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 $w$:
  – $h(x) \equiv h_w(x) \equiv h(w,x)$
  – 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** $f(z) = \begin{cases} 
0 & \text{if } z < 0 \\
1 & \text{if } z \geq 0 
\end{cases}$

**Perceptron**

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

**Logistic Regression**

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

**ReLU**

$identity$ $f(z) = z$

**Linear Regression**

Lecture 01
Polynomial curve fitting is Linear Regression:

\[ x = \varphi(x) = [1, x, x^2, \ldots, x^M]^T \]

\[ h(x) = w^T x \]
Perceptron

- Assume classes $T = \{c_1, c_2\} = \{1, 0\}$.
- Training set is $(x_1, t_1), (x_2, t_2), \ldots (x_n, t_n)$.
  $x = [1, x_1, x_2, \ldots, x_k]^T$
  $h(x) = \text{step}(w^T x)$

Lecture 01
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 $(x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)$.
  
  $x = [1, x_1, x_2, \ldots, x_k]^T$

  $h(x) = \sigma(w^T x)$

$$h_w(x) = \frac{1}{1 + \exp(-w^T x)}$$
Logistic Regression for Binary Classification

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

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

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

• How do we train a logistic regression model?
  – What **error/cost function** to minimize?
Logistic Regression Learning

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

• Least Squares error function?

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

  – Differentiable \( \Rightarrow \) can use gradient descent \( \checkmark \)
  – Non-convex \( \Rightarrow \) not guaranteed to find the global optimum \( \times \)
Maximum Likelihood

Training set is \( D = \{ \langle x_n, t_n \rangle | t_n \in \{0,1\}, n \in 1\ldots N \} \)

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

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

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

- The negative log-likelihood (cross entropy) **error function**: \[
E(w) = -\ln p(t | x) = -\sum_{n=1}^{N} \left\{ t_n \ln h_n + (1 - t_n) \ln(1 - h_n) \right\}
\]
Maximum Likelihood Learning for Logistic Regression

• The **ML** solution is:

\[ w_{ML} = \arg \max \ p(t \mid w) = \arg \min \ E(w) \]

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

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

- Use a Gaussian prior over the parameters:
  \[ w = [w_0, w_1, \ldots, w_M]^T \]

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

- Bayes’ Theorem:
  \[ p(w | t) = \frac{p(t | w)p(w)}{p(t)} \propto p(t | w)p(w) \]

- MAP solution:
  \[ w_{MAP} = \arg \max_w p(w | t) \]
Regularized Logistic Regression

- **MAP solution:**

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

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

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

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

\[
= \arg \min_{\mathbf{w}} E_D(\mathbf{w}) + \frac{\alpha}{2} \mathbf{w}^T \mathbf{w}
\]

\[
E_D(\mathbf{w}) = -\sum_{n=1}^{N} \left\{ t_n \ln y_n + (1 - t_n) \ln(1 - y_n) \right\} \quad \text{data term}
\]

\[
E_w(\mathbf{w}) = \frac{\alpha}{2} \mathbf{w}^T \mathbf{w} \quad \text{regularization term}
\]
Regularized Logistic Regression

- **MAP** solution:
  
  \[ w_{MAP} = \arg \min_w E_D(w) + E_w(w) \]

- **ML** solution is given by \( \nabla E(w) = 0 \).

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

  where \( h_n = \sigma(w^T x_n) \)

- Cannot solve analytically => solve numerically:
  - (stochastic) gradient descent [PRML 3.1.3], Newton Raphson iterative optimization [PRML 4.3.3], conjugate gradient, L-BFGS.
Interlude: Gradient Descent
Machine Learning is Optimization

- Parametric ML involves minimizing an **objective function** $J(w)$:
  - Also called **cost function**, **loss function**, or **error function**.
  - Want to find $\hat{w} = \text{argmin}_w J(w)$

- Numerical optimization procedure:
  1. Start with some guess for $w^0$, set $\tau = 0$.
  2. Update $w^\tau$ to $w^{\tau+1}$ such that $J(w^{\tau+1}) \leq J(w^{\tau})$.
  3. Increment $\tau = \tau + 1$.
  4. Repeat from 2 until $J$ cannot be improved anymore.
Gradient Descent Algorithm

• Want to minimize a function $J : \mathbb{R}^n \rightarrow \mathbb{R}$.
  – $J$ is differentiable and convex.
  – Compute gradient of $J$ i.e. direction of steepest increase:

$$\nabla J(w) = \left[ \frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_n} \right]$$

