CS 4900/5900: Machine Learning

Logistic Regression

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Supervised Learning

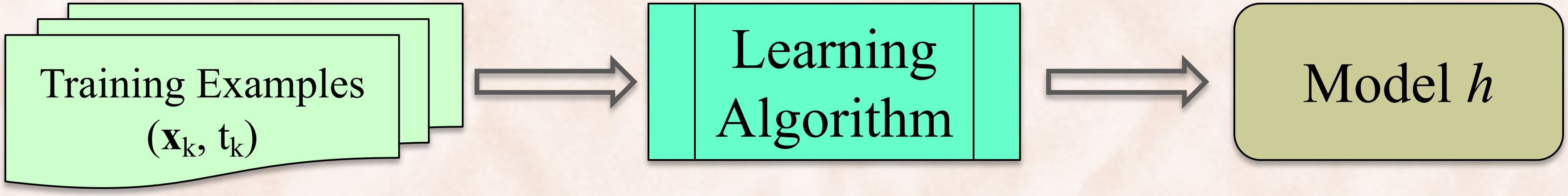
- **Task** = learn an (unkown) function $t : X \rightarrow T$ that maps input instances $\mathbf{x} \in X$ to output targets $t(\mathbf{x}) \in T$:
  - **Classification**:
    - The output $t(\mathbf{x}) \in T$ is one of a finite set of discrete categories.
  - **Regression**:
    - The output $t(\mathbf{x})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

Training

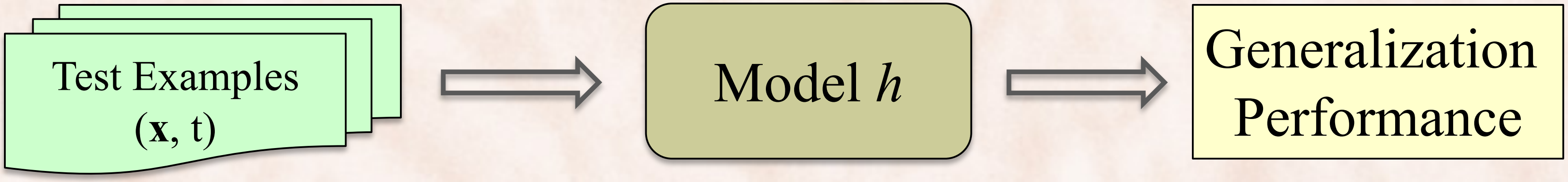

Training Examples

$(\mathbf{x}_k, t_k)$

Learning Algorithm

Model h

Testing


Test Examples

$(\mathbf{x}, t)$

Model h

Generalization Performance

Parametric Approaches to Supervised Learning

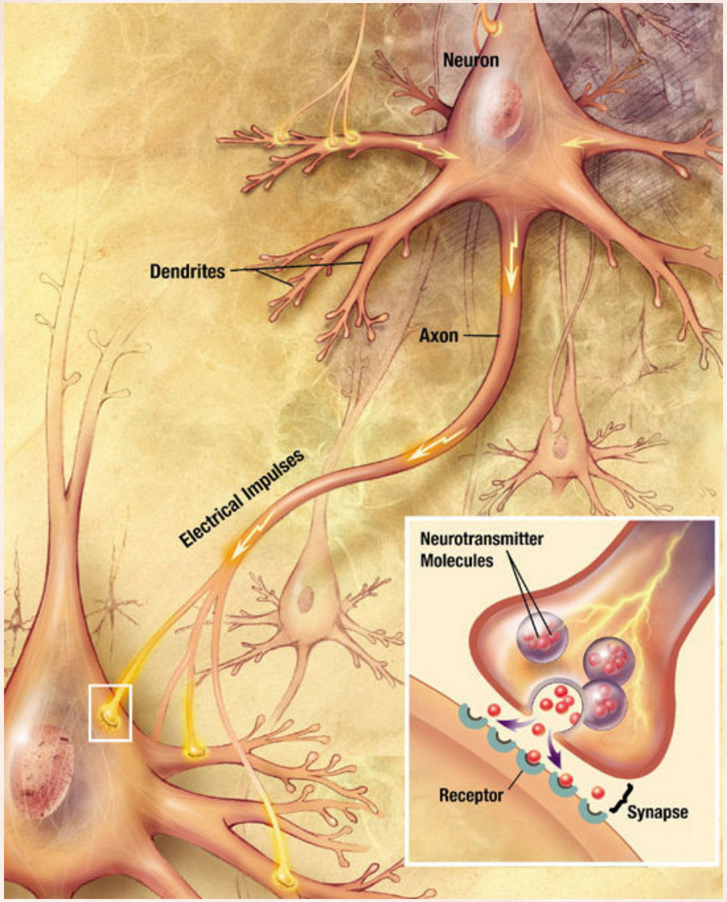
- **Task** = build a function $h(\mathbf{x})$ such that:
  - h matches t well on the training data:

