A Rebirth, one of several
Multi-layer Perceptrons II

• Classic perceptrons threshold linear functions
  – $f(x) = h(w \cdot x + b)$
  – $h()$ is a threshold-based activation function
  – Converts activations into decisions

• But if we want to combine perceptrons?
  – Thresholding individual perceptrons is not useful
  – Replacing $h()$ with identity would allow us to sum linear responses
  – But a sum of linear responses is just another linear response
Sigmoid Activation Functions

• $f(x) = s(w \cdot x + b)$

$y = \tanh(x)$

$y = \left(1 + e^{-x}\right)^{-1}$
Activation Function Properties

• Activation functions **must**
  – Be non-linear

• Activation functions **may**
  – map an infinite domain to a finite range
    • Like [-1, 1] (for tanh) or [0, 1] (for logistic)
    • Keeps values from growing too large/small
    • Sometimes called “squashing”
  – Have non-zero derivatives everywhere
    • Useful for training
Backpropagation

- Backpropagation is the algorithm that describes how we update weights in a network, given:
  - Training samples
  - Training labels
  - A 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
  - …
Goals For Today

• Walk you through the math of backpropagation
  – Complicated, but just calculus
  – Almost universal: modifiable for different node types (see previous slide)
• Today’s derivation assumes multi-layer perceptrons
  ➢ $z(x) = wx + b$
  ➢ $a(x) = h(z(x)) = h(wx + b)$
• Remember the chain rule from calculus:
  ➢ $f(x) = g(h(x)) \rightarrow f'(x) = g'(h(x))h'(x)$
Simple Neural Network

Layer 1

$N^1_1$
$N^1_2$
$N^1_3$
$N^1_4$

Layer 2

$N^2_1$
$N^2_2$

Layer 3

$N^3_1$
$N^3_2$

Notation: superscripts are layers, subscripts are node numbers
Setup for Training: Cost

Layer 1  Layer 2  Layer 3  Cost

\( N^1_1 \)  \( N^2_1 \)  \( N^3_1 \)  \( Y_1 \)
\( N^1_2 \)  \( N^2_2 \)  \( N^3_2 \)  \( Y_2 \)
\( N^1_3 \)
\( N^1_4 \)

(training labels)
Cost Functions

- Cost functions measure the gap between the network output and the ideal output.
- Two necessary properties:
  1. An average over samples: \( C = \frac{1}{n} \sum_x C_x \)
  2. Function of output activations: \( C = C(\alpha_l) \)
- Example: mean squared error
  \[
  C = \frac{1}{2n} \sum_x \| y(x) - a^L(x) \|^2
  \]
δS : local derivatives as error measures

Layer 1  Layer 2  Layer 3  Cost

\[ \Delta C = \frac{\partial C}{\partial z_j^l} \Delta z_j^l \]

Subtle change from previous diagram, now showing cost C not training label.
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.
  - $\frac{\partial C}{\partial z^l_j}$ is already optimal!
  - $\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}$
Recap - 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 $a_j^l = h(z_j^l)$
Output Layer

- \( \delta_j^L = \frac{\partial C}{\partial a_j^L} h'(z_j^L) \) by the chain rule

- \( \frac{\partial C}{\partial a_j^L} \) is the partial derivative of C with respect to the activation of output unit j
  - If C is LMS (slide #6)
    - \( \frac{\partial C}{\partial a_j^L} = a_j^L(x) - y(x) \)
    - The difference between the output & desired output
Output Layer (cont.)

- \( h'(z_j^L) \) is the derivative of the non-linear transfer function at \( z_j^L \)
- If \( h(x) = \tanh(x) \), \( \sigma'(x) = 1 - \tanh^2(x) \)
- If \( h(x) = (1 + e^{-x})^{-1} \),
  \[
  \sigma'(x) = \sigma(x)(1 - \sigma(x))
  \]
- \( \delta_j^L = \left( a_j^L(x) - y(x) \right) \left( 1 - \tanh^2 \left( z_j^L(x) \right) \right) \text{ or } \]
  \( \delta_j^L = \left( a_j^L(x) - y(x) \right) \left( a_j^L(x) \left( 1 - a_j^L(x) \right) \right) \)

3/29/20

CS 510, Image Computation, ©Ross Beveridge
\( \delta^L \text{ given } \delta^{L+1} \)

- \( \delta^l_j = \sigma'(z_j^l) \sum_k w_{kj}^{l+1} \delta_k^{l+1} \)
- \( \sigma' \) is computed as on previous slide
- The RHS is just the sum of the impacts
- This is where \textit{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

Layer 1

Layer 2

Layer 3

Cost

Compute intermediate $\delta_s$

Compute output layer $\delta_s$
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 $\delta_j^l \equiv \frac{\partial c}{\partial z_j^l}$ (slide #8)

• And that $z_j^l = w_j^l x + b$

• So $\frac{\partial c}{\partial b_j^l} = \delta_j^l$

• So $b_j^l \leftarrow (1 - \alpha) b_j^l - \alpha \delta_j^l$
  – Where $\alpha$ is a learning rate
  – Regulates how much you react to each sample
Adjusting w’s

\[ \frac{\partial c}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \]

• So \( w_{jk}^l \leftarrow (1 - \alpha)w_{jk}^l - \alpha a_k^{l-1} \delta_j^l \)
  
  – 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
  – A cost function

• Network nodes may be
  – Non-linear perceptrons (the most common)
  – Convolutional units
  – Pooling units
  – Batch normalization units
  – …
Step Back! Other Resources

• Modulo some notation ambiguity the previous formula-based presentation if fine, but for some of us unsatisfying

• It is best to approach the task of understanding backpropogation simultaneously from three angles.
  1. Mathematical formulas (just finished)
  2. Develop an internal visualization
  3. Running code
May I Recommend

What is backpropagation really doing? | Deep learning, chapter 3

1,698,016 views • Nov 3, 2017
Back propagation with TensorFlow

(Updated for TensorFlow 1.0, at March 6th, 2017)

When I first read about neural network in Michael Nielsen’s Neural Networks and Deep Learning, I was excited to find a good source that explains the material along with actual code. However there was a rather steep jump in the part that describes the basic math and the part that goes about implementing it, and it was especially apparent in the numpy-based code that implements backward propagation.

So, in order to explain it better to myself, and learn about TensorFlow in the process, I took it upon myself to implement the first network in the book using TensorFlow by two means. First, manually defining the back propagation step, and the second - letting TensorFlow do the hard work using automatic differentiation.
As we will discuss in lecture today, I expect everyone to setup a TF 1.14 environment and play with this code.