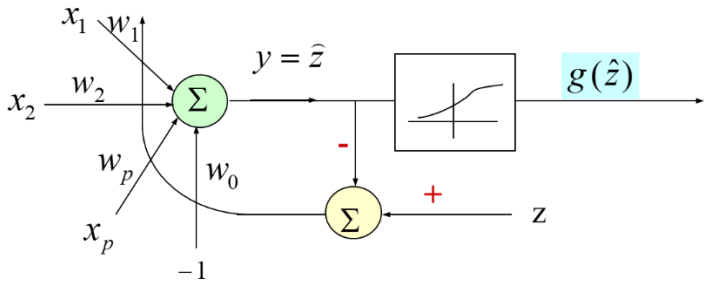
$$Quadratic\ \varepsilon - insensitive:$$
$$L_\varepsilon(e) = \begin{cases} e^2 - \varepsilon^2; |e| > \varepsilon \\ 0; |e| \le \varepsilon \end{cases}$$



Rosenblatt's Perceptron can categorize linearly separable classes in low dimensions



Widrow-Hoff's LMS: can approximate functions:



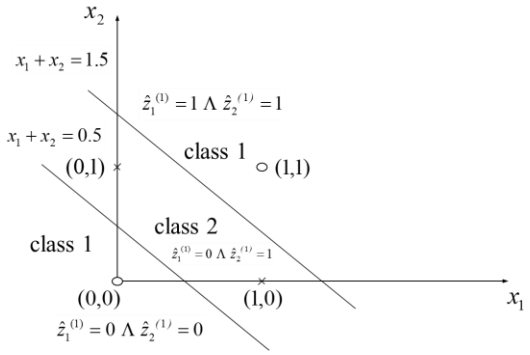
Logistic comes naturally from binary classification. Softmax for multiple classes.

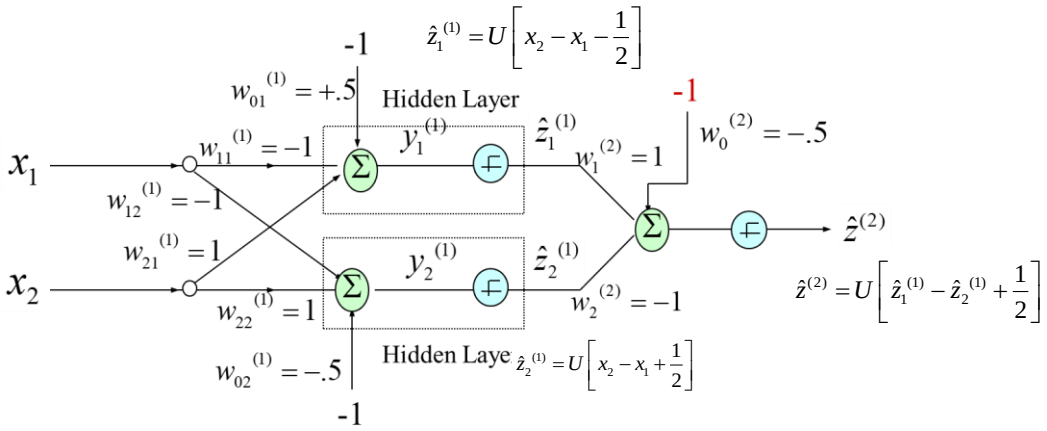
But can't solve XOR problem!

Multi-layer perceptron: What if we have multiple layers…. Just like our vision system

- The cortex of human brain can be seen as a multilayer architecture with 5-10 layers dedicated only to our visual system (Hubel and Wiesel, 1962). Using networks with more layers can lead to more compact representations of the input-output mapping.