    => 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 **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:

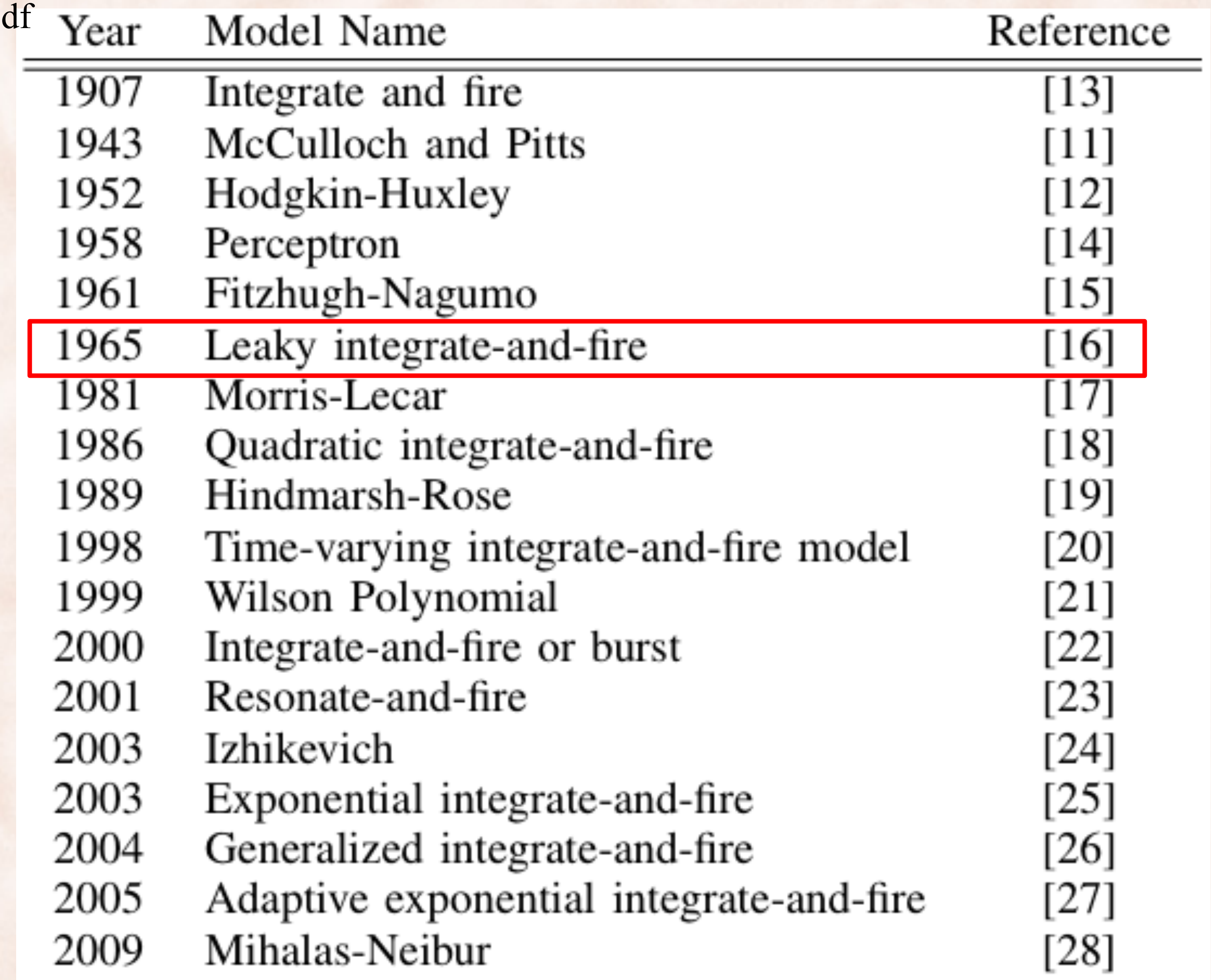
- *carries nerve signals to other neurons.*

Synapses are molecular structures between axon terminals and other neurons:

- *where the communication takes place.*

Neuron Models

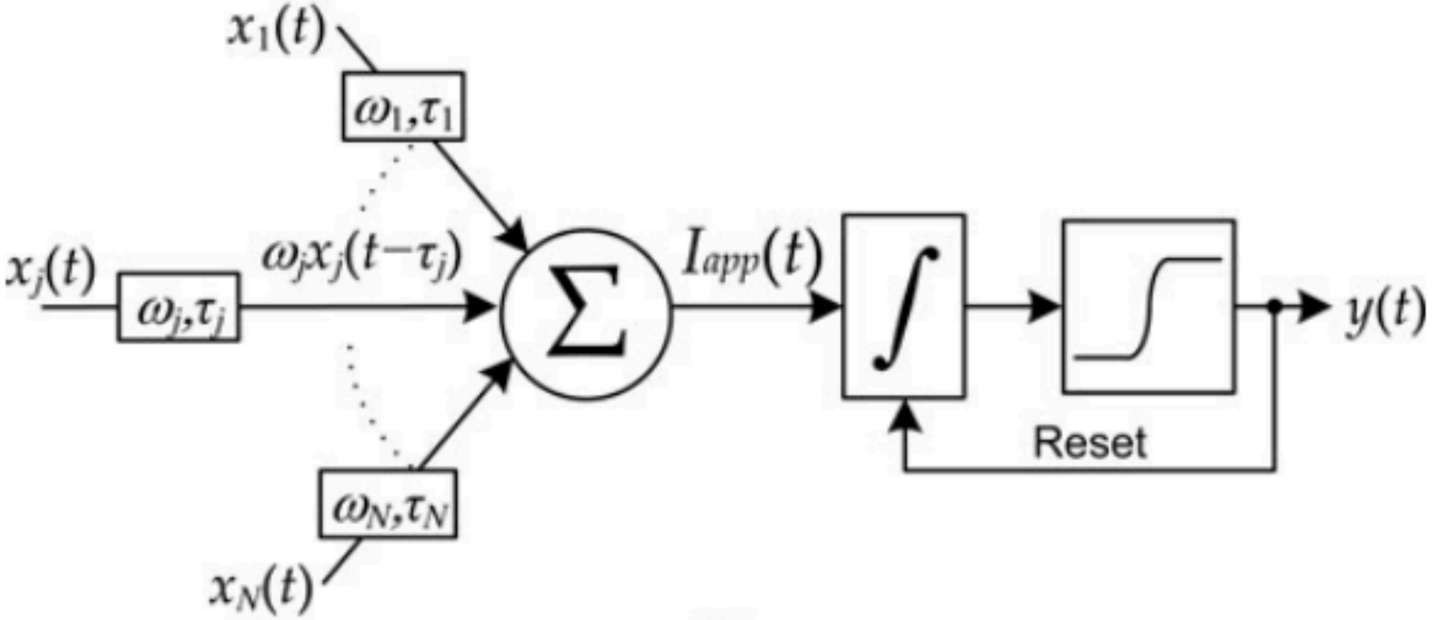
https://www.research.ibm.com/software/IBMResearch/multimedia/IJCNN2013.neuron-model.pdf



