CS 6890: Deep Learning

Linear Regression

Logistic Regression

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

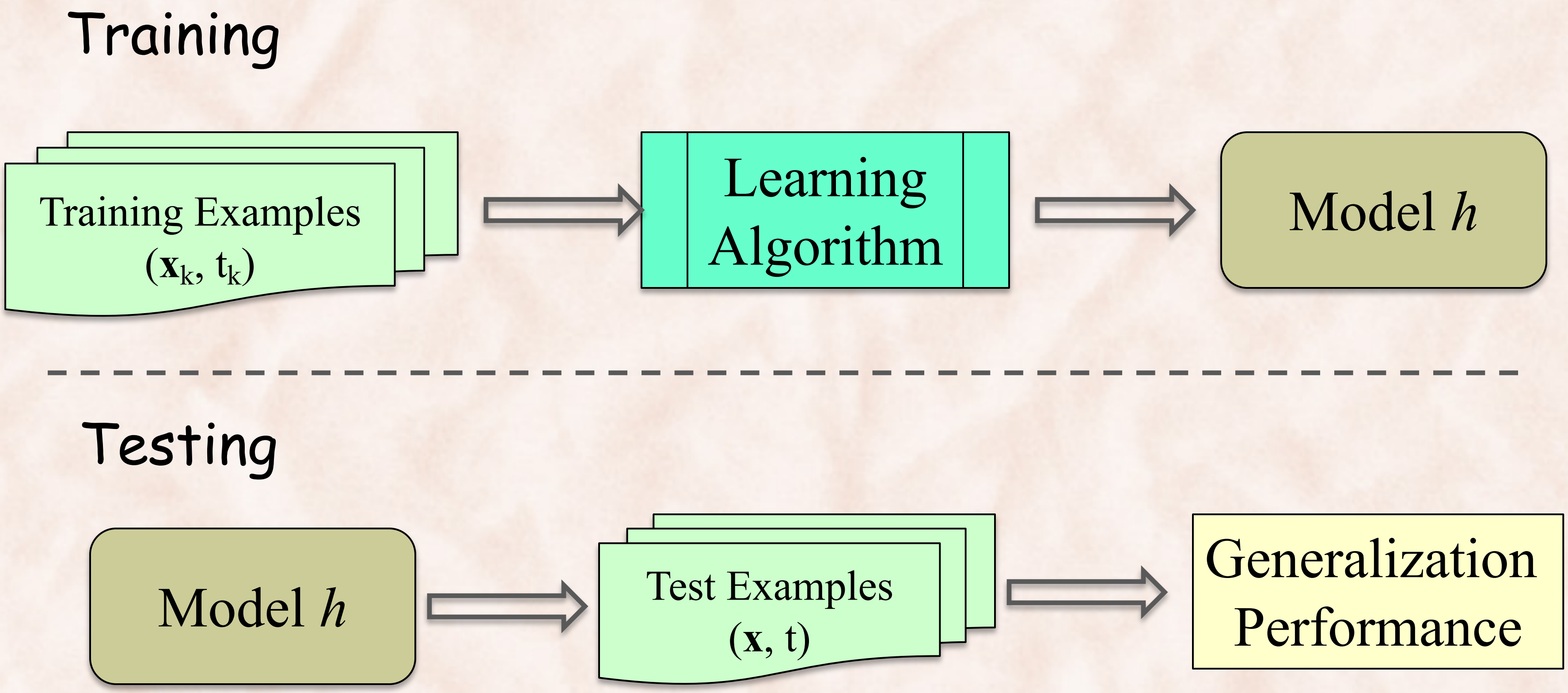
Supervised Learning

- **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