- Hastad switching lemma (1987) – Boolean circuits representable with a polynomial number of nodes with k layers may require an exponential number of nodes with (k-1) layers (e.g., parity) … more depth is better. Deep networks may require less number of weights than a shallow representation
- Humans think in terms of hierarchies: Feature hierarchies identified in deep networks can potentially be shared among multiple tasks





$\hat{z}_1^{(1)} = U\left[x_2 - x_1 - \frac{1}{2}\right]$ $\hat{z}_2^{(1)} = U\left[x_2 - x_1 + \frac{1}{2}\right]$ $\hat{z}^{(2)} = U\left[z_1^{(1)} - z_2^{(1)} + \frac{1}{2}\right]$

x_1	x_2	$\hat{z}_1^{(1)}$	$\hat{z}_2^{(1)}$	$\hat{z}^{(2)}$	
0	0	0	1	0	Class 1 (o)
0	1	1	1	1	Class 2 (x)
1	0	0	0	1	Class 2 (x)
1	1	0	1	0	Class 1 (o)

Alternate solution $\hat{z}_1^{(1)} = U[x_1 + x_2 - 1.5]$ $\hat{z}_2^{(1)} = U\left[x_1 + x_2 - \frac{1}{2}\right]$ $\hat{z}^{(2)} = U\left[\hat{z}_2^{(1)} - 2\hat{z}_1^{(1)} - \frac{1}{2}\right]$

x_1	x_2	$\hat{z}_1^{(1)}$	$\hat{z}_2^{(1)}$	$\hat{z}^{(2)}$	
0	0	0	0	0	Class 1 (o)
1	0	0	1	1	Class 2 (x)
0	1	0	1	1	Class 2 (x)
1	1	1	1	0	Class 1 (o)

- Universal approximation theorem: MLP is a universal approximator
- Training MLPs: Layer 0 inputs; Layer L outputs
- $\hat{z}_i(l) = i^{th}$ output of l^{th} *layer or state at layer l*

  $= g(\hat{y}_i(l)) == g\left(\sum_{j=0}^{M(l-1)} w_{ij}(l)\hat{z}_j(l-1)\right)$; $\underline{\hat{z}}(l) = \underline{g}(W(l)\underline{z}^{(l-1)})$

  If $g(z_i) = \frac{1}{1+e^{-\alpha z_i}} \Rightarrow g' = \alpha g(1-g)$

  If $g(z_i) = \tanh(\alpha z_i) \Rightarrow g' = \alpha(1-g^2)$
- Nested function

$$\Rightarrow \hat{z}_i(L) = g\left[\sum_{j_{L-1}=0}^{M(L-1)} w_{ij_{(L-1)}}(L)\cdots\cdots g\left[\sum_{j_1=0}^{M(1)} w_{j_2 j_1}(2)\cdots\cdots g\left[\sum_{j_0=0}^{M(0)} w_{j_1 j_0}(1)\ \underbrace{z_{j_0}(0)}_{\text{input features}}\right]\right]\right]$$

- Cost function: cross entropy, MSE,….
- Want to use incremental gradient descent or stochastic gradient descent algorithm

$$w_{ij}^{(n+1)}(l) = w_{ij}^{(n)}(l) - \eta\frac{\partial J}{\partial w_{ij}^{(n)}(l)} = w_{ij}^{(n)}(l) + \Delta w_{ij}^{(n)}(l)$$

- Back propagation algorithm is an essentially a chain rule

$$L(x) = L(f(h(g(x))$$

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial h}\frac{\partial h}{\partial g}\frac{\partial g}{\partial x} = \lambda_g\frac{\partial g}{\partial x}$$

$$\lambda_f = \frac{\partial L}{\partial f}; \lambda_h = \frac{\partial L}{\partial h} = \lambda_f\frac{\partial f}{\partial h};$$

$$\lambda_g = \frac{\partial L}{\partial g} = \lambda_h\frac{\partial h}{\partial g}$$

$$x \to g \to h \to f \to L$$

$$\lambda_g\frac{\partial g}{\partial x} \leftarrow \lambda_g \leftarrow \lambda_h \leftarrow \lambda_f$$

$$\frac{\partial J}{\partial w_{ij}(l)} = \frac{\partial J}{\partial \hat{z}_i(l)}\frac{\partial \hat{z}_i(l)}{\partial \hat{y}_i(l)}\frac{\partial \hat{y}_i(l)}{\partial w_{ij}(l)} == \lambda_i(l)g'(\hat{y}_i(l))\hat{z}_j(l-1); \lambda_i(l) = \frac{\partial J}{\partial \hat{z}_i(l)}$$