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

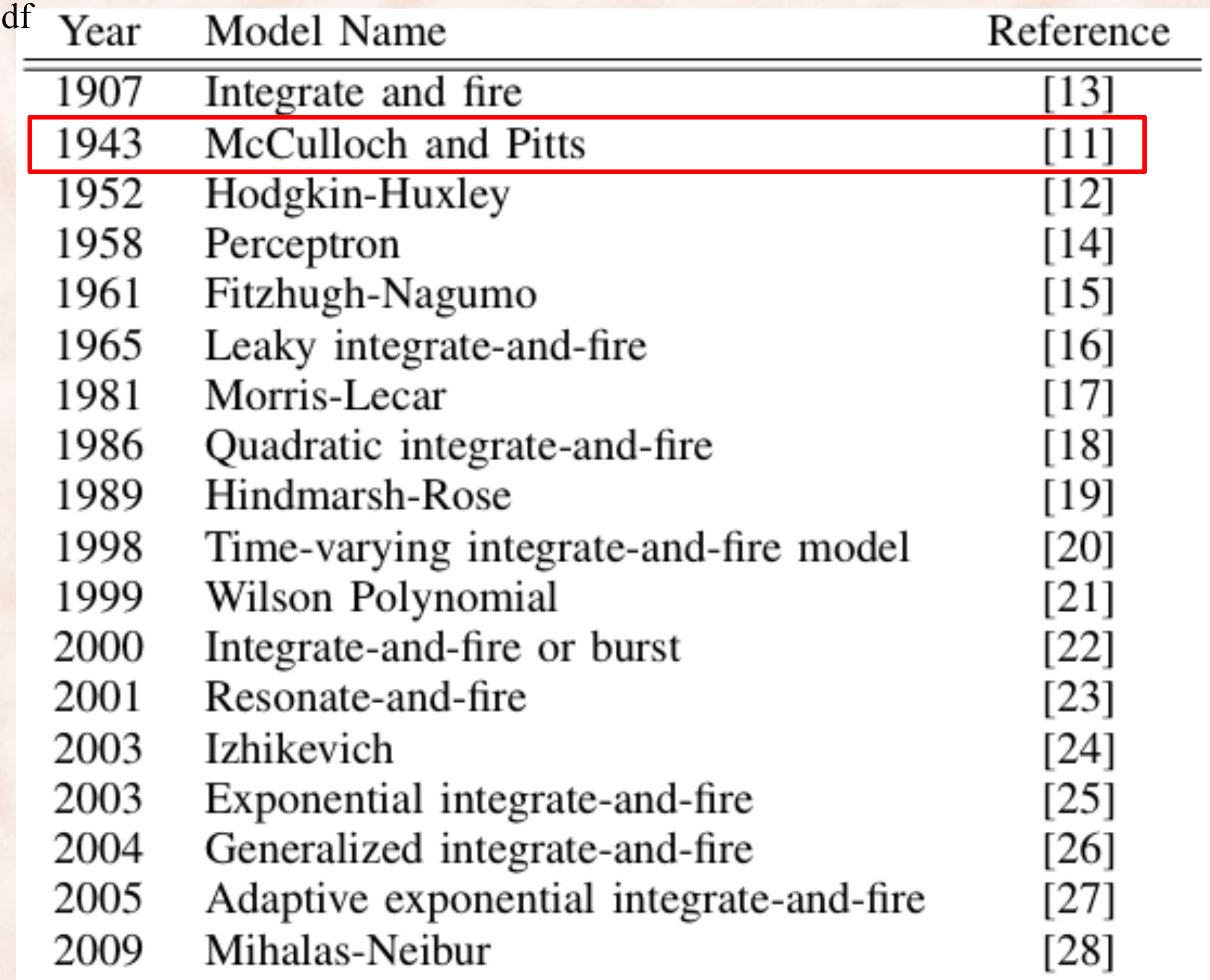


(b)

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_{reset}. The resulting spike is sent to other neurons in the network.

