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 (unknown) function \( t : X \rightarrow T \) that maps input instances \( x \in X \) to output targets \( t(x) \in T \):
  - **Classification**:
    - The output \( t(x) \in T \) is one of a finite set of discrete categories.
  - **Regression**:
    - The output \( t(x) \in T \) is continuous, or has a continuous component.

- Target function \( t(x) \) is known (only) through (noisy) set of training examples:
  \[ (x_1,t_1), (x_2,t_2), \ldots (x_n,t_n) \]
Supervised Learning

Training

Training Examples \((x_k, t_k)\) → Learning Algorithm → Model \(h\)

Testing

Model \(h\) → Test Examples \((x, t)\) → Generalization Performance
Parametric Approaches to Supervised Learning

- **Task** = build a function \( h(x) \) 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 \( w \):
  - \( h(x) = h_w(x) = h(w,x) \)
  - 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


<table>
<thead>
<tr>
<th>Year</th>
<th>Model Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1907</td>
<td>Integrate and fire</td>
<td>[13]</td>
</tr>
<tr>
<td>1943</td>
<td>McCulloch and Pitts</td>
<td>[11]</td>
</tr>
<tr>
<td>1952</td>
<td>Hodgkin-Huxley</td>
<td>[12]</td>
</tr>
<tr>
<td>1958</td>
<td>Perceptron</td>
<td>[14]</td>
</tr>
<tr>
<td>1961</td>
<td>Fitzhugh-Nagumo</td>
<td>[15]</td>
</tr>
<tr>
<td>1965</td>
<td>Leaky integrate-and-fire</td>
<td>[16]</td>
</tr>
<tr>
<td>1981</td>
<td>Morris-Lecar</td>
<td>[17]</td>
</tr>
<tr>
<td>1986</td>
<td>Quadratic integrate-and-fire</td>
<td>[18]</td>
</tr>
<tr>
<td>1989</td>
<td>Hindmarsh-Rose</td>
<td>[19]</td>
</tr>
<tr>
<td>1999</td>
<td>Wilson Polynomial</td>
<td>[21]</td>
</tr>
<tr>
<td>2000</td>
<td>Integrate-and-fire or burst</td>
<td>[22]</td>
</tr>
<tr>
<td>2001</td>
<td>Resonate-and-fire</td>
<td>[23]</td>
</tr>
<tr>
<td>2003</td>
<td>Izhikevich</td>
<td>[24]</td>
</tr>
<tr>
<td>2003</td>
<td>Exponential integrate-and-fire</td>
<td>[25]</td>
</tr>
<tr>
<td>2004</td>
<td>Generalized integrate-and-fire</td>
<td>[26]</td>
</tr>
<tr>
<td>2005</td>
<td>Adaptive exponential integrate-and-fire</td>
<td>[27]</td>
</tr>
<tr>
<td>2009</td>
<td>Mihalas-Neibur</td>
<td>[28]</td>
</tr>
</tbody>
</table>
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 $I_{app}(t)$, 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