Parametric Approaches to Supervised Learning

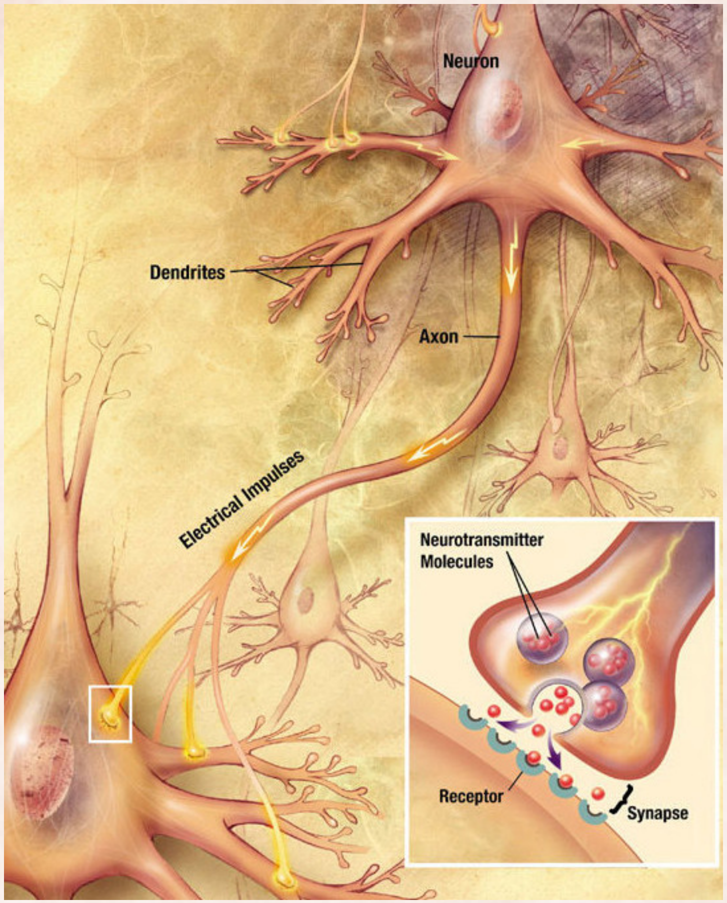
- **Task** = build a function $h(\mathbf{x})$ such that:
  - h matches t well on the training data:

    => 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.*

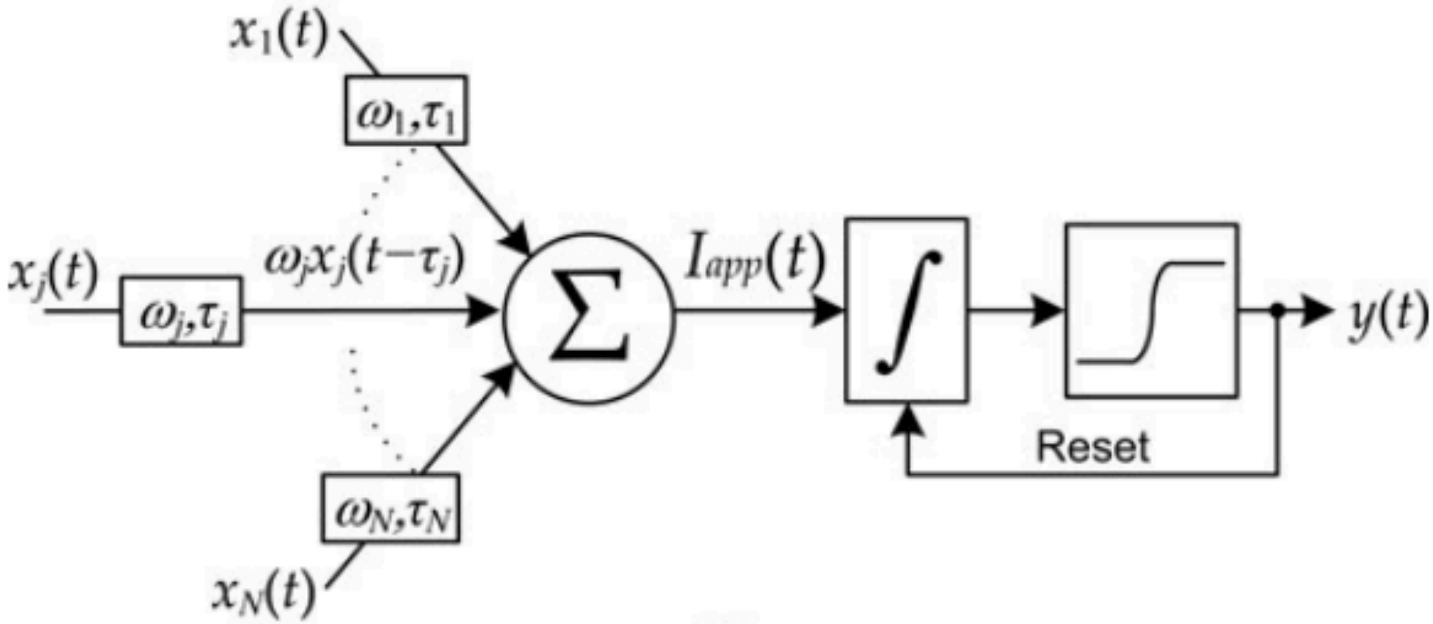
Neuron Models

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)

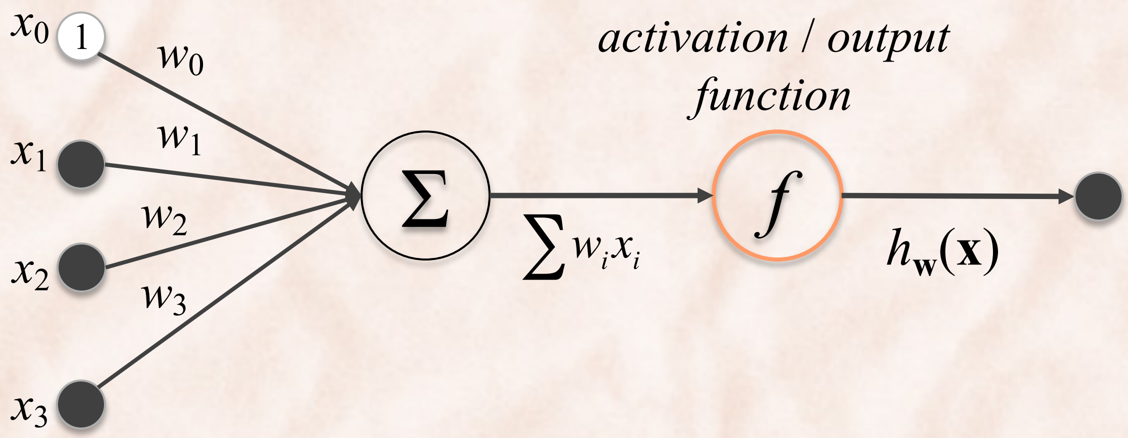
Fig. 2. (a) Illustration and (b) functional description of a leaky integrate-and-fire neuron. Weighted and delayed input signals are summed into the input current $I_{\mathrm{app}}(t)$, which travel to the soma and perturb the internal state variable, the voltage V. Since V is hysteric, the soma performs 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_{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