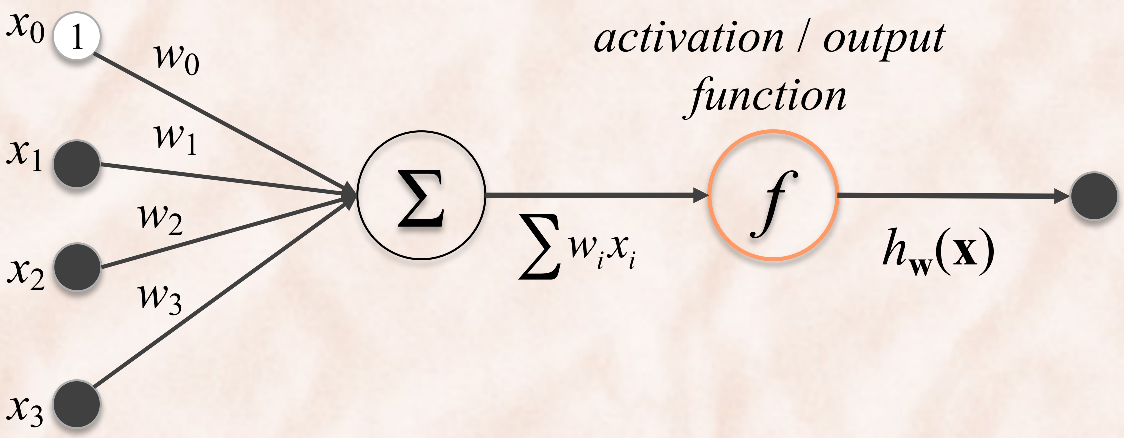
Neuron Models

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



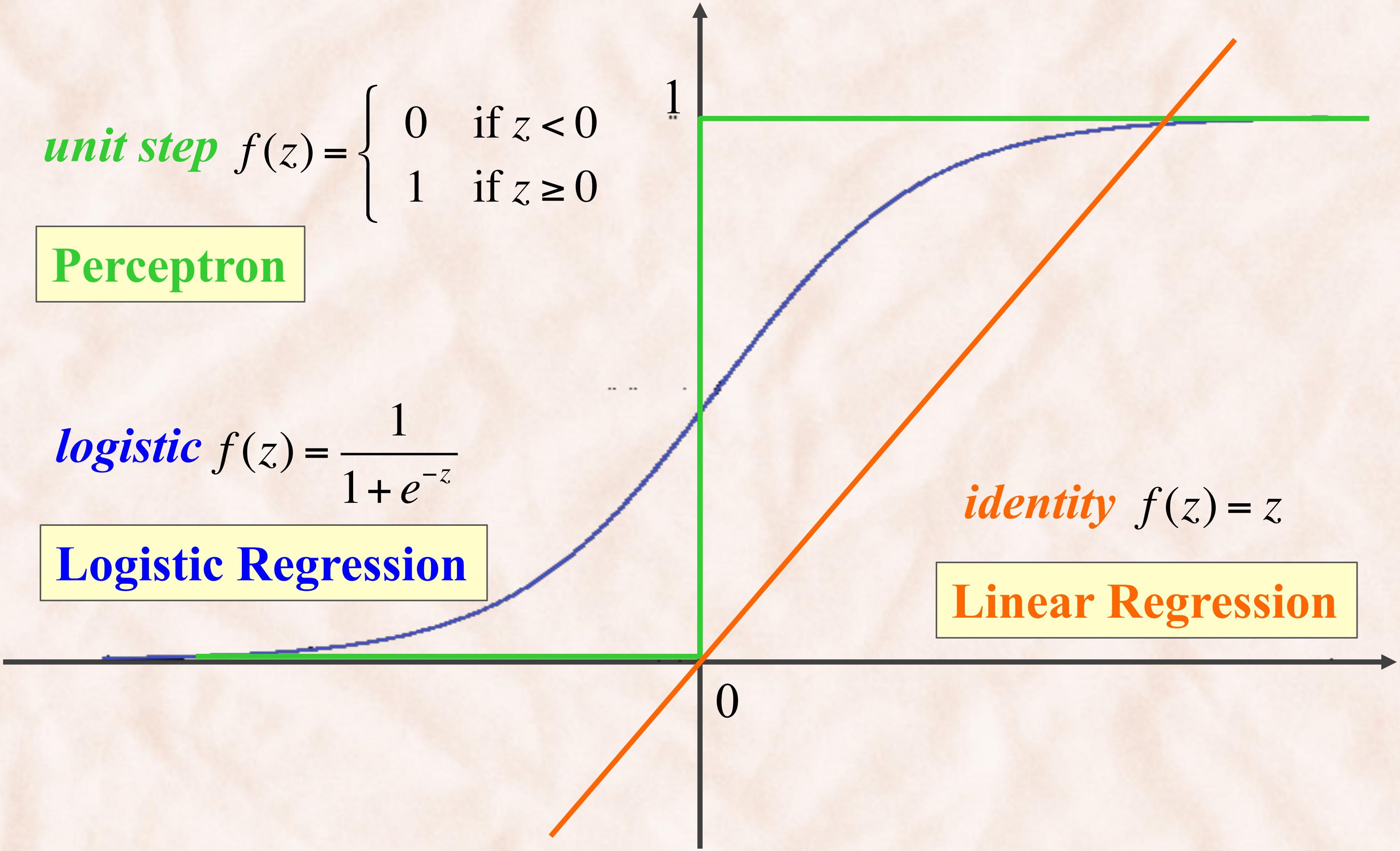
Year	Model Name	Reference
1907	Integrate and fire	[13]
1943	McCulloch and Pitts	[11]
1952	Hodgkin-Huxley	[12]
1958	Perceptron	[14]
1961	Fitzhugh-Nagumo	[15]
1965	Leaky integrate-and-fire	[16]
1981	Morris-Lecar	[17]
1986	Quadratic integrate-and-fire	[18]
1989	Hindmarsh-Rose	[19]
1998	Time-varying integrate-and-fire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