<table>
<thead>
<tr>
<th>Year</th>
<th>Model Name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1907</td>
<td>Integrate and fire</td>
<td>[13]</td>
</tr>
<tr>
<td>1943</td>
<td>McCulloch and Pitts</td>
<td>[11]</td>
</tr>
<tr>
<td>1952</td>
<td>Hodgkin-Huxley</td>
<td>[12]</td>
</tr>
<tr>
<td>1958</td>
<td>Perceptron</td>
<td>[14]</td>
</tr>
<tr>
<td>1961</td>
<td>Fitzhugh-Nagumo</td>
<td>[15]</td>
</tr>
<tr>
<td>1965</td>
<td>Leaky integrate-and-fire</td>
<td>[16]</td>
</tr>
<tr>
<td>1981</td>
<td>Morris-Lecar</td>
<td>[17]</td>
</tr>
<tr>
<td>1986</td>
<td>Quadratic integrate-and-fire</td>
<td>[18]</td>
</tr>
<tr>
<td>1989</td>
<td>Hindmarsh-Rose</td>
<td>[19]</td>
</tr>
<tr>
<td>1999</td>
<td>Wilson Polynomial</td>
<td>[21]</td>
</tr>
<tr>
<td>2000</td>
<td>Integrate-and-fire or burst</td>
<td>[22]</td>
</tr>
<tr>
<td>2001</td>
<td>Resonate-and-fire</td>
<td>[23]</td>
</tr>
<tr>
<td>2003</td>
<td>Izhikevich</td>
<td>[24]</td>
</tr>
<tr>
<td>2003</td>
<td>Exponential integrate-and-fire</td>
<td>[25]</td>
</tr>
<tr>
<td>2004</td>
<td>Generalized integrate-and-fire</td>
<td>[26]</td>
</tr>
<tr>
<td>2005</td>
<td>Adaptive exponential integrate-and-fire</td>
<td>[27]</td>
</tr>
<tr>
<td>2009</td>
<td>Mihalas-Neibur</td>
<td>[28]</td>
</tr>
</tbody>
</table>
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** \( f(z) = \begin{cases} 
0 & \text{if } z < 0 \\
1 & \text{if } z \geq 0 
\end{cases} \)

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

**Perceptron**

**ramp** \( f(z) = \max(0, z) \)

**ReLU**

**identity** \( f(z) = z \)

**Logistic Regression**

**Linear Regression**
Linear Regression

- Polynomial curve fitting is Linear Regression:
  \[ x = \phi(x) = [1, x, x^2, \ldots, x^M]^T \quad h(x) = w^T x \]

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

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

**Use gradient descent**
Logistic Regression for Binary Classification

- Used for binary classification:
  - Labels $T = \{C_1, C_2\} = \{1, 0\}$
  - Output $C_1$ iff $h(x) = \sigma(w^T x) > 0.5$

- Training set is $(x_1, t_1), (x_2, t_2), \ldots (x_n, t_n)$.
  - $x = [1, x_1, x_2, \ldots, x_k]^T$
Logistic Regression for Binary Classification

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

\[
p(C_1 | x) = \sigma(w^T x) = \frac{1}{1 + \exp(-w^T x)}
\]

\[
p(C_2 | x) = 1 - \sigma(w^T x) = \frac{\exp(-w^T x)}{1 + \exp(-w^T x)}
\]

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

  Use gradient descent
Maximum Likelihood

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

Let $h_n = p(C_1 \mid x_n) \iff h_n = p(t_n = 1 \mid x_n) = \sigma(w^T x_n)$

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

- The **likelihood function** is $p(t \mid w) = \prod_{n=1}^{N} h_n^{t_n} (1-h_n)^{(1-t_n)}$

- The **negative log-likelihood** (cross entropy) error function:

$$
E(w) = -\ln p(t \mid x) = -\sum_{n=1}^{N} \{ t_n \ln h_n + (1-t_n) \ln(1-h_n) \}$$
Maximum Likelihood Learning for Logistic Regression

- The **ML** solution is:
  \[ w_{ML} = \arg \max_w p(t \mid w) = \arg \min_w E(w) \]

- **ML** solution is given by \( \nabla E(w) = 0 \).
  - Cannot solve analytically \( \Rightarrow \) solve numerically with gradient based methods: (stochastic) gradient descent, conjugate gradient, L-BFGS, etc.
  - Gradient is (prove it):
    \[ \nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T \]
Implementation: Vectorization of LR

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

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T
\]
– Assume example \( x_n \) is stored in column \( X[:,n] \) in data matrix \( X \).

```python
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]
```

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Implementation: Vectorization of LR

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

\[
\nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T
\]

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

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Implementation: Vectorization of LR