1. Set learning rate $\eta = 0.001$ (or other small value).
2. Start with some guess for $w^0$, set $\tau = 0$.
3. Repeat for epochs $E$ or until $J$ does not improve:
4. $\tau = \tau + 1$.
5. $w^{\tau+1} = w^\tau - \eta \nabla J(w^\tau)$
Learning Rates vs. GD Behavior

http://scs.ryerson.ca/~aharley/neural-networks/

- If learning rate too big => oscillating behavior.
- If learning rate too small => slow convergence.
The Learning Rate

- Use **line search** (backtracking line search, conjugate gradient, …).
- Use **second order methods** (Newton’s method, L-BFGS).
  - Requires computing or estimating the Hessian.
- Use a simple learning rate **annealing schedule**:
  - Start with a relatively large value for the learning rate.
  - Decrease the learning rate as a function of the number of epochs or as a function of the improvement in the objective.
- Use **adaptive learning rates**:
  - Adagrad, Adadelta, RMSProp, Adam.
Gradient Descent: Non-Convexity & Plateaus
Gradient Descent: Saddle Points
Gradient Descent: Ravines
Gradient Descent: Ravines

- **Ravines** are areas where the surface curves much more steeply in one dimension than another.
  - Common around local optima.
  - GD oscillates across the slopes of the ravines, making slow progress towards the local optimum along the bottom.

- **Use momentum** to help accelerate GD in the relevant directions and dampen oscillations:
  - Add a fraction of the past *update vector* to the current update vector.
    - The momentum term increases for dimensions whose previous gradients point in the same direction.
    - It reduces updates for dimensions whose gradients change sign.
    - Also reduces the risk of getting stuck in local minima.
Gradient Descent & Momentum

Vanilla Gradient Descent:

\[ \mathbf{v}^{\tau+1} = \eta \nabla J(\mathbf{w}^\tau) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \mathbf{v}^{\tau+1} \]

Gradient Descent w/ Momentum:

\[ \mathbf{v}^{\tau+1} = \gamma \mathbf{v}^\tau + \eta \nabla J(\mathbf{w}^\tau) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \mathbf{v}^{\tau+1} \]

\( \gamma \) is usually set to 0.9 or similar.
Momentum & Nesterov Accelerated Gradient

GD with Momentum:
\[ v^{\tau+1} = \gamma v^\tau + \eta \nabla J(w^\tau) \]
\[ w^{\tau+1} = w^\tau - v^{\tau+1} \]

Nesterov Accelerated Gradient:
\[ v^{\tau+1} = \gamma v^\tau + \eta \nabla J(w^\tau - \gamma v^\tau) \]
\[ w^{\tau+1} = w^\tau - v^{\tau+1} \]

By making an anticipatory update, NAGs prevents GD from going too fast => significant improvements when training RNNs.
Gradient Descent Optimization Algorithms

- **Momentum.**
- **Nesterov Accelerated Gradient (NAG).**
- Adaptive learning rates methods:
  - Idea is to perform larger updates for infrequent params and smaller updates for frequent params, by accumulating previous gradient values for each parameter.
    - **Adagrad:**
      - Divide update by sqrt of sum of squares of past gradients.
    - **Adadelta:** use exponential decay for past gradients.
    - **RMSProp.**
    - **Adaptive Moment Estimation (Adam)**
RMSProp

- Element-wise gradient: $g_i^t = \nabla_{w_i} J(w_t)$
- Gradient is $g_t = [g_1^t, g_2^t, \ldots, g_K^t]$
- Element-wise square gradient: $g_t^2 = g_t \circ g_t$

RMSProp:

$$E_t[g^2] = \gamma E_{t-1}[g^2] + (1 - \gamma) g_t^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E_t[g^2] + \epsilon}} g_t$$

$\gamma$ is usually set to 0.9, $\eta$ is set to 0.001
Adam: Adaptive Moment Estimation

- Maintain an exponentially decaying average of past gradients (1st m.) and past squared gradients (2nd m.):
  1) \( m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \)
  2) \( v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t^2 \)

- Biased towards 0 during initial steps, use bias-corrected first and second order estimates:
  1) \( \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \)
  2) \( \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \)
Adam: Adaptive Moment Estimation

- First and second moment:
  \[
  m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
  v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t^2
  \]

- Bias-correction:
  \[
  \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \text{and} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}
  \]

\[
\text{Adam:} \\
\quad w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}
\]
Visualization

- Adagrad, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances.
  - Insofar, **Adam** might be the best overall choice.
Variants of Gradient Descent

\[ \mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \nabla J(\mathbf{w}^\tau) \]

- Depending on how much data is used to compute the gradient at each step:
  - **Batch gradient descent**:
    - Use all the training examples.
  - **Stochastic gradient descent** (SGD):
    - Use one training example, update after each.
  - **Minibatch gradient descent**.
    - Use a constant number of training examples (minibatch).
Batch GD vs. Stochastic GD
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:
\[
\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \quad \text{where } \varepsilon = 0.0001
\]
Implementation: Gradient Checking

• If $\theta$ is a vector of parameters $\theta_i$,
  – Compute numerical derivative with respect to each $\theta_i$.
  – Aggregate all derivatives into numerical gradient $G_{\text{num}}(\theta)$.