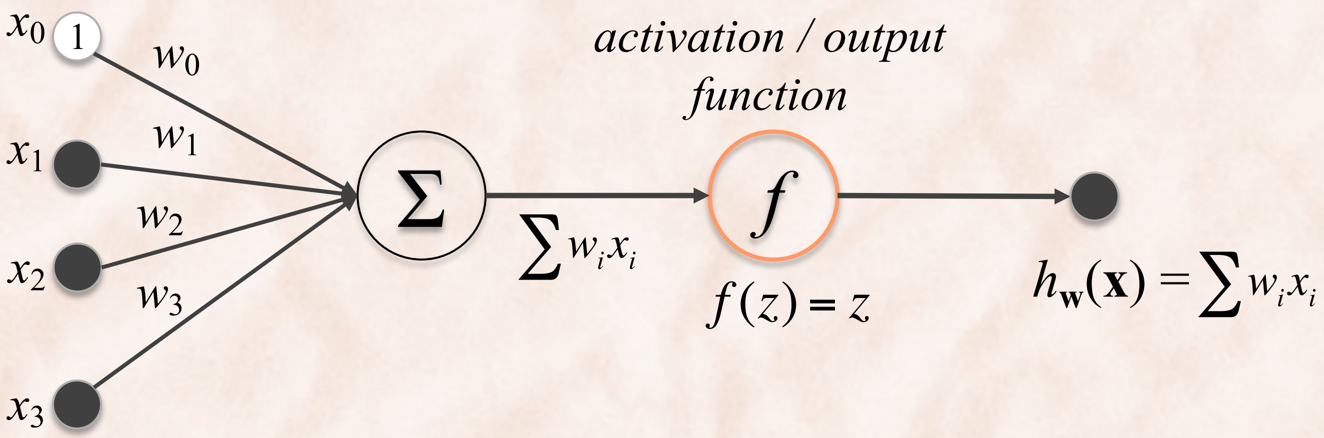
logistic $f(z) = \dfrac{1}{1 + e^{-z}}$

Logistic Regression

identity $f(z) = z$

Linear Regression

Linear Regression

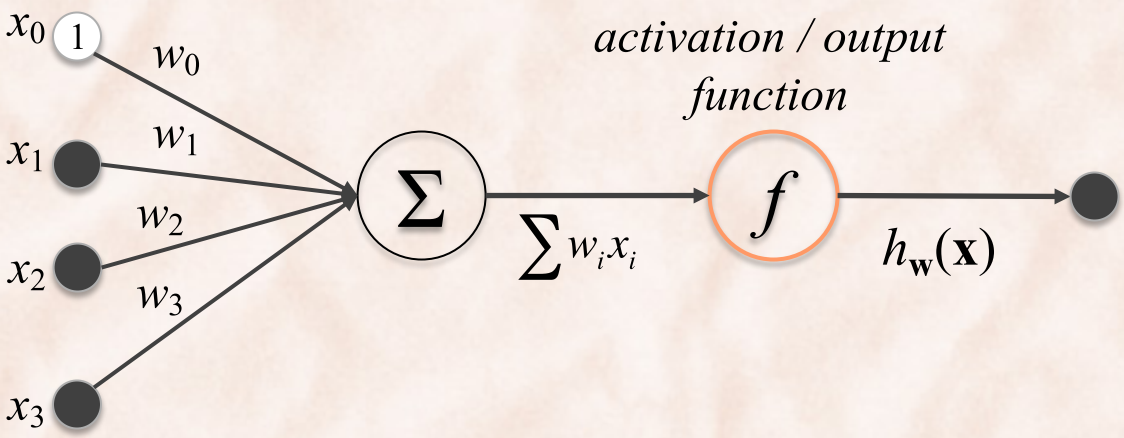


- Polynomial curve fitting is Linear Regression:

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

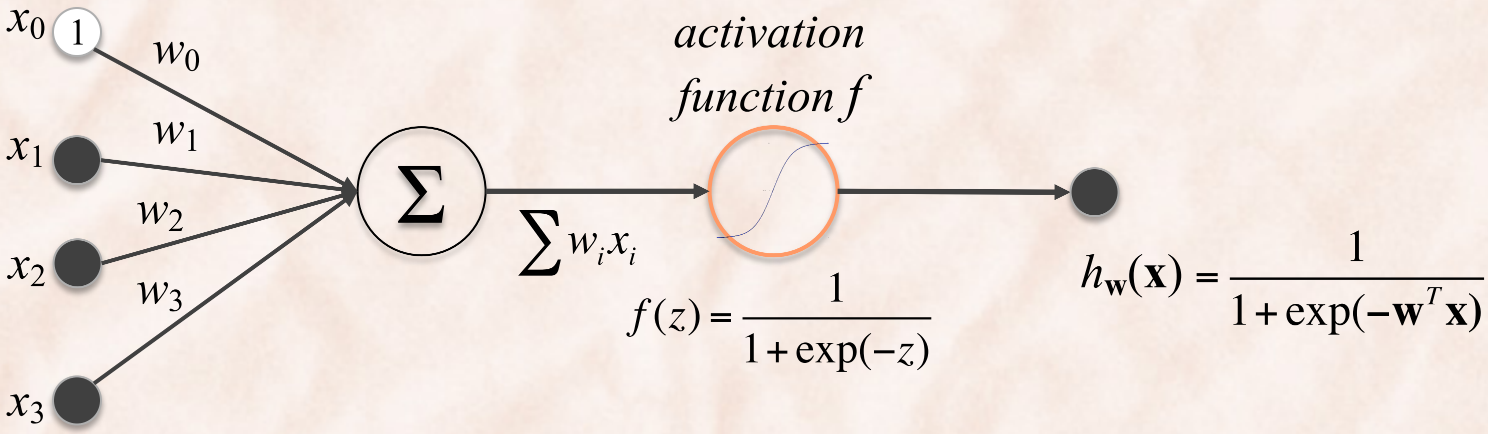
  $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$

