# Deep Learning: Shallow Supervised Learning

## Lecture 01

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

*bunescu@ohio.edu*

# Supervised Learning

- **Task** = learn an (unkown) function $t : X \rightarrow T$ that maps input instances $\mathbf{x} \in X$ to output targets $t(\mathbf{x}) \in T$:
  - **Classification**:
    - The output $t(\mathbf{x}) \in T$ is one of a finite set of discrete categories.
  - **Regression**:
    - The output $t(\mathbf{x}) \in T$ is continuous, or has a continuous component.

- Target function $t(\mathbf{x})$ is known (only) through (noisy) set of training examples:

$$(\mathbf{x}_1,t_1), (\mathbf{x}_2,t_2), \ldots (\mathbf{x}_n,t_n)$$

# Supervised Learning

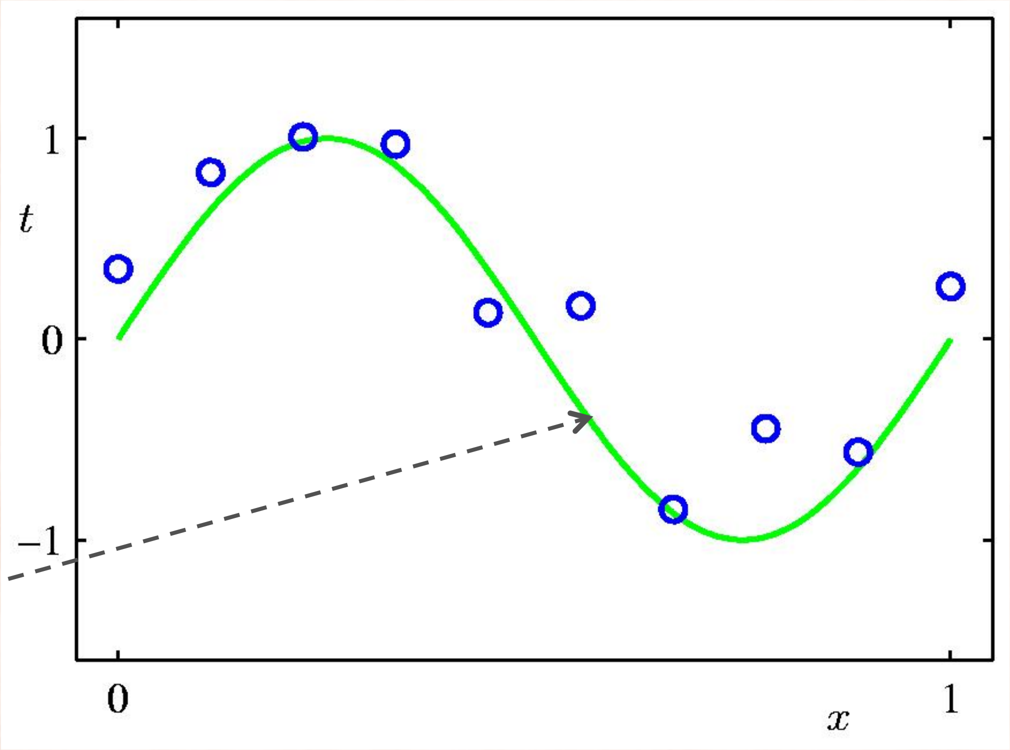
- **Task** = learn an (unknown) function $t : X \rightarrow T$ that maps input instances $\mathbf{x} \in X$ to output targets $t(\mathbf{x}) \in T$:
  - function $t$ is known (only) through (noisy) set of training examples:
    - Training/Test data: $(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), \ldots (\mathbf{x}_n, t_n)$

- **Task** = build a function $h(\mathbf{x})$ such that:
  - $h$ matches $t$ well on the *training data*:

    => $h$ is able to fit data that it has seen.
  - $h$ also matches target $t$ well on *test data*:

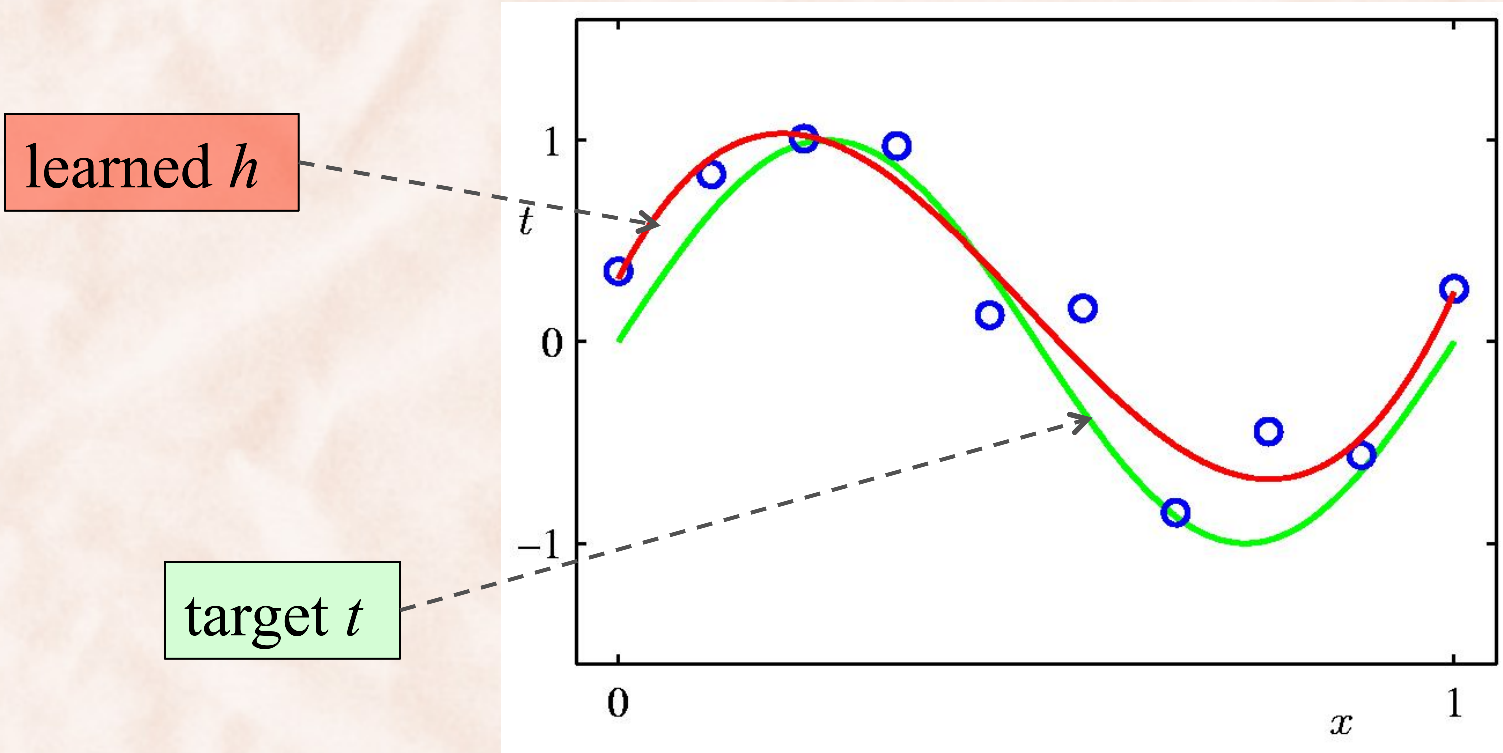
    => $h$ is able to generalize to unseen data.

# Regression: Curve Fitting



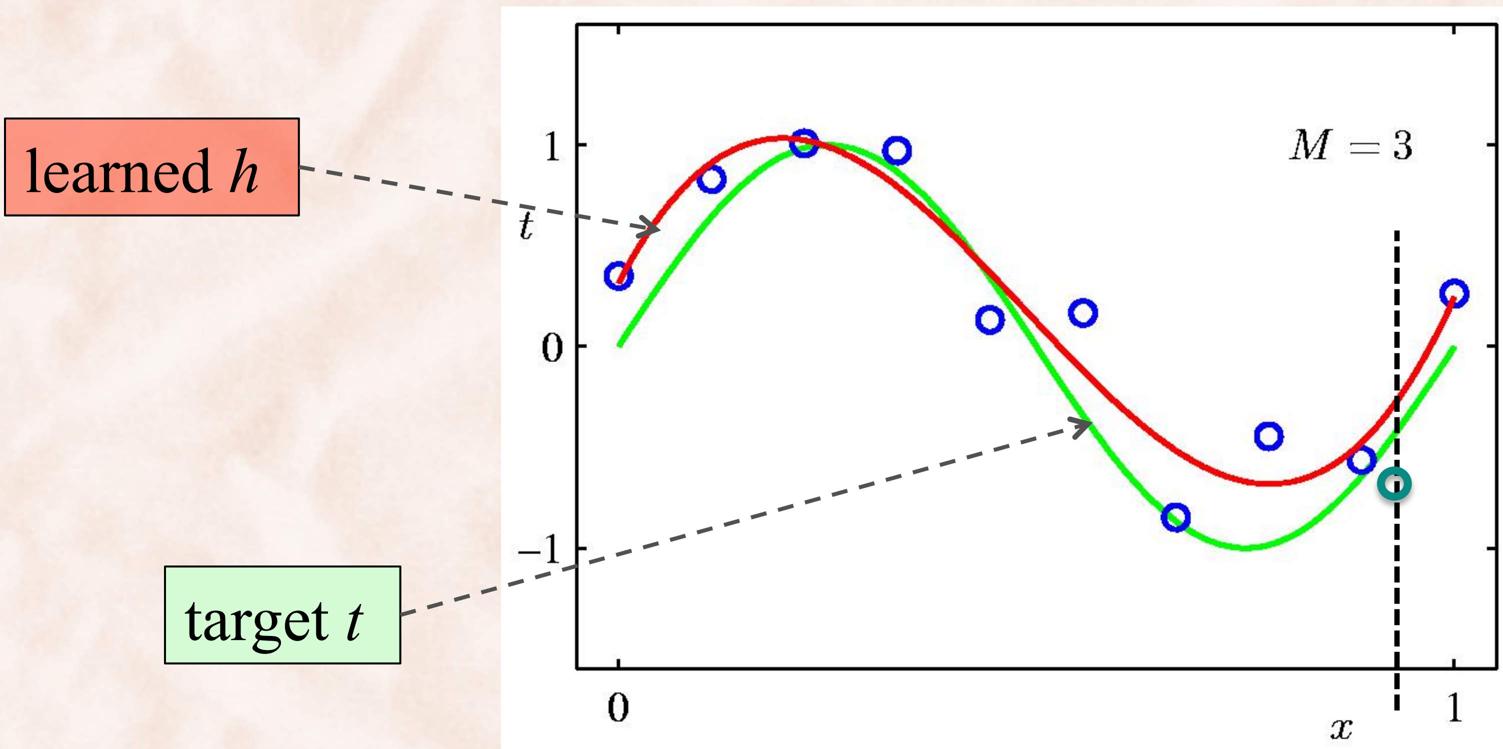

target $t$

# Regression: Curve Fitting



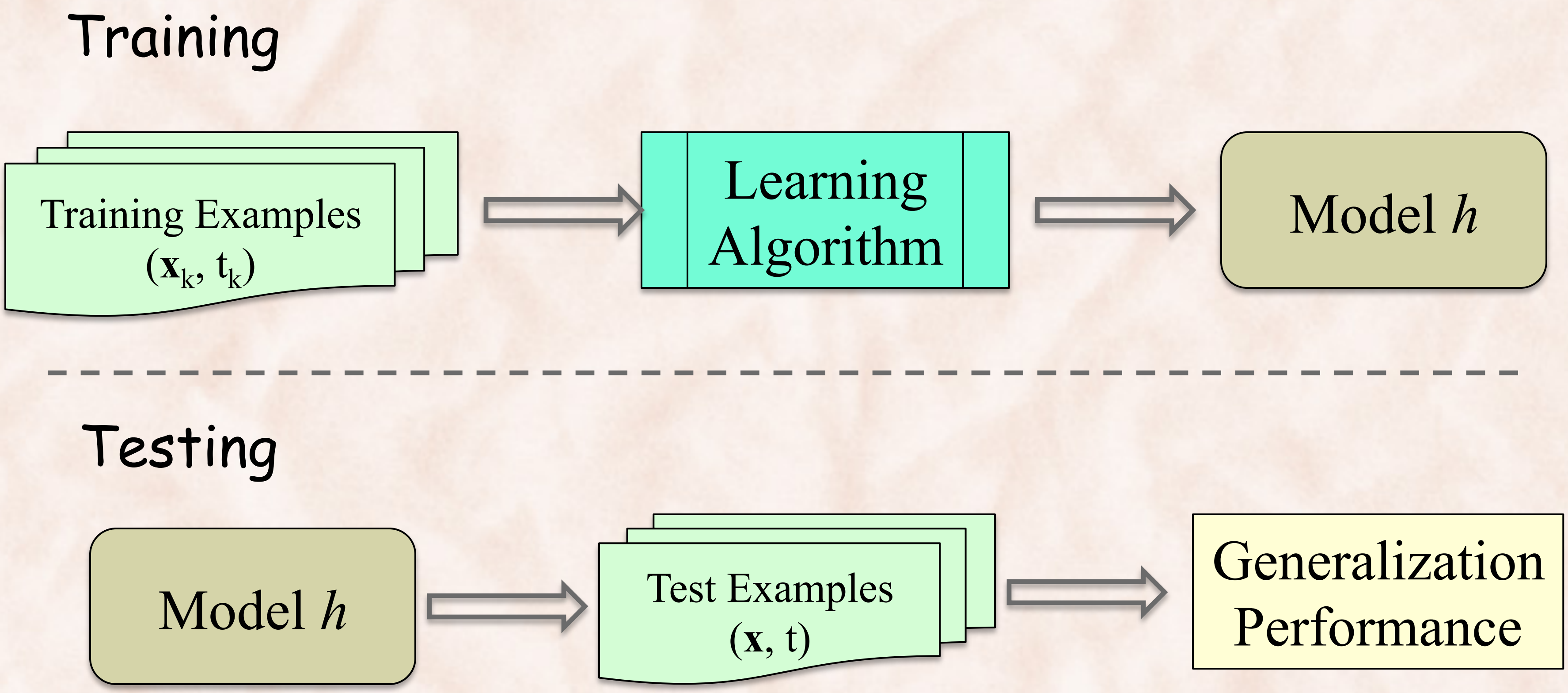

- **Training**: Build a function $h(\mathbf{x})$, based on training examples $(\mathbf{x}_1,t_1)$, $(\mathbf{x}_2,t_2)$, … $(\mathbf{x}_N,t_N)$

# Regression: Curve Fitting




- **Testing**: for arbitrary (unseen) instance $\mathbf{x} \in X$ , compute target output $h(\mathbf{x})$; want it to be close to $t(\mathbf{x})$.

