Neural Networks

Wei Xu

(many slides from Greg Durrett and Philipp Koehn)
Recap: Loss Functions

- Hinge (SVM)
- Logistic
- 0-1 (ideal)
- Perceptron
Recap: Logistic Regression

\[ P(y = + | x) = \text{logistic}(w^\top x) \]

\[ P(y = + | x) = \frac{\exp(\sum_{i=1}^{n} w_i x_i)}{1 + \exp(\sum_{i=1}^{n} w_i x_i)} \]

- To learn weights: maximize discriminative log likelihood of data \( P(y|x) \)

\[ \mathcal{L}(x_j, y_j = +) = \log P(y_j = + | x_j) \]

\[ = \sum_{i=1}^{n} w_i x_{ji} - \log \left( 1 + \exp \left( \sum_{i=1}^{n} w_i x_{ji} \right) \right) \]

sum over features
Recap: Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp\left(w^\top f(x, y)\right)}{\sum_{y' \in Y} \exp\left(w^\top f(x, y')\right)}$$

Why? Interpret raw classifier scores as probabilities

too many drug trials, too few patients

Health: +2.2
Sports: +3.1
Science: -0.6

$$w^\top f(x, y)$$ unnormalized probabilities

$$\exp$$ probabilities must be \geq 0

$$\sum_{y' \in Y}$$ sum over output space to normalize

Why? Interpret raw classifier scores as probabilities

log(0.21) = -1.56

0.21 0.77 0.02 probabilities must sum to 1

0.21 0.77 0.02 probabilites

L(x_j, y_j^*) = log P(y_j^*|x_j)

1.00 0.00 0.00 correct (gold) probabilities
Recap: Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp \left( w^\top f(x, y) \right)}{\sum_{y' \in \mathcal{Y}} \exp \left( w^\top f(x, y') \right)}$$

sum over output space to normalize

- Training: maximize $\mathcal{L}(x, y) = \sum_{j=1}^{n} \log P(y_j^*|x_j)$

$$= \sum_{j=1}^{n} \left( w^\top f(x_j, y_j^*) - \log \sum_{y} \exp(w^\top f(x_j, y)) \right)$$
This Lecture

- Neural network history
- Neural network basics
- Feedforward neural networks + backpropagation
- Applications
- Implementing neural networks (if time)
History: NN “dark ages”

- ConvNets: applied to MNIST by LeCun in 1998
- LSTMs: Hochreiter and Schmidhuber (1997)

https://www.youtube.com/watch?v=FwFduRA_16Q&feature=youtu.be
2008-2013: A glimmer of light...

- Collobert and Weston 2011: “NLP (almost) from scratch”
  - Feedforward neural nets induce features for sequential CRFs (“neural CRF”)
  - 2008 version was marred by bad experiments, claimed SOTA but wasn’t, 2011 version tied SOTA
- Krizhevsky et al. (2012): AlexNet for vision
- Socher 2011-2014: tree-structured RNNs working okay
2014: Stuff starts working


- Sutskever et al. (2014) + Bahdanau et al. (2015): seq2seq + attention for neural MT (LSTMs work for NLP?)

- Chen and Manning (2014) transition-based dependency parser (even feedforward networks work well for NLP?)

- 2015: explosion of neural nets for everything under the sun
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 / AdaDelta / 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 representations to have the right continuous semantics
Neural Networks: motivation

- Linear classification: \( \arg\max_y w^\top f(x, y) \)

- How can we do nonlinear classification? Kernels are too slow...

