CSEP 517: NLP
Introduction to Neural Nets
Autumn 2018

Luke Zettlemoyer
University of Washington

[Many slides from Yejin Choi, Carlos Guestrin]
Next several slides are from Carlos Guestrin, Luke Zettlemoyer
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

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

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

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

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

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?
  - $0.5 + x_1 + \ldots + x_n$
  - $(-n+0.5) + x_1 + \ldots + x_n$
- Can learn majority?
  - $(-0.5 \times n) + x_1 + \ldots + x_n$
- 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:
  \[ \text{out}(x) = g(w_0 + \sum_i w_i x_i) \]

- 1-hidden layer:
  \[ \text{out}(x) = g \left( w_0 + \sum_k w_k g \left( w_0^k + \sum_i w_i^k x_i \right) \right) \]

- 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>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??
A network:

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>00010000</td>
<td>.03 .05 .02</td>
<td>→ 00001000</td>
</tr>
<tr>
<td>00001000</td>
<td>.22 .99 .99</td>
<td>→ 00000100</td>
</tr>
<tr>
<td>00000100</td>
<td>.80 .01 .98</td>
<td>→ 00000010</td>
</tr>
<tr>
<td>00000010</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

\[^1\text{Inception 5 (GoogLeNet)}\]

\[^1\text{Inception 7a}\]

\[^1\text{Going Deeper with Convolutions, [C. Szegedy et al, CVPR 2015]}\]
LEARNING: BACKPROPAGATION
Error Backpropagation

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

for brevity: \( \tilde{\theta} = \{ w_{ij}, w_{jk}, w_{kl} \} \)
Error Backpropagation

- 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:

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

$$z_j = g(a_j) \quad z_k = \quad z_l =$$

The diagram illustrates the flow of information through the nodes, with $x_0, x_1, x_2, \ldots, x_P$ as inputs, and $f(x, \tilde{\theta})$ as the final output.
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

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

\[ = \frac{1}{N} \sum_{n=0}^{N} \frac{1}{2} \left( y_n - \sum_k w_{kl} \left( \sum_j w_{jk} \left( \sum_i w_{ij} x_{n,i} \right) \right) \right)^2 \]
Taking Partial Derivatives...
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

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

Diagram of a neural network with input $x_0, x_1, x_2, ... x_P$, hidden layers $z_i, z_j, z_k$, output layer $a_l$, and final 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

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

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

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

$$= \frac{1}{N} \sum_n \delta_{l,n} z_{k,n}$$

Diagram:
Error Backpropagation

Repeat for all previous layers

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

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

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

\[
\begin{align*}
    w_{ij}^{t+1} &= w_{ij}^t - \eta \frac{\partial R}{w_{ij}} \\
    w_{jk}^{t+1} &= w_{jk}^t - \eta \frac{\partial R}{w_{kl}} \\
    w_{kl}^{t+1} &= w_{kl}^t - \eta \frac{\partial R}{w_{kl}}
\end{align*}
\]
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
  \[ \text{MSE} = \frac{1}{2}(y - f(x))^2 \]
Regularization

• Regularization by objective term

\[ \mathcal{L}(\theta) = \sum_{i=1}^{n} \max\{0, 1 - (\hat{y}_c - \hat{y}_{c'})\} + \lambda ||\theta||^2 \]

  – Modify loss with L1 or L2 norms

• Less depth, smaller hidden states, early stopping

• Dropout
  – Randomly delete parts of network during training
  – Each node (and its corresponding incoming and outgoing edges) dropped with a probability p
  – P is higher for internal nodes, lower for input nodes
  – The full network is used for testing
  – Faster training, better results
Convergence of backprop

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

• Multilayer neural nets (with nonlinearity) are 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