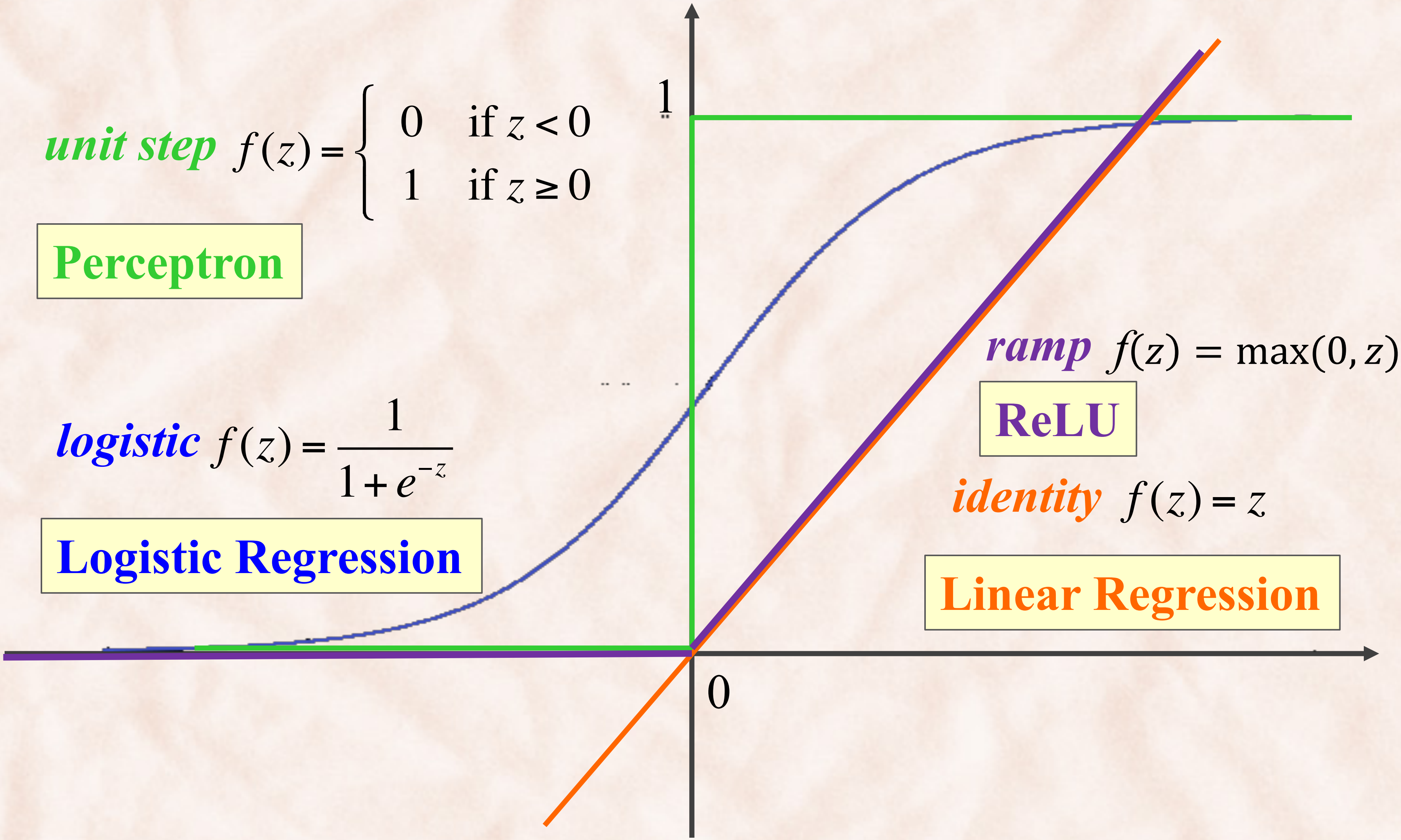
Year	Model Name	Reference
1907	Integrate and fire	[13]
1943	McCulloch and Pitts	[11]
1952	Hodgkin-Huxley	[12]
1958	Perceptron	[14]
1961	Fitzhugh-Nagumo	[15]
1965	Leaky integrate-and-fire	[16]
1981	Morris-Lecar	[17]
1986	Quadratic integrate-and-fire	[18]
1989	Hindmarsh-Rose	[19]
1998	Time-varying integrate-and-fire 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