- Want to learn intermediate conjunctive features of the input

  the movie was not all that good

  \( I[\text{contains not} \& \text{contains good}] \)
Neural Networks: XOR

- Let’s see how we can use neural nets to learn a simple nonlinear function
- Inputs $x_1, x_2$
  
  (generally $x = (x_1, \ldots, x_m)$)
- Output $y$
  
  (generally $y = (y_1, \ldots, y_n)$)

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y = x_1 \text{ XOR } x_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Neural Networks: XOR

\[ y = a_1 x_1 + a_2 x_2 \]

\[ y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2) \]

(looks like action potential in neuron)
Neural Networks: XOR

\[
y = a_1 x_1 + a_2 x_2
\]

\[
y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)
\]

\[
y = -x_1 - x_2 + 2 \tanh(x_1 + x_2)
\]

"or"
Neural Networks: XOR

\[ y = -2x_1 - x_2 + 2 \tanh(x_1 + x_2) \]

the movie was not all that good
Neural Networks

Linear model: \[ y = \mathbf{w} \cdot \mathbf{x} + b \]
\[ y = g(\mathbf{w} \cdot \mathbf{x} + b) \]
\[ y = g(\mathbf{W} \mathbf{x} + \mathbf{b}) \]

Nonlinear transformation
Warp space
Shift

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
Neural Networks

Linear classifier  Neural network  ...possible because we transformed the space!

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
Deep Neural Networks

The output of the first layer is given by:

\[ z = g(Vy + c) \]

The output of the second layer is given by:

\[ y = g(Wx + b) \]
\[ z = g(Vg(Wx + b) + c) \]

The output of the first layer is:

\[ z = V(Wx + b) + c \]

“Feedforward” computation (not recurrent)

Check: what happens if no nonlinearity?

More powerful than basic linear models?
Deep Neural Networks

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
Feedforward Networks,
Backpropagation
Simple Neural Network

[Diagram of a neural network with nodes and weights:]

- Node 1 with weights 3.7, 2.9, 3.7, 2.9, 1, 1, 3.7, 2.9, 1, 1.

One innovation: bias units (no inputs, always value 1)

https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf
Try out two input values

- Sample Input:
  - 1.0
  - 0.0

- Hidden unit computation:
  - $\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1.0 \times 1.5) = \text{sigmoid}(2.1) = 0.90$
  - $\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1.0 \times 4.5) = \text{sigmoid}(1.6) = 0.17$
Try out two input values

Hidden unit computation

\[
\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1+e^{-2.2}} = 0.90
\]

\[
\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1+e^{1.6}} = 0.17
\]
Output unit computation

\[
sigmoid(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = sigmoid(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76
\]
Network implements XOR

- $h_0$ is OR
- $h_1$ is AND

<table>
<thead>
<tr>
<th>Input $x_0$</th>
<th>Input $x_1$</th>
<th>Hidden $h_0$</th>
<th>Hidden $h_1$</th>
<th>Output $y_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.12</td>
<td>0.02</td>
<td>0.18 → 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.88</td>
<td>0.27</td>
<td>0.74 → 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.73</td>
<td>0.12</td>
<td>0.74 → 1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.99</td>
<td>0.73</td>
<td>0.33 → 0</td>
</tr>
</tbody>
</table>
How do we adjust the weights?

Computed output: \( y = 0.76 \)

Correct output: \( t = 1.0 \)

Q: how do we adjust the weights?
Derivative of Sigmoid

- Sigmoid function:

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

- Derivative:

\[
\frac{d}{dx} \left( \frac{1}{1 + e^{-x}} \right) = \frac{d}{dx} \frac{1}{1 + e^{-x}}
\]

\[
= 0 \times (1 - e^{-x}) - (-e^{-x})
\]

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

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

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

\[
= sigmoid(x)(1 - sigmoid(x))
\]
Final Layer Update

- Linear combination of weights:  \( s = \sum_k w_k h_k \)
- Activation function:  \( y = \text{sigmoid}(s) \)
- Error (L2 norm):  \( E = \frac{1}{2}(t - y)^2 \)
- Derivative of error with regard to one weight  \( w_k \):
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}
  \]
Final Layer Update (1)

- Linear combination of weights:  \( s = \sum_k w_k h_k \)
- Activation function:  \( y = \text{sigmoid}(s) \)
- Error (L2 norm):  \( E = \frac{1}{2}(t - y)^2 \)
- Derivative of error with regard to one weight \( w_k \):
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}
  \]

- Error \( E \) is defined with respect to \( y \):
  \[
  \frac{dE}{dy} = \frac{d}{dy} \frac{1}{2}(t - y)^2 = -(t - y)
  \]
Final Layer Update (2)

