Neural networks for NLP

Feed-forward NNs

Recurrent NNs

Convolutional NNs

Transformer

Always coupled with word embeddings...
This Lecture

- Feedforward Neural Networks
- Applications
  - Neural Bag-of-Words Models
  - Feedforward Neural Language Models
- The training algorithm: Back-propagation
Neural Networks: History
NN “dark ages”

- Neural network algorithms date from the 80s
- ConvNets: applied to MNIST by LeCun in 1998
- Long Short-term Memory Networks (LSTMs): Hochreiter and Schmidhuber 1997
- Henderson 2003: neural shift-reduce parser, not SOTA
2008-2013: A glimmer of light

- Collobert and Weston 2011: “NLP (almost) from Scratch”
  - Feedforward NNs can replace “feature engineering”
  - 2008 version was marred by bad experiments, claimed SOTA but wasn’t, 2011 version tied SOTA

- Krizhevsky et al, 2012: AlexNet for ImageNet Classification

- Socher 2011-2014: tree-structured RNNs working okay

Credits: Greg Durrett
2014: Stuff starts working

  - ConvNets work for NLP!

- Sutskever et al, 2014: sequence-to-sequence for neural MT
  - LSTMs work for NLP!

- Chen and Manning 2014: dependency parsing
  - Even feedforward networks work well for NLP!

- 2015: explosion of neural networks for everything under the sun

Credits: Greg Durrett
Why didn’t they work before?

- **Datasets too small**: for MT, not really better until you have 1M+ parallel sentences (and really need a lot more)

- **Optimization not well understood**: good initialization, per-feature scaling + momentum (Adagrad/Adam) work best out-of-the-box
  - Regularization: dropout is pretty helpful
  - Computers not big enough: can’t run for enough iterations

- Inputs: need **word embeddings** to represent continuous semantics
The “Promise”

- Most NLP works in the past focused on human-designed representations and input features

<table>
<thead>
<tr>
<th>Var</th>
<th>Definition</th>
<th>Value in Fig. 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>count(positive lexicon) $\in$ doc</td>
<td>3</td>
</tr>
<tr>
<td>$x_2$</td>
<td>count(negative lexicon) $\in$ doc</td>
<td>2</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$\begin{cases} 1 &amp; \text{if “no” } \in \text{doc } \ 0 &amp; \text{otherwise} \end{cases}$</td>
<td>1</td>
</tr>
<tr>
<td>$x_4$</td>
<td>count(1st and 2nd pronouns $\in$ doc)</td>
<td>3</td>
</tr>
<tr>
<td>$x_5$</td>
<td>$\begin{cases} 1 &amp; \text{if “!” } \in \text{doc } \ 0 &amp; \text{otherwise} \end{cases}$</td>
<td>0</td>
</tr>
<tr>
<td>$x_6$</td>
<td>log(word count of doc)</td>
<td>$\ln(64) = 4.15$</td>
</tr>
</tbody>
</table>

- **Representation learning** attempts to automatically learn good features and representations

- **Deep learning** attempts to learn multiple levels of representation on increasing complexity/abstraction
Feed-forward Neural Networks
Feed-forward NNs

- Input: $x_1, \ldots, x_d$
- Output: $y \in \{0, 1\}$
Neural computation

Computation units: neurons
An artificial neuron

- A neuron is a computational unit that has scalar inputs and an output.
- Each input has an associated weight.
- The neuron multiplies each input by its weight, sums them, applies a **nonlinear function** to the result, and passes it to its output.
Neural networks

- The neurons are connected to each other, forming a **network**
- The output of a neuron may feed into the inputs of other neurons
A neuron can be a binary logistic regression unit

\[ f(z) = \frac{1}{1 + e^{-z}} \]

\[ h_{\mathbf{w}, b}(\mathbf{x}) = f(\mathbf{w}^T \mathbf{x} + b) \]
A neural network
= many layers of classifiers all learned at once, some providing features for others

* If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs...
* which we can feed into another logistic regression function
Mathematical Notation

- **Input layer:** $x_1, \ldots, x_d$

- **Hidden layer 1:** $h_1^{(1)}, h_2^{(1)}, \ldots, h_{d_1}^{(1)}$

  $h_1^{(1)} = f(W_{1,1}^{(1)} + W_{1,2}^{(1)} x_2 + \ldots + W_{1,d}^{(1)} x_d + b_1^{(1)})$

  $h_2^{(1)} = f(W_{2,1}^{(1)} + W_{2,2}^{(1)} x_2 + \ldots + W_{2,d}^{(1)} x_d + b_2^{(1)})$

  \[ \ldots \]