# Supervised Learning

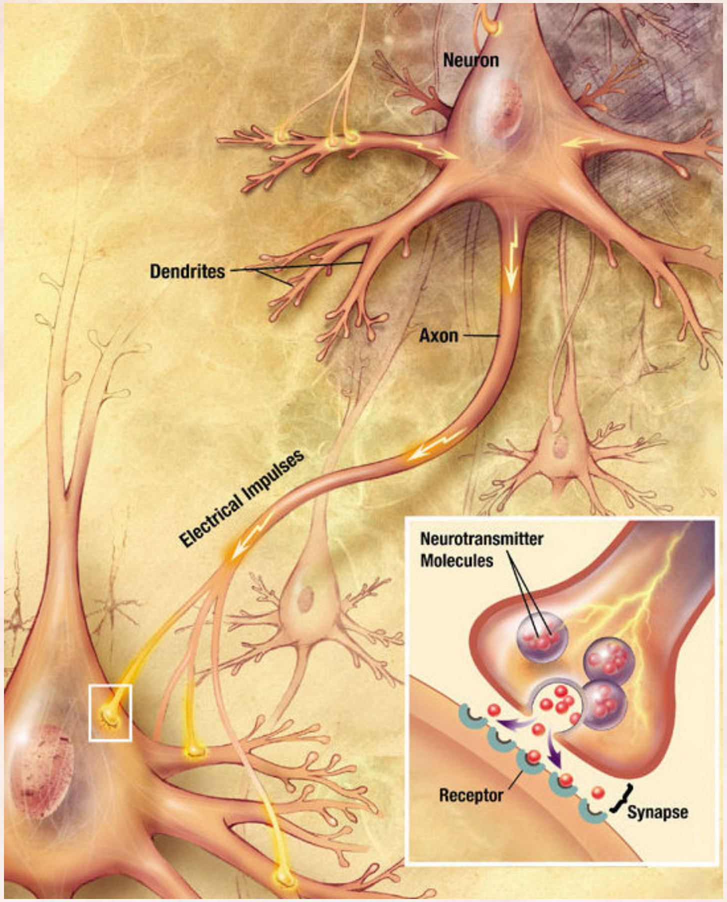


# Parametric Approaches to Supervised Learning

- **Task** = build a function $h(\mathbf{x})$ such that:
  - $h$ matches $t$ well on the training data:
    => $h$ is able to fit data that it has seen.
  - $h$ also matches $t$ well on test data:
    => $h$ is able to generalize to unseen data.

- **Task** = choose $h$ from a "nice" *class of functions* that depend on a vector of parameters $\mathbf{w}$:
  - $h(\mathbf{x}) \equiv h_{\mathbf{w}}(\mathbf{x}) \equiv h(\mathbf{w},\mathbf{x})$
  - **what classes of functions are "nice"?**

# Neurons




**Soma** is the central part of the neuron:
- *where the input signals are combined.*

**Dendrites** are cellular extensions:
- *where majority of the input occurs.*

**Axon** is a fine, long projection:
- *carries nerve signals to other neurons.*

**Synapses** are molecular structures between axon terminals and other neurons:
- *where the communication takes place.*

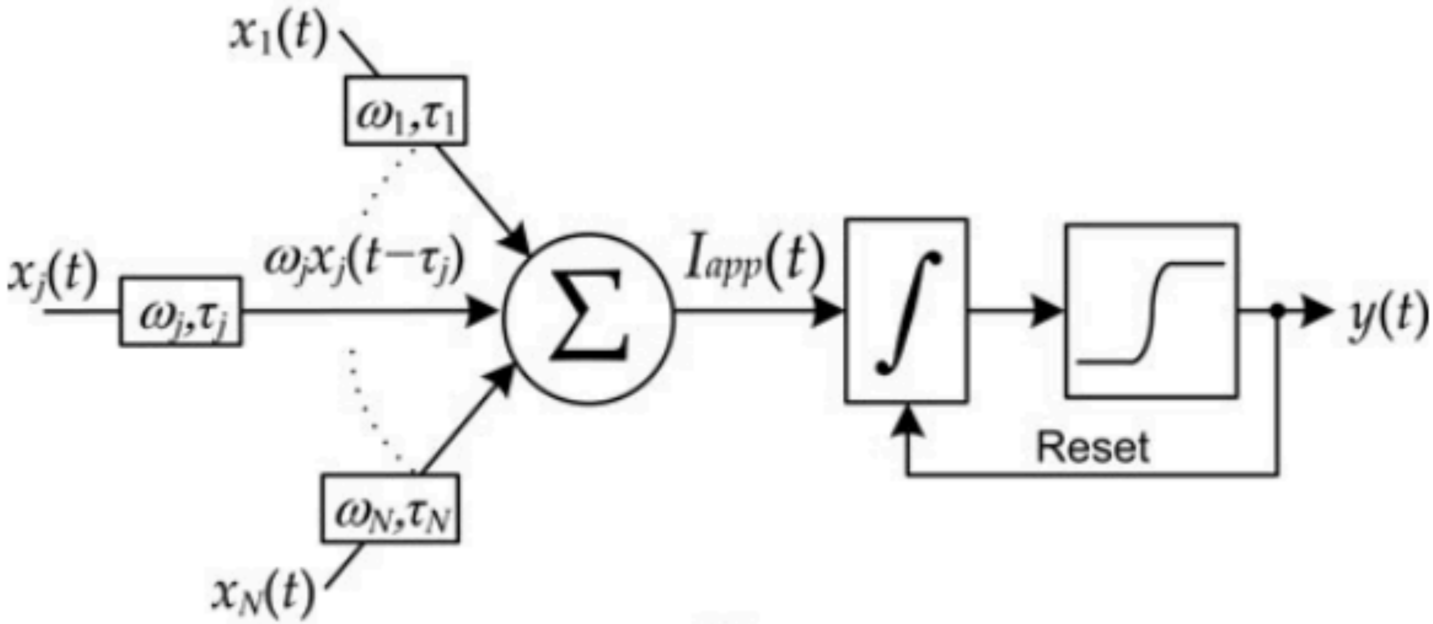
# Neuron Models

https://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf

| Year | Model Name | Reference |
|---|---|---|
| 1907 | Integrate and fire | [13] |
| 1943 | McCulloch and Pitts | [11] |
| 1952 | Hodgkin-Huxley | [12] |
| 1958 | Perceptron | [14] |
| 1961 | Fitzhugh-Nagumo | [15] |
| 1965 | Leaky integrate-and-fire | [16] |
| 1981 | Morris-Lecar | [17] |
| 1986 | Quadratic integrate-and-fire | [18] |
| 1989 | Hindmarsh-Rose | [19] |
| 1998 | Time-varying integrate-and-fire model | [20] |
| 1999 | Wilson Polynomial | [21] |
| 2000 | Integrate-and-fire or burst | [22] |
| 2001 | Resonate-and-fire | [23] |
| 2003 | Izhikevich | [24] |
| 2003 | Exponential integrate-and-fire | [25] |
| 2004 | Generalized integrate-and-fire | [26] |
| 2005 | Adaptive exponential integrate-and-fire | [27] |
| 2009 | Mihalas-Neibur | [28] |

# Spiking/LIF Neuron Function

http://ee.princeton.edu/research/prucnal/sites/default/files/06497478.pdf




(b)

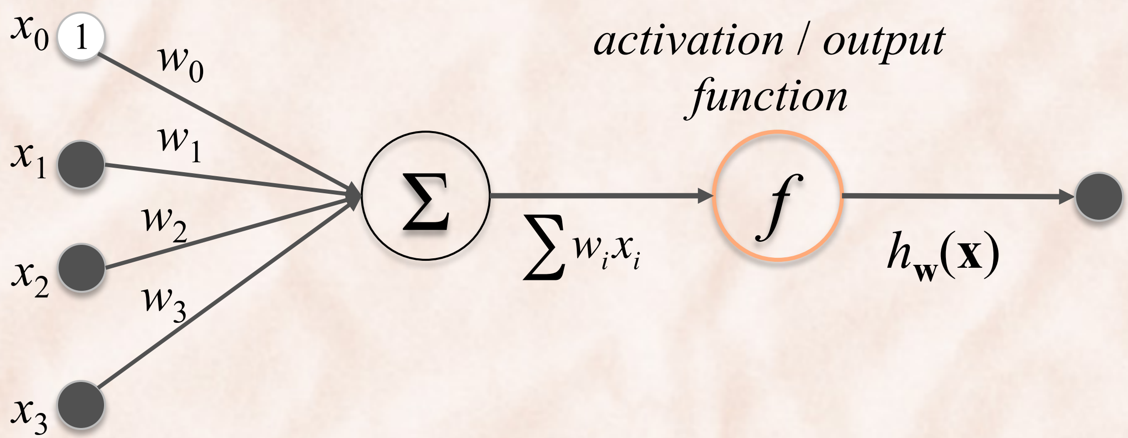
Fig. 2. (a) Illustration and (b) functional description of a leaky integrate-and-fire neuron. Weighted and delayed input signals are summed into the input current $I_{\text{app}}(t)$, which travel to the soma and perturb the internal state variable, the voltage $V$. Since $V$ is hysteric, the soma performs integration and then applies a threshold to make a spike or no-spike decision. After a spike is released, the voltage $V$ is reset to a value $V_{\text{reset}}$. The resulting spike is sent to other neurons in the network.

# Neuron Models

https://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf

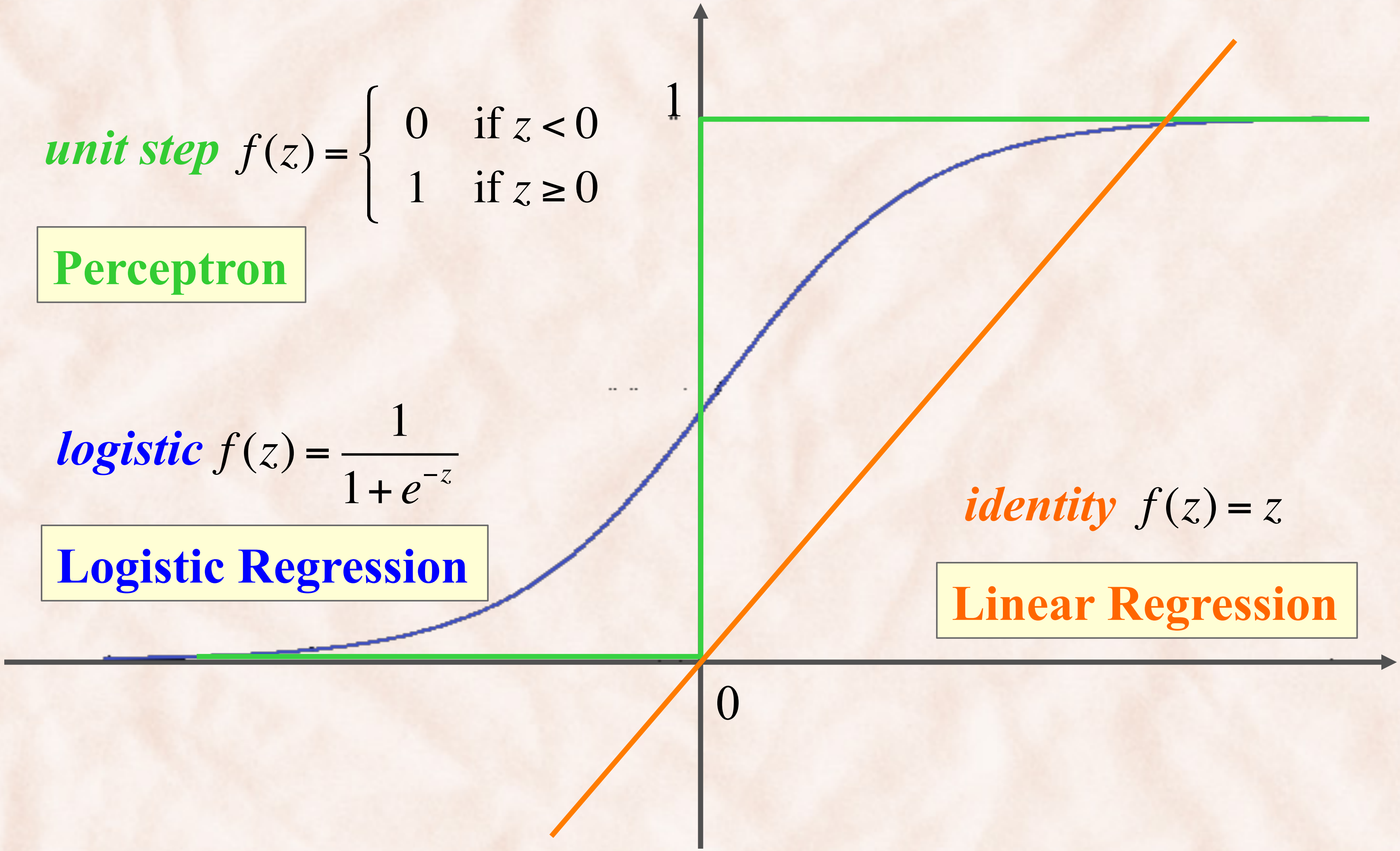
| Year | Model Name | Reference |
|---|---|---|
| 1907 | Integrate and fire | [13] |
| 1943 | McCulloch and Pitts | [11] |
| 1952 | Hodgkin-Huxley | [12] |
| 1958 | Perceptron | [14] |
| 1961 | Fitzhugh-Nagumo | [15] |
| 1965 | Leaky integrate-and-fire | [16] |
| 1981 | Morris-Lecar | [17] |
| 1986 | Quadratic integrate-and-fire | [18] |
| 1989 | Hindmarsh-Rose | [19] |
| 1998 | Time-varying integrate-and-fire model | [20] |
| 1999 | Wilson Polynomial | [21] |
| 2000 | Integrate-and-fire or burst | [22] |
| 2001 | Resonate-and-fire | [23] |
| 2003 | Izhikevich | [24] |
| 2003 | Exponential integrate-and-fire | [25] |
| 2004 | Generalized integrate-and-fire | [26] |
| 2005 | Adaptive exponential integrate-and-fire | [27] |
| 2009 | Mihalas-Neibur | [28] |

# McCulloch-Pitts Neuron Function




- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights $w_i$ correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through an **activation / output function**.