- Linear combination of weights: \( s = \sum_k w_k h_k \)
- Activation function: \( y = \text{sigmoid}(s) \)
- Error (L2 norm): \( E = \frac{1}{2}(t - y)^2 \)
- Derivative of error with regard to one weight \( w_k \):
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}
  \]

- \( y \) with respect to \( s \) is \( \text{sigmoid}(s) \):
  \[
  \frac{dy}{ds} = \frac{d}{ds} \text{sigmoid}(s) = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)
  \]
Final Layer Update (3)

- Linear combination of weights: \( s = \sum_k w_k h_k \)
- Activation function: \( y = \text{sigmoid}(s) \)
- Error (L2 norm): \( E = \frac{1}{2}(t - y)^2 \)
- Derivative of error with regard to one weight \( w_k \):
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k}
  \]

- \( s \) is weighted linear combination of hidden node values \( h_k \):
  \[
  \frac{ds}{dw_k} = \frac{d}{dw_k} \sum_k w_k h_k = h_k
  \]
Putting it All Together

- Derivative of error with regard to one weight $w_k$:
  \[
  \frac{dE}{dw_k} = \frac{dE}{dy} \frac{dy}{ds} \frac{ds}{dw_k} = -(t - y) \ y(1 - y) \ h_k
  \]
  error          derivative of sigmoid: $y'$

- Weighted adjustment will be scaled by a fixed learning rate $\mu$:
  \[
  \Delta w_k = \mu (t - y) \ y' \ h_k
  \]
Multiple Output Nodes

- Previous slides discussed the situation with only one output node:
  
  \[ E = \frac{1}{2}(t - y)^2 \quad \Delta w_k = \mu (t - y) y' h_k \]

- But, typically neural networks have multiple output nodes

- Error is computed over all j output nodes:
  
  \[ E = \sum_j \frac{1}{2}(t_j - y_j)^2 \]

- Weights are adjusted according to the node they point to:
  
  \[ \Delta w_{j \leftarrow k} = \mu(t_j - y_j) y'_j h_k \]
Hidden Layer Update

- In a hidden layer, we do not have a target output value
- But, we can compute how much each node contributed to downstream error
- Definition of error term of each node:
  \[ \delta_j = (t_j - y_j) \, y_j' \]

- Back-propagate the error term:
  \[ \delta_i = \left( \sum_j w_{j \leftarrow i} \delta_j \right) \, y_i' \]

- Universal update formula:
  \[ \Delta w_{j \leftarrow k} = \mu \, \delta_j \, h_k \]
Our Example

- Computed output: \( y = 0.76 \)
- Correct output: \( t = 1.0 \)

Q: how do we adjust the weights?
Our Example

- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$):

  \[
  \delta_G = (t - y) y' = (1 - .76) 0.181 = .0434
  \]

  \[
  \Delta w_{GD} = \mu \delta_G h_D = 10 \times .0434 \times .90 = .391
  \]

  \[
  \Delta w_{GE} = \mu \delta_G h_E = 10 \times .0434 \times .17 = .074
  \]

  \[
  \Delta w_{GF} = \mu \delta_G h_F = 10 \times .0434 \times 1 = .434
  \]
Our Example

- Computed output: \( y = .76 \)
- Correct output: \( t = 1.0 \)
- Final layer weight updates (learning rate \( \mu = 10 \)):

\[
\begin{align*}
\delta_G &= (t - y) \ y' = (1 - .76) \ 0.181 = .0434 \\
\Delta w_{GD} &= \mu \ \delta_G \ h_D = 10 \times .0434 \times .90 = .391 \\
\Delta w_{GE} &= \mu \ \delta_G \ h_E = 10 \times .0434 \times .17 = .074 \\
\Delta w_{GF} &= \mu \ \delta_G \ h_F = 10 \times .0434 \times 1 = .434
\end{align*}
\]
Hidden Layer Updates