- **Hidden layer 2:** $h_1^{(2)}, h_2^{(2)}, \ldots, h_{d_2}^{(2)}$

  $h_1^{(2)} = f(W_{1,1}^{(2)} h_1^{(1)} + W_{1,2}^{(2)} h_2^{(1)} + \ldots + W_{1,d_1}^{(2)} h_{d_1}^{(1)} + b_1^{(2)})$

  $h_2^{(2)} = f(W_{2,1}^{(2)} h_1^{(1)} + W_{2,2}^{(2)} h_2^{(1)} + \ldots + W_{2,d_1}^{(2)} h_{d_1}^{(1)} + b_2^{(2)})$

  \[ \ldots \]

- **Output layer:**

  $y = \sigma(w_1^{(o)} h_1^{(2)} + w_2^{(o)} h_2^{(2)} + \ldots + w_{d_2}^{(o)} h_{d_2}^{(2)} + b^{(o)})$
Matrix Notation

- **Input layer:** \( x \in \mathbb{R}^d \)

- **Hidden layer 1:**
  \[
  h_1 = f(\mathbf{W}^{(1)} x + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}
  \]
  \( \mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d} \), \( \mathbf{b}^{(1)} \in \mathbb{R}^{d_1} \)

- **Hidden layer 2:**
  \[
  h_2 = f(\mathbf{W}^{(2)} h_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}
  \]
  \( \mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1} \), \( \mathbf{b}^{(2)} \in \mathbb{R}^{d_2} \)

- **Output layer:**
  \[
  y = \sigma(\mathbf{w}^{(o)} \cdot h_2 + b^{(o)})
  \]

*: \( f \) is applied element-wise

\[
  f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]
  \]
Mathematical Notation, side by side

- Input layer: $x_1, \ldots, x_d$

- Hidden layer 1: $h_1^{(1)}, h_2^{(1)}, \ldots, h_{d_1}^{(1)}$
  
  $h_1^{(1)} = f(W_{1,1}^{(1)} + W_{1,2}^{(1)}x_2 + \ldots + W_{1,d}^{(1)}x_d + b_1^{(1)})$
  
  $h_2^{(1)} = f(W_{2,1}^{(1)} + W_{2,2}^{(1)}x_2 + \ldots + W_{2,d}^{(1)}x_d + b_2^{(1)})$
  
  \ldots

- Hidden layer 2: $h_1^{(2)}, h_2^{(2)}, \ldots, h_{d_2}^{(2)}$
  
  $h_1^{(2)} = f(W_{1,1}^{(2)}h_1^{(1)} + W_{1,2}^{(2)}h_2^{(1)} + \ldots + W_{1,d_1}^{(2)}h_{d_1}^{(1)} + b_1^{(2)})$
  
  $h_2^{(2)} = f(W_{2,1}^{(2)}h_1^{(1)} + W_{2,2}^{(2)}h_2^{(1)} + \ldots + W_{2,d_1}^{(2)}h_{d_1}^{(1)} + b_2^{(2)})$
  
  \ldots

- Output layer:
  
  $y = \sigma(w_1^{(o)}h_1^{(2)} + w_2^{(o)}h_2^{(2)} + \ldots + w_{d_2}^{(o)}h_{d_2}^{(2)} + b^{(o)})$

- Input layer: $x \in \mathbb{R}^d$

- Hidden layer 1:
  
  $h_1 = f(W^{(1)}x + b^{(1)}) \in \mathbb{R}^{d_1}$
  
  $W^{(1)} \in \mathbb{R}^{d_1 \times d}, b^{(1)} \in \mathbb{R}^{d_1}$

- Hidden layer 2:
  
  $h_2 = f(W^{(2)}h_1 + b^{(2)}) \in \mathbb{R}^{d_2}$
  
  $W^{(2)} \in \mathbb{R}^{d_2 \times d_1}, b^{(2)} \in \mathbb{R}^{d_2}$

- Output layer:
  
  $y = \sigma(w^{(o)} \cdot h_2 + b^{(o)})$
Why non-linearities?