# Activation Functions




***unit step*** $f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

**Perceptron**

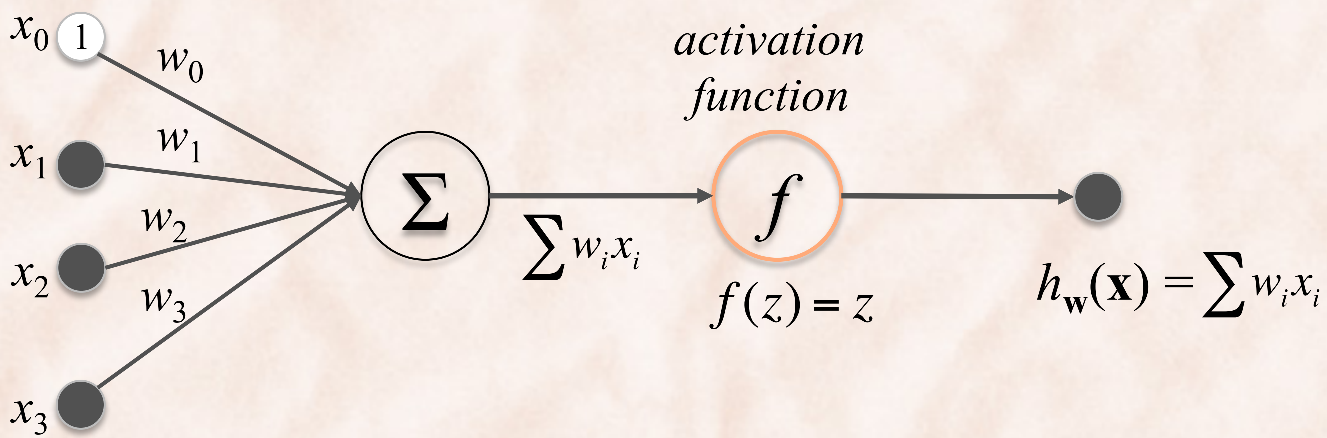
***logistic*** $f(z) = \dfrac{1}{1+e^{-z}}$

**Logistic Regression**

***identity*** $f(z) = z$

**Linear Regression**

# Linear Regression

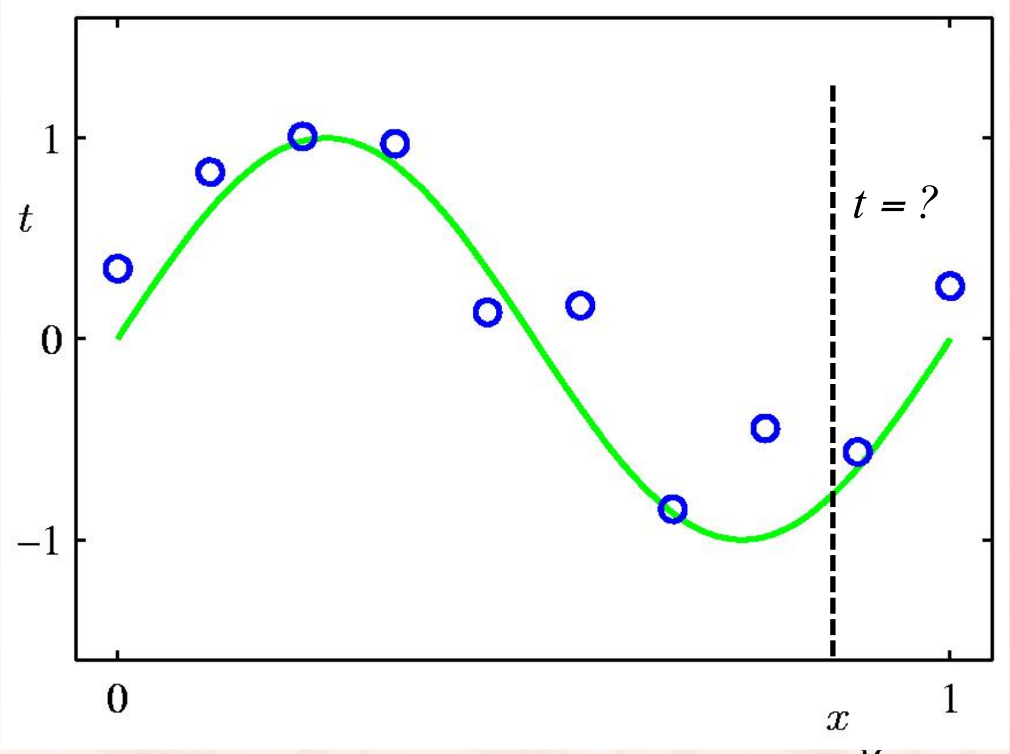



- Polynomial curve fitting is Linear Regression:

  $\mathbf{x} = \varphi(x) = [1, x, x^2, ..., x^M]^T$

  $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$

# Polynomial Curve Fitting




$$h(x) = h(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

*parameters* *features*

# Polynomial Curve Fitting

- Learning = finding the "right" parameters $\mathbf{w}^T = [w_0, w_1, \ldots, w_M]$
  - Find $\mathbf{w}$ that minimizes an *error / cost function* $E(\mathbf{w})$ which measures the misfit between $h(x_n, \mathbf{w})$ and $t_n$.
  - Expect that: $h(x_n,\mathbf{w})$ performing well on training examples $x_n \Rightarrow h(x,\mathbf{w})$ will perform well on arbitrary test examples $x \in X$.

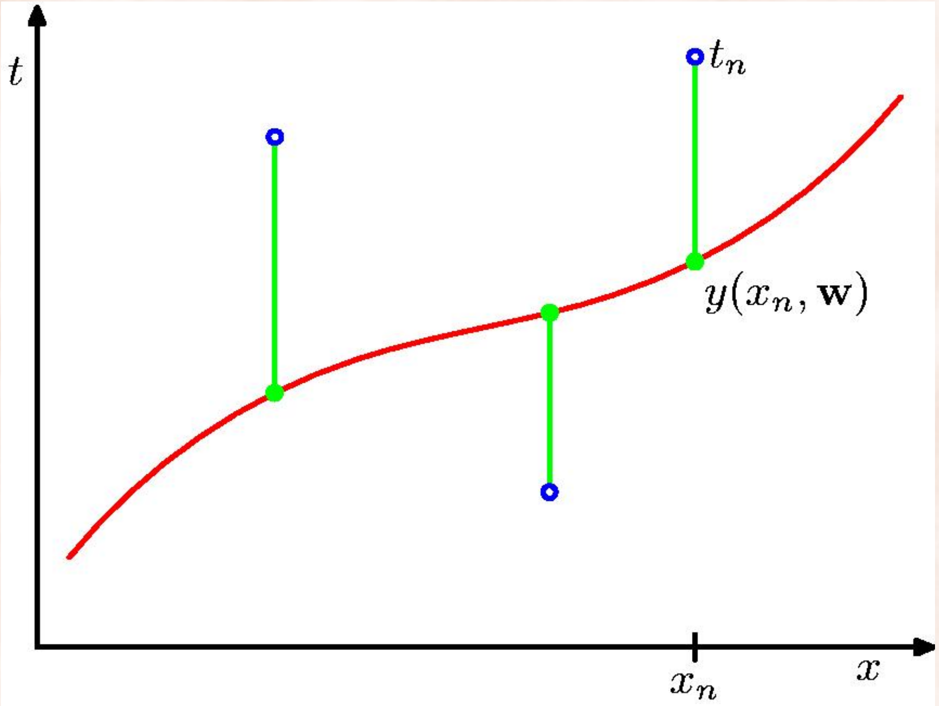
Inductive Learning Hyphotesis

- Least Squares error function:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{h(x_n, \mathbf{w}) - t_n\}^2$$

why squared?

# Polynomial Curve Fitting




$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{h(x_n, \mathbf{w}) - t_n\}^2$$

- How do we find **w*** that minimizes $E(\mathbf{w})$?

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E(\mathbf{w})$$

# Polynomial Curve Fitting

- *Least Square* solution is found by solving a set of M + 1 linear equations:

$$\sum_{j=0}^{M} A_{ij} w_j = T_i \text{ , where } A_{ij} = \sum_{n=1}^{N} x_n^{i+j} \text{, and } T_i = \sum_{n=1}^{N} t_n x_n^i$$

- Prove it.

# Gradient Descent (Batch)

- Want to minimize a function $f : R^{n} \rightarrow R$.
  - $f$ is differentiable and convex.
  - compute gradient of $f$ i.e. *direction of steepest increase*:

$$\nabla f(\mathbf{w}) = \left[ \frac{df}{dw_1}(\mathbf{w}), \frac{df}{dw_2}(\mathbf{w}), \ldots, \frac{df}{dw_k}(\mathbf{w}) \right]$$

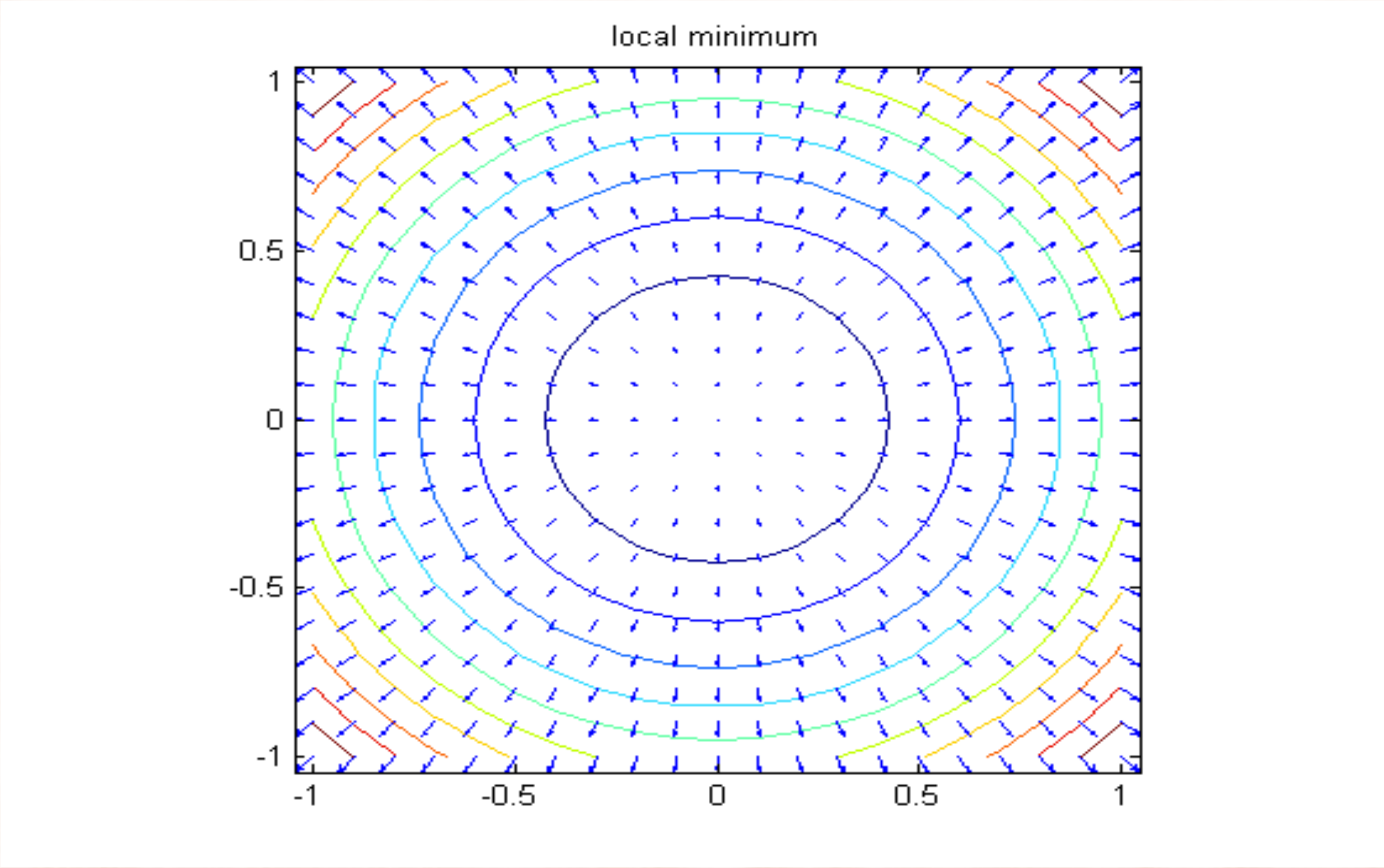
  - choose a sequence of points $\mathbf{w}^1$, $\mathbf{w}^2$, … and a learning rate $\eta$ such that:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla f(\mathbf{w}^{\tau})$$

