CSE 490 U: Deep Learning
Spring 2016

Yejin Choi

Some slides from Carlos Guestrin, Andrew Rosenberg, Luke Zettlemoyer
Human Neurons

- **Switching time**
  - \( \sim 0.001 \) second
- **Number of neurons**
  - \( 10^{10} \)
- **Connections per neuron**
  - \( 10^{4-5} \)
- **Scene recognition time**
  - 0.1 seconds
- **Number of cycles per scene recognition?**
  - 100 \( \rightarrow \) much parallel computation!
Perceptron as a Neural Network

This is one neuron:

– Input edges $x_1 \ldots x_n$, along with basis
– The sum is represented graphically
– Sum passed through an activation function $g$

$$g = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
-1 & \text{otherwise}
\end{cases}$$
Sigmoid Neuron

\[ g(w_0 + \sum_{i} w_i x_i) = \frac{1}{1 + e^{-(w_0 + \sum_{i} w_i x_i)}} \]

Just change \( g \)!

- Why would we want to do this?
- Notice new output range \([0,1]\). What was it before?
- Look familiar?
Optimizing a neuron

We train to minimize sum-squared error

\[
\ell(W) = \frac{1}{2} \sum_j [y^j - g(w_0 + \sum_i w_i x^i)]^2
\]

\[
\frac{\partial \ell}{\partial w_i} = - \sum_j [y_j - g(w_0 + \sum_i w_i x^i)] \frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x^i)
\]

\[
\frac{\partial}{\partial w_i} g(w_0 + \sum_i w_i x^i) = x^i g'(w_0 + \sum_i w_i x^i)
\]

\[
\frac{\partial \ell(W)}{\partial w_i} = - \sum_j [y^j - g(w_0 + \sum_i w_i x^i)] x^i \cdot g'(w_0 + \sum_i w_i x^i)
\]

Solution just depends on \(g'\): derivative of activation function!
Sigmoid units: have to differentiate $g$

\[
\frac{\partial \ell(W)}{\partial w_i} = -\sum_j [y^j - g(w_0 + \sum_i w_i x_i^j)] x_i^j \ g'(w_0 + \sum_i w_i x_i^j)
\]

\[
g(x) = \frac{1}{1 + e^{-x}} \quad g'(x) = g(x)(1 - g(x))
\]

\[
\delta^j = [y^j - g(w_0 + \sum_i w_i x_i^j)] g^j(1 - g^j)
\]

\[
g^j = g(w_0 + \sum_i w_i x_i^j)
\]

\[
\omega_i \leftarrow \omega_i + \eta \sum_j x_i^j \delta^j
\]
Perceptron, linear classification, Boolean functions: $x_i \in \{0, 1\}$

- Can learn $x_1 \lor x_2$?
  - $-0.5 + x_1 + x_2$

- Can learn $x_1 \land x_2$?
  - $-1.5 + x_1 + x_2$

- Can learn any conjunction or disjunction?
  - $0.5 + x_1 + ... + x_n$
  - $(-n + 0.5) + x_1 + ... + x_n$

- Can learn majority?
  - $(-0.5n) + x_1 + ... + x_n$

- What are we missing? The dreaded XOR!, etc.
Going beyond linear classification

Solving the XOR problem

\[ y = x_1 \text{ XOR } x_2 = (x_1 \land \neg x_2) \lor (x_2 \land \neg x_1) \]

\[ v_1 = (x_1 \land \neg x_2) = -1.5 + 2x_1 - x_2 \]

\[ v_2 = (x_2 \land \neg x_1) = -1.5 + 2x_2 - x_1 \]

\[ y = v_1 \lor v_2 = -0.5 + v_1 + v_2 \]
Hidden layer

• Single unit:

\[ out(x) = g(w_0 + \sum_i w_i x_i) \]

• 1-hidden layer:

\[ out(x) = g \left( w_0 + \sum_k w_k g(w_0 + \sum_i w_i^k x_i) \right) \]

• No longer convex function!
Example data for NN with hidden layer

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>

Can this be learned??
Learned weights for hidden layer

Learned hidden layer representation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>.89 .04 .08</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>.01 .11 .88</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>.01 .97 .27</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>.99 .97 .71</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>.03 .05 .02</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>.22 .99 .99</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>.80 .01 .98</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>.60 .94 .01</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Why “representation learning”?