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

\[
\frac{\left\| G_{\text{num}}(\theta) - G_{\text{imp}}(\theta) \right\|}{\left\| G_{\text{num}}(\theta) + G_{\text{imp}}(\theta) \right\|} \leq 10^{-6}
\]
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:**

\[
C_\ast = \arg \max_{C_k} \ p(C_k \mid x) \\
= \arg \max_{C_k} \ \frac{\exp(w_k^T x)}{\sum_j \exp(w_j^T x)} \\
= \arg \max_{C_k} \ \exp(w_k^T x) \\
= \arg \max_{C_k} \ w_k^T x
\]

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

Lecture 01
Softmax Regression

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

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

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

- The **ML** solution is:
  \[
  w_{ML} = \arg \min_w E_D(w)
  \]

- The **gradient** is (prove it):
  \[
  \nabla_{w_k} E_D(w) = -\frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - p(C_k | x_n) \right) x_n
  \]
  \[
  = -\frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - \frac{\exp(w_k^T x_n)}{Z(x_n)} \right) x_n
  \]

  \[
  \nabla E_D(w) = \left[ \nabla_{w_1}^T E_D(w), \nabla_{w_2}^T E_D(w), \ldots, \nabla_{w_K}^T E_D(w) \right]^T
  \]
Regularized Softmax Regression

- The new **cost** function is:

\[ E(w) = E_D(w) + E_w(w) \]

\[ = - \frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln \frac{\exp(w_k^T x_n)}{Z(x_n)} + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k \]

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

\[ \nabla_{w_k} E(w) = - \frac{1}{N} \sum_{n=1}^{N} \left( \delta_k(t_n) - p(C_k | x_n) \right) x_n^T + \alpha w_k^T \]
Softmax Regression

• **ML** solution is given by $\nabla E_D(w) = 0$.
  - Cannot solve analytically.
  - Solve numerically, by plugging $[\text{cost}, \text{gradient}] = [E_D(w), \nabla E_D(w)]$ 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 \([\text{cost}, \text{gradient}]\):

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

  => need to compute, for \(k = 1, ..., K\):

  \[
  \begin{align*}
  \text{output} \ p(C_k | x_n) &= \frac{\exp(\mathbf{w}_k^T \mathbf{x}_n))}{\sum_j \exp(\mathbf{w}_j^T \mathbf{x}_n))} \\
  \text{Overflow when} \ \mathbf{w}_k^T \mathbf{x}_n \text{are too large.}
  \end{align*}
  \]
Implementation: Preventing Overflows

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

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

$$p(C_k \mid x_n) = \frac{\exp(w_k^T x_n - c))}{\sum_j \exp(w_j^T x_n - c)}$$
Implementation: Vectorization of LR

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

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

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

```python
ggrad = 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))
```

Lecture 03
Implementation: Vectorization of LR

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

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

\[
\text{grad} = \text{np.zeros}(K)
\]

for n in range(N):
    \[ \text{grad} = \text{grad} + (\text{sigmoid}(w\cdot\text{dot}(X[n])) - t[n]) \cdot X[n] \]

---

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

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

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

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

---

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
Vectorization of Softmax

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

  - \[\text{cost} = -\frac{1}{N} \sum_{n=1}^{N} \sum_{k=1}^{K} \delta_k(t_n) \ln p(C_k \mid x_n) + \frac{\alpha}{2} \sum_{k=1}^{K} w_k^T w_k\]

  - \[\text{gradient}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k \mid x_n)) x_n^T + \alpha w_k^T\]

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

\[\text{from scipy.sparse import coo_matrix}\]
\[\text{groundTruth = coo_matrix((np.ones(N, dtype = np.uint8), (labels, np.arange(N)))), toarray()}\]

Lecture 04
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 $\text{grad}_k = -\frac{1}{N} \sum_{n=1}^{N} (\delta_k(t_n) - p(C_k | x_n)) x_n^T + \alpha w_k^T$

  ▪ Gradient = $[\text{grad}_1 | \text{grad}_2 | \ldots | \text{grad}_K]$

  – 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()