- Sum-of-squares error: $E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{ \mathbf{w}^T \mathbf{x}_n - t_n \}^2$

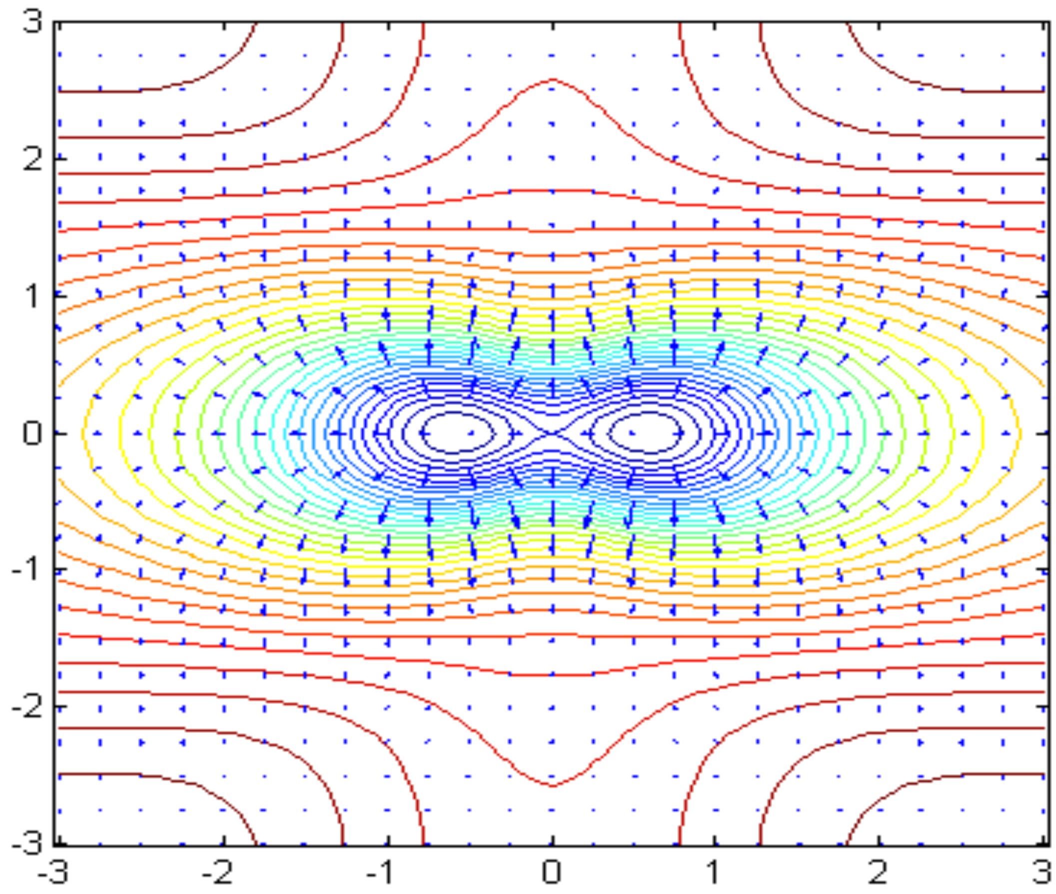
# Gradient Descent: Convex Objective

http://www2.math.umd.edu/~jmr/241/gradients.html



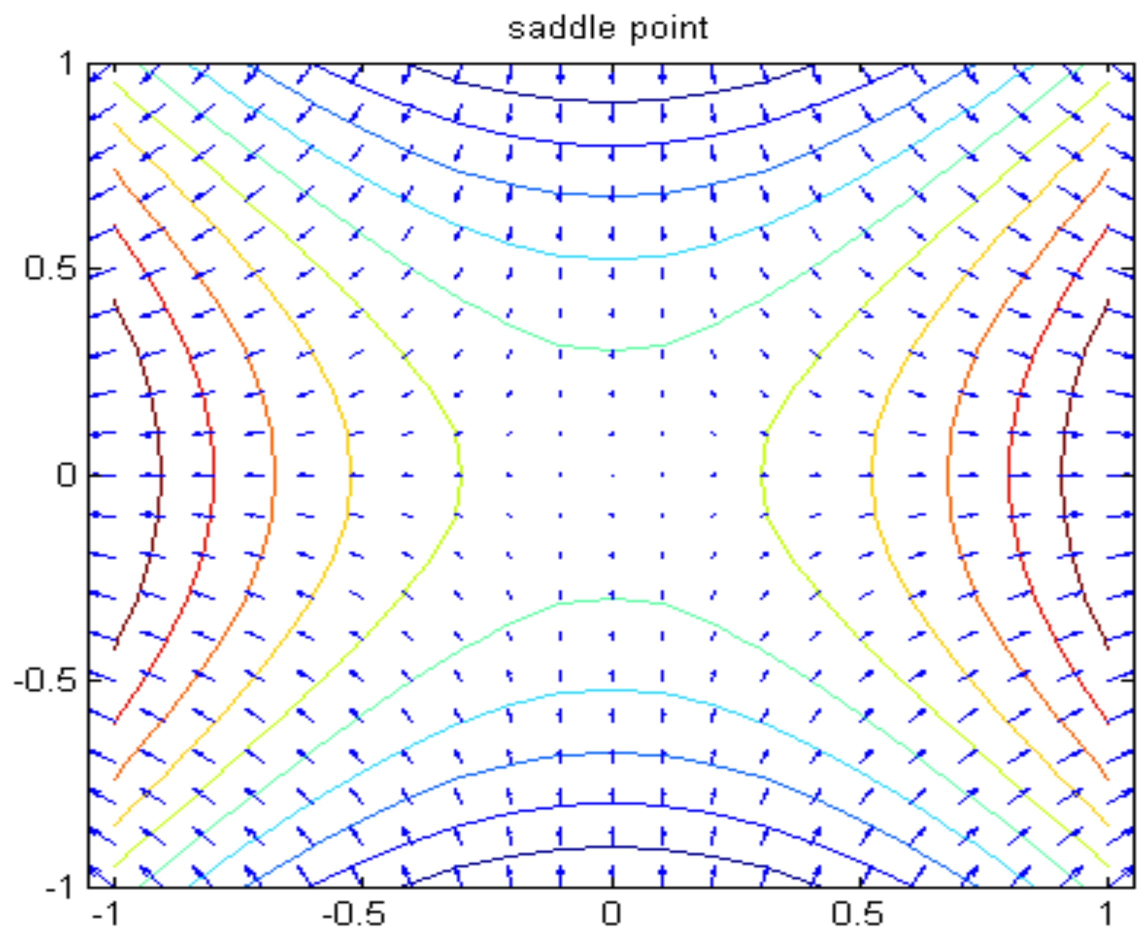
# Gradient Descent: Non-Convex Objective

http://www2.math.umd.edu/~jmr/241/gradients.html

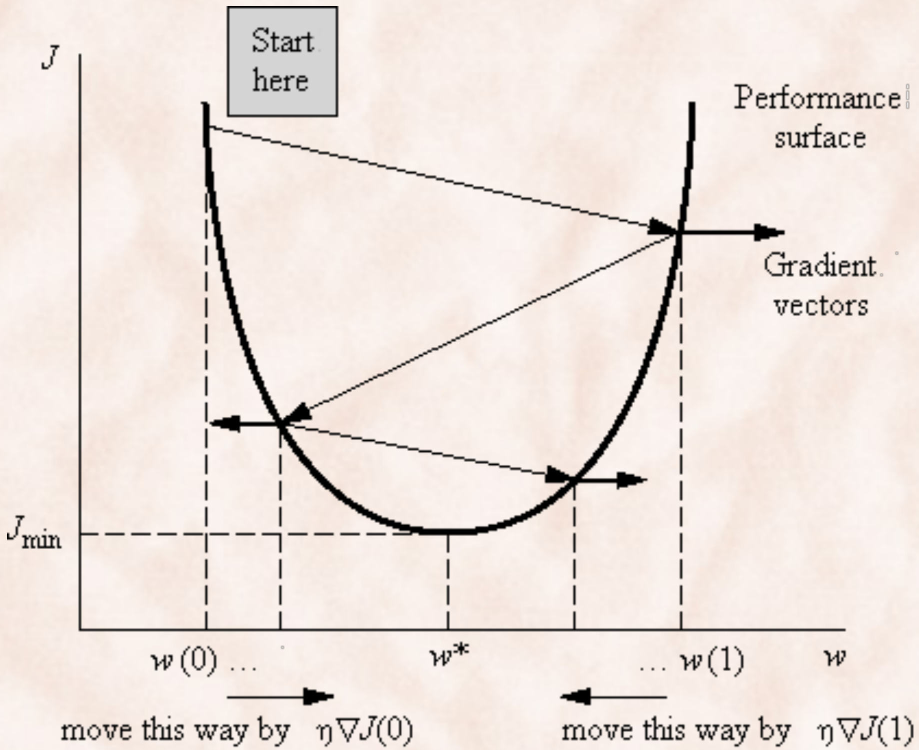


# Gradient Descent: Saddle Points & Plateaus

http://www2.math.umd.edu/~jmr/241/gradients.html



# Gradient Descent: Zig-Zagging Behavior



# Stochastic Gradient Descent (Online)

- Decompose error function in sum of example errors:

$$E(\mathbf{w}) = \sum_{n=1}^{N} \boxed{\frac{1}{2}\left(\mathbf{w}^T\mathbf{x}_n - t_n\right)^2} = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- Update parameters **w** after each example, sequentially:

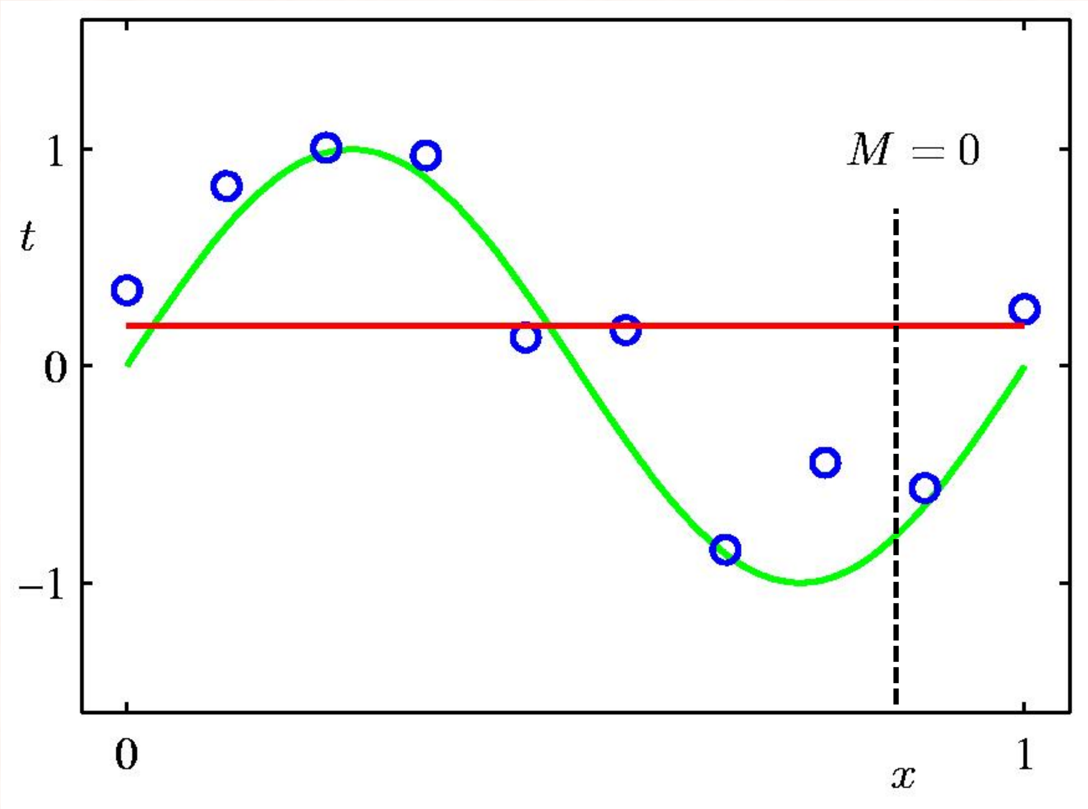
$$\begin{aligned}\mathbf{w}^{(\tau+1)} &= \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}) \\ &= \mathbf{w}^{(\tau)} + \eta (t_n - \mathbf{w}^{(\tau)T}\mathbf{x}_n)\mathbf{x}_n\end{aligned}$$

=> the *least-mean-square* (LMS) algorithm.