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 a monotonic **activation / output function**.

Logistic Regression



$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x})}$$

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

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

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

- 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).

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 **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:

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

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

- Bayes' Theorem:

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

- MAP solution:

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

Regularized Logistic Regression

- MAP solution:

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

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

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

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

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

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

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

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 $\mathbf{w}$

- 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:

$$
\begin{aligned}
C_* &= \arg\max_{C_k} p(C_k \mid \mathbf{x}) \\
&= \arg\max_{C_k} \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\boxed{\sum_j \exp(\mathbf{w}_j^T \mathbf{x})}} \\
&= \arg\max_{C_k} \exp(\mathbf{w}_k^T \mathbf{x}) \\
&= \arg\max_{C_k} \mathbf{w}_k^T \mathbf{x}
\end{aligned}
$$

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

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

Softmax Regression

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

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

convex in **w**

- The Maximum Likelihood solution is:

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

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

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

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

Regularized Softmax Regression

- The new **cost** function is:

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

$$= -\frac{1}{N}\sum_{n=1}^{N} \ln \frac{\exp\left(\mathbf{w}_{t_n}^T \mathbf{x}_n\right)}{Z(\mathbf{x}_n)} + \frac{\alpha}{2}\|\mathbf{w}\|^2$$

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

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

Softmax Regression

- ML solution is given by $\nabla E_D(\mathbf{w}) = 0$.
  - Cannot solve analytically.
  - Solve numerically, by pluging [*cost*, *gradient*] = $[E(\mathbf{w}), \nabla E(\mathbf{w})]$ 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 $k = 1, \ldots, K$:

- *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 $\mathbf{w}_k{}^{\mathrm{T}}\mathbf{x}_n$ are too large.

Implementation: Preventing Overflows

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

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

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

Implementation: Gradient Checking

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

- Mathematical definition of derivative:

$$\frac{d}{d\theta}J(\theta) = \lim_{\varepsilon \to 0} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

- 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 $\boldsymbol{\theta}_i$.
    - Create a vector $\mathbf{v}$ that is ε in position i and 0 everywhere else:
      - *How do you do this without a for loop in NumPy?*
    - Compute $G_{\text{num}}(\boldsymbol{\theta}_i) = (J(\boldsymbol{\theta} + \mathbf{v}) - J(\boldsymbol{\theta} - \mathbf{v})) / 2\varepsilon$
  - Aggregate all derivatives into numerical gradient $G_{\text{num}}(\boldsymbol{\theta})$.

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

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

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))
```

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** = [**grad**$_1$ | **grad**$_2$ | … | **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()`

import scipy

- scipy.sparse.coo_matrix()

```
groundTruth = coo_matrix((np.ones(numCases, dtype = np.uint8),
                         (labels, np.arange(numCases)))).toarray()
```

- scipy.optimize:
  - scipy.optimize.fmin_l_bfgs_b()