- Neural networks can learn much more complex functions and nonlinear decision boundaries

The capacity of the network increases with more hidden units and more hidden layers

How if we remove activation function?
**Activation functions**

sigmoid

\[ f(z) = \frac{1}{1 + e^{-z}} \]

\[ f'(z) = f(z) \times (1 - f(z)) \]

**tanh**

\[ f(z) = \frac{e^{2z} - 1}{e^{2z} + 1} \]

\[ f'(z) = 1 - f(z)^2 \]

**ReLU (rectified linear unit)**

\[ f(z) = \max(0, z) \]

\[ f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases} \]

Advantages of ReLU?
Activation functions

Problems of ReLU?  “dead neurons”

Leaky ReLU

\[
f(z) = \begin{cases} 
    z & z \geq 0 \\
    0.01z & z < 0 
\end{cases}
\]
Loss functions

- **Binary classification**

  \[ y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)}) \]

  \[ \mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y) \]

- **Multi-class classification** (\(C\) classes)

  \[ y_i = \text{softmax}_i(\mathbf{W}^{(o)} \mathbf{h}_2 + \mathbf{b}^{(o)}) \quad \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}, \mathbf{b}^{(o)} \in \mathbb{R}^C \]

  \[ \mathcal{L}(y, y^*) = - \sum_{i=1}^{C} y_i^* \log y_i \quad \text{softmax}_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \]

  The question again becomes how to compute: \( \nabla_\theta \mathcal{L}(\theta) \)

  \[ \theta = \{ \mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \mathbf{w}^{(o)}, \mathbf{b}^{(o)} \} \]

- **Regression**

  \[ y = \mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)} \]

  \[ \mathcal{L}_{\text{MSE}}(y, y^*) = (y - y^*)^2 \]
Optimization

\[ \theta^{(t+1)} = \theta^{(t)} - \eta \nabla_{\theta} J(\theta) \]

- Logistic regression is convex: one global minimum
- Neural networks are non-convex and not easy to optimize
- A class of more sophisticated “adaptive” optimizers that scale the parameter adjustment by an accumulated gradient.
  - Adam
  - Adagrad
  - RMSprop
  - ...

(Ruder 2016): An overview of gradient descent optimization algorithms
Dropout

- Form of regularization NNs

- **Idea:** “Handicap” NN by removing hidden units **stochastically**
  - set each hidden unit in a layer to 0 with probability $p$ during training ($p = 0.5$ usually works well)
  - scale outputs by $1/(1 - p)$
  - hidden units forced to learn more general patterns

- **Test time:** Simply compute identity
What can we compute with NNs?
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

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

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

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

\[ y = v_1 \lor v_2 \]
\[ = -0.5 + v_1 + v_2 \]
Example data for NN with hidden layer

A target function:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000</td>
<td>10000000</td>
</tr>
<tr>
<td>010000000</td>
<td>01000000</td>
</tr>
<tr>
<td>001000000</td>
<td>00100000</td>
</tr>
<tr>
<td>000100000</td>
<td>00010000</td>
</tr>
<tr>
<td>000010000</td>
<td>00001000</td>
</tr>
<tr>
<td>000001000</td>
<td>00000100</td>
</tr>
<tr>
<td>000000100</td>
<td>00000010</td>
</tr>
<tr>
<td>000000010</td>
<td>00000001</td>
</tr>
<tr>
<td>000000001</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>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>.22 .99 .99</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>.80 .01 .98</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>.60 .94 .01</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>.60 .94 .01</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Applications
Neural Bag-of-Words (NBOW)

- Deep Averaging Networks (DAN) for Text Classification

(Iyyer et al. 2015): Deep Unordered Composition Rivals Syntactic Methods for Text Classification
Word embeddings: re-train or not?

• Word embeddings can be treated as parameters too!

$$\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, w^{(o)}, b^{(o)}, E_{emb}\}$$

• When the training set is small, don’t re-train word embeddings (think of them as features!).

• Most cases: initialize word embeddings using pre-trained ones (word2vec, Glove) and re-train them for the task

• When you have enough data, you can just randomly initialize them and train from scratch (e.g. machine translation)
Neural Bag-of-Words (NBO\textsuperscript{W})

<table>
<thead>
<tr>
<th>Model</th>
<th>RT</th>
<th>SST fine</th>
<th>SST bin</th>
<th>IMDB</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAN-ROOT</td>
<td>—</td>
<td>46.9</td>
<td>85.7</td>
<td>—</td>
<td>31</td>
</tr>
<tr>
<td>DAN-RAND</td>
<td>77.3</td>
<td>45.4</td>
<td>83.2</td>
<td>88.8</td>
<td>136</td>
</tr>
<tr>
<td>DAN</td>
<td>80.3</td>
<td>47.7</td>
<td>86.3</td>
<td>89.4</td>
<td>136</td>
</tr>
<tr>
<td>NBO\textsuperscript{W}-RAND</td>
<td>76.2</td>
<td>42.3</td>
<td>81.4</td>
<td>88.9</td>
<td>91</td>
</tr>
<tr>
<td>NBO\textsuperscript{W}</td>
<td>79.0</td>
<td>43.6</td>
<td>83.6</td>
<td>89.0</td>
<td>91</td>
</tr>
<tr>
<td>Bi\textsuperscript{NB}</td>
<td>—</td>
<td>41.9</td>
<td>83.1</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>NBSVM-bi</td>
<td>79.4</td>
<td>—</td>
<td>—</td>
<td>91.2</td>
<td>—</td>
</tr>
</tbody>
</table>
Feedforward Neural LMs