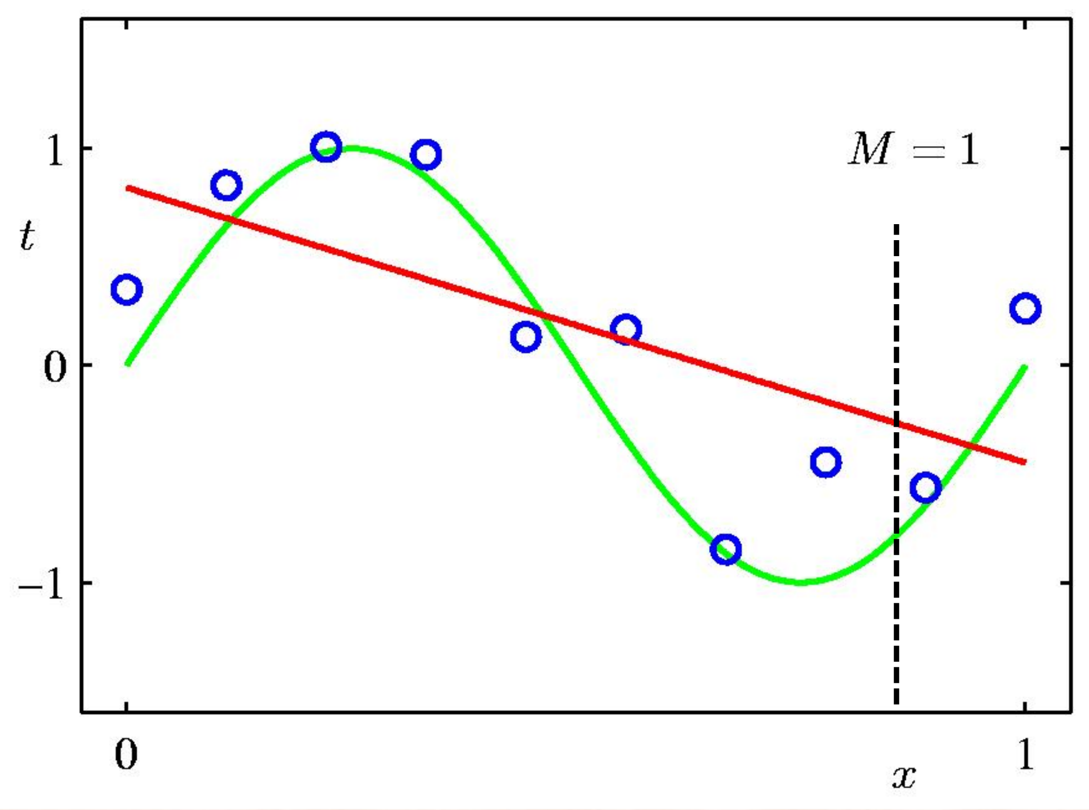
# Polynomial Curve Fitting

- Generalization = how well the parameterized $h(x,\mathbf{w}^*)$ performs on arbitrary (unseen) test instances $x \in X$.
- Generalization performance depends on the value of M.

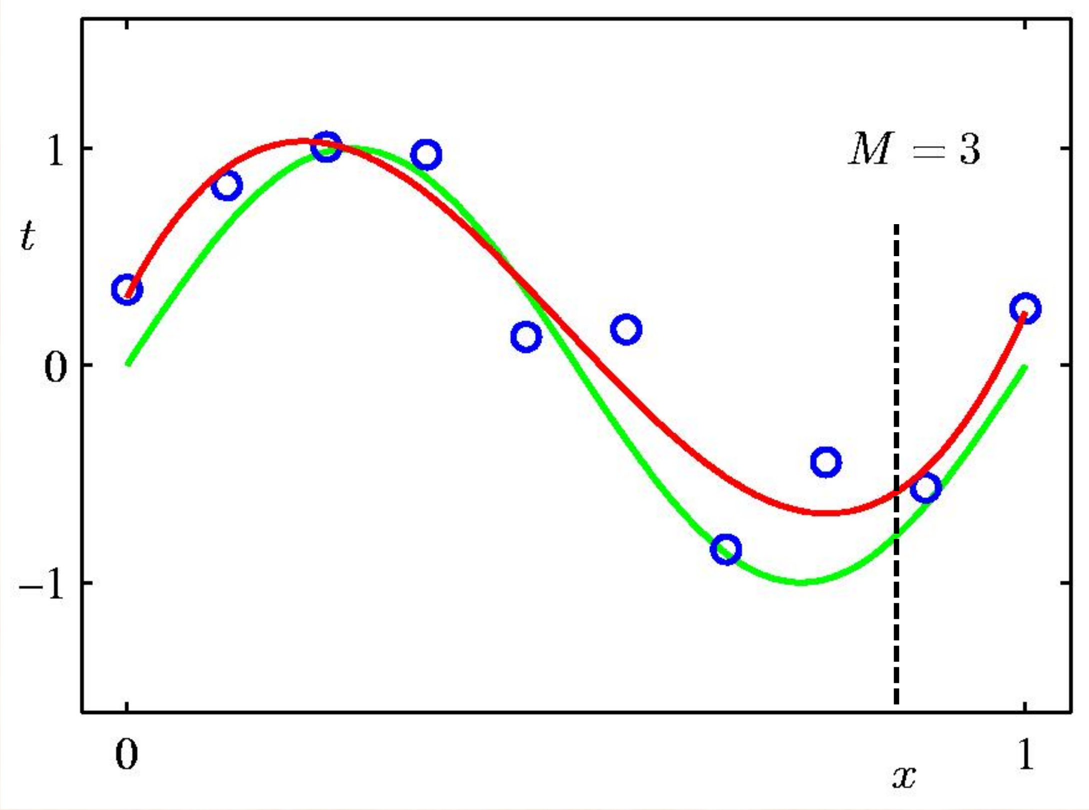
# $0^{th}$ Order Polynomial



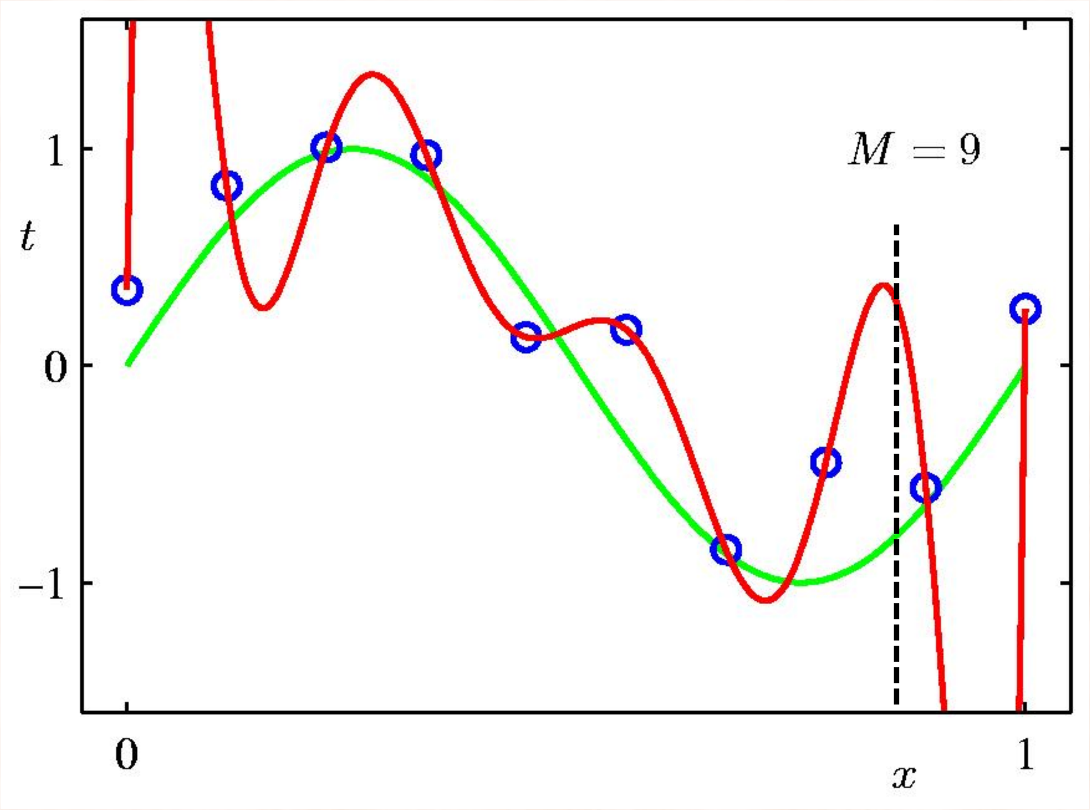
# $1^{st}$ Order Polynomial



# $3^{rd}$ Order Polynomial



# $9^{th}$ Order Polynomial



# Polynomial Curve Fitting

- Model Selection: choosing the order M of the polynomial.
  - Best fit obtained with M = 3.
  - M = 9 obtains poor fit, even though it fits training examples perfectly:
    - But M = 9 polynomials subsume M = 3 polynomials!

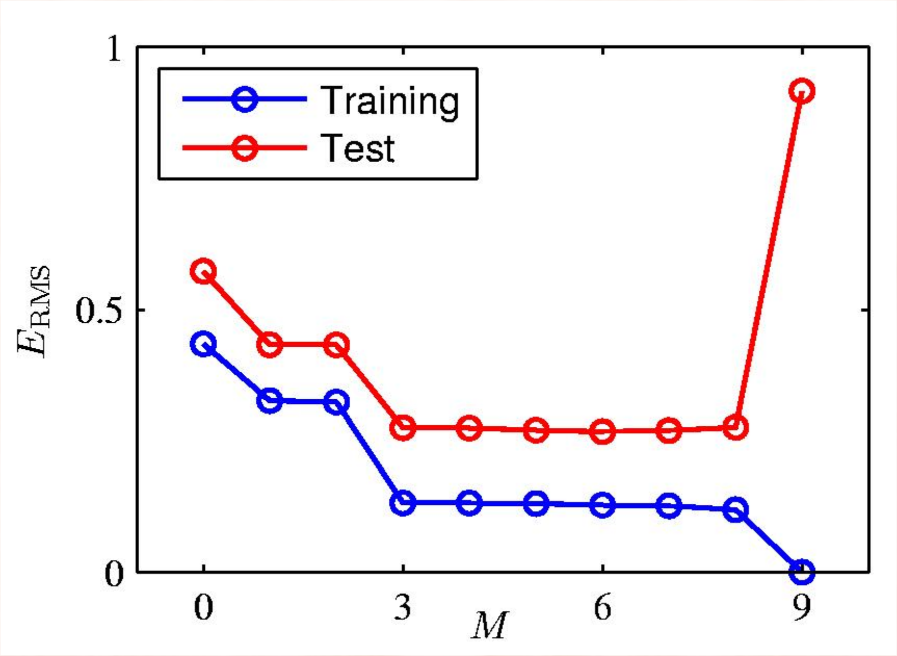
- Overfitting ≡ good performance on training examples, poor performance on test examples.

# Overfitting

- Measure fit using the Root-Mean-Square (RMS) error:

$$E_{RMS}(\mathbf{w}) = \sqrt{\frac{\sum_n \left(\mathbf{w}^T \mathbf{x}_n - t_n\right)^2}{N}}$$

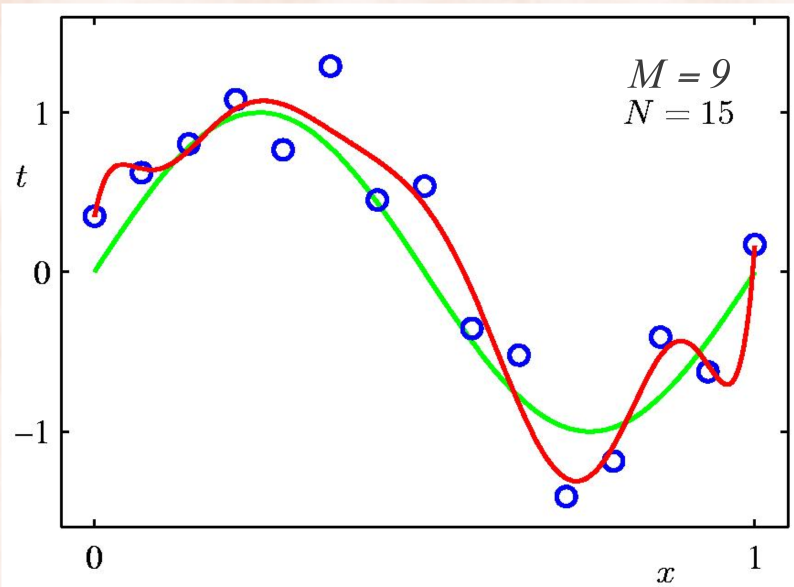
- Use 100 random test examples, generated in the same way:

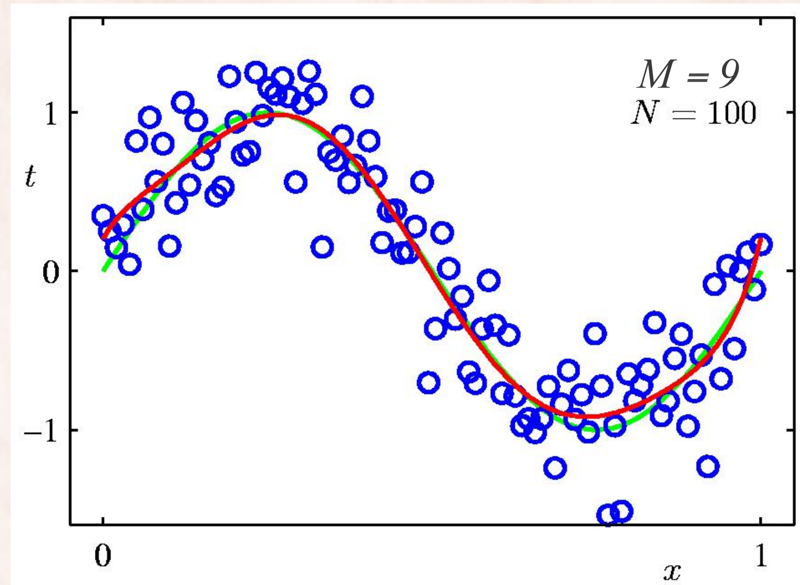


# Over-fitting and Parameter Values

| | $M=0$ | $M=1$ | $M=3$ | $M=9$ |
|---|---|---|---|---|
| $w_0^\star$ | 0.19 | 0.82 | 0.31 | 0.35 |
| $w_1^\star$ | | -1.27 | 7.99 | 232.37 |
| $w_2^\star$ | | | -25.43 | -5321.83 |
| $w_3^\star$ | | | 17.37 | 48568.31 |
| $w_4^\star$ | | | | -231639.30 |
| $w_5^\star$ | | | | 640042.26 |
| $w_6^\star$ | | | | -1061800.52 |
| $w_7^\star$ | | | | 1042400.18 |
| $w_8^\star$ | | | | -557682.99 |
| $w_9^\star$ | | | | 125201.43 |

# Overfitting vs. Data Set Size




- More training data ⇒ less overfitting.
- What if we do not have more training data?
  - Use **regularization**.