- MaxEnt (multinomial logistic regression):
  \[ y = \text{softmax}(w \cdot f(x, y)) \]

- NNs:
  \[ y = \text{softmax}(w \cdot \sigma(Ux)) \]
  \[ y = \text{softmax}(w \cdot \sigma(U^{(n)}(...\sigma(U^{(2)}\sigma(U^{(1)}x)))))) \]

You design the feature vector

Feature representations are “learned” through hidden layers
Very deep models in computer vision

\(^1\)Inception 5 (GoogLeNet)

Inception 7a

\(^1\)Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]
RECURRENT NEURAL NETWORKS
Recurrent Neural Networks (RNNs)

- Each RNN unit computes a new hidden state using the previous state and a new input
  \[ h_t = f(x_t, h_{t-1}) \]
- Each RNN unit (optionally) makes an output using the current hidden state
  \[ y_t = \text{softmax}(V h_t) \]
- Hidden states \( h_t \in R^D \) are continuous vectors
  - Can represent very rich information
  - Possibly the entire history from the beginning
- Parameters are shared (tied) across all RNN units (unlike feedforward NNs)
Recurrent Neural Networks (RNNs)

- Generic RNNs:
  \[ h_t = f(x_t, h_{t-1}) \]
  \[ y_t = \text{softmax}(Vh_t) \]

- Vanilla RNN:
  \[ h_t = \tanh(Ux_t + Wh_{t-1} + b) \]
  \[ y_t = \text{softmax}(Vh_t) \]
Many uses of RNNs

1. Classification (seq to one)

- Input: a sequence
- Output: one label (classification)
- Example: sentiment classification

\[
h_t = f(x_t, h_{t-1})
\]

\[
y = \text{softmax}(Vh_n)
\]
Many uses of RNNs
2. one to seq

• Input: one item
• Output: a sequence
• Example: Image captioning

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(Vh_t)$$
Many uses of RNNs
3. sequence tagging

• Input: a sequence
• Output: a sequence (of the same length)
• Example: POS tagging, Named Entity Recognition
• How about Language Models?
  – Yes! RNNs can be used as LMs!
  – RNNs make markov assumption: T/F?

\[ h_t = f(x_t, h_{t-1}) \]
\[ y_t = \text{softmax}(Vh_t) \]
4. Language models

- Input: a sequence of words
- Output: one next word
- Output: or a sequence of next words
- During training, $x_t$ is the actual word in the training sentence.
- During testing, $x_t$ is the word predicted from the previous time step.
- Does RNN LMs make Markov assumption?
  - i.e., the next word depends only on the previous N words

$$h_t = f(x_t, h_{t-1})$$
$$y_t = \text{softmax}(V h_t)$$
Many uses of RNNs

5. **seq2seq (aka “encoder-decoder”)**

- Input: a sequence
- Output: a sequence (of *different* length)
- Examples?

\[
\begin{align*}
    h_t &= f(x_t, h_{t-1}) \\
    y_t &= \text{softmax}(Vh_t)
\end{align*}
\]
Many uses of RNNs

4. seq2seq (aka “encoder-decoder”)

- Conversation and Dialogue
- Machine Translation
Many uses of RNNs

4. seq2seq (aka “encoder-decoder”)

Parsing!
- “Grammar as Foreign Language” (Vinyals et al., 2015)

\[(S (NP NNP )_{NP} (VP VBZ (NP DT NN )_{NP} )_{VP} )_{S}\]

John has a dog
Recurrent Neural Networks (RNNs)

- **Generic RNNs:**
  \[
  h_t = f(x_t, h_{t-1})
  \]
  \[
  y_t = \text{softmax}(Vh_t)
  \]

- **Vanilla RNN:**
  \[
  h_t = \tanh(Ux_t + Wh_{t-1} + b)
  \]
  \[
  y_t = \text{softmax}(Vh_t)
  \]
Recurrent Neural Networks (RNNs)

- Generic RNNs: \( h_t = f(x_t, h_{t-1}) \)
- Vanilla RNNs: \( h_t = \tanh(Ux_t + Wh_{t-1} + b) \)
- LSTMs (Long Short-term Memory Networks):

\[
\begin{align*}
i_t &= \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)}) \\
f_t &= \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \\
o_t &= \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)}) \\
\tilde{c}_t &= \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \tanh(c_t)
\end{align*}
\]

There are many known variations to this set of equations!
LSTMS (Long Short-Term Memory Networks)

Figure by Christopher Olah (colah.github.io)
LSTMS (LONG SHORT-TERM MEMORY NETWORKS)

sigmoid: [0, 1]

Forget gate: forget the past or not

\[ f_t = \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)}) \]