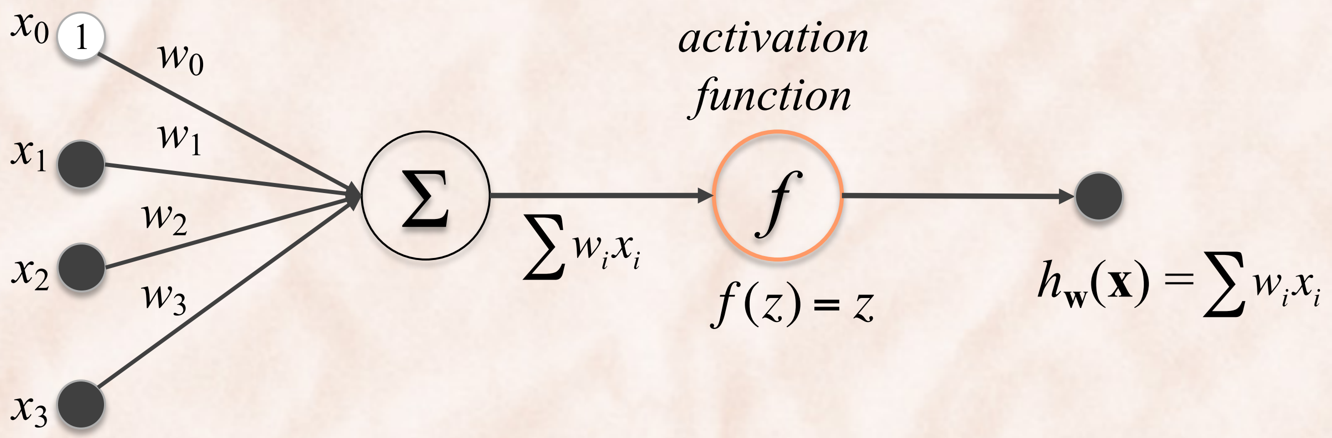


- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights w_i correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.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

Activation Functions



Linear Regression



- Polynomial curve fitting is Linear Regression:

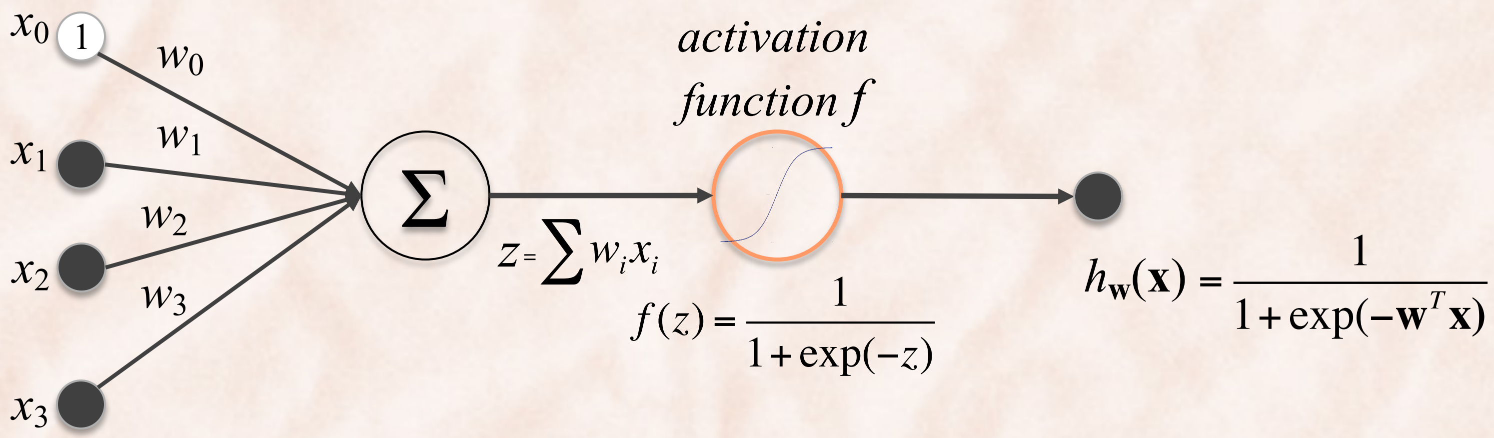
  $\mathbf{x} = \varphi(x) = [1, x, x^2, \ldots, x^M]^T$ $\qquad h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$

- What **error/cost function** to minimize?