# Regularization

- **Parameter norm penalties** (term in the objective).
- Limit parameter norm (constraint).
- Dataset augmentation.
- Dropout.
- Ensembles.
- Semi-supervised learning.
- Early stopping.
- Noise robustness.
- Sparse representations.
- Adversarial training.

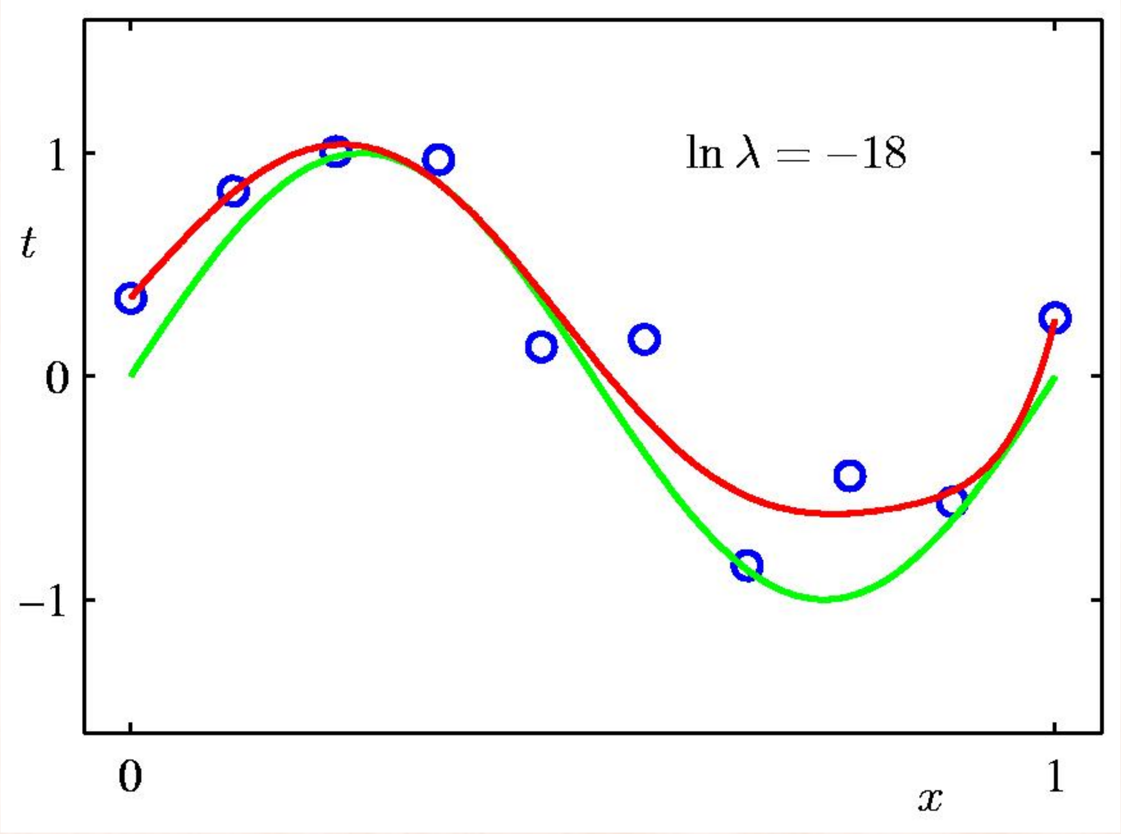
# Regularization

- Penalize large parameter values:

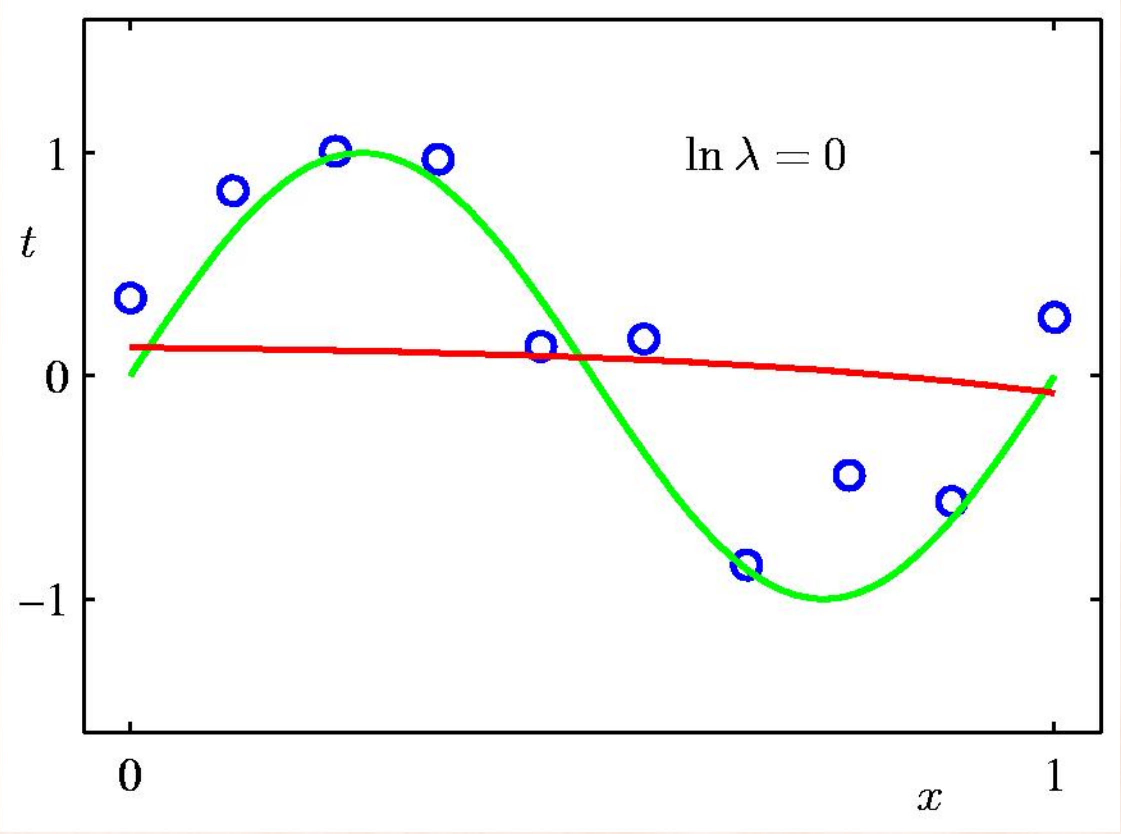
$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{h(x_n,\mathbf{w}) - t_n\}^2 + \underbrace{\frac{\lambda}{2}\|w\|^2}_{\textit{regularizer}}$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E(\mathbf{w})$$

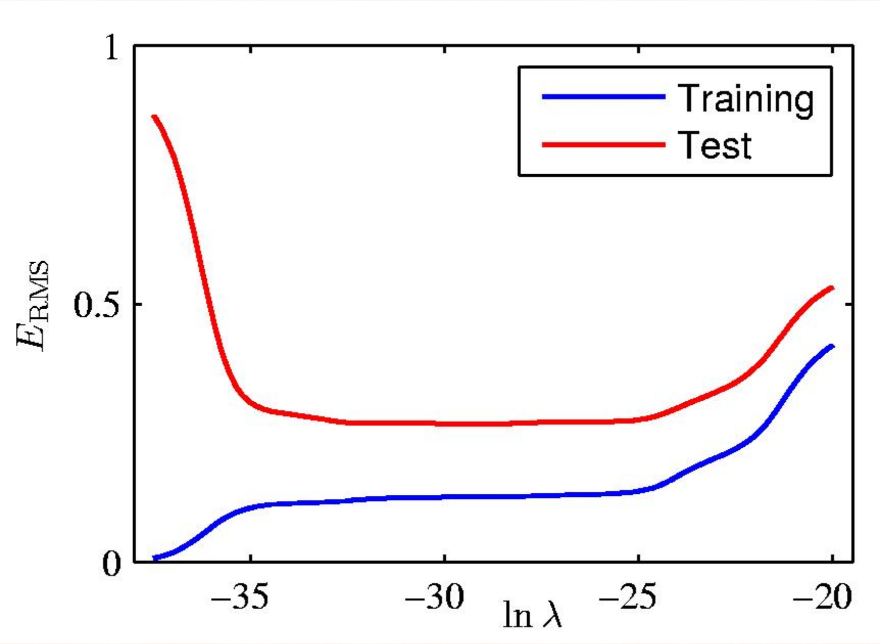
# $9^{th}$ Order Polynomial with Regularization



# $9^{th}$ Order Polynomial with Regularization



# Training & Test error vs. $\ln \lambda$

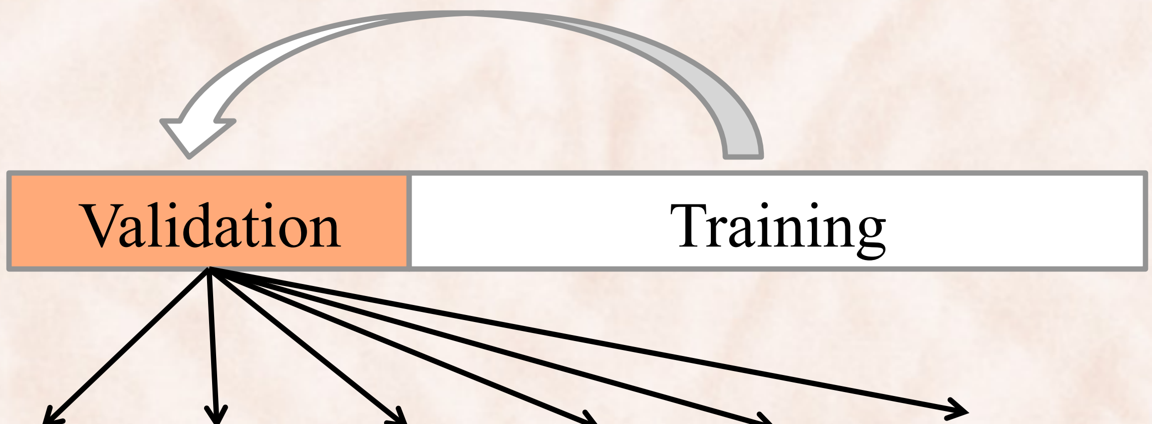



How do we find the optimal value of λ?

# Model Selection

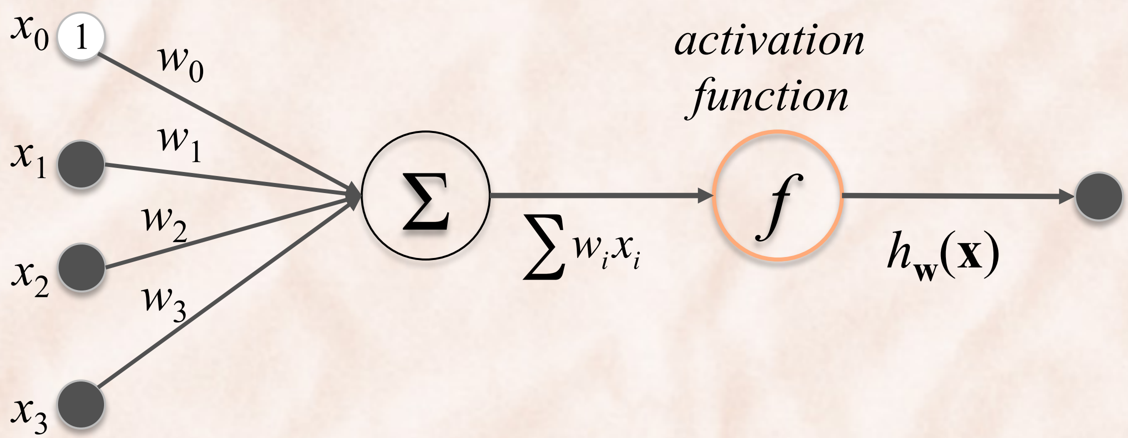
- Put aside an independent *validation set*.
- Select parameters giving best performance on validation set.