Figure by Christopher Olah (colah.github.io)
LSTMs (LONG SHORT-TERM MEMORY NETWORKS)

sigmoid: [0, 1]

\[
sigmoid(x) = \frac{1}{1 + e^{-x}}
\]

\[
tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

Forget gate: forget the past or not
\[
f_t = \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)})
\]

Input gate: use the input or not
\[
i_t = \sigma(U^{(i)} x_t + W^{(i)} h_{t-1} + b^{(i)})
\]

New cell content (temp):
\[
\tilde{c}_t = \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)})
\]

Figure by Christopher Olah (colah.github.io)
**LSTMS (LONG SHORT-TERM MEMORY NETWORKS)**

sigmoid: \([0,1]\)

\[
\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}
\]

\[
sigmoid: [0,1]
\]

\[
\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}
\]

tanh: \([-1,1]\)

\[
\text{tanh}(x)
\]

---

Forget gate: forget the past or not

\[
f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)})
\]

Input gate: use the input or not

\[
i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)})
\]

New cell content (temp):

\[
\tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)})
\]

New cell content:

- mix old cell with the new temp cell

\[
c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t
\]

Figure by Christopher Olah (colah.github.io)
**LSTMs (LONG SHORT-TERM MEMORY NETWORKS)**

Output gate: output from the new cell or not
\[ o_t = \sigma(U^{(o)}x_t + W^{(o)}h_{t-1} + b^{(o)}) \]

Hidden state:
\[ h_t = o_t \circ \tanh(c_t) \]

---

Forget gate: forget the past or not
\[ f_t = \sigma(U^{(f)}x_t + W^{(f)}h_{t-1} + b^{(f)}) \]

Input gate: use the input or not
\[ i_t = \sigma(U^{(i)}x_t + W^{(i)}h_{t-1} + b^{(i)}) \]

New cell content (temp):
\[ \tilde{c}_t = \tanh(U^{(c)}x_t + W^{(c)}h_{t-1} + b^{(c)}) \]

New cell content:
- mix old cell with the new temp cell
\[ c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \]

---

Figure by Christopher Olah (colah.github.io)
LSTMS (LONG SHORT-TERM MEMORY NETWORKS)

Forget gate: forget the past or not
Input gate: use the input or not
Output gate: output from the new cell or not

\[
\begin{align*}
  f_t &= \sigma(U^{(f)} x_t + W^{(f)} h_{t-1} + b^{(f)}) \\
  i_t &= \sigma(U^{(i)} x_t + W^{(i)} h_{t-1} + b^{(i)}) \\
  o_t &= \sigma(U^{(o)} x_t + W^{(o)} h_{t-1} + b^{(o)}) \\
  \tilde{c}_t &= \tanh(U^{(c)} x_t + W^{(c)} h_{t-1} + b^{(c)}) \\
  c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t
\end{align*}
\]

New cell content (temp):
New cell content:
  - mix old cell with the new temp cell

Hidden state:
\[
h_t = o_t \circ \tanh(c_t)
\]
vanishing gradient problem for RNNs.

- The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity).
- The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network ‘forgets’ the first inputs.

Example from Graves 2012
For simplicity, all gates are either entirely open (‘O’) or closed (‘—’).
The memory cell ‘remembers’ the first input as long as the forget gate is open and the input gate is closed.
The sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.
Recurrent Neural Networks (RNNs)

- **Generic RNNs:**
  \[ h_t = f(x_t, h_{t-1}) \]

- **Vanilla RNNs:**
  \[ h_t = \tanh(U x_t + W h_{t-1} + b) \]

- **GRUs (Gated Recurrent Units):**
  \[ z_t = \sigma(U^{(z)} x_t + W^{(z)} h_{t-1} + b^{(z)}) \]
  \[ r_t = \sigma(U^{(r)} x_t + W^{(r)} h_{t-1} + b^{(r)}) \]
  \[ \tilde{h}_t = \tanh(U^{(h)} x_t + W^{(h)} (r_t \circ h_{t-1}) + b^{(h)}) \]
  \[ h_t = (1 - z_t) \circ h_{t-1} + z_t \circ \tilde{h}_t \]

Less parameters than LSTMs. Easier to train for comparable performance!
Recursive Neural Networks

- Sometimes, inference over a tree structure makes more sense than sequential structure.
- An example of compositionality in ideological bias detection (red $\rightarrow$ conservative, blue $\rightarrow$ liberal, gray $\rightarrow$ neutral) in which modifier phrases and punctuation cause polarity switches at higher levels of the parse tree.
Recursive Neural Networks