```
theta, _, _ = fmin_l_bfgs_b(softmaxCost, theta,
                            args = (numClasses, inputSize, decay, images, labels),
                            maxiter = 100, disp = 1)
```

  - scipy.optimize.fmin_cg()
  - scipy.minimize

https://docs.scipy.org/doc/scipy-0.10.1/reference/tutorial/optimize.html

Multiclass Logistic Regression (K ≥ 2)

1) Train one weight vector per class [PRML Chapter 4.3.4]:

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

2) More general approach:

$$p(C_k \mid \mathbf{x}) = \frac{\exp(\mathbf{w}^T \varphi(\mathbf{x}, C_k))}{\sum_j \exp(\mathbf{w}^T \varphi(\mathbf{x}, C_j))}$$

– Inference:

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

Logistic Regression (K ≥ 2)

2) Inference in more general approach:

$$
\begin{aligned}
C_* &= \arg\max_{C_k} p(C_k \mid \mathbf{x}) \\
&= \arg\max_{C_k} \frac{\exp(\mathbf{w}^T \varphi(\mathbf{x}, C_k))}{\boxed{\sum_j \exp(\mathbf{w}^T \varphi(\mathbf{x}, C_j))}} \\
&= \arg\max_{C_k} \exp(\mathbf{w}^T \varphi(\mathbf{x}, C_k)) \\
&= \arg\max_{C_k} \mathbf{w}^T \varphi(\mathbf{x}, C_k)
\end{aligned}
$$

$Z(\mathbf{x})$ the partition function.

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

Logistic Regression (K ≥ 2) with ML

- The negative log-likelihood error function is:

$$E_D(\mathbf{w}) = -\ln\prod_{n=1}^{N} p(t_n \mid \mathbf{x}_n) = \boxed{-\sum_{n=1}^{N} \ln\frac{\exp(\mathbf{w}^T\varphi(\mathbf{x}_n, t_n))}{Z(\mathbf{x}_n)}}$$

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

$\boxed{\textit{convex in } \mathbf{w}}$

- The gradient is (prove it):

$$\nabla E_D(\mathbf{w}) = \left[\frac{\partial E_D(\mathbf{w})}{\partial w_0}, \frac{\partial E_D(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial E_D(\mathbf{w})}{\partial w_M}\right]$$

$$\frac{\partial E_D(\mathbf{w})}{\partial w_i} = -\sum_{n=1}^{N}\varphi_i(\mathbf{x}_n, t_n) + \sum_{n=1}^{N}\sum_{k=1}^{K} p(C_k \mid \mathbf{x}_n)\varphi_i(\mathbf{x}_n, C_k)$$

Logistic Regression (K ≥ 2) with ML

- Set $\nabla E_D(\mathbf{w}) = 0 \Rightarrow$ ML solution satisfies:

$$\sum_{n=1}^{N} \varphi_i(\mathbf{x}_n, t_n) = \sum_{n=1}^{N} \sum_{k=1}^{K} p(C_k \mid \mathbf{x}_n) \varphi_i(\mathbf{x}_n, C_k)$$

$\Rightarrow$ for every feature φ_i, the *observed value* on D should be the same as the *expected value* on D!

- Solve numerically:
  - Stochastic gradient descent [chapter 3.1.3].
  - Newton Raphson iterative optimization (large Hessian!).
  - Limited memory Newton methods (e.g. L-BFGS).

The Maximum Entropy Principle

- Principle of Insufficient Reason
- Principle of Indifference
  - can be traced back to Pierre Laplace and Jacob Bernoulli.

➢ A. L. Berger, S. A. Della Pietra, and V. J. Della Pietra. 1996. A maximum entropy approach to natural language processing. Computational Linguistics, 22(1).

  - “*model all that is known and assume nothing about that which is unknown*”.
  - “*given a collection of facts, choose a model consistent with all the facts, but otherwise as uniform as possible*”.

Maximum Likelihood ⇔ Maximum Entropy

1) Maximize conditional likelihood:

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

$$p(\mathbf{t} \mid \mathbf{w}) = \prod_{n=1}^{N} p_{\mathbf{w}}(t_n \mid \mathbf{x}_n) = \prod_{n=1}^{N} \frac{\exp(\mathbf{w}^T \varphi(\mathbf{x}_n, t_n))}{Z(\mathbf{x}_n)}$$

2) Maximize conditional entropy:

$$p_{ME} = \arg\max_{p} \sum_{n=1}^{N} \sum_{k=1}^{K} - p(C_k \mid \mathbf{x}_n) \log p(C_k \mid \mathbf{x}_n)$$

subject to:

$$\sum_{n=1}^{N} \varphi(\mathbf{x}_n, t_n) = \sum_{n=1}^{N} \sum_{k=1}^{K} p(C_k \mid \mathbf{x}_n) \varphi(\mathbf{x}_n, C_k)$$

$$p_{ME}(t_n \mid \mathbf{x}_n) = p_{\mathbf{w}_{ML}}(t_n \mid \mathbf{x}_n) = \frac{\exp(\mathbf{w}_{ML}^T \varphi(\mathbf{x}_n, t_n))}{Z(\mathbf{x}_n)}$$