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

\[ \nabla E(w) = \sum_{n=1}^{N} (h_n - t_n)x_n^T \]

\[ \text{grad} = X.\text{dot}(\text{sigmoid}(w.\text{dot}(X)) - t) \]

```python
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```
Softmax Regression = Logistic Regression for Multiclass Classification

• Multiclass classification:
  \( T = \{C_1, C_2, \ldots, C_K\} = \{1, 2, \ldots, K\} \).

• Training set is \((x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)\).
  \[
  x = [1, x_1, x_2, \ldots, x_M]
  \]
  \[
  t_1, t_2, \ldots, t_n \in \{1, 2, \ldots, K\}
  \]

• One weight vector per class [PRML 4.3.4]:
  \[
  p(C_k | x) = \frac{\exp(w_k^T x)}{\sum_j \exp(w_j^T x)}
  \]
Softmax Regression ($K \geq 2$)

- **Inference:**

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

  $$= \arg \max_{C_k} \exp(w_k^T x)$$

  $$= \arg \max_{C_k} \exp(w_j^T x)$$

  $$= \arg \max_{C_k} w_k^T x$$

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

  *Use gradient descent*
Softmax Regression

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

\[
E_D(w) = -\frac{1}{N} \ln \prod_{n=1}^{N} p(t_n | x_n) \\
= -\frac{1}{N} \sum_{n=1}^{N} \ln \frac{\exp(w^T_{t_n} x_n)}{Z(x_n)} \\
= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln \frac{\exp(w_k^T x_n)}{Z(x_n)}
\]

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

*convex in \( w \)*
Softmax Regression

- The **ML** solution is:
  \[
  \mathbf{w}_{ML} = \arg \min_{\mathbf{w}} E_D(\mathbf{w})
  \]

- The **gradient** is (prove it):
  \[
  \nabla_{\mathbf{w}_k} E_D(\mathbf{w}) = -\frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - p(C_k | \mathbf{x}_n) \right) \mathbf{x}_n
  \]
  \[
  = -\frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n)}{Z(\mathbf{x}_n)} \right) \mathbf{x}_n
  \]
  \[
  \nabla E_D(\mathbf{w}) = \left[ \nabla^T_{\mathbf{w}_1} E_D(\mathbf{w}), \nabla^T_{\mathbf{w}_2} E_D(\mathbf{w}), \ldots, \nabla^T_{\mathbf{w}_K} E_D(\mathbf{w}) \right]^T
  \]
Implementation

- Need to compute $[cost, gradient]$:
  
  - $cost = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k \mid x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k$
  
  - $gradient_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k \mid x_n)) x_n^T + \alpha w_k^T$

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

  - $output \ p(C_k \mid x_n) = \frac{\exp(w_k^T x_n)}{\sum_j \exp(w_j^T x_n)}$

  Overflow when $w_k^T x_n$ are too large.
Implementation: Preventing Overflows

- Subtract from each product $w_k^T x_n$ the maximum product:

$$c = \max_{1 \leq k \leq K} w_k^T x_n$$

$$p(C_k | x_n) = \frac{\exp(w_k^T x_n - c)}{\sum_j \exp(w_j^T x_n - c)}$$
Vectorization of Softmax

- Need to compute \([\text{cost}, \text{gradient}]\):

\[
\begin{align*}
\text{cost} &= -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k \\
\text{gradient}_k &= -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T
\end{align*}
\]

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

```python
from scipy.sparse import coo_matrix

groundTruth = coo_matrix((np.ones(N, dtype = np.uint8),
                          (labels, np.arange(N)))).toarray()
```
Vectorization of Softmax

- Compute $cost = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k | x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k$

  - Compute matrix of $w_k^T x_n$.

  - Compute matrix of $w_k^T x_n - c_n$.

  - Compute matrix of $\exp(w_k^T x_n - c_n)$.

  - Compute matrix of $\ln p(C_k | x_n)$.

  - Compute log-likelihood.
Vectorization of Softmax

- Compute $\textbf{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k$

  - Gradient $= [\textbf{grad}_1 | \textbf{grad}_2 | \ldots | \textbf{grad}_K]$
  
    - Compute matrix of $p(C_k | x_n)$.
    - 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()