- NNs connected as a tree
- Tree structure is fixed a priori
- Parameters are shared, similarly as RNNs

Example from Iyyer et al., 2014
LEARNING: BACKPROPAGATION
Error Backpropagation

- Model parameters: \( \tilde{\theta} = \{ w_{ij}^{(1)}, w_{jk}^{(2)}, w_{kl}^{(3)} \} \)

for brevity: \( \tilde{\theta} = \{ w_{ij}, w_{jk}, w_{kl} \} \)
• Model parameters: $\tilde{\theta} = \{w_{ij}, w_{jk}, w_{kl}\}$

• Let $a$ and $z$ be the input and output of each node
Error Backpropagation

\[ z_j = g(a_j) \]

\[ a_j = \sum_i w_{ij} z_i \]
• Let $a$ and $z$ be the input and output of each node

\[
a_j = \sum_i w_{ij} z_i \quad a_k = \quad a_l = \quad \]

\[
z_j = g(a_j) \quad z_k = \quad z_l = \quad \]

\[
f(x, \bar{\theta})
\]
• Let \( a \) and \( z \) be the input and output of each node

\[
    a_j = \sum_i w_{ij} z_i \quad a_k = \sum_j w_{jk} z_j \quad a_l = \sum_k w_{kl} z_k
\]

\[
    z_j = g(a_j) \quad z_k = g(a_k) \quad z_l = g(a_l)
\]
Training: minimize loss

\[ R(\theta) = \frac{1}{N} \sum_{n=0}^{N} L(y_n - f(x_n)) \]

Empirical Risk Function

\[ = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} (y_n - f(x_n))^2 \]

\[ = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - g \left( g \left( g \left( x_{n,i} \right) \right) \right) \right)^2 \]
Training: minimize loss

$$R(\theta) = \frac{1}{N} \sum_{n=0}^{N} L(y_n - f(x_n))$$

Empirical Risk Function

$$= \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} (y_n - f(x_n))^2$$

$$= \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - \sum_k w_{kl} g \left( \sum_j w_{jk} g \left( \sum_i w_{ij} x_{n,i} \right) \right) \right)^2$$

Empirical Risk Function

$$= \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - \sum_k w_{kl} g \left( \sum_j w_{jk} g \left( \sum_i w_{ij} x_{n,i} \right) \right) \right)^2$$
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial^2 (y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial \frac{1}{2} (y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}} \right]$$
**Error Backpropagation**

Optimize last layer weights \( w_{kl} \)

\[
L_n = \frac{1}{2} (y_n - f(x_n))^2
\]

\[
\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]
\]

Calculus chain rule

\[
\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial \frac{1}{2} (y_n - g(a_{l,n}))^2}{\partial a_{l,n}} \right] \left[ \frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n}) g'(a_{l,n}) \right] z_{k,n}
\]

Diagram of a neural network with inputs \( x_0, x_1, x_2, \ldots, x_P \), weights \( w_{ij}, w_{jk}, w_{kl} \), and output \( f(x, \tilde{\theta}) \).
Error Backpropagation

Optimize last layer weights $w_{kl}$

$$L_n = \frac{1}{2} (y_n - f(x_n))^2$$

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right]$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial^2}{2} (y_n - g(a_{l,n}))^2 \right] \left[ \frac{\partial z_{k,n} w_{kl}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n}) g'(a_{l,n}) \right] z_{k,n}$$

Calculus chain rule

$$\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n}) g'(a_{l,n}) \right] z_{k,n} = \frac{1}{N} \sum_n \delta_{l,n} z_{k,n}$$

Diagram of a neural network with layers and weights.
Error Backpropagation

Repeat for all previous layers

\[
\frac{\partial R}{\partial w_{kl}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{l,n}} \right] \left[ \frac{\partial a_{l,n}}{\partial w_{kl}} \right] = \frac{1}{N} \sum_n \left[ -(y_n - z_{l,n})g'(a_{l,n}) \right] z_{k,n} = \frac{1}{N} \sum_n \delta_{l,n}z_{k,n}
\]

\[
\frac{\partial R}{\partial w_{jk}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{k,n}} \right] \left[ \frac{\partial a_{k,n}}{\partial w_{jk}} \right] = \frac{1}{N} \sum_n \left[ \sum_l \delta_{l,n}w_{kl}g'(a_{k,n}) \right] z_{j,n} = \frac{1}{N} \sum_n \delta_{k,n}z_{j,n}
\]

