Adaptive Supertagging [Clark & Curran, 2007]

Start with an initial prob. cutoff $\beta$

<table>
<thead>
<tr>
<th>He</th>
<th>reads</th>
<th>the</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NP$</td>
<td>$(S[pss]\backslash NP)/NP$</td>
<td>$NP/N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>
Adaptive Supertagging [Clark & Curran, 2007]

Prune a category, if its probability is below $\beta$ times the prob. of the best category

He | reads | the | book
---|---|---|---
$NP$ | $(S[pss]\backslash NP)/NP$ | $NP/N$ | $N$
Decrease $\beta$ if no spanning analysis

<table>
<thead>
<tr>
<th>He</th>
<th>reads</th>
<th>the</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NP$</td>
<td>$(S[pss] \backslash NP) / NP$</td>
<td>$NP / N$</td>
<td>$N$</td>
</tr>
<tr>
<td>$N$</td>
<td>$(S \backslash NP) / NP$</td>
<td>$NP / NP$</td>
<td>$(S \backslash NP) / NP$</td>
</tr>
<tr>
<td>$N / N$</td>
<td>$S \backslash NP$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decrease $\beta$ if no spanning analysis

<table>
<thead>
<tr>
<th>He</th>
<th>reads</th>
<th>the</th>
<th>book</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>$(S[pss]\backslash NP)/NP$</td>
<td>$NP/N$</td>
<td>$N$</td>
</tr>
<tr>
<td>N</td>
<td>$(S\backslash NP)/NP$</td>
<td>$NP/NP$</td>
<td>$(S\backslash NP)/NP$</td>
</tr>
<tr>
<td>N/N</td>
<td>$S\backslash NP$</td>
<td>$NP/NP$</td>
<td>$(S\backslash NP)/NP$</td>
</tr>
<tr>
<td>NP/NP</td>
<td>$(S[pt]\backslash NP)/NP$</td>
<td>$NP/NP$</td>
<td>$(S[dcl]\backslash NP)/NP$</td>
</tr>
</tbody>
</table>
Neural Networks

Neural Networks can be built for different input, output types.

- Outputs can be:
  - Linear, single output (Linear)
  - Linear, multiple outputs (Linear)
  - Single output binary (Logistic)
  - Multi output binary (Logistic)
  - 1 of k Multinomial output (Softmax)

- Inputs can be:
  - A scalar number
  - Vector of Real numbers
  - Vector of Binary

Goal of training: Given the training data (inputs, targets) and the architecture, determine the model parameters.

Model Parameters for a 3 layer network:
- Weight matrix from input layer to the hidden ($W_{jk}$)
- Weight matrix from hidden layer to the output ($W_{kj}$)
- Bias terms for hidden layer
- Bias terms for output layer

Our strategy will be:
- Compute the error at the output
- Determine the contribution of each parameter to the error by taking the differential of error wrt the parameter
- Update the parameter commensurate with the error it contributed.
When building a neural network, the designer would choose the following hyper parameters and non-linearities based on the application characteristics:

- Number of hidden layers
- Number of hidden units in each layer
- Learning rate
- Regularization coefficient
- Number of outputs
- Type of output (linear, logistic, softmax)
- Choice of Non linearity at the output layer and hidden layer (See next slide)
- Input representation and dimensionality
Commonly used non-linearities (fig: courtesy Socher)

logistic ("sigmoid")
$$f(z) = \frac{1}{1 + \exp(-z)}.$$  

f'(z) = f(z)(1 - f(z))

**tanh**
$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$  

f'(z) = 1 - f(z)^2

**hard tanh**
$$\tanh(z) = 2\logistic(2z) - 1$$

**soft sign**
$$\text{softsign}(z) = \frac{z}{1 + |z|}$$

**rectified linear (ReLU)**
$$\text{rect}(z) = \max(z, 0)$$
Objective Functions and gradients

• Linear – Mean squared error
  \[ E(w) = \frac{1}{2N} \sum_{n=1}^{N} (t_n - y_n)^2 \]

• Logistic with binary classifications: Cross Entropy Error

• Logistic with k outputs: k > 2: Cross Entropy Error

• Softmax: 1 of K multinomial classification: Cross Entropy Error, minimize NLL

• In all the above cases we can show that the gradient is: \((y_k - t_k)\) where \(y_k\) is the predicted output for the output unit \(k\) and \(t_k\) is the corresponding target
High Level Backpropagation Algorithm

• Apply the input vector to the network and forward propagate. This will yield the activations for hidden layer(s) and the output layer
  • \( net_j = \sum_i w_{ji} z_i \)
  • \( z_j = h(net_j) \) where \( h \) is your choice of non linearity. Usually it is sigmoid or tanh. Rectified Linear Unit (ReLU) is also used.