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

Use *gradient descent*

Logistic Regression for Binary Classification



- Used for binary classification:
  - Labels T = $\{C_1, C_2\} = \{1, 0\}$
  - Output C_1 iff $h(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) > 0.5$
- Training set is $(\mathbf{x}_1,t_1), (\mathbf{x}_2,t_2), \ldots (\mathbf{x}_n,t_n)$.

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

Logistic Regression for Binary Classification

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

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

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

- What **error/cost function** to minimize?
  - The **negative log-likelihood**.

Use *gradient descent*

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):

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

Implementation: Vectorization of LR

- **Version 1**: Compute gradient component-wise.

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

  - Assume example $\mathbf{x}_n$ is stored in column X[:,n] in data matrix X.

```
grad = np.zeros(K)
for n in range(N):
    h = sigmoid(w.dot(X[:,n])
    temp = h − t[n]
    for k in range(K):
      grad[k] = grad[k] + temp * X[k,n]
```

```
def sigmoid(x):
   return 1 / (1 + np.exp(−x))
```

Implementation: Vectorization of LR

- **Version 2**: Compute gradient, partially vectorized.

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

```
grad = np.zeros(K)
for n in range(N):
    grad = grad + (sigmoid(w.dot(X[n])) − t[n]) * X[n]
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(−x))
```

Implementation: Vectorization of LR

- **Version 3**: Compute gradient, vectorized.

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

grad = X.dot(sigmoid(**w**.dot(X)) − **t**)

```
def sigmoid(x):
    return 1 / (1 + np.exp(−x))
```

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), … ($\mathbf{x}_n$,t_n).

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

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

- 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(**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 by minimizing the **negative log-likelihood**.

Use *gradient descent*

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 $\mathbf{w}$

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):

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

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

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

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 $k = 1, \ldots, K$:

  - *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:

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

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

Vectorization of Softmax

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

=> compute ground truth matrix G such that G[k,n] = $\delta_k(t_n)$

```
from scipy.sparse import coo_matrix
groundTruth = coo_matrix((np.ones(N, dtype = np.uint8),
                          (labels, np.arange(N)))).toarray()
```

Vectorization of Softmax

- Compute $cost = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}\delta_k(t_n)\ln p(C_k \mid \mathbf{x}_n) + \frac{\alpha}{2}\sum_{k=1}^{K}\mathbf{w}_k^T\mathbf{w}_k$
  - Compute matrix of $\mathbf{w}_k^T\mathbf{x}_n$.
  - Compute matrix of $\mathbf{w}_k^T\mathbf{x}_n - c_n$.
  - Compute matrix of $\exp(\mathbf{w}_k^T\mathbf{x}_n - c_n)$.
  - Compute matrix of $\ln p(C_k|\mathbf{x}_n)$.
  - Compute log-likelihood.

Vectorization of Softmax

- Compute $\mathbf{grad}_k = -\frac{1}{N}\sum_{n=1}^{N}\left(\delta_k(t_n) - p(C_k \mid \mathbf{x}_n)\right)\mathbf{x}_n^T + \alpha\mathbf{w}_k^T$
  - **Gradient** = [$\mathbf{grad}_1$ | $\mathbf{grad}_2$ | … | $\mathbf{grad}_K$]

– Compute matrix of $p(C_k|\mathbf{x}_n)$.

– Compute matrix of gradient of data term.

– Compute matrix of gradient of regularization term.

Vectorization of Softmax

- Useful Numpy functions:
  - np.dot()
  - np.amax()
  - np.argmax()
  - np.exp()
  - np.sum()
  - np.log()
  - np.mean()