\[
\frac{\partial R}{\partial w_{ij}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{j,n}} \right] \left[ \frac{\partial a_{j,n}}{\partial w_{ij}} \right] = \frac{1}{N} \sum_n \left[ \sum_k \delta_{k,n}w_{jk}g'(a_{j,n}) \right] z_{i,n} = \frac{1}{N} \sum_n \delta_{j,n}z_{i,n}
\]
Backprop Recursion

\[ a_j = \sum_i w_{ij} z_i \]

\[ z_j = g(a_j) \]

\[ \frac{\partial R}{\partial w_{jk}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{k,n}} \right] \left[ \frac{\partial a_{k,n}}{\partial w_{jk}} \right] = \frac{1}{N} \sum_n \left[ \sum_l \delta_{l,n} w_{kl} g'(a_{k,n}) \right] z_{j,n} = \frac{1}{N} \sum_n \delta_{k,n} z_{j,n} \]

\[ \frac{\partial R}{\partial w_{ij}} = \frac{1}{N} \sum_n \left[ \frac{\partial L_n}{\partial a_{j,n}} \right] \left[ \frac{\partial a_{j,n}}{\partial w_{ij}} \right] = \frac{1}{N} \sum_n \left[ \sum_k \delta_{k,n} w_{jk} g'(a_{j,n}) \right] z_{i,n} = \frac{1}{N} \sum_n \delta_{j,n} z_{i,n} \]
Learning: Gradient Descent

\[
\begin{align*}
    w^{t+1}_{ij} &= w^t_{ij} - \eta \frac{\partial R}{w_{ij}} \\
    w^{t+1}_{jk} &= w^t_{jk} - \eta \frac{\partial R}{w_{kl}} \\
    w^{t+1}_{kl} &= w^t_{kl} - \eta \frac{\partial R}{w_{kl}}
\end{align*}
\]
Backpropagation

- Starts with a forward sweep to compute all the intermediate function values.
- Through backprop, computes the partial derivatives recursively.
- A form of dynamic programming:
  - Instead of considering exponentially many paths between a weight $w_{ij}$ and the final loss (risk), store and reuse intermediate results.
- A type of automatic differentiation. (There are other variants e.g., recursive differentiation only through forward propagation.)
Backpropagation

- TensorFlow (https://www.tensorflow.org/)
- Torch (http://torch.ch/)
- Theano (http://deeplearning.net/software/theano/)
- CNTK (https://github.com/Microsoft/CNTK)
- cnn (https://github.com/clab/cnn)
- Caffe (http://caffe.berkeleyvision.org/)

Primary Interface Language:
- Python
- Lua
- Python
- C++
- C++
- C++
Cross Entropy Loss (aka log loss, logistic loss)

- Cross Entropy
  \[ H(p, q) = - \sum_y p(y) \log q(y) \]

- Related quantities
  - Entropy
    \[ H(p) = \sum_y p(y) \log p(y) \]
  - KL divergence (the distance between two distributions p and q)
    \[ D_{KL}(p||q) = \sum_y p(y) \log \frac{p(y)}{q(y)} \]
    \[ H(p, q) = E_p[-\log q] = H(p) + D_{KL}(p||q) \]

- Use Cross Entropy for models that should have more probabilistic flavor (e.g., language models)
- Use Mean Squared Error loss for models that focus on correct/incorrect predictions
  \[ \text{MSE} = \frac{1}{2} (y - f(x))^2 \]
RNN Learning: **Backprop Through Time** (BPTT)

- Similar to backprop with non-recurrent NNs
- But unlike feedforward (non-recurrent) NNs, each unit in the computation graph repeats the exact same parameters...
- Backprop gradients of the parameters of each unit as if they are different parameters
- When updating the parameters using the gradients, use the average gradients throughout the entire chain of units.
Convergence of backprop

• Without non-linearity or hidden layers, learning is convex optimization
  – Gradient descent reaches \textbf{global minima}

• Multilayer neural nets (with nonlinearity) are \textbf{not convex}
  – Gradient descent gets stuck in local minima
  – Selecting number of hidden units and layers = fuzzy process
  – NNs have made a HUGE comeback in the last few years
    • Neural nets are back with a new name
      – Deep belief networks
      – Huge error reduction when trained with lots of data on GPUs
Overfitting in NNs

• Are NNs likely to overfit?
  – Yes, they can represent arbitrary functions!!!

• Avoiding overfitting?
  – More training data
  – Fewer hidden nodes / better topology
  – Random perturbation to the graph topology ("Dropout")
  – Regularization
  – Early stopping