$$\ln\lambda \in \{-40,-35,-30,-25,-20,-15\}$$



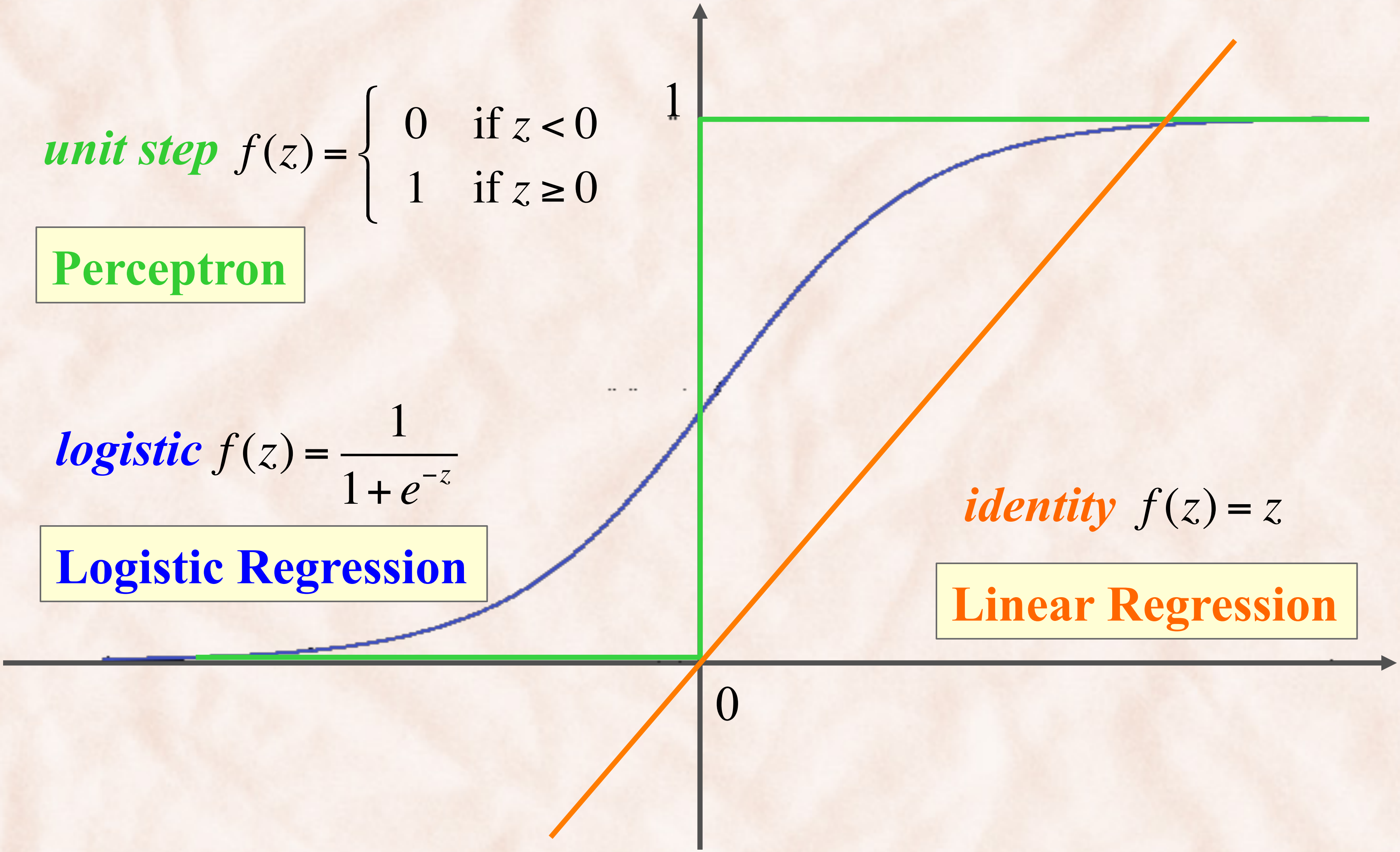

| $\ln\lambda$ | -40 | -35 | -30 | -25 | -20 | -15 |
|---|---|---|---|---|---|---|
| $E_{RMS}$ | 1.05 | 0.30 | $\boxed{0.25}$ | 0.27 | 0.52 | 0.55 |

# Neuron Function

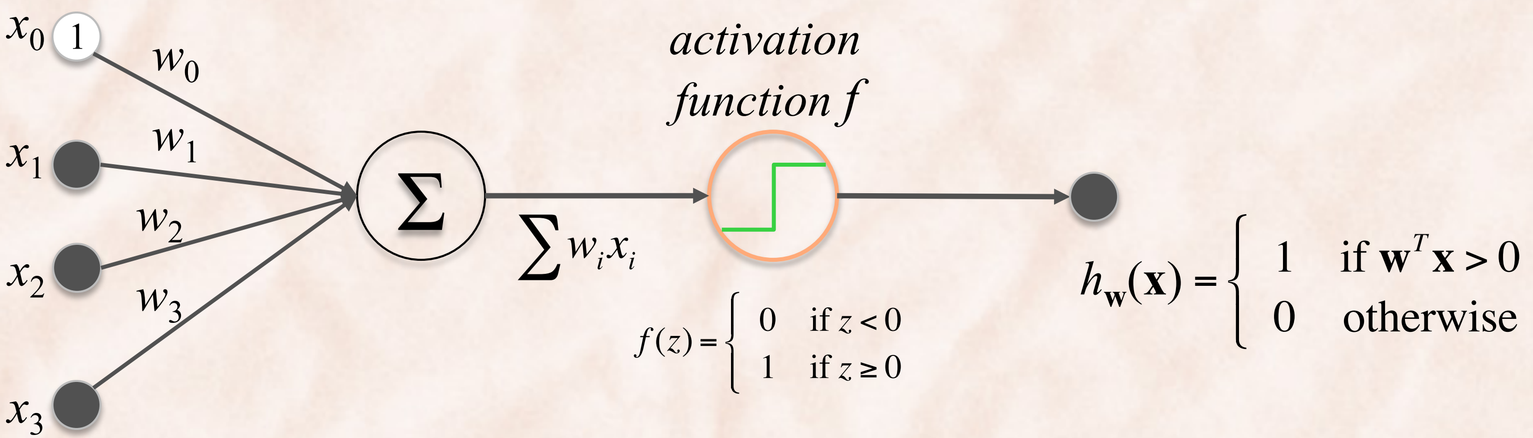



- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights $w_{\mathrm{i}}$ correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through a monotonic **activation function**.

# Activation Functions



# Perceptron




- Assume classes $T = \{c_1, c_2\} = \{1, 0\}$.
- Training set is $(\mathbf{x}_1,t_1), (\mathbf{x}_2,t_2), \ldots (\mathbf{x}_n,t_n)$.

  $\mathbf{x} = [1, x_1, x_2, \ldots, x_k]^T$

  $h(\mathbf{x}) = step(\mathbf{w}^T\mathbf{x})$

# Perceptron Learning

- Learning = finding the "right" parameters $\mathbf{w}^T = [w_0, w_1, \ldots, w_k]$
  - Find $\mathbf{w}$ that minimizes an *error function* $E(\mathbf{w})$ which measures the misfit between $h(\mathbf{x}_n, \mathbf{w})$ and $t_n$.
  - Expect that $h(\mathbf{x},\mathbf{w})$ performing well on training examples $x_n \Rightarrow h(x,\mathbf{w})$ will perform well on arbitrary test examples $\mathbf{x} \in X$.

- **Least Squares** error function?

$$E(\mathbf{w}) = \frac{1}{2}\boxed{\sum_{n=1}^{N}\{h(\mathbf{x}_n, \mathbf{w}) - t_n\}^2}$$

\# of mistakes

# Least Squares vs. Perceptron Criterion

- **Least Squares** => cost is # of misclassified patterns:
  - Piecewise constant function of $\mathbf{w}$ with discontinuities.
  - Cannot find closed form solution for $\mathbf{w}$ that minimizes cost.
  - Cannot use gradient methods (gradient zero almost everywhere).

- **Perceptron Criterion**:
  - Want $\mathbf{w}^T\mathbf{x}_n > 0$ for $t_n = 1$, and $\mathbf{w}^T\mathbf{x}_n < 0$ for $t_n = 0$.

  $\Rightarrow$ would like to have $\mathbf{w}^T\mathbf{x}_n(2t_n - 1) > 0$ for all patterns

  $\Rightarrow$ want to minimize $-\mathbf{w}^T\mathbf{x}_n(2t_n - 1)$ for all missclassified patterns M.

$$\Rightarrow \text{minimize } \boxed{E_P(\mathbf{w}) = -\sum_{n \in M} \mathbf{w}^T \mathbf{x}_n (2t_n - 1)}$$

# Stochastic Gradient Descent

- Update parameters $\mathbf{w}$ sequentially after each mistake:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_P(\mathbf{w}^{(\tau)}, \mathbf{x}_n)$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \eta \mathbf{x}_n (2t_n - 1)$$

- The magnitude of $\mathbf{w}$ is inconsequential => set $\eta = 1$.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \mathbf{x}_n (2t_n - 1)$$

# The Perceptron Algorithm: Two Classes

1. **initialize** parameters $\mathbf{w} = 0$
2. **for** $i = 1 \ldots n$
3. $\quad h_i = step(\mathbf{w}^{\mathrm{T}}\mathbf{x}_{\mathrm{i}})$
4. $\quad$ **if** $y_{\mathrm{i}} \neq t_i$ **then**
5. $\quad\quad \mathbf{w} = \mathbf{w} + \mathbf{x}_{\mathrm{i}}(2t_i - 1)$

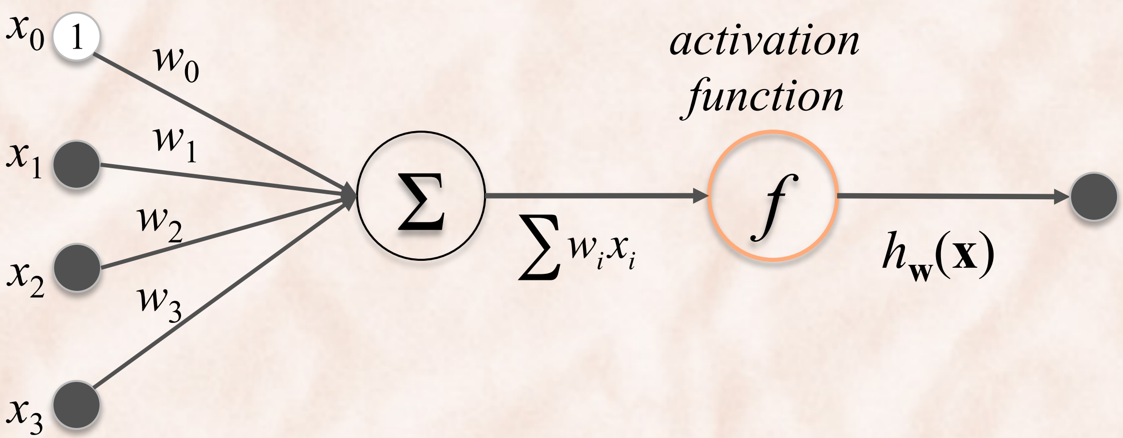
Repeat:
a) until convergence.
b) for a number of epochs E.

Theorem [Rosenblatt, 1962]:

If the training dataset is linearly separable, the perceptron learning algorithm is guaranteed to find a solution in a finite number of steps.

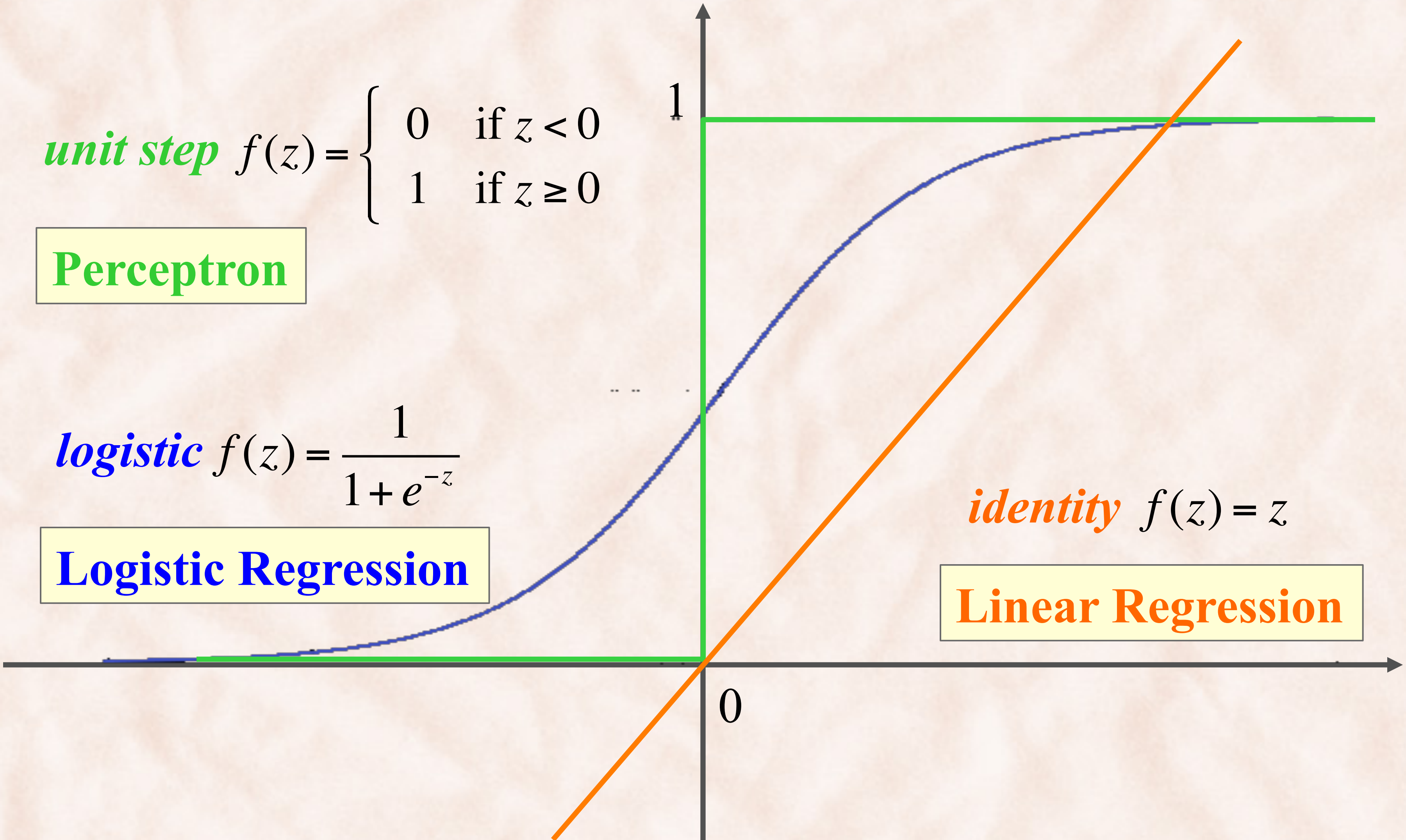
- see Theorem 1 (Block, Novikoff) in [Freund & Schapire, 1999].