$$\lambda_i(L) = \frac{\partial J}{\partial \hat{z}_i(L)} = \hat{z}_i(L) - z_i$$

$$\lambda_i(l) = \frac{\partial J}{\partial \hat{z}_i(l)} = \sum_{j=0}^{M(l+1)} \frac{\partial J}{\partial \hat{z}_j(l+1)} \frac{\partial \hat{z}_j(l+1)}{\partial \hat{y}_j(l+1)} \frac{\partial \hat{y}_j(l+1)}{\partial \hat{z}_i(l)} = \sum_{j=0}^{M(l+1)} \lambda_j(l+1) \frac{\partial \hat{z}_j(l+1)}{\partial \hat{y}_j(l+1)} \frac{\partial \hat{y}_j(l+1)}{\partial \hat{z}_i(l)}$$

$$= \sum_{j=0}^{M(l+1)} \lambda_j(l+1) g'(\hat{y}_j(l+1)) w_{ji}(l+1)$$

$$\underline{\lambda}(l) = W^T(l+1)\ Diag\left(g'(\underline{\hat{y}}(l+1))\right)\underline{\lambda}(l+1)\ \ldots.linear\ running\ backwards\ \text{in}\,time$$

$$W^{n+1}(l) = W^n(l) - \eta^{(n)}\ Diag\begin{bmatrix} g'(\hat{y}_0(l)) & & \\ & \ddots & \\ & & g'(\hat{y}_{M(l)}(l)) \end{bmatrix} .\underline{\lambda}(l)\underline{\hat{z}}^T(l-1)$$

Slight variation in implementation:

$$\delta_i(l) = \frac{\partial E}{\partial \hat{y}_i(l)} = \frac{\partial E}{\partial \hat{z}_i(l)} \frac{\partial \hat{z}_i(l)}{\partial \hat{y}_i(l)} = \lambda_i(l) g'(\hat{y}_i(l))$$

$$\delta_i(l) = \left[\sum_{j=0}^{M(l+1)} \delta_j(l+1) w_{ji}(l+1)\right] g'(\hat{y}_i(l)); \delta_i(L) = \left(\hat{z}_i(L) - z_i\right) g'(\hat{y}_i(L))$$

$$\underline{\delta}(l) = Diag\begin{bmatrix} g'(\hat{y}_0(l)) & & \\ & \ddots & \\ & & g'(\hat{y}_{M(l)}(l)) \end{bmatrix} W^T(l+1)\underline{\delta}(l+1)$$

$$W^{n+1}(l) = W^n(l) - \eta^{(n)}\ \underline{\delta}(l)\underline{\hat{z}}^T(l-1)$$

Discuss implementation: need for shuffling data; multiple stopping criteria (cost, gradient, weights,…), acceleration schemes, MEKA algorithm

Backprop practicalities:

- Nonlinearities
- Weight initialization (Xavier initialization)
- Layer-dependent step sizes (Smaller for higher layers because multipliers are smaller for lower layers)
- Batch normalization
- Target values for Classes: one hot coding
- Minibatch averaged gradient. Typical batch sizes from 32 to 256.
- Training with Noise: generate surrogate data with small noise added (e.g., 0.1 variance for scaled data). Good for unbalanced case.
- Training with hints: e.g., in fault diagnosis, add another output for the fault severity during training only. Remove it for testing.
- Randomized (Shuffled) training. Scale data to have zero mean and unit variance or between (0.15 0.85) for sigmoid or (-1.716 1.716) for tanh function. Use ReLU or Leaky ReLU. Restart if training error does not decrease fast enough. Multi-start
- Minimum training set size N = 10* number of weights

    - Perform cross validation or bootstrap. At worst, split N as follows: 65% Training; 10% Validation (use these to evaluate errors in training); 25% Testing (should not be seen and used only once)
    - Network size from prediction error
    - Adaptive step sizes: Momentum, adagrad, adam, RMSprop,…
    - NLP techniques: Conjugate gradient, Memoryless Quasi-Newton method, MEKA. 2nd order methods need big batch sizes, like 10,000.
    - Regularization (Network pruning techniques) to improve generalization. Build the simplest possible model or try to drive small weights to zero
    - Weight eliminator
    - Network pruning
    - Dropout
    - Early stopping
    - Cost function: cross entropy, MSE
- Predictive posterior for regression and classification
- Semi-supervised learning
- Variations