- **Hidden node D:**
  \[ \delta_D = \left( \sum_j w_{j \rightarrow d} \delta_j \right) y'_D = w_{gD} \delta_g y'_D = 4.5 \times 0.0434 \times 0.0898 = 0.0175 \]
  \[ \Delta w_{DA} = \mu \delta_D h_A = 10 \times 0.0175 \times 1.0 = 1.75 \]
  \[ \Delta w_{DB} = \mu \delta_D h_B = 10 \times 0.0175 \times 0.0 = 0 \]
  \[ \Delta w_{DC} = \mu \delta_D h_C = 10 \times 0.0175 \times 1 = 1.75 \]

- **Hidden node E:**
  \[ \delta_E = \left( \sum_j w_{j \rightarrow e} \delta_j \right) y'_E = w_{gE} \delta_g y'_E = -5.2 \times 0.0434 \times 0.2055 = -0.0464 \]
  \[ \Delta w_{EA} = \mu \delta_E h_A = 10 \times -0.0464 \times 1.0 = -0.464 \]
  etc.

https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf
Logistic Regression with NNs

\[ P(y|x) = \frac{\exp(w^T f(x, y))}{\sum_{y'} \exp(w^T f(x, y'))} \]

\[ P(y|x) = \text{softmax} \left( [w^T f(x, y)]_{y \in Y} \right) \]

\[ \text{softmax}(p)_i = \frac{\exp(p_i)}{\sum_{i'} \exp(p_{i'})} \]

\[ P(y|x) = \text{softmax}(W f(x)) \]

\[ P(y|x) = \text{softmax}(W g(V f(x))) \]

- Single scalar probability
- Compute scores for all possible labels at once (returns vector)
- softmax: exponentiates and normalizes a given vector
- Weight vector per class; \( W \) is \([\text{num classes} \times \text{num feats}]\)
- Now one hidden layer
We can think of a neural network classifier with one hidden layer as building a vector $z$ which is a hidden layer representation (i.e. latent features) of the input, and then running standard logistic regression on the features that the network develops in $z$. 
Training Neural Networks

\[ P(y|x) = \text{softmax}(Wz) \quad z = g(Vf(x)) \]

- Maximize log likelihood of training data

\[ \mathcal{L}(x, i^*) = \log P(y = i^*|x) = \log (\text{softmax}(Wz) \cdot e_{i^*}) \]

- \( i^* \): index of the gold label

- \( e_i \): 1 in the \( i \)th row, zero elsewhere. Dot by this = select \( i \)th index

\[ \mathcal{L}(x, i^*) = Wz \cdot e_{i^*} - \log \sum_j \exp(Wz) \cdot e_j \]
Computing Gradients

\[ \mathcal{L}(x, i^*) = Wz \cdot e_{i^*} - \log \sum_j \exp(Wz) \cdot e_j \]

- Gradient with respect to \( W \)

\[ \frac{\partial}{\partial W_{ij}} \mathcal{L}(x, i^*) = \begin{cases} 
    z_j - P(y = i|x)z_j & \text{if } i = i^* \\
    -P(y = i|x)z_j & \text{otherwise}
\end{cases} \]

- Looks like logistic regression with \( z \) as the features!
Neural Networks for Classification

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

\[ \frac{\partial L(x, i^*)}{\partial W} = z(e_{i^*} - P(y|x)) = z \cdot \text{err(root)} \]
Computing Gradients

\[ \mathcal{L}(\mathbf{x}, i^*) = W \mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W \mathbf{z}) \cdot e_j \]

- Gradient with respect to \( \mathbf{z} \) [some math...]

\[ \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = W_{i^*} - \sum_j P(y = j | \mathbf{x}) W_j \]

\[ \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = \text{err}(\mathbf{z}) = W^\top \text{err}(\text{root}) \]
\[ \text{dim} = d \]

\[ \text{err}(\text{root}) = e_{i^*} - P(y|\mathbf{x}) \]
\[ \text{dim} = \text{num\_classes} \]
Backpropagation: Picture

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

- Can forget everything after \( z \), treat it as the output and keep backpropping

\[
\frac{\partial L(x, i^*)}{\partial z} = err(z) = W^\top err(\text{root})
\]
Computing Gradients: Backpropagation

\[ \mathcal{L}(\mathbf{x}, i^*) = W \mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W \mathbf{z}) \cdot e_j \]

- Gradient with respect to \( V \): apply the chain rule

\[
\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}}
\]

\[ \text{dim} = \text{num\_classes} \]

\[ \text{dim} = d \]

\[ z = g(V f(x)) \]

Activations at hidden layer

\[ \text{err}(\text{root}) = e_{i^*} - P(y | x) \]

\[ \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = \text{err}(\mathbf{z}) = W^T \text{err}(\text{root}) \]
Computing Gradients: Backpropagation

\[ \mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_{j=1}^{m} \exp(W\mathbf{z} \cdot e_j) \]

- Gradient with respect to \( V \): apply the chain rule

\[ \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{V_{ij}} \quad \frac{\partial \mathbf{z}}{V_{ij}} = \frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial V_{ij}} \]

- First term: gradient of nonlinear activation function at \( \mathbf{a} \) (depends on current value)

- Second term: gradient of linear function

- Straightforward computation once we have \( \text{err}(\mathbf{z}) \)

\[ \mathbf{z} = g(Vf(\mathbf{x})) \]

Activations at hidden layer
\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

\[ \text{err(root)} = e_{i^*} - P(y|x) \]

\[ \frac{\partial L(x, i^*)}{\partial V_{ij}} = \frac{\partial L(x, i^*)}{\partial z} \frac{\partial z}{\partial V_{ij}} = \text{err}(z) \frac{\partial z}{\partial V_{ij}} \]

\[ \frac{\partial L(x, i^*)}{\partial z} = \text{err}(z) = W^T \text{err(root)} \]
Backpropagation

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

- Step 1: compute \( err(\text{root}) = e_{i^*} - P(y|x) \) (vector)
- Step 2: compute derivatives of \( W \) using \( err(\text{root}) \) (matrix)
- Step 3: compute \( \frac{\partial L(x, i^*)}{\partial z} = err(z) = W^T err(\text{root}) \) (vector)
- Step 4: compute derivatives of \( V \) using \( err(z) \) (matrix)
- Step 5+: continue backpropagation (compute \( err(f(x)) \) if necessary...)

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

- Step 1: compute \( err(\text{root}) = e_{i^*} - P(y|x) \) (vector)
- Step 2: compute derivatives of \( W \) using \( err(\text{root}) \) (matrix)
- Step 3: compute \( \frac{\partial L(x, i^*)}{\partial z} = err(z) = W^T err(\text{root}) \) (vector)
- Step 4: compute derivatives of \( V \) using \( err(z) \) (matrix)
- Step 5+: continue backpropagation (compute \( err(f(x)) \) if necessary...)

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]
Backpropagation: Takeaways

- Gradients of output weights $W$ are easy to compute — looks like logistic regression with hidden layer $z$ as feature vector.

- Can compute derivative of loss with respect to $z$ to form an “error signal” for backpropagation.

- Easy to update parameters based on “error signal” from next layer, keep pushing error signal back as backpropagation.

- Need to remember the values from the forward computation.
Applications
NLP with Feedforward Networks

- Part-of-speech tagging with FFNNs

  Fed raises **interest rates** in order to ...

  previous word

- Word embeddings for each word form input

  ~1000 features here — smaller feature vector than in sparse models, but every feature fires on every example

- Weight matrix learns position-dependent processing of the words

other words, feats, etc.

Botha et al. (2017)
NLP with Feedforward Networks

- Hidden layer mixes these different signals and learns feature conjunctions

Botha et al. (2017)
NLP with Feedforward Networks

- Multilingual tagging results:

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc.</th>
<th>Wts.</th>
<th>MB</th>
<th>Ops.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gillick et al. (2016)</td>
<td>95.06</td>
<td>900k</td>
<td>-</td>
<td>6.63m</td>
</tr>
<tr>
<td>Small FF</td>
<td>94.76</td>
<td>241k</td>
<td>0.6</td>
<td>0.27m</td>
</tr>
<tr>
<td>+Clusters</td>
<td>95.56</td>
<td>261k</td>
<td>1.0</td>
<td>0.31m</td>
</tr>
<tr>
<td>$\frac{1}{2}$ Dim.</td>
<td>95.39</td>
<td>143k</td>
<td>0.7</td>
<td>0.18m</td>
</tr>
</tbody>
</table>

- Gillick used LSTMs; this is smaller, faster, and better

Botha et al. (2017)
Deep Averaging Networks: feedforward neural network on average of word embeddings from input

\[ h_1 = f(W_1 \cdot av + b_1) \]
\[ h_2 = f(W_2 \cdot h_1 + b_2) \]

\[ av = \frac{1}{4} \sum_{i=1}^{4} c_i \]

Predator \( c_1 \) is \( c_2 \) a \( c_3 \) masterpiece \( c_4 \)

Iyyer et al. (2015)
# Sentiment Analysis

<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>NBOW-RAND</td>
<td>76.2</td>
<td>42.3</td>
<td>81.4</td>
<td>88.9</td>
<td>91</td>
</tr>
<tr>
<td>NBOW</td>
<td>79.0</td>
<td>43.6</td>
<td>83.6</td>
<td>89.0</td>
<td>91</td>
</tr>
<tr>
<td>BiNB</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>
<tr>
<td>RecNN*</td>
<td>77.7</td>
<td>43.2</td>
<td>82.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RecNTN*</td>
<td>---</td>
<td>45.7</td>
<td>85.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRecNN</td>
<td>---</td>
<td>49.8</td>
<td>86.6</td>
<td></td>
<td>431</td>
</tr>
<tr>
<td>TreeLSTM</td>
<td>---</td>
<td>50.6</td>
<td>86.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCNN*</td>
<td>---</td>
<td>48.5</td>
<td>86.9</td>
<td>89.4</td>
<td></td>
</tr>
<tr>
<td>PVEC*</td>
<td>---</td>
<td>48.7</td>
<td>87.8</td>
<td>92.6</td>
<td></td>
</tr>
<tr>
<td>CNN-MC</td>
<td>81.1</td>
<td>47.4</td>
<td>88.1</td>
<td></td>
<td>2,452</td>
</tr>
<tr>
<td>WRRBM*</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>89.2</td>
<td></td>
</tr>
</tbody>
</table>

- Bag-of-words
- Tree RNNs / CNNS / LSTMS

- Wang and Manning (2012)
- Iyyer et al. (2015)
- Kim (2014)
Coreference Resolution

- Feedforward networks identify coreference arcs

Clark and Manning (2015), Wiseman et al. (2015)
Implementation Details
Computing gradients is hard!

Automatic differentiation: instrument code to keep track of derivatives

\[ y = x \times x \quad \text{codegen} \quad (y, dy) = (x \times x, 2 \times x \times dx) \]

Computation is now something we need to reason about symbolically

Use a library like PyTorch or TensorFlow. This class: PyTorch
Define forward pass for \( P(y|x) = \text{softmax}(Wg(Vf(x))) \)

```python
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Computation Graphs in Pytorch

\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

def make_update(input, gold_label):
    ffnn.zero_grad()  # clear gradient variables
    probs = ffnn.forward(input)
    loss = torch.neg(torch.log(probs)).dot(gold_label)
    loss.backward()
    optimizer.step()
Training a Model

Define a computation graph

For each epoch:

   For each batch of data:
      Compute loss on batch
      Autograd to compute gradients and take step

Decode test set
Batching

- Batching data gives speedups due to more efficient matrix operations

- Need to make the computation graph process a batch at the same time

```python
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label):
    ...
    probs = ffnn.forward(input)  # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

- Batch sizes from 1-100 often work well
Regularization: Dropout

- Very simple!
- In each forward pass, randomly set the activations for some nodes (neurons) to zero.
- Probability of dropping is a hyper-parameter; 0.5 is common.