Lecture 6: Neural Networks

Alan Ritter

(many slides from Greg Durrett)
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)

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. + Bahdanau et al.: seq2seq for neural MT (LSTMs work for NLP?)

- Chen and Manning 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 Net Basics
Neural Networks

- Linear classification: \( \text{argmax}_y w^\top f(x, y) \)

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

- Want to learn intermediate conjunctive features of the input

  \( \text{the movie was not all that good} \)

  \( \text{l[contains not & contains good]} \)
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 $y = (y_1, \ldots, y_n)$)

\[
x_2 \\
\downarrow \\
| \\
| \\
1 \quad 0 \\
| \\
\downarrow \\
| \\
0 \quad 1 \\
| \\
\hline
x_1
\]

<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

\[ y = a_1 x_1 + a_2 x_2 \]

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

“or”

(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

Input

First Layer

Second Layer

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

output of first layer

“Feedforward” computation (not recurrent)

Check: what happens if no nonlinearity?
More powerful than basic linear models?

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

Adopted from Chris Dyer
Deep Neural Networks

Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/
Feedforward Networks, Backpropagation
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)
- \text{softmax}: exps and normalizes a given vector
- Weight vector per class; \( W \) is [num classes x num feats]
- Now one hidden layer
Neural Networks for Classification

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

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

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

- Gradient with respect to $W$

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

- Looks like logistic regression with $z$ as the features!
Neural Networks for Classification

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

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

 Activation at hidden layer

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

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

\[ \text{some math...} \]

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

\[ \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = \text{err}(\mathbf{z}) = W^\top \text{err}(\text{root}) \]
\[ P(y|x) = \text{softmax}(Wg(Vf(x))) \]

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

- 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

Botha et al. (2017)
Hidden layer mixes these different signals and learns feature conjunctions.
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
Sentiment Analysis

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input

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>WRRBBM*</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>89.2</td>
<td>—</td>
</tr>
</tbody>
</table>

**Bag-of-words**
- Iyyer et al. (2015)

**Tree RNNs / CNNs / LSTMs**
- Wang and Manning (2012)
- Kim (2014)
Coreference Resolution

- Feedforward networks identify coreference arcs

President Obama signed...

He later gave a speech...

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

Automatic differentiation: instrument code to keep track of derivatives

\[ y = x \times x \quad \rightarrow \quad (y, dy) = (x \times x, 2 \times x \times dx) \]

ecodegen

Computation is now something we need to reason about symbolically

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

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

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

\( e_i^*: \) one-hot vector of the label (e.g., \([0, 1, 0]\))

```python
ffnn = FFNN()
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 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
Next Time

- More implementation details: practical training techniques
- Word representations / word vectors
- word2vec, GloVe