Backpropagation

Backpropagation is the algorithm that describes how we update weights in a network, given:
- Training samples
- Training labels
- An cost function

It's used for (almost) all networks.

Network nodes may be:
- Non-linear perceptrons (the most common)
- Convolutional units
- Pooling units
- Batch normalization units
- ...
Partial derivatives as error measures

- Imagine you want to change the output \( z_l^j \) by \( \Delta z_l^j \)
- Then \( \Delta C = \frac{\partial C}{\partial z_l^j} \Delta z_l^j \)
- If \( \left| \frac{\partial C}{\partial z_l^j} \right| \) is large, then \( C \) becomes smaller by giving \( \Delta z_l^j \) the opposite sign
- But if \( \left| \frac{\partial C}{\partial z_l^j} \right| \) is near zero, then \( \Delta z_l^j \) doesn’t matter.
- \( \Delta C = \frac{\partial C}{\partial z_l^j} \Delta z_l^j \)

So where are we?

- We can optimize on a per-sample basis
  - Because the cost function is an average
- Minimizing the \( \delta_s \) optimizes the net
  - The \( \delta_s \) depend on the data samples
- But how do we minimize the \( \delta_s \)?
- We will assume that nodes have non-linear functions, so \( a_l^j = \sigma(z_l^j) \)

Output Layer

- \( \delta_l^j = \frac{\partial C}{\partial a_l^j} \sigma'(z_l^j) \) by the chain rule
- \( \frac{\partial C}{\partial a_l^j} \) is the partial derivative of \( C \) with respect to the activation of output unit \( j \)
  - If \( C \) is LMS (slide #6)
    - \( \frac{\partial C}{\partial a_l^j} = a_l^j(x) - y(x) \)
    - The difference between the output & desired output
      - Explains the \( \frac{1}{2} \) in LMS

Output Layer (cont.)

- \( \sigma'(z_l^j) \) is the derivative of the non-linear transfer function at \( z_l^j \)
  - If \( \sigma(x) = \tanh(x) \), \( \sigma'(x) = 1 - \tanh^2(x) \)
  - If \( \sigma(x) = (1 + e^{-x})^{-1} \), \( \sigma'(x) = \sigma(x)(1 - \sigma(x)) \)
  - \( \delta_l^j = \left(a_l^j(x) - y(x)\right)(1 - \tanh^2(z_l^j(x))) \) or
  - \( \delta_l^j = \left(a_l^j(x) - y(x)\right)\left(a_l^j(x)(1 - a_l^j(x))\right) \)

\( \delta_l^j \) given \( \delta_l^{L+1} \)

- \( \delta_l^j = \sigma'(z_l^j) \sum_k w_{kj}^{l+1} \delta_k^{l+1} \)
  - \( \sigma' \) is computed as on previous slide
  - The RHS is just the sum of the impacts
  - This is where backpropagation comes from
    - Calculate \( \delta_s \) for output layer
    - Then recursively compute \( \delta_s \) for previous layers
Computing $\delta$s...

Layer 1

Layer 2

Layer 3

Cost

Compute intermediate $\delta$s (slide #12)

Compute output layer $\delta$s (slide #11)

So...

• Given an input $x$ and output $y$:
  – We can compute $\delta^l_j$ for every node $j$ at every level $l$
  – Minimizing the $\delta$s will optimize the network
    • Relative to this sample
    • So we need to adjust the weights $w_i$ and $b$ to reduce the $\delta$s
      • But just a little for each input/output pair
      • So we can optimize across all samples

Adjusting $b$

• Remember that $\delta^l_j \equiv \frac{\partial C}{\partial x_j}$ (slide #8)
• And that $z^l_j = w^l_j x + b$
• So $\frac{\partial C}{\partial b_j} = \delta^l_j$
• So $b^l_j \leftarrow (1 - \alpha) b^l_j - \alpha \delta^l_j$
  – Where $\alpha$ is a learning rate
  – Regulates how much you react to each sample

Adjusting w’s

• $\frac{\partial C}{\partial w^l_{jk}} = a_k^l - 1 \delta^l_j$
• So $w^l_{jk} \leftarrow (1 - \alpha) w^l_{jk} - \alpha a_k^l - 1 \delta^l_j$
  – Where $\alpha$ is the same learning rate as before
  – We are collectively minimizing the deltas by heading downhill in the $k+1$ dimensional space defined by $w$ & $b$

Backpropagation (redux)

• Backpropagation updates weights in a network, given
  – Training samples
  – Training labels
  – An cost function
• Network nodes may be
  – Non-linear perceptrons (the most common)
  – Convolutional units
  – Pooling units
  – Batch normalization units
  – …