• Evaluate the error \( \delta_k \) for all the output units
  \( \delta_k = o_k - t_k \) where \( o_k \) is the output produced by the model and \( t_k \) is the target provided in the training dataset

• Backpropagate the \( \delta \)'s to obtain \( \delta_j \) for each hidden unit \( j \)
  \( \delta_j = h'(z_j) \sum_k w_{kj} \delta_k \)

• Evaluate the required derivatives
  \( \frac{\partial E}{\partial W_{ji}} = \delta_j z_i \)
Recurrent neural networks (RNN)
Recurrent neural networks

• Use the same computational function and parameters across different time steps of the sequence
• Each time step: takes the input entry and the previous hidden state to compute the output entry
• Loss: typically computed every time step
Figure from *Deep Learning*, by Goodfellow, Bengio and Courville
Recurrent neural networks

Math formula:

\[ a^{(t)} = b + W s^{(t-1)} + U x^{(t)} \]
\[ s^{(t)} = \tanh(a^{(t)}) \]
\[ o^{(t)} = c + V s^{(t)} \]
\[ \hat{y}^{(t)} = \text{softmax}(o^{(t)}) \]

Figure from Deep Learning, Goodfellow, Bengio and Courville
Advantage

• Hidden state: a lossy summary of the past
• Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
• Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)
Advantage

• Hidden state: a lossy summary of the past
• Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
• Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

• Yet still powerful (actually universal): any function computable by a Turing machine can be computed by such a recurrent network of a finite size (see, e.g., Siegelmann and Sontag (1995))
Figure 4: A simple recurrent network.
Recurrent Network Variations

- This network can theoretically learn contexts arbitrarily far back
- Many structural variations
  - Elman/Simple Net
  - Jordan Net
  - Mixed
  - Context sub-blocks, etc.
  - Multiple hidden/context layers, etc.
  - Generalized row representation
- How do we learn the weights?
Simple Recurrent Training – Elman Training

- Can think of net as just being a normal MLP structure where part of the input happens to be a copy of the last set of state/hidden node activations. The MLP itself does not even need to be aware that the context inputs are coming from the hidden layer.
- Then can train with standard BP training.
- While network can theoretically look back arbitrarily far in time, Elman learning gradient goes back only 1 step in time, thus limited in the context it can learn.
  - Would if current output depended on input 2 time steps back.
- Can still be useful for applications with short term dependencies.
BPTT – Backprop Through Time

- BPTT allows us to look back further as we train
- However we have to pre-specify a value $k$, which is the maximum that learning will look back
- During training we *unfold* the network in time as if it were a standard feedforward network with $k$ layers
  - But where the weights of each unfolded layer are the same (shared)
- We then train the unfolded $k$ layer feedforward net with standard BP
- Execution still happens with the actual recurrent version
- Is not knowing $k$ apriori that bad? How do you choose it?
  - Cross Validation, just like finding best number of hidden nodes, etc., thus we can find a good $k$ fairly reasonably for a given task
  - But problematic if the amount of state needed varies a lot
- $k$ is the number of feedback/context blocks in the unfolded net.
- Note $k=1$ is just standard MLP with no feedback
- 1st block $h(0)$ activations are just initialized to a constant or 0 so $k=1$ is still same as standard MLP, so just leave it out for feedforward MLP
- Last context block is $h(k-1)$
- $k=2$ is Elman training
Training RNN

• Principle: unfold the computational graph, and use backpropagation
• Called back-propagation through time (BPTT) algorithm
• Can then apply any general-purpose gradient-based techniques
Training RNN

• Principle: unfold the computational graph, and use backpropagation
• Called back-propagation through time (BPTT) algorithm
• Can then apply any general-purpose gradient-based techniques

• Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters
Recurrent neural networks

Math formula:

\[ a^{(t)} = b + W s^{(t-1)} + U x^{(t)} \]
\[ s^{(t)} = \tanh(a^{(t)}) \]
\[ o^{(t)} = c + V s^{(t)} \]
\[ \hat{y}^{(t)} = \text{softmax}(o^{(t)}) \]

Figure from *Deep Learning*, Goodfellow, Bengio and Courville
Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$
Gradient at $o^{(t)}$:

$$\frac{\partial L}{\partial o^{(t)}_i} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o^{(t)}_i} = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}$$

Figure from Deep Learning, Goodfellow, Bengio and Courville
Gradient at $s^{(\tau)}$:

$$(\nabla_{o^{(\tau)}} L) \frac{\partial o^{(\tau)}}{\partial s^{(\tau)}} = (\nabla_{o^{(\tau)}} L) V$$

Figure from *Deep Learning*,
Goodfellow, Bengio and Courville
Gradient at $s^{(t)}$:

$$
(\nabla_{s^{(t+1)}} L) \frac{\partial s^{(t+1)}}{\partial s^{(t)}} + (\nabla_{o^{(t)}} L) \frac{\partial o^{(t)}}{\partial s^{(t)}}
$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville
Gradient at parameter $V$:

$$\sum_t (\nabla_{o(t)} L) \frac{\partial o(t)}{\partial V} = \sum_t (\nabla_{o(t)} L) s(t)^T$$
Dealing with the vanishing/exploding gradient in RNNs