- N-gram models: $P(\text{mat}|\text{the cat sat on the})$

Input layer (context size $n = 5$):

$$x = [\mathbf{e}_{\text{the}}; \mathbf{e}_{\text{cat}}; \mathbf{e}_{\text{sat}}; \mathbf{e}_{\text{on}}; \mathbf{e}_{\text{the}}] \in \mathbb{R}^{dn}$$

Hidden layer

$$h = \text{tanh}(Wx + b) \in \mathbb{R}^{h}$$

Output layer (softmax)

$$z = Uh \in \mathbb{R}^{|V|}$$

$$P(w = i | \text{context}) = \text{softmax}_i(z)$$

(Bengio et al. 2003): A Neural Probabilistic Language Model
Backpropagation

How to compute gradients?
Backpropagation

- It’s taking derivatives and applying chain rule!

- We’ll **re-use** derivatives computed for higher layers in computing derivatives for lower layers so as to minimize computation.

- Good news is that modern automatic differentiation tools did all for you!
  - Implementing backprop by hand is like programming in assembly language.
Deriving gradients for Feedforward NNs

Input: \( x \)

\[
h_1 = \tanh(W_1 x + b_1)
\]

\[
h_2 = \tanh(W_2 h_1 + b_2)
\]

\[
y = \sigma(W^T h_2 + b)
\]

\[
\mathcal{L}(y, y^*) = -y^* \log y - (1 - y^*) \log (1 - y)
\]

\[
\frac{\partial \mathcal{L}}{\partial w} = ? \quad \frac{\partial \mathcal{L}}{\partial b} = ?
\]

\[
\frac{\partial \mathcal{L}}{\partial W_2} = ? \quad \frac{\partial \mathcal{L}}{\partial b_2} = ?
\]

\[
\frac{\partial \mathcal{L}}{\partial W_1} = ? \quad \frac{\partial \mathcal{L}}{\partial b_1} = ?
\]
Deriving gradients for Feedforward NNs

\[ z_1 = W_1 x + b_1 \quad h_1 = \tanh(z_1) \]
\[ z_2 = W_2 h_1 + b_2 \quad h_2 = \tanh(z_2) \]
\[ y = \sigma(w^T h_2 + b) \]

\[ \frac{\partial L}{\partial b} = y - y^* \quad \frac{\partial L}{\partial w} = (y - y^*)h_2 \quad \frac{\partial L}{\partial h_2} = (y - y^*)w \]

\[ \frac{\partial L}{\partial z_2} = (1 - h_2^2) \circ \frac{\partial L}{\partial h_2} \]

\[ \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial z_2} h_1^T \quad \frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2} \quad \frac{\partial L}{\partial h_1} = W_2^T \frac{\partial L}{\partial z_2} \]

\[ \frac{\partial L}{\partial z_1} = (1 - h_1^2) \circ \frac{\partial L}{\partial h_1} \quad \frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial z_1} x^T \quad \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1} \]
Computational graphs

\[ x \rightarrow Wx + z \rightarrow f \rightarrow h \rightarrow s \]

\[ x \rightarrow Wx + z \rightarrow f \rightarrow h \rightarrow s \]

Credits: Chris Manning
An example

\[ f(x, y, z) = (x + y) \max(y, z) \]
\[ x = 1, y = 2, z = 0 \]

\[ a = x + y \]
\[ b = \max(y, z) \]
\[ f = ab \]

Compute the gradients yourself!
Backpropagation in general computational graph

- Forward propagation: visit nodes in topological sort order
  - Compute value of node given predecessors
- Backward propagation:
  - Initialize output gradient as 1
  - Visit nodes in reverse order and compute gradient wrt each node using gradient wrt successors

\[
\frac{\partial L}{\partial x} = \sum_{i=1}^{n} \frac{\partial L}{\partial y_i} \frac{\partial y_i}{\partial x}
\]

\{y_1, \ldots, y_n\} = \text{successors of } x