# Neuron Function

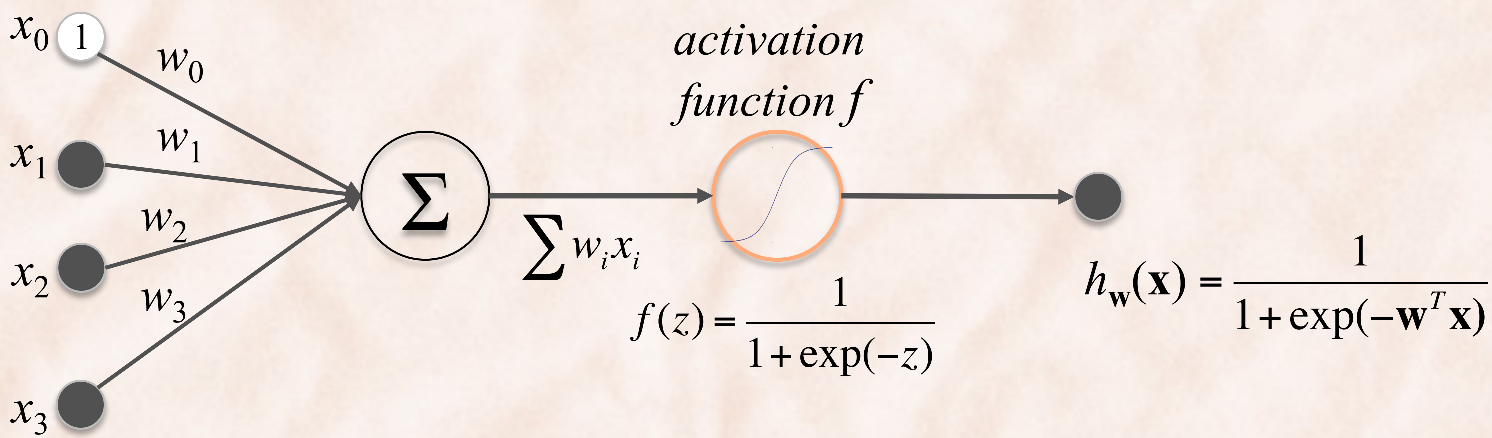



- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights $w_i$ correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through a monotonic **activation function**.

# Activation Functions



# Logistic Regression




- Can be used for both classification and regression:
  - Classification: $T = \{C_1, C_2\} = \{1, 0\}$.
  - Regression: $T = [0, 1]$ (i.e. output needs to be normalized).
- Training set is $(\mathbf{x}_1,t_1)$, $(\mathbf{x}_2,t_2)$, … $(\mathbf{x}_n,t_n)$.

$$\mathbf{x} = [1, x_1, x_2, ..., x_k]^T$$

$$h(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})$$

# Logistic Regression for Binary Classification

- Model output can be interpreted as **posterior class probabilities**:

$$p(C_1 \mid \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x}))}$$

$$p(C_2 \mid \mathbf{x}) = 1 - \sigma(\mathbf{w}^T \mathbf{x}) = \frac{\exp(-\mathbf{w}^T \mathbf{x})}{1 + \exp(-\mathbf{w}^T \mathbf{x})}$$

- How do we train a logistic regression model?
  - What **error/cost function** to minimize?

# Logistic Regression Learning

- Learning = finding the "right" parameters $\mathbf{w}^T = [w_0, w_1, \ldots, w_k]$
  – Find $\mathbf{w}$ that minimizes an *error function* $E(\mathbf{w})$ which measures the misfit between $h(\mathbf{x}_n, \mathbf{w})$ and $t_n$.
  – Expect that $h(\mathbf{x},\mathbf{w})$ performing well on training examples $\mathbf{x}_n \Rightarrow$ $h(\mathbf{x},\mathbf{w})$ will perform well on arbitrary test examples $\mathbf{x} \in X$.

- **Least Squares** error function?

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{h(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

  – Differentiable => can use gradient descent ✓
  – Non-convex => not guaranteed to find the global optimum ✗

# Maximum Likelihood

Training set is $D = \{\langle \mathbf{x}_n, t_n \rangle \mid t_n \in \{0,1\}, n \in 1 \ldots N\}$

Let $h_n = p(C_1 \mid \mathbf{x}_n) \Leftrightarrow h_n = p(t_n = 1 \mid \mathbf{x}_n) = \sigma(\mathbf{w}^T \mathbf{x}_n)$

**Maximum Likelihood (ML)** principle: find parameters that maximize the likelihood of the labels.

- The likelihood function is $p(\mathbf{t} \mid \mathbf{w}) = \prod_{n=1}^{N} h_n^{t_n} (1 - h_n)^{(1-t_n)}$
- The negative log-likelihood (cross entropy) error function:

$$E(\mathbf{w}) = -\ln p(\mathbf{t} \mid \mathbf{x}) = -\sum_{n=1}^{N} \{ t_n \ln h_n + (1 - t_n) \ln(1 - h_n) \}$$

# Maximum Likelihood Learning for Logistic Regression

- The ML solution is:

$$\mathbf{w}_{ML} = \arg\max_{\mathbf{w}} p(\mathbf{t} \mid \mathbf{w}) = \arg\min_{\mathbf{w}} \boxed{E(\mathbf{w})}$$

*convex in* **w**

- ML solution is given by $\nabla E(\mathbf{w}) = 0$.
  - Cannot solve analytically => solve numerically with gradient based methods: (stochastic) gradient descent, conjugate gradient, L-BFGS, etc.
  - Gradient is (prove it):

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (h_n - t_n)\mathbf{x}_n^T$$

# Regularized Logistic Regression

- Use a Gaussian prior over the parameters:

$$\mathbf{w} = [w_0, w_1, \ldots , w_M]^{\mathrm{T}}$$

$$p(\mathbf{w}) = N(\mathbf{0}, \alpha^{-1}\mathbf{I}) = \left(\frac{\alpha}{2\pi}\right)^{(M+1)/2} \exp\left\{-\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\right\}$$

- Bayes' Theorem:

$$p(\mathbf{w} \mid \mathbf{t}) = \frac{p(\mathbf{t} \mid \mathbf{w})\, p(\mathbf{w})}{p(\mathbf{t})} \propto p(\mathbf{t} \mid \mathbf{w})\, p(\mathbf{w})$$

- MAP solution:

$$\mathbf{w}_{MAP} = \arg\max_{\mathbf{w}} p(\mathbf{w} \mid \mathbf{t})$$

# Regularized Logistic Regression

- MAP solution:

$$\mathbf{w}_{MAP} = \arg\max_{\mathbf{w}} p(\mathbf{w} \mid \mathbf{t}) = \arg\max_{\mathbf{w}} p(\mathbf{t} \mid \mathbf{w}) p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} - \ln p(\mathbf{t} \mid \mathbf{w}) p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} - \ln p(\mathbf{t} \mid \mathbf{w}) - \ln p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} E_D(\mathbf{w}) - \ln p(\mathbf{w})$$

$$= \arg\min_{\mathbf{w}} E_D(\mathbf{w}) + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} \qquad \boxed{= \arg\min_{\mathbf{w}} E_D(\mathbf{w}) + E_{\mathbf{w}}(\mathbf{w})}$$

$$E_D(\mathbf{w}) = -\sum_{n=1}^{N} \{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \} \dashrightarrow \boxed{\textit{data term}}$$

$$E_{\mathbf{w}}(\mathbf{w}) = \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} \dashrightarrow \boxed{\textit{regularization term}}$$

# Regularized Logistic Regression

- MAP solution:

$$\mathbf{w}_{MAP} = \arg\min_{\mathbf{w}} \boxed{E_D(\mathbf{w}) + E_{\mathbf{w}}(\mathbf{w})}$$

*still convex in* **w**

- ML solution is given by $\nabla E(\mathbf{w}) = 0$.

$$\nabla E(\mathbf{w}) = \nabla E_D(\mathbf{w}) + \nabla E_{\mathbf{w}}(\mathbf{w}) = \sum_{n=1}^{N} (h_n - t_n)\mathbf{x}_n^T + \alpha \mathbf{w}^T$$

where $h_n = \sigma(\mathbf{w}^T \mathbf{x}_n)$

- Cannot solve analytically => solve numerically:
  - (stochastic) gradient descent [PRML 3.1.3], Newton Raphson iterative optimization [PRML 4.3.3], conjugate gradient, LBFGS.

# Softmax Regression = Logistic Regression for Multiclass Classification

- Multiclass classification:

  $T = \{C_1, C_2, ..., C_K\} = \{1, 2, ..., K\}$.

- Training set is $(\mathbf{x}_1, t_1), (\mathbf{x}_2, t_2), \dots (\mathbf{x}_n, t_n)$.

  $\mathbf{x} = [1, x_1, x_2, ..., x_M]$

  $t_1, t_2, \dots t_n \in \{1, 2, ..., K\}$

- One weight vector per class [PRML 4.3.4]:

$$p(C_k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}))}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}$$

# Softmax Regression (K ≥ 2)

- Inference:

$$C_* = \arg\max_{C_k} p(C_k \mid \mathbf{x})$$

$$= \arg\max_{C_k} \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\boxed{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}}$$

*$Z(\mathbf{x})$ a normalization constant*

$$= \arg\max_{C_k} \exp(\mathbf{w}_k^T \mathbf{x})$$

$$= \arg\max_{C_k} \mathbf{w}_k^T \mathbf{x}$$

- Training using:
  - Maximum Likelihood (ML)
  - Maximum A Posteriori (MAP) with a Gaussian prior on **w**.

# Softmax Regression

- The **negative log-likelihood** error function is:

$$E_D(\mathbf{w}) = -\frac{1}{N}\ln\prod_{n=1}^{N} p(t_n \mid \mathbf{x}_n)$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\ln\frac{\exp(\mathbf{w}_{t_n}^T\mathbf{x}_n)}{Z(\mathbf{x}_n)}$$

$$= -\boxed{\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln\frac{\exp(\mathbf{w}_k^T\mathbf{x}_n)}{Z(\mathbf{x}_n)}}$$

*convex in* **w**

where $\delta_t(x) = \begin{cases} 1 & x = t \\ 0 & x \neq t \end{cases}$ is the *Kronecker delta* function.

# Softmax Regression

- The ML solution is:

$$\mathbf{w}_{ML} = \arg\min_{\mathbf{w}} E_D(\mathbf{w})$$

- The **gradient** is (prove it):

$$\nabla_{\mathbf{w}_k} E_D(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n)}{Z(\mathbf{x}_n)}\right)\mathbf{x}_n$$

$$\nabla E_D(\mathbf{w}) = \left[\nabla^T_{\mathbf{w}_1} E_D(\mathbf{w}), \nabla^T_{\mathbf{w}_2} E_D(\mathbf{w}), \ldots, \nabla^T_{\mathbf{w}_K} E_D(\mathbf{w})\right]^T$$

# Regularized Softmax Regression

- The new **cost** function is:

$$E(\mathbf{w}) = E_D(\mathbf{w}) + E_{\mathbf{w}}(\mathbf{w})$$

$$= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln\frac{\exp(\mathbf{w}_k^T\mathbf{x}_n)}{Z(\mathbf{x}_n)} + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$$

- The new **gradient** is (prove it):

$$\nabla_{\mathbf{w}_k}E(\mathbf{w}) = -\frac{1}{N}\sum_{n=1}^{N}\big(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\big)\mathbf{x}_n^T + \alpha\mathbf{w}_k^T$$

# Softmax Regression

- ML solution is given by $\nabla E_D(\mathbf{w}) = 0$ .
  - Cannot solve analytically.
  - Solve numerically, by pluging [*cost*, *gradient*] = $[E_D(\mathbf{w}), \nabla E_D(\mathbf{w})]$ values into general convex solvers:
    - L-BFGS
    - Newton methods
    - conjugate gradient
    - (stochastic / minibatch) gradient-based methods.
      - gradient descent (with / without momentum).
      - AdaGrad, AdaDelta
      - RMSProp
      - ADAM, ...

# Implementation

- Need to compute [*cost*, ***gradient***]:
  - *cost* $= -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln p(C_k \mid \mathbf{x}_n) + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$
  - *gradient*$_k$ $= -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n^T + \alpha\mathbf{w}_k^T$

=> need to compute, for $k$ = 1, ..., K:

- *output* $p(C_k \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T\mathbf{x}_n))}{\sum_j \exp(\mathbf{w}_j^T\mathbf{x}_n)}$

Overflow when $\mathbf{w}_k{}^T\mathbf{x}_n$ are too large.

# Implementation: Preventing Overflows

- Subtract from each product $\mathbf{w}_k{}^{\mathrm{T}}\mathbf{x}_n$ the maximum product:

$$c = \max_{1 \le k \le K} \mathbf{w}_k^T \mathbf{x}_n$$

$$p(C_k \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n - c))}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x}_n - c)}$$

# Implementation: Gradient Checking

- Want to minimize $J(\theta)$, where $\theta$ is a scalar.

- Mathematical definition of derivative:

$$\frac{d}{d\theta} J(\theta) = \lim_{\varepsilon \to \infty} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

- Numerical approximation of derivative:

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \qquad \text{where } \varepsilon = 0.0001$$

# Implementation: Gradient Checking

- If $\boldsymbol{\theta}$ is a vector of parameters $\boldsymbol{\theta}_i$,
  - Compute numerical derivative with respect to each $\boldsymbol{\theta}_i$.
  - Aggregate all derivatives into numerical gradient $G_{\mathrm{num}}(\boldsymbol{\theta})$.

- Compare numerical gradient $G_{\mathrm{num}}(\boldsymbol{\theta})$ with implementation of gradient $G_{\mathrm{imp}}(\boldsymbol{\theta})$:

$$\frac{\left\| G_{num}(\boldsymbol{\theta}) - G_{imp}(\boldsymbol{\theta}) \right\|}{\left\| G_{num}(\boldsymbol{\theta}) + G_{imp}(\boldsymbol{\theta}) \right\|} \leq 10^{-6}$$