- Gradient clipping – for large gradients – type of adaptive LR
- Linear self connection near one for gradient – Leaky unit
- Skip connections
  - Make sure can be influenced by units $d$ skips back, still limited by amount of skipping, etc.
- Time delays and different time scales
- LSTM – Long short term memory - Current state of the art
  - Gated recurrent network
  - Keeps self loop to maintain state and gradient constant as long as needed – self loop is gated by another learning node - forget gate
  - Learns when to use and forget the state
Other Recurrent Approaches

- **LSTM**
- **RTRL** – Real Time Recurrent Learning
  - Do not have to specify a $k$, will look arbitrarily far back
  - But note, that with an expectation of looking arbitrarily far back, you create a very difficult problem expectation
  - Looking back more requires increase in data, else overfit – Lots of irrelevant options which could lead to minor accuracy improvements
  - Have reasonable expectations
  - $n^4$ and $n^3$ versions – too expensive in practice
- Recursive Network – Dynamic tree structures
- Reservoir computing: Echo State Networks and Liquid State machines
- Hessian Free Learning
- Tuned initial states and momentum
- Neural Turing Machine – RNN which can learn to read/write memory
- Relaxation networks – Hopfield, Boltzmann, Multcons, etc.
Supertagging with a RNN

• Using only dense features
  – word embedding
  – suffix embedding
  – capitalization

• The input layer is a concatenation of all embeddings of all words in a context window
Supertagging with a RNN

... bought some books and ...
Supertagging with a RNN

... bought some books and ...

Diagram with nodes connected by arrows, indicating a neural network structure.
Supertagging with a RNN

... bought some books and ...

...
Supertagging with a RNN

... bought some books and ...

[Diagram with arrows and labels indicating the process of supertagging with an RNN]
Supertagging with a RNN

... bought some books and ...
### 1-best Supertagging Results: dev

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;C (gold POS)</td>
<td>92.60</td>
<td>-</td>
</tr>
<tr>
<td>C&amp;C (auto POS)</td>
<td>91.50</td>
<td>0.57</td>
</tr>
<tr>
<td>NN</td>
<td>91.10</td>
<td>21.00</td>
</tr>
<tr>
<td>RNN</td>
<td>92.63</td>
<td>-</td>
</tr>
<tr>
<td>RNN+dropout</td>
<td>93.07</td>
<td>2.02</td>
</tr>
</tbody>
</table>

Table 1: 1-best tagging accuracy and speed comparison on CCGBank Section 00 with a single CPU core (1,913 sentences), tagging time in secs.
1-best Supertagging Results: test

<table>
<thead>
<tr>
<th>Model</th>
<th>Section 23</th>
<th>Wiki</th>
<th>Bio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C&amp;C (gold POS)</td>
<td>93.32</td>
<td>88.80</td>
<td>91.85</td>
</tr>
<tr>
<td>C&amp;C (auto POS)</td>
<td>92.02</td>
<td>88.80</td>
<td>89.08</td>
</tr>
<tr>
<td>NN</td>
<td>91.57</td>
<td>89.00</td>
<td>88.16</td>
</tr>
<tr>
<td>RNN</td>
<td>93.00</td>
<td>90.00</td>
<td>88.27</td>
</tr>
</tbody>
</table>

Table 2: 1-best tagging accuracy comparison on CCGBank Section 23 (2,407 sentences), Wikipedia (200 sentences) and Bio-GENIA (1,000 sentences).
Multi-tagging Results: dev

![Graph showing multi-tagging accuracy vs ambiguity level for different models: RNN, RNN + dropout, NN, and C&C. The graph plots multi-tagging accuracy on the y-axis and ambiguity level on the x-axis. The RNN + dropout model shows the highest accuracy across all ambiguity levels.](image-url)
Multi-tagging Results: test
## Final Parsing Results

<table>
<thead>
<tr>
<th>CCGBank Section 23</th>
<th>Wikipedia</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LP</td>
</tr>
<tr>
<td>C&amp;C</td>
<td>86.24</td>
</tr>
<tr>
<td>(NN)</td>
<td>86.71</td>
</tr>
<tr>
<td>(RNN)</td>
<td><strong>87.68</strong></td>
</tr>
<tr>
<td>C&amp;C</td>
<td>86.24</td>
</tr>
<tr>
<td>(NN)</td>
<td>86.71</td>
</tr>
<tr>
<td>(RNN)</td>
<td><strong>87.68</strong></td>
</tr>
</tbody>
</table>

Table 3: Parsing test results (auto POS). We evaluate on all sentences (100% coverage) as well as on only those sentences that returned spanning analyses (% cov.). RNN and NN both have 100% coverage